Towards Structured Algebraic Programming

Daniele G. Spampinato
Computing Systems Lab
Huawei Zurich Research Center
Zurich, Switzerland
daniele.giuseppe.spampinato@huawei.com

Denis Jelovina
Computing Systems Lab
Huawei Zurich Research Center
Zurich, Switzerland
denis.jelovina@huawei.com

Jiawei Zhuang
HiSilicon
Huawei Technologies Co., Ltd.
Shenzhen, China
zhuangjiawei@hisilicon.com

Albert-Jan N. Yzelman
Computing Systems Lab
Huawei Zurich Research Center
Zurich, Switzerland
albertjan.yzelman@huawei.com

Abstract

Structured matrices and tensors exhibiting properties such as symmetry and fixed non-zero patterns are known for making algorithms and data storage more efficient. Due to emerging power and efficiency constraints required by the scale of modern scientific, machine learning, and edge computing applications, algorithm experts are revisiting traditional linear algebra algorithms with the goal of making new structures appear. Such structures often result from new numerical approximations that would greatly benefit from a more flexible linear algebra interface than standard BLAS and LAPACK, allowing for mixed precision and data types to appear in place of traditional floating-point operations. Algebraic programming interfaces, like GraphBLAS, while naturally abstracting the algebras of their operations, do not explicitly capture structured, densely stored arrays. In this paper, we present a preliminary design of a new algebraic programming interface for structured containers with template-generic, non-zero patterns. This interface offers to backend implementations the possibility of integrating more compile-time pattern information in the loop-based implementation of primitive operations as well as in the formulation of array accesses. We demonstrate its ability to specify important dense matrix decomposition algorithms and argue its ability to expose high-performance backends.

CCS Concepts: • **Computing methodologies** → **Algebraic algorithms**; **Linear algebra algorithms**; • **Mathematics of computing** → **Mathematical software performance**.

Keywords: algebraic programming, mathematical software, dense linear algebra, structured matrices

ARRAY '23, June 18, 2023, Orlando, FL, USA

ACM Reference Format:
Daniele G. Spampinato, Denis Jelovina, Jiawei Zhuang, and Albert-Jan N. Yzelman. 2023. Towards Structured Algebraic Programming . In *Proceedings of the 9th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming (ARRAY '23), June 18, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3589246.3595373

1 Introduction

The GraphBLAS interface [9] enables implementing graph algorithms using linear algebraic operations [26]: graphs represented in terms of their sparse adjacency or incidence matrices and linear algebra primitives, such as matrix-vector (mxv) or matrix-matrix (mxm) multiplication, are used as algorithmic building blocks. Critically, GraphBLAS promotes an algebraic programming style: the algebraic structures (*e.g.*, semirings[1]) used by the primitives are explicitly passed as input arguments, enabling a small set of primitives to specify a large space of algorithms. For example, the same mxv primitive can be used to implement both a step of the single-source shortest path algorithm with the min-plus semiring (*i.e.*, $< \mathbb{R} \cup \{+\infty\}, \min, +, +\infty, 0 >$) as well as a step of k-hop reachability with the Boolean semiring (*i.e.*, $< \{T, F\}, or, and, F, T >$) [51].

Such an idea could also be adopted for expressing modern adaptations of dense linear algebra algorithms, using mixed floating- and fixed-point precision as well as integer arithmetic [10, 46]. Revisited functionalities may include LAPACK [5] algorithms, like the Cholesky, LU, and QR decompositions, eigensolvers, and linear systems solvers. These functionalities are typically expressed in terms of lower-level linear algebraic operations on structured input and/or output matrices, *i.e.*, containers with special non-zero patterns (*e.g.*, triangular and band matrices) and/or entry patterns (*e.g.*, symmetric and orthogonal matrices). Structured containers enable faster and more accurate algorithms, as well as more efficient memory storage schemes [19]. Capturing

[1]Denoted $< \mathbb{T}, \oplus, \otimes, 0, 1 >$, a semiring is a set $\mathbb{T}$ equipped with two binary operations $\oplus$ and $\otimes$ with identities 0 and 1, respectively (more in Sec. 3.1).

structured containers is therefore a critical requirement for an algebraic programming interface aiming at supporting current and future numerical applications. While sparse storage schemes, for example the compressed row and column storage (CRS and CCS), could represent structured containers (in particular structurally sparse containers such as diagonal and band matrices) dense applications best store matrices and vectors using compact layouts. Doing so avoids the memory indirection typically incurred by sparse storage schemes and achieves higher compression rates. Gareev et al. [18] demonstrate that high-performance optimization techniques used to optimize dense mxm (and more generally tensor contraction) [33, 47] can also be applied when the operation is generalized over semirings. However, the basic concept of a general dense container alone is not sufficient for the specification of a large number of LAPACK-Level functionalities.

To bridge this gap, this paper proposes the design of an algebraic programming interface where low-level algebraic primitives, like mxv and mxm, on generic semirings can be applied to densely-stored structured containers with generic patterns. We build our idea on the Algebraic Programming (ALP)/GraphBLAS [2], a modern C++ library mathematically equivalent to the GraphBLAS C interface, and dub our prototype framework ALP/Dense. It contributes:

- a new generic type for structured containers that decouples the formulation of container non-zero patterns (*e.g.*, tridiagonal), from the formulation of container storage schemes (Sec. 4.1). Container nonzero structure types allow for compile-time specialization of loop constraints to the non-zero pattern, while the storage scheme type allows the symbolic mapping of logical elements to their memory locations; and
- container views (*e.g.*, gathering a sub-matrix) can transform both the non-zero pattern and the container access function (Sec. 4.2). Chain of views could be further optimized, *e.g.*, the composition of static views (*i.e.*, views introducing runtime-invariant changes) may be performed at compile time.
- The paper, furthermore, shows that ALP/Dense allows implementing LAPACK-level functionalities, such as LU, QR, and Eigen decompositions (Sec. 5.1); and
- demonstrates that the presence of efficient building blocks for the ALP/Dense primitives results in ALP/-Dense algorithms competitive with highly-tuned, dense linear algebra libraries (Sec. 5.2).

2 Related Work

Mathematical interfaces, such as the BLAS [12, 13, 30], have pioneered the area of performance portability by exposing a relatively small set of primitive operations mainly responsible for the performance of scientific applications when executed on parallel systems (Sec. 3.2). Several vendor (*e.g.*, Intel oneMKL [25], Nvidia cuBLAS [34], and Huawei KML [24]) and open source (*e.g.*, OpenBLAS [3] and BLIS [47]) libraries exist, which provide highly-tuned implementations of the BLAS for modern CPUs and accelerators, including for small-sized input and output matrices (typically focusing on gemm or Level 3 operations, *e.g.*, libxsmm [23], BLASFEO [17], and libShalom [50]). However, the power and performance demands of modern applications, whether deployed on large exascale systems or small automotive and IoT edge devices, are pushing towards the limits of what a static library interface, like the BLAS, can offer [10, 47]. Enhancing the flexibility and automatic performance optimization of mathematical frameworks has therefore become a major area of investigation. Here, we briefly discuss some lines of work related to our own research direction.

Algebraic Programming Interfaces. The idea of capturing different algorithms by generalizing low-level numerical primitives over their algebraic structures has had noticeable impact in the sparse linear algebra and graph algorithms community [26, 41]. Implementations of the GraphBLAS [11] and equivalent C++ APIs, *e.g.*, ALP/GraphBLAS [4] and GBTL [1], enable generic operations on sparse matrix and vector containers, while the LLVM/Polly extension in [18] demonstrates that the generic dense tensor contractions can still leverage BLIS-like analytical modeling and optimizations [31, 47]. Our work builds on ALP/GraphBLAS extending its algebraic interface to dense, structured containers. The BLIS-based approach in [18] could be extended to take static information from the container types (*e.g.*, their non-zero patterns and storage schemes) into account.

