Mosaic: An Interoperable Compiler for Tensor Algebra

Manya Bansal, Olivia Hsu, Kunle Olukotun, Fredrik Kjolstad
Sparse Tensor Algebra is important to many domains

- Data Analytics
- Scientific Computing
- Machine Learning
Explosion in number of high-performance systems
<table>
<thead>
<tr>
<th>System Type</th>
<th>Sparse</th>
<th>Tensor Properties</th>
<th>Platform</th>
</tr>
</thead>
<tbody>
<tr>
<td>Library</td>
<td>✗</td>
<td>✓</td>
<td>CPU</td>
</tr>
<tr>
<td>Library</td>
<td>✓</td>
<td>✗</td>
<td>GPU</td>
</tr>
<tr>
<td>Compiler</td>
<td>✓</td>
<td>✗</td>
<td>Capstan</td>
</tr>
</tbody>
</table>

**Stardust: Compiling Sparse Tensor Algebra to a Reconfigurable Dataflow Architecture**

Olivia Hsu  
Stanford University  
Stanford, CA, USA  
oshu@stanford.edu

Alexander Rucker  
Stanford University  
Stanford, CA, USA  
arucker@stanford.edu

Tian Zhao  
Stanford University  
Stanford, CA, USA  
tzhao@stanford.edu

Kunle Olukotun  
Stanford University  
Stanford, CA, USA  
kunle@stanford.edu

Frederik Kahlele  
Stanford University  
Stanford, CA, USA  
kahlele@stanford.edu
Many ways of computing the same thing

\[ A = B \odot (CD) \]

(SDDMM)
Many ways of computing the same thing

The Tensor Algebra Compiler
FREDRIK KJOLSTAD, Massachusetts Institute of Technology, USA
SHOAIB KAMIL, Adobe Research, USA
STEPHEN CHOU, Massachusetts Institute of Technology, USA
DAVID LUCATO, French Alternative Energies and Atomic Energy Commission, France
SAMAN AMARASINGHE, Massachusetts Institute of Technology, USA

Stardust: Compiling Sparse Tensor Algebra to a Reconfigurable Dataflow Architecture
Olivia Elia
Stanford University
sevilia@stanford.edu
Alexander Badler
Stanford University
abadder@stanford.edu
Tian Zhao
Stanford University
htian@stanford.edu
Kris Okamoto
Stanford University
kro@stanford.edu
Fredrik Kjolstad
Stanford University
fkjolstad@stanford.edu

\[ A = B \circ (C \cdot D) \]
(SDDMM)
Many ways of computing the same thing

The Tensor Algebra Compiler
FREDRIK KJOLSTAD, Massachusetts Institute of Technology, USA
SJOAB KAMIL, Adobe Research, USA
STEPHEN CHOU, Massachusetts Institute of Technology, USA
DAVID LUCATO, French Alternative Energies and Atomic Energy Commission, France
SAMAN AMARASINGHE, Massachusetts Institute of Technology, USA

Stardust: Compiling Sparse Tensor Algebra to a Reconfigurable Dataflow Architecture
Olivia Elia
New York University
New York, NY
elie@nyu.edu
Alexander Badler
New York University
New York, NY
abadler@nyu.edu
Tao Zhao
New York University
New York, NY
taozhao@nyu.edu
Kenda Okamoto
New York University
New York, NY
kendo@nyu.edu
Frederik Kjolstad
Stanford University
Stanford, CA, USA
kjl@stanford.edu

$A = B \odot (CD)$
(SDDMM)
Many ways of computing the same thing

\[ A = B \odot (CD) \]

(SDDMM)
Many ways of computing the same thing

\[ A = B \circ (CD) \]

\((\text{SDDMM})\)
Many ways of computing the same thing

\[ A = B \odot (CD) \]

(SDDMM)
SDDMM Performance

\[ A = B \odot (CD) \]

![Graph showing runtime (ms) against nonzeros (%) for different libraries and operations.]

