DisposableElementsAttr

Constant propagation with garbage collectible MLIR ElementsAttr

Soren Lassen, Groq Inc.
December 2022

This presentation proposes a new garbage collectible MLIR ElementsAttr attribute and ways to use it in constant propagation and to garbage collect it between compiler passes.

Constant propagation

Constant Folding:

```
func @foo() -> tensor<1xf32> {
  %0 = "onnx.Constant"() {value = dense<[1.0]> : tensor<1xf32>} : () -> tensor<1xf32>
  %1 = "onnx.Constant"() {value = dense<[2.0]> : tensor<1xf32>} : () -> tensor<1xf32>
  %2 = "onnx.Add"(%0, %1) : (tensor<1xf32> , tensor<1xf32>) -> tensor<1xf32>
  %3 = "onnx.Constant"() {value = dense<[3.0]> : tensor<1xf32>} : () -> tensor<1xf32>
  %4 = "onnx.Add"(%2, %3) : (tensor<1xf32> , tensor<1xf32>) -> tensor<1xf32>
  "std.return"(%4) : (tensor<1xf32>) -> ()
}
```

If we call `onnx-mlir-op --constprop-onnx`, we will get:

```
func @foo() -> tensor<1xf32> {
  %0 = "onnx.Constant"() {value = dense<[6.0]> : tensor<1xf32>} : () -> tensor<1xf32>
  "std.return"(%0) : (tensor<1xf32>) -> ()
}
```

From onnx-mlir docs/ConstPropagationPass.md

See onnx-mlir docs

Rewrites to enable Constant Folding:

```
// Use commutativity to normalize constants
// in the second position of Add.
def AddConstCommutative1 : Pat<
  // From add(c, x).
  (ONNXAddOp (ONNXConstantOp:$c ..), $x),
  // To add(x, c).
  (ONNXAddOp $x, $c),
  // To avoid infinite loop, constrain the first
  // arguments to be anything but a constant.
  [(IsNotAConstant:$x)]>;

// Use associativity to add constants together.
def AddConstAssociative1 : Pat<
  // From add(add(x, c1), c2).
  (ONNXAddOp
    (ONNXAddOp $x,(ONNXConstantOp:$c1 ..)),
    (ONNXConstantOp:$c2 ..)),
  // To add(x, add(c1, c2)).
  (ONNXAddOp
    $x,
    (ONNXAddOp $c1, $c2)),
    [(IsNotAConstant:$x)]>;
```

From onnx-mlir src/Transform/ONNX/ConstProp.td

First, what is const prop? It largely consists of constant folding and rewrites to enable constant folding.

Constant folding is when an operation's inputs are all constants and we replace the operation with the result of applying it to the constant inputs, as in the addition example on the left.

The rewrite rules on the right pushes constants in an expression tree together in the rightmost expression tree branch, which can then be constant folded.

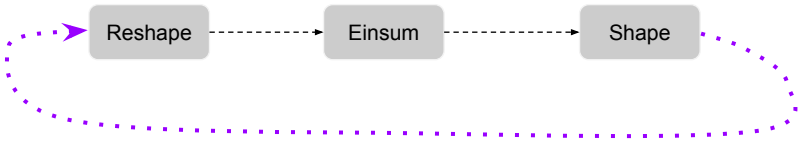
Const prop can do some compile time reduction of the input graph but another important role of const prop is to enable other compiler analyses and transformations.

Constant propagation

1. Canonicalization is another way to replace an operation with a constant
   a. Example: **“onnx.Shape”(%arg) : tensor<2x1xf32> -> tensor<2xi64>**
      canonicalizes to **“onnx.Constant() {value = dense<[2,1]> : tensor<2xi64>}**
2. Constant propagation, Canonicalization, and Shape Inference are mutually dependent
   a. Example: **“onnx.Reshape(%arg, %cst)”** has shape **%cst**

Motivation

1. XLNet model: Reshape -> Einsum -> Shape -> Reshape dependency, repeated dozens of times
   a. Reshape shape inference needs const prop
   b. Einsum decomp needs shape inference
   c. Shape canonicalization needs shape inference



2. Solutions:
   a. Compiler repeats shape inference, canonicalization, const prop, decomposition many times
   b. Or compiler combines shape inference, const prop, etc in a hybrid top-down pass...

