Compiling ONNX Neural Network Models Using MLIR

Tian Jin[1,†1,*] Gheorghe-Teodor Bercea[1] Tung D. Le[2,*] Tong Chen[1]
Gong Su[1] Haruki Imai[2] Yasushi Negishi[2] Anh Leu[1] Kevin O'Brien[1]
Kiyokuni Kawachiya[2] Alexandre E. Eichenberger[1]

Abstract: Deep neural network models are becoming increasingly popular and have been used in various tasks such as computer vision, speech recognition, and natural language processing. Machine learning models are commonly trained in a resource-rich environment and then deployed in a distinct environment such as high availability machines or edge devices. To assist the portability of models, the open-source community has proposed the Open Neural Network Exchange (ONNX) standard. In this paper, we present a high-level, preliminary report on our onnx-mlir compiler, which generates code for the inference of deep neural network models described in the ONNX format. Onnx-mlir is an open-source compiler implemented using the Multi-Level Intermediate Representation (MLIR) infrastructure recently integrated in the LLVM project. Onnx-mlir relies on the MLIR concept of dialects to implement its functionality. We propose here two new dialects: (1) an ONNX specific dialect that encodes the ONNX standard semantics, and (2) a loop-based dialect to provide for a common lowering point for all ONNX dialect operations. Each intermediate representation facilitates its own characteristic set of graph-level and loop-based optimizations respectively. We illustrate our approach by following several models through the proposed representations and we include some early optimization work and performance results.

1. Introduction

Deep neural network models have been used widely for various tasks such as computer vision, speech recognition, and natural language processing. The success of such models was mainly originated from the development of accelerators, especially GPU accelerators, back in 2012 [3]. Since then, many deep learning frameworks, such as Torch, Caffe, Theano, and TensorFlow, have been developed to facilitate the training and inferencing of deep neural network models, which significantly speeds up the explosion of deep learning in many areas. However, training and inferencing are often done on different environments due to their different optimization characteristics. For example, a model is trained using a large-scale distributed system since it might need weeks or months to finish, and can then be used on light-weight devices such as Internet of Things or mobile phones for inferencing. Hence, it is desirable to dynamically rewrite a trained model so that it runs efficiently on a target environment.

Many deep learning frameworks utilize a highly-optimized library written for a target accelerator. Rewriting a model for inferencing consists of replacing the operations in the model with the function calls in the library. While such a library-call approach simplifies the rewritten procedure and would lead to improved performance, it exposes the following drawbacks. Firstly, the number of models that can be rewritten is limited by the provided functions in the library. Secondly, it is often the case that users need to install additional packages to make the library work well. Thirdly, it lacks the ability to tailor code specific to different problems since the same function may be used for them.

We tackle these drawbacks by developing a compiler that rewrites a trained model to native code for a target hardware. It uses many mature optimization techniques developed during the long history of compiler, such as the ability to tailor code for a specific problem, memory optimizations, and parallelization. Our compiler is completely based on open-source software. In particular, we chose Open Neural Network Exchange (ONNX) [1] as a format to represent the input model of our compiler. ONNX is an open-source machine-independent format and widely used for exchanging neural network models. It has been actively maintained by and contributed from open source communities. Our compiler was written using Multi-level Intermediate Representation (MLIR) [5], a modern open source compiler infrastructure for multi-level intermediate representations and a subproject inside LLVM [4].

Our compiler is completely open-sourced and a subproject

[1] IBM T.J. Watson Research Center,
1101 Kitchawan Rd, Yorktown Heights, NY 10598, USA
[2] IBM Research - Tokyo,
19-21, Nihonbashi Hakozaki-cho, Chuo-ku, Tokyo 103-8510, Japan
[†1] Presently with Massachusetts Institute of Technology (MIT)
[*] Corresponding authors: Tung D. Le (tung@jp.ibm.com), Tian Jin (tianjin@mit.edu)

inside the ONNX project[†1] . Although it is still under development, it can already compile some popular models such MNIST and ResNet50 to native code on x86 machines, IBM Power Systems[†2] , and IBM System Z[†3] . In this paper, we will introduce our compiler by

- presenting its overall design and architecture of the compiler,
- introducing two new dialects: onnx dialect to encode the ONNX standard semantics, and krnl dialect to provide for a common lowering point for all ONNX dialect operations.
- introducing optimization passes such as graph rewriting, constant propagation, and memory management, and
- discussing some problems we encountered when emitting native code for different architectures.

