ECE408-report

Group Member: yutao2, jiey3, fz7

Baseline Results:

1. M1.1:

We got the following output:

```
Loading fashion-mnist data... done
Loading model... done
EvalMetric: {'accuracy': 0.8673}
9.99user 4.17system 0:05.52elapsed 256%CPU (0avgtext+0avgdata 1633608maxresident)k
0inputs+2624outputs (0major+27585minor)pagefaults 0swaps
```

The elapsed time of the whole python program is 0:05.52.

2. M1.2:

We got the following output:

```
Loading fashion-mnist data... done
Loading model...[23:45:37] src/operator/././cudnn_algoreg-inl.h:112: Running performance tests
to find the best convolution algorithm, this can take a while... (setting env variable
MXNET_CUDNN_AUTOTUNE_DEFAULT to 0 to disable)
done
EvalMetric: {'accuracy': 0.8673}
1.86user 1.01system 0:08.19elapsed 35%CPU (0avgtext+0avgdata 908356maxresident)k
331064inputs+3136outputs (1199major+156292minor)pagefaults 0swaps
```

The accuracy is 0.8673, the same as that in M1.1.

The elapsed time of the whole python program is 0:08.19.

3. M1.3:

Part of the results are:

308 Profiling result:

Time(%) Time Calls Avg Min Max Name

37.07% 50.402ms 1 50.402ms 50.402ms 50.402ms void cudnn::detail::implicit_convolve_sgemm

28.85% 39.234ms 1 39.234ms 39.234ms 39.234ms sgemm_sm35*ldg_tn*128x8x256x16x32

14.25% 19.374ms 2 9.6869ms 460.86us 18.913ms void cudnn::detail::activation_fw_4d_kernel

10.66% 14.492ms 1 14.492ms 14.492ms 14.492ms void cudnn::detail::pooling_fw_4d_kernel

308 API calls:

Time(%) Time Calls Avg Min Max Name

44.17% 1.84312s 18 102.40ms 18.424us 921.40ms cudaStreamCreateWithFlags

27.33% 1.14035s 10 114.03ms 645ns 321.98ms cudaFree**

24.85 % 1.03694s 24 43.206ms 238.22us 1.02980s cudaMemGetInfo

These are the most time-comsuming kernels or API calls. The GPU spends most of its time on the convolution kernel and stream creating.

4. M2.1:

The output is:

```
Loading fashion-mnist data... done
Loading model... done
Op Time: 9.116229
Correctness: 0.8562 Model: ece408-high
```

For the ece408-low model with data size of 10000:

```
* Running python m2.1.py ece408-low 10000
New Inference
Loading fashion-mnist data... done
Loading model... done
Op Time: 9.769963
Correctness: 0.629 Model: ece408-low
```

For the ece408-high model with data size of 10000:

```
* Running python m2.1.py ece408-high 10000
New Inference
Loading fashion-mnist data... done
Loading model... done
Op Time: 9.058432
Correctness: 0.8562 Model: ece408-high
```

The implementation has the expected correctness.

yutao2: build the CPU implementation.

jiey3: move on to explore basic GPU implementation.

fz7: move on to explore basic GPU implementation.

5. M3.1:

Simple matrix multiplication is performed here. The significant parts in NVPROF are listed below.

