

Dimension Analysis by Union-Find over Index Space: Flattening, Reshape Propagation and Operator Fusion in HFusion

Internal Technical Report

Hocky Yudhiono
Huawei Technologies
hocky.yudhiono@huawei.com

October 2026

Abstract

Tensor compilers for vector accelerators repeatedly face the same structural problem: *aliasing* operations such as `tensor.expand_shape` and `tensor.collapse_shape` sit in the middle of operator chains, tensors carry more dimensions than the hardware benefits from, and every downstream optimization (fusion, vectorization, tiling) has to reason about axes that are logically one loop but syntactically many. This report describes three techniques in the HFUSION pipeline of the AscendNPU-IR compiler that address this problem from one shared viewpoint: *shape facts are an equivalence relation over index positions*. (i) The *Dimension Analyzer* assigns a node to every tensor dimension of every allocated value, relates nodes with union-find structures, and starts from the optimistic assumption that all adjacent dimensions can be merged; it then removes merge links wherever an operation (reduce, broadcast, transpose, slice, ...) or a function boundary distinguishes two neighboring axes. The surviving components yield one globally consistent collapse for the whole function, which the *tidy flattener* realizes with reshapes only at the function boundary. (ii) *PropagateReshape* removes aliasing reshape operations from the interior of operator chains by sinking collapses and hoisting expansions. (iii) *OpFusion* forms fusion groups over a topological order of the cleaned-up graph. We describe the algorithms, their invariants and complexity, and illustrate them on running examples taken from the compiler's regression tests. The report presents the designs and their mechanisms; performance measurements are left to a follow-up study.

1 Introduction

Deep-learning graphs reach a kernel compiler with shapes that reflect the *framework's* view of the computation, not the hardware's. Activations are viewed as `[batch, heads, seq, dim]`, reshaped to `[batch*heads, seq, dim]`, transposed, reshaped back, and so on. Most of these reshapes are *aliasing*: they change how a buffer is indexed, not what it contains. In MLIR's tensor dialect they appear as `tensor.expand_shape` and `tensor.collapse_shape` [5]. Three problems follow when such operations are left in the middle of an operator chain.

1. **They block fusion.** A fusion pass that groups producer-consumer chains must either stop at a reshape or reason about two different iteration spaces inside one fused region.
2. **They hide structure.** Two adjacent axes that every operation in a function treats identically are, logically, one loop; but the IR shows two, and with unit dimensions or broadcast-expanded operands it often shows many more.
3. **They inflate rank.** Higher rank means more loop levels, more index arithmetic and more strided access descriptors on a vector accelerator whose throughput comes from long contiguous runs.

The three obvious responses are each local. A per-operation flattener collapses the loops of one operation without knowing what its neighbors do, so it inserts reshapes between operations whose choices disagree. A reshape-folding pattern removes a reshape next to one specific operation kind. A fusion pass that tolerates reshapes pushes the complexity into the fused region. What is missing is a *global* account of which axes in the function are the same axis and which neighboring axes are never separated by anything.

Central observation. Every shape-related question the compiler asks has the form “are these two dimension positions the same, or are they adjacent and inseparable?”. Both are equivalence-style relations over *index positions*, and equivalence relations over a growing universe are exactly what union–find structures maintain in near-linear time [1, 2, 3]. This report builds three optimizations on that observation.

Contributions.

1. **Dimension Analyzer** (Section 3). A formulation of dimension analysis as union–find over a flat *index space* in which every tensor dimension of every allocated value is a node and every allocation is closed by a barrier node. The analysis is *optimistic*: all adjacent dimensions start connected, and only links are ever removed—at reduces, broadcasts, transposes, slices, mutated axes and function edges. Two union–find structures separate “same extent” from “same axis role”, which makes concatenation, padding and slicing expressible without losing precision.
2. **Tidy flattening** (Section 4). A flattener that consumes the analysis and applies *one* consistent collapse to the whole function, so that reshapes occur only at its entry and exit, in contrast to the per-operation *greedy* mode.
3. **PropagateReshape** (Section 5). A set of rewrite patterns that eliminate aliasing operations from the interior of operator chains by moving collapses toward the outputs and expansions toward the inputs.
4. **OpFusion** (Section 6). A fusion pass that forms groups by walking a topological order of the graph once reshapes have been propagated out of the way.

All four are implemented in the open-source AscendNPU-IR compiler. Examples in this report are taken from its regression tests; we state explicitly which snippets are verbatim test expectations and which are derived by hand. We have not measured performance in this report (Section 8 lists the planned experiments), and we are careful to separate what the code does from what we expect it to buy.

Organization. Section 2 fixes notation. Sections 3 and 4 present the analyzer and the tidy flattener, Sections 5 and 6 present the other two techniques, Section 7 shows how they compose, and Section 10 records limitations honestly.

2 Background and Problem Statement

Tensors, reshapes and reassociation. We work on MLIR-style tensor programs [5]. A `tensor.collapse_shape` merges groups of *adjacent* dimensions; the grouping is a *reassociation* such as `[[0],[1,2]]`, which turns `5x6x6` into `5x36`. A `tensor.expand_shape` is the inverse. Both are aliasing: no element moves. Merging dimensions i and $i+1$ is only legal if no consumer needs to address them separately.

Operations that distinguish axes. Elementwise operations relate the axes of all their operands position by position. Other operations *separate* neighboring axes: *reduce* removes some axes and keeps others; *broadcast* adds axes; *transpose* reorders them; *slices* take a sub-range of an axis, which also breaks contiguity of the axes to its left; *concat*, *pad*, *cumulative* operations and *gather* change the extent or meaning of a single axis (“mutate” it). Function arguments and results are a boundary: the caller fixes their shapes.

Problem. Given a function, choose for every tensor value a reassociation such that (a) every operation can be rewritten to act on the collapsed shapes with the same semantics, (b) the choices for all values are mutually consistent, so no reshape is needed between two operations in the interior, and (c) rank is as small as the constraints allow. Our solution computes the *coarsest* consistent collapse by starting from “everything adjacent is mergeable” and removing what the operations forbid.

Notation. We write $\text{rank}(t)$ for the number of dimensions of a tensor value t and $d_k(t)$ for its k -th dimension. A *union-find* structure maintains a partition of $\{0, \dots, N-1\}$ with FIND and JOIN; with union by size and path compression both run in amortized $O(\alpha(N))$ time [2, 3].

3 The Dimension Analyzer

The Dimension Analyzer answers one question for a whole function: *which tensor axes are the same axis, and which neighboring axes can never be told apart by any operation?* Its answer is a set of union-find structures over an *index space* plus a *connectivity flag* per node. This section defines both, gives the per-operation rules, and walks the running example through the algorithm. The consumer that turns the answer into rewritten IR is the tidy flattener (Section 4).

3.1 From shape reasoning to index space

A per-operation flattener looks at one operation and picks a collapse of its loops. Two operations that share a tensor may pick different collapses, and every disagreement costs a reshape. The Dimension Analyzer instead gives *every tensor dimension of every allocated value* a node in one flat index space, and expresses every constraint as a relation between nodes.

Nodes. For each *allocation*—a tensor-typed block argument, or the result of a “head” operation (`tensor.empty`, `arith.constant`, `bufferization.to_tensor`, or a reshape)—a block of rank + 1 consecutive ids is reserved. The first rank ids are the dimension nodes, in axis order. The last id is a *barrier* node, which is never joined to anything and has both connection flags cleared (Figure 1). Consequently, in the flat index space two nodes are adjacent only if they are consecutive axes of the same allocation, and no group of nodes can straddle two tensors.

Views. Not every value is an allocation. An elementwise result *aliases* the view of its operands; a reduce result is a new view that lists the operand’s surviving nodes; a transpose result is a permutation of the operand’s view. A *view* is the ordered list of node ids a value’s dimensions refer to. Derived views reuse existing node ids, so analyzing a long chain of operations adds union-find joins but no new nodes. Reshape operations, `func.return` and `materialize_in_destination` are *scope barriers*: they delimit the region the analysis may reason about and are not joined through.

3.2 Two relations, one set of flags

The analyzer maintains two union-find structures over the same nodes (Figure 2).

