

# SIG OpenXLA Community Meeting

---

October 18, 2022

# What is OpenXLA?

**Open, state-of-art ML compiler ecosystem, built collaboratively with Hardware & Software partners, using the best of XLA & MLIR.**

# Introductions

---

# Welcome!

- Welcome to any new attendees? What are you looking to focus on?
- SIG member organizations:
  - Alibaba
  - AMD
  - Apple
  - ARM
  - AWS
  - Google
  - Intel
  - Meta
  - NVIDIA

# SIG Collaboration

---

Reference material for our new collaborators

# Our Meetings

- Monthly on Zoom, 3rd Tuesday @ 8AM PT
- Rotating meeting host & scribe
- Proposed agenda shared by host week prior in [GitHub Discussions](#)
- Meeting minutes & slides shared publicly in the [meetings archive](#) the day after
- Meetings should include:
  - Development updates
  - Design proposals
  - Community topics

# Our Collaboration Channels

Channel	Content	Access	Archive
<a href="#">GitHub organization</a>	Code, Design proposals, PRs, Issues, Roadmaps	Public	N/A
<a href="#">Community repository</a>	Governance, Meetings, Code of conduct	Public	Public
<a href="#">Community discussions</a>	Meta discussions on openxla/community repo	Public	Public
<a href="#">Technical discussions</a>	Technical discussions on individual repos: xla, stablehlo	Public	Public
<a href="#">Discord</a>	Sync discussions	Open invites	Archived chats
<a href="#">Community meetings</a>	Monthly live meetings	Public	Public agenda, slides, meeting minutes
<i>SIG Google Drive</i>	<i>Shared docs, decks</i>	<i>Read-only to non members</i>	<i>Indefinite</i>

# Development Updates

---



# StableHLO Compatibility

---

Kevin Gleason, Google

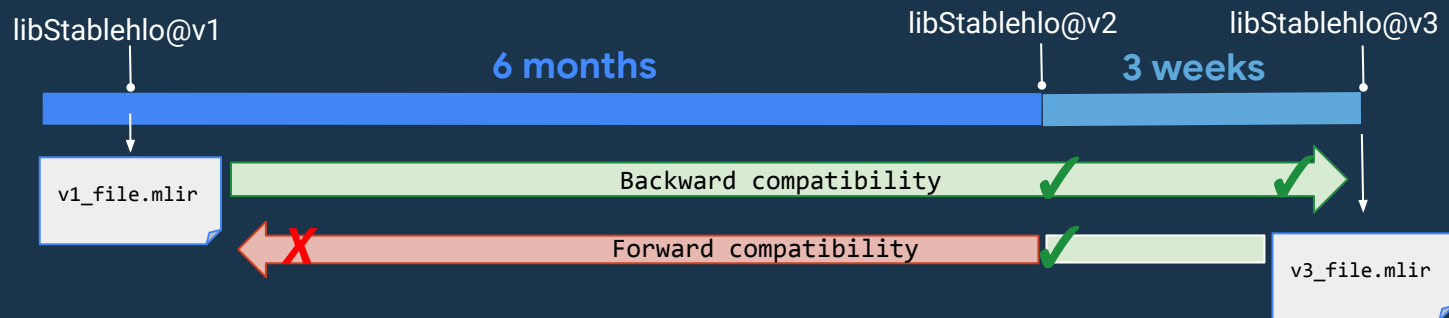
[gleasonk@google.com](mailto:gleasonk@google.com)

# Agenda

- Compatibility Requirements
- Upgrade and Downgrade Hooks
- Compatibility Prototype
- Next Steps

# Requirements *(feedback wanted!)*

R1	libStablehlo provides 6 months of backward compatibility
R2	libStablehlo provides 3 weeks of forward compatibility
R3	Source compatibility for C, C++ and Python APIs within libStablehlo is an aspirational goal.



