Name: *Wentao Zhang*
NetID: *wentao4*
Section: *AL1*

ECE 408/CS483 Milestone 3 Report

File Submitted Clarification

Optimization 1, 2, 3 are submitted, tested and profiled in ***new_forward_fp32.cu***.

Optimization 4, 6 are submitted, tested and profiled in ***new_forward_fp16.cu***.

Optimization 5, 7 are submitted, tested and profiled in ***new_forward_final.cu.***

Optimization 8 is submitted, tested and profiled in ***new_forward.cu***

For leaderboard submission, please use ***new_forward.cu***.

For op time check, please rename ***new_forward_final.cu*** *to* ***new_forward.cu*** and check it.

1. 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	*0.17596ms*	*0.63716 ms*	*5.160s*	*0.86*
1000	*1.6283ms*	*6.2782ms*	*46.817s*	*0.886*
10000	*15.9715ms*	*62.4858ms*	*8m10.069s*	*0.8714*

2. **Optimization 1:** ***Use constant Memory to save the convolution kernel***

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

    I used constant Memory to save the convolution kernel, because constant memory can be accessed in shorter time.

    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?

This optimization requires us to save kernel in the constant memory can access the constant memory during inference.

For implementation, I defined a global variable ***kernel*** *and using* ***cudaMemcpySymbol*** *to copy the host kernel to constant memory. Then I replace the global memory access with constant memory access in the kernel.*

I think this optimization can work well because constant memory can be accessed in very short time.

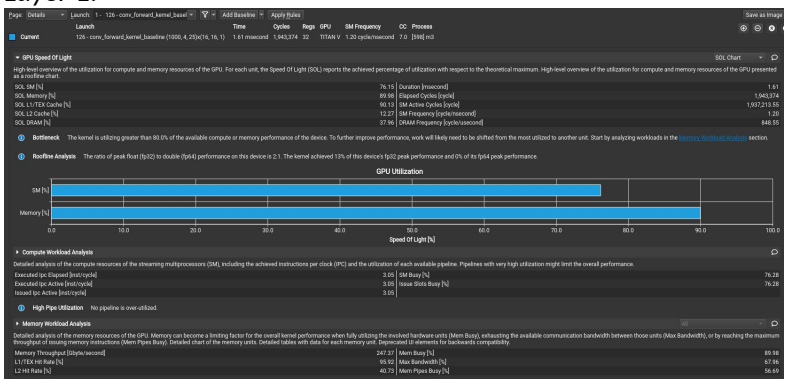
c. List the Op Times, whole program execution time, and accuracy for batch size of 100, 1k, and 10k images using this optimization (including any previous optimizations also used).

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.366427 ms*	*0.75584 ms*	*5.420s*	*0.86*
1000	*1.562 ms*	*5.724 ms*	*46.252s*	*0.886*
10000	*14.8253 ms*	*56.9035 ms*	*8m10.362s*	*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).

This implementation is successful and it improve the the op time slightly (~5%-7%) when the batch size become larger. I profile the baseline kernel and constant memory kernel. Here are my result on two kernel invoke baseline model using batch size 1000

Layer 1:



Layer 2:

After add the constant memory
Layer 1:

Layer2:

You can see that the memory utilization rate goes down and SM utilization goes up, and the performance improved.

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

CUDA C++ Programming Guide

3. **Optimization 2:** ***Tunning with restrict and loop unrolling***

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

      Then I implemented restrict and loop unrolling because with '__restrict__' keyword, the access of pointer can be optimized by compiler. And proper loop unrolling can let compiler make full use of registers.

   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 add __**restrict**__ keyword before the parameter in the kernel.*
      Using __restrict__ can let the compiler knows that two point cannot overlap, which can reduce the time of memory access.

      And using loop unroll can let the compiler use more registers which can be accessed in very short time.