Flexible Dense Linear Algebra APIs. Linear algebra libraries, for example, Armadillo [39], Eigen [21], xtensors [32], and Numpy/Scipy [22, 49], provide a tradeoff between performance and flexibility, offering a flexible C++ and/or Python interface to BLAS- and LAPACK-like functionalities with sparse and dense vector, matrix, and tensor containers. Some of these libraries (*e.g.*, Armadillo and Scipy) implicitly assume containers have properties when used with operations, such as matrix decompositions. Others support specific structured views on general containers (*e.g.*, Eigen supports `selfadjointView` and `TriangularView` methods on matrices). While certain optimizations on general containers are supported by most of these libraries (*e.g.*, lazy evaluation on general matrices), the lack of an explicit organization of containers' patterns often leads to suboptimal performance in the presence of more complex structured expressions [7, 36]. In contrast, ALP/Dense provides a mechanism for capturing and classifying patterns at the type level, making this information available at compile time for both correctness and performance optimization purposes.

Dense Linear Algebra Compilers. Several domain-specific compilers explicitly handle and take advantage of structured containers. Systems like Cl1ck [14] and Linnea [7] identify efficient ways to express dense linear algebra algorithms in terms of BLAS and LAPACK calls, driven by

patterns (which they call properties) of the structured matrices in the input expression. The concept of partially ordered set of non-zero patterns described in Sec. 4.1 is inspired by Linnea. LGen [44] describes a polyhedral-based approach to generate SIMD-vectorized C code starting from the non-zero and entry patterns of the matrices in an input basic linear algebra expression of fixed size. ALP/Dense also builds on a similar concept of non-zero pattern. It trades the flexibility of a full polyhedral description (at the price of involving expensive external libraries) with a more restricted class of patterns (captured via C++ metaprogramming) useful for expressing LAPACK-level algorithms. The Lift [35] compiler allows to define nested array types where inner arrays may have different sizes (including dependent on outer ones), capturing triangular matrices and stencils. GBTLX [38] extends GBTL with a SPIRAL-based [15] code generation backend for certain sparse optimization patterns supported by SPIRAL. Finally, [18] and [45] show how to integrate BLIS-level optimization of gemm-like operations within a general-purpose LLVM compiler infrastructure [28], while [8] expresses a similar optimization strategy with MLIR [29].

3 Background

In this section, we briefly introduce some basic concepts from ALP/GraphBLAS, the algebraic programming interface ALP/Dense builds on, and standard dense linear algebra APIs, which our work aims at capturing with a flexible interface.

3.1 ALP/GraphBLAS

ALP/GraphBLAS [2, 51] is a sequential, STL-compatible C++ programming interface mathematically equivalent to GraphBLAS. Its API exposes three major concepts: Opaque vector and matrix containers, algebraic structures (*e.g.*, binary operators, monoids, and semirings) and primitives, *i.e.*, a basic set of operations on containers, such as element-wise operations, mxv, and mxm. Monoids are algebraic structures denoted $M =< \mathbb{T}, \oplus, 0 >$ where $\oplus$ is an associative binary operator[2] on set $\mathbb{T}$ with identity $0 \in \mathbb{T}$, while semirings[2], denoted $S =< \mathbb{T}, \oplus, \otimes, 0, 1 >$, are formed combining a commutative, additive monoid (*e.g.*, M) with a multiplicative one $< \mathbb{T}, \otimes, 1 >$, where multiplication distributes over addition and multiplication by 0 annihilates elements in $\mathbb{T}$. Primitives compute on opaque containers using algebraic structures explicitly passed as additional input. Consider the following use example of `grb::foldl`[3] primitive:

```
1| grb::Monoid<
2|     grb::operators::min< T >,
3|     grb::identities::infinity
4| > Mon;
5| grb::Vector< T > x( n ), y( n );
6| ... // Building vectors x and y
7| grb::foldl( y, x, Mon );
```

which folds all elements in an ALP/GraphBLAS vector x into the corresponding elements from an input/output vector y using the `Mon` monoid, *i.e.*, $y_i = \min(y_i, x_i)$. Two remarks about the mathematical specification of this operation are useful in our context.

Opaque Containers and Backends. First, use of opaque ALP/GraphBLAS containers decouples the mathematical notion of vector exposed to the algorithm developers (here referred to as the users of the library) from the memory control aspects, which can be freely governed by the framework, *e.g.*, choosing a suitable sparse storage scheme. The user can control this at a high-level by compiling an ALP program targeting a specific backend. ALP backends encode different levels of parallelism that may provided by the hardware target, ranging from auto-vectorization to automatic shared- and distributed-memory parallelization, or a mixture of any of these. Backend choices are made via flags passed to the compiler. ALP/GraphBLAS provides a compiler wrapper for simplifying compilation. For example, the following command targets a shared memory-parallel backend:

```
$ alpcxx -b shmem -o app.x app.cpp
```

where the flag (i.e., `-b`) will trigger the inclusion of appropriate compiler and linker flags needed to target the chosen backend. The choice of an ALP/GraphBLAS backend drives the automatic optimization process of the framework, including making decisions about parallel memory allocation. Critically however, the user code remains unchanged and always looks like standard, sequential C++.

Algebras and Algebraic Type Traits. The implementation of an algebraic primitive, like the `grb::foldl` in the example above, can make sure that algebraic properties hold at compile time using type traits. For instance, if the selected operator is non-associative then a monoid cannot be formed and a compiler error is raised.

In ALP/Dense, we build on these two concepts of opaque containers and template-based type traits to leverage compile-time decisions on dense structured containers.

3.2 Dense Linear Algebra Libraries

The Linear Algebra Package (LAPACK) offers important linear algebra algorithms frequently occurring in scientific applications, including the Cholesky, QR, LU, eigenvalue, and singular value decompositions. LAPACK is one of the early example of performance portable libraries. Its algorithms are expressed in terms of lower-level operations: the Basic Linear Algebra Subprograms (BLAS). The BLAS defines three levels of basic dense functions, *i.e.*, vector-vector (Level 1), matrix-vector (Level 2), and matrix-matrix (Level 3) operations. Building LAPACK on top of BLAS (ideally in terms of the highest possible level, *i.e.*, BLAS 3) allows to exploit

[2] ALP/GraphBLAS allows an even more general formulation of the operators with different domain and range sets [51].

[3] Note that in ALP fold primitives are overloaded on the input/output types. In this example, folding applies M element-wise accumulating on the left operand. Were `y` a scalar, folding would reduce vector `x` into it.

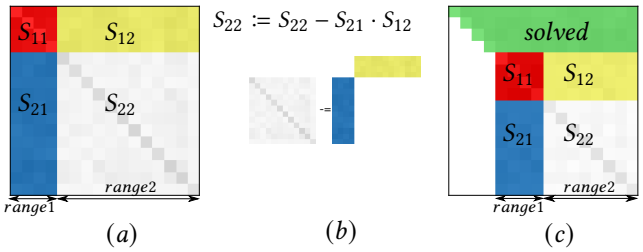

Figure 1. (a)–(b): Partitions of SPD matrix S during the first iteration of a blocked variant of the Cholesky decomposition (`BChol`). S_{11} and S_{22} are SPD, $Up(S_{11})$ is upper triangular. Since S is SPD, only $Up(S_{11})$ and S_{12} are updated (c)).

the cache hierarchy, instruction-level parallelism, and vector functional units of modern architectures.

We take the Cholesky decomposition to illustrate the typical layout of an LAPACK functionality as well as possible structured matrices appearing in this context. In particular, we consider the upper triangular, in-place, blocked variant of the Cholesky algorithm (in short `BChol`) as illustrated in Fig 1. `BChol` computes the (unique) factorization $U^T U = S$ of a real, symmetric (or complex, Hermitian) positive-definite matrix S, where U is an upper triangular matrix directly stored in the upper part of S (in-place version).

Each iteration of `BChol` consists of three updates of the upper part of the original matrix S as showed in Fig. 1:

1. Solve a smaller (unblocked) `UChol`:
$$S_{11} := \texttt{UChol}(S_{11}), \quad \text{Fig. 1(a) red update}$$
2. BLAS 3 triangular solve:
$$Up(S_{11}) \cdot S_{12} := S_{12}, \quad \text{Fig. 1(a) yellow}$$
3. BLAS 3 rank-k update:
$$S_{22} := S_{22} - S_{21} S_{12}, \quad \text{Fig 1(b).}$$

The algorithm progresses iteratively trough diagonal blocks from the top-left corner down to the bottom-right one. During the kth iteration, the kth row panel of the resulting matrix is calculated, while all previous ones have been already updated (green panel in Fig 1(c)).

