AsiaLLVM Developers' Meeting, June 10, 2025, Tokyo, Japan

ONNX-MLIR: An MLIR-based Compiler for ONNX AI models

Tung D. Le (speaker), Alexandre E Eichenberger, Tong Chen, Haruki Imai, Yasushi Negishi, Kiyokuni Kawachiya

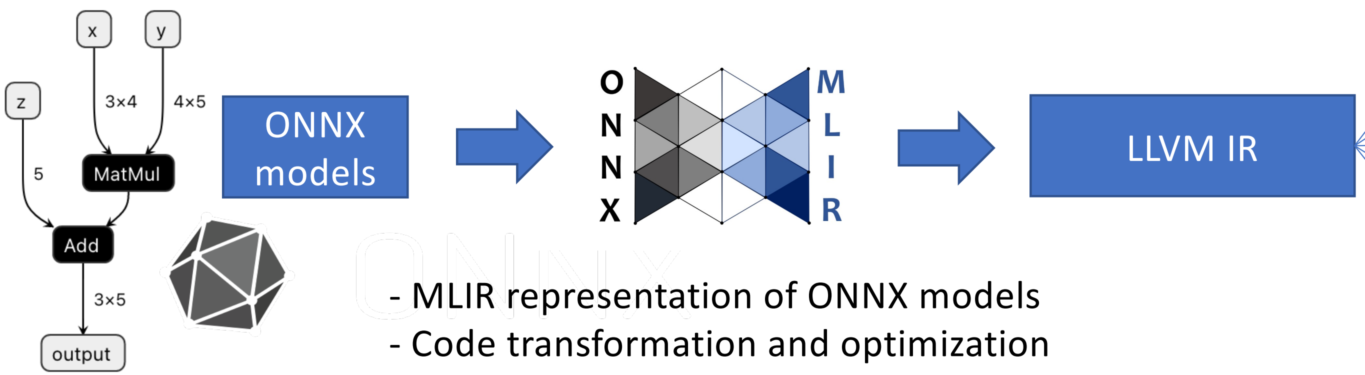
IBM Research

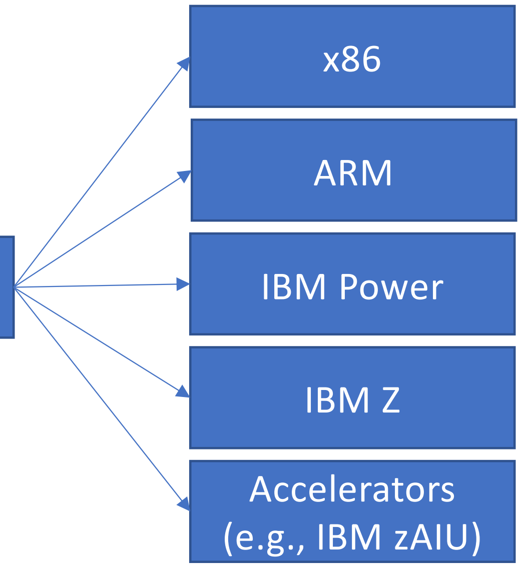
Presenting the work of many other people!

https://github.com/onnx/onnx-mlir

What is onnx-mlir?

- Compile an ONNX AI model to an optimized binary for inferencing using
  - MLIR to perform high-level transformations
  - LLVM to perform low-level optimizations and code generation
- GitHub (open-source): https://github.com/onnx/onnx-mlir
  - Initially developed by IBM Research since 2019
  - > 100 contributors from AMD, ByteDance, Groq, Microsoft, etc.
  - Cited by ~90 scientific papers

