Torch-MLIR

Sean Silva (Google/IREE) & Anush Elangovan (nod.ai)

IREE Team At Google

- Started the Torch-MLIR (nee npcomp) team because of the pressing need to have inbound connections from PyTorch –> MLIR broadly.
- Dedicated to keeping this effort independent from a project perspective (i.e. IREE needs a good connection to PyTorch, but so does everyone)
- Torch-MLIR Team members:
  - Sean Silva
  - Yi Zhang
  - Stella Laurenzo (emeritus)
- Complements other frontend integrations (TFLite, TOSA, TF, etc)
- Most investment right now going to Torch-MLIR and TFLite/TOSA

nod.ai

- **What we do:** Turnkey A.I Silicon Enablement, complementing your internal teams
- **How we do it:** MLIR + Custom Codegen / Auto-scheduling for your silicon
- Been around for 8+yrs with last **4+ focused on A.I performance**
- If you build A.I Silicon talk to us
    - We work with a lot of you on this call from edge to large clusters. Thank you for your support.
- If working at the intersection of ML, CS, Math and HW excites you talk to us stdin@nod.com

- Anush Elangovan - Founder / CEO
    - Built Chromebooks at Google previously, @Agnilux, Network Security @FireEye, DC Networking @ Cisco
    - anush@nod.com
- Harsh Menon - CTO
    - Built flying cars at KittyHawk
    - harsh@nod.com

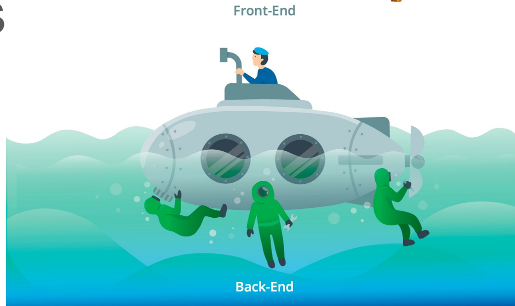
- Nod.ai contributors: Ramiro, George, Dan, Stan and more ramping up

