ECE 408 Final Project

School: UIUC

Team Name: Shrek

Team: Alex Fan (`zhenfan3`), Brandon Van (`jbvan2`), Joshua Song (`jssong3`)

Milestone 1

Top 10 time consuming kernels:

1. `volta_scudnn_128x32_relu_interior_nn_v1`
2. `void cudnn::detail::implicit_convolve_sgemm<float, float, int=1024, int=5, int=5, int=3, int=3, int=3, int=1, bool=1, bool=0, bool=1>(int, int, int, float const *, int, float*, cudnn::detail::implicit_convolve_sgemm<float, float, int=1024, int=5, int=5, int=3, int=3, int=3, int=1, bool=1, bool=0, bool=1>*, kernel_conv_params, int, float, float, int, float, float, int, int)`
3. `volta_sgemm_128x128_tn`
4. `void cudnn::detail::activation_fw_4d_kernel<float, float, int=128, int=1, int=4, cudnn::detail::tanh_func<float>>(cudnnTensorStruct, float const *, cudnn::detail::activation_fw_4d_kernel<float, float, int=128, int=1, int=4, cudnn::detail::tanh_func<float>>, cudnnTensorStruct*, float, cudnnTensorStruct*, int, cudnnTensorStruct*)`
5. `void cudnn::detail::pooling_fw_4d_kernel<float, float, cudnn::detail::maxpooling_func<float, cudnnNanPropagation_t=0>, int=0, bool=0>(cudnnTensorStruct, float const *, cudnn::detail::pooling_fw_4d_kernel<float, float, cudnn::detail::maxpooling_func<float, cudnnNanPropagation_t=0>, int=0, bool=0>, cudnnTensorStruct*, cudnnPoolingStruct, float, cudnnPoolingStruct, int, cudnn::reduced_divisor, float)`
6. `void mshadow::cuda::MapPlanLargeKernel<mshadow::sv::saveto, int=8, int=1024, mshadow::expr::Plan<mshadow::Tensor<mshadow::gpu, int=2, float>, float>, mshadow::expr::Plan<mshadow::expr::ScalarExp<float>, float>>(mshadow::gpu, unsigned int, mshadow::Shape<int=2>, int=2, int)`
7. `void mshadow::cuda::SoftmaxKernel<int=8, float, mshadow::expr::Plan<mshadow::Tensor<mshadow::gpu, int=2, float>, float>, mshadow::expr::Plan<mshadow::Tensor<mshadow::gpu, int=2, float>, float>>(mshadow::gpu, int=2, unsigned int)`
8. `void mshadow::cuda::MapPlanKernel<mshadow::sv::saveto, int=8, mshadow::expr::Plan<mshadow::Tensor<mshadow::gpu, int=2, float>, float>, mshadow::expr::Plan<mshadow::expr::ScalarExp<float>, float>>(mshadow::gpu, unsigned int, mshadow::Shape<int=2>, int=2)`
9. `Volta_sgemm_32x32_sliced1x4_tn`
10. `void mshadow::cuda::MapPlanKernel<mshadow::sv::plusto, int=8, mshadow::expr::Plan<mshadow::Tensor<mshadow::gpu, int=2, float>, float>, mshadow::expr::Plan<mshadow::expr::Broadcast1DExp<mshadow::Tensor<mshadow::gpu, int=1, float>, float, int=2, int=1>, float>>(mshadow::gpu, unsigned int, mshadow::Shape<int=2>, int=2)`

Top 10 time-consuming API calls:

1. `cudaMemGetInfo`
2. `cudaFree`
3. `cudaFuncSetAttribute`
4. `cudaMemcpy2DAsync`
5. `cudaStreamSynchronize`
6. `cudaMalloc`
7. `cudaGetDeviceProperties`
8. `cuDeviceGetAttribute`
9. `cudaEventCreate`
10. `cudaEventCreateWithFlags`

Difference between kernels and API calls:

The API calls are the CUDA calls that provide an extension to the C language. They facilitate configuration of the parallel computing device - actions include allocation of memory and transfer of data to and from. A kernel function is code intended to run on the parallel device. Upon calling, it launches multiple threads to process different parts of the data in parallel.