# Supported Dialects and Operations, Attributes, Types

Dialect	Supported Ops, Attributes and Types
Arith	AddOp, CmpOp, CmpIPredicateAttr, ConstantOp, DivSIOp, ExtSIOp, IndexCastOp, MaxSIOp, MinSIOp, MulOp, SelectOp, SubOp, TruncOp
Builtin	No ops, but all attributes and types.
CHLO	All ops, attributes and types.
Func	CallOp
MLProgram	All ops, attributes and types.
SparseTensor	Aspirational (pending the sparsity RFC which is expected in Q4 2022).
StableHLO	All ops, attributes and types except CustomCallOp whose semantics is implementation-defined.
Tensor	CastOp, DimOp, FromElementsOp

# IR Upgrades

- Provides backward compatibility
- For 6 months, support dialect upgrades
- Payloads have version number, allowing tooling to error if out of compatibility window

Rename `stablehlo.sub` → `stablehlo.subtract`

v39

```
func.func @upgrade(%arg0: tensor<2xi1>) -> () {  
  %0 = stablehlo.sub %arg0, %arg0 : tensor<2xi1>  
  return  
}
```

upgrade

v40

```
func.func @upgrade(%arg0: tensor<2xi1>) -> () {  
  %0 = stablehlo.subtract %arg0, %arg0 : tensor<2xi1>  
  return  
}
```

# IR Downgrades

- Provides forward compatibility when possible, else warn / error
- For 3 weeks after dialect changes, target the previous version
- Can be processed in v39, and can be upgraded by v40

Rename `stablehlo.sub` → `stablehlo.subtract`

v40

```
func.func @upgrade(%arg0: tensor<2xi1>) -> () {  
  %0 = stablehlo.subtract %arg0, %arg0 : tensor<2xi1>  
  return  
}
```

downgrade

v39

```
func.func @upgrade(%arg0: tensor<2xi1>) -> () {  
  %0 = stablehlo.sub %arg0, %arg0 : tensor<2xi1>  
  return  
}
```

# Compatibility Prototype Tool / API

## Compatibility Tool

```
stablehlo-translate --compat [flags] file.mlir
  --target=<version>    Target version to generate (Default to minimum supported)
  --emit-asm            Output textual assembly (Default generate bytecode)
```

## Compatibility APIs

```
// namespace mlir::stablehlo
OwningOpRef<Operation *> parseWithCompat(llvm::SourceMgr &sourceMgr,
                                          MLIRContext *context);

LogicalResult writeWithCompat(Operation *topLevelOperation,
                              int64_t targetVersion,
                              bool emitAssembly,
                              llvm::raw_ostream &output);
```

# Compatibility Prototype Example

```
void registerSubOpChanges() {  
    // Change log:  
    //   Version 39: SubOp<"stablehlo.sub"> exists  
    //   Version 40: SubOp<"stablehlo.sub"> -> SubtractOp<"stablehlo.subtract">  
    // Backward compatibility: Support v39 and after.  
    // Forward compatibility: Target v39 for printing.  
  
    // Upgrade <v40 -> 40: [sub --> subtract]  
    addUpgrade("stablehlo.sub", 40,  
        [&](Operation *op, int64_t fromVer) -> LogicalResult {  
            renameOperation(op, "stablehlo.subtract");  
            return success();  
        });  
  
    // Downgrade v40 -> 39: [subtract --> sub]  
    addDowngrade("stablehlo.subtract", 39,  
        [&](Operation *op, int64_t fromVer) -> LogicalResult {  
            renameOperation(op, "stablehlo.sub");  
            return success();  
        });  
}
```



