ECE 459 - Lab 3

Jackson Guo
j84guo@uwaterloo.ca

1 Kernels

In “cuda” mode, forward propagation of one image through the neural network is accomplished by two CUDA kernels.

1.1 2D Convolution and ReLU

The first kernel accepts one 100x100 image and performs 2D convolution by passing ten 5x5 filters over the image with a stride of 5 - the output has dimensions 10x20x20. Each output element is then passed through a ReLU function, which zeroes any negative values.

```
extern "C" __global__ void conv2d_relu(// 100x100
                                       const double *image,
                                      // 10x5x5
                                       const double *filters,
                                      // 10x20x20
                                       double *output)
```

The kernel accepts three tensors: a 100x100 image, a 10x5x5 set of filters and a 10x20x20 output. As with all tensors/grids/matrices indexed in this report and associated code, each of these parameters is laid out in row-major order and elements are of type double. The image and filters are read-only, while the output tensor can be written to.

When the kernel is launched, each thread is responsible for computing one or more of the 10x20x20=4000 output elements. If there are more threads than output elements, each thread computes at most one - but if there are less threads than output elements, the threads determine which ones they’re responsible for via a grid-stride loop.

```
// Index into flattened output array
for (int output_i = blockIdx.x * blockDim.x + threadIdx.x;
    output_i < 4000;
    output_i += gridDim.x * blockDim.x) {
   double sum = 0;
   // ...COMPUTE SUM HERE...
   output[output_i] = sum;
}
```

In the grid-stride loop, the thread uses `output_i` as an index into the output array - this is initialized to the thread’s index in the flattened grid

```
output_i = blockIdx.x * blockDim.x + threadIdx.x
```

and incremented by the number of threads in the grid.

```
output_i += gridDim.x * blockDim.x
```

To compute an output, the sum must first find the filter it is going to use based on the output index.

```
// 5x5 filter
const int filter_num = output_i / 400;
const double *filter = filters + filter_num * 5 * 5;
```

Then, it must determine the top-left corner of the 5x5 tile in the image that it will position the filter over - this is done by computing the row and column of the output value in the current filter's 20x20 output and multiplying those by five.

```
// 20x20 filter output
const int filter_output_i = output_i % 400;
const int filter_output_r = filter_output_i / 20;
const int filter_output_c = filter_output_i % 20;

// Top-left row, col in 5x5 image tile corresponding to filter output
const int image_r = filter_output_r * 5;
const int image_c = filter_output_c * 5;
```

The sum resulting from the current filter being positioned on the 5x5 tile described by the coordinates `image_r, image_c` is computed using two nested loops.

```
// Accumulate sum
double sum = 0;
for (int r = 0; r < 5; r++) {
   for (int c = 0; c < 5; c++) {
       const int i = (image_r + r) * 100 + (image_c + c);
       const int j = r * 5 + c;
       sum += image[i] * filter[j];
   }
}
```

Finally, ReLU is applied and the resulting value is written to the output array.

```
if (sum < 0) {
   sum = 0;
}
output[output_i] = sum;
```

The kernel is launched with 1024 blocks, each containing 32 threads. The reasoning for using many blocks is to ensure that each of the GPU's symmetric multiprocessors (SM), which are assigned blocks of threads to execute, are kept busy. Each block has a multiple of 32 (in this case, exactly 32) threads since each of an SM's cores can execute instructions from a 32-thread warp simultaneously in single-instruction multiple-thread (SIMT) fashion. The stream is synchronized so that the kernel completes before the next is launched.

```
launch!(module.conv2d_relu<<<1024, 32, 0, stream>>>(
   device_input.as_device_ptr(),
   self.conv_layer.as_device_ptr(),
   device_conv2d_output.as_device_ptr()
))?;
self.stream.synchronize()?;
```

1.2 Vector-Matrix Multiplication

1.2.1 Parallelism and Caching

Next, the 10x20x20 output of the ReLU layer is treated as a flattened 1x4000 vector and multiplied by a 4000x10 matrix of weights, producing the 1x10 output layer. Although the weight matrix is logically 4000x10, it is laid out in memory as 10x4000 (transposed) in the sample code, presumably to improve data locality when iterating sequentially through the weights of each output neuron.

To compute the vector-matrix product with low latency, several levels of parallelism in the problem are exploited. Furthermore (in point 3), on-chip shared memory is leveraged to reduce the overhead of accessing slower global device memory.

