Modular Construction and Optimization of the UZP Sparse Format for SpMV on CPUs

ALONSO RODRÍGUEZ-IGLESIAS, CITIC, Universidade da Coruña, Spain
SANTOSHKUMAR T. TONGLI, Colorado State University, USA
EMILY TUCKER, Colorado State University, USA
LOUIS-NOËL POUCHET, Colorado State University, USA
GABRIEL RODRÍGUEZ, CITIC, Universidade da Coruña, Spain
JUAN TOURIÑO, CITIC, Universidade da Coruña, Spain

Sparse data structures are ubiquitous in modern computing, and numerous formats have been designed to represent them. These formats may exploit specific sparsity patterns, aiming to achieve higher performance for key numerical computations than more general-purpose formats such as CSR and COO.

In this work we present UZP, a new sparse format based on polyhedral sets of integer points. UZP is a flexible format that subsumes CSR, COO, DIA, BCSR, etc., by raising them to a common mathematical abstraction: a union of integer polyhedra, each intersected with an affine lattice. We present a modular approach to building and optimizing UZP: it captures equivalence classes for the sparse structure, enabling the tuning of the representation for target-specific and application-specific performance considerations. UZP is built from any input sparse structure using integer coordinates, and is interoperable with existing software using CSR and COO data layout. We provide detailed performance evaluation of UZP on 200+ matrices from SuiteSparse, demonstrating how simple and mostly unoptimized generic executors for UZP can already achieve solid performance by exploiting $\mathcal{Z}$-polyhedra structures.

CCS Concepts: • **General and reference** → **Performance**; • **Theory of computation** → *Vector / streaming algorithms.*

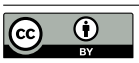
Additional Key Words and Phrases: sparse linear algebra, code generation, sparse format, polyhedral compilation, SIMD vectorization

ACM Reference Format:
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

1 Introduction

Sparse computations typically trade off the regularity and ease-to-optimize of dense versions for the benefits of reducing storage and computation time, by avoiding operations that produce a zero value (e.g., multiplications by zero). While the throughput may decrease compared to a dense implementation, sparse implementations compensate by reducing the total number of operations to perform.

Authors' Contact Information: Alonso Rodríguez-Iglesias, CITIC, Universidade da Coruña, Spain, alonso.rodriguez@udc.es; Santoshkumar T. Tongli, Colorado State University, USA, Santoshkumar.T@colostate.edu; Emily Tucker, Colorado State University, USA, Emily.Tucker@colostate.edu; Louis-Noël Pouchet, Colorado State University, USA, pouchet@colostate.edu; Gabriel Rodríguez, CITIC, Universidade da Coruña, Spain, gabriel.rodriguez@udc.es; Juan Touriño, CITIC, Universidade da Coruña, Spain, juan.tourino@udc.es.



ACM 2475-1421/2025/6-ART232
https://doi.org/10.1145/3729335

Sparse computations are typically implemented using specialized data structures that can conveniently represent arbitrary sets of nonzero coordinates. They must cope with the *irregular* nature of these sets of nonzeros, in contrast to the *regular* loops and accesses used in dense computations. In sparse computations, the set of nonzero coordinates to operate on is typically stored explicitly, in contrast to dense codes where the coordinates are *computed* using, e.g., affine functions of the loop iterators. Such approaches enable the modeling of arbitrary sparsity, irrespective of any "structure" it may expose, making formats like CSR, CSC, COO, BCSR, etc. general-purpose. However, format-specific *executor* programs need to be created to implement a particular computation on these stored nonzero coordinates [23], and these executors must be *optimized* to the particular hardware targeted [4, 36, 46].

On the other hand, dense computations are typically implemented using regular loops and simple array accesses, where indices are computed as a function of loop iterators, enabling aggressive optimizations such as tiling and parallelization [5] including efficient SIMD vectorization [22]. There is a large landscape of frameworks to optimize dense computations, and especially dense linear algebra, for a multitude of hardware targets. As dense linear algebra computations often fit the restrictions of *polyhedral compilation*, i.e., that loops and array accesses are represented using affine functions of the loop iterators, numerous polyhedral frameworks can automatically optimize such dense computations. Recently, the Affine MLIR dialect [25] has gained popularity as it bridges the gap between expressing regular dense computations (e.g., arising from deep learning applications) and the polyhedral optimizers that have proved successful over the past decade [5, 43]. In this work, we target the expression of sparse computations as a union of smaller dense computations.

Approaches developed to improve the performance of sparse computations range from Inspector/Executor (I/E) frameworks using general-purpose sparse formats [35, 40], where the sparse structure is traversed and manipulated (partitioned, reordered) prior to computation to enable a more efficient executor program; to sparsity-specific approaches where code is generated at compile-time for one specific set of nonzero coordinates, removing all indirection arrays [2] and enabling effective SIMD vectorization [9, 22, 47]. In between, numerous formats exploiting some form of regularity in the sparsity have also been developed, e.g., DIA and BCSR, to alleviate part of the performance penalty of using structure-agnostic general-purpose formats like CSR or COO.

An open question, recurring for practitioners, is to figure out which format(s) perform best for a particular sparsity structure, and how best to optimize the computation (executors) for a particular hardware target. In this work we propose an alternate approach, which reconciles sparse and dense computations and their optimizations. We present a *tunable unified sparse representation format*, inspired from recent results in reconstructing sparse sequences of integer tuples as unions of polyhedra and lattices [2, 34]. We introduce UZP, a novel and highly flexible sparse format that exploits a simple yet fundamental observation: that *any set of sparse integer coordinates can be equivalently represented as a union of dense sets of such coordinates*. To an extreme, an arbitrary set of n sparse coordinates can be represented with n dense sets containing 1 coordinate each, preserving generality. To another extreme, e.g., if only points along the diagonal are nonzero, a single dense set $\{(i, j) : 0 \leq i < n \wedge i = j\}$ is sufficient. This idea generalizes to sparse tensors of arbitrary dimensionality, which UZP supports natively. In UZP, or Union of $\mathcal{Z}$-Polyhedra, these regular sets are modeled as the intersection of a polyhedron of integer points and an affine multidimensional integer lattice [19]. UZP supports strides between coordinates represented within one $\mathcal{Z}$-polyhedron and therefore can exploit some forms of structure in the sparsity patterns. We make the following contributions:

- We introduce the UZP sparse format, its design principles and flexibility, and its ability to seamlessly subsume other formats such as DIA and BCSR. This is presented in Sec. 2.

- We present a *modular* approach to build UZP from a sparse structure, enabling the design of simple generic executors for UZP targeting here SpMV computations on CPUs, computing on UZP by using regular computations without any indirection array. This is presented in Sec. 3.
- UZP is designed to seamlessly enable alternate representations of the same sparse structure. We discuss techniques to quickly build UZP from any sparse structure, and *tune* the UZP representation for different objectives such as data compression or performance of a particular application. This is presented in Sec. 4.
- We extensively evaluate UZP on a set of 229 matrices broadly selected from SuiteSparse, targeting SpMV computations on single and multi-core CPUs. We compare performance against Intel MKL, sparsity-specific code generation approaches, and other custom SpMV executors demonstrating the merits and trade-offs of UZP. This is presented in Sec. 5.

2 The UZP Sparse Format

2.1 Motivation and Prior Work

Augustine et al. [2] demonstrated the feasibility of using polyhedra and lattices to compress the set of nonzero coordinates of a sparse matrix. Intuitively, a coordinate in the n-dimensional sparse data can be represented as an $n+1$ integer tuple $(\vec{i}, data)$ where $\vec{i}$ is the n-dimensional nonzero coordinate (nzc), and $data$ the corresponding location in the data vector of the actual data value for this nzc. Using this encoding, they attempt to build $\mathcal{Z}$-polyhedra [19] that group several, possibly non-consecutive, nzc together. To perform a computation on the sparse structure, code that scans these $\mathcal{Z}$-polyhedra and therefore computes the actual nzc coordinates can be generated. Precisely, *polyhedral code generation* [3] produces a dense, regular loop nest that, when executing, computes exactly the integer tuples captured by the $\mathcal{Z}$-polyhedra. A $\mathcal{Z}$-polyhedron is defined as follows.

Definition 2.1 ($\mathcal{Z}$-polyhedron). *A $\mathcal{Z}$-polyhedron is the image of a k-dimensional integer polyhedron P by an $m \times k$ affine integer lattice L. P is defined by a conjunction of affine inequalities, which are affine expressions of the dimensions of P. Similarly, L is a function represented as a matrix of integer coefficients, with m rows and k columns.*

Note we do not restrict the input dimension size k, and $m = n + 1$ using the encoding above. That is, the image $L(P)$ of P by L (equivalently, the intersection of P and L) models the nzc and the corresponding position(s) in the data vector.

For illustration, suppose we manipulate a sparse matrix with 6 nzc: $(0,0,0)$, $(0,71,1)$, $(1,42,2)$, $(2,44,3)$, $(3,46,4)$, $(42,42,5)$ where the last element of each 3-tuple represents the position in the data vector, here following CSR/COO storage. A possible representation of this sparse structure uses 6 $\mathcal{Z}$-polyhedra: $P_1 : \{(i,j,d) : i = 0 \wedge j = 0 \wedge d = 0\}$ and $L_1 : \{(i,j,d) \rightarrow (i,j,d)\}$; $P_2 : \{(i,j,d) : i = 0 \wedge j = 71 \wedge d = 1\}$ and $L_2 = L_1$, etc. However, this is arguably of limited use versus a traditional COO representation: we use one polyhedron per nzc.

Instead, *compressing* several nzc into a single polyhedron can be achieved: $P_3 : \{(l) : 1 \leq l \leq 3\}$ and $L_3 : \{(l) \rightarrow (i,j,d) : i = l \wedge j = 40 + 2l \wedge d = l + 1\}$ directly captures $(1,42,2)$, $(2,44,3)$, $(3,46,4)$ *in a single polyhedral structure*, reducing the number of polyhedra needed to model the full sparse structure by 2. As we demonstrated in prior work, this approach is robust to seemingly unstructured sparsity, discovering when feasible strided repetitive (regular) patterns in the nzc set. For example, Fig. 1 is reconstructed using 870 $\mathcal{Z}$-polyhedra using up to 8 dimensions ($k = 8$) for them [2, 16].

