Lab 3: CNN Accelerator Design-1

The lab is due on Nov. 15, 2024 (make sure you keep on top of things and the two parts below)

This lab has 2 main parts: (1) quantizing and pruning a network to make it more hardware (FPGA) friendly, and (2) designing and implementing two critical NN operators.

Part 1: Creating an Accelerator-Friendly CNN

The CNN model we created in Lab1 is based on FP32 arithmetic. We now optimize it using integer quantization and pruning to make it more hardware-friendly and reduce compilation times for an FPGA. This part of the lab has three main steps and is PyTorch based.

Baseline Model: Below is the baseline model that we will use throughout this lab. We pre-trained the model. You can find the PyTorch code in the Colab Notebook linked below and the actual trained parameters on S3: lab3_ref.ipynb

```
aws s3 cp s3://mlxlo-2024-ut-austin/shared/saved_myconvnet.pt .
```

```
class MyConvNet(nn.Module):
    def __init__(self, args):
        super(MyConvNet, self).__init__()
        self.conv1 = nn.Conv2d(1, 16, kernel_size=3, stride=1,
                               padding=1)
        self.act1  = nn.ReLU(inplace=True)
        self.pool1 = nn.MaxPool2d(kernel_size=2)
        self.conv2 = nn.Conv2d(16, 32, kernel_size=3, stride=1,
                               padding=1)
        self.act2  = nn.ReLU(inplace=True)
        self.pool2 = nn.MaxPool2d(kernel_size=2)
        self.lin2  = nn.Linear(7*7*32, 10)

    def forward(self, x):
        x = self.conv1(x)
        x = self.act1(x)
        x = self.pool1(x)
        x = self.conv2(x)
        x = self.act2(x)
        x = self.pool2(x)
        x = x.view(x.size(0), -1)
        x = self.lin2(x)
        return x
```

Step 1: Quantization

Here we explore several integer quantization schemes of increasing effectiveness. We use integer quantization with a scaling factor per tensor/layer. [Note: a Colab notebook posted on Canvas contains a lot of useful starter code for your work.]

Task 1: Weight Quantization

Using PyTorch, quantize model *weights* for several quantization bit widths (num_bits). (For this task, keep activation representation unchanged.) Explore test accuracy vs. num_bits for the following values of num_bits: 4, 8, 12, and 16.

a) *Full-range*: first, use a quantization scheme that covers the entire range of weights found in FP32. ([q_min = min(weight), q_max = max(weight)]. Compare for:
(i) Symmetric signed quantization; and (ii) Asymmetric quantization.

b) *Optimal range settings for better quantization*: Repeat (a) by modifying the range [q_min, q_max] covered by your quantization scheme to remove outliers. See if you can get better accuracy.

(c) *Per-layer quantization:* Modify the PyTorch procedure to allow layer-specific quantization (specific scaling factors). Report accuracy changes.

Task 2: Activation + Weight Quantization

Extend Task 1 by introducing simulated quantization of activations. (Note: this implies you need to requantize the result of FC/CONV layers.) For this experiment, use only full-range asymmetric quantization for both weights and activations. Compare to the accuracy you got for this setting in Task 1.

Step 2: Pruning

Task 1: Structured Pruning: Here we explore structured pruning: pruning of entire filters and/or channels in CONV layers for more efficient execution on an accelerator. To mimic the filter/channel pruning and identify its impact on accuracy, we will fill the filters being pruned with zeros in PyTorch (though they should be rearranged after training for reduced matrix dimensions on the accelerator). You may find the following tutorial helpful.

Starting with MyConvNet, try both filter-wise and channel-wise pruning and explain how you determined the target filters/channels (hint: recall the threshold and fraction pruning methods from Lab1).

Minimize the model size (in KB) while achieving test accuracy of >85%. Report the model size. Only use structured pruning.

Task 2: Combining Pruning and Quantization: We now combine quantization and pruning to reduce the model size further while meeting a certain accuracy target:

a) For accuracy targets of 80%, 85%, and 90%, what is the minimum model size you achieve with the pruning and quantization methods of this lab, respectively? Explain your strategies to achieve the accuracy goal while minimizing model size.

b) Which technique (quantization or pruning) was more effective in terms of reducing model size for a small degradation of accuracy? Why?

