Linalg is All You Need to Optimize Attention

Li-Wen Chang, Chengji Yao, Ziheng Jiang, Xuanrun Zhang,
Yuanqiang Liu, Hongyu Zhu, Zhekun Zhang, and Xin Liu
ByteDance

Overview

Motivation

Linalg dialect in MLIR has been proven to be capable of delivering composable and efficient code generation through fusion, tiling, and vectorization, for various compute patterns. However, the current implementation is not enough to handle sophisticated patterns, like attentions. We propose multiple enhancement for the Linalg implementation, and make it robust to support common optimization strategies of an attention pattern.

```
func.func @single_attention(Q, KT, V) {
  …
  QK = Matmul ins (Q, KT) outs (Zeros1)
  M = ReduceMax d(1) ins (QK) outs (NegInf)
  E = Generic ins (QK, M) outs (Empty1)  // exp(QK - M)
  A = ReduceSum d(1) ins (E) outs (Zeros2)
  D = Generic ins (E, A) outs (Empty2)  // E/A
  O = Matmul ins (D, V) outs (Zeros3)
  return O
}
```

Tile [T0, 0, T2],
Interchange [2, 1, 0]

```
func.func @single_attention(Q, KT, V) {
  …
  C1 = scf:for … iter_args(Arg1 = Empty3') {              // reduction loop
   C2 = scf.for … iter_args(Arg2 = Arg1) {                // parallel loop
    SliceQK = Matmul ins (SliceQ, SliceKT) outs (Zeros1')
    SliceQK' = Matmul ins (SliceQ, KT) outs (Zeros1'')  // recompute entire seq
    SliceM = ReduceMax d(1) ins (SliceQK') outs (NegInf')  // entire seq
    SliceE = Generic ins (SliceQK, SliceM) outs (Empty1')    // exp(SliceQK - SliceM)
    SliceE' = Generic ins (SliceQK', SliceM) outs (Empty1'')   // recompute entire seq
    SliceA = ReduceSum d(1) ins (SliceE') outs (Zeros2')  // entire seq
    SliceD = Generic ins (SliceE, SliceA) outs (Empty2')  // SliceE/SliceA
    SliceO = Matmul ins (SliceD, SliceV) outs (Zeros3')
    O' = tensor.insert_slice SliceO into Arg2
    scf.yield O'
   }
   scf.yield C2
  }
  return C1
}
```

Current Tile-and-Fuse Result contains too much recomputing

Linalg Tile-and-Fuse Mechanism

The upstream Linalg Tile-and-Fuse relies on *TilingInterface* and *tileConsumerAndFuseProducerGreedilyUsingSCFForOp.*

```
func.func @fusion(…) {
  …
  B = ProducerB(…, C, …)
  …
  A = ProducerA(…, B, …)
  …
  R = ConsumerRoot(…, A, …)
  …
}
```

Regular Tiling
@ ConsumerRoot

Enqueue SliceA after

```
func.func @fusion(…) {
  …
  B = ProducerB(…, C, …)
  …
  A = ProducerA(…, B, …)
  …
  R = scf.for … iter_args(…, A' = A, …) {
   SliceA = tensor.extract_slice A'
   SliceR = ConsumerRoot(…, SliceA, …)
   R' = tensor.insert_slice SliceR into …
   scf.yield R'
  }
  …
}
```

Dequeue @ SliceA then
Tile and Fuse @ ProducerA

Enqueue
SliceB after

Linalg Fusion Enhancement

Enable Consumer-Rewriting

Tiling a specific op might need *rewriting its consumer operations*, not just *rewriting the tiling op* and *replacing its consumers' operands*.

```
func.func @fusion(…) {
  …
  R = scf.for … iter_args(…, C' = C, …) {
   SliceC = tensor.extract_slice C'
   SliceB = ProducerB(…, SliceC, …)      // When Tiling ProducerB
   SliceA = ProducerA(…, SliceB, …, D)   // a SliceB's consumer
     ➔ rewrite to e.g. SliceA = OpX(.., D, SliceB, …)
   SliceR = consumerRoot(…, SliceA, …)
   R' = tensor.insert_slice SliceR into …
   scf.yield R'
  }
  …
}
```

Fix Fusion for Reduction Tiling

A slice producer may or may not to be fused during reduction tiling. If a operand has a semantic as an initial value in reduction, its slice producer should NOT be fused.

Solution:
Avoid enqueuing a slice, when the above criteria meets.

Add New Linalg Ops

Adding new structure-like Linalg ops might not be *necessary* to optimize attention, but it can simplify the optimization process by avoiding matching generic ops and enabling the above new features.