For the ece408-high model with data size of 10000:

```
==313== NVPROF is profiling process 313, command: python m3.1.py
Loading model... done
Op Time: 0.503840
Correctness: 0.8562 Model: ece408-high
==313== Profiling application: python m3.1.py
==313== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max       Name
85.45%       503.73ms 1           503.73ms  503.73ms  503.73ms  void
mxnet::op::forward_kernel<mshadow::gpu, float>(float*, mxnet::op::forward_kernel<mshadow::gpu,
float> const *, mxnet::op::forward_kernel<mshadow::gpu, float> const , int, int, int, int, int,
int)
6.48%        38.177ms 1           38.177ms  38.177ms  38.177ms  sgemm_sm35ldg_tn128x8x256x16x3
3.29%        19.372ms 2           9.6862ms  454.14us  18.918ms  void
cudnn::detail::activation_fw4d_kernel<float, float, int=128, int=1, int=4,
cudnn::detail::tanh_func<float>>(cudnnTensorStruct, float const *,
cudnn::detail::activation_fw4d_kernel<float, float, int=128, int=1, int=4,
cudnn::detail::tanh_func<float>>, cudnnTensorStruct, float, cudnnTensorStruct, int,
cudnnTensorStruct*)
2.44%        14.386ms 1           14.386ms   14.386ms   14.386ms  void
cudnn::detail::pooling_fw4d_kernel<float, float, cudnn::detail::maxpooling_func<float,
cudnnNanPropagation_t=0>, int=0>(cudnnTensorStruct, float const *,
cudnn::detail::pooling_fw4d_kernel<float, float, cudnn::detail::maxpooling_func<float,
cudnnNanPropagation_t=0>, int=0>, cudnnTensorStruct*, cudnnPoolingStruct, float,
cudnnPoolingStruct, int, cudnn::reduced_divisor, float)
...
==313== API calls:
Time(%)      Time     Calls       Avg       Min       Max  Name
 42.05%  1.95477s        18  108.60ms  17.474us  977.02ms  cudaStreamCreateWithFlags
 26.25%  1.22033s        10  122.03ms     889ns  344.22ms  cudaFree
 18.68%  868.47ms        23  37.760ms  235.02us  861.78ms  cudaMemGetInfo
 10.84%  503.74ms         1  503.74ms  503.74ms  503.74ms  cudaDeviceSynchronize
  1.67%  77.804ms        25  3.1121ms  5.2740us  41.741ms  cudaStreamSynchronize
  0.25%  11.525ms         8  1.4406ms  8.2440us  5.5929ms  cudaMemcpy2DAsync
  0.14%  6.3779ms        41  155.56us  9.5740us  1.0945ms  cudaMalloc
  0.03%  1.3591ms         4  339.78us  325.18us  356.60us  cuDeviceTotalMem
  0.02%  937.92us       114  8.2270us     625ns  341.91us  cudaEventCreateWithFlags
  0.02%  914.88us         4  228.72us  35.711us  787.40us  cudaStreamCreate
  0.02%  885.68us       352  2.5160us     244ns  76.231us  cuDeviceGetAttribute
  0.01%  540.87us        23  23.516us  9.8190us  72.343us  cudaLaunch
  0.01%  457.03us         6  76.171us  36.803us  119.21us  cudaMemcpy
  ...
```