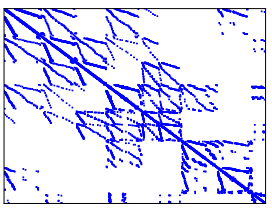


Fig. 1. HB/can_1072: 12444 nzc

2.2 Design Principles of UZP

We now present our novel polyhedral-based sparse format, UZP. A key objective of UZP, in contrast to other works [2, 22], is to enable the easy development of *a class of generic executors* that are independent of the specific sparsity structure, but still leverage the compression of the nonzero coordinates into polyhedra. Generic executors operating on UZP are presented in Sec. 3. Returning to the example in Sec. 2.1, an explicit list of all polyhedra used may contain a lot of redundancy. In particular, the same *polyhedral shape* (e.g., a diagonal of 3 elements strided by 2) may occur numerous times, as a result of the reconstruction approach used, which mines for repetitions of such shapes [2, 22]. UZP splits this representation, separating the description of the prototype shapes from the list of origins on which they are applied, removing redundancy and significantly reducing storage size. In addition, UZP contains segments that allow storing (parts of) the sparse structure in the classical CSR and COO formats. Indeed, for highly unstructured sparsity (e.g., polyhedra typically model 2 or less nzc) benefitting from a polyhedral representation, in terms of storage as well as performance, is unlikely. Enabling mixed representation within a single format greatly facilitates the tuning of the representation itself, and its interoperability with existing layouts, as discussed in Sec. 4.

A UZP representation is therefore composed of 5 distinct sections: (a) a dictionary of parametric shapes, that are parameterized $\mathcal{Z}$-polyhedra; (b) a list of origins, on which to apply a particular shape; (c) a CSR-encoded segment, (d) a COO-encoded segment and (e) a data vector.

2.2.1 Dictionary of Shapes and Origins. UZP splits the encoding of $\mathcal{Z}$-polyhedra into two structures: *parameterized* $\mathcal{Z}$-polyhedra, and *origins* they shall be applied on. For example, P_3 is encoded as $P^1 : \{(l) : 1 \leq l \leq 3\}$ and $L^1 : \{(l) \rightarrow (i, j, d) : i = I_1 + l \wedge j = I_2 + 40 + 2l \wedge d = D + l + 1\}$ where $I_1, I_2, D \in \mathbb{Z}$ are constant parametric integers. That is, I_1, I_2 and D are offsets (for coordinates, and position in the data vector respectively). We have therefore $P_3 : (0, 0, 0, (P^1, L^1))$. That is, applying the origin $(0, 0, 0)$ to this parametric $\mathcal{Z}$-polyhedron (i.e., setting $I_1 = 0$, $I_2 = 0$, $D = 0$) leads to describing exactly P_3. This representation allows to store separately the description of shapes from the description of the origins on which each shape is applied. Consequently, if a prototype shape occurs multiple times in the sparse structure, only the various origins and associated shape identifiers need to be stored. An origin is simply an $m + 2$-dimensional integer tuple, e.g. $(0, 0, 0, id(P^1, L^1))$. This aspect is essential to reduce storage versus a classical explicit polyhedral representation. It is also essential to *enable the design of simple executors, that amount to a simple parametric regularly strided loop nest*, as detailed in the next section.

2.2.2 UZP Tuners. UZP fundamentally exposes a large set of equivalence classes, between different encodings of the same sparse structure. One may use different polyhedral shapes to compress the nzc, i.e., different reconstruction approaches lead to equivalent but different UZP representations. But, conveniently, the list of origins can be reordered in the format itself: this directly influences the schedule of operations implemented in the executor.

UZP is designed to enable the development of *tuners*: tools that optimize a UZP representation into another UZP one, for example trading off compression (size of the representation) versus performance (e.g., ability to efficiently vectorize the computation) by changing the shapes being used or the order in which they are iterated. We discuss this approach in Sec. 4.

2.2.3 Insertion and Deletion of Nonzeros. Many sparse formats, including CSR, implement storage of the actual data following a lexicographic ordering of the nonzero coordinates. This makes insertion/deletion of nzc tedious, possibly requiring to recompute/shift all arrays used for the sparse representation. In UZP, the insertion of a new nonzero coordinate, at any position, is an $O(1)$ operation: it simply amounts to adding a new origin at the end of the list, using a single-point

shape. This feature is possible because we explicitly encode the position in the data vector in UZP, via the integer lattice, and thanks to the generic executors presented in Sec. 3. However deletion of a nonzero coordinate still requires possibly shifting the `data` array, and re-generation of the shape/origin associated with this deleted point.

2.2.4 Limitations of UZP. By design, UZP groups possibly nonconsecutive points in the original sparse structure in the same shape. The processing of each individual shape by the executors being atomic, it constrains the order within which the nonzero coordinates are traversed, and a lexicographic ordering (e.g., as in CSR) is not implemented anymore, which may be detrimental to data locality without careful tuning.

UZP also heavily relies on branching: loops, possibly with very small trip counts, are used to compress nzc. As such, a high ratio of branches to nzc can introduce a significant penalty, especially if polyhedra of very few points are used (e.g., less than 4), and/or origins are sorted in a way that results in a high rate of branch mispredictions.

UZP may also have a storage size exceeding, e.g., CSR if the number of points compressed into each polyhedron is low (e.g., averaging less than 2 points per instantiated shape). To partially overcome this issue UZP supports a hybrid format, where an arbitrary set of points (those not efficiently compressed into polyhedra) can be modeled using the classical CSR or COO formats. We refer to these points as *unincorporated points* below.

2.3 Comparison of UZP with Other Formats

In terms of storage, UZP typically consumes $m + 2$ integers per origin, for an m-dimensional sparse structure. The size of the dictionary is typically negligible in comparison to the list of origins.

2.3.1 CSR and CSC. These formats typically use at least $O(nnz)$ metadata (for the column or row index), and compression versus COO is achieved irrespective of any structure within the sparse coordinates. UZP needs $O(4n)$ where n is the number of origins. That is, if on average there are more than 4 nzc per shape in the reconstructed structure (i.e., there are $O(nnz/4)$ origins), UZP will necessarily have a smaller footprint than CSR or CSC. Note the $data$ value is optional, and one can reorder the data according to the list of origins and remove this 4th component, reducing to O(3n) for a 2D sparse matrix. Note that we also ignore here the additional metadata proportional to the number of columns (resp. rows) needed in CSR (resp. CSC).

2.3.2 ELLPack. Similarly, ELL requires $O(nnz)$ metadata, in addition to possible additional storage for zero elements in the data vector. Profitability of UZP is analogous to CSR/CSC.

2.3.3 DIA and HDC. The DIA format allows the representation of dense diagonals using simply one element: the offset describing the position of the diagonal. In UZP a diagonal, dense or strided, wherever its starting point, is represented using one origin, that is up to 4 elements for a 2D matrix. This is negligible overhead versus DIA. Note that the generic executor for a UZP diagonal shape and the typical DIA generic executor are essentially identical, and therefore can deliver identical performance. HDC is a hybrid format combining DIA and CSR for elements outside the diagonal [18]; HDC is fully subsumed by UZP.

2.3.4 CSF. Compressed Sparse Fiber [38] has gained popularity to represent higher-dimensional tensors and their tiling/permutations [27]. CSF can achieve compression, especially when there are identical prefixes between nzc (i.e., identical values on the x first dimensions of the nzc). Similarly, this can be achieved when mining for shapes modeling an identical prefix in UZP. Overall, CSF will use at least one element per nzc, giving it an $O(nnz)$ footprint. UZP can achieve further compression as described above. UZP supports arbitrary shape sizes, including different shapes for different

parts of the matrix, subsuming formats such as Adaptive tiling [21] which supports different tile sizes for different tiles within a matrix.

Numerous other sparse formats can be encoded in UZP, making it a nearly one-size-fits-all representation, which can be easily converted to other formats such as CSR or DIA.

2.4 Implementation of UZP

We have implemented both the UZP format and the associated generic executors, presented in the next section, 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 as illustrated later in this paper. We summarize below some key aspects of the format implementation and illustrate with Fig. 2.

2.4.1 Dictionary of Shapes. A shape is a parametric $\mathcal{Z}$-polyhedron, and 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 UZP, columns of this matrix can be skipped when the shape is hyper-rectangular, as only extremal vertices are needed to represent the shape. For example 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, leading to the set $(42+0..42+8 \times 128+0..128+8)$.

2.4.2 List of Origins. An origin is an $m+2$-tuple of integer coefficients, including the first m-dimensional vertex (e.g., $(42,128)$ above), the *data* offset, and the shape identifier in the dictionary. The list of origins may never be larger than the nnz.

2.4.3 Modeling Data. The `data` vector stores the actual data values for each nzc. While it is possible to constrain the reconstruction of the sparse structure into polyhedral shapes to operate on a fixed data vector (e.g., the exact data vector of an existing CSR matrix representation, to ease integration in a full application flow of data-specific SpMV codes without any data conversion [2]), UZP allows reordering the data vector, for example, so that elements accessed by a single shape applied to a single origin appear consecutively in the `data` vector, in the lexicographic order of coordinates generated by this shape. This simplifies the representation of origins and shapes as described above.

2.4.4 Modeling Unincorporated Points. Finally, UZP contains an optional section, typically used to encode points that cannot be efficiently captured by $\mathcal{Z}$-polyhedral shapes: instead of modeling them using a single-point shape and the associated origins, we support modeling these points using the classical CSR and COO formats, with their separate `data` segment.

2.4.5 Example. Fig. 2 exemplifies the representation of a sparse matrix in CSR, COO, and two possible UZP representations. The dictionary encodes here hyper-rectangular shapes, but we also support more general shapes. Taking `s1` on the bottom left, it describes a 1D shape. Shapes are encoded as a polyhedron and a lattice. The first two values describe the 1D polyhedron vertices (e.g. $l : 0 \leq l \leq 2$, so we store 0, 2). The next values describe the coefficients of an integer affine lattice $(l) \rightarrow (i,j,d)$. That is 0, 0, 2, 0, 1, 0 corresponds to $i = 0*l+0$, $j = 2*l+0$, $d = 1*l+0$. Origins are then applied to this $\mathcal{Z}$-polyhedron, e.g. the origin $(0,0,0,s1)$ on the left UZP example, which leads to producing the tuples $(0,0,0)$, $(0,2,1)$ and $(0,4,2)$ which correspond to the first three nzc, S-C-A. Optionally, a CSR or COO fragment can be used to encode some nzc, but equivalently they can be encoded with a polyhedron of one point, as shown in the right UZP example (s4). Note that shapes of multiple dimensionalities can coexist in the dictionary, as shown in the example on the right in which s1 is a 2D shape (defined as $l_1, l_2 : 0 \leq l_1 \leq 1 \ \wedge \ 0 \leq l_2 \leq 2$, that is 0, 1, 0, 2 and the lattice

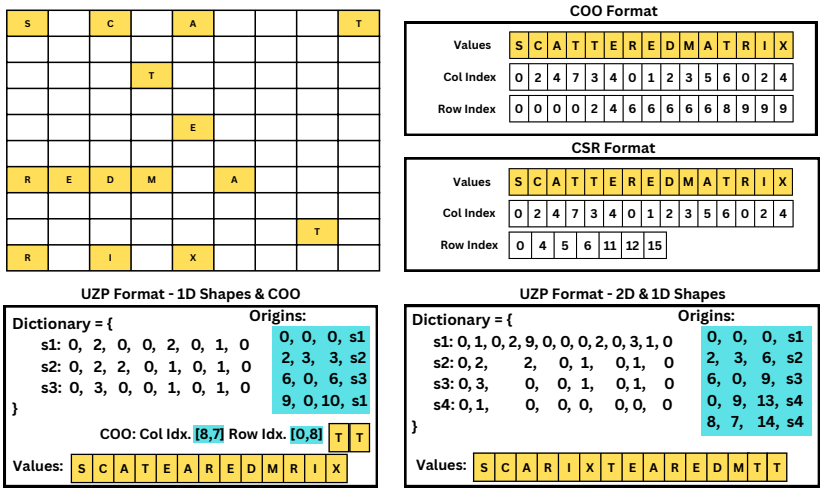


Fig. 2. Different possible representations of the same sparse matrix: CSR, COO and two possible UZPs.

