Modelling linear algebra kernels as polyhedral volume operations

Karl F. A. Friebel
Technische Universität Dresden
Dresden, Germany
karl.friebel@tu-dresden.de

Asif Ali Khan
Technische Universität Dresden
Dresden, Germany
asif_ali.khan@tu-dresden.de

Lorenzo Chelini
Intel Switzerland
lorenzo.chelini@intel.com

Jeronimo Castrillon
Technische Universität Dresden
Dresden, Germany
jeronimo.castrillon@tu-dresden.de

Abstract

Linear algebra de-facto dominates an entire branch of current hard- and software design efforts focused on bringing faster and more efficient machine learning and signal processing kernels. For many years, compilers have leveraged the well-defined semantics of linear algebra operations for optimization, with lots of recent research around machine learning. Due to the structure of the operations the polyhedral model is an ideal fit for reasoning about linear algebra programs. In this paper, we present a related model in which such programs are represented as sequences of operations on indexed volumes characterized by their element-wise dependencies. We show how this model opens a way towards efficiently implementing compound kernels by partitioning and specializing over index domains. We demonstrate how memory layout, partitioning (e.g., tiling) and compile-time sparsity are exploited using this model.

ACM Reference Format:
Karl F. A. Friebel, Asif Ali Khan, Lorenzo Chelini, and Jeronimo Castrillon. 2023. Modelling linear algebra kernels as polyhedral volume operations. In *Proceedings of 13th International Workshop on Polyhedral Compilation Techniques (IMPACT'23).* ACM, New York, NY, USA, 10 pages.

1 Introduction

Linear algebra computations are central to many application domains, including scientific computing, data analytics, and machine learning. These kernels are well understood, and as a result, hardware vendors have developed highly efficient hand-optimized libraries for them. For dense linear algebraic kernels, several highly efficient implementations exist that are built around the basic linear algebra subroutines (BLAS) [11], LAPACK [2] or similar libraries [22]. However, these libraries are complex and are constrained on input sizes and interfaces.

A large body of work leverages the polyhedral model [7] to optimize linear algebra kernels [5, 9]. Depending on the optimization target, polyhedral compilers often implement complex transformations on the iteration space to improve data locality, maximize parallelism and optimize data and memory layouts [3, 8, 13, 16, 18]. The `linalg` dialect [1] in the recent *multi level intermediate representation* (MLIR) compiler infrastructure [10] enables reasoning about linear algebra kernels at a higher abstraction, supports dense and sparse data types and empowers specializing over them [4]. It also accounts for mapping to efficient library implementations when available, simplifying code generation.

Both polyhedral and MLIR-based compilers essentially operate on multidimensional arrays. These also serve as descriptors for the memory structure, linking memory to computations on values. We generalize over arrays and define a *volume* V as an aggregate value comprising indexed element values (scalars). A volume is characterized by an *index domain* $\operatorname{dom} V$, which is a polyhedron of all element indices. Elements in a volume are referred to by index tuples $\mathbf{i} \in \operatorname{dom} V$ into the volume $V[\mathbf{i}] = V[i_1, \ldots, i_N]$ where N is the rank of the volume. Volumes permit arbitrary polyhedral index domains and are intended to model the structural properties of data, as well as memory when needed.

In this paper, we leverage volumes and propose a model that simplifies the data flow analysis of the polyhedral model by specializing it for tensor programs. Compared to the polyhedral model, which mainly targets schedule and layout optimizations, the proposed model targets a) performing dynamic shape inference of input/output/intermediate volumes, b) affine pattern matching to identify regions of interest (e.g., sparse regions in a volume) and specialize over them, and c) separating computation from memory. The latter is enabled by an operational semantics based on general affine volumes as opposed to strictly hyperrectangular ones. We also describe how the proposed model can be used on the MLIR `linalg` abstraction to generalize tiling and other partitioning transformations to arbitrary tensor programs.

IMPACT'23, January 16, 2023, Toulouse, France
2023.

2 Background

This section provides a short summary of closely related works and the concerns they share. It also provides background on the polyhedral model and the MLIR infrastructure for tensor algebra applications.

2.1 Polyhedral compilers

In polyhedral compilers, a program is modeled using statements and a schedule. A statement is defined over an iteration domain, with each tuple naming one statement instance. The schedule assigns schedule tuples to statement instances, defining a total lexicographical ordering. It encodes the order of execution of the statement instances.

Every input program has a default execution schedule called the *reference* schedule. The interaction of statement instances with memory accesses is modeled with *access relations* that map statement instances to array accesses. Access relations are used to determine whether a given schedule, after certain transformations, is valid or not, i.e., whether or not it respects the program data dependencies. A polyhedral scheduler is an ILP solver that finds schedules that optimize some metrics (e.g., parallelism, locality) while respecting program's true data dependencies in the reference schedule. The polyhedral dataflow analysis can use the read and write memory access relations to compute RAW, WAW and RAR dependencies for a given schedule.

2.2 The MLIR infrastructure

MLIR is a compiler infrastructure that supports multiple intermediate representations operating at different abstraction levels [10]. It has similarities to traditional SSA representations like LLVM IR, but hosts first-class concepts via dialect extensions, e.g., polyhedral modelling. Values in MLIR are defined by assignment and remain immutable. Assignments come from operations, which combine operands and state into results. Side-effect-free operations do not manipulate any state, and so they are (pure) functions of their operands.

The `tensor` and `linalg` dialects are amongst the major MLIR dialects that can model tensor programs as graphs of pure functions. No loop structure or other control flow is explicit in any of their primitives. Instead, each operation has an implicit iteration domain and is defined by a pointwise output expression. The most general form of this is the `linalg.generic` operation, which models all possible `linalg` operations using index maps and an element-wise body.

2.3 Related work

Many new compilers use polyhedral optimization techniques to improve a program's interaction with the memory hierarchy of the target device. Such tools [9, 14, 25] exploit cache hierarchies and shared memory usage to improve throughput. Polyhedral techniques can also be applied when these memory subsystems are generated as part of the compilation process [16]. This work does not present such an end-to-end compiler, but a reasoning framework.

ISL [19] remains the state-of-the-art framework in terms of solver latency and schedule quality for general polyhedral programs. It achieves this by using heuristics that perform very well on loosely constrained problems. For a complex set of constraints, e.g., for library matching, it can become prohibitively slow [17]. This is the result of the heuristics failing, which causes a fallback to the Feautrier scheduler [21]. Consequently, different compilers have opted to implement their own heuristics to address their specific concerns.