For the ece408-low model with data size of 10000:

```
==311== NVPROF is profiling process 311, command: python m3.1.py ece408-low 10000
Loading model... done
Op Time: 0.428593
Correctness: 0.629 Model: ece408-low
==311== Profiling result:
Time(%)      Time     Calls     Avg      Min      Max  Name
83.33%       428.49ms 1         428.49ms 428.49ms 428.49ms  void
mxnet::op::forward_kernel<mshadow::gpu, float>(float*, mxnet::op::forward_kernel<mshadow::gpu,
float> const *, mxnet::op::forward_kernel<mshadow::gpu, float> const , int, int, int, int, int,
int)
7.62%        39.182ms 1         39.182ms 39.182ms 39.182ms  sgemm_sm35_ldg_tn_128x8x256x16x32
3.77%        19.383ms 2         9.6916ms 461.59us 18.922ms  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*)
2.82%        14.501ms 1         14.501ms  14.501ms 14.501ms  void
cudnn::detail::pooling_fw_4d_kernel<float, float, cudnn::detail::maxpooling_func<float,
cudnnNanPropagation_t=0>, int=0>(cudnnTensorStruct, float const *,
cudnn::detail::pooling_fw_4d_kernel<float, float, cudnn::detail::maxpooling_func<float,
cudnnNanPropagation_t=0>, int=0>, cudnnTensorStruct*, cudnnPoolingStruct, float,
cudnnPoolingStruct, int, cudnn::reduced_divisor, float)
  1.21%      6.2393ms  13       479.94us  1.5360us  4.3149ms  [CUDA memcpy HtoD]
  0.71%      3.6600ms  1        3.6600ms  3.6600ms  3.6600ms  sgemm_sm35_ldg_tn_64x16x128x8x32
  ...
==311== API calls:
Time(%)      Time     Calls       Avg       Min       Max  Name
 42.60%  1.85107s        18  102.84ms  17.640us  925.18ms  cudaStreamCreateWithFlags
 26.08%  1.13328s        10  113.33ms     753ns  325.26ms  cudaFree
 18.97%  824.30ms        23  35.839ms  236.95us  817.50ms  cudaMemGetInfo
  9.86%  428.51ms         1  428.51ms  428.51ms  428.51ms  cudaDeviceSynchronize
  1.81%  78.747ms        25  3.1499ms  5.4760us  42.526ms  cudaStreamSynchronize
  0.29%  12.544ms         8  1.5680ms  12.063us  4.3994ms  cudaMemcpy2DAsync
  0.15%  6.5509ms        41  159.78us  9.1420us  1.1257ms  cudaMalloc
  0.15%  6.5422ms         4  1.6356ms  26.468us  6.4352ms  cudaStreamCreate
  0.03%  1.3608ms         4  340.20us  338.68us  343.31us  cuDeviceTotalMem
  0.02%  863.67us       352  2.4530us     247ns  74.805us  cuDeviceGetAttribute
  0.01%  616.07us       114  5.4040us     526ns  135.51us  cudaEventCreateWithFlags
  0.01%  498.61us        23  21.678us  10.369us  59.764us  cudaLaunch
  0.01%  342.64us         6  57.107us  17.475us  117.77us  cudaMemcpy
  ...
```

Above implementation matches the expected correctness exactly.

In milestone 3, we split our work as follows:

yutao2: build the simple matrix multiplication implementation based on CPU version.

jiey3: move on next phase to explore the potential optimization such as tiling to improve the performance.

fz7: move on next phase to explore the potential optimization such as tiling to improve the performance.

Optimization Approach and Results

1.Basic kernel with forward path of convolutional layer

First, we implement the convolutional layer using a basic CUDA kernel. The input image is 28*28 and output image is 24X24. This kernel has a high level of parallelism and uses constant memory but do not use any shared memory. The parameters of the two kernels are shown as in Table. 1.

item	basic matrix multiplication
runtime	396ms
control divergence	yes
memory coalesce	no

