CME213 Final Project Final Report

Brad Huang

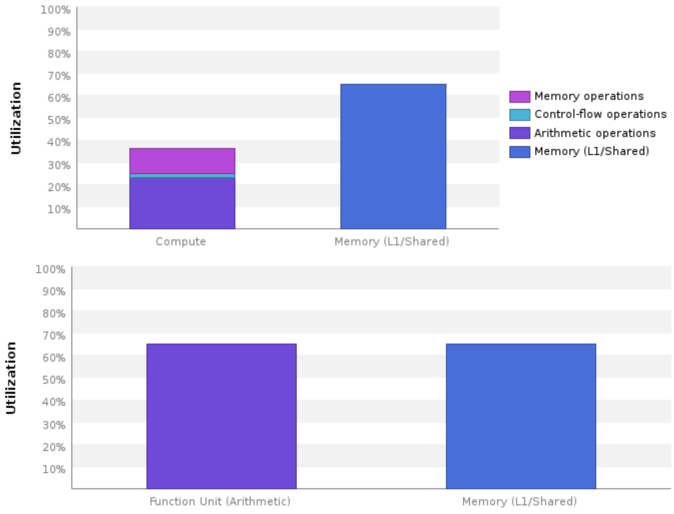
June 2016

1 GEMM Optimization

I implemented the optimization of GEMM as described in the Appendix of the final project's handout 1. With some optimization on unrolling the for loops and caching intermediate results, the resulting running times are:

M	N	K	Iteration	cuBLAS	myGEMM	My Runtime / cuBLAS
800	1000	784	10	0.0468709	0.0857	1.828
800	10	1000	10	0.0043869	0.00665593	1.517
1600	2000	1568	100	3.26495	6.47858	1.984
1600	20	2000	100	0.160203	0.172668	1.078

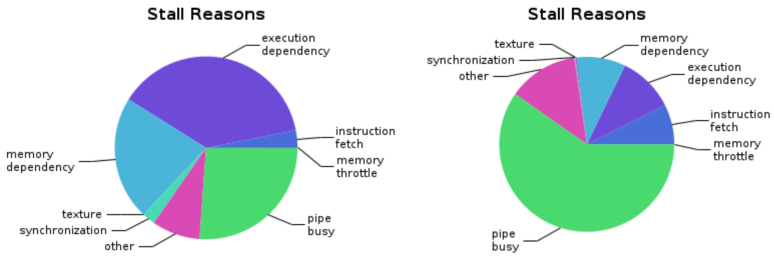
When analyzing the first two runs under NVVP, 67.2% of GPU time is spent on my kernel, and 32.8% is spent on cuBLAS' kernel. Let's compare the utilizations of the two:



We can see that my implementation is still memory bound, while cuBLAS' implementation is not bound by memory or computation. Nonetheless, the memory access pattern for the optimized GEMM is closer to cuBLAS' GEMM. Compared to the unoptimized version (one thread for each element in D), my optimized GEMM has a much higher shared memory bandwith, which is close to that of cuBLAS:

Statistics or Bandwidth (GB/s)	Naive GEMM	Optimized GEMM	cuBLAS
L1 Local Load	0	653.496	0
L1 Local Store	0	1.432	0
Shared Load	0	574.072	596.793
Shared Store	0	35.586	75.36
L1 Global	642.307	72.002	2.954
L1 Global	18.832	713.735	3.007
L2 Cache Read	116.101	52.974	3.007
L2 Cache Write	18.832	2.201	3.007
L2 Texture Cache	N/A	N/A	66.184
Global L2 Transaction/Access	114.3	32	16
Shared Transaction/Access	N/A	8	N/A

From the latency analyses of the two kernels, we can see that, to further optimize this kernel, we'll have to increase the compute utilization - such as potentially removing some unnecessary checks or paralleling the method further (since each thread handles 16 elements in D, the parallelization is not as good). Or maybe include the use of texture memory, as the memory statistics of the cuBLAS kernel shows that it is using texture memory.



The left is my optimized GEMM, and the left is cuBLAS.