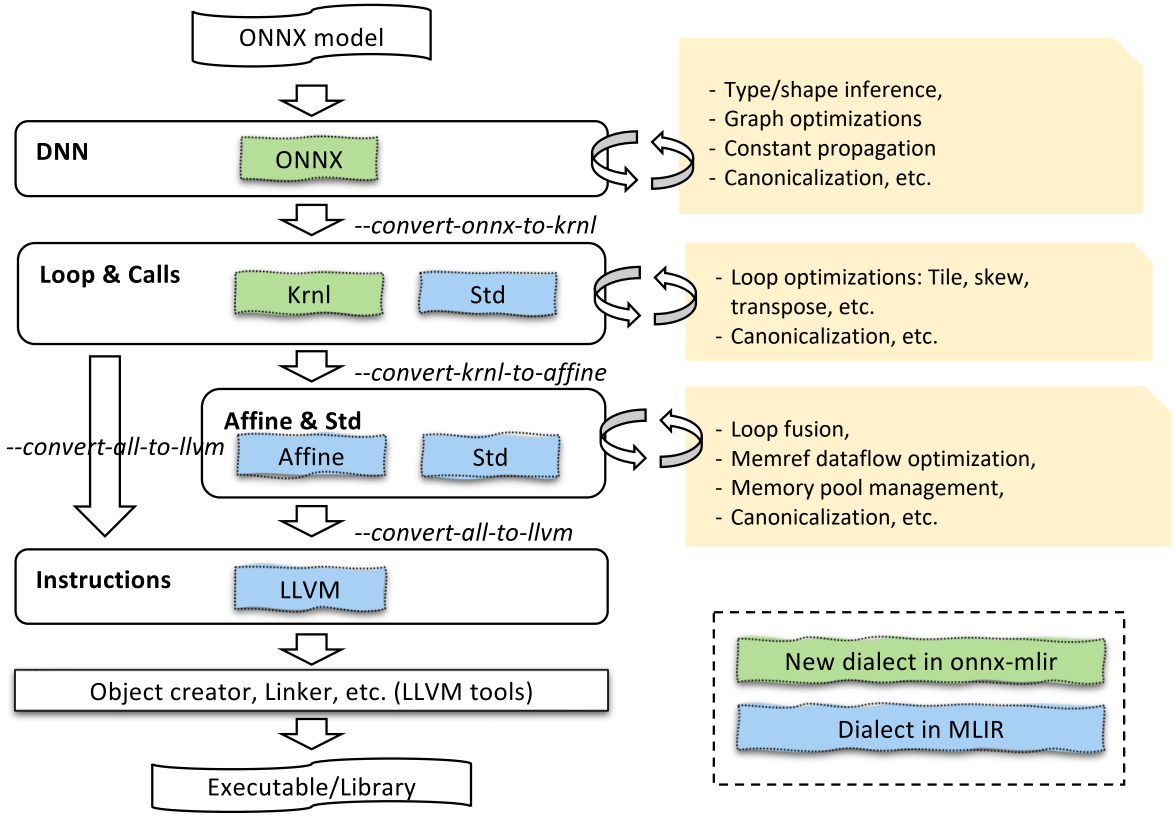




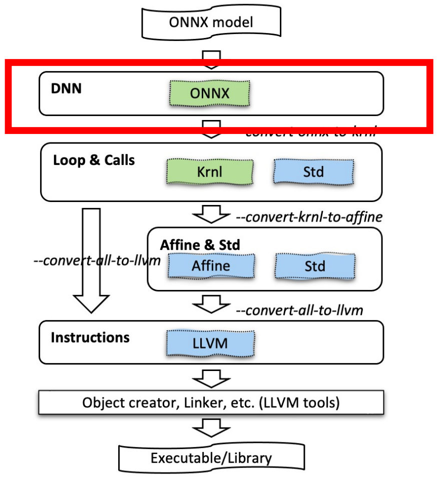
Design goals

- A reference ONNX dialect in MLIR
- Easy to write optimizations for CPU and custom accelerators
  - From high-level (e.g., graph level) to low-level (e.g., instruction level)
- Easy to deploy
  - Stand-alone driver and runtime support in C/C++/Java/Python
  - Integration into other applications
- Continuously tested
  - Unit tests for each operation and ONNX model zoo
  - x86, Arm, Power, z/Architecture
  - Windows, Linux, z/OS, macOS
  - C/C++/Java/Python

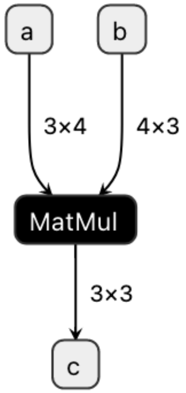
Core Dialects and Transformations



ONNX Dialect



- Presentation of an ONNX model in MLIR language
- Additional operation, EntryPoint, to specify the entry function for doing inference



Importer

```
module {
  func.func @main_graph(%arg0: tensor<3x4xf32>, %arg1: tensor<4x3xf32>) -> tensor<3x3xf32>
                                  attributes {input_names = ["a", "b"], output_names = ["c"]} {
    %0 = "onnx.MatMul"(%arg0, %arg1) : (tensor<3x4xf32>, tensor<4x3xf32>) -> tensor<3x3xf32>
    return %0 : tensor<3x3xf32>
  }
  "onnx.EntryPoint"() {func = @main_graph} : () -> ()
}
```

ONNX Model

ONNX Model represented in MLIR language

ONNX Dialect

- ONNX Dialect is automatically generated by a python script from the ONNX specification

Add

Performs element-wise binary addition (

This operator supports **multidirectional**

(Opset 14 change): Extend supported ty

Version

This version of the operator has been av

Other versions of this operator: 1, 6, 7, 1

Inputs

A *(differentiable)* : *T*
First operand.

