Mosaic: An Interoperable Compiler for Tensor Algebra

MANYA BANSAL, Stanford University, USA
OLIVIA HSU, Stanford University, USA
KUNLE OLUKOTUN, Stanford University, USA
FREDRIK KJOLSTAD, Stanford University, USA

We introduce Mosaic, a sparse tensor algebra compiler that can bind tensor expressions to external functions of other tensor algebra libraries and compilers. Users can extend Mosaic by adding new functions and bind a sub-expression to a function using a scheduling API. Mosaic substitutes the bound sub-expressions with calls to the external functions and automatically generates the remaining code using a default code generator. As the generated code is fused by default, users can productively leverage both fusion and calls to specialized functions within the same compiler. We demonstrate the benefits of our dual approach by showing that calling hand-written CPU and specialized hardware functions can provide speedups of up to 206× against fused code in some cases, while generating fused code can provide speedups of up to 3.57× against code that calls external functions in other cases. Mosaic also offers a search system that can automatically map an expression to a set of registered external functions. Both the explicit binding and automatic search are verified by Mosaic. Additionally, the interface for adding new external functions is simple and general. Currently, 38 external functions have been added to Mosaic, with each addition averaging 20 lines of code.

CCS Concepts: • **Software and its engineering** → **Source code generation**; *Domain specific languages*; • **Mathematics of computing** → *Mathematical software performance*.

Additional Key Words and Phrases: sparse tensor algebra, compilation, external functions, automated search

ACM Reference Format:
Manya Bansal, Olivia Hsu, Kunle Olukotun, and Fredrik Kjolstad. 2023. Mosaic: An Interoperable Compiler for Tensor Algebra. *Proc. ACM Program. Lang.* 7, PLDI, Article 122 (June 2023), 26 pages. https://doi.org/10.1145/3591236

1 INTRODUCTION

Sparse tensor algebra is an important computational language that describes multilinear expressions over dense and sparse tensors. It is used in many domains, including scientific computing, engineering, and machine learning. Performance is crucial in these domains, resulting in a proliferation of libraries for CPUs [Gough 2009; Huang et al. 2017; Intel 2009; Lawson et al. 1979; Whaley and Petitet 2005], GPUs [Dalton et al. 2014; M Naumov 2010], vector processors [Choquette et al. 2021; Intel 2011; Stephens et al. 2018], and domain-specific hardware [Chen et al. 2018b; Dadu et al. 2019; He et al. 2020; Hegde et al. 2019; Jouppi et al. 2017; Pal et al. 2018; Qin et al. 2020; Rucker et al. 2021; Srivastava et al. 2020a,b; Zhang et al. 2021a]. These libraries have been optimized at great expense and effort. As a result, most tensor algebra expressions are written as a sequence of calls to libraries that compute different sub-expressions.

Authors' addresses: Manya Bansal, manya227@stanford.edu, Stanford University, USA; Olivia Hsu, owhsu@stanford.edu, Stanford University, USA; Kunle Olukotun, kunle@stanford.edu, Stanford University, USA; Fredrik Kjolstad, kjolstad@stanford.edu, Stanford University, USA.

2475-1421/2023/6-ART122
https://doi.org/10.1145/3591236

Fusing operations can lead to better performance, thus breaking library boundaries [Ahrens et al. 2022; Hsu et al. 2023; Kjolstad et al. 2017; Yadav et al. 2022]. Fused sparse tensor algebra operations may even have better asymptotic complexity. Moreover, generating bespoke code for an expression and specializing the loop order can avoid expensive tensor transposes and reshapes. Due to the cost of developing specialized fused operations, researchers have proposed compilers that can automatically generate fused code for both dense and sparse tensor algebra [Bik et al. 2022; Kjolstad et al. 2017; Mutlu et al. 2020; Tian et al. 2021; Ye et al. 2023; Zhao et al. 2022]. Although these compilers generalize to any tensor algebra expression, they cannot match the performance of libraries for expressions that are sufficiently important to have been hand optimized. For example, dense matrix multiplication when implemented on specialized hardware can result in a speedup of two orders of magnitude. Therefore, the best performance for an expression may require fused code (e.g., as in sparse sampled dense-dense matrix multiplication), calls to libraries (e.g., as in dense matrix multiplication), or a mix of fused code and calls to libraries.

No current sparse tensor algebra compiler can mix generated specialized and fused code with calls to libraries or domain-specific hardware, leaving application developers to write low-level code by hand. The application developer must write code to traverse and compute on sparse data structures, fuse sparse expressions, and tile, transpose, and reshape sparse tensors to fit library APIs. Writing such code is a laborious and error-prone process. Moreover, since the performance benefits of loop ordering, tiling strategy, and external functions depend on both the data and the machine, it is necessary to explore many different optimization strategies in a large design space. Without compiler support to automate the low-level code generation and to assist with this design-space exploration, application developers leave performance on the table.

Even with a way to automatically generate a mix of fused code and library calls, writing an optimal program is still challenging. Due to the sheer number of libraries available, there is a combinatorial explosion in the number of choices available for code generation. Enabling programmers to specify schedules offers a rich space of optimizations and provides the possibility of incorporating domain expertise [Ragan-Kelley et al. 2012]. However, programmers must completely specify all transformations, including picking constants to tile with and loops to reorder or fuse. Some users may not have the time or expertise to write such precise schedules. On the other hand, a completely automatic scheduler that tunes every parameter can be too slow for quick design space exploration.

Prior work on sparse tensor algebra systems has only solved parts of the problem of mixing specialized/fused sparse tensor algebra code with calls to libraries and hardware. Several compilers have been developed that can generate specialized and fused imperative code for sparse tensor algebra expressions, including TACO [Kjolstad et al. 2017], the MLIR SparseTensor Dialect [Bik et al. 2022], COMET [Mutlu et al. 2020], SparseTIR [Ye et al. 2023], and the Sparse Polyhedral Framework [Strout et al. 2018; Venkat et al. 2015]. Other systems reduce sparse linear or tensor algebra expressions to a fixed set of library calls, such as MATLAB [MATLAB 2010], Julia [Bezanson et al. 2017], TTB [Bader and Kolda 2006, 2007], and CTF [Solomonik and Hoefler 2015; Solomonik et al. 2014]. Moreover, the AMOS compiler [Zheng et al. 2022] generates code that mixes bespoke code with calls to domain-specific hardware for dense tensor algebra. And, the Exo compiler [Ikarashi et al. 2022] lets users manually compose different instructions to implement algorithms that can be expressed with affine loops, which includes dense tensor algebra. However, none can currently provide both fusion and function calls from external libraries for sparse tensor algebra compilation.

We propose Mosaic, a modular compiler for sparse tensor algebra that users can extend with library functions and specialized hardware to externally compute whole expressions or sub-expressions. Building on prior work on the TACO compiler [Kjolstad et al. 2017], Mosaic can also generate specialized and fused imperative code for those expressions or sub-expressions where no suitable external function or hardware exists. With Mosaic, users can write partial schedules

Table 1. The tensor algebra systems (and their features) used in Section 8 to evaluate Mosaic. Mosaic composes function calls from different libraries with generated code to compute any sparse tensor algebra expressions.

Tensor Algebra System	System Type (Language)	Features				
		Data Representation			Tensor	Platform
		Arbitrary Order	Dense	Sparse	Properties	
`Intel AVX` [Intel 2011]	Intrinsic (C)	✘	✔	✘	✘	CPU
`BLAS` [Lawson et al. 1979]	Library (C)	✘	✔	✘	✔	CPU
`Intel MKL` [Intel 2009]	Library (C)	✘	✔	✔	✔	CPU
`GSL` [Gough 2009]	Library (C)	✘	✔	✘	✔	CPU
`TBLIS` [Matthews 2016]	Library (C)	✔	✔	✘	✘	CPU
`TACO` [Kjolstad et al. 2017]	Compiler (C/C++)	✔	✔	✔	✘	CPU
`cuSPARSE` [M Naumov 2010]	Library (C)	✘	✔	✔	✘	GPU
`Stardust` [Hsu et al. 2022]	Compiler (Scala/Spatial)	✔	✔	✔	✘	Capstan

that map a sub-expression to an external function. Mosaic checks whether the mapping is valid against a user-provided specification, tiles and reshapes the expression to fit within the constraints of the function, and then completes the schedule. However, it is up to the user to ensure that the external function actually computes what the user-provided specification claims. To further increase programmer productivity, we also provide a completely automated search mechanism. This mechanism provides a list of valid schedules, but does not select the most performant schedule.

Mosaic is a productive and interactive system that helps developers identify ways to map an expression to external functions, produces code to transform data structures to match external function APIs, and generates any parts of the expression that cannot be mapped to external functions. Our contributions are:

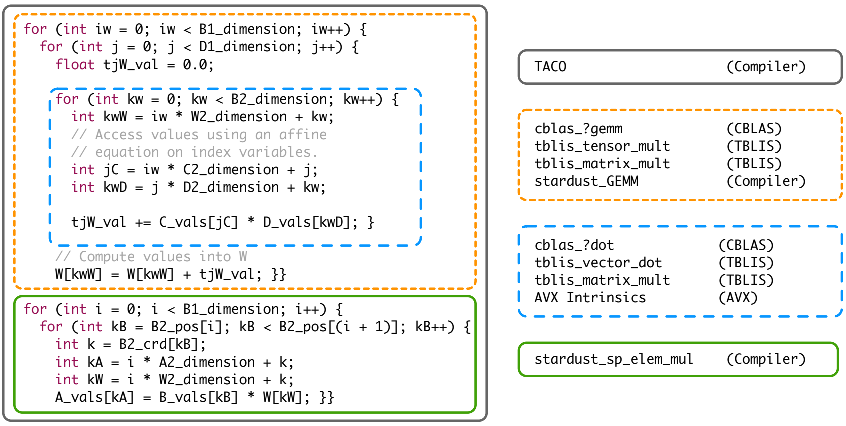
- An *external function interface* that defines the algorithm to generate code targeting a given external function and specifies the tensor algebra expressions it can compute;
- *Composable scheduling commands* that map sub-expressions to external functions;
- An *automatic verification and mapping algorithm* that rewrites expressions to identify valid mappings and fills in partial schedules; and
- A *code generation algorithm* that generates external function calls wherever a mapping has been made and generates code natively for sub-computations that have not been mapped.

To evaluate our contributions, we show that Mosaic can outperform a homogeneous compiler, sometimes even adding two orders of magnitude speedup. We also demonstrate the generality of our approach by adding 38 external functions from eight existing tensor algebra systems (see Table 1) and compiling tensor algebra expressions that cannot be fully computed by just composing calls to different functions. Finally, we show that Mosaic's search system finds many bindings that require reshaping operators within an expression.

2 MOTIVATING EXAMPLE

Consider a user who wants to compute Alternating Least Squares [Koren et al. 2009] and needs a fast implementation of sampled dense-dense matrix multiplication (SDDMM). SDDMM is expressed in tensor index notation as $A_{ij} = \sum_k B_{ij} \cdot C_{ik} \cdot D_{kj}$, i.e., the dense multiplication of tensors C and D is sampled using the sparse matrix B. Figure 1 shows an implementation of SDDMM and a number of possible library functions that can be used to replace sub-computations. Using these functions, there are 19 possible ways to rewrite the existing code.

In order to rewrite the code to utilize external functions, users must refactor the surrounding program to interface correctly with each function. Computations that are mapped to a function need to be isolated from the rest of the program and the inputs and outputs to different computations



```
for (int iw = 0; iw < B1_dimension; iw++) {
  for (int j = 0; j < D1_dimension; j++) {
    float tjW_val = 0.0;

    for (int kw = 0; kw < B2_dimension; kw++) {
      int kwW = iw * W2_dimension + kw;
      // Access values using an affine
      // equation on index variables.
      int jC = iw * C2_dimension + j;
      int kwD = j * D2_dimension + kw;

      tjW_val += C_vals[jC] * D_vals[kwD]; }
    // Compute values into W
    W[kwW] = W[kwW] + tjW_val; }}
for (int i = 0; i < B1_dimension; i++) {
  for (int kB = B2_pos[i]; kB < B2_pos[(i + 1)]; kB++) {
    int k = B2_crd[kB];
    int kA = i * A2_dimension + k;
    int kW = i * W2_dimension + k;
    A_vals[kA] = B_vals[kB] * W[kW]; }}
```

```
TACO                    (Compiler)

cblas_?gemm             (CBLAS)
tblis_tensor_mult       (TBLIS)
tblis_matrix_mult       (TBLIS)
stardust_GEMM           (Compiler)

cblas_?dot              (CBLAS)
tblis_vector_dot        (TBLIS)
tblis_matrix_mult       (TBLIS)
AVX Intrinsics          (AVX)

stardust_sp_elem_mul    (Compiler)
```

Fig. 1. Generated code for sampled dense-dense matrix multiplication (SDDMM). Color-coded boxes indicate external functions that could be used for corresponding sub-computations.

need to be rewired. Users must also observe the functions' calling conventions and initialize system-specific objects. If inputs need to be tiled to fit in specialized memories, more finicky changes (indexing correctly into each operand) are required. Writing optimized code for a fixed set of functions is already challenging, doing the same for 19 possible function placements is unfeasible.

