LEGO: A Layout Expression Language for Code Generation of Hierarchical Mapping

Amir Mohammad Tavakkoli
Kahlert School of Computing
University of Utah
Salt Lake City, USA
amir.tavakkoli@utah.edu

Cosmin E. Oancea
Dept. of Computer Science
University of Copenhagen
Copenhagen, Denmark
cosmin.oancea@di.ku.dk

Mary Hall
Kahlert School of Computing
University of Utah
Salt Lake City, USA
mhall@cs.utah.edu

***Abstract*—We describe LEGO, a new approach to optimizing data movement whereby code is expressed as a layout-independent computation and composed with layouts for data and computation. This code generator organization derives complex indexing expressions associated with hierarchical parallel code and data movement for GPUs. LEGO maps from layout specification to indexing expressions, and can be integrated into existing compilers and code templates. It facilitates the exploration of data layouts in combination with other optimizations. We demonstrate LEGO's integration with the Triton and MLIR compilers, and with CUDA templates. We show that LEGO is capable of deriving performance competitive with Triton, and shows broad applicability for data and thread layout mapping optimizations in its integration with CUDA and MLIR.**

***Index Terms*—data layout, MLIR compiler, domain-specific optimization tools**

I. Introduction

Now that Moore's Law and Dennard scaling no longer drive performance improvements, researchers have turned to architecture specialization and domain-specific programming systems for further scaling gains [1]. Data movement is now the dominant cost in execution time and energy [2], and optimizations to reduce data movement must take center stage.

Most commonly, optimizing data movement involves *reordering computation* to modify memory access order; this reordering allows the computation to exploit reuse of data in nearby *fast* memory, especially cache and registers using loop transformations such as tiling [3] and unroll-and-jam [4]. Notably, polyhedral compiler frameworks, dating back to mid 1980s [5] – and more recently Pluto [6], PPCG [7], and Polygeist [8], among others – represent a dynamic instance of a statement in a multi-dimensional loop iteration space as an integer point in the statement's polyhedron [9]. This mathematical representation facilitates the composition of complex transformation sequences as statement instance reorderings.

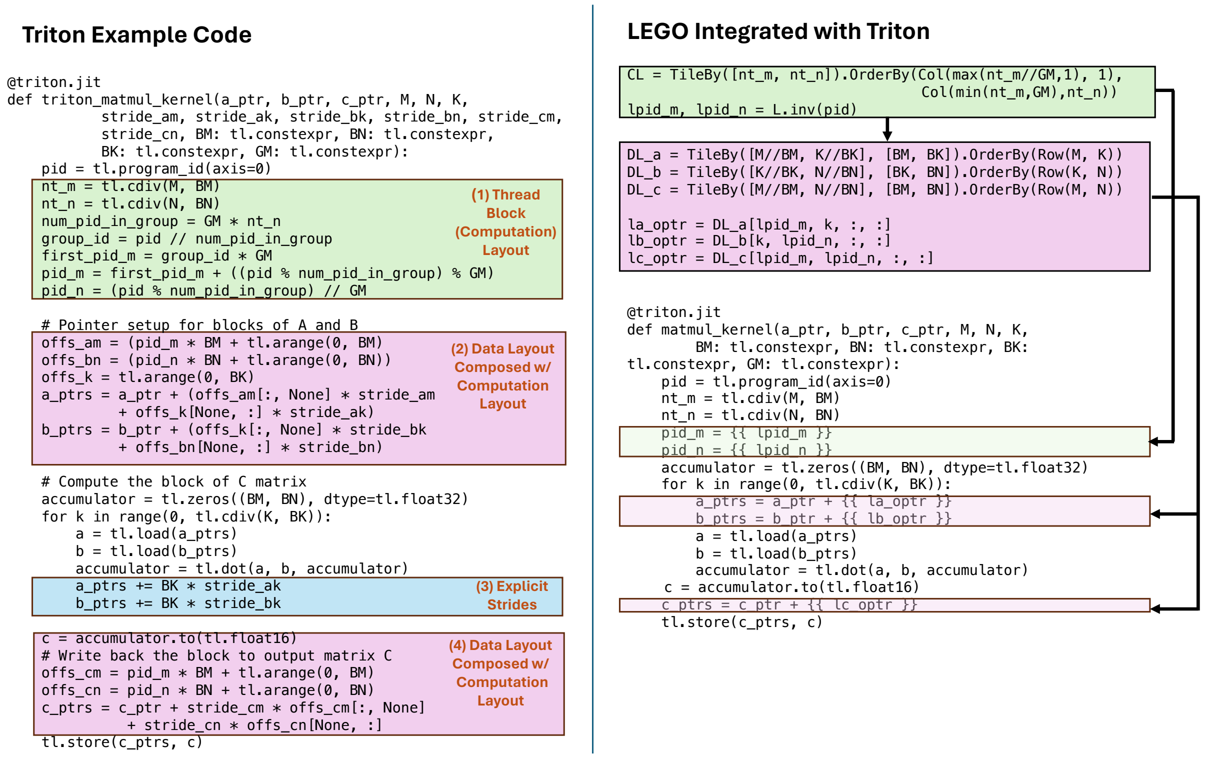
As an alternative to statement reordering, a system can *change the layout of data in memory* to more closely match the order in which the computation accesses it. For example, the standard layout for a 2-dimensional array in a C or C++ compiler is row-major order, whereby adjacent elements in a row are stored contiguously in memory, and elements in the same column are strided by the length of the row. But improved spatial reuse and reduced data movement has been demonstrated by alternatives to row-major order [10, 11, 12, 13, 14]. Further, controlling data layout helps achieve performance portability across architectures, matching layout to size and bandwidth of each architecture's memory hierarchy [15, 16, 17, 18, 19].

Recently, a body of work targeting vector and matrix processors in GPUs uses *both data and computation layout* as well as data movement specification to decompose computations and order data to match the inputs and outputs of these accelerator units, e.g., Fireiron [20], CuTe [21], Graphene [22] and Triton [23, 24]. Such systems *restrict* indexing expressions to encode *linear* formulas that are represented in terms of *strides* or *binary matrices*, which makes specification tedious and error-prone. Despite the availability of implementations of some of these, integration into other tools and generalization for other architectural features remain limited.

This paper presents LEGO, a layout abstraction that increases generality and facilitates adoption of this essential capability by other frameworks. LEGO can express any bijective mapping between the logical and reordered index space that are represented as *permutations*, thus omitting strides. Permutations can be *linear*, e.g., of (entire) dimensions, or *irregular*, represented by user-defined functions. LEGO's lack of explicit strides eliminates low-level index calculations, making code simpler and more expressive than frameworks like Triton. Its high-level building blocks reduce both mathematical complexity and overall code size, allowing users to modify computations simply by changing layouts without altering core logic. Once the layouts and connections are defined, LEGO automatically generates a bijective mapping for the whole ensemble, thus serving both as a high-level programming abstraction and a tool for high-performance code generation. LEGO's algebra is also extended beyond this bijective restriction to support partial tiles and common injective mappings.

This paper makes the following contributions:

- a general abstraction for bijective layouts, which can express both computation and data,
- an implementation reproducible from the paper and accessible to other frameworks via open-source software,
- a demonstration of lowering to Triton, CUDA, and MLIR supporting irregular layouts such as an anti-diagonal,


Triton Example Code
(1) Thread Block (Computation) Layout
(2) Data Layout Composed w/ Computation Layout
(3) Explicit Strides
(4) Data Layout Composed w/ Computation Layout
LEGO Integrated with Triton

```
@triton.jit
def triton_matmul_kernel(a_ptr, b_ptr, c_ptr, M, N, K,
           stride_am, stride_ak, stride_bk, stride_bn, stride_cm,
           stride_cn, BM: tl.constexpr, BN: tl.constexpr,
           BK: tl.constexpr, GM: tl.constexpr):
    pid = tl.program_id(axis=0)
    nt_m = tl.cdiv(M, BM)
    nt_n = tl.cdiv(N, BN)
    num_pid_in_group = GM * nt_n
    group_id = pid // num_pid_in_group
    first_pid_m = group_id * GM
    pid_m = first_pid_m + ((pid % num_pid_in_group) % GM)
    pid_n = (pid % num_pid_in_group) // GM

    # Pointer setup for blocks of A and B
    offs_am = (pid_m * BM + tl.arange(0, BM)
    offs_bn = (pid_n * BN + tl.arange(0, BN))
    offs_k = tl.arange(0, BK)
    a_ptrs = a_ptr + (offs_am[:, None] * stride_am
             + offs_k[None, :] * stride_ak)
    b_ptrs = b_ptr + (offs_k[:, None] * stride_bk
             + offs_bn[None, :] * stride_bn)

    # Compute the block of C matrix
    accumulator = tl.zeros((BM, BN), dtype=tl.float32)
    for k in range(0, tl.cdiv(K, BK)):
        a = tl.load(a_ptrs)
        b = tl.load(b_ptrs)
        accumulator = tl.dot(a, b, accumulator)
        a_ptrs += BK * stride_ak
        b_ptrs += BK * stride_bk

    c = accumulator.to(tl.float16)
    # Write back the block to output matrix C
    offs_cm = pid_m * BM + tl.arange(0, BM)
    offs_cn = pid_n * BN + tl.arange(0, BN)
    c_ptrs = c_ptr + stride_cm * offs_cm[:, None]
             + stride_cn * offs_cn[None, :]
    tl.store(c_ptrs, c)
```

```
CL = TileBy([nt_m, nt_n]).OrderBy(Col(max(nt_m//GM,1), 1),
                                  Col(min(nt_m,GM),nt_n))
lpid_m, lpid_n = L.inv(pid)

DL_a = TileBy([M//BM, K//BK], [BM, BK]).OrderBy(Row(M, K))
DL_b = TileBy([K//BK, N//BN], [BK, BN]).OrderBy(Row(K, N))
DL_c = TileBy([M//BM, N//BN], [BM, BN]).OrderBy(Row(M, N))

la_optr = DL_a[lpid_m, k, :, :]
lb_optr = DL_b[k, lpid_n, :, :]
lc_optr = DL_c[lpid_m, lpid_n, :, :]


@triton.jit
def matmul_kernel(a_ptr, b_ptr, c_ptr, M, N, K,
        BM: tl.constexpr, BN: tl.constexpr, BK:
tl.constexpr, GM: tl.constexpr):
    pid = tl.program_id(axis=0)
    nt_m = tl.cdiv(M, BM)
    nt_n = tl.cdiv(N, BN)
    pid_m = {{ lpid_m }}
    pid_n = {{ lpid_n }}
    accumulator = tl.zeros((BM, BN), dtype=tl.float32)
    for k in range(0, tl.cdiv(K, BK)):
        a_ptrs = a_ptr + {{ la_optr }}
        b_ptrs = b_ptr + {{ lb_optr }}
        a = tl.load(a_ptrs)
        b = tl.load(b_ptrs)
        accumulator = tl.dot(a, b, accumulator)
     c = accumulator.to(tl.float16)
    c_ptrs = c_ptr + {{ lc_optr }}
    tl.store(c_ptrs, c)
```

Fig. 1: Matrix multiplication expressed in Triton (left) and using LEGO to instantiate Triton (right). The latter version describes layouts for thread block and data at a high level; LEGO automatically derives the complex indexing expressions. The resulting code generated by LEGO is shown in Figure 10.