Hybrid pass: combine shape inference, canonicalization, decomposition, const propagation

1. Goals: make frontend passes effective, robust, fast, and memory efficient
   a. "effective": find a way to apply all the shape inference, const prop, etc in long dependency chains
   b. "robust": avoid the need for artful pass ordering and repetition
   c. fast and memory efficient: of course
2. Hybrid pass:
   a. Combine into a single top-down pass patterns from all of shape inference, canonicalization, decomposition, const prop
   b. Cascade through the graph and do all the things in a single pass
   c. Express shape inference as a set of rewrite patterns (Draft PR)
   d. **Make onnx-mlir const prop interoperate with other patterns** (this presentation)
3. Caveats: needs more due diligence
   a. TODO: figure out how to play together with patterns that cannot be applied top-down
   b. TODO: avoid limiting the power and flexibility we get from MLIR's pass infrastructure

onnx-mlir constant propagation with buffer pool

1. Heap allocates intermediate constant tensors
   a. Deallocates them after the pass and replaces them with proper DenseElementsAttr
   b. Thus avoids the construction of intermediate DenseElementsAttr tensors which are “immortal”
2. Maintains high numerical precision
   a. Intermediate heap allocated tensor elements are **double** or **int64_t**..ar
3. Intermediate heap allocated tensors are represented as non-standard attributes
   a. Not interoperable with rest of code base beyond the const prop rewrite patterns
4. Not thread safe
5. onnx-mlir wants to replace buffer pool with better alternative (link to issue)
   a. Proposal: **DenseResourceElementsAttr** (draft PR)
   b. Alternative proposal: **DisposableElementsAttr** (this presentation, draft PR)

The garbage collectible attribute in this presentation is an attempt to replace the buffer pool with an interoperable alternative that preserves the two key features of the buffer pool, which are: 1st the ability to deallocate intermediate constants produced during const prop; 2nd maintaining high numerical precision of multi-hop constant folding.

DisposableElementsAttr: garbage collectible alternative to DenseElementsAttr

1. Data can be “wide” high precision cpp type, e.g. double for float16, int64_t for int32_t
2. Data can be owned or unowned, owned data can be garbage collected
3. Data is reference counted (shared_ptr) and can be shared
4. Data can be “strided” and present a view, like pytorch tensors
5. Data can apply an element-wise transformation on demand when accessed

6. Interoperable:
   a. Implements same ElementsAttr interface as DenseElementsAttr
   b. Hides implementation complexity behind convenient interface
   c. Prints the same as DenseElementsAttr (no lit tests changes)
7. Efficient:
   a. Low memory usage (apart from wide cpp types): shared buffers, strides, garbage collection
   b. Avoids most memory copies
   c. Inlines functions in inner loops without template instantiation combinatorial explosion

7

The new garbage collectible attribute is called DisposableElementsAttr (DispEA).

It is designed to enable constant propagation with low memory usage and high numerical precision.

It’s meant as a plug-in replacement for DenseElementsAttr to represent tensor constants in onnx-mlir.

DispEA implements the “ElementsAttr” interface and, fortunately, the onnx-mlir code access DenseElementsAttr through the methods of the “ElementsAttr” interface, so minimum code changes are needed to interoperate with the new DispEA.

The interface access methods abstract from the underlying representation which allows DispEA to do all of the following:

(1) The underlying data can use a “wide” higher precision data type than the tensor element type. For example, float32 or float16 can be represented with float64 to maximize numerical precision of constant propagation.

