387 lines (326 sloc) | 15.3 KB ...

```
/* Copyright 2021 The TensorFlow Authors. All Rights Reserved.

Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at

    http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License.
==============================================================================*/

#include <sys/types.h>

#include <string>

#include "mlir/Dialect/StandardOps/IR/Ops.h"
#include "mlir/IR/AffineMap.h"
#include "mlir/IR/BuiltinOps.h"
#include "mlir/IR/BuiltinTypes.h"
#include "mlir/IR/MLIRContext.h"
#include "mlir/IR/Operation.h"
#include "mlir/IR/OperationSupport.h"
#include "mlir/IR/TypeRange.h"
#include "mlir/Transforms/GreedyPatternRewriteDriver.h"
```

```
#include "llvm/ADT/DenseMap.h"
#include "llvm/ADT/DenseSet.h"
#include "llvm/ADT/STLExtras.h"
#include "llvm/ADT/StringExtras.h"
#include "llvm/ADT/iterator_range.h"
#include "llvm/Support/Alignment.h"
#include "llvm/Support/Casting.h"
#include "llvm/Support/ErrorOr.h"
#include "tensorflow/compiler/mlir/hlo/include/mlir-hlo/Analysis/shape_component_analysis.h"
#include "tensorflow/compiler/mlir/hlo/include/mlir-hlo/Dialect/mhlo/IR/hlo_ops.h"
#include "tensorflow/compiler/mlir/hlo/include/mlir-hlo/Dialect/mhlo/transforms/rewriters.h"
#include "tensorflow/compiler/mlir/tfrt/jit/transforms/tf_cpurt_passes.h"

namespace tensorflow {
namespace {

using llvm::ArrayRef;
using llvm::SmallVector;

using mlir::AffineExpr;
using mlir::AffineMap;
using mlir::failure;
using mlir::FuncOp;
using mlir::FunctionPass;
using mlir::Location;
using mlir::LogicalResult;
using mlir::MLIRContext;
using mlir::OpBuilder;
using mlir::RankedTensorType;
using mlir::ShapeComponentAnalysis;
using mlir::success;
using mlir::TypeRange;
using mlir::Value;
using mlir::ValueRange;
using mlir::arith::ConstantIndexOp;
using mlir::arith::ConstantOp;
using mlir::arith::IndexCastOp;

namespace linalg = mlir::linalg;
namespace mhlo = mlir::mhlo;
namespace shape = mlir::shape;
namespace tensor = mlir::tensor;

#define GEN_PASS_CLASSES
#include "tensorflow/compiler/mlir/tfrt/jit/transforms/tf_cpurt_passes.h.inc"

// ------------------------------------------------------------------------- //

// Rewrite shape.cstr_broadcastable with constant witness if can prove that
```

```
78  // shapes are broadcastable from the symbolic shapes.
79
80  class CstrBroadcastableOpLowering
81      : public mlir::OpRewritePattern<shape::CstrBroadcastableOp> {
82   public:
83    using Base = OpRewritePattern<shape::CstrBroadcastableOp>;
84
85    explicit CstrBroadcastableOpLowering(MLIRContext* ctx);
86
87    LogicalResult matchAndRewrite(shape::CstrBroadcastableOp op,
88                                  mlir::PatternRewriter& rewriter) const override;
89  };
90
91  CstrBroadcastableOpLowering::CstrBroadcastableOpLowering(MLIRContext* ctx)
92      : Base(ctx) {}
93
94  // Returns true if all of bcasted_shapes can be broadcasted with output_shape.
95  bool isKnownBroadcastable(ShapeComponentAnalysis& analysis,
96                            ValueRange bcasted_shapes, Value output_shape) {
97    auto output_shape_dims = analysis.dimensionsForShapeTensor(output_shape);
98    if (!output_shape_dims) return false;
99    for (Value shape : bcasted_shapes) {
100     auto shape_dims = analysis.dimensionsForShapeTensor(shape);
101     if (!shape_dims) return false;
102     // Iterate backwards over the smallest input shape.
103     for (auto zip : llvm::zip(llvm::reverse(*output_shape_dims),
104                               llvm::reverse(*shape_dims))) {
105       const auto& first = std::get<0>(zip);
106       const auto& second = std::get<1>(zip);
107       // TODO(ezhulenev): What to do with dimensions statically known to be
108       // zero?