$(l_1, l_2) \longrightarrow (i, j, d)$ e.g. $i = 9 * l_1 + 0 * l_2 + 0$). This shape now requires 3 integer coefficients per output dimension instead of 2, and when the origin $(0, 0, 0, s1)$ is applied, it captures the nzc S-C-A-R-I-X.

3 Generic Executors for UZP-GenEx

Prior work using $\mathcal{Z}$-polyhedra to encode sparse structures targeted specifically the generation of sparsity-specific programs, from that representation [2, 22]. We aim in this work to remove this limitation, enabling the design of *generic executors*, similarly to traditional sparse formats, that can then be tuned for a particular objective or hardware targeted.

A central aspect of UZP is the ability to benefit from coordinate compression using $\mathcal{Z}$-polyhedra while also *enabling the development of simple generic executors for various computations*, such as SpMV illustrated in Sec. 5. These executors perform (strided) dense computations, and can be seamlessly optimized by polyhedral optimizers. They implement a complete traversal of all nonzero coordinates to form a computation. 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, 22], however at the cost of some performance potential lost by lack of code specialization [22], as shown below.

3.1 Design Principles of the Generic Executors

The generic executor scans all nonzero coordinates by (1) enumerating all origins and, for each, (2) scanning the corresponding shape using a loop nest to compute the original nonzero coordinates, and using them to perform the desired computation. While it is possible to create a single "generic" parametric polyhedral loop nest that can handle arbitrary polyhedral shapes, to facilitate the C/C++ compiler's optimizations of the executor code, we instead explicitly provide a prototype parameterized loop nest for each dimensionality case. This remains a small and practical set, as we never use shapes above 8D, and model only sparse tensors of 1D to 4D, leading to 80 cases.

We exemplify this by showing below a generic 2D shape executor code, for SpMV, on UZP. Note we perform strength reduction to present the compiler with a clear induction for the lattice coordinates. For simplicity, we assume below the data vector has been laid out to use the unit lattice to compute the data position of every element.

```
double* const data_vector = uzp_matrix->data;
int idx_i, idx_j;
// Retrieve the shape information from origin:
int a_data_pos = orig.data_offset;
// Retrieve the shape parameters:
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 the 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 part:
        y[idx_i] += data_vector[a_data_pos] * x[idx_j];
        // Increment lattice offset:
        idx_i += offset_idx_i;
        idx_j += offset_idx_j;
        a_data_pos += 1;
}   }
```

We remark that, for any computation that requires a single traversal of the sparse matrix without any specific ordering constraint, this generic executor can be used and an inlinable function/macro can be called for the computation-specific part that captures the specific point-wise computation to be performed, using `idx_i`, `idx_j` and `a_data_pos`.

3.2 Performance Considerations

To improve performance, we rely on the C/C++ compiler's optimization of the executor code. To enable analysis and optimization of these strided loops, *specializing* the code for different values of the loop strides and lattices explicitly in the executor provides sufficient information at compile-time to enable their auto-vectorization. We exemplify below one such specialization for a 1D shape representing a dense vertical vector of coordinates:

```
// Specialized 1D shape #0: lattice[0] = 1,
// lattice[1] = 0, lattice[2] = 0, stride[0] = 1,
// start[0] = 0, arbitrary end[0] supported.
int idx_i = orig.coordinates[0];
int idx_j = orig.coordinates[1];
for (int i = 0; i <= shape.end[0]; i += 1) {
  // Computation-specific part:
  y[idx_i] += data_vector[a_data_pos] * x[idx_j];
  // Increment lattice offset:
  idx_i += 1;
  a_data_pos += 1;
}
```

We have specialized 1D and 2D shapes, for dense and strided horizontal, vertical, and diagonal shapes. A unique `int` hash key, identifying the specialized case in the code, is associated to each shape described in the dictionary. Then a simple `switch` statement on this key is implemented in the executor, calling the specialized version when available and defaulting to the generic shape executors as described above otherwise. Unambiguously, versioning and semi-manual SIMD vectorization bypassing compilers' cost models shall be deployed for improved performance of these generic executors. In this paper, we limit to explicitly versioning a handful of shapes, using auto-vectorization by GCC. *Tuning the performance of these generic executors for a particular hardware target is outside the scope of the present paper*, as we focus instead on evaluating the performance of mostly *the simplest loop forms* needed for correct UZP executors, a sort of base-case performance scenario to expose the limitations of UZP.

3.3 Data-Specific versus Generic Executors

Generic executors bring convenience by separating the data from the code it is executed on, allowing their independent optimizations, and may still enable powerful automatic optimizations by compilers as they are implemented using regular affine loop nests. However, *specialized* sparsity-specific code, e.g., where data and code are folded in the same binary program and both *data-specific* and machine-specific SIMD instructions are generated [22] are still expected to provide a significant performance advantage. In particular, MACVETH performs across-loop optimizations for SIMD such as packing small yet independent reductions in the same vector to improve vector occupancy in the presence of loops with small trip counts.

We illustrate this with Fig. 3, which plots the single-core performance in GFLOPS of MACVETH and UZP for a set of 229 SuiteSparse matrices described in Sec. 5. Note that, while for 96% of matrices MACVETH improves performance over MKL, the generic executor improves for only 61% of them. This remains a respectable improvement, motivating the use of UZP and generic executors even if they are vastly unoptimized. Yet it also displays the merits of sparsity-specific specialized code generation on top of UZP.

Analyzing with hardware counters, from a vectorization point of view in data-specific codes 90% of the FLOPs in the experimental dataset are executed using AVX2 operations (i.e., 256-bit vectors), vs. only 34% in the generic executor, itself automatically vectorized by GCC. This means a 2.5x increase in the number of FLOP instructions issued by the generic executor program, reducing efficiency. Together with other overheads, including a 4.2x increase in the number of memory accesses, as well as a 116x increase in the branch instruction count, this adds up to a net 5.8x increase in the total number of instructions executed. This ultimately offsets the 725x decrease in instruction cache misses vs. MACVETH, resulting in an overall lower performance.

4 Conversion to UZP and Tuning

4.1 Building UZP from a Sparse Structure

We have implemented Polyhedrator using Rust, a new tool to quickly mine for specific polyhedral shapes in a trace of nonzero coordinates. In contrast to prior work [2], the shape of the polyhedra used to compress nzc is not discovered by the mining algorithm itself, but instead are pre-defined (by the user or a tool) prior to reconstruction, and are input to our mining tool. This enables a predictable and fast reconstruction time, while *tuners* are deployed afterwards to build more complex and larger shapes as feasible, as explained in Sec. 4.3.

The worst-case complexity is $O(s \times nnz)$ where s is the number of prototype shapes we look for, in the set of nnz. That is, the user provides the set of polyhedral patterns $\{p_1, \ldots, p_n\}$ to be mined, which may contain any $\mathcal{Z}$-polyhedra description of any dimensionality, specified through

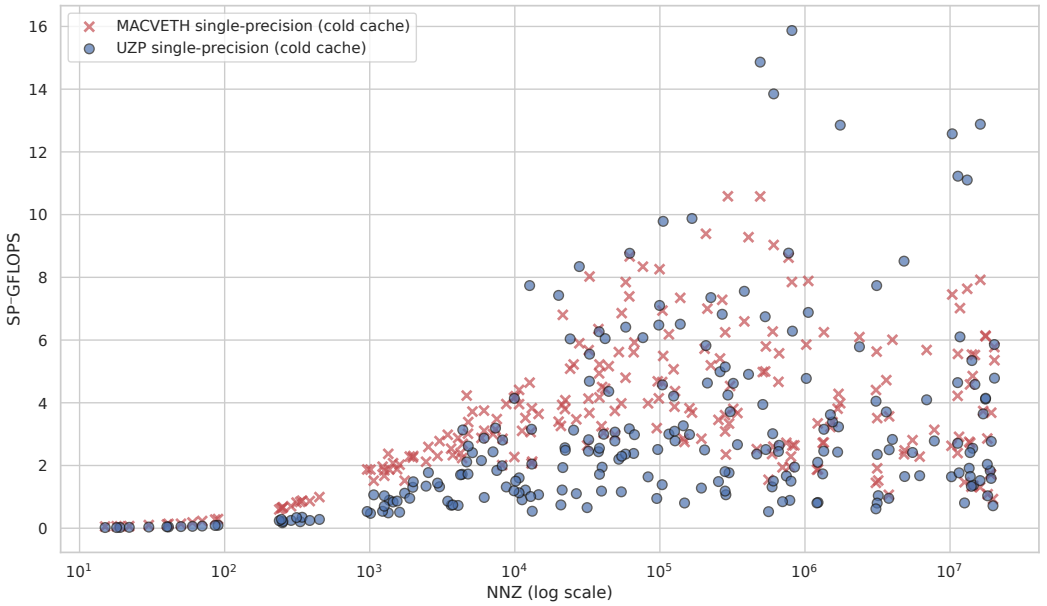


Fig. 3. UZP Generic Executor versus MACVETH performance

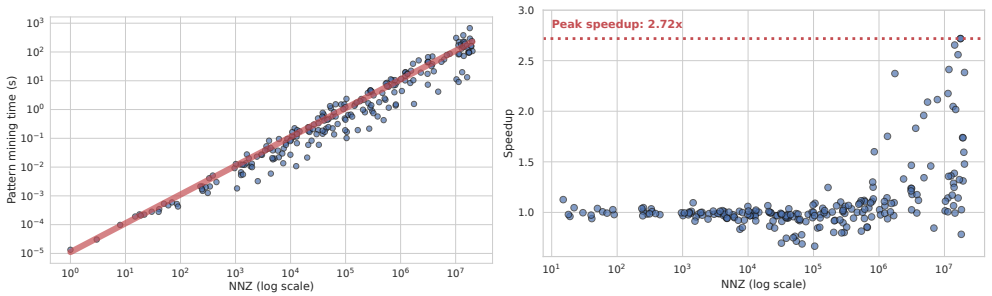
its domain, strides, and lattices. A list of nonzero coordinates $\{c_1, \ldots, c_m\}$ describing the sparsity structure (e.g., from an existing CSR representation) is then repeatedly scanned. This mining boils down to checking, for each c_i and for each pattern p_j in the search set, whether p_j exists in the structure when using c_i as origin. Given that choosing c_i inside a given pattern precludes its inclusion in a different one, this approach does not guarantee to find the reconstruction that minimizes the number of origins, exhaustive search may be needed to reach optimality. Instead we perform the search in a greedy *pattern-first* fashion, in which all nonzeros in the matrix are checked as starting points for each p_j in order. In this mode, a point c_i will never be chosen into a pattern p_j if it belongs inside another pattern $p_k, k < j$.