Data Layouts. As noted in [47], the BLAS fixes a very strict storage layout for matrices, exposing it to the user. For example, matrices must be provided in column-major order (row-major is also an option in the analogous C interface CBLAS) and strides between elements are not allowed along a matrix's leading dimension. In ALP/Dense, the goal is to hide this complexity from the user, allowing the framework to flexibly express and control storage layouts.

4 Structured ALP/Dense Extension

In this section, we describe in more details the proposed ALP/Dense framework extension to support operations on structured containers. In particular, we will focus on two major aspects. First, their internal representation in ALP. Second, their role in the implementation of ALP/Dense primitives with two representative ALP/Dense backends, *i.e.*, the reference sequential and parallel backends.

4.1 Structured Containers

An ALP/Dense structured container accepts additional template parameters for conveying non-zero patterns' information. As mentioned in Sec. 1, in this paper we focus on dense patterns. For example, an upper triangular matrix of size $n \times n$ with elements in `T` can be declared as follows:

```
1| alp::Matrix<
2|     T, alp::structures::UpperTriangular,
3|     alp::Dense
4| > U( n );
```

Listing 1. Declaration of an upper triangular matrix $U \in \mathbb{T}^{n \times n}$ in ALP (we may drop the *alp* namespace in future examples for the sake of clarity).

In the code snippet above, Line 3 selects one of the predefined non-zero patterns supported by ALP, while Line 4 specifies that the non-zero area in the upper triangular part of the matrix is dense, *i.e.*, a static mapping between any element in that area and a location in memory should be provided by the framework at compile time. Notice that nothing says whether the zero part of the matrix should be stored or not, or whether non-zero elements follow a row-major order. As mentioned in Sec. 3.1, ALP/Dense leverages opaque containers to maintain full control on the layout of the data and their access. The pattern and density arguments are handles provided to the ALP/Dense user to specify the mathematical layout of the container. Any storage layout decision is delegated to ALP/Dense and may be based, *e.g.*, on the choice of a specific backend as discussed later in this section.

Remarks on ALP/Dense Vectors and Scalars. In the remainder of the paper, we will mostly focus on matrices for introducing and explaining new concepts. However, unless stated otherwise, those concepts equally apply to ALP/Dense vectors and scalars. For the sake of our discussion, a vector of length n and a scalar can be intuitively treated as special matrix instances of size $n \times 1$ and 1×1, respectively. Next, we explain how non-zero patterns are organized in ALP/Dense and internally associated, by the framework, with a particular storage scheme.

ALP/Dense Non-Zero Patterns. ALP/Dense represents non-zero patterns as C++ empty classes. The goal of such classes is to capture key geometric features of patterns and logical implications between them for compile-time optimizations and error handling. Currently, ALP/Dense statically describes the following properties of a non-zero pattern:

1. **Band Intervals**: The location of one or more non-zero band intervals, where a band interval defines as an area of the matrix between two diagonals $[\ell, u)$, $\ell, u \in \mathbb{Z}$, $\ell < u$. Diagonal tags are centered around the main diagonal (the 0-diagonal) and specified as

template arguments. For example, a tridiagonal matrix has a single non-zero band interval $[-1, 2)$. If a band is only limited by the size of a matrix on one side, ALP/Dense represents its interval as open-ended on that side. For example, we denote the band interval of the upper part of an upper triangular matrix as $[0,)$. A general rectangular matrix is defined as having a completely open-ended interval: $(,)$. Finally, a tuple of intervals could be used to capture the union of multiple bands. *e.g.*, the matrix $\begin{bmatrix} 0 & a & 0 \\ b & 0 & c \\ 0 & d & 0 \end{bmatrix}$ is associated with the tuple $([-1, 0), [1, 2))$. This idea is similar to the polyhedral concept of structural information in [44]. By limiting the scope of representation to a union of band intervals, however, we make processing this information via metaprogramming practical, while still capturing important non-zero patterns appearing in linear algebra applications.

2. **`is_a`-Relation:** Intuitively, if a non-zero pattern S_B has all properties of a pattern S_A we can invariably state that a matrix with pattern S_B is also a matrix with pattern S_A. For example, an upper triangular matrix is both a square and an upper trapezoidal matrix. Such `is_a`-relation on non-zero patterns forms a partially ordered set [7] that can be used to verify runtime-invariant predicates via C++ type traits as part of ALP/Dense implementations.

Figure 2 summarizes the two properties of non-zero patterns captured in ALP/Dense. Currently, the interface comprises major non-zero patterns required by the algorithms discussed in Sec. 5.1, such as tridiagonal and trapezoidal ones. Entry patterns, such as symmetric and orthogonal, can also be included in the `is_a` poset and used as a pattern argument in the declaration of a container. ALP/Dense internal developers may use them to identify static optimization opportunities (*e.g.*, always accessing the same part of a symmetric matrix S due to the property $S_{i,j} = S_{j,i}$) via type traits. Notice that capturing explicitly the entry relation of such patterns (*e.g.*, in a similar way to the access functions discussed shortly for non-zero patterns) may enable the framework to apply some of these optimization automatically. We plan to investigate this in future work.

As previously discussed, the non-zero pattern of a structured container is specified by the user without making any decision about the way data is stored in memory. This choice is delegated to the ALP/Dense framework as we now explain.

Selecting an ALP/Dense Storage Scheme. An ALP/Dense storage scheme is an internal component of the framework that interfaces the mathematical layout of a container and its memory instantiation. The process of selecting a storage scheme is both backend- and pattern-specific. The backend choice influences how containers are distributed

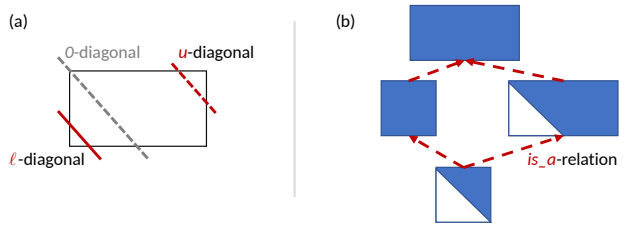

Figure 2. Properties of a non-zero pattern in ALP/Dense: (a) its static band intervals, denoted $[\ell, u)$ with respect to the main diagonal; (b) The `is_a`-relation among available non-zero patterns, here exemplified by the set of general, square, upper trapezoidal, and upper triangular patterns.

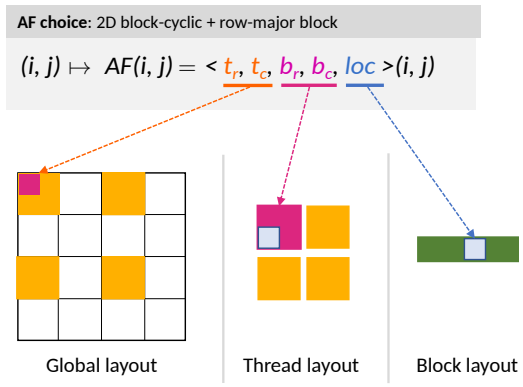

Figure 3. Possible choice of a hierarchical access function (AF) for a dense, general matrix when targeting the combination of shared-memory parallel and sequential reference backends: A 2D block-cyclic storage in shared memory, where each local block is stored in row-major order. The *loc* coordinate is expressed as a storage polynomial.

and allocated across threads as well as the way they are accessed. The latter is described via an access function (AF), *i.e.*, a function that maps non-zero coordinates of a container to a physical location in memory (*e.g.*, $(i, j \mapsto AF(i, j)$ for a matrix). The range of the AF is generally multidimensional. For example, as shown in Fig. 3, the ALP/Dense reference backend currently defines a flat 1D memory layout while its combination with the shared-memory backend adds additional dimensions to enable a 2D block-cyclic layout.