1. First, each dot product between a logical column of the weight matrix (weight vector) and the ReLU output (ReLU vector) can be computed independently, in parallel.
2. Second, in each dot product, corresponding pairs of elements between the two vectors can be multiplied together independently, in parallel, before being summed.
3. Third, in each dot product, the summation step can itself be done in parallel using the parallel reduction programming pattern.
    a. When implementing parallel reduction using CUDA, a block of threads is responsible for reducing a sub-array of a large linear array.
    b. While a single thread performing reduction over n contiguous elements would require an O(n) step array traversal, a block of n threads can perform the reduction in O(log n) steps by reducing the sub-array into half at each step.
        i. In one time step, each of the n threads reduces (zero or) one pair of elements, which is what allows the array to be reduced into half each time.
    c. Since there are only ten weight vectors (less than the number of SM's on the GPU), **multiple** thread blocks are used to reduce each weight vector - each will

be responsible for one or more segments (i.e. subarrays) of the weight vector and each reduced segment will be combined at the end by host code.

d. The segment over some index range [i, j] that a block is responsible for is an array containing the products of pairs elements from the weight and ReLU vectors corresponding to those indices, i.e. segment[i] = weights[i] * relu[i].
    i. Since parallel reduction requires frequent read/write access to the segment in memory, each block will cache its segment in shared memory. Shared memory is on-chip and allocated per thread-block - it is several orders of magnitude faster than the device's global memory.

1.2.1 Code

This kernel accepts the ReLU layer's output (conv2d_output), the weights (weights), and scratch space for each block to record the reduction result of its segments (block_outputs).

```
extern "C" __global__ void flattened_to_dense(// 10x20x20
                                               const double *conv2d_output,
                                              // 10x4000
                                               const double *weights,
                                              // 10xgridDim.x
                                               double *block_outputs)
```

Shared memory is statically allocated with a size of 4000 doubles - each block of threads will have its own shared memory array and will likely not have to use all 4000 slots, only the subarray from zero with length equal to the segment length. This shared memory array could have been allocated dynamically on kernel launch, but I coded it statically for convenience, knowing that no segment would be larger than 4000.

```
// Cache, it is assumed that 4000 >= blockDim.x
__shared__ double x[4000];
```

The blocks are launched in a 2D grid, where gridDim.x is the number of blocks assigned to each weight vector and gridDim.y is ten - each row (fixed y-coordinate) contains the blocks for the weight vector associated with a particular output neuron. As before, the stream is synchronized so that the results are completely computed before the host code tries to retrieve them.

```
launch!(module.flattened_to_dense<<<(NUM_BLOCKS_PER_OUTPUT_NEURON,
OUT_LAYER_SIZE as u32), 256, 0, stream>>>(
   device_conv2d_output.as_device_ptr(),
   self.output_layer.as_device_ptr(),
   device_block_outputs.as_device_ptr(),
   NUM_BLOCKS_PER_OUTPUT_NEURON
))?;
self.stream.synchronize()?;
```

Thus, threads in a block can determine which weight vector they are reducing using the block's y-index.

```
// Output neuron
const int output_neuron = blockIdx.y;
```

Likewise, threads in a block determine the segment they are responsible for using the block's x-index; an outer loop iterates through the segments a block is responsible for. Note that the segment length is at most the number of threads in the block.

```
// For each segment of the row that this block is responsible for
for (int segment_start = blockIdx.x * blockDim.x; segment_start < 4000;
segment_start += gridDim.x * blockDim.x) {
      // ...REDUCE SEGMENT...
}
```

Before a block starts reducing a segment, its threads must first read the pairs of weight and ReLU output values into shared memory. For each (weight, ReLU output) pair, their product is computed and stored in shared memory. All elements of the segment must be stored into shared memory before reduction, which is why a __syncthreads() barrier is used after the global memory accesses and multiplication.

```
// Populate cache with this segment
const int segment_len = min(blockDim.x, 4000 - segment_start);
if (threadIdx.x < segment_len) {
   x[threadIdx.x] = weights[output_neuron * 4000 + segment_start + threadIdx.x]
* conv2d_output[segment_start + threadIdx.x];
}
__syncthreads();
```

Reduction of the current segment is implemented by an inner loop which repeatedly reduces the segment by half at each step. Note that a barrier is used at the end of each iteration of the inner loop, to ensure that the reduction from n to n/2 is complete before going from n/2 to n/4, and so on.

```
double *segment = x;
int right = segment_len - 1;
while (0 < right) {
   const int mid = right / 2;
   if (((right + 1) % 2) == 0) {
       if (threadIdx.x <= mid) {
           segment[threadIdx.x] += segment[right - threadIdx.x];
       }
   } else {
       if (threadIdx.x < mid) {
           segment[threadIdx.x] += segment[right - threadIdx.x];
       }
```

```
    }
    right = mid;
    __syncthreads();
}
```

At the end of the current segment's reduction, the result is stored in segment[0]. The thread with index 0 in the block is responsible for adding the segment's reduction result to the block scratch space - each block has one double in the scratch space to combine the results of each segment that it reduced. Note that a barrier is used at the end of each segment's reduction (end of each iteration of the outer loop), so that one segment is entirely reduced before the block moves on to reducing another one.

```
if (threadIdx.x == 0) {
    const int block_outputs_i = output_neuron * num_blocks_per_output_neuron +
blockIdx.x;
    block_outputs[block_outputs_i] += segment[0];
}
__syncthreads();
```