Among the shared concerns of these tools is the application of transform patterns such as fusion and tiling, which directly target memory access patterns. MLIR `linalg` [1] already offers limited (see section 3) support for both. However, these transforms are not order-independent [24]. In sequences of such transforms, e.g., to infer the tile shapes of intermediaries [14], polyhedral modelling can reach its limits [23]. In [24], an alternative output-to-input polyhedral code-generation strategy is proposed, which is a subset of the framework presented in this paper.

Languages such as TensorComprehensions [18] provide an expression-centric view of tensor programs, closer to SSA form. Here, alternative terminology establishes a connection between assignments and memory. A *physical tensor* is one which will have all of its elements stored in memory, as opposed to a *virtual tensor*, which is not stored at all [15]. Compilers such as TVM [6] use this to separate transient values from memory. The ordering problem of fusion with other transforms thus takes the form of a modified CSE problem. In this paper, we assume purely virtual expressions, which leads to the same traversal as in [24].

Views, which are index reassociations on tensors that never require recomputation, are commonly modelled as virtual tensors. Only special types of views are commonly in use, such as *offset-size-stride* views, receiving special optimizations. In our proposed framework, we offer higher generality without special handling.

3 Motivation

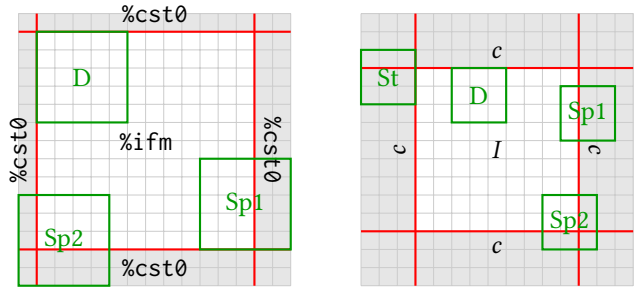
Figure 1 is an example snippet of a convolutional layer written in the MLIR `linalg` dialect. Its consists of an asymmetrical padding ①, a strided convolution ② and a leaky ReLU activation function ③.

In this program, we can derive additional facts about structural sparsity at compile-time. We know that `%pad0` in ① inserts a sparse boundary. Figure 2a graphically shows how the convolution function in ② will observe `%pad0` while performing the convolution.

In general (cf. fig. 2b), we expect every output element of a padded convolution to belong to one of the following sparsity regions: D for dense (assuming I is dense), St for static

```
^krnl0(%ifm: tensor<1x3x512x512xf32>):
①%pad0 = tensor.pad %ifm low[0,0,1,1] high
      ↪ [0,0,2,2] {
    ^bb0(%i0: index, %i1: index, %i2: index, %i3:
       ↪ index):
      tensor.yield %cst0 : f32
  } : tensor<1x3x512x512xf32> to tensor<1
      ↪ x3x515x515xf32>
②%conv0 = linalg.conv_2d_nchw_fchw {
      dilations = dense<1> : tensor<2xi64>,
      strides = dense<2> : tensor<2xi64>
    }
    ins(%pad0, %wgt0: tensor<1x3x515x515xf32>,
       ↪ tensor<8x3x5x5xf32>)
    outs(%conv0.init: tensor<1x8x256x256xf32>)
    -> tensor<1x8x256x256xf32>
③%act0 = linalg.generic #elementwise_traits
      ins(%conv0: tensor<1x8x256x256xf32>)
      outs(%act0.init: tensor<1x8x256x256xf32>) {
    ^bb0(%a: f32, %b: f32):
      %0 = arith.maxf %a, %cst0 : f32
      %1 = arith.mulf %0, %cst0_01 : f32
      %2 = arith.addf %a, %1 : f32
      linalg.yield %2 : f32
  } -> tensor<1x8x256x256xf32>
```

Figure 1. A convolutional layer in MLIR `linalg` (excerpt).



(a) From Figure 1. **(b)** In general.

Figure 2. Sparsity regions in a padded convolution.

(constants c only), Sp1 for sparse edges, and Sp2 for sparse corners. Prior research has demonstrated that, if identified correctly, region-specific optimizations in such scenarios can lead to considerably faster convolution implementations [4]. However, to generate efficient code that combines all these optimizations, computations have to be partitioned based on facts inferred from the data (sparse or dense).

State-of-the-art compilers are limited in detecting and optimizing for regions at this fine granularity for various reasons as outlined below.

MLIR: In MLIR `linalg`, the `TilingInterface` implements a codegen strategy that extracts a tile loop from an operation. However, it can only do this for an individual operation, and only if its index maps are permuted identities, i.e., the loops are directly correlated with output dimensions. The `tile-and-fuse` strategy attempts to remedy this, but is limited to a single producer-consumer pair at a time, which must also exhibit permuted identity mapping between produced and consumed indices. For example, this strategy is still incapable of fusing a pad *producer* with a generic consumer.

Our model aims to directly remedy this. `linalg` was designed around a future affine abstraction, which MLIR Presburger is now set to become. Similarly, we reproduced some of our results using a modified version of this library, in anticipation of future feature completeness.

Polyhedral compilers: A generic polyhedral compiler such as PET [20] can schedule each instance of a statement for every iteration index tuple separately. As a result, it is perfectly capable of performing tiling and fusing for not just tensor programs, but all that can be modeled by SCoPs. At the same time, this vastly increases the search space, and makes it more difficult to generate code comprised of fixed building blocks, e.g., required for library-based offloading.

Our model provides a more lightweight alternative to full-blown polyhedral rescheduling. Specialized for tensor programs, it requires less effort while still producing results exploitable by rescheduling. In future work, we intend to use general rescheduling for smaller subprograms guided by information gathered at the volume model.

We propose a volume-centric approach to modelling tensor programs to address these issues. Adopting an expression-based view of tensor programs, as opposed to a loop-based one, is not only natural in `linalg`, but can also simplify transformation ordering problems. In side-effect free form, these programs allow us to use a single common value abstraction throughout, regardless of how compute and memory will be arranged later. To support a variety of shapes while still being able to fall back on readily available mathematical and software frameworks, we limit these to unions of parametric integer polyhedra. To avoid confusion and overloading existing terminology, we call these values *volumes*.

4 Volume-based dataflow analysis

Our goal is to detect sparsity and other structural properties, and act on them by partitioning computations. Since computation implicitly follows the volumes, the underlying problem is how properties transfer between volumes via operations. We approach this using a specialized but simplified version of polyhedral dataflow analysis. We call this the *volume model* of a tensor program.

