One-Shot Function Bufferization of Tensor Programs

MLIR Open Design Meeting (13 Jan 2022)
Matthias Springer (springerm@google.com), Nicolas Vasilache (ntv@google.com)
Outline

- **Bufferization**: Allocating + assigning memref buffers to tensor values.
- Current bufferization solutions in MLIR
  - **Core bufferization**: Multiple passes (one per dialect), conservative (always copy on write)
  - **One-Shot Bufferization** (**Comprehensive Bufferize**): Single pass, comes with an analysis (copy buffers only if deemed necessary).
- This talk:
  - Why we need something better than core bufferization. Design + Implementation sketch of One-Shot Bufferization.
  - How we can **consolidate** both bufferization solutions into a single one while maintaining compatibility with existing code.
  - How users can (gradually) **extend** the new bufferization with their own ops.
  - Focusing on function body bufferization. (No focus on **Module Bufferization**.)
What is Bufferization?
And why it is difficult.
Challenges in Bufferization

*Bufferization:* Convert IR with tensors into IR with memrefs.

*Challenge 1:* **Use as little memory as possible.** I.e., try to keep the number of memory allocations ("buffers") small and try to reuse existing buffers when possible.

*Challenge 2:* **Copy as little memory as possible.**
Example

```plaintext
func @foo(%a : tensor<?xf32>, %f : f32, %idx0 : index, %idx1 : index)
  -> (f32, f32)
{
  %b = tensor.insert %f into %a[%idx0] : tensor<?xf32>
  %c = tensor.extract %a[%idx1] : tensor<?xf32>
  %d = tensor.extract %b[%idx1] : tensor<?xf32>

  return %c, %d : f32, f32
}
```
Example: Tensor Values

```plaintext
func @foo(%a : tensor<?xf32>, %f : f32, %idx0 : index, %idx1 : index)
  -> (f32, f32)
{
  %b = tensor.insert %f into %a[%idx0] : tensor<?xf32>
  %c = tensor.extract %a[%idx1] : tensor<?xf32>
  %d = tensor.extract %b[%idx1] : tensor<?xf32>

  return %c, %d : f32, f32
}
```
Example: Ops with Tensor Semantics

```clojure
func @foo(%a : tensor<?xf32>, %f : f32, %idx0 : index, %idx1 : index)
  -> (f32, f32)
{
  %b = tensor.insert %f into %a[%idx0] : tensor<?xf32>
  %c = tensor.extract %a[%idx1] : tensor<?xf32>
  %d = tensor.extract %b[%idx1] : tensor<?xf32>
  return %c, %d : f32, f32
}
```

func @foo(%a : tensor<?xf32>, %f : f32, %idx0 : index, %idx1 : index)
-> (f32, f32) {

  %b = tensor.insert %f into %a[%idx0] : tensor<?xf32>
  %c = tensor.extract %a[%idx1] : tensor<?xf32>
  %d = tensor.extract %b[%idx1] : tensor<?xf32>

  return %c, %d : f32, f32
}

memref.store into buffer(%a)

memref.load from buffer(%a) before the store (RaW conflict)
func @foo(%a : tensor<?xf32>, %f : f32, %idx0 : index, %idx1 : index) 
-> (f32, f32)
{
  %c = tensor.extract %a[%idx1] : tensor<?xf32>
  %b = tensor.insert %f into %a[%idx0] : tensor<?xf32>
  %d = tensor.extract %b[%idx1] : tensor<?xf32>

  return %c, %d : f32, f32
}
What if we Swap Ops?

```
func @foo(%a: tensor<?xf32>, %f: f32, %idx0 : index, %idx1 : index)
  -> (f32, f32)
{
  %c = tensor.extract %a[%idx1]
  %b = tensor.insert %f into %a[%idx0]
  %d = tensor.extract %b[%idx1]

  return %c, %d : f32, f32
}
```

Not something that we currently consider.
Example: One Possible Bufferization

```
func @foo(%a : memref<?xf32>, %f : f32, %idx0 : index, %idx1 : index)
  -> (f32, f32)
{
  %b = memref.alloc(...) : memref<?xf32>
  memref.copy %a, %b : memref<?xf32>
  memref.store %f, %b[%idx0] : memref<?xf32>
  %c = memref.load %a[%idx1] : memref<?xf32>
  %d = memref.load %b[%idx1] : memref<?xf32>

  return %c, %d : f32, f32
}
```