The remainder of the paper is organized as follows. In Sec. 2, we briefly discuss ONNX and MLIR on which our compiler is based. In Sec. 3, we introduce our compiler, its design principle, and architecture. We also discuss in this section two new dialects, i.e., onnx and krnl, and some optimization passes. In Sec. 4, we present some preliminary experiemental results for MNIST and ResNet50 models on IBM Power Systems. Finally, we conclude our paper and discuss future work in Sec. 5.

2. Background

2.1 ONNX

Open Neural Network Exchange (ONNX) [1] is an open source format for artificial intelligence models, including both deep learning and traditional machine learning. It defines an extensible computational graph model, operators, and standard data types, which provides a common IR for different frameworks. There are two ONNX variants: the neural-network-only ONNX variant recognizes only tensors as input and output types, while the classic machine learning ONNX-ML also recognizes sequences and maps. ONNX-ML extends the ONNX operator set with machine learning algorithms that are not based on neural networks. In this paper, we focus on the neural-network-only ONNX variant and refer to it as just ONNX.

In ONNX, the top-level structure is a 'Model' to associate metadata with a graph. Operators in ONNX are divided into a set of primitive operators and functions, where a function is an operator whose calculation can be expressed via a subgraph of other operators. A graph is used to describe a function. There are lists of nodes, inputs, outputs, and initializers (constant values or default values for inputs) in a graph. An acyclic dataflow graph is constructed as a topological sort of the list of nodes in the graph. Each node in a graph contains the name of the operator it invokes, inputs, outputs, and attributes associated with the operator. Inputs and outputs can be marked as variadic or optional. There are three data types used to define inputs and outputs, i.e., 'Tensor', 'Sequence', and 'Map'.

ONNX uses the Protocol Buffers[†4] definition language for its syntax. Listing 1 shows an example of an ONNX model for the LeakyRelu operator. There is one node in the graph (Lines 4–13), which is associated with LeakyRelu, and has

Listing 1: ONNX model for LeakyRelu operator (printed using 'protoc' command).

```
1  ir_version: 3
2  producer_name: "backend-test"
3  graph {
4    node {
5      input: "x"
6      output: "y"
7      op_type: "LeakyRelu"
8      attribute {
9        name: "alpha"
10       f: 0.1
11       type: FLOAT
12     }
13   }
14   name: "test_leakyrelu"
15   input {
16     name: "x"
17     type {
18       tensor_type {
19         elem_type: 1
20         shape {
21           dim {
22             dim_value: 3
23           }
24           dim {
25             dim_value: 4
26           }
27           dim {
28             dim_value: 5
29           }
30         }
31       }
32     }
33   }
34   output {
35     name: "y"
36     type {
37       tensor_type {
38         elem_type: 1
39         shape {
40           dim {
41             dim_value: 3
42           }
43           dim {
44             dim_value: 4
45           }
46           dim {
47             dim_value: 5
48           }
49         }
50       }
51     }
52   }
53 }
54 opset_import {
55   version: 9
56 }
```

[†1] https://github.com/onnx/onnx-mlir
[†2] https://www.ibm.com/it-infrastructure/power/power9
[†3] https://www.ibm.com/it-infrastructure/z/hardware
[†4] https://developers.google.com/protocol-buffers

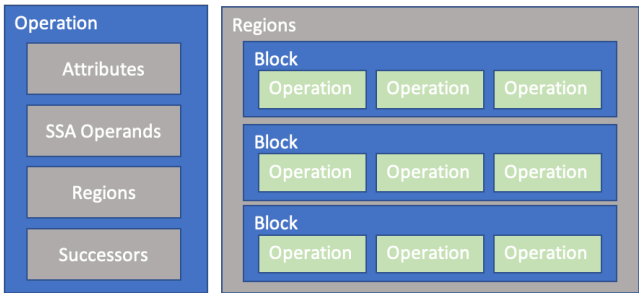

Fig. 1: Operations and Regions in MLIR.

one input, one output, and one attribute. The input and output tensors have the shape of ⟨3x4x5⟩ and element type of float32 (elem_type: 1 at Lines 19 and 38).

2.2 MLIR

Multi-level Intermediate Representation (MLIR) [5] is a modern compiler infrastructure which is reusable and extensible. It reduces the cost of building domain-specfic compilers by facilitating the design and implementation of code generators, translators, and optimizers at different abstraction levels. MLIR is a subproject of the LLVM project [6] and has many similarities to the LLVM compiler infrastructure [4]. In this section, we briefly review some of the features in MLIR that were used to build our compiler. For more information about MLIR, one can refer to a previous study [5]. Readers who are familiar with MLIR can skip this section.