      *For loop unroll implementation, I used **#pragma unroll [n]** instruction to tell the compiler to unroll the loop for n times.*

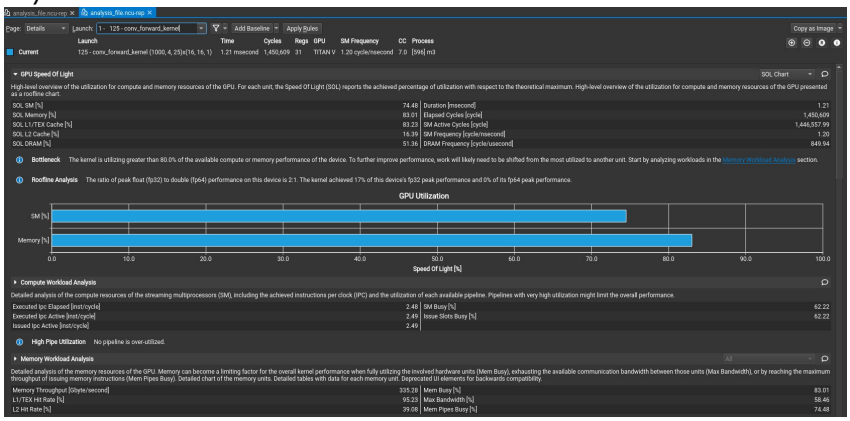
      I think it can improve the performance because the more computation can happen betweeen registers and can synergize with the previous optimization.

   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).

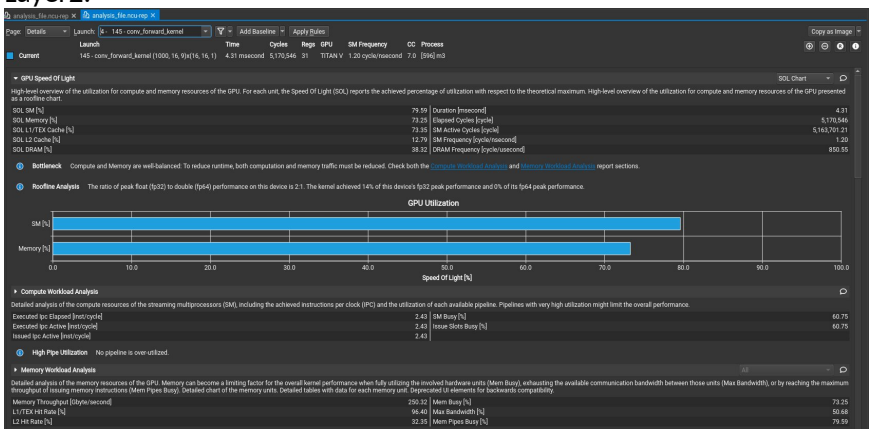
Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.18702 ms*	*0.51661 ms*	*4.882s*	*0.86*
1000	*1.29562 ms*	*4.41658 ms*	*46.920s*	*0.886*
10000	*11.1509 ms*	*43.0152 ms*	*8m3.109s*	*0.8714*

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).
It turned out that this optimization works well, it improve the op time by about 30%. I compare the result with the result of previous optimization using Nsight-Compute.

Layer1:



Layer2:



For the same kernel, we can see the duration of first layer reduced from 1.47ms to 1.21ms, and cycle reduced from 1766049 cycles to 1450609 cycles.
The second layer reduced from 5.61ms to 4.31ms, and cycle reduced from 6764573 to 5170546. This shows that the optimization do make improvement.

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

CUDA C++ Programming Guide

4. **Optimization 3: Tiled shared memory convolution**

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

I implemented tiled shared memory convolution. I think it can improve the performance because it uses shared memory which can be accessed 50-100 times faster than the global memory.

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?

This optimization requires every thread to load a tile of data from global memory to shared memory in parallel and then compute the convlution using the shared memory.

I think this optimization can improve the performance since it make full use of computing power.

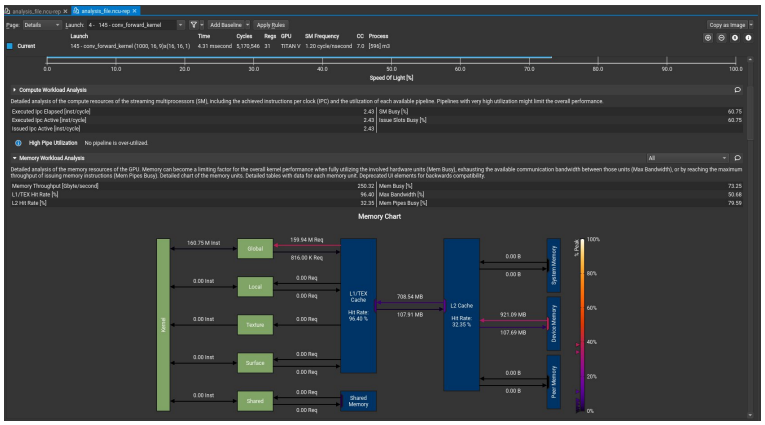
And it can synergize with previous optimization since we can constant memory access and loop unrolling is also appliable while we save some of the data to shared memory.