- a demonstration of ease of use and performance competitive with state of the art.

Layout algebras such as LEGO are an important component of emerging compiler frameworks with data/thread tile abstractions, and can facilitate layout exploration. To fully exploit data/thread tiles requires integration with statement reordering optimizations such as the previously-described polyhedral compilers – a substantial implementation effort. In this paper, we demonstrate LEGO's use as a frontend tool to generate optimized indexing expressions automatically from data and thread layout specifications, and rely on backend compiler frameworks to perform loop iteration space reorderings.

II. Motivation and Overview

In this section, we discuss how layout descriptions are being used in existing tools to simplify the development of high-performance GPU implementations of matrix multiply, using a tiled, hierarchical approach to achieve data locality in registers and shared memory, and leveraging matrix processors. We motivate LEGO's generalization of the specification of the layout and automatic generation of the indexing expressions.

Matrix Multiplication Using Triton: High-performance implementations of matrix multiply for GPUs can be achieved with Triton [23] programs, as shown in Figure 1 (left). The Triton program calculates the memory offsets for the input and output matrices, loads the necessary elements from A and B, and subsequently performs the dot product to compute the matrix multiplication, storing the result in C. In the example, the code describes 2-D tiles to reuse inputs and produce output tiles. The compiler detects these tiles and optimizes the `load` and `store` operators to move data through the GPU memory hierarchy and generates the code for the `dot` operator. As a result of the compiler's careful management of the memory hierarchy and mapping to Tensor Cores, this implementation demonstrates competitive performance with the cuBLAS library.

To derive the Triton program in Figure 1 (left) nevertheless requires the programmer to write complex code, enclosed in colored boxes, to express computation and data layout specifications, specifically: (1) the thread-block-level computation layout (green); (2) the layout in global memory of the input matrices A, and B and hints at their 2D tiles, composed with the computation layout (pink); (3) an explicit stride for A, and B (blue); and, (4) the layout in global memory of output matrix C (pink). In particular, the thread block layout in the green box is non-standard and has been found to perform better than a 2D row-major order. At the inner level, program IDs are grouped with a group size of GM, while the outer level defines the overall ordering of these groups, with both levels using a column-major order. Not only is a significant portion of the code dedicated to complex index calculations, but the implementation is tightly coupled to the program instance layouts, fixed K iteration space layout, and the data layout of matrices A, B, and C.

Matrix Multiplication Using Graphene: Graphene [22] is an intermediate representation for specifying data layout and data movement using the shape algebra of CuTe [21], a part of NVIDIA's CUTLASS library. Graphene improves upon the interface for Triton by: (1) supporting more general data layouts of strided rectangular regions, as specified using a shape algebra; and (2) generating the complex index expressions automatically through a mapping from the shape algebra. A performance engineer writes a template in the Graphene IR that expresses data and thread layouts, which is instantiated by the Graphene compiler. A simple Graphene template for Matrix Multiplication takes 22 lines of specification (see Figure 8 in reference [22]).

LEGO Improvements: Like Graphene, LEGO derives index expressions from layout specifications, freeing the programmer from providing these low-level details. As compared to Graphene, LEGO eliminates explicit stride specifications in the layout definition (section III-C) and extends support to any bijective mapping from multidimensional coordinates to contiguous linear space, an aspect not supported by previous work. In Figure 1 (right), at the top, we show LEGO specification for thread block layout (2D column major order shown in green box). Next, the data tiles and their composition with the thread-block layout in row major for the input and output matrices are specified (pink box). The Triton kernel is now much simpler; LEGO instantiates the thread-block code and computes data addresses, reducing the number of arithmetic operations the user must specify from 31 to just 9.

Moreover, LEGO is a building block for compilers or code generators that applies to computations beyond tensors or specific tensor cores. This point is demonstrated with CUDA code, the Triton and MLIR compilers (sections IV and V).

III. LEGO Specification

The discussion is organized as follows: section III-A presents in an intuitive fashion how the LEGO pieces are composed, section III-B shows the LEGO grammar and uses it to define the semantics of a LEGO ensemble from the semantics of individual pieces, and section III-C compares the expression of CuTe/Graphene and LEGO layouts.

A. LEGO by Example & Mathematical Intuition

LEGO elevates data layout to a first-class design consideration. The user defines a logical view of the index space together with reordering transformations, which can be (de)composed hierarchically and chained horizontally.

Figure 2 demonstrates a simple use case that defines a LEGO layout for a flat buffer consisting of $N = 24$ elements. The user specifies the logical view of a multi-dimensional array as part of expressing the target algorithm. This is shown in the left column as an array A of shape 6×4, whose elements correspond to the flat index space of the logical view, e.g., $A[4, 1] = 4 \cdot 4 + 1 = 17$.

Next, the user would like to reorder in memory the elements of the logical view. The first step in this process is to define a hierarchy of some q levels of tiles, each of the same dimensionality d:

$$N = (n_1^1 \times \ldots \times n_d^1) \times \ldots \times (n_1^q \times \ldots \times n_d^q) \qquad (1)$$

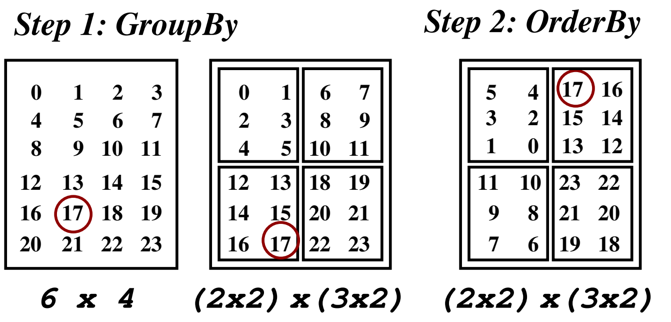


Fig. 2: Logical view – reshape – permute, hierarchically.

The middle column of Figure 2 demonstrates this step for $q = 2$ and $d = 2$, creating a 4D array whose outer and inner tiles have shapes 2×2 and 3×2, respectively:

$$N = (n_1^1 \times n_2^1) \times (n_1^2 \times n_2^2) = (2 \times 2) \times (3 \times 2)$$

Note that this step is just a reshape operation applied to the logical layout that does not change yet the physical layout, i.e., laying down the elements of the array in the middle column in increasing order of inner dimensions still results in $[0, 1, \ldots, 22, 23]$.

The second step reorders the elements of tiles by defining permutations (possibly) at each of the q levels of the hierarchy: *The general case* is covered by a pair of user-defined functions implementing a bijection between the index space of the corresponding tile and its flat space. For example, the right column of the figure shows that elements of each innermost tile are reordered according to the user-defined permutation $p_{n_1^2,n_2^2}(i, j) = (n_1^2\text{-}1\text{-}i) \cdot n_2^2 + (n_2^2\text{-}1\text{-}j)$, which reverses the elements on each of the two dimensions.

For ease of use, LEGO also supports a *specialized case* that interchanges the dimensions of a tile by some statically known permutation σ of $[1, \ldots, d]$. The right column of Figure 2 uses $\sigma = [2, 1]$ on the outermost tile level to transpose the locations of the inner tiles. Such a reordering allows the user to bypass the hassle of writing functions, and may enable further simplifications of index computation and analyses. The layout transformation of Figure 2 is expressed as:

$$\textbf{GroupBy}([6, 4], \textbf{OrderBy}(\textbf{RegP}([2, 2], [2, 1]), \textbf{GenP}([3, 2], p, p^{-1}))))$$

where $[6, 4]$ specifies the shape of the logical view. The **OrderBy** construct specifies that the outer tile level, of shape $[2, 2]$, is reordered by transposing its dimensions, hence $\sigma = [2, 1]$, and the elements of the inner tiles, of shape $[3, 2]$, are reordered by the user-defined permutation p (whose inverse p^{-1} is not shown).

The user interface with the layout consists of two functions: (1) `apply`, which maps a logical-view index to its flat physical position; and, (2) `inv` that performs the reverse. For example, `apply`$([4, 1]) = 6$ and `inv`$(6) = [4, 1]$. The rationale is that element 17 at index $[4, 1]$ in the logical view is ultimately placed in memory at position 6, corresponding to index $[0, 1, 0, 0]$ of the 4D array shown on the right:

$$
\begin{array}{lll l}
\sigma_d & ::= & [\overline{k}^d] & \text{Ct. Perm. of } [1 \ldots d] \\
Tile_d & ::= & [\overline{e}^d] & \text{Sizes of a } d\text{-dim tile} \\
Prm_d & ::= & \mathbf{RegP}(Tile_d, \sigma_d) & \text{Regular Perm} \\
 & | & \mathbf{GenP}(Tile_d, f, f^{inv}) & \text{Irregular Perm} \\
OrderBy & ::= & \mathbf{OrderBy}(\overline{Prm_d}^{q_2}) & \\
GroupBy & ::= & \mathbf{GroupBy}(\overline{Tile_{d'}}^{q_1},\ \overline{OrderBy}^{q_3}) &
\end{array}
$$

$$
\begin{array}{llll}
e & ::= & k & \text{Ct.} \in \mathbb{Z} \\
 & | & x & \text{Var.} \\
 & | & e + e & \text{Add} \\
 & | & e * e & \text{Mul.} \\
 & | & \ldots & \text{Other}
\end{array}
$$

Notation:

q, d	sequence size
h, k	seq. iterators
i, j	indices
n, m	int expressions

Notation: $\overline{o}^q$ is a sequence $o_1, \ldots, o_q$ of q objects of some kind. d' and d denote the dimensionality of a tiling hierarchy.

Fig. 3: Grammar: $GroupBy$ gives the logical view of an index space whose elements are reordered by a chain of $OrderBy$.

- 17 belonged to tile $[1, 0]$ of the array in the middle column and transposition has brought its tile to position $[0, 1]$
- within its tile, 17 was placed as the last index of both dimensions, and reverting them brings it in position $[0, 0]$.

Mathematical Intuition: The mathematical glue that binds the multi-layered components are the well known *canonical bijections*, denoted $\mathcal{B}$ and $\mathcal{B}^{-1}$, that connect a multi-dimensional index space to its corresponding flat space. For example, the index transformation between the logical view and the reshaped tile hierarchy—i.e., between the left and middle columns of Figure 2—is obtained by (1) flattening the logical-view index by applying $\mathcal{B}$, and then by (2) unflattening the resulted index in the tiled space by applying $\mathcal{B}^{-1}$.

LEGO enables the user to express piece-wise bijections that document the reordering performed at each level of the tile hierarchy, and provides an automatic procedure that combines these into one bijection **B** that covers the whole index space. This essentially allows the user to work in a suitable logical space, say of shape $n'_1 \times \ldots \times n'_{d'}$, while LEGO transparently performs the mapping to the reordered flat (physical) space by means of the `apply` bijection:

$$\mathbf{B} \circ \mathcal{B}^{-1}_{(n^1_1 \cdot \ldots \cdot n^1_d)\ \cdot\ \ldots\ \cdot\ (n^q_1 \cdot \ldots \cdot n^q_d)} \circ \mathcal{B}_{n'_1, \ldots, n'_{d'}}$$

with $n^1_1 \ldots n^q_d$ defined in Equation Eq. (1). As well, since bijections are reversible, one can find the logical multi-dimensional index corresponding to a physical one by using the `inv` bijection, inferred as:

$$\mathcal{B}^{-1}_{n'_1, \ldots, n'_{d'}} \circ \mathcal{B}_{(n^1_1 \cdot \ldots \cdot n^1_d)\ \cdot\ \ldots\ \cdot\ (n^q_1 \cdot \ldots \cdot n^q_d)} \circ \mathbf{B}^{-1}$$

Finally, LEGO allows to chain reordering **OrderBy** transformations, by similarly gluing them with canonical bijections. The next section III-B presents the LEGO grammar and details its implementation.

