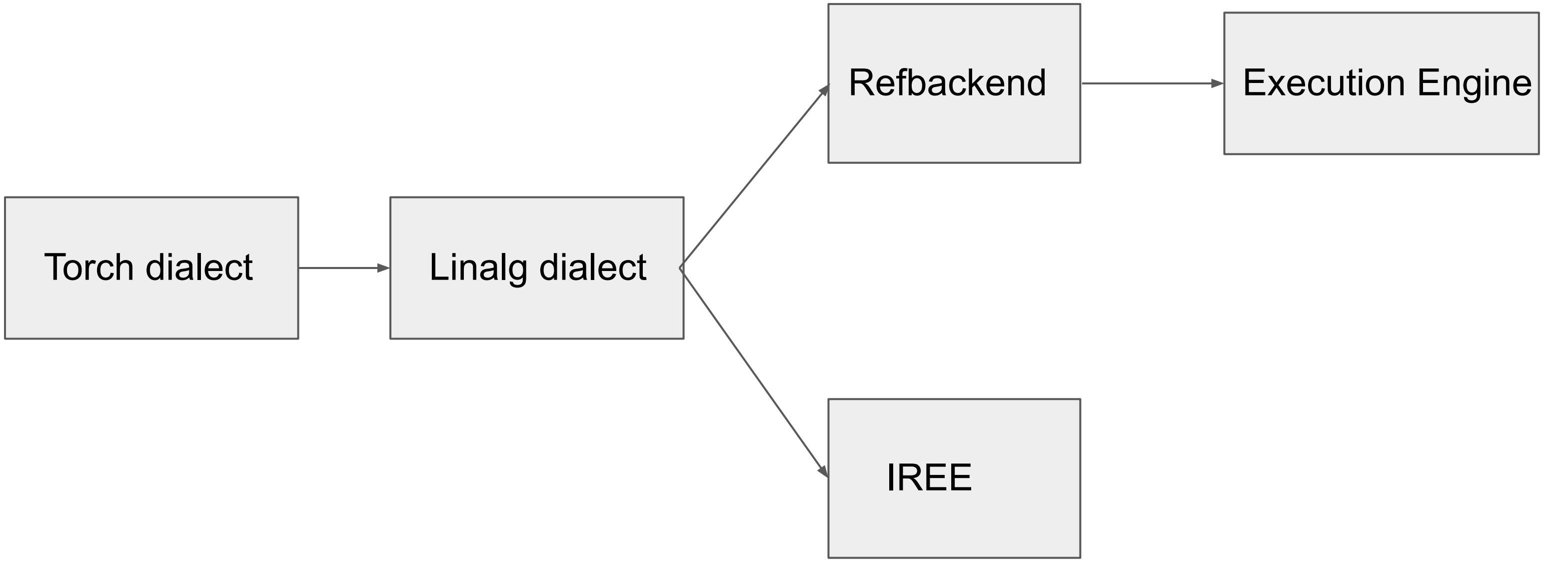
Anatomy of Linalg.generic

Yi Zhang

End to End Torch Module Execution



Linalg ops overview

Linalg defines payload carrying operations which implement the **structured** ops.

- Structure ops carry out computations on tensor or buffers like contractions or convolutions.
- Can be further lowered to loops or to affine expressions with computation in the loop body.

Linalg ops overview

Linalg defines a small set of commonly used named ops

linalg-generalize-named-ops

Named Ops → **Linalg.generic Ops**

linalg.conv

linalg.batch_matmul

linalg.pooling

Example tensor operation

perform a sum reduction along the H,W dimensions of a tensor<NxCxHxW>, resulting in a tensor<NxC>.

Example tensor operation

perform a sum reduction along the H,W dimensions of a tensor<NxCxHxW>, resulting in a tensor<NxC>.

```
%6 = linalg.generic {
    indexing_maps =
              [affine_map<(d0, d1, d2, d3) -> (d0, d1, d2, d3)>,
               affine_map<(d0, d1, d2, d3) -> (d0, d1)>],

    iterator_types = ["parallel", "parallel", "reduction", "  reduction"]}

    ins(%1 : tensor<?x?x?x?xf32>) outs(%5 : tensor<?x?xf32>) {

      ^bb0(%arg1: f32, %arg2: f32):  // no predecessors
           %17 = arith.addf %arg2, %arg1 : f32
           linalg.yield %17 : f32

     } -> tensor<?x?xf32>
```

