**Name:** *Lizhuang Zheng*
**NetID:** *Lzheng17*
**Section:** *ZJ1*

# 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 | *0.235181ms* | *0.858519ms* | *0m1.580s* | *0.86* |
| 1000 | *2.19411ms* | *8.33744ms* | *0m10.200s* | *0.886* |
| 5000 | *10.8326ms* | *41.6817ms* | *0m48.498s* | *0.871* |

*Baseline Nsys:*

```
Time(%)      Total Time       Calls         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  --------------------------------
   74.9      1077668953          20      53883447.6           28622       583849151  cudaMemcpy
   16.1       231715604          20      11585780.2            2416       225532838  cudaMalloc
    7.3       104651021          16       6540688.8             811        82819748  cudaDeviceSynchronize
    1.1        16447834          10       1644783.4           15546        16239841  cudaLaunchKernel
    0.6         9074469          20        453723.5            2328         6068658  cudaFree



Generating CUDA Kernel Statistics...

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

Time(%)      Total Time   Instances         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  --------------------------------
  100.0       104613970           6      17435661.7            9504        82818262  conv_forward_kernel
    0.0            2880           2          1440.0            1376            1504  do_not_remove_this_kernel
    0.0            2656           2          1328.0            1312            1344  prefn_marker_kernel


CUDA Memory Operation Statistics (nanoseconds)

Time(%)      Total Time  Operations         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  --------------------------------
   92.2       977031076           6     162838512.7           12640       582951968  [CUDA memcpy DtoH]
    7.8        82341716          14       5881551.1            1120        42712355  [CUDA memcpy HtoD]


CUDA Memory Operation Statistics (KiB)

            Total      Operations            Average            Minimum            Maximum  Name
-----------------  --------------  -----------------  -----------------  -----------------  --------------------------------
        1723922.0               6           287320.4            148.535          1000000.0  [CUDA memcpy DtoH]
         545660.0              14            38975.7              0.004           288906.0  [CUDA memcpy HtoD]
```

1. **Optimization 1: Tiled shared memory convolution (2 points)**

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

    *I chose the Tiled shared memory convolution, because I think using shared memory will decrease the access time for input feature maps and masks, since there is some reuse of values in each block.*

    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 each block we use two shared memories, one for loading sub-tiles of input feature maps and another for loading sub-tiles of masks.*

    *I think the optimization will improve the performance since shared memory access is much faster than the global memory access.*

    *This optimization is implemented directly based on the M2 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 | *0.823724ms* | *2.13124ms* | *0m1.575s* | *0.86* |
| 1000 | *8.19428ms* | *21.3309ms* | *0m10.278s* | *0.886* |
| 5000 | *40.6632ms* | *106.344ms* | *0m48.653ms* | *0.871* |

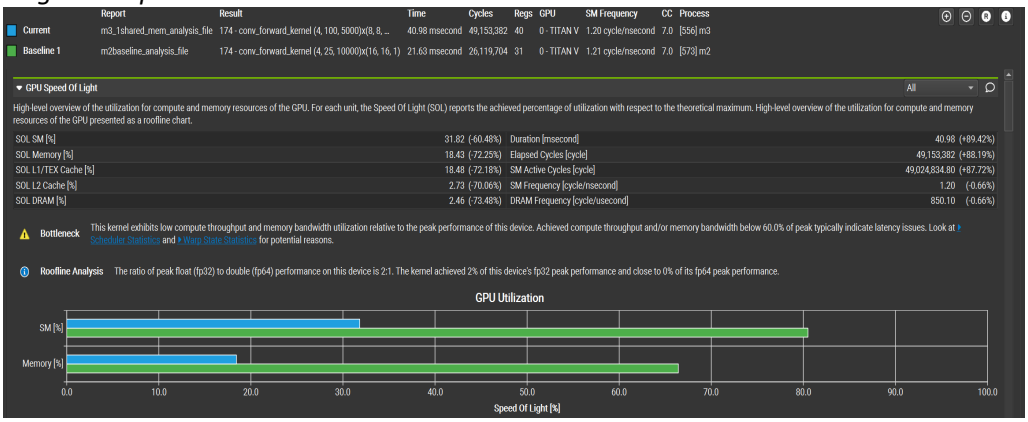
    d. Was implementing this optimization successful in improving performance? Why or why not? Include profiling results from *nsys* and *Nsight-Compute* to justify your answer, directly comparing to your baseline (or the previous optimization this one is built off of).

