

ECE408 Final Project Report

Teamname: spicy-chicken

Members: Xiaocong Yu(xy21), Dawei Wang(dwang56), Kexuan Zou(kzou3)

Milestone 1

List of kernel calls that collectively consume more than 90% of the program time

Type	Time(%)	Time	Calls	Avg	Min	Max	Name
GPU activities:	40.07%	16.772ms	20	838.61us	1.1200us	16.152ms	[CUDA memcpy HtoD]
	20.15%	8.4355ms	1	8.4355ms	8.4355ms	8.4355ms	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)							
	11.82%	4.9474ms	1	4.9474ms	4.9474ms	4.9474ms	volta_cgemm_64x32_tn
	7.04%	2.9486ms	2	1.4743ms	25.087us	2.9235ms	void
op_generic_tensor_kernel<int=2, float, float, float, int=256, cudnnGenericOp_t=7,							
cudnnNanPropagation_t=0, cudnnDimOrder_t=0, int=1>(cudnnTensorStruct, float*, cudnnTensorStruct, float const *, cudnnTensorStruct, float const *, float, float, float, float, dimArray, reducedDivisorArray)							
	5.71%	2.3909ms	1	2.3909ms	2.3909ms	2.3909ms	void fft2d_c2r_32x32<float,
bool=0, bool=0, unsigned int=1, bool=0, bool=0>(float*, float2 const *, int, int, int, int, int, int, int, int, float, float, cudnn::reduced_divisor, bool, float*, float*, int2, int, int)							
	5.59%	2.3403ms	1	2.3403ms	2.3403ms	2.3403ms	volta_sgemm_128x128_tn
	4.56%	1.9076ms	1	1.9076ms	1.9076ms	1.9076ms	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)							
	4.21%	1.7638ms	1	1.7638ms	1.7638ms	1.7638ms	void fft2d_r2c_32x32<float,
bool=0, unsigned int=0, bool=0>(float2*, float const *, int, int, int, int, int, int, int, int, int, int, cudnn::reduced_divisor, bool, int2, int, int)							

List of API calls that collectively consume more than 90% of the program time

Type	Time(%)	Time	Calls	Avg	Min	Max	Name
API calls:	41.37%	3.00690s	22	136.68ms	13.759us	1.58874s	cudaStreamCreateWithFlags
	33.85%	2.46087s	24	102.54ms	87.306us	2.45588s	cudaMemGetInfo
	21.29%	1.54779s	19	81.463ms	948ns	413.98ms	cudaFree

Explanation of the difference between kernels and API calls

“Summary mode is the default operating mode for nvprof. In this mode, nvprof outputs a single result line for each kernel function and each type of CUDA memory copy/set performed by the application. For each kernel, nvprof outputs the total time of all instances of the kernel or type of memory copy as well as the average, minimum, and maximum time. The time for a kernel is the kernel execution time on the device. By default, nvprof also prints a summary of all the CUDA runtime/driver API calls. Output of nvprof (except for tables) are prefixed with ==<pid>==, <pid> being the process ID of the application being profiled.” -- CUDA Toolkit Reference

From the official reference for nvprof tool, we know that the list for kernels and API are different since the API calls are mostly executed at CPU side (host code that may or may not invoke GPU), whereas the kernel calls are executed at GPU side (device code). There list some driver API calls including cudaGetDevice, cuDeviceGetName and Runtime API calls including cudaFunctionSetAttr, cudaMemsetAsync, etc.

API calls deals with collecting information for NVCC during compile time to help generate architecture and computing-ability specific executable code for CPU and GPU, and invoking kernel function on device so it will be executed much more times than kernel calls. Whereas Kernel time usage information above is collected for single kernel, so you can see the calculation functions are only called once. Time mostly comprised of kernel calculation function and cudaMemcpy from the GPU shared memory.

Output of rai running MXNet on CPU & GPU

Running /usr/bin/time python m1.1.py

Loading fashion-mnist data... done

Loading model... done

New Inference

EvalMetric: {'accuracy': 0.8236}

9.18user 3.70system

Time used 0:05.38 elapsed 239%CPU

(0avgtext+0avgdata2470492maxresident)k0inputs+2824outputs (0major+669491minor)pagefaults

0swaps

Time used for m1.1py

0:05.38 elapsed 239%CPU

Running /usr/bin/time python m1.2.py

Loading fashion-mnist data... done

Loading model... done

New Inference

EvalMetric: {'accuracy': 0.8236}

4.31user 3.25system

0:04.23 elapsed

178%CPU

(0avgtext+0avgdata2858044maxresident)k8inputs+1728outputs (0major+663219minor)pagefaults

0swaps

Time used for ml.2py

0:04.23 elapsed 178%CPU

Milestone 2

Dataset	Correctness	Op Time	Program execution time	Percentage of CPU
IMG 100	0.84	Op Time: 0.035449 Op Time: 0.077439	0:01.03elapsed	568%CPU
IMG 1000	0.852	Op Time: 0.242293 Op Time: 0.753023	0:02.03elapsed	364%CPU
IMG 10000	0.8397	Op Time: 2.509002 Op Time: 7.785874	0:12.08elapsed	174%CPU

