[Public]

Compiler for DNN Workloads on AMD GPUs

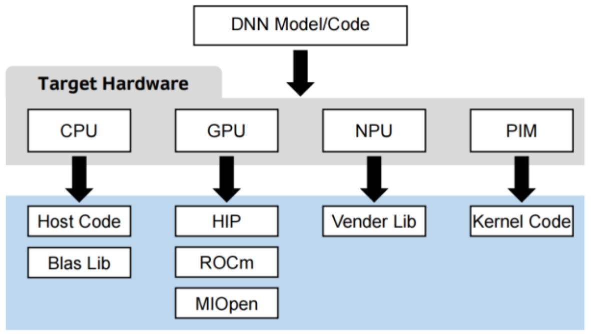
Jumin Lee

Yonsei University, Korea

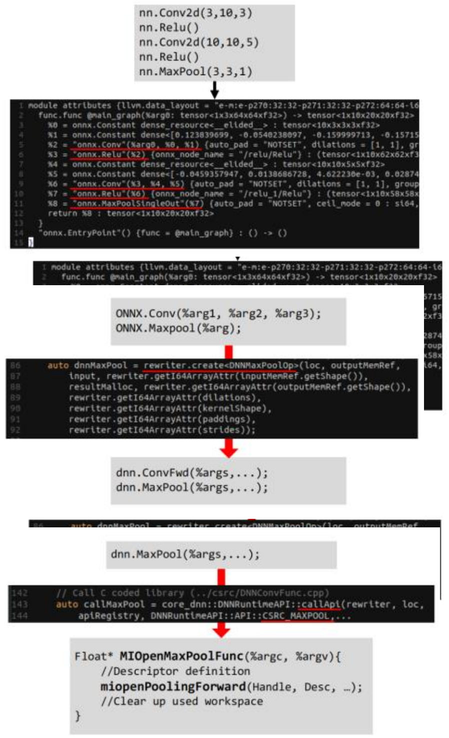
OpenHW2023

INTRODUCTION

The motivation for a DNN Compiler lies in the **diverse architecture environments**, where each hardware platform utilizes unique APIs. This diversity requires the creation of specific code generating paths for different hardware configurations. The challenge arises from the need to adapt and optimize code generation paths to **meet the specific requirements of each hardware environment**. Therefore, the development of a DNN Compiler addresses this challenge by providing a flexible and efficient solution that can dynamically adjust to the varied architectures encountered in different hardware platforms.



Motivation of DNN Compiler



```
nn.Conv2d(3,10,3)
nn.Relu()
nn.Conv2d(10,10,5)
nn.Relu()
nn.MaxPool(3,3,1)
```

```
module attributes {llvm.data_layout = "e-m:e-p270:32:32-p271:32:32-p272:64:64-i6
  func.func @main_graph(%arg0: tensor<1x3x64x64xf32>) -> tensor<1x10x20x20xf32>
    %0 = onnx.Constant dense_resource<__elided__> : tensor<10x3x3x3xf32>
    %1 = onnx.Constant dense<[0.123839699, -0.0540238097, -0.159999713, -0.15715
    %2 = "onnx.Conv"(%arg0, %0, %1) {auto_pad = "NOTSET", dilations = [1, 1], gr
    %3 = "onnx.Relu"(%2) {onnx_node_name = "/relu/Relu"} : (tensor<1x10x62x62xf3
    %4 = onnx.Constant dense_resource<__elided__> : tensor<10x10x5x5xf32>
    %5 = onnx.Constant dense<[-0.0459357947, 0.0138686728, 4.622230e-03, 0.02874
    %6 = "onnx.Conv"(%3, %4, %5) {auto_pad = "NOTSET", dilations = [1, 1], group
    %7 = "onnx.Relu"(%6) {onnx_node_name = "/relu_1/Relu"} : (tensor<1x10x58x58x
    %8 = "onnx.MaxPoolSingleOut"(%7) {auto_pad = "NOTSET", ceil_mode = 0 : si64,
    return %8 : tensor<1x10x20x20xf32>
  }
  "onnx.EntryPoint"() {func = @main_graph} : () -> ()
}
```

```
module attributes {llvm.data_layout = "e-m:e-p270:32:32-p271:32:32-p272:64:64-i6
  func.func @main_graph(%arg0: tensor<1x3x64x64xf32>) -> tensor<1x10x20x20xf32>
```

```
ONNX.Conv(%arg1, %arg2, %arg3);
ONNX.Maxpool(%arg);
```