Running MXNet on the CPU:

Output:

```
✱ Running /usr/bin/time python m1.1.py
Loading fashion-mnist data... done
Loading model... done
New Inference
EvalMetric: {'accuracy': 0.8177}
```

Run time:

```
20.95user 6.05system 0:14.19elapsed 190%CPU (0avgtext+0avgdata 5954620maxresident)k
0inputs+2856outputs (0major+1580062minor)pagefaults 0swaps
```

Running MXNet on the GPU:

Output:

```
✱ Running /usr/bin/time python m1.2.py
Loading fashion-mnist data... done
Loading model... done
New Inference
EvalMetric: {'accuracy': 0.8177}
```

Run time:

```
4.24user 2.57system 0:04.62elapsed 147%CPU (0avgtext+0avgdata 2846512maxresident)k
0inputs+4568outputs (0major+706410minor)pagefaults 0swaps
```

Milestone 2

Output:

```
✱ Running /usr/bin/time python m2.1.py 10000
Loading fashion-mnist data... done
Loading model... done
New Inference
Op Time: 26.134832
Op Time: 154.410258
Correctness: 0.8171 Model: ece408
191.63user 6.42system 3:05.15elapsed 106%CPU (0avgtext+0avgdata 5953104maxresident)k
0inputs+2856outputs (0major+2264713minor)pagefaults 0swaps
```

Op Times:

```
Op Time: 26.134832
Op Time: 154.410258
```

Execution Time:

```
191.63user 6.42system 3:05.15elapsed 106%CPU (0avgtext+0avgdata 5953104maxresident)k
0inputs+2856outputs (0major+2264713minor)pagefaults 0swaps
```

Milestone 3

Output:

Number of Images: 100

```
	Loading fashion-mnist data... done
	Loading model... done
	New Inference
	Op Time: 0.000651
	Op Time: 0.001620
	Correctness: 0.85 Model: ece408
```

Number of Images: 1000

```
	Loading fashion-mnist data... done
	Loading model... done
	New Inference
	Op Time: 0.006198
	Op Time: 0.016068
	Correctness: 0.827 Model: ece408
```

Number of Images: 10000

```
	Loading fashion-mnist data... done
	Loading model... done
	New Inference
	Op Time: 0.064318
	Op Time: 0.147558
	Correctness: 0.8171 Model: ece408
```

Performance:

According to `forward1_analysis.nvprof` and `forward2_analysis.nvprof`,

- The performance of the kernel is most likely limited by the latency of arithmetic or memory operations;
- There are low global memory load efficiency and low warp execution issues;
- Global memory load and store are not properly aligned, which leads to inefficient use of memory bandwidth;
- Divergence branches lower warp execution efficiency, which leads to inefficient use of the GPU’s compute resources.

Milestone 4

Optimization 1: Unroll/Shared-Memory Matrix Multiply

Description:
Reduce the original convolutional layer by unrolling inputs such that all elements needed for each output element are stored as a sequential block; the resulting operation consists of one matrix multiplication.

Times & Correctness (10000 images):
Op Time: 0.059692
Op Time: 0.140799
Correctness: 0.8171 Model: ece408

Performance Analysis:
A significant portion of the execution time is spent doing the matrix multiply operation, which can be drastically improved using different tiling schemes. In the matrix multiply kernel, we have a 69.7% shared memory efficiency and we see that busy pipelines result in 41.1% of stalling. This is due to certain compute units not being available which causes a performance bottleneck, limiting the execution times that we can achieve.

Optimization 2: Weight matrix (kernel values) in constant memory

Description:
Move the weight matrix, which is accessed many times, to constant memory in order to improve memory read times. Constant memory has better performance than global memory but worse performance when compared to shared memory or register file.

Times & Correctness (10000 images):
Op Time: 0.174848
Op Time: 0.419835
Correctness: 0.8171 Model: ece408

Performance Analysis:
Analyzing this optimization, we can see that a large portion of the computation time is spent on matrix multiplication (GEMM) which is expected behavior. When we look at the issue stall reasons, they are primarily split between execution dependencies and data requests. This means that we can lower the execution times by improving instruction parallelism by possibly using shared memory or different memory access patterns. In addition, we see that the `memcpy` operations are not efficient and require a large portion of the time compared to the actual kernel executions.

Optimization 3: Shared Memory convolution

Description:
Make parts of the image data, which is needed for convolutions, loaded into the shared memories first, so that the threads in a block pull the data from the same shared memory. Reading from shared memories is faster than from global memories. Since it involves lots of reuses of data within a block of threads, putting them into shared memories will make it faster than reading from global memories every time.

Time & Correctness (10000 images):
Op Time: 0.059888
Op Time: 0.137068
Correctness: 0.8171 Model: ece408

Performance Analysis:
For this optimization, we see that around 90% of the compute time is spent by the convolution kernel. When we analyze the performance of this kernel alone, there is a low level of kernel concurrency. This leads us to believe that our memory access patterns or other operation patterns are causing stalls which limit the extent to which we are able to run in parallel. In addition, the profiler tells us that a good portion of the execution time is spend on memcpy operations.