Dense ALP/Dense Access Functions. The non-zero pattern of a container may also affect the choice of the storage scheme as certain patterns (*e.g.*, the upper part of the matrix declared in Listing 1) may be stored more compactly by allocating memory only for non-zero elements. To capture this, the innermost dimension of an AF normally associated with local storage is defined in ALP/Dense using polynomial objects, dubbed storage polynomials. For example, the generic storage polynomial declared below:

```
1| using namespace alp::internal;
```

```
  storage::polynomials::BivariateQuadratic<
      Ai2, Aj2, Aij, Ai, Aj, A0, D
  > P( ai2, aj2, aij, ai, aj, a0 );
```

instantiates the following bivariate, quadratic polynomial:

$$P(i,j) = \frac{1}{D}\left(A_{i2}a_{i2}i^2 + A_{j2}a_{j2}j^2 + A_{ij}a_{ij}ij + A_i a_i i + A_j a_j j + A_0 a_0\right) \quad (1)$$

where i, j, A_*, $a_* \in \mathbb{Z}$ and $D \in \mathbb{Z} \setminus 0$. Such polynomials are expressive enough to capture classic LAPACK storage schemes. Template coefficients A_* enable compile-time optimizations (*e.g.*, dropping zero terms) and the denominator D allows representation of a compact storage, such as the LAPACK's packed storage. For example, the linear 2D polynomial $P_R(i,j) = in + j$, associated with a conventional storage for an $m \times n$ matrix in row-major order, is defined as:

```
BivariateQuadratic< 0, 0, 0, 1, 1, 0, 1 >
  P_R( 0, 0, 0, n, 1, 0);
```

while the polynomial $P_U(i,j) = (j^2 + 2i + j)/2$ defined below:

```
BivariateQuadratic< 0, 1, 0, 2, 1, 0, 2 >
  P_U( 0, 1, 0, 1, 1, 0 );
```

may be selected as a packed, column-major storage format for the matrix in Listing 1. Finally, notice that union of k bands could be associated with a piecewise, bivariate, quadratic polynomial with exactly k intervals.

Building on ideas from domain-specific frameworks, such as SPIRAL [15, 16] and the polyhedral isl library [48], ALP/Dense storage polynomials are represented symbolically to enable further optimizations in the presence of container views as we discuss next.

4.2 ALP/Dense Views

Views enable casting a different perspective on ALP/Dense containers (or part of them) without additional memory allocation. Examples of views are gathering a sub-matrix, transposing a matrix, and permuting its rows and/or columns. Table 1 lists view categories currently supported in ALP/Dense. We differentiate between three categories of views: static, dynamic, and functor views. Next, we introduce each category in more details beginning with static views which may affect the access function of a container at compile-time.

Static Views. Static views affect the internal interface to an ALP/Dense container at compile-time by (i) redefining its non-zero pattern and/or (ii) redefining its AF.

Statically Affecting the Non-Zero Pattern. Consider the static view interface in Tab. 1(a), which could be used to define, for instance, an upper triangular view of a general square matrix. This type of view is allowed in ALP/Dense as long as the target non-zero pattern is more specialized than the input one, according to the `is_a`-poset discussed in Sec. 4.1 and shown in Fig. 2 for the upper triangular example. The `alp::get_view` assesses the validity of a view only when the verification step can be carried out at compile-time based on the `is_a`-poset or at runtime if doable in constant time. For example viewing a general $m \times n$ matrix as square only requires checking that $m = n$. More expensive checks (*e.g.*, whether a full-rank view of a square matrix is valid) may be defined and activated in ALP/Dense for debugging purposes. Finally, notice that a view that changes the non-zero pattern of a container may redefine the semantics of a container but has no effect on its AF. For example, a triangular view of a square matrix may involve accessing fewer elements but the location of those elements would be identical to (and defined by) the original square pattern.

Statically Affecting the Access Function. Static views, such as those listed in Tab. 1(b)–(f), may affect the AF of a target container, including the formulation of its storage polynomial. Take for instance the gather view in Tab. 1(b), which allows the selection of a sub-container, optionally explicitly changing its non-zero pattern. For matrices, the elements in a submatrix are selected at runtime using row and column (strided) ranges as shown in Fig. 4(b). Such range information can be statically fused into the storage polynomial. In particular, a range `alp::range(b, M, s)` selects $m = \lceil (M-b)/s \rceil$ elements within range $[b, M)$ at stride s. As an affine map, it is expressed as follows:

$$\{0, \ldots, m-1\} \to \{0, \ldots, M-1\}; i \mapsto b + si. \quad (2)$$

Suppose we use it to select m rows from the $M \times N$ matrix `A` as shown below:

```
Matrix<T, structures::General, Dense> A(M, N);
auto gA = get_view(A, range(b, M, s), range(N));
```

where matrix `A` is initially assigned a row-major format with storage polynomial $P_R(r,c) = rN + c$ and `alp::range(N)` reduces to the identity map $\{0, \ldots, N-1\} \to \{0, \ldots, N-1\}; j \mapsto j$. Based on the code above, ALP/Dense can optimize the polynomial storage of view `gA` yielding the equivalent of the polynomial declaration below:

```
BivariateQuadratic< 0, 0, 0, 1, 1, 1, 1 >
  P_gA( 0, 0, 0, s * N, 1, b * N);
```

This formulation is obtained from the following composition of the strided and identity maps on rows and columns, respectively, and the polynomial P_R:

$$P_R(r = b + si, c = j) = (b + si)N + j = sNi + j + bN. \quad (3)$$

Differently from the view in Tab. 1(a), which is analogous to applying a full range to both rows and columns, the validity of a target non-zero pattern of a sub-container can only be checked at runtime as it always depends on the range information. Consider for instance Fig. 4(a) that shows possible examples of valid submatrix views with change in non-zero pattern on an upper triangular matrix. The general submatrix (black box) is valid only if the coordinates of the bottom-left and top-right corners of the submatrix fall within the band interval of the source upper triangular matrix. Also, in this scenario, only constant-time checks can be defined unless in debug mode.

Table 1. Selection of ALP/Dense views [A] and equivalent Eigen [E] expressions. **(a, b, g)**: [A] M_1 is a container view with any supported `DstStr` pattern (*e.g.*, `UpperTriangular, Symmetric, Tridiagonal, etc.`) valid according to `is_a`-relation with `M`. [E] `xxx` ∈ {`triangular, selfAdjoint`} and `X` any acceptable argument for `xxx` (*e.g.*, `triangularView< Upper >`). **(b, c, d)**: [A] `rng_*` an `alp::range(beg, end, inc)` . [E] `seq_*` a `Eigen::seq(beg, end, inc)` . **(g)**: [A] `p_*` of type `alp::vector< size_t/int/... >` . [E] `p_*` of type `std::vector< size_t/int/... >` or several other acceptable array-like containers. **(h)**: [A] Functor views overload low-rank primitives, `alp::outer` is used here as a possible example. M_2 has `Symmetric` or `General` pattern depending on `alp::is_commutative< ⊗ >::value`. [E] The use of `noalias()` enables lazy evaluation of the outer product. Assume `v` is of type `Eigen::Matrix< T, Dynamic, 1 >`. Implementing `T.operator*` according to the semantic of ⊗ would mimic the use in [A]. Note that varying operator in [A] may require varying type `T` in [E]. Symmetry has to be explicitly expressed if desired, *e.g.*, via `selfadjointView`.

Category	ALP/Dense Interface	Eigen-equivalent	
Static	M_1 = `get_view< DstStr >( M );`	`M.xxxView< X >();`	(a)
	M_1 = `get_view< DstStr >( M, rng_r, rng_c );`	`M( seq_r, seq_c ).xxxView< X >();`	(b)
	`v = get_view( M, rng_r, c );`	`M( seq_r, c );`	(c)
	`v = get_view( M, r, rng_c );`	`M( r, seq_c );`	(d)
	M^t = `get_view< transpose >( M );`	`M.transpose();`	(e)
	`d = get_view< diagonal >( M );`	`M.diagonal();`	(f)
Dynamic Functor	M_1 = `get_view< DstStr >( M, p_r, p_c );`	`M( p_r, p_c ).xxxView< X >();`	(g)
	M_2 = `alp::outer(v, ⊗ );`	M_2`.noalias() = v * v.transpose();`	(h)

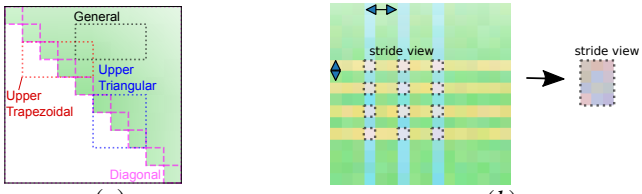

Figure 4. (a) Static views on an upper triangular matrix (*e.g.*, a general (black box) and upper triangular view (blue box)). Non-zeros are shown in green, while zeros are in white. (b) A strided matrix view (on the right) obtained selecting rows and columns at a stride from the original matrix (left in (b)).