(2) The underlying data can point to long lived constant data, like the underlying raw data of a DenseElementsAttr, or it can be heap allocated memory owned by the attribute that’s deallocated when the attribute is garbage collected, or it can point to the memory mapping of a file owned by the attribute that’s released when the attribute is garbage collected. [More precisely, it’s the

(1) attribute _storage_ that owns memory or memory mapping and which is garbage collected, but I gloss over that detail here.]

(2) The underlying data is reference counted with shared_ptr, which allows different attribute to share underlying data.

(3) An attribute can present a strided view of the data, like pytorch tensors, so you can transpose, reshape, and broadcast without changing the underlying data.

(4) An attribute can represent an element-wise transformation of the data which is applied on demand when the data is accessed.

While the “wide” higher precision data can increase the memory footprint, the other features minimize footprint of constants constructed during constant propagation, and support their garbage collection after the constant propagation pass.

Some other design criteria are speed, code size, and printing.

DenseResourceElementsAttr exists and fulfills some of the requirements but doesn’t abstract from the underlying representation, in particular it doesn’t support a representation with a wider data type. However, it validates the concept of heap allocated backing storage which can be deallocated.

Performance: Speed and Memory Usage

1. Tested on models from the model zoo
   a. resnet152-v1-7, arcfaceresnet100-8 did the most constant propagation
   b. mostly elementwise binary ops, approximately 800 ops, 50M elements total

2. DisposableElementsAttr sped up const prop 20x compared to existing buffer pool:
   a. ConstPropONNXToONNXPass timing went from 17s to 0.8s on a MacBook

3. DisposableElementsAttr more than halved the total memory usage:
   a. Peak memory usage dropped 56% from 1.3GB to 570MB
   b. Memory footprint after ONNX frontend passes dropped 70% from 515MB to 151MB

See the details in PR #1874 description

DisposableElementsAttributeStorage

```
struct DisposableElementsAttributeStorage : public AttributeStorage {

        ShapedType type;

        ArrayRef<int64_t> strides;

        onnx_mlir::DataType bufferDType;

        shared_ptr<llvm::MemoryBuffer> buffer;

        function<void(StringRef, MutableArrayRef<WideNum>)> reader;

        // constructor etc …  (doesn't participate in storage uniqueing)
};
```


This is how DisposableElementsAttr is stored. It has a ShapedType like any ElementsAttr and I'll describe the other fields and the DataType and WideNum data types in the following slides.

One technical thing is bypassing MLIR's storage uniquer: Instances of DisposableElementsAttributeStorage are not deduplicated by the storage uniquer because it's infeasible to do that for the MemoryBuffer and reader function.

The bufferDType field can express that buffer contains elements of a different data type than the ShapedType's element type. This is needed to represent const prop intermediate data in higher precision.

enum class onnx_mlir::DataType

```
enum class DataType { FLOAT = 1, UINT8 = 2, INT8 = 3, ...}; // mirrors onnx::TensorProto_DataType

assert( mlirTypeOfDType(DataType::FLOAT).isF32() == true );
assert( dtypeOfMlirType(mlir::FloatType::getF32(ctx)) == DataType::FLOAT );

static_assert( is_same_v<CppType<DataType::FLOAT>, float> );
static_assert( toDType<float> == DataType::FLOAT );

DenseElementsAttr splatAbs(DenseElementsAttr splat) {
  return dispatchByMlirType(splat.getType().getElementType(), [splat](auto dtype) {
    using cpptype = CppType<dtype>;
    cpptype v = dense.getSplatValue<cpptype>();
    if constexpr (!is_integral_v<cpptype>)      v = fabs(static_cast<double>(v));
    else if constexpr (is_signed_v<cpptype>) v = llabs(static_cast<int64_t>(v));
    else return splat; // abs is identity on uint
    return DenseElementsAttr::get(splat.getType(), v); });
}
```


DataType is an enum that replicates ONNX's Tensor::DataType enum.

I defined mappings between DataType and mlir::Type. DataType is sometimes more convenient. For instance you can use a constant enum as a template parameter and e.g. translate from enum to cpp type and back with templates.

