UZP: Representing sparse matrices as polyhedra

Alonso Rodríguez-Iglesias*,
Gabriel Rodríguez*, Juan Touriño*,
Louis-Noël Pouchet†

* *CITIC, Universidade da Coruña, Spain*
† *Colorado State University, USA*

ABSTRACT

Sparse data structures are ubiquitous in modern computing, and numerous formats have been designed to represent them. These formats may exploit specific sparsity patterns, so that higher performance for key numerical computations can be achieved than when using more general-purpose formats, such as CSR or 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. UZP 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.
KEYWORDS: sparse linear algebra; sparse format; polyhedral compilation

1 Introduction

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

Sparse computations are typically implemented using specialized data structures, that can conveniently represent arbitrary sets of non-zero coordinates. They cope with the *irregular* nature of these sets of non-zeros, by contrast to the *regular* loops and accesses in dense computations. That is, the set of non-zero coordinates to operate on is typically stored explicitly, in contrast to dense computations where the coordinates are *computed* using e.g. affine functions of the loop iterators. Such approach enables 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 non-zero coordinates [KKC+17], and these executors must be *optimized* to the particular hardware targeted [VD+02, BG09, S+15].

[1]E-mail: alonso.rodriguez@udc.es

2 The UZP Sparse Format

2.1 Motivation and Prior Work

This novel approach to store sparse matrices is backed by many prior works. In one of them, Augustine et al. [ASPR19] demonstrated the feasibility of using polyhedra and lattices to compress a set of non-zero coordinates of a sparse matrix. Intuitively, a coordinate in the n-dimensional sparse data is represented as a $n+1$ integer tuple $(\vec{i}, data)$ where $\vec{i}$ is the n-dimensional non-zero 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 [GR07] 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 compute the actual nzc coordinates can be generated. Precisely, *polyhedral code generation* [Bas04] produces a dense, regular loop nest that when executing computes exactly the integer tuples captured by the $\mathcal{Z}$-polyhedra.

2.2 Design Principles

The new UZP format stores sparse matrices by representing its nzc and data as a collection of n-dimensional $\mathcal{Z}$-polyhedra. These polyhedra can be of any shape and dimensionality in the UZP file format, and although UZP supports any type of polyhedron, in this work we are just going to focus on 1-d polyhedra as a starting point.

Say we have a loop that traverses a matrix, accessing the values in coordinates: (1,1), (2,3), (3,5), (4,7), (5,9), (6,11), (7,13), (8,15). We can note that these 8 matrix accesses, starting from origin (1,1), have a constant stride of (1,2). It is clear that we could represent this access pattern as a collection of {origin, polyhedron$_{md}$}, where the m-dimensional[2] polyhedron has lattice and n elements: {origin, {lattice, n}}. This would translate into {(1,1), {(1,2), 8}$_{1d}$} and subsequently generate the following loop[3]:

```
for(int i = 0; i < 8; i++){
    <access to> A[1+1*i][1+2*i]
}
```

In order to represent a sparse matrix as a UZP (Union of $\mathcal{Z}$-Polyhedra) file, efficient pattern search (Sec 3) is crucial. To achieve that, we must code a high performance program that generates such pattern collection. As inputs to that program we have both the input sparse matrix, as well as a user-tunable pattern list, inside which we write the 1-d patterns we want to find in order of precedence.

2.3 Internal Representation and Data Compression

The UZP File Format is divided into several differentiated sections: Header, Dictionary of Shapes (DoSh), List of Origins (LO), Vector of not-Included/Incorporated Data (ninc data), Vector of Included/Incorporated Data (inc data). These sections and their contents are described more in depth in Figure 1.

Based on this generic representation we obtain substantial compression benefits over other storage systems such as CSC, CSR, BCSR, DIA, etc, as the UZP file can be tuned in such a way so that it uses the most optimal storage system for each non-zero distribution.

---

[2]Trailing md is for clarification purposes only. It is not necessary at all and can be easily inferred.

[3]Note that, although the for loop is encoding a 2-d sparsity structure, it uses a 1-d iteration polyhedron to scan it.

Header	DoSh	LO	ninc data	inc data
nnz, inc data len, nrows, ncols, dimensions, shapes, data_ptr...	{ shape_id: _, lenghts: _, strides: _, lattice: _, } ...	total origins [(shape_id, row, col, data_offset), ...]	type of encoding: { 0: CSR; 1: CSC; 2: COO } [data] { CSR: row_ptr, col_idx CSC: col_ptr, row_idx, COO: row_idx, col_idx }	data in the order of traversal described by the LO, sorted by shape_id, followed by the values pointed by the ninc data coords

Figure 1: UZP simplified file structure

3 Pattern Search

Pattern searching plays a crucial role in the generation of a UZP file. Efficient creation and distribution of UZP files are the most important parts of the UZP file generation process.

1. *File size*: in order to generate UZP files that achieve better compression ratios than typical compressed sparse matrix formats, we need to have a list of patterns to search, favouring common and long patterns over less common, shorter ones.

2. *Creation time*: we have implemented a fast, Rust-based tool that performs pattern checking in a scalable way. Figure 2 shows the relation between the size of the matrix and the export time for an input set of 230 matrices extracted from SuiteSparse[4].