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).

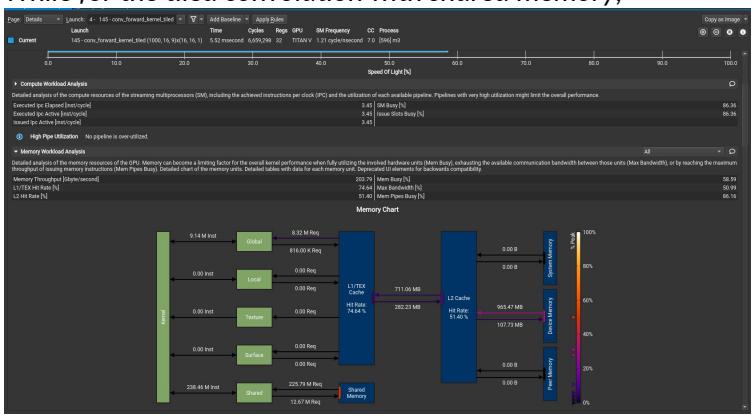
Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.19779 ms*	*0.64118 ms*	*4.830s*	*0.86*
1000	*1.33216 ms*	*5.746 ms*	*47.148s*	*0.886*
10000	*11.4665 ms*	*55.4457 ms*	*8m21.273s*	*0.8714*

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).

Compare with the baseline, the performance improve a bit, but it ***can not outpeform previous optimization****. It improved compare with baseline because it make use of shared memory, but it did not improve compare with previous optimization because the* ***lower cache hit rate****.*
I capture the memory workload information, the first picture is from ***Optimization 2***



It shows that the cache L1 hit rate is 96.4% and L2 cache hit rate is 32.35%, the memory throughput is 250.32Gb/s.
While for the tiled convolution with shared memory,



The memory throughput is significantly lower than the previous one, so the performance become bad. We decide not to include this optimization in the final veriosn.

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

CUDA C++ Programming Guide

5. **Optimization 4: Fixed point (FP16) arithmetic**

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

I choose to implement FP16 computation because it use 16 bits to save float so that the memory load time and computation time can be reduced.

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?

Normal float occupies 32 bits in the machine while FP16 computation use 16 bit in computation with special instructions.

For implementation, I define another kernel called ***to_half*** *to convert the input into float16 format in advance. Then use built-in function* ***__hadd*** *and* ***__hmul*** *to perform addition and multiplication.*

I believe it can improve the performance because it can compute two times faster and depend less on memory load.

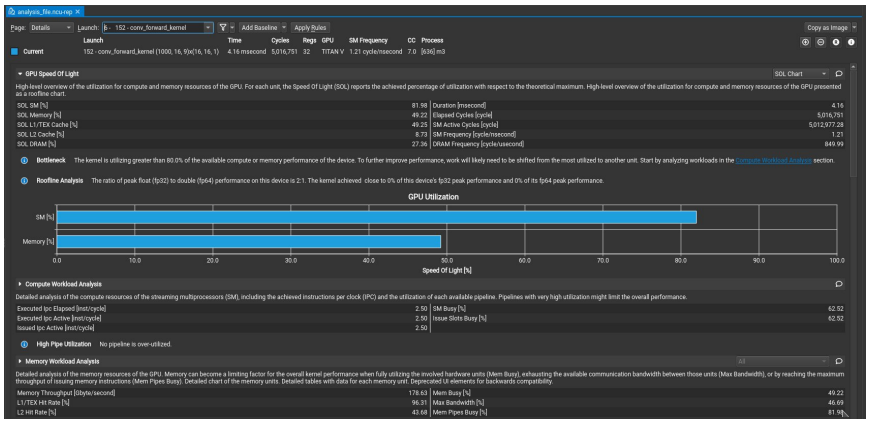
c. List the Op Times, whole program execution time, and accuracy for batch size of 100, 1k, and 10k images using this optimization (including any previous optimizations also used).