Users can describe possible function placements for SDDMM by choosing one of the three options in Mosaic:

A full schedule: Users, such as performance engineers, may know exactly which functions to utilize and how to tile and reshape tensors in the sub-expression to meet the constraints of the function. Such users can specify these transformations precisely and then bind a sub-expression to an external function using the `bind` scheduling command (Figure 2). In this case, Mosaic ensures that every transformation and function mapping is correct.

A partial schedule: If a user wants to try a particular function for a fixed sub-expression, but does not know whether any code transformations are required to do so, they can use the `map` scheduling command (Figure 3). Mosaic will automatically discover a valid binding (if possible) by tiling and reshaping the tensors of that fixed sub-expression to match the constraints of the function.

An automatically generated schedule: Finally, if the user wants to explore the design space, Mosaic can automatically search the space of possible mappings (lines 2-3, Figure 4) and return a list of valid schedules. Then, the user can select one schedule out of all possible schedules (line 6, Figure 4).

```
// SDDMM in einsum notation.
stmt=A(i,j)=B(i,j)*C(i,k)*D(k,j)
// Precompute C*D in W and use
// iw,jw,kw as index vars in the code.
stmt.precompute(C*D,W,{i,j,k},
                {iw,jw,kw})
// Split loop kw by 4 into ki and ko.
     .split(kw,ko,ki,4)
// Push ki to be the inner most loop.
     .reorder(iw, jw, ko, ki)
// Consider iw, jw to be constant.
     .fix(iw,jw)
// Bind the reduction of kw to AVXAdd().
     .bind(AVXAdd(), C*D)
```

Fig. 2. Full schedule for targeting SDDMM to AVX vector add.

```
stmt = stmt.map(AVXAdd(), C*D)
```

Fig. 3. Partial Schedule for targeting SDDMM to AVX vector add.

```
1 // Register AVX to Mosaic.
2 register(AVXAdd());
3 vec<stmts> schedules =
4           stmt.getAllSchedules();
5 // Pick a schedule to apply.
6 A.compile(schedules[i])
```

Fig. 4. Automatically search and find all schedules that use AVX vector add.

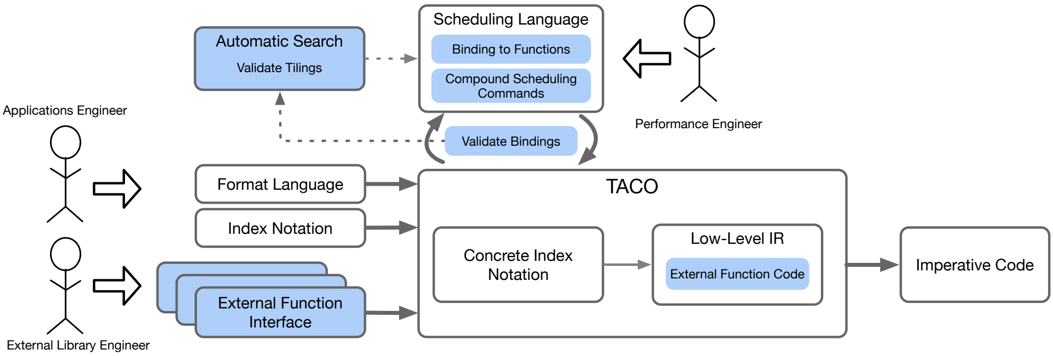


Fig. 5. System overview of our Mosaic compiler with blue components signifying new contributions. Dotted arrows signify the automatic mapping loop, which an end-user may optionally enable. Mosaic enables independent development among experts: application engineers may write programs, performance engineers may pick function substitutions, and system developers may add external libraries.

3 OVERVIEW

Mosaic compiles tensor algebra expressions to a mix of natively generated code and external function calls. That is, while lowering a tensor algebra expression, it glues together library functions, filling in any blanks where no function is available with generated code. In this way, it gives users the ability to write performant code using highly optimized functions, while preserving generality. Figure 5 shows how different pieces of Mosaic interact, and we describe each piece below.

Mosaic is implemented as an extension to the TACO compiler [Kjolstad et al. 2017], an open-source domain-specific compiler for sparse tensor algebra embedded in C++. As in TACO, users describe computations using tensor index notation (Einsum notation), a format language [Chou et al. 2018], and a scheduling language [Senanayake et al. 2020]. These languages combine to form concrete index notation (CIN), an abstract loop based IR [Kjølstad et al. 2019]. We add another domain-specific input language called the *external function interface* that adds new library functions to Mosaic. We extend the scheduling language with commands that specify function placement.

External functions are added as extensions through an external function interface (described in Section 4). In this work, an external function is any function that computes one or more tensor algebra expressions. Some libraries with functions that fall into this category are listed in Table 1. The external function interface provides two key pieces of information in order to correctly substitute a sub-computation with a call to an external function: the calling convention associated with the function and a description of the function's capability. A function's capability is characterized by the set of tensor algebra expressions it computes and the constraints it imposes over those expressions. Mosaic characterizes a function's capability using a *capability description*. The external function interface of a function only needs to be written once for a single function, therefore, other users can include pre-written descriptions like a library.

Mosaic introduces new *scheduling commands* (Section 5) that integrate with TACO's scheduling framework, which consists of commands that form a composable set of rewrite rules that can transform concrete index notation expressions by reordering, splitting and fusing loops. With the additional commands provided by Mosaic, users can also bind computation to external functions. Or, users can map a computation to a function, and Mosaic will automatically discover any valid tiling or reshaping transformation that permits this mapping. Thus, Mosaic can complete under-specified schedules to expose valid mappings. Before binding to a function, Mosaic also validates mappings by translating constraints defined on tensor order and dimension to an SMT query.

Finally, we also provide a fully *automatic mapping* (Section 6) solution to end-users. Validity and speed are the two core pillars of the automatic mapping process. To ensure validity, we check whether the schedule matches the function's capabilities using an SMT solver. The capability language is critical for not only validity, but also the speed of the mapping process. By exploiting information embedded in the compute capability language to guide the search, Mosaic can apply guided rewrites to expose valid mappings. Using this solution, users can register the plugin and transparently schedule existing TACO programs to a combination of different functions in two simple lines of code. The automatic-mapping solution only returns a list of valid mappings; users can add an autoscheduler that ranks or prioritizes the returned mappings. We leave this autoscheduler as future work.

Extensibility and modularity are at the core of Mosaic. Users can not only extend Mosaic by adding external functions, but they can also change Mosaic's default code generation algorithm by adding a code generator. Users can add an autoscheduler that selects a mapping made by Mosaic's automatic mapping process. Each component of Mosaic—the user program, schedules, and plugins—is independent of the other. A user can write a program, system experts can write external function plugins, and performance engineers can write schedules.

4 EXTERNAL FUNCTION INTERFACE

Users of Mosaic can add a new external function by supplying a description of its calling convention and compute capabilities. The calling convention tells Mosaic how to call the function. The compute capabilities description tells Mosaic what set of tensor algebra expressions the external function can compute, and is used to verify the correctness of user-requested bindings and to guide an automated search.

To add new external functions to Mosaic, users write an *external function interface*. A sample interface for vector addition using AVX intrinsics is given in Figure 6. Each external function interface consists of seven pieces of information:

A calling description that provides the name, return type, and arguments for the external function (line 6, Figure 6).

Setup code that is called before calling the function. The setup code may initialize specialized memories, pack data into function-specific data structures and user-defined formats, allocate additional memory, and configure meta-data values (line 9, Figure 6).

Teardown code that is called after the function. The teardown code can be used to check for error codes, unpack data from function-specific data structures and user-defined formats, and free allocated memory (line 12, Figure 6).

The function capabilities describe the expressions the function computes (line 14, Figure 6).

Include paths are paths to files where setup and teardown code is declared (line 17, Figure 6).

Library paths denote the path to the shared library where the external function and functions called during setup and teardown are defined (line 19, Figure 6).

Build flags denote any Makefile flags that should be used to compile the code (line 21, Figure 6).

An external function interface need only be written once for each external function. Because an external function interface is simply a C++ class, users may also embed code generators as plugins to Mosaic. Interfaces for code generators will not have a fixed definition or declaration at mapping time, but will emit a concrete definition or declaration when a successful mapping is identified.

4.1 Calling Convention

A calling description (Line 6, Figure 6) provides information that is used to generate code to call an external function. It consists of a *name*, *return type*, and a *list of arguments*. Arguments can be other

```
class VecSumAVX : public FunctionAbstraction {
 public:
  GslTensorPlus() : x(Tensor(Float32,  1, {Dense})) ... {}
                               // datatype, #dimensions, format for dimension.
  // CallingDescriptions written in Mosaic's IR.
  CallingDescription getCallingDescription(){return x_avx = _mm256_add_ps(y_avx, z_avx);}
  // Initialize y_avx, z_avx.
  vector<CallingDescription> getSetup(){return {y_avx = _mm256_load_ps(y),
                                                z_avx = _mm256_load_ps(z)}}
  // Store result into x from x_avx.
  vector<CallingDescription> getTeardown(){return {_mm256_storeu_ps(y, &x_avx)};}
  FuncCapabilityStmt getFuncCapabilities(){stmt =  x(i) = y(i) + z(i);
                                            return stmt.where(i==8);} // Constraint length of i.
  // Name and path of include file.
  vector<string, string> getIncludePaths(){return {{"immintrin.h", path}}}
  // Name and path of shared library object. For AVX, we have none.
  vector<string> getLibraryPaths(){return {};}
  // Add "-mavx2" to the Makefile flags.
  vector<string> getCFlags(){return {"-mavx2"}}
 private:
    Tensor x, y, z;
    FuncObject x_avx, y_avx;}
```

Fig. 6. Sample external function abstraction for _mm256_add_ps in Intel AVX intrinsics written in C++. Users inherit from Mosaic's abstract class `FuncAbstraction`. Each virtual function corresponds to a field of the external function abstraction.

calling descriptions, library objects, or tensor metadata. Using the recursive definition of arguments that includes other calling descriptions, users can perform any pre-processing or format change on function arguments. Additionally, libraries like TBLIS and GSL pack data into special structs that must be initialized before the function call. Through the addition of library objects as argument types, Mosaic can declare such objects during the code generation phase. These objects are then used by later parts of the code. Finally, tensor metadata includes commonly used parameters like an array of dimensions, the dimension of a particular rank, and the array of tensor values.

4.2 Compute Capabilities

When replacing sub-expressions with external functions, Mosaic ensures only valid substitutions are performed. That is, Mosaic determines whether a sub-computation lies in the space of a *function's capabilities.* A function's capabilities are the set of expressions a function can calculate, as well as any constraints that the inputs must satisfy.

Specifying the semantics of external functions for sparse tensor computations presents a unique challenge. While some functions compute a single expression, others can calculate an infinite number of expressions. And, some functions are only suitable for tensors with a specified mathematical property or a restricted sparsity pattern. In addition, constraints of exotic hardware must also be expressible. To capture such a wide variety of interface semantics, three components work together:

A capability language: An tensor index notation language with added support to describe tensors with unspecified rank and to impose constraints on expressions.

A checker function: A user-defined function that takes as input an expression and returns true or false indicating a successful or failed match.

Tensor properties: A specification of the accepted tensor operands, including formats, mathematical properties, and sparsity patterns.