Milestone 3

100

Loading fashion-mnist data... done

Loading model... done

New Inference

Op Time: 0.000073

Op Time: 0.000217

Correctness: 0.84 Model: ece408

4.41user 3.45system 0:04.24elapsed 185%CPU (0avgtext+0avgdata 2760292maxresident)k

0inputs+4576outputs (0major+619191minor)pagefaults 0swaps

1000

Loading fashion-mnist data... done

Loading model... done

New Inference

Op Time: 0.000532

Op Time: 0.001969

Correctness: 0.852 Model: ece408

4.30user 3.37system 0:04.18elapsed 183%CPU (0avgtext+0avgdata 2748216maxresident)k

8inputs+4576out

puts (0major+616435minor)pagefaults 0swaps

10000

Loading fashion-mnist data... done

Loading model... done

New Inference

Op Time: 0.005501

Op Time: 0.021360

Correctness: 0.8397 Model: ece408

4.38user 3.35system 0:04.38elapsed 176%CPU (0avgtext+0avgdata 2832252maxresident)k

8inputs+4576outputs (0major+660053minor)pagefaults 0swaps

Loading fashion-mnist data... done

==278== NVPROF is profiling process 278, command: python m3.1.py

Loading model... done

New Inference

Op Time: 0.005137

Op Time: 0.020465

Correctness: 0.8397 Model: ece408

==278== Profiling application: python m3.1.py

==278== Profiling result:

Type	Time(%)	Time	Calls	Avg	Min	Max	Name
GPU activities:	51.04%	25.540ms	2	12.770ms	5.1011ms	20.438ms	
mxnet::op::forward_kernel(float*, float const *, float const *, int, int, int, int, int, int, int)							
	33.69%	16.858ms	20	842.92us	1.0880us	16.470ms	[CUDA memcpy HtoD]
	4.75%	2.3745ms	2	1.1873ms	724.35us	1.6502ms	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)							
	4.62%	2.3096ms	2	1.1548ms	20.640us	2.2890ms	volta_sgemm_32x128_tn
	3.22%	1.6134ms	2	806.68us	22.432us	1.5909ms	void
op_generic_tensor_kernel<int=2, float, float, float, int=256, cudnnGenericOp_t=7,							
cudnnNanPropagation_t=0, cudnnDimOrder_t=0, int=1>(cudnnTensorStruct, float*, cudnnTensorStruct,							
float const *, cudnnTensorStruct, float const *, float, float, float, float, dimArray, reducedDivisorArray)							
	2.09%	1.0436ms	1	1.0436ms	1.0436ms	1.0436ms	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)							
API calls:	42.30%	3.04868s	22	138.58ms	14.623us	1.55085s	cudaStreamCreateWithFlags
	34.60%	2.49384s	22	113.36ms	93.244us	2.48890s	cudaMemGetInfo
	21.09%	1.52040s	18	84.467ms	804ns	407.55ms	cudaFree
0.51%	36.510ms	912	40.033us	298ns	10.056ms		cudaFuncSetAttribute
	0.48%	34.501ms	9	3.8334ms	22.910us	16.629ms	cudaMemcpy2DAsync
	0.39%	27.938ms	6	4.6564ms	3.4350us	20.442ms	cudaDeviceSynchronize
	0.25%	17.942ms	66	271.85us	5.9540us	9.6978ms	cudaMalloc
	0.08%	5.6985ms	12	474.88us	7.0040us	5.2797ms	cudaMemcpy

Milestone 4

Optimization for CNN forward kernel

Three optimizations for MS4:

- Constant memory
- Unroll and shared-memory matrix multiply
- Shared memory in convolution

Clarification for performance:

In MS4, the GPU on the computing server assigns at most two two job for single GPU, so the competition between jobs may cause the computing performance a little bit lower than previous plain implementation.

Performance in default implementation for MS3, plain GPU convolution operations are Op Time:

0.005501 Op Time: 0.021360.

1.Weight matrix (kernel values) in constant memory

Runtime of the optimized program:

```
*Running nvprof -o timeline.nvprof python m4.1.py
Loading fashion-mnist data... done
==279== NVPROF is profiling process 279, command: python m4.1.py
Loading model... done
New Inference
Op Time: 0.005337
Op Time: 0.020313
Correctness: 0.8397 Model: ece408
==279== Generated result file: /build/timeline.nvprof
```

nvvp profiler analysis:

Name	Invocations	Avg. Duration	Regs	Static SMem	Avg. Dynamic SMem
memset (0)	0	0 ns	0	0	0
void mshadow::cuda::MapPlanKernel<mshadow::sv::saveto, int=8, r	13	2.471 µs	16	0	0
void mshadow::cuda::MapPlanKernel<mshadow::sv::saveto, int=8, r	1	4.832 µs	26	0	0
void mshadow::cuda::MapPlanKernel<mshadow::sv::plusto, int=8, r	2	12.096 µs	16	0	0
void mshadow::cuda::SoftmaxKernel<int=8, float, mshadow::expr::f	1	67.679 µs	20	1024	0
void mshadow::cuda::MapPlanLargeKernel<mshadow::sv::saveto, i	1	150.717 µs	22	0	0
void op_generic_tensor_kernel<int=2, float, float, float, int=256, cu	2	808.035 µs	16	0	0
void mshadow::cuda::MapPlanLargeKernel<mshadow::sv::saveto, i	2	1.18702 ms	22	0	0
volta_sgemm_32x128_tn	2	2.6846 ms	55	16384	0
void cudnn::detail::pooling_fw_4d_kernel<float, float, cudnn::detai	1	4.13773 ms	48	0	3872
mxnet::op::forward_kernel_consmem(float*, float const *, int, int, i	2	28.21216 ms	32	0	0

