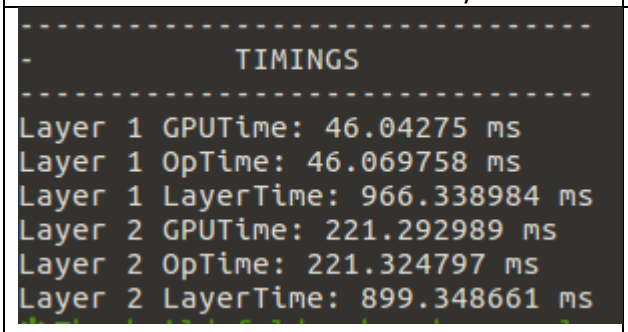
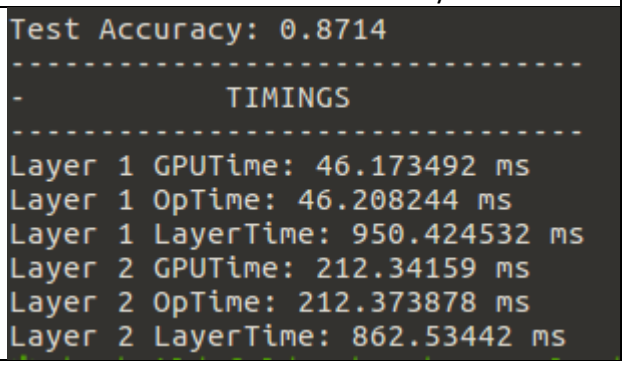
PM4
Jimmy Zhang
Andrew Gaeck
Martin Juskelis

Optimization: Constant Memory of the Weight Kernels

Identified: Using Constant memory is a topic covered in depth in class, including in the convolution lectures. No profiling required to approach this.

Why it would be fruitful: Constant memory should speed up the convolution as each thread should only be reading half the number of global memory elements (reading only data, not kernel as well). Less global memory reads should improve run time.

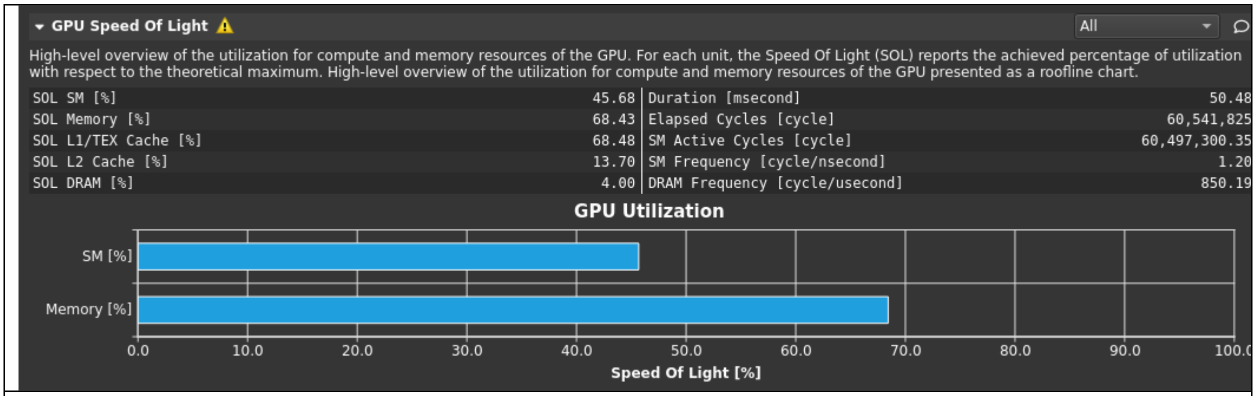
Effect: This optimization had the intended effect on the runtime, reducing the amount of time the CPU runs. However, the effect on GPU runtime isnegligible. This is evident in the attached timing results. These timings were run multiple times with constant memory outperforming in CPU times.