*It is not successful in improving performance, because we need to take account of the existence of Stride "S." Every time we load a sub-tile of input feature map, we need to cover [(TILE_WIDTH-1)*S+MASK_WIDTH]^2 elements (here MASK_WIDTH=K), but as the stride being large, the overlap between needed places (TILE_WIDTH^2 values, each place is S positions away from others) and halos (K^2 areas around each needed place, required to be convolved with the mask sub-tile) will become less, even no overlap if the stride is large enough, making it almost no reuse among a block. What's more, in this situation, we even load more unnecessary values into the shared memory, which is a big waste of resources.*

*Nsys:*

```
Generating CUDA API Statistics...
CUDA API Statistics (nanoseconds)

Time(%)      Total Time       Calls         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ---------------------------------
   61.0       552532423          20      27626621.1           31344       286718982  cudaMemcpy
   21.0       190085554          20       9504277.7            2321       186644123  cudaMalloc
   16.4       148349722          16       9271857.6             907       106798867  cudaDeviceSynchronize
    1.4        12434650          10       1243465.0           25433        12157185  cudaLaunchKernel
    0.3         2789750          20        139487.5            2364          628189  cudaFree



Generating CUDA Kernel Statistics...

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

Time(%)      Total Time   Instances         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ---------------------------------
  100.0       148316211           6      24719368.5           33824       106788675  conv_forward_kernel
    0.0            2752           2          1376.0            1376            1376  do_not_remove_this_kernel
    0.0            2624           2          1312.0            1280            1344  prefn_marker_kernel


CUDA Memory Operation Statistics (nanoseconds)

Time(%)      Total Time  Operations         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ---------------------------------
   92.8       505902184           6      84317030.7           12799       285956046  [CUDA memcpy DtoH]
    7.2        38989824          14       2784987.4            1152        20959287  [CUDA memcpy HtoD]


CUDA Memory Operation Statistics (KiB)

            Total      Operations            Average            Minimum            Maximum  Name
----------------  --------------  -----------------  -----------------  -----------------  ---------------------------------
        862672.0               6           143778.7            148.535           500000.0  [CUDA memcpy DtoH]
        276206.0              14            19729.0              0.004           144453.0  [CUDA memcpy HtoD]
```

*Nsight-Compute:*



*From nsys we can see that CUDA kernel time increases compared with the M2 baseline, due to the data copy into shared memory. But from Nsight-Compute, we can see that GPU utilization is actually decreased.*

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

*Previous Lab: Lab 3 Tiled Matrix Multiplication.*

2. **Optimization 2: Weight matrix (kernel values) in constant memory (0.5 point)**

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

*I chose to implement Weight matric (kernel values) in constant memory, because constant memory access is faster than the global memory access, and the weight matrix are highly reused by each block, which is very suitable to use constant 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?

*Instead of copying the host_mask into the device global memory, we use cudaMemcpyToSymbol to copy host_mask into the constant memory Mask_c, and for each block, we take values from the constant memory directly.*

*I think this optimization will increase performance of the forward convolution, because accessing constant memory is faster than accessing global memory, and for the masks, we need to access them multiple times during computation, it should have better performance than Optimization 1 where we only use shared memory.*

*This optimization is based on Optimization 1.*

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 | *0.767434ms* | *2.20522ms* | *0m1.560s* | *0.86* |
| 1000 | *7.60375ms* | *22.1306ms* | *0m10.772s* | *0.886* |
| 5000 | *37.9277ms* | *110.012ms* | *0m49.237s* | *0.871* |

