Chair for Computer Architecture and Parallel Systems
School of Computation, Information and Technology
Technical University of Munich

Lab 2: Integrating CNN Layers

Practical Course: Accelerating CNNs using PL

05.05.2023

1 Exercise

After Implementing the different layers in the last lab it is now time to connect the layers together and perform the first classifications. First copy your implementation of the last lab into the current lab folder, replacing the `kernels.cpp` file, afterwards complete the `Tensor * CNN::inference(Tensor *)` method of the CNN class and run the different tests, and measure the time spent in each of the layers.
To complete the lab implement the inference function such that all tests pass, in addition measure the time spent in each layer of the Convolutional Neural Network.
Your CNN implementation should now be able to perform inference for any CNN containing the in the previous lab described layers with some conditions:
The Softmax Layer is only used as the last layer and after a Linear Layer no convolutional or Pooling layers are used. The information of the different layers are stored as a structure in the layers vector, which contain the type, sequence and configurations of the different layers. The weights and biases are loaded from the pytorch model and read using the same functions as in the previous lab. Run `gentest.py` to generate the test cases and if you are using windows don't forget to change target in the Makefile to `lab2.exe` from `lab2.bin`.

2 Test

To test the correctness of your implementation you should run the `lab2.cpp` file, where the test cases are generated once again using pytorch. The Lab comes with 7 pre-defined networks that you can use to test your algorithm, as well as to perform inference. The net `testNet` is only for testing purposes and has no real application purposes. The four networks `smallNet`, `mediumNet`, `largeNet` and `giantNet` are the CNNs that are going to be used for benchmarking the Software performance and implemented on Hardware later in the course. They take as an input a 3-channel 128×128 px RGB image and classify it into 100 different classes, the 4 CNNs for this lab are not pre-trained and can't predict anything at the moment.

The other two networks VGG11 and VGG16 are pre-trained networks using the ImageNet (ILSVRC2012) dataset and can be used to classify images into 1000 different classes. They both take `3x224x224` Tensors as input, for example a 3-channel RGB image you can test the effectiveness of the networks using the example images provided or input your own images and see what it predicts. The file `convertImg.py` in the utils folder converts an image to a tensor that can be read inside the `lab2.cpp` file. To classify an image pass 1 followed by the file name to the executable `$./lab2.bin 1 image.tensor`

You can find the vectors defining the different models in the `nets.cpp` file in the utils folder. The tests can be run by passing the value 0 when calling the function `$./lab2.bin 0$`.

3 Timing the Layers

To optimize an algorithm it is crucial to know, where most of the computational time is spent and then focus on optimizing those part of the algorithm. To time the layers you can use the functions `mtick()` and `mtock()` and analyze where the most time is spent. The `print_timing()` method prints the execution times stored in the runtime array, as can be seen in the main file. To benchmark the performance we don't have to check the correctness of the outputs, this can be chosen by passing the value 2 to the function `$.lab2.bin 2`.

4 Layer Structure

The Layer structure vector contains the information about the different layers used. Loop through the vector and perform the operation according to the layer type . The important attributes of the Structure during inference are the pointers to the Output Tensor (Z), the Weight Tensor array (`W`) and the bias Tensor (`B`). The structure also contain information such as `output_size` (Size of Z),`kernel_width` (Size of W), `input_channels` (`size[0]` of previous layers Z). If the Layer does not have any weights, the weight and bias pointer are `NULL`.
The value `in_place` chooses whether the `ReLU` layer stores its result to a new Tensor or modifies the input Tensor (The pointer Z is automatically set to the previous layer Z). The attribute `pad` stores the size of the zero-padding, **to perform zero padding you can use the** `padTensor()` function that returns a new padded Tensor (declared in file `tensor.h`). Don't forget to free the Tensor after usage using `delete`.

```
enum struct Layer_Type : uint32_t {Linear = 0,Pool = 1 ,
  ReLU = 2,Conv = 3,Softmax = 4};

typedef struct{
  Layer_Type type;
  uint32_t output_size[3];
  uint32_t kernel_width;
  uint32_t input_channels;
  uint32_t pad;
  bool in_place = false;
  Tensor * Z;
  Tensor * W;
  Tensor * B;
} CNN_layer_struct;
```