Chaining Static Views. The same fusion optimization process also takes place when three or more static views are chained. ALP/Dense currently supports tree-like view chains as part of a single scope, where new views are fully contained within previously defined ones without overlapping with more than one view.

Dynamic Views. Dynamic views are a second category of views supported by ALP/Dense which enable permuting the content of a container based on runtime information. The view in Tab. 1(g) takes as an input three ALP/Dense containers, *i.e.*, a matrix `A` and two vectors `p_r` and `p_c`. Vector `p_r` (`p_c`) contains integer elements and is interpreted as a permutation applied to the rows (columns) of matrix `A`. Figure 5 shows an example of dynamic view (right matrix) with row permutation. In particular, the linear algebraic semantic of this view can be expressed as the multiplication of the permutation matrix corresponding to the permutation `p_r` with matrix `A`:

$$P_{p_r} A = \left[e_{p_r(0)}, \ldots, e_{p_r(M-1)} \right]^T A, \tag{4}$$

where $A \in \mathbb{T}^{M \times N}$ and e_i is the ith standard basis vector of $\mathbb{T}^M$. The general view in Tab. 1(g) (with two permutation vectors) permutes both rows and columns of `A`: $P_{p_r} A P_{p_c}^T$.

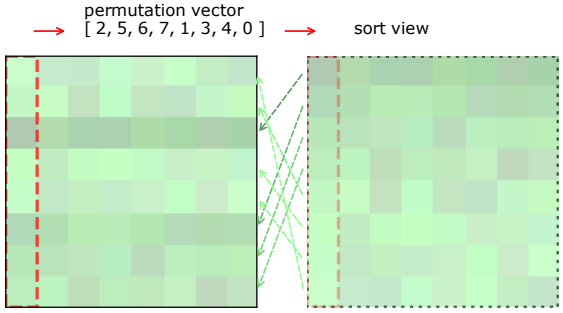

Figure 5. The right matrix shows a dynamic row-permuting view of the left matrix according to the permutation vector p_r. Vector p_r was generated by the `alp::sort` primitive applied to the the first column of the matrix on the left.

Permutation vectors are ALP/Dense containers and may be computed using ALP/Dense primitives. For example, the `alp::sort` function may be used to compute the sorting permutation p_r of the first column of the left matrix in Fig. 5 according to a given ALP/Dense partial order $\geq \subseteq \mathbb{T}^2$.

Chaining Static and Dynamic Views. Dynamic and static views may be chained. Interposing dynamic views between static ones breaks a view chain into a number of static sub-chains to which the fusion mechanism described above may be separately applied.

Functor Views. Functor views are overloaded ALP/Dense primitives which return views on low-rank output. Table 1(h) shows the example of the outer product primitive between

a vector v and its own transpose using an operator $\otimes$. The output its a view on the resulting matrix S. Notice that matrix S may have generic or symmetric pattern depending on the commutativity of operator $\otimes$. If matrix S is just an intermediate temporary container with no substantial reuse, capturing it as a view can save on memory allocation while computing its elements according to the semantics of the primitive (*i.e.*, $S_{i,j} = x_i \otimes x_j$) can be performed on the fly via C++ lambda functions (functors) whenever required by subsequent accesses. Notice, that the `alp::outer` functor view is used just as a representative example. Similarly, a number of other low-rank and element-wise (*e.g.*, `alp::conjugate`) ALP/Dense primitives may provide overloaded views.

Materializing a View. If needed, any kind of views can be materialized in memory via a call to the `alp::set` primitive. The output container needs to match the type (including element type and non-zero pattern) of the input view. Static and dynamic views are also acceptable output containers as they address valid memory.

4.3 ALP/Dense Primitives

In the remainder of this section, we describe how structured containers and views can support the implementation of ALP/Dense primitives. In particular, we will discuss their role in the context of both the reference sequential and parallel ALP/Dense backends.

Sequential Backend. Non-zero patterns associated with structured containers can be used to specialize ALP/Dense primitives, consequently minimizing the amount of redundant computation and data access. For example, consider the reference implementation of the `alp::foldl(A, B, Mon)` primitive applied to two banded matrices $A, B \in \mathbb{T}^{M\times N}$ with a monoid Mon $:=< \mathbb{T}, \oplus, 0 >$. Assuming A and B have band intervals $[l_A, u_A)$ and $[l_B, u_B)$, respectively, the primitive computes the following:

$$A_{i,j} := A_{i,j} \oplus B_{i,j},$$
$$\left\{(i,j) \in \mathbb{Z}^2 \mid \begin{array}{c} 0 \le i < M, \\ \max(0, i + l_B) \le j < \min(N, i + u_B) \end{array}\right\}, \quad (5)$$

granted that the following condition:

$$[l_B, u_B) \subseteq [l_A, u_A), \quad (6)$$

holds on the band intervals. Notice that A is not assigned outside of (5), as $A_{i,j} \oplus 0 = A_{i,j}$. Condition (6) can be verified at compile time via type traits along with other conditions on the validity of the monoid (Sec. 3.1). The iteration space described in (5) can be constructed precisely by the primitive's loops via type traits on the type of B, thus only accessing data strictly necessary for the computation. Internally, each logical access to $A_{i,j}$ and $B_{i,j}$ is translated into storage coordinates via the AF associated with each container (for example, the AF in Fig. 3(a) assuming that a row-major order storage is selected for the two matrices).

Similar logic can be applied to the combination of supported structured containers and all other ALP/Dense primitives enabling the synthesis of tight iteration spaces and associated memory maps. In Sec. 6 we will discuss how we plan to take this type information into account in the development of future high-performance sequential backends with compiler support.

Shared Memory-Parallel Backend. When targeting a parallel ALP/Dense backend, such as the shared-memory one, the AF becomes hierarchical (as shown in Fig. 3(b)) with a coordinate space that depends on the chosen data distribution. ALP/Dense provides an additional internal view mechanism for developers of primitives, which allows referencing local blocks as sequential ALP/Dense containers. For example, consider implementing the primitive `alp::eWiseMul(A, alpha, beta, Sr)` with the following specification:

$$A := A \otimes \alpha \oplus \beta, \quad A \in \mathbb{T}^{M\times N}, \alpha, \beta \in \mathbb{T}, \quad (7)$$

under the semiring $Sr :=< \mathbb{T}, \oplus, \otimes, 1, 0 >$. Suppose the target backend is a combination of the shared-memory parallel backend and the sequential reference one. When targeting a backend, ALP/Dense containers are automatically specialized to it. Here, we indicate this with an additional template parameter in the `Matrix` type, *e.g.*, `Matrix< ..., shmem >` for a matrix in our current example. The internal `get_view` function can be used to cast the computation in terms of the same primitive applied to local sequential blocks `Aloc` of size $m_b \times n_b$ as shown in the code sketch below:

```
1  RC eWiseMul( Matrix< ..., shmem > A, ... alpha, ...,
       ... Sr ) {
2    const auto &da = internal::getStorageScheme( A )
3                               .getDistribution();
4
5    #pragma omp parallel
6    {
7      thread = config::OMP::current_thread_ID();
8      th_coo = da.getThreadCoords( thread );
9      blk_grd = da.getLocalBlockGridDims( th_coo );
10
11     for( auto &blk_coo: blk_grd ) {
12       // Assuming shmem::seq == reference, Aloc has
       type
13       // Matrix< ... , reference >
14       auto Aloc = internal::get_view< shmem::seq >(
15           A, th_coo, blk_coo
16         );
17         // Calls func on L.21
18         eWiseMul( Aloc, alpha, beta, Sr );
19     } }
20 }
21
22 // Implementation of eWiseMul for reference containers
23 RC eWiseMul( Matrix< ..., reference > A, ... , ... Sr )
       { ... }
```

Differently from the user-level `get_view`, the internal one associated with the shared-memory backend takes coordinates from a thread and a local block grid, both defined by the distribution of the storage scheme. For example, assuming the shared-memory storage scheme associated with A selects a 2D block-cyclic distribution like the one in Fig. 3(b),

Aloc would be referencing one of the yellow blocks in the figure, with a layout specified by the `loc` storage polynomial. The internal `get_view` produces a sequential view container `Aloc` suitable for the sequential backend selected in combination with the parallel one (*e.g.*, reference in the example in Fig. 3(b)). Currently, ALP/Dense only implements view mechanisms for general matrices when selecting parallel backends, but we are working on extending it to include any supported structured container.

