An Approach for Tiling and Fusing Dependent Reductions in Linalg

MLIR Developer Meetup - Oct 2026

Charitha Saumya
Intel MLIR Team

intel.

Tile, Fuse Producers/Consumers, and Repeat

```
// prologue: A + c
%a = linalg.add ins(%A, %cst) outs(%e0)

// GEMM
%c = linalg.matmul ins(%a, %B) outs(%C)

// epilogue: exp
%d = linalg.exp ins(%c) outs(%D)
```

Tile the GEMM
Fuse producers
Fuse consumers



```
%r = scf.forall (%i, %j) = (0, 0) to (M, N)
       step (tM, tN) shared_outs(%o = %D) {
  %At = tensor.extract_slice %A[%i, 0] [tM, K]
  %at = linalg.add ins(%At, %cst) outs(%e0t)
  %Bt = tensor.extract_slice %B[0, %j] [K, tN]
  %Ct = tensor.extract_slice %C[%i, %j] [tM, tN]
  %ct = linalg.matmul ins(%at, %Bt) outs(%Ct)
  %Ot = tensor.extract_slice %o[%i, %j] [tM, tN]
  %dt = linalg.exp ins(%ct) outs(%Ot)
  scf.forall.in_parallel {
    tensor.parallel_insert_slice %dt into
        %o[%i, %j] [tM, tN]
  }
}
```

Great support for tiling along parallel/reduction dims and fusion along parallel dims

How about Dependent Reductions?

```
// R1: reduce along the reduction dim
%m = linalg.reduce {red_kind1}
       ins(%x) outs(%init1)
       dimensions = [1]
// R1's result, broadcast along it
%p = linalg.generic {...}
       ins(%x, %m) outs(%init)
// R2: reduce along the same dim
%s = linalg.reduce {red_kind2}
       ins(%p) outs(%init2)
       dimensions = [1]
```

Tile R2
Fuse producers



```
%s = scf.for %j = 0 to N step tN iter_args(%a) {
  %xt = tensor.extract_slice %x[0, %j] [M, tN]
  // R1 fused at FULL extent: reads all of x
  %m  = linalg.reduce {red_kind1} ins(%x) outs(%i1)
  %pt = linalg.generic {...} ins(%xt, %m) outs(%et)
  %st = linalg.reduce {red_kind2} ins(%pt) outs(%a)
  scf.yield %st
}
```

R2's tile needs R1's result over the *whole* row ⇒ R1 can't be tiled

It re-reads all of `x` in *every* iteration ⇒ bad performance

Settle for Separate Loops

```
// R1: reduce along the reduction dim
%m = linalg.reduce {red_kind1}
       ins(%x) outs(%init1)
       dimensions = [1]
// R1's result, broadcast along it
%p = linalg.generic {...}
       ins(%x, %m) outs(%init)
// R2: reduce along the same dim
%s = linalg.reduce {red_kind2}
       ins(%p) outs(%init2)
       dimensions = [1]
```

Tile R1
Tile R2
No fusion



```
%m = scf.for %j = 0 to N step tN iter_args(%a) {
  %xt = tensor.extract_slice %x[0, %j] [M, tN]
  %mt = linalg.reduce {red_kind1} ins(%xt) outs(%a)
  scf.yield %mt
}
%s = scf.for %j = 0 to N step tN iter_args(%a) {
  %xt = tensor.extract_slice %x[0, %j] [M, tN]
  %pt = linalg.generic {...} ins(%xt, %m) outs(%et)
  %st = linalg.reduce {red_kind2} ins(%pt) outs(%a)
  scf.yield %st
}
```

Better: R1 is computed once, no recompute

Not the best: x is read twice ⇒ costly when the reduction dim is huge

Can we do better? Online Softmax

$$m = \max_j x_j$$

$$s = \sum_j e^{x_j - m}$$

$$\mathrm{softmax}(x)_j = \frac{e^{x_j - m}}{s}$$

$$e^{x_j - m_{\text{new}}} = e^{x_j - m_{\text{old}}} \cdot e^{m_{\text{old}} - m_{\text{new}}}$$

```
m = -inf;  s = 0
for each tile x_t of x:
  m_new = max(m, max(x_t))
  s = s * exp(m - m_new)     // correction
      + sum(exp(x_t - m_new))
  m = m_new
```

One pass over `x`: rescale the running sum whenever the max changes
Existing approaches: high-level ops, complex pattern matching

No transform primitive that composes with the existing tile/fuse infra?