B *(differentiable)* : *T*
Second operand.

Outputs

C *(differentiable)* : *T*
Result, has same element type as two

Type Constraints

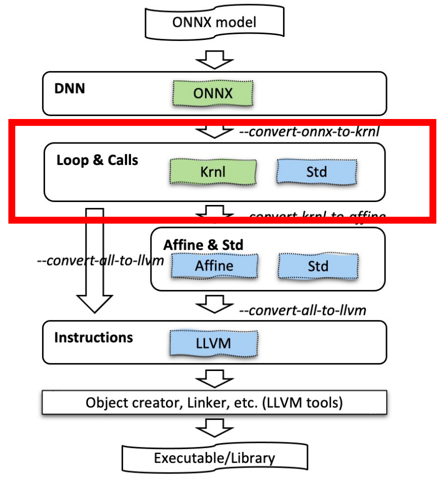
T *: tensor(uint8), tensor(uint16), tens*
tensor(float16), tensor(float), tensor(
Constrain input and output types to al

```
def ONNXAddOp:ONNX_Op<"Add",
  [Pure, DeclareOpInterfaceMethods<ShapeInferenceOpInterface>]> {
  let summary = "ONNX Add operation";
  let description = [{
  Performs element-wise binary addition (with Numpy-style broadcasting support).

  This operator supports **multidirectional (i.e., Numpy-style) broadcasting**; for more details please
check [the doc](Broadcasting.md).

  (Opset 14 change): Extend supported types to include uint8, int8, uint16, and int16.
  }];
  let arguments = (ins AnyTypeOf<[TensorOf<[UI8]>, TensorOf<[UI16]>, TensorOf<[UI32]>,
                                  TensorOf<[UI64]>, TensorOf<[I8]>, TensorOf<[I16]>,
                                    TensorOf<[I32]>, TensorOf<[I64]>, TensorOf<[F16]>,
                                    TensorOf<[F32]>, TensorOf<[F64]>, TensorOf<[BF16]>]>:$A,
                       AnyTypeOf<[TensorOf<[UI8]>, TensorOf<[UI16]>, TensorOf<[UI32]>,
                                    TensorOf<[UI64]>, TensorOf<[I8]>, TensorOf<[I16]>,
                                    TensorOf<[I32]>, TensorOf<[I64]>, TensorOf<[F16]>,
                                    TensorOf<[F32]>, TensorOf<[F64]>, TensorOf<[BF16]>]>:$B);
  let results = (outs AnyTypeOf<[TensorOf<[UI8]>, TensorOf<[UI16]>, TensorOf<[UI32]>,
                                  TensorOf<[UI64]>, TensorOf<[I8]>, TensorOf<[I16]>,
                                  TensorOf<[I32]>, TensorOf<[I64]>, TensorOf<[F16]>,
                                  TensorOf<[F32]>, TensorOf<[F64]>, TensorOf<[BF16]>]>:$C);
  let hasVerifier = 1;
  let hasCanonicalizer = 1;
}
```

Version Handling

- Each ONNX operator has multiple versions
  - Each version may have different specification, e.g.
    - Squeeze version 1 has one input (data) and one attribute (axes)
    - Squeeze version 11 the same as version 1
    - Squeeze version 13 has two inputs (data and axes)
- onnx-mlir supports lowering for the latest version only.
- When importing an operator with an old version, it is converted into the **closest** newer version, e.g.
  - We have two rewritten rules for Squeeze to convert
    - version 1 to 11
    - version 11 to 13
  - If there were a new version, say 15, only one rule is needed to convert version 13 to 15

Krnl Dialect



- Operations to define loop iterations:
  - krnl.define_loops to define loops, krnl.iterate to iterate over loops
- Operations for optimizations:
  - krnl.block for tiling, krnl.permute for permutation, etc.