Similar to LLVM, MLIR is a three-address static single assignment (SSA)-based IR, where values are defined before use and have a scope defined by their dominance relations. Operations may produce zero or more results, and each operation is a distinct SSA value with its own type defined by the type system. The type system in MLIR is open, and one can define application-specific types. There are a number of primitive types, e.g., integers, as well as aggregate types for tensors and memory buffers, e.g., 'Tensor' and 'MemRef' types. A Tensor type is abstracted and does not have a pointer to the data while a MemRef type is a lower representation, referring to a region of memory. In MLIR, Tensor and MemRef types are syntactically represented as tensor$\langle D_1 \times D_2 \times \ldots \times D_N \times$dtype$\rangle$ and memref$\langle D_1 \times D_2 \times \ldots \times D_N \times$dtype$\rangle$, respectively, where $D_1, D_2, \ldots, D_N$ are intergers representing the dimensions of a tensor or memref, and dtype is the type of the elements in a tensor or memref, e.g., f32 for float32. $\langle D_1 \times D_2 \times \ldots \times D_N \rangle$ is called the shape of a tensor or memref. Tensor and MemRef types can be unranked when their shapes are unknown. In MLIR, unranked Tensor and MemRef types are syntactically represented as tensor$\langle * \times$dtype$\rangle$ and memref$\langle * \times$dtype$\rangle$, respectively.

An *operation* is the unit of code in MLIR. To define an operation, a TableGen-based [7] specification for an operation descriptor is used. Figure 1 shows the structure of an operation. An operation has a list of SSA operands and may have attributes that store static information. An operation can hold a *region* which is a list of *blocks*. A *block* contains a list of operations and ends with a *terminator* operation that may have *successor* blocks to which the control flow may be transferred. That being said, *nested regions* becomes a first-class concept in MLIR, which is efficient to represent control flow graphs. A *function* is an operation with a single region and attributes. A *module* is an operation with a single region containing a single block and terminated by a dummy operation.

To develop a compiler using MLIR, users often need to define *dialects* and *optimization passes*. A dialect serves as an abstraction level or intermediate representation, and an optimization pass is to enable optimization at an abstraction level or transformation among abstraction levels.

There are dialects in MLIR that are ready to use, e.g., llvm, std, scf, and affine. The llvm dialect is a low-level dialect. It wraps the LLVM IR types and instructions into MLIR types and operations. The std dialect includes standard operations such as load, store, addi, addf, absf, and call. The scf dialect defines control flow operations such as for and if. The affine dialect provides an abstraction for affine operations and analyses.

Optimization passes can be roughly classified into three categories: general transformation, conversion, and dialect-specific. General transformation passes includes common passes such as 'canonicalize' pass for operation canonicalization, 'CSE' pass to eliminate common sub-expressions, and passes to print IR information such as 'print-op-graph', 'print-op-stats', and 'print-cfg-graph'. Conversion passes are to convert operations in one dialect to operations in another dialect, e.g., 'convert-std-to-llvm' pass to convert standard operations into LLVM instructions. Finally, dialect-specific passess are for transformation in a dialect, e.g., 'affine-loop-unroll-jam' pass to unroll and jam affine loops in the affine dialect. MLIR passes can be expressed via Declarative Rewriting Rules (DRRs) using tablegen records or via writing code in C++.

To denote an operation in a dialect, we explicitly use a form of dialect_name.operation_name. For example, std.load means the operation load of dialect std. Optimization passes are named with prefix '--', for example, --canonicalize is the canonlicalization pass.

Listing 2 shows an example for calculating the exponential of a given input tensor, element-wise, using std and affine dialects. The top level is a module containing a function 'exp'. The function 'exp' accepts one input that is of memref type, and produces an output of the same type. The memory for the output is allocated via std.alloc (Line 3). There is a nested loop (Lines 4–10), iterating over dimensions of the inputs using affine.for, loading each element from the input using affine.load (Line 6), computing the exponential using std.exp (Line 7), and storing the result in the output using affine.store (Line 8). The output of the function is finally returned using std.return.

Listing 2: Compute the exponential of a tensor in MLIR.

```
module {
  func @exp(arg0: memref<3x4xf32>) -> memref<3x4xf32> {
    %1 = std.alloc() : memref<3x4xf32>
    affine.for %arg1 = 0 to 3 {
      affine.for %arg2 = 0 to 4 {
        %2 = affine.load %arg0[%arg1, %arg2] : memref<3x4xf32>
        %3 = std.exp %2 : f32
        affine.store %3, %1[%arg1, %arg2] : memref<3x4xf32>
      }
    }
    std.return %1 : memref<3x4xf32>
  }
}
```