Solution:
Add Softmax, Diag Op, and extend BatchMatmul to n-D

Other Enhanced Features

These enhanced features may or may not directly contribute to optimize attention patterns, but can either extend a fusion scope or support more fusion patterns.

To extend fusion scopes:
•*Simplify tensor dim* to avoid false breaking of dependency from stopping fusion when dynamic shapes happen.
•*Enable tensor expand_shape and collapse_shape tiling* to extend fusion crossing expand_shape/collapse_shape.

To support more fusion patterns:
•Support fusion having intermediates as outputs.

Attention Results

FlashAttention-style Tiling

```
func.func @multi_head_attention(Q, KT, V) {
  …
  QK = BatchMatmul ins (Q, KT) outs (Zeros1)
  S:4 = Softmax d(3) ins (QK) outs (Empty1, NegInf, Zeros2, Empty2)
  O = BatchMatmul ins (S#0, V) outs (Zeros3)
  return O
}
```

Tile [T0, 0, T2, 0, T4],
Interchange [0, 1, 4, 3, 2]

```
func.func @multi_head_attention(Q, KT, V) {
  …
  C1:3 = scf:for … iter_args(Arg1 = NegInf, Arg2 = Zero2, Arg3 = Zero3) {
   C2:3 = scf.for … iter_args(Arg4 = Arg1, Arg5 = Arg2, Arg6 = Arg3) {
    C3:3 = scf.for … iter_args(Arg7 = Arg4, Arg8 = Arg5, Arg9 = Arg6) {
     …
     SliceQK = BatchMatmul ins (SliceQ, SliceKT) outs (Zeros1')
     SliceS:4 = Softmax d(3) ins (SliceQK) outs (Empty1', SliceArg7, SliceArg8, Empty2')
     D = Diag ins (SliceS#3, Empty3)
     P = BatchMatmul ins (D, SliceArg9) outs (Zeros4)
     SliceO = BatchMatmul ins (SliceS#0, SliceV) outs (P)
     M' = tensor.insert_slice SliceS#1 into Arg7
     A' = tensor.insert_slice SliceS#2 into Arg8
     O' = tensor.insert_slice SliceO into Arg9
     scf.yield M' A', O'
    }
    scf.yield C3#0, C3#1, C3#2
   }
   scf.yield C2#0, C2#1, C2#2
  }
  return C1#2
}
```

Megatron-style (Split-head) Tiling

```
func.func @multi_head_attention_with_prologue_proj(X, Wq, Wk, Wv) {
  …
  BWq = Broadcast ins (Wq) out (Empty1)
  Q = BatchMatmul ins (X, BWq) outs (Zeros1)
  BWk = Broadcast ins (Wk) out (Empty2)
  K = BatchMatmul ins (X, BWk) outs (Zeros2)
  RQ = Expand_Shape (Q)  [0, 1, [2, 3]]
  TQ = Transpose ins (RQ) outs (Empty3) [0, 2, 1, 3]
  RK = Expand_Shape (K) [0, 1, [2, 3]]
  TK = Transpose ins (RK) outs (Empty4) [0, 2, 3, 1]
  QK = BatchMatmul ins (TQ, TK) outs (Zeros3)
  S:4 = Softmax d(3) ins (QK) outs (Empty5, NegInf, Zeros4, Empty6)
  …
  return O
}
```

Tile [0, T1, 0, 0, 0],
Interchange [0, 1, 2, 3, 4]

```
func.func @multi_head_attention_with_prologue_proj(X, Wq, Wk, Wv) {
  …
  C = scf.for … iter_args(Arg1 = ZerosN) {
   …
   BWq = Broadcast ins (SliceWq) out (Empty1')
   Q = BatchMatmul ins (X, BWq) outs (Zeros1')
   BWk = Broadcast ins (SliceWk) out (Empty2')
   K = BatchMatmul ins (X, BWk) outs (Zeros2')
   RQ = Expand_Shape (Q)  [0, 1, [2, 3]]
   TQ = Transpose ins (RQ) outs (Empty3') [0, 2, 1, 3]
   RK = Expand_Shape (K) [0, 1, [2, 3]]
   TK = Transpose ins (RK) outs (Empty4') [0, 2, 3, 1]
   QK = BatchMatmul ins (TQ, TK) outs (Zeros3')
   S:4 = Softmax d(3) ins (QK) outs (Empty5', NegInf', Zeros4', Empty6')
   …
  }
  return C
}
```

On-going Work

We are working on *variable length support* for a multi-head attention pattern.