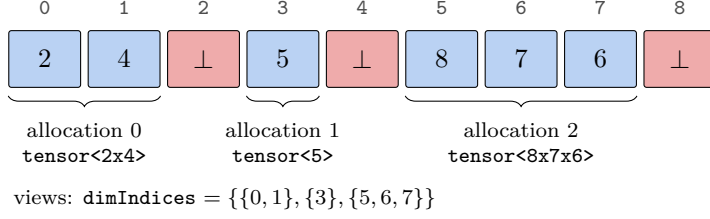


Figure 1: **The index space.** Every tensor dimension of every allocated value (a function argument or a “head” operation result such as `tensor.empty`) gets one node; consecutive node ids are consecutive axes. Each allocation is closed by a *barrier* node (red) that is never joined and has both connection flags cleared, so no merge can ever span two allocations. A *view* is the ordered list of node ids a value’s dimensions refer to; many values share one view, and values derived by reduce, broadcast or transpose get new views over *existing* node ids rather than new nodes. (Layout from the example in the analyzer’s initializer.)

Equivalent extent, $\mathcal{E}$. Two nodes are related when the compiler can prove they have the same extent. This structure carries the static size (or “dynamic”) of each class and, for a dynamic class, the earliest defining site (allocation, axis), so that a dynamic extent can later be rebuilt with a single `tensor.dim`.

Structural, $\mathcal{S}$. Two nodes are related when they play the *same axis role*, even if extents differ. $\mathcal{S}$ is coarser than $\mathcal{E}$. It is joined by every join that $\mathcal{E}$ sees, plus the joins for axes whose extent changes (concatenation, padding, slicing).

Keeping both is what lets one framework treat `concat` naturally. The concatenated axis of $a:3$, $b:5$ and $r:8$ is a single logical axis, so it must be cut or merged in unison across the three tensors ($\mathcal{S}$), but their extents must never be unified ($\mathcal{E}$). All connectivity state lives on $\mathcal{S}$.

Connectivity state. Each node n carries two flags $L(n), R(n) \in \{\text{true}, \text{false}\}$, initially true for dimension nodes, and an *element kind* $\kappa(n) \in \{\text{HasMutation} \prec \text{NoMutation} \prec \text{Unit}\}$: Unit for an extent-1 axis, NoMutation otherwise, and HasMutation for an axis an operation changes the meaning of (concatenated, padded, cumulatively scanned, gathered, ...). Two positions $i-1$ and i of a view w may be merged into one collapse group iff

$$R(w[i-1]) \wedge L(w[i]) . \quad (1)$$

The analysis never sets a flag back to true. It is *optimistic*: assume the whole function is one big contiguous loop nest, and let each operation take away what it cannot allow.

3.3 Per-operation rules

Table 1 lists what each operation family contributes. Three primitives appear: `joinShape( $a, b$ )` joins in both $\mathcal{E}$ and $\mathcal{S}$; `joinCollapser( $a, b$ )` joins in $\mathcal{S}$ only; `disconnect( $a, b$ )` clears $R(a)$ and $L(b)$, on the nodes and on their $\mathcal{S}$ roots. Operations are visited in pre-order and, for every tensor operand, the operation’s rule is applied; since joins commute, the visiting order affects only which view a value happens to alias first, not the resulting partition.

3.4 Cutting at view boundaries

Many cuts need no explicit rule. Once all joins are in, every view is checked by a single *view-boundary rule* (procedure MARKBROKEN in Algorithm 1): for each maximal run of consecutive node ids $[n, \dots, m]$ in a view, the pair to the left of n and the pair to the right of m are cut, both in the view’s neighbors and in the global neighbors $(n-1, n)$ and $(m, m+1)$. Equivalently:

Operation	Relation added	Cuts / element kind
Elementwise (parallel <code>linalg/hfusion</code> : <code>map</code> , <code>fill</code> , <code>cast</code> , <code>select</code> , <code>generic</code>)	All equal-rank operands and results alias one view, or are <code>joinShaped</code> position by position. Indexing maps are not consulted; scalar operands are skipped.	None.
<code>broadcast</code> , dims D	Input view = output view minus D (a new view over existing nodes), merged with the input's own view by <code>joinShape</code> .	<code>disconnect</code> at every boundary between kept and dropped axes.
<code>reduce</code> , dims D	Output view = input view minus D , merged with the init.	Same boundary cuts; an axis read by <code>linalg.index</code> in the body is cut from both neighbors.
<code>transpose</code> , π	Output view is a permutation of the input view; <i>no join</i> (node ids are reused).	Cut between output positions $j, j+1$ unless $\pi^{-1}(j+1) = \pi^{-1}(j) + 1$ (Figure 3).
<code>matmul</code> , <code>batch_matmul</code>	Output view = $[M, N]$ from the operands (K absent); batch variants add the batch axes.	Boundary cuts between the surviving axes of different operands.
<code>concat</code> , dim d	Axes $\neq d$: <code>joinShape</code> . Axis d : <code>joinCollapser</code> .	Axis d is <code>HasMutation</code> .
<code>pad</code>	Axes with zero padding: <code>joinShape</code> ; padded axes: <code>joinCollapser</code> .	Padded axes are <code>HasMutation</code> .
<code>extract_slice</code> , <code>insert_slice</code> , <code>subview</code>	Per axis <code>joinCollapser</code> (super, sub), skipping rank-reduced axes.	A contiguity mask over the super-view (full-extent inner axes, static strides) cuts group boundaries.
<code>cumsum</code> , <code>cumprod</code> ; <code>gather</code>	Non-scan (non-gather) axes: <code>joinShape</code> ; the scan (gather) axis: <code>joinCollapser</code> .	Scan / gather axis is <code>HasMutation</code> .
<code>flip</code> , <code>sort</code> , <code>interleave</code> , <code>embedding_gather</code>	Operands alias results; index axes joined to the leading result axes.	The affected axis is cut from both neighbors.
<code>scf.for</code> , <code>scf.if</code> , <code>scf.yield</code>	Region arguments, yielded values and results are aliased to their counterparts.	None.
<code>reshapes</code> , <code>func.return</code> , <code>materialize_in_destination</code>	Not joined through.	Head / tail scope barriers.
<code>func.call</code> (any- where)	—	The analysis aborts and the pass is a no-op.

Table 1: What each operation family contributes to the index-space relations. The first four rows carry the intuition: elementwise operations only *join*; reduce, broadcast and transpose *cut*; and operations that change an axis tag it `HasMutation`.

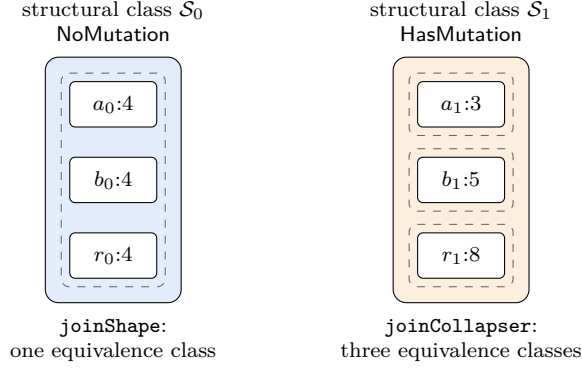


Figure 2: **Two relations over one index space**, shown for `r = tensor.concat(a, b) dim=1` with `a:4x3`, `b:4x5`, `r:4x8`. *Dashed boxes* are classes of the *equivalent-extent* relation $\mathcal{E}$ (“these axes have the same extent”, which carries static sizes and can reconstruct a dynamic size); *solid boxes* are classes of the *structural* relation $\mathcal{S}$ (“these axes play the same role”, even if their extents differ). Axis 0 is joined by `joinShape` in both relations. The concatenated axis is joined by `joinCollapser` in $\mathcal{S}$ only—extents 3, 5 and 8 must not be unified—and is tagged **HasMutation**. All cut flags live on $\mathcal{S}$ roots, so an axis that changes extent is still cut or merged consistently across every tensor it appears in.

