Structured Operations: Modular Design of Code Generators for Tensor Compilers

Nicolas Vasilache[0000−0002−4096−3325], Oleksandr Zinenko[0000−0003−1978−0222], Aart J.C. Bik[0000−0002−0333−7413], Mahesh Ravishankar, Thomas Raoux, Alexander Belyaev, Matthias Springer[0000−0001−5077−7131], Tobias Gysi, Diego Caballero, Stephan Herhut, Stella Laurenzo, and Albert Cohen[0000−0002−8866−5343]

Google

Abstract. The performance of machine learning systems heavily relies on code generators tailored to tensor computations. We propose an approach to the design and implementation of such code generators leveraging the natural structure of tensor algebra and illustrating the progressive lowering of domain-specific abstractions in the MLIR infrastructure.

1 Introduction

This article tackles the design and implementation code generators for *portable* and *performance-portable* tensor operations in Machine Learning (ML) frameworks.[1] As of today, ML frameworks have to make an exclusive choice among tensor compilers such as XLA [22], Glow [15], TVM [5], Triton [19], TACO [10], polyhedral compilers [21]. All of these eventually produce some flavor of LLVM IR, but they come with incompatible abstractions and implementations, do not compose, and have complex front-end/back-end compatibility matrices. We propose a portable set of abstractions, composition rules and refinements to break out of this silo-ed world.

We leverage MLIR, a compiler infrastructure that drastically reduces the entry cost to define and introduce new abstraction levels for building domain-specific Intermediate Representations (IRs) [11]. It is part of the LLVM project and follows decades of established practices in production compiler construction. Yet, while MLIR provides much needed infrastructure, the problem remains of defining portable intermediate abstractions and a progressive refinement strategy for tensor compilers. Our strategy involves alternating cycles of top-down and bottom-up thinking: (1) top-down relates to making primitives available to the programmer that *gradually decompose* into smaller building blocks with unsurprisingly good performance; while (2) bottom-up is concerned with the creation of building blocks that are well-suited to each hardware architecture and their *gradual composition*, connecting to top-down thinking.

[1] This version is a preprint of a paper with the same title published in the proceedings of LCPC 2022 `https://doi.org/10.1007/978-3-031-31445-2_10`.

The techniques and abstractions described in this paper are used at Google for CPU and GPU code generation, including XLA-based flows [22] and IREE [9] on mobile and edge devices. The extended version of the paper [20] provides additional examples and details the design and implementation.

2 Overview of the Code Generation Flow

MLIR reduces the cost to define, compose and reuse abstractions for the construction of domain-specific compilers. It offers a comprehensive collection of compiler construction solutions by: (1) standardizing Static Single Assignment (SSA) form representations and data structures, (2) unifying compiler analyses and transformations across semantic domains through generic programming concepts such as operation traits and interfaces, (3) providing a declarative system for operations with nested regions and domain-specific type systems, and (4) providing a wide range of services including documentation, parsing/printing logic, location tracking, multithreaded compilation, pass management, etc.

MLIR is designed around the principles of parsimony, progressivity and traceability [11]. The code generation approach presented in this paper has largely contributed to the establishment of these principles and actively leverages them. The Internal Representation (IR) is fully extensible, allowing for user-defined operations (instructions), attributes and types. IR components that are expected to work together are grouped into *dialects*, which can be seen as the IR analog of dynamic libraries. Unlike earlier compilation flows offering a multi-level IR, MLIR affords and encourages the mix of different dialects in a single unit of compilation at any point in the compilation flow. For example, a high-level tensor operation may co-exist with low-level hardware instructions on vector elements in the same function. This provides a great level of modularity, composition and optionality: different abstractions can be assembled into solving a particular problem, instead of having to solve all problems in a unique representation.

2.1 Structured Operations

Optimizations for numerical computing have traditionally focused on loop nests. The associated analyses consider memory dependences on individual array elements and aliasing [1]. They are well-suited when starting from an input language like C or Fortran where the problem is already specified in terms of loops over data residing in pre-allocated memory. When focusing on a specific domain such as Machine Learning (ML), we have the luxury of programs defined at a much higher level of abstraction than loops. This allows to revisit classical loop optimizations like fusion, tiling or vectorization, operating at the highest possible level of abstraction. It eliminates the need for complex analyses retrieving information from lower-level code. Advantages include reduced complexity and maintenance cost while also enabling support for sparse storage and computation that is impractical to model at the loop level. We refer to this approach as *structured code generation* since the compiler primarily leverages structural

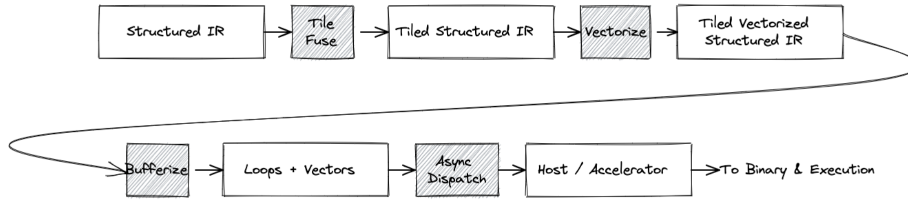

Fig. 1. Bird's eye view of structured code generation.

Fig. 2. Visual description of a typical MLIR compiler flow.

```
%value_definition = "dialect.operation"(%value_use) {attribute_name = #attr_kind<"value">} ({
// Regions contain blocks.
^block(%block_argument: !argument_type):
  "dialect.further_operation"()[^successor] : () -> ()
^successor: // more operations below
}) : (!operand_type) -> !result_type<"may_be_parameterized">
```

Fig. 3. MLIR concepts in the generic syntax.

information readily available in the source code, and *structured operations* refer to the operations amenable to structured code generation.

Figure 1 depicts the coarse-grained steps and levels of abstraction involved in a structured code generation flow for a typical tensor compiler. The starting point (Structured IR) is composed of tensor algebra operations, organized as a functional program over dense and sparse tensors. From this level we move to a tiled structured level, which introduces loops by gradually tiling structured operations into ones operating on smaller tensors. One also perform fusion of tensor operations at this level. The final granularity of operations is chosen to make their hardware mapping efficient.

A typical example is to tile matrix multiplication according to a cache hierarchy, lowering block-wise matrix multiplications on sub-tensors to a retargetable vector abstraction (possibly involving padding or vector masking), and eventually targeting a super-optimized microkernel in assembly language. Such a tiling and refinement flow is made possible through the preservation of high-level knowledge about the operations.

What makes *structured code generation* highly composable and reusable is that tiling and fusion are both fully generic in the operations and data types they operate upon. These transformations only assume a monotonic (regarding set inclusion), structural decomposition pattern associated with computations and composite data. Both dense and sparse tensor algebra exhibit such block-wise decomposition patterns, and the code generation abstractions and infrastructure generically applies to both.

Up until now, computations took place on immutable tensor values. The next step lowers this to a representation on side-effecting buffers and nested loops. More optimizations on loops and memory accesses happen at this level, leveraging existing affine analyses and loop optimizations as implemented in MLIR, and that have been largely explored in the literature.

In the final step, one may translate the representation directly to the `llvm` dialect of MLIR for sequential execution on CPU, or offload a GPU kernel, or split up loops into `async` blocks for a task parallel runtime, etc.