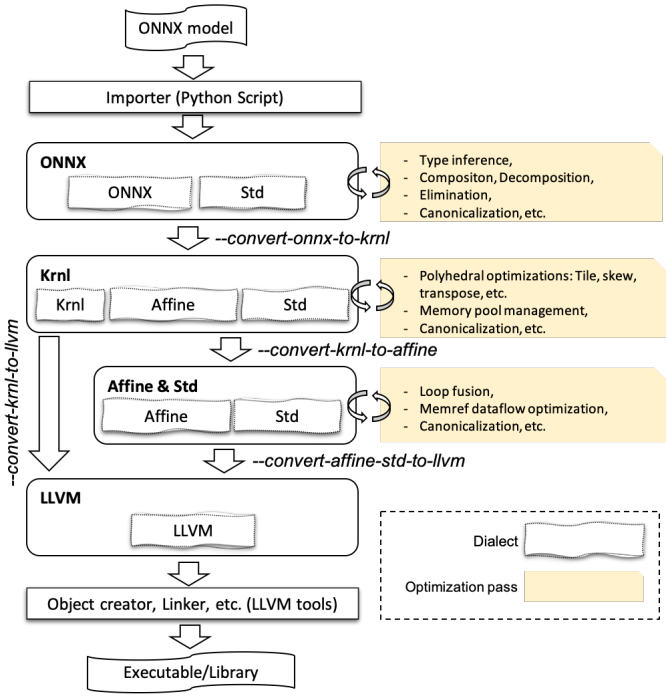

Fig. 2: Architecture of onnx-mlir. Names prefixed with '--' are passes.

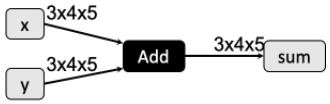

Fig. 3: ONNX model for element-wise addition.

3. Compiling ONNX Models

This section introduces our compiler, *onnx-mlir*. We first discuss its overall architecture. We then introduce two new dialects, onnx and krnl dialects. Finally, we present MLIR passes for carrying out optimization.

3.1 Overview

Figure 2 shows the overall architecture of *onnx-mlir*. The input is an ONNX model, and the output is a library containing the compiled code. The output library contains an entry function called '_dyn_entry_point_main_graph' whose inputs and outputs are similar to the ONNX model's inputs and outputs, respectively. To carry out inference with the output library, users write their program to call the entry function by passing inputs to the function and obtain results.

There are five main dialects in *onnx-mlir*, i.e., onnx, krnl, affine, std and llvm, organized into four abstraction levels. Two new dialects, onnx and krnl, are discussed in Sections 3.2 and 3.3, respectively. The first abstraction level is a high-level representation of ONNX operations. It consists of operations in onnx and std dialects, where the onnx dialect is automatically generated via *an importer* that is a python script. The second abstraction level includes krnl, affine and std dialects. krnl dialect provides a representation that is suitable for loop optimizations, which is able to carry out affine transformations such as tile, skew, and permutation easily. It plays as an intermediate dialect for efficiently lowering the onnx dialect into low-level dialects (e.g., affine, std and llvm). The third abstraction level includes affine and std dialects where existing optimization passes in MLIR can be freely applied. The forth abstraction level includes only llvm dialect that is realy to generate bitcode.

There are MLIR passes for converting one dialect to another, and for doing optimizations at a specific dialect. onnx dialect is converted to krnl dialect via pass --convert-onnx-to-krnl. Then krnl dialect (except some of its operations) is converted into affine and std dialects via pass --convert-krnl-to-affine. The remaining operations in krnl dialect and operations in affine and std dialects are directly converted into instructions in llvm via pass --convert-krnl-to-llvm. The right side of Fig. 2 shows optimization passes that can be carried out at each abstraction level.

We only enumerate the important optimizations here, and the list of optimization passes is not exhaustive.

Before discussing dialects and optimization passes in detail, we give a brief running example and go through dialects in *onnx-mlir*. This example is a testcase model in ONNX that performs element-wise binary addition. Figure 3 shows this ONNX model of the testcase. Operation add accepts two tensors of type ⟨3x4x5xf32⟩ (element type is float 32) and returns a result tensor, i.e., sum, of the same type. Listings 3, 4, and 5 show emitted programs in different dialects onnx, krnl, affine, respectively. We omit the program in llvm

Listing 3: Operation add in onnx dialect, generated using importer.

```
module {
  func @main_graph( %arg0:tensor<3x4x5xf32>, %arg1:tensor<3x4x5xf32>) -> tensor<*xf32> {
    %0 = "onnx.Add"(%arg0, %arg1) : (tensor<3x4x5xf32>, tensor<3x4x5xf32>) -> tensor<*xf32>
    std.return %0 : tensor<*xf32>
  }
  "onnx.EntryPoint"() {func = @main_graph, numInputs = 2 : i32, numOutputs = 1 : i32} : () -> ()
}
```