Compute
80.4% mxnet::op::forward_kernel_consmem(float*, float const *, ..
7.6% volta_sgemm_32x128_tn
5.9% void cudnn::detail::pooling_fw_4d_kernel<float, float, cudn...
3.4% void mshadow::cuda::MapPlanLargeKernel<mshadow::sv::S...
2.3% void op_generic_tensor_kernel<int=2, float, float, float, int=..
0.2% void mshadow::cuda::MapPlanLargeKernel<mshadow::sv::S...
0.1% void mshadow::cuda::SoftmaxKernel<int=8, float, mshadow...
0.0% void mshadow::cuda::MapPlanKernel<mshadow::sv::saveto,...
0.0% void mshadow::cuda::MapPlanKernel<mshadow::sv::plusto, ..
0.0% void mshadow::cuda::MapPlanKernel<mshadow::sv::saveto,...
0.0% memset (0)

Results

Low Memcpy/Kernel Overlap [0 ns / 28.21733 ms = 0%]

The percentage of time when memcopy is being performed in parallel with kernel is low.

Low Kernel Concurrency [4.736 μ s / 61.7964 ms = 0%]

The percentage of time when two kernels are being executed in parallel is low.

Low Memcpy Throughput [2.777 GB/s avg, for memcpys accounting for 3.7% of all memcopy time]

The memory copies are not fully using the available host to device bandwidth.

Low Memcpy Overlap [0 ns / 4.48 μ s = 0%]

The percentage of time when two memory copies are being performed in parallel is low.

The profile list memory usage for different media of storage, including L1 cache(shared memory), Unified cache, L2 Cache, Device memory(global memory for GPU)

Description:

Applied constant memory for the convolution kernel. Originally, each thread in the grid and block will fetch data in the convolution kernel during the iterations of channels and all the element in index range $[0, K*K - 1]$. By utilizing the constant memory, all block threads in the grid can benefit from this since the requirement for global memory bandwidth will be significantly reduced. At the same time, accessing constant memory is somehow more efficient than accessing shared memory for certain access pattern. Referenced to the lecture slides 7-8, memory access for shared memory and constant memory is 5 clock-cycles whereas global memory access is 500 clock-cycles.

Analysis:

Experiment is defaultly ran with dataset 10000. Here the Op Time for two convolution step in MS4 are 0.005337 and 0.020313 s separately.

This optimization mostly focus on reducing memory access cost. Originally, the GPU implementation will require $H_{out} * W_{out} * M * C * K * K$ times global memory access for kernel data, letting alone the hardware cache optimization. After the constant memory is applied, all those memory turned into memory access that is efficiency equivalent to shared memory.

2. Unrolling + shared-memory matrix multiply

Runtime of the optimized program:

```
*Running nvprof -o timeline.nvprof python m4.1.py
Loading fashion-mnist data... done
==279== NVPROF is profiling process 279, command: python m4.1.py
Loading model... done
New Inference
Op Time: 0.130809
Op Time: 0.144876
Correctness: 0.8397 Model: ece408
==279== Generated result file: /build/timeline.nvprof
*Running nvprof --kernels "__gemm" --analysis-metrics -o gemm_analysis.nvprof p
ython m4.1.py
```

nvvp profiler analysis:

Name	Invocations	Avg. Duration	Regs	Static SMem	Avg. Dynamic SMem
memset (0)	0	0 ns	0	0	0
void mshadow::cuda::MapPlanKernel<mshadow::sv::plusto, int=8,	2	2.08 µs	16	0	0
void mshadow::cuda::MapPlanKernel<mshadow::sv::saveto, int=8,	14	2.148 µs	16	0	0
void mshadow::cuda::SoftmaxKernel<int=8, float, mshadow::expr::	1	4.096 µs	20	1024	0
void mshadow::cuda::MapPlanKernel<mshadow::sv::saveto, int=8,	1	4.416 µs	26	0	0
mxnet::op::__gemm(float*, float*, float*, int, int, int, int, int)	200	4.802 µs	32	2048	0
mxnet::op::__unroll(int, int, int, int, float*, float*)	200	6.228 µs	30	0	0
void op_generic_tensor_kernel<int=2, float, float, float, int=256, cu	2	10.432 µs	16	0	0
void mshadow::cuda::MapPlanKernel<mshadow::sv::saveto, int=8,	2	13.215 µs	16	0	0
void cudnn::detail::pooling_fw_4d_kernel<float, float, cudnn::deta	1	13.727 µs	48	0	3872
volta_sgemm_128x32_tn	1	16.224 µs	55	16384	0
volta_sgemm_32x32_sliced1x4_tn	1	67.328 µs	86	32768	0

