ECE 459: Programming for Performance
Assignment 3

Zahin Mohammad

March 17, 2021

1 CNN and GPU

The following sections will discuss how the Kernel and host code work to perform for a convolutional neural network.

1.1 Kernel

The kernel code is split into two layers:

- Convolution + ReLU
- Output

The convolution layer kernel code performs computes the dot product with a $5x5$ filter for a single section of the input image at a time. This can be done because the dot products are all calculated in parallel. The kernel has a size of $10x20x20$ (same size as output),to easily coordinate between blocks and grids on which dot product to compute.

```
extern "C" __global__ void ConvolutionLayerAndReLU(
    const double input[INPUT_DIM][INPUT_DIM],
    const double conv_layer[CONV_LAYER_SIZE][FILTER_DIM][FILTER_DIM],
    double out[CONV_LAYER_SIZE][CONV_OUT_DIM][CONV_OUT_DIM])
{
    int filter_index = blockIdx.x;
    int filter_row_index = threadIdx.x;
    int filter_col_index = threadIdx.y;

    double prod = 0.0;
    #pragma unroll
    for (int y = 0; y < FILTER_DIM; y++)
        #pragma unroll
        for (int x = 0; x < FILTER_DIM; x++)
            prod += input[filter_row_index*FILTER_DIM + y][filter_col_index*FILTER_DIM + x]

    // ReLU
    prod = fmax(prod, 0.0);
```

```
    out[filter_index][filter_row_index][filter_col_index] = prod;
}
```

The output layer has a kernel size of $10x32$ but needs to compute the dot product of a 4000 length vector 10 times. Instead of a $10x4000$ kernel size, it was found it was better to have a $10x32$ kernel size and have the kernel perform the partial dot product batched in groups of $\frac{4000}{32} = 125$. As threads will be modifying the same address due to the partial dot product, the updates are made using the *atomicAdd* function, which will perform an atomic add.

```
extern "C" __global__ void OutputLayer(
    const double input[OUT_NEURON_DIM],
    const double output_layer[OUT_LAYER_SIZE][OUT_NEURON_DIM],
    double out[OUT_LAYER_SIZE])
{
    int out_index = blockIdx.x;
    int layer_index = threadIdx.x;
    double prod = 0.0;
    #pragma unroll
    for (int j= 0; j<125; j++){
        int i = layer_index*125 + j;
        prod += input[i]*output_layer[out_index][i];
    }
    atomicAdd(&out[out_index], prod);
}
```

1.2 Host Code

The host code can be described in 3 sections:

- Initialization
- Buffers
- Launching Kernel

The host code is largely based off the example RustCuda code present in lectures 22 and 23, with the primary difference being how the work items are split.

The grid size for the convolution layer (with ReLU) is 10, and the the block size is $5x5$. This decision was made to correspond to the output size of the convolution layer which is $10x20x20$.

The grid size for the output layer is 10, and the block size is 32. Through testing it was found that a block size of 32 works best as the input is of size 40000 which is divisible by 32. The algorithm for the output layer is further described in the kernel section of the report.

```
pub fn init(cnn: &Cnn) -> Result<Self, Box<dyn Error>> {
    rustacuda::init(CudaFlags::empty())?;
    let device = Device::get_device(0)?;
    let ctx = Context::create_and_push(ContextFlags::MAP_HOST | ContextFlags::SCHED_AUTO, d
    let ptx = CString::new(include_str!("../kernel/kernel.ptx"))?;
    let cuda_context = CudaContext{
```

```
        conv_layer: DeviceBox::new(&cnn.conv_layer).unwrap(),
        output_layer: DeviceBox::new(&cnn.output_layer).unwrap(),
        module: Module::load_from_string(&ptx)?,
        stream: Stream::new(StreamFlags::NON_BLOCKING, None)?,
        _context: ctx,
    };
    Ok(cuda_context)
}

pub fn compute(&mut self, input: &InputMatrix) -> Result<OutputVec, Box<dyn Error>> {
    // Initialize input/output buffers
    let mut input = DeviceBox::new(input).unwrap();
    let mut _conv_output = DeviceBox::new(&[[[0.0; CONV_OUT_DIM]; CONV_OUT_DIM]; CONV_LAYER
    let mut output = OutputVec([0.0; OUT_LAYER_SIZE]);
    let mut _output = DeviceBox::new(&output)?;
    let block_size = BlockSize::xy(CONV_OUT_DIM as u32, CONV_OUT_DIM as u32);
    let stream = &self.stream;
    let module = &self.module;

    unsafe {
        // Layer 1: Convolution Layer + ReLu
        let _ = launch!(module.ConvolutionLayerAndReLU<<<CONV_LAYER_SIZE as u32, block_size
            input.as_device_ptr(),
            self.conv_layer.as_device_ptr(),
            _conv_output.as_device_ptr()
        ));
        // Layer 2: Output Layer
        let _ = launch!(module.OutputLayer<<<OUT_LAYER_SIZE as u32, 32 as u32, 0, stream>>>
            _conv_output.as_device_ptr(),
            self.output_layer.as_device_ptr(),
            _output.as_device_ptr()
        ));
    }
    // Wait for Kernel to finish executing
    self.stream.synchronize()?;
    _output.copy_to(&mut output)?;
    Ok(output)
}
```

2 GPU vs CPU

When comparing the Cuda code to the CPU code, it was found that the Cuda code is ≈ 3 times faster than the cpu code. This is due to the Cuda code computing many dot products in parallel, enough to offset the performance hit from transferring data to and from the GPU.

```
    Finished release [optimized] target(s) in 0.05s
     Running `target/release/lab3 cuda input/cnn.csv input/in.csv output/out_cuda.csv`
```

```
24674 microseconds of actual work done
    Finished release [optimized] target(s) in 0.04s
     Running `target/release/lab3 cpu input/cnn.csv input/in.csv output/out.csv`
70555 microseconds of actual work done
Comparison finished
```