With these things you can dispatch on an element type. Here we take a splat DenseElementsAttr, where "splat" means that all the tensor's elements have the same value, and we return another splat DenseElementsAttr with the absolute value, computed using either fabs on doubles or llabs on long long ints.

The dispatch function makes it super easy to do template instantiation of every combination of input and output types to all operations, but we run the risk of generating an excessive number of code paths. If we ignore strings and complex numbers, there are 13 different floating point and signed and unsigned integer types.

The example illustrates that they can all be promoted without loss of precision to one of the 64-bit wide data types DOUBLE, INT64, UINT64.

union WideNum

```
// Untagged union. Must be tagged by a DType, or corresponding mlir::Type.
union WideNum { double dbl; int64_t i64; uint64_t u64; };

template <typename WSRC, typename WDST> // WSRC, WDST are 64 bit wide types
void wideCast(MutableArrayRef<WideNum> nums) {
  for (WideNum &n : nums)
    n = reinterpret_cast<WideNum>(static_cast<WDST>((reinterpret_cast<WSRC>(n)));
}
void castElementType(MutableArrayRef<WideNum> nums, Type srcType, Type dstType) {
  DataType src = wideDTypeOfMlirType(srcType), dst = wideDTypeOfMlirType(dstType);
  if (src == DOUBLE && dst == INT64)     wideCast<double, int64_t>(nums);
  if (src == DOUBLE && dst == UINT64)  wideCast<double, uint64_t>(nums);
  if (src == INT64     && dst == DOUBLE) wideCast<int64_t, double>(nums);
  if (src == INT64     && dst == UINT64)  wideCast<int64_t, uint64_t>(nums);
  if (src == UINT64  && dst == DOUBLE) wideCast<uint64_t, double>(nums);
  if (src == UINT64  && dst == INT64)     wideCast<uint64_t, int64_t>(nums);
}
```


Building on the buffer pool idea of representing data with high precision, const prop with DisposableElementsAttr defaults to representing numbers with 64 bits, which can be represented as this union named WideNum. (The name is a riff on the arbitrary precision “BigNum” data type.)

In many cases we can process data promoted to 64 bits as opaque WideNums, e.g. in const prop of Slice and Scatter and Gather we can move around elements uninterpreted as 64 bis WideNums without need for templates.

In other cases where we need to interpret the bits, we just need to consider the 3 wide types instead of all the 13 DataTypes.

E.g. to cast one type to another, if we represent the elements as WideNums we only need to consider 3x3 combinations of source and destination types, instead of 13x13. Here’s the code to cast between WideNums, where the 3x3 cases boil down to only 6 when we ignore casting to the same type.

ArrayRef<int64_t> strides

1. Like pytorch strides (more or less)
   a. Broadcast (ONNX Expand op) can always be expressed as strides metadata
   b. Transpose (ONNX Transpose op)
   c. Reshape (ONNX Reshape, Squeeze, Unsqueeze, Flatten ops): sometimes
2. Generalizes “splat” from DenseElementsAttr: splat is the special case of broadcasting a scalar

Like pytorch tensor views. You can broadcast and transpose a DisposableElementsAttr by creating a new one that shares buffer and reader and just has different strides. The same goes for reshape, unless it has already been transposed to incompatible strides.

shared_ptr<llvm::MemoryBuffer> buffer

1. llvm::MemoryBuffer:
   a. Can point to unowned long lived memory.
      i. Can be used to wrap DenseElementsAttr.
   b. Can own heap allocated memory and deallocate it on when destroyed.
      i. Can be used with garbage collector.
   c. Can own mmap of a file and close it when destroyed.
      i. Can be used with garbage collector.
2. shared_ptr:
   a. Reference counted
      i. MemoryBuffer is destroyed when last occurrence is garbage collected

llvm::MemoryBuffer is a handy memory wrapper. We already use it to read files when we parse an ONNX protobuf model into MLIR datastructures.

A MemoryBuffer can point to unowned memory that has a longer life time than the buffer. This can be used to wrap a DisposableElementsAttr around a DenseElementsAttr because a DenseElementsAttr lives forever. E.g. you can transpose or broadcast a DenseElementsAttr that way without touching or allocating memory.