For integer sets and relations, and operations on them, the following definitions adopt the terminology and semantics of ISL [19]. This includes the usage of union sets and relations,

which can have multiple disjuncts, thus extend operational semantics. With respect to SSA programs, our notation and semantics are consistent with MLIR [10].

4.1 Volumes

All values in our SSA programs are tensors, which in simplified terms are hyperrectangular indexed families of scalars (multidimensional arrays). In our volume model, their direct equivalents are the arrays (cf. def. 4.7), which are a specialization of the more general volumes (cf. def. 4.1). We characterize every volume using a unqiue identifier and an index domain set.

Definition 4.1 (Volume)**.** A *volume* V is an indexable aggregate of values with a polyhedral index domain $\operatorname{dom} V \subset \mathbb{Z}^N$, where N is the *rank* of the volume.

Definition 4.2 (Element)**.** An *element* $V[i_1, \ldots, i_N] = V[\mathbf{i}]$ is the value of volume V at index $\mathbf{i} \in \operatorname{dom} V$.

The index spaces of different volumes are distinguished by their respective volume identifiers. Consequently, all elements are also uniquely identified by their index tuple. Values without structure (scalars) are also modeled as volumes.

Lemma 4.3. *Given a volume V with a rank of 0, it follows that $\operatorname{dom} V = \{\emptyset\}$, i.e., it is indexed only by the 0-tuple.*

Definition 4.4 (Scalar)**.** A *scalar* S is a volume of rank 0. We abbreviate $S = S[] = S[\emptyset]$.

Volumes permit arbitrary polyhedral index domains. They are intended to model structural properties of data. Multidimensional arrays serve as descriptors for data organization in memory.

Definition 4.5 (Hyperrectangle)**.** A set $S \subset \mathbb{Z}^N$ that is congruent with its box hull $\{\mathbf{s} : \operatorname{lexmin} S \leq \mathbf{s} \leq \operatorname{lexmax} S\}$ is called *hyperrectangular*.

Lemma 4.6. *A hyperrectangle R is uniquely defined by its offset $\Delta_\mathbf{0} R = \operatorname{lexmin} R$ and size $\Delta_\square R = \operatorname{lexmax} R - \Delta_\mathbf{0} R + \mathbf{1}$.*

Definition 4.7 (Array)**.** An *array* A is a volume with a hyperrectangular index domain such that $\Delta_\mathbf{0} \operatorname{dom} A = \mathbf{0}$.

Indexing operations can be generalized via affine maps. These views and their special cases play an important role in partitioning data, and consequently computations.

Definition 4.8 (View)**.** A *view* $\Pi : \operatorname{dom} V_\text{sub} \to \operatorname{dom} V_\text{sup}$ is an affine map that defines a subvolume $V_\text{sub}[\mathbf{i}] = V_\text{sup}[\Pi(\mathbf{i})]$ of the supervolume V_sup.

Definition 4.9 (Slice)**.** A *slice* $\Xi : \mathbf{i} \mapsto \Delta_\mathbf{0} \Xi + \mathbf{i} \odot \Delta_\mathbf{i+1} \Xi$ is a view defined by an offset $\Delta_\mathbf{0} \Xi$ and a stride $\Delta_\mathbf{i+1} \Xi$ vector.

4.2 Operations

The volume model is intended to transpose facts from volumes to other volumes via operations. These operations derive result elements (cf. def. 4.2) from operand elements, which we can model using element-wise data dependencies.

Definition 4.10 (Operation)**.** An *operation* X is a side-effect free function producing M result values from N operand values

$$X(O_1, \ldots, O_N) \mapsto R_1, \ldots, R_M$$

Definition 4.11 (Volume element map)**.** A *volume element map* is a union of affine maps between volume index domains. The volume element map $\mathcal{M}_X$ of an operation X encodes the dependencies of its result elements on its operand elements

$$\mathcal{M}_X : \bigcup_j \bigcup_k \{R_j \to O_k\}$$

Lemma 4.12. *Volume element maps have an implied context*

$$\mathcal{M}_X \subseteq \bigcup_j \operatorname{dom} R_j \times \bigcup_k \operatorname{dom} O_k$$

and can thus be defined by their gist, i.e., a map that has the same intersection with the context as the subset.

To allow for shape inference, in the following we will assume that $\operatorname{dom} R_j$ is unknown, and thus exclude it from the context.

4.2.1 `LinalgOp`.

An MLIR operation `X` implementing the `LinalgOp` interface associates each operand and result V with an indexing map $\mathcal{I}_V : \mathbb{Z}^L \to \operatorname{dom} V$. $\mathbb{Z}^L$ is the iteration space of the operation, where L is the number of loops.

The iteration domain $\operatorname{dom} \mathtt{X}$ of this operation is implicit, and is given by

$$\operatorname{dom} \mathtt{X} = \bigcap_j \operatorname{dom} \left(\mathcal{I}_{R_j} \underset{\text{rg}}{\cap} \operatorname{dom} R_j \right) \text{[1]}$$

where R_j enumerates all result values.

To compute the volume element map $\mathcal{M}_\mathtt{X}$, all results must be put in relation to all operands

$$\mathcal{M}_\mathtt{X} := \bigcup_j \bigcup_k \left(\left(\mathcal{I}_{R_j}^{-1} \underset{\text{rg}}{\cap} \operatorname{dom} \mathtt{X} \right) \circ \mathcal{I}_{O_k} \right)$$

Note that we intersect with the iteration domain, so that bounds not in the implicit context are explicit.

`X` might use SSA values U that are not operands of `X`, inside its element-wise body. For each (R_j, U), a pessimistic disjunct $\{\operatorname{dom} R_j \mapsto \operatorname{dom} U\}$ needs to be added.

4.3 Programs

Our model uses the SSA form to represent programs that manipulate volumes. Additionally, we disallow control flow in programs. In common terminology, a program must be a single basic block. As a result, the volume-wise dependencies are directly encoded in the use-def DAG of operations.

[1] $A \underset{\text{rg}}{\cap} B$ is the intersection of A with $\{\operatorname{dom} A \mapsto B\}$

Definition 4.13 (Assignment)**.** An *assignment* is an SSA statement that defines one or more values by applying an operation

$$D_1, \ldots, D_M \leftarrow X(U_1, \ldots, U_N)$$

where D_i are the definitions and U_j are the uses.

Definition 4.14 (Program)**.** A *program* P is an unordered sequence of assignments, plus a set of incoming definitions and exiting uses. It is well-formed iff the graph associating each use with its unique definition is acyclic.