compute payload

Components of a generic op

- iterator types
- indexing maps
- input/output tensors
- compute payload

Iterator types

```
%6 = linalg.generic {
    indexing_maps =
            [affine_map<(d0, d1, d2, d3) -> (d0, d1, d2, d3)>,
             affine_map<(d0, d1, d2, d3) -> (d0, d1)>],

    iterator_types = ["parallel", "parallel", "reduction", "reduction"]}

    ins(%1 : tensor<?x?x?x?xf32>) outs(%5 : tensor<?x?xf32>) {

      ^bb0(%arg1: f32, %arg2: f32):  // no predecessors
          %17 = arith.addf %arg2, %arg1 : f32
          linalg.yield %17 : f32

    } -> tensor<?x?xf32>
```

```
for d0 := …
  for d1 := …
    for d2 := …
      for d3 := …
```

Iterator types

```
SmallVector<StringRef, 4> iteratorTypesSum{"parallel", "parallel",
                                            "reduction", "reduction"};
```

```
%6 = linalg.generic {
  indexing_maps =
        [affine_map<(d0, d1, d2, d3) -> (d0, d1, d2, d3)>,
         affine_map<(d0, d1, d2, d3) -> (d0, d1)>],

  iterator_types = ["parallel", "parallel", "reduction", "reduction"]}

  ins(%1 : tensor<?x?x?x?xf32>) outs(%5 : tensor<?x?xf32>) {

    ^bb0(%arg1: f32, %arg2: f32):  // no predecessors
       %17 = arith.addf %arg2, %arg1 : f32
       linalg.yield %17 : f32

 } -> tensor<?x?xf32>
```

Indexing maps

```
%6 = linalg.generic {
  indexing_maps =
          [affine_map<(d0, d1, d2, d3) -> (d0, d1, d2, d3)>,
           affine_map<(d0, d1, d2, d3) -> (d0, d1)>],

  iterator_types = ["parallel", "parallel", "reduction", "reduction"]}

  ins(%1 : tensor<?x?x?x?xf32>) outs(%5 : tensor<?x?xf32>) {

    ^bb0(%arg1: f32, %arg2: f32):  // no predecessors
        %17 = arith.addf %arg2, %arg1 : f32
        linalg.yield %17 : f32

  } -> tensor<?x?xf32>
```

Indexing maps

```
affine_map<(d0, d1, d2, d3) -> (d1+1, 2*d2, d3)>
```

```
for d0 := …
  for d1 := …
    for d2 := …
      for d3 := …
```

The left hand side are the induction variables for each nested loops

Indexing maps

```
affine_map<(d0, d1, d2, d3) -> (d1+1, 2*d2, d3)>
```

```
for d0 := …
  for d1 := …
    for d2 := …
      for d3 := …
```

a[d1+1][2*d2][d3]

Indexing maps

affine_map<(d0, d1, d2, d3) -> (d0, d1, d2, d3)>

identity map

```
for d0 := …
  for d1 := …
    for d2 := …
      for d3 := …

a[d0][d1][d2][d3]
```

affine_map<(d0, d1, d2, d3) -> (d0, d1)>

```
for d0 := …
  for d1 := …
    for d2 := …
      for d3 := …

b[d0][d1]
```

Indexing maps

```
affine_map<(d0, d1, d2, d3) -> (d0, d1, d2, d3)>
```

input

```
for d0 := …
 for d1 := …
  for d2 := …
   for d3 := …

a[d0][d1][d2][d3]
```

```
affine_map<(d0, d1, d2, d3) -> (d0, d1)>
```

output

```
for d0 := …
 for d1 := …
  for d2 := …
   for d3 := …

b[d0][d1]
```

accumulated sum of the inner two dimension:

b[d0][d1] += a[d0][d1][d2][d3]

Indexing maps

```
affine_map<(d0, d1, d2, d3) -> (d0, d1, d2, d3)>
```

output

```
for d0 := …
  for d1 := …
    for d2 := …
      for d3 := …

a[d0][d1][d2][d3]
```

```
affine_map<(d0, d1, d2, d3) -> (d0, d1)>
```

input

```
for d0 := …
  for d1 := …
    for d2 := …
      for d3 := …

b[d0][d1]
```

broadcast the 2 dimensions:

a[d0][d1][d2][d3] = b[d0][d1]