B. Building Blocks & Lowering Algorithm

Figure 3 presents the LEGO grammar: A **GroupBy** consists of (1) a hierarchical tile decomposition on some arbitrary but fixed number q_1 of levels, such that each tile has the same dimensionality d', together with (2) a chain of reordering **OrderBy** transformations.

Notation: o_k denotes k^{th} object from sequence $\overline{o}^q = o_1, \ldots, o_q$ and $\overline{o}^{h=q_1 \ldots q_2}$ creates a new sequence from objects $o_{q_1}, \ldots, o_{q_2}$.

σ_d^{-1} is obtained by scattering $[1, \ldots, d]$ at the positions of σ_d.

$$\mathcal{B}_{\overline{n}^q}(\overline{i}^q) \ = \ i_1 \cdot \textstyle\prod_{k=2}^q n_k \ + \ \ldots \ + \ i_{q-1} \cdot n_q \ + \ i_q$$

$$\mathcal{B}^{-1}_{\overline{n}^q}(\ i\ ) \ = \ \textbf{if } q = 1 \textbf{ then } i \textbf{ else } (\mathcal{B}^{-1}_{\overline{n}^{h=1 \ldots q-1}}(i/n_q),\ i \,\%\, n_q)$$

$$
\begin{aligned}
&\mathbf{GenP}([\overline{n}^d],\ f_{\overline{n}^d},\ f^{inv}_{\overline{n}^d})::\texttt{apply}(\overline{i}^d) \ = \ f_{\overline{n}^d}(\overline{i}^d) \\
&\mathbf{GenP}([\overline{n}^d],\ f_{\overline{n}^d},\ f^{inv}_{\overline{n}^d})::\texttt{inv}(\ i_{flat}\ ) \ = \ f^{inv}_{\overline{n}^d}(\ i_{flat}\ ) \\
&\mathbf{GenP}([\overline{n}^d],\ f_{\overline{n}^d},\ f^{inv}_{\overline{n}^d})::\texttt{dims}() \ = \ \overline{n}^d
\end{aligned}
$$

$$
\begin{aligned}
&\mathbf{RegP}([\overline{n}^d],\ \sigma_d)::\texttt{apply}(\overline{i}^d) \ = \ \mathcal{B}_{\sigma_d(\overline{n}^d)}(\ \sigma_d(\overline{i}^d)\ ) \\
&\mathbf{RegP}([\overline{n}^d],\ \sigma_d)::\texttt{inv}(\ i_{flat}\ ) \ = \ \sigma_d^{-1}(\ \mathcal{B}^{-1}_{\sigma_d(\overline{n}^d)}(\ i_{flat})\ ) \\
&\mathbf{RegP}([\overline{n}^d],\ \sigma_d)::\texttt{dims}() \ = \ \overline{n}^d
\end{aligned}
$$

```
OrderBy($\overline{Perm_d}^q$)::apply( $\overline{i}^{d \cdot q}$ ) =
  $i_{flat} \leftarrow 0$
  for $Perm \in \overline{Perm_d}^q$ and $k \in 0 \ldots q-1$ do
      $\overline{n}^d \leftarrow Perm$.dims();    $\overline{i_{cur}}^d \leftarrow \overline{i}^{h=k \cdot d+1 \ldots k \cdot d+d}$
      $i^{cur}_{flat} \leftarrow Perm$.apply($\overline{i_{cur}}^d$)
      $i_{flat} \leftarrow i^{cur}_{flat} + i_{flat} \cdot \prod_{h=1}^d (n_h)$
  return $i_{flat}$
OrderBy($\overline{Perm_d}^q$)::inv( $i_{flat}$ ) =
  $\overline{i} \leftarrow$ empty sequence
  for $Perm \in$ reverse($\overline{Perm_d}^q$) do
      $\overline{n}^d \leftarrow Perm$.dims();    $p \leftarrow \prod_{h=1}^d (n_h)$
      $i^{cur}_{flat} \leftarrow i_{flat} \% p$;          $i_{flat} \leftarrow i_{flat} / p$
      $\overline{i} \leftarrow Perm$.inv($i^{cur}_{flat}$), $\overline{i}$
  return $\overline{i}$
OrderBy($\overline{Perm_d}^q$)::dims( ) = $\overline{n} \leftarrow$ empty sequence
  for $Perm \in \overline{Perm_d}^q$ do  $\overline{n} \leftarrow \overline{n}$, $Perm$.dims()
  return $\overline{n}$
```

Fig. 4: Semantics of `apply` and `inv` of **OrderBy** Blocks.

An **OrderBy** defines its own d-dimensional tile hierarchy on some q_2 levels by means of a sequence of permutations Prm_d. Prm_d has two constructors: **GenP** denotes a general permutation of the *elements* of a tile, by a user-defined function f, whose inverse is f^{inv}. **RegP** denotes a regular (constant) permutation σ of the *dimensions* of a tile, i.e., if the logical shape of the tile is $\overline{n}^d$ then the reordered shape is $\sigma(\overline{n}^d)$. Finally, a tile is represented by its shape, as a list of dimension sizes, which are expressions.

Of course, the total number of elements of the hierarchical tiling defined by **GroupBy** must equal that of each of the chained **OrderBy**s. In practice, tiles within an **OrderBy** or **GroupBy** do not have to share the same dimensionality, e.g., one may use a 1-D grid of 3-D blocks; we use this restriction to simplify the presentation.

LEGO's interface to the user consists of an `apply` and `inv` functions that can be called on a **GroupBy** block: `apply` receives as argument a multi-dimensional index corresponding to the logical shape of **GroupBy**, and results in the corresponding flat index in the (reordered) physical layout, while `inv` does the opposite. We define this functionality by a syntax-directed translation [25], detailed in Figures 4 and 5, which

```
$\textbf{GroupBy}(\ ([\overline{n^1}^d], \ldots, [\overline{n^{q_g}}^d]),\ \overline{O}^v)$::apply( $\overline{i}^{d \cdot q_g}$ )  $=$
  $i_{flat} = \mathcal{B}_{(n_1^1 \cdot \ldots \cdot n_d^{q_g})}(\ \overline{i}^{d \cdot q_g}\ )$
  for $O\ \in$ reverse($\overline{O}^v$) do
      $\overline{n'^1}^{d'}, \ldots, \overline{n'^{q_o}}^{d'} \leftarrow O$.dims();
      $\overline{i'}^{d' \cdot q_o} \leftarrow \mathcal{B}^{-1}_{n'^1_1 \cdot \ldots \cdot n'^{q_o}_{d'}}(\ i_{flat}\ )$
      $i_{flat} \leftarrow O$.apply( $\overline{i'}^{d' \cdot q_o}$ )
  return $i_{flat}$
$\textbf{GroupBy}(\ ([\overline{n^1}^d], \ldots, [\overline{n^{q_g}}^d]),\ \overline{O}^v)$::inv( $i_{flat}$ )  $=$
  for $O\ \in\ \overline{O}^v$ do
      $\overline{n'^1}^{d'}, \ldots, \overline{n'^{q_o}}^{d'} \leftarrow O$.dims();
      $\overline{i'}^{d' \cdot q_o} \leftarrow O$.inv( $i_{flat}$ )
      $i_{flat} \leftarrow \mathcal{B}_{n'^1_1 \cdot \ldots \cdot n'^{q_o}_{d'}}(\ \overline{i'}^{d' \cdot q_o}\ )$
  return $\mathcal{B}^{-1}_{(n_1^1 \cdot \ldots \cdot n_d^{q_g})}(\ i_{flat}\ )$
$\textbf{GroupBy}(\ ([\overline{n^1}^d], \ldots, [\overline{n^{q_g}}^d]),\ \overline{O}^v)$::dims( ) $= \overline{n^1}^d, \ldots, \overline{n^{q_g}}^d$
```

Fig. 5: Semantics of `apply` and `inv` of **GroupBy** Blocks

implements the `apply`, `inv` and `dims` functions for each syntactic category of the LEGO language by combining the functionality of its syntactic constituents (where `dims` is used to track the dimension sizes of a space).

GenP simply applies the provided user-defined functions f, f^{inv}. **RegP**'s `apply` flattens the index by applying the canonical bijection $\mathcal{B}$ in the physical (permuted) layout, hence the dimensions and index are permuted by σ_d. Its `inv` unflattens the index by $\mathcal{B}^{-1}$ using the physical (permuted) dimensions and recovers the logical-space index by permuting back the physical index by the inverse of σ_d, which is obtained by scattering $[1, \ldots, d]$ at the positions of σ_d.

OrderBy's `apply` traverses the tiling space from outermost inwards, and at each steps flattens and accumulates the corresponding part of the index; `inv` unflattens the index from innermost outwards.

Finally, Figure 5 shows the lowering algorithm of **GroupBy**: its `apply` first flattens its index in its logical space, and then traverses the chain of reordering transformations $\overline{O}^v$ in reverse order, and for each one, denoted O, it remaps the flat index to O's logical space by $\mathcal{B}^{-1}$, and applies the reordering. **GroupBy**'s `inv` traverses $\overline{O}^v$ forward, and for each reorder O, it applies its inverse, and then flattens it according to O's logical space. Ultimately, the resulting index is unflattened in **GroupBy**'s space.

Figure 6 demonstrates a more complex example that uses a 6×6 logical view, depicted in the left column, whose elements correspond for convenience to the flat index space $[0, \ldots, 35]$.

The middle column shows a reordering transformation O_2 that tiles the logical view into a 2×2 grid of 3×3 blocks. This is achieved by a **RegP** permutation that first stripmines each of the logical-view dimensions of size 6 into two smaller dimensions of sizes 2×3, and then interchanges the middle (second and third) dimensions, i.e.,

O_2 = **OrderBy**(**RegP**([2,3,2,3], σ=[1,3,2,4]))

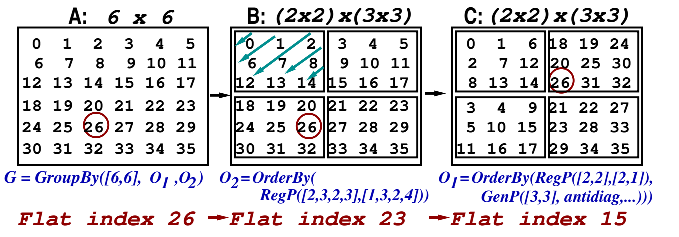

A: 6 x 6
B: (2x2)x(3x3)
C: (2x2)x(3x3)
G = GroupBy([6,6], O1 ,O2)
O2=OrderBy(RegP([2,3,2,3],[1,3,2,4]))
O1=OrderBy(RegP([2,2],[2,1]), GenP([3,3], antidiag,...)))
Flat index 26 → Flat index 23 → Flat index 15

Fig. 6: $\mathbf{2} \times \mathbf{2} \times \mathbf{3} \times \mathbf{3}$ tiling followed by transposing the outer dimensions and applying anti-diagonal permutation in the inner $\mathbf{3} \times \mathbf{3}$ blocks. The logical view is a $\mathbf{6} \times \mathbf{6}$ matrix.

```
def antidiag(n, i, j):
  antidg = i + j + 1
  if(antidg <= n):
    return i + (antidg*(
         antidg-1))/2
  else:
    antidg = 2*n - antidg
    gauss = (antidg * (
         antidg-1))/2
    return n*n - n + i -
         gauss
```

```
def antidiag$^{inv}$(n, x0):
  S = n*(n+1) / 2
  $x$ = x0 if x0 < S else n*n-1 -
       x0
  antidg = $\lfloor\sqrt{2 * x}\rfloor$
  antidg += ( $x$ >= (antidg*(
       antidg+1))/2 )
  i = $x$ - antidg*(antidg-1)/2
  j = antidg - i - 1
  return (i,j) if x0 < S else (n
       -1-i, n-1-j)
```

Fig. 7: Anti-Diagonal Permutation of an $n \times n$ Logical Space

The right column of Figure 6 applies another reordering O_1 that similarly uses a hierarchical space of a 2×2 grid of 3×3 blocks, in which the grid is transposed (**RegP**) and the elements of each block are permuted (**GenP**) such that they are laid out in the order in which they appear on the block's $2 \cdot 3 - 1$ antidiagonals. In practice, the dimensions d and q are omitted, being implicitly inferred from the shape of the arguments:

O_1 = **OrderBy**(**RegP**([2,2],σ=[2,1]), **GenP**([3,3], antidiag,..))