We also support a "hierarchical reconstruction" mode. In this case, the input to the tool is already a UZP representation. It is scanned for instances of the same shapes, and the origins obtained are then mined again against the prototype shapes. Alternatively, we may employ the TRE approach [33, 34] to look for grouping origins, using more general and complex polyhedral shapes. By applying hierarchical reconstruction, low-dimensional pieces are fused together, achieving higher compression. Generic executors can specialize execution strategies for hierarchically reconstructed pieces, applying optimizations such as, e.g., register tiling. Exploring this angle is left as future work.

4.2 Experiments Building UZP

We generated UZP files for the 229 matrices in the experimental set described in Sec. 5, using a list of 664 1-dimensional vertical, horizontal, and diagonal patterns with different strides. *This set of shapes is excessively large compared to what is required, in that we again display a form of worst-case time* when a very high amount of different shapes is evaluated. In practice, using smaller shapes and tuners as described below to merge into larger shapes would achieve the same result but reduce UZP generation time accordingly.

Fig. 4a displays this generation time. Mining time grows mostly linearly with the number of nonzeros, given the number of shapes is a constant here (664); however, as points get incorporated, the set of origins to consider is constant or decreases with iterations. Our implementation achieves a throughput of about 90K nonzeros per second using a single CPU core. Note that parallelizing the conversion process and tiling are simple accelerations; however, we have not implemented them.



(a) (b)

Fig. 4. Left: UZP generation time. Right: speedup of locality-tuned UZP vs. randomly-sorted origins.

4.3 Tuning UZP

Obviously, the UZP representation generated is directly dependent on the list of prototype shapes being mined for. One can extensively change the list of simple shapes with varying strides, as described above. But another category of *tuning* can be easily deployed on UZP, by design, and in a manner independent from the optimization of the generic executors. We use two different tuners in our experiments in Sec. 5, described in the subsections below.

4.3.1 Merging Shapes. This tuner merges together contiguous small shapes into a larger shape, and works in conjunction with the prototype shapes list used for the initial mining. For example, assume the sparse matrix is in fact a dense diagonal matrix. We may mine for diagonals blocks of 4 consecutive elements only, to reduce the number of shapes mined, which leads to $n/4$ such origins in the UZP produced. Then, a *merging tuner* inspects the origins using the same identical shape, and creates a single, larger polyhedral shape that captures contiguous repetitions of this shape. Here a single origin is found, now applied to a polyhedral shape of n points instead of 4. This approach complements hierarchical reconstruction, and can speed up reconstruction overall, by only initially mining for small-sized template shapes and aggregating them in larger shapes in a second step, instead of mining for different scalings of the same shape.

We implemented this tuner, and it is deployed in all experiments reported in Sec. 5. Alg. 1 displays the skeleton of this tuner. Note that this algorithm can create multiple shapes with the same polyhedral description; for simplicity, it adds these (possibly redundant) shapes temporarily to the dictionary, a final normalization pass implements the removal of redundant shapes and adjusts the origins accordingly instead.

4.3.2 Reordering Origins for Data Locality. The order of the pieces in a UZP file can significantly impact spatial and temporal data locality, as well as branch (mis)prediction in the generic executors. We have implemented a tuner that reorders the origins in an SpMV file, favoring grouping shapes by locality along the x vector in SpMV computations, and compared the obtained results against a random ordering of origins in Fig. 4b.

Algorithm 1 Aggregation of shapes in input UZP file

```
1:  O := Origins(UZP_in)
2:  D := Dictionary(UZP_in)
3:  for each s in D do do
4:      O_s⃗ := extractOriginsWithShapeId(O, s)
5:      O_s⃗ := reverseLexicographicSort(O_s⃗)
6:      LastMatchedOrig := O_s⃗[1] // first element of O_s⃗
7:      NewShape := s
8:      for i in 2..|O_s⃗| do do
9:          orig := O_s⃗[i]
10:         Poly := ZPolyhedralUnion(NewShape(LastMatchedOrig), s(orig))
11:         if Poly.Lattice == s.Lattice then // Merge to a larger shape with same lattice
12:             NewShape := ExtractParametricShape(Poly)
13:             AddToSet(MergedOrigs, orig)
14:         else// Cannot merge further, reset
15:             AddToDictionary(D, NewShape)
16:             LastMatchedOrig := orig
17:             NewShape := s
18:         end if
19:     end for
20: end for
21: NewOrigins := O - MergedOrigs
22: RemoveDuplicatesInDictAndUpdateOrigins(D, NewOrigins)
23: return UZP(D, NewOrigins)
```

This tuner only alters the order of origins, and amounts to implementing a lexicographic sorting of the origins following the (j, i) order, to ensure maximal spatial/temporal locality for the x vector but at the expense of locality along the y vector. The data vector is reordered accordingly, to follow the new order for origins and ensuring perfect spatial locality along the sparse matrix data. This tuner is deployed in all the experiments reported in Sec. 5. Note that it is easy to build a more advanced tuner that would preserve data locality along x and y by simply implementing a "tiled" scheduling order for the origins (instead of a column-first ordering) while ensuring footprint constraints for the subset of x and y elements touched by a tile.

4.4 From UZP to CSR

To maintain interoperability with existing applications, it is critical to be able to quickly convert UZP to other traditional formats. Indeed, while UZP construction can be constrained to preserve the ordering of the data vector, for seamless integration in a full application using, e.g., CSR, it can also be reordered in UZP and/or split between different data segments in UZP. Our tool integrates this conversion back to CSR. In our experiments, the longest conversion time is 1.04 seconds, across the entire experimental set, for matrix `Mazaheri/bundle_adj` containing 20M nonzeros.

5 Experimental Results

We experiment on the 229 matrices selected by Horro et al. [22] by sieving the matrices below 20M nonzeros in SuiteSparse [16] using a similar technique to the one employed by Augustine et al. [2]. Note that the complexity of our UZP generator is mostly linear in the number of nonzeros, and

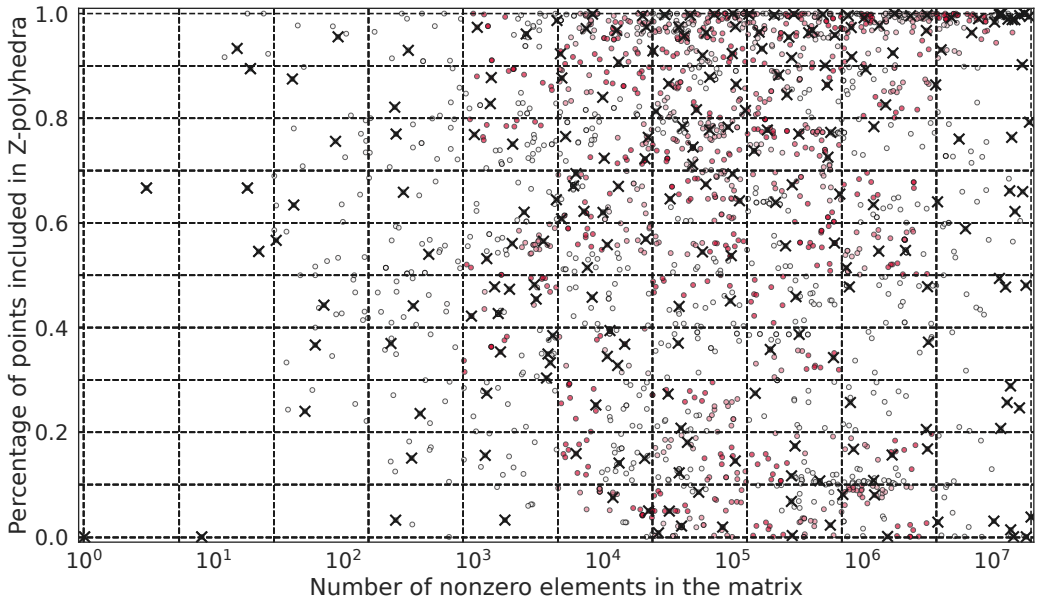


Fig. 5. Sieve of the 2,754 SuiteSparse matrices below 20M nonzeros. Selected matrices are marked with 'X's.

UZP generation time is displayed in Fig. 4a above. The full open-source UZP tools[1] used in this paper are available on Zenodo [30, 31].

Matrices are classified according to the decile they belong to in terms of number of nonzeros and percentage of them that is 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 selected so that the probability density of the sample matches the original one. Fig. 5 displays this selection process.

In this section, we focus on the analysis of the experimental performance of SpMV, executed on double-precision floating-point data. Experiments were run on an Intel Core i9 12900K with 128 GiB of RAM. All runs were repeated 10 times; we report the best performance achieved for each experiment after removing outliers, identified as measurements that deviate more than 3σ from the mean. The CPU frequency was fixed at the nominal base of 3.2 GHz to prevent thermal constraints from introducing experimental variability. Both the data and code segments were allocated 2 MiB hugepages for all the experimental versions. Codes were compiled using GCC 11.4.0. In all cases we introduce `-O2 -ffast-math -ftree-vectorize -floop-unroll-and-jam -march=native`. The PolyBench [29] testing harness was used for performance measurements. Prefetching of the text segment [2] was included in the linking process for data-specific codes. Using the same basic setup and running on the 16 logical P-cores of this 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 performance for SpMV computations of 5.5 GFLOPS.