- Intel MKL Dot
- Intel MKL GEMM
- TACL SDDMM
- CBLAS GEMM
- TBLIS GEMM
- GSL GEMV
SDDMM Performance

\[ A = B \circ (CD) \]

![Graph showing performance of SDDMM with varying nonzeros and matrix dimensions. The graph compares different libraries like Intel MKL Dot, Intel MKL GEMM, TACO SDDMM, CBLAS GEMM, TBLIS GEMM, and GSL GEMV. The y-axis represents runtime in milliseconds, and the x-axis shows nonzeros as a percentage.](image-url)
SDDMM Performance

\[ A = B \circ (CD) \]
SDDMM Performance

\[ A = B \odot (CD) \]
SDDMM Performance

\[ A = B \odot (CD) \]

Matrix Dimension (n)

<table>
<thead>
<tr>
<th>Nonzeros (%)</th>
<th>Runtime (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel MKL Dot</td>
<td>10</td>
</tr>
<tr>
<td>Intel MKL GEMM</td>
<td>7</td>
</tr>
<tr>
<td>TACO SDDMM</td>
<td>8</td>
</tr>
<tr>
<td>CBLAS GEMM</td>
<td>10</td>
</tr>
<tr>
<td>Intel MKL GEMV</td>
<td>2.5</td>
</tr>
<tr>
<td>TBLIS GEMM</td>
<td>5</td>
</tr>
<tr>
<td>GSL GEMV</td>
<td>10</td>
</tr>
<tr>
<td>CBLAS GEMV</td>
<td>1.25</td>
</tr>
</tbody>
</table>

Sparse

Runtime (ms)

Nonzeros (%)
Levering both fusion and hand-optimized library calls is crucial for achieving good performance
But, adapting old code to utilize new functions is hard.

\[
A = B \odot (CD)
\]

Sparse

TACO (Compiler)

\[
\begin{align*}
\text{cblas\_?gemm} & \quad (\text{CBLAS}) \\
\text{tblis\_tensor\_mult} & \quad (\text{TBLIS}) \\
\text{tblis\_matrix\_mult} & \quad (\text{TBLIS}) \\
\text{stardust\_GEMM} & \quad (\text{Compiler}) \\
\text{cblas\_?dot} & \quad (\text{CBLAS}) \\
\text{tblis\_vector\_dot} & \quad (\text{TBLIS}) \\
\text{tblis\_matrix\_mult} & \quad (\text{TBLIS}) \\
\text{AVX Intrinsics} & \quad (\text{AVX}) \\
\text{stardust\_sp\_elem\_mul} & \quad (\text{Compiler})
\end{align*}
\]
Our Goal

MKL

CuSPARSE

GSL

Stardust

TBLIS

AVX

BLAS
Our Goal
Overview of Mosaic

Separation of Responsibilities!

Applications Engineer

External Library Engineer

Automatic Search

Scheduling Language
- Binding to Functions
- Compound Scheduling Commands

Validate Bindings

Performance Engineer

External Function Interface

Concrete Index Notation

Low-Level IR
- External Function Code

Imperative Code

TACO

Our Contributions
Overview of Mosaic

Separation of Responsibilities!

Automatic Search

External Function Interface

Scheduling Language
- Binding to Functions
- Compound Scheduling Commands

Performance Engineer

Validate Bindings

TACO

Concrete Index Notation

Low-Level IR
- External Function Code

Imperative Code

Applications Engineer

Our Contributions

External Library Engineer
External Function Interface

*What* does the function compute?

*How* to target the function?
What does the function compute?

Dense Linear Algebra

Space of all Tensor Algebra Expressions

Dense Tensor Contractions

Sparse Tensor Algebra

Capstan

CPU

CPU

TBLIS is a library and framework for performing tensor operations, especially tensor contraction, using efficient native algorithms.