While this flow is but a first stake in the ground, it already demonstrates how to achieve a modular and composable system, following a progressive lowering principle. Every step is materialized in the IR and very little load-bearing logic is hidden in the form of complex static analyses and heuristics in C++. It is designed with optionality in mind: more operations and data types can be implemented that do not fit the current code generation stack of abstractions.

2.2 Structured Code Generation in MLIR

The MLIR infrastructure builds on the success of LLVM IR while providing unprecedented extensibility. MLIR has an open, easily extensible set of instructions, called *operations* that typically represent the dynamic semantics of the program. Operations can represent anything from hardware instructions, or even hardware itself, to building blocks for machine learning models such as layers or blocks thereof. They define and use values, which represent units of immutable data in SSA form. The compile-time knowledge about values is captured in *types*, and the knowledge about operations is captured in *attributes*. Attribute and type systems are similarly open and extensible. IR objects can be logically grouped together in libraries, called *dialects*. The MLIR IR has a recursive structure where operations may have additional *regions* containing a graph of (basic) *blocks*, which in turn contain further operations. Figure 3 illustrates key MLIR concepts: the IR has an open set of attributes, operations and types; operations may recursively contain regions of blocks holding operations themselves.

In addition to common components such as the compiler pass infrastructure, MLIR provides tools to manage its extensibility, many of which evolved or were specifically designed to support the code generation flow presented in this document. In particular, MLIR features attribute, operation and type *interfaces* similar to object-oriented programming languages allowing one to work with abstract properties rather than (fixed) lists of supported concepts. Interfaces can be implemented separately from operations, and mixed in using MLIR's registration mechanism, thus fully separating IR concepts from transformations.

Let us now review the dialects we defined, listed in increasing level of abstraction. Any of these dialects can be mixed with others or simply bypassed if it does not provide a useful abstraction for a particular case.

The `vector` dialect provides a fixed-rank, static shape, *n-D* vector type, such as `vector<4x3x8xf32>`, as wel operations that conceptually extend traditional 1-D vector instructions to arbitrary rank. Such operations decompose progressively into lower-rank variants, and eventually lower to LLVM vector instructions.

The `gpu` dialect defines a retargetable *GPU programming model*. It features abstractions common to SIMT platforms, such as host/device code separation, a workitem/group (thread/block) execution model, communication and synchronization primitives, etc. This dialect can be produced from the `vector` dialect and can be lowered to platform-specific dialects such as `nvvm`, `rocdl` or `spirv`.

The `memref` dialect introduces the `memref` data type, which is the main representation for *n-D* memory buffers in MLIR, the entry point to the side-effecting memory-based operations, the way to interoperate with external C code. The dialect also provides the operations to manage buffer allocation, aliasing (memref *views*) and access. Unlike traditional pointers, `memref`s are multi-dimensional buffers with explicit layout that allows for decoupling the indexing scheme from the underlying storage: `memref<10x10xf32, strides: [1,10]>` affords column-major access while having row-major storage.

The `tensor` dialect operates on an abstract n-D tensor type with no specified representation in memory. Later in the compilation flow, sufficiently small tensors of static shape may be placed directly in (vector) registers while larger or dynamically-sized tensors are put into memory storage thanks to the bufferization process. Tensor values are *immutable* and subject to SSA semantics. Operations on tensors are generally free of side-effects. This allows classical compiler transformations such as peephole optimizations, constant sub-expression and dead code elimination, or loop-invariant code motion to apply seamlessly to tensor operations regardless of their underlying complexity. *Since tensor values are immutable, they cannot be written into. Instead, "value insertion" operations create new tensors with a value or a subset thereof replaced.* [2]

The `scf` or *structured control flow* dialect provides operations that represent looping and conditionals (e.g. regular `scf.for` and `scf.while` loops without early exit as well as an `scf.if` conditional construct) and *embeds them into the SSA+regions form* of MLIR. This is structured at a higher-level of abstraction than a control flow graph. Notably, `scf` loop operations may yield SSA values and compose with other operations and dialects with either side-effecting or value-based semantics.

The `linalg` dialect provides higher-level compute primitives that operate on both multiple containers, including `tensor` and `memref`. These primitives can decompose into versions of themselves operating on structured *subsets* of the original input data and producing similarly structured subsets of their results. They also capture program invariants and structural information, such as reduction patterns or the independence of certain parts of the computation.

The `sparse_tensor` dialect provides the types and transformations required to make sparse tensor types first-class citizens within the MLIR compiler infrastructure [2]. It bridges high-level `linalg` operations on sparse tensors with lower-level operations on the actual sparse storage schemes that save memory and avoid performing redundant work.

---

[2] This is analogous to the design of `struct` in LLVM IR: `%1 = insertvalue {f64, f32, i32} %0, f32 42.0, 1` defines a new value `%1` that holds the same elements as `%0` except for the element at position `1` that now holds `42.0`.

Figure 2 summarizes a typical flow. At the end of the transformation process, MLIR produces low-level dialects common to multiple compilation paths. The `llvm` dialect closely mirrors LLVM IR and can be *translated* to LLVM IR before being handed off to the LLVM compiler to produce machine code. This dialect reuses built-in MLIR types such as `i32` or `f32` scalars. To support performance-critical scenarios involving specific hardware instructions, it can also be mixed with low-level platform-specific dialects: `nvvm`, `rocdl`, `x86vector`, `arm_neon`, `arm_sve`, `amx`, etc. These dialects partly mirror the corresponding sets of LLVM IR intrinsic functions. Beyond making these instructions first-class operations, these dialects also provide higher-level abstractions that make use of MLIR's extensible type system and interfaces. For example, the

```
arm_neon.2d.sdot : vector<4x4xi8>, vector<4x4xi8> to vector<4xi32>
```

operation is naturally expressed on a MLIR multidimensional vector type. Before converting to LLVM IR, it is first lowered to

```
arm_neon.intr.sdot : vector<16xi8>, vector<16xi8> to vector<4xi32>
```

that operates on flattened 1-D vectors to match LLVM's convention. The full example is provided in the extended version of the paper [20].

3 Transformations

Let us illustrate important transformations available on structured operations. We first consider a 1-D convolution named `linalg.conv_1d_nwc_wcf` and its lowering to a tiled, padded and vectorized form. The highest level input is shown in Figure 4 (left). It operates on SSA values (immutable), with layout information as annotations for a future bufferization step (more on this later).

The operation is fully defined by the following expression, indexed over a 5-D rectangular iteration domain, and using/defining 3-D tensors:[3]

$$O[n, w, f] = I[n, w + k_w, c].K[k_w, c, f]$$

The iteration domain is implicit in the operation expression: it is such that iterators span the entire operands' shape. In the example, this yields the inequalities

$$0 \leq n < O.0, \quad 0 \leq w < O.1, \quad 0 \leq f < O.2, \quad 0 \leq k_w < K.0, \quad 0 \leq c < K.1$$

where $O.d$ denotes the size of the d-th dimension of O. The derivation for these quantities follows the same rules as Tensor Comprehensions [21]. They can be derived with successive applications of Fourier-Motzkin elimination [16].