Instead of allocating: Maybe we can reuse another buffer that is not being used at the moment? (Challenge #1)
Why We Need Something Better
Example: Core Bufferization

```plaintext
%t0 = ...
%t1 = vector.transfer_write %data1, %t0[%c0]
%t2 = vector.transfer_write %data2, %t1[%c5]
%t3 = vector.transfer_write %data3, %t2[c10]
```

// Do something with %t3

Always copy a buffer before it is modified. Simple, no need to worry about RaW conflicts.
Where we really don’t want Copies

%1 = tensor.extract_slice %arg1[%iv][%c10][1]

%2 = vector.transfer_write %v, %1[%pos]

%3 = tensor.insert_slice %2 into %arg1[%iv][%c10][1]

This should bufferize without memory copies.
Why Improve Bufferization?

Core bufferization is conservative: Always copy

\[
%1 = \text{tensor.extract_slice} \ %\text{arg1}[%\text{iv}][%\text{c10}][1]
\]

\[
%2 = \text{vector.transfer_write} \ %v, \ %1[%\text{pos}]
\]

\[
%3 = \text{tensor.insert_slice} \ %2 \ \text{into} \ %\text{arg1}[%\text{iv}][%\text{c10}][1]
\]

This should bufferize without memory copies.
%arg1_memref = bufferization.to_memref %arg1

%0 = memref.subview %arg0[%iv] [%c10] [1]
vector.transfer_write %v, %0[%pos]

%3 = bufferization.to_memref %arg1_memref
It can get even more complicated...

```plaintext
%r1 = scf.for %iv = %c0 to %ub step %c10 iter_args(%arg1 = %arg0) ... {
  %1 = tensor.extract_slice %arg1[%iv][%c10][1]
  %r2 = scf.if %cond ... {
    %2 = vector.transfer_write %v, %1[%pos]
    scf.yield %2
  } else {
    scf.yield %1
  }
  %3 = tensor.insert_slice %r2 into %arg1[%iv][%c10][1]
  scf.yield %3
}
```

This should bufferize without memory copies.
How To Use Bufferization?
From a user’s perspective...
Example: Core Bufferization (1)

```mlir
func @foo(%sz : index, %f : f32, %idx1 : index, %idx2 : index) -> f32 {
  %t = linalg.init_tensor [%sz] : tensor<?xf32>
  %1 = tensor.insert %f into %t[%idx1] : tensor<?xf32>
  %2 = tensor.extract %1[%idx2] : tensor<?xf32>
  return %2 : f32
}
```
Example: Core Bufferization (1)

```mlir
// RUN: mlir-opt %s --linalg-bufferize --tensor-bufferize --finalizing-bufferize

func @foo(%sz : index, %f : f32, %idx1 : index, %idx2 : index) -> f32 {
    %t = linalg.init_tensor [%sz] : tensor<?xf32>
    %1 = tensor.insert %f into %t[%idx1] : tensor<?xf32>
    %2 = tensor.extract %1[%idx2] : tensor<?xf32>
    return %2 : f32
}
```
Example: Core Bufferization (2)

```mlir
// RUN: mlir-opt %s --linalg-bufferize --tensor-bufferize --finalizing-bufferize

func @foo(%sz : index, %f : f32, %idx1 : index, %idx2 : index) -> f32 {
  %t_m = memref.alloc(%sz) : memref<?xf32>
  %t = bufferization.to_tensor %t_m : memref<?xf32>
  %1 = tensor.insert %f into %t[%idx1] : tensor<?xf32>
  %2 = tensor.extract %1[%idx2] : tensor<?xf32>
  return %2 : f32
}
```
Example: Core Bufferization (2)

```mlir
// RUN: mlir-opt %s --linalg-bufferize --tensor-bufferize --finalizing-bufferize

func @foo(%sz : index, %f : f32, %idx1 : index, %idx2 : index) -> f32 {
  %t_m = memref.alloc(%sz) : memref<?xf32>
  %t = bufferization.to_tensor %t_m : memref<?xf32>
  %1 = tensor.insert %f into %t[%idx1] : tensor<?xf32>
  %2 = tensor.extract %1[%idx2] : tensor<?xf32>
  return %2 : f32
}
```
Example: Core Bufferization (2)