Table 1

Basic kernel with forward path of convolutional layer is performed here. The significant parts in NVPROF are listed below.

```
==310== NVPROF is profiling process 310, command: python m3.1.py
Op Time: 0.396217
Correctness: 0.8562 Model: ece408-high
==310== Profiling application: python m3.1.py
==310== Profiling result:
Time(%)      Time     Calls   Avg       Min       Max      Name
78.51%      376.56ms     1   376.56ms  376.56ms  376.56ms  void
mxnet::op::forward_kernel<mshadow::gpu, float>(float*, mxnet::op::forward_kernel<mshadow::gpu,
float> const *, mxnet::op::forward_kernel<mshadow::gpu, float> const , int, int, int, int, int,
int)
8.06%       38.661ms     1   38.661ms  38.661ms  38.661ms  sgemm_sm35_ldg_tn_128x8x256x16x32
4.09%       19.612ms     1   19.612ms  19.612ms  19.612ms  void
mshadow::cuda::MapPlanLargeKernel<mshadow::sv::saveto, int=8, int=1024,
mshadow::expr::Plan<mshadow::Tensor<mshadow::gpu, int=4, float>, float>,
mshadow::expr::Plan<mshadow::expr::BinaryMapExp<mshadow::op::mul,
mshadow::expr::ScalarExp<float>, mshadow::Tensor<mshadow::gpu, int=4, float>, float, int=1>,
float>>(mshadow::gpu, unsigned int, mshadow::Shape<int=2>, int=4, int)
4.04%       19.359ms     2  9.6797ms  454.94us  18.905ms  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*)
3.00%       14.383ms     1  14.383ms  14.383ms  14.383ms  void
cudnn::detail::pooling_fw_4d_kernel<float, float, cudnn::detail::maxpooling_func<float,
cudnnNanPropagation_t=0>, int=0>(cudnnTensorStruct, float const *,
cudnn::detail::pooling_fw_4d_kernel<float, float, cudnn::detail::maxpooling_func<float,
cudnnNanPropagation_t=0>, int=0>, cudnnTensorStruct*, cudnnPoolingStruct, float,
cudnnPoolingStruct, int, cudnn::reduced_divisor, float)
1.27%      6.0785ms     13  467.58us  1.5360us  4.1529ms  [CUDA memcpy HtoD]
==310== API calls:
Time(%)      Time     Calls       Avg       Min       Max  Name
 42.54%  1.84059s        18  102.25ms  16.818us  919.97ms  cudaStreamCreateWithFlags
 25.85%  1.11814s        10  111.81ms     681ns  319.85ms  cudaFree
 20.04%  867.09ms        23  37.699ms  236.84us  860.45ms  cudaMemGetInfo
  9.15%  396.03ms         1  396.03ms  396.03ms  396.03ms  cudaDeviceSynchronize
  1.76%  76.286ms        25  3.0514ms  5.7980us  40.230ms  cudaStreamSynchronize
```

2.Basic kernel of shared memory

In next step, we tried to use the shared memory and constant memory to reduce the latency due to memory bandwidth. In kernel one, we used constant memory to save the elements of mask which has size of 50 * 1 * 5 * 5. The input size is (tile_width+k-1) * (tile_width+k-1), while the output size is (tile_width) * (tile_width). Each block will load an input image and it can achieve data reuse since we do not have to access elements in the global memory every time. The Table. 2 is the analysis of two kernels using shared and constant memory. The analysis is shown in Table. 2.

item	basic matrix with shared memory
runtime	329ms
control divergence	yes
memory coalesce	yes