We experimented with six different versions of the SpMV computation: 1) a vanilla CSR version shown in Listing 1; 2) MKL version 2024.1.0 using function `mkl_sparse_s_mv()`; 3) the same MKL version but using BCSR encoding through `mkl_sparse_convert_bsr()`; 4) our generic executor

[1]UZP: https://github.com/UDC-GAC/uzp-sparse-format

described in Sec. 3, labeled UZP GenEx; 5) a CSR5 SpMV kernel [26]; and 6) the publicly available GitHub source for the partially-strided codelets approach [8, 9].

In UZP files, pieces are aggregated as described in Sec. 4.3 and sorted to exploit locality over the dense vector in the computation during an inexpensive inspection phase. Similarly, for MKL experiments, the inspector-executor capabilities are employed by invoking the `mkl_sparse_optimize()` function outside the timed kernel. Symmetric matrices are expanded to their full size in all cases.

```
1 #pragma omp for private(j) nowait
2 for(i = 0; i < N; ++i)
3   for(j = row_ptr[i]; j < row_ptr[i+1]; ++j)
4     y[i] += A[j] * x[col_idx[j]];
```

Listing 1. CSR executor for SpMV. OpenMP is disabled for single-threaded experiments.

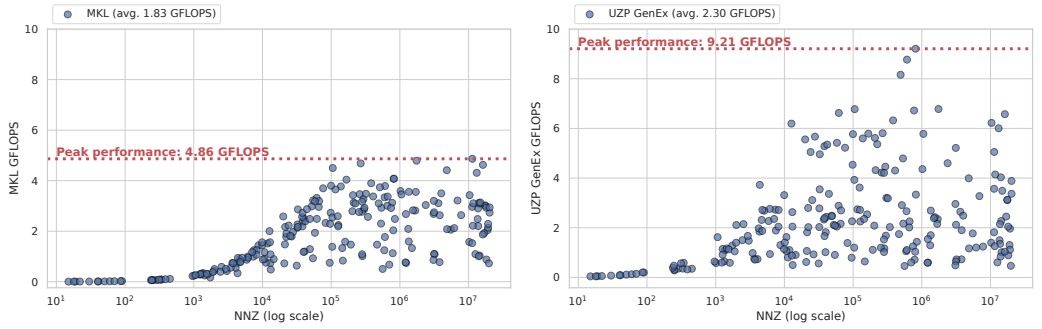
5.1 Single-Threaded Performance

The performance of these kernels under cold cache, single-threaded conditions is shown in Figs. 6 and 8. For these experiments, the SpMV kernel is executed exactly once after loading the data and flushing the caches, highlighting raw performance. UZP GenEx achieves the best overall performance, with an average of 2.29 GFLOPS across the entire experimental set (CSR: 1.63, CSR5: 1.39, MKL: 1.83). While the performance of both CSR and CSR5 remains consistently below that of UZP, MKL starts closing the gap for matrices above 500K nonzeros. Investigating the data carefully, we observe that the probability for GenEx to improve over MKL is driven by the percentage of unincorporated points, and whether few shapes are needed to capture the majority of incorporated points. Specifically, looking only at matrices with more than 10K nonzeros, the probability for MKL to outperform GenEx for matrices with more than 20% unincorporated points is 72%, where for matrices with less than 10% unincorporated points the situation is reversed.

As for MKL-BCSR and PSC, we single them out as they do not produce results on the full experimental set. Averages are restricted to the corresponding set of valid experiments for each kernel. The MKL-BCSR implementation requires the block size to be a multiple of both the number of rows and columns of the matrix. If the two are co-primes, no feasible block size selection exists. For each matrix, we selected all feasible block sizes between 2 and 32, and show the one which obtains the best raw performance in Fig. 6. Results are obtained for 187 matrices, with an average performance on this subset of 1.75 GFLOPS (MKL: 1.89, UZP: 2.37). MKL-BCSR outperforms both UZP and MKL for a total of 30 matrices, on which the average number of spurious computation overhead (i.e., computations on explicit zeros introduced by BCSR) is kept below the 2x limit (average: 1.91x). The average MKL-BCSR performance on this subset is 2.91 GFLOPS (MKL: 2.02, UZP: 1.98). For the remaining 157 matrices, MKL-BCSR introduces on average 5.27x spurious FLOPs overhead, achieving an average performance of 1.52 GFLOPS (MKL: 1.86, UZP: 2.44).

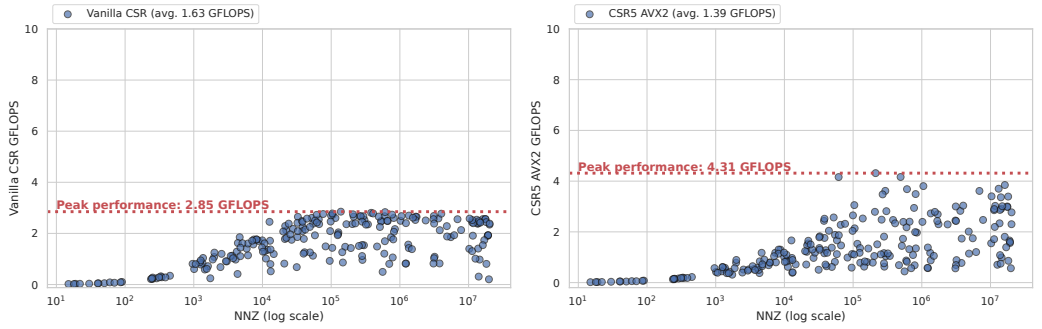
Regarding PSC, we modify the source code to normalize the experimental setup by introducing the PolyBench/C harness. We reproduce the cold cache environment in our tests: the matrix is read, the cache is flushed, and a single repetition of the SpMV kernel is run. Symmetric matrices are expanded for fair comparison with all other kernel versions in our setup. This approach produces results for a total of 166 matrices[2], obtaining an average performance of 1.71 GFLOPS (MKL: 1.97, UZP: 2.49). The main driver of performance is how well the sparsity structure of a matrix conforms to the predefined codelet shapes. PSC performs better than both MKL and UZP GenEx for 16 matrices, for which PSC emits only 2.15 instructions per FLOP (MKL: 2.43, UZP: 2.52) for an average performance of 3.51 GFLOPS (MKL: 2.64, UZP: 2.75). For the remaining 150 matrices it emits 22.83

[2]We suspect possible incompatibilities with some MTX formats on the SuiteSparse website.



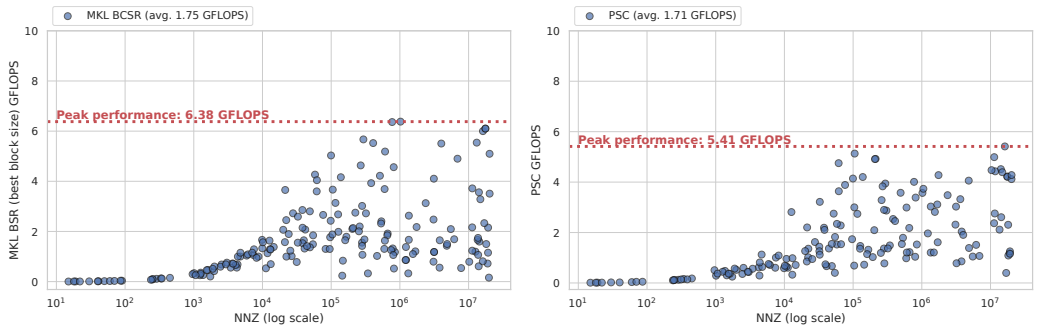
(a) MKL

(b) UZP GenEx



(c) Vanilla CSR

(d) CSR5 AVX2



(e) MKL-BCSR

(f) PSC

Fig. 6. 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.

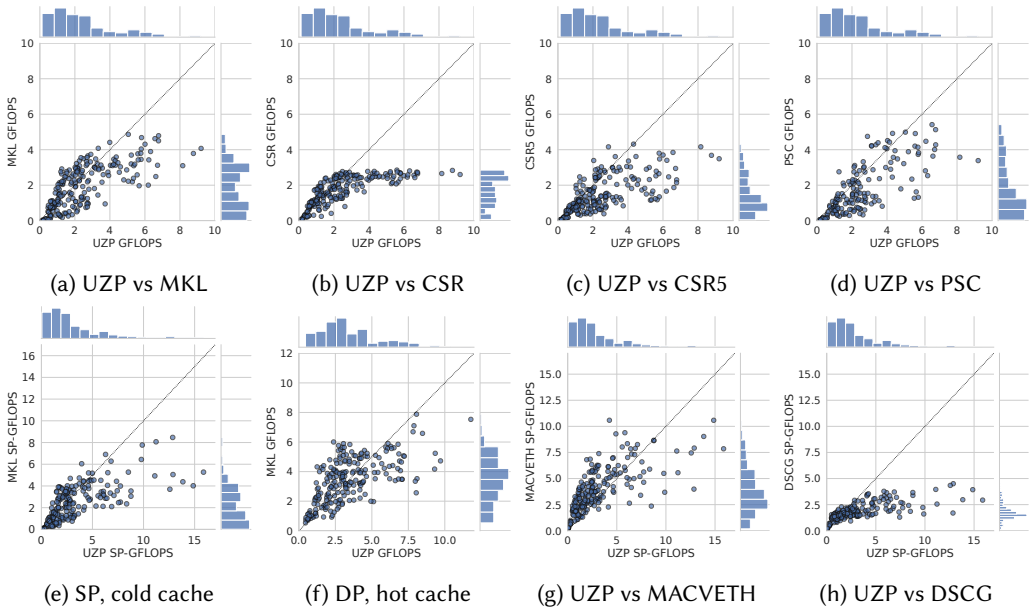
instructions per FLOP (MKL: 14.20, UZP: 5.03) for an average performance of 1.52 GFLOPS (MKL: 1.90, UZP: 2.46).

We draw generic performance considerations by carefully analyzing selected performance counters. The reasons for the superior performance of UZP GenEx with respect to MKL are essentially a better memory performance, as well as a more efficient vectorization. On the memory performance side, UZP incurs 0.29/0.016 misses per FLOP at the L2/L3 levels (MKL: 0.45/0.031, MKL-BCSR:0.65/0.072). As for vectorization, MKL uses on-the-fly zero-filling when traversing the sparse structure for emitting the operations, whereas GenEx employs the statically-provided UZP

information. As a result, MKL executes 19% more FLOPs than strictly required to compute the SpMV across the experimental set, precisely 1.38 billion FLOPs emitted vs 1.13 billion useful FLOPs. GenEx emits a much tighter 1.16 billion FLOPs.