// RUN: mlir-opt %s --linalg-bufferize --tensor-bufferize --finalizing-bufferize

func @foo(%sz : index, %f : f32, %idx1 : index, %idx2 : index) -> f32 {
  %t_m = memref.alloc(%sz) : memref<?xf32>
  %t = bufferization.to_tensor %t_m : memref<?xf32>
  %1 = tensor.insert %f into %t[%idx1] : tensor<?xf32>
  %2 = tensor.extract %1[%idx2] : tensor<?xf32>
  return %2 : f32
}

- Wrap in to_tensor / to_memref.
- Allocate + copy %t.
- Rewrite to memref.store into copy.
- Use to_tensor(copy) instead of %1 from now on.
func @foo(%sz : index %f : f32, %idx1 : index, %idx2 : index) -> f32 {
  %t_m = memref.alloc(%sz) : memref<?xf32>
  %d = memref.dim %t_m, %c0 : memref<?xf32>
  %copy = memref.alloc(%d) : memref<?xf32>
  memref.copy %t_m, %copy : memref<?xf32>
  memref.store %f, %copy[%idx1] : memref<?xf32>
  %2 = memref.load %copy[%idx2] : memref<?xf32>
  return %2 : f32
}
One-Shot Bufferization: Final Result

```mlir
// RUN: mlir-opt %s -test-comprehensive-function-bufferize

func @foo(%sz : index, %f : f32, %idx1 : index, %idx2 : index) -> f32 {
  %t_m = memref.alloc(%sz) {alignment = 128 : i64} : memref<?xf32>
  memref.store %f, %t_m[%idx1] : memref<?xf32>
  %2 = memref.load %t_m[%idx2] : memref<?xf32>
  memref.dealloc %t_m : memref<?xf32>
  return %2 : f32
}
```

- single pass
- no copy
Call One-Shot Bufferization Programmatically

Call bufferization directly wherever you need it.

```java
BufferizationOptions options;
// Set bufferization options
if (failed(runComprehensiveBufferize(op_to_bufferize, options)))
    return failure();
```
How Does One-Shot Bufferize Work?
A look behind the scenes...
One-Shot Bufferization is...

- Currently still called *Comprehensive Bufferize*, to be renamed and moved to the bufferization dialect.
- **Monolithic**: Whole function bufferization in a single pass.
- **Compatible** with the existing core bufferization passes.
- **Greedy**: Makes bufferization decisions based on *heuristics*. Solving bufferization perfectly is probably NP-hard.
- Designed to run after other transformations (e.g., tiling, fusing, vectorization, ...)

What is One-Shot Bufferize?

- Analysis of tensor SSA use-def chains

Analyze IR and decide where to insert copies. Could be replaced with a different analysis (and different heuristics).
What is One-Shot Bufferize?

- Analysis of tensor SSA use-def chains
- **BufferizableOpInterface**
  - An op interface that specifies bufferization properties of bufferizable ops.
  - Used by the analysis to understand how an op behaves.
  - Also contains the rewrite logic.

```cpp
LogicalResult MyOp::bufferize(RewriterBase &rewriter,
                               const BufferizationState &state) {
  // Create a new op with memref semantics.
  ValueRange bufferizedResults = /*...*/
  replaceOpWithBufferizedValues(rewriter, this, bufferizedResults);
  return success();
}
```
What is One-Shot Bufferize?

- Analysis of tensor SSA use-def chains
- **BufferizableOpInterface**
- Op interface implementations

One per bufferizable op. Can be implemented by the op directly or be provided as an external model.

```cpp
struct InsertOpInterface
:
public BufferizableOpInterface::ExternalModel<InsertOpInterface, InsertOp>

bool bufferizesToMemoryRead(OpOperand &operand) const {
    return true;
}

bool bufferizesToMemoryWrite(OpOperand &operand) const {
    return true;
}

OpResult getAliasingOpResult(OpOperand &operand) const {
    return op->getOpResult(0);
}

SmallVector<OpOperand *> getAliasingOpOperand(OpResult opResult) const {
    return {&op->getOpOperand(1) /*dest*/};
}
```
What is One-Shot Bufferize?