Indexing maps

input

```
affine_map<(d0, d1) -> (d1, d0)>
```

```
for d0 := …
  for d1 := …

a[d1][d0]
```

output

```
affine_map<(d0, d1) -> (d0, d1)>
```

```
for d0 := …
  for d1 := …

b[d0][d1]
```

transpose the input:

b[d0][d1] = a[d1][d0]

Indexing maps

```
SmallVector<AffineExpr, 2> ncExprs;
ncExprs.push_back(mlir::getAffineDimExpr(0, context));
ncExprs.push_back(mlir::getAffineDimExpr(1, context));
auto ncIndexingMap = AffineMap::get(
    /*dimCount=*/4,
    /*symbolCount=*/0, ncExprs, context);
SmallVector<AffineMap, 2> indexingMaps = {
    rewriter.getMultiDimIdentityMap(4), // input
    ncIndexingMap,                       // output
};
```

```
%6 = linalg.generic {
  indexing_maps =
        [affine_map<(d0, d1, d2, d3) -> (d0, d1, d2, d3)>,
         affine_map<(d0, d1, d2, d3) -> (d0, d1)>],

  iterator_types = ["parallel", "parallel", "reduction", "reduction"]}

  ins(%1 : tensor<?x?x?x?xf32>) outs(%5 : tensor<?x?xf32>) {

    ^bb0(%arg1: f32, %arg2: f32):  // no predecessors
      %17 = arith.addf %arg2, %arg1 : f32
      linalg.yield %17 : f32

  } -> tensor<?x?xf32>
```

Indexing maps

```
SmallVector<AffineExpr, 2> ncExprs;
ncExprs.push_back(mlir::getAffineDimExpr(0, context));
ncExprs.push_back(mlir::getAffineDimExpr(1, context));
auto ncIndexingMap = AffineMap::get(
    /*dimCount=*/4,
    /*symbolCount=*/0, ncExprs, context);
SmallVector<AffineMap, 2> indexingMaps = {
    rewriter.getMultiDimIdentityMap(4), // input
    ncIndexingMap,                      // output
};
```

```
%6 = linalg.generic {
    indexing_maps =
            [affine_map<(d0, d1, d2, d3) -> (d0, d1, d2, d3)>,
             affine_map<(d0, d1, d2, d3) -> (d0, d1)>],

    iterator_types = ["parallel", "parallel", "reduction", "reduction"]}

    ins(%1 : tensor<?x?x?x?xf32>) outs(%5 : tensor<?x?xf32>) {

      ^bb0(%arg1: f32, %arg2: f32):  // no predecessors
          %17 = arith.addf %arg2, %arg1 : f32
          linalg.yield %17 : f32

  } -> tensor<?x?xf32>
```

Indexing maps

```cpp
SmallVector<AffineExpr, 2> ncExprs;
ncExprs.push_back(mlir::getAffineDimExpr(0, context));
ncExprs.push_back(mlir::getAffineDimExpr(1, context));
auto ncIndexingMap = AffineMap::get(
    /*dimCount=*/4,
    /*symbolCount=*/0, ncExprs, context);
SmallVector<AffineMap, 2> indexingMaps = {
    rewriter.getMultiDimIdentityMap(4), // input
    ncIndexingMap,                      // output
};
```

```
%6 = linalg.generic {
    indexing_maps =
            [affine_map<(d0, d1, d2, d3) -> (d0, d1, d2, d3)>,
             affine_map<(d0, d1, d2, d3) -> (d0, d1)>],

    iterator_types = ["parallel", "parallel", "reduction", "reduction"]}

    ins(%1 : tensor<?x?x?x?xf32>) outs(%5 : tensor<?x?xf32>) {

      ^bb0(%arg1: f32, %arg2: f32):  // no predecessors
          %17 = arith.addf %arg2, %arg1 : f32
          linalg.yield %17 : f32

  } -> tensor<?x?xf32>
```