Without Constant Memory	With Constant Memory
	

The effect of using Constant memory was not as great as we anticipated. This may be because the global reads on the kernel data were already pretty efficient (With all threads in a warp using the same memory value) and not much time was spent on these global reads.

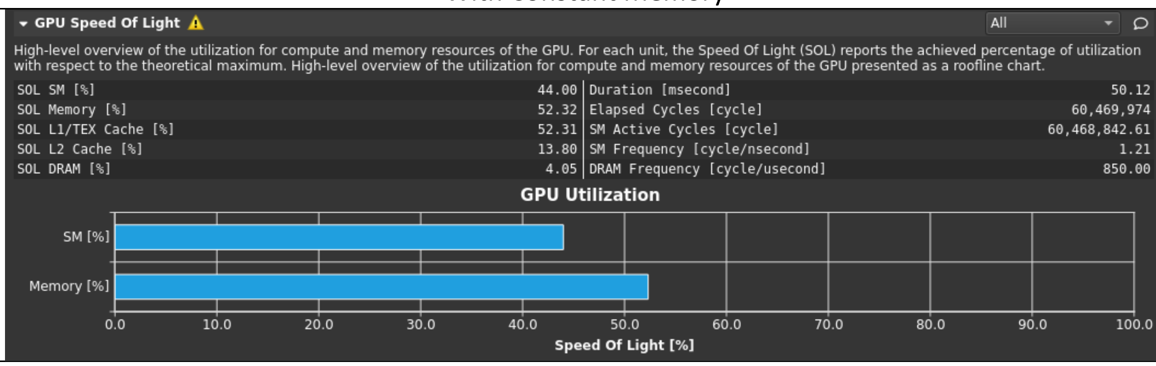
What's interesting is the nv-sight-cu output. Here, nv-sight is stating that the performance decreases when using constant memory (although it correctly states that constant memory takes less time to run). This may be because the global memory reads on the kernel data were highly efficient, inflating the SOL values on the memory. This would explain why even though constant memory isn't affecting the run time, the SOL of memory decreases by about 15%.

Without Constant Memory



XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

With Constant Memory



From Nsys we come up with some explanations for this behavior. The time spent in the convolution kernel is unaffected constant memory. Furthermore, the time spent on cudaMemcpyToSymbol is less than the time saved on cudaMalloc and cudaMemcpy. Improvements to the convolution kernel aside, using constant memory is worthwhile just for the time saved in allocating and writing memory for the kernels.

Furthermore, the pipes are more active with the constant memory approach, meaning we are better utilizing our memory bandwidth.

References: None. Extensively covered in lecture to the point where we were able to implement without even using any documentation.

Organization: This optimization was simple, so only one member (Andrew) ended up working on it. The difficulty was in merging this with the other optimizations.

Optimization: Input and Kernel Unrolling and Tiled Matrix Multiplication

Identified: Baseline convolution is highly parallel but loads the same value multiple times from global memory.

Why it would be fruitful: By unrolling the input and kernels, we can use matrix multiplication instead of convolution. Matrix multiplication is more straightforward, and has be optimized for less flow divergence, and tiling through shared memory can increase data reuse, reducing global memory reads.

Effect: We encountered an issue where at high batch sizes > 6000, the unrolled x matrix becomes too large, and we run out of memory. We intend to implement built in unrolling as a future optimization, which avoids the need of saving a giant unrolled x matrix.

For now we just simply split the convolution into batches of 2000 so that we don't use too much memory:

```
Test Accuracy: 0.8716
--------------------------------
-            TIMINGS
--------------------------------
Layer 1 GPUTime: 1077.795718 ms
Layer 1 OpTime: 1590.93109 ms
Layer 1 LayerTime: 1950.31598 ms
Layer 2 GPUTime: 783.023658 ms
Layer 2 OpTime: 1160.855999 ms
Layer 2 LayerTime: 1438.295952 ms
```

Despite the optimization, performance was quite worse. We think this is due the additional global memory stores and writes from the unrolling, where the overhead of the additional global memory accesses from the unrolling kernels overshadows and benefit from using shared memory.