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.119351 ms*	*0.427188 ms*	*5.067s*	*0.86*
1000	*1.09954 ms*	*4.17577 ms*	*49.724s*	*0.887*
10000	*10.5644 ms*	*42.8472 ms*	*8m0.979s*	*0.8716*

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 implementation is successful, it can improve the performance the op time slightly.
Layer1:

Layer2:



*Compare with **Optimization 2**, for first layer, the duration **reduced from 1.21ms to 1.10ms**. For second layer, the duration **reduced from 4.31ms to 4.16ms.***
This is slight improvement. Theoretically, it will give a 2x speed up, but float point instructions only account for a small number of instruction among all instructions

SOL SM Breakdown	
SOL SM: Inst Executed Pipe Adu [%]	81.86
SOL IDC: Request Cycles Active [%]	74.28
SOL SM: Issue Active [%]	68.21
SOL SM: Inst Executed [%]	68.21
SOL SM: Mio2rf Writeback Active [%]	41.86
SOL SM: Inst Executed Pipe Lsu [%]	41.68
SOL SM: Mio Inst Issued [%]	41.31
SOL SM: Mio Pq Read Cycles Active [%]	38.03
SOL SM: Mio Pq Write Cycles Active [%]	37.90
SOL SM: Pipe Alu Cycles Active [%]	37.13
SOL SM: Pipe Fma Cycles Active [%]	28.42
SOL SM: Inst Executed Pipe Fp16 [%]	18.95
SOL SM: Pipe Shared Cycles Active [%]	18.95
SOL SM: Inst Executed Pipe Xu [%]	7.58
SOL SM: Inst Executed Pipe Cbu Pred On Any [%]	7.58
SOL SM: Inst Executed Pipe Ipa [%]	0
SOL SM: Inst Executed Pipe Tex [%]	0
SOL SM: Pipe Fp64 Cycles Active [%]	0
SOL SM: Pipe Tensor Cycles Active [%]	0

*According to profile result, only about 20%, so the improvement is slight. But we still decide to include this optimization to final version because it **reduce the memory load by a half** and there is still slight improvement.*

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

CUDA C++ Programming Guide

6. **Optimization 5: Multiple kernel implementations for different layer sizes**

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

I chose to implement different kernel implementations for different layer size because in previous implementation I found that different block organization layout or parameter for different layer gives different op time. So I chose to implement it to achieve better result.

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?

For different layer, based on different input parameter, such as H, W, C, M, use different block organization. I think it can improve the forward convolution because there it is reasonable that different data can be processed using different configuration. Because in this part I use most of the previous code, I believe it can synergize with the previous optimization.

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	*0.11787 ms*	*4.3122 ms*	*5.278s*	*0.86*
1000	*1.06009 ms*	*3.25258 ms*	*50.674s*	*0.887*
10000	*10.5561 ms*	*32.3514 ms*	*8m00.590s*	*0.8716*

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 implementation is successful because it improve the op time of second layer by a lot. I change the grid and block layout, and the result improved. I used **nsys** to profile the result. The result shows that changed layerout of second layer improved both in cycles and total run times. We decided to include this optimization in the final version.*

*We can demostrate the effectiveness of this optimization by comparing the **nsys** kernel statistics result on 10000 batch size.*

Baseline profile of optimization of kernel is profile of optimization 4

```
Generating CUDA Kernel Statistics...

Generating CUDA Memory Operation Statistics...
CUDA Kernel Statistics (nanoseconds)

Time(%)      Total Time   Instances         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ------------------------------
   97.5        54609793           2      27304896.5        10964064        43645729  conv_forward_kernel
    2.5         1386328           2        693164.0          643037          743291  to_half
    0.0            2816           2          1408.0            1376            1440  do_not_remove_this_kernel
    0.0            2720           2          1360.0            1344            1376  prefn_marker_kernel
```

We can see the total time of two kernel is 54609793.
After apply different implementation for different layers, we have

```
Generating CUDA Kernel Statistics...

Generating CUDA Memory Operation Statistics...
CUDA Kernel Statistics (nanoseconds)

Time(%)      Total Time   Instances         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ------------------------------
   71.4        31156656          50        623133.1          612477          629853  conv_forward_kernel_0
   25.2        10983341           4       2745835.2         2743635         2748915  conv_forward_kernel_large
    3.5         1519642          54         28141.5           14720           192447  to_half
    0.0            2656           2          1328.0            1280            1376  prefn_marker_kernel
    0.0            2656           2          1328.0            1248            1408  do_not_remove_this_kernel
```

The total conv kernel time add up to 31156656 + 10983341 = 42139997, which is a significant improvement compare with using the same implementation.