Table 2

Basic kernel with shared memory is performed here. The significant parts in NVPROF are listed below.

```
==310== NVPROF is profiling process 310, command: python m3.1.py
Loading model... done
Op Time: 0.329201
Correctness: 0.8562 Model: ece408-high
==310== Profiling application: python m3.1.py
==310== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
75.03%     309.60ms      1      309.60ms  309.60ms  309.60ms  void
mxnet::op::forward_kernel<mshadow::gpu, float>(float*, mxnet::op::forward_kernel<mshadow::gpu,
float> const *, mxnet::op::forward_kernel<mshadow::gpu, float> const , int, int, int, int, int,
int)
9.39%      38.731ms      1      38.731ms  38.731ms  38.731ms  sgemm_sm35_ldg_tn_128x8x256x16x32
4.74%      19.576ms      1      19.576ms  19.576ms  19.576ms  void
mshadow::cuda::MapPlanLargeKernel<mshadow::sv::saveto, int=8, int=1024,
mshadow::expr::Plan<mshadow::Tensor<mshadow::gpu, int=4, float>, float>,
mshadow::expr::Plan<mshadow::expr::BinaryMapExp<mshadow::op::mul,
mshadow::expr::ScalarExp<float>, mshadow::Tensor<mshadow::gpu, int=4, float>, float, int=1>,
float>>(mshadow::gpu, unsigned int, mshadow::Shape<int=2>, int=4, int)
4.70%     19.380ms       2      9.6899ms  454.75us  18.925ms  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*)
3.49%     14.394ms       1      14.394ms  14.394ms  14.394ms  void
cudnn::detail::pooling_fw_4d_kernel<float, float, cudnn::detail::maxpooling_func<float,
cudnnNanPropagation_t=0>, int=0>(cudnnTensorStruct, float const *,
cudnn::detail::pooling_fw_4d_kernel<float, float, cudnn::detail::maxpooling_func<float,
cudnnNanPropagation_t=0>, int=0>, cudnnTensorStruct*, cudnnPoolingStruct, float,
cudnnPoolingStruct, int, cudnn::reduced_divisor, float)
 1.50%    6.1767ms       13     475.13us  1.5670us  4.2601ms  [CUDA memcpy HtoD]

==310== API calls:
Time(%)      Time     Calls       Avg       Min       Max  Name
 43.08%  1.83955s        18  102.20ms  17.219us  919.46ms  cudaStreamCreateWithFlags
 26.43%  1.12855s        10  112.86ms     728ns  321.08ms  cudaFree
 20.45%  873.45ms        23  37.976ms  235.18us  866.75ms  cudaMemGetInfo
  7.71%  329.07ms         1  329.07ms  329.07ms  329.07ms  cudaDeviceSynchronize
  1.79%  76.316ms        25  3.0527ms  5.4080us  40.236ms  cudaStreamSynchronize
  0.29%  12.521ms         8  1.5651ms  12.029us  4.3493ms  cudaMemcpy2DAsync
  0.15%  6.2584ms        41  152.64us  11.939us  1.1036ms  cudaMalloc
  0.03%  1.3392ms         4  334.80us  322.82us  339.79us  cuDeviceTotalMem
  0.02%  836.14us       352  2.3750us     244ns  62.721us  cuDeviceGetAttribute
  0.02%  786.53us       114  6.8990us     617ns  289.85us  cudaEventCreateWithFlags
  0.01%  471.28us        24  19.636us  9.5730us  48.515us  cudaLaunch
  0.01%  444.95us         4  111.24us  37.290us  318.62us  cudaStreamCreate
  0.01%  417.20us         6  69.533us  25.763us  119.63us  cudaMemcpy
```

3.Basic kernel of convolutional layer setting B as Z axis

In order to maximize the parallel execution, we set the B as Z axis, which has a better performance than setting M as Z axis. In next step, we tried to use the constant memory to reduce the latency due to high iteration times. We used constant memory to save the elements of mask which has size of 50X1X5X5. The Table. 3 is the analysis of two kernels using shared and constant memory. The analysis is shown in Table. 3.

item	basic matrix multiplication setting B as Z axis
runtime	302ms
control divergence	yes
memory coalesce	yes

Table 3

Basic kernel with shared and constant memory is performed here. The significant parts in NVPROF are listed below.

```
==310== NVPROF is profiling process 310, command: python m3.1.py
Op Time: 0.302154
Correctness: 0.8562 Model: ece408-high
==310== Profiling application: python m3.1.py
==310== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
73.05%     282.54ms      1      282.54ms  282.54ms  282.54ms  void
mxnet::op::forward_kernel<mshadow::gpu, float>(float*, mxnet::op::forward_kernel<mshadow::gpu,
float> const *, mxnet::op::forward_kernel<mshadow::gpu, float> const , int, int, int, int, int,
int)
9.84%      38.075ms      1      38.075ms  38.075ms  38.075ms  sgemm_sm35_ldg_tn_128x8x256x16x32
5.05%      19.541ms      1      19.541ms  19.541ms  19.541ms  void
mshadow::cuda::MapPlanLargeKernel<mshadow::sv::saveto, int=8, int=1024,
mshadow::expr::Plan<mshadow::Tensor<mshadow::gpu, int=4, float>, float>,
mshadow::expr::Plan<mshadow::expr::BinaryMapExp<mshadow::op::mul,
mshadow::expr::ScalarExp<float>, mshadow::Tensor<mshadow::gpu, int=4, float>, float, int=1>,
float>>(mshadow::gpu, unsigned int, mshadow::Shape<int=2>, int=4, int)
5.01%     19.374ms       2      9.6871ms  455.26us  18.919ms  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*)
==310== API calls:
Time(%)      Time     Calls       Avg       Min       Max  Name
 43.68%  1.99770s        18  110.98ms  17.336us  998.66ms  cudaStreamCreateWithFlags
 26.54%  1.21381s        10  121.38ms     902ns  344.01ms  cudaFree
 20.91%  956.24ms        23  41.575ms  236.17us  949.32ms  cudaMemGetInfo
  6.60%  302.04ms         1  302.04ms  302.04ms  302.04ms  cudaDeviceSynchronize
  1.70%  77.940ms        25  3.1176ms  5.8530us  41.964ms  cudaStreamSynchronize
  0.29%  13.361ms         8  1.6702ms  17.560us  6.6625ms  cudaMemcpy2DAsync
  0.15%  7.0471ms        41  171.88us  11.672us  1.1769ms  cudaMalloc
  0.03%  1.4006ms         4  350.15us  341.98us  369.34us  cuDeviceTotalMem
  0.02%  946.71us       114  8.3040us     674ns  445.28us  cudaEventCreateWithFlags
  0.02%  863.60us       352  2.4530us     245ns  67.102us  cuDeviceGetAttribute
  0.02%  856.46us         4  214.11us  53.042us  614.99us  cudaStreamCreate
  0.01%  545.23us        24  22.718us  11.450us  65.263us  cudaLaunch
  0.01%  496.09us         6  82.681us  43.466us  134.82us  cudaMemcpy
```

