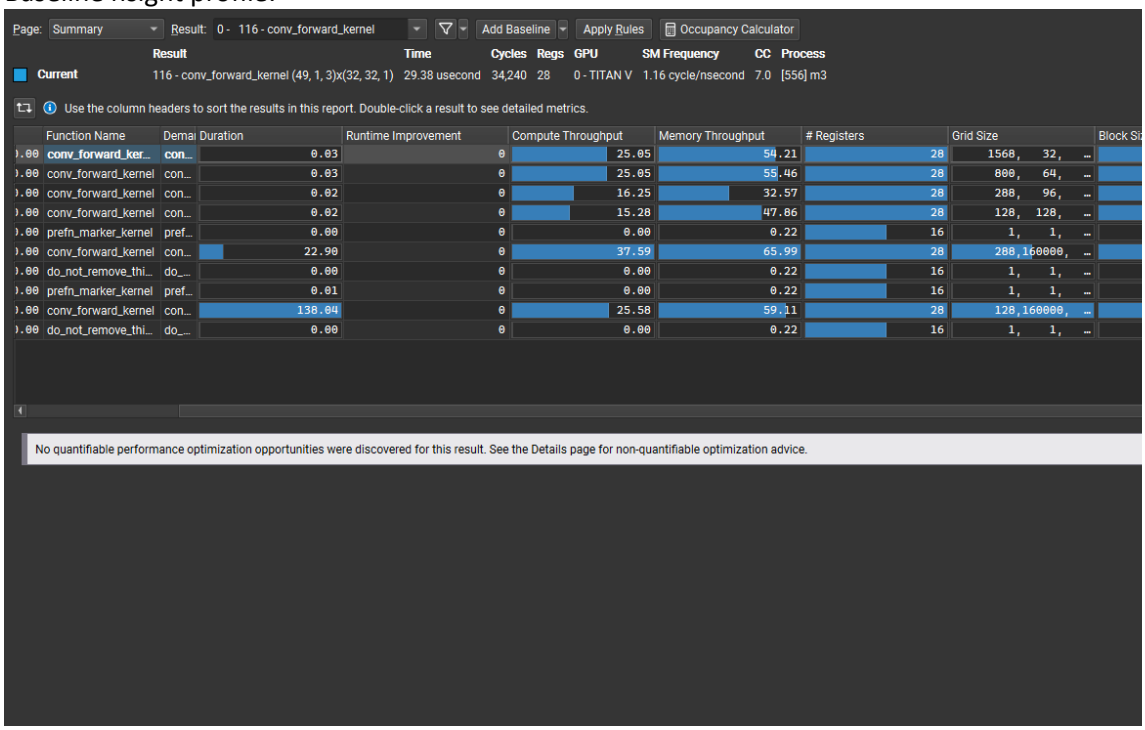
Name: *Hanliang Jiang*
NetID: *hj33*
Section: *AL2*

ECE 408/CS483 Milestone 3 Report

0. List Op Times, whole program execution time, and accuracy for batch size of 100, 1k, and 5k images from your basic forward convolution kernel in milestone 2. This will act as your baseline this milestone. Note: **Do not** use batch size of 10k when you profile in *--queue rai_amd64_exclusive*. We have limited resources, so any tasks longer than 3 minutes will be killed. Your baseline M2 implementation should comfortably finish in 3 minutes with a batch size of 5k (About 1m35 seconds, with nv-nsight).

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.475203 ms*	*2.78089 ms*	*3.256093 ms*	*0.86*
1000	*4.61965 ms*	*27.746 ms*	*32.36565 ms*	*0.886*
5000	*22.9112 ms*	*138.052 ms*	*160.9632 ms*	*0.871*

Baseline nsight profile:



1. **Optimization 1*: Tiled shared memory convolution***

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

      Tiled shared memory convolution

   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?