3.1 Tiling

We only describe the simplest form of tiling. Other tiling variants, how to generate parallel SPMD code, various forms of fusion can be found in the MLIR

[3] The operation also allows specifying sizes and strides, omitted for simplicity.

```
#id3d = affine_map<(d0,d1,d2)->(d0,d1,d2)>
{linalg.buffer_layout = #id3d,
 linalg.inplaceable = false}

{linalg.buffer_layout = #id3d,
 linalg.inplaceable = true}

!tI = type tensor<1x990x32xf32>
!tK = type tensor<3x32x64xf32>
!tO = type tensor<1x988x64xf32>

func @conv_1d_nwc_wcf_main(%I: !tI #in, %K: !tK #in,
                           %0: !tO #out) -> !tO {
  %cst = arith.constant 0.000000e+00 : f32
  %0 = linalg.fill(%cst, %0) : f32, !tO -> !tO
  %1 = linalg.conv_1d_nwc_wcf
    ins(%I, %K : !tI, !tK) outs(%0 : !tO) -> !tO
  return %1 : !tO
}
```

tile[1,8,32,1,8]

```
!tDyn = type tensor<?x?x?xf32>
%1 = scf.for %n  = 0 to 1 step 1 iter_args(%O1 = %0) -> (!tO) {
  %2 = scf.for %w  = 0 to 988 step 8 iter_args(%O2 = %O1) -> (!tO) {
    %3 = scf.for %f  = 0 to 64 step 32 iter_args(%O3 = %O2) -> (!tO) {
      %4 = scf.for %kw = 0 to 3 step 1 iter_args(%O4 = %O3) -> (!tO) {
        %5 = scf.for %c  = 0 to 32 step 8 iter_args(%O5 = %O4) -> (!tO) {
          %8 = tensor.extract_slice %I[%n, %w + %kw, %c]
                [1, min(8, 990 - %w - %f), 8] [1, 1, 1] : !tI to !tDyn
          %9 = tensor.extract_slice %F[%kw, %c, %f]
                [1, 8, 32] [1, 1, 1] : !tK to !tDyn
          %11 = tensor.extract_slice %arg12[%n, %w, %f]
                 [1, min(8, 988 - %w), 32] [1, 1, 1] : !tO to !tDyn
          %12 = linalg.conv_1d_nwc_wcf
              ins(%8, %9 : !tDyn, !tDyn) outs(%11 : !tDyn) -> !tDyn
          %13 = tensor.insert_slice %12 into %O4[%n, %w, %f]
                 [1, min(8, 988 - %w), 32] [1, 1, 1] : !tDyn into !tO
          scf.yield %13 : !tO }
        scf.yield %5 : !tO }
      scf.yield %4 : !tO }
    scf.yield %3 : !tO }
  scf.yield %2 : !tO }
```

```
%c1 = arith.constant 1 : index

%6 = affine.apply affine_map
      <(d0,d1)->(d0+d1)>
      (%arg5, %arg9)

%7 = affine.min affine_map<
      (d0)->(8,990-d0)>(%6)
```

Fig. 4. Tiling a convolution introduces loops and induction variables. Parts in italic are simplified for clarity and expanded in callouts. New concepts underscored: (left) operations on immutable tensors, (right) induction variables and tensor slicing.

```
!tDyn = type tensor<?x?x?xf32>
%1 = scf.for %n  = 0 to 1 step 1 iter_args(%O1 = %0) -> (!tO) {
  %2 = scf.for %w  = 0 to 988 step 8 iter_args(%O2 = %O1) -> (!tO) {
    %3 = scf.for %f  = 0 to 64 step 32 iter_args(%O3 = %O2) -> (!tO) {
      %4 = scf.for %kw = 0 to 3 step 1 iter_args(%O4 = %O3) -> (!tO) {
        %5 = scf.for %c  = 0 to 32 step 8 iter_args(%O5 = %O4) -> (!tO) {
          %8 = tensor.extract_slice %I[%n,%w+%kw,%c]
                [1,min(8,990-%w-%f),8][1,1,1] : !tI to !tDyn
          %9 = tensor.extract_slice %F[%kw,%c,%f]
                [1,8,32][1,1,1] : !tK to !tDyn
          %11 = tensor.extract_slice %arg12[%n,%w,%f]
                 [1,min(8,988-%w),32][1,1,1] : !tO to !tDyn
          %12 = linalg.conv_1d_nwc_wcf
              ins(%8, %9 : !tDyn, !tDyn) outs(%11 : !tDyn) -> !tDyn
          %13 = tensor.insert_slice %12 into %O4[%n,%w,%f]
                 [1,min(8,988-%w),32][1,1,1] : !tDyn into !tO
          scf.yield %13 : !tO }
        scf.yield %5 : !tO }
      scf.yield %4 : !tO }
    scf.yield %3 : !tO }
  scf.yield %2 : !tO }
```

pad

```
!tISlice = type tensor<1x8x8xf32>
!tKFSlice = type tensor<1x8x32xf32>

%1 = scf.for %w = /*...*/ iter_args(%O1 = %0) -> (!tO) {
  %3 = linalg.init_tensor [3, 4, 1, 8, 8] : tensor<?x?x1x8x8xf32>
  %PI = scf.for %kw = /*...*/ iter_args(%I1 = %I) -> (tensor<?x?x1x8x8xf32>) {
    %8 = scf.for %c = /*...*/ iter_args(%I2 = %I1) /*...*/ {
      %11 = tensor.extract_slice %I[0, %w+%kw, %c][1,min(8,990-%w-%kw),8][1,1,1]
      %13 = linalg.pad_tensor %11 nofold low[0,0,0] high[0,8-min(8,990-%w-%kw),0] {
      ^bb0(/*tensor indices*/):
        linalg.yield 0.000000 : f32
      } : tensor<1x?x8xf32> to !tISlice
      %14 = tensor.insert_slice %13 into %I2 [1,%c ceildiv 8,0,0,0]
                                             [1,1,1,8,8][1,1,1,1,1]
      scf.yield %14 : tensor<?x?x1x8x8xf32> }
    scf.yield %8 : tensor<?x?x1x8x8xf32> }

  %5 = scf.for %f = /*...*/ iter_args(%O2 = %O1) -> (!tO) {
    %6 = scf.for %kw = /*...*/ iter_args(%O3 = %O2) -> (!tO) {
      %7 = scf.for %c = /*...*/ iter_args(%O4 = %O3) -> (!tO) {
        %8 = tensor.extract_slice %F[%kw,%c,%f][1,8,32] [1,1,1] /*...*/
        %9 = tensor.extract_slice %O4[0,%w,%f][1,min(8,988-%w),32][1,1,1] /*...*/
        %12 = tensor.extract_slice %PI[%kw,%c ceildiv 8,0,0,0][1,1,1,8,8][1,1,1,1,1]
        %13 = linalg.pad_tensor %8 nofold low[0,0,0] high[0,0,0] /*...*/
        %15 = linalg.pad_tensor %9 low[0,0,0] high[0, 8-min(8,988-%w),0] /*...*/
        %16 = linalg.conv_1d_nwc_wcf
            ins(%12, %13 : !tISlice, !tKFSlice) outs(%15 : !tKFSlice) -> !tKFSlice
        %17 = tensor.extract_slice %16[0,0,0][1,min(8,988-%w),32][1,1,1] /*...*/
        %18 = tensor.insert_slice %17 into %O4[0, %w, %f][1,min(8,988-%w),32][1,1,1]
        scf.yield %18 : !tO }
      scf.yield %7 : !tO }
    scf.yield %6 : !tO }
  scf.yield %5 : !tO }
```

Fig. 5. Padding a tiled operation to obtain fixed-size tensors (highlighted). Parts in italic are simplified for brevity. Constants in roman font are attributes, those in italic are `arith.constant` operation results.

documentation. In this simple form, tiling introduces `scf.for` loops as well as subset operations (`tensor.extract_slice` and `tensor.insert_slice`) to access tiled subsets, see Figure 4 (right). The tiled form of the operation is itself a `linalg.conv_1d_nwc_wcf` operating on the tiled subsets. The derivation of dense subsets is obtained by computing the image of the iteration domain by the indexing function for each tensor. Non-dense iteration domains and subsets involve dialect extensions and inspector-executor [10] code generation that are outside the scope of this paper.