A MemoryBuffer can also own heap allocated memory and deallocate it on destruction. This can be used for garbage collection with a garbage collector which can figure out when the storage of DisposableElementsAttr is unreachable.

A MemoryBuffer can also own a memory mapping of a file and release it on destruction. This can be used with ONNX graphs with so-called external data where tensor data is stored in standalone files, or we could write large tensors to files during compilation to save memory.

reader(StringRef, MutableArrayRef<WideNum>)

1. reader performs element-wise transformation from bufferDType to “element type”
   a. “element type” = WideNum tagged by type.getElementType()
      i. i.e. the WideNum is a double/i64/u64 depending on whether type.getElementType() is float or int, signed or unsigned
2. reads StringRef from MemoryBuffer, writes WideNums to array
   a. transforms as many elements as the size of the array
      i. reader runs the innermost loop, avoids function invocation of reader for every element
3. can be composed with a follow-on transformation that mutates the array again.
   a. thus we can compose an element-wise transformation onto any DisposableElementsAttr.
   b. e.g. const prop unary element-wise op, like ONNX Neg, Sqrt, Cast.
   c. or const prop of a binary element-wise op where one arg is splat.

DisposableElementsAttr: access methods

```
class DisposableElementsAttr
    : public AttrBase<.. DisposableElementsAttributeStorage, ElementsAttr::Trait, TypedAttr::Trait> {
      public:
        // ElementsAttr access:
        ShapedType getType() const;
        bool isSplat() const;
        template <typename X> X getSplatValue() const;
        template <typename X> iterator_range<X> getValues() const; // slow if strides not contiguous

        // Bulk access:
        // Copies out the elements in a flat array in row-major order.
        void readWideNums(MutableArrayRef<WideNum> dst) const;
        // Returns buffer with all elements. Zero copy if bufferDType matches, data is not strided, transformed.
        template <typename X> onnx_mlir::ArrayBuffer<X> getArray() const;
```


All the funky data representation is hidden from the attribute API. You can access it like any ElementsAttr, which is in fact how onnx-mlir accesses to DenseElementsAttr, so almost no changes are needed to put the new attribute into use. The only thing is that the getValues() iterator range access is slow because of how the ElementsAttr interface plumbs things.

Therefore I've added two efficient bulk access methods for all the elements as a flat array. To minimize copies, the caller has a choice to get it copied out to a specified array with the readWideNums() method, or get a read-only array with the getArray() method.

ArrayBuffer is a little bit like MemoryBuffer - it can act as just a "pass through" ArrayRef or it can own a heap allocated vector which is deallocated when it exits lexical scope, unless it's moved. It enables us to offer optimistic zero-copy access to the underlying data, or hold the copied out data if needed.

ElementsAttrBuilder & onnx-mlir const prop

```
class ElementsAttrBuilder { public: using DispElmsAttr = DisposableElementsAttr; // abbreviation
        ElementsAttrBuilder(DisposablePool &disposablePool /* instances pool for GC */);

        template <typename T> using Filler = function<void(MutableArrayRef<T>)>;
        DispElmsAttr fromWideNums(ShapedType type, const Filler<WideNum> &filler);

        DispElmsAttr fromElementsAttr(ElementsAttr elements); // wraps without copy

        using Transformer = function<void(MutableArrayRef<WideNum>)>;
        DispElmsAttr transform(DispElmsAttr elms, Type transformedElementType, Transformer transformer);
        DispElmsAttr castElementType(DispElmsAttr elms, Type newElementType);
        DispElmsAttr transpose(DispElmsAttr elms, ArrayRef<uint64_t> perm);
        DispElmsAttr reshape(DispElmsAttr elms, ArrayRef<int64_t> reshapedShape);
        DispElmsAttr expand(DispElmsAttr elms, ArrayRef<int64_t> expandedShape);
```


All instantiation of DisposableElementsAttr is delegated to the friend class ElementsAttrBuilder.
It has a reference to a DisposablePool which records all DisposableElementsAttr instances for the purposes of garbage collection.