109       // Numpy can only broadcast [0] with [1], however Tensorflow can broadcast
110       // [0] with any dimension size, and produces dimension of size [0].
111       // Currently we'll conservatively return failure and will not proceed with
112       // a rewrite.
113       if (first.isConstant(0) || second.isConstant(0)) return false;
114       // If either shape has a static one dimension the broadcast will always
115       // succeed.
116       if (first.isConstant(1) || second.isConstant(1)) continue;
117       // Otherwise dims have to be equal.
118       if (first != second) return false;
119     }
120   }
121   return true;
122 }
123
124 LogicalResult CstrBroadcastableOpLowering::matchAndRewrite(
125     shape::CstrBroadcastableOp op, mlir::PatternRewriter& rewriter) const {
126   ShapeComponentAnalysis shape_component_analysis;
```

```
  if (!isKnownBroadcastable(shape_component_analysis, op.getShapes(),
                            op.getShapes().front()))
    return failure();

  // Replace constraint with a true witness.
  rewriter.replaceOpWithNewOp<shape::ConstWitnessOp>(op, true);

  return success();
}

// Replace shape.broadcast with a shape if it's statically known.
class BroadcastOpLowering final
    : public mlir::OpRewritePattern<shape::BroadcastOp> {
 public:
  explicit BroadcastOpLowering(MLIRContext* ctx) : OpRewritePattern(ctx) {}

  LogicalResult matchAndRewrite(shape::BroadcastOp op,
                                mlir::PatternRewriter& rewriter) const override;
};

// Returns a shape tensor if the shapes can be broadcasted to a known shape.
// Will either return one of the shapes or a generated mix of the shapes.
llvm::Optional<Value> simplifyBroadcast(ShapeComponentAnalysis& analysis,
                                        ValueRange shapes, Location loc,
                                        OpBuilder* builder) {
  // First find the input shape with the largest rank.
  SmallVector<ArrayRef<ShapeComponentAnalysis::SymbolicDimension>> shapes_found;
  size_t maxRank = 0;
  for (auto shape : llvm::enumerate(shapes)) {
    auto found_shape = analysis.dimensionsForShapeTensor(shape.value());
    if (!found_shape) return {};
    shapes_found.push_back(*found_shape);
    maxRank = std::max(maxRank, found_shape->size());
  }

  SmallVector<const ShapeComponentAnalysis::SymbolicDimension*>
      joined_dimensions(maxRank);
  SmallVector<std::pair<Value, int64_t>> shape_and_rank_for_dim(maxRank);
  for (auto shape : llvm::enumerate(shapes_found)) {
    for (auto dim : llvm::enumerate(llvm::reverse(shape.value()))) {
      // 1 dimensions don't contribute to the final result.
      if (dim.value().isConstant(1)) continue;
      // If it's not a 1 dimension it will be present in the result. Remember
      // where it came from.
      auto index = maxRank - dim.index() - 1;
      if (!joined_dimensions[index]) {
        joined_dimensions[index] = &dim.value();
        shape_and_rank_for_dim[index] =
            std::make_pair(shapes[shape.index()], shape.value().size());
```

```
176         continue;
177       }
178       // Bail if the dimensions are neither equal nor 1.
179       if (*joined_dimensions[index] != dim.value()) return {};
180     }
181   }
182   // If the output is the same as one of the inputs just return that.
183   if (llvm::is_splat(shape_and_rank_for_dim) &&
184       shape_and_rank_for_dim[0].first) {
185     return shape_and_rank_for_dim[0].first;
186   }
187   // Otherwise rematerialize the shape from the pieces we have.
188   SmallVector<Value> elements;
189   for (int i = 0; i != maxRank; ++i) {
190     // 1 dimensions are filtered above, recreate the constant.
191     if (!shape_and_rank_for_dim[i].first) {
192       auto one = builder->getIntegerAttr(
193           shapes[0].getType().cast<RankedTensorType>().getElementType(), 1);
194       elements.push_back(builder->create<ConstantOp>(loc, one));
195       continue;
196     }
197     // Extract from one of the shapes, accounting for the reverse indexing
198     // performed by broadcast.