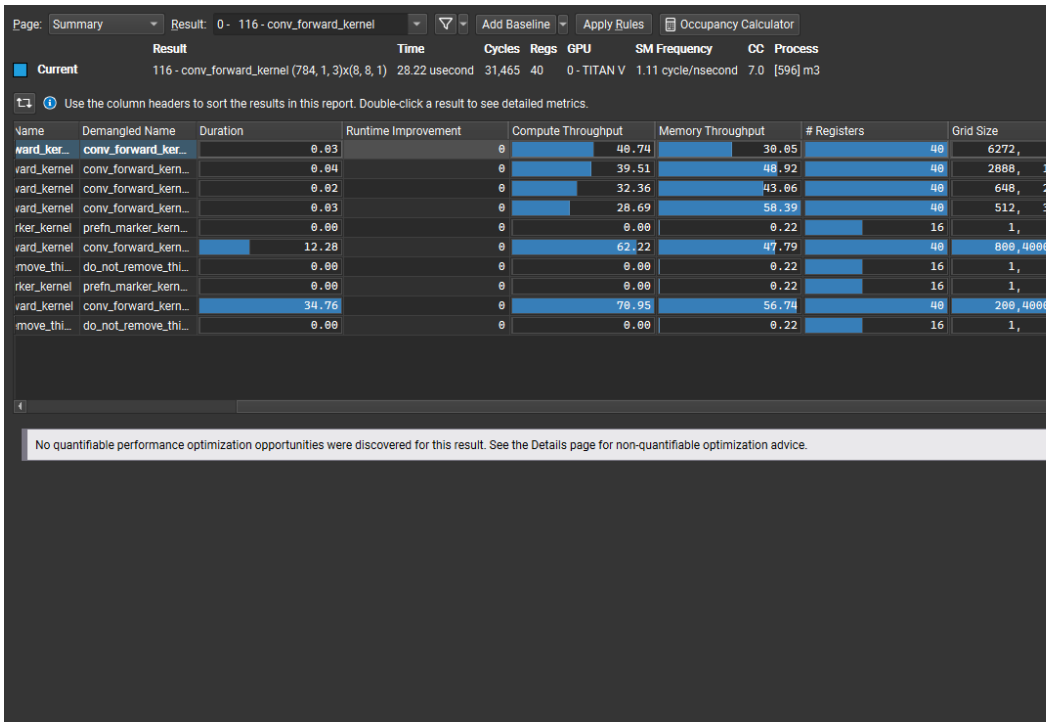
      I loaded both mask and input tile into the shared memory.
      The optimization definitely improve performance, for loading in shared memory for many times can reduce data transportation time. And tiled technique can reuse partial results.
      Yes, it is compatible with the baseline.

   c. List the Op Times, whole program execution time, and accuracy for batch size of 100, 1k, and 5k images using this optimization (including any previous optimizations also used).

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.254627 ms*	*0.68127 ms*	*0.935897 ms*	*0.86*
1000	*2.44275 ms*	*6.96609 ms*	*<exec_time>*	*0.886*
5000	*12.1753 ms*	*34.9122 ms*	*47.0875 ms*	*0.871*

   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).

      Of course.
      Transporting data with shared memory instead of global memory can increase the speed of data transfer, so data throughput rate is higher.



Take the compute throughput in the second layer for example, baseline is 25.58 while this optimation is 70.95. It increases a lot!

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

I referred my MP4 code.

2. **Optimization 2: Tuning with restrict and loop unrolling**

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

Tuning with restrict and loop unrolling

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?

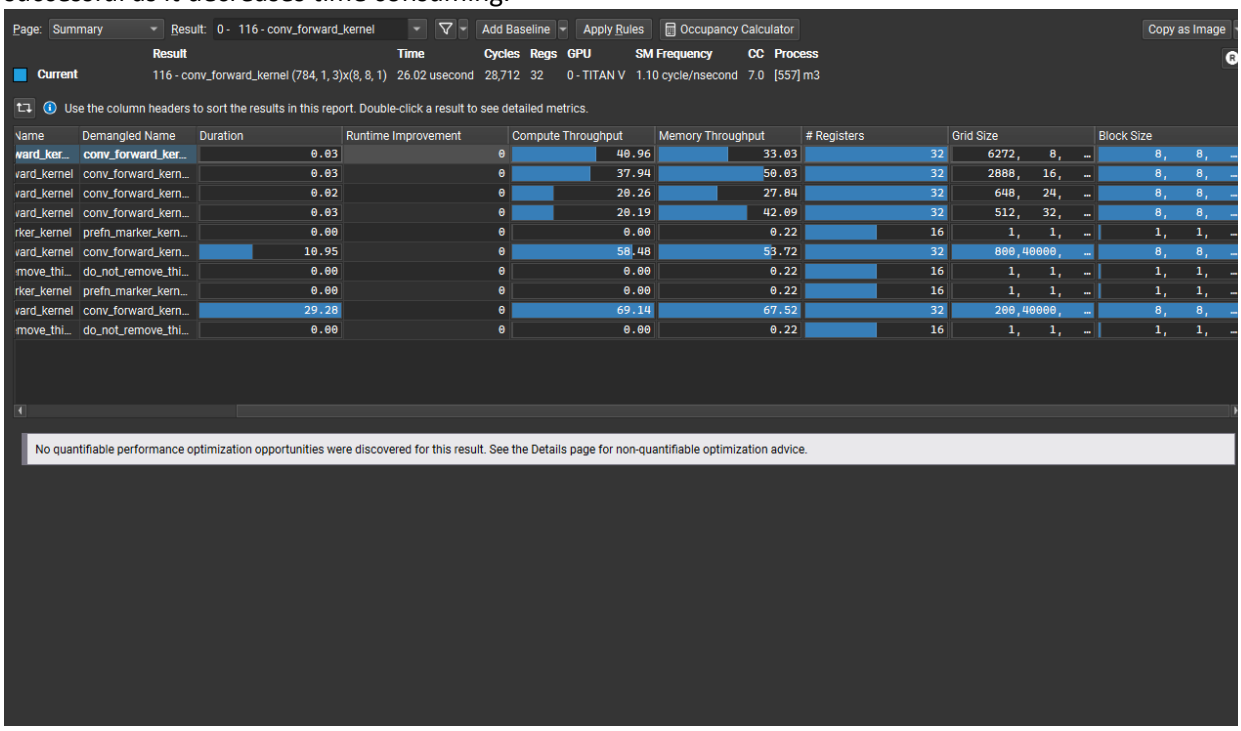
I added a restricted type on the input, mask and output of the kernel, and rewrote the loop into a verbose format.
I think this would increase the performance, as __restrict__ can help the compiler better optimize the code, so does the verbose format of a loop.
Yes, it does.