Compute
51.8% mxnet::op::__unroll(int, int, int, int, float*, float*)
40.0% mxnet::op::__gemm(float*, float*, float*, int, int, int, in...
2.8% volta_sgemm_32x32_sliced1x4_tn
1.3% void mshadow::cuda::MapPlanKernel<mshadow::sv::sav...
1.1% void mshadow::cuda::MapPlanKernel<mshadow::sv::sav...
0.9% void op_generic_tensor_kernel<int=2, float, float, float...
0.7% volta_sgemm_128x32_tn
0.6% void cudnn::detail::pooling_fw_4d_kernel<float, float, c...
0.2% void mshadow::cuda::MapPlanKernel<mshadow::sv::sav...
0.2% void mshadow::cuda::MapPlanKernel<mshadow::sv::plu...
0.2% void mshadow::cuda::SoftmaxKernel<int=8, float, msha...
0.4% memset (0)

Results

Low Memcpy/Kernel Overlap [0 ns / 648.025 μ s = 0%]

The percentage of time when memcopy is being performed in parallel with kernel is low.

Low Kernel Concurrency [0 ns / 2.29089 ms = 0%]

The percentage of time when two kernels are being executed in parallel is low.

Low Memcpy Throughput [383.775 MB/s avg, for memcpys accounting for 7.2% of all memcopy time]

The memory copies are not fully using the available host to device bandwidth.

Low Memcpy Overlap [0 ns / 3.488 μ s = 0%]

The percentage of time when two memory copies are being performed in parallel is low.

Low Compute Utilization [2.40337 ms / 4.91913 s = 0%]

The multiprocessors of one or more GPUs are mostly idle.

Description:

The second optimization we implemented is unfolding and duplicating the inputs to the convolution kernel so that a general matrix multiplication can be used instead of traditional convolution operation. As described in the book, a kernel `__unroll()` is used to unroll input data `x.dptr_`. After this operation, a general-purpose matrix multiplication kernel (similar to MP3), `__gemm()`, is invoked to calculate the output. Since `w.dptr_` has dimension `[C, M, K, K]` and is already compatible with the unrolled version `x.dptr_`, nothing needs to be down for `w.dptr_` prior to the `__gemm()` call.

Analysis:

According to timeline profile of the nvvp output, general-purpose matrix multiplication introduces a huge overhead, and this is the first bottleneck of the slow performance of this optimization. Also, for each feature map in the batch, two kernels are launched in sequence: `unroll()` which unrolls `x.dptr_` followed by `gemm()` which performs the matrix multiplication. The two kernels are purely sequential and lack any level of parallelism.

3. Shared Memory convolution

Runtime of the optimized program:

```
*Running nvprof -o timeline.nvprof python m4.1.py
Loading fashion-mnist data... done
==279== NVPROF is profiling process 279, command: python m4.1.py
Loading model... done
New Inference
Op Time: 0.006043
Op Time: 0.035648
Correctness: 0.8397 Model: ece408
==279== Generated result file: /build/timeline.nvprof
```

nvvp profiler analysis:

Name	Invocations	Avg. Duration	Regs	Static SMem	Avg. Dynamic SMem
memset (0)	0	0 ns	0	0	0
void mshadow::cuda::MapPlanKernel<mshadow::sv::saveto, int=8,	13	2.503 µs	16	0	0
void mshadow::cuda::MapPlanKernel<mshadow::sv::saveto, int=8,	1	5.088 µs	26	0	0
void mshadow::cuda::MapPlanKernel<mshadow::sv::plusto, int=8,	2	11.92 µs	16	0	0
void mshadow::cuda::SoftmaxKernel<int=8, float, mshadow::expr:	1	75.136 µs	20	1024	0
void mshadow::cuda::MapPlanLargeKernel<mshadow::sv::saveto,	1	152.383 µs	22	0	0
void op_generic_tensor_kernel<int=2, float, float, float, int=256, cu	2	805.705 µs	16	0	0
void cudnn::detail::pooling_fw_4d_kernel<float, float, cudnn::deta	1	1.04447 ms	48	0	3872
volta_sgemm_32x128_tn	2	1.15908 ms	55	16384	0
void mshadow::cuda::MapPlanLargeKernel<mshadow::sv::saveto,	2	1.18249 ms	22	0	0
mxnet::op::forward_kernel_shmem(float*, float const *, float cons	2	22.34373 ms	40	0	1700

Compute
85.4% mxnet::op::forward_kernel_shmem(float*, float const ...
4.5% void mshadow::cuda::MapPlanLargeKernel<mshadow::....
4.4% volta_sgemm_32x128_tn
3.1% void op_generic_tensor_kernel<int=2, float, float, float...
2.0% void cudnn::detail::pooling_fw_4d_kernel<float, float, c...
0.3% void mshadow::cuda::MapPlanLargeKernel<mshadow::....
0.1% void mshadow::cuda::SoftmaxKernel<int=8, float, msha...
0.1% void mshadow::cuda::MapPlanKernel<mshadow::sv::sa...
0.0% void mshadow::cuda::MapPlanKernel<mshadow::sv::plu...
0.0% void mshadow::cuda::MapPlanKernel<mshadow::sv::sa...
0.0% memset (0)

Results

⚠ **Low Memcpy/Kernel Overlap** [0 ns / 17.2892 ms = 0%]

The percentage of time when memcpy is being performed in parallel with kernel is low.

⚠ **Low Kernel Concurrency** [0 ns / 47.0083 ms = 0%]

The percentage of time when two kernels are being executed in parallel is low.