# Compatibility Prototype Example

```
gleasonk@gleasonk-linux:~/compat$ cat sub.mlir
func.func private @test_sub(%arg0: tensor<2xi1>) -> () attributes {stablehlo.compat_version = 37 : i32} {
  %0 = "stablehlo.sub"(%arg0, %arg0) : (tensor<2xi1>, tensor<2xi1>) -> tensor<2xi1>
  func.return
}

gleasonk@gleasonk-linux:~/compat$ stablehlo-translate --compat --target=40 sub.mlir
func.func private @test_sub(%arg0: tensor<2xi1>) attributes {stablehlo.compat_version = 40 : i64} {
  %0 = stablehlo.subtract %arg0, %arg0 : tensor<2xi1>
  return
}

gleasonk@gleasonk-linux:~/compat$ stablehlo-translate --compat --target=38 sub.mlir
"func.func"() ({
^bb0(%arg0: tensor<2xi1>):
  %0 = "stablehlo.sub"(%arg0, %arg0) : (tensor<2xi1>, tensor<2xi1>) -> tensor<2xi1>
  "func.return"() : () -> ()
}) {function_type = (tensor<2xi1>) -> (), stablehlo.compat_version = 38 : i64, sym_name = "test_sub", sym_
) -> ()
```

# Next Steps

- In Q4 2022:
  - Propose an MLIR RFC about stability of the bytecode format.
  - Start a discussion about stability guarantees for the Builtin dialect.
  - Finish the compatibility tool, hold a design review for the StableHLO Compatibility RFC
- We aim to ship a working solution by the end of the year.

# Q&A

---

# GmlSt : Tiling and Fusion for XLA

Frederik Gossen, Google

[frgossen@google.com](mailto:frgossen@google.com)

# Tiling and fusion

**Goal.** Break operation down to tiles to ...

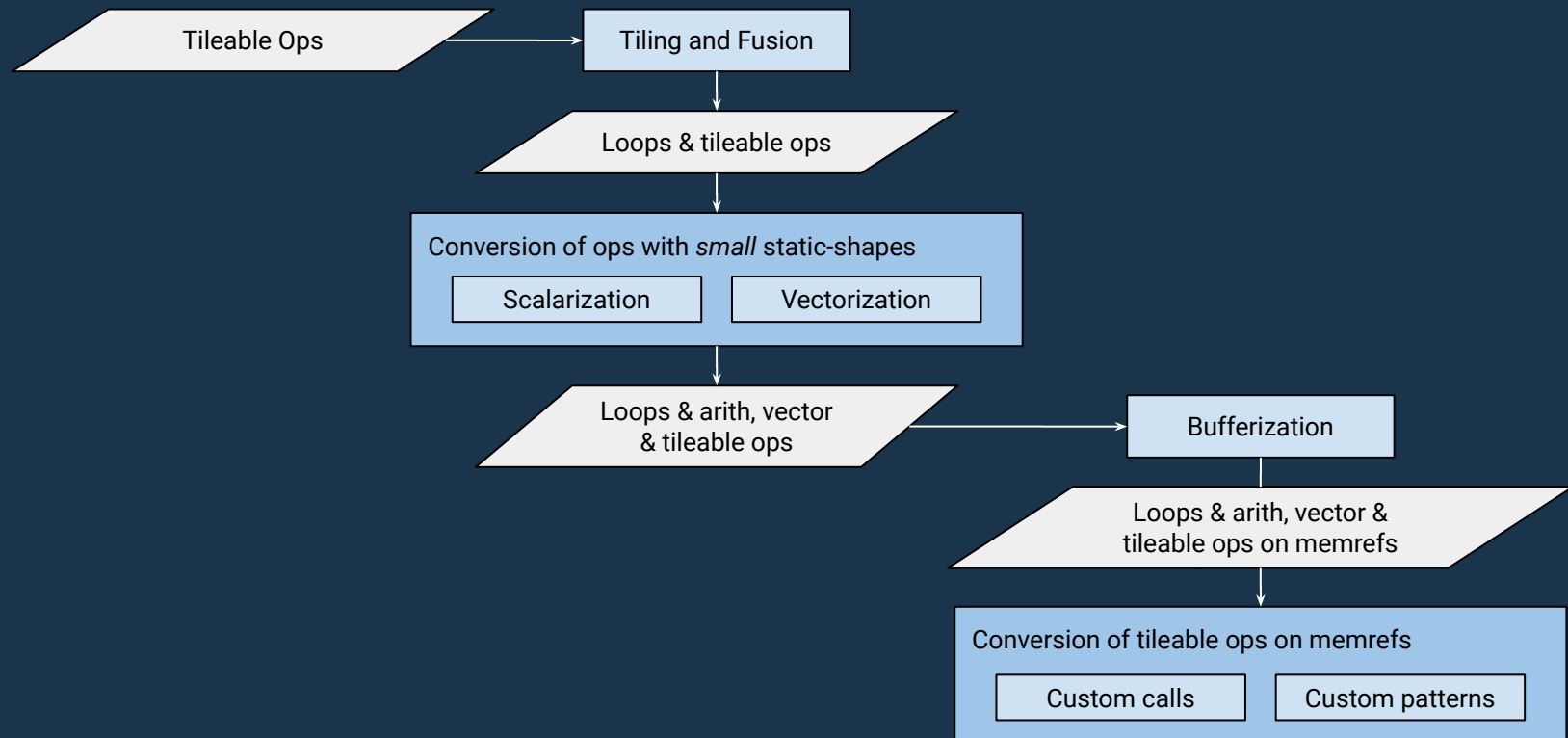
- ... make better use of the cache
- ... map nicely to device-specific concepts,  
e.g. blocks/warps (GPU), vector instr. (CPU)