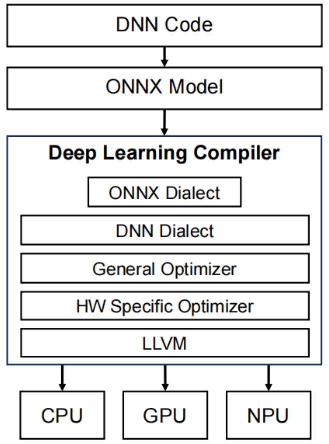
```
    auto dnnMaxPool = rewriter.create<DNNMaxPoolOp>(loc, outputMemRef,
        input, rewriter.getI64ArrayAttr(inputMemRef.getShape()),
        resultMalloc, rewriter.getI64ArrayAttr(outputMemRef.getShape()),
        rewriter.getI64ArrayAttr(dilations),
        rewriter.getI64ArrayAttr(kernelShape),
        rewriter.getI64ArrayAttr(paddings),
        rewriter.getI64ArrayAttr(strides));
```

```
dnn.ConvFwd(%args,...);
dnn.MaxPool(%args,...);
```

```
dnn.MaxPool(%args,...);
```

```
    // Call C coded library (../csrc/DNNConvFunc.cpp)
    auto callMaxPool = core_dnn::DNNRuntimeAPI::callApi(rewriter, loc,
        apiRegistry, DNNRuntimeAPI::API::CSRC_MAXPOOL,...
```

```
Float* MIOpenMaxPoolFunc(%argc, %argv){
    //Descriptor definition
    miopenPoolingForward(Handle, Desc, …);
    //Clear up used workspace
}
```



ONNX MLIR based system

The ONNX graph-to-code converter serves as a publicly available compiler facilitating the transformation of ONNX models to the MLIR framework. **This converter** operates by providing an intermediate representation of DNN operations, including the definition of DNN-related operations and an abstract API for each hardware. Leveraging LLVM calls for **backend APIs**, the compiler ensures a 1-to-1 mapping of DNN dialect to AMD GPU code. Furthermore, it handles translation for hardware-specific libraries and code, while optimizations are conducted at the DNN dialect level. This integrated approach **enables efficient and hardware-adaptive compilation of ONNX models** into the MLIR framework.

CREATIVE DESING

On board test by AMD GPUs

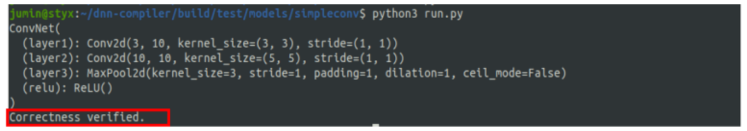
RESULT

Correctness verification

– Evaluated with **1000 iterations** each with random value input

– Convolution with maxpooling and ReLU is found to be correct

– **Extension with ROCblas** is expected in future work



```
jumin@styx:~/dnn-compiler/build/test/models/simpleconv$ python3 run.py
ConvNet(
  (layer1): Conv2d(3, 10, kernel_size=(3, 3), stride=(1, 1))
  (layer2): Conv2d(10, 10, kernel_size=(5, 5), stride=(1, 1))
  (layer3): MaxPool2d(kernel_size=3, stride=1, padding=1, dilation=1, ceil_mode=False)
  (relu): ReLU()
)
Correctness verified.
```

Future Work

- Extension on other benchmarks

– Current implementation is limited on fixed sized CNNs

– Extension on transformer or LLMs is expected

• Optimizations on Multi-tenant benchmarks

– Scheduling in multi tenant workloads are needed

– Memory optimization for scaled situations are considered

• Expansion of backends

– Not only GPU, but NPU and cluster target backends