On their side, CSR, CSR5, and PSC also present degraded memory behavior with respect to that of UZP GenEx in terms of L2/L3 misses per FLOP (CSR: 0.33/0.021, CSR5: 0.30/0.017, PSC: 0.52/0.023), and execute approximately 1.5x more instructions than UZP GenEx and MKL. All the FLOPs in the CSR kernel are scalar, whereas CSR5 emits almost exclusively 256-bit AVX2 instructions for a grand total of 1.44 billion FLOPs (a 27% excess).

The plot on Fig. 7a shows the joint distribution of the performance obtained for UZP GenEx vs. MKL. There are two main insights from this figure. The first one is that, as detailed in the previous paragraphs, the joint distribution is skewed towards the bottom of the diagonal line splitting the figure, i.e., the GenEx version performs better in general than the MKL one. The second has to do with the maximal performance obtained by each kernel. We developed the single-threaded roofline of this machine, and found the DRAM bandwidth to be 37.6 GB/s. Taking this into account, a double-precision two-stream computation with no memory reuse performing two FLOPs per iteration, which approximates the SpMV kernel in Listing 1, would be theoretically limited to 3.8 GFLOPS (37.6 GB/s multiplied by the arithmetic intensity of $^2/_{20}$ FLOPs/B). However, if we approximate the read of `x[:]` as a scalar, assuming an input matrix with a single dense column, this peak could be increased to 9.4 GFLOPS. According to this calculation, the maximum performance obtainable by a single-threaded SpMV kernel should lie somewhere between 3.8 and 9.4 GFLOPS, depending on the matrix sparsity structure, the computation schedule, and the sparse storage format. We observe maximum performances of 4.86 GFLOPS for MKL and 9.21 GFLOPS for UZP GenEx. The latter represents 97% of peak performance. The remaining subplots in Fig. 7 show the joint performance distributions of other kernels w.r.t. UZP GenEx.

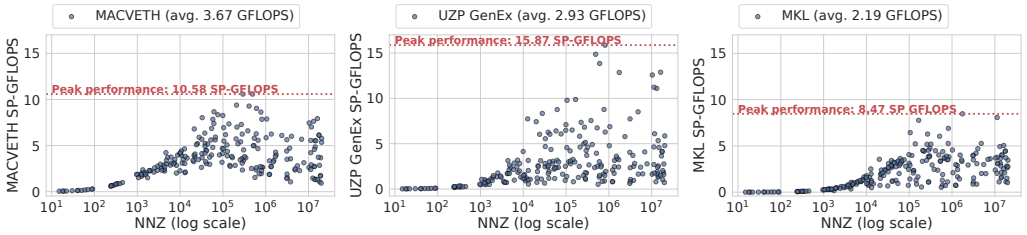


(a) UZP vs MKL (b) UZP vs CSR (c) UZP vs CSR5 (d) UZP vs PSC

(e) SP, cold cache (f) DP, hot cache (g) UZP vs MACVETH (h) UZP vs DSCG

Fig. 7. Joint performance (GFLOPS) plots of different kernel versions.

5.1.1 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 average 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. 8.



(a) MACVETH (b) UZP GenEx (c) MKL

Fig. 8. Performance (GFLOPS) of single-precision, cold cache kernels.

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. 7e.

5.1.2 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[3]). 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. 7f.

5.1.3 Data-Specific Codes. As detailed in Sec. 3.3, 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. 7g 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. [22] 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.

[3]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 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 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. 7.

5.2 Parallel Scaling

We experiment with double-precision, cold cache parallel workloads for the main four generic executors: UZP GenEx, MKL, CSR5, and PSC. For MKL, we linked against the threaded version of the MKL library. For the UZP generic executor, we parallelize the loop dispatching each piece of the sparse matrix using a `parallel for` OpenMP pragma, using static scheduling. Note that in this section, the results using 1-thread do not directly correspond to the single-threaded experiments in the previous section. These have been re-run explicitly using 1-threaded OpenMP versions, to account for the basic thread creation and management overhead.

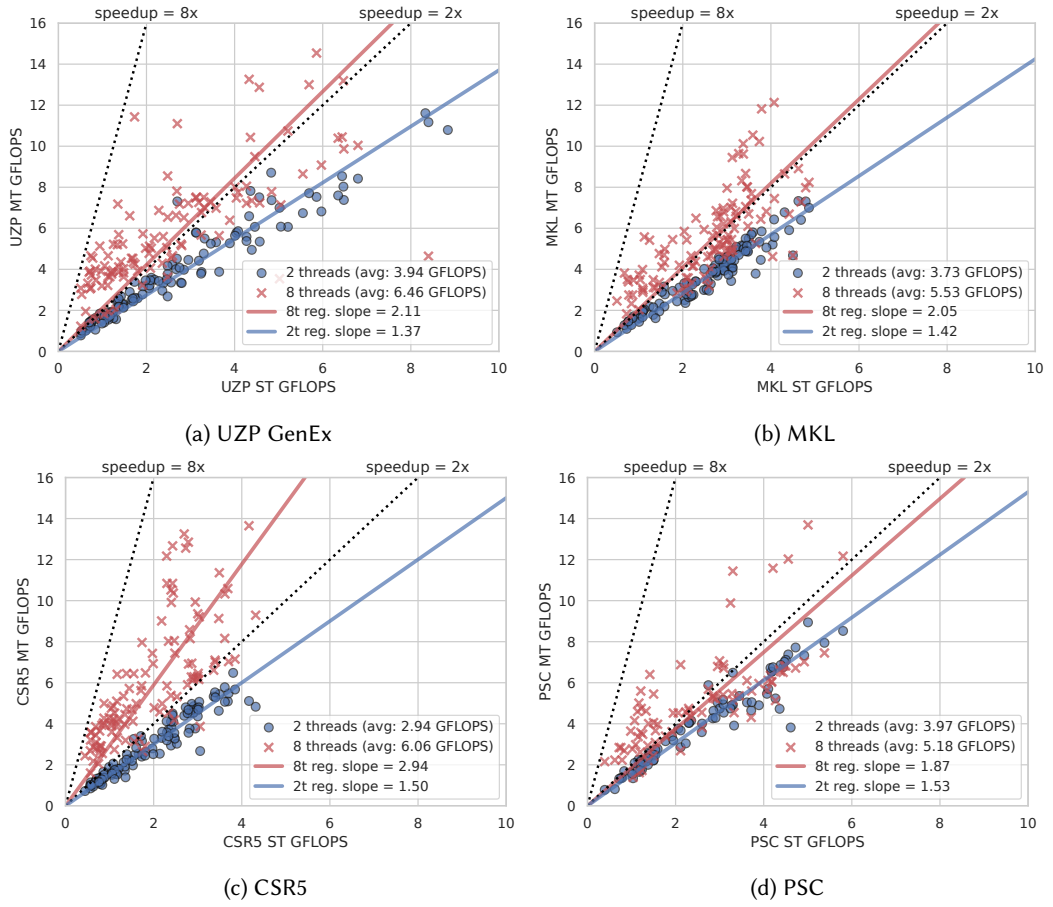
Figure 9 presents the obtained scaling data, restricted to matrices above 100K nonzeros to improve readability (smaller matrices achieve very small speedups and clutter the plot). UZP GenEx retains the best average parallel performances of 3.94 and 6.46 GFLOPS, respectively (MKL: 3.73 and 5.53, CSR5: 2.94 and 6.06, PSC: 3.97 and 5.18), with peak performances of 11.61 and 29.19 GFLOPS (MKL: 7.33 and 12.13, CSR5: 6.48 and 17.72, PSC: 8.94 and 13.69).

5.3 Performance Distribution

We complement our analysis by observing the relationship between the existence of diagonals in the matrix and the associated performance and compression achieved. Indeed, numerous matrices in SuiteSparse contain one or more diagonals. To better observe the effectiveness of UZP and Intel MKL, we compute a "diagonal" metric for each of the 229 sparse matrices as the ratio between the number of nonzeros in a diagonal divided by the total number of nonzeros. That is, a value of 1 gives a purely diagonal matrix, a value of 0 means no point along any diagonal occurs. For fast sampling of this metric, we limit to searching for pairs of points along a diagonal orientation, with a stride of 5. While more refined metrics can be designed, this simple criterion already provides insightful data shown below.

Fig. 10a displays the count of matrices as a function of the "diagonality" metric, split into ten ranges. We see that the dataset contains 67 matrices with at least 90% of points along the diagonal (last bar on the right), out of which 42 have more than 50K nonzeros, and 25 less than 50K. We also display the minimum, average, and maximum GFLOPS achieved in single-threaded, double-precision, cold cache setting for each cluster. On the left (black) is the Intel MKL performance, and on the right (red) is the GenEx performance. We observe that the raw performance mostly follows diagonality, with the maximum average performance for both MKL and GenEx reached for the highest diagonality.

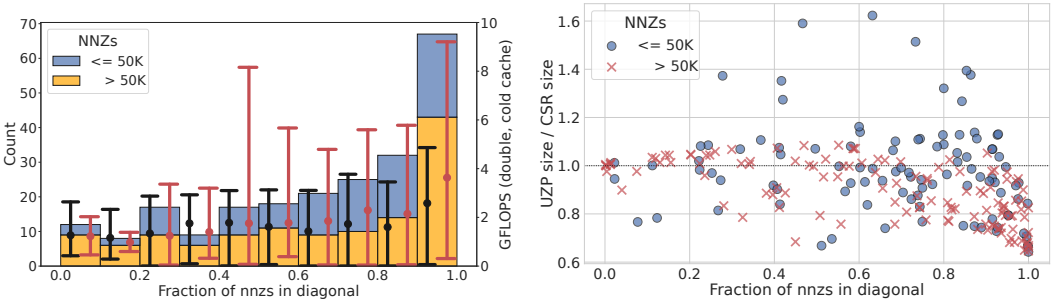
Fig. 10b displays the compression ratio of UZP vs. CSR (a value below 1 means the footprint of the final file in UZP is smaller than the CSR one), as a function of the "diagonality" metric. We



(a) UZP GenEx (b) MKL

(c) CSR5 (d) PSC

Fig. 9. 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.



(a) (b)

Fig. 10. Performance (left) and compression rates (right) as a function of matrix diagonality.

typically achieve good compression for more diagonal-based matrices, in particular for the larger ones, but compression can also be achieved over the entire spectrum.

Finally, Fig. 11 displays diagonality versus number of nonzeros (x-axis), showing a lack of correlation between both in the studied dataset, as dominantly diagonal matrices are occurring over the full range of matrix sizes.