Outline

- What/Why/Roadmap

- The `torch` dialect

- Frontends
- Backends

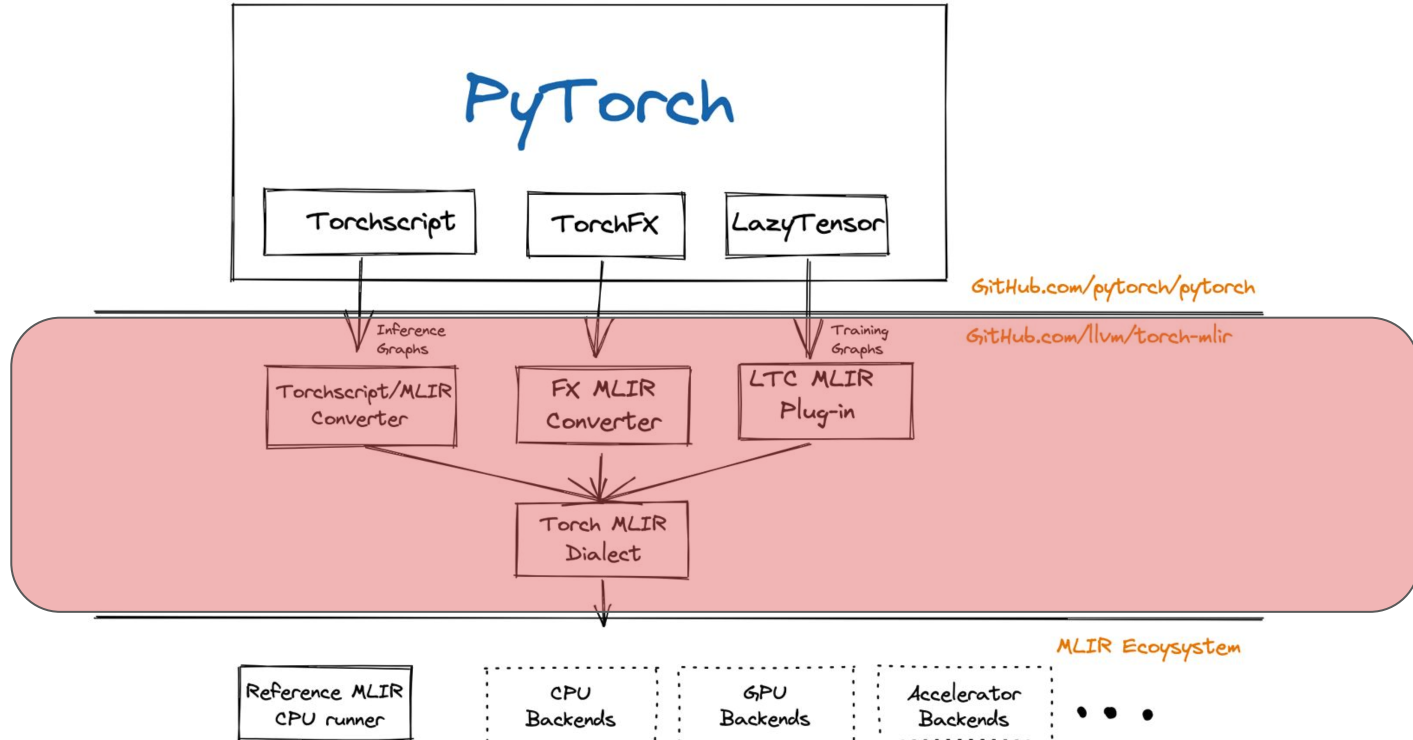


- Demo

What is Torch-MLIR?

Confluence of two large ecosystems - PyTorch and LLVM/MLIR

What is Torch-MLIR?

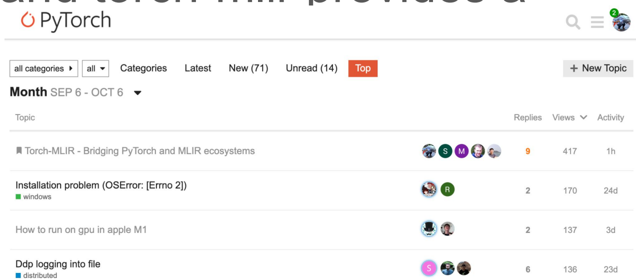


What is Torch-MLIR?

- Aim to centralize PyTorch/MLIR interop code paths into a single project
  - Downstreams focus on their unique value instead of building yet another PyTorch/MLIR frontend.
  - Native import for popular PyTorch model hubs - torchvision, Huggingface NLP etc
- LLVM incubator project (upstream parts into PyTorch if it makes sense)
- Dual licensed (LLVM + PyTorch BSD) to facilitate code movement upstream
- https://github.com/llvm/torch-mlir or #torch-mlir on LLVM discord
- RFC to build with PyTorch Upstream:
  - https://github.com/pytorch/pytorch/pull/65880

Why Torch-MLIR ?

- Lets Silicon Vendors focus on their differentiation and their backend without a need to keep up with the Frameworks.
- I have seen more than five custom implementations of mapping from PyTorch to MLIR. All redundant.
    - Like writing Clang C++ frontend for every LLVM backend / ISA.
- There is a need. Most viewed post this month. 7 vendors expressed support.
    - Express your support too: https://discuss.pytorch.org/t/torch-mlir-bridging-pytorch-and-mlir-ecosystems/133151
- Every platform needs the same building blocks and torch-mlir provides a clean abstraction for the backends



The Future:

- Validated Model/Op Support
  - Huggingface, torchvision etc
- Tighter integration with PyTorch
  - Solidify LazyTensorCore, TorchFX path
  - Potentially have a native LTC->MLIR plugin, like LTC->XLA (in torch_xla)
- First class training and inference support
- Compatibility test suite for Silicon Vendors
  - Push button qualification of PyTorch integration
  - Benchmarks as part of CTS
- Example dialects for customers to introduce custom functionality.
- Silicon Vendors: What would make your life easier ?

The `torch` dialect

The `torch` dialect

- Auto-generated from source-of-truth Torch op registry
    - All Torch programs bottom out on the same set of dispatchable kernels
- A few hand-defined ops for modeling program structures, such as class types, instances, etc. (relevant for advanced features of TorchScript)
- Types for faithfully modeling Torch/Python type system.
    - !torch.tensor + !torch.vtensor (value-semantic tensor)
    - E.g. !torch.tensor<[3,4,?],unk>, !torch.vtensor<*,f32>, …
    - Builtin tensor does not:
        - Model non-value semantics
        - Have first-class support for unknown dtype (element type)
    - !torch.int, !torch.float, !torch.list<T>, !torch.dict<K, V>
    - class types