Step 3: Exporting Model Weights

Interfacing with the Vitis-generated accelerator is most easily done by dumping the weights and quantization parameters of each layer. You may find it helpful to dump various intermediate values and results to files as well to aid in debugging.

(Task 1) Write a `save_weights(path)` member function of MyConvNet that exports the quantized weights (Uniform Asymmetric Quantization) of each layer into a file. You will load these parameters and weights will be provided as input data to your accelerator in the future lab (Lab 4).

Part 2: Commonly Used NN Operators

In this part of the lab, you will extend the matrix multiplication unit that you designed in Lab 2 and make it more flexible. We provide host-side code and a testbench to help your design validation process. Your implementation can conform to the provided C++ API below. Our skeleton code is designed such that the implementation of each operator may be completed in parallel if you choose to distribute work among team members. These tasks must be completed on AWS.

Skeleton Code

For your convenience, we provide host code, a testbench, and C++ function signatures for each op you are expected to implement. Each operator is contained within its own folder. Implement your operators within our provided skeletons.

When testing your code with the testbench, use the following workflow.

```
make cbin TARGET=sw_emu PLATFORM=$AWS_PLATFORM
```

It builds C binaries of (<kernel>.cpp, <kernel>.hpp, and <kernel>_tb.cpp) and the binaries are dropped into csim.

Operator Definitions

Function signatures for each of the operations below are available in the skeleton code.

```
aws s3 cp s3://mlxlo-2024-ut-austin/shared/lab3_skel.tar.gz .
```

(Task 1) Matmul:

Write a kernel that performs matrix multiplication over blocks and conforms to the API and skeleton code below. The operator must be quantized using the quantization scheme described in the following paper: Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference. Specifically, we will only use the simpler scheme of Section 2.2, but please read the entire paper because it contains important concepts that will help you. For example, you may find Equation 10 in the paper useful, though it appears after Section 2.2. The interface and description of the operator is shown below. For the report, use u=w=64, v=32.

To aid you a bit in understanding the paper, we will clarify some of the notation of Section 2.2. Specifically, the paper introduces some ambiguity with respect to the symbol M0 at the end of section 2.2. M is a scaling multiplier defined by the scheme that allows all numbers to be represented as quantized integers. M is a real (floating point) number, so will need to itself be quantized. This is where M0 comes in, but think of M0 as having two representations: $M0_{frac}$ and $M0_{int}$. The idea of M0 is to retain as much precision as possible from the real-valued M. $M0_{frac}$ is thus chosen as a scaled version of M ($M0_{frac} = 2^n*M$) such that $M0_{frac}$ is in the range [0.5, 1). It can then be quantized to retain maximum precision into the fixed-point (integer type) $M0_{int} = M0_{frac}*2^N$; N is chosen based on the precision of $M0_{int}$.

You will need to decide on the values of n and N and how much to shift the result to get the final value when implementing the kernel. Note that the intermediate values can overflow if you do not choose the correct precision (data type) for them when computing the quantized matrix using Equation 4; declare your variables wisely inside the kernel.

The function `matmul` implements matrix-matrix multiplication of the form: `out = A*B`. The shape dimensions are: `A` is [u x v], `B` is [v x w], and `out` is [u x w]. You will implement the block with quantized weights and activation (Uniform Asymmetric Quantization). The elements of a matrix are quantized into `uint8` and have a global scale and a zero point. Note that the scale (also approximated with an integer) can be computed on the host and will be passed to the kernel. The kernel returns a quantized result (`uint8_t* out`). The other inputs to the kernel are `uint32_t M0, uint32_t n, uint8_t* a, uint8_t* b, uint8_t zero_a, uint8_t zero_b, uint8_t zero _out,` which are the scale computed by the host, quantized matrices and their zero points, and `uint32_t u, uint32_t v, uint32_t w`, the shape of the matrices.

```
void matmul(
    uint8_t* out,              // quantized result for the output matrix
    const uint32_t M0,         // scale for quantization; computed on host
    const uint32_t n,          // shift for M0_frac
    const uint8_t* a,          // quantized input matrix A
    const uint8_t* b,          // quantized input matrix B
    const uint8_t  zero_a,     // zero point for A (precomputed)
    const uint8_t  zero_b,     // zero point for B (precomputed)
    const uint8_t  zero_out,   // zero point for output matrix (precomputed)
    uint32_t u,                // matrix dimension u (A is u x v)
    uint32_t v,                // matrix dimension v (B is v x w)
    uint32_t w                 // matrix dimension w (OUT is u x w)
)
```