(Original) Loop definition

Affine schedule for blocking/tiling

```
%ii = krnl.define_loops 1
%ib, %il = krnl.block %ii 2 : (!krnl.loop)->(!krnl.loop, !krnl.loop)
krnl.iterate(%ib, %il) with (%ii -> %i = 0 to 10) {
  %foo = std.addi %i, %i : index
}
```

Scalar computation using original loops

Iteration domain using original loops

Convert to Affine dialect

```
#map0 = affine_map<(d0) -> (d0)>
#map1 = affine_map<(d0) -> (d0 + 2)>
affine.for %arg0 = 0 to 10 step 2 {
  affine.for %arg1 = #map0(%arg0) to #map1(%arg0) {
    %0 = addi %arg1, %arg1 : index
  }
}
```

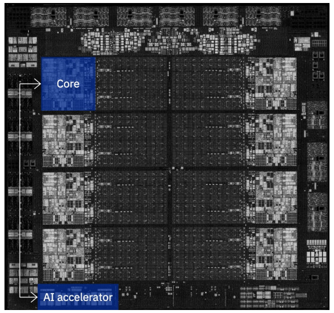
Index Expression

- Problem:
  - **ONNX Shape Inference**: generate literal values or question marks at *compute time*.
  - **Shape Lowering**: generate literals or create operations that compute shapes at *runtime*.
- Index expressions (IndexExpr and subclasses)
  - Polymorphic class that represents computations over shapes (e.g. add/ceil/select…).
- ShapeHelper (ONNXOpShapeHelper and subclasses)
  - Encapsulate how to compute the shape for a give ONNX operation.
  - Each ONNX operation defines its own/reuse a subclass.
  - Runtime code is generated if a builder is given. Otherwise, question marks.

IBM Telum on-chip AI accelerator (zAIU)

- A new **on-chip accelerator** for AI on IBM Z machines
  - High-speed and real-time inferencing at scale
  - More than 6 TFLOPs of 16-bit floating point processing power

- IBM Z Deep Learning Library (zDNN)
  - A very thin wrapper in C for neural-network-processing-assist (NNPA) instructions
  - A set of primitives: matmul, conv, lstm, etc.
  - E.g., *zdnn_status* ***zdnn_add****(*
    *const zdnn_ztensor *input_a,*
    *const zdnn_ztensor *input_b,*
    *zdnn_ztensor *output);*
  - Open sourced at: https://github.com/IBM/zDNN



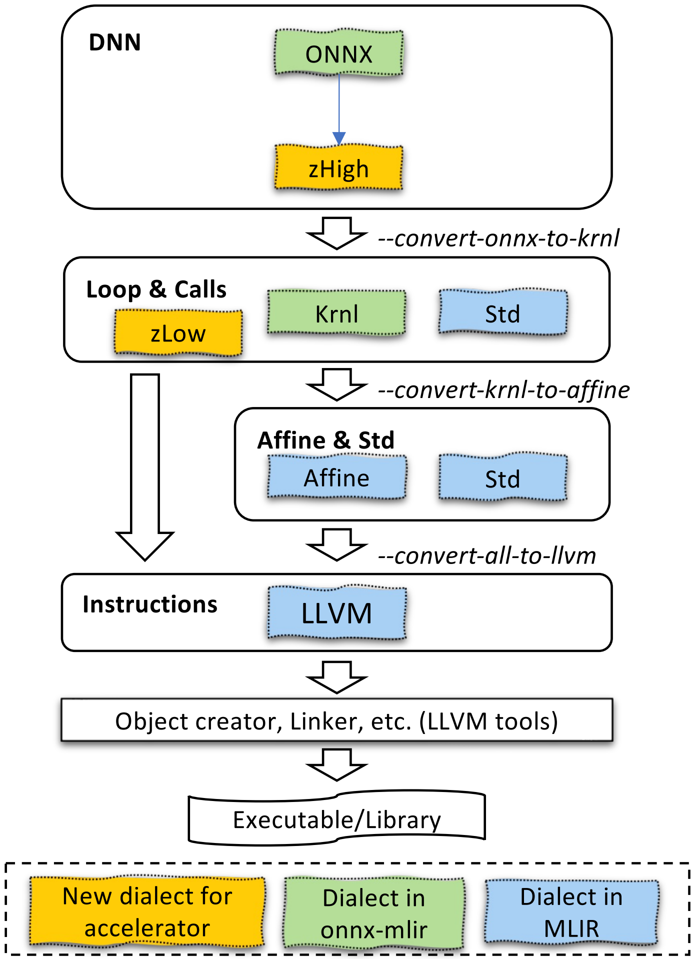
IBM Telum Chip in z16 mainframes

IBM z16 mainframe

Support zAIU Accelerator

- zHigh represents high-level operations for accelerator
  - Tensor-based representation of zDNN APIs
  - Operations for for data transformation