Finally, the pseudocode of the user-defined antidiagonal permutation of a $n \times n$ logical space, and its inverse, are shown in Figure 7. One can verify that the element at index $[4, 2]$ in the 6×6 logical view (left column), i.e., representing 26, is reordered by O_2 in the middle column to flat index 23 (i.e., multi-dimensional index [1,0,1,2]), and then by O_1 in the right column to physical index 15 (i.e., [0,1,2,0]). Conversely, one can use `inv` to compute that the flat physical index 15 corresponds to the logical-view index $[4, 2]$.

Correctness: The algorithm presented in Figures 4 and 5 provably implements a bijection, i.e., is correct by construction, if two assumptions hold: *First*, the user-defined function(s) f^{inv} (and $f^{inv}_{\overline{n}^d}$) of **GenP**($[\overline{n}^d]$, $f_{\overline{n}^d}$, $f^{inv}_{\overline{n}^d}$) actually implement a bijection (and its inverse) from the multidimensional space $\mathcal{Z}_{n_1} \times \ldots \times \mathcal{Z}_{n_d}$ to the flat space $\mathcal{Z}_{n_1 \cdot \ldots \cdot n_d}$. This is currently left as user's responsibility. *Second*, the total number of elements of the multidimensional spaces defined by the **GroupBy** and each of its contained **OrderBy** constructs is the same. This can be cheaply verified dynamically and even hoisted outside recurrences that do not affect these sizes.

Classification of Bijective Layouts: A single **OrderBy** construct using only regular permutations (**RegP**) implements an *affine* layout in the `apply` direction—i.e., a formula such as $i_! \cdot s_1 + \ldots + i_q \cdot s_q$, where $i_{1,\ldots,q}$ are the target indices and $s_{1,\ldots,q}$ are constant strides (a product of dimension sizes).

Sequencing two or more **OrderBy** constructs using only regular permutations *does not necessarily result in an affine layout* because the canonical bijections ($\mathcal{B}$ and $\mathcal{B}^{-1}$) applied at the borders introduce division and modulo operations that are not guaranteed to simplify away. Finally, user-defined permutations (**GenP**) allow *fully arbitrary* (bijective) layouts, e.g., using indirect arrays [12], and quadratic (polynomial) indexing as in the anti-diagonal permutation of Figure 7.

Notation and Syntactic Sugar: For convenience of presentation, this section has used the grammar in Figure 3. The rest of the paper uses a notation that chains reordering and the final grouping transformations by means of dots:

$$\begin{aligned}&\textbf{GroupBy}([6,6]).\\&\textbf{OrderBy}(\textbf{RegP}([2,3,2,3],[1,3,2,4])).\\&\textbf{OrderBy}(\textbf{RegP}([2,2],[2,1]),\\&\qquad\qquad \textbf{GenP}([3,3],\text{antidiag},\text{antidiag}^{inv}))\end{aligned} \tag{2}$$

As well, we define syntactic sugar for common operations:

$$\begin{array}{rcl}\text{Row}([n_1,\ldots,n_d]) & \equiv & \textbf{RegP}([n_1,\ldots,n_d],\ [1,2,\ldots,d])\\ \text{Col}([n_1,\ldots,n_d]) & \equiv & \textbf{RegP}([n_d,\ldots,n_1],\ [d,\ldots,2,1])\\ \text{TileBy}_{q\times d}\ (& \equiv & \textbf{GroupBy}([\overline{n^1}^d,\ldots,\overline{n^q}^d]).\\ {[\overline{n^1}^d],\ldots,[\overline{n^q}^d]}) & & \textbf{OrderBy}(\textbf{RegP}([\overline{n^1}^d,\ldots,\overline{n^q}^d],\sigma_{d\times q}))\\ \text{TileOrderBy}_{q\times d} & \equiv & \textbf{GroupBy}(P_d^1,\ldots,P_d^q).\\ (P_d^1,\ldots,P_d^q) & & \textbf{OrderBy}(\ \textbf{RegP}(\\ & & \quad \sigma_{d\times q}(P_d^1.dims,..,P_d^q.dims,\sigma_{d\times q}^{-1}))\end{array}$$

$$\textbf{where } \sigma_{d\times q} = \text{flatten}(A), \textbf{with } A : [d][q]\textbf{int}, A_{k,h} = k + 1 + d \cdot h$$

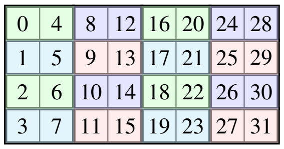
Row and Col define row- and column-major layouts, corresponding to permuting dimensions by identity and by $[d,\ldots,1]$. TileBy$_{q\times d}$ denotes hierarchical tiling of d dimensions on q levels, e.g., TileBy$_{3\times2}$ and TileBy$_{2\times3}$ have permutations $\sigma_{2\times3} = [1,3,5,2,4,6]$ and $\sigma_{3\times2} = [1,4,2,5,3,6]$, respectively, and applying these permutations to their logical dimensions results, as expected, in the physical spaces $(n_1^1 \times n_1^2 \times n_1^3) \times (n_2^1 \times n_2^2 \times n_2^3)$, and $(n_1^1 \times n_1^2) \times (n_2^1 \times n_2^2) \times (n_3^1 \times n_3^2)$. TileOrderBy similarly defines a hierarchical-tiling reordering.

C. Comparison with CuTe/Graphene Algebra

We identify two primary distinctions between the LEGO and CuTe/Graphene shape algebra representations.

Elimination of Explicit Strides: LEGO supports all of the strided, rectangular layouts that can be expressed in the shape algebra for CuTe and Graphene. A significant difference in the shape specification is that the CuTe/Graphene shape algebra requires the performance programmer to provide the strides for the layout, whereas LEGO derives the strides internally from the hierarchical tiling specification. Table I compares the LEGO and CuTE/Graphene layout specification for some of the various layouts used in the figures in this paper. The simple tiled layout for the input matrices of Figure 1 describes a 4D tiled data layout; the CuTe/Graphene layout in the third column linearizes the four dimensions to derive a stride. The tiled representation for CuTe/Graphene describing $G \circ O_2$ in Figure 6 expresses the original layout A of 6×6 on the top row. On the second row, the stride is specified: 6 between rows and 1 between columns within a row. The resulting layout B is (2×2) × (3×3), with a stride of 18 between block rows, and 3 between block columns. The stride is 6 elements across tiles in the row dimension, and 1 in the column dimension.

Even with this relatively simple tiled example, the need to specify strides already muddies the layout description. However, the specification becomes more complex with the example of Figure 8, which matches Figure 4d in the Graphene paper [22]. In this case, as depicted in the figure, the goal is to create tiles (denoted by locations with the same color) that are not contiguous in either dimension. In LEGO's formulation, this layout is simply a permutation of the five dimensions resulting from the tiling. In contrast, Graphene expresses the layout with complex multi-dimensional strides.



0	4	8	12	16	20	24	28
1	5	9	13	17	21	25	29
2	6	10	14	18	22	26	30
3	7	11	15	19	23	27	31

Fig. 8: Example layout that is non-contiguous in 2 dimensions: LEGO and Graphene layout specifications shown in Table I.

Extended Layout Support: LEGO is not limited to strided layouts; it also accommodates additional layouts that require complex indexing expressions beyond rectangular, strided layouts. For example, the antidiagonal layout for O_1 in Figure 6, whose implementation is described in Figure 7 and Equation 2, cannot be supported by the CuTe/Graphene shape algebra. Because LEGO can represent any bijective mapping between physical and logical layout, it can represent this antidiagonal, and, as discussed in Section VII, provides a foundation for other commonly-used bijective layouts.

D. Beyond Bijective Layouts

In some cases, LEGO primitives can be composed to support certain layouts that are not bijective. Importantly, we support partial tiles where the tile size does not evenly divide the problem size, adopting a similar approach to the oversampling method in CuTe [21] as follows. A new constructor **ExpandBy**, as illustrated in Figure 9, performs the necessary widening/narrowing conversions between a physical d-dimensional space $\overline{n}^d$ whose sizes do not evenly divide the tiles, and an extended one $\overline{n'}^d$ that does, such that the bijective layout G is safely applied in the expanded space. Specifically, `apply` projects a logical index through G to a flat index in the expanded layout, unfolds it via the canonical bijection $\mathcal{B}$, accepts it only if the coordinates fall within the original physical extents, and reports the corresponding flat position in the original space (otherwise -1). Conversely, `inv` lifts to an original flat index, re-flattens it in the expanded space via the canonical bijections, and then inverts through G.

TABLE I: Comparison of LEGO and CuTe/Graphene layouts for examples in figures and performance results.