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

None

7. **Optimization 6: *Kernel fusion for unrolling and matrix-multiplication***

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

To further boost the peformance, I chose to implement fused matrix unrolling and matrix multiplication because it is frequently mentioned and theoritically improve the performance because of more parallelization and make full use of shared memory.

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?

In this optimization, we first find the output location
W_out = W - K + 1;
w_out = w % W_out;
h_out = w / W_out;
Where w_out and h_out is also the beginning location of input
*Then load a tile of matrix into the shared memory, to fully leverage the shared memory and reduce the overhead of synchronization, I choose to load whole kernel of size **M * C * K * K** into shared memory collaboratively and a big tile of input of size **K * K * C * TILE_WIDTH** to the shared memory, FP16 calculation enable us to peform this loading.*

*Assume that **cur_x** is the intermediate variable we used for traversing column **C * K * K**, we map it to **c, p, q** using*
c = cur_x / kernel_len;
tmp_idx = cur_x - c * kernel_len;
p = tmp_idx / K;
q = tmp_idx % K;
*Where **kernel_len = K * K***
*Then we can load using normal indexing like **x(b, c, h_out + p, w_out + q)** to the shared memory.*

Then we perform matrix multiplication in shared memory to get the result.

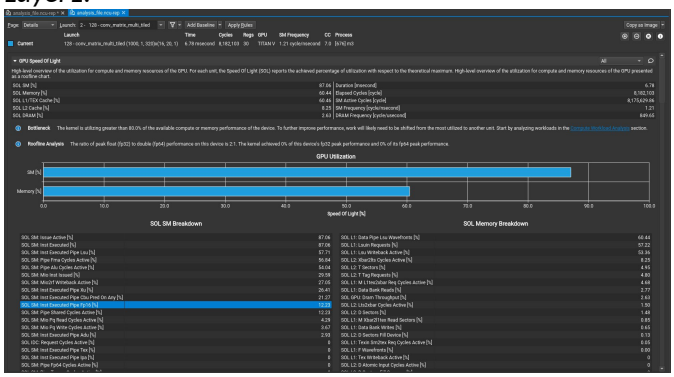
To implement this and test the performance of fused unrolled matrix multiplication, I turned off the optimization of multiple implementation for different layer size.

This optimization can not synergize with the previous optimization because it is completely different from naive convolution.

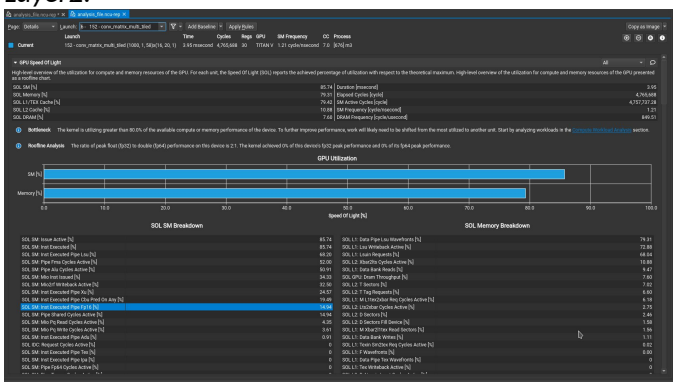
c. List the Op Times, whole program execution time, and accuracy for batch size of 100, 1k, and 10k images using this optimization (including any previous optimizations also used).

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.692826 ms*	*0.409086 ms*	*5.423s*	*0.86*
1000	*6.82276 ms*	*3.9851 ms*	*49.959s*	*0.887*
10000	*67.7911 ms*	*39.4842 ms*	*8m26.183s*	*0.8716*

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 optimization is partially successful because it improve the op time of the second layer but not the first the layer even though I set different parameter for the first layer. Here is the profile of two layer.
Layer1:



Layer2:



The profile result shows that the utilization of SM and memory system is high. And for layer2, it even outpeform the result of ***optimization 2****, but it can not outperform*

optimiaztion 5 even after carefully parameter adjusting. The implementation is correct but the result is not satisfying. So I decide not to include this optimization into the final version.