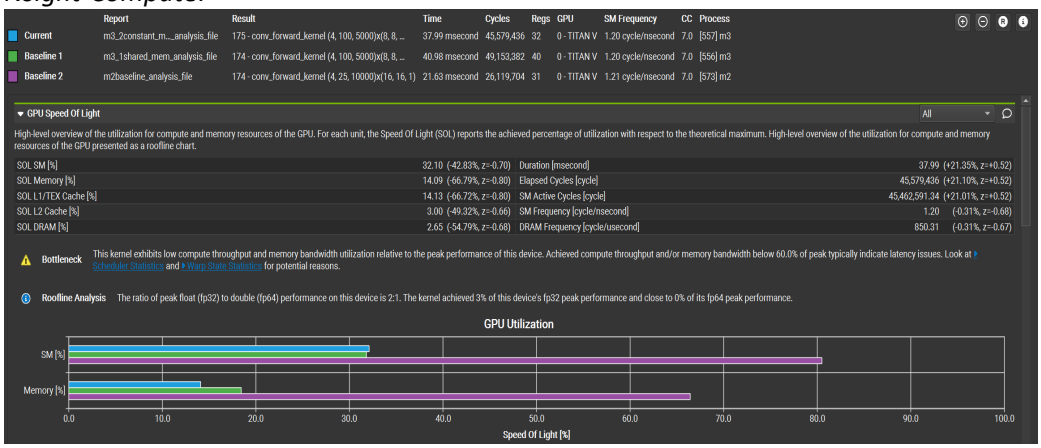
d. Was implementing this optimization successful in improving performance? Why or why not? Include profiling results from *nsys* and *Nsight-Compute* to justify your answer, directly comparing to your baseline (or the previous optimization this one is built off of).

*It is not successful in improving performance, there is almost no significant change. Maybe it is because the time it needs to access global memory is similar to the time it needs to copy into shared memory and access shared memory.*

*Nsys:*

```
Generating CUDA API Statistics...
CUDA API Statistics (nanoseconds)

Time(%)      Total Time       Calls         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ------------------------------
   59.1       541493166          14      38678083.3           32377       291538778  cudaMemcpy
   22.5       205960866          20      10298043.3            2523       202661899  cudaMalloc
   16.4       150728575          16       9420535.9             850       111828365  cudaDeviceSynchronize
    1.6        14442543          10       1444254.3           25122        14177684  cudaLaunchKernel
    0.3         2756121          20        137806.0            2204          586448  cudaFree
    0.1         1003377           6        167229.5          105495          188800  cudaMemcpyToSymbol



Generating CUDA Kernel Statistics...

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

Time(%)      Total Time   Instances         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ------------------------------
  100.0       150688996           6      25114832.7           39104       111822563  conv_forward_kernel
    0.0            2816           2          1408.0            1344            1472  do_not_remove_this_kernel
    0.0            2560           2          1280.0            1280            1280  prefn_marker_kernel


CUDA Memory Operation Statistics (nanoseconds)

Time(%)      Total Time  Operations         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ------------------------------
   91.5       492428410           6      82071401.7           23167       290771071  [CUDA memcpy DtoH]
    8.5        45915108          14       3279650.6            1152        24014201  [CUDA memcpy HtoD]


CUDA Memory Operation Statistics (KiB)

            Total      Operations            Average            Minimum            Maximum  Name
----------------  --------------  -----------------  -----------------  -----------------  ------------------------
        862672.0               6           143778.7            148.535           500000.0  [CUDA memcpy DtoH]
        276206.0              14            19729.0              0.004           144453.0  [CUDA memcpy HtoD]
```

*Nsight-Compute:*




*From nsys, we can see that cudaMemcpy time + cudaMemcpyToSymbol time is almost the same as cudaMemcpy time in Optimization 1. And the kernel time is even higher. In Nsight-Compute, the GPU memory utilization is even lower than Optimization 1.*

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

*Previous Lab: Lab 4: Convolution.*
*Lecture slides.*

3. **Optimization 3: *Shared memory matrix multiplication and input matrix unrolling (3 points)***