Listing 4: Operation add in krnl dialect, generated by applying passes --shape-inference and --convert-onnx-to-krnl.

```
module {
  func @main_graph(%arg0: memref<3x4x5xf32>, %arg1: memref<3x4x5xf32>) -> memref<3x4x5xf32> {
    %0 = alloc() : memref<3x4x5xf32>
    %1:3 = krnl.define_loops 3
    krnl.iterate(%1#0, %1#1, %1#2) with (%1#0 -> %arg2 = 0 to 3, %1#1 -> %arg3 = 0 to 4, %1#2 -> %arg4 = 0 to 5) {
      %2 = affine.load %arg0[%arg2, %arg3, %arg4] : memref<3x4x5xf32>
      %3 = affine.load %arg1[%arg2, %arg3, %arg4] : memref<3x4x5xf32>
      %4 = std.addf %2, %3 : f32
      affine.store %4, %0[%arg2, %arg3, %arg4] : memref<3x4x5xf32>
    }
    std.return %0 : memref<3x4x5xf32>
  }
  "krnl.entry_point"() {func = @main_graph, numInputs = 2 : i32, numOutputs = 1 : i32} : () -> ()
}
```

Listing 5: Operation add in affine dialect, generated by applying the pass --convert-krnl-to-affine.

```
module {
  func @main_graph(%arg0: memref<3x4x5xf32>, %arg1: memref<3x4x5xf32>) -> memref<3x4x5xf32> {
    %0 = alloc() : memref<3x4x5xf32>
    affine.for %arg2 = 0 to 3 {
      affine.for %arg3 = 0 to 4 {
        affine.for %arg4 = 0 to 5 {
          %1 = affine.load %arg0[%arg2, %arg3, %arg4] : memref<3x4x5xf32>
          %2 = affine.load %arg1[%arg2, %arg3, %arg4] : memref<3x4x5xf32>
          %3 = std.addf %1, %2 : f32
          affine.store %3, %0[%arg2, %arg3, %arg4] : memref<3x4x5xf32>
        }
      }
    }
    std.return %0 : memref<3x4x5xf32>
  }
  "krnl.entry_point"() {func = @main_graph, numInputs = 2 : i32, numOutputs = 1 : i32} : () -> ()
}
```

due to space limitations.

In onnx dialect, operations are represented similarly to their descriptions in ONNX. The ONNX model is converted into the function main_graph. To generate an entry point function into which users feed their inputs, we create a helper operation in the onnx dialect, i.e., onnx.EntryPoint, which keeps meta-data in the operation's attributes such as function name to call and the number of inputs and outputs.

In krnl dialect, operation onnx.Add is translated into a loop-based computation represented by operations in the krnl dialect, where scalar computation is represented by primitive operations in the affine and std dialects. We can apply loop optimizations, such as tile, skew, or transpose, to loop-based computation. At this level, we allocate memory for output tensors, and memory management can be performed.

In affine dialect, optimized loop-based computation in krnl dialect is translated into affine.for loops. At this level, we still have an operation in krnl, i.e., krnl.entry_point. Such an operation is not related to the main computation and will be directly converted to llvm. Operations in the affine dialect will be converted to operations in the std and scf dialects before being lowered to instructions in the llvm dialect.

3.2 onnx dialect

onnx dialect is the first abstraction level in *onnx-mlir* and represents an ONNX model in MLIR language. We wrote a python script to automatically import ONNX operations into the tablegen-based operation definitions in MLIR. These imported operations are organized into the onnx dialect. Thanks to tablegen, the operation definition in the onnx dialect is quite similar to the operation description in ONNX, where we are able to represent all necessary information, such as inputs, outputs, attributes, and description, into a single tablegen-based definition in human-readable textual form.

Listing 6: Tablegen-based definition for operation relu.

```
1 def ONNXLeakyReluOp:ONNX_Op<"LeakyRelu",
2   [NoSideEffect, DeclareOpInterfaceMethods<ShapeInferenceOpInterface>]> {
3     let summary = "ONNX LeakyRelu operation";
4     let description = [{"LeakyRelu takes ... "}];
5     let arguments = (ins AnyTypeOf<[TensorOf<[F16]>, TensorOf<[F32]>, TensorOf<[F64]>]>:$X,
          DefaultValuedAttr<F32Attr, "0.01">:$alpha);
6     let results = (outs AnyTypeOf<[TensorOf<[F16]>, TensorOf<[F32]>, TenorOf<[F64]>]>:$Y);
7     let extraClassDeclaration = [{ ... }];
8 }
```

We also created a new operation in the onnx dialect, i.e., onnx.EntryPoint to keep information related to the dynamic list of inputs in an ONNX model. This operation will be lowered to generate the entry function '_dyn_entry_point_main_graph' of the generated library.