Assignments can be associated volume element maps through their operations. A well-formed program with live-ins I and live-outs O is also an operation $O_1, \ldots, O_M \leftarrow P(I_1, \ldots, I_N)$, and has a volume element map $\mathcal{M}_P$.

Definition 4.15 (Lenient composition)**.** The *lenient composition* $A \circ^{\text{id}} B$ extends composition of unions of affine maps

$$A \circ^{\text{id}} B = A \circ (B \cup \text{id}(\text{range}\, A \setminus \text{dom}\, B))$$

where $\text{id}\, X$ is the identity over X.

Using the lenient composition, we can exploit the SSA property to compute the volume element map of a program in a single pass over the DAG in use-def order, instead of having to compute a transitive closure (cf. lemma A.5).

5 Usage

In this section we demonstrate the use of the volume model by means of the example in Figure 1. In summary, the steps involved in our example are 1) building the dependency model, 2) applying patterns and facts, 3) projecting to volumes of interest and 4) inspecting the sets and relations.

5.1 Building the dependency model

In our running example, see Figure 1, `%act0` is the output, and `%ifm` is the input. Hence, we are interested in $\mathcal{M}_{\texttt{krnl0}} : \text{dom}\, \texttt{act0} \to \text{dom}\, \texttt{ifm}$.

$$\text{dom}\, \texttt{act0} := \{\texttt{act0}[\mathbf{i}] : \mathbf{0} \le \mathbf{i} < [1, 8, 256, 256]\}$$
$$\text{dom}\, \texttt{ifm} := \{\texttt{ifm}[\mathbf{i}] : \mathbf{0} \le \mathbf{i} < [1, 3, 512, 512]\}$$

We also observe that $\mathcal{M}_{\texttt{krnl0}} = \mathcal{M}_3 \circ \mathcal{M}_2 \circ \mathcal{M}_1$. In other words, we can construct the program's dependency map from the per-operation maps (cf. lemma A.5).

Inspection of ③ shows us that it is an embarassingly parallel element-wise operation

$$\mathcal{M}_3 := \{\texttt{act0}[\mathbf{i}] \mapsto \texttt{conv0}[\mathbf{i}]\}$$

Operation ② is a convolution (cf. appendix A.2.7). Since there are batch and channel dimensions, not all dimensions are treated equally and the resulting map depends on the organization of the volume, i.e., the semantics of the dimensions. In `linalg`, this is not encoded in the type of the value, but in the operation name: `conv_2d_`**`nchw_fchw`**. Via the `LinalgOp` interface, we infer (cf. section 4.2.1)

$$\begin{aligned}\mathcal{M}_2 := \{\ &\texttt{conv0}[n, f, y, x] \mapsto \texttt{pad0}[n, c, 2y + a, 2x + b] \\ &\quad : 0 \le a < 5 \wedge 0 \le b < 5\} \\ \cup\ &\{\texttt{conv0}[n, f, y, x] \mapsto \texttt{wgt}[f, c, a, b]\}\end{aligned}$$

Operations like `tensor.pad` perform an element-wise selection between operands, based on the result index. In the case of ①, we construct (cf. appendix A.2.6)

$$\begin{aligned}\mathcal{M}_1 := \{\ &\texttt{pad0}[n, c, h, w] \mapsto \texttt{ifm}[n, c, h - 1, w - 1] \\ &\quad : 1 \le h < 513 \wedge 1 \le w < 513\} \\ \cup\ \{\ &\texttt{pad0}[n, c, h, w] \mapsto \texttt{cst0} \\ &\quad : h < 1 \vee w < 1 \vee h \ge 513 \vee w \ge 513\}\end{aligned}$$

Entering all this into ISL [19], we obtain $\mathcal{M}_{\texttt{krnl0}}$ as

```
{ act0[n, f, h, w] -> ifm[n, c, y, x]
    : y >= 2h - 1 and 0 <= y <= 511 and y <= 3 + 2h
      and x >= 2w - 1 and 0 <= x <= 511
      and x <= 3 + 2w;
  act0[n, f, h, w] -> cst0[]
    : h >= 255 or w >= 255 or h <= 0 or w <= 0 }
```

5.2 Applying facts

A fact takes the form of some predicate we assign to elements of volumes. For example, we know that `%cst0` is a compile-time constant. We now want to ask what elements of `%act0` are also compile-time constant, or derived from them, i.e., we want to know its sparsity.

In this example, we simply range-intersect $\mathcal{M}_{\texttt{krnl0}}$ with dom `cst0`, which gives us the second disjunct. By taking the domain of that disjunct, we transfer the fact onto `%act0`

$$\begin{aligned}I_c &= I_{\texttt{Sp1}} \cup I_{\texttt{Sp2}} \\ &= \{\texttt{act0}[n, f, h, w] : h \ge 255 \vee w \ge 255 \vee h \le 0 \vee w \le 0\}\end{aligned}$$

Recalling Figure 2a, we see that this is the union of 4 half-spaces that form a boundary with radius 1, which is what we expect for the output (stride = 2).

We can also use $\mathcal{M}_{\texttt{krnl0}}$ to apply partitionings. A tiling of h and w with tile sizes T_h and T_w respectively reads

$$\begin{aligned}\mathcal{M}_T := \{\ &\texttt{act1}[n, f, q_h, m_h, q_w, m_w] \mapsto \\ &\quad \texttt{act0}[n, f, T_h q_h + m_h, T_w q_w + m_w] \\ &\quad\quad : 0 \le m_h < T_h \wedge 0 \le m_w < T_w\}\end{aligned}$$

where q_x is the quotient or tile index, and m_x is the remainder or in-tile index. By computing $\mathcal{M}_T \circ \mathcal{M}_{\texttt{krnl0}}$, we effectively compute the volume element map of the tiled program. These tile sizes can be arbitrary, but are assumed to be fixed. For variable T_x and q_x, the expression $T_x q_x$ is no longer affine, but quasipolynomial. For reasons stated in section 3, we stick to Presburger algebra.

5.3 Inspecting the result

One very basic use of the model is to perform feed-forward shape inference. In that case, the facts we apply are the domains of the operands, which may even involve dynamic size parameters. The domain of the volume element map gives us the index domain of the result. From there, we can apply a number of techniques, e.g., compute the box hull.