⚠ **Low Memcpy Throughput** [266.222 MB/s avg, for memcpys accounting for 0.2% of all memcpy time]

The memory copies are not fully using the available host to device bandwidth.

⚠ **Low Memcpy Overlap** [0 ns / 5.76 μ s = 0%]

The percentage of time when two memory copies are being performed in parallel is low.

Description:

Since the previous approach, unrolling with general-purpose matrix multiply is relative slow due to global memory access and the subsequent matrix multiply kernel, we considered the shared memory convolution, which first loads input data `x.dptr_` into kernel shared memory; after this memory access pattern need not to be coalesced. Using the shared memory technique, global memory access is reduced, to $1/\text{TILE_SIZE}$ that of the original. This approach invokes a single kernel, `forward_kernel_consmem()`.

Analysis:

With shared memory implemented, the usage for global memory significantly decreased. For constant memory optimization, we have about 2.7 GB/s of memory throughput, but in contrast, we have only 266 MB/s of memory throughput for shared memory optimization. In addition, this optimization gives runtime close to the constant memory optimization and is greatly faster than the unroll optimization. As a result, we consider this shared memory optimization as the best one among the three optimizations we implemented so far.

Milestone 5

1. Multiple kernel implementations for different layer sizes

Runtime of the optimized program:

```
done
New Inference
Op Time: 0.004730
Op Time: 0.011295
Correctness: 0.8397 Model: ece408
4.44user 3.32system 0:04.26elapsed 182%CPU (0avgtext+0avg
```

nvvp profiler analysis:

Name	Invocations	Avg. Duration	Regs	Static SMem	Avg. Dynamic SMem
memset (0)	0	0 ns	0	0	0
void mshadow::cuda::MapPlanKernel<mshadow::sv::saveto, int=8, mshadow::	13	2.163 µs	16	0	0
void mshadow::cuda::MapPlanKernel<mshadow::sv::saveto, int=8, mshadow::	1	4.96 µs	26	0	0
void mshadow::cuda::MapPlanKernel<mshadow::sv::plusto, int=8, mshadow::	2	11.888 µs	16	0	0
void mshadow::cuda::SoftmaxKernel<int=8, float, mshadow::expr::Plan<msha	1	75.2 µs	20	1024	0
void mshadow::cuda::MapPlanLargeKernel<mshadow::sv::saveto, int=8, int=1	1	156.703 µs	22	0	0
void op_generic_tensor_kernel<int=2, float, float, float, int=256, cudnnGeneri	2	812.057 µs	16	0	0
void cudnn::detail::pooling_fw_4d_kernel<float, float, cudnn::detail::maxpool	1	1.05657 ms	48	0	3872
void mshadow::cuda::MapPlanLargeKernel<mshadow::sv::saveto, int=8, int=1	2	1.20754 ms	22	0	0
volta_sgemm_32x128_tn	2	1.25069 ms	55	16384	0
mxnet::op::forward_kernel_consmem_small(float*, float const *, int, int, int, in	1	4.77788 ms	32	0	0
mxnet::op::forward_kernel_consmem_large(float*, float const *, int, int, int, in	1	11.43529 ms	32	0	0

Compute
47.4% mxnet::op::forward_kernel_consmem_large(float*, float ...
19.8% mxnet::op::forward_kernel_consmem_small(float*, float...
10.4% volta_sgemm_32x128_tn
10.0% void mshadow::cuda::MapPlanLargeKernel<mshadow::s...
6.7% void op_generic_tensor_kernel<int=2, float, float, float, in...
4.4% void cudnn::detail::pooling_fw_4d_kernel<float, float, cud...
0.6% void mshadow::cuda::MapPlanLargeKernel<mshadow::sv::...
0.3% void mshadow::cuda::SoftmaxKernel<int=8, float, mshado...
0.1% void mshadow::cuda::MapPlanKernel<mshadow::sv::savet...
0.1% void mshadow::cuda::MapPlanKernel<mshadow::sv::plust...
0.0% void mshadow::cuda::MapPlanKernel<mshadow::sv::savet...
0.0% memset (0)

Results

⚠ **Low Kernel / Malloc Efficiency** [24.1107 ms / 16.4758 ms = 1.463]

The amount of time performing compute is low relative to the amount of time required for malloc.

⚠ **Low Malloc/Kernel Overlap** [0 ns / 16.4758 ms = 0%]

The percentage of time when malloc is being performed in parallel with kernel is low.

⚠ **Low Kernel Concurrency** [0 ns / 18.71747 ms = 0%]

The percentage of time when two kernels are being executed in parallel is low.

⚠ **Low Malloc Throughput** [279.434 MB/s avg, for mallocs accounting for 0.2% of all malloc time]

The memory copies are not fully using the available host to device bandwidth.

⚠ **Low Malloc Overlap** [0 ns / 5.888 μs = 0%]

The percentage of time when two memory copies are being performed in parallel is low.

Description:

We customized two constant buffers for this optimization. We dugged into the details of the model used in the submission and found that there were two layers of the kernel whose sizes are $6*1*5*5$ and $16*6*5*5$ separately and the image size is $10000 * 1 * 48 * 48$. We realize that we can reduce the time cost if we treat two models differently according to the specific filter kernel model size. So we declare two macros for two different kernel size and two constant memory globally.