Let us chose tile sizes `1x8x32x1x8`. Some of these do not divide tensor sizes; as a result the boundary tiles are subject to full/partial tile separation. There is no single static tensor type that is valid for every loop iteration; the tiled tensor type `!tDyn` must be relaxed to a dynamically shaped tensor, whose corresponding dynamic tile sizes are %8, %9 and %11. Later canonicalization steps kick in to refine the types that can be determined to be partially static. The resulting `scf.for` loops perform iterative yields of the full tensor value that is produced

at each iteration of the loop nest. Since tensor values are immutable, new values are produced by each `tensor.insert_slice` and `scf.yield`.

3.2 Padding and Packing

Dynamic tile shapes can hamper vectorization which requires static sizes. There are multiple mitigating options: multi-level loop peeling/versioning to isolate a statically known constant part, possibly combined with masked vector operations at domain boundaries; or padding to a larger static size (the padding value must be neutral for the consuming operation). In Figure 5, the `tensor.pad` operation deals with the latter option. Its size is obtained by subtracting the dynamic tile size from the static tile size. Elements in the padded region are set to the `%cst` value. A *nofold* attribute enforces additional padding to avoid cache line split.

Note that padding operations can often be hoisted out of tile loops, storing padded tiles in a packed, higher-dimensional tensor. This amortizes copying cost and makes tiles contiguous in memory (reducing TLB misses). The amount of hoisting is configurable per tensor, to trade memory consumption for copy benefits. In the example, input tensor padding is hoisted by 3 loops. This introduces an additional tile loop nest to precompute padded tiles and insert them into a packed tensor of type `tensor<?x?x1x8x8xf32>` containing all padded tiles. This also results in the actual computations accessing the packed tensor `%12= tensor.extract_slice %PI...`.

3.3 Vectorization

After tiling and padding, the convolution operands are statically shaped and amenable to vectorization, see Figure 6 (left). In the current IR, only 2 types of operations need to be vectorized: `tensor.pad` and `linalg.conv1d_nwc_wcf`. The vector dialect additionally provides a first-class representation for high-intensity operations. Figure 6 (right) illustrates one of these, `vector.contract`.

3.4 Bufferization

Bufferization is the process of materializing `tensor` values into (`memref`) buffers. It typically occurs late in the compilation flow. To achieve good performance, it is essential to allocate and copy as little memory as possible. As a result, buffers should be reused and updated in-place whenever possible.

Allocating a new buffer for every memory write is always safe, but wastes memory and introduces unnecessary copies. On the other hand, reusing a buffer and writing to it in-place can result in invalid bufferization if the original data at the overwritten memory location must be read at a later point of time. When performing transformations, one must be careful to preserve program semantics exposed by dependencies [1]. The right-hand side of Figure 7 illustrates a potential *Read-after-Write (RaW) conflict* that prevents in-place bufferization. The problem of efficient bufferization is related to register coalescing, the register allocation sub-task associated with the elimination of register-to-register moves.

```
!tISlice = type tensor<1x8x8xf32>
!tKFSlice = type tensor<1x8x32xf32>

%1 = scf.for %w = /*...*/ iter_args(%01 = %0) -> (!t0) {
  %3 = linalg.init_tensor [3, 4, 1, 8, 8] : tensor<?x?x1x8x8xf32>
  %PI = scf.for %kw = /*...*/ iter_args(%I1 = %I) -> (tensor<?x?x1x8x8xf32>) {
    %8 = scf.for %c = /*...*/ iter_args(%I2 = %I1) /*...*/ {
      %11 = tensor.extract_slice %I[0, %w+%kw, %c][1,min(8,990-%w-%kw),8][1,1,1]
      %13 = linalg.pad_tensor %11 nofold low[0,0,0] high[0,8-min(8,990-%w-%kw),0] {
      ^bb0(/*tensor indices*/):
        linalg.yield 0.000000 : f32
      } : tensor<1x?x8xf32> to !tISlice
      %14 = tensor.insert_slice %13 into %I2 [1,%c ceildiv 8,0,0,0]
                                             [1,1,1,8,8][1,1,1,1,1]
      scf.yield %14 : tensor<?x?x1x8x8xf32> }
    scf.yield %8 : tensor<?x?x1x8x8xf32> }

  %5 = scf.for %f = /*...*/ iter_args(%02 = %01) -> (!t0) {
    %6 = scf.for %kw = /*...*/ iter_args(%03 = %02) -> (!t0) {
      %7 = scf.for %c = /*...*/ iter_args(%04 = %03) -> (!t0) {
        %8 = tensor.extract_slice %F[%kw,%c,%f][1,8,32] [1,1,1] /*...*/
        %9 = tensor.extract_slice %04[0,%w,%f][1,min(8,988-%w),32][1,1,1] /*...*/
        %12 = tensor.extract_slice %PI[%kw,%c ceildiv 8,0,0,0][1,1,1,8,8][1,1,1,1,1]
        %13 = linalg.pad_tensor %8 nofold low[0,0,0] high[0,0,0] /*...*/
        %15 = linalg.pad_tensor %9 low[0,0,0] high[0, 8-min(8,988-%w),0] /*...*/
        %16 = linalg.conv_1d_nwc_wcf
             ins(%12, %13 : !tISlice, !tKFSlice) outs(%15 : !tKFSlice) -> !tKFSlice
        %17 = tensor.extract_slice %16[0,0,0][1,min(8,988-%w),32][1,1,1] /*...*/
        %18 = tensor.insert_slice %17 into %04[0, %w, %f][1,min(8,988-%w),32][1,1,1]
        scf.yield %18 : !t0 }
      scf.yield %7 : !t0 }
    scf.yield %6 : !t0 }
  scf.yield %5 : !t0 }
```