In host code, the results for the blocks that were responsible for each weight vector are reduced one more time, producing the final 1x10 output vector.

```
let mut block_outputs = [[0f64; NUM_BLOCKS_PER_OUTPUT_NEURON as usize];
OUT_LAYER_SIZE];
device_block_outputs.copy_to(&mut block_outputs)?;

let mut output = OutputVec{0: [0f64; OUT_LAYER_SIZE]};
for i in 0usize..OUT_LAYER_SIZE as usize {
    for j in 0usize..NUM_BLOCKS_PER_OUTPUT_NEURON as usize {
        output.0[i] += block_outputs[i][j];
    }
}
Ok(output)
```

2 Host and Device Code

2.1 CUDA Context, Device, Stream

A structure was provided in the sample code which holds all information required to launch kernels and obtain their results.

```
pub struct CudaContext {
    conv_layer: DeviceBox<ConvLayer>,
    output_layer: DeviceBox<OutputLayer>,
    module: Module,
    stream: Stream,
```

```
    _context: Context,
}
```

On initialization, the underlying CUDA driver API is set up, a handle to a GPU device is obtained, and a CUDA context is created containing all the necessary state for working with that device. A CUDA stream is also created, in which operations like kernel launches and host-device/device-host memory transfers will be enqueued.

```
rustacuda::init(CudaFlags::empty())?;
let device = Device::get_device(0)?;
let _context = Context::create_and_push(ContextFlags::MAP_HOST |
ContextFlags::SCHED_AUTO, device)?;
let stream = Stream::new(StreamFlags::NON_BLOCKING, None)?;
```

2.2 PTX Object File, Rustacuda Module

The kernels' source code is contained in a separate CUDA C/C++ file called **kernel/kernel.cu** - this is compiled by **nvcc** through a Cargo build script called **build.rs** as a pre-build step. The resulting PTX object file is embedded into the host source code as a static string literal using the include_str! macro, and Rustacuda builds a module object from the string. The Rustacuda module is subsequently used for kernel launches.

```
let ptx = CString::new(include_str!("../kernel/kernel.ptx"))?;
let module = Module::load_from_string(&ptx)?;
```

2.3 Host-Device Memory Transfers

Before launching a kernel, memory must be allocated on the GPU (device memory) to hold the tensors the kernel needs. Rustacuda's DeviceBox and DeviceBuffer can each be constructed with a reference to a tensor in host memory. Device memory allocations are made using the underlying CUDA driver (e.g. cudaMalloc) which have the same size as the tensors in host memory, and the host memory's contents are copied to the device to initialize the new device memory.

```
let mut device_input = DeviceBox::new(input)?;
let mut device_conv2d_output = DeviceBox::new(&ConvOutput{0: [[[0f64;
CONV_OUT_DIM]; CONV_OUT_DIM]; CONV_LAYER_SIZE]})?;

const NUM_BLOCKS_PER_OUTPUT_NEURON: u32 = 8;
let mut device_block_outputs = DeviceBuffer::from_slice(&[[0f64;
NUM_BLOCKS_PER_OUTPUT_NEURON as usize]; OUT_LAYER_SIZE])?;
```

2.3.1 DeviceCopy trait, newtype Pattern

Note that types which are to be copied to the GPU by Rustacuda must have the DeviceCopy trait - from the docs, this marker trait indicates that the type "can be duplicated simply by

copying bits" and "does not contain a reference to memory which is not accessible to the device". These traits are implemented by the sample code in **src/cnn.rs**.

Note also that the device copyable types provided by the sample code (e.g. ConvLayer) are tuple structures implementing the newtype pattern - a way to provide extra type safety by indicating the purpose of a variable in its type. The tuple structs do not change the memory layout of their contents thanks to the use of `#[repr(transparent)]` - hence, kernels can simply accept input tensors as pointers.
.

2.4 Asynchronous Kernel Launch, Stream Synchronization

Since kernel launches are asynchronous, the stream they are launched on is synchronized to ensure the kernel completes before moving on. Launching is done in an unsafe block since a kernel launch is invoking non-Rust code - the Rust compiler cannot make its usual safety guarantees about code in the PTX object file.

```
unsafe {
    launch!(module.conv2d_relu<<<1024, 32, 0, stream>>>(
        device_input.as_device_ptr(),
        self.conv_layer.as_device_ptr(),
        device_conv2d_output.as_device_ptr()
    ))?;
}
self.stream.synchronize()?;
```

2.5 Device-Host Memory Transfers

Finally, output data computed by the kernels on the device need to be copied back to the host once the kernel is done.

```
let mut block_outputs = [[0f64; NUM_BLOCKS_PER_OUTPUT_NEURON as usize];
OUT_LAYER_SIZE];
device_block_outputs.copy_to(&mut block_outputs)?;
```

As mentioned in section 1, the host code performs a final reduction of the vector-matrix multiplication kernel's output.