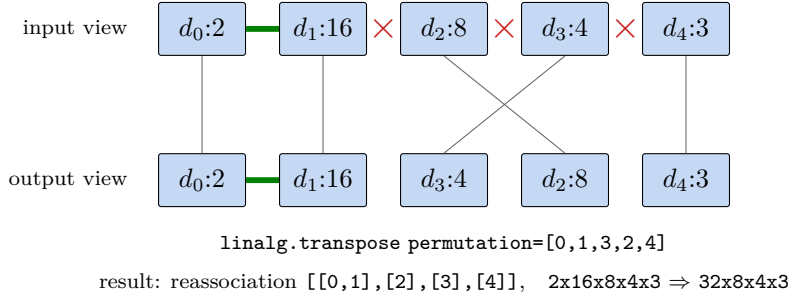


Figure 3: **The view-boundary rule on a transpose.** A pair of neighboring nodes stays connected only if *every* view that contains either of them contains both, adjacent and in the same order. Here d_0, d_1 keep their relative order and adjacency in both views (green link), so they may merge. In the output view d_1 is followed by d_3 , d_2 is followed by d_4 , and d_3, d_2 are reversed, so the pairs (d_1, d_2) , (d_2, d_3) and (d_3, d_4) are cut ($\times$). The collapsed shape `32x8x4x3` matches the expectation in the compiler’s tidy-flatten test for this permutation.

A global pair of neighboring nodes stays connected iff every view that contains either of them contains both, adjacent and in order.

This single rule is what separates function arguments from function results (the arguments’ views are whole allocations; a result view that skips or reorders axes breaks the run), reduce from broadcast (their views are proper sub-runs), and transposed axes from one another (Figure 3). The explicit `disconnect` calls of Table 1 cover only the cases that the view alone does not reveal, such as the axis a reduction body indexes.

3.5 Propagating cuts across classes

A cut recorded on one node must hold for every node of its structural class, otherwise the same logical pair would be merged in one tensor and kept apart in another. `PROPAGATEBROKEN` (Algorithm 1) therefore (1) takes the conjunction of the flags over each class and spreads it back to every member, with the element kind taking the minimum in `HasMutation` $\prec$ `NoMutation` $\prec$ `Unit`, and (2) scans the `HasMutation` classes: a mutated axis may share a group only with `Unit` neighbors

up to the next barrier or cut, so any non-unit neighbor is separated from it. The conjunction is repeated afterwards, because the scan may have introduced new cuts.

3.6 Worked example

Figure 4 traces the `complex_ops_1` case of the compiler’s tidy-flatten tests, in which a reduce, an elementwise add and a broadcast share one function. The input is:

```
func.func @complex_ops_1(%arg0: tensor<5x6x6xf32>, %arg1: tensor<6x6xf32>,
                        %arg2: tensor<5x6x6xf32>) -> (tensor<5x6x6xf32>, tensor<5x6x6xf32>) {
  %1 = add(%arg0, %arg2)                // elementwise
  %3 = reduce %arg0 dimensions = [0]    // 5x6x6 -> 6x6
  %5 = add(%3, %arg1)                  // elementwise on 6x6
  %7 = broadcast %5 dimensions = [0]    // 6x6 -> 5x6x6
  return %1, %7
}
```

The test expects `%arg0` and `%arg2` to be collapsed from $5 \times 6 \times 6$ to 5×36 , `%arg1` from 6×6 to 36, and the two results to be expanded from 5×36 back to $5 \times 6 \times 6$. (The listing above is abbreviated to the operations that matter; the test uses `linalg/hfusion` spellings.)

The analysis reaches this in four moves. **(1)** `ALLOCATE` creates nodes for the three arguments and the two `tensor.empty` results. **(2)** The two adds and the reduce/broadcast rules join the axes into three classes: *A* (extent 5, the leading axis of `arg0`, `arg2`, `%1`), *B* and *C* (extent 6, shared by every tensor in the function). The leading axis of the broadcast result is a fourth class: nothing in the function relates it to *A*. **(3)** `MARKBROKEN` sees the reduce output and the broadcast input as the view $[B, C]$, a run that starts at *B*, so it cuts $A|B$. **(4)** `PROPAGATEBROKEN` makes that cut hold in every tensor; nothing cuts $B|C$. The surviving groups are $\{A\}$ and $\{B, C\}$, giving the shapes above.

A negative example. Where every axis is separated, the analysis correctly does nothing. In the `broadcast_mul_reduce` case of the same test file a reduce and a broadcast over *different* axes of related tensors cut every adjacent pair, and the test asserts the absence of any `tensor.collapse_shape`. A per-operation flattener cannot tell this case from the example above without looking at the neighboring operations, which is the point of analyzing the whole function.

3.7 Properties

The following properties follow from the construction; we state them as design invariants rather than machine-checked theorems.

Property 1 (Isolation). *No collapse group spans two allocations.*

Reason. Barrier nodes are never joined and have both flags cleared, so Equation (1) is false at every allocation boundary.

Property 2 (Monotonicity). *Connectivity flags are only cleared, and element kinds only decrease.*

Reason. No procedure sets a flag to true after initialization. Hence each of the $2N$ flags changes at most once and every cut-propagation phase terminates. (The `HasMutation` scan is a single pass over class roots rather than a fixpoint; see Section 10.)

Property 3 (Class-uniform decisions). *After `PROPAGATEBROKEN`, all nodes of one structural class carry identical flags and kind.*

Reason. `ANDANDSPREAD` writes the class conjunction back to every member and runs last.

Algorithm 1 Tidy dimension analysis. All flags start true for dimension nodes; the procedures only ever clear flags.

```

1: procedure ANALYZE(function  $f$ )
2:   if  $f$  contains func.call then
3:     return ▷ conservative: pass is a no-op
4:   end if
5:   ALLOCATE( $f$ ) ▷ nodes for arguments and head ops; one barrier each
6:   for each operation  $o$  of  $f$  in pre-order do
7:     for each tensor operand  $v$  of  $o$  do
8:       PROCESS( $o, v$ ) ▷ Table 1
9:     end for
10:  end for
11:  for each view  $w$  do
12:    MARKBROKEN( $w$ )
13:  end for
14:  PROPAGATEBROKEN()
15:  if dynamic extents must not be fused then
16:    BREAKDYNAMICSHAPES(); PROPAGATEBROKEN()
17:  end if
18:  REWRITE( $f$ ) ▷ Section 4
19: end procedure

20: procedure PROPAGATEBROKEN
21:   ANDANDSPREAD()
22:   for each  $\mathcal{S}$ -root  $r$  with  $\kappa(r) = \text{HasMutation}$  do
23:     if not ONLYUNITSUNTILBARRIER( $r, -1$ ) then
24:       DISCONNECT( $r-1, r$ )
25:     end if
26:     if not ONLYUNITSUNTILBARRIER( $r, +1$ ) then
27:       DISCONNECT( $r, r+1$ )
28:     end if
29:   end for
30:   ANDANDSPREAD()
31: end procedure

32: procedure ANDANDSPREAD
33:   for each node  $n$  do
34:      $r \leftarrow \text{FIND}_{\mathcal{S}}(n)$ ;  $L(r) \leftarrow L(r) \wedge L(n)$ ;  $R(r) \leftarrow R(r) \wedge R(n)$ ;  $\kappa(r) \leftarrow \min(\kappa(r), \kappa(n))$ 
35:   end for
36:   for each node  $n$  do
37:     copy  $L, R, \kappa$  of  $\text{FIND}_{\mathcal{S}}(n)$  to  $n$ 
38:   end for
39: end procedure

40: procedure ONLYUNITSUNTILBARRIER(position  $p$ , direction  $\delta$ )
41:   while  $p$  and  $p+\delta$  are connected do
42:     if  $\kappa(p+\delta) \neq \text{Unit}$  then
43:       return false
44:     end if
45:      $p \leftarrow p + \delta$ 
46:   end while
47:   return true ▷ reached a barrier or a cut
48: end procedure