Op autogeneration: `torch.aten.relu`

Info extracted from Torch registry

```
JitOperator 'aten::relu : (Tensor) ->
(Tensor)':
  MLIR op name = torch.aten.relu
  MLIR td def name = Torch_AtenReluOp
  namespace = aten
  unqualified_name = relu
  overload_name =
  is_c10_op = True
  is_vararg = False
  is_varret = False
  is_mutable = False
  arguments:
    arg: {'name': 'self', 'type':
'Tensor', 'pytype': 'Tensor'}
  returns:
    ret: {'name': '', 'type': 'Tensor',
'pytype': 'Tensor'}
```

ODS

```
def Torch_AtenReluOp : Torch_Op<"aten.relu", [
   AllowsTypeRefinement,
   HasValueSemantics
 ]> {
 let summary = "Generated op for `aten::relu :
(Tensor) -> (Tensor)`";
 let arguments = (ins
   AnyTorchTensorType:$self
 );
 let results = (outs
   AnyTorchTensorType:$result
 );
 let assemblyFormat = "$self attr-dict `:`
type($self) `->` type($result)";
}
```

Op autogeneration: `torch.aten.relu_`

```
JitOperator 'aten::relu_ : (Tensor) -> (Tensor)':
  MLIR op name = torch.aten.relu_
  MLIR td def name = Torch_AtenRelu_Op
  namespace = aten
  unqualified_name = relu_
  overload_name =
  is_c10_op = True
  is_vararg = False
  is_varret = False
  is_mutable = True
  arguments:
    arg: {'name': 'self', 'type': 'Tensor',
'pytype': 'Tensor', 'alias_info': {'is_write':
True, 'before': ['alias::a'], 'after':
['alias::a']}}
  returns:
    ret: {'name': '', 'type': 'Tensor', 'pytype':
'Tensor', 'alias_info': {'is_write': True,
'before': ['alias::a'], 'after': ['alias::a']}}
```

```
def Torch_AtenRelu_Op : Torch_Op<"aten.relu_",
[
   IsTrailingUnderscoreInplaceVariant,
   AllowsTypeRefinement
 ]> {
 let summary = "Generated op for `aten::relu_ :
(Tensor) -> (Tensor)`";
 let arguments = (ins
   AnyTorchTensorType:$self
 );
 let results = (outs
   AnyTorchTensorType:$result
 );
 let assemblyFormat = "$self attr-dict `:`
type($self) `->` type($result)";
}
```

Key `torch` Dialect Transformations

- ReduceOpVariants
  - Reduce similar ops to a smaller canonical set of ops
- MaximizeValueSemantics
  - Convert as much of the program as possible to value semantics
- RefineTypes
  - Propagate types throughout the program (including dtypes)
- GlobalizeObjectGraph
  - Turn TorchScript object graph into flat list of globals.

ReduceOpVariants: value-semantic ops

```
func @f(%arg0: !torch.tensor<[],f32>) -> !torch.tensor<[],f32> {
 %0 = torch.aten.tanh %arg0 : !torch.tensor<[],f32> -> !torch.tensor<[],f32>
 return %0 : !torch.tensor<[],f32>
}
```

⇒

```
func @f(%arg0: !torch.tensor<[],f32>) -> !torch.tensor<[],f32> {
 %0 = torch.copy.to_vtensor %arg0 : !torch.vtensor<[],f32>
 %1 = torch.aten.tanh %0 : !torch.vtensor<[],f32> -> !torch.vtensor<[],f32>
 %2 = torch.copy.to_tensor %1 : !torch.tensor<[],f32>
 return %2 : !torch.tensor<[],f32>
}
```

ReduceOpVariants: trailing "_" inplace variants

```
func @f(
      %arg0: !torch.tensor<[2],f32>, %arg1: !torch.tensor<[2],f32>) -> (!torch.tensor<[2],f32>, !torch.tensor<[2],f32>) {
 %int1 = torch.constant.int 1
 %0 = torch.aten.add_.Tensor %arg0, %arg1, %int1 : !torch. tensor<[2],f32>, !torch.tensor<[2],f32>, !torch.int ->
!torch.tensor<[2],f32>
 return %0, %arg0 : !torch.tensor<[2],f32>, !torch.tensor<[2],f32>
}
```

