CS 483/ECE 408: Applied Parallel Programming
the_cuda_corps
Larry Poon (Chicago), Gerald Kozel (Chicago), & Waleed Ramahi (UIUC)

Milestone 1

[kernel calls that collectively consume more than 90% of the program time]:

Time(%)	Time	Calls	Avg	Min	Max	Name
34.07%	118.49ms	9	13.166ms	13.149ms	13.180ms	fermiPlusCgemmLDS 128_batched
27.01%	93.927ms	1	93.927ms	93.927ms	93.927ms	cudnn::detail::implicit _convolve_sgemm
12.69%	44.128ms	9	4.9031ms	2.6906ms	6.2766ms	fft2d_c2r_32x32
8.2%	28.514ms	1	28.514ms	28.514ms	28.514ms	sgemm_sm35_ldg_tn _128x8x256x16x32
6.42%	22.331ms	14	1.5951ms	1.5360us	21.504ms	[CUDA memcpy HtoD]
4.07%	14.156ms	2	7.0781ms	252.38us	13.904ms	cudnn::detail::activati on_fw_4d_kernel

[CUDA API calls that collectively consume more than 90% of the program time.]:

Time(%)	Time	Calls	Avg	Min	Max	Name
43.61%	1.93912s	18	107.73ms	15.637us	969.21ms	cudaStreamCreateWi thFlags
27.11%	1.20548s	10	120.55ms	1.2190us	342.20ms	cudaFree
20.62%	917.01ms	27	33.963ms	236.99us	908.89ms	cudaMemGetInfo

API calls and kernel calls are both parts of the CUDA programming interface. Kernels are functions defined by the __global__ specifier which we are used to using in our homeworks. API calls are executed on the host, but have access to the global memory of the device. Kernel calls are executed on the device itself. The runtime API consists of cudaMemcpyTosymbol, cudaMalloc, and other functions used to access global variables. Generally the main tradeoff between the two is that kernel launches are more complex to implement but provide more fine-grained control while the runtime API makes device code management easier and cleaner. There's no significant performance difference between API and kernel calls, based on the statistics in the tables above.

[Output of rai running MXNet on the CPU]:

```
✱ Running /usr/bin/time python m1.1.py
Loading fashion-mnist data...
done
Loading model...
done
New Inference
EvalMetric: {'accuracy': 0.8444}
```

[m1.1.1.py program run time]:

```
13.26user 11.98system 0:11.66elapsed 216%CPU (0avgtext+0avgdata
2821748maxresident)k
0inputs+2624outputs (0major+38226minor)pagefaults 0swaps
```

[Output of rai running MXNet on the GPU]:

```
✱ Running /usr/bin/time python m1.2.py
Loading fashion-mnist data...
done
Loading model...
[23:46:06] 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
New Inference
EvalMetric: {'accuracy': 0.8444}
```

[m1.2.py program run time]:

```
2.29user 1.06system 0:02.82elapsed 118%CPU (0avgtext+0avgdata 1138624maxresident)k
0inputs+3136outputs (0major+153677minor)pagefaults 0swaps
```

CS 483/ECE 408: Applied Parallel Programming
the_cuda_corps
Larry Poon (Chicago), Gerald Kozel (Chicago), & Waleed Ramahi (UIUC)

Milestone 2

10,000 images

[m2.1.py program output]:

```
Loading fashion-mnist data...
done
Loading model...
done
New Inference
Op Time: 7.463296
Op Time: 25.670930
Correctness: 0.8451 Model: ece408
```

[m2.1.py program runtime]:

```
37.64user 1.29system 0:36.93elapsed 105%CPU (0avgtext+0avgdata 2813404maxresident)k
0inputs+0outputs (0major+35786minor)pagefaults 0swaps
```

100 images

[m2.1.py program output]:

```
Loading fashion-mnist data...
done
Loading model...
done
New Inference
Op Time: 0.074735
Op Time: 0.255830
Correctness: 0.88 Model: ece408
```

[m2.1.py program runtime]:

```
1.10user 0.58system 0:01.09elapsed 154%CPU (0avgtext+0avgdata 175624maxresident)k
0inputs+0outputs (0major+31500minor)pagefaults 0swaps
```

10 images

[m2.1.py program output]:

```
Loading fashion-mnist data...
done
Loading model...
done
New Inference
Op Time: 0.007455
Op Time: 0.025613
Correctness: 1.0 Model: ece408
```

[m2.1.py program runtime]:

```
0.76user 0.40system 0:00.72elapsed 160%CPU (0avgtext+0avgdata 169980maxresident)k
0inputs+0outputs (0major+29052minor)pagefaults 0swaps
```

Milestone 3

10,000 images

[m3.1.py program output]:

```
✱ Running python m3.1.py
Loading fashion-mnist data...
done
Loading model...
done
New Inference
Op Time: 0.203084
Op Time: 0.508383
Correctness: 0.8451 Model: ece408
```