4. Four-level parallelism with tiling

We load the filter W[m, c] into the shared memory. All threads collaborate to copy the portion of the input X[n,c,.,.] that is required to compute the output tile into the shared memory array X_shared. Compute partial sum of output Y_shared[n, m,.,.]. We need to allocate shared memory for the input block X_tile_width * X_tile_width, where X_tile_width = TILE_WIDTH + K-1. In addition we also need to allocate shared memory for convolution filter. So the total amount of shared memory will be (TILE_WIDTH + K-1)* (TILE_WIDTH + K-1)+ K*K . The use of shared memory tiling results in a very high level of acceleration in the execution of the kernel.

Basic kernel with shared and constant memory is performed here. The significant parts in NVPROF are listed below.

```
==310== NVPROF is profiling process 310, command: python m3.1.py
Loading model... done
Op Time: 0.183244
Correctness: 0.8562 Model: ece408-high
==310== Profiling application: python m3.1.py
==310== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
60.87%     163.66ms      1      163.66ms  163.66ms  163.66ms  mxnet::op::forward_kernel(float*,
float const *, float const *, int, int, int, int, int, int, int)
14.58%     39.202ms      1      39.202ms  39.202ms  39.202ms  sgemm_sm35_ldg_tn_128x8x256x16x32
7.27%      19.559ms      1      19.559ms  19.559ms  19.559ms  void
mshadow::cuda::MapPlanLargeKernel<mshadow::sv::saveto, int=8, int=1024,
mshadow::expr::Plan<mshadow::Tensor<mshadow::gpu, int=4, float>, float>,
mshadow::expr::Plan<mshadow::expr::BinaryMapExp<mshadow::op::mul,
mshadow::expr::ScalarExp<float>, mshadow::Tensor<mshadow::gpu, int=4, float>, float, int=1>,
float>>(mshadow::gpu, unsigned int, mshadow::Shape<int=2>, int=4, int)
7.21%      19.385ms      2      9.6925ms  459.16us  18.926ms  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*)
5.38%      14.469ms      1      14.469ms  14.469ms  14.469ms  void
cudnn::detail::pooling_fw_4d_kernel<float, float, cudnn::detail::maxpooling_func<float,
cudnnNanPropagation_t=0>, int=0>(cudnnTensorStruct, float const *,
cudnn::detail::pooling_fw_4d_kernel<float, float, cudnn::detail::maxpooling_func<float,
cudnnNanPropagation_t=0>, int=0>, cudnnTensorStruct*, cudnnPoolingStruct, float,
cudnnPoolingStruct, int, cudnn::reduced_divisor, float)
2.31%      6.2071ms      13    477.47us  1.5360us  4.2722ms  [CUDA memcpy HtoD]
1.36%      3.6570ms       1    3.6570ms  3.6570ms  3.6570ms  sgemm_sm35_ldg_tn_64x16x128x8x32
==310== API calls:
Time(%)      Time     Calls       Avg       Min       Max  Name
 44.68%  1.94391s        18  107.99ms  16.235us  974.78ms  cudaStreamCreateWithFlags
 27.04%  1.17633s        10  117.63ms  1.0830us  338.25ms  cudaFree
 21.41%  931.54ms        23  40.502ms  236.22us  924.75ms  cudaMemGetInfo
  4.21%  183.20ms         1  183.20ms  183.20ms  183.20ms  cudaDeviceSynchronize
  1.81%  78.612ms        25  3.1445ms  5.8480us  42.462ms  cudaStreamSynchronize
  0.23%  9.8672ms         4  2.4668ms  32.086us  9.7278ms  cudaStreamCreate
  0.19%  8.3734ms         8  1.0467ms  10.123us  4.3189ms  cudaMemcpy2DAsync
  0.18%  7.7813ms         6  1.2969ms  27.730us  7.5136ms  cudaMemcpy
  0.15%  6.5801ms        41  160.49us  10.202us  1.1155ms  cudaMalloc
  0.03%  1.3999ms         4  349.98us  338.70us  378.42us  cuDeviceTotalMem
  0.02%  994.46us       114  8.7230us     614ns  439.08us  cudaEventCreateWithFlags
  0.02%  949.95us       352  2.6980us     247ns  81.163us  cuDeviceGetAttribute
  0.01%  572.82us        24  23.867us  11.240us  52.457us  cudaLaunch
  0.01%  222.10us        30  7.4030us     608ns  86.712us  cudaSetDevice
```