⇒

```
func @f(%arg0: !torch.tensor<[2],f32>, %arg1: !torch.tensor<[2],f32>)
-> (!torch.tensor<[2],f32>, !torch.tensor<[2],f32>) {
 %int1 = torch.constant.int 1
 %0 = torch.copy.to_vtensor %arg0 : !torch.vtensor<[ 2],f32>
 %1 = torch.copy.to_vtensor %arg1 : !torch.vtensor<[ 2],f32>
 %2 = torch.aten.add.Tensor %0, %1, %int1 : !torch.vtensor<[ 2],f32>, !torch.vtensor<[2],f32>, !torch.int ->
!torch.vtensor<[2],f32>
 %3 = torch.copy.to_tensor %2 : !torch. tensor<[2],f32>
 %4 = torch.copy.to_vtensor %3 : !torch.vtensor<[ 2],f32>
 torch.overwrite.tensor %4 overwrites %arg0 : !torch.vtensor<[ 2],f32>, !torch.tensor<[2],f32>
 return %arg0, %arg0 : !torch.tensor<[2],f32>, !torch.tensor<[2],f32>
}
```

MaximizeValueSemantics: trivial example

```
func @f(%arg0: !torch.vtensor, %arg1: !torch.vtensor) -> (!torch.vtensor, !torch.vtensor) {
 %0 = torch.copy.to_tensor %arg0 : !torch.tensor
 %equal_to_arg0 = torch.copy.to_vtensor %0 : !torch.vtensor
 torch.overwrite.tensor %arg1 overwrites %0 : !torch.vtensor, !torch.tensor
 %equal_to_arg1 = torch.copy.to_vtensor %0 : !torch.vtensor
 return %equal_to_arg0, %equal_to_arg1 : !torch.vtensor, !torch.vtensor
}
```

⇒

```
func @f(%arg0: !torch.vtensor, %arg1: !torch.vtensor) -> (!torch.vtensor, !torch.vtensor) {
 return %arg0, %arg1 : !torch.vtensor, !torch.vtensor
}
```

MaximizeValueSemantics: view-like ops

```
func @f(%arg0: !torch.vtensor) -> !torch.vtensor {
 %int0 = torch.constant.int 0
 %0 = torch.copy.to_tensor %arg0 : !torch.tensor
 %1 = torch.aten.unsqueeze %0, %int0 : !torch.tensor, !torch.int -> !torch.tensor
 %2 = torch.aten.unsqueeze %1, %int0 : !torch.tensor, !torch.int -> !torch.tensor
 %3 = torch.copy.to_vtensor %2 : !torch.vtensor
 return %3 : !torch.vtensor
}
```

⇒

```
func @f(%arg0: !torch.vtensor) -> !torch.vtensor {
 %int0 = torch.constant.int 0
 %0 = torch.aten.unsqueeze %arg0, %int0 : !torch.vtensor, !torch.int -> !torch.vtensor
 %1 = torch.aten.unsqueeze %0, %int0 : !torch.vtensor, !torch.int -> !torch.vtensor
 return %1 : !torch.vtensor
}
```

```python
class MyConvLayer(torch.nn.Module):
    def __init__(self):
        self.conv = torch.nn.Conv2d(2, 10, 3, bias=False)
    def forward(self, x):
        return torch.relu(self.conv(x))
```

Note: all op inputs are Value's in PyTorch semantics! (no Attribute's).

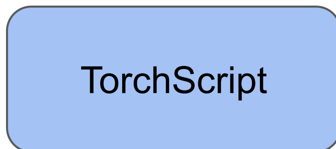
Easy to pattern match into Attribute's as needed though.

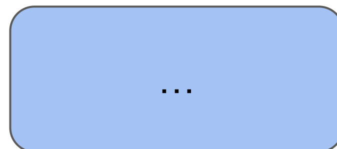
(+ some annotations for input types) ⇒

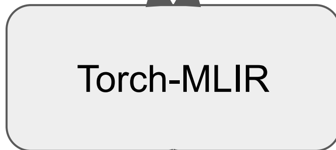
```
func @forward(%arg0: !torch.vtensor<[?,?,?,?],f32>) -> !torch.vtensor<[?,?,?,?],f32> {
 %int1 = torch.constant.int 1
 %int0 = torch.constant.int 0
 %0 = torch.vtensor.literal(opaque<"_", "0xDEADBEEF"> : tensor<10x2x3x3xf32>) :
!torch.vtensor<[10,2,3,3],f32>
 %none = torch.constant.none
 %1 = torch.prim.ListConstruct %int1, %int1 : (!torch.int, !torch.int) -> !torch.list< !torch.int>
 %2 = torch.prim.ListConstruct %int0, %int0 : (!torch.int, !torch.int) -> !torch.list< !torch.int>
 %3 = torch.prim.ListConstruct %int1, %int1 : (!torch.int, !torch.int) -> !torch.list< !torch.int>
 %4 = torch.aten.conv2d %arg0, %0, %none, %1, %2, %3, %int1 : !torch.vtensor<[?,?,?,?], f32>,
!torch.vtensor<[10,2,3,3],f32>, !torch.none, !torch.list<!torch.int>, !torch.list<!torch.int>,
!torch.list<!torch.int>, !torch.int -> !torch.vtensor<[?,?,?,?], f32>
 %5 = torch.aten.relu %4 : !torch.vtensor<[?,?,?,?], f32> -> !torch.vtensor<[?,?,?,?],f32>
 return %5 : !torch.vtensor<[?,?,?,?],f32>
}
```

