Name: *Chia-Chi, Chen*
NetID: *chiachi5*
Section: *AL1*

ECE 408/CS483 Milestone 3 Report

0. List Op Times, whole program execution time, and accuracy for batch size of 100, 1k, and 10k images from your basic forward convolution kernel in milestone 2. This will act as your baseline this milestone.

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.240738ms*	*0.705865ms*	*1.224s*	*0.86*
1000	*1.77481ms*	*6.34676ms*	*10.029s*	*0.886*
10000	*15.8306ms*	*61.3222ms*	*1m38.320s*	*0.8714*

1. **Optimization 1: *Kernel fusion for unrolling and matrix-multiplication***

   a. Which optimization did you choose to implement and why did you choose that optimization technique.

   Kernel fusion for unrolling and matrix-multiplication

   After I tried two separate kernels, the first kernel unrolling matrix k and matrix x, followed by the second kernel tiled matrix multiply, the performance did not improve much. To transform the convolution into matrix multiply, there are some overheads allocating memory for the unrolling matrix and performing the unrolling transformation. Kernel fusion seems to be an appropriate approach to reduce these overheads.

   b. How does the optimization work? Did you think the optimization would increase performance of the forward convolution? Why? Does the optimization synergize with any of your previous optimizations?

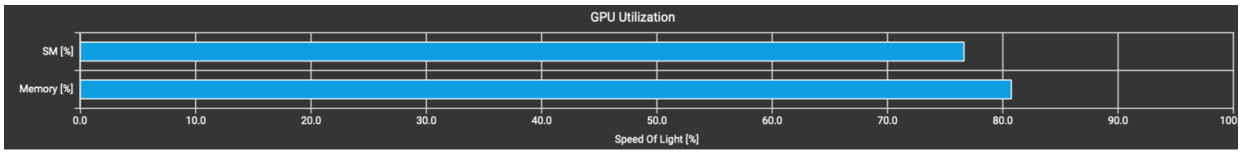
   The kernel fusion method combines both unrolling and tiled matrix multiply into one single kernel. The kernel unrolls the matrix while fetching the inputs tiles into the shared memory. This eliminates extra memory to be allocated for temporary unrolled matrix, eliminates the overhead for launching another kernel for unrolling, and also eliminates one global memory read and write for data transferring in the unrolling kernel. This optimization is a further improvement base on the two separate kernel - Shared memory matrix multiplication and input matrix unrolling.

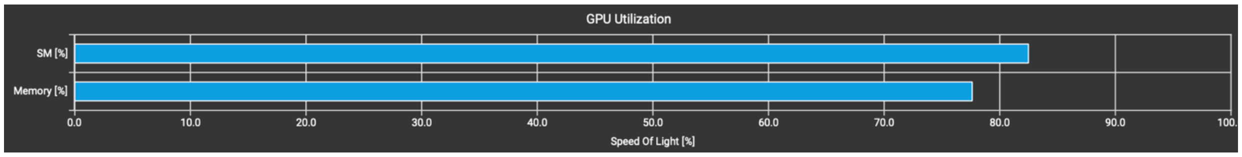
c. List the Op Times, whole program execution time, and accuracy for batch size of 100, 1k, and 10k images using this optimization (including any previous optimizations also used).

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.651913ms*	*0.528109ms*	*1.221s*	*0.86*
1000	*4.5806ms*	*3.52363ms*	*10.177s*	*0.886*
10000	*43.897ms*	*33.3302ms*	*1m38.062s*	*0.8714*