From Nsys, we can see that most of the GPU time is at x_unroll, So, most of the additional time is overhead from the unrolling.

Time(%)	Total Time	Instances	Average	Minimum	Maximum	Name
83.8	2042944416	10	204294441.6	129166807	280598929	x_unroll
16.2	395834466	10	39583446.6	29912160	51884194	matrixMultiplyShared
0.0	5312	2	2656.0	2528	2784	k_unroll
0.0	2944	2	1472.0	1408	1536	do_not_remove_this_kernel
0.0	2816	2	1408.0	1280	1536	prefn_marker_kernel

However the time spent in the matrix multiplication (503 ms) is less than the baseline convolution.

So we hope that when we add kernel fusion for the unrolling, speedup can be recovered.

References: None. Extensively covered in lecture to the point where we were able to implement without even using any documentation.

Organization: This optimization was done by Jimmy Zhang

Optimization: Fixed point (FP16) arithmetic

Identified: It was mentioned as a potential optimization in the README

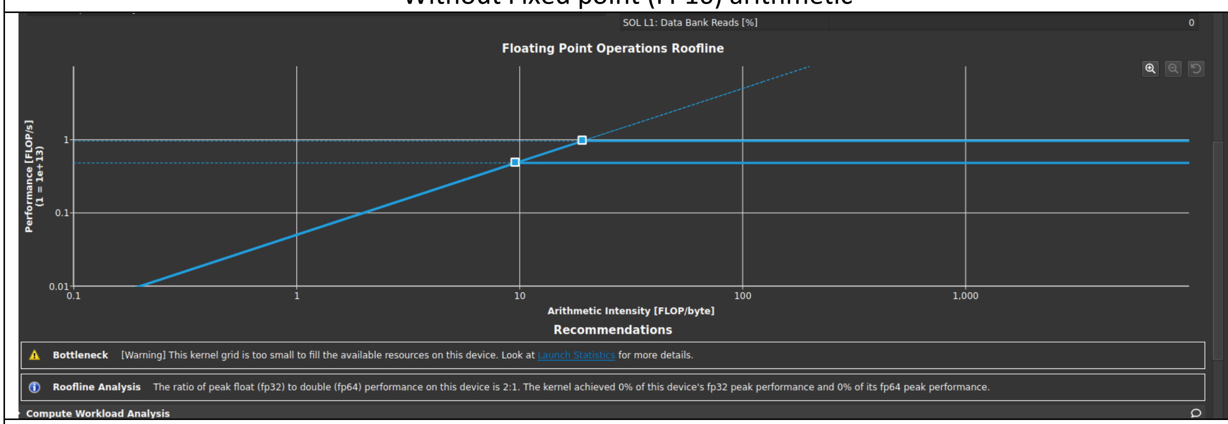
Why it would be fruitful: When doing floating point operations, GPUs can see 2X-8X speedup on FP16 over FP32 and it uses half the space as a normal float (source linked in references).

Effect: The optimization caused a small performance increase - it took a few percent off the running time.

Without Fixed point (FP16) arithmetic	With Fixed point (FP16) arithmetic
Test Accuracy: 0.8714 -------------------------------- - TIMINGS -------------------------------- Layer 1 GPUTime: 1173.196895 ms Layer 1 OpTime: 1704.23812 ms Layer 1 LayerTime: 2072.338205 ms Layer 2 GPUTime: 926.924287 ms Layer 2 OpTime: 1330.130839 ms Layer 2 LayerTime: 1601.007262 ms	Test Accuracy: 0.8716 -------------------------------- - TIMINGS -------------------------------- Layer 1 GPUTime: 1167.023317 ms Layer 1 OpTime: 1678.613633 ms Layer 1 LayerTime: 2039.645427 ms Layer 2 GPUTime: 898.606913 ms Layer 2 OpTime: 1295.848503 ms Layer 2 LayerTime: 1570.401161 ms