Fig.	**LEGO**	**CuTe/Graphene**
1	TileBy($[M/BM, K/BK], [BM, BK]$).OrderBy(Row(M, K))	$(\begin{bmatrix} M/BM & K/BK & BM & BK \\ K*BM & BK & K & 1 \end{bmatrix})$
6mid	GroupBy($[6, 6]$).OrderBy(RegP($[2, 3, 2, 3], [1, 3, 2, 4]$))	$(\begin{bmatrix} 2, & 2 \\ 18, & 3 \end{bmatrix}) \cdot (\begin{bmatrix} 3, & 3 \\ 6, & 1 \end{bmatrix})$
8	GroupBy($[2, 2, 2, 2, 2]$).OrderBy(RegP($[2, 2, 2, 2, 2], [5, 2, 4, 3, 1]$))	$(\begin{bmatrix} 2, & 2 \\ 1, & 8 \end{bmatrix}) \cdot (\begin{bmatrix} 2, & (2,2) \\ 2, & (4,16) \end{bmatrix})$
12b	GroupBy($[R, R], [T, T]$).OrderBy(Row($R*T, R*T$))	$(\begin{bmatrix} R & R & T & T \\ RT^2 & T^2 & T & 1 \end{bmatrix})$
12c	TileBy($[N/B, N/B, N/B], [B, B, B]$).OrderBy(Row($N/B, N/B, N/B$), Row($B, B, B$))	$(\begin{bmatrix} N/B & N/B & N/B & B & B & B \\ N^2B & NB^2 & B^3 & B^2 & B & 1 \end{bmatrix})$

```
$ExpandBy$ ::= **ExpandBy**($\overline{Tile_d}$, $\overline{Tile_d}$, $GroupBy$)
**ExpandBy**($[\overline{n}^d], [\overline{n'}^d], G$)::apply( $\overline{s}^q$ ) =
    $i_{\text{flat}} \leftarrow G$::apply( $\overline{s}^q$ )
    $\overline{i'}^d \leftarrow \mathcal{B}^{-1}_{n'_1 \cdot \ldots \cdot n'_d}( i_{\text{flat}} )$
    **if** $(i'_1 < n_1) \wedge \cdots \wedge (i'_d < n_d)$ **then**
        **return** $\mathcal{B}_{n_1 \cdot \ldots \cdot n_d}( \overline{i'}^d )$
    **else return** $-1$
**ExpandBy**($[\overline{n}^d], [\overline{n'}^d], G$)::inv( $i_{\text{flat}}$ ) =
    $\overline{i}^d \leftarrow \mathcal{B}^{-1}_{n_1 \cdot \ldots \cdot n_d}( i_{\text{flat}} )$
    $i'_{\text{flat}} \leftarrow \mathcal{B}_{n'_1 \cdot \ldots \cdot n'_d}( \overline{i}^d )$
    **return** $G$::inv( $i'_{\text{flat}}$ )
```

Fig. 9: Grammar and (`apply`/`inv`) semantics of **ExpandBy**.

To accommodate injective layouts such as broadcasting $(i, j) \mapsto i$ or $(i, j) \mapsto j$ and even-mapping $i \mapsto 2i$, we restrict the language to exporting only `apply` (not `inv`) and to using exactly one **GroupBy** followed by an **OrderBy** of the same shape, where that **OrderBy** contains a single **GenP** that may be injective. For the remainder of the paper, we focus on bijective layouts (i.e., **GenP** being bijective).

IV. Integrating LEGO into Ecosystems

As demonstrated in previous sections, LEGO establishes an algebraic framework independent of any compilation system. We see it as an important tool that can be integrated into a compiler or code generator, particularly to support data/thread tile abstractions for GPUs, but also applicable to threaded CPU code generation and future heterogeneous hardware. To demonstrate the power of the LEGO indexing mapping from layout specification to code generation, it was essential to integrate LEGO into mature ecosystems and rely on these to optimize code resulting from the layout mapping.

In this section, we describe the integration of LEGO into Triton, CUDA, and MLIR. The integration with Triton and CUDA illustrates a straightforward implementation using Python. The incorporation within MLIR underscores the tool's versatility.

A. Code Generation via Instantiating Templates

Our implementation for generating Triton and CUDA code utilizes an approach in which the user supplies code containing placeholders, and separately-defined layouts. The placeholders, marked using the Jinja2 [26] syntax {{ }}, are intended to represent index expressions or layout-specific logic. LEGO then generates appropriate symbolic expressions based on the user-defined layout and replaces the corresponding placeholders within the template. This process offloads the complexity of constructing low-level index calculations from the user. An example of this specification was shown in Figure 1 (right).

For this purpose, the LEGO algebra is integrated into the SymPy framework [27], a Python library for symbolic mathematics. This integration enables advanced symbolic reasoning and high-level manipulation of index expressions, including algebraic simplification. However, SymPy does not have all the necessary information to generate the optimized index expression. In particular, it lacks details about the range of variables used to index into the layout. We propagate this range information through the layout and develop a custom SymPy expression traversal that leverages these range constraints to simplify the index expressions. Moreover, since our algebra involves modulo and floor-division operations, we apply seven custom simplifications summarized in Table II. Each rule's side-conditions (e.g. non-negativity and upper-bound checks) are proved by the Z3 SMT solver [28] using the index ranges derived from the layout specification. In addition, users can provide their own constraints to the system to further simplify the expression. The indexing code is then generated by the Python and C printers provided by SymPy.

For integration with Triton, we have introduced specialized slicing syntax analogous to NumPy's slice notation [29]. Specifically, when a user employs a colon (`:`) to denote the entire dimension—specified through TileBy—the system generates a corresponding `tl.arange`, whose bounds are derived from the layout specifications. Furthermore, Triton mandates that the upper and lower bounds of this range be known at the time of compilation. We show the final Triton code generated by this process in Figure 10, starting with the input from Figure 1 (right). We also evaluated whether pre-expanding terms in index expressions before invoking SymPy's simplification routines (including our range-simplification pass) improves performance compared with simplifying the unexpanded expressions, since expansion can reveal additional optimization opportunities. In the NW benchmark, skipping pre-expansion produced better performance by minimizing the total number of operations. In the LUD benchmark, pre-expansion helped by exposing simplifications that lowered the operation count and improved runtime. To accommodate both cases, we use a simple cost model that counts operations in the generated expression and selects the variant with the lowest count, choosing the unexpanded form for NW and the expanded form for LUD.

TABLE II: Integer division and modulo simplification rules.

Pattern	Result	Condition
`(d*q + r) % d`	`r % d`	$d \neq 0$

`(d*q + r) / d`	`q`
	`q + r / d`
		$d \neq 0,\ 0 \leq r < d$
		otherwise

`(x % d) / d`	`0`	$d > 0$
`x / a`	`0`	$a > 0,\ 0 \leq x < a$
`x % a`	`x`	$a > 0,\ 0 \leq x < a$
`(n + y) / 1`	`n + (y / 1)`	$n \in \mathbb{Z}$
`a*(x / a) + x % a`	`x`	$a \neq 0$

```
@triton.jit
def matmul_kernel(
        a_ptr, b_ptr, c_ptr,
        M, N, K,
        BM: tl.constexpr, BN: tl.constexpr, BK: tl.constexpr,
        GM: tl.constexpr,
        ACTIVATION: tl.constexpr
):
    pid = tl.program_id(axis=0)
    nt_m = tl.cdiv(M, BM)
    nt_n = tl.cdiv(N, BN)
    pid_m = pid % min(GM, nt_m) + (((pid)//(nt_n*min(GM, nt_m))) % max(1,
    ((nt_m)//(GM))))*min(GM, nt_m)
    pid_n = ((pid % (nt_n*min(GM, nt_m)))//(min(GM, nt_m)))
    accumulator = tl.zeros((BM, BN), dtype=tl.float32)
    for k in range(0, tl.cdiv(K, BK)):
        a_ptrs = a_ptr + BK*k + K*(BM*pid_m + ((tl.arange(0, BM))[:, None])) +
        ((tl.arange(0, BK))[None, :])
        b_ptrs = b_ptr + BN*pid_n + N*(BK*k + ((tl.arange(0, BK))[:, None])) +
        ((tl.arange(0, BN))[None, :])
        a = tl.load(a_ptrs)
        b = tl.load(b_ptrs)
        accumulator = tl.dot(a, b, accumulator)
    c = accumulator.to(tl.float16)
    c_ptrs = c_ptr + BN*pid_n + N*(BM*pid_m + ((tl.arange(0, BM))[:, None])) +
    ((tl.arange(0, BN))[None, :])
    tl.store(c_ptrs, c)
```

Fig. 10: LEGO layouts instantiated into Triton template of Figure 1 (right).

B. End-to-end Code Generation in MLIR

MLIR facilitates end-to-end code generation through its robust dialect system. We integrate LEGO into MLIR by using the previously-described SymPy expression simplification, creating a custom SymPy printer using the MLIR Python bindings. In this framework, the layout algebra is implemented with the `arith` and `affine` dialects for arithmetic and control flow operations, the `memref` and `vector` dialects for managing memory operations, and the `gpu` dialect providing GPU code generation primitives. This approach leverages Python bindings to ensure compatibility with both standard MLIR dialects and custom user dialects, which can take advantage of LEGO layout algebra for their specialized implementations. A single MLIR file is then generated, encapsulating both layout information and compute code.

By integrating LEGO into MLIR, there is the potential for broader adoption in domain-specific frameworks, including but not limited to tensor computations. We demonstrate this implementation here, but such an integration with other dialects will be the subject of future work.

V. Evaluation

The goal of the performance evaluation is to demonstrate LEGO data and thread-block layouts support code generation in a variety of contexts, and integrated with various state-of-the-art frameworks.

1) *Triton to demonstrate integration in state-of-the-art DSLs*, e.g., targeting tensor-core utilization. We match Triton's performance while simplifying the input specification.
2) *CUDA to demonstrate integration in a mainstream GPU programming language*, and the benefits of a richer set of data and thread-block layouts.
3) *MLIR to demonstrate integration in mainstream compiler frameworks*.

LEGO data layouts used in the experiments include 2D tiles that are in row-major or column-major order. We also employ a novel 3D brick data layout in the CUDA experiments. For thread layouts, 2D thread blocks in row major order are common, but we also use 3D thread blocks for bricks, an antidiagonal for NW, and a thread coarsening pattern for LUD (please refer to Table I).

Table III summarizes the one-time code-generation and simplification latency for each application on a personal laptop (Apple M2 Max). The generation and simplification time of the per-application code ranges from sub-second to several seconds. This overhead is limited to index-generation time and does not affect steady-state execution.

TABLE III: Per-application code generation and simplification.

Benchmark	Generation time
Layernorm FWD + BWD	0.33 s
Grouped GEMM	0.65 s
Softmax	0.05 s
Matmul (each variant)	1.11 s
LUD	0.87 s
NW	0.46 s
Bricks (Cube/Star)	5.95 / 18.07 s
Transpose (Naive/SMEM)	1.07 / 1.15 s

In this study, the experiments were executed using an NVIDIA Ampere A100 80GB GPU, deployed on an AMD EPYC 7513 processor with 32 cores under a CentOS operating system. The experimental framework was configured with LLVM (commit 556ec4a), Triton 3.2.0, PyTorch 2.5.1, and CUDA 12.4. Each benchmark was executed 25 times for warm-up, followed by 100 repetitions for data collection, and the mean performance value from these repetitions is reported.

A. Triton Benchmarks

In this study, we evaluated the LEGO framework using five benchmarks obtained from the official Triton repository: group GEMM, LayerNorm Fwd, LayerNorm Bwd, softmax, and matrix multiplication in FP16. These benchmarks were

TABLE IV: Arithmetic ops before and after optimization.

Operator	Original Ops	Optimized Ops
LayerNorm (FWD)	6	1
LayerNorm (BWD)	4	0
Softmax	4	0
Grouped GEMM	20	6
Matmul	31	9

selected due to their computational heterogeneity and their frequent application in machine learning workloads. Performance of LEGO versions was measured for three problem sizes against reference implementations from the Triton repository, as shown in Figure 11. We use Triton as our baseline. PyTorch's CUDA backend dispatches matrix multiplications to cuBLAS, resulting in vendor-optimized kernels.