**Tiling.** Break an operation down to subsets/tiles and process them separately.

**Fusion.** Pull producers of inputs into a tiled implementation.

**XLA.** Moving to tile-based fusion and code emission,  
instead of point-based.

# Pipeline Overview



# GmlSt Design Goals

- **Nested tiling.** Tile to multiple levels, e.g. blocks, warps, threads.
- **Dynamism.** Tile & fuse dynamic ops, e.g. bcast, concat, gather, and scatter.
- **Compose** with existing ops and tiling, e.g. fuse into existing loops.
- **Subsets.** Be ready for subset types beyond tiles.

# GmlSt Characteristics

- Works on destination-style IR

```
%init = tensor.empty [%d0, %d1] : tensor<?x?xf32>  
%ab = linalg.map "add" ins(%a, %b) outs(%init)
```

- Separates subset computation and subset materialization

```
%t0 = gml_st.tile [%i, %j] [32, 8] [1, 1] : !gml_st.tile<32x8>  
%lhs_sub = gml_st.materialize %lhs[%t0] : tensor<32x8xf32>
```

- Pack subsets and pass them as one value
  - More readable IR (subjective), flexible wrt. future subset types



# Examples

---

# Examples

- Tiling and fusing element-wise ops
- Concat
- Reduction
- Nested tiling

# Tile & Fuse Element-wise

---

# Tiling and Fusing Element-wise Ops I

```
func.func @cwise(%a: tensor<?x?xf32>, %b: tensor<?x?xf32>, %c: tensor<?x?xf32>)  
  -> tensor<?x?xf32> {  
  
    %ab = mhlo.add %a, %b : tensor<?x?xf32>  
    %abc = mhlo.multiply %ab, %c : tensor<?x?xf32>  
    func.return %abc  
  }
```



# Tiling and Fusing Element-wise Ops II

```
func.func @cwise(%a: tensor<?x?xf32>, %b: tensor<?x?xf32>, %c: tensor<?x?xf32>)  
  -> tensor<?x?xf32> {  
    %init = tensor.empty [%d0, %d1] : tensor<?x?xf32>  
    %ab = linalg.map "add" ins(%a, %b) outs(%init)  
    %abc = linalg.map "mul" ins(%ab, %c) outs(%init)  
    func.return %abc  
  }
```