Capability Language. The capability language, shown in Figure 7, describes the expressions that a function can compute. The capability language expands upon tensor index notation (or einsum notation) and can describe classes of expressions by defining tensors with unspecified rank. For such expressions, constraints over dimension size (the size of each rank) and order (the number of

$$\begin{array}{llllllll}
\textit{Index Variable} & i & \textit{Concrete Index Variable List} & i* & \textit{Constant Integer} & \text{int} & \textit{Index into an Index-Variable List} & \text{index}
\end{array}$$

$$\begin{array}{rlll}
\textit{Capability Description} & CD & ::= & INS \text{ when } ILS \mid INS \\
\textit{Index Notation Statement} & INS & ::= & \text{forall}_i\ S \mid a = E \mid a \mathrel{+}= E \\
\textit{Index Notation Expression} & E & ::= & a \mid \text{int} \mid E + E \mid E * E \mid \ldots \\
\textit{Index List Statement} & ILS & ::= & jc \mid \forall(i,\ \mathcal{I},\ ILS) \mid \exists(i,\ \mathcal{I},\ ILS) \\
\textit{Joint Condition} & jc & ::= & c \land c \mid c \lor c \\
\textit{Tensor Accesses} & a & ::= & \mathcal{T}(\mathcal{I}) \mid \mathcal{T}(i*) \\
\textit{Dynamic Index Variable List} & \mathcal{I} & ::= & \mathcal{I}\ \mathcal{I} \mid i\ \mathcal{I} \mid \mathcal{I}\ i \mid \text{range}(\ldots) \mid \text{range}(\text{int}\ldots) \mid \text{range}(\ldots\text{int}) \mid \text{range}(\text{int}\ldots\text{int}) \\
\textit{Condition} & c & ::= & e \mathrel{!}= e \mid e == e \mid e <= e \mid e >= e \mid e < e \mid e > e \\
\textit{Binary Op} & b & ::= & e + e \mid e - e \mid e * e \mid e/e \mid e\%e \\
\textit{Index List Expression} & e & ::= & b \mid p \mid \text{int} \\
\textit{Property} & p & ::= & \text{order}(\mathcal{I}) \mid \text{dimension}(\mathcal{I}(\text{index})) \mid \text{product}(\mathcal{I})
\end{array}$$

Fig. 7. Context-free grammar of our compute capability language, which augments index notation to include dynamic lists of index variables with constraints.

ranks) are described using set-builder notation. Therefore, the capability language can describe both functions that compute a fixed expression, like the `cblas_saxpy` vector addition function from the `CBLAS` library, and functions that compute many expressions, like the `tblis_tensor_mult` any-order tensor contraction function from the `TBLIS` library.

The compute capability language defines the capabilities of functions that can compute a class of expressions by letting users index into tensors with an unspecified number of ranks. In order to achieve this, the capability language allows tensors to be indexed by a dynamically sized index variable list (see Dynamic Index Variable List in Figure 7) in addition to a fixed set of indices.

A dynamic index variable list may need to satisfy certain constraints in order to lie in the capability of a function. For example, Stardust—a tensor algebra compiler targeting specialized hardware—requires that the total number of values in a tensor does not exceed 65,536 (due to memory constraints) even though it does not restrict tensor rank. In this case, no matter what the tensor rank, the product of each index variable's dimension in the index list must not exceed 65,536. To specify such constraints over a dynamically sized index variable list, the capability language provides three language constructs:

(1) `condition`: A `condition` node can describe constraints over index variables and properties of a dynamically sized index list like the order of an index list, the product of the dimension of index variables in an index list, etc.
(2) $\forall$(`iterator`, `index list`, `condition`) : The $\forall$ node describes a condition that must be true for all elements of a dynamically sized index list.
(3) $\exists$(`iterator`, `index list`, `condition`) : The $\exists$ node describes a condition that must be true for at least one element of a dynamically sized index list.

Therefore, the capability description $\texttt{product}(\mathcal{I}) < 65,536$, where $\mathcal{I}$ is the dynamically sized index list used to index into tensors in Stardust's description, can be used to specify the memory restriction of Stardust mappings.

In addition to iterating through index lists, the capability language also provides a concatenation operation for index lists that can be used to concatenate index lists with index variables or other index lists. By doing so, it is possible to specify different constraints on the indices being concatenated. For example, consider a tensor $\mathcal{T}$ with unfixed rank using IJ where $I = range(...)$ (i.e. a dynamically sized index list) and $J = [m, n]$ where m, n are concrete index variables. Due to the concatenation of I with J, the minimum order of $\mathcal{T}$ is set to 2. When matching $\mathcal{T}$ to a concrete tensor, A, that is used in the expression we want to schedule, the last two index variables that are being used to index into A are captured by m and n. Any constraints on m and n are then applied to the captured index variables.

Checker Function. As there are constraints that the capability language cannot express, Mosaic supplements the capability language with a checker function. The checker function is a C++ function that takes an expression written in tensor index notation and returns a boolean indicating whether the expression lies within the function's capability. Checker functions can be used to describe subtle constraints of highly specialized emerging hardware. For example, the checker function can be used to reject a function mapping for specialized hardware if the hardware is already in use.

Although the checker function is strictly more expressive than the compute capability language, it is opaque to other parts of Mosaic. If an external function writer only provides a checker function, we would need to call the checker function to check for a match on every transformation for every sub-expression for every function, limiting automatic mapping solutions from exploiting the structure of the function capabilities to guide the search. By combining both, we get full expressibility through the checker function, and fast search space exploration through the language. Working together, these two approaches enable us to do a coarse-grained match using the language and then an optional fine-grained match using the checker function.

Tensor Properties. Most external functions can only compute on input tensors of a specified format. In Mosaic, similar to TACO, users specify the formats of tensors used in the capability description using the Format Language [Chou et al. 2018] introduced in prior work.

In order to ensure correctness, Mosaic also gives users the ability to annotate a tensor with a *property*. Some functions may further restrict input tensors that have special mathematical properties or sparsity patterns. For example, `MKL` has a matrix multiply function optimized for Hermitian matrices. In this case, Mosaic must ensure that only a Hermitian matrix is mapped to this function. Users can tag tensors with properties to indicate such special characteristics. Properties have no semantic meaning associated with them. This tagging system can be used to indicate different types of sparsity, and mathematical properties.

To tag tensors with properties, we provide a `tag` argument to the tensor format. We show an example of a dense symmetric matrix: `Tensor<float>T("T",{dense, dense},Property::symmetric)`. Additionally, we give users a way to describe the interaction between different tags and operations to avoid tedious user tagging. Mosaic allows users to describe these rules using an index expression language variant that index expression with tag objects instead of tensor objects. Mosaic then propagates tags based on the user-provided tag rules.

5 BINDING EXPRESSIONS TO EXTERNAL FUNCTIONS

Many modern domain-specific programming systems use scheduling languages to guide program optimization [Chen et al. 2007, 2018a; Ragan-Kelley et al. 2012; Senanayake et al. 2020; Yi 2012]. A clean scheduling language separates the rewrite system and code generation from the decisions about what rewrites to apply. This design greatly increases the productivity of performance engineers and simplifies work on automatic optimizations. The scheduling commands typically include classical compiler loop optimizations (interchange, strip-mining, flattening, tiling, vectorization, and parallelization), but also commands to move computations into or out of loop nests such as Halide's compute-at command or TACO's precompute command.

Mosaic extends TACO's scheduling language [Kjølstad et al. 2019; Senanayake et al. 2020; Yadav et al. 2022] with new commands that can be used to bind a tensor algebra expression to an external function. These commands can be used to substitute either full expressions or sub-expressions with an external function. Mosaic will generate any code that is needed to transform the sparse data structures of tensors to match the function's API (see Section 7). In addition, Mosaic will verify, using an SMT solver, that the sub-expression conforms to the compute capabilities of the function.

5.1 The Bind Command

The bind command replaces a sub-expression in a tensor algebra expression with an external function after Mosaic has verified that the replacement is valid. As a result, Mosaic's code generator emits code that calls that function, instead of generating imperative code to compute the sub-expression. Mosaic also generates supporting code to transform the arguments to the function into the data structures expected by the function and code to transform the result back to the data structures specified by the result tensor.

The bind command `S' = S.bind(s, f)` is applied to a statement `S`, given in the concrete index notation, which it then rewrites. It takes as its inputs the sub-statement `s` to replace and the function `f` to replace it with. The bind command either returns a rewritten statement `S'` or an error message if the index expression implemented by `s` is not in the capability set of `f`.

Binding Validation. To ensure the correctness of binding `s` to `f`, Mosaic validates the binding against two things that are associated with `f`: the compute capability description, written in the compute capability language, and the checker function, written in C++. If `f` has a checker function and it returns false when `s` is supplied as an input, then Mosaic cannot bind `s` to `f`. If the checker function returns true, Mosaic validates the binding against the capability description.

There are four steps to check whether `s` lies in the space of expressions defined by the capability description of `f`. First, Mosaic ensure that the datatypes of the operands of `f` matches that of the tensors in `s`. Second, Mosaic checks whether the operators used in `s` are the same as the operators used in the capability description. If these two checks pass, then Mosaic can associate each tensor in `s` with a tensor in the capability description. Third, for each associated tensor, Mosaic matches the tensor's index variables used in the user-provided expression (concrete index variables) to the tensor's index variable used in the capability language (abstract index variables). Fourth, Mosaic checks whether all constraints described over the abstract index variables are satisfied by the assigned concrete index variables. To do so, Mosaic explicitly enumerates the corresponding constraints over each assigned concrete index variable and generates code targeting the Z3 theorem prover [de Moura and Bjørner 2008]. Figure 8 shows how Mosaic generates the theorem prover inputs for this step of the validation. After step two, there may be several options for matching concrete index variables to abstract index variables due to the concatenation operator in the compute capability language. If the selected matching passes the rest of the validation steps, we bind `s` to `f`. Otherwise, we test other possible matchings.

```
// Generate Z3 query from Capability Language's AST
void GenerateZ3Query(ASTNode node){
  switch(node->type){
    case ∀(i, I, ILS):
      for(i ∈ Concrete(I)):
       conditions[i] = GenerateZ3Query(
            ILS(i))
      emit z3.And(conditions.join(","))
      break
   case ∃(i, I, ILS):
      for(i ∈ Concrete(I)):
       conditions[i] = GenerateZ3Query(
            ILS(i))
      emit z3.Or(conditions.join(","))
      break
    case (Joint_Condition):
      //node.op is ∧ or ∨.
      emit node.op(GenerateZ3Query(node.RHS)
                   GenerateZ3Query(node.LHS))
      break
    case (Condition):
      //node.op is ==, != , ≤, ≥, >, <.
      emit GenerateZ3Query(node.RHS) node.op
          GenerateZ3Query(node.LHS)
      break
    case (Binary_Op):
      //node.op is +, −, %, ∗, /.
      emit GenerateZ3Query(node.op_1) node.op
          GenerateZ3Query(node.op_2)
      break
    case (....)
```

Fig. 8. Z3 code generation for a subset of the compute capability AST nodes (defined in Figure 7). Mosaic builds an SMT query that validates a given binding.

5.2 The Map Command

To increase productivity, Mosaic provides a compound command, `map`, that provides partial scheduling automation. When provided with a sub-expression, `map` is tasked with binding the whole sub-expression to `f`. Unlike `bind`, where users must explicitly specify how the computation gets associated with the interface and are tasked with tiling, reshaping, and precomputing the appropriate expressions, `map` tiles and reshapes the sub-expression to fit within the constraints of a specified function. Users do not need to search, read, or understand the minute details of the documentation for the external function they want to use. Essentially, using the map command, the user isolates a sub-expression that exactly matches f and leaves the rest of the transformations that are necessary for correctness, such as tiling and reshaping, to Mosaic.

The `map` command is applied to a concrete index notation statement and takes as input a sub-statement `s` and an interface `f` to produce `S'`, the result of applying `S.map(s, f)`. It uses several other scheduling commands to rewrite the CIN statement and target the sub-statement to an external interface. Some of these commands include existing commands (`split`, `fuse`, `precompute`, `reorder` and `parallelize`) from the TACO scheduling API [Senanayake et al. 2020]. However, we also add the `promote` and `fix` commands to Mosaic. These commands modify the sub-expressions, either by changing the shape of tensors or the scope of the sub-expression, to fit the hardened capabilities of an external function. The `map` command calls the automatic search at Step 3 of the automatic searching algorithm (Section 6.2) to transform the expression and validate the mapping. In Section 6, we describe how our automatic search machinery factorizes sub-expressions from a larger expression to bind to different functions.