- zLow represents low-level operations on actual memory
  - Memory buffer allocation
  - Operation's signature is like zDNN API's one



Representation of zTensor in MLIR

Tensor in ONNX dialect

```
tensor<4x8xf32>
```

to zHigh dialect

zTensor in zHigh dialect

```
tensor<4x8xf32, #zhigh.encoding<{dataLayout = "2D"}>>
```

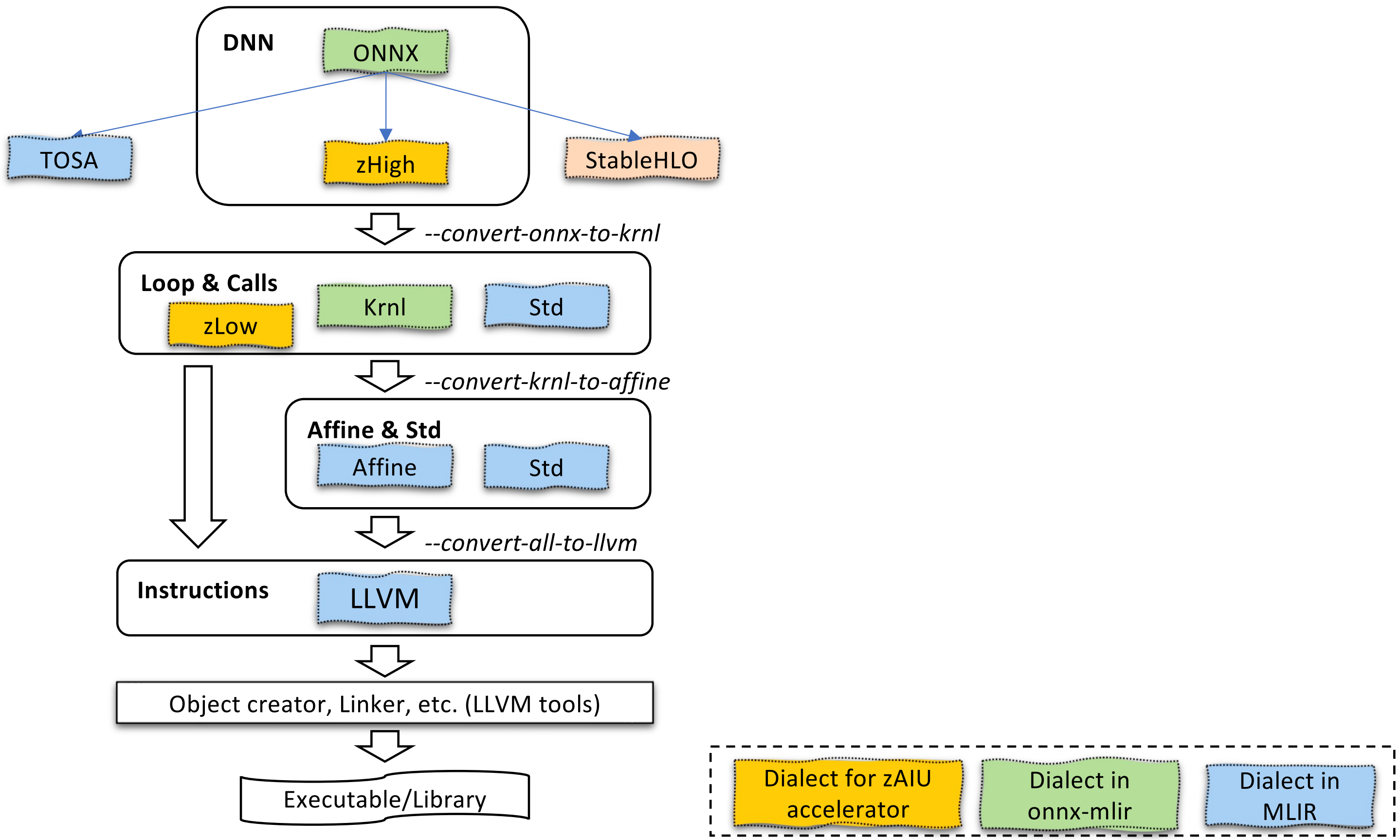
to zLow dialect: buffer allocation, element mapping using affine functions