5 Contributors
What does the function compute?

<table>
<thead>
<tr>
<th>Capability Language</th>
<th>Tensor Properties</th>
<th>Checker Function</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
First Attempt: Reuse Application Language

X(i) = X(i) + Y(i)
X(i, k) = Y(i, j) + Y(j, k)

+ Constraint Language
>, <, ==, !=

A(i) = C(i) + B(i)
A(i, i) = C(i, i) + B(i, i)
A(i, i, k) = C(i, i, i) + B(i, i, i)

...
Capability Language

\[ A(i) = C(i) + B(i) \]
\[ A(i, i) = C(i, i) + B(i, i) \]
\[ A(i, i, k) = C(i, i, i) + B(i, i, i) \]
\[ \cdots \]

\[ X(I) = X(I) + Y(I) \]
\[ \uparrow \]
Dynamic order

Adding Constraints

∀ ∈
What does the function compute?

- Capability
- Language
- Tensor Properties
- Checker Function

What does the function accept?
What does the function compute?

Checker Function

Any C++ Function

Capability Language
Tensor Properties
What does the function compute?
External Function Interface

What does the function compute?

How to target the function?
How to target the function?

1. Setup  2. Function Call  3. Teardown
How to target the function?

```c
tblis_tensor var1;
tblis_tensor var2;
tblis_tensor result;

tblis_init_tensor_s_helper_row_major(&var1, D->dimensions, 2, C_vals);
tblis_init_tensor_s_helper_row_major(&var2, D->dimensions, 2, D_vals);
tblis_init_tensor_s_helper_row_major(&result, D->dimensions, 2, A2463);

tblis_tensor_mult(NULL, NULL, &var1, "ij", &var2, "jk", &result, "ik");

free_tblris_tensor(&var1);
free_tblris_tensor(&var2);
free_tblris_tensor(&result);
```

Sample Function Call (TBLIS Library)
How to target the function?

tblis_tensor var1;
tblis_tensor var2;
tblis_tensor result;

tblis_init_tensor_s_helper_row_major(&var1, D->dimensions, 2, C_vals);
tblis_init_tensor_s_helper_row_major(&var2, D->dimensions, 2, D_vals);
tblis_init_tensor_s_helper_row_major(&result, D->dimensions, 2, A2463);

tblis_tensor_mult(NULL, NULL, &var1, "ij", &var2, "jk", &result, "ik");

free_tblis_tensor(&var1);
free_tblis_tensor(&var2);
free_tblis_tensor(&result);
How to target the function?

tblis_tensor var1;
tblis_tensor var2;
tblis_tensor result;

tblis_init_tensor_s_helper_row_major(&var1, D->dimensions, 2, C_vals);
tblis_init_tensor_s_helper_row_major(&var2, D->dimensions, 2, D_vals);
tblis_init_tensor_s_helper_row_major(&result, D->dimensions, 2, A2463);

tblis_tensor_mult(NULL, NULL, &var1, "ij", &var2, "jk", &result, "ik");

free_tblis_tensor(&var1);
free_tblis_tensor(&var2);
free_tblis_tensor(&result);
External Function Interface

*What does the function compute?*

*How to target the function?*
Overview of Mosaic

Separation of Responsibilities!

Automatic Search

Scheduling Language
- Binding to Functions
- Compound Scheduling Commands

Validate Bindings

Performance Engineer

External Function Interface

TACO
- Concrete Index Notation
- Low-Level IR
  - External Function Code

Imperative Code

Applications Engineer

External Library Engineer

Our Contribution
Overview of Mosaic

Separation of Responsibilities!