To further study the performance distribution, we trained simple decision trees to predict the expected GFLOPS or potential to compress versus CSR. The dominant features for the decision are the number of nonzeros, the diagonality, and, as a second-order discriminant, the average number of nonzeros per row.

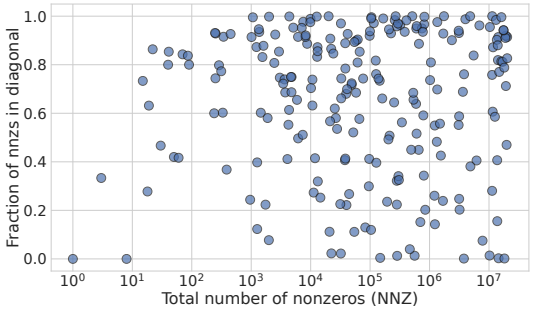


Fig. 11. Diagonality versus number of nonzeros.

5.4 UZP File Size Analysis

Finally, Fig. 12 shows a breakdown of file sizes into the proportion of its three main components: size of UZP metadata, size of the CSR/COO metadata if any, and size of the double-precision data values for the non-zero elements, which total to 100% of a UZP file (1.00 of the total file size). We also display the overhead that would be incurred by storing all unincorporated points as 1-point polyhedral shapes, instead of using a CSR/COO fragment for them, as bars above 1. That is, a value of 1.25 means 25% additional space would be needed to store the matrix without using CSR/COO encoding for unincorporated points. Except for the smaller matrices, the (incompressible) data values represent the largest proportion of the files. On aggregate, the UZP metadata takes up 17.5% of the total dataset storage size, compared to 9.9% for the CSR/COO fragment metadata and 72.6% for the data themselves.

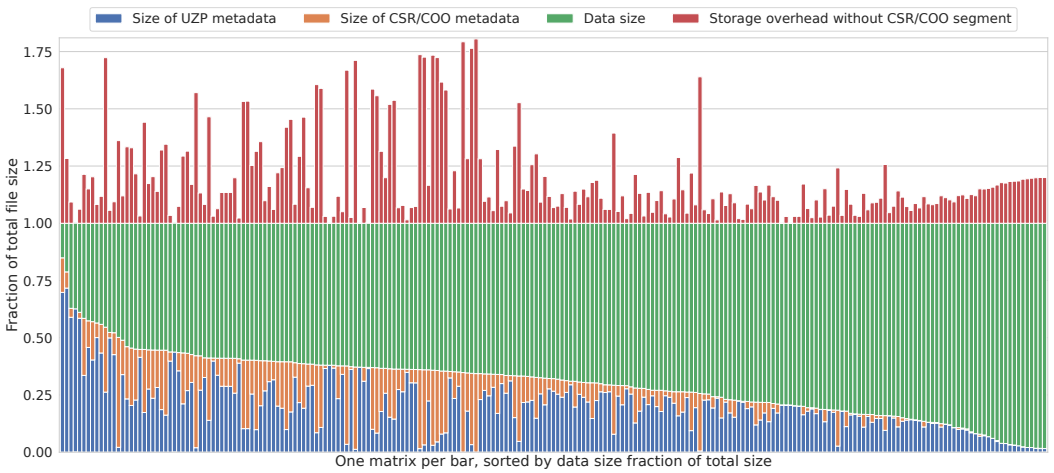


Fig. 12. Breakdown of the storage size of UZP files

We remark that UZP tuners geared towards minimizing the size of the metadata, along with using more general $\mathcal{Z}$-polyhedra shapes during the reconstruction process can further reduce the UZP file sizes, Fig. 12 displays the exact UZP files we evaluated for performance in this section.

6 Related Work

Saltz et al. [35] proposed runtime techniques for distributed memory parallelization of irregular applications [28, 35, 37]. These were augmented with compiler approaches that automatically generated parallel code [1, 15, 44]. Ravishankar et al. [32] exploit runtime regularity to produce polyhedrally-optimizable executor code in specific cases. Sukumaran-Rajam and Clauss [41] also detect runtime regularity using linear interpolation and regression models, selecting optimizations in a speculative fashion. Cheshmi et al. [12] develop an inspection technique to fuse two sparse kernels, generating parallel code optimized for locality and load balance.

The Sparse Polyhedral Framework [24, 39, 42] employed *uninterpreted function symbols* to (over-)approximate an irregular computation into a polyhedral one, which is ideal for generating I/E code at compile time. The same advantages and limitations occur: the generated code will be valid for any input sparse matrix, but will not exploit opportunities to customize the program for a specific matrix.

Sympiler [10, 11] is an I/E compiler that inspects the sparsity structure of an input to generate ad-hoc executor code. Indirection arrays are not completely removed, and may appear in loop bounds and access functions. The TACO compiler [14, 23] is a framework for generating optimized (sparse) tensor computations, supporting a large variety of sparse input formats and, in particular, composing different sparse formats and generating the associated executor programs.

Augustine et al. [2] proposed DSCG, and approach that mines for regularity in the sparsity structure of a matrix by computing the polyhedral shapes that capture the nzc, eventually generating data-specific codes that remove the use of indirection arrays. This approach exploits $\mathcal{Z}$-polyhedra to represent non-zero coordinates, and rely on the compiler auto-vectorization capabilites to optimize the matrix-specific executor file generated. Horro et al. [22] leveraged DSCG by developing a custom local rescheduling and advanced SIMD synthesis approach, including the fusion of independent reductions. MACVETH provides systematic performance improvements over DSCG, however these approaches lack the modularity and flexibility offered by UZP: one must compile the executor code for each target sparse matrix, leading to binaries which can have a size proportional to their nnz. In contrast, UZP develops *generic executors* whose size are independent of the nnz, and store the information about the polyhedral sets used to represent a given sparse structure in a separate file, which itself can be tuned and communicated easily.

Herholz et al. [20] unroll the computation into a graph that is then optimized to remove redundant computations and regroup expressions to take advantage of the hardware.Wilkinson et al. [47] apply unroll-and-jam followed by data compression to specific sparsity structures, enabling register tiling of SpMM computations. These techniques also leverage (a limited set of) $\mathcal{Z}$-polyhedral shapes to encode the computations.

Partially-Strided Codelets (PSC) have been proposed to efficiently execute sparse computations on multi-core CPUs [9]. PSC are also polyhedral shapes, albeit much more restricted than those considered by, e.g., Augustine et al. [2]. These can therefore also be represented in UZP. PSC focuses on a small number of possible codelets for which an efficient implementation is available (e.g., from existing libraries or via manual generation) and deploys a partitioning approach at inspection time to map nzc to these codelets, easing locality load-balancing for parallel computations. In contrast, UZP provides a flexible format to represent nearly arbitrary polyhedral shapes to encode a sparse structure, and provides a modular and decoupled approach to reconstruction, tuning (e.g., to favor locality or storage size compression) and execution. PSC [9] is tuned towards efficiently executing large sparse matrices on multi-core CPUs using a small number of predefined shapes, while in this work we prioritized scaling the reconstruction to a large number of candidate shapes, and tuning for smaller matrices (less than 20M nnz) and cold-cache / single-core situations. Indeed, UZP GenEx

currently outperforms PSC for about 76% of matrices tested in single-core, cold-cache configuration. Future work includes the development of tuners for other objectives, such as to improve scalability of multi-core execution and further improve data locality for additional performance.

While CSR remains the most common sparse storage format, many works have improved over its shortcomings, in particular to tune the storage format to specific computations. For example, Vuduc [45] presented an automated system for generating efficient implementations of SpMV on CPUs, while Williams et al. [49] moved toward multi-core platforms with the implementation of parallel SpMV kernels. For GPUs, Bell and Garland [4] implemented sparse matrix formats in CUDA and proposed the HYB approach (hybrid of ELL and COO).

Block-based formats [6, 13] such as BCSR, BELLPACK, or CSB are similar to UZP in that they exploit regularity. By design, blocks represent contiguous sets of coordinates, including storage for explicit zeros. Often the block size is constant for the entire matrix structure. Our approach does not have any of these restrictions. DCSR, RPCSR, and DCSC [7, 48] compress the index information to reduce the bandwidth consumed by the sparse computation. CVR [50] is a format focused on SpMV, simultaneously processing multiple rows within the input matrix. AlphaSparse [17] optimizes SpMV for GPUs, creating custom storage formats for a given sparsity structure. Note that many of these formats can also be represented in UZP.

7 Conclusion

We presented UZP, the Union of $\mathcal{Z}$-Polyhedra sparse format, and the associated tools to build UZP and compute with it. By proposing a modular approach to code generation, separating out the analysis of the sparse structure, tuning the representation for compression, and code generation strategies for, e.g., SpMV on CPUs, we removed several key limitations of monolithic prior works, and enable future research on the various components of this flow individually. UZP is a flexible format that encodes the sparsity structure as a union of integer polyhedra, each intersected with an affine lattice. It carries the benefits of using unions of dense computations to implement a sparse computation, while facilitating the generation of specialized sparsity-specific codes for improved performance. UZP seamlessly models strided dense sub-regions, subsuming other sparse formats.

In this work, we specifically targeted SpMV computations on multi-core CPUs, evaluating against a variety of formats and a highly-optimized matrix-specific SIMD code generation approach. Future work includes the development of advanced target-specific tuning strategies for CPUs, including fusion of small shapes for better SIMD occupancy, and a lightweight generic executor generator for sparse linear algebra expressions using UZP.

Acknowledgments

This work was supported in part by the U.S. National Science Foundation award #2009020; by grant PID2022-136435NB-I00, funded by MICIU/AEI/10.13039/501100011033 and by "ERDF A way of making Europe", EU; and by the predoctoral grant of Alonso Rodríguez-Iglesias FPU2022/01651, funded by the Ministry of Science, Innovation and Universities. CITIC, as a center accredited for excellence within the Galician University System and a member of the CIGUS Network, receives subsidies from the Department of Education, Science, Universities, and Vocational Training of the Xunta de Galicia. Additionally, it is co-financed by the EU through the FEDER Galicia 2021-27 operational program (Ref. ED431G 2023/01).

Artifact Statement

The latest version of the evaluated artifact is available under DOI: 10.5281/zenodo.15048005. It includes the necessary instructions, software, and scripts to reproduce the experiments in this paper. UZP is also available at https://github.com/UDC-GAC/uzp-sparse-format.