# Tiling and Fusing Element-wise Ops III

```
func.func @cwise(%a: tensor<?x?xf32>, %b: tensor<?x?xf32>, %c: tensor<?x?xf32>)
  -> tensor<?x?xf32> {
    %init = tensor.empty [%d0, %d1] : tensor<?x?xf32>
    %ab = linalg.map "add" ins(%a, %b) outs(%init)
    %abc = gml_st.parallel (%i, %j) = (%c0, %c0) to (%d0, %d1) step (%c4, %c8) {
      %tile = gml_st.tile [%i, %j] [%td0, %td1] [1, 1] : !gml_st.tile<?x?>

      %ab_sub = gml_st.materialize %ab[%tile] : tensor<?x?xf32>
      %c_sub = gml_st.materialize %c[%tile] : tensor<?x?xf32>
      %init_sub = gml_st.materialize %init[%tile] : tensor<?x?xf32>

      %abc_sub = linalg.map "mul" ins(%ab_sub, %c_sub) outs(%init_sub)
      gml_st.set_yield %abc_sub into %init[%tile]
    } : tensor<?x?xf32>
    func.return %abc
  }
```



# Tiling and Fusing Element-wise Ops IV

```
func.func @cwise(%a: tensor<?x?xf32>, %b: tensor<?x?xf32>, %c: tensor<?x?xf32>)
  -> tensor<?x?xf32> {
    %init = tensor.empty [%d0, %d1] : tensor<?x?xf32>

    %abc = gml_st.parallel (%i, %j) = (%c0, %c0) to (%d0, %d1) step (%c4, %c8) {
      %tile = gml_st.tile [%i, %j] [%td0, %td1] [1, 1] !gml_st.tile<?x?>
      %a_sub = gml_st.materialize %a[%tile] : tensor<?x?xf32>
      %b_sub = gml_st.materialize %b[%tile] : tensor<?x?xf32>
      %c_sub = gml_st.materialize %c[%tile] : tensor<?x?xf32>
      %init_sub = gml_st.materialize %init[%tile] : tensor<?x?xf32>
      %ab_sub = linalg.map "add" ins(%a_sub, %b_sub) outs(%init_sub)
      %abc_sub = linalg.map "mul" ins(%ab_sub, %c_sub) outs(%init_sub)
      gml_st.set_yield %abc_sub into %init[%tile]
    } : tensor<?x?xf32>
    func.return %abc
  }
```



# Tile Concat

---



# Materialize Ops are an Entry Point to Fusion

- Allows to fuse w/o the presence of a specific loop
  - Concat can be tiled to a switch statement
  - Allows for different loop kinds,  
e.g. parallel and sequential



# Concat I

```
func.func @concat(%a : tensor<?x?xi32>, %b : tensor<?x?xi32>) -> tensor<?x?xi32> {  
  %init_ab = tensor.empty [%d0, %d1] : tensor<?x?xi32>  
  %ab = linalg.map "add" ins(%a, %b) outs(%init_ab)  
  %init_ccat = tensor.empty [%d0, %dcat] : tensor<?x?xi32>  
  
  %ccat = thlo.concatenate ins(%ab, %a) outs(%init_ccat : tensor<?x?xi32>) { dimension = 1 }  
  
  func.return %concat  
}
```



# Concat II

```
func.func @concat(%a : tensor<?x?xi32>, %b : tensor<?x?xi32>) -> tensor<?x?xi32> {  
  %init_ab = tensor.empty [%d0, %d1] : tensor<?x?xi32>  
  %ab = linalg.map "add" ins(%a, %b) outs(%init_ab)  
  %init_ccat = tensor.empty [%d0, %dcat] : tensor<?x?xi32>  
  
  %ccat = gml_st.parallel (%i, %j) = (%c0, %c0) to (%d0, %dcat) step (%c4, %c1) {  
  
    %ccat_sub = scf.if %subset_in_a {  
      %tile = gml_st.tile [%i, %j] [%td0, 1] [1, 1]  
      %a_sub = gml_st.materialize %ab[%tile] : tensor<?x?xf32>  
      scf.yield %a_sub  
    } else {  
  
      ...  
    }  
  }  
}
```