```

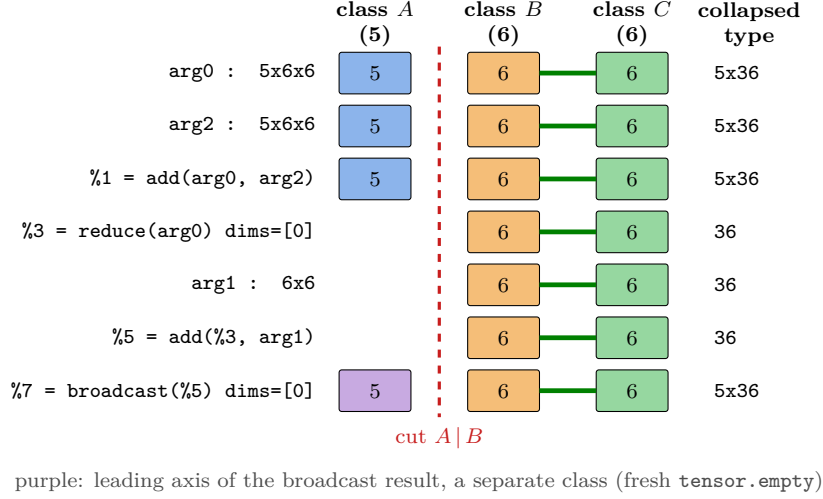


Figure 4: **Running example** (the `complex_ops_1` case of the tidy-flatten tests; shapes verbatim, the class bookkeeping is derived). The function returns `%1` and `%7`. Each box is one node of the index space; color is its structural class. The `adds` join dimensions position by position. The reduce output and the broadcast input are the views $[B, C]$, a run that starts at B rather than at the first node of its allocation, so the view-boundary rule cuts $A | B$. This cut is stored on the class, so it holds in *every* tensor, including those that never meet a reduce. Nothing separates B from C (green), so they merge everywhere: arguments $5 \times 6 \times 6$ become 5×36 and 6×6 becomes 36 . No interior value needs a reshape; the two results are expanded back to $5 \times 6 \times 6$ at the function boundary.

Property 4 (Global consistency). *Whether two neighboring positions of a view are merged depends only on the structural classes of the two nodes. Hence, if two views contain positions in the same pair of classes, they are grouped identically, and the operands of an elementwise operation receive identical collapsed shapes.*

Reason. Combine Equation (1) with the previous property. This is the property that removes interior reshapes: operations that share an axis never disagree about whether it is merged with its neighbor.

Complexity. Let N be the number of nodes ($\sum(\text{rank} + 1)$ over allocations) and E the number of operations. Joins and finds run in $O(\alpha(N))$ amortized. The join walk costs $O(\sum_o k_o^2 r \alpha(N))$ for operations with k_o operands of rank r , because each operand is processed with all operands re-bound; this is linear for bounded arity. View-boundary cuts and each `ANDANDSPREAD` are $O(N\alpha(N))$. Our implementation has two super-linear spots in the rewrite phase (Section 10); the analysis itself is near-linear in the size of the function.

3.8 Dynamic shapes and register-based targets

Fusing several dynamic extents into one group makes the collapsed extent a product of runtime values. When requested (and in all `HFusion` pipelines), `BREAKDYNAMICSHAPES` scans each maximal connected run left to right and cuts in front of every dynamic axis after the first, so that each resulting group contains at most one dynamic extent and its collapsed size is a static multiple of a single runtime value. For register-based targets, a further pass cuts sub-byte (`i1`) select masks that cross a function boundary when the inner extent is not a multiple of eight. Both passes only clear flags and are followed by another `PROPAGATEBROKEN`.

	Greedy	Tidy
Scope of a decision	one elementwise operation	the whole function
Cross-operation analysis	none	union-find over index space
Reduce / broadcast / transpose	not handled	handled by cut rules
Slices, concat, pad, scans, gather	not handled	mutation and contiguity rules
Reshapes inserted	wherever neighbors disagree	function entry and exit only

Table 2: The two flatten modes of `hfusion-flatten-ops`.

3.9 The same analyzer in other roles

The analyzer is a library, not only a flattening pass. The HIVM variant reuses the same base structures to choose tiling axes; its joins are *transactional*—buffered per operation and filtered so that no merge joins two axes of the same tensor—and dropped merges are remembered so that broadcast axes can still be tiled. The auto-scheduler reads the same union-find structures to compute an anchor axis for a kernel. We describe only the flattening role in this report.

4 Tidy Flattening

The analysis of Section 3 decides *what* may be merged. The tidy flattener (`hfusion-flatten-ops` with `flatten-mode=tidy`) applies that decision to the IR, and does so for the whole function at once.

4.1 Greedy versus tidy

The pass has two modes. *Greedy* is the default option value: it applies upstream MLIR’s iteration-space collapsing to each elementwise operation independently, with no analysis across operations and no support for reduce, broadcast or transpose. *Tidy* first runs the Dimension Analyzer over the entire function and then rewrites every operation consistently. The HFusion pipelines select tidy mode.

4.2 From flags to reassociation

For a value with view $w = [w_0, \dots, w_{r-1}]$, positions $i-1$ and i fall into the same collapse group iff $R(w_{i-1}) \wedge L(w_i)$ (Equation (1)). Reading the groups left to right yields a reassociation, for example $[[0], [1, 2]]$ for $5 \times 6 \times 6$. Because the flags are uniform on each structural class (Property 3), every value that shares a class gets the same grouping, and no per-operation negotiation is needed.

4.3 The rewrite

The rewrite has three parts, in this order.

FLATTEN-IN. Immediately after each allocation (function argument or head operation) a `tensor.collapse_shape` with the computed reassociation is inserted, unless it would be the identity. Uses are redirected to the collapsed value, except `tensor.dim` uses and uses by head/tail operations; the collapsed value is mapped to the same view as the original so that later steps see consistent information.

Operations. The operations are visited in pre-order from a worklist. Each is rebuilt on the collapsed operands, with result types taken from the collapsed destination operand and with its attributes remapped through the reassociation (Table 3).

Operation	Rewrite on collapsed operands
reduce	<code>dimensions</code> remapped to group indices; <code>linalg.index</code> in the body remapped.
broadcast	<code>dimensions</code> remapped through the output's groups.
transpose	Permutation rebuilt from the reassociation.
cumsum, cumprod,	Axis attribute remapped to the group containing it (the mutated axis is alone or joined only with unit axes).
concat, gather	Only the single mutated axis of each group contributes padding; others get zero.
pad	Offsets, sizes and strides recomputed per group: offsets become $\sum_i \text{offset}_i \cdot \text{cumSize}_i$; static sizes multiply.
extract_slice, insert_slice, subview	Indices linearized within each group.
tensor.extract, tensor.insert	

Table 3: How operations are rewritten after the groups are known.

FLATTEN-OUT. For every tail operation (a function result, a reshape, or a store into a destination operand) a `tensor.expand_shape` restores the original type at the function boundary. Dynamic sizes are recovered from the original slice sizes or from `tensor.dim` on the earliest defining axis recorded in $\mathcal{E}$ (Section 3.2). Afterwards reshape canonicalization and folding of `tensor.empty` clean up redundant reshapes. Running `fold-tensor-empty` *before* flattening avoids dynamic-shape barriers introduced by an expanded `tensor.empty`.

4.4 Examples

All shapes below are verbatim expectations from the tidy-flatten tests.

Broadcast over the last axis. A broadcast along the last axis of a 1024x1024x1024 result collapses the two leading axes:

```
%in  = tensor.collapse_shape %arg0 [[0,1]]      : 1024x1024      into 1048576
%out  = tensor.collapse_shape %empty [[0,1],[2]] : 1024x1024x1024 into 1048576x1024
%r    = linalg.broadcast ins(%in : 1048576) outs(%out : 1048576x1024) dimensions = [1]
%res  = tensor.expand_shape %r [[0,1],[2]]      : 1048576x1024 into 1024x1024x1024
```

(The remapped `dimensions = [1]` is derived from the remapping rule; the test checks the shapes.)

Mutated axes. A mutated axis stays alone unless its neighbors are unit axes. For a cumulative sum over 4x2x6x4x8: with `cum_dims=[2]` the result is 8x6x32 (groups `[[0,1],[2],[3,4]]`); with `cum_dims=[0]` it is 4x384 (groups `[[0],[1,2,3,4]]`). A gather over 2x4x64 with `axis=2` becomes 8x32 with `axis=1`.

Dynamic extents. For an elementwise chain over `?x?`, the default run collapses to `?`; the run with dynamic-shape splitting enabled leaves the two dynamic axes uncollapsed (Section 3.8).