References

[1] G. Agrawal, J. Saltz, and R. Das. 1995. Interprocedural Partial Redundancy Elimination and its Application to Distributed Memory Compilation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI*. La Jolla, CA, USA, 258–269. https://doi.org/10.1145/223428.207157

[2] Travis Augustine, Janarthanan Sarma, Louis-Noël Pouchet, and Gabriel Rodríguez. 2019. Generating Piecewise-Regular Code from Irregular Structures. In *40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI*. Phoenix, AZ, USA, 625–639. https://doi.org/10.1145/3314221.3314615

[3] C. Bastoul. 2004. Code Generation in the Polyhedral Model Is Easier Than You Think. In *13th International Conference on Parallel Architectures and Compilation Techniques, PACT*. Antibes, France, 7–16. https://doi.org/10.1109/PACT.2004.1342537

[4] N. Bell and M. Garland. 2009. Implementing Sparse Matrix-Vector Multiplication on Throughput-Oriented Processors. In *ACM/IEEE Conference on High Performance Computing, SC*. Portland, OR, USA. https://doi.org/10.1145/1654059.1654078

[5] Uday Bondhugula, Albert Hartono, J. Ramanujan, and P. Sadayappan. 2008. A Practical Automatic Polyhedral Program Optimization System. In *29th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI*. Tucson, AZ, USA, 101–113. https://doi.org/10.1145/1375581.1375595

[6] Aydin Buluç, Jeremy T. Fineman, Matteo Frigo, John R. Gilbert, and Charles E. Leiserson. 2009. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In *21st Annual Symposium on Parallelism in Algorithms and Architectures, SPAA*. Calgary, AB, Canada, 233–244. https://doi.org/10.1145/1583991.1584053

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

[8] Kazem Cheshmi. 2023. Partially Strided Codelet GitHub repository. https://github.com/sparse-specialize/partially-strided-codelet Commit: c03d0593411c8afc9c6861de152695c453358a04.

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

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

[11] K. Cheshmi, S. Kamil, M. M. Strout, and M. M. Dehnavi. 2018. 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. https://doi.org/10.1109/SC.2018.00065

[12] Kazem Cheshmi, Michelle Strout, and Maryam Mehri Dehnavi. 2023. 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. https://doi.org/10.1145/3581784.3607097

[13] J.W. Choi, A. Singh, and R.W. Vuduc. 2010. Model-Driven Autotuning of Sparse Matrix-Vector Multiply on GPUs. In *15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP*. Bangalore, India, 115–126. https://doi.org/10.1145/1837853.1693471

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

[15] R. Das, P. Havlak, J. Saltz, and K. Kennedy. 1995. Index Array Flattening Through Program Transformation. In *ACM/IEEE Supercomputing Conference, SC*. San Diego, CA, USA. https://doi.org/10.1145/224170.224420

[16] Timothy A. Davis and Yifan Hu. 2011. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.* 38, 1 (2011). https://doi.org/10.1145/2049662.2049663

[17] Zhen Du, Jiajia Li, Yinshan Wang, Xueqi Li, Guangming Tan, and Ninghui Sun. 2022. 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. https://doi.org/10.1109/SC41404.2022.00071

[18] Takeshi Fukaya, Koki Ishida, Akie Miura, Takeshi Iwashita, and Hiroshi Nakashima. 2021. Accelerating the SpMV kernel on standard CPUs by exploiting the partially diagonal structures. *arXiv preprint arXiv:2105.04937* (2021). https://doi.org/10.48550/arXiv.2105.04937

[19] Gautam Gupta and Sanjay Rajopadhye. 2007. The Z-Polyhedral Model. In *12th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming, PPoPP*. San Jose, CA, USA, 237–248. https://doi.org/10.1145/1229428.1229478

[20] Philipp Herholz, Xuan Tang, Teseo Schneider, Shoaib Kamil, Daniele Panozzo, and Olga Sorkine-Hornung. 2022. Sparsity-Specific Code Optimization using Expression Trees. *ACM Trans. Graph.* 41, 5 (2022). https://doi.org/10.1145/3520484

[21] Changwan Hong, Aravind Sukumaran-Rajam, Israt Nisa, Kunal Singh, and P. Sadayappan. 2019. Adaptive sparse tiling for sparse matrix multiplication. In *24th Symposium on Principles and Practice of Parallel Programming, PPoPP*. Washington, DC, USA, 300–314. https://doi.org/10.1145/3293883.3295712

[22] Marcos Horro, Louis-Noël Pouchet, Gabriel Rodríguez, and Juan Touriño. 2022. Custom High-Performance Vector Code Generation for Data-Specific Sparse Computations. In *International Conference on Parallel Architectures and Compilation Techniques, PACT*. Chicago, IL, USA, 160–171. https://doi.org/10.1145/3559009.3569668

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

[24] A. LaMielle and M. Strout. 2010. *Enabling Code Generation within the Sparse Polyhedral Framework.* Technical Report CS-10-102. Colorado State University. https://www.cs.colostate.edu/TechReports/Reports/2010/tr10-102.pdf

[25] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling compiler infrastructure for domain specific computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization, CGO*. Seoul, South Korea, 2–14. https://doi.org/10.1109/CGO51591.2021.9370308

[26] Weifeng Liu and Brian Vinter. 2015. CSR5: An Efficient Storage Format for Cross-Platform Sparse Matrix-Vector Multiplication. In *29th ACM on International Conference on Supercomputing, ICS*. Newport Beach, CA, USA, 339–350. https://doi.org/10.1145/2751205.2751208

[27] Israt Nisa, Jiajia Li, Aravind Sukumaran-Rajam, Prasant Singh Rawat, Sriram Krishnamoorthy, and P. Sadayappan. 2019. An efficient mixed-mode representation of sparse tensors. In *International Conference for High Performance Computing, Networking, Storage and Analysis, SC*. Denver, CO, USA. https://doi.org/10.1145/3295500.3356216

[28] R. Ponnusamy, J.H. Saltz, and A.N. Choudhary. 1993. Runtime Compilation Techniques for Data Partitioning and Communication Schedule Reuse. In *ACM/IEEE Conference on Supercomputing, SC*. Portland, OR, USA, 361–370. https://doi.org/10.1145/169627.169752

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

[30] L.-N. Pouchet, G. Rodriguez, and colleagues. 2025. Artifact for PLDI'25 Modular Construction and Optimization of the UZP Sparse Format for SpMV on CPUs. https://doi.org/10.5281/zenodo.15240673.

[31] L.-N. Pouchet, G. Rodriguez, and colleagues. 2025. The UZP sparse format. https://github.com/UDC-GAC/uzp-sparse-format.

[32] M. Ravishankar, R. Dathathri, V. Elango, L.-N. Pouchet, J. Ramanujam, A. Rountev, and P. Sadayappan. 2015. Distributed Memory Code Generation for Mixed Irregular/Regular Computations. In *20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP*. San Francisco, CA, USA, 65–75. https://doi.org/10.1145/2688500.2688515

[33] G. Rodríguez, J. M. Andión, M. T. Kandemir, and J. Touriño. 2016. Trace-based Affine Reconstruction of Codes. In *14th International Symposium on Code Generation and Optimization, CGO*. Barcelona, Spain, 139–149. https://doi.org/10.1145/2854038.2854056

[34] G. Rodríguez, M. T. Kandemir, and J. Touriño. 2019. Affine Modeling of Program Traces. *IEEE Trans. Comput.* 68, 2 (2019), 294–300. https://doi.org/10.1109/TC.2018.2853747

[35] J. Saltz, K. Crowley, R. Mirchandaney, and H. Berryman. 1990. Run-time Scheduling and Execution of Loops on Message Passing Machines. *J. Parallel Distrib. Comput.* 8, 4 (1990), 303–312. https://doi.org/10.1016/0743-7315(90)90129-D

[36] Naser Sedaghati, Te Mu, Louis-Noel Pouchet, Srinivasan Parthasarathy, and P. Sadayappan. 2015. Automatic selection of sparse matrix representation on GPUs. In *29th ACM on International Conference on Supercomputing, ICS*. Newport Beach, CA, USA, 99–108. https://doi.org/10.1145/2751205.2751244

[37] S. Sharma, R. Ponnusamy, B. Moon, Y.-S. Hwang, R. Das, and J. Saltz. 1994. Run-time and Compile-time Support for Adaptive Irregular Problems. In *ACM/IEEE Conference on Supercomputing, SC*. Washington, DC, USA, 97–106. https://doi.org/10.1109/SUPERC.1994.344269

[38] Shaden Smith and George Karypis. 2015. Tensor-matrix products with a compressed sparse tensor. In *5th Workshop on Irregular Applications: Architectures and Algorithms, IA*^{3}. Austin, TX, USA. https://doi.org/10.1145/2833179.2833183

[39] M.M. Strout, G. George, and C. Olschanowsky. 2012. Set and Relation Manipulation for the Sparse Polyhedral Framework. In *25th International Workshop on Languages and Compilers for Parallel Computing, LCPC*. Tokyo, Japan, 61–75. https://doi.org/10.1007/978-3-642-37658-0_5

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

[41] A. Sukumaran-Rajam and P. Clauss. 2016. The Polyhedral Model of Nonlinear Loops. *ACM Trans. Archit. Code Optim.* 12, 4 (2016). https://doi.org/10.1145/2838734

[42] A. Venkat, M.S. Mohammadi, J. Park, H. Rong, R. Barik, M.M. Strout, and M. Hall. 2016. Automating Wavefront Parallelization for Sparse Matrix Computations. In *International Conference for High Performance Computing, Networking, Storage and Analysis, SC*. Salt Lake City, UT, USA. https://doi.org/10.1109/SC.2016.40

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

[44] R. von Hanxleden, K. Kennedy, C. Koelbel, R. Das, and J. Saltz. 1992. Compiler analysis for irregular problems in Fortran D. In *6th International Workshop on Languages and Compilers for Parallel Computing, LCPC*. New Haven, CT, USA, 97–111. https://doi.org/10.1007/3-540-57502-2_42

[45] R.W. Vuduc. 2004. *Automatic Performance Tuning of Sparse Matrix Kernels*. Ph. D. Dissertation. University of California.

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

[47] Lucas Wilkinson, Kazem Cheshmi, and Maryam Mehri Dehnavi. 2023. Register Tiling for Unstructured Sparsity in Neural Network Inference. *Proceedings of the ACM on Programming Languages* 7, PLDI (2023), 1995–2020. https://doi.org/10.1145/3591302

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

[49] S. Williams, L. Oliker, R.W. Vuduc, J. Shalf, K.A. Yelick, and J. Demmel. 2009. Optimization of Sparse Matrix-vector Multiplication on Emerging Multicore Platforms. *Parallel Comput.* 35, 3 (2009), 178–194. https://doi.org/10.1145/1362622.1362674

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

Received 2024-11-15; accepted 2025-03-06