199     Value index = builder->create<ConstantIndexOp>(
200         loc, i - maxRank + shape_and_rank_for_dim[i].second);
201     elements.push_back(builder->create<tensor::ExtractOp>(
202         loc, shape_and_rank_for_dim[i].first, index));
203   }
204   return Value(builder->create<tensor::FromElementsOp>(loc, elements));
205 }
206
207 LogicalResult BroadcastOpLowering::matchAndRewrite(
208     shape::BroadcastOp op, mlir::PatternRewriter& rewriter) const {
209   ShapeComponentAnalysis shape_component_analysis;
210   auto new_broadcast = simplifyBroadcast(
211       shape_component_analysis, op.getShapes(), op.getLoc(), &rewriter);
212   if (!new_broadcast) return failure();
213   rewriter.replaceOp(op, {*new_broadcast});
214   return success();
215 }
216
217 // ------------------------------------------------------------------------- //
218
219 // Rewrite mhlo.dynamic_broadcast_in_dim operation into linalg.generic operation
220 // if can infer the indexing maps for the operand from the symbolic shapes.
221 class DynamicBroadcastInDimOpLowering
222     : public mlir::OpRewritePattern<mhlo::DynamicBroadcastInDimOp> {
223  public:
224   using Base = OpRewritePattern<mhlo::DynamicBroadcastInDimOp>;
```

```
225
226   explicit DynamicBroadcastInDimOpLowering(MLIRContext* ctx);
227
228   LogicalResult matchAndRewrite(mhlo::DynamicBroadcastInDimOp op,
229                                 mlir::PatternRewriter& rewriter) const override;
230 };
231
232 DynamicBroadcastInDimOpLowering::DynamicBroadcastInDimOpLowering(
233     MLIRContext* ctx)
234     : Base(ctx) {}
235
236 // Check if broadcasting `from` to `to_shape` is statically known to only have
237 // dimensions that never expand or always expand.
238 llvm::Optional<AffineMap> isNonExpandingBroadcast(
239     ShapeComponentAnalysis& analysis, Value from, Value to_shape) {
240   auto in_shape = analysis.dimensionsForShape(from);
241   auto out_shape = analysis.dimensionsForShapeTensor(to_shape);
242   if (!in_shape || !out_shape) return {};
243
244   SmallVector<AffineExpr> input_map_exprs;
245   size_t rank = out_shape->size();
246   MLIRContext* ctx = (*out_shape)[0].expr.getContext();
247   size_t d = 0;
248   auto affine_zero = getAffineConstantExpr(0, ctx);
249   for (auto zip :
250        llvm::zip(llvm::reverse(*in_shape), llvm::reverse(*out_shape))) {
251     const auto& in = std::get<0>(zip);
252     const auto& out = std::get<1>(zip);
253     bool extend = in.isConstant(1) && !out.isConstant(1);
254     input_map_exprs.push_back(extend ? affine_zero
255                                      : getAffineDimExpr(rank - d - 1, ctx));
256     ++d;
257
258     // Bail if this is neither a known expansion nor a known non-expansion.
259     if (!extend && in != out) return {};
260   }
261   // Any leading dimensions will be expanded.
262   input_map_exprs.resize(in_shape->size(), affine_zero);
263   std::reverse(input_map_exprs.begin(), input_map_exprs.end());
264   return AffineMap::get(/*dimCount=*/rank,
265                         /*symbolCount=*/0, input_map_exprs, ctx);
266 }
267
268 LogicalResult DynamicBroadcastInDimOpLowering::matchAndRewrite(
269     mhlo::DynamicBroadcastInDimOp op, mlir::PatternRewriter& rewriter) const {
270   MLIRContext* ctx = getContext();
271
272   auto in_type = op.operand().getType().dyn_cast<RankedTensorType>();
273   auto out_type = op.getResult().getType().dyn_cast<RankedTensorType>();
```

```
274   if (!in_type || !out_type) return failure();
275
276   // Check that broadcast is right-aligned (numpy style), so that operand
277   // dimensions broadcasted to match inner-most dimensions of the output.
278   auto bcast_dims = op.broadcast_dimensions().getValues<int64_t>();
279   auto expected_bcast_dims = llvm::seq<int64_t>(
280       out_type.getRank() - in_type.getRank(), out_type.getRank());
281   if (!llvm::equal(bcast_dims, expected_bcast_dims)) return failure();
282
283   ShapeComponentAnalysis shape_component_analysis;
284   auto input_map = isNonExpandingBroadcast(
285       shape_component_analysis, op.operand(), op.output_dimensions());
286   if (!input_map) return failure();
287
288   // Resolve dynamic output dimensions for the `linalg.init_tensor` operation.