Transpose and slices. The transpose of Figure 3 is rewritten between 32x8x4x3 operands. A slice that takes 24x128 out of 24x256 is not collapsed, because taking a sub-range of the inner axis breaks contiguity of the pair.

5 PropagateReshape

Tidy flattening (Section 4) is only as good as the region the analyzer can see. Reshape operations are *scope barriers* for the Dimension Analyzer: the result of a reshape starts a fresh dimension space, and a reshape in the middle of a function therefore splits the function into independent segments. PropagateReshape (`propagate-reshape`) removes that obstacle. It rewrites the function so that aliasing operations (`tensor.expand_shape`, `tensor.collapse_shape`) appear only at the function boundary, and the computation in between runs on the finest iteration space the reshapes imply.

5.1 Normal form

The target normal form has a simple statement. A reshape is *valid* (in the sense used by the pass's test oracle) iff

1. every user of the reshape is a non-compute operation (another reshape, a return, ...), or
2. the operation defining its source is a non-compute operation (an argument, `tensor.empty`, a constant, `bufferization.to_tensor`, or another reshape).

In words: *expansions bubble up toward the inputs; collapses sink toward the outputs*. The pass is one greedy application of a set of local rewrite patterns; it does not construct the normal form directly.

5.2 The core rules

R1: CollapseDown. A collapse feeding an elementwise operation is pushed below it. The operation is re-created on the pre-collapse shape, its destination and side operands are expanded, and its result is collapsed again (all other users of the old collapse keep it):

```
%c = tensor.collapse_shape %x [[0,1],[2],[3]] : 2x3x7x8 into 6x7x8
%y = linalg.elemwise_unary ins(%c) outs(%i : 6x7x8)
==>
%i2 = tensor.expand_shape %i [[0,1],[2],[3]] : 6x7x8 into 2x3x7x8
%y2 = linalg.elemwise_unary ins(%x : 2x3x7x8) outs(%i2)
%y = tensor.collapse_shape %y2 [[0,1],[2],[3]]
```

Operands of different rank (scalars) are untouched. The `expand(collapse(...))` pair this creates on a side operand cancels under reshape canonicalization.

R2: ExpandUp. The mirror image: an expand that follows an elementwise operation is hoisted above it, so the operation runs on the expanded operands; any other user of the old result receives a collapse.

R3: SwapCollapseExpand. A collapse immediately followed by an expand blocks both rules. If they are not exact inverses, the pair is rewritten through a *common refinement* of the two reassociations,

$$\text{expand}(\text{collapse}(x)) \implies \text{collapse}'(\text{expand}'(x)),$$

after which the expand can keep climbing and the collapse can keep sinking (Figure 5, row 2). Exact inverse pairs simply cancel.

R4: NearEndExpandDown. On the memory-based target, an expand whose users are all single-input elementwise operations that feed only the return is pushed *down* to the return instead of up. It runs at higher pattern benefit than ExpandUp, so the two never fight over the same operation: ExpandUp declines when all users are out-operations.

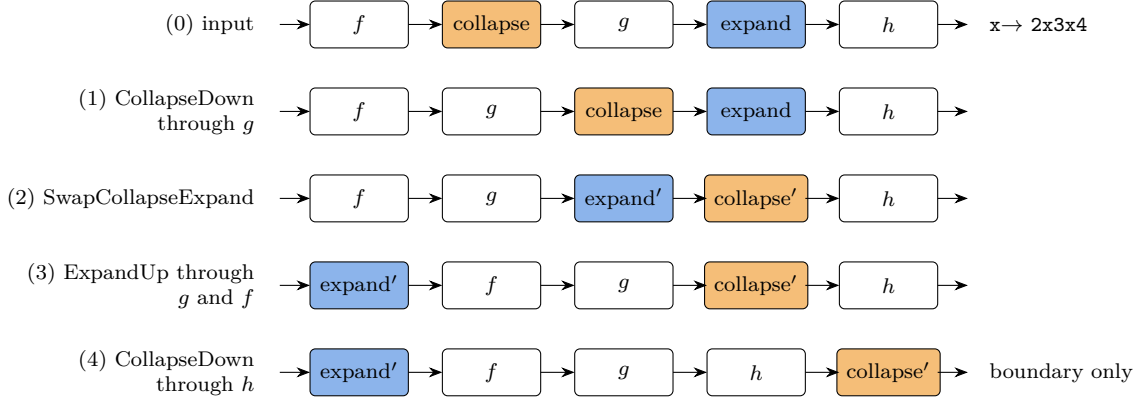


Figure 5: **Reshapes migrate to the boundary.** f, g, h are elementwise operations; *collapse* merges axes, *expand* splits them. Initially (row 0) $x:2 \times 3 \times 4$ is collapsed to 6×4 before g and expanded to $6 \times 2 \times 2$ before h , so g and h work on different iteration spaces. Collapses sink past elementwise operations (row 1). A collapse followed by an expand that is *not* its inverse is swapped through a common refinement (row 2): expand' : $2 \times 3 \times 4 \rightarrow 2 \times 3 \times 2 \times 2$, $\text{collapse}'$: $2 \times 3 \times 2 \times 2 \rightarrow 6 \times 2 \times 2$. The expand then climbs to the argument (row 3) and the collapse sinks to the result (row 4). In row 4 every interior operation sees the same iteration space and the only reshapes are at the boundary. (The swap in row 2 is derived by hand from the swap rule; the individual steps follow the rules described in the text.)

Operation	Collapse moves below	Expand moves above
elementwise	Re-create on X ; expand side operands and destination by G ; collapse the result.	Re-create on Z ; expand operands by H ; collapse other users.
reduce, dims D	$D' = \bigcup_{d \in D} G_d$; surviving groups $[G_i : i \notin D]$, renumbered, give the result's re-association; the init is expanded by it.	Reduced axes become placeholder groups; surviving axes take H 's groups; $D' =$ positions of the placeholders.
broadcast, dims B	Mirror of reduce: placeholders at broadcast positions; $B' =$ their positions.	$B' = \bigcup_{b \in B} H_b$; input reassociation is H minus the broadcast groups.
transpose, π	$\pi' = \text{concat}_i G_{\pi(i)}$; result groups are consecutive blocks of sizes $ G_{\pi(i)} $.	Symmetric, with π^{-1} .
extract_slice, insert_slice	Legal only if the 1-D slice over a collapsed group is a hyperrectangle in the expanded index space (below).	Same test, with expand and slice swapped.
concat, pad	One <i>candidate</i> axis in the group carries the concat (pad); all other axes of the group must be unit or divide every operand extent.	Same.
interleave, deinterleave	Last-axis extent scaled by the channel count; needs a static, divisible last axis.	Mirror.

Table 4: Reassociation algebra used when a reshape crosses an axis-sensitive operation. Reduce is worked through in Figure 6.

5.3 Reassociation algebra through other operations

Elementwise operations are shape-polymorphic, so moving a reshape across them is mechanical. The interesting cases are operations that give particular axes meaning. The rules are written in terms of the groups $G_0, \dots, G_{m-1}$ of a collapse $X \rightarrow Y$ (a partition of the dimensions of X into runs of adjacent axes) and $H_0, \dots, H_{m-1}$ of an expand $Y \rightarrow Z$. Groups are renumbered consecutively after groups are added or dropped; a placeholder group $\{-1\}$ stands for a new singleton axis.