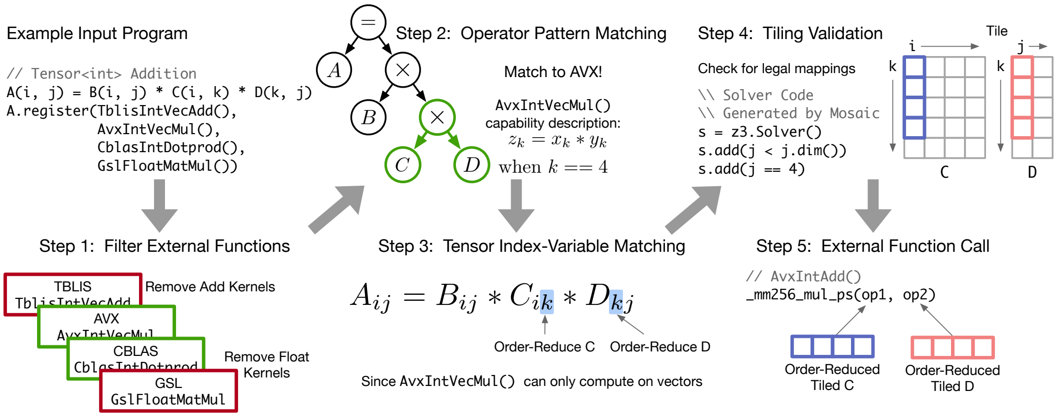
Promotion. The `promote` command adds an additional dimension of size 1 to a given tensor at the provided index, creating an equivalent tensor with higher order. This command enables `map` to use functions that operate on higher-order tensors to target computations that have lower-order inputs. For example, `map` can bind matrix-multiply functions to matrix-vector computations with `promote`. `Promote` is called on a CIN statement as `S.promote(`$\mathcal{T}$`, int)` where $\mathcal{T}$ is a tensor and `int` specifies the position that the extra dimension is inserted at. For a tensor of order n, int can range from $[0, n]$ inclusive since a dimension can be inserted between any n index variables.

Fix. The `fix` command enables `map` to target functions operating on lower-order tensors to computations that use high-order inputs. For example, the contraction at index j of a matrix-matrix multiply computation $A_{ik} = \sum_j B_{ij} * C_{jk}$ can be computed using a dot product function if indices for each i and k combination are fixed. The `fix` command specifies the index variables to ignore while mapping the computation. These indices are effectively held constant for the mapped computation.

6 AUTOMATED SEARCH FOR BINDINGS

While the scheduling commands in Section 5 provide users explicit control over function placement, some users may want to replace such fine-grained control with more automation. For example, a user may not have the time or expertise to make scheduling decisions, or have legacy code that they want to speedup without much effort. To meet the needs of such users, Mosaic provides an automated search mechanism that finds all valid bindings. Given a set of registered external functions, the automated search mechanism returns all the schedules that use those functions.

The search machinery consists of five steps shown in Figure 9. First, it filters all external functions whose parameter and return datatypes are different from the user-specified computation. Second, it applies mathematical rewrites, looking for equivalent operator patterns (see Section 6.1). Third, it matches index variables in the compute capability description to index variables in the user-defined statement (see Section 6.2), effectively resolving conditions on tensor order. Fourth, it performs a



```
// Tensor<int> Addition
A(i, j) = B(i, j) * C(i, k) * D(k, j)
A.register(TblisIntVecAdd(),
           AvxIntVecMul(),
           CblasIntDotprod(),
           GslFloatMatMul())
```

```
\\ Solver Code
\\ Generated by Mosaic
s = z3.Solver()
s.add(j < j.dim())
s.add(j == 4)
```

```
// AvxIntAdd()
_mm256_mul_ps(op1, op2)
```

Fig. 9. Steps in the automatic searching process.

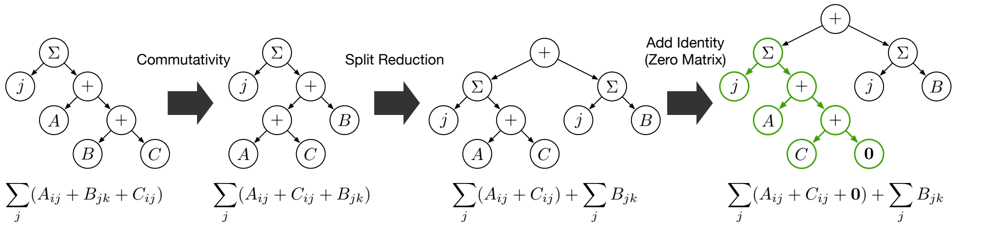

Commutativity
Split Reduction
Add Identity (Zero Matrix)
Σ
+
j
A
B
C
0
$\sum_j (A_{ij} + B_{jk} + C_{ij})$
$\sum_j (A_{ij} + C_{ij} + B_{jk})$
$\sum_j (A_{ij} + C_{ij}) + \sum_j B_{jk}$
$\sum_j (A_{ij} + C_{ij} + \mathbf{0}) + \sum_j B_{jk}$

Fig. 10. Series of mathematical rewrites to map to a function that has the capability: $\sum_j A_{ij} + B_{ij} + C_{ij}$.

tiling validation step that explores and validates potential index variable tilings (see Section 6.3) and ensures that constraints on dimension size are satisfied. Fifth, a check against the checker function described in Section 4 is performed. If the check returns true, the search is complete and the external function is called during code generation. If not, the search algorithm considers all permutations of rewrites up to a provided depth.

Optionally, users can tell Mosaic to pick a schedule out of all possible schedules at random. Therefore, with just two additional lines of code, users can target their legacy code to other functions through Mosaic's automatic search mechanism. Programmers may also choose to add an autoscheduler that ranks all possible scheduling options, which we leave as future work.

6.1 Operator Pattern Matching

The goal of this stage is to simply match the operator(s) in the user-provided sub-expression with the operator(s) in the external functions. The search algorithm ignores all information about tensor order and dimension leaving them to be resolved at a later stage. For example, if the user-defined expression performs an addition of two tensors, any function that computes additions of two tensors of any order (where a scalar is a tensor of order 0) will be identified as a potential match. Note that since tensor algebra also includes reductions over index variables, Mosaic considers reduction as another operator to match for correctness. For example, in the expression $a_i = b_i \cdot c_i$, substituting $b_i \cdot c_i$ with a dot product $\alpha = \sum_i x_i \cdot y_i$ will give an incorrect result even though the element-wise multiplication $(\cdot)$ operator pattern matches.

To expose matches, the algorithm applies basic mathematical rewrites (see Figure 10 for an example) such as distributivity, associativity, commutativity, splitting reductions (when all operands of the reductions are being added) and adding an identity operand. These rewrites are guided by the compute capability description. For example, if a function targets the addition of three operands, and the user-provided expression consists of two operands, adding an identity tensor is more fruitful than applying a commutativity or associativity transformation. Since Mosaic is rewriting expressions at a high-level mathematical IR (concrete index notation), it is easy to validate that the rewrites preserve the semantic meaning of the original expression.

6.2 Tensor Index-Variable Matching

At this stage, the operator pattern of the sub-statement is the same as that of the compute capability of the function, and Mosaic can associate each operand in the compute capability description with a corresponding tensor in the user-defined expression. For each pair of associated tensors, the algorithm checks whether it is possible to match the index variables used to index into the two tensors. To do so, Mosaic must check that (1) any reduction or free variable is matched with another reduction or free variable respectively, and (2) once an index variable has been mapped, it is mapped to the same index across all other tensor accesses.

However, there are many possible ways of selecting matching index variables. These choices emerge because of tensor reshape transformations and the concatenation operation between dynamically sized index lists. Because of the `fix` command, Mosaic can restrict the combination of indices under consideration. And, because of the `promote` command, Mosaic can increase the order of the tensor, giving the algorithm more indices to work with. As a heuristic, Mosaic never `promotes` the order of the tensor to be greater than the minimum order required by the function. Moreover, the concatenation operation between dynamically sized index lists can introduce several choices for how to split indices into concatenated dynamically sized index lists.

We use a brute-force search to enumerate index-variable choices. In practice, exhaustively searching this space is fast, adding a negligible slow-down in compilation time. This low overhead is due to two reasons. First, the tensors we need to schedule rarely have order greater than 4. In fact, only 5% of distinct expressions inputted into the TACO website [Kjølstad et al. 2022] contain tensors of order greater than 4. Second, we have not seen an interface that needs more than one dynamically sized index list. With only one list, the mapping for dynamically sized index lists is fixed, and there is no search space to explore. A list of legal mappings is sent to the next stage.

6.3 Tiling Validation

After the search mechanism has assigned each index variable to another variable or added it to a dynamic index list, any constraints that have been specified over the indices using the compute capability language (described in Figure 7) must be satisfied. To check that the constraints are satisfied, Mosaic generates a query targeting the Z3 theorem prover using the algorithm described in Figure 8 and validation from Section 5.1. However, this query is too restrictive as it neglects potential tilings of index variables. To include such tilings, we loosen the constraint that the index variable size must be equal to the size of the dimension it indexes into, and now permit sizes of index variables that are less than the sizes of their respective dimensions. With this weaker constraint, the algorithm checks for the satisfiability of the new model. If the model is unsatisfiable, Mosaic will give up. But, if the model is satisfiable, Mosaic will query the model to return valid values for every index variable's dimensions, adding an additional constraint requiring that the product of future tilings exceeds the product of previously returned tilings and stopping once an unsatisfiable model is reached. The largest tiling (the tiling that gives the largest product of the dimensions of the index variables) is selected as the final dimensions of the tiled tensor.

7 CODE GENERATION

In this section, we describe the code generation for the `map` command. The code generation produces code that calls external functions for any sub-expressions that have been mapped, orchestrates data

```
1 // Full Plus3T Expression
2 A(i, j, k) = B(i, j, k) + C(i, j, k) + D(i, j, k)
3
4 // Schedule sub-computation to TblisPlus3 which calculates
5 // X_I = Y_I + X_I where I = range(...)
6 A.getSchedule().bind(B(i, j, k)+C(i, j, k), TblisPlus3())
```

```
1  // Code for Plus3T where B(i,j,k) + C(i,j,k)
2  // is mapped to tblis_add_tensor.
3  void compute(taco_tensor_t * A, B, C, D){
4   // Emit variables to access tensor metadata
5   int A1_dim = A->dimensions[0];
6   float * A_vals = (float *) B->vals;
7   ...
8   // Declare and init. workspace tensor W.
9   float*W=malloc(sizeof(float)*num_val);
10  for (int i = 0; ...)
11   for (int j = 0; ...)
12    for (int k = 0; ...)
13     int index = ((i*C2_dim)+j)*C3_dim+k
14     // Copy the second operand into W.
15     W[index] = C_vals[index];
16
17  // Emit object declarations.
18  tblis_tensor t1; tblis_tensor t2;
19 // Emit setup functions.
20 tblis_init_tensor_helper_s(&t1, B_dim, 3, B);
21 tblis_init_tensor_helper_s(&t2, W_dims, 3, W);
22
23 // Call function that computes
24 // and stores result into t2
25 tblis_tensor_add(NULL,NULL,&t1,"ijk",&t2,"ijk");
26
27 // No teardown, compute result directly into W array
28
29 // Compute A(i,j,k)=W(i,j,k)+D(i,j,k) using TACO code
30 for (int i = 0; ...)
31   for (int j = 0; ...)
32     for (int k = 0; ...)
33       int index = ((i*C2_dim)+j)*C3_dim+k
34       // Copy the second operand into W.
35       A_vals[index] = W[index]+D_vals[index];
36
37 // Free the workspace tensor.
38 ... }
```

Fig. 11. Generated code for a scheduled Plus3T computation.

movement between the function and surrounding code, and generates code to natively compute sub-computations that have not been mapped.

The map command given in Section 5 replaces the sub-expression that is being computed by the function with a workspace tensor. First, this workspace tensor is declared (line 9, Figure 11). The result of the function call will be stored in the workspace tensor. Functions may store the result tensor back into an input operand. For example, tblis_tensor_add in the TBLIS library stores the result of the tensor addition into the second operand. So, before the code generation can start emitting code for the function call, we may need to copy the operand into the result workspace tensor (lines 11-16, Figure 11). During the next stages of code generation, the second operand is replaced by the workspace tensor.