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

      *I chose to implement Shared memory matrix multiplication and input matrix unrolling, because originally every input tile is loaded M times (M is the number of output feature maps), which is not efficient in global memory bandwidth, so using unrolled matrix multiplication will allow duplicated input features to be shared among output feature maps, no need to load input feature tiles multiple times.*

   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 each image we unroll the input feature maps into a (C*K*K)-row and (H_out*W_out)-column matrix, and use the unrolled convolution masks to multiply it, so that we can get the output feature matrix with dimensions (M * (H_out*W_out)). Note that as we use the linearized index, the (M * (C*K*K)) mask matrix can be seen as already unrolled; and for the output matrix, the situation is similar.*
      *I first implement the unroll using CPU host code, and then implement the unroll using an extra GPU kernel.*

      *I think this optimization will increase the performance because doing shared memory matrix multiplication may be faster than the original ones. However, CPU unrolling may cost extra CPU host resources and longer CPU times, and using extra GPU kernel will definitely increase GPU Op-time to some extent.*

      *This optimization is based on the baseline, and the GPU version is synergized with the CPU version.*

   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).

      CPU unrolling:

| Batch Size | Op Time 1 | Op Time 2 | Total Execution Time | Accuracy |
|---|---|---|---|---|
| 100 | *0.426838ms* | *1.54382ms* | *0m2.093s* | *0.86* |
| 1000 | *3.69845ms* | *13.7846ms* | *0m14.621s* | *0.886* |
| 5000 | *18.2071ms* | *68.8156ms* | *1m9.107s* | *0.871* |

GPU unrolling:

| Batch Size | Op Time 1 | Op Time 2 | Total Execution Time | Accuracy |
|---|---|---|---|---|
| 100 | *1.34526ms* | *2.04341ms* | *0m1.546s* | *0.86* |
| 1000 | *8.49249ms* | *18.502ms* | *0m10.504s* | *0.886* |
| 5000 | *41.4166ms* | *92.209ms* | *0m50.173s* | *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).

*Not successful in improving performance, I think it may because unrolling uses extra resources (whatever CPU or GPU resources), and there are many duplicated elements in the unrolled input feature map matrix.*

*From nsys below, for CPU unrolling, we can see that the conv_forward_kernel time is cut down a little bit compared with the baseline. But the cudaMemcpy and cudaMalloc times increase a lot.*
*For GPU unrolling, we can see that we have got an extra unroll_Kernel which takes more kernel time. But the cudaMemcpy time is significantly cut down compared with both the CPU version and the baseline. The cudaMalloc time is also shorter than the CPU version, but still longer than the baseline, since we are allocating more spaces for unrolling.*

*From Nsight-Compute, the conv_forward_kernel takes up more SM (this kernel is the same for both versions). And for the GPU version, the unroll_Kernel takes up more memory usage.*