Automatic Search
Scheduling Language
- Binding to Functions
- Compound Scheduling Commands

External Function Interface
Concrete Index Notation
- Low-Level IR
  - External Function Code

TACO
Imperative Code

Performance Engineer

External Library Engineer

Applications Engineer

Our Contribution
Example: AVXDot for SDDMM

\[ A = B \odot (CD) \]
Scheduling command describe function placement

```
// SDDMM in einsum notation.
stmt = A(i,j) = B(i,j)*C(i,k)*D(k,j)
```

**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

= · :

is sampled using the sparse matrix . 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.

```
// SDDMM in einsum notation.
stmt = A(i,j) = B(i,j)*C(i,k)*D(k,j)
```

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.

```
register(AVXAdd());
vec<stmts> schedules = stmt.getAllSchedules();
A.compile(schedules[i])
```

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

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 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 the 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 the 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).

**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.
Scheduling command describe function placement

```
1 // SDDMM in einsum notation.
2 stmt = A(i, j) = B(i, j) * C(i, k) * D(k, j)
3 // Precompute C*D in W and use
4 // iw, jw, kw as index vars in the code.
5 stmt . precompute (C*D, W, {i, j, k},
6                        {iw, jw, kw})
```

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

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

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

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 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 the 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 the 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).
Scheduling command describe function placement

```plaintext
// 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)
```

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

\[ \Gamma_{89} = \mathcal{\mathcal{\Gamma}}_{89} \times \mathcal{\mathcal{\Gamma}}_{89} \times \mathcal{\mathcal{\Gamma}}_{89}, \]

i.e., the dense multiplication of tensors \(\mathcal{\mathcal{\Gamma}}_{89}\) and \(\mathcal{\mathcal{\Gamma}}_{89}\) is sampled using the sparse matrix \(\mathcal{\mathcal{\Gamma}}_{89}\). 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.

```plaintext
1 // SDDMM in einsum notation.
2 stmt = A(i,j) = B(i,j) * C(i,k) * D(k,j)
3 // Precompute C*D in W and use
4 // iw,jw,kw as index vars in the code.
5 stmt.precompute(C*D, W, {i, j, k},
                  {iw, jw, kw})
6 // Split loop kw by 4 into ki and ko.
7 .split(kw, ko, ki, 4)
```

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

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

Fig. 4. Automatically search and find all schedules that use AVX vector add.
Scheduling command describe function placement

```c
// SDDMM in einsum notation.
stmt=A(i,j)=B(i,j)\times C(i,k)\times 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)
```
Scheduling command describe function placement

// 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)
Scheduling command describe function placement

```plaintext
// 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)
```
**Fuzzy scheduling boosts productivity.**

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

```java
1 // SDDMM in einsum notation.
2 stmt=A(i,j)=B(i,j)*C(i,k)*D(k,j)
3 // Precompute C*D in W and use
4 // iw, jw, kw as index vars in the code.
5 stmt.precompute(C*D,W,{i,j,k},
6     {iw,jw,kw})
7 // Split loop kw by 4 into ki and ko.
8     .split(kw,ko,ki,4)
9 // Push ki to be the inner most loop.
10     .reorder(iw, jw, ko, ki)
11 // Consider iw, jw to be constant.
12     .fix(iw,jw)
13 // Bind the reduction of kw to AVXAdd().
14     .bind(AVXAdd(), C*D)
```
Automatic Search for Design Space Exploration

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

\[ \hat{\mathbf{V}} = \mathbf{A} \cdot \mathbf{B} \cdot \mathbf{C} \cdot \mathbf{D}, \]

i.e., the dense multiplication of tensors \( \mathbf{A} \) and \( \mathbf{B} \) is sampled using the sparse matrix \( \mathbf{C} \cdot \mathbf{D} \). 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.

```c++
// 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.
stmt.split(kw,ko,ki,4)
// Push ki to be the inner most loop.
stmt.reorder(iw, jw, ko, ki)
// Consider iw, jw to be constant.
stmt.fix(iw,jw)
// Bind the reduction of kw to AVXAdd().
stmt.bind(AVXAdd(), C*D)
```

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

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

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

```c++
// Register AVX to Mosaic.
register(AVXAdd());
vec<stmts> schedules = stmt.getAllSchedules();
// Pick a schedule to apply.
A.compile(schedules[i])
```

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

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 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 the 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 the 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).