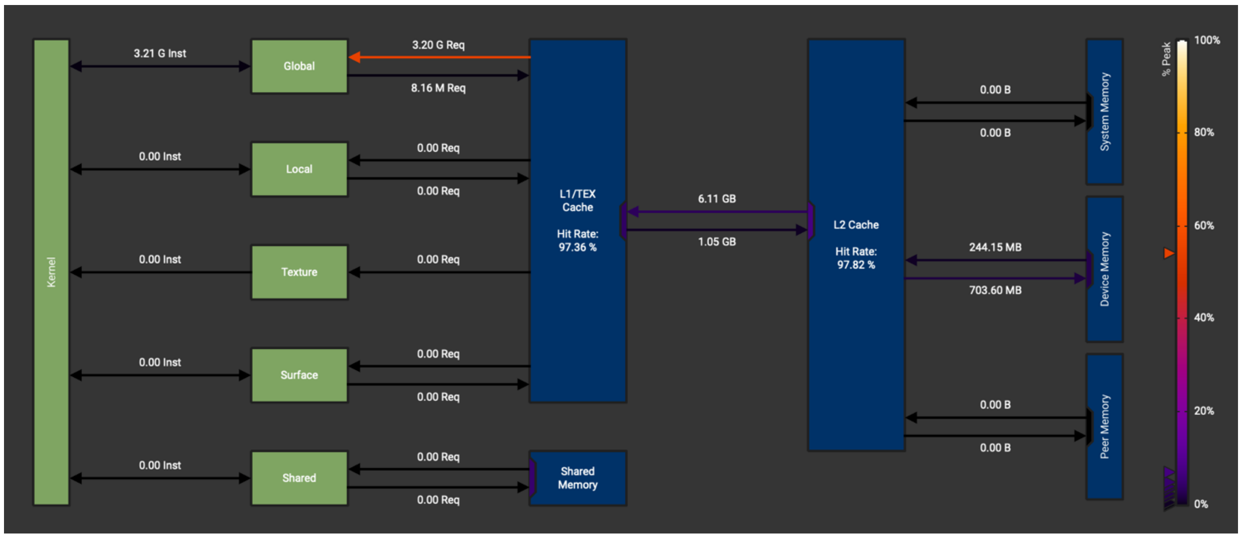
d. Was implementing this optimization successful in improving performance? Why or why not? Include profiling results from *nsys* and *Nsight-Compute* to justify your answer, directly comparing to your baseline (or the previous optimization this one is built off of).

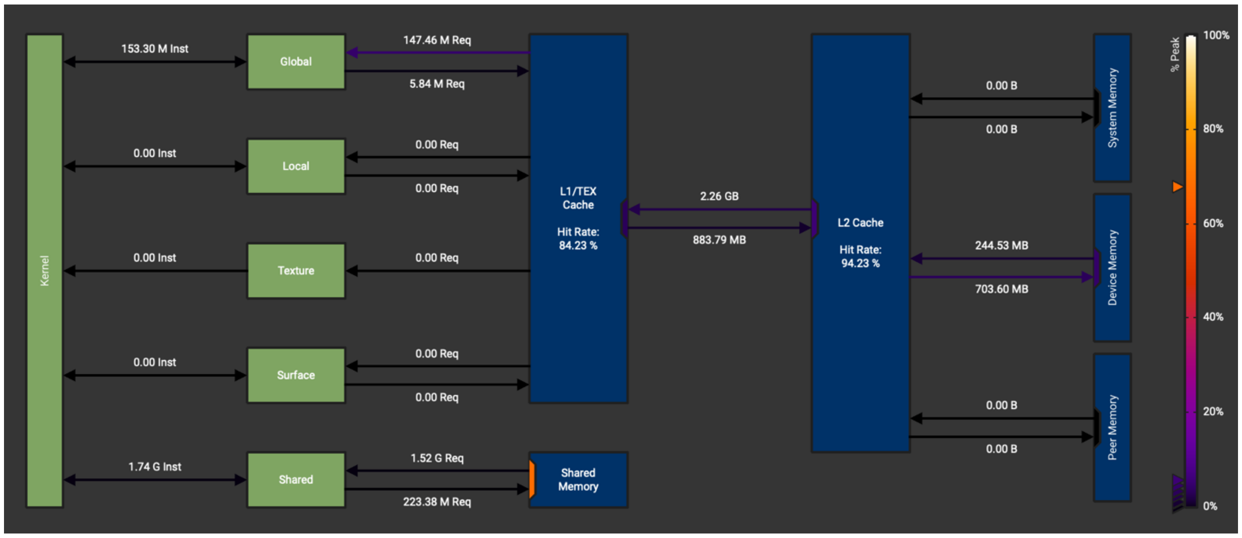
The first screenshot shows the basic convolution. The second screenshot shows the kernel fusion optimization. The SM increases, which implies better usage of the GPU computation, while the memory decreases.