Indexing maps

```
SmallVector<AffineExpr, 2> ncExprs;
ncExprs.push_back(mlir::getAffineDimExpr(0, context));
ncExprs.push_back(mlir::getAffineDimExpr(1, context));
auto ncIndexingMap = AffineMap::get(
    /*dimCount=*/4,
    /*symbolCount=*/0, ncExprs, context);
SmallVector<AffineMap, 2> indexingMaps = {
    rewriter.getMultiDimIdentityMap(4), // input
    ncIndexingMap,                      // output
};
```

```
%6 = linalg.generic {
    indexing_maps =
            [affine_map<(d0, d1, d2, d3) -> (d0, d1, d2, d3)>,
             affine_map<(d0, d1, d2, d3) -> (d0, d1)>],

    iterator_types = ["parallel", "parallel", "reduction", "reduction"]}

    ins(%1 : tensor<?x?x?x?xf32>) outs(%5 : tensor<?x?xf32>) {

      ^bb0(%arg1: f32, %arg2: f32):  // no predecessors
          %17 = arith.addf %arg2, %arg1 : f32
          linalg.yield %17 : f32

     } -> tensor<?x?xf32>
```

Indexing maps

```
SmallVector<AffineExpr, 2> ncExprs;
ncExprs.push_back(mlir::getAffineDimExpr(0, context));
ncExprs.push_back(mlir::getAffineDimExpr(1, context));
auto ncIndexingMap = AffineMap::get(
    /*dimCount=*/4,
    /*symbolCount=*/0, ncExprs, context);
SmallVector<AffineMap, 2> indexingMaps = {
    rewriter.getMultiDimIdentityMap(4), // input
    ncIndexingMap,                      // output
};
```

```
%6 = linalg.generic {
    indexing_maps =
            [affine_map<(d0, d1, d2, d3) -> (d0, d1, d2, d3)>,
             affine_map<(d0, d1, d2, d3) -> (d0, d1)>],

    iterator_types = ["parallel", "parallel", "reduction", "reduction"]}

    ins(%1 : tensor<?x?x?x?xf32>) outs(%5 : tensor<?x?xf32>) {

      ^bb0(%arg1: f32, %arg2: f32):  // no predecessors
          %17 = arith.addf %arg2, %arg1 : f32
          linalg.yield %17 : f32

  } -> tensor<?x?xf32>
```

Indexing maps

```cpp
SmallVector<AffineExpr, 2> ncExprs;
ncExprs.push_back(mlir::getAffineDimExpr(0, context));
ncExprs.push_back(mlir::getAffineDimExpr(1, context));
auto ncIndexingMap = AffineMap::get(
    /*dimCount=*/4,
    /*symbolCount=*/0, ncExprs, context);
SmallVector<AffineMap, 2> indexingMaps = {
    rewriter.getMultiDimIdentityMap(4), // input
    ncIndexingMap,                      // output
};
```

```mlir
%6 = linalg.generic {
    indexing_maps =
              [affine_map<(d0, d1, d2, d3) -> (d0, d1, d2, d3)>,
               affine_map<(d0, d1, d2, d3) -> (d0, d1)>],

    iterator_types = ["parallel", "parallel", "reduction", "reduction"]}

    ins(%1 : tensor<?x?x?x?xf32>) outs(%5 : tensor<?x?xf32>) {

      ^bb0(%arg1: f32, %arg2: f32):  // no predecessors
            %17 = arith.addf %arg2, %arg1 : f32
            linalg.yield %17 : f32

  } -> tensor<?x?xf32>
```

Indexing maps

```
SmallVector<AffineExpr, 2> ncExprs;
ncExprs.push_back(mlir::getAffineDimExpr(0, context));
ncExprs.push_back(mlir::getAffineDimExpr(1, context));
auto ncIndexingMap = AffineMap::get(
    /*dimCount=*/4,
    /*symbolCount=*/0, ncExprs, context);
SmallVector<AffineMap, 2> indexingMaps = {
    rewriter.getMultiDimIdentityMap(4), // input
    ncIndexingMap,                      // output
};
```

```
%6 = linalg.generic {
   indexing_maps =
          [affine_map<(d0, d1, d2, d3) -> (d0, d1, d2, d3)>,
           affine_map<(d0, d1, d2, d3) -> (d0, d1)>],

   iterator_types = ["parallel", "parallel", "reduction", "reduction"]}

   ins(%1 : tensor<?x?x?x?xf32>) outs(%5 : tensor<?x?xf32>) {

     ^bb0(%arg1: f32, %arg2: f32):  // no predecessors
        %17 = arith.addf %arg2, %arg1 : f32
        linalg.yield %17 : f32

  } -> tensor<?x?xf32>
```