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.
Automatic Search for Design Space Exploration

Example Input Program

```cpp
// Tensor<int> Addition
A(i, j) = B(i, j) + 5
A.register(AvxIntAdd(),
          CblasIntMul(),
          TblisIntMul(),
          GslFloatAdd())
```

Step 1: Filter External Functions

- AVX
- CBLAS
- TBLIS
- GSL
- FloatAdd()

Step 2: Operator Pattern Matching

Match to AVX!

```
AvxIntAdd() capability description:
2k = x_k + y_k
where k == 4
```

Step 3: Tensor Index-Variable Matching

```
A_{ij} = B_{ij} + 5
```

Order-Reduce B    Broadcast 5

Since AvxIntAdd can only compute on vectors

Step 4: Tiling Validation

Check for legal mappings

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

Tile

Step 5: External Function Call

```
// AvxIntAdd()
_mm256_add_ps(op1, op2)
```

Order-Reduced Tiled B  Broadcasted 5
We perform a lines of code (LOC) study on the external function abstractions in Mosaic. The LOC (TTTP), introduced in Section Table 3. A subset of the external functions plugged into Mosaic for experiments described in Section Vol. 1, No. 1, Article . Publication date: November 2022.

That loops through all valid schedules, sorting them by application run time. The user may either pick one (or a few) to run or employ a naive autotuner that Mosaic is allowed to explore. For this experiment, the depth was set to choices. To limit the number of choices, users can set a maximum depth of mathematical rewrites one of the 3 registered function, or we leave it unscheduled. Because of this, we get involved in the TTTP computation. Each pair of matrices has 4 possible options: either we bind it to these functions. There are ber of mappings results from the number of ways in which Mosaic can choose two dense matrices functions are registered into Mosaic. The large num-

ber of mappings results from the number of ways in which Mosaic can choose two dense matrices one dot product and two matrix-multiplication search algorithm is able to ing the automatic search machinery with no user-

functions (38) we were able to plug in to Mosaic in a limited amount of time. And is now usable by all Mosaic users. Table 8.5 External Function Abstraction Study and is now usable by all Mosaic users. Table 8.5 External Function Abstraction Study

8.6 Evaluation of the Search Finally, we look at the performance of Mosaic when we compute an SpMMAdd. Similar to results

Functions that compute two or more expressions are underlined a

Table 22. The time it takes to find valid tilings for

<table>
<thead>
<tr>
<th>Name</th>
<th>Expression</th>
<th>BLAS</th>
<th>GSL</th>
<th>TBLIS</th>
<th>AVX</th>
<th>Stardust</th>
<th>cuSPARSE</th>
<th>MKL</th>
</tr>
</thead>
<tbody>
<tr>
<td>VecAdd</td>
<td>( A_i = B_i + C_i )</td>
<td>10</td>
<td>15</td>
<td>16</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Saxpy</td>
<td>( A_i = B_i + y D_i )</td>
<td>11</td>
<td>20</td>
<td>19</td>
<td>15</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dot</td>
<td>( \alpha = B_i * C_i )</td>
<td>12</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GEMV</td>
<td>( A_i = B_{ij} * C_j )</td>
<td>11</td>
<td>20</td>
<td>30</td>
<td>43</td>
<td>21</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SGEMV</td>
<td>( A_i = \alpha * B_{ij} * C_j + \beta * D_i )</td>
<td>11</td>
<td>20</td>
<td>30</td>
<td>43</td>
<td>21</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SpMV</td>
<td>( A_i = \alpha * B_{ij} * C_j + \beta * D_i )</td>
<td>12</td>
<td>19</td>
<td>19</td>
<td>30</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SpMMAdd</td>
<td>( A_{ik} = B_{ij} + C_{ij} )</td>
<td>12</td>
<td>19</td>
<td>19</td>
<td>30</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GEMM</td>
<td>( A_{ik} = B_{ij} * C_{jk} )</td>
<td>12</td>
<td>19</td>
<td>19</td>
<td>30</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SGEMM</td>
<td>( A_{ik} = \alpha * B_{ij} * C_{jk} + \beta * C_{ik} )</td>
<td>12</td>
<td>19</td>
<td>19</td>
<td>30</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TTM</td>
<td>( A_{ijk} = B_{ijl} * C_{kl} )</td>
<td>18</td>
<td>30</td>
<td>18</td>
<td>30</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Plus3</td>
<td>( A_{ijk} = B_{ijl} + C_{ijk} )</td>
<td>18</td>
<td>30</td>
<td>18</td>
<td>30</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Evaluation