c. List the Op Times, whole program execution time, and accuracy for batch size of 100, 1k, and 5k images using this optimization (including any previous optimizations also used).

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.225541 ms*	*0.56205 ms*	*0.787591 ms*	*0.86*
1000	*2.1972 ms*	*5.89211 ms*	*8.08931 ms*	*0.886*
5000	*10.7501 ms*	*29.1418 ms*	*39.8919 ms*	*0.871*

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).

This implementation was not as successful regarding throughput, as compared to the previous optimization. The compute throughput was 70.95 for last one, but this one decreases to 69.14. The reason for that is unrolling loop requires less work, so throughput decrease. However, this is still successful as it decreases time consuming.



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

Cuda documentation: CUDA Pro Tip: Optimize for Pointer Aliasing | NVIDIA Technical Blog

3. **Optimization 3: Fixed point (FP16) arithmetic**
(Delete this section blank if you did not implement this many optimizations.)

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

Fixed point (FP16) arithmetic

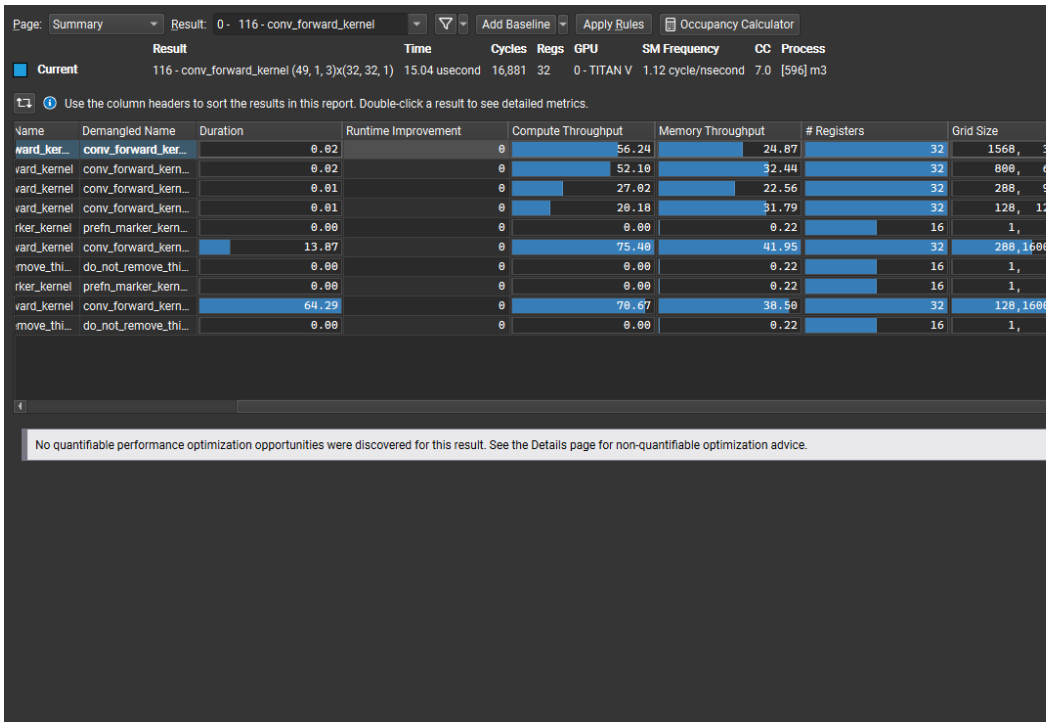
b. How does the optimization work? Did you think the optimization would increase performance of the forward convolution? Why? Does the optimization synergize with any of your previous optimizations?