In our example in fig. 1, shapes are not an issue, but sparsity is. To generate code specialized for sparsity, we first compute the domains of the sparsity regions. We already determined I_c as the subset of dom `act0` that involves `%cst0`

$$\begin{aligned} I_{\mathrm{D}} &= \operatorname{dom} \texttt{act0} \setminus I_c \\ &= \{\texttt{act0}[n, f, h, w] : 0 < h \leq 254 \wedge 0 < w \leq 254\} \\ I_{\mathrm{St}} &= I_c \setminus \operatorname{dom} \texttt{act0} = \emptyset \end{aligned}$$

To separate I_{Sp1} and I_{Sp2} from I_c, we can inspect its disjuncts. All elements of I_{Sp2} are contained within two boundary half-spaces, and thus two disjuncts. Figure 3a shows the resulting partitions, i.e., projecting fig. 2a from `%pad0` onto `%act0`.

Algorithm 1 outlines how code can be generated for hyperrectangular partitions using this information. We split `%act0` into subvolumes by intersecting the domain of $\mathcal{M}_{\texttt{krnl0}}$ with any of the I_x domains. The range of the result will tell us what subvolumes of the operands are needed to compute the result subvolume. For hyperrectangular partitions, operations like `linalg.generic` can simply be cloned for the new operands.

Algorithm 1: Generating code for hyperrectangular partitions.

```
%res' ← Uninitialized(dom %res);
foreach disjoint hyperrect out_rect in I_x do
    in_rects ← range(M ∩_dom out_rect)²;
    %op:N' ← ExtractSlice(%op:N,in_rects);
    %part ← CloneOps(%op:N');
    %res' ← InsertSlice(%part,%res',out_rect);
end
return %res';
```

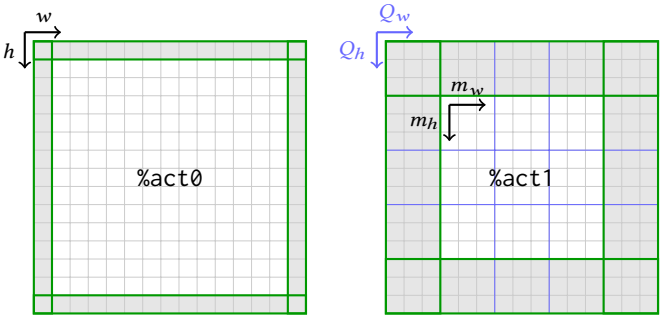
Suppose we want to compute I_D using a tiled implementation. Using the tiled volume element map we constructed earlier, we change the tile indices q_i into parameters

```
[QH, QW] -> { act1[n, f, qh = QH, mh, qw = QW, mw]
  -> ifm[n, c, y, x]
    : 0 <= mh <= 15 and 0 <= mw <= 15
      and y >= 32QH + 2mh - 1 and 0 <= y <= 511
      and y <= 3 + 32QH + 2mh
      and x >= 32QW + 2mw - 1 and 0 <= x <= 511
      and x <= 3 + 32QW + 2mw }
```

We reject all tiles using I_c, i.e., all tiles on the edge. This becomes a bound on the parameter domain

```
[QH, QW] -> { : 0 < QH < 15 and 0 < QW < 15 }
```

If the parameter domain has multiple disjuncts, we can recursively apply the partitioning techniques. Figure 3b shows the new coordinate system and the tiles in each partition.



(a) Element-wise **(b)** Tiled

Figure 3. Output of the convolutional layer from fig. 1.

Algorithm 2 outlines how code can be generated for such a tiling. The parameters Q_i are the induction variables of our tile loops, which are constant within an iteration. As a result, we can match index domains in $\mathcal{M}$ to offset, stride and size of slices (cf. def. 4.9)In our example, the slice offset in `%ifm` is $[0, 0, 32Q_h - 1, 32Q_w - 1]$. The `InsertSlice` and `ExtractSlice` functions are simply constructors for these MLIR `tensor` ops, which take place at runtime.

Algorithm 2: Generating tiled loops.

```
M' ← tile_map ∘ M;
%res' ← Uninitialized(dom %res);
foreach tile dim pair q_i, m_i in tile_map do
    iv ← CreateAndEnterLoop(min Q_i to max Q_i);
    iv_dom ← [Q_i = iv] → {[..., q_i = Q_i, ...]};
    M' ← M' ∩_dom iv_dom;
end
in_rects ← MatchSlice(range M');
out_rect ← MatchSlice(dom M');
%op:N' ← ExtractSlice(%op:N,in_rects);
%tile ← CloneOps(%op:N');
%res' ← InsertSlice(%tile,%res',out_rect);
return %res';
```

The advantage of the volume approach is that it enables reasoning about arbitrarily large fused programs. However, since the model does not carry information about the operations that are performed on volume elements, generating fused code requires additional work. `linalg` operations offer great simplicity through their implicit iteration domains, and

[2] $A \underset{\text{dom}}{\cap} B$ is the intersection of A with $\{B \mapsto \text{range } A\}$

their bodies define polyhedral model statements. To support fusion with operations like `tensor.pad` without partitioning, a polyhedral scheduling algorithm can be used.

5.4 Pitfalls

Following the directions given in section 4.2.1, $\mathcal{M}_1$ would contain a disjunct $\{\texttt{act0[i]} \mapsto \texttt{cst0[]}\}$. This leads to an undesirable $\mathcal{M}_{\texttt{krnl0}}$, in which the use of `%cst0` in ③ shadows the one that defines the sparsity in ①. In other words, the choice of def-use subgraph matters, and it depends on the pattern to be applied, implying that it must be already known.

This problem is an artifact of treating all volumes as assumed-virtual tensors (cf. section 2.3). This was done so that we might potentially achieve a model usable independent of this ordering problem. Indeed, this can be achieved by splitting the SSA values. Conceptually, we rewrite the SSA program such that each definition has up to one use only. Alternatively, this happens virtually, by assigning each def-use edge a unique volume identifier. However, this breaks the previous $1:1$ correspondence between volume elements and statement instances, making reuse in code generation harder. Alternatively, the behavior of other virtual tensor compilers can be replicated by cloning only those values that do not entail recomputation, e.g., the constants as above.

5.5 MLIR Presburger

As stated earlier, a focus of this work was to reuse existing software frameworks to perform the reasoning. While lacking in features compared to ISL, the Fast Presburger Library, which is the implementation of MLIR Presburger, is readily accessible in an MLIR project. To perform the operations described in this section, only minor modifications were needed. This is mostly due to the API being incomplete or restricting access to some required internals, and MLIR specializations overriding behavior by shadowing.