5. Reduction of convolutional layer to matrix multiplication

We can build an even faster convolutional layer by reducing it to matrix multiplication and then using simple multiplication. The central idea of this method is unfolding and duplicating of the inputs to the convolutional kernel in such way that all elements needed to compute one output element will be stored as one sequential block. This will reduce the forward operation of the convolutional layer to one large matrix-matrix multiplication[1].

First we will rearrange all input elements. Since the results of the convolutions are summed across input features, the input features can be concatenated into one large matrix. Each row of this matrix contains all the input values necessary to compute one element of an output feature. This means that each input element will be replicated multiple times. However, due to limitation of time, we cannot implement this method.

6.Fast Fourier Transformation (FFT) for convolutional layer

Fast Fourier Transformation (FFT) is a highly parallel “divide and conquer” algorithm for the calculation of Discrete Fourier Transformation of single, or multidimensional signals and it is good at large, power of two sized data processing. It can be efficiently implemented using the CUDA programming model and the CUDA distribution package includes CUFFT, a CUDA-based FFT library, whose API is modeled after the widely-used CPU-based “FFTW” library. The basic outline of Fourier-based convolution is: • Apply direct FFT to the convolution kernel, • Apply direct FFT to the input data array (or image), • Perform the point-wise multiplication of the two preceding results, • Apply inverse FFT to the result of the multiplication. However, even though we write a bug free code utilizing FFT to do convolution, but we failed. Because the version of the CUDA on the server do not support this library. But we failed.

7.Work split

yutao2: build the tiling convolution kernel and explore the potential optimization.

jiey3: move on next phase to explore the potential optimization such as tiling to improve the performance.

fz7: attempt to achieve unroll method of convolution, Fast Fourier transformation

References

1.3rd-Edition-Chapter16-case-study-DNN-FINAL-corrected

Suggestions

When we submit our solution and wait to be executed, there might be some problem on the waiting list set. The screen will show ' waiting for the server to process your request' and we wait in the line. However, sometimes we wait for five minutes to queue in line while the same time other teammate who submits solution later than me will execute earlier than me. So we think there might be some bugs in the waiting list set.