*Nsys:*

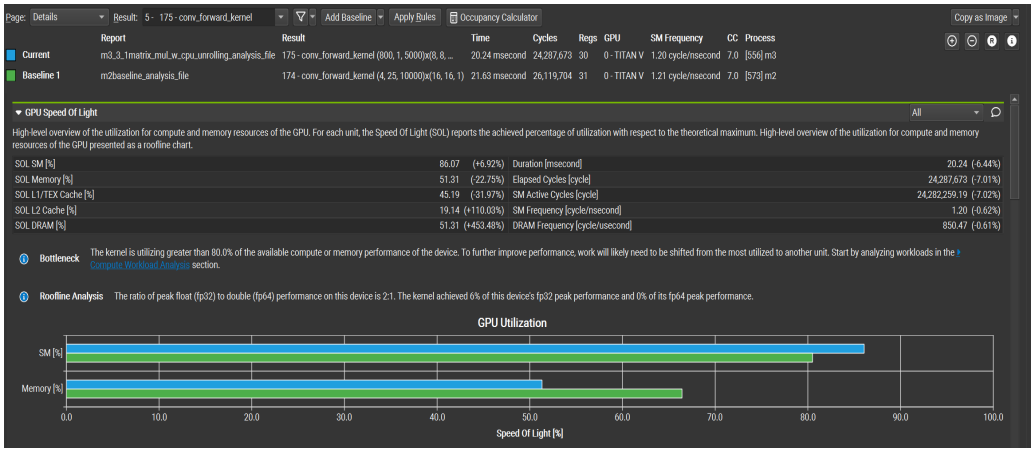
*CPU unrolling:*

```
Generating CUDA API Statistics...
CUDA API Statistics (nanoseconds)

Time(%)      Total Time       Calls         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ----------------------------
   64.2      1990963728          14     142211694.9           31129       835390496  cudaMemcpy
   18.3       569026422          20      28451321.1            3357       169984656  cudaMalloc
   10.3       319107056          20      15955352.8            4657       123121052  cudaFree
    7.2       222577139          16      13911071.2             894       120469147  cudaDeviceSynchronize
    0.0          971951          10         97195.1           25830          338507  cudaLaunchKernel
    0.0          419339           6         69889.8           47307           89177  cudaMemcpyToSymbol



Generating CUDA Kernel Statistics...

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

Time(%)      Total Time   Instances         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ----------------------------
  100.0        99566211           6      16594368.5            9184        78940002  conv_forward_kernel
    0.0            4736           2         2368.0            1408            3328  do_not_remove_this_kernel
    0.0            2656           2         1328.0            1312            1344  prefn_marker_kernel


CUDA Memory Operation Statistics (nanoseconds)

Time(%)      Total Time  Operations         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ----------------------------
   74.6      1468311952          14     104879425.1            1184       835249452  [CUDA memcpy HtoD]
   25.4       500656922           6      83442820.3           12608       288574421  [CUDA memcpy DtoH]


CUDA Memory Operation Statistics (KiB)

             Total      Operations            Average            Minimum            Maximum  Name
----------------  --------------  -----------------  -----------------  -----------------  ----------------------
       10563126.0              14           754509.0              0.004          6125000.0  [CUDA memcpy HtoD]
         862672.0               6           143778.7            148.535           500000.0  [CUDA memcpy DtoH]
```

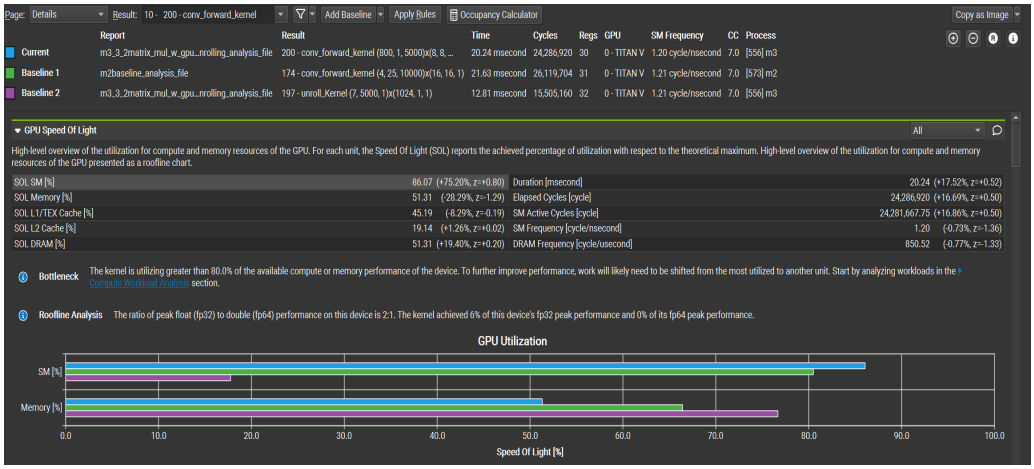
*GPU unrolling:*

```
Generating CUDA API Statistics...
CUDA API Statistics (nanoseconds)

Time(%)      Total Time       Calls         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ----------------------------
   46.3       578636940          14      41331210.0           29323       293253302  cudaMemcpy
   32.0       400544005          26      15405538.7            2979       295280684  cudaMalloc
    9.6       120135538          22       5460706.3             807        77590029  cudaDeviceSynchronize
    8.7       109351246          16       6834452.9            5628       109010825  cudaLaunchKernel
    3.3        40729548          26       1566521.1            3013        23744240  cudaFree
    0.1         1516394           6        252732.3          106530          709787  cudaMemcpyToSymbol



Generating CUDA Kernel Statistics...

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

Time(%)      Total Time   Instances         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ----------------------------
   81.9        98324688           6      16387448.0            7456        77580952  conv_forward_kernel
   18.1        21728945           6       3621490.8            6079        12207170  unroll_Kernel
    0.0            2880           2          1440.0            1440            1440  do_not_remove_this_kernel
    0.0            2496           2          1248.0            1248            1248  prefn_marker_kernel


CUDA Memory Operation Statistics (nanoseconds)

Time(%)      Total Time  Operations         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ----------------------------
   91.9       523332208           6      87222034.7           23488       292438463  [CUDA memcpy DtoH]
    8.1        46347320          14       3310522.9            1152        24007847  [CUDA memcpy HtoD]


CUDA Memory Operation Statistics (KiB)

            Total      Operations            Average            Minimum            Maximum  Name
----------------  --------------  -----------------  -----------------  -----------------  ----------------------
         862672.0               6           143778.7            148.535           500000.0  [CUDA memcpy DtoH]
         276206.0              14            19729.0              0.004           144453.0  [CUDA memcpy HtoD]
```