vectorize

```
!vecI = type vector<1x8x8xf32>
!vecKF = type vector<1x8x32xf32>
!vecO = type vector<1x8xf32>
#proj_012 = affine_map<(d0,d1,d2,d3) -> (d0,d1,d3)
#proj_32 = affine_map<(d0,d1,d2,d3) -> (d3,d2)

%1 = scf.for %w = /*...*/ iter_args(%01 = %0) -> (!t0) {
  %3 = linalg.init_tensor [3, 4, 1, 8, 8] : tensor<?x?x1x8x8xf32>
  %PI = scf.for %kw = /*...*/ iter_args(%I1 = %I)->(tensor<?x?x1x8x8xf32>) {
    %8 = scf.for %c = /*...*/ iter_args(%I2 = %I1) /*...*/ {
      %11 = tensor.extract_slice %I /*...*/
      %12 = vector.transfer_read %11[0,0,0], 0.000000
            {in_bounds=[true,false,true]} : tensor<1x?x8xf32>, !vecI
      %13 = vector.transfer_write %12, %arg8[%kw,%c ceildiv 8,0,0,0]
            {in_bounds=[true,true,true]} : !vecI, tensor<?x?x1x8x8xf32>
      scf.yield %13 : tensor<?x?x1x8x8xf32> }
    scf.yield %8 : tensor<?x?x1x8x8xf32> }

  %5 = scf.for %f = /*...*/ iter_args(%02 = %01) -> (!t0) {
    %7 = tensor.extract_slice %02[0,%w,%f][1,min(8,988-%w),32][1,1,1]/*...*/
    %8 = vector.transfer_read %7[0,0,0], 0.000000
      {in_bounds=[true,false,true]} : tensor<1x?x32xf32>, !vecKF
    %9 = scf.for %kw  = /*...*/ iter_args(%03 = %8) -> (!vecO) {
      %12 = scf.for %c = /*...*/ iter_args(%04 = %03) -> (!vecO) {
        %14 = vector.transfer_read %PI[%kw,%c ceildiv 8,0,0,0], 0.000000
              {in_bounds=[true,true,true]} : tensor<?x?x1x8x8xf32>, !vecI
        %15 = vector.transfer_read %F[%kw,%c,%f], 0.000000
              {in_bounds=[true,true,true]} : tensor<3x32x64xf32>, !vecKF
        %16 = vector.extract %15[0] : !vecKF
        %17 = vector.contract {indexing_maps=[#proj_012,#proj_32,#proj_012],
          iterator_types=["parallel","parallel","parallel","reduction"]}
          %14, %16, %04 : !vecI, vector<8x32xf32> into !vecKF
        scf.yield %17 : !vecO }
      scf.yield %7 : !vecO }
    %A = vector.transfer_write %9, %7[0,0,0]
      {in_bounds=[true,false,true]} : !vecKF, tensor<1x?x32xf32>
    %B = tensor.insert_slice %10 into %02[0,%w,%f][1,min(8,988-%w),32][1,1,1]
    scf.yield %B : !t0 }
  scf.yield %5 : !t0 }
```

Fig. 6. Fixed-size operations can be directly vectorized.

We propose a bufferization interface freeing upstream passes of the risk of incurring a performance penalty when high-level transformations result in unexpected allocation and copying. It is based on the so-called *destination-passing style*: one of the tensor arguments is singled out and tied with the resulting tensor for in-place bufferization. This singled-out tensor argument is called an *output* tensor; see the left-hand side of Figure 7. After lowering, output tensors are similar to C++ *output parameters* that are passed as non-const references and used for returning the result of a computation. Yet the tie between an output tensor (argument) and the operation's result serve as a bufferization constraint with no observable impact on the functional semantics; in particular, output tensors still appear as immutable. During bufferization, only output tensors are considered when looking for a buffer to write the result of an operation into.

The rationale is two-fold: first of all it provides a non-ambigious, "unsurprising" mechanism for driving bufferization choices from higher level optimization algorithms; second, notice the ubiquitous sub-setting operations resulting from tiling and structured control flow (`extract_slice`, `insert_slice` and `scf.yield`) naturally consume their tensor argument, making them ideal candidates for in-place bufferization. A comprehensive example is shown in Figure 8. The trade-off is that upstream compilation passes are responsible of rewriting the IR in destination-passing style. In particular, we believe that a global copy elimination problem can be formalized on top of destination-passing style, offering the best of both worlds in terms of allowing passes to optimize memory usage at a global scale, while enabling a robust, in-place bufferization path for the important special case of refining structured operations.

During bufferization, before modifying the, an analysis decides for each tensor OpOperand `%t` whether *buffer(*`%t`*)* (*in-place bufferization*) or a copy thereof (*out-of-place bufferization*), denoted by *copy(buffer(*`%t`*))*, should be used with

Fig. 7. Left-hand side: output tensor arguments, tied with the result of an operation, in destination-passing style. Right-hand side: example of a read-after-write conflict.

Fig. 8. Bufferization assigns tensor values to buffers, taking into account function-level annotations `#in`, `#out` from Figure 4. Data flow is replaced by side effects, unnecessary values are crossed out on the left. "Computational payload" dialects such as `linalg` and `vector` are designed to support both `tensor` and `memref` (buffer) containers.

the new memref operation. The analysis simulates *a future in-place bufferization of the OpOperand* and checks if a RaW conflict can be found under this assumption. If not, the analysis greedily commits to this in-place bufferization decision. Furthermore, the analysis stores the fact that the OpOperand and its potentially aliasing OpResult are now *known to alias*, by merging their alias sets. The search for RaW conflicts only involves the traversal of tensor SSA use-def chains. The extended version of the paper [20] details the procedure and covers special cases such as partial updates, cast operations and initialization.

3.5 Lowering of Multidimensional Vector Operations to LLVM

At this point, the IR has reached a level of abstraction close to nested loops with vector intrinsics in C, except that we operate on multi-dimensional vectors. In the simplest case, multi-dimensional `vector.transfer` operations lower to multiple 1-D `vector.load` and `vector.store` operations. When supported by hardware, they can also lower to n-D operations and DMA transfers. In more complex cases, transfer operations lower to a combination of broadcast, tranposition and

```
!MIView = type memref<1x?x8xf32, offset: ?, strides: [31680,32,1]>
!MOView = type memref<1x?x32xf32, offset: ?, strides: [63232,64,1]>
!MBuf = type memref<3x4x1x8x8xf32>

func @conv_1d_nwc_wcf_main(%I: memref<1x990x32xf32>,
                           %F: memref<3x32x64xf32>, %O: memref<1x988x64xf32>) {
  %0 = memref.alloc() {alignment = 128 : i64} : !MBuf
  scf.for %w = /*...*/ {
    scf.for %kw = /*...*/ {
      scf.for %c = /*...*/ {
        %5 = memref.subview %I[0,min(8,988-%w,%kw][1,min(8,990-%w-%kw),8][1,1,1]
             : memref<1x990x32xf32> to !MIView
        %6 = vector.transfer_read %5[0,0,0], /*...*/ : !MIView, !vecI
        vector.transfer_write %6, %0[%kw,%c ceildiv 8,0,0,0] /*...*/ !vecI, !MBuf
      }}

    scf.for %f = /*...*/ {
      %2 = memref.subview %O[0,%w,%f][1,min(8,988-%w),32][1,1,1]
           : memref<1x988x64xf32> to !MOView
      %3 = vector.transfer_read %2[0,0,0], /*...*/ : !MOView, !vecKF
      %4 = scf.for %kw  = /*...*/ iter_args(%O3 = %3) -> (!vecO) {
        %5 = scf.for %c = /*...*/ iter_args(%O4 = %O3) -> (!vecO) {
          %14 = vector.transfer_read %0[%kw,%c ceildiv 8,0,0,0], /*...*/
                : memref<3x4x1x8x8xf32>, !vecI
          %15 = vector.transfer_read %F[%kw,%c,%f], /*...*/
                : memref<3x32x64xf32>, !vecKF
          %16 = vector.extract %15[0] : !vecKF
          %17 = vector.contract /*...*/
          scf.yield %17 : !vecO }
        scf.yield %7 : !vecO }
      vector.transfer_write %4, %2[0,0,0] /*...*/ : !vecKF, !MOView
    }}
  memref.dealloc %0 : !MBuf
  return }
```