The first screenshot shows the basic convolution. The second screenshot shows the kernel fusion optimization. We can see that the basic convolution has very high global memory access while the L1 Cache and L2 Cache hit rate is extremely high. On the other hand, the kernel fusion kernel has a much lower global memory access and the hit rate is also lower. This explains why the memory drops as the previous screenshots show. Moreover, we can see that the kernel fusion kernel has utilized quite a lot on the shared memory.





e. What references did you use when implementing this technique?

Lecture slide 13 on week 7.

2. **Optimization 2: *Multiple kernel implementations for different layer sizes***

   a. Which optimization did you choose to implement and why did you choose that optimization technique.

      Multiple kernel implementations for different layer sizes

      Comparing the basic convolution kernel and the kernel fusion kernel, we can see that the kernel fusion only optimized the second layer. Since the two layers have different input dimensions, it is reasonable to implement two separate kernels for the two different layers.

   b. How does the optimization work? Did you think the optimization would increase performance of the forward convolution? Why? Does the optimization synergize with any of your previous optimizations?

      We can easily separate the two different layers by the value M. For the first layer, we will launch the basic convolution kernel. On the other hand, we will launch the kernel fusion kernel for the second layer. By separating the kernels for different layers, we can gain the benefit of both kernels for different input dimensions.

   c. List the Op Times, whole program execution time, and accuracy for batch size of 100, 1k, and 10k images using this optimization (including any previous optimizations also used).

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.251549ms*	*0.417646ms*	*1.148s*	*0.86*
1000	*1.67603ms*	*3.39131ms*	*9.990s*	*0.886*
10000	*15.8337ms*	*33.1199ms*	*1m38.643s*	*0.8714*

d. Was implementing this optimization successful in improving performance? Why or why not? Include profiling results from *nsys* and *Nsight-Compute* to justify your answer, directly comparing to your baseline (or the previous optimization this one is built off of).

Since this is just a simple observation from the previous attempts, I'll not include nsys nor Nsight-Compute here. This optimization results from combining the best performance of the previous two kernels to reduce the overall time.

e.

method	Op Time 1	Op Time 2
Basic Convolution	*15.8306ms*	*61.3222ms*
Kernel Fusion	*43.897ms*	*33.3302ms*
Multiple Kernel	*15.8337ms*	*33.1199ms*

f. What references did you use when implementing this technique?

   This technique does not need any references.

3. **Optimization 3: Sweeping various parameters to find best values (block sizes, amount of thread coarsening) + *Weight matrix (kernel values) in constant memory***

   a. Which optimization did you choose to implement and why did you choose that optimization technique.

      1. *Sweeping various parameters to find best values (block sizes, amount of thread coarsening)*
      2. *Weight matrix (kernel values) in constant memory*

      We should be able to improve the performance by exploiting different combinations of the parameters to find the optimal parameters for each kernel. I included constant memory here because I also tried different combinations in using constant memory for the weight matrix.

   b. How does the optimization work? Did you think the optimization would increase performance of the forward convolution? Why? Does the optimization synergize with any of your previous optimizations?

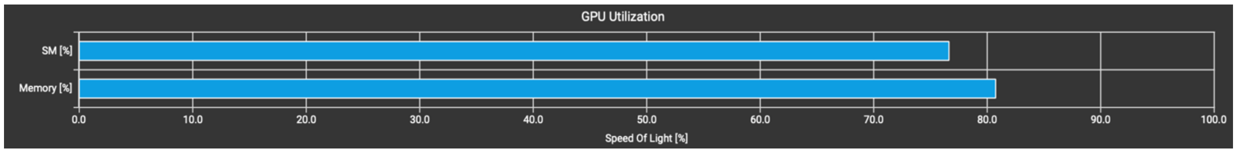
Different parameters will lead to different performance. Therefore, our goal is to find the optimal parameters by trying each combination one by one. This is possible only because task we are working on will not change and all the implementation is fixed. I found that for tile width of 16 performs the best among others. Also, my experiments show that using constant memory on the first layer will improve the performance, while the second layer will not. I think it is because the size of weight matrix is way larger in the second layer, so the overhead copying data is greater. Moreover, the second layer utilizes the shared memory for the weight matrix, which works similar as the constant memory.