[Note: Expected Time: T2.xlarge SW-EMU < 5mins, HW-EMU compilation < 10mins, simulation < 20 mins]

```
Test Case:
Matrix A
tensor([[ 0.0000000000, -0.5000000000,  1.0000000000, -1.5000000000],
        [ 2.0000000000, -2.5000000000,  3.0000000000, -3.5000000000],
        [ 4.0000000000, -4.5000000000,  0.0000000000, -0.5000000000],
        [ 1.0000000000, -1.5000000000,  2.0000000000, -2.5000000000]],
       dtype=torch.float64)

Matrix B
tensor([[0.0000000000, 0.5000000000, 1.0000000000, 1.5000000000],
        [2.0000000000, 2.5000000000, 3.0000000000, 3.5000000000],
        [4.0000000000, 4.5000000000, 0.0000000000, 0.5000000000],
        [1.0000000000, 1.5000000000, 2.0000000000, 2.5000000000]],
       dtype=torch.float64)

Real-Valued Product matrix G = A*B
tensor([[  1.5000000000,   1.0000000000,  -4.5000000000,  -5.0000000000],
        [  3.5000000000,   3.0000000000, -12.5000000000, -13.0000000000],
        [ -9.5000000000, -10.0000000000, -10.5000000000, -11.0000000000],
        [  2.5000000000,   2.0000000000,  -8.5000000000,  -9.0000000000]],
       dtype=torch.float64)

n  6
M0 and Zout
tensor(1249445032) tensor(201., dtype=torch.float64)
S_a, S_b, S_out
tensor(0.0333333333,        dtype=torch.float64)        tensor(0.0176470588,
dtype=torch.float64) tensor(0.0647058824, dtype=torch.float64)
Za, Zb
tensor(135., dtype=torch.float64) tensor(0., dtype=torch.float64)

matmul output (quantized)
tensor([[224., 216., 131., 123.],
        [255., 246.,   8.,   0.],
        [ 54.,  45.,  39.,  31.],
        [239., 231.,  69.,  61.]], dtype=torch.float64)

MATMUL RESULT: Dequantized Output
tensor([[  1.4882352941,   0.9705882353,  -4.5294117647,  -5.0470588235],
        [  3.4941176471,   2.9117647059, -12.4882352941, -13.0058823529],
        [ -9.5117647059, -10.0941176471, -10.4823529412, -11.0000000000],
        [  2.4588235294,   1.9411764706,  -8.5411764706,  -9.0588235294]],
       dtype=torch.float64)
```

- We encourage you to produce the same result using python first (not required).

(Task 2) Im2col:

Write functional blocks that support the im2col transformation. For the report, let `src_m=src_n=32, src_c=3, kernel_size_m=kernel_size_n=3`.

The function `im2col` implements a data transformation which "lays out" a convolution as a matrix multiplication. src_m and src_n specify the size of the a matrix (to support padding) and src_c specifies the number of input channels. kernel_size_m and kernel_size_n specify the size of the convolutional window that is passed across the A matrix as part of the transformation. The function should correctly pad if the transform size is not an even multiple of the matrix size. You can read more about this operation here: Unfold — PyTorch 2.1 documentation. Note that you do not not need to implement dilation, padding or stride. Assume dilation=1, padding=1, stride=1 to match your conv usage above.

```
void im2col(
    const uint8_t *in,
    uint8_t *out,
    uint32_t src_m,
    uint32_t src_n,
    uint32_t src_c,
    uint32_t kernel_size_m,
    uint32_t kernel_size_n
)
```

[Note: Expected Time: T2.xlarge SW-EMU < 5mins, HW-EMU compilation < 10mins, HW simulation < 20mins]

(Task 3) Once your two operator designs have been tested in HW_EMU report the following about each design:

- Resource usage (summarize in a written format – no tables)
- Latency / Throughput / Cycle Counts for different stages
- A written description of your design

Part 3: Submission Instructions

1. Submit a PDF to Gradescope.

2. Upload your code,reports,etc to the S3 folder for your team

From your instance:
Make an archive

```
aws s3 cp lab3_archive.tgz s3://mlxlo-2024-ut-austin/work/mlxlo-2024-team<TEAM_NUMBER>/
```