5 Evaluation

In this section, we analyze the new structural extension of ALP/Dense from both an expressiveness and performance point of view. In particular, we try two answer two questions:

1. What kind of workloads can be expressed in ALP/-Dense based on the new extensions introduced in the previous section (Sec. 5.1)?
2. What performance we can expect from future high-performance backend implementations (Sec. 5.2)?

5.1 Expressiveness

The current prototype of the ALP/Dense framework is publicly available [4], and collects various algorithmic implementations including:

- Householder tridiagonalisation;
- Inverse of a symmetric, positive-definite matrix;
- The Cholesky decomposition;
- LU decomposition;
- QR decomposition;
- Singular value decomposition; and
- Symmetric/Hermitian eigenvalue decomposition.

Each backend is associated to a specific include sub-folder, *e.g.*, *include/alp/reference* for the reference backend implementation. ALP Algorithms are available in the `include/alp/algorithms` folder. For instance, file `cholesky.hpp` also contains the implementation of `BChol` discussed in Sec. 3.2. We use the latter as an expressiveness case study and refer the reader to [4] for further algorithmic implementations.

The Cholesky Decomposition in ALP. In the ALP/-Dense implementation of `BChol`, operations are performed on partitions of matrix S as shwon in Fig. 1. In particular, the unblocked version of the Cholesky decomposition is applied to square diagonal blocks:

```
auto S11 = get_view( S, range1, range1 );
cholesky_uptr( S11, ring ); //unblocked version
```

The diagonal view S_{11} inherits the SPD pattern from matrix S which enables passing it to the `cholesky_uptr` function. Following, the linear system $Up(S_{11}) \cdot S_{12} := S_{12}$ is solved using (in-place) forward substitution on the upper triangular view of S_{11} and the general panel S_{12}:

```
auto S12 = get_view< General >( S, range1, range2 );
algorithms::forwardsubstitution(
  get_view< UpperTriangular >( S11 ), S12, ring );
```

Finally, the rank-k update of the square diagonal view S_{22} is computed using the `alp::mxm` and `alp::foldl` primitives:

```
// S22t is also symmetric
size_t n = ncols(S12);
Matrix< T, typename S::structure > S22t( n );
set( S22t, zero );
mxm( S22t,
     get_view< view::transpose >(S12), S12, ring );
auto S22 = get_view( S, range2, range2 );
foldl( S22, S22t, minus );
```

Notice that the `mxm` and `foldl` primitives should investigate the patterns of its parameters to determine that only half of the matrix needs to be computed. Further, the storage scheme of S_{22} is compile-time deducible from the one of S, making each internal access to S^t_{22} yield a suitable access to the memory of S.

5.2 Performance

An ALP/Dense high-performance, sequential backend is currently work in progress. However, we show projected performance obtained with our framework using a *dispatch* backend. Compiling with this backend, ALP/Dense intercepts specific combinations of primitives provided by us and maps them to BLAS calls. The dispatch backend allows for independent benchmarking of:

- sequential algorithms, where operations are dispatched to an external library, and
- parallel algorithms where operations on blocks are dispatched to an external library.

Experimental Setup. All experiments reported here are performed on Kunpeng 920-4826 machine (2 sockets, 96 cores) with 500GB DRAM. Symbols in plots show the mean value for an experiment repeated 30 times, while the shaded area around the symbols (small and not visible in some cases) indicates the corresponding standard deviation. In shared-memory parallel experiments, we also use thread pinning, in order to improve the memory locality, by setting the environment variable GOMP_CPU_AFFINITY="0-15 24-39 48-63 72-87" to reflect the NUMA domains of a Kunpeng 920-4826 machine. ALP dispatches to KML BLAS, a vendor BLAS implementation optmized for Kunpeng processors [24].

Sequential. We compare ALP/Dense implementation of the blocked Cholesky decomposition compiled with a dispatch backend with the Netlib LAPACK [6]. Both link against the sequential version of KML BLAS. We use two block sizes $BS = 128$ and $BS = 64$, and show that the performance of ALP/Dense is on par with the one of Netlib LAPACK implementation, as shown in the upper panel of Fig. 6, full green line with circles (ALP/Dense $BS = 128$) and full orange line with triangles (ALP/Dense $BS = 64$) compared to full blue line with square (LAPACK). The blocked Cholesky decomposition with fixed block sizes delivers similar and better performances for large matrices (matrix size larger than 2000), for instance, for matrix size $N = 4000$, execution

time of ALP/Dense block algorithm is 7% shorter than the LAPACK execution time.

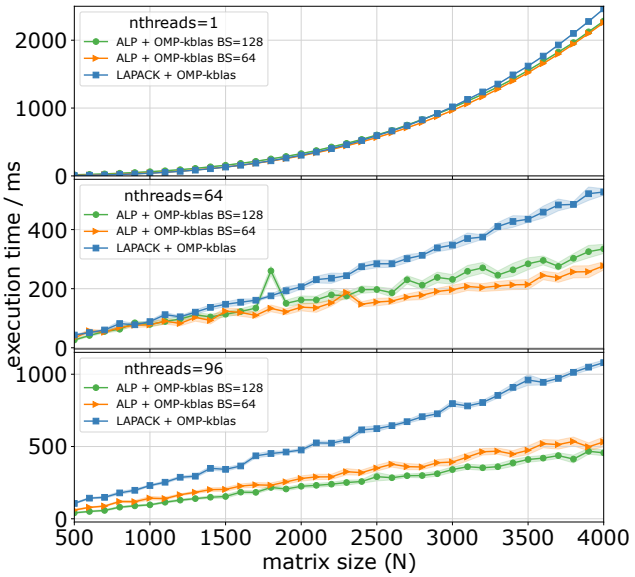

Figure 6. Execution time of ALP/Dense Cholesky block-decomposition (less is better) as a function of (square-) matrix size N. Block-decomposition using block sizes $BS = 128$ (green lines with circles), $BS = 64$ (orange lines with triangles) and Netlib LAPACK implementation (blue lines with squares). Tests are performed using 1 (upper panel), 64 (middle panel) and 96 (bottom panel) threads.

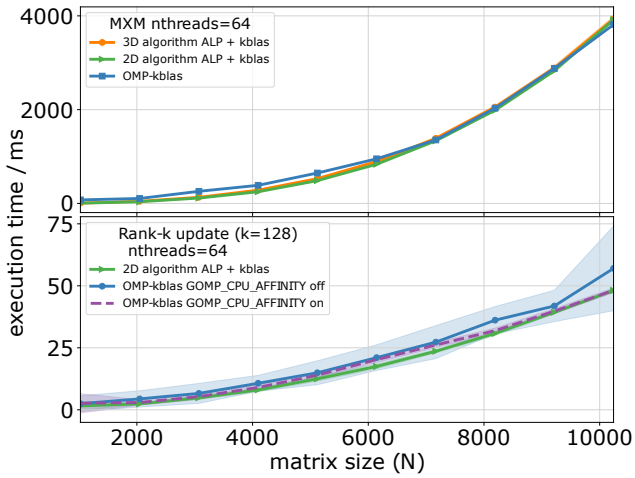

Figure 7. ALP/Dense (upper panel) multiplication of two square $N \times N$ matrices, execution time (less is better), for ALP/Dense 3D (orange line with circles), ALP/Dense 2D (green line with triangles) and Kunpeng OMP-BLAS (blue line with squares), using 64 threads. Execution time (bottom panel) of ALP/Dense (green line with triangles) and Kunpeng OMP-BLAS *rank-k update* (C += $A \times B$, where C is a square $N \times N$ matrix, A is $N \times k$, and B is $k \times N$, rectangular matrix, ($k = 128$ in all cases), using 64 threads, with thread pinning enabled (blue line with squares) and disabled (magenta dashed line).