One example of an extension implemented on top of MLIR Presburger is the previously mentioned `MatchSlice` function. Given an affine set as coefficients C of its constraint system, algorithm 3 attempts to recover the offset and stride vectors of the slice it represents. In a typical application, local invariants such as tile indices are promoted to parameter constants. The `MatchSlice` then attempts to recover affine expressions for each dimension describing the start offset constant, uniform stride between elements and number of elements.

Algorithm 3: Matching offset and stride of a slice.

```
foreach set dimension d of C do
    of ← lower bound on d;
    eq ← single equality involving d;
    if no eq found then
        sd ← 1;
    end
    else
        of ← constant part of eq;
        loc ← single local var in eq;
        if no loc found then
            sd ← 0;
        end
        else
            sd ← coeff on loc in eq;
            locOf, locSd ← GetOfSd(domain of loc);
            of ← of + sd · locOf;
            sd ← sd · locSd;
        end
    end
end
```

6 Conclusion and future work

Volumes generalize over array and tensor structures. In this paper, we show how volumes simplify the polyhedral data flow analysis. We describe how dynamic shape inference, fine-grained affine pattern matching and generalized tiling and fusing of operations in the input program can use this information. For a simple convolutional layer example with asymmetric padding, we describe how state-of-the-art compilers struggle to separate the different regions of the program, which may lead to sub-optimal performance. We demonstrate that our model can conveniently identify all kernel regions, i.e., dense, sparse, constant, and corners. This empowers specializing over these regions, e.g., by using sparse compiler optimizations for sparse regions or offloading them to a sparsity-aware hardware target and mapping dense regions to external library calls. In future work, we plan to use the MLIR presburger arithmetic [12] to implement our model in an MLIR compiler that targets heterogeneous systems. The hierarchical abstractions of the compiler will utilize the application (sparsity, shape) and system (CPU, GPU, accelerators) information to map regions to the most suited hardware targets.

Acknowledgments

We thank the anonymous reviewers for their invaluable input towards the placement and improvement of this research. This work is partially funded by the EU Horizon 2020 Programme under grant agreement No 957269 (EVEREST) and the German Research Council (DFG) through the HetCIM project (502388442) under the Priority Program on 'Disruptive Memory Technologies' (SPP 2377).

References

[1] [n. d.]. MLIR linalg dialect. https://mlir.llvm.org/docs/Dialects/Linalg/. Accessed: 2022-11-23.

[2] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. 1999. *LAPACK Users' Guide* (third ed.). Society for Industrial and Applied Mathematics, Philadelphia, PA.

[3] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. 2019. Tiramisu: A polyhedral compiler for expressing fast and portable code. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 193–205.

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

[5] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008. A Practical Automatic Polyhedral Parallelizer and Locality Optimizer. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Tucson, AZ, USA) *(PLDI '08)*. Association for Computing Machinery, New York, NY, USA, 101–113. https://doi.org/10.1145/1375581.1375595

[6] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. {TVM}: An automated {End-to-End} optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 578–594.

[7] Paul Feautrier and Christian Lengauer. 2011. *Polyhedron Model*. 1581–1592. https://doi.org/10.1007/978-0-387-09766-4_502

[8] Tobias Grosser, Albert Cohen, Justin Holewinski, Ponuswamy Sadayappan, and Sven Verdoolaege. 2014. Hybrid hexagonal/classical tiling for GPUs. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*. 66–75.

[9] Tobias Grosser, Hongbin Zheng, Raghesh Aloor, Andreas Simbürger, Armin Größlinger, and Louis-Noël Pouchet. 2011. Polly-Polyhedral optimization in LLVM. In *Proceedings of the First International Workshop on Polyhedral Compilation Techniques (IMPACT)*, Vol. 2011. 1.

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

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

[12] Arjun Pitchanathan, Kunwar Shaanjeet Singh Grover, Michel Weber, and Tobias Grosser. [n. d.]. Bringing Presburger Arithmetic to MLIR with FPL. ([n. d.]).

[13] Louis-Noël Pouchet, Uday Bondhugula, Cédric Bastoul, Albert Cohen, Jagannathan Ramanujam, Ponnuswamy Sadayappan, and Nicolas Vasilache. 2011. Loop transformations: convexity, pruning and optimization. *ACM SIGPLAN Notices* 46, 1 (2011), 549–562.

[14] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. *SIGPLAN Not.* 48, 6 (jun 2013), 519–530. https://doi.org/10.1145/2499370.2462176

[15] Norman A. Rink and Jeronimo Castrillon. 2019. TeIL: A Type-Safe Imperative Tensor Intermediate Language. In *Proceedings of the 6th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming - ARRAY 2019*. ACM Press, Phoenix, AZ, USA, 57–68. https://doi.org/10.1145/3315454.3329959

[16] Stephanie Soldavini, Karl F. A. Friebel, Mattia Tibaldi, Gerald Hempel, Jeronimo Castrillon, and Christian Pilato. 2022. Automatic Creation of High-Bandwidth Memory Architectures from Domain-Specific Languages: The Case of Computational Fluid Dynamics. *ACM Trans. Reconfigurable Technol. Syst.* (sep 2022). https://doi.org/10.1145/3563553 Just Accepted.

[17] Mahdi Soltan Mohammadi, Kazem Cheshmi, Ganesh Gopalakrishnan, Mary Hall, Maryam Mehri Dehnavi, Anand Venkat, Tomofumi Yuki, and Michelle Mills Strout. 2018. Sparse matrix code dependence analysis simplification at compile time. *ArXiv e-prints* (2018), arXiv–1807.

[18] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2018. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *arXiv preprint arXiv:1802.04730* (2018).

[19] Sven Verdoolaege. 2010. Isl: An Integer Set Library for the Polyhedral Model. In *Mathematical Software – ICMS 2010*, David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, Komei Fukuda, Joris van der Hoeven, Michael Joswig, and Nobuki Takayama (Eds.). Vol. 6327. Springer Berlin Heidelberg, Berlin, Heidelberg, 299–302. https://doi.org/10.1007/978-3-642-15582-6_49

[20] Sven Verdoolaege and Tobias Grosser. 2012. Polyhedral extraction tool. In *Second International Workshop on Polyhedral Compilation Techniques (IMPACT'12), Paris, France*, Vol. 141.