Next, the compiler emits setup-function calls. Arguments are recursively lowered, and special user-defined objects used as arguments are declared before calls to setup functions. Similarly, the compiler emits calls to perform the computation and the teardown (lines 20-29, Figure 11). As an optimization, when the whole expression can be targeted to a single external function, the code generation does not use a temporary workspace to store the result of the function call. The original result tensor is used as the result tensor for the function call.

If no external function interfaces have been registered to Mosaic, then we default to TACO's code generation algorithm. TACO expresses the whole range of tensor algebra expressions and produces fused code by default. For sparse inputs, fused code can run asymptotically faster, while mapping computations to separate functions results in unfused code. A default compiler producing fast fused code gives users an additional option to choose the extent of fused versus unfused code.

8 EVALUATION

We evaluate the performance of Mosaic's generated code and its ability to search and find successful mappings. We contrast the performance of Mosaic's default code generator (TACO), to expressions

Table 2. The full expressions used to evaluate our compiler where sparse tensors are bolded.

Name	Expression	Name	Expression	Name	Expression
GEMV	$a_i = \sum_j B_{ij} c_j$	GEMM	$A_{ij} = \sum_j B_{ij} C_{jk}$	SDDMM	$A_{ij} = \sum_k \mathbf{B}_{ij} C_{ik} D_{jk}$
SpMV	$a_i = \sum_j \mathbf{B}_{ij} c_j$	SpMM	$A_{ij} = \sum_j \mathbf{B}_{ij} C_{jk}$	TTTP4	$A_{ijkl} = \sum_m B_{ijkl} * C_{im} D_{jm} E_{km} F_{lm}$
TTV	$A_{ij} = \sum_k B_{ijk} c_k$	Block-Sparse SpMM	$A_{ijkl} = \mathbf{B}_{ijmn} * C_{mnkl}$	SpMMAdd	$A_{ij} = \mathbf{B}_{ij} + \mathbf{C}_{ij}$

that are lowered to a mix of calls to external functions and TACO code using Mosaic. Our results show that there are regimes where fused code is the most performant and other regimes where a mix of generated code and external function calls is the most performant. Thus, we provide evidence for the utility of Mosaic's ability to mix generated code with calls to libraries.

Mosaic also enables a quick and systematic search over the design space created by the combination of sub-expressions and external functions. We demonstrate the quality of our search by evaluating along two axes: the number of discovered bindings and the speed of the automatic mapper. We also demonstrate that by dividing the specification of a function's capability into a checker function and a compute capability language, we get both expressibility and search speed.

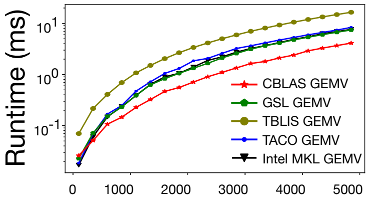
8.1 Methodology

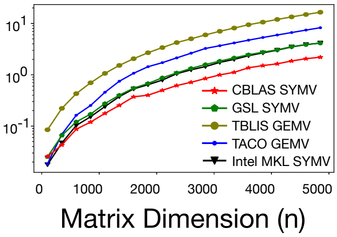
We evaluate Mosaic on real-world tensor expressions from prior work [Kjolstad et al. 2017; Singh et al. 2022] (see Table 2). The studies in Section 8.2, Section 8.3, and Section 8.4 use dense or uniformly random sparse synthetic data. Synthetic data lets us control and vary computation regimes, and demonstrate their underlying performance tradeoffs. We sweep the tensor dimension and sparsity (where applicable) of the synthetic data. All tensors are square with tensor dimension n referring to the size of all tensor ranks, and varying sparsity refers to changing the percentage of nonzero values. We also include benchmarks in Section 8.3 run on real-world sparse matrices from the SuiteSparse Matrix Collection [Davis and Hu 2011] listed in Table 3.

For our evaluation, we register 38 external functions from eight tensor algebra systems to Mosaic. Table 1 lists each tensor algebra system's features, backend platform, and programming language. AVX [Intel 2011] is a set of single instruction multiple data (SIMD) extensions to the x86 instruction set made available through Intel Intrinsics. CBLAS [Lawson et al. 1979], GSL [Gough 2009], and Intel MKL [Intel 2009] are all fixed-function, linear algebra libraries. We use the OpenBLAS implementation for the CBLAS library. GSL calls BLAS headers that are implemented by the Automatically Tuned Linear Algebra Software (ATLAS) project [Whaley and Petitet 2005] under the hood. TBLIS [Matthews 2016] is a library for dense tensor operations based on the BLIS framework [Van Zee and van de Geijn 2015]. As opposed to translating tensor operations into matrix operations and using BLAS to compute the decomposed pieces, TBLIS decomposes the operation into simpler, optimized functions. The cuSPARSE library [M Naumov 2010] is designed to run on NVIDIA GPUs and provides subroutines that can be used to compute linear algebra expressions on sparse matrices. Stardust [Hsu et al. 2022] compiles sparse tensor algebra to the Capstan reconfigurable dataflow accelerator [Rucker et al. 2021]. Stardust is unique in our evaluation since it is a compiler, is not natively embedded in the C ecosystem, and compiles to a hardware accelerator backend. We use the same methodology as in the original Stardust and Capstan papers [Hsu et al. 2022; Rucker et al. 2021], with details of the evaluation methodology provided below. [1]

All experiments are run on an AWS EC2 g4dn.xlarge instance with an NVIDIA Tesla T4 GPU and a four-socket Xeon(R) Platinum 8259CL CPU. The CPU has a 64 KiB L1 data and instruction cache, 2000 KiB L2 cache, 3580 KiB L3 cache, and 16082 MB RAM. The machine runs Ubuntu 22.04.1

[1] Stardust is written in C++ and generates Spatial code, a Scala-embedded DSL for accelerator design [Koeplinger et al. 2018]. All Spatial applications are compiled [Zhang et al. 2021b] and run using the same cycle-accurate Capstan simulator modeling 4 channels of HBM-2E (at 1800 GB/s) [Kim et al. 2016] and a non-ideal network [Zhang et al. 2019].





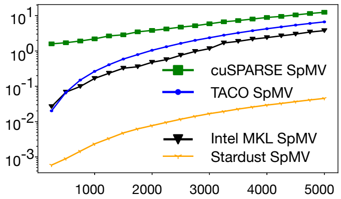


Fig. 12. Log performance for varying matrix dimensions when using Mosaic to bind functions to dense GEMV.

Fig. 13. Log performance across matrix dimension for SYMV (a dense, symmetric GEMV).

Fig. 14. Log performance across various matrix dimension for SpMV where $\%\text{nnz}_B = 20\%$.

LTS and is clocked at 2500 MHz. All code is compiled using GCC 11.3.0 with `-O3 -ffast-math` optimizations on. For all benchmarks, we report the median runtime of 10 iterations with 10 warmup iterations. To generate pure TACO code, we simply do not register any plugins to Mosaic, kicking off its default code generator.

8.2 Comparing Systems

In this section, we quantitatively show that hand-written functions and generated code from different systems have performance benefits in different situations. Thus, we motivate a system like Mosaic that can automatically map expressions to a mix of external functions and generated code. We compare the performance of several systems on matrix-vector multiplication using three types of matrices: a dense matrix (GEMV), a symmetric matrix (SYMV), and a sparse matrix (SpMV). Across the three types of matrix data, no one system (including TACO) is sufficient to outperform the rest. Therefore, it is beneficial for users to explore several options through Mosaic to find the most performant implementation for their use-case.

For dense GEMV, calling a CBLAS and GSL function provides an average of 1.9× and 1.1× speedup respectively (Figure 12) over the code generated by TACO. This difference is to be expected since TACO is not optimized for dense code. TACO computes GEMV using a naive implementation whereas BLAS is a 40-year-old, hand-optimized library and GSL uses an ATLAS implementation of BLAS, which tunes for machine-specific quirks.

For the symmetric GEMV computation, we notice an average of 2.9× and 1.8× speedup when we use the BLAS and MKL library implementations respectively over TACO-generated code (Figure 13). The three libraries—MKL, GSL and CBLAS—have special functions for computing matrix-vector products with symmetric matrices (SYMV). SYMV implementations allow these systems to deliver better performance over TACO by exploiting the mathematical structure of the data.

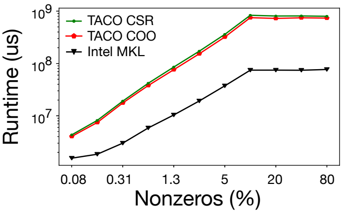


Fig. 15. Log performance comparing tensor formats coordinate list (COO) and compressed sparse row (CSR) for SpMMAdd, where $n = 10,000$.

For the SpMV computation (Figure 14), we cannot compare against GSL and CBLAS as these libraries do not have functions that perform sparse linear algebra operations. In Figure 14, we see an order of magnitude difference between the Stardust and TACO runtimes because Stardust compiles to the Capstan accelerator, a dataflow architecture built for sparse computations. As the GPU is built for inference, we do not see an advantage of targeting cuSPARSE.

Finally, we also show the benefits of paying the penalty of runtime format conversion to target an external library. Figure 15 shows that the TACO implementation for SpMMAdd using the COO format is slightly more performant than the one using the CSR format. However, when the COO matrix is converted to CSR at runtime to meet the format constraints of the MKL library, we see a maximum speedup of 10.1× over the TACO COO implementation.

Figures 12 to 14 show performance variation due to the underlying mathematical properties of the data, the availability of specialized hardware, and the difference in data structure formats. This section shows that systems optimize for different subsets of these factors, and Sections 8.3 and 8.4 will show that this specialization can be leveraged by generating performant code through Mosaic.

8.3 Leveraging System Specialization: SDDMM

For expressions involving sparse tensors, even for a fixed expression and fixed format, a fixed mapping does not always yield maximum performance benefits. As the sparsity of the tensor changes, the optimal function to map to a sub-expression also changes. Because of this behavior, we show that there exist computations that benefit from both fused and factorized code optimizations and they can be scheduled through Mosaic.

We compare the performance of sampled dense-dense matrix multiplication (SDDMM, see Table 2) across various tensor dimensions and sparsities over a set of external functions mapped using Mosaic. We chose SDDMM since it is a core building block, and often the bottleneck, for many applications including graph learning, matrix completion, and alternating least squares.

For a low percentage of nonzeros $\%\text{nnz}_B < 0.24\%$, Figure 16 shows that fused code generated by TACO generally outperforms code that calls external functions. When mapping SDDMM to a dense-dense matrix multiplication function, Mosaic needs to insert a temporary to store the result of the function call. This insertion produces unfused code: CD is computed first and then multiplied by the compressed tensor B. The resulting asymptotic complexity is proportional to the total number of dense elements ($O(n^3)$ here). TACO, on the other hand, produces fused code, multiplying elements of C and D only when there is a corresponding nonzero in the sparse input B. The runtime is then $O(\text{nnz}_B * n)$ where nnz_B denotes the number of nonzeroes in B, which is very small when density is low (and sparsity is high). Any performance gained from a fast dense-dense matrix-multiply is lost to computing redundant values.

For a high percentage of nonzeros $\%\text{nnz}_B > 0.24\%$, we can see the benefit of a fast dense matrix multiplication function. Even though the unfused code requires more storage, is still calculating redundant values, and is asymptotically worse, BLAS and GSL are poised to take advantage of machine specific information and recuperate the cost of doing extra work. Moreover, the run time for dense-function mappings remain constant as shown in Figure 16 since the amount of work for dense systems does not change with the sparsity of B. However, as the number of nonzeros increase, TACO's runtime steadily rises, resulting in an order of magnitude slowdown.[2]

When we fix the $\%\text{nnz}_B$ to be 40% and sweep dimension (Figure 19), we see an average of 31.5×, 21.7× and 18.8× speedup over TACO when the multiplication of CD is mapped to dense matrix multiplication functions provided by BLAS, GSL and TBLIS respectively. The fused code generated by TACO suffers because the percentage of nonzeroes is high, and GCC cannot use usual optimization strategies to analyze deep loop nests that iterate over sparse data structures.