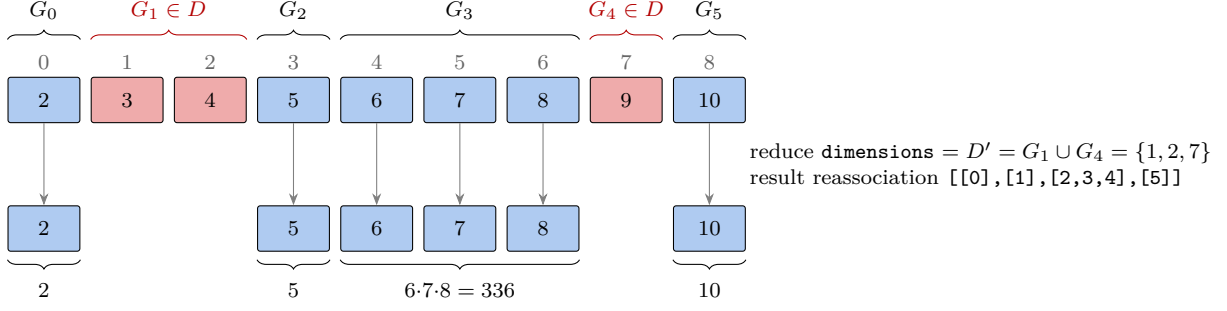


Figure 6: **Reassociation algebra for a collapse feeding a reduce** (collapse_reduce regression case). The original program collapses a $2 \times 3 \times 4 \times 5 \times 6 \times 7 \times 8 \times 9 \times 10$ value with groups $[[0], [1, 2], [3], [4, 5, 6], [7], [8]]$ and then reduces collapsed dimensions $D = \{1, 4\}$. Propagation moves the reduce *above* the collapse: it reduces the union of the groups named by D (red), and the surviving groups, renumbered, become the reassociation of the collapse that is re-applied to the result: $2 \times 5 \times 6 \times 7 \times 8 \times 10 \rightarrow 2 \times 5 \times 336 \times 10$. Both reduce dimensions and result grouping match the test’s expected output.

Slices and the hyperrectangle test. A slice over a collapsed group is a 1-D arithmetic progression of linear indices offset + $k \cdot \text{stride}$, $0 \leq k < \text{size}$. It can be moved across the collapse iff those indices, delinearized against the group’s expanded shape, form a hyperrectangle: the values in each expanded axis are an arithmetic progression of distinct values, and the product of the per-axis counts equals size. For example, a slice $[3] [6] [1]$ of the collapsed 12 (from 4×3) is the rectangle $[1, 0] [2, 3] [1, 1]$, and the slice moves; a slice starting at an unaligned offset is not a rectangle and the collapse stays where it is. Only static shapes are handled.

Strict versus loose reassociation. Swapping uses the strict compatibility check first (same number of groups, each group either collapsing or expanding but not both). If that fails, a loose two-pointer sweep over the source and result shapes computes a common refinement: equal extents emit one axis; if one extent divides the other the quotient is carried forward; otherwise the rewrite fails. Loose refinement can introduce extra unit axes, which motivates the termination guard below.

5.4 Boundaries, hardware-driven restrictions and termination

Where propagation stops. A reshape is not moved through a `tensor.empty` or constant (nothing to move it into), nor is an expand lifted when all its users are out-operations (reshapes, returns, `materialize_in_destination`). Regions are not crossed unless the single-use condition holds; `scf.for` is handled by expanding the yield operand, the iteration argument and the init together.

Hardware-driven restrictions. Propagation is not purely algebraic. An expand that only adds unit axes is *not* lifted through a load boundary (`bufferization.to_tensor`), because on the memory-based target doing so breaks stride alignment and can overflow the on-chip unified buffer; only expansions of non-unit axes become a “zero-cost” memref view on the load side. Similarly, a separate pass (`fold-collapse-into-alloc-with-load`) folds a unit-dimension collapse into the allocation that feeds a load, so that later data-layout inference sees the genuine matrix rank rather than a unit-padded scaffold.

Termination. The pass is a single greedy pattern application that fails if it does not converge. The informal measure is that every rewrite refines or cancels, and non-unit axes can only be refined finitely often. Unit axes, however, can be created again, so the register-based target

bounds the number of unit axes a propagation may introduce (default 10), and expansion through `linalg.fill` is disabled on the memory-based target after a non-termination was found. We do not claim confluence: the outcome can depend on pattern benefit and use-list order, and correctness is checked by the validity predicate of Section 5.1 rather than by a unique normal form.

5.5 Why it runs before flattening

The ordering is deliberate. The analyzer treats existing reshapes, empties, constants and `to_tensor` results as *head* operations that start a fresh dimension space and reshapes and returns as *tail* operations that must be re-expanded (Section 3.1). A reshape in the middle of a function therefore cuts the union-find analysis into segments, and the flattener cannot make one decision for them. Moving reshapes to the boundary lets the analyzer see a single connected region. Conversely, the pass must not run if flattening will not, since it would then merely move reshapes around; a performance regression on a kernel class where flattening is skipped is what motivated the guard.

6 OpFusion

After `PropagateReshape` and tidy flattening the function is a graph of compute operations on a small number of ranks, with reshapes only at its boundary. OpFusion (`hfusion-fuse-ops`) partitions that graph into *fusion groups*, outlines each group into its own device function, and leaves the remainder to a single-operation outlining pass. Its algorithm is a greedy contraction of def-use edges, ordered by a topological ranking of the graph and guarded by an acyclicity test.

6.1 Setting

Input and output. The input is a value-semantics host function carrying a *fusion kind* attribute (Section 6.2). For each accepted group, a new device function is created containing the group’s operations plus an *auxiliary closure*—scalar computations, constants, `tensor.dim`, `tensor.empty` and shape operations that the group needs—and the caller is rewritten to call it. The group’s live-out values become results; *single* and *single-aggressive* output modes instead split a group into one cone per output (the latter duplicating computation).

Operation patterns. Each operation is classified into a coarse *pattern*: elementwise, last-axis or other-axis reduce, last-axis or other-axis broadcast, transpose, matmul, collective, or opaque. Operations with an unknown pattern (calls, `scf`, unregistered dialects) never join a group. The classification of reduces and broadcasts as *last-axis* depends on the ranks produced by the preceding flatten (Section 7).

6.2 Fusion kinds as legality tables

Whether two neighboring operations may be fused depends on the target schedule. OpFusion expresses this as a *fusion kind*; each kind selects a pairwise legality table over patterns (Table 5). A function’s kind is inferred by trying kinds in order of increasing permissiveness and returning the first for which the analysis yields exactly one group containing every compute operation (and, for the matmul kind, exactly one output).

6.3 Block formation by topological contraction

State. Every operation of the block starts as a singleton set of a disjoint-set forest. A *quotient graph* over the sets keeps, for each pair of sets, whether the edge is *fusible*: the conjunction of the pairwise legality over all operation edges between them. Kahn’s algorithm assigns each set a *topological rank*, the length of the longest path from a source.

Kind	Pairs that may fuse
PURE_ELEWISE	Elementwise-class operations among themselves (including slices, concat, pad, reshape, transpose); no reduce, broadcast or matmul.
ANY_PB	As above, plus broadcasts.
LAST_AXIS_PBR	Plus last-axis reduce. Reduce or broadcast may feed elementwise or broadcast consumers; reduce→reduce and other-axis reduce are excluded.
ANY_PBR	Also other-axis reduce and reduce→reduce.
SHALLOW_CV	Everything above plus matmul (except other-axis reduce).
SHALLOW_VV	Any two vector-pattern operations.
MIX_CV	Matmul followed by elementwise, and elementwise chains; elementwise→matmul is not allowed.
MIX_C2	Collective communication adjacent to matmul.
SINGLE_CUBE	Never fusible.

Table 5: Fusion kinds as legality tables.

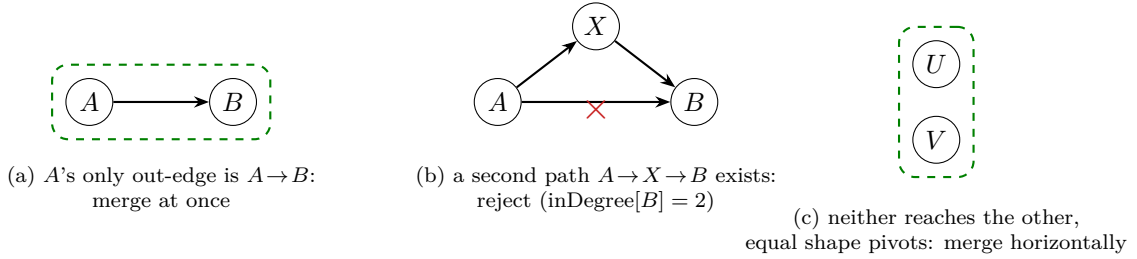


Figure 7: **Acyclicity test for group merging.** Groups are the nodes of a quotient graph that must stay a DAG. (a) Contracting $A \rightarrow B$ is always safe when B is A 's only successor. (b) If there is another path from A to B , contracting the edge would put X both after and before the merged node: a cycle. The test is a bounded search from A over nodes whose topological rank does not exceed B 's; reaching B twice means a second path exists. (c) Independent groups can be merged *horizontally*; they must have no path in either direction, and compatible shapes.