<table>
<thead>
<tr>
<th>Name</th>
<th>Expression</th>
<th>BLAS</th>
<th>GSL</th>
<th>TBLIS</th>
<th>AVX</th>
<th>Stardust</th>
<th>cuSPARSE</th>
<th>MKL</th>
</tr>
</thead>
<tbody>
<tr>
<td>VecAdd</td>
<td>$A_i = B_i + C_i$</td>
<td>16</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Saxpy</td>
<td>$A_i = B_i + yD_i$</td>
<td>10</td>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dot</td>
<td>$\alpha = B_i * C_i$</td>
<td>12</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GEMV</td>
<td>$A_i = B_{ij} * C_j$</td>
<td>11</td>
<td>20</td>
<td>19</td>
<td></td>
<td></td>
<td></td>
<td>15</td>
</tr>
<tr>
<td>SGEMV</td>
<td>$A_i = \alpha * B_{ij} * C_j + \beta * D_i$</td>
<td>11</td>
<td>20</td>
<td></td>
<td></td>
<td></td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>SpMV</td>
<td>$A_i = \alpha * B_{ij} * C_j + \beta * D_i$</td>
<td></td>
<td></td>
<td>30</td>
<td></td>
<td>43</td>
<td>21</td>
<td></td>
</tr>
<tr>
<td>SpMMAdd</td>
<td>$A_{ik} = B_{ij} + C_{ij}$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>30</td>
<td>18</td>
</tr>
<tr>
<td>GEMM</td>
<td>$A_{ik} = B_{ij} * C_{jk}$</td>
<td>12</td>
<td>19</td>
<td>19</td>
<td>30</td>
<td></td>
<td></td>
<td>17</td>
</tr>
<tr>
<td>SGEMM</td>
<td>$A_{ik} = \alpha * B_{ij} * C_{jk} + \beta * C_{ik}$</td>
<td>12</td>
<td>19</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>17</td>
</tr>
<tr>
<td>TTM</td>
<td>$A_{ijk} = B_{ij} * C_{kl}$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>18</td>
<td>30</td>
</tr>
<tr>
<td>Plus3</td>
<td>$A_{ijk} = B_{ijk} + C_{ijk}$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>18</td>
<td>30</td>
</tr>
</tbody>
</table>
## Evaluation