Overall performance is nearly identical between the LEGO and Triton benchmarks. PyTorch/cuBLAS outperforms both for most benchmarks at 2k size, but as the problem size gets larger, the LEGO-generated code is able to fully utilize the tensor cores. For matrix multiplication experiments, we used power-of-two square matrices. We selected configurations that avoided partial tiling in the inputs, thereby eliminating the need for load/store masking in the Triton kernel, ensuring a fair comparison. Four variations of matrix multiplication were generated using the generic kernel template discussed in the previous section, with the only modification being the data layout for matrices A and B. The transposed version employs a column-major layout (Col), while the non-transposed version utilizes a row-major layout (Row); for example, in the case of AB^T, where A is $Row(M,K)$ and B is $Col(K,N)$. This highlights the flexibility of LEGO code generation, demonstrating that by merely altering the data layout, different implementations of matrix multiplication can be achieved. As illustrated in Figure 11, LEGO achieves performance on matrix multiplication comparable to that of Triton and PyTorch/cuBLAS.

For the remaining benchmarks, LEGO and Triton outperform PyTorch/cuBLAS in some cases, although the difference is small with softmax. LEGO also outperforms Triton on the LayerNorm Fwd benchmark because the example in the original repository uses a for loop with an explicit step, which Triton's codegen handles less efficiently than loops with a manually incremented step. For LayerNorm Bwd, we benchmark only the backward pass and skip the forward pass.

For program specification, arithmetic operations in user-defined code were reduced across evaluated components (Table IV), demonstrating LEGO's ability to generate high-performance Triton kernels with simpler expressions

B. CUDA Benchmarks

To illustrate layouts not supported by the other frameworks and the efficacy of exploring different data or thread-block layouts, we present three examples in Figure 12.

The first is the NW benchmark from the Rodinia benchmark suite [30]. Its CUDA implementation consists of two kernels that are called in a loop executed on the host. The kernels utilize a $(b+1)\times(b+1)$ buffer `buff` that is maintained in shared memory, and whose elements on each anti-diagonal are updated in parallel. Since Rodinia requires b to be a multiple of 16 and b is also the size of the CUDA block, it follows that the read and write accesses of the original code exhibit stride b, resulting in expensive bank conflicts.

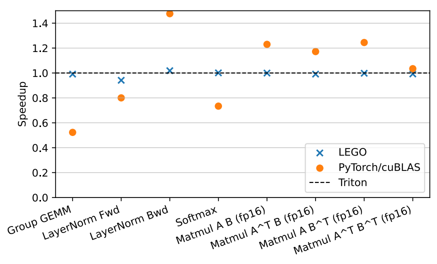
We optimize `buff`'s layout by applying the anti-diagonal reordering (permutation) shown in Figure 7, which uses the indexing expressions generated by LEGO, and by overloading the `[]` operator to redirect logical accesses from the original code. This requires the definition of a small wrapper class for arrays and the modification of only two lines of the original code. LEGO's layout description is shown in Equation 2 of Section III-B. As demonstrated in Figure 12a, this layout transformation improves performance from $1.4\times$ up to $2.1\times$ by reducing shared memory bank conflict and warp stalls.

The second CUDA example is LUD, also from the Rodinia benchmark suite. For this example, we apply a common optimization called thread coarsening, whereby the amount of work performed by each thread is increased [31, 32]. But what distinguishes our approach is that thread coarsening is re-imagined as a layout optimization. LEGO's thread-block layout optimization binds values to both the total number of threads in each dimension and the bounds on the outer loop in the thread. The layout description is provided in Table I, and the resulting performance and roofline are shown in Figure 12b and Figure 13a. Although the baseline code uses a logical LUD block size of 16×16 and a one-to-one correspondence to the CUDA block size, the best performance is obtained with an LUD block size of 64×64 and a coarsening factor of 4, which keeps the CUDA block size at 16×16 yet executes more work per thread block and achieves best block-level parallelism.

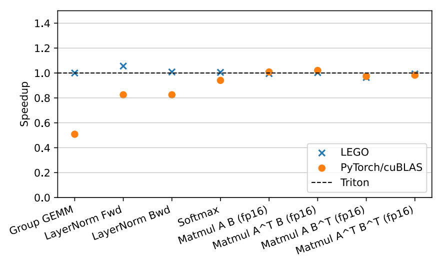
The final example illustrates a modified data layout for 3D stencil computations. The benchmark is based on the array and brick data layout code in Zhou et al. [33]. Bricks are 3D subdomains stored in contiguous memory, so that spatially adjacent data related to a block of computation are also physically adjacent, thus eliminating unnecessary data movement over strided data when a conventional row-major layout is used [12]. The brick layout – in the last row of Table I – is a 6D object; the CuTe/Graphene layout expresses a stride for each dimension. The experiment compares a row-major layout to a brick layout for 3D cube-shaped (27-pt and 125-pt) and star-shaped (7-pt, 13-pt, 19-pt, and 27-pt) stencils. As shown in Figures 12c and 13b, we observe speedups of $3.4\times$–$3.9\times$ across all stencil types solely from changing the data layout, even without integration with vector code generation described in Zhou et al. [12].

C. MLIR Benchmark

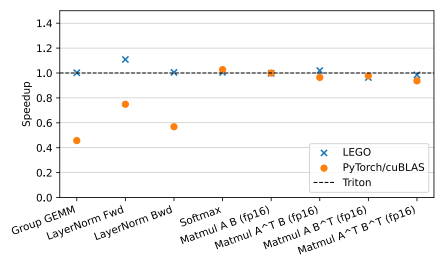
To demonstrate an integration into a compiler framework, we evaluate LEGO's MLIR GPU code with a 2D transpose operation, a simple example to showcase optimization of data movement. Table V shows a comparison between the LEGO-MLIR implementation, compiled from MLIR, and the baseline code from the NVIDIA CUDA, compiled with `nvcc`. In the



(a) $N = 2048$

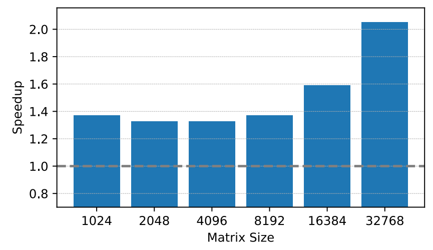


(b) $N = 4096$

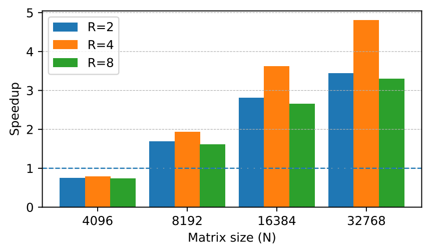


(c) $N = 8192$

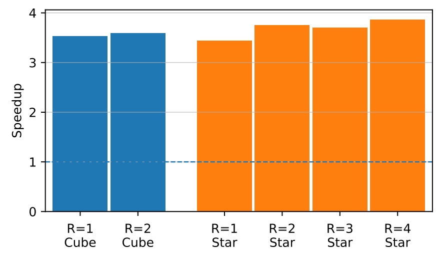
Fig. 11: Performance comparison of Triton and PyTorch against the generated code using LEGO.



(a) NW

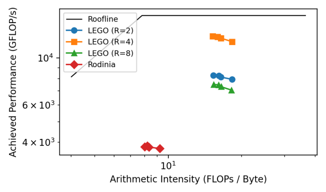


(b) LUD

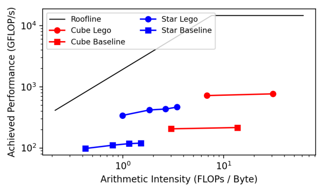


(c) Stencil

Fig. 12: Performance comparison of CUDA benchmarks against Rodinia benchmark for NW and LUD. We compare brick vs. array layout for stencil examples.



(a) LUD



(b) Stencil

Fig. 13: Roofline performance for LUD & Stencil benchmarks.

Naive code, the input and output matrices are read/written from global memory, resulting in uncoalesced global memory accesses. **Smem+Coalesced** stages data in shared memory (another layout in LEGO) so that all global memory accesses are coalesced. Results are listed as throughput (GB/s). In spite of using different compilation frameworks, performance results are comparable, with a slight advantage to LEGO for generating linearized array accesses.

TABLE V: Comparison of LEGO performance on 2D transpose with CUDA SDK baseline using MLIR. Performance is reported in GB/s throughput, so higher numbers are better.

	Naive			**Smem+Coalesced**		
Size	*2048*	*4096*	*8192*	*2048*	*4096*	*8192*
CUDA-SDK	212.0	175.8	175.4	670.0	718.2	735.7
LEGO-MLIR	206.8	178.0	190.7	681.7	741.2	759.4

VI. Related Work

Deriving high-performance implementations of tensor computations is a fertile area of active research. We will focus this section on a narrow set of prior work that integrates data layout and/or data movement into code generation.

Data movement specifications: Historically, data copy was applied in compilers to reorganize submatrices, especially to avoid conflict misses in cache or stage data in explicitly managed storage [34, 35]. CUDA-CHiLL incorporated data-copy into its scheduling language to copy data to/from global memory, shared memory, and texture memory in GPUs [36]. More recently, Fireiron and MDH enrich these data movement specifications for GPUs [37, 38].

Data layout for sparse tensors: Specifying data layout is central to optimizing sparse matrix and tensor computations, where the representation of only nonzero elements varies to better exploit their structure. Moreover, loop optimizations must be reformulated whenever loop indices iterate over a sparse dimension of a tensor [39, 40, 41, 42]. TACO [43] introduced an approach to co-iteration over multiple sparse tensors, where the intersection (for multiply) or the union (for addition) of the nonzero locations must be identified. The user specifies the layout along with the computation in Einstein notation, and the compiler generates the code for the input with the specified layout. To improve the performance and take advantage of the optimized data layouts, code transformations were later enabled in TACO through a scheduling language [44]. Where logical indices may not have corresponding physical entries, the inverse mapping from physical to logical indices can be used to find corresponding elements in other tensors during co-iteration, as done in dlcomp [45].

Data layout for performance portability: Data layouts such as Kokkos View [15], and `std::mdspan` in C++ 2023, abstract away the underlying data organization in memory for performance portability. Their underlying data layouts exploit the hierarchical nature of GPUs and CPU/GPU systems. In structured grid computations, fine-grained data blocking, where logically adjacent three-dimensional subdomains are stored in contiguous memory, have been shown to significantly reduce data movement [13, 12, 14]. TiDA [16, 46] uses coarse-grained data blocking, where the entire grid is tiled into sub-grids, each with its own ghost zone.

Data layout and thread layout applied to tensors: As previously noted, LEGO is most closely connected to the approaches of Graphene [22] and Triton [23, 24], which are focused on utilizing tensor cores. Graphene uses the same layout specification for data layout/movement and thread/block layout, representing general strided rectangular regions, as in Figure 8. As shown in Figure 1, Triton's layout specification provides a slice of each tensor, but requires explicit stride calculations. Very recently, linear layouts have been introduced; these are internal to the Triton compiler and not exposed at the source code level [24]. The layout description eliminates the need for explicit strides, but instead uses a linear algebra formulation where a layout is described with a binary matrix. Moreover, linear layout does not support user-defined bijective layouts that are nonlinear. At the application level, a tensor library called einops [47] exposes a tensor notation to describe tensor structure, facilitating the integration of tensor libraries.