lower vectors

```
!MBufView = memref<1x?x8xf32, offset: 0, strides: [64,8,1]>
!MIDynView = memref<1x?x8xf32, offset: 0, strides: [?,?,?]>

%11 = memref.subview %0 /*...*/
%12 = scf.if min(8,988-%w) >= 8 {
  scf.yield %11 as !MIDynView
} else {
  linalg.fill(0.000000, %0) : f32, memref<1x8x8xf32>
  %21 = memref.subview %11/*...*/ : !MIView to !MIView
  %22 = memref.subview %0/*...*/ : memref<1x8x8xf32> to !MBufView
  linalg.copy(%21, %22) : !MIView, !MBufView
  scf.yield %0 as !MIDynView }

%13 = vector.load %12[0,0,0] : !MBufView, vector<8xf32>
vector.store %13, %3[%kw,%c ceildiv 8,0,0,0] : !MBuf, vector<8xf32>
/*... etc ...*/
%20 = vector.load %12[0,0,7] : !MBufView, vector<8xf32>
vector.store %20, %3[%kw,%c ceildiv 8,0,7,0] : !MBuf, vector<8xf32>

%37 = vector.load %0[%kw,%c ceildiv 8,0,0,0] : !MBuf, vector<8xf32>
%61 = vector.extract %37[0] : vector<8xf32>
%62 = vector.insert %61, dense<0.000000e+00> [0, 0] : f32 into vector<8x8xf32>
/*... more 1D vectors inserted into %62 to produce %64,%68, ...*/
%44 = vector.load %0[%kw,%c ceildiv 8,0,7,0] : !MBuf, vector<8xf32>
%187 = vector.extract %44[7] : vector<8xf32>
%188 = vector.insert %187, %186 [7, 7] : f32 into vector<8x8xf32>

%45 = vector.load %F[%kw,%c,%f] : memref<3x32x64xf32>, vector<32xf32>
/*... etc ...*/
%59 = vector.load %F[%kw,%c+7,%f] : memref<3x32x64xf32>, vector<32xf32>

%189 = vector.extract %188[0] : vector<8x8xf32>

%60 = vector.extract %O4[0] : vector<1x8x32xf32>
%190 = vector.outerproduct %189, %45, %60
       {kind = #vector.kind<add>} : vector<8xf32>, vector<32xf32>
/*... third operand of outerproduct is the result of the previous one ...*/
%203 = vector.extract %188[7] : vector<8x8xf32>
%204 = vector.outerproduct %203, %59, %202
       {kind = #vector.kind<add>} : vector<8xf32>, vector<32xf32>
%205 = vector.broadcast %204 : vector<8x32xf32> to vector<1x8x32xf32>
```

Fig. 9. The `vector` dialect can be lowered progressively to simpler operations on 1-D vectors. Illustrated on lowering contractions to outer products, with parts in italic simplified for brevity and repetitive parts omitted. Lower-level vector operations require constant indices and are produced by unrolling the outer dimensions.

```
%r = vector.transfer_read %m1[0,0], 0.0f {
  in_bounds=[true,true],
  permutation_map=affine_map<(d0,d1)->(d1,d0)>}
  : memref<2x16xf32>, vector<8x1xf32>
```

(b)

```
%vt = vector.transfer_read %m1[0,0], 0.0f {
  in_bounds=[true,true]}
  : memref<2x16xf32>, vector<1x8xf32>
%r = vector.transpose %vt, [1, 0]
  : vector<1x8xf32> to vector<8x1xf32>
```

(c)

```
%l = vector.load %m1[%c0, %c0]
  : memref<2x16xf32>, vector<8xf32>
%cast = vector.shape_cast %l
  : vector<8xf32> to vector<1x8xf32>
%r = vector.transpose %cast, [1, 0]
  : vector<1x8xf32> to vector<8x1xf32>
```

(a)

```
%a = vector.transfer_read %m0[0], 0.0f {
  in_bounds=[true,true],
  permutation_map=affine_map<(d0)->(d0,0)>}
  : memref<2xf32>, vector<2x2xf32>
%b = vector.transfer_read %m1[0,0], 0.0f {
  in_bounds=[true,true],
  permutation_map=affine_map<(d0,d1)->(d1,d0)>}
  : memref<2x16xf32>, vector<16x2xf32>
%v = vector.broadcast 0.0f : f32 to vector<2x16xf32>
%d = vector.contract #matmul_trait %a, %b, %v
  : vector<2x2xf32>, vector<16x2xf32>
vector.transfer_write %d, %m2[0,0] {
  in_bounds=[true,true]}
  : vector<2x16xf32>, memref<2x16xf32>
```

(a)

```
%b0 = vector.extract_strided_slice %b
  {offsets=[0,0], sizes=[8,2], strides=[1,1]}
  : vector<16x2xf32> to vector<8x2xf32>
%v0 = vector.extract_strided_slice %v
  {offsets=[0,0], sizes=[2,8], strides=[1,1]}
  : vector<2x16xf32> to vector<2x8xf32>
%d0 = vector.contract #matmul_trait %a, %b0, %v0
  : vector<2x2xf32>, vector<8x2xf32>
%r0 = vector.insert_strided_slice %d0, %v
  {offsets=[0,0], strides=[1,1]}
  : vector<2x8xf32> into vector<2x16xf32>

%b1 = vector.extract_strided_slice %b
  {offsets=[8,0], sizes=[8,2], strides=[1,1]}
  : vector<16x2xf32> to vector<8x2xf32>
%v1 = vector.extract_strided_slice %v
  {offsets=[0,8], sizes=[2,8], strides=[1,1]}
  : vector<2x16xf32> to vector<2x8xf32>
%d1 = vector.contract #matmul_trait %a, %b1, %v1
  : vector<2x2xf32>, vector<8x2xf32>
%d = vector.insert_strided_slice %d1, %r0
  {offsets=[0,8], strides=[1,1]}
  : vector<2x8xf32> into vector<2x16xf32>
```

(d)

```
%a0 = vector.transpose %a, [1, 0]
  : vector<1x1xf32> to vector<1x1xf32>
%b0 = vector.transpose %b, [1, 0]
  : vector<8x1xf32> to vector<1x8xf32>
%a1 = vector.extract %a0[0] : vector<1x1xf32>
%b1 = vector.extract %b0[0] : vector<1x8xf32>
%d = vector.outerproduct %a1, %b1, %acc {
  kind = #vector.kind<add>} : vector<1xf32>,vector<8xf32>
```

(e)

```
%a0 = vector.transpose %a, [1, 0]
  : vector<8x1xf32> to vector<1x8xf32>
%a1 = vector.extract %a0[0] : vector<1x8xf32>
%b0 = vector.extract %b[0, 0] : vector<1x1xf32>
%bsplat = vector.splat %b0 : vector<8xf32>
%a2 = vector.extract %a1[0] : vector<1x8xf32>
%d = vector.fma %a2, %bsplat, %acc : vector<8xf32>
```

Fig. 10. Progressive lowering of the `vector` operations representing matrix product: (a) vector unrolling to the target shape $2 \times 8 \times 2$ introduces vector slice manipulation; (b) the transfer permutation is materialized as a `transpose` operation; (c) 1-D transfers become loads with shape adaptation; (d) the contraction rewrites into a sequence of outer products over a vector accumulator, and each one finally lowers to (e) fused multiply-add instructions.

masked scatter/gather. In the particular case where the `vector.transfer` cannot be determined to be in-bounds, one must resort to an additional separation between full and partial transfer, akin to the full and partial tile separation for non-divisible tile sizes. This is illustrated in Figure 9 (right) in the `else` block

around the `linalg.copy(%21, %22)` operation. The progressive lowering process and resulting code for our example has several dozen operations, as shown in Figure 10. The detailed discussion of this process is provided in the extended version of the paper [20].