# Concat II

```
func func @concat(%a : tensor<?x?xi32>, %b : tensor<?x?xi32>) -> tensor<?x?xi32> {  
  %init_ab = tensor.empty [%d0, %d1] : tensor<?x?xi32>  
  %ab = linalg.map "add" ins(%a, %b) outs(%init_ab)  
  %ccat = tensor.empty [%d0, %dcat] : tensor<?x?xi32>  
  
  %ccat = gml_st.parallel (%i, %j) = (%c0, %c0) to (%d0, %dcat) step (%c4, %c1) {  
    %ccat_sub = scf.if %c1.get_in_a {  
      %tile = gml_st.tile [%i, %j] [%d0, 1] [1, 1]  
      %a_sub = gml_st.materialize [%ab] [%tile] : tensor<?x?xf32>  
      scf.yield %a_sub  
    } else {  
      ...  
    }  
  }  
}
```



# Tile Reductions

---

# Tile Reductions

Reductions can be tiled to

- ``gml_st.parallel`` with an accumulator or to
- ``gml_st.for`` as a sequential implementation.



# Reduction I

```
func.func @reduce(%arg: tensor<32x16xf32>) -> tensor<32xf32> {  
  %init = tensor.empty [32] : tensor<32xf32>  
  %fill = linalg.fill ins(%c0) outs(%init) : tensor<32xf32>  
  %sum = linalg.reduce "sum" reduction_dim = 1 ins(%arg) outs(%fill)  
  func.return %sum  
}
```



# Reduction II

```
func.func @reduce(%arg: tensor<32x16xf32>) -> tensor<32xf32> {  
  %init = tensor.empty [32] : tensor<32xf32>  
  %fill = linalg.fill ins(%c0) outs(%init) : tensor<32xf32>  
  %sum = gml_st.parallel (%i, %j) = (%c0, %c0) to (%c32, %c16) step (%c4, %c8) {  
    %t0 = gml_st.tile [%i, %j] [%t0d0, %t0d1] [1, 1] : !gml_st.tile<?x?>  
    %arg_sub = gml_st.materialize %arg[%t0] : !gml_st.tile<4x8>  
    %t1 = gml_st.tile [%i] [%t1d0] [1] : !gml_st.tile<4>  
    %fill_sub = gml_st.materialize %fill[%t1] : tensor<4xf32>  
    %sum_sub = linalg.reduce "sum" reduction_dim = 1 ins(%arg_sub) outs(%fill_sub)  
    gml_st.set_yield %sum_sub into %fill[%t1] acc (%a, %b: tensor<4xf32>) {  
      %ab = linalg.map "add" ins(%a) outs(%b)  
      gml_st.yield %ab  
    }  
  } : tensor<32xf32>  
  func.return %sum  
}
```





# Nested Tiling

---

# Nested Tiling

Repeated tiling of one operation



# Nested Tiling I

```
func.func @ewise(%a: tensor<?x?xf32>, %b: tensor<?x?xf32>) -> tensor<?x?xf32> {  
  %init = tensor.empty [%d0, %d1] : tensor<?x?xf32>  
  %ab = linalg.map "add" ins(%a, %b) outs(%init)  
  func.return %ab  
}
```



# Nested Tiling II

```
func.func @ewise(%a: tensor<?x?xf32>, %b: tensor<?x?xf32>) -> tensor<?x?xf32> {  
  %init = tensor.empty [%d0, %d1] : tensor<?x?xf32>  
  %ab = gml_st.parallel (%i, %j) = (%c0, %c0) to (%d0, %d1) step (%c128, %c512) {  
    %t0 = gml_st.tile [%i, %j] [%t0d0, %t0d1] [1, 1] : !gml_st.tile<?x?>  
    %a_sub = gml_st.materialize %a[%t0] : tensor<?x?xf32>  
    %b_sub = gml_st.materialize %b[%t0] : tensor<?x?xf32>  
    %init_sub = gml_st.materialize %init[%t0] : tensor<?x?xf32>  
    %ab_sub = linalg.map "add" ins(%a_sub, %b_sub) outs(%init_sub)  
    gml_st.set_yield %ab_sub into %init[%t0]  
  } : tensor<?x?xf32>  
  func.return %ab  
}
```