100 images

[m3.1.py 100 program output]:

```
✱ Running python m3.1.py 100
Loading fashion-mnist data...
done
Loading model...
done
New Inference
Op Time: 0.001499
Op Time: 0.004704
Correctness: 0.88 Model: ece408
```

10 images

[m3.1.py 10 program output]:

```
✱ Running python m3.1.py 10
Loading fashion-mnist data...
done
Loading model...
done
New Inference
Op Time: 0.000259
Op Time: 0.000765
Correctness: 1.0 Model: ece408
```

Our approach to optimization

The sequential implementation involves nested loops (7 for-loops), we wanted to implement a basic parallel implementation to speed up the execution in a way that workload are evenly distributed among thread blocks. We first layout the output feature maps into a giant output feature map, with dimension of (B * H_out , M * W_out), where B is the number of batches and M is the number of output feature maps in each batch. Then, we distribution the tasks in the thread grid, such that each thread would process a output of a corresponding output feature map. The time complexity is reduced because the work is now evenly distributed, and each thread will have O(C*K*K) work, in contrast, it had O(B*M*H*W*C*K*K) amount of work for a single process originally.

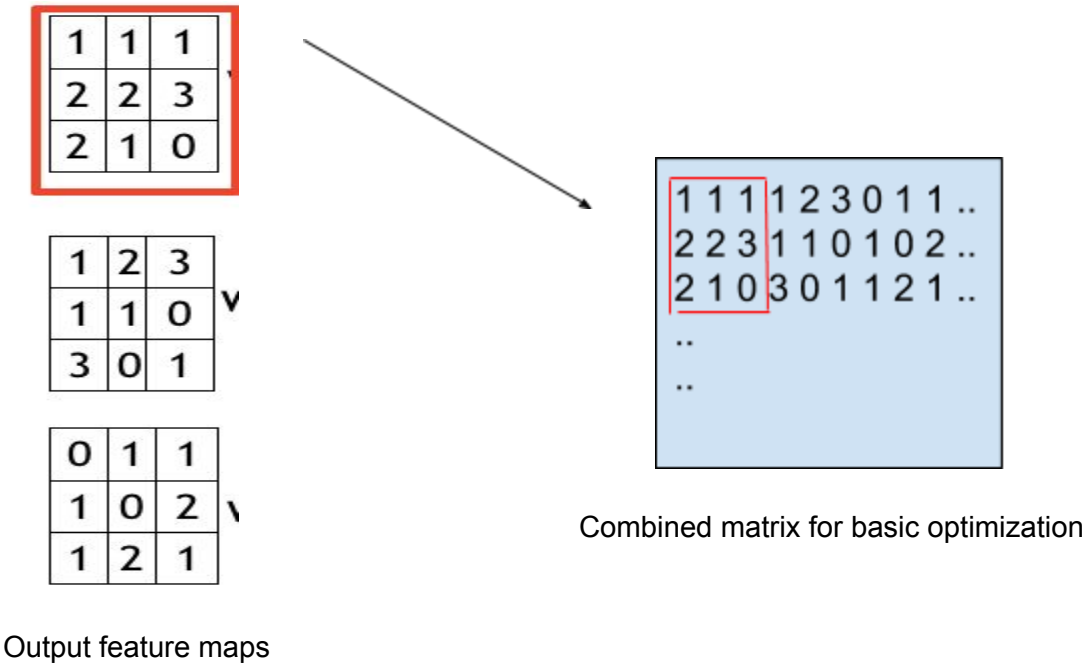

Figure 1. Example of mapping between output feature maps to a matrix used for optimization in this milestone

Initial Performance Results

Unsurprisingly, the basic, parallel implementation has improved in time complexity compare to the sequential version. In milestone 2, the program ‘m2.1.py 100’ took about 0.074735 in the first layer and 0.255830 for the second layer convolution, while in milestone 3, with a simple, parallel implementation, operation time has decreased to 0.002174 and 0.004704, respectively (the execution details generated by nvvp are shown below).