We tried to use two different kernel functions to treat the convolution process differently for different input and output model, such that we didn't use shared memory in the small kernel since the overhead is considered little bit heavy for this convolution with small scale. But the memory access would become $44 * 44 * 25 * B$ (shared memory implementation will only cost $48 * 48 * B$ times), let alone the cache mechanism in the block. So we turn to the block dimensions. The decision for which kernel to use is made before we invoke the kernel in the forward function by the filter kernel size.

Analysis:

The output model dimension is $6 * 44 * 44$ for small kernel and $16 * 18 * 18$ for large kernel. As for improving control divergence, we choose block size $18 * 18$ for larger kernel, and $20 * 20$ for smaller kernel. So that there will be less blocks in perspective of gridDim.z will be decreased. And there will be no control divergence in each grid block. By using two separate constant memory, we can customize each layer for better performance and fine tune various parameters, as will be discussed in part 2 "Parameter sweeping, loop unrolling, overhead reduction" below.

2. Parameter sweeping, loop unrolling, overhead reduction

Runtime of the optimized program:


```

done
New Inference
Op Time: 0.005094
Op Time: 0.012116
Correctness: 0.8397 Model: ece408
4.52user 3.45system 0:04.28elapsed 185%CPU (0avgtext+0avgdata 2829556maxresident)k
0inputs+4704outputs (0major+659743minor)pagefaults 0swaps

```

nvvp profiler analysis:

Name	Invocations	Avg. Duration	Regs	Static SMem	Avg. Dynamic SMem
memset (0)	0	0 ns	0	0	0
void mshadow::cuda::MapPlanKernel<mshadow::sv::saveto, int=8, r	13	2.136 µs	16	0	0
void mshadow::cuda::MapPlanKernel<mshadow::sv::saveto, int=8, r	1	4.832 µs	26	0	0
void mshadow::cuda::MapPlanKernel<mshadow::sv::plusto, int=8, r	2	11.84 µs	16	0	0
void mshadow::cuda::SoftmaxKernel<int=8, float, mshadow::expr::f	1	75.071 µs	20	1024	0
void mshadow::cuda::MapPlanLargeKernel<mshadow::sv::saveto, i	1	154.367 µs	22	0	0
void op_generic_tensor_kernel<int=2, float, float, float, int=256, cud	2	805.961 µs	16	0	0
void cudnn::detail::pooling_fw_4d_kernel<float, float, cudnn::detail	1	1.04534 ms	48	0	3872
volta_sgemm_32x128_tn	2	1.15345 ms	55	16384	0
void mshadow::cuda::MapPlanLargeKernel<mshadow::sv::saveto, i	2	1.18929 ms	22	0	0
mxnet::op::__shmem_1(float*, float const *, int, int)	1	5.01759 ms	30	0	1700
mxnet::op::__shmem_2(float*, float const *, int, int, int, int)	1	12.13539 ms	31	0	2804

Compute
48.9% mxnet::op::__shmem_2(float*, float const *, int, int, int, int)
20.2% mxnet::op::__shmem_1(float*, float const *, int, int)
9.6% void mshadow::cuda::MapPlanLargeKernel<mshadow::sv::saveto, int=...
9.3% volta_sgemm_32x128_tn
6.5% void op_generic_tensor_kernel<int=2, float, float, float, int=256, cudn...
4.2% void cudnn::detail::pooling_fw_4d_kernel<float, float, cudnn::detail::m...
0.6% void mshadow::cuda::MapPlanLargeKernel<mshadow::sv::saveto, int=...
0.3% void mshadow::cuda::SoftmaxKernel<int=8, float, mshadow::expr::Pla...
0.1% void mshadow::cuda::MapPlanKernel<mshadow::sv::saveto, int=8, msh...
0.1% void mshadow::cuda::MapPlanKernel<mshadow::sv::plusto, int=8, msh...
0.0% void mshadow::cuda::MapPlanKernel<mshadow::sv::saveto, int=8, msh...
0.0% memset (0)

Results
Low Kernel / Memcpy Efficiency [24.79292 ms / 16.71074 ms = 1.484] The amount of time performing compute is low relative to the amount of time required for memcpy.
Low Memcpy/Kernel Overlap [0 ns / 16.71074 ms = 0%] The percentage of time when memcpy is being performed in parallel with kernel is low.
Low Kernel Concurrency [0 ns / 19.46266 ms = 0%] The percentage of time when two kernels are being executed in parallel is low.
Low Memcpy Throughput [366.563 MB/s avg, for memcpys accounting for 0.3% of all memcpy time] The memory copies are not fully using the available host to device bandwidth.
Low Memcpy Overlap [0 ns / 7.904 µs = 0%] The percentage of time when two memory copies are being performed in parallel is low.

Description:

We have demonstrated that in the previous part “Multiple kernel implementations for different layer sizes”, that adapting different kernels for different layer sizes gives us a huge gain in terms of runtime, since inappropriately determined block size may cause control divergence for convolutional layers with different input (and therefore output) size. However, due to the computational power constraint imposed by the hardware, we have to fine tune parameters both based on the output of nvvp profiler and runtime of forward passes for both layers. To further reduce the overhead, combination of optimization techniques such as loop unrolling and memory address calculation are also used.