Distributed Data Layouts: Common array layouts, such as tiled, row- and column-major, have been supported for a long time as directives in languages for distributed programming, such as High-Performance Fortran [48]. ZPL [49] separates the definition of the hardware abstraction from the manner in which data is mapped to the hardware, and Sequoia [50] and Legion [18] build on this idea to support, for example, (1) hierarchical definition of the hardware, (2) efficient data movement through memory hierarchy, (3) overlapped partitioning of data, (4) control over placement of data and computation, (5) support for accelerators, and (6) overlapping communication and computation. Finally, various DSLs, such as DISTAL [17] and SpDISTAL [51], use the Legion runtime system to implement sparse and dense tensor algebras, which allow user specification of communication patterns and of the data layout at per-node and across-nodes level, by means of scheduling languages.

Array Dependence Analyses: A rich body of work has used layouts of some sort or another in the quest of optimizing affine and non-affine programs. For non-affine programs, such as molecular-dynamics simulations, inspector-executor techniques have been devised to reorder the data and iteration space at runtime [52, 53], in a way that optimizes temporal and spatial locality. For example, the inspector code computes a *permutation* of the data/iterations that is used by the statically-generated executor code.

Work on automatic parallelization of non-affine loops [54, 55] tests at runtime sufficient conditions for statically irreducible queries that model loop independence. These can be represented as predicated extensions of polyhedral systems [6, 56] or as languages [57, 58] that build on linear-memory access descriptors (LMADs).

LMADs [59, 60] generalize Python-like slicing by allowing a global-memory offset together with a list that pairs up the length of each logical dimension with its total stride—i.e., the number of memory elements that are jumped to advance to the next element in that dimension, similar to Graphene.

LMADs have also been used in Futhark [61] to support various optimizations that are not expressible in a pure IR, and more relevant, to allow change-of-layout transformations to be applied on arrays at $O(1)$ cost, i.e., without manifestation in memory. While Figure 3 of [61] hints that any (straight-line) reordering sequence can be modeled by a chain of LMADs, subsequent work [62], presenting the memory lowering, clarifies that Futhark supports at $O(1)$ cost only reorderings that are expressible by *one* LMAD.

In comparison, LEGO supports reordering chains that may require several LMADs, such as `B` (O_2) in Figure 6, and non-linear (user-defined) patterns, e.g., `C` (O_1).

VII. Conclusion

This paper has described LEGO, a layout algebra to support tiled, hierarchical high-performance code generation. The key advance in LEGO is that it eliminates the need to specify strides in hierarchical layouts, thus simplifying layout specification. It is also a standalone Python code that can provide a bijective mapping of computation and data layout to/from program index space, thus eliminating the need for programmers to derive complex indices manually. It facilitates exploration of layouts in combination with other optimizations. We have demonstrated LEGO's integration with CUDA templates, the Triton and MLIR compilers, and its role in generating high-performance implementations.

LEGO's support extends beyond the strided, rectangular layouts of the CuTe/Graphene shape algebra, enabling arbitrary permutations of elements, such as the anti-diagonal example and the brick data layout presented in this paper. Layout composition can be used to express data movement to and from GPU shared memory, while also supporting optimizations such as thread coarsening and the reduction of shared memory bank conflicts. As future work, we plan to further explore the full range of layouts and integration with other systems. Related supplementary materials are available in the repository [63].

Acknowledgment

We thank John Regehr and David Yue for their feedback on the manuscript. We are especially grateful to the anonymous reviewers for their careful reading, detailed comments, and insightful suggestions, which significantly improved the clarity and quality of this work. The support and resources from the Center for High-Performance Computing at the University of Utah are gratefully acknowledged. Research reported in this publication was supported by the National Science Foundation through award CCF-2107556, and Novo Nordisk Foundation through award NNF24OC0090447.

Appendix

A. Abstract

This artifact contains the source code of the LEGO framework and the scripts used to execute and evaluate all benchmarks in the paper. LEGO provides an algebraic, compiler-agnostic framework for specifying and transforming memory layouts. Through integrations with Triton, CUDA, and MLIR, we compare LEGO-generated kernels with existing implementations and demonstrate that careful data layout reorganization can achieve state-of-the-art performance or significantly improve performance.

B. Artifact check-list (meta-information)

- **Data set:** provided as an artifact
- **Hardware:** NVIDIA A100 80GB GPU and AMD EPYC 7513 CPU
- **Output:** Benchmark figures and throughput table
- **How much disk space required (approximately)?:** 40GB
- **How much time is needed to prepare workflow (approximately)?:** 2 hours
- **How much time is needed to complete experiments (approximately)?:** 1.5 hours
- **Publicly available?:** Yes
- **Code licenses (if publicly available)?:** MIT
- **Archived (provide DOI)?:** 10.5281/zenodo.17633994

C. Description

1) How delivered: 10.5281/zenodo.17633994

2) Hardware dependencies: NVIDIA A100 80GB GPU

3) Software dependencies: LLVM (commit `556ec4a`), Triton 3.2.0, PyTorch 2.5.1, CUDA 12.4, Python 3.12.4, Ninja 1.12.1, CMake 3.26.5, GCC 11.2.0

D. Installation

1) **Clone the LLVM**
```
$ git clone https://github.com/llvm/llvm-project.git
$ cd llvm-project && git checkout 556ec4a
```
2) **Build and install LLVM/MLIR at the required commit**
```
$ mkdir build && cd build
$ cmake -G Ninja ../llvm -DCMAKE_BUILD_TYPE=Release \
  -DLLVM_ENABLE_PROJECTS="mlir" \
  -DLLVM_TARGETS_TO_BUILD="X86;NVPTX" \
  -DMLIR_ENABLE_CUDA_RUNNER=ON \
  -DMLIR_ENABLE_BINDINGS_PYTHON=ON \
  -DPython3_EXECUTABLE="$(which python)" \
  -DLLVM_BUILD_EXAMPLES=OFF && ninja
```
3) **Set the path to LLVM/MLIR build path**
```
$ export MLIR_BUILD_FOLDER="$(pwd)"
```

E. Experiment workflow

From the root of the artifact repository, the experiments can be reproduced with the following steps:

1) **Create the virtual environment and install Python packages:**
```
$ bash ./setup.sh && source venv/bin/activate
```
2) **Generate all kernel source code:**
```
$ bash ./gen_all_kernel.sh
```
3) **Run all benchmarks and produce figures and tables:**
```
$ bash ./run_all_kernels.sh
```
This will execute all benchmarks and generate the figures and tables reported in the paper.

F. Evaluation and expected result

The generated figures for the evaluation section (11, 12, 13) and Table V are located in the `./figures` folder in the root of the artifact directory.

References

[1] C. E. Leiserson, N. C. Thompson, J. S. Emer, B. C. Kuszmaul, B. W. Lampson, D. Sanchez, and T. B. Schardl, "There's plenty of room at the top: What will drive computer performance after moore's law?" *Science*, vol. 368, no. 6495, p. eaam9744, 2020. [Online]. Available: https://www.science.org/doi/abs/10.1126/science.aam9744

[2] P. Kogge and J. Shalf, "Exascale computing trends: Adjusting to the "new normal"' for computer architecture," *Computing in Science & Engineering*, vol. 15, no. 6, pp. 16–26, 2013.

[3] M. Wolfe, "More iteration space tiling," in *Proceedings of the 1989 ACM/IEEE Conference on Supercomputing*, ser. Supercomputing '89. New York, NY, USA: Association for Computing Machinery, 1989, p. 655–664. [Online]. Available: https://doi.org/10.1145/76263.76337

[4] S. Carr and K. Kennedy, "Improving the ratio of memory operations to floating-point operations in loops," *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 6, p. 1768–1810, Nov. 1994. [Online]. Available: https://doi.org/10.1145/197320.197366

[5] P. Feautrier, "Some efficient solutions to the affine scheduling problem. part ii. multidimensional time," *Int J Parallel Prog*, vol. 21, p. 389–420, 1992.

[6] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "A practical automatic polyhedral parallelizer and locality optimizer," *SIGPLAN Not.*, vol. 43, no. 6, p. 101–113, Jun. 2008. [Online]. Available: https://doi.org/10.1145/1379022.1375595

[7] S. Verdoolaege, J. Carlos Juega, A. Cohen, J. Ignacio Gómez, C. Tenllado, and F. Catthoor, "Polyhedral parallel code generation for cuda," *ACM Trans. Archit. Code Optim.*, vol. 9, no. 4, Jan. 2013. [Online]. Available: https://doi.org/10.1145/2400682.2400713

[8] W. S. Moses, L. Chelini, R. Zhao, and O. Zinenko, "Polygeist: Raising c to polyhedral mlir," in *Proceedings of the 30th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '21. IEEE Press, 2024, p. 45–59. [Online]. Available: https://doi.org/10.1109/PACT52795.2021.00011

[9] A. Thangamani, V. Loechner, and S. Genaud, "A survey of general-purpose polyhedral compilers," *ACM Trans. Archit. Code Optim.*, vol. 21, no. 4, Nov. 2024. [Online]. Available: https://doi.org/10.1145/3674735

[10] D. S. Wise, J. D. Frens, Y. Gu, and G. A. Alexander, "Language support for morton-order matrices," *ACM Sigplan Notices*, vol. 36, no. 7, pp. 24–33, 2001.

[11] S. Chatterjee, V. V. Jain, A. R. Lebeck, S. Mundhra, and M. Thottethodi, "Nonlinear array layouts for hierarchical memory systems," in *Proceedings of the 13th international conference on Supercomputing*, 1999, pp. 444–453.

[12] T. Zhao, P. Basu, S. Williams, M. Hall, and H. Johansen, "Exploiting reuse and vectorization in blocked stencil computations on cpus and gpus," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3295500.3356210

[13] M. Araya-Polo, F. Rubio, R. de la Cruz, M. Hanzich, J. M. Cela, and D. P. Scarpazza, "3d seismic imaging through reverse-time migration on homogeneous and heterogeneous multi-core processors," *Sci. Program.*, vol. 17, no. 1-2, Jan. 2009.

[14] C. Yount, J. Tobin, A. Breuer, and A. Duran, "Yask—yet another stencil kernel: A framework for hpc stencil code-generation and tuning," in *2016 Sixth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing (WOLFHPC)*, 2016.

[15] Kokkos, "Kokkos view multidimensional arrays," 2024. [Online]. Available: https://kokkos.org/kokkos-core-wiki/ProgrammingGuide/View.html

[16] D. Unat, T. Nguyen, W. Zhang, M. N. Farooqi, B. Bastem, G. Michelogiannakis, A. Almgren, and J. Shalf, "Tida: High-level programming abstractions for data locality management," in *International Conference on High Performance Computing*. Springer, 2016, pp. 116–135.

[17] R. Yadav, A. Aiken, and F. Kjolstad, "Distal: the distributed tensor algebra compiler," in *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, ser. PLDI 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 286–300. [Online]. Available: https://doi.org/10.1145/3519939.3523437

[18] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Legion: Expressing locality and independence with logical regions," in *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2012, pp. 1–11.