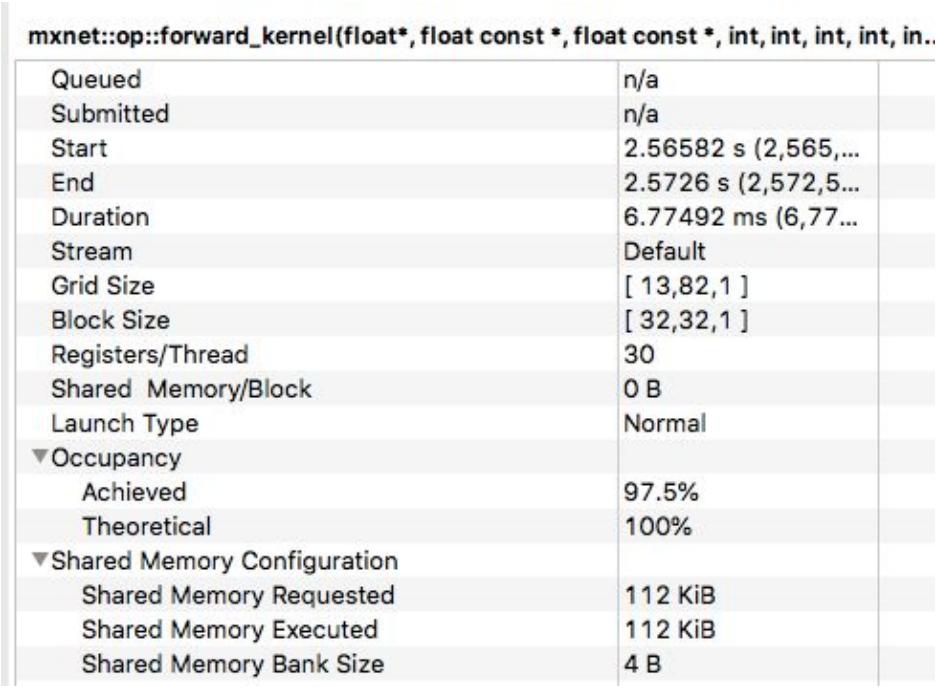

Figure 2. Kernel execution summary

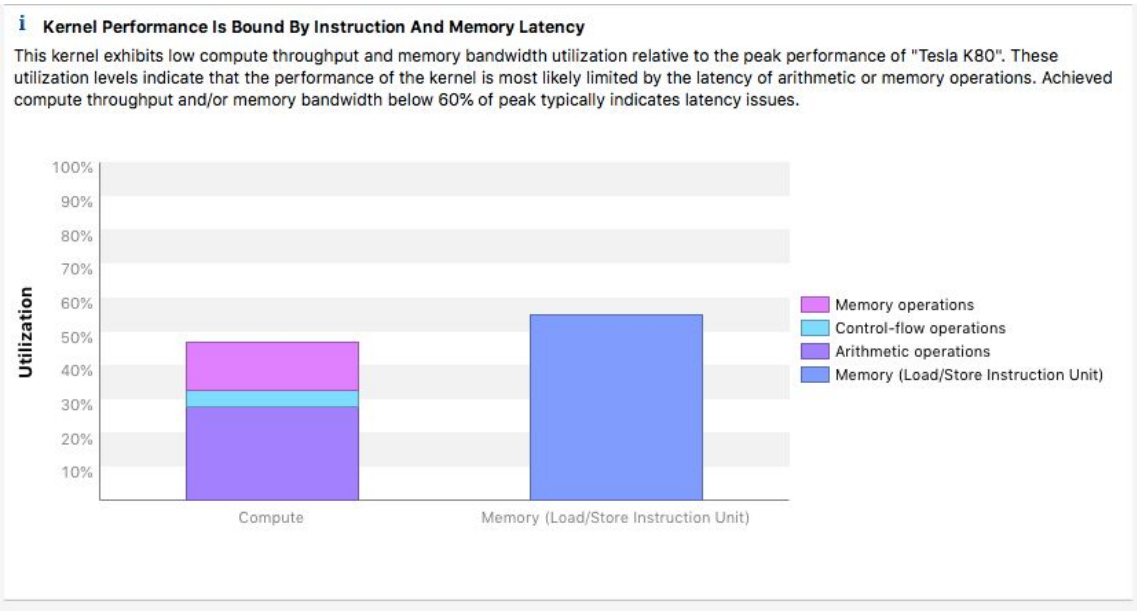

Figure 3. “Forward_kernel” individual kernel analysis

Results

Memory Bandwidth And Utilization

The following table shows the memory bandwidth used by this kernel for the various types of memory on the device. The table also shows the utilization of each memory type relative to the maximum throughput supported by the memory. More...

	Transactions	Bandwidth	Utilization
L1/Shared Memory			
Local Loads	0	0 B/s	
Local Stores	0	0 B/s	
Shared Loads	0	0 B/s	
Shared Stores	0	0 B/s	
Global Loads	19919250	149.725 GB/s	
Global Stores	102200	896.265 MB/s	
Atomic	0	0 B/s	
L1/Shared Total	20021450	150.622 GB/s	Idle Low Medium High Max
L2 Cache			
L1 Reads	31707000	149.725 GB/s	
L1 Writes	189800	896.265 MB/s	
Texture Reads	0	0 B/s	
Noncoherent Reads	0	0 B/s	
Atomic	0	0 B/s	
Total	31896800	150.622 GB/s	Idle Low Medium High Max
Texture Cache			
Reads	0	0 B/s	Idle Low Medium High Max
Device Memory			
Reads	104244	492.256 MB/s	
Writes	218215	1.03 GB/s	
Total	322459	1.523 GB/s	Idle Low Medium High Max
ECC Overhead	116425	549.777 MB/s	

Figure 4. Memory bandwidth and Utilization