Listing 6 shows a tablegen-based definition for the relu operation imported via the importer in *onnx-mlir*. The operation description is represented in the 'description' field (Line 4). Inputs and attributes are represented in the 'arguments' field, while outputs were represented in the 'results' field (Lines 5–6). All inputs and outputs will be imported as a tensor in MLIR. The importer automatically infers element types for inputs, attributes, and outputs. However, the shape of a tensor will be inferred via the --shape-inference pass, which is a trait in the LeakyRelu operation (Line 2). MLIR generates a C++ class definition for an operation from its tablegen-based definition. If users want to define custom declaration in the class, it can be done via the 'extraClassDeclaration' field (Line 7).

3.3 krnl dialect

A computation kernel in a neural network workload has local structural simplicity in which loop nests are often simple, e.g., hyper-rectangle and statements carry quite straightforward arithmetic semantics. Such a characteristic is quite suitable to be represented in a polyhedral model for optimization [8]. krnl dialect aims to host both loop optimization and scalar semantic optimization in a single representation. It is expected to provide interpretability where not only is polyhedral representation readable but it also makes program semantics (or what to execute) and program schedules (how and when to execute) independent. In other words, our goal is to optimize not only programs but also the composition of individual schedules, which is a feature that is often lacking in other existing systems.

Below is an example that defines a nested loop in krnl:

```
1 %ii, %jj = krnl.define_loops 2
2 krnl.iterate(%ii, %jj) with (%ii -> %i = 0 to 10, %jj -> %j = 0 to 10) {
3   %foo = std.addi %i, %j : index
4 }
```

where krnl.define_loops defines *two loops*, called ii and jj. These loop variables will be used to express both program semantics and schedules. Operation krnl.iterate semantically accepts two types of loop variables: variables for original loops and variables for scheduled loops. In syntactic sugar form, we separate the two types of loops by the keyword with, i.e. (scheduled loops) with (original loops). Induction variables, e.g., i and j in the above example, will be defined by using original loops. If there is no schedule (e.g. block, skew, etc.), the scheduled loops are similar to the original loops.

Now, we insert a schedule for blocking or tiling. Without loss of generality, we define just one loop instead of two.

```
1 %ii = krnl.define_loops 1
2 %ib, %il = krnl.block %ii 2 : (!krnl.loop)->(!krnl.loop, !krnl.loop)
3 krnl.iterate(%ib, %il) with (%ii -> %i = 0 to 10) {
4   %foo = std.addi %i, %i : index
5 }
```

Operation krnl.block (Line 2) takes a loop and integer as inputs, where the integer is the tile size with which we want to carry out blocking. Results are two loop variables: one for the outer loop and the other for the inner loop. The two loops will be used as the result of scheduling and be passed to krnl.iterate (Line 3). It is worth noting that the original loops and computation in krnl.iterate remained *unchanged* while inserting a schedule, which is exactly what we want for seperating program semantics and schedules in our krnl dialect.

The --convert-krnl-to-affine pass automatically generates optimized affine.for based loops as follows.

```
1 #map0 = affine_map<(d0) -> (d0)>
2 #map1 = affine_map<(d0) -> (d0 + 2)>
3 affine.for %arg0 = 0 to 10 step 2 {
4   affine.for %arg1 = #map0(%arg0) to #map1(%arg0) {
5     %0 = addi %arg1, %arg1 : index
6   }
7 }
```

The outer affine.for iterates with step 2 i.e., the tile size, and the inner affine.for iterates over the elements in a tile.

Other schedules, such as skew and permutation are used in a similar manner. All schedules are composable and can be nested.

3.4 Optimization Passes

In this section, we discuss some of the optimization passes in *onnx-mlir*. Thanks to the expressive power of MLIR, many optimizations can be expressed easily via Declarative Rewriting Rules (DRRs) using tablegen records or writing

code in C++.

3.4.1 Operation Decomposition

In ONNX, many operations can be expressed using other basic operations. For example, ReduceL1 over a vector x is mathematically calculated by summing up the absolute values of the elements in x. In other words, we have

$$\text{ReduceL1} = \text{ReduceSum}\ (\text{Abs}\ x)$$

We only need to lower a subset of operations in the onnx dialect to krnl dialect, while the remaining operations in the onnx dialect will be decomposed into operations in the subset.