When/Why does this work?

$$m = \max_j x_j$$

$$\mathrm{softmax}(x)_j = \frac{e^{x_j - m}}{s}$$

$$s = \sum_j e^{x_j - m}$$

$$= \sum_j \underbrace{e^{x_j}}_{\text{data}} \cdot \underbrace{e^{-m}}_{\text{R1}}$$

Stage $i-1$

$$s_{i-1} = \sum_{j<i} e^{x_j - m_{i-1}} \tag{1}$$

$$= e^{x_1} e^{-m_{i-1}} + e^{x_2} e^{-m_{i-1}} + \cdots + e^{x_{i-1}} e^{-m_{i-1}} \tag{2}$$

Stage i

$$m_i = \max(m_{i-1}, x_i) \tag{3}$$

$$s_{i-1} \cdot \frac{1}{e^{-m_{i-1}}} = e^{x_1} + e^{x_2} + \cdots + e^{x_{i-1}} \tag{4}$$

$$s_{i-1} \cdot \boxed{\frac{e^{-m_i}}{e^{-m_{i-1}}}} = e^{x_1 - m_i} + e^{x_2 - m_i} + \cdots + e^{x_{i-1} - m_i} \tag{5}$$

$$s_i = s_{i-1} \cdot e^{m_{i-1} - m_i} + e^{x_i - m_i} \tag{6}$$

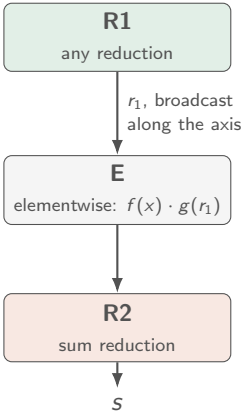
correction factor

R2 is a *sum*, and each summed term is $f(x) \cdot g(r_1)$ (multiplicatively separable)
Multiplication distributes over addition $\Rightarrow$ one rescale corrects every term

Legality Conditions

For a reduction chain R1 $\rightarrow$ E $\rightarrow$ R2:

1. **R1** can be *any* reduction
2. **E** (elementwise) uses R1's result *broadcast* along R1's reduction axis
3. E's use of R1 is *multiplicatively separable*: $E(x, r_1) = f(x) \cdot g(r_1)$
4. **R2** is a *sum* reduction that reduces over the same axis as R1



Multiplicative Separability

A lightweight *dataflow analysis* over E's body proves $E(x, r_1) = f(x) \cdot g(r_1)$

Facts, propagated from inputs to yield

- {IND}: independent of R1
- {ADD}: R1 enters *additively*
- {MUL}: R1 enters *multiplicatively*

Transfer rules

`add, sub`	keep {ADD}
`mul, div`	keep {MUL}
`exp`	{ADD} → {MUL}
`neg` / `abs`	keep all / keep {MUL}
other ops	no facts

Example: softmax, $E = \exp(x - m)$

```
// %x is {IND}, %m is {ADD, MUL}
%d = arith.subf %x, %m    // {ADD}
%e = math.exp %d          // {MUL}
linalg.yield %e           // {MUL}
```

Yield is {MUL} and depends on `%m` ⇒ separable ✓

Accept E only if its yield is {MUL} and depends on R1

Deriving Online Correction

1. Separability gives $E(x, r_1) = f(x) \cdot g(r_1)$. To isolate r_1's effect, build $E'(r_1) = g(r_1)$ by making $f(x) = 1$
2. Compute the *forward slice* of r_1 in E's body
3. Set every slice operand that does not depend on r_1 to the op's *neutral element* (0 for `add`/`sub`, 1 for `mul`/`div`)
4. Evaluate E' at r_1^{old} and r_1^{new}: correction $= E'(r_1^{\text{new}})/E'(r_1^{\text{old}})$

Example: $E(x, r_1) = \exp(x - r_1)$

```
// E
%d = arith.subf %x, %r1
%e = math.exp %d
linalg.yield %e
```

```
// E': %x := 0.0
%c0 = arith.constant 0.0
%d = arith.subf %c0, %r1
%e = math.exp %d
linalg.yield %e
```

$$E'(r_1) = \exp(0.0 - r_1)$$

$$\text{correction} = \frac{\exp(0.0 - r_1^{\text{new}})}{\exp(0.0 - r_1^{\text{old}})}$$

Proposal: *structured.fuse_dependent_reduction_ops*

Three operand handles, one result:

1. `R1` — already tiled producer reduction (tiled using the `FullReduction` strategy, i.e. `tile_using_for`). Can carry multiple such reductions.
2. `E` — all-parallel elementwise op.
3. `R2` — still-untiled consumer reduction.