Vertical phase. All def-use edges (u, v) are collected and sorted by $(\text{rank}[v], \text{rank}[u])$ ascending, so that producers are contracted into consumers in topological order. For each candidate whose pattern pair is legal for the current kind, the pass tries to merge the two sets, subject to the checks below.

Acyclicity (Figure 7). Merging the sets of u and v must leave the quotient graph a DAG, so that every group is *convex*: no path leaves a group and re-enters it. If A 's only out-edge is $A \rightarrow B$, the merge is safe. Otherwise a breadth-first search starts at A , expands only nodes whose rank is at most $\text{rank}[B]$, and counts how often each node is reached; if B is reached more than once there is a second path through some X , and the merge is rejected. The rank bound is what makes the test cheap, and processing candidates in ascending consumer rank keeps $\text{rank}[A] \leq \text{rank}[B]$.

Horizontal phase. Groups that remain, are *compute-bearing*, and are independent can be merged side by side. Roots are sorted by rank and tried pairwise, up to a configurable cap on successful merges. A pair is rejected if either set can reach the other (the same search, unpruned, in both directions), or if their *shape pivots* disagree: for the elementwise kinds the destination shapes of pivot operations (elementwise, broadcast, fill) must be equal; for the matmul kind the matmul shapes must match. Dynamic shapes block horizontal fusion for every kind and block vertical fusion for the shallow kinds.

Algorithm 2 Block formation (one fusion kind).

```
1: procedure FUSEBLOCK(ops of one block)
2:   singleton sets; build quotient graph with fusible-edge flags; compute topological ranks
3:    $cand \leftarrow$  all def-use edges  $(u, v)$ , sorted by  $(\text{rank}[v], \text{rank}[u])$ 
4:   for  $(u, v) \in cand$  do
5:     if pattern pair  $(u, v)$  is legal for the kind then
6:       TRYJOIN( $u, v$ , vertical)
7:     end if
8:   end for
9:    $roots \leftarrow$  set roots that contain a compute op; sort by rank
10:  for  $i < j$  over  $roots$  until the horizontal cap is reached do
11:    TRYJOIN( $roots_i, roots_j$ , horizontal)
12:  end for
13:  return sets with at least two compute ops (and a matmul, if the kind requires one)
14: end procedure

15: procedure TRYJOIN( $u, v$ , mode)
16:    $A \leftarrow \text{FIND}(u)$ ,  $B \leftarrow \text{FIND}(v)$ ; if  $A = B$  then return
17:   reject on reduce-rank / reduce-axis mismatch, node-type rules, or (horizontal) shape-pivot
     mismatch
18:   reject if the merge would create a cycle in the quotient graph ▷ Figure 7
19:   union by size; merge quotient edges; set rank to the maximum; merge pattern summaries
20: end procedure
```

Acceptance. A group is accepted if it contains at least two compute operations; the shallow and mixed matmul kinds additionally require a matmul. Everything else is left to single-operation outlining.

Complexity. With N operations and E def-use edges, setup and ranking are $O(N + E)$ and the candidate sort is $O(E \log E)$. A vertical candidate costs $O(1)$ when the producer has out-degree one and $O(N + E)$ for the search otherwise, giving $O(E \log E + E(N + E))$ in the worst case. The horizontal phase tries $O(K^2)$ pairs for K roots, each with up to two searches; the cap bounds the number of *successes*, not attempts. There is no upper bound on group size and no memory-footprint model in this pass.

6.4 Outlining

Each accepted group becomes a device function. The pass collects the auxiliary closure by backward search from the group (pulling in scalar producers, constants, shape computations and buffers), optionally keeping output buffers in the caller and passing them as arguments, and forward search for zero-rank elementwise users whose operands are already inside. If the caller is a host function and multi-kernel mode is off, the function signature is kept stable by repeating each output once per outside use, so that the host calling convention is preserved.

6.5 Position in the design

OpFusion is deliberately simple: a legality table, a topological ordering and a cycle test. It does not consult the Dimension Analyzer and relies on the earlier passes to hand it a graph without interior reshapes and with canonical ranks. It is distinct from the later *vector-function fusion*, which fuses vector operations *inside* a kernel into register-targeted vector functions under a buffer budget; OpFusion works on whole kernels over tensors.

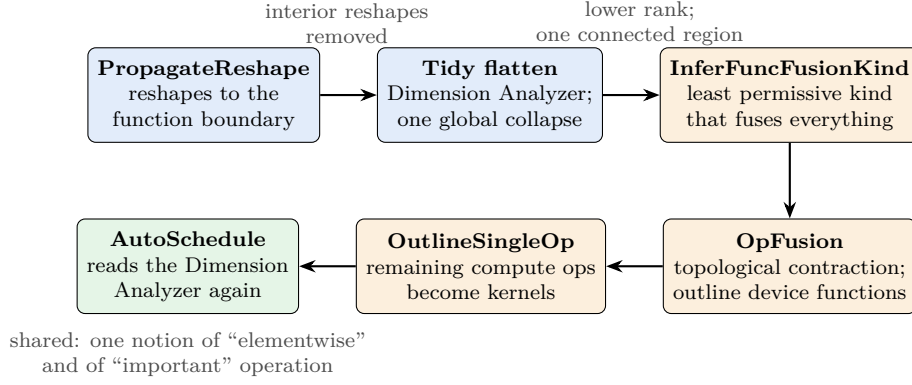


Figure 8: **Composition in the default HFusion pipeline.** Each stage prepares the invariant the next one relies on. In multi-kernel mode the reshape and flatten stages run *after* fusion, on the outlined device functions only. The Triton-kernel path skips operator fusion.

7 Composition in the HFusion Pipeline

The three techniques are stages of one pipeline, and each exists to establish an invariant for the next (Figure 8).

Order. In the default (non-multi-kernel) pipeline the order is: canonicalization (which turns `tensor.reshape` into expand/collapse), `PropagateReshape`, `fold-tensor-empty`, tidy flatten, reshape canonicalization, fusion-kind inference, `OpFusion`, single-operation outlining, symbol cleanup, and finally auto-scheduling. In multi-kernel mode the reshape and flatten stages run after fusion and only on the outlined device functions. The register-based pipeline has the same structure.

Invariants between stages.

- **PropagateReshape $\Rightarrow$ flatten.** After `PropagateReshape` the interior contains no reshape, so the analyzer sees one connected region rather than a chain of head/tail-delimited segments (Section 5.5).
- **Flatten $\Rightarrow$ fusion.** After tidy flattening ranks are small and consistent across operations. `OpFusion`’s pattern classification—“reduce over the last axis”, “broadcast of the last axis”, equal destination shapes for horizontal fusion—is a statement about axes of the flattened shapes.
- **Fusion $\Rightarrow$ scheduling.** The auto-scheduler works on outlined kernels and reuses the Dimension Analyzer to find an anchor axis from the same union-find structures.

Shared infrastructure. The stages share small pieces of vocabulary rather than data structures. `PropagateReshape`, the flattener and `OpFusion` use one definition of “elementwise operation”; the analyzer’s anchor selection uses `OpFusion`’s notion of an “important” (compute) pattern. `OpFusion` itself does not consume the analyzer; its dependence on the earlier passes is through the shapes they produce.

8 Evaluation Scope

This report presents the designs and their mechanisms. It does not report performance measurements, and it makes no quantitative claim about speedup, compile time or code size.

Evidence in this report. The compiler’s regression suite already asserts the structural claims on small cases: the tidy-flatten tests check collapsed shapes (Section 4.4), the PropagateReshape tests check the validity predicate of Section 5.1, and the OpFusion tests check group formation for each fusion kind. These establish that the mechanisms behave as described; they say nothing about benefit.

Planned measurements. A follow-up study would address five questions.

- Q1** Does tidy flattening remove interior reshapes and reduce rank relative to greedy flattening?
- Q2** Does PropagateReshape enlarge the region tidy flattening can optimize, i.e. does flattening collapse more with it than without?
- Q3** How many kernels does OpFusion produce, and how large are the groups, with and without the preceding passes?
- Q4** What is the compile-time cost of each pass, and how does the analyzer scale with function size?
- Q5** What is the end-to-end kernel latency effect on the target hardware?