Shared-Memory Parallel. In a first experiment, without modifying the ALP/Dense blocked algorithm, shared-memory parallel performance is inspected for 64 and 96 threads (middle and bottom panel, respectively, in Fig. 6) linking the same ALP program compiled with the dispatch backend in the sequential experiment above against the shared-memory parallel version of KML BLAS. Netlib LAPACK is also linked against the same version of the BLAS library. In particular, for matrix size $N = 4000$, ALP/Dense blocked algorithm execution time is shorter by a factor 1.6 and 2.4 when using 64 and 96 (all available) threads, respectively.

In a second experiment, we evaluate the performance of the `alp::mxm` primitive compiled with a combination of the shared memory-parallel and the sequential dispatch backends. The parallel `alp::mxm` is implemented based on the 2.5D matrix multiplication algorithm in [43]. In particular, we use threads in both a 2D (ALP/Dense 2D) and a 3D (ALP/-Dense 3D) grid configuration. The sequential `alp::mxm` is dispatched to a sequential version of `gemm` from KML BLAS. We compare against the shared-memory implementation of `gemm` from KML BLAS. In the upper panel in Fig. 7 we see the performance comparison for the operation C += $A \times B$, where A, B and C are square $N \times N$ matrices, between ALP/-Dense 3D (orange line with circles), ALP/Dense 2D (green line with triangles) and KML BLAS (blue line with squares), using 64 threads, which results in comparable performances in all three cases. The bottom panel in Fig. 7 shows performance comparisons between the ALP/Dense 2D and KML BLAS (blue line with squares) rank-k update, *i.e.*, C += $A \times B$ operation, where C is square $N \times N$ matrix, A is $N \times k$ and B is $k \times N$ size matrix, and $k = 128$ in all cases (same as the block size used in the mxm experiment). In the ALP/Dense implementation, all matrices are blocked into $BS = 128$ sized blocks, and redistributed in memory according to 2D scheme. The tests have been performed using 64 OpenMP threads, where we have imposed OpenMP thread binding. We see that ALP/Dense 2D outperforms or stays within standard deviation of the performance of KML BLAS. Additionally, ALP/Dense 2D shows much smaller performance variations than KML BLAS library with thread pinning disabled.

6 Limitations and Future Work

In this section, we point out some of the current limitations in our approach and briefly discuss how we intend to address them in future work.

High-performance ALP/Dense. Currently, most of the proposed ALP/Dense features are implemented as part of a

reference backend at the sequential level. We plan to introduce a high-performance sequential backend as well, leveraging optimization methodologies from past library and compiler experiences [8, 18, 47]. In particular, we plan to further leverage the static knowledge from the pattern of a container and its access function in the optimization process.

Exploiting Entry Patterns. Structured containers, such as symmetric, skew symmetric, and Vandermonde matrices exhibit entry patterns which can currently only be captured implicitly by including *ad-hoc* patterns with a general non-zero band in the `is_a` poset discussed in Sec. 4.1. We believe that a modern algebraic programming interface should support new algorithmic formulations by easily incorporating old and newly emerging container structures [20, 37], even if not (yet) considered by established HPC interfaces, such as BLAS and LAPACK. For this reason, we plan to extend our pattern description mechanism at the container type level to also include entry patterns along with non-zero patterns.

Unified Sparse and Dense API. The interface of ALP/-Dense builds on ALP/GraphBLAS, which supports general sparse matrices and vectors. In the future, we plan to investigate a unified ALP interface with the possibility to define the non-zero pattern of a container as either dense or sparse (*e.g.*, in Listing 1), enabling primitives to operate on mixed storage schemes. Further, extending the approach to tensor containers may define a truly flexible algebraic interface for multidimensional array programming [18, 27, 40, 42].

7 Conclusion

In this paper, we proposed a new algebraic programming interface to express primitive operations on structured containers with densely-stored non-zero patterns, such as band and triangular matrices. The container type decouples the formulation of its non-zero pattern from the one of its storage schemes, allowing users to build structured mathematical formulations, backends and to pair them with appropriate storage schemes and access functions. A compiler can use static information from the container types to verify that primitives are used on conformant containers as well that they are implemented using tight loop bounds. We demonstrated that this interface can be integrated in a modern C++ algebraic programming API and used to express and compute important structured linear algebra algorithms occurring in scientific computing and machine learning applications. If supplied with an efficient primitive implementations, algorithms based on our interface can achieve performance competitive with high-performance numerical libraries.

Acknowledgments

We thank the anonymous ALP contributors as well as the anonymous reviewers for their constructive comments.

References

[1] 2020. GraphBLAS Template Library (GBTL), v. 3.0. https://github.com/cmu-sei/gbtl.
[2] 2021. ALP/GraphBLAS. https://github.com/Algebraic-Programming/ALP.
[3] 2022. OpenBLAS – An optimized BLAS library. https://www.openblas.net/.
[4] 2023. ALP/Dense. https://github.com/Algebraic-Programming/ALP/tree/array23-artifact.
[5] Edward Anderson, Zhaojun Bai, Christian Bischof, L Susan Blackford, James Demmel, Jack Dongarra, Jeremy Du Croz, Anne Greenbaum, Sven Hammarling, Alan McKenney, et al. 1999. *LAPACK users' guide*. SIAM.
[6] Edward Anderson and Jack Dongarra. 1990. *Lapack working note 19: Evaluating block algorithm variants in LAPACK*. University of Tennessee.
[7] Henrik Barthels. 2021. *Linnea: A Compiler for Mapping Linear Algebra Problems onto High-Performance Kernel Libraries*. Dissertation. RWTH Aachen University. https://hpac.cs.umu.se/~barthels/publications/dissertation.pdf
[8] Uday Bondhugula. 2020. High Performance Code Generation in MLIR: An Early Case Study with GEMM. *CoRR* abs/2003.00532 (2020). arXiv:2003.00532
[9] Benjamin Brock, Aydın Buluç, Timothy Mattson, Scott McMillan, and José Moreira. 2021. The GraphBLAS C API Specification: Version 2.0.0. https://graphblas.org/docs/GraphBLAS_API_C_v2.0.0.pdf.
[10] Qinglei Cao, Sameh Abdulah, Rabab Alomairy, Yu Pei, Pratik Nag, George Bosilca, Jack Dongarra, Marc G. Genton, David E. Keyes, Hatem Ltaief, and Ying Sun. 2022. Reshaping Geostatistical Modeling and Prediction for Extreme-Scale Environmental Applications. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '22)*. Article 2, 12 pages.
[11] Timothy A. Davis. 2019. Algorithm 1000: SuiteSparse:GraphBLAS: Graph Algorithms in the Language of Sparse Linear Algebra. *ACM Trans. Math. Softw.* 45, 4, Article 44 (dec 2019), 25 pages. https://doi.org/10.1145/3322125
[12] J. J. Dongarra, Jeremy Du Croz, Sven Hammarling, and I. S. Duff. 1990. A Set of Level 3 Basic Linear Algebra Subprograms. *ACM Trans. Math. Softw.* 16, 1 (mar 1990), 1–17. https://doi.org/10.1145/77626.79170
[13] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Richard J. Hanson. 1988. An Extended Set of FORTRAN Basic Linear Algebra Subprograms. *ACM Trans. Math. Softw.* 14, 1 (mar 1988), 1–17. https://doi.org/10.1145/42288.42291
[14] D. Fabregat-Traver and P. Bientinesi. 2011. Automatic Generation of Loop-Invariants for Matrix Operations. In *2013 13th International Conference on Computational Science and Its Applications*. 82–92. https://doi.org/10.1109/ICCSA.2011.41
[15] Franz Franchetti, Tze-Meng Low, Thom Popovici, Richard Veras, Daniele G. Spampinato, Jeremy Johnson, Markus Püschel, James C. Hoe, and José M. F. Moura. 2018. SPIRAL: Extreme Performance Portability. *Proceedings of the IEEE, special issue on "From High Level Specification to High Performance Code"* 106, 11 (2018).
[16] Franz Franchetti, Yevgen Voronenko, and Markus Püschel. 2005. Formal Loop Merging for Signal Transforms. In *Programming Languages Design and Implementation (PLDI)*. 315–326.
[17] Gianluca Frison, Dimitris Kouzoupis, Tommaso Sartor, Andrea Zanelli, and Moritz Diehl. 2018. BLASFEO: Basic Linear Algebra Subroutines for Embedded Optimization. *ACM Trans. Math. Softw.* 44, 4, Article 42 (jul 2018), 30 pages. https://doi.org/10.1145/3210754
[18] Roman Gareev, Tobias Grosser, and Michael Kruse. 2018. High-Performance Generalized Tensor Operations: A Compiler-Oriented Approach. *ACM Trans. Archit. Code Optim.* 15, 3, Article 34 (sep 2018), 27 pages. https://doi.org/10.1145/3235029

[19] Gene H. Golub and Charles F. Van Loan. 2013. *Matrix Computations* (fourth ed.). The Johns Hopkins University Press.