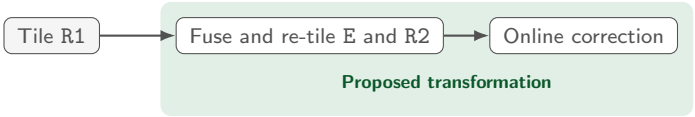
```
%loop = ..... -> !transform.any_op   // R1: the already tiled reduction loop
%e    = ..... -> !transform.any_op   // E : the elementwise term
%r2   = ..... -> !transform.any_op   // R2: the consumer reduction

%fused = transform.structured.fuse_dependent_reduction_ops %e, %r2 into %loop
    : (!transform.any_op, !transform.any_op, !transform.any_op)
    -> !transform.any_op
```

Returns the fused loop, now carrying `E` and `R2`

The op can be applied repeatedly to fuse chains of dependent reductions

Main Stages of the Transformation



- E is fused first (`scf::tileAndFuseConsumer`), then immediately re-tiled to match R1's tile
- R2 is fused against the slice that E yields
- Emit online correction for the R2 accumulator
- Implemented as an `OpRewritePattern` on R2

Input: softmax, max already tiled

```
for j in tiles(N):                  // R1
  m = max(m, max(x[j]))
p = exp(x - m)                      // E
s = sum(p)                          // R2
```

After fusion: one loop

```
for j in tiles(N):
  m_new = max(m, max(x[j]))         // R1
  p     = exp(x[j] - m_new)         // E, re-tiled
  s     = s * exp(m - m_new)        // online correction
  s     = s + sum(p)                // R2, re-tiled
  m     = m_new
```

Example: Standard to Fused Attention

- `x` is the score matrix: `x = Q @ K`$^{\mathtt{T}}$
- All reductions are over the inner `j` axis
- Normalization is deferred (flash-attention style)

$$
\begin{aligned}
R1\ (\max): &\quad m[i] = \max_j x[i,j] \\
E\ (\text{term}): &\quad p[i,j] = \exp(x[i,j] - m[i]) \\
R2a\ (\text{sum}): &\quad s[i] = \textstyle\sum_j p[i,j] \\
R2b\ (P@V): &\quad o'[i,d] = \textstyle\sum_j p[i,j]\, V[j,d] \\
\text{div}: &\quad o[i,d] = o'[i,d]/s[i]
\end{aligned}
$$

A single `E` feeds two consumer reductions $\Rightarrow$ requires 2 applications of reduction fusion

Input IR

```
%m = scf.for %iv = 0 to 512 step 32                    // R1: row max, already tiled
       iter_args(%acc_m = %ninf_64) {
  %xt   = tensor.extract_slice %x[0, %iv] [64, 32]     // score tile
  %mold = tensor.extract_slice %acc_m[0] [64]
  %mt   = linalg.generic ins(%xt) outs(%mold)          // max(x_tile, acc_m)
  %im   = tensor.insert_slice %mt into %acc_m[0] [64]
  scf.yield %im
}
%p = linalg.generic ins(%x, %m) outs(%empty_64x512)    // E: exp(x - m), full [64, 512]
%s = linalg.generic ins(%p) outs(%zero_64)             // R2a: row sum, sum_j p
%o = linalg.generic ins(%p, %V) outs(%zero_64x64)      // R2b: P @ V, sum_j p * V
%out = linalg.generic ins(%o, %s) outs(%empty_64x64)   // normalize: o / s
```

Attention input shapes: `Q @ K`^{T} (64×512), `K`, `V` (512×64)

The Transform Schedule

```
%gen = transform.structured.match ops{["linalg.generic"]} in %arg0 .....
%r1, %e, %r2a, %r2b, %div = transform.split_handle %gen .....     // R1, E, R2a (sum), R2b (P @ V), div
%loop = transform.structured.match ops{["scf.for"]} in %arg0 .....

%fa = transform.structured.fuse_dependent_reduction_ops           // Fuse chain 1: max -> E -> row sum
        %e, %r2a into %loop .....

%e2 = transform.get_producer_of_operand %r2b[0] .....             // re-derive E (handle consumed)
%fb = transform.structured.fuse_dependent_reduction_ops           // Capture fused loop %fa and fuse
        %e2, %r2b into %fa .....                                  // chain 2: max + sum -> E -> P @ V
```