I converted the float data to fp16 data(__half) when I access them, and converted the result back, and add to the output.
I think this would help with the performance, for fp16

c. List the Op Times, whole program execution time, and accuracy for batch size of 100, 1k, and 5k images using this optimization (including any previous optimizations also used).

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.295333 ms*	*1.29508 ms*	*1.59413 ms*	*0.86*
1000	*2.8096 ms*	*12.9779 ms*	*15.7875 ms*	*0.886*
5000	*13.8789 ms*	*64.7007 ms*	*78.5796 ms*	*0.871*

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).
Yes. Converting float to fp16 itself requires much local computing, so computer throughput increase a lot.
Below is a screenshot by Nsight Compute. Compared to baseline, compute throughput increases from 25.58 to70.67, which increases a lot!



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

Cuda documentation: [CUDA Math API :: CUDA Toolkit Documentation (nvidia.com)]

4. *Optimization 4: Using Streams to overlap computation with data transfer (Delete this section blank if you did not implement this many optimizations.)*

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

Using Streams to overlap computation with data transfer

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?

I used two streams to handle different batches simultaneously, if more than two batches are presented, the two streams take turns to process the following batches. It would increase the performance a lot. Processing tasks with two streams increases parallelism.
It synergizes with tiled shared memory approach.

c. List the Op Times, whole program execution time, and accuracy for batch size of 100, 1k, and 5k images using this optimization (including any previous optimizations also used).

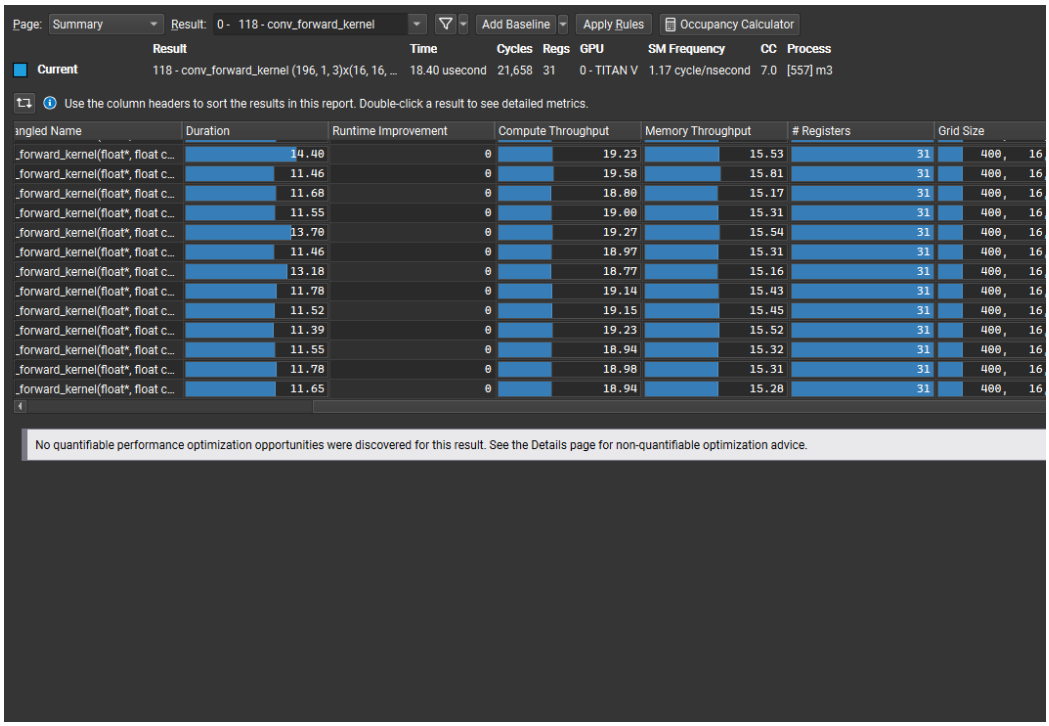
Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.002338*	*0.003888 ms*	*0.006226 ms*	*0.86*
1000	*0.00446 ms*	*0.004568 ms*	*0.009028 ms*	*0.886*
5000	*0.005905 ms*	*0.005463 ms*	*0.011368 ms*	*0.871*

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).

Yes.

Processing tasks in streams increases parallelism, which improves computation within a period of time, and it improves performance.

Comparing this approach with the baseline, as shown below, although this one may show a lower per-kernel compute throughput, this one executed the kernel for far more times, which means that overall throughput is far higher that baseline.



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

The lecture.