An Introduction to Practical 2

Lecture 4 for Advanced Deep Learning Systems

Aaron Zhao, Imperial College London, a.zhao@imperial.ac.uk

Table of contents

Introduction

Introduction

Two labs are in Practical 2

- Lab 3: A quantization search using MASE.
- Lab 4 (software stream): A toy Network Architecture Search (NAS) using MASE.
- Lab 4 (hardware stream): Writing and testing a fully-connected layer in SystemVerilog.

Deliverable

- A Markdown file: with all answers (plots, tables ...) of the questions and optional questions.
- Corresponding code in your forked repository.

Examination (15%)

- Submission requires the Markdown files only.
- Lab oral to check on your code and Q&A.

Lab 3: A quantization search using MASE

Problem setup

We allow multi-precision, different layers can use a different precision setup We would like to have at most $X\%$ accuracy degradation, and focus on quantizing the computationally heavy layers (eg. linear, convolution).

- If the network has N layers.
- Each layer has M quantization choices.
- N^M search space.

Classic Approach

```python
class JSC_Tiny(nn.Module):
    def __init__(self, info, qparam):
        super(JSC_Tiny, self).__init__()
        self.seq_blocks = nn.Sequential(
            # 1st LogicNets Layer
            nn.BatchNorm1d(16),  #  batch norm layer
            QuantizedLinear(16, 5, qparam),  # linear layer
        )

    def forward(self, x):
        return self.seq_blocks(x)
...
for qparam in search_space:
  evaluate(model(info, qparam))
```

```
  ...
```

MASE

The classic method is not very scalable because it interleaves network definition with the quantization optimization, what if

- We have a new network
- We have a new optimization or a bunch of optimizations

```
for i, config in enumerate(search_spaces):
  mg = quantize_transform_pass(ori_mg, config)
  evaluate(mg)
```

MASE

The classic method is not very scalable because it interleaves network definition with the quantization optimization, what if

- We have a new network
- We have a new optimization or a bunch of optimizations

```
for i, config in enumerate(search_spaces):
  mg = quantize_transform_pass(ori_mg, config)
  evaluate(mg)
```

MASEGraph Deepdive

The MaseGraph implementation largely relies on the torch fx_graphs.

When traversing an fx_graph, you actually need two components, that are the MASEGraph.fx_graph itself and MASEGraph.modules. One can imagine the fx_graph is a skeleton, it records minimal information

- node.op
- node.target
- node.name
- node.args
- node.kwargs

```
for i, config in enumerate(search_spaces):
  mg = quantize_transform_pass(ori_mg, config)
  evaluate(mg)
```

MASEGraph Deepdive

node.op is "placeholder"

- node.name is set to the variable name for the input
- node.target not used
- node.args not used
- node.kwargs not used

MASEGraph Deepdive

node.op is "call_function"

- node.name is function name
- node.target is the actual function
- node.args is the function arguments
- node.kwargs is the kwargs

node.op is "call_module"

- node.name is module name
- node.target is also the module name
- node.args is the function arguments
- node.kwargs is the kwargs

MASEGraph Deepdive

```
for node in graph.fx_graph.nodes:
  args, kwargs = None, None
  if node.op == "placeholder":
    result = dummy_in[node.name]
  ...
  elif node.op == "call_function":
    args = load_arg(node.args, env)
    kwargs = load_arg(node.kwargs, env)
    result = node.target(*args, **kwargs)
  elif node.op == "call_module":
    args = load_arg(node.args, env)
    kwargs = load_arg(node.kwargs, env)
    result = graph.modules[node.target](*args, **kwargs)
  ...
```

Full code available in the implementation of add_common_metadata pass.

Lab 4 (software stream): A toy Network Architecture Search (NAS) using MASE

What is Network Architecture Search?

We want to pick the optimal architecture $a \in \mathcal{A}$ from a set of architectures $\mathcal{A}$.

At the same time, we want to pick the optimal parameters $w^*(a)$ for the architecture a.

$$\begin{aligned} min_{a \in \mathcal{A}} \mathcal{L}_{val}(w^*(a), a) \\ s.t. w^*(a) = argmin_w(\mathcal{L}_{train}(w, a)) \end{aligned} \tag{1}$$

The idea of multiplied channels

```python
class JSC_Three_Linear_Layers(nn.Module):
  def __init__(self):
    super(JSC_Three_Linear_Layers, self).__init__()
    self.seq_blocks = nn.Sequential(
        nn.BatchNorm1d(16),  # 0
        nn.ReLU(16),  # 1
        nn.Linear(16, 16),  # linear seq_2
        nn.ReLU(16),  # 3
        nn.Linear(16, 16),  # linear seq_4
        nn.ReLU(16),  # 5
        nn.Linear(16, 5),  # linear seq_6
        nn.ReLU(5),  # 7
    )


  def forward(self, x):
    return self.seq_blocks(x)
```

The idea of multiplied channels

```
class JSC_Three_Linear_Layers(nn.Module):
  def __init__(self):
      super(JSC_Three_Linear_Layers, self).__init__()
      self.seq_blocks = nn.Sequential(
          nn.BatchNorm1d(16),
          nn.ReLU(16),
          nn.Linear(16, 32),  # output scaled by 2
          nn.ReLU(32),  # scaled by 2
          nn.Linear(32, 64),  # input scaled by 2 but output scaled by 4
          nn.ReLU(64),  # scaled by 4
          nn.Linear(64, 5),  # scaled by 4
          nn.ReLU(5),
      )

  def forward(self, x):
      return self.seq_blocks(x)
```

The idea of multiplied channels

- The idea is to scale the input and output channels of the linear layer by a constant factor.
- Consecutive linear layers must be scaled by the same factor.
- Search through all the possible factors (brute force and Bayesian).