3.6 Overview of Sparse Code Generation

Let us now briefly describe how sparse code generation fits into the picture. As stated earlier, the `sparse_tensor` dialect bridges high-level operations on sparse tensors types with lower-level operations on the actual sparse storage [2]. To this end, the dialect introduces a tensor encoding attribute derived from the formats of the Tensor Algebra Compiler (TACO) [10]. Figure 11 illustrates this on matrix product over Doubly Compressed Sparse Column (DCSC) storage.

Rewrite rules specific to the `sparse_tensor` dialect lower the kernel to a sparse storage scheme and imperative constructs that only store and iterate over the nonzero elements. This approach to automatic sparse code generation was pioneered by [3,4] in the context of sparse linear algebra, and later generalized to sparse tensor algebra in [10]. The details of these rewrite rules are outside the scope of this paper, but they follow the structured code generation philosophy described earlier. Variations include the use of sparse index sets and slightly more complicated bufferization due to the compound nature of sparse storage schemes. The rewrite rules similarly interoperate with `linalg`, `tensor`, `memref`, `scf`, and `vector` abstractions, thereby lowering a sparsity-agnostic definition of a kernel into a form that fully exploits the sparsity of tensors as well as all performance features of the target architecture.

Figure 12 illustrates the approach on the prototypical sparse matrix-vector product `x = Ab`. The Compressed Sparse Row (CSR) format is suitable for a matrix `A` with no specific sparsity pattern. Nested `scf` loops iterate over the outer dense dimension and over the nonzero elements of the compressed inner dimension by means of an indirection. If `A` is actually a structured sparse matrix where most columns are empty but nonzero the columns are dense, the `CDC` format is suitable, favoring column-wise access and compressing the outermost dimension (now columns) only. Specializing for this format for a sparse 8192×8192 matrix and a vector length of 16, the sparse rewriting rules compose with vectorization to yield sparse vector code that skips over empty column while performing the innermost dense update using vectors.

3.7 Discussion

The transformations we introduced are legal by design, in the sense that their legality and applicability derive from an operation's properties and structure.

The traditional compilation for numerical computing [1] revolves around:

- Legality: what transformations can be applied without changing the observed program semantics? Legality conditions are often checked through static analyses. They may be performed upfront or on-demand, and their results may be updated as the IR is transformed.

```
#DCSC = #sparse_tensor.encoding<{
  dimLevelType = [ "compressed", "compressed" ],
  dimOrdering = affine_map<(i,j) -> (j,i)>
}>

#CSR = #sparse_tensor.encoding<{ dimLevelType = [ "dense", "compressed" ] }>

// Dense linalg.matmul on tensors
%0 = linalg.matmul ins(%a, %b: tensor<10x20xf32>, tensor<20x30xf32>) outs(%c: tensor<10x30xf32>) -> tensor<10x30xf32>

// Same matmul with SpMM kernel
%0 = linalg.matmul ins(%a, %b: tensor<10x20xf32, #CSR>, tensor<20x30xf32>) outs(%c: tensor<10x30xf32>) -> tensor<10x30xf32>

// Same matmul with SpMSpM kernel
%0 = linalg.matmul ins(%a, %b: tensor<10x20xf32, #CSR>, tensor<20x30xf32, #DCSC>) outs(%c: tensor<10x30xf32>) -> tensor<10x30xf32>
```

simply add a sparse attribute to the tensor type

Fig. 11. Definition of sparse tensor layouts and their usage in a matrix multiplication.

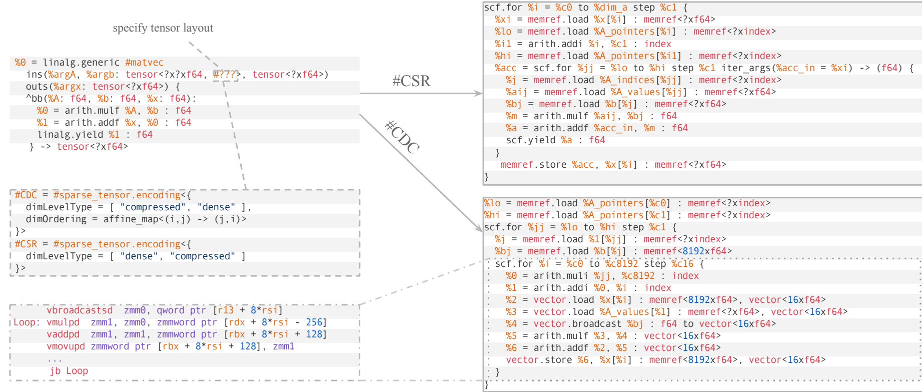

Fig. 12. Lowering sparse matrix-vector multiplication.

– Applicability: how complex is the IR matching process for finding where to apply a transformation? Applicability also encompasses considerations related to the loss of high-level semantic information and the ability to apply subsequent transformations.
– Profitability: what are the transformations deemed beneficial for a given metric? For example, polyhedral compilers often focus on finding an objective function to minimize (universal or target-specific) [21], while autotuners may rely on a learned performance model to accelerate search [23].

It is of central importance to control the abstractions on which the transformation legality, applicability and profitability questions relate to. The finer-grained the IR, the more general and canonical the representation, but also the more intractable the analyses and transformations. Indeed, canonicalization to some flavor SSA CFG such as LLVM IR has proven invaluable in enabling the reuse of common infrastructure for middle-end and back-end compilers. But lowering abstractions and domain knowledge too quickly reduces the amount of structure available to derive transformations from. While a loop nest is a net abstraction gain compared to a CFG for the application of loop transformations, important information is still lost. It induces non-trivial phase ordering issues: e.g., loop fusion to enhance temporal locality may alter the ability to recognize an efficient BLAS-2 or BLAS-3 implementation in a numerical library.

The higher-level abstractions we propose facilitate the declarative specification of transformations. It makes it possible to target individual operations in the IR rather than large multi-operation constructs such as loops and control flow graphs. This is the case of tiling, fusion and unrolling.[4] Loops and other constructs may be produced as a result of transformations, but they rarely need to be targeted by further high-level transformations. On the other hand, target specifications can be arbitrarily complex and may use the pattern-matching infrastructure available in MLIR such as the PDL dialect.

Furthermore, transformation orchestration calls for a meta-programming dialect suitable for IR manipulation. Such a dialect makes it possible to capture *transformation schedules* [13] in the IR itself. Transformations schedules can be stored, analyzed, parameterized, and even *shipped separately* from the main compiler. This is paramount to retargeting a tensor compiler to new hardware or to port it to a different ML framework. A declarative approach also facilitates the design of custom passes by selecting specific rewrite rules.

Finally, the multi-level nature of MLIR makes it possible to build higher-level dialects to define IR transformations superimposed on the existing infrastructure and handled by progressive lowering. For example, the transformation sequence and some of the parameters can be reified into a new "strategy" operation that gets lowered into primitive transformation operations with the lowering also specified declaratively. Just as with any other dialect, IR modules using such meta-programming dialects can be created programmatically from any language. The textual or binary IR format enables loosely coupled communication between the front-end language—in which the transformation is written—and the compiler infrastructure. The expressiveness of such transformations is similar to that of RISE/ELEVATE [8] but without restrictions to the specification language.