Step 1: *E* Fused — Still at Full Extent

```
%r:2 = scf.for %iv = 0 to 512 step 32                    // new iter_arg %acc_p: E's full result
         iter_args(%acc_m = %ninf_64, %acc_p = %empty_64x512) {
  %xt   = tensor.extract_slice %x[0, %iv] [64, 32]
  %mold = tensor.extract_slice %acc_m[0] [64]
  %mt   = linalg.generic ins(%xt) outs(%mold)            // max(x_tile, acc_m)
  %xf   = tensor.extract_slice %x[0, 0] [64, 512]        // Fused E clone reads all the cols along the
  %pdst = tensor.extract_slice %acc_p[0, 0] [64, 512]    // reduction dim (512 cols) and writes
  %pt   = linalg.generic ins(%xf, %mt) outs(%pdst)       // exp(x - m_new) along the FULL row
  %im   = tensor.insert_slice %mt into %acc_m[0] [64]
  %ip   = tensor.insert_slice %pt into %acc_p[0, 0] [64, 512]
  scf.yield %im, %ip
}                                                        // outside: row sum, original E, P @ V, div
```

Step 2: E Re-Tiled to the Reduction Tile

```
%r:2 = scf.for %iv = 0 to 512 step 32                    // Reduction tile size = 32
         iter_args(%acc_m = %ninf_64, %acc_p = %empty_64x512) {
  %xt   = tensor.extract_slice %x[0, %iv] [64, 32]
  %mold = tensor.extract_slice %acc_m[0] [64]
  %mt   = linalg.generic ins(%xt) outs(%mold)            // max(x_tile, acc_m)
  %pdst = tensor.extract_slice %acc_p[0, %iv] [64, 32]   // E cut to the [64, 32] tile
  %pt   = linalg.generic ins(%xt, %mt) outs(%pdst)       // exp(x_tile - m_new)
  %im   = tensor.insert_slice %mt into %acc_m[0] [64]
  %ip   = tensor.insert_slice %pt into %acc_p[0, %iv] [64, 32]
  scf.yield %im, %ip
}
```

Step 3: The Row Sum Fused (and Re-Tiled)

```
%r:3 = scf.for %iv = 0 to 512 step 32
         iter_args(%acc_m = %ninf_64, %acc_p = %empty_64x512, %acc_s = %zero_64) {
  %xt   = tensor.extract_slice %x[0, %iv] [64, 32]
  %mold = tensor.extract_slice %acc_m[0] [64]
  %mt   = linalg.generic ins(%xt) outs(%mold)       // max(x_tile, acc_m)
  %pdst = tensor.extract_slice %acc_p[0, %iv] [64, 32]
  %pt   = linalg.generic ins(%xt, %mt) outs(%pdst)  // exp(x_tile - m_new)
  %sold = tensor.extract_slice %acc_s[0] [64]       // Row sum fused against the E tile
  %st   = linalg.generic ins(%pt) outs(%sold)       // (acc_s + sum_j p_tile)
  ... yields: %mt, %pt, %st
}
```

Sum's only input was E, which is already re-tiled. So re-tiling is a no-op in this case

Step 4: Online Correction — Chain 1 Complete

```
%r:3 = scf.for %iv = 0 to 512 step 32
         iter_args(%acc_m = %ninf_64, %acc_p = %empty_64x512, %acc_s = %zero_64) {
  %mt   = linalg.generic ins(%xt) outs(%mold)          // max(x_tile, acc_m)
  %pt   = linalg.generic ins(%xt, %mt) outs(%pdst)     // exp(x_tile - m_new)
  %sold = tensor.extract_slice %acc_s[0] [64]
  %tnew = linalg.generic ins(%mt) outs(%empty_64)      // Online correction: E'(m_new) = exp(0.0 - m_new)
  %told = linalg.generic ins(%mold) outs(%empty_64)    // E'(m_old) = exp(0.0 - m_old)
  %f    = linalg.elementwise <div> ins(%tnew, %told)   // correction factor = E'(m_new)/E'(m_old)
  %sc   = linalg.elementwise <mul> ins(%sold, %f)      // acc_s_rescaled = acc_s * factor
  %st   = linalg.generic ins(%pt) outs(%sc)            // sc + sum_j p_tile
  ... yields: %mt, %pt, %st
}
```