2 Neural Network Optimization

2.1 MPI Optimization

I did not make any changes to the MPI part of the NN implementation because my method is the one with the least memory transactions I can think of. I broadcast all of W^T, b and X^T, y^T to each process before the training starts, and copy all of them into GPU memory. During each iteration, I compute the ranges of samples/columns that each process should be responsible for. This way each process can directly index into their GPU copies of X^T and y^T (since the matrices are column major) to get the samples without needing to scatter the data every time. After feedforward and backpropagation for each process, I only have to copy the gradients, two copies of dW and db, from GPU tp CPU, perform an `MPI_Allreduce()` on the gradients, and then copy the reduced gradients back to the GPU to update each process' own GPU copies of W and b. I think this is the minimal amount of memory transfer, both between processes and between CPU and GPU, that we can get for this MPI problem.

2.2 CUDA Kernel Optimization

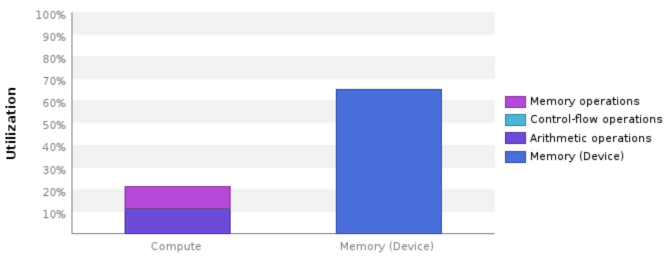
As outlined in my previous report, I have chosen the four kernels occupying the most GPU time to optimize. These are the statistics from the previous report:

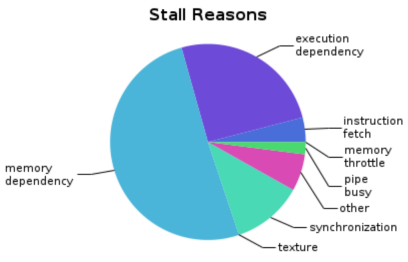
Wrapper	Kernel	Time	Compute Util	Memory Util	Main Stall Reason
myAllocGEMM	simpleAllocGEMM	74.3	30	65	Pipe Busy
computeZ	gpuVecGEMM	18.7	48	25	Pipe Busy
columnSums	gpuColSums	4.9	8	5	Execution Dependency
transpose	gpuTranspose	1.7	12	65	Memory Dependency

2.2.1 myAllocGEMM

Since the `simpleAllocGEMM` is very similar to GEMM and the only difference is that the result is not stored in place, I simply transferred the implementation of `myGEMM` to the kernel.

The resulting kernel is still memory bound, with slightly lower compute utilization. The reduced compute utilization comes from the fact that each thread now computes 16 elements in the resulting matrix, while previously one thread handles only one element, leading to higher parallelization. However, the kernel now has a much higher L1/Shared memory bandwidth because of the introduction of shared memory in the kernel.



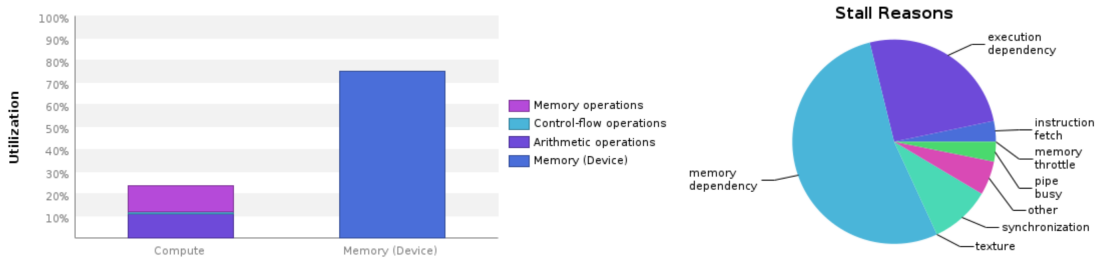


The main stall reason is now memory dependency and execution dependency. The memory dependency may be due to the increased use of arrays in the kernel and the use of shared memory. Execution dependency probably comes from more branching and synchronization required for the kernel.

2.2.2 computeZ

similar to `myAllocGEMM`, I also transferred the implementation of `myGEMM` to the kernel `gpuVecGEMM`.

The result is very similar to the previous section, except that the memory utilization is even higher for this kernel - which is reasonable since the memory access to the matrix C in the previous kernel is more expensive than accessing a vector b in this kernel.