Finally, we test Mosaic's performance on real-world data from the SuiteSparse matrix collection [Davis and Hu 2011] on the matrices listed in Table 3. We order all matrices from the SuiteSparse matrix collection with respect to the number of non-zeroes they contain, limiting the maximum number of non-zeroes at 50,000 due to machine memory constraints. From this ordered list, we select, at random, 4 matrices each from the 50 matrices with the least, median, and largest number

[2] There is no benefit in mapping dot-product functions to the dense matrix multiplication. Mosaic needs to create temporaries for vector data collection, causing slowdown, and dot-product bindings `fix` the schedule to an ikj-loop ordering, resulting in an inner-product algorithm that performs asymptotically worse [Gustavson 1978; Hsu et al. 2023; Zhang et al. 2021a].

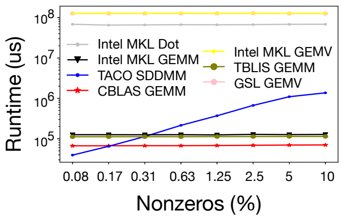


Fig. 16. Log performance of mapped SDDMM using Mosaic as B's number of nonzeros vary, where B's dimension is $n = 2000$.

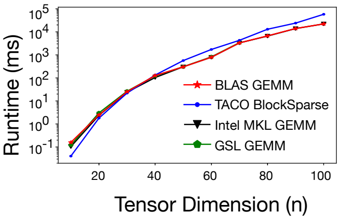


Fig. 17. Log performance of mapped block-sparse matrix multiply using Mosaic across varying dimension, where %nnz$_B$ = 5%.

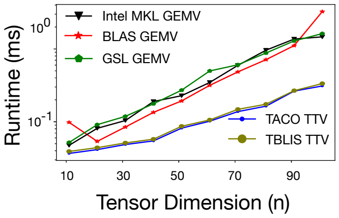


Fig. 18. Log performance of dense tensor-times-vector (TTV) across varying tensor dimensions.

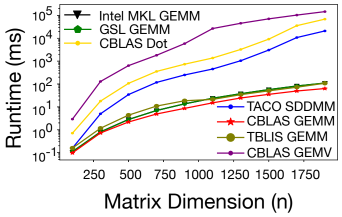


Fig. 19. Log performance of external functions mapped to SDDMM using Mosaic as the dimension of B is varied when %nnz$_B$ = 40%

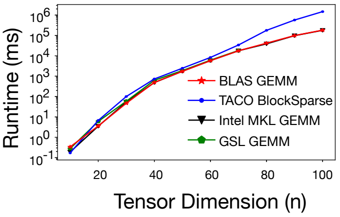


Fig. 20. Log performance of mapped block-sparse matrix multiply using Mosaic across varying dimension, where %nnz$_B$ = 20%

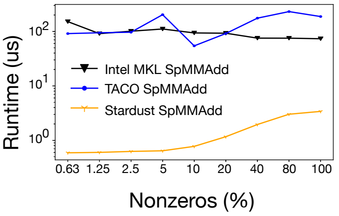


Fig. 21. Log performance of sparse matrix-matrix addition (SpMMAdd) across varying number of nonzeros, where $n = 200$.

Table 3. Matrices from the SuiteSparse matrix collection [Davis and Hu 2011].

Name	Domain	Dimensions	Nonzeros	Density (%)
Trec5	Combinatorial Problem	3×7	12	57.14
Ragusa18	Directed Weighted Graph	23×23	64	12.09
lpi_bgprtr	Linear Programming Problem	20×40	70	8.75
lp_sc50b	Linear Programming Problem	50×78	148	3.79
cavity02	Subsequent Computational Fluid Dynamics Problem	317×317	5,923	5.89
cavity03	Subsequent Computational Fluid Dynamics Problem	317×317	7,311	7.28
lp_nug08	Linear Programming Problem	$912 \times 1,632$	7,296	0.49
m3plates	Acoustics Problem	$11,107 \times 11,107$	6,639	0.01
IG5-12	Combinatorial Problem	$2,296 \times 2,875$	46,260	0.70
g7jac020sc	Economic Problem	$5,850 \times 5,850$	42,568	0.12
gemat1	Power Network Problem	$4,929 \times 10,595$	46,591	0.09
mimo28x28_system	Eigenvalue/Model Reduction Problem	$13,251 \times 13,251$	48,737	0.03

of nonzeros to instantiate the sparse matrix in the SDDMM expression. Figure 22 shows that as the dimension of the sparse matrices increases, the benefits of using a faster dense-dense matrix multiply eclipse the cost of doing redundant work. However, as density decreases, the cost of doing redundant work undoes the benefit of a fast multiply. While MKL and BLAS are 0.28× and 0.31× as fast as the TACO implementation for the `lp_sc50b` matrix, the performance of MKL and BLAS improves to be 1.33× and 1.69× faster than the TACO baseline for the `cavity02` matrix. The maximum speedup is observed in the case of the `IG5-12` matrix, with MKL and BLAS being 3.57× and 6.48× faster than the TACO implementation. However, MKL and BLAS are only 0.29× and 0.57× as fast as TACO for the `mimo28x28_system` matrix, which has a density of only 0.03%

8.4 Performance Comparisons on Real-World Expressions

We demonstrate Mosaic's ability to bind expressions to commonly used functions on a wide range of real-world expressions using the `map` scheduling command (Section 5). That is, after we

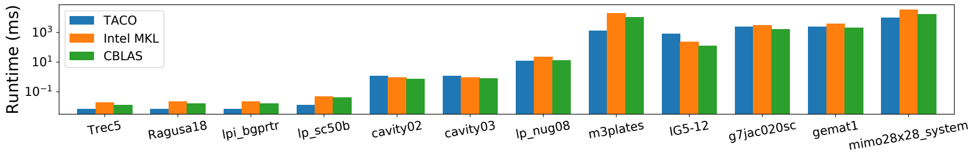


Fig. 22. Log performance of external functions mapped to SDDMM using Mosaic for SuiteSparse matrices.

indicated a function substitution through the map command, Mosaic was able to fix indices and tile computation to fit within the constraints of the function. Through Mosaic, users can rapidly compare the performance of possible factorizations side-by-side.

We evaluate performance on block-sparse matrix-matrix multiplication (Block-Sparse SpMV), tensor-times-vector multiplication (TTV), and sparse matrix-matrix addition (SpMMAdd). Block-sparse matrix-matrix multiplication demonstrates higher-order sparse computation that can be mapped to dense functions (see Figures 17 and 20). TTV demonstrates Mosaic's ability to handle higher-order expressions (see Figure 18). Finally, SpMMAdd demonstrates an expression with additions and two sparse operands (see Figure 21).

Block-sparse matrices contain dense blocks of values that are spread uniformly throughout the matrix. These blocks effectively act as dense matrices inside a larger sparse matrix. So, while computing the product of a block-sparse matrix with a dense matrix, we can use the blocks of values as inputs to a dense-dense matrix multiply function. Figure 17 and Figure 20 show the results of this mapping with 5% and 20% nonzeroes respectively. For 5% of nonzeroes and $n < 40$, TACO outperforms other mappings. But, as the dimension increases, BLAS delivers a 2.4× speedup. To call a dense-dense matrix multiply function, Mosaic packs the blocked values into an input array that corresponds to the same format the external function is expecting data in. When dimension $n < 40$, an optimized matrix multiply cannot recover the cost of this packing and unpacking. However, as n increases, this cost becomes negligible and we reap the benefits of a hand-optimized matrix multiply. For the 20% sparsity case, we see a similar trend. Since the number of nonzeroes is already higher at lower dimensions, we do not see much difference between the TACO and BLAS implementations. However, we see a 7.4× speedup over TACO as the dimension increases.

Next, we evaluate Mosaic's performance on the TTV expression when it is mapped to GEMV and TTV functions. From Figure 18, we notice that neither the GEMV nor the TTV functions can compete with TACO. The optimizations discovered by GCC on the TACO code trump optimizations performed by TBLIS. We found that when running the TTV benchmark on GCC 7.5.0, a lower version than what is shown, TBLIS outperforms TACO. While underlying systems and libraries evolve, handwritten code does not. Therefore, having a system like Mosaic that can incorporate new systems and automatically generate code that targets a new selection of functions helps users rapidly adapt old code to new developments (like new GCC versions).

Finally, we look at the performance of Mosaic when computing SpMMAdd (see Figure 21). Similar to results found in [Hsu et al. 2022], Stardust boosts performance by up to 173× over TACO.

8.5 External Function Abstraction Study

We perform a lines of code (LOC) study on the external function abstractions in Mosaic. The LOC numbers from Table 4 demonstrate that the development of external function abstractions for Mosaic is relatively straightforward since each expression requires 20 lines of code on average. Furthermore, each of these external functions only needs to be written once and is usable by all Mosaic users. Table 4 also shows a subset of the total number of external functions (38) we were able to plug in to Mosaic in a limited amount of time.

Table 4. A subset of the external functions plugged into Mosaic for experiments described in Section 8. Functions that compute two or more expressions are underlined after their first instance (denoting duplicates).

Name	Expression	Lines of Code (LOC)						
		BLAS	GSL	TBLIS	AVX	Stardust	cuSPARSE	MKL
VecAdd	$A_i = B_i + C_i$			16	16			
Saxpy	$A_i = B_i + \gamma D_i$	10	15					
Dot	$\alpha = B_i * C_i$	12	16	17	18			
GEMV	$A_i = B_{ij} * C_j$	11	20	19				15
SGEMV	$A_i = \alpha * B_{ij} * C_j + \beta * D_i$	$\underline{11}$	$\underline{20}$			30		
SpMV	$A_i = \alpha * B_{ij} * C_j + \beta * D_i$			30			43	21
SpMMAdd	$A_{ik} = B_{ij} + C_{ij}$					30		18
GEMM	$A_{ik} = B_{ij} * C_{jk}$	12	19	$\underline{19}$		30		17
SGEMM	$A_{ik} = \alpha * B_{ij} * C_{jk} + \beta * C_{ik}$	$\underline{12}$	$\underline{19}$					$\underline{17}$
TTM	$A_{ijk} = B_{ijl} * C_{kl}$			$\underline{18}$		30		
Plus3	$A_{ijk} = B_{ijk} + C_{ijk}$			18		30		

Table 5. End-to-end times and number of distinct matching found by Mosaic (excluding redundant mappings generated by consecutively adding/multiplying by an identity operand).

Name	Expression	# Registered Functions = 3		# Registered Functions = 9	
		End-to-End Runtime (s)	# Mappings	End-to-End Runtime (s)	# Mappings
VecAdd	$A_i = B_i + C_i$	0.027195	3	0.439966	7
Dot	$\alpha = B_i * C_i$	0.034908	3	0.102808	7
GEMV	$A_i = B_{ij} * C_j$	0.037356	3	0.107068	7
SGEMM	$A_{ik} = B_{ij} * C_{jk} + C_{ik}$	0.124247	5	0.338617	13
Plus3	$A_{ijk} = B_{ijk} + C_{ijk}$	0.037763	3	1.19884	7
SDDMM	$A_{ik} = \mathbf{B}_{ik} * C_{ij} * D_{jk}$	0.09578	2	0.351564	13

8.6 Evaluation of the Search

System	Capability Language	Checker Linear Search	Checker Random Search
Stardust	0.421 sec	21.36 sec	1.72 sec
AVX	0.0201 sec	0.00163 sec	14.027 sec

Fig. 23. Valid tiling times for the AVX and Stardust functions when using the compute capability language or an opaque checker function (using both a linear and random search of the checker function).

We present the end-to-end runtime and number of mappings found for the automatic mapping algorithm in Table 5. For this experiment, we ran our search twice with 3 and 9 external functions registered to Mosaic: (CBLAS); and Gemm (CBLAS, MKL, GSL), Dot (CBLAS, MKL, GSL) and VecAdd (CBLAS, MKL, AVX). Additionally, the depth of mathematical rewrites was set to 3. We notice that the runtime of the search algorithm depends on the complexity of the expression i.e. the scope of mathematical rewrites and scales linearly with the number of registered functions. We also see a significant jump in the VecAdd and Plus3 expressions as the number of registered functions increases. In both expressions, the mapper found valid AVX mappings and since the AVX interface requires tiling and has a compute capability language constraint, extra time is spent generating and executing the resultant Z3 query.