# Nested Tiling III

```
func.func @ewise(%a: tensor<?x?xf32>, %b: tensor<?x?xf32>) -> tensor<?x?xf32> {
  %init = tensor.empty [%d0, %d1] : tensor<?x?xf32>
  %ab = gml_st.parallel (%i, %j) = (%c0, %c0) to (%d0, %d1) step (%c128, %c512) {
    %t0 = gml_st.tile [%i, %j] [%t0d0, %t0d1] [1, 1] : !gml_st.tile<?x?>
    %a_sub = gml_st.materialize %a[%t0] : tensor<?x?xf32>
    %b_sub = gml_st.materialize %b[%t0] : tensor<?x?xf32>
    %init_sub = gml_st.materialize %init[%t0] : tensor<?x?xf32>
    %ab_sub = gml_st.parallel (%k, %l) = (%c0, %c0) to (%t0d0, %t0d1) step (%c4, %c8) {
      %t1 = gml_st.tile [%k, %l] [%t1d0, %t1d1] [1, 1] : !gml_st.tile<?x?>
      %a_sub_sub = gml_st.materialize %a_sub[%t1] : tensor<?x?xf32>
      %b_sub_sub = gml_st.materialize %b_sub[%t1] : tensor<?x?xf32>
      %init_sub_sub = gml_st.materialize %init_sub[%t1] : tensor<?x?xf32>
      %ab_sub_sub = linalg.map "add" ins(%a_sub_sub, %b_sub_sub) outs(%init_sub_sub)
      gml_st.set_yield %ab_sub_sub into %init_sub[%t1]
    } : tensor<?x?xf32>
    gml_st.set_yield %ab_sub into %init[%t0]
  } : tensor<?x?xf32>
  func.return %ab
}
```



# Nested Tiling IV

```
func.func @ewise(%a: tensor<?x?xf32>, %b: tensor<?x?xf32>) -> tensor<?x?xf32> {  
  %init = tensor.empty [%d0, %d1] : tensor<?x?xf32>  
  %ab = gml_st.parallel (%i, %j) = (%c0, %c0) to (%d0, %d1) step (%c128, %c512) {  
    %t0 = gml_st.tile [%i, %j] [%t0d0, %t0d1] [1, 1] : !gml_st.tile<?x?>  
  
    %init_sub = gml_st.materialize %init[%t0] : tensor<?x?xf32>  
    %ab_sub = gml_st.parallel (%k, %l) = (%c0, %c0) to (%t0d0, %t0d1) step (%c4, %c8) {  
      %t1 = gml_st.tile [%i + %k, %j + %l] [%t1d0, %t1d1] [1, 1] : !gml_st.tile<?x?>  
      %a_sub_sub = gml_st.materialize %a[%t1] : tensor<?x?xf32>  
      %b_sub_sub = gml_st.materialize %b[%t1] : tensor<?x?xf32>  
      %init_sub_sub = gml_st.materialize %init_sub[%t1] : tensor<?x?xf32>  
      %ab_sub_sub = linalg.map "add" ins(%a_sub_sub, %b_sub_sub) outs(%init_sub_sub)  
      gml_st.set_yield %ab_sub_sub into %init_sub[%t1]  
    } : tensor<?x?xf32>  
    gml_st.set_yield %ab_sub into %init[%t0]  
  } : tensor<?x?xf32>  
  func.return %ab  
}
```