[19] M. Lakshminarasimhan, O. Antepara, T. Zhao, B. Sepanski, P. Basu, H. Johansen, M. Hall, and S. Williams, "Bricks: A high-performance portability layer for computations on block-structured grids," *The International Journal of High Performance Computing Applications*, vol. 38, no. 6, pp. 549–567, 2024. [Online]. Available: https://doi.org/10.1177/10943420241268288

[20] B. Hagedorn, A. S. Elliott, H. Barthels, R. Bodik, and V. Grover, "Fireiron: A data-movement-aware scheduling language for gpus," in *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 71–82. [Online]. Available: https://doi.org/10.1145/3410463.3414632

[21] V. Thakkar, P. Ramani, C. Cecka, A. Shivam, H. Lu, E. Yan, J. Kosaian, M. Hoemmen, H. Wu, A. Kerr, M. Nicely, D. Merrill, D. Blasig, F. Qiao, P. Majcher, P. Springer, M. Hohnerbach, J. Wang, and M. Gupta, "Cutlass," https://github.com/NVIDIA/cutlass, 2023, version 3.0.0. License: BSD-3-Clause. [Online]. Available: https://github.com/NVIDIA/cutlass/tree/v3.0.0

[22] B. Hagedorn, B. Fan, H. Chen, C. Cecka, M. Garland, and V. Grover, "Graphene: An ir for optimized tensor computations on gpus," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, 2023, pp. 302–313.

[23] P. Tillet, H. T. Kung, and D. Cox, "Triton: an intermediate language and compiler for tiled neural network computations," in *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, ser. MAPL 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 10–19. [Online]. Available: https://doi.org/10.1145/3315508.3329973

[24] K. Zhou, M. Lezcano, A. Goucher, A. Rakhmati, J. Niu, J. Lebar, P. Szczerbuk, P. Bell, P. Tillet, T. Raoux, and Z. Moudallal, "Linear layouts: Robust code generation of efficient tensor computation using $\mathbb{F}_2$," 2025. [Online]. Available: https://arxiv.org/abs/2505.23819

[25] T. A. Mogensen, *Introduction to Compiler Design*, 1st ed. Springer Publishing Company, Incorporated, 2011.

[26] Pallets, "Jinja: A very fast and expressive template engine." https://github.com/pallets/jinja, accessed: 2025-04-13.

[27] A. Meurer, C. P. Smith, M. Paprocki, O. Čertík, S. B. Kirpichev, M. Rocklin, A. Kumar, S. Ivanov, J. K. Moore, S. Singh, T. Rathnayake, S. Vig, B. E. Granger, R. P. Muller, F. Bonazzi, H. Gupta, S. Vats, F. Johansson, F. Pedregosa, M. J. Curry, A. R. Terrel, v. Roučka, A. Saboo, I. Fernando, S. Kulal, R. Cimrman, and A. Scopatz, "Sympy: symbolic computing in python," *PeerJ Computer Science*, vol. 3, p. e103, Jan. 2017. [Online]. Available: https://doi.org/10.7717/peerj-cs.103

[28] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[29] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, "Array programming with NumPy," *Nature*, vol. 585, no. 7825, pp. 357–362, Sep. 2020. [Online]. Available: https://doi.org/10.1038/s41586-020-2649-2

[30] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *2009 IEEE International Symposium on Workload Characterization (IISWC)*, 2009, pp. 44–54.

[31] P. Barua, J. Shirako, and V. Sarkar, "Cost-driven thread coarsening for gpu kernels," in *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3243176.3243196

[32] I. R. Ivanov, O. Zinenko, J. Domke, T. Endo, and W. S. Moses, "Retargeting and respecializing gpu workloads for performance portability," in *Proceedings of the 2024 IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '24. IEEE Press, 2024, p. 119–132. [Online]. Available: https://doi.org/10.1109/CGO57630.2024.10444828

[33] T. Zhao, S. Williams, M. Hall, and H. Johansen, "Delivering performance-portable stencil computations on cpus and gpus using bricks," in *2018 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. IEEE, 2018, pp. 59–70.

[34] O. Temam, E. D. Granston, and W. Jalby, "To copy or not to copy: a compile-time technique for assessing when data copying should be used to eliminate cache conflicts," in *Procs. of ACM/IEEE Conf. on Supercomputing*, ser. Supercomputing'93. ACM, 1993, p. 410–419.

[35] M. M. Baskaran, U. Bondhugula, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan, "Automatic data movement and computation mapping for multi-level parallel architectures with explicitly managed memories," in *Princ. Pract. of Par. Prog.*, 2008, p. 1–10.

[36] M. Khan, P. Basu, G. Rudy, M. Hall, C. Chen, and J. Chame, "A script-based autotuning compiler system to generate high-performance cuda code," *ACM Trans. Archit. Code Optim. (TACO)*, vol. 9, no. 4, 2013.

[37] B. Hagedorn, A. Elliott, H. Barthels, R. Bodik, and V. Grover, "Fireiron: a data-movement-aware scheduling language for gpus," in *Procs. of Int. Conf. on Par. Arch. and Compilation Tech. (PACT)*, 2020.

[38] A. Rasch, R. Schulze, D. Shabalin, A. C. Elster, S. Gorlatch, and M. Hall, "(de/re)-compositions expressed systematically via mdh-based schedules," in *Int. Conf. on Comp. Constr. (CC)*. ACM, 2023, p. 61–72.

[39] M. Strout, A. LaMielle, L. Carter, J. Ferrante, B. Kreaseck, and C. Olschanowsky, "An approach for code generation in the sparse polyhedral framework," *Parallel Computing*, vol. 53, 2016.

[40] A. Venkat, M. Shantharam, M. Hall, and M. Strout, "Non-affine extensions to polyhedral code generation," in *IEEE/ACM Int. Symp. on Code Gen. and Opt. (CGO)*, 2014.

[41] A. Venkat, M. Hall, and M. Strout, "Loop and data transforma-

tions for sparse matrix code," in *ACM SIGPLAN Conf. on Prog. Lang. Design and Impl. (PLDI)*, 2015.

[42] A. Venkat, M. S. Mohammadi, J. Park, H. Rong, R. Barik, M. M. Strout, and M. Hall, "Automating wavefront parallelization for sparse matrix computations," in *Procs. Int. Conf. on High Perf. Comp., Networking, Storage and Analysis*, ser. SC '16. IEEE Press, 2016.

[43] F. Kjolstad, S. Kamil, S. Chou, D. Lugato, and S. Amarasinghe, "The tensor algebra compiler," *Proc. of the ACM on Prog. Lang.*, vol. 1, no. OOPSLA, pp. 1–29, 2017.

[44] R. Senanayake, C. Hong, Z. Wang, A. Wilson, S. Chou, S. Kamil, S. Amarasinghe, and F. Kjolstad, "A sparse iteration space transformation framework for sparse tensor algebra," *Proc. of the ACM on Prog. Lang.*, vol. 4, no. OOPSLA, pp. 1–30, 2020.

[45] T. Zhao, T. Popoola, M. Hall, C. Olschanowsky, and M. Strout, "Polyhedral specification and code generation of sparse tensor contraction with co-iteration," *ACM Trans. Archit. Code Optim. (TACO)*, vol. 20, no. 1, dec 2022.

[46] D. Unat, A. Dubey, T. Hoefler, J. Shalf, M. Abraham, M. Bianco, B. L. Chamberlain, R. Cledat, H. C. Edwards, H. Finkel, K. Fuerlinger, F. Hannig, E. Jeannot, A. Kamil, J. Keasler, P. H. J. Kelly, V. Leung, H. Ltaief, N. Maruyama, C. J. Newburn, and M. Pericás, "Trends in data locality abstractions for hpc systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 10, 2017.

[47] A. Rogozhnikov, "Einops: Clear and reliable tensor manipulations with einstein-like notation," 2022, accepted for oral presentation at ICLR'22, https://openreview.net/pdf?id=oapKSVM2bcj.

[48] D. B. Loveman, "High performance fortran," *IEEE Parallel Distrib. Technol.*, vol. 1, no. 1, p. 25–42, Feb. 1993. [Online]. Available: https://doi.org/10.1109/88.219857

[49] S. Deitz, B. Chamberlain, and L. Snyder, "Abstractions for dynamic data distribution," in *Ninth International Workshop on High-Level Parallel Programming Models and Supportive Environments, 2004. Proceedings.*, 2004, pp. 42–51.

[50] K. Fatahalian, D. R. Horn, T. J. Knight, L. Leem, M. Houston, J. Y. Park, M. Erez, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan, "Sequoia: programming the memory hierarchy," in *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, ser. SC '06. New York, NY, USA: Association for Computing Machinery, 2006, p. 83–es. [Online]. Available: https://doi.org/10.1145/1188455.1188543

[51] R. Yadav, A. Aiken, and F. Kjolstad, "Spdistal: compiling distributed sparse tensor computations," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '22. IEEE Press, 2022.

[52] C. Ding and K. Kennedy, "Improving cache performance in dynamic applications through data and computation reorganization at run time," in *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, ser. PLDI '99. New York, NY, USA: Association for Computing Machinery, 1999, p. 229–241. [Online]. Available: https://doi.org/10.1145/301618.301670

[53] M. M. Strout and P. D. Hovland, "Metrics and models for reordering transformations," in *Proceedings of the 2004 Workshop on Memory System Performance*, ser. MSP '04. New York, NY, USA: Association for Computing Machinery, 2004, p. 23–34. [Online]. Available: https://doi.org/10.1145/1065895.1065899

[54] S. Moon and M. W. Hall, "Evaluation of predicated array data-flow analysis for automatic parallelization," in *Proceedings of the Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '99. New York, NY, USA: Association for Computing Machinery, 1999, p. 84–95. [Online]. Available: https://doi.org/10.1145/301104.301112

[55] C. E. Oancea and A. Mycroft, *Set-Congruence Dynamic Analysis for Thread-Level Speculation (TLS)*. Berlin, Heidelberg: Springer-Verlag, 2008, p. 156–171. [Online]. Available: https://doi.org/10.1007/978-3-540-89740-8_11

[56] M. W. Hall, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, and M. S. Lam, "Interprocedural Parallelization Analysis in SUIF," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 27(4), pp. 662–731, 2005.

[57] S. Rus, L. Rauchwerger, and J. Hoeflinger, "Hybrid analysis: static & dynamic memory reference analysis," in *Proceedings of the 16th International Conference on Supercomputing*, ser. ICS '02. New York, NY, USA: Association for Computing Machinery, 2002, p. 274–284. [Online]. Available: https://doi.org/10.1145/514191.514229

[58] C. E. Oancea and L. Rauchwerger, "Scalable conditional induction variables (civ) analysis," in *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 213–224. [Online]. Available: http://dl.acm.org/citation.cfm?id=2738600.2738627

[59] Y. Lin and D. A. Padua, "Demand-driven interprocedural array property analysis," in *Proceedings of the 12th International Workshop on Languages and Compilers for Parallel Computing*, ser. LCPC '99. Berlin, Heidelberg: Springer-Verlag, 1999, p. 303–317.

[60] Y. Paek, J. Hoeflinger, and D. Padua, "Efficient and precise array access analysis," *ACM Trans. Program. Lang. Syst.*, vol. 24, no. 1, p. 65–109, Jan. 2002. [Online]. Available: https://doi.org/10.1145/509705.509708

[61] P. Munksgaard, T. Henriksen, P. Sadayappan, and C. Oancea, "Memory optimizations in an array language," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '22. IEEE Press, 2022.

[62] P. Munksgaard, C. Oancea, and T. Henriksen, "Compiling a functional array language with non-semantic memory information," in *Proceedings of the 34th Symposium on Implementation and Application of Functional Languages*, ser. IFL '22. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3587216.3587218

[63] A. M. Tavakkoli, C. Oancea, and M. Hall, "Lego: A layout expression language for code generation of hierarchical mapping - artifact," Nov. 2025. [Online]. Available: https://doi.org/10.5281/zenodo.17633994