289   SmallVector<Value> output_dyn_dimensions;
290   Location loc = op.getLoc();
291   int64_t rank = out_type.getRank();
292   for (size_t d = 0; d < rank; ++d) {
293     int64_t output_dim = out_type.getShape()[d];
294
295     // Skip static output dimensions, they will be resolved from the shape.
296     if (output_dim >= 0) continue;
297
298     // Resolve the dynamic size of the output dimension.
299     Value output_dyn_dim = rewriter.create<tensor::ExtractOp>(
300         loc, op.output_dimensions(),
301         ValueRange{rewriter.create<ConstantIndexOp>(loc, d)});
302
303     // Symbolic shape analysis might have given us an i32 or i64. Cast to index.
304     if (!output_dyn_dim.getType().isIndex())
305       output_dyn_dim = rewriter.create<IndexCastOp>(loc, output_dyn_dim,
306                                                     rewriter.getIndexType());
307
308     output_dyn_dimensions.push_back(output_dyn_dim);
309   }
310
311   // Create a linalg.tensor_init operation to initialize output.
312   Value init = rewriter.create<linalg::InitTensorOp>(loc, output_dyn_dimensions,
313                                                      out_type.getShape(),
314                                                      out_type.getElementType());
315
316   // Output indexing map is an identity with `rank` number of loops.
317   AffineMap output_map = AffineMap::getMultiDimIdentityMap(rank, ctx);
318
319   // All iterators are parallel.
320   SmallVector<llvm::StringRef> iterator_types(rank, "parallel");
321
322   rewriter.replaceOpWithNewOp<linalg::GenericOp>(
```

```
      op, /*resultTensorTypes=*/TypeRange{init.getType()},
      /*inputs=*/ValueRange{op.operand()},
      /*outputs=*/ValueRange{init},
      /*indexingMaps=*/llvm::makeArrayRef({*input_map, output_map}),
      /*iteratorTypes=*/iterator_types,
      [&](OpBuilder& nested_builder, Location nested_loc, ValueRange args) {
        nested_builder.create<linalg::YieldOp>(nested_loc, args[0]);
      });

  return success();
}

// ----------------------------------------------------------------------------- //
// Optimize function based on the symbolic shape attributes.
// ----------------------------------------------------------------------------- //

struct SymbolicShapeOptimizationPass
    : public SymbolicShapeOptimizationBase<SymbolicShapeOptimizationPass> {
  SymbolicShapeOptimizationPass() = default;

  explicit SymbolicShapeOptimizationPass(bool constraints_only) {
    this->optimize_only_constraints = constraints_only;
  }

  void runOnFunction() override {
    FuncOp func = getFunction();

    MLIRContext* ctx = &getContext();
    mlir::RewritePatternSet patterns(ctx);

    // Rewrite constraints based on the symbolic shapes.
    patterns.insert<CstrBroadcastableOpLowering>(ctx);
    // Rewrite shape.broadcast based on the symbolic shapes.
    patterns.insert<BroadcastOpLowering>(ctx);

    // Move broadcasts up across mhlo operations to enable more opportunities
    // for constraints and broadcasts optimizations. These patterns are only
    // applicable if we do not lower mhlo broadcasts to linalg.generic.
    if (optimize_only_constraints)
      mlir::mhlo::PopulateBroadcastsPropagationPatterns(ctx, &patterns);

    // Rewrite broadcasts based on the symbolic shapes if enabled.
    if (!optimize_only_constraints)
      patterns.insert<DynamicBroadcastInDimOpLowering>(ctx);

    // Add shape dialect canonicalization patterns to fold shape operations
    // after constraints are replaced with constant witness.
    mlir::Dialect* shape_dialect = ctx->getLoadedDialect<shape::ShapeDialect>();
    for (auto* op : ctx->getRegisteredOperations()) {
```

```
372       if (op->dialect.getTypeID() == shape_dialect->getTypeID())
373         op->getCanonicalizationPatterns(patterns, ctx);
374     }
375
376     (void)mlir::applyPatternsAndFoldGreedily(func, std::move(patterns));
377   }
378 };
379
380 }  // namespace
381
382 std::unique_ptr<FunctionPass> CreateSymbolicShapeOptimizationPass(
383     bool constraints_only) {
384   return std::make_unique<SymbolicShapeOptimizationPass>(constraints_only);
385 }
386
387 }  // namespace tensorflow
```