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

None

1. **Optimization 7: Sweeping various parameters to find best values**

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

I choose to tweak the parameters to improve the performance based on the different implementation of normal convolution layer which used in optimization 5. This is because I can have a chance to choose optimal parameters for 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?

For this optimization, we choose different block dimension, different grid dimension, and different unrolling factor for different layer.
I think this optimization can improve the performane because different size of input requires different level of thread coarsening.
And it can synergize with the previous optimization.

I chose TILE_WIDTH=16 for first kernel and TILE_WIDTH=32 for second kernel. The unroll factor for channel loop is 1 and 2 respectively.

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	*0.115887 ms*	*0.857705 ms*	*5.234s*	*0.86*
1000	*1.04763 ms*	*3.17042 ms*	*50.989s*	*0.886*
10000	*10.8679 ms*	*30.2002 ms*	*8m10.818s*	*0.8716*

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 optimiaztion improve the performance slightly because the parameters I have chosen is very close to the optimal one, so we can only witness very small improvement. The **nsys** result and op time improvement on 10000 batch size can prove the effectiveness of parameter tuning.*

Before tunning the parameters, we have (as we mentioned above)

```
Generating CUDA Kernel Statistics...

Generating CUDA Memory Operation Statistics...
CUDA Kernel Statistics (nanoseconds)

Time(%)      Total Time   Instances         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ------------------------------
   71.4        31156656          50        623133.1          612477          629853  conv_forward_kernel_0
   25.2        10983341           4       2745835.2         2743635         2748915  conv_forward_kernel_large
    3.5         1519642          54         28141.5           14720          192447  to_half
    0.0            2656           2          1328.0            1280            1376  prefn_marker_kernel
    0.0            2656           2          1328.0            1248            1408  do_not_remove_this_kernel
```

After we tuning the parameters, we got

```
Generating CUDA Kernel Statistics...

Generating CUDA Memory Operation Statistics...
CUDA Kernel Statistics (nanoseconds)

Time(%)      Total Time   Instances         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  --------------------------------
   69.7        28530999          50        570620.0          562151          574566  conv_forward_kernel_0
   26.6        10869685           4       2717421.2         2714150         2720229  conv_forward_kernel_large
    3.7         1506624          54         27900.4           14528          192535  to_half
    0.0            2624           2          1312.0            1248            1376  do_not_remove_this_kernel
    0.0            2528           2          1264.0            1184            1344  prefn_marker_kernel
```

The overall time add up to 28530999 + 10869685 = 39400684, which is a significant improvement of 42139997. This prove the effectiveness of my optimization.

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

None

1. **Optimization 8: Using Streams to overlap computation with data transfer**

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

I chose to implement streaming for overall time optimization because I profile the whole running time and it shows that memory copy account for large amount of time. So I need to reduce the memory copy time so that I can improve the overall running time.

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?

*First the input and output memory are converted to **pinned memory** using **cudaHostRegister** so that the host virtual memory pages are specially marked and they cannot be paged out. This allows a overlap between **device to host** and **host to device** memory copy, which save a lot of time.*

*I used **cudaMemcpyAsync** instead of **cudaMemcpy** so that we can overlap memory copy and execution.*

*I also change the structure of the template file and merge memory allocation and memory copy to one function call so that we can avoid the **cudaDeviceSynchronize** to cut off the stream.*

I think it can synergize with previous optimization. But it can not improve the op time because this optimization improve the layer time and overall time.

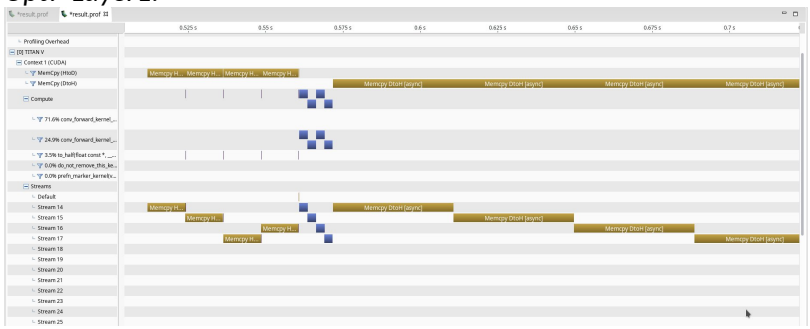
c. List the Op Times, whole program execution time, and accuracy for batch size of 100, 1k, and 10k images using this optimization (including any previous optimizations also used).