Input/output operands

```
%6 = linalg.generic {
    indexing_maps =
            [affine_map<(d0, d1, d2, d3) -> (d0, d1, d2, d3)>,
             affine_map<(d0, d1, d2, d3) -> (d0, d1)>],

    iterator_types = ["parallel", "parallel", "reduction", "reduction"]}

    ins(%1 : tensor<?x?x?x?xf32>) outs(%5 : tensor<?x?xf32>) {

      ^bb0(%arg1: f32, %arg2: f32):  // no predecessors
          %17 = arith.addf %arg2, %arg1 : f32
          linalg.yield %17 : f32

  } -> tensor<?x?xf32>
```

compute payload

Input/output operands

- Define the iteration space
  - decide the bounds of induction variables

Input/output operands

- Define the iteration space
  - decide the bounds of induction variables
- output operands
  - shape-only tensor: decide the Linalg operation result shape

```
ins(%1 : tensor<?x?x?x?xf32>) outs(%5 : tensor<?x?xf32>) {

  ^bb0(%arg1: f32, %arg2: f32):  // no predecessors
      %17 = “some operation” %arg1 : f32
      linalg.yield %17 : f32

} -> tensor<?x?xf32>
```

Input/output operands

- Define the iteration space
  - decide the bounds of induction variables
- output operands
  - shape-only tensor: decide the Linalg operation result shape
  - init tensor: used for destructive update

```
ins(%1 : tensor<?x?x?x?xf32>) outs(%5 : tensor<?x?xf32>) {

  ^bb0(%arg1: f32, %arg2: f32):  // no predecessors
      %17 = arith.addf %arg2, %arg1 : f32
      linalg.yield %17 : f32

} -> tensor<?x?xf32>
```

out[d0][d1] += in[d0][d1][d2][d3]

Compute payload

```
%6 = linalg.generic {
    indexing_maps =
            [affine_map<(d0, d1, d2, d3) -> (d0, d1, d2, d3)>,
             affine_map<(d0, d1, d2, d3) -> (d0, d1)>],

    iterator_types = ["parallel", "parallel", "reduction", "reduction"]}

    ins(%1 : tensor<?x?x?x?xf32>) outs(%5 : tensor<?x?xf32>) {

      ^bb0(%arg1: f32, %arg2: f32):  // no predecessors
          %17 = arith.addf %arg2, %arg1 : f32
          linalg.yield %17 : f32

    } -> tensor<?x?xf32>
```

compute payload

Compute payload

```
%6 = linalg.generic {
    indexing_maps =
          [affine_map<(d0, d1, d2, d3) -> (d0, d1, d2, d3)>,
           affine_map<(d0, d1, d2, d3) -> (d0, d1)>],

    iterator_types = ["parallel", "parallel", "reduction", "reduction"]}

    ins(%1 : tensor<?x?x?x?xf32>) outs(%5 : tensor<?x?xf32>) {

      ^bb0(%arg1: f32, %arg2: f32):  // no predecessors
          %17 = arith.addf %arg2, %arg1 : f32
          linalg.yield %17 : f32

  } -> tensor<?x?xf32>
```

compute payload

Compute payload

```
[&](OpBuilder &b, Location loc, ValueRange args) {
  Value input = args[0], sum = args[1];
  Value result = rewriter.create<arith::AddFOp>(
      loc, sum, input);
  b.create<linalg::YieldOp>(loc, result);
})
```

```
%6 = linalg.generic {
    indexing_maps =
            [affine_map<(d0, d1, d2, d3) -> (d0, d1, d2, d3)>,
             affine_map<(d0, d1, d2, d3) -> (d0, d1)>],

    iterator_types = ["parallel", "parallel", "reduction", "reduction"]}

    ins(%1 : tensor<?x?x?x?xf32>) outs(%5 : tensor<?x?xf32>) {

      ^bb0(%arg1: f32, %arg2: f32):  // no predecessors
          %17 = arith.addf %arg2, %arg1 : f32
          linalg.yield %17 : f32

    } -> tensor<?x?xf32>
```

compute payload

Compute payload

```
%6 = linalg.generic {
    indexing_maps =
        [affine_map<(d0, d1, d2, d3) -> (d0, d1, d2, d3)>,
         affine_map<(d0, d1, d2, d3) -> (d0, d1)>],

    iterator_types = ["parallel", "parallel", "reduction", "reduction"]}

    ins(%1 : tensor<?x?x?x?xf32>) outs(%5 : tensor<?x?xf32>) {

      ^bb0(%arg1: f32, %arg2: f32):  // no predecessors
        ….
        linalg.index  dim : index
        ….

    } -> tensor<?x?xf32>
```