Analysis:

There are two convolution layers with different input data shape. Both layers have kernel size $K = 5$. For the first layer the input size is $[10,000, 1, 48, 48]$ and $M = 6$. Minimum runtime is achieved when $TILE_SIZE = 16$. For the second layer the input size is $[10,000, 6, 22, 22]$ and $M = 16$. Minimum runtime is achieved when $TILE_SIZE = 22$. (Notice that, theoretically, to avoid control divergence, $TILE_SIZE$ should divide output size.)

To make good use of compiler, `#pragma unroll` is used to automatically unroll loops in an effort to free some computational resources. Finally, we pre-calculate memory addresses given a set of specific input shapes for this MP. This also helps us to remove some function parameters and reduce overhead of address generation and kernel launching.

3. Reduction Tree

Runtime of the optimized program:

```
done
New Inference
Op Time: 0.006505
Op Time: 0.035581
Correctness: 0.8397 Model: ece408
4.53user 3.30system 0:04.47elapsed 175%CPU (0avgtext+0avgdata 2830308maxresident)
```

nvvp profiler analysis:

Name	Invocations	Avg. Duration	Regs	Static SMem	Avg. Dynamic SMem
memset (0)	0	0 ns	0	0	0
void mshadow::cuda::MapPlanKernel<mshadow::sv::saveto, int=8, msh	13	2.171 μ s	16	0	0
void mshadow::cuda::MapPlanKernel<mshadow::sv::saveto, int=8, msh	1	5.024 μ s	26	0	0
void mshadow::cuda::MapPlanKernel<mshadow::sv::plusto, int=8, msh	2	11.952 μ s	16	0	0
void mshadow::cuda::SoftmaxKernel<int=8, float, mshadow::expr::Plan	1	75.647 μ s	20	1024	0
void mshadow::cuda::MapPlanKernel<mshadow::sv::saveto, int=8	1	157.631 μ s	22	0	0
void op_generic_tensor_kernel<int=2, float, float, float, int=256, cudnn	2	810.92 μ s	16	0	0
void cudnn::detail::pooling_fw_4d_kernel<float, float, cudnn::detail::m	1	1.05529 ms	48	0	3872
void mshadow::cuda::MapPlanLargeKernel<mshadow::sv::saveto, int=8	2	1.21108 ms	22	0	0
volta_sgemm_32x128_tn	2	1.25477 ms	55	16384	0
mxnet::op::forward_kernel_reduction_tree(float*, float const *, float co	2	21.05445 ms	32	0	3584

Compute
84.2% mxnet::op::forward_kernel_reduction_tree(float*, float const *, float const *, int, int, int, ..
5.0% volta_sgemv_32x128_tn
4.8% void mshadow::cuda::MapPlanLargeKernel<mshadow::sv::saveto, int=8, int=1024, mshad...
3.2% void op_generic_tensor_kernel<int=2, float, float, float, int=256, cudnnGenericOp_t=7, cu...
2.1% void cudnn::detail::pooling_fw_4d_kernel<float, float, cudnn::detail::maxpooling_func<fl...
0.3% void mshadow::cuda::MapPlanLargeKernel<mshadow::sv::saveto, int=8, int=1024, mshad...
0.2% void mshadow::cuda::SoftmaxKernel<int=8, float, mshadow::expr::Plan<mshadow::Tensor...
0.1% void mshadow::cuda::MapPlanKernel<mshadow::sv::saveto, int=8, mshadow::expr::Plan<...
0.0% void mshadow::cuda::MapPlanKernel<mshadow::sv::plusto, int=8, mshadow::expr::Plan<...
0.0% void mshadow::cuda::MapPlanKernel<mshadow::sv::saveto, int=8, mshadow::expr::Plan<...
0.0% memset (0)

⚠ Low Memcpy/Kernel Overlap [0 ns / 16.87068 ms = 0%]

The percentage of time when memcpy is being performed in parallel with kernel is low.

⚠ Low Kernel Concurrency [0 ns / 44.6212 ms = 0%]

The percentage of time when two kernels are being executed in parallel is low.

⚠ Low Memcpy Throughput [266.492 MB/s avg, for memcpys accounting for 0.2% of all memcpy time]

The memory copies are not fully using the available host to device bandwidth.

⚠ Low Memcpy Overlap [0 ns / 5.728 μs = 0%]

The percentage of time when two memory copies are being performed in parallel is low.

Description:

The goal of this optimization is to reduce addition time for matrix convolution. In this optimization, instead of add the products directly during convolution, we store the temporary values of input channel(C dimension) in a shared memory with size $C * \text{TILE_SIZE} * \text{TILE_SIZE}$ and then using reduction tree method to compute sums of C dimension. This means that after doing reduction sum, shared memory (0, threadIdx.y, threadIdx.x) for all valid threadIdx.y, threadIdx.x will contain the final result and we just need to copy them back to output matrices.

Analysis:

In the project instruction, it mentions the optimization of input channel reduction using reduction tree. Nevertheless, we find out that there is no way to implement this feature because of the dimensions do not match. Instead, we find out that it is possible to use reduction tree inside convolution. Our method will speed up the add operations for matrix convolution, thus, when input channel dimension is very large, this optimization will have better performance. However, since we use another shared memory to store temporary values, it needs extra read and write to shared memory and there also exist more control divergence in thread blocks. Unfortunately, after carefully examine the input data, we found that input channel C is very small ($C < 10$ in most of the cases). Thus, the overhead for accessing share memory will overweight the speed up for add operations using reduction tree which gave us worse runtime compared with other optimization methods.