Using the DRRs in MLIR, operation decomposition is concisely written as the following pattern:

```
1 def ReduceL1Pattern: Pat<
2   (ReduceL1Op $x, $axes, $keepdims),
3   (ReduceSumOp (AbsOp $x), $axes, $keepdims)
4 >;
```

where ReduceL1Op, ReduceSumOp, and AbsOp are programmable forms of operations onnx.ReduceL1, onnx.ReduceSum, and onnx.Abs respectively. Variables x, axes, and keepdims are for keeping input values of operation ReduceL1Op. The pattern 'ReduceL1Pattern' contains a source pattern to match a graph of one operation ReduceL1Op (Line 2) and a destination pattern to generate a graph of two operations ReduceSumOp and AbsOp (Line 3). Whenever an operation ReduceL1Op appears in an ONNX model, it will be replaced with a combination of ReduceSumOp and AbsOp.

3.4.2 Shape Inference

The --shape-inference pass attempts to infer shapes for all tensors in a program at onnx. The pass traverses all operations in a program, infers the shapes of tensors with unrank shapes (i.e. tensor⟨∗xf32⟩), propagates the ranked shapes to consuming operations, and terminates once all tensors have ranked shapes. For one operation, if its inputs have static shapes, it is likely that the --shape-inference pass will be able to infer static shapes for its outputs. If the inputs have dynamic shapes (e.g. tensor⟨?x?x?xf32⟩), the outputs will also have dynamic shapes also, except for some operations whose output tensors' shapes are specified in the operation attributes.

3.4.3 Graph Rewriting

Graph rewriting is a powerful optimization tool. It is intensively applied to neural networks since calculation in a neural network is expressed via a dataflow graph. In MLIR, graph rewriting rules are conveniently represented using DRRs.

For example, the following rule is to fuse onnx.Add and onnx.MatMul into a single operation onnx.Gemm under the condition that the result of MatMulOp is only consumed by AddOp:

```
1 def MulAddToGemmPattern : Pat<
2   (AddOp (MatMulOp:$res $m1, $m2), $m3),
3   (GemmOp $m1, $m2, $m3),
4   [(HasOneUse $res)]
5 >;
```

Another example is to remove an IdentityOp operation by passing its input directly to its consuming operations.

```
1 def IdentityEliminationPattern : Pat<
2   (ONNXIdentityOp $arg),
3   (replaceWithValue $arg)
4 >;
```

Users can write as many rewriting rules as possible in the same manner.

3.4.4 Constant propagation

Constant propagation is a well-known optimization in compilers. In *onnx-mlir*, we created a pass to do this during compilation. There are two key ideas in constant propagation: (1) if all the inputs of an operation are constant, compute its outputs during compilation and remove the operation, (2) if there is a mix of constant and non-constant inputs, normalize the operation. Normalization is to increase the possibility of constant propagation and strongly depends on the mathematical properties of an operation. Below are some normalization rules in *onnx-mlir* for the onnx.Add operation whose properties are associative and communicative.

(1) $c + x \Rightarrow x + c$

(2) $(x + c_1) + c_2 \Rightarrow x + (c_1 + c_2)$

(3) $(x + c) + y \Rightarrow (x + y) + c$

(4) $x + (y + c) \Rightarrow (x + y) + c$

(5) $(x + c_1) + (y + c_2) \Rightarrow (x + y) + (c_1 + c_2)$

where x and y are non-constant values, and c, c_1, and c_2 are constant values. Normalization rules are expressed by using the DRRs in MLIR.

4. Preliminary Experiments

4.1 ONNX operation support and testcases

ONNX provides a set of test cases for each operation. When we support any operation in *onnx-mlir*, we enable its ONNX test cases to check whether the operation behaves correctly and produces correct result. At the time of writing this paper, *onnx-mlir* supports 51 operations out of 139 operations in ONNX, including important operations such as convolution, pooling, Gemm, and LSTM. These are enough to compile and execute major networks such as MNIST and ResNet50. On the GitHub repository of *onnx-mlir*, we enable continuous integration on different environments, i.e., Windows, Linux, and Docker environments, and different systems, i.e., x86 machines, IBM Power Systems, and System Z. All supported operations have passed tests on the above environments.

4.2 MNIST and ResNet50

In this section, we present some of our preliminary results for two neural network models in the ONNX Model Zoo: MNIST and ResNet50 [2]. The MNIST[†5] and ResNet50[†6]

[†5] https://github.com/onnx/models/tree/master/vision/classification/mnist

[†6] https://github.com/onnx/models/tree/master/vision/classification/resnet

models have already been trained in the CNTK and Caffe2 frameworks, respectively. We ran inferences on the given test data set in each model. The experiments were conducted on a machine with 2.3-GHz POWER9 processors. For *onnx-mlir*, graph rewriting and canonicalization passes were enabled. In this paper, we only provide a reference implementation that is not optimized, thus performance measurements are not applicable.