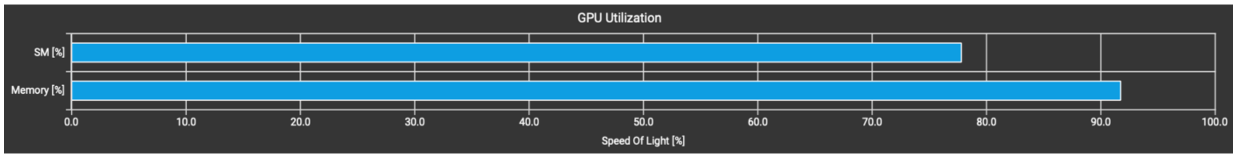
c. List the Op Times, whole program execution time, and accuracy for batch size of 100, 1k, and 10k images using this optimization (including any previous optimizations also used).

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.310946ms*	*0.543687ms*	*1.177s*	*0.86*
1000	*1.10464ms*	*3.36385ms*	*10.092s*	*0.886*
10000	*10.2514ms*	*32.9557ms*	*1m39.118s*	*0.8714*

d. Was implementing this optimization successful in improving performance? Why or why not? Include profiling results from *nsys* and *Nsight-Compute* to justify your answer, directly comparing to your baseline (or the previous optimization this one is built off of).

Compared with the basic convolution (first screenshot), we can see that the memory utilization increases. This is because we use the constant memory for weight matrix in the first layer.





e. What references did you use when implementing this technique?

   This technique does not need any references.

4. **Optimization 4:** ***Using Streams to overlap computation with data transfer***

   a. Which optimization did you choose to implement and why did you choose that optimization technique.

      <u>*Using Streams to overlap computation with data transfer*</u>

      We can see from that the Op time is roughly 35ms, whereas the layer time is around 600ms. This shows that the overall time is dominated by the memory transfer between CPU and GPU. To further improve the performance we should overlap computation with data transferring.

   b. How does the optimization work? Did you think the optimization would increase performance of the forward convolution? Why? Does the optimization synergize with any of your previous optimizations?

      We first create 3 streams for copying to device, kernel computation, and copying from device. We then divide the input X into several partitions. We can start our kernel immediately when one of the partitions is transferred to the GPU. Similarly, we transfer the results back as long as the computation is complete.

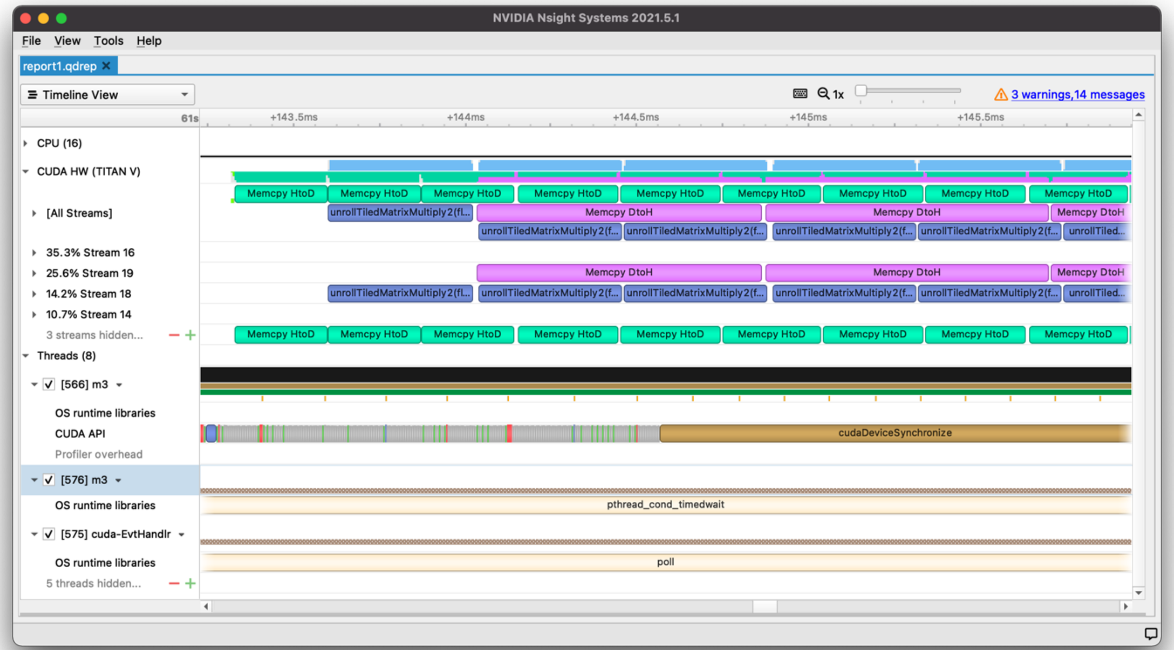
   c. List the Op Times, whole program execution time, and accuracy for batch size of 100, 1k, and 10k images using this optimization (including any previous optimizations also used).