MemRef in zLow dialect

```
#tiling = affine_map<(d0, d1) -> (0, d1/64, 0, d0/32, d0%32, d1%64)>
%0 = memref.alloc() {alignment = 4096 : i64} : memref<4x8xf16, #tiling>
```

Original shape is preserved throughout the transformation

Other lowering paths



Memory buffer optimization

- onnx-mlir initially had its own bufferization to optimize buffer reuse
- MLIR introduced a bufferization
  - We updated onnx-mlir to use the new bufferization
- MLIR introduced a new bufferization
  - We updated onnx-mlir to use the new bufferization
  - Found that runtime performance was lost by 2x, and reported to MLIR
  - We introduced a compile flag in onnx-mlir to switch between the old and new one
- Finally the new bufferization has worked well and we are using it

Attribute for big contants

- onnx-mlir initially used DenseElementsAttr for storing learned weights
- Memory consumption during compilation was really big
  - Memory increased after each constant folding => peak memory was 5x large than the model size
- Temporary solution:
  - Manually allocate buffers for constant folding
  - Only create DenseElementsAttr at the end of constant folding
  - Memory consumption is still 2x of the model size.
- MLIR introduced ElementsAttr interface that allows defining custom storage.
- onnx-mlir created DisposableElementsAttr based on ElementsAttr
  - Memory consumption is close to the model size

LLVM tools: slow compilation time for AI models

```
Total Execution Time: 174.1031 seconds
----Wall Time----  ----Name----
  0.0032 (  0.0%)  [onnx-mlir] Loading Dialects
  1.5719 (  0.9%)  [onnx-mlir] Importing ONNX Model to MLIR Module from "bert-base-uncased-onnx-18.onnx"
 32.4585 ( 18.6%)  [onnx-mlir] Compiling and Optimizing MLIR Module
 84.0119 ( 48.3%)  [onnx-mlir] Translating MLIR Module to LLVM and Generating LLVM Optimized Bitcode (llvm opt)
 54.6438 ( 31.4%)  [onnx-mlir] Generating Object from LLVM Bitcode (llvm llc)
  1.3087 (  0.8%)  [onnx-mlir] Linking and Generating the Output Shared Library
  0.1052 (  0.1%)  Rest
174.1031 (100.0%)  Total
```

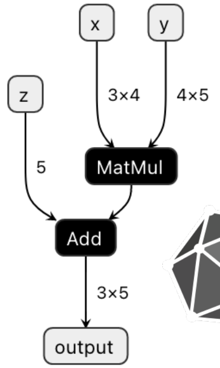
Exporting all constant values to an external file before calling LLVM tools

```
Total Execution Time: 139.0119 seconds
----Wall Time----  ----Name----
  0.0033 (  0.0%)  [onnx-mlir] Loading Dialects
  1.5862 (  1.1%)  [onnx-mlir] Importing ONNX Model to MLIR Module from "bert-base-uncased-onnx-18.onnx"
 32.7726 ( 23.6%)  [onnx-mlir] Compiling and Optimizing MLIR Module
 46.6286 ( 33.5%)  [onnx-mlir] Translating MLIR Module to LLVM and Generating LLVM Optimized Bitcode (llvm opt)
 57.9360 ( 41.7%)  [onnx-mlir] Generating Object from LLVM Bitcode (llvm llc)
  0.0663 (  0.0%)  [onnx-mlir] Linking and Generating the Output Shared Library
  0.0189 (  0.0%)  Rest
139.0119 (100.0%)  Total
```

Interacting with MLIR community

- We submit patches in MLIR when needed
  - Big-endian related issues
  - Memref normalization in the case of dynamic dimensions
  - Affine loop fusion enhancement

- We actively follow and involve in discussion on https://discourse.llvm.org/



ONNX models → ONNX MLIR → LLVM IR → x86 / ARM / IBM Power / IBM Z / Accelerators (e.g., IBM zAIU)

- MLIR representation of ONNX models
- Code transformation and optimization
- GitHub: https://github.com/onnx/onnx-mlir

Thank you for your attention!