[20] Albert Gu, Karan Goel, and Christopher Re. 2022. Efficiently Modeling Long Sequences with Structured State Spaces. In *International Conference on Learning Representations*. https://openreview.net/forum?id=uYLFoz1vlAC

[21] Gaël Guennebaud, Benoît Jacob, et al. 2010. Eigen. https://eigen.tuxfamily.org/.

[22] Charles R. Harris et al. 2020. Array programming with NumPy. *Nature* 585, 7825 (Sept. 2020), 357–362. https://doi.org/10.1038/s41586-020-2649-2

[23] Alexander Heinecke, Greg Henry, Maxwell Hutchinson, and Hans Pabst. 2016. LIBXSMM: Accelerating Small Matrix Multiplications by Runtime Code Generation. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '16)*. Article 84, 11 pages. https://doi.org/10.5555/3014904.3015017

[24] Ltd Huawei Technologies Co. 2023. KML Kunpeng Math Library. https://www.hikunpeng.com/developer/boostkit/library/math. Accessed: 2032-03-07.

[25] Intel. 2023. Intel oneAPI Math Kernel Library. https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl.html.

[26] Jeremy Kepner and John Gilbert. 2011. *Graph Algorithms in the Language of Linear Algebra*. Society for Industrial and Applied Mathematics. https://doi.org/10.1137/1.9780898719918

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

[28] C. Lattner and V. Adve. 2004. LLVM: a compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization (CGO)*. 75–86. https://doi.org/10.1109/CGO.2004.1281665

[29] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques A. Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *IEEE/ACM International Symposium on Code Generation and Optimization, CGO*. 2–14. https://doi.org/10.1109/CGO51591.2021.9370308

[30] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. 1979. Basic Linear Algebra Subprograms for Fortran Usage. *ACM Trans. Math. Softw.* 5, 3 (sep 1979), 308–323. https://doi.org/10.1145/355841.355847

[31] Tze Meng Low, Francisco D. Igual, Tyler M. Smith, and Enrique S. Quintana-Orti. 2016. Analytical Modeling Is Enough for High-Performance BLIS. *ACM Trans. Math. Softw.* 43, 2, Article 12 (aug 2016), 18 pages. https://doi.org/10.1145/2925987

[32] Johan Mabille, Sylvain Corlay, and Wolf Vollprecht. 2016. xtensor. https://github.com/xtensor-stack/xtensor.

[33] Devin A. Matthews. 2018. High-Performance Tensor Contraction without Transposition. *SIAM Journal on Scientific Computing* 40, 1 (2018), C1–C24. https://doi.org/10.1137/16M108968X

[34] Nvidia. 2023. cuBLAS. https://docs.nvidia.com/cuda/cublas/.

[35] Federico Pizzuti, Michel Steuwer, and Christophe Dubach. 2019. Position-Dependent Arrays and Their Application for High Performance Code Generation. In *Proceedings of the 8th ACM SIGPLAN International Workshop on Functional High-Performance and Numerical Computing (FHPNC 2019)*. 14–26. https://doi.org/10.1145/3331553.3342614

[36] Christos Psarras, Henrik Barthels, and Paolo Bientinesi. 2022. The Linear Algebra Mapping Problem. Current State of Linear Algebra Languages and Libraries. *ACM Trans. Math. Softw.* 48, 3, Article 26 (sep 2022), 30 pages. https://doi.org/10.1145/3549935

[37] Zhen Qin, Xiaodong Han, Weixuan Sun, Bowen He, Dong Li, Dongxu Li, Yuchao Dai, Lingpeng Kong, and Yiran Zhong. 2023. Toeplitz Neural Network for Sequence Modeling. In *The Eleventh International Conference on Learning Representations*. https://openreview.net/forum?id=IxmWsm4xrua

[38] Sanil Rao, Anurag Kutuluru, Paul Brouwer, Scott McMillan, and Franz Franchetti. 2020. GBTLX: A First Look. In *2020 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–7. https://doi.org/10.1109/HPEC43674.2020.9286231

[39] Conrad Sanderson and Ryan Curtin. 2016. Armadillo: a template-based C++ library for linear algebra. *Journal of Open Source Software* 1, 2 (2016), 26. https://doi.org/10.21105/joss.00026

[40] Martin D. Schatz, Tze Meng Low, Robert A. van de Geijn, and Tamara G. Kolda. 2014. Exploiting Symmetry in Tensors for High Performance: Multiplication with Symmetric Tensors. *SIAM Journal on Scientific Computing* 36, 5 (2014), C453–C479. https://doi.org/10.1137/130907215

[41] Stanislav G. Sedukhin and Marcin Paprzycki. 2012. Generalizing Matrix Multiplication for Efficient Computations on Modern Computers. In *Parallel Processing and Applied Mathematics*, Roman Wyrzykowski, Jack Dongarra, Konrad Karczewski, and Jerzy Waśniewski (Eds.). 225–234. https://doi.org/10.1007/978-3-642-31464-3_23

[42] Shruti Shivakumar, Jiajia Li, Ramakrishnan Kannan, and Srinivas Aluru. 2021. *Efficient Parallel Sparse Symmetric Tucker Decomposition for High-Order Tensors*. 193–204. https://doi.org/10.1137/1.9781611976830.18

[43] Edgar Solomonik and James Demmel. 2011. Communication-Optimal Parallel 2.5D Matrix Multiplication and LU Factorization Algorithms. In *Euro-Par 2011 Parallel Processing*, Emmanuel Jeannot, Raymond Namyst, and Jean Roman (Eds.). 90–109.

[44] Daniele G. Spampinato and Markus Püschel. 2016. A Basic Linear Algebra Compiler for Structured Matrices. In *International Symposium on Code Generation and Optimization (CGO)*. 117–127.

[45] Xing Su, Xiangke Liao, and Jingling Xue. 2017. Automatic Generation of Fast BLAS3-GEMM: A Portable Compiler Approach. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization (CGO '17)*. 122–133.

[46] Yaohung Tsai. 2020. *Mixed-Precision Numerical Linear Algebra Algorithms: Integer Arithmetic Based LU Factorization and Iterative Refinement for Hermitian Eigenvalue Problem*. Dissertation. University of Tennessee. https://trace.tennessee.edu/utk_graddiss/6094

[47] Field G. Van Zee and Robert A. van de Geijn. 2015. BLIS: A Framework for Rapidly Instantiating BLAS Functionality. *ACM Trans. Math. Softw.* 41, 3, Article 14 (jun 2015), 33 pages. https://doi.org/10.1145/2764454

[48] Sven Verdoolaege. 2010. isl: An Integer Set Library for the Polyhedral Model. In *Mathematical Software – ICMS 2010*. Springer Berlin Heidelberg, 299–302.

[49] Pauli Virtanen et al. 2020. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods* 17 (2020), 261–272. https://doi.org/10.1038/s41592-019-0686-2

[50] Weiling Yang, Jianbin Fang, Dezun Dong, Xing Su, and Zheng Wang. 2021. LIBSHALOM: Optimizing Small and Irregular-Shaped Matrix Multiplications on ARMv8 Multi-Cores. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '21)*. Article 72, 14 pages. https://doi.org/10.1145/3458817.3476217

[51] A. N. Yzelman, D. Di Nardo, J. M. Nash, and W. J. Suijlen. 2020. A C++ GraphBLAS: specification, implementation, parallelisation, and evaluation. (2020). Preprint.

Received 2023-03-31; accepted 2023-04-21