Lab 4 (hardware stream): Writing and testing a fully-connected layer in SystemVerilog

The goal of Lab4

Automatically generate a fully-connected layer in SystemVerilog, and test it using Cocotb.

Cocotb is a COroutine based COsimulation TestBench environment for verifying VHDL and SystemVerilog RTL using Python.

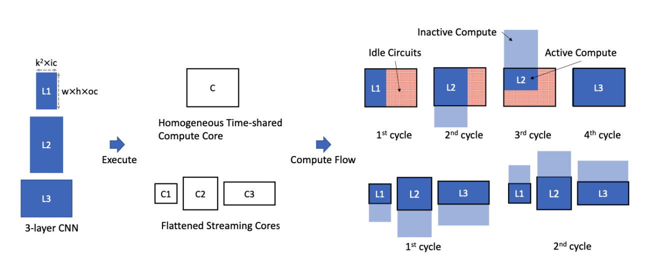
EmitVerilog

- Classic source to source generation
- Directly generate SystemVerilog from MaseGraph

```python
from chop.passes.graph.transforms import (
    emit_verilog_top_transform_pass,
    emit_internal_rtl_transform_pass,
    emit_bram_transform_pass,
    emit_verilog_tb_transform_pass,
)
```

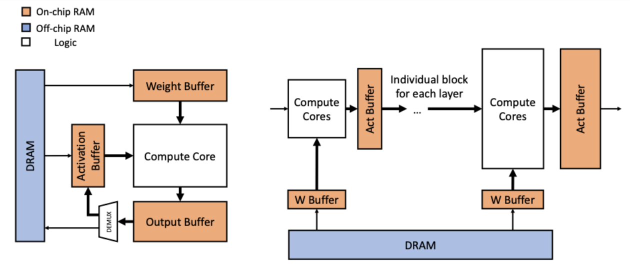
EmitVerilog

- Generate functional elements (RTL)
- Generate memory components (BRAM)
- Dataflow accelerator design without making use of the DRAM



Dataflow accelerator design: Overview

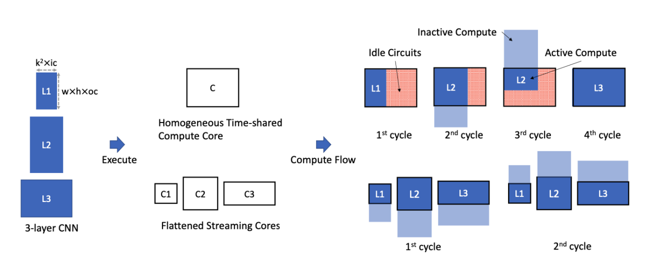
- A homogeneous Big Compute Core (normal design, ASIC)
- A series of tailored small compute cores (dataflow design, FPGA)



Dataflow accelerator design

Advantages

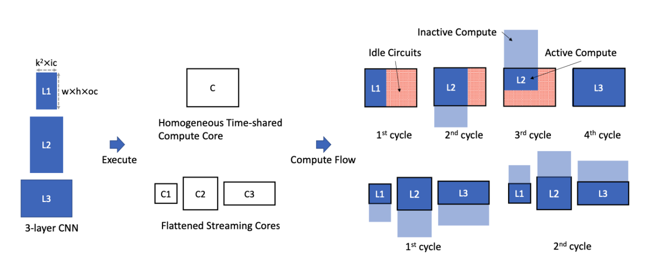
- No complex control flow (minimal or no ISA design)
- (Almost) no waste of resources
- (Almost) fixed memory access pattern
- Deep pipeline



Dataflow accelerator design

Disadvantages

- Re-program hardware for each new network
- Scalability issues
- If DRAM is utilized, hard to achieve great performance by filling up all pipeline stages



The compute pattern

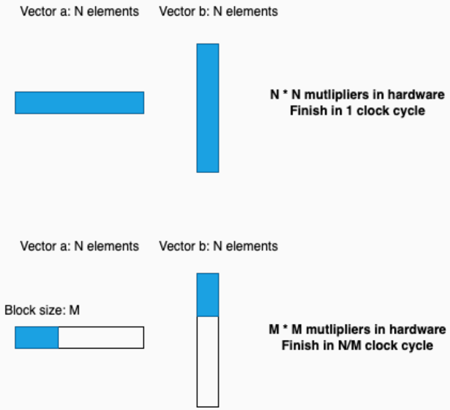
Simple blocking

```python
# Breaking the vector into blocks
for i in range(0, n, block_size):
    # calculate end val considering the last block which can be smaller than block_size
    end_val_i = min(i + block_size, n)

    # Retrieving block of a
    sub_a = a[i : end_val_i]
    # Retrieving corresponding elements from vector
    sub_b = b[i : end_val_i]

    # multiplication, actual hardware dimension is (block_size, 1)
    result += np.dot(sub_a, sub_b)
```

The compute pattern



- $N >> M$, this gives you a chance to do a trade-off between resources and latency simply by changing M.
- Blocking can happen in a 2D shape!