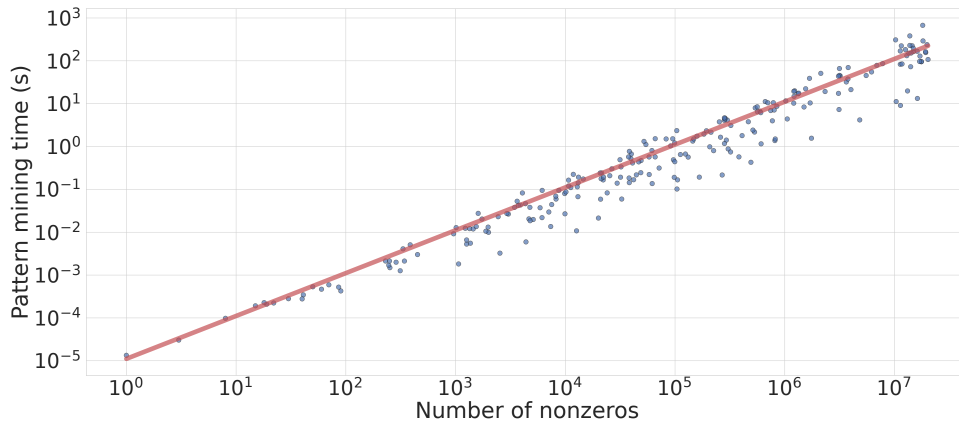

Figure 2: UZP file generation time

3.1 Pattern Search Algorithm

The pattern search algorithm traverses all cells in the sparse matrix, searching for user pre-defined patterns. A flowchart detailing the process of pattern search can be seen in Figure 3. When all base patterns are found, grouping and dimensionality augmentation, which are briefly defined below, but not discussed in depth in this work, becomes possible.

[4] `https://sparse.tamu.edu`

- *Grouping*: We call grouping to the union of one or more n-d polyhedron into a single n-d polyhedron, e.g. two consecutive (1-d) horizontal lines of sizes 8 and 8 can be merged into a single 16 element line, which translates into a 1-level for loop.
- *Augmentation*: We call augmentation to the union of one or more n-d polyhedron into a single {n+1}-d polyhedron, e.g. three non-consecutive lines of size 8, with their origins strided by (3,2), can become a 2-d polyhedron, translating into a 2-level for loop.

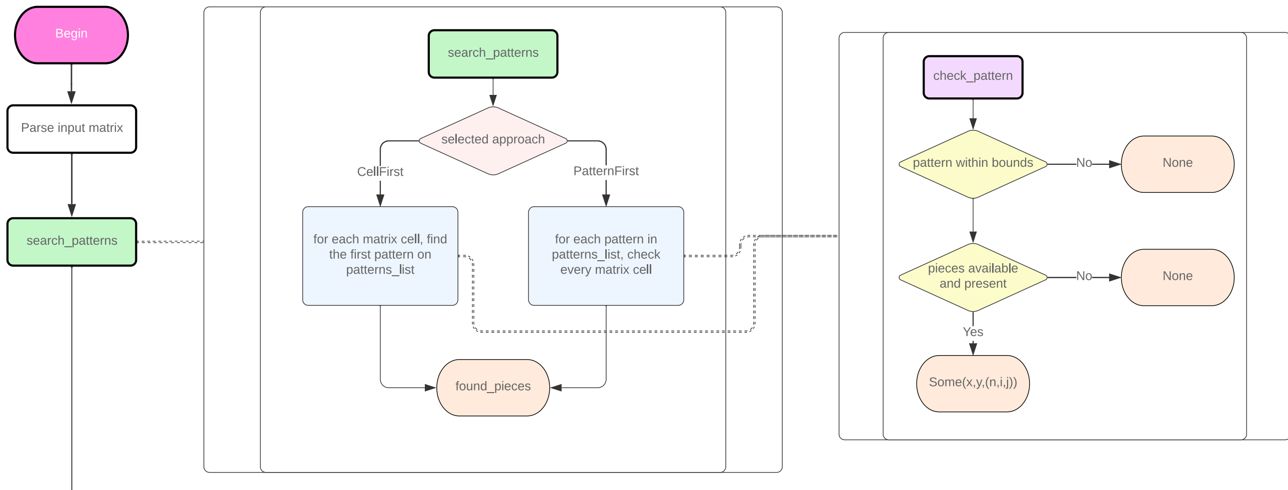

Figure 3: Initial pattern search algorithm

References

[ASPR19] Travis Augustine, Janarthanan Sarma, Louis-Noël Pouchet, and Gabriel Rodríguez. Generating piecewise-regular code from irregular structures. In *PLDI 2019*, page 625–639, 2019.

[Bas04] C. Bastoul. Code generation in the polyhedral model is easier than you think. In *13th PACT*, pages 7–16, Antibes, France, 2004. IEEE.

[BG09] N. Bell and M. Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *ACM/IEEE Conference on HPC, SC*, 11 2009.

[GR07] Gautam Gupta and Sanjay Rajopadhye. The z-polyhedral model. In *12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 237–248, 2007.

[KKC+17] F. Kjolstad, S. Kamil, S. Chou, D. Lugato, and S. Amarasinghe. The tensor algebra compiler. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):77, 2017.

[S+15] Naser Sedaghati et al. Automatic selection of sparse matrix representation on gpus. In *ACM on International Conference on Supercomputing*, pages 99–108, 2015.

[VD+02] Rich Vuduc, Demmel, et al. Performance optimizations and bounds for sparse matrix-vector multiply. In *SC'02*, pages 26–26. IEEE, 2002.

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).
Grant PID2022-136435NB-I00, funded by MCIN/AEI/ 10.13039/501100011033 and by "ERDF A way of making Europe", EU. Predoctoral grant of Alonso Rodríguez-Iglesias ref. FPU2022/01651, funded by the Ministry of Science, Innovation and Universities.