linalg.index 0 == d0

linalg.index 1 == d1

…..

```
%6 = linalg.generic {
    indexing_maps =
            [affine_map<(d0, d1, d2, d3) -> (d0, d1, d2, d3)>,
             affine_map<(d0, d1, d2, d3) -> (d0, d1)>],

    iterator_types = ["parallel", "parallel", "reduction", "reduction"]}

    ins(%1 : tensor<?x?x?x?xf32>) outs(%5 : tensor<?x?xf32>) {

      ^bb0(%arg1: f32, %arg2: f32):  // no predecessors
            %17 = arith.addf %arg2, %arg1 : f32
            linalg.yield %17 : f32

    } -> tensor<?x?xf32>
```

```
Value sumPool2d = rewriter
                      .create<linalg::GenericOp>(
                          loc, initTensor0.getType(),
                          /*inputs=*/input, /*outputs=*/initTensor0,
                          /*indexingMaps=*/indexingMaps,
                          /*iteratorTypes=*/iteratorTypesSum,
                          [&](OpBuilder &b, Location loc, ValueRange args) {
                            Value input = args[0], sum = args[1];
                            Value result = rewriter.create<arith::AddFOp>(
                                loc, sum, input);
                            b.create<linalg::YieldOp>(loc, result);
                          })
                      .getResult(0);
```

```
%6 = linalg.generic {
  indexing_maps =
        [affine_map<(d0, d1, d2, d3) -> (d0, d1, d2, d3)>,
         affine_map<(d0, d1, d2, d3) -> (d0, d1)>],

  iterator_types = ["parallel", "parallel", "reduction", "reduction"]}

  ins(%1 : tensor<?x?x?x?xf32>) outs(%5 : tensor<?x?xf32>) {

    ^bb0(%arg1: f32, %arg2: f32):  // no predecessors
         %17 = arith.addf %arg2, %arg1 : f32
         linalg.yield %17 : f32

  } -> tensor<?x?xf32>
```

```
Value sumPool2d = rewriter
                      .create<linalg::GenericOp>(
                          loc, initTensor0.getType(),
                          /*inputs=*/input, /*outputs=*/initTensor0,
                          /*indexingMaps=*/indexingMaps,
                          /*iteratorTypes=*/iteratorTypesSum,
                          [&](OpBuilder &b, Location loc, ValueRange args) {
                            Value input = args[0], sum = args[1];
                            Value result = rewriter.create<arith::AddFOp>(
                                loc, sum, input);
                            b.create<linalg::YieldOp>(loc, result);
                          })
                      .getResult(0);
```

```
%6 = linalg.generic {
    indexing_maps =
            [affine_map<(d0, d1, d2, d3) -> (d0, d1, d2, d3)>,
             affine_map<(d0, d1, d2, d3) -> (d0, d1)>],

    iterator_types = ["parallel", "parallel", "reduction", "reduction"]}

    ins(%1 : tensor<?x?x?x?xf32>) outs(%5 : tensor<?x?xf32>) {

      ^bb0(%arg1: f32, %arg2: f32):  // no predecessors
          %17 = arith.addf %arg2, %arg1 : f32
          linalg.yield %17 : f32

    } -> tensor<?x?xf32>
```

```
Value sumPool2d = rewriter
                      .create<linalg::GenericOp>(
                          loc, initTensor0.getType(),
                          /*inputs=*/input, /*outputs=*/initTensor0,
                          /*indexingMaps=*/indexingMaps,
                          /*iteratorTypes=*/iteratorTypesSum,
                          [&](OpBuilder &b, Location loc, ValueRange args) {
                            Value input = args[0], sum = args[1];
                            Value result = rewriter.create<arith::AddFOp>(
                                loc, sum, input);
                            b.create<linalg::YieldOp>(loc, result);
                          })
                      .getResult(0);
```

```
%6 = linalg.generic {
    indexing_maps =
            [affine_map<(d0, d1, d2, d3) -> (d0, d1, d2, d3)>,
             affine_map<(d0, d1, d2, d3) -> (d0, d1)>],

    iterator_types = ["parallel", "parallel", "reduction", "reduction"]}

    ins(%1 : tensor<?x?x?x?xf32>) outs(%5 : tensor<?x?xf32>) {

      ^bb0(%arg1: f32, %arg2: f32):  // no predecessors
          %17 = arith.addf %arg2, %arg1 : f32
          linalg.yield %17 : f32

    } -> tensor<?x?xf32>
```