Static metrics would be the number of `expand_shape/collapse_shape` operations remaining in the interior, mean and maximum tensor rank, and the number and mean size of fusion groups; compile time would be measured per pass and against the node count N ; latency would be measured on device over warm-up and repeated runs. The ablation would compare (A) greedy flatten only, (B) tidy flatten only, (C) PropagateReshape plus tidy flatten, and (D) C plus OpFusion; running fusion with and without a preceding flatten isolates the dependence discussed in Section 7. Workloads would include elementwise, reduce and broadcast micro-kernels, normalization and attention sub-graphs, and representative production sub-graphs where permitted.

9 Related Work

This section situates the work briefly; it is not an exhaustive survey.

Compiler infrastructure. MLIR [5] provides the tensor and `linalg` dialects on which this work is built, including `expand_shape/collapse_shape` with reassociation maps and patterns that fold reshapes into structured operations or collapse the iteration dimensions of a single operation. Those patterns act on one operation at a time; the analysis here instead works on the whole function, so that the collapse decision is consistent across operations.

Tensor compilers and schedules. TVM [6], Halide [8], Tensor Comprehensions [9] and Triton [7] treat loop structure through schedules or through a kernel language; shape simplification is typically a by-product of lowering rather than a separate analysis over the whole graph.

Operator fusion. DNNFusion [10] and AStitch [11] study fusion beyond simple producer–consumer chains, using operator classification and memory-hierarchy-aware grouping respectively. OpFusion in this report is smaller in ambition: it contracts def–use edges under per-kind legality tables, with a topological-rank acyclicity test, and relies on earlier passes to simplify shapes.

Unification-based analyses. Treating a relation as an equivalence maintained by union–find has a long history in program analysis [4], with the data structure itself analyzed in [1, 2, 3]. The Dimension Analyzer applies the same idea to *axes* of tensors and adds an optimistic connectivity layer—adjacency flags that are only ever cleared—on top of the equivalence relations.

Target. The target is the Ascend NPU family [12], whose vector units reward long contiguous runs and whose on-chip buffers motivate several of the guards in Section 5.4.

10 Limitations and Future Work

We record the known weaknesses of each component plainly.

Dimension Analyzer and tidy flatten.

- The scan that separates mutated axes from non-unit neighbors is a *single pass* over class roots, not a fixpoint; we do not prove that it reaches the same result as iterating to a fixpoint.
- The analysis is conservative in the presence of `func.call`: it gives up and the pass is a no-op for the whole function.
- Dynamic extents are handled by forbidding more than one per group, which is safe but loses collapse opportunities.
- The rewrite phase has two super-linear spots (a worklist that erases from the front of a vector-backed set, and a linear membership test on the argument list); the analysis itself is near-linear. Both are straightforward to replace with hashed or indexed structures.
- Correctness rests on the per-operation rules of Table 1: an operation that gives its axes distinct roles needs a rule before it can be flattened safely. An operation without a rule falls to a default: all-parallel operations receive the elementwise rule, a few structural operations (concat, pad, slices, loops, subviews) are handled by shared rules, and any other operation is reported as “unchecked” in debug builds; the traversal discards that result and continues rather than aborting. A declarative, interface-based description of an operation’s axis behavior would make adding operations safer than extending a dispatch function.

PropagateReshape.

- There is no confluence result; the outcome depends on pattern benefit and traversal order.
- Termination is protected by guards (a unit-axis threshold, disabled expansion through `fill`) found empirically, not by a proved decreasing measure.
- Per-operation propagation is implemented as long dispatch chains; only a few operations (loops, multi-result and annotation operations) go through an extension interface. Regularizing this is future work.
- Slice propagation requires static shapes and the hyperrectangle condition.

OpFusion.

- **No cost model.** Group size is unbounded and there is no memory-footprint or occupancy estimate in this pass; fusion decisions are purely legality-driven.
- **Hand-written tables.** Legality is a set of per-kind pairwise rules plus a handful of special cases (shape pivots, dynamic shapes, matmul placement). They encode experience with particular kernel shapes rather than a derived criterion.
- **Kind inference** runs the analysis once per candidate kind (up to nine times per function) instead of deriving the kind in one pass.
- Horizontal fusion considers all pairs of roots; the cap bounds successes, not attempts.

Future work. Fuse the analyzer’s equivalence classes with OpFusion’s shape-pivot test, so that “same shape” is decided once; replace per-kind tables with a small constraint language; bound fusion by an estimate of on-chip buffer use; and obtain the measurements outlined in Section 8.

Acknowledgments

The author thanks the colleagues who reviewed, extended and maintain the implementation in AscendNPU-IR.

A Provenance of the Designs and of the Implementation

This appendix separates what the public repository shows from what rests on the author’s own records.

A.1 What the public repository shows

- The implementation is in the open-source AscendNPU-IR repository (<https://gitcode.com/Ascend/AscendNPU-IR>), licensed under Apache-2.0, with source headers reading “Copyright (c) Huawei Technologies Co., Ltd. 2025”.
- The public history begins with a single squashed commit (780c62557, 2025-05-21). The Dimension Analyzer, the flattener, PropagateReshape and OpFusion are not in it. They first appear together in a bulk-import commit (2d1ae329b, 2025-10-09, “Add HFusion”). Development history before that date is therefore not visible in the public repository, and per-file blame for these components mostly reflects the import and later maintenance.
- Commits under the author’s name that touch PropagateReshape are 1f9e966ab (2026-02-13, a fix to the expand-from-collapse helper), 940f87a06 (2026-07-27, a guard against unsafe reshape propagation) and 6f924a1bc (2026-07-28, a port of the register-based PropagateReshape code). The author’s other commits in this repository concern other components.
- The components have since been extended by other contributors, and several of the more recent commits are synchronizations between branches.

The public history therefore cannot, by itself, establish where the designs originated. That is a statement about the repository, not about the designs.

A.2 Statement of the author

The union–find formulation of dimension analysis over an index space with barrier nodes and optimistic cut flags (Section 3); its use for tidy flattening (Section 4); the PropagateReshape normal form and rewrite approach (Section 5); and the topological-contraction design of OpFusion (Section 6) originate with the author.

A.3 Implementation status of examples in this report

Shapes quoted as “verbatim” come from the compiler’s regression tests. Figures that depict class bookkeeping (for example, the coloring in Figure 4 and the intermediate rows of Figure 5) are derived by hand from the algorithms and were not produced by running the compiler.

References

- [1] B. A. Galler and M. J. Fischer. An improved equivalence algorithm. *Communications of the ACM*, 7(5):301–303, 1964.

- [2] R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22(2):215–225, 1975.
- [3] R. E. Tarjan and J. van Leeuwen. Worst-case analysis of set union algorithms. *Journal of the ACM*, 31(2):245–281, 1984.
- [4] B. Steensgaard. Points-to analysis in almost linear time. In *POPL*, 1996.
- [5] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko. MLIR: Scaling compiler infrastructure for domain specific computation. In *CGO*, 2021.
- [6] T. Chen et al. TVM: An automated end-to-end optimizing compiler for deep learning. In *OSDI*, 2018.
- [7] P. Tillet, H. T. Kung, and D. Cox. Triton: An intermediate language and compiler for tiled neural network computations. In *MAPL*, 2019.
- [8] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *PLDI*, 2013.
- [9] N. Vasilache et al. Tensor Comprehensions: Framework-agnostic high-performance machine learning abstractions. *arXiv:1802.04730*, 2018.
- [10] W. Niu, J. Guan, Y. Wang, G. Agrawal, and B. Ren. DNNFusion: Accelerating deep neural networks execution with advanced operator fusion. In *PLDI*, 2021.
- [11] Z. Zheng et al. AStitch: Enabling a new multi-dimensional optimization space for memory-intensive ML training and inference on modern SIMT architectures. In *ASPLOS*, pages 359–373, 2022.
- [12] H. Liao et al. Ascend: a scalable and unified architecture for ubiquitous deep neural network computing (industry track). In *HPCA*, 2021.