Frontends & Backends

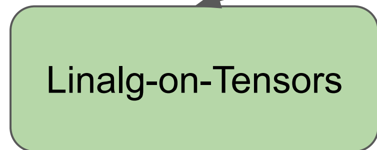
Frontends

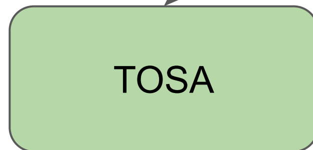


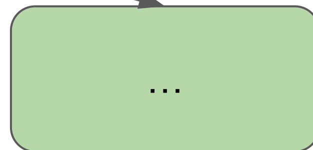


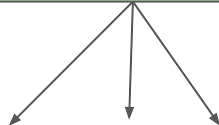


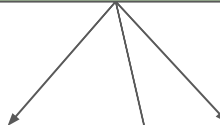
Backends

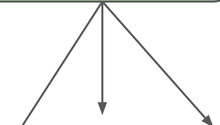












Frontends

Torch Frontends

- TorchScript - high fidelity Python subset (whole program compilation model)
  - A lot of early torch-mlir (nee npcomp) was done in the context of the TorchScript frontend
- TorchFX - symbolic Python-level graph tracing
  - In-progress support for a pure-python TorchFX importer using our Python bindings
- LazyTensorCore - device-level graph tracing
  - In-progress support for seamless under-the-hood graph tracing + dispatch to your backend

TorchScript to MLIR importer

- Well-defined TorchScript IR (torch::jit::{Node,Block})
  - Very similar to MLIR actually.
- Well-defined TorchScript runtime representation (c10::IValue)
- <1kLOC to systematically import entire representation
- Whole program capture (Python subset)
  - :) Great for generating standalone deployable artifacts
  - :(Not so great if you just want a graph of tensor ops, and don’t care about (/can’t handle) the rest of the program
- int, float, list<T>, dict<K,V>, class types
  - Accurately models Python (Torch) type ontology.

```python
class TestModule(torch.nn.Module):
    def __init__(self):
        self.arange = torch.arange(4)
    def forward(self, x):
        return x * self.arange
```

⇒

```
func private @__torch__.TestModule.forward(
     %arg0: !torch.nn.Module<"__torch__.TestModule">, %arg1: !torch.tensor) -> !torch.tensor {
 %2 = torch.prim.GetAttr %arg0["arange"] : !torch.nn.Module<"__torch__.TestModule"> -> !torch.tensor
 %3 = torch.aten.mul.Tensor %arg1, %2 : !torch.tensor, !torch.tensor -> !torch.tensor
 return %3 : !torch.tensor
}
torch.class_type @__torch__.TestModule  {
 torch.attr "arange" : !torch.tensor
 torch.method "forward", @__torch__.TestModule.forward
}
%0 = torch.tensor.literal(dense<[0, 1, 2, 3]> : tensor<4xsi64>) : !torch.tensor<[4],si64>
%1 = torch.nn_module  {
 torch.slot "arange", %0 : !torch.tensor<[4],si64>
} : !torch.nn.Module<"__torch__.TestModule">
```

GlobalizeObjectGraph

```
func private @  torch  .TestModule.forward(
      %arg0: !torch.nn.Module<"__torch__.TestModule">, %arg1: !torch.tensor) -> !torch.tensor {

 %2 = torch.prim.GetAttr %arg0["arange"] : !torch.nn.Module<"  torch  .TestModule"> -> !torch.tensor
 %3 = torch.aten.mul.Tensor %arg1, %2 : !torch.tensor, !torch.tensor -> !torch.tensor
 return %3 : !torch.tensor
}
…
%1 = torch.nn_module  { // Identified as root module.
 torch.slot "arange", %0 : !torch.tensor<[4],si64>
} : !torch.nn.Module<"__torch__.TestModule">
```