4 Related Work

The extended version of the paper surveys related compilers and infrastructure [20]: ONNX (`https://onnx.ai`), XLA [22], Halide [13], TVM [5], Fireiron [6], LIFT [18], Multi-Dimensional Homomorphisms [14], Elevate [7], Glenside [17], Tensor Comprehensions [21], PolyMage [12]. It also discusses the interaction with lower level code generators. The crux of the matter is that structured operations capture the common abstractions and transformations underlying the different flavors of tensor compilers. From the mathematical specification down to super-optimized blocks of vector instructions, from polyhedral to domain-specific and algebraic forms, with or without explicit scheduling languages.

5 Conclusion

We presented the composable multi-level intermediate representation and transformations that underpin tensor code generation in MLIR. This so-called "structured code generation" approach leverages the natural decomposition of tensor

[4] Some transformations such as software pipelining remain naturally attached to loops.

algebra operations, doing away with static analyses and applicability checks on low-level IR. The resulting design is modular and built with optionality in mind. Abstractions span data structures and control flow with both functional (SSA form) and imperative (side-effecting) semantics; they serve as generic building blocks for composable, interoperable, retargetable tensor compilers.

References

1. Allen, R., Kennedy, K.: Optimizing Compilers for Modern Architectures: A Dependence-Based Approach. Morgan Kaufmann (2001)
2. Bik, A., Koanantakool, P., Shpeisman, T., Vasilache, N., Zheng, B., Kjolstad, F.: Compiler support for sparse tensor computations in MLIR. ACM Trans. Archit. Code Optim. **19**(4) (sep 2022). `https://doi.org/10.1145/3544559`
3. Bik, A.J.: Compiler Support for Sparse Matrix Computations. Ph.D. thesis, Department of Computer Science, Leiden University (1996), iSBN 90-9009442-3
4. Bik, A.J., Brinkhaus, P.J., Knijnenburg, P.M., Wijshoff, H.A.: The automatic generation of sparse primitives. Transactions on Mathematical Software **24**, 190–225 (1998), `https://doi.org/10.1145/290200.287636`
5. Chen, T., Moreau, T., Jiang, Z., Zheng, L., Yan, E., Shen, H., Cowan, M., Wang, L., Hu, Y., Ceze, L., et al.: TVM: An automated end-to-end optimizing compiler for deep learning. In: 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18). pp. 578–594. USENIX Association (2018), `https://www.usenix.org/conference/osdi18/presentation/chen`
6. Hagedorn, B., Elliott, A.S., Barthels, H., Bodik, R., Grover, V.: Fireiron: A data-movement-aware scheduling language for gpus. In: Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques. p. 71–82. PACT '20, Association for Computing Machinery, New York, NY, USA (2020). `https://doi.org/10.1145/3410463.3414632`
7. Hagedorn, B., Lenfers, J., Koehler, T., Gorlatch, S., Steuwer, M.: A language for describing optimization strategies. CoRR **abs/2002.02268** (2020), `https://arxiv.org/abs/2002.02268`
8. Hagedorn, B., Lenfers, J., Kundefinedhler, T., Qin, X., Gorlatch, S., Steuwer, M.: Achieving high-performance the functional way: A functional pearl on expressing high-performance optimizations as rewrite strategies. Proceedings of ACM on Programming Languages **4**(ICFP) (Aug 2020). `https://doi.org/10.1145/3408974`
9. IREE Developers: IREE: the Intermediate Representation Execution Environment (2021), `https://google.github.io/iree/`
10. Kjolstad, F., Kamil, S., Chou, S., Lugato, D., Amarasinghe, S.: The tensor algebra compiler. Proc. ACM Program. Lang. **1**(OOPSLA) (Oct 2017). `https://doi.org/10.1145/3133901`
11. Lattner, C., Amini, M., Bondhugula, U., Cohen, A., Davis, A., Pienaar, J., Riddle, R., Shpeisman, T., Vasilache, N., Zinenko, O.: MLIR: Scaling compiler infrastructure for domain specific computation. In: 2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO). pp. 2–14. IEEE/ACM, IEEE/ACM (2021), `https://doi.org/10.1109/CGO51591.2021.9370308`
12. Mullapudi, R.T., Vasista, V., Bondhugula, U.: Polymage: Automatic optimization for image processing pipelines. In: Özturk, Ö., Ebcioglu, K., Dwarkadas, S. (eds.) Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS

2015, Istanbul, Turkey, March 14-18, 2015. pp. 429–443. ACM (2015). https://doi.org/10.1145/2694344.2694364
13. Ragan-Kelley, J., Barnes, C., Adams, A., Paris, S., Durand, F., Amarasinghe, S.: Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. ACM SIGPLAN Notices **48**(6), 519–530 (2013), https://doi.org/10.1145/2499370.2462176
14. Rasch, A., Schulze, R., Gorlatch, S.: Generating portable high-performance code via multi-dimensional homomorphisms. In: 2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT). pp. 354–369. IEEE, Seattle, WA (2019), https://doi.org/10.1109/PACT.2019.00035
15. Rotem, N., Fix, J., Abdulrasool, S., Deng, S., Dzhabarov, R., Hegeman, J., Levenstein, R., Maher, B., Satish, N., Olesen, J., Park, J., Rakhov, A., Smelyanskiy, M.: Glow: Graph lowering compiler techniques for neural networks. CoRR **abs/1805.00907** (2018), http://arxiv.org/abs/1805.00907
16. Schrijver, A.: Theory of Linear and Integer Programming. Wiley (1986)
17. Smith, G.H., Liu, A., Lyubomirsky, S., Davidson, S., McMahan, J., Taylor, M.B., Ceze, L., Tatlock, Z.: Pure tensor program rewriting via access patterns (representation pearl). CoRR **abs/2105.09377** (2021), https://arxiv.org/abs/2105.09377
18. Steuwer, M., Remmelg, T., Dubach, C.: Lift: A functional data-parallel ir for high-performance gpu code generation. In: 2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO). pp. 74–85. IEEE/ACM (2017). https://doi.org/10.1109/CGO.2017.7863730
19. Tillet, P., Kung, H.T., Cox, D.: 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. p. 10–19. MAPL 2019, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3315508.3329973
20. Vasilache, N., Zinenko, O., Bik, A.J.C., Ravishankar, M., Raoux, T., Belyaev, A., Springer, M., Gysi, T., Caballero, D., Herhut, S., Laurenzo, S., Cohen, A.: Composable and modular code generation in MLIR: A structured and retargetable approach to tensor compiler construction. CoRR **abs/2202.03293** (2022), https://arxiv.org/abs/2202.03293
21. Vasilache, N., Zinenko, O., Theodoridis, T., Goyal, P., Devito, Z., Moses, W.S., Verdoolaege, S., Adams, A., Cohen, A.: The next 700 accelerated layers: From mathematical expressions of network computation graphs to accelerated gpu kernels, automatically. ACM Transactions on Architecture and Code Optimization (TACO) **16**(4), 1–26 (2019), https://doi.org/10.1145/3355606
22. XLA team within Google: XLA: TensorFlow, Compiled. Google Developers Blog (2017), https://developers.googleblog.com/2017/03/xla-tensorflow-compiled.html
23. Zheng, L., Jia, C., Sun, M., Wu, Z., Yu, C.H., Haj-Ali, A., Wang, Y., Yang, J., Zhuo, D., Sen, K., Gonzalez, J.E., Stoica, I.: Ansor: Generating high-performance tensor programs for deep learning. In: 14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020. pp. 863–879. USENIX Association (2020), https://www.usenix.org/conference/osdi20/presentation/zheng