- Analysis of tensor SSA use-def chains
- **BufferizableOpInterface**
- Op interface implementations
- Lightweight driver that stitches everything together

1. Walk IR in certain order and analyze each tensor op.
2. Rewrite all bufferizable ops with a **RewritePattern**. The rewrite pattern calls `BufferizableOpInterface::bufferize`.  

Overview: BufferizableOpInterface

- bool bufferizesToMemoryRead(OpOperand&)
- bool bufferizesToMemoryWrite(OpOperand&)
- OpResult getAliasingOpResult(OpOperand&)
- SmallVector<OpOperand *> getAliasingOpOperand(OpResult)
- LogicalResult bufferize(Operation*, RewriterBase&, BufferizationState&)

There are a few more...
Definition: Aliasing OpOperand / OpResult

(maybe) aliasing OpOperand / OpResult pair

%r = tensor.insert %f into %t[%c0] : tensor<?xf32>

buffer(%r) == buffer(%t)
or: buffer(%r) is a newly allocated buffer.

In the design document called “tied OpOperand / OpResult pair”.

Definition: Aliasing OpOperand / OpResult

Definition: Aliasing OpOperand / OpResult

\( %r = \text{tensor.insert} \ %f \ \text{into} \ %t[\%c0] : \text{tensor}<\text{xf32}> \)

(maybe) aliasing OpOperand / OpResult pair

buffer(%r) == buffer(%t)
or: buffer(%r) is a newly allocated buffer.

buffer(%r) is either buffer(%t) or a newly allocated buffer. We do not consider other buffers!

⇒ Destination-Passing Style
Definition: Aliasing OpOperand / OpResult

\( \%0 = \text{vector.transfer_write} \ %v, \ %A[\%c0, \ %c0] : \text{vector}<5x6xf32>, \text{tensor}<10x20xf32> \)

(maybe) aliasing OpOperand / OpResult pair

\text{buffer}(\%0) == \text{buffer}(\%A) \\ or: \text{buffer}(\%0) \text{ is a newly allocated buffer.}
Definition: Aliasing OpOperand / OpResult

(maybe) aliasing OpOperand / OpResult pair

\[
\%r = \text{tensor.insert_slice } \%t0 \text{ into } \%t1[5][10][1] : \text{tensor<}\text{xf32}> \text{ into } \text{tensor<}\text{xf32>}
\]

\[
\text{buffer}(%r) == \text{buffer}(%t1)
\]
or: buffer(%r) is a newly allocated buffer.
Definition: Aliasing OpOperand / OpResult

has no (maybe) aliasing OpOperand

```
%r = "tosa.matmul"(%a, %b) : (tensor<?x?xf32>, tensor<?x?xf32>) -> tensor<?x?xf32>
```

buffer(%r) is a newly allocated buffer. Op is not in destination-passing style.

There is no destination ("output") tensor among the OpOperands that could be used for bufferization. Bufferizes same as core bufferization.
Definition: Bufferizes to Read / Write

bufferizes to memory read

\[%r = \text{tensor.insert } \%f \text{ into } \%t[\%c0] : \text{tensor\langle?xf32\rangle}\%

bufferizes to memory write

buffer(\%t) is read and buffer(\%t) is written.

This is a property of the OpOperand, not the SSA Value!
Definition: Bufferizes to Read / Write

bufferizes to memory read

bufferizes to memory read

bufferizes to memory write

buffer(%t) is read
buffer(%t) is read and buffer(%t) is written.
Definition: Bufferizes to Read / Write

bufferizes to memory write

%r = linalg.fill %cst, %t : f32, tensor<?xf32> -> tensor<?xf32>

buffer(%t) is written.

Conceptually, this is identical to:

%r = tensor.generate %sz { ^bb0(%i : index):
    yield %cst : f32
} : tensor<?xf32>

(but this is not in destination-passing style)
Summary: **BufferizableOpInterface**

- `bool bufferizesToMemoryRead(OpOperand& o)`:
  Is the buffer(o) read?
- `bool bufferizesToMemoryWrite(OpOperand& o)`:
  Is the buffer(o) written?
- `OpResult getAliasingOpResult(OpOperand& o)`:
  If o bufferizes in-place: Return OpResult r where buffer(o) may == buffer(r) at runtime.
- `SmallVector<OpOperand *> getAliasingOpOperand(OpResult r)`:
  Return all OpOperands o where if o bufferizes in-place, buffer(o) may == buffer(r) at runtime.
- `LogicalResult bufferize(Operation*, RewriterBase&, BufferizationState&)`:
  Bufferize the op.