Table 1: Run inferencing with MNIST and ResNet50 on a POWER9 machine. Time in seconds.

Model	Compilation time	Inference time
MNIST	0.237	0.001
ResNet50	7.661	7.540

Table 1 shows the running times for the MNIST and ResNet50 models when doing inferencing. For each model, we measured the compilation time for compiling the model to native code and inference time for running the native code with real inputs. MNIST is a small model with two convolutional operations, one max pooling operation and a matrix multiplication followed by an element-wise addition. Compiling the MNIST model and carrying out inferencing was rather fast, i.e., finished in less than one second. In the MNIST model, the graph rewriting rule MulAddToGemmPattern mentioned in Sec. 3.4.3 was applied to fuse matrix multiplication and element-wise addition into a Gemm operation. ResNet50 is a complex deep model consisting of 50 layers of operations such as convolutions and poolings. The model is about 100 megabytes including learned weights. For ResNet50, the current version of *onnx-mlir* does not have any optimization applied to the model during compilation. However, we believe that the compilation time looks reasonable and the inference time is not so slow. We hope that once we integrate important optimizations, such as polyhedral optimizations, SIMD optimization, and loop fusion in near future, the inference time will be significantly reduced.

4.3 Supported Systems

Although *onnx-mlir* is completely built upon widely-used open source software such as ONNX and MLIR, we found a problem related to supporting different systems. In particular, we could not run ONNX models on Linux on IBM System Z (s390-linux) because the big-endian format was not well-supported in ONNX and MLIR. There are two reasons for such a problem. First, a large amount of public input data and models in ONNX are stored in little-endian format. Hence, they must be converted to big-endian format before they are used in a big-endian system. Second, we found that constant values in ONNX models are not correctly loaded in MLIR. LLVM was well-supported in big-endian, but MLIR was not. We created two patches to solve this problem: one in ONNX[†7] and one in MLIR[†8] , and they are now available at the master branches of ONNX and MLIR. As a result, *onnx-mlir* now supports Linux on x86 (x86-Linux), Linux on Power Systems (ppc64le-Linux), Linux on IBM Z (s390-Linux), and Windows.

[†7] `https://github.com/onnx/onnx/pull/2633`
[†8] `https://reviews.llvm.org/D78076`

5. Conclusion

We are developing an open source compiler called *onnx-mlir* for compiling ONNX models into native code. MLIR was used as an infrastructure to build the compiler, and two novel dialects were introduced, i.e., onnx and krnl. We also discussed some optimizations such as graph rewriting and constant propagation. It is worth noting that new optimizations can be easily integrated into *onnx-mlir* thanks to the MLIR infrastructure. In the future, we will add more optimizations, e.g., polyhedral optimization, loop fusion, SIMD optimization, and enable code generation for accelerators.

Acknowledgements

We acknowledge the ONNX standard for hosting the project and the external contributors for their contributions.

References

[1] Bai, J., Lu, F., Zhang, K. et al.: ONNX: Open Neural Network Exchange, GitHub (online), available from ⟨https://github.com/onnx/onnx⟩ (accessed 2020-07-01).

[2] He, K., Zhang, X., Ren, S. and Sun, J.: Deep Residual Learning for Image Recognition, *CoRR*, Vol. abs/1512.03385 (online), available from ⟨http://arxiv.org/abs/1512.03385⟩ (2015).

[3] Krizhevsky, A., Sutskever, I. and Hinton, G. E.: ImageNet Classification with Deep Convolutional Neural Networks, *International Conference on Neural Information Processing Systems (NIPS)*, pp. 1097–1105 (2012).

[4] Lattner, C. and Adve, V.: LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation, San Jose, CA, USA, pp. 75–88 (2004).

[5] Lattner, C., Amini, M., Bondhugula, U., Cohen, A., Davis, A., Pienaar, J., Riddle, R., Shpeisman, T., Vasilache, N. and Zinenko, O.: MLIR: A Compiler Infrastructure for the End of Moore's Law, (online), available from ⟨http://arxiv.org/abs/2002.11054⟩ (2020).

[6] LLVM: The LLVM Project, LLVM (online), available from ⟨https://github.com/llvm/llvm-project⟩ (accessed 2020-07-01).

[7] LLVM: TableGen, LLVM (online), available from ⟨https://llvm.org/docs/TableGen/⟩ (accessed 2020-07-01).

[8] Pouchet, L.-N., Bastoul, C., Cohen, A. and Cavazos, J.: Iterative optimization in the polyhedral model: Part II, multidimensional time, *ACM SIGPLAN Notices*, Vol. 43, No. 6, pp. 90–100 (2008).