Loop is again a tiled reduction loop (now returning two reductions) ⇒ it can be the producer of the next chain

Step 5: P @ V Fusion

```
%r:5 = scf.for %iv = 0 to 512 step 32
         iter_args(%acc_m = %ninf_64, %acc_p1 = %empty_64x512, %acc_s = %zero_64,
                   %acc_p2 = %empty_64x512, %acc_o = %zero_64x64) {
  // %mt (max), %st (sum) fused in previous step
  %vt    = tensor.extract_slice %V[%iv, 0] [32, 64]        // V slice is re-tiled
  %oold  = tensor.extract_slice %acc_o[0, 0] [64, 64]      // Online correction for P @ V acc
  %tnew2 = linalg.generic ins(%mt) outs(%empty_64x64)      // E'(m_new) = exp(0.0 - m_new)
  %told2 = linalg.generic ins(%mold) outs(%empty_64x64)    // E'(m_old) = exp(0.0 - m_old)
  %f2    = linalg.elementwise <div> ins(%tnew2, %told2)    // correction factor = E'(m_new)/E'(m_old)
  %oc    = linalg.elementwise <mul> ins(%oold, %f2)        // acc_o_rescaled = acc_o * factor
  %ot    = linalg.generic ins(%p2, %vt) outs(%oc)          // oc + sum_j p2 * V_tile
  // insert result tensors into the running accumulators
  ... yields: %mt, %p1, %st, %p2, %ot
}
%out = linalg.generic ins(%r#4, %r#2) outs(%empty_64x64)   // only the divide is left: o / s
```

`V` must be re-tiled (unlike before) because it's an extra input for R2

Result is a fused online loop with max, sum and `P @ V` accumulators

Evaluation

Hardware

- Intel Battlemage (BMG) GPU

Attention kernel

- **Input IR**: standard attention kernel written in Linalg, lowered for end-to-end execution using the upstream XeGPU pipeline
- **Sizes**: `Z = 2` (batch), `H = 8` heads, head dim = `64`, `f16`; `n_ctx` $\in \{1024, 4096, 8192, 16384\}$
- **Baseline (unfused)**: same kernel tiled/fused with existing upstream transforms only — no fusion on the reduction dim, score matrix recomputed for each reduction

Results: Attention

n_ctx	unfused	reduction fusion (this RFC)
1024	1.00x	**2.49x**
4096	1.00x	**2.69x**
8192	1.00x	**2.59x**
16384	1.00x	**2.19x**

Reduction fusion gives 2.2x–2.7x speedup over the unfused baseline

Results: Softmax and L1/L2 Normalization (BMG)

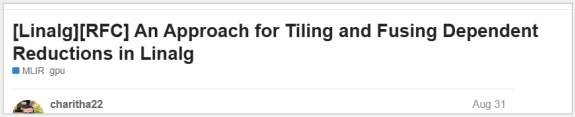
- All three are a max reduction → `E` → sum reduction chain, with a different `E` in each kernel
- Baseline (unfused): 3 loops vs. 2 loops with fusion

kernel	`E`	correction derived from `E`
softmax	`exp(x - m)`	`exp(m_old - m_new)`
L1 normalize	`abs(x / m)`	`m_old / m_new`
L2 normalize	`(x / m)^2`	`(m_old / m_new)^2`

reduction dim	baseline	softmax	L1 norm.	L2 norm.
4096	1.00x	**1.23x**	**1.22x**	**1.22x**
8192	1.00x	**1.23x**	**1.22x**	**1.23x**
16384	1.00x	**1.21x**	**1.20x**	**1.21x**

The transformation automatically derives the correction factor for each `E`

Current Status



LLVM Lighthouse contributions

[linalg][transforms] Add legality checks for fusing dependent reductions. #295

Merged tkarna merged 10 commits into llvm:main from charithaintc:dependant_reduction_legality 3 weeks ago

[linalg][transfrom] A transform extension for tiling and fusing dependent reductions. #297

Merged tkarna merged 7 commits into llvm:main from charithaintc:linalg_reduction_fusion 3 weeks ago

Thank you!

Questions?