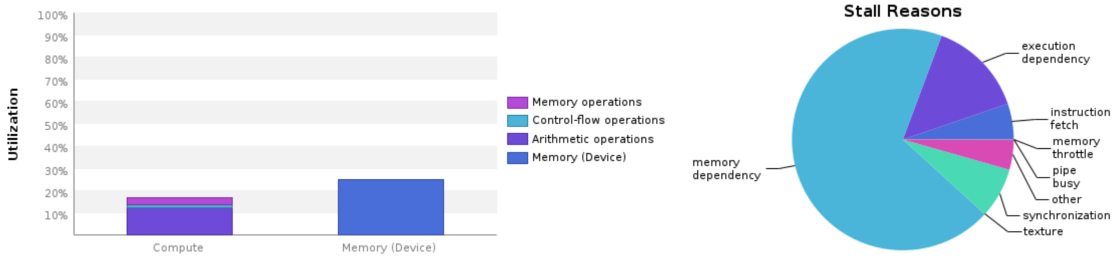


2.2.3 columnSums

I adopted the reduction code presented in lecture to optimize this kernel. For each column, I allocated one 256x1 block for it. Each of the 256 threads compute a partial sum based on the length of the column, and then I used a shared memory array to compute the final total sum.

The compute and memory utilizations are two to five times higher than those of the previous kernel, although the kernel is still latency bound. The reason is that the task is inherently smaller and requires fewer threads to perform it, so utilization of the full GPU is unlikely.

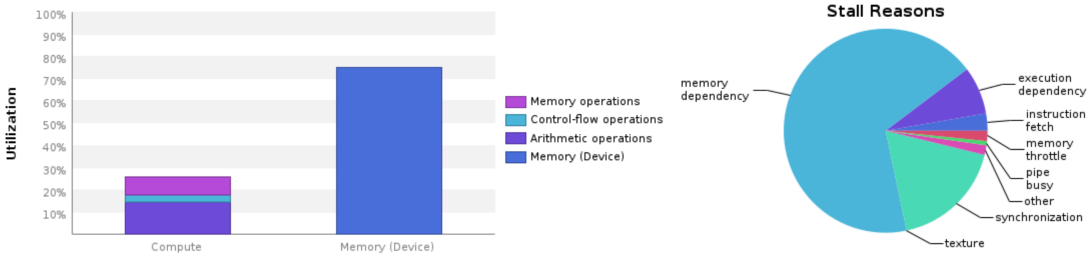
The main stall reason has shifted from execution dependency to memory dependency. The optimized kernel has more parallelization, so the execution dependency is decreased, and memory dependency prevails.



2.2.4 transpose

I adopted the transpose code presented in lecture here, except that the global memory reading and writing has been modified to be coalesced when the matrix is using column major storage.

The compute and memory utilizations are higher than those of the previous kernel by about 10%, although the kernel is still memory bandwidth bound.



2.3 Overall Changes

The overall runtime of the neural network is only improved very slightly, around 10% in speedup, with the kernel optimizations:

Grading Mode	CPU Runtime	GPU Runtime	Speedup	Previous Speedup
1	118.584	41.8269	2.833	2.563
2	29.7955	11.5513	2.579	2.423
3	23.8216	2.06716	11.524	8.945

However, the optimization does show up in the NVVP kernel analysis. The top seven kernels occupying the most time are:

Wrapper	Kernel	Time (%)
myAllocGEMM	allocGEMM	55.8
computeZ	gpuVecGEMM	41.0
transpose	fastTranspose	2.3
elemMults	gpuElemMults	0.4
mySigmoid	gpuSigmoid	0.3
columnSums	gpuColSums	0.2
matAdd	gpuMatAdd	0.1

The optimization priorities of `allocGEMM` and `gpuVecGEMM` are now much higher than those of the rest kernels. The kernels are called the most number of times and are responsible for the most complicated computations among all of the kernels, so it is natural that they occupied the most time and have the highest priorities. But `gpuColSums` and `fastTranspose` have been optimized enough that they are now not more prioritized than the other kernels. The increased percentage of `transpose` time is only due to the larger reductions in the other optimized kernels.