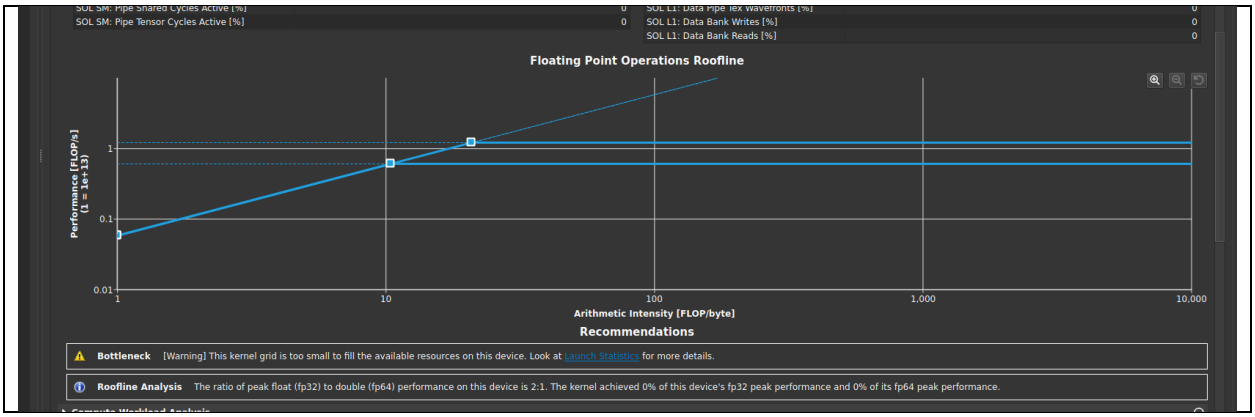
The effect is not large, which implies that the floating point operations are not the bottleneck and that there must be some other operation (such as memory reads) that is the limiting factor.

Without Fixed point (FP16) arithmetic



XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

With Fixed point (FP16) arithmetic



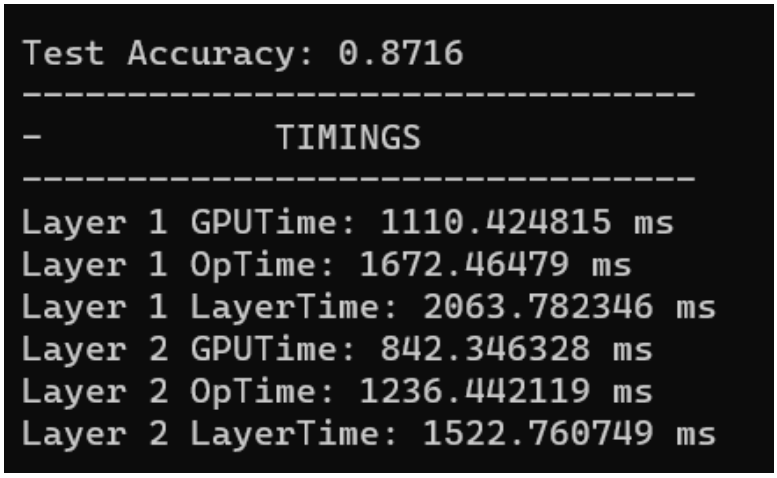
From Nsys we can see that the FLOPS are indeed a little bit faster, but there are other bottlenecks.

References: https://medium.com/@prrama/an-introduction-to-writing-fp16-code-for-nvidias-gpus-da8ac000c17f

Organization: One team member (Martin) worked on this optimization.

All optimizations combined:

Combining tiled matrix multiplication, constant weight memory, and fp16 we get the following timings:



These times are roughly equivalent to just the unrolling optimization. Again there is significant overhead from loop unrolling. Kernel fusion should expose benefits of constant memory.