<table>
<thead>
<tr>
<th>Name</th>
<th>Expression</th>
<th>BLAS</th>
<th>GSL</th>
<th>TBLIS</th>
<th>AVX</th>
<th>Stardust</th>
<th>cuSPARSE</th>
<th>MKL</th>
</tr>
</thead>
<tbody>
<tr>
<td>VecAdd</td>
<td>$A_i = B_i + C_i$</td>
<td>16</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Saxpy</td>
<td>$A_i = B_i + \gamma D_i$</td>
<td>10</td>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dot</td>
<td>$\alpha = B_i * C_i$</td>
<td>12</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GEMV</td>
<td>$A_i = B_{ij} * C_j$</td>
<td>11</td>
<td>20</td>
<td>19</td>
<td></td>
<td></td>
<td></td>
<td>15</td>
</tr>
<tr>
<td>SGEMV</td>
<td>$A_i = \alpha * B_{ij} * C_j + \beta * D_i$</td>
<td>11</td>
<td>20</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>30</td>
</tr>
<tr>
<td>SpMV</td>
<td>$A_i = \alpha * B_{ij} * C_j + \beta * D_i$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>43</td>
</tr>
<tr>
<td>SpMMAAdd</td>
<td>$A_{ik} = B_{ij} + C_{ij}$</td>
<td>12</td>
<td>19</td>
<td>19</td>
<td>30</td>
<td></td>
<td></td>
<td>18</td>
</tr>
<tr>
<td>GEMM</td>
<td>$A_{ik} = B_{ij} * C_{jk}$</td>
<td>12</td>
<td>19</td>
<td></td>
<td>30</td>
<td></td>
<td></td>
<td>17</td>
</tr>
<tr>
<td>SGEMM</td>
<td>$A_{ik} = \alpha * B_{ij} * C_{jk} + \beta * C_{ik}$</td>
<td>12</td>
<td>19</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>17</td>
</tr>
<tr>
<td>TTM</td>
<td>$A_{ijk} = B_{ijl} * C_{kl}$</td>
<td>18</td>
<td></td>
<td>30</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Plus3</td>
<td>$A_{ijk} = B_{ijk} + C_{ijk}$</td>
<td>18</td>
<td>30</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Average LOC: 20

Total # Interfaces: 38

- The table shows the lines of code (LOC) for different expressions in the Mosaic system across various libraries and systems.
- The expressions are categorized into different functions such as VecAdd, Saxpy, Dot, GEMV, SGEMV, SpMV, SpMMAAdd, GEMM, SGEMM, TTM, and Plus3.
- Each expression is evaluated on different systems and libraries, including BLAS, GSL, TBLIS, AVX, Stardust, cuSPARSE, and MKL.
- The LOC values range from 16 to 43, with an average LOC of 20.
- The total number of interfaces plugged into Mosaic for these expressions is 38.

### Evaluation Result

- The evaluation of the search algorithm shows how Mosaic can discover and schedule tensor-times-tensor product (TTTP) computations.
- The table provides a detailed comparison of the LOC for different expressions across various systems and libraries, demonstrating the effectiveness of Mosaic in optimizing computational tasks.

---

---

---
SDDMM Performance

\[ A = B \odot (CD) \]

Matrix Dimension (n):
- 250
- 500
- 750
- 1500
- 1750
- 1250
- 1000

Nonzeros (%):
- 10
- 6
- 10
- 5
- 10
- 3
- 4

Runtime (ms):
- Intel MKL Dot: 0.08
- Intel MKL GEMM: 0.17
- TACO SDDMM: 0.31
- CBLAS GEMM: 2.5
- Intel MKL GEMV: 5
- TBLIS GEMM: 10
- GSL GEMV: 1.25
- CBLAS GEMV: 0.63

SDDMM Performance

Sparse
Evaluation: Blocked Sparse Matrix Multiplication

\[ A = BC \]

Blocked Sparse Matrix

Runtime (ms)

Tensor Dimension (n)

Fixed % Non-Zeroes = 5%
Evaluation: Blocked Sparse Matrix Multiplication

\[ A = BC \]

Tensor Dimension (n)

Runtime (ms)

Fixed % Non-Zeroes = 5%

Blocked Sparse Matrix
Evaluation: Blocked Sparse Matrix Multiplication

$A = BC$

 Blocked Sparse Matrix

Runtime (ms)

Tensor Dimension (n)

Fixed % Non-Zeroes = 5%
Evaluation: Blocked Sparse Matrix Multiplication

\[ A = BC \]

Blocked Sparse Matrix

![Graph showing runtime vs. tensor dimension for different libraries](image)

- BLAS GEMM
- TACO BlockSparse
- Intl MKL GEMM
- GSL GEMM

Fixed % Non-Zeroes = 5%
Mosaic: An Interoperable Compiler