There are a few more...
Analysis

For each tensor OpOperand o: Decide whether buffer(o) can be used in the bufferized op directly or a copy of buffer(o) must be used.

```
%1 = tensor.insert %arg1 into %0[%arg2]
{___inplace___} = ["none", "true", "none"] : tensor<*xf32>
```

“true” means “use buffer(%0)”
Analysis

For each tensor OpOperand o: Decide whether buffer(o) can be used in the bufferized op directly or a copy of buffer(o) must be used.

```mlir
// RUN: mlir-opt %s -test-comprehensive-function-bufferize="test-analysis-only"
func @foo(%arg0: index, %arg1: f32, %arg2: index, %arg3: index) -> f32 {
  %0 = tensor.generate %arg0 ... : tensor<?xf32>
  %1 = tensor.insert %arg1 into %0[%arg2]
      {__inplace__ = ["none", "true", "none"]} : tensor<?xf32>
  %2 = tensor.extract %1[%arg3] {__inplace__ = ["true", "none"]} : tensor<?xf32>
  return %2 : f32
}
```
Analysis

For each tensor OpOperand o: Decide whether buffer(o) can be used in the bufferized op directly or a copy of buffer(o) must be used.

// RUN: mlir-opt %s -test-comprehensive-function-bufferize="test-analysis-only"
func @foo(%arg0: index, %arg1: f32, %arg2: index, %arg3: index) -> f32 {
  %0 = tensor.generate %arg0 ... : tensor<?xf32>
  %1 = tensor.insert %arg1 into %0[%arg2]
      {__inplace__ = ["none", "false", "none"]} : tensor<?xf32>
  %2 = tensor.extract %0[%arg3] {__inplace__ = ["true", "none"]} : tensor<?xf32>
  return %2 : f32
}
What if %0 Bufferizes In-place?

For each tensor OpOperand o: Decide whether buffer(o) can be used in the bufferized op directly or a copy of buffer(o) must be used.

```mlir
// RUN: mlir-opt %s -test-comprehensive-function-bufferize="test-analysis-only"
func @foo(%arg0: index, %arg1: f32, %arg2: index, %arg3: index) -> f32 {
  %0 = tensor.generate %arg0 ... : tensor<xf32>
  %1 = tensor.insert %arg1 into %0[%arg2]
      {__inplace__ = ["none", "???", "none"]} : tensor<xf32>
    %2 = tensor.extract %0[%arg3] {__inplace__ = ["true", "none"]} : tensor<xf32>
  return %2 : f32
}
```
Analysis Algorithm Sketch
Do this for every tensor OpOperand.

- Assume that the OpOperand bufferizes in-place.
- Enumerate all “in-place memory write” and “memory read” combinations of the same tensor. Find the “last write” of the read. Check if there’s a conflict.

```plaintext
%0 = tensor.generate %arg0 ... : tensor<?xf32>
%1 = tensor.insert %arg1 into %0[%arg2]
    {__inplace__ = ["none", "???", "none"]} : tensor<?xf32>
%2 = tensor.extract %4[%arg3] {__inplace__ = ["true", "none"]} : tensor<?xf32>
```

Conflict Candidates:
- read = %0
- maybe conflicting write = %0
- last write = %0
What if %3 Bufferizes In-place?

We don't know what %3 is. If buffer(%3) == buffer(%0), there would be conflict!

%0 = tensor.generate %arg0 ... : tensor<?xf32>
...
%1 = tensor.insert %arg1 into %3[%arg2]
    {__inplace__ = ["none", "???", "none"]} : tensor<?xf32>
%2 = tensor.extract [%arg3] {__inplace__ = ["true", "none"]} : tensor<?xf32>
What if %3 Bufferizes In-place?

We don’t know what %3 is. If buffer(%3) == buffer(%0), there would be conflict!

Solution: Analysis maintains alias sets. Take into account all reads/write of an entire alias set.
If %3 bufferizes in-place:          {{%0}, {%1, %3}
If %3 bufferizes out-of-place:      {{%0}, {%1}, {%3}}
%3 Bufferizes In-place!