```
Value sumPool2d = rewriter
                      .create<linalg::GenericOp>(
                          loc, initTensor0.getType(),
                          /*inputs=*/input, /*outputs=*/initTensor0,
                          /*indexingMaps=*/indexingMaps,
                          /*iteratorTypes=*/iteratorTypesSum,
                          [&](OpBuilder &b, Location loc, ValueRange args) {
                            Value input = args[0], sum = args[1];
                            Value result = rewriter.create<arith::AddFOp>(
                                loc, sum, input);
                            b.create<linalg::YieldOp>(loc, result);
                          })
                      .getResult(0);
```

```
%6 = linalg.generic {
    indexing_maps =
        [affine_map<(d0, d1, d2, d3) -> (d0, d1, d2, d3)>,
         affine_map<(d0, d1, d2, d3) -> (d0, d1)>],

    iterator_types = ["parallel", "parallel", "reduction", "reduction"]}

    ins(%1 : tensor<?x?x?x?xf32>) outs(%5 : tensor<?x?xf32>) {

      ^bb0(%arg1: f32, %arg2: f32):  // no predecessors
          %17 = arith.addf %arg2, %arg1 : f32
          linalg.yield %17 : f32

    } -> tensor<?x?xf32>
```

```
Value sumPool2d = rewriter
                      .create<linalg::GenericOp>(
                          loc, initTensor0.getType(),
                          /*inputs=*/input, /*outputs=*/initTensor0,
                          /*indexingMaps=*/indexingMaps,
                          /*iteratorTypes=*/iteratorTypesSum,
                          [&](OpBuilder &b, Location loc, ValueRange args) {
                            Value input = args[0], sum = args[1];
                            Value result = rewriter.create<arith::AddFOp>(
                                loc, sum, input);
                            b.create<linalg::YieldOp>(loc, result);
                          })
                      .getResult(0);
```

```
%6 = linalg.generic {
    indexing_maps =
            [affine_map<(d0, d1, d2, d3) -> (d0, d1, d2, d3)>,
             affine_map<(d0, d1, d2, d3) -> (d0, d1)>],

    iterator_types = ["parallel", "parallel", "reduction", "reduction"]}

    ins(%1 : tensor<?x?x?x?xf32>) outs(%5 : tensor<?x?xf32>) {

      ^bb0(%arg1: f32, %arg2: f32):  // no predecessors
          %17 = arith.addf %arg2, %arg1 : f32
          linalg.yield %17 : f32

    } -> tensor<?x?xf32>
```

```
Value N = getDimOp(rewriter, loc, input, 0);
Value C = getDimOp(rewriter, loc, input, 1);
Value initTensor = rewriter.create<linalg::InitTensorOp>(
    loc, ValueRange{N, C}, elementType);
Value c0 = rewriter.create<arith::ConstantOp>(
    loc, FloatAttr::get(elementType, 0.0));
Value initTensor0 =
    rewriter.create<linalg::FillOp>(loc, c0, initTensor).getResult(0);

SmallVector<AffineExpr, 2> ncExprs;
ncExprs.push_back(mlir::getAffineDimExpr(0, context));
ncExprs.push_back(mlir::getAffineDimExpr(1, context));
auto ncIndexingMap = AffineMap::get(
    /*dimCount=*/4,
    /*symbolCount=*/0, ncExprs, context);
SmallVector<AffineMap, 2> indexingMaps = {
    rewriter.getMultiDimIdentityMap(4), // input
    ncIndexingMap,                      // output
};
SmallVector<StringRef, 4> iteratorTypesSum{"parallel", "parallel",
                                           "reduction", "reduction"};
Value sumPool2d = rewriter
                      .create<linalg::GenericOp>(
                          loc, initTensor0.getType(), input, initTensor0,
                          /*indexingMaps=*/indexingMaps,
                          /*iteratorTypes=*/iteratorTypesSum,
                          [&](OpBuilder &b, Location loc, ValueRange args) {
                            Value input = args[0], sum = args[1];
                            Value result = rewriter.create<arith::AddFOp>(
                                loc, sum, input);
                            b.create<linalg::YieldOp>(loc, result);
                          })
                      .getResult(0);
```

tensor operation:

perform a sum reduction along the H,W dimensions of a tensor<NxCxHxW>, resulting in a tensor<NxC>.

Thank you!