Benefits of the Compute Capability Language. To check whether the compute capability language gives us any benefits over the checker function, we timed how quickly Mosaic can discover mappings when both descriptions or only the checker function are given. We ran this experiment for the AVX and Stardust functions since they have the most interesting language constraints. AVX requires vectors to be exactly of size 4 (with floating-point data), and Stardust only allows tensors that contain less than 65,536 values. A tiling transformation is necessary to target expressions containing large tensors that do not fit these constraints. When finding a concrete tile size using only the checker function, we use two strategies. First, we use a linear search from 1 to 65,536 and call the checker function on each tile size to validate correctness. After finding a valid tiling, we maximize tile size by continuing the linear search until we find a tile size that the checker function rejects. We also use a randomized search between 1 and 65,536. When the checker function returns a match on a randomly chosen tile size, a linear search starts to find the maximum tile size.

The times presented using both the linear and random search are noted in Figure 23. Picking a single strategy to step through transformations is hard because of the range of functions available. While the linear increment was beneficial for small tilings used in AVX, a random increment was much more productive in finding larger tilings for Stardust. Using information about Stardust, we were able to bound our search, otherwise, picking a maximum tile size may be a challenging decision in itself. By encoding such information in the compute capability language, we are able to apply precise rewrites without having to search for concrete parameters (like tile sizes) in the dark.

9 RELATED WORK

Mosaic is the first programming system for sparse tensor algebra that can generate code that mixes fully generated code with calls to external functions. It uses a scheduling language to reshape and bind expressions to functions, verifies that the bindings are correct, and also provides an automated search system that given an expression and one or more functions, will return to the user schedules that take advantage of those functions. In this section, we discuss other programming systems for sparse tensor algebra, other scheduling languages, fully-automated scheduling systems, libraries, and domain-specific hardware that could be used by our system.

9.1 Programming Systems for Sparse Tensor Algebra

The traditional way to compute sparse linear and tensor algebra expressions is to write a sequence of calls to libraries like Intel MKL [Intel 2009] or cuSPARSE [M Naumov 2010]. Several systems have been designed to automate this mapping, such as MATLAB's sparse support [Gilbert et al. 1992], the MATLAB Tensor Toolbox [Kolda and Bader 2006], Julia [Bezanson et al. 2017], and CTF [Solomonik and Hoefler 2015; Solomonik et al. 2014]. These systems have excellent performance for expressions for which they have a suitable function to call, but their performance can suffer for other expressions, as these have to be factorized to use a fixed set of available functions. These systems are also hard-coded to utilize specific functions, and a user cannot add new functions to this set.

More recently, several compilers have been developed that compile sparse tensor algebra expressions to imperative code. These include the TACO compiler [Kjolstad et al. 2017], the MLIR SparseTensor Dialect [Bik et al. 2022], and the Sparse Polyhedral Framework [Zhao et al. 2022]. These systems can compile sparse and dense tensor algebra expressions all the way down to imperative code, but cannot mix the generated code with calls to external functions. Thus, their performance suffers where a library can compute a specific expression—such as dense GEMM—faster than their generated code. Some compilers, like SparseTIR [Ye et al. 2023], can leverage hand-optimized code for CPUs or domain-specific architecture like the NVIDIA Tensor Cores [NVIDIA 2022b]. However, unlike Mosaic, SparseTIR cannot generate efficient fused code with coiteration and cannot be extended to utilize other external functions.

9.2 Scheduling Languages, Binding, and Automatic Scheduling

Starting with the Halide compiler [Ragan-Kelley et al. 2012], many domain-specific programming systems have adopted scheduling languages, including TVM [Chen et al. 2018a], TACO [Kjølstad et al. 2019; Senanayake et al. 2020], Lift/Elevate [Hagedorn et al. 2020]. The CHiLL [Chen et al. 2007] and POET [Yi 2012] compilers also enable the separate scheduling of C loops. The scheduling language of the Exo compiler [Ikarashi et al. 2022] includes commands to substitute code for specialized user-defined instruction. Out of these systems, the Exo compiler is the closest to our approach, however, its affine loop-nest IR is not suitable for sparse computations. Exo is also tailored for working with low-level instructions. Mosaic, in contrast, can compile sparse tensor algebra and can replace whole sub-computations with calls to external functions.

In addition to manual approaches to scheduling, researchers have explored automatic scheduling systems [Adams et al. 2019; Anderson et al. 2021; Chen et al. 2018a; Mullapudi et al. 2016; Ragan-Kelley et al. 2013; Zheng et al. 2022]. Out of these systems, the AMOS [Zheng et al. 2022] compiler is closest to our approach. The Amos compiler is an automatic compilation framework for spatial hardware targeting dense tensor algebra. Although their compute abstraction is similar to Concrete Index Notation, they target domain-specific accelerators. Mosaic on the other hand, lives within the C ecosystem and can target both CPUs and specialized hardware, albeit the external function must handle hierarchical memory accesses itself.

9.3 Libraries and Domain-Specific Hardware

There are a large number of libraries that bundle hand-written for sparse and dense linear and tensor algebra computations. Examples for dense tensor algebra include the BLAS library [Lawson et al. 1979], Intel MKL [Intel 2009], the GNU Scientific Library [Gough 2009], and NVIDIA CUTLASS [NVIDIA 2022a]. More recently, researchers have developed hand-optimized libraries for dense tensor algebra, most prominently the TBLIS library [Matthews 2016]. Sparse linear and tensor algebra libraries are also common, including Intel MKL, and NVIDIA cuSPARSE [M Naumov 2010]. Moreover, in the last eight years, many domain-specific architectures have been developed that accelerate specific expressions and, in some cases, a class of expressions. These domain-specific architectures include both fixed-function accelerators [He et al. 2020; Pal et al. 2018; Qin et al. 2020; Srivastava et al. 2020a; Zhang et al. 2021a] and more general reconfigurable accelerators [Dadu et al. 2019; Hegde et al. 2019; Hsu et al. 2023; Rucker et al. 2021; Srivastava et al. 2020b]. The Mosaic compiler is designed to take advantage of these libraries and domain-specific architectures by generating code that composes them and that fills sub-expressions that cannot be mapped to available external functions or hardware.

10 CONCLUSION

We propose Mosaic, a system that can compose externally defined library functions to implement an arbitrary sparse tensor algebra expression, unlocking good performance where hand-optimized implementations or specialized hardware exist. It fills in the gaps that are not provided by the libraries, guaranteeing generality in both expressions and data structures, as well as fusion. As opposed to writing code that is hard-wired to utilize a small set of fixed libraries, users can choose Mosaic, allowing them to access these libraries under the umbrella of a single host compiler. Because of the completion of partial schedules, users can use external functions with ease without having to sift through documentation for hours. Because Mosaic also validates bindings and automatically generates code, users can have more trust in optimizations and their supporting code. We hope this empowers users to utilize upcoming, state-of-the-art libraries with a few lines of code and minimal refactoring. We also hope that performance engineers contribute to Mosaic's growing library of external functions so that every Mosaic user may benefit.

ACKNOWLEDGMENTS

We thank Jason Ansel, Pranav Bansal, James Dong, Timothy Gu, Praneeth Kolichala, Scott Kovach, Rubens Lacouture, Luca Pistor, Alexander J. Root, Shiv Sundram, Rohan Yadav, and Bobby Yan for their helpful feedback. This work was supported in part by the National Science Foundation under Grant CCF-2143061 and 1937301 and the GRFP Fellowship, the Google Research Scholar program, and Stanford Data Analytics for What's Next (DAWN) Affiliate Program. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the aforementioned funding agencies.

ARTIFACT

The Mosaic compiler code is open-source and can be found at the mosaic repository on GitHub. Instructions for reproducibility and reusability are available on an archived version on Zenodo [Bansal et al. 2023] and at the mosaic-artifact repository on GitHub. Benchmarking results depend on access to specialized hardware and vary based on external library versions and machine configuration.

REFERENCES

Andrew Adams, Karima Ma, Luke Anderson, Riyadh Baghdadi, Tzu-Mao Li, Michaël Gharbi, Benoit Steiner, Steven Johnson, Kayvon Fatahalian, Frédo Durand, and Jonathan Ragan-Kelley. 2019. Learning to Optimize Halide with Tree Search and Random Programs. *ACM Trans. Graph.* 38, 4, Article 121 (jul 2019), 12 pages. https://doi.org/10.1145/3306346.3322967

Peter Ahrens, Fredrik Kjolstad, and Saman Amarasinghe. 2022. Autoscheduling for Sparse Tensor Algebra with an Asymptotic Cost Model. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 269–285. https://doi.org/10.1145/3519939.3523442

Luke Anderson, Andrew Adams, Karima Ma, Tzu-Mao Li, Tian Jin, and Jonathan Ragan-Kelley. 2021. Efficient Automatic Scheduling of Imaging and Vision Pipelines for the GPU. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 109 (October 2021), 28 pages. https://doi.org/10.1145/3485486

Brett W. Bader and Tamara G. Kolda. 2006. Algorithm 862: MATLAB Tensor Classes for Fast Algorithm Prototyping. *ACM Trans. Math. Softw.* 32, 4 (dec 2006), 635–653. https://doi.org/10.1145/1186785.1186794

Brett W. Bader and Tamara G. Kolda. 2007. Efficient MATLAB Computations with Sparse and Factored Tensors. *SIAM J. Sci. Comput.* 30, 1 (dec 2007), 205–231. https://doi.org/10.1137/060676489

Manya Bansal, Olivia Hsu, Kunle Olukotun, and Fredrik Kjolstad. 2023. *Artifact for Mosaic: An Interoperable Compiler for Tensor Algebra.* https://doi.org/10.5281/zenodo.7710602

Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah. 2017. Julia: A Fresh Approach to Numerical Computing. *SIAM Rev.* 59, 1 (2017), 65–98. https://doi.org/10.1137/141000671 arXiv:https://doi.org/10.1137/141000671

Aart Bik, Penporn Koanantakool, Tatiana Shpeisman, Nicolas Vasilache, Bixia Zheng, and Fredrik Kjolstad. 2022. Compiler Support for Sparse Tensor Computations in MLIR. *ACM Trans. Archit. Code Optim.* 19, 4, Article 50 (sep 2022), 25 pages. https://doi.org/10.1145/3544559

Chun Chen, Jacqueline Chame, and Mary W. Hall. 2007. *CHiLL : A Framework for Composing High-Level Loop Transformations.* Technical Report.

Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018a. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation* (Carlsbad, CA, USA) *(OSDI'18)*. USENIX Association, USA, 579–594.

Yu-Hsin Chen, Tien-Ju Yang, Joel Emer, and Vivienne Sze. 2018b. Eyeriss v2: A Flexible Accelerator for Emerging Deep Neural Networks on Mobile Devices. https://doi.org/10.48550/ARXIV.1807.07928

Jack Choquette, Wishwesh Gandhi, Olivier Giroux, Nick Stam, and Ronny Krashinsky. 2021. NVIDIA A100 Tensor Core GPU: Performance and Innovation. *IEEE Micro* 41, 2 (2021), 29–35. https://doi.org/10.1109/MM.2021.3061394

Stephen Chou, Fredrik Kjolstad, and Saman Amarasinghe. 2018. Format Abstraction for Sparse Tensor Algebra Compilers. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 123 (oct 2018), 30 pages. https://doi.org/10.1145/3276493

Vidushi Dadu, Jian Weng, Sihao Liu, and Tony Nowatzki. 2019. Towards General Purpose Acceleration by Exploiting Common Data-Dependence Forms. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO '52)*. Association for Computing Machinery, New York, NY, USA, 924–939. https://doi.org/10.1145/3352460.3358276

Steven Dalton, Nathan Bell, Luke Olson, and Michael Garland. 2014. Cusp: Generic Parallel Algorithms for Sparse Matrix and Graph Computations. http://cusplibrary.github.io/ Version 0.5.0.

Timothy A. Davis and Yifan Hu. 2011. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.* 38, 1, Article 1 (dec 2011), 25 pages. https://doi.org/10.1145/2049662.2049663

Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Budapest, Hungary) *(TACAS'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 337–340.

John R Gilbert, Cleve Moler, and Robert Schreiber. 1992. Sparse matrices in MATLAB: Design and implementation. *SIAM journal on matrix analysis and applications* 13, 1 (1992), 333–356.

Brian Gough. 2009. *GNU scientific library reference manual.* Network Theory Ltd.