Solution: Analysis maintains alias sets. Take into account all reads/write of an entire alias set.

If %3 bufferizes in-place: {(%0), (%1, %3)}
Where is the Heuristic?

- The order in which OpOperands are analyzed affects the order in which conflicts are found.
- There could be multiple out-of-place bufferization candidates to avoid a conflict. Once a conflict becomes apparent, the OpOperand that is currently analyzed is chosen to bufferize out-of-place.
- Examples for possible heuristics:
  - Analyze ops in a FuncOp top-to-bottom.
  - Analyze ops in a FuncOp bottom-to-top.
  - First analyze all InsertSliceOps in a FuncOp, then the remaining ops top-to-bottom.
Extensibility: Conflict Detection

- Ops may not be reading/writing the entire OpOperand.
- E.g.: `tensor.insert_slice` does not read the overwritten part of `dest`.
- Yet, `bufferizesToMemoryRead/Write` is just a boolean (yes/no).
- Ops can specify “read” / “conflicting write” pairs that are not a conflict:

  ```cpp
  BufferizableOpInterface::isNotConflicting(
    OpOperand *uRead, OpOperand *uConflictingWrite)
  ```
Unifying One-Shot Bufferization and Core Bufferization
Why Unify the Bufferizations?

- **Less confusing** for users. There’s only one bufferize to choose.
- Users of core bufferization can benefit from **better bufferizations** and have a clear path for **gradually migrating to one-shot bufferization**.
- **Code cleanup**: No fundamental reason for having two bufferizations.
Compatibility

- One-Shot Bufferize and Core Bufferization are compatible. They use the same contract at the bufferization boundary (to_memref/to_tensor).
- They can be used together, but **One-Shot Bufferize must run first**.

// RUN: mlir-opt %s | \
  -test-comprehensive-function-bufferize= \ 
    "allow-return-memref allow-unknown-ops create-deallocs=0" | \ 
  -bufferize-my-own-dialect -bufferize-my-other-dialect | \ 
  -finalizing-bufferize -buffer-deallocation
Compatibility

- One-Shot Bufferize and Core Bufferization are compatible. They use the same contract at the bufferization boundary (`to_memref/to_tensor`).
- They can be used together, but One-Shot Bufferize must run first.

```// RUN: mlir-opt %s | \
   -test-comprehensive-function-bufferize=allow-return-memref allow-unknown-ops create-deallocs=0 | 
   -bufferize-my-own-dialect -bufferize-my-other-dialect | 
   -finalizing-bufferize -buffer-deallocation
```

`to_memref/to_tensor` are internal ops and special variants of `unrealized_conversion_cast`. They...
- should not leak across pass boundaries,
- are not compatible with the analysis (e.g., the result of `to_tensor` can alias with anything),
- are only used to connect the two bufferizations,
- never appear in a fully bufferized program.
Compatibility

- One-Shot Bufferize and Core Bufferization are compatible. They use the same contract at the bufferization boundary (to_memref/to_tensor).
- They can be used together, but **One-Shot Bufferize must run first**.

```bash
// RUN: mlir-opt %s | \\
  -test-comprehensive-function-bufferize= \\
  "allow-return-memref allow-unknown-ops create-deallocs=0 \\
  dialect-filter='tensor,vector,scf'" | \\
  -bufferize-my-own-dialect -bufferize-my-other-dialect | \\
  -finalizing-bufferize -buffer-deallocation
```
Outline of Steps

- Move Comprehensive Bufferize (One-Shot Bufferize) to the bufferization dialect and rename it to just *bufferization*.
- **Switch impl. of core bufferization passes** to BufferizableOpInterface. NFC from a user’s perspective. A single rewrite pattern that calls bufferize without an analysis. For ops that are not supported in Comprehensive Bufferize: Move existing implementation into op interface.
- **Gradually update existing users** of partial bufferization to One-Shot Bufferization. This is optional. But users will get better bufferization results if they do make the switch.
- **Delete all partial bufferization passes** once they have no users anymore. We probably want to keep them around for unit tests. (As test passes.) This is a longer-term goal.
Switch Core Bufferization Passes