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*1.86249 ms*	*1.49721 ms*	*5.357s*	*0.86*
1000	*17.8265 ms*	*12.4047 ms*	*50.521s*	*0.887*
10000	*177.478 ms*	*122.129 ms*	*8m17.680s*	*0.8716*

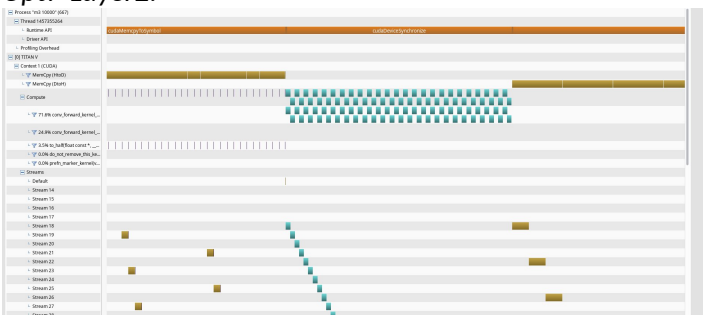
d. Was implementing this optimization successful in improving performance? Why or why not? Include profiling results from *nsys* and *Nsight-Compute* to justify your answer, directly comparing to your baseline (or the previous optimization this one is built off of).

*This implementation is successful although the op time and overall time did not improve, but if we use nvvp to plot the running process and find that the **Host2Device** transfer, **Computation**, **Device2Host** transfer are overlaped. And I got a good rank on the leader board. Here are the comparison between the profile result of **optimization 7 and optimization 8***

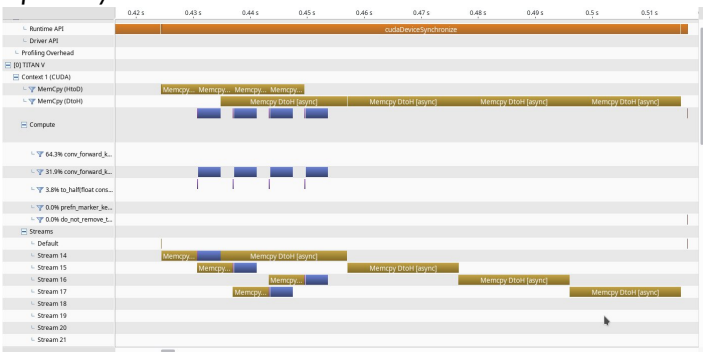
Opt7-Layer1:



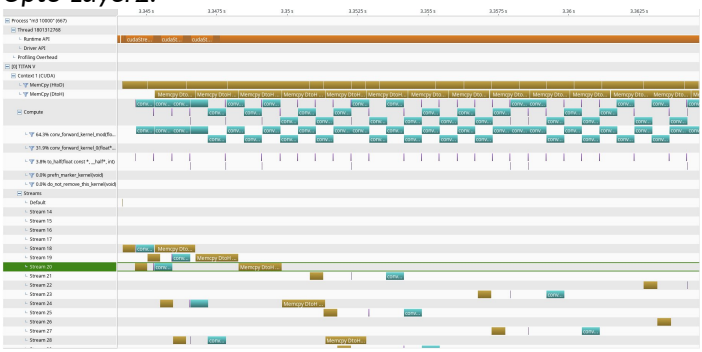
Opt7-Layer2:



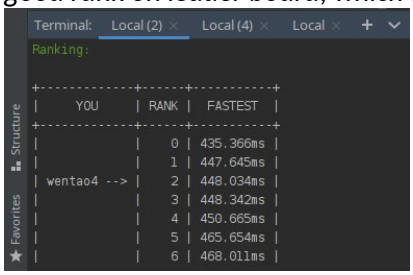
Opt8-Layer1:



Opt8-Layer2:



*We can observe that **the computation, and bi-directional memory copy are overlapped**. This lead to significant improvement of performance. As a result, I got a good rank on leader board, which also prove the effectiveness of this optimization.*



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

https://developer.download.nvidia.com/CUDA/training/StreamsAndConcurrencyWebinar.pdf