      Since I implement all of this optimization code within conv_forward_gpu_prolog, the Op time is not accurate. Therefore, I'll list the Layer time instead.

Batch Size	Layer Time 1	Layer Time 2	Total Execution Time	Accuracy
100	*3.424ms*	*2.81375ms*	*1.199s*	*0.86*
1000	*32.2723ms*	*25.1438ms*	*9.8362s*	*0.886*
10000	*332.67ms*	*213.475ms*	*1m38.352s*	*0.8714*

d. Was implementing this optimization successful in improving performance? Why or why not? Include profiling results from *nsys* and *Nsight-Compute* to justify your answer, directly comparing to your baseline (or the previous optimization this one is built off of).

As we can see from nsys, we successfully overlap the memory copy of 2 directions (host to device and device to host) and also the kernel computation.



Moreover, we can see that the time spend on memcpy from device to host is almost thrice as much as the time on memcpy from host to device. This means that we'll have limited overlapping improvement, and the rest of the time will be the memcpy from device to host. Therefore, I also overlap the time spending on host, calling cudaHostRegister four times as I partitioned the host_y array. As a result, we will not have to wait until the whole array is pinned before we execute the memcpy and our kernel functions.

e. What references did you use when implementing this technique?

1. *https://developer.nvidia.com/blog/how-optimize-data-transfers-cuda-cc/*
2. *https://developer.nvidia.com/blog/how-overlap-data-transfers-cuda-cc/*
3. *https://blog.csdn.net/junparadox/article/details/50633641*
4. *https://stackoverflow.com/questions/31450020/why-not-cudamemcpyasynchost-to-device-and-cuda-kernel-are-parallel*
5. *https://github.com/NVIDIA-developer-blog/code-samples/blob/master/series/cuda-cpp/overlap-data-transfers/async.cu*
6. *https://github.com/cwpearson/nvidia-performance-tools*

5. **Other Optimizations**

I've also tried the following optimization, but they do not improve the performance and even make increase the overall time. I think the reason is that the bottleneck is bounded by copying data back and forth from the host and the device. As long as we cannot reduce the time here, there will be a lower bound of the total execute time.

Fixed point (FP16) arithmetic
1. https://docs.nvidia.com/cuda/cuda-math-api/group__CUDA__MATH____HALF2__ARITHMETIC.html#group__CUDA__MATH____HALF2__ARITHMETIC
2. https://docs.nvidia.com/cuda/cuda-math-api/group__CUDA__MATH____HALF__MISC.html#group__CUDA__MATH____HALF__MISC

Register-tiled matrix multiplication
1. https://github.com/Huanghongru/SGEMM-Implementation-and-Optimization/blob/master/matmul_CompOpt.cu
2. https://github.com/cwpearson/nvidia-performance-tools