This is the new implementation of -tensor-bufferize.

```cpp
struct TensorBufferizePass : public TensorBufferizeBase<TensorBufferizePass> {
    void runOnFunction() override {
        auto options = std::make_unique<BufferizationOptions>();
        options->allowReturnMemref = true;
        options->allowUnknownOps = true;
        options->createDeallocs = false;
        options->addToDialectFilter<tensor::TensorDialect>();
        AlwaysCopyBufferizationState state(options);
        return bufferizeOp(getFunction(), state);
    }
};
```

just bufferize, no analysis
Questions / Discussion
Related Docs / Discourse Posts

- https://llvm.discourse.group/t/open-mlir-meeting-1-13-2021-one-shot-function-bufferization-of-tensor-programs/5197/2
- https://llvm.discourse.group/t/rfc-dialect-for-bufferization-related-ops/4712
## Comparison of Bufferizations

<table>
<thead>
<tr>
<th>Core Bufferization</th>
<th>One-Shot Bufferization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiple passes (one per dialect)</td>
<td>Single pass, whole function bufferization</td>
</tr>
<tr>
<td>Partial bufferization possible (via to_tensor / to_memref ops)</td>
<td>Unknown ops are wrapped in to_tensor/to_memref ops. Those cannot be bufferized any further. ⇒ Cannot run one-shot bufferization after partial bufferization!</td>
</tr>
<tr>
<td>DialectConversion patterns</td>
<td>Op interface + external model impl. + analysis + RewritePatterns</td>
</tr>
<tr>
<td>Conservatively insert buffer copy on every memory write. Remove copies in a separate pass after a memref-based analysis (not currently implemented).</td>
<td>Perform tensor-based analysis first, insert copies only when deemed necessary.</td>
</tr>
<tr>
<td>Buffer deallocation via BufferDeallocationPass</td>
<td>Buffer deallocation automated for allocs that do not escape block boundaries. Otherwise, use BufferDeallocationPass.</td>
</tr>
</tbody>
</table>
Current State of Bufferization

MLIR Transforms
- BufferHoistingPass
- BufferLoopHoistingPass
- PromoteBuffersToStackPass
- BufferResultsToOutParamsPass

Similar functionality

ComprehensiveBufferize
- ComprehensiveModuleBufferizePass
- (ComprehensiveFunctionBufferizePass)
- BufferizableOpInterface
  + external model implementations

Tensor dialect
- TensorBufferizePass

SCF dialect
- SCFBufferizePass

Standard dialect
- FuncBufferizePass

Bufferization dialect
- FinalizingBufferizePass
- BufferDeallocationPass

Directives
- tensor → memref DialectConversion patterns
Example: tensor.insert (1 / 2)

%r = tensor.insert %f into %dest[%pos] : f32 into tensor<?xf32>

struct InsertOpInterface : public BufferizableOpInterface::ExternalModel<InsertOpInterface, tensor::InsertOp> {
  bool bufferizesToMemoryRead(Operation *op, OpOperand &opOperand) {
    return true;
  }

  bool bufferizesToMemoryWrite(Operation *op, OpOperand &opOperand) {
    return true;
  }

  OpResult getAliasingOpResult(Operation *op, OpOperand &opOperand /*dest*/) {
    return op->getOpResult(0);
  }

  LogicalResult bufferize(Operation *op, RewriterBase &rewriter, const BufferizationState &state);
};

Interface methods are called only for tensor-typed OpOperands / OpResults.
LogicalResult InsertOpInterface::bufferize(Operation *op, RewriterBase &rewriter,
const BufferizationState &state) {
  auto insertOp = cast<tensor::InsertOp>(op);

  Value destMemref = *state.getBuffer(rewriter, insertOp->getOpOperand(1) /*dest*/);

  rewriter.create<memref::StoreOp>(insertOp.getLoc(), insertOp.scalar(),
                                  destMemref, insertOp.indices());

  replaceOpWithBufferizedValues(rewriter, insertOp, destMemref);

  return success();
}
Definition: Aliasing OpOperand / OpResult

OpResult is (maybe) aliasing one (or both) of the the two tensor OpOperands.

\[
%r = \text{std.select } %c, \; %t1, \; t2 : \text{tensor<?xf32>}
\]

- \(\text{buffer}(%r) = \text{buffer}(%t1)\)
- \(\text{or: } \text{buffer}(%r) = \text{buffer}(%t2)\)
- \(\text{or: } \text{buffer}(%r)\) is a newly allocated buffer