# Q&A

---

# XLA Runtime + CUDA Graphs

---

RFC: Cuda Graphs support in XLA



# XLA Runtime Supports Multiple Exported Functions

- XLA today always exports a single entry point function from the HLO module
- XLA runtime modules do not have this limitation, and can export multiple functions
- Exported functions are callable from the C++ API: `Executable::Execute(int ordinal, Arguments arg)`

```
rt.export @add ordinal 0
rt.export @mul ordinal 1

func @add(%arg0: tensor<?xf32>, %arg1: tensor<?xf32>)
  -> tensor<?xf32> {
    %0 = mhlo.add %arg0, %arg1 : tensor<?xf32>
    return %0 : tensor<?f32>
  }

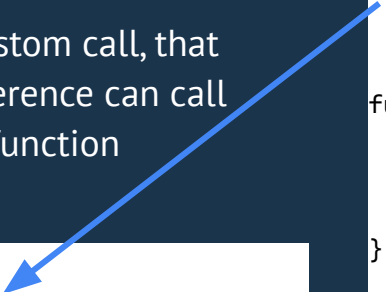
func @mul(%arg0: tensor<?xf32>, %arg1: tensor<?xf32>)
  -> tensor<?xf32> {
    %0 = mhlo.mul %arg0, %arg1 : tensor<?xf32>
    return %0 : tensor<?f32>
  }
```



# XLA Custom Calls Support Function References

- XLA runtime can pass a reference to an exported function to custom calls
- Executable calls into a custom call, that itself using a function reference can call back into the executable function

```
CustomCall::Bind("call_ref")  
  .Arg<Memref>()  
  .Arg<Memref>()  
  .Attr<FunctionRef>("func")  
  .To([](Memref a, Memref b, FunctionRef func) {  
    return func(a, b);  
  });
```



```
rt.export @add ordinal 0  
rt.export @mul ordinal 1  
  
func @call_ref(%arg0: tensor<?xf32>,  
              %arg1: tensor<?xf32>)  
  -> tensor<?xf32> attributes { rt.custom_call = "call_ref" }  
  
func @impl(%arg0: tensor<?xf32>, %arg1: tensor<?xf32>)  
  -> tensor<?xf32> {  
    %0 = mhlo.add %arg0, %arg1 : tensor<?xf32>  
    return %0 : tensor<?f32>  
  }  
  
func @main(%arg0: tensor<?xf32>, %arg1: tensor<?xf32>)  
  -> tensor<?xf32> {  
    %0 = call @call_ref(%arg0, %arg1) { func = @impl }  
      : (tensor<?xf32>, tensor<?xf32>) -> tensor<?xf32>  
    return %0 : tensor<?f32>  
  }
```



# XLA Exports Graph Capture Functions

- XLA runtime implements a custom call that takes a reference to “graph capture” function
- Before calling graph capture function runtime sets up CUDA graph building “context”.
- Instead of a graph capture XLA can provide a set of custom calls that build graph using explicit API

```
CustomCall::Bind("xla.graph.launch")
  .Arg<RemainingArgs>()
  .Attr<FunctionRef>("capture")
  .To([](RemainingArgs args, FunctionRef func) {
    if (pointers_changed(args)) {
      cudaStreamBeginCapture();
      Status captured = func(a, b);
      cudaStreamEndCapture();
      UpdateGraph();
    }
    return ExecuteCachedGraph(args, func);
  });
```

```
rt.export @main ordinal 0
rt.export @graph.capture ordinal 1

// This function does not execute "add". Once it's lowered
// to runtime HAL it adds an "add" command to the "command
// buffer" (cuda graph).
func @graph.capture(%arg0: tensor<?xf32>,
                   %arg1: tensor<?xf32>) -> ... {
  %0 = mhlo.add %arg0, %arg1 : tensor<?xf32>
  return %0 : tensor<?f32>
}

func @main(%arg0: tensor<?xf32>,
          %arg1: tensor<?xf32>) -> ... {
  // Xla runtime is responsible for capturing graphs,
  // caching them, and submitting to the compute stream.
  %0 = call @xla.graph.launch(%arg0, %arg1)
    { capture = @graph.capture }
    : (tensor<?xf32>, tensor<?xf32>) -> tensor<?xf32>
  return %0 : tensor<?f32>
}
```



# Q&A

---

# Let's continue to discuss on GitHub!

[github.com/openxla/xla/discussions](https://github.com/openxla/xla/discussions)