Fred G. Gustavson. 1978. Two Fast Algorithms for Sparse Matrices: Multiplication and Permuted Transposition. *ACM Trans. Math. Softw.* 4, 3 (1978).

Bastian Hagedorn, Johannes Lenfers, Thomas Koehler, Sergei Gorlatch, and Michel Steuwer. 2020. A Language for Describing Optimization Strategies. arXiv:2002.02268 [cs.PL]

Xin He, Subhankar Pal, Aporva Amarnath, Siying Feng, Dong-Hyeon Park, Austin Rovinski, Haojie Ye, Yuhan Chen, Ronald Dreslinski, and Trevor Mudge. 2020. *Sparse-TPU: Adapting Systolic Arrays for Sparse Matrices.* Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3392717.3392751

Kartik Hegde, Hadi Asghari-Moghaddam, Michael Pellauer, Neal Crago, Aamer Jaleel, Edgar Solomonik, Joel Emer, and Christopher W Fletcher. 2019. Extensor: An accelerator for sparse tensor algebra. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture.* 319–333.

Olivia Hsu, Alexander Rucker, Tian Zhao, Kunle Olukotun, and Fredrik Kjolstad. 2022. Stardust: Compiling Sparse Tensor Algebra to a Reconfigurable Dataflow Architecture. https://doi.org/10.48550/ARXIV.2211.03251

Olivia Hsu, Maxwell Strange, Ritvik Sharma, Jaeyeon Won, Kunle Olukotun, Joel S. Emer, Mark A. Horowitz, and Fredrik Kjølstad. 2023. The Sparse Abstract Machine. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 710–726. https://doi.org/10.1145/3582016.3582051

Jianyu Huang, Devin A. Matthews, and Robert A. van de Geijn. 2017. Strassen's Algorithm for Tensor Contraction. *CoRR* abs/1704.03092 (2017). arXiv:1704.03092 http://arxiv.org/abs/1704.03092

Yuka Ikarashi, Gilbert Louis Bernstein, Alex Reinking, Hasan Genc, and Jonathan Ragan-Kelley. 2022. Exocompilation for Productive Programming of Hardware Accelerators. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 703–718. https://doi.org/10.1145/3519939.3523446

2009. *Intel Math Kernel Library. Reference Manual.* Intel Corporation, Santa Clara.

2011. *Intel Advanced Vector Extensions Programming Reference.* Intel Corporation, Santa Clara, USA. https://www.intel.com/content/dam/develop/external/us/en/documents/36945

Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. *SIGARCH Comput. Archit. News* 45, 2 (jun 2017), 1–12. https://doi.org/10.1145/3140659.3080246

Yoongu Kim, Weikun Yang, and Onur Mutlu. 2016. Ramulator: A Fast and Extensible DRAM Simulator. *IEEE Comput. Archit. Lett.* 15, 1 (Jan. 2016), 45–49. https://doi.org/10.1109/LCA.2015.2414456

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

Fredrik Kjølstad, Stephen Chou, and Saman Amarasinghe. 2022. Taco: The tensor algebra compiler. http://tensor-compiler.org/

Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. 2017. The Tensor Algebra Compiler. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 77 (oct 2017), 29 pages. https://doi.org/10.1145/3133901

David Koeplinger, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, Stefan Hadjis, Ruben Fiszel, Tian Zhao, Luigi Nardi, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. 2018. Spatial: A Language and Compiler for Application Accelerators. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018)*. Association for Computing Machinery, New York, NY, USA, 296–311. https://doi.org/10.1145/3192366.3192379

Tamara G. Kolda and Brett W. Bader. 2006. MATLAB Tensor Toolbox, Version 00. https://www.osti.gov/biblio/1230898

Yehuda Koren, Robert Bell, and Chris Volinsky. 2009. Matrix Factorization Techniques for Recommender Systems. *Computer* 42, 8 (2009), 30–37. https://doi.org/10.1109/MC.2009.263

Chuck L Lawson, Richard J. Hanson, David R Kincaid, and Fred T. Krogh. 1979. Basic linear algebra subprograms for Fortran usage. *ACM Transactions on Mathematical Software (TOMS)* 5, 3 (1979), 308–323.

P Vandermersch M Naumov, LS Chien. 2010. Cusparse library. *GPU Technology Conference (GTC)* (2010).

MATLAB. 2010. *version 7.10.0 (R2010a).* The MathWorks Inc., Natick, Massachusetts.

Devin Matthews. 2016. High-Performance Tensor Contraction without BLAS. *CoRR* abs/1607.00291 (2016). arXiv:1607.00291 http://arxiv.org/abs/1607.00291

Ravi Teja Mullapudi, Andrew Adams, Dillon Sharlet, Jonathan Ragan-Kelley, and Kayvon Fatahalian. 2016. Automatically Scheduling Halide Image Processing Pipelines. *ACM Trans. Graph.* 35, 4, Article 83 (jul 2016), 11 pages. https://doi.org/10.1145/2897824.2925952

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

NVIDIA. 2022a. *CUTLASS.* NVIDIA. https://developer.nvidia.com/blog/cutlass-linear-algebra-cuda/

NVIDIA. 2022b. *Tensor Cores.* NVIDIA. https://www.nvidia.com/en-us/data-center/tensor-cores/

Subhankar Pal, Jonathan Beaumont, Dong-Hyeon Park, Aporva Amarnath, Siying Feng, Chaitali Chakrabarti, Hun-Seok Kim, David Blaauw, Trevor Mudge, and Ronald Dreslinski. 2018. OuterSPACE: An Outer Product Based Sparse Matrix Multiplication Accelerator. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA).* 724–736. https://doi.org/10.1109/HPCA.2018.00067

Eric Qin, Ananda Samajdar, Hyoukjun Kwon, Vineet Nadella, Sudarshan Srinivasan, Dipankar Das, Bharat Kaul, and Tushar Krishna. 2020. SIGMA: A Sparse and Irregular GEMM Accelerator with Flexible Interconnects for DNN Training. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA).* 58–70. https://doi.org/10.1109/HPCA47549.2020.00015

Jonathan Ragan-Kelley, Andrew Adams, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Frédo Durand. 2012. Decoupling Algorithms from Schedules for Easy Optimization of Image Processing Pipelines. *ACM Trans. Graph.* 31, 4, Article 32 (July 2012), 12 pages. https://doi.org/10.1145/2185520.2185528

Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13).* Association for Computing Machinery, New York, NY, USA, 519–530. https://doi.org/10.1145/2491956.2462176

Alexander Rucker, Matthew Vilim, Tian Zhao, Yaqi Zhang, Raghu Prabhakar, and Kunle Olukotun. 2021. Capstan: A Vector RDA for Sparsity. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture* (Virtual Event, Greece) *(MICRO '21).* Association for Computing Machinery, New York, NY, USA, 1022–1035. https://doi.org/10.1145/3466752.3480047

Ryan Senanayake, Changwan Hong, Ziheng Wang, Amalee Wilson, Stephen Chou, Shoaib Kamil, Saman Amarasinghe, and Fredrik Kjølstad. 2020. A Sparse Iteration Space Transformation Framework for Sparse Tensor Algebra. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 158 (Nov. 2020), 30 pages. https://doi.org/10.1145/3428226

Navjot Singh, Zecheng Zhang, Xiaoxiao Wu, Naijing Zhang, Siyuan Zhang, and Edgar Solomonik. 2022. Distributed-memory tensor completion for generalized loss functions in python using new sparse tensor kernels. *J. Parallel Distributed Comput.* 169 (2022), 269–285. https://doi.org/10.1016/j.jpdc.2022.07.005

Edgar Solomonik and Torsten Hoefler. 2015. Sparse Tensor Algebra as a Parallel Programming Model. arXiv:1512.00066 [cs.MS]

Edgar Solomonik, Devin Matthews, Jeff R. Hammond, John F. Stanton, and James Demmel. 2014. A massively parallel tensor contraction framework for coupled-cluster computations. *J. Parallel and Distrib. Comput.* 74, 12 (2014), 3176–3190. https://doi.org/10.1016/j.jpdc.2014.06.002

Nitish Srivastava, Hanchen Jin, Jie Liu, David Albonesi, and Zhiru Zhang. 2020a. MatRaptor: A Sparse-Sparse Matrix Multiplication Accelerator Based on Row-Wise Product. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO).* 766–780. https://doi.org/10.1109/MICRO50266.2020.00068

Nitish Srivastava, Hanchen Jin, Shaden Smith, Hongbo Rong, David Albonesi, and Zhiru Zhang. 2020b. Tensaurus: A Versatile Accelerator for Mixed Sparse-Dense Tensor Computations. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA).* 689–702. https://doi.org/10.1109/HPCA47549.2020.00062

Nigel Stephens, Stuart Biles, Matthias Boettcher, Jacob Eapen, Mbou Eyole, Giacomo Gabrielli, Matt Horsnell, Grigorios Magklis, Alejandro Martinez, Nathanaël Prémillieu, Alastair Reid, Alejandro Rico, and Paul Walker. 2018. The ARM Scalable Vector Extension. *CoRR* abs/1803.06185 (2018). arXiv:1803.06185 http://arxiv.org/abs/1803.06185

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

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

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

Anand Venkat, Mary Hall, and Michelle Strout. 2015. Loop and Data Transformations for Sparse Matrix Code. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI '15)*. Association for Computing Machinery, New York, NY, USA, 521–532. https://doi.org/10.1145/2737924.2738003

R. Clint Whaley and Antoine Petitet. 2005. Minimizing development and maintenance costs in supporting persistently optimized BLAS. *Software: Practice and Experience* 35, 2 (2005), 101–121. https://doi.org/10.1002/spe.626 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.626

Rohan Yadav, Alex Aiken, and Fredrik Kjolstad. 2022. DISTAL: The Distributed Tensor Algebra Compiler. (2022), 286–300. https://doi.org/10.1145/3519939.3523437

Zihao Ye, Ruihang Lai, Junru Shao, Tianqi Chen, and Luis Ceze. 2023. SparseTIR: Composable Abstractions for Sparse Compilation in Deep Learning. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 660–678. https://doi.org/10.1145/3582016.3582047

Qing Yi. 2012. POET: A Scripting Language for Applying Parameterized Source-to-Source Program Transformations. *Softw. Pract. Exper.* 42, 6 (jun 2012), 675–706. https://doi.org/10.1002/spe.1089

Guowei Zhang, Nithya Attaluri, Joel S. Emer, and Daniel Sanchez. 2021a. Gamma: Leveraging Gustavson's Algorithm to Accelerate Sparse Matrix Multiplication. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 687–701. https://doi.org/10.1145/3445814.3446702

Yaqi Zhang, Alexander Rucker, Matthew Vilim, Raghu Prabhakar, William Hwang, and Kunle Olukotun. 2019. Scalable Interconnects for Reconfigurable Spatial Architectures. In *Proceedings of the 46th International Symposium on Computer Architecture* (Phoenix, Arizona) *(ISCA '19)*. Association for Computing Machinery, New York, NY, USA, 615–628. https://doi.org/10.1145/3307650.3322249

Yaqi Zhang, Nathan Zhang, Tian Zhao, Matt Vilim, Muhammad Shahbaz, and Kunle Olukotun. 2021b. SARA: Scaling a Reconfigurable Dataflow Accelerator. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. 1041–1054. https://doi.org/10.1109/ISCA52012.2021.00085

Tuowen Zhao, Tobi Popoola, Mary Hall, Catherine Olschanowsky, and Michelle Strout. 2022. Polyhedral Specification and Code Generation of Sparse Tensor Contraction with Co-Iteration. *ACM Trans. Archit. Code Optim.* 20, 1, Article 16 (dec 2022), 26 pages. https://doi.org/10.1145/3566054

Size Zheng, Renze Chen, Anjiang Wei, Yicheng Jin, Qin Han, Liqiang Lu, Bingyang Wu, Xiuhong Li, Shengen Yan, and Yun Liang. 2022. AMOS: Enabling <U>A</U>Utomatic <U>M</U>Apping for Tensor Computations <U>O</U>n <U>S</U>Patial Accelerators with Hardware Abstraction. In *Proceedings of the 49th Annual International Symposium on Computer Architecture* (New York, New York) *(ISCA '22)*. Association for Computing Machinery, New York, NY, USA, 874–887. https://doi.org/10.1145/3470496.3527440

Received 2022-11-10; accepted 2023-03-31