I listed most of the instance creation methods here and const prop largely works through these. Notice that the inner workings of strides, bufferDType, llvm::MemoryBuffer, and reader are largely hidden.

ElementsAttrBuilder can manipulate all these as a friend class of DisposableElementsAttr and it puts it to use in the last 5 methods which in most cases are able to construct new instances that share the memory of the input ElementsAttr.

reshape() can just wrap the input with new shape and strides if everything aligns, otherwise it copies the underlying data into a layout that works.

Const prop can fold constants for the obvious ONNX ops using the last 5 methods, otherwise it calls fromWideNums() and constructs all the elements in place without strides and transform in a MemoryBuffer that ElementsAttrBuilder creates. Const prop just needs to work withWideNums and know that they are “tagged” by the element type of the passed in ShapedType.

I migrated all the onnx-mlir buffer pool const prop for ONNX to

DisposableElementsAttr, including scatter, gather, slice, split, and concat, and I believe that I demonstrated that the API works. In particular, the implementation complexity doesn't leak through the API to make the ONNX const prop code more complicated, so we reap the benefits without adding application complexity.

GarbageCollection: DisposablePool

```
class DisposablePool : public DialectInterface::Base<DisposablePool> {
public:
  static DisposablePool &create(MLIRContext *context); // calls onnxDialect.addInterface(..)
  static DisposablePool *get(MLIRContext *context);      // calls onnxDialect.getRegisteredInterface(..)

  // Disposes every DisposableElementsAttr in the pool which is unreachable (doesn't appear in moduleOp).
  void garbageCollectUnreachable(ModuleOp moduleOp);

  // Disposes every DisposableElementsAttr and in moduleOp replaces each with a DenseElementsAttr.
  void scrub(ModuleOp moduleOp);
private:
  unordered_set<DisposableElementsAttributeStorage *> pool;

  void insert(DisposableElementsAttr disposable);
  friend class ElementsAttrBuilder; // allow access to insert()
};
```


The garbage collection works by recording DispElmsAttr instances when they are created. The pool has a method garbageCollectUnreachable() which we call between ONNX frontend passes, and a scrub() method that's called just before ONNX is lowered (to Krnl or other lowering targets).

GarbageCollection: DisposableGarbageCollector

```
struct DisposableGarbageCollector : public PassInstrumentation {

        DisposablePool &disposablePool;

        void runAfterPass(Pass *pass, Operation *op) override {
            if (ModuleOp moduleOp = dyn_cast<ModuleOp>(op))
                disposablePool.garbageCollectUnreachable(moduleOp);
        }
};
```


We register a garbage collector as a PassInstrumentation with the PassManager and it runs after every pass and, if it's a module pass, it calls DisposablePool to garbage collect everything that's unreachable from the module op.

DisposableElementsAttr: printing and parsing

1. DisposableElementsAttr is an implementation detail so would be nice if it would just print “transparently” as a DenseElementsAttr
   a. Then we don’t need to change all the lit tests
   b. (No need to parse DisposableElementsAttr, DenseElementsAttr always ok when we parse mlir assembly)

2. But MLIR insists on prefixing the attribute printing with “#onnx.” prefix

3. Idea 1: Intercept the mlir assembly output in the main() function (implemented in PR #1874)
   a. In the onnx-mlir and onnx-mlir-opt executables
   b. Regex replace #onnx.disposable<...> with dense<...>

4. Idea 2: With new ONNXConstantOp custom assembly format (PR #1898), print DisposableElementsAttr like DenseElementsAttr

Next steps

1. Run the proposal by the MLIR community
   a. conflicts with MLIR best practice and roadmap?
   b. anything worth adding to MLIR?

2. Use DisposableElementsAttr to implement fold methods, see issue #1718 and draft PR #1902

3. Incorporate constant folding into hybrid pass in draft PR #1770