[21] Sven Verdoolaege and Gerda Janssens. 2017. Scheduling for PPCG. https://doi.org/10.13140/RG.2.2.28998.68169

[22] Field G. Van Zee, Ernie Chan, Robert A. van de Geijn, Enrique S. Quintana-Ortí, and Gregorio Quintana-Ortí. 2009. The libflame Library for Dense Matrix Computations. *Computing in Science & Engineering* 11, 6 (2009), 56–63. https://doi.org/10.1109/MCSE.2009.207

[23] Jie Zhao and Albert Cohen. 2019. Flextended Tiles: A Flexible Extension of Overlapped Tiles for Polyhedral Compilation. *ACM Trans. Archit. Code Optim.* 16, 4, Article 47 (dec 2019), 25 pages. https://doi.org/10.1145/3369382

[24] Jie Zhao and Peng Di. 2020. Optimizing the Memory Hierarchy by Compositing Automatic Transformations on Computations and Data. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 427–441. https://doi.org/10.1109/MICRO50266.2020.00044

[25] Jie Zhao, Bojie Li, Wang Nie, Zhen Geng, Renwei Zhang, Xiong Gao, Bin Cheng, Chen Wu, Yun Cheng, Zheng Li, Peng Di, Kun Zhang, and Xuefeng Jin. 2021. AKG: Automatic Kernel Generation for Neural Processing Units Using Polyhedral Transformations. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 1233–1248. https://doi.org/10.1145/3453483.3454106

A Volume model

A.1 Volumes

Example. A volume U representing an $M \times N$ upper-right triangular matrix can be represented by

$$\operatorname{dom} U := \{[i, j] : 0 \leq i < M \wedge i \leq j < N\}$$

which encodes its sparsity property by definition.

We could define a dense array A as

$$A\left[iN - \frac{1}{2}(i-1)\,i + j\right] = U[i, j]$$

but not without involving a quasipolynomial expression. This precludes further reasoning using Presburger algebra.

Suppose $R \leftarrow U \odot V$[3], where V is an upper-left triangular matrix of the same shape, then Presburger algebra can infer the sparsity of the result R

$$\begin{aligned}\operatorname{dom} R &= \operatorname{dom} U \cap \operatorname{dom} V \\ &= \{[i,j] : 0 \le i < M \wedge i \le j < (N-i)\}\end{aligned}$$

A.2 Operations

A.2.1 View. The *extraction* of a subvolume from a supervolume has already been defined

$$\mathcal{M}_{\text{ext},\Pi} := \Pi$$

The *insertion* of a subvolume V_{sub} into a supervolume V_{sup} is an index-based selection

$$\begin{aligned}\mathcal{M}_{\text{ins},\Pi} := &\left\{R[\mathbf{r}] \mapsto V_{\text{sup}}[\mathbf{r}] : \mathbf{r} \notin \operatorname{Im}\Pi\right\} \\ &\cup \left\{R[\mathbf{r}] \mapsto V_{\text{sub}}\left[\Pi^{-1}(\mathbf{r})\right] : \mathbf{r} \in \operatorname{Im}\Pi\right\}\end{aligned}$$

where $\operatorname{Im}\Pi = \Pi(\operatorname{dom}\Pi)$, and Π^{-1} is the inverse of Π.

A.2.2 Reduction. A *reduction* $R \leftarrow \text{red}_{\oplus,j}\, O$ reduces the rank of volume O by applying a reduction $\oplus$ along an axis j

$$R[\mathbf{r}] = \bigoplus_k O\left[r_1, \ldots, r_{j-1}, k, r_j, \ldots, r_N\right]$$

$$\mathcal{M}_{\text{red},j} := \left\{R[\mathbf{r}] \mapsto O\left[r_1, \ldots, r_{j-1}, k, r_j, \ldots, r_N\right] : k \in \mathbb{Z}\right\}$$

A.2.3 Diagonal. The $s, t : s < t$ *diagonal* R of a volume O is a view into O where the indices at s and t are equal

$$\mathcal{M}_{\text{diag},s=t} := \{R[\mathbf{r}] \mapsto O[r_1, \ldots, r_s, \ldots, r_{t-1}, r_s, r_t, \ldots, r_N]\}$$

A.2.4 Outer product. The *outer product* $R \leftarrow A \otimes B$ and its volume element map are given by

$$R[\mathbf{a} +\!\!+ \mathbf{b}] = A[\mathbf{a}]\, B[\mathbf{b}]$$

$$\mathcal{M}_{\otimes} := \{R[\mathbf{a} +\!\!+ \mathbf{b}] \to A[\mathbf{a}]\} \cup \{R[\mathbf{a} +\!\!+ \mathbf{b}] \to B[\mathbf{b}]\}$$

where $\mathbf{a} +\!\!+ \mathbf{b} = [a_1, \ldots, a_N, b_1, \ldots, b_M]$, i.e. a concatenation of the index tuples.

A.2.5 Inner product. An *inner product* $R \leftarrow A \cdot B$ is a contraction of $A \otimes B$. A *contraction* is a reduction over a diagonal. The diagonal is ambiguous for a combined rank > 2. For consistency with matrix-matrix multiplication, we use the diagonal over the straddling adjacent dimensions, and so

$$\mathcal{M}_{\cdot} := \mathcal{M}_{\text{red},N} \circ \mathcal{M}_{\text{diag},N=N+1} \circ \mathcal{M}_{\otimes}$$

where N is the rank of volume A.

[3] $\odot$ is the element-wise product

A.2.6 Padding. Adding a boundary around an array A using values from a volume B, selected based on index, is called *padding*

$$\mathcal{M}_{\text{pad},\mathbf{lo},\mathbf{hi},\Pi} := R[\mathbf{r}] \mapsto \begin{cases} A[\mathbf{r} - \mathbf{lo}] & \mathbf{lo} \le \mathbf{r} < (\Delta_\square R - \mathbf{hi}) \\ B[\Pi(\mathbf{r})] & \text{otherwise} \end{cases}$$

where $\mathbf{lo}$ and $\mathbf{hi}$ indicate the size of the boundary in all dimensions, $\Pi : \operatorname{dom} R \to \operatorname{dom} B$ and $\Delta_\square R = \Delta_\square A + \mathbf{hi} + \mathbf{lo}$.

Example. To apply a 3×3 stencil to a scalar field F, we introduce a periodic boundary of radius 1. We note that $\mathbf{lo} = [1, 1]$ and $\mathbf{hi} = [1, 1]$.

To achieve a periodic boundary, we set $B \leftarrow F$, and define

$$\Pi(\mathbf{i}) \mapsto \mathbf{i} - \mathbf{lo} \mod \Delta_\square F$$

where the modulo is applied element-wise.