Extra Credit

Kernel fusion implementation

Runtime of the optimized program:






```
done
New Inference
Op Time: 0.005319
Op Time: 0.013196
Correctness: 0.8397 Model: ece408
4.54user 3.38system 0:04.33elapsed 181%CPU (0avg
```

nvvp profiler analysis:

Name	Invocations	Avg. Duration	Regs	Static SMem	Avg. Dynamic SMem
memset (0)	0	0 ns	0	0	0
void mshadow::cuda::MapPlanKernel<mshadow::sv::saveto, int=8, n	13	2.163 µs	16	0	0
void mshadow::cuda::MapPlanKernel<mshadow::sv::saveto, int=8, n	1	4.896 µs	26	0	0
void mshadow::cuda::MapPlanKernel<mshadow::sv::plusto, int=8, n	2	11.967 µs	16	0	0
void mshadow::cuda::SoftmaxKernel<int=8, float, mshadow::expr::F	1	75.391 µs	20	1024	0
void mshadow::cuda::MapPlanLargeKernel<mshadow::sv::saveto, ir	1	156.35 µs	22	0	0
void op_generic_tensor_kernel<int=2, float, float, float, int=256, cud	2	812.425 µs	16	0	0
void cudnn::detail::pooling_fw_4d_kernel<float, float, cudnn::detai	1	1.05497 ms	48	0	3872
void mshadow::cuda::MapPlanLargeKernel<mshadow::sv::saveto, ir	2	1.21133 ms	22	0	0
volta_sgemm_32x128_tn	2	1.24968 ms	55	16384	0
mxnet::op::forward_kernel_fusion_kernel_1(float*, float const *, int	1	5.03723 ms	30	0	1700
mxnet::op::forward_kernel_fusion_kernel_2(float*, float const *, int	1	12.15678 ms	31	0	2804

Compute
48.4% mxnet::op::forward_kernel_fusion_kernel_2(float*, float const *, i...
20.1% mxnet::op::forward_kernel_fusion_kernel_1(float*, float const *, i...
10.0% volta_sgemm_32x128_tn
9.7% void mshadow::cuda::MapPlanLargeKernel<mshadow::sv::saveto, i...
6.5% void op_generic_tensor_kernel<int=2, float, float, float, int=256, cu...
4.2% void cudnn::detail::pooling_fw_4d_kernel<float, float, cudnn::deta...
0.6% void mshadow::cuda::MapPlanLargeKernel<mshadow::sv::saveto, i...
0.3% void mshadow::cuda::SoftmaxKernel<int=8, float, mshadow::expr::...
0.1% void mshadow::cuda::MapPlanKernel<mshadow::sv::saveto, int=8, ...
0.1% void mshadow::cuda::MapPlanKernel<mshadow::sv::plusto, int=8, ...
0.0% void mshadow::cuda::MapPlanKernel<mshadow::sv::saveto, int=8, ...
0.0% memset (0)

Results

 Low Kernel / Malloc Efficiency [25.09654 ms / 18.13552 ms = 1.384]
The amount of time performing compute is low relative to the amount of time required for malloc.
 Low Malloc/Kernel Overlap [0 ns / 18.13552 ms = 0%]
The percentage of time when malloc is being performed in parallel with kernel is low.
 Low Kernel Concurrency [0 ns / 19.69649 ms = 0%]
The percentage of time when two kernels are being executed in parallel is low.
 Low Malloc Throughput [278.892 MB/s avg, for mallocs accounting for 0.2% of all malloc time]
The memory copies are not fully using the available host to device bandwidth.
 Low Malloc Overlap [0 ns / 5.856 μs = 0%]
The percentage of time when two memory copies are being performed in parallel is low.

Description:

When testing our code on unroll+gemm matrix multiplication implementation, we found that the kernel invocation times are extremely large. We have to invoke two kernel(unroll and gemm kernel, one for loading the data in to a device memory, the other for performing computation on the preprocessed data) separately for batch size times, whereas the batch is usually in large extent. We decide to fuse the kernel into one, that is, the kernel will perform both data preprocessing and matrix multiplication jobs in the same kernel. To improve the utilization of the global memory access, we decide to load tile of data (format of gemm matrix) in each iteration along the column of first matrix(equivalent to row of the second matrix) to shared memory(size of TILE_SIZE * TILE_SIZE). The weight of convolution filter kernel is loaded as well, and since the indexing for kernel is already in place, we don't have to perform address translation on that. Then in the following computation, we perform exactly the same thing as what we do in the __gemm kernel in milestone 3.

Analysis:

In the current implementation we use thread in each Block for loading and computing of one feature point in the output. The Block size is fixed with TILE*SIZE * TILE*SIZE. For Grid dimension z, we assign batch dimension along this axis, so each block is in charge of a block size feature map for certain B.

This kernel will dramatically improve the performance regarding to the plain implementation of unroll and gemm given the size of output channel and number of output feature are fit with TILE_SIZE. The main improvement is mainly due to reduced overhead for kernel launching..