*Nsight-Compute:*
*CPU unroll:*




*GPU unroll:*




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

*Textbook: Programming Massively Parallel Processors, 4th Edition, Chapter 16 Deep Learning.*
*Previous Lab: Lab 3 Tiled Matrix Multiplication.*

4. **Optimization 4: Kernel fusion for unrolling and matrix-multiplication (2 points)**

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

*I chose to implement Kernel fusion for unrolling and matrix-multiplication, because I think compared with launching two kernels for unrolling, launching only one kernel should be faster, since kernel launching really takes 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?

*For kernel fusion, we deleted the unroll_Kernel used in Optimization 3 GPU version. And actually now we are not unrolling all elements at once. Instead, every time we want to load a sub-tile of input feature maps for matrix multiplication, we do a conceptual unrolling: using index mapping to fetch correct data from the original input feature maps and arrange them into the order of an unrolled matrix. Then we use this sub-tile of the conceptual unrolled matrix to do the matrix multiplication just as in Optimization 3.*

*I think this optimization will increase performance compared with Optimization 3 GPU version, because now we do not need to allocate new device memory for unrolled matrix, and now we have only one kernel rather than two, which should save some kernel launching time.*

*The optimization synergize with my Optimization 3 (GPU version).*

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 | *0.537133ms* | *0.73279ms* | *0m1.596s* | *0.86* |
| 1000 | *5.28054ms* | *7.30228ms* | *0m10.288s* | *0.886* |
| 5000 | *26.1775ms* | *36.4892ms* | *0m48.818s* | *0.871* |

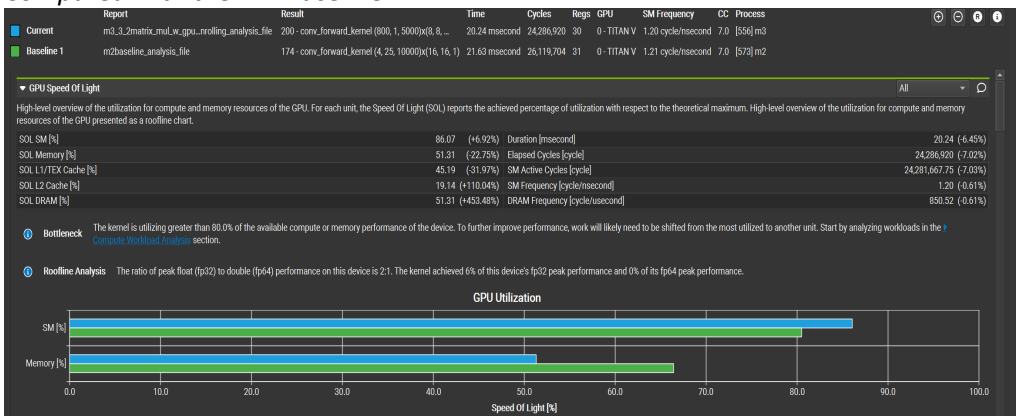
d. Was implementing this optimization successful in improving performance? Why or why not? Include profiling results from *nsys* and *Nsight-Compute* to justify your answer, directly comparing to your baseline (or the previous optimization this one is built off of).