A.2.7 Convolution. A *convolution* on arrays $R \leftarrow I * W$ with strides $\mathbf{sd}$ and dilations $\mathbf{dl}$ is modeled by

$$\begin{aligned}\mathcal{M}_* := &\{R[\mathbf{r}] \mapsto I[\mathbf{sd} \odot \mathbf{r} + \mathbf{dl} \odot \mathbf{w}] : \mathbf{w} \in \operatorname{dom} W\} \\ &\cup \{R[\mathbf{r}] \mapsto W[\mathbf{w}]\}\end{aligned}$$

which assumes all dimensions are treated equally. Note that the bounds on $\mathbf{w}$ in the first disjunct are explicit because they are not in the implicit context of that map.

A.3 Programs

Lemma A.1. *The volume element map of an SSA operation is transitively closed.*

Proof. Let the element-wise dependencies of an SSA operation O be given by its volume element map $\mathcal{M}_O$.

By definition, each volume is an SSA value. Due to the SSA property, the def-use graph is acyclic, and thus a value cannot be involved in its own definition. Therefore, range $\mathcal{M}_O \cap \operatorname{dom} \mathcal{M}_O = \emptyset$, and $\mathcal{M}_O \circ \mathcal{M}_O = \emptyset$. □

Lemma A.2. *A transitively closed map is a fixpoint under lenient composition with itself.*

Proof. Let A be transitively closed, i.e., $A = \bigcup_{N=1} A^N$.

Expanding the lenient composition gives

$$\begin{aligned}X &= A \circ^{\text{id}} A \\ &= A \circ (A \cup \text{id}(\text{range}\, A \setminus \operatorname{dom} A)) \\ &= (A \circ A) \cup (A \circ \text{id}(\text{range}\, A)) \\ &= A^2 \cup A\end{aligned}$$

And equality follows

$$\begin{aligned}X &\subseteq A \cup A^2 \cup \bigcup_{N=1} A^N \\ A &\subseteq A \cup A^2\end{aligned}$$

□

Lemma A.3. *The volume element map of a program P consisting of assignments $A \in P$, live-out definitions O and live-in uses I is given by*

$$\mathcal{M}_P = \left(\bigcup_{A \in P} \mathcal{M}_A\right)^+ \cap \{o \mapsto i : o \in O, i \in I\}$$

Proof. Let $P^+ = \bigcup_{A \in P} \mathcal{M}_A$ and $a_i \leftarrow A(x_j) : A \in P$. Due to the SSA property, every value and thus volume has exactly one definition.

If $x_j \notin I$, then its definition $x_j \leftarrow X(y_k)$ must be in P, and $\{a_i \mapsto y_k\} \subset P^+$. Suppose $\nexists i \in I : \{a_i \mapsto i\} \subset P^+$, then a_i must be a live-in definition, and therefore $A \notin P$, which is contradictory. □

Lemma A.4. *The volume element map of the composition of two SSA operations is the lenient composition of their volume element maps.*

Proof. Let A and B be SSA operations, and $A \circ B$ only define the outputs of A.

Since all unused outputs of B will not appear in the volume element map, we write

$$\mathcal{M}_{A \circ B} = (\mathcal{M}_A \cup \mathcal{M}_B)^+ \cap \{O_A \rightarrow I_B \cup I_A \setminus O_B\}$$

where the intersection with the context ensures only live-in and live-out values appear in the map.

Both maps are transitively closed, and due to SSA $\mathcal{M}_A^2 = \mathcal{M}_B^2 = \emptyset$. We can simplify the transitive closure to

$$\mathcal{M}_{A \circ B} = (\mathcal{M}_A \circ \mathcal{M}_B \cup \mathcal{M}_A \cup \mathcal{M}_B) \cap \{O_A \rightarrow I_{A \circ B}\}$$

The intersection of $\mathcal{M}_B$ with the context is trivially empty, while $\mathcal{M}_A \circ \mathcal{M}_B$ is a strict subset of the context. We obtain

$$\begin{aligned} \mathcal{M}_{A \circ B} &= (\mathcal{M}_A \circ \mathcal{M}_B) \cup \left(\mathcal{M}_A \setminus_{\text{rg}} O_B\right) \\ &= \mathcal{M}_A \circ (\mathcal{M}_B \cup \text{id}\,(\text{range}\,\mathcal{M}_A \setminus \text{dom}\,\mathcal{M}_B)) \\ &= \mathcal{M}_A \circ^{\text{id}} \mathcal{M}_B \end{aligned}$$

□

Lemma A.5. *The volume element map of a program is the lenient composition of all its assignments' volume maps in ALAP use-def order.*

Proof. Let P be a program with live-in values I and live-out values O. Let $G_0 = \emptyset$ and $\mathcal{M}_{G_0} = \text{id}\,O$. Mark all live-out edges of the use-def graph (using virtual nodes $\notin P$) as visited.

While there are still nodes (assignments) not visited, select any assignment A_{i+1} where all outgoing edges are marked as visited. Since the use-def graph is a DAG, this is always possible. Suppose candidate A had an unvisited outgoing edge to $B \notin P$, and could thus never be visited. Then, that edge would be in O, and must therefore have been marked during initialization. Suppose candidate A had an unvisited outgoing edge to $B \in P$, then B would need to be visited first. A deadlock can only occur when a path $B \rightarrow A$ exists, which violates the acyclic property.

Visit the selected assignment A_{i+1}, computing

$$G_{i+1} = G_i \cup \{A_{i+1}\}$$
$$\mathcal{M}_{G_{i+1}} = \mathcal{M}_{G_i} \circ^{\text{id}} \mathcal{M}_{A_{i+1}}$$

and marking all incoming edges. Since all outgoing edges of A_{i+1} are marked before it is visited, all its uses are already part of $\mathcal{M}_{G_i}$, i.e., $\nexists A_j \in P : A_j \notin G_i$, range $\mathcal{M}_{A_j} \cap$ dom $\mathcal{M}_{A_{i+1}} \neq \emptyset$, so it must not be visited again.

When $G_i = P$, terminate with $\mathcal{M}_P = \mathcal{M}_{G_i}$. Because all nodes are visited in this manner, range $\mathcal{M}_{G_i} = I$. Suppose there was a value $V \in$ range $\mathcal{M}_{G_i} : V \notin I$, defined by an assignment A_V. Since all incoming nodes of A_V were visited after it, V can not be a live-out of any predecessor. For V to be a live-out of A_V, $\{V \mapsto V\} \subset \mathcal{M}_{A_V}$ must be true, which violates the acyclic property. □