⇒

```
// Easy to inline @arange after this transformation - mutation/aliasing/etc. easy to analyze.
torch.global_slot @arange : !torch. tensor  {
 %0 = torch.tensor.literal(dense<[0, 1, 2, 3]> : tensor<4xsi64>) : !torch.tensor<[4],si64>
 torch.global_slot.init %0 : !torch. tensor<[4],si64>
}
func @forward(%arg0: !torch. tensor) -> !torch.tensor {
 %0 = torch.global_slot.get @arange : !torch.tensor
 %1 = torch.aten.mul.Tensor %arg0, %0 : !torch. tensor, !torch.tensor -> !torch.tensor
 return %1 : !torch.tensor
}
```

Backends

Interop with builtin tensor type.

- !torch.vtensor converts trivially to builtin `tensor` if dtype is known
- Linalg-on-tensors lowering
  - 100% dynamic shapes + runtime error guards for mismatching dimensions!
    - Requires statically inferred rank
- TOSA, MHLO soon? Let’s do this :)
- This also involves lowering !torch.int to i64 and !torch.float to f64.
- The TorchConversion (`torch_c`) dialect facilitates interop with builtin types.
- Hacky reference flow using linalg-on-tensors + bufferize + ctypes + PyExecutionEngine
  - Easy to plug in a real linalg-on-tensors capable backend compiler/runtime instead.

Linalg-on-tensors for Conv2D example from earlier

```
#map = affine_map<(d0, d1, d2, d3) -> (d0, d1, d2, d3)>
module  {
 func @forward(%arg0: tensor<?x2x?x?xf32>) -> tensor<?x?x?x?xf32> {
   %cst = constant opaque<"_", "0xDEADBEEF"> : tensor<10x2x3x3xf32>
   %c-2_i64 = constant -2 : i64
   %cst_0 = constant 0.000000e+00 : f32
   %c0 = constant 0 : index
   %c2 = constant 2 : index
   %c3 = constant 3 : index
   %0 = tensor.dim %arg0, %c0 : tensor<?x2x?x?xf32>
   %1 = tensor.dim %arg0, %c2 : tensor<?x2x?x?xf32>
   %2 = tensor.dim %arg0, %c3 : tensor<?x2x?x?xf32>
   %3 = index_cast %1 : index to i64
   %4 = addi %3, %c-2_i64 : i64
   %5 = index_cast %4 : i64 to index
   %6 = index_cast %2 : index to i64
   %7 = addi %6, %c-2_i64 : i64
   %8 = index_cast %7 : i64 to index
   %9 = linalg.init_tensor [%0, 10, %5, %8] : tensor<?x10x?x?xf32>
   %10 = linalg.fill(%cst_0, %9) : f32, tensor<?x10x?x?xf32> -> tensor<?x10x?x?xf32>
   %11 = linalg.conv_2d_nchw_fchw {dilations = dense<1> : vector<2xi64>, strides = dense<1> : vector<2xi64>} ins(%arg0, %cst : tensor<?x2x?x?xf32>, 
tensor<10x2x3x3xf32>) outs(%10 : tensor<?x10x?x?xf32>) -> tensor<?x10x?x?xf32>
   %12 = linalg.generic {indexing_maps = [#map, #map], iterator_types = [ "parallel", "parallel", "parallel", "parallel"]} ins(%11 : tensor<?x10x?x?xf32>) outs(%9 
: tensor<?x10x?x?xf32>) {
   ^bb0(%arg1: f32, %arg2: f32):  // no predecessors
     %14 = cmpf ugt, %arg1, %cst_0 : f32
     %15 = select %14, %arg1, %cst_0 : f32
     linalg.yield %15 : f32
   } -> tensor<?x10x?x?xf32>
   %13 = tensor.cast %12 : tensor<?x10x?x?xf32> to tensor<?x?x?x?xf32>
   return %13 : tensor<?x?x?x?xf32>
 }
}
```

E2E testing framework

- Op-level correctness test suite
- Larger models correctness tests (ResNet, BERT, …)
- Plug in your compiler+runtime with a small Python file
    - Can live in your downstream repo – does not need to be made public / pushed upstream.
- Supports complex flows like running on prototype hardware, remote lab devices, etc.
- XFAIL support for incremental bringup
- Excellent error handling/diagnostics for when things don’t go as expected

Demos

Join us:
#torch-mlir on LLVM discord
https://github.com/llvm/torch-mlir