*Compared with Optimization 3 GPU version, it is successful to improve the performance, but if compared with the baseline, it is still not good enough.*
*I think although we use only one kernel to reduce kernel launching time, and do not need to allocate extra memory space for unrolled matrix, we need to do more works in a kernel, such as computing indices, which takes up more GPU resources.*

*Nsys:*

```
Generating CUDA API Statistics...
CUDA API Statistics (nanoseconds)

Time(%)      Total Time       Calls         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ----------------------------------
   38.8       559887497          14      39991964.1           27635       294971203  cudaMemcpy
   34.0       491816305          20      24590815.3            2591       178113342  cudaMalloc
   22.7       328400186          20      16420009.3            3486        75192151  cudaFree
    4.4        63330287          16       3958142.9             892        36515360  cudaDeviceSynchronize
    0.1          871745           6        145290.8           58378          195272  cudaMemcpyToSymbol
    0.0          407392          10         40739.2           22539          116912  cudaLaunchKernel



Generating CUDA Kernel Statistics...

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

Time(%)      Total Time   Instances         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ----------------------------------
  100.0        63287992           6      10547998.7            9632        36512756  conv_forward_kernel
    0.0            2784           2          1392.0            1344            1440  do_not_remove_this_kernel
    0.0            2592           2          1296.0            1280            1312  prefn_marker_kernel


CUDA Memory Operation Statistics (nanoseconds)

Time(%)      Total Time  Operations         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ----------------------------------
   91.2       493235307           6      82205884.5           23200       294232944  [CUDA memcpy DtoH]
    8.8        47769145          14       3412081.8            1152        25789261  [CUDA memcpy HtoD]


CUDA Memory Operation Statistics (KiB)

            Total      Operations            Average            Minimum            Maximum  Name
----------------  --------------  -----------------  -----------------  -----------------  ----------------------------
        862672.0               6           143778.7            148.535           500000.0  [CUDA memcpy DtoH]
        276206.0              14            19729.0              0.004           144453.0  [CUDA memcpy HtoD]
```

*Nsight-Compute:*
*Compared with the M2 Baseline:*



*Compared with the Optimization 3 GPU version:*


*From nsys, we can see that cudaMalloc time is decreased compared with Optimization 3 (GPU version). And the conv_forward_kernel time is much shorter than Optimization 3 (GPU version). Also we do not have unroll_Kernel, which saves more kernel time. But compared with the M2 baseline, although we can see a large cut down in cudaMemcpy time, the cudaMalloc is still larger, due to the allocation of shared memory. By the way, the conv_forward_kernel time is cut down to the half.*

*From Nsight-Compute, we have more SM utilization and less memory utilization than the baseline, but we have much less memory utilization compared with the two kernels of the Optimization 3 (GPU version).*

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

*Textbook: Programming Massively Parallel Processors, 4th Edition, Chapter 16 Deep Learning.*
*Previous Lab: Lab 3 Tiled Matrix Multiplication.*
*Lecture Slides: Lecture 12 Computation in Deep Neural Networks*

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

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

*I chose to implement Fixed point (FP16) arithmetic, because FP16 takes only 16 bits (2 bytes) instead of the FP32 float (32 bits, 4 bytes), so transferring FP16 and Memcpy will take less 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?

*We need first include <mma.h> and use namespace nvcuda, then I use an extra kernel to convert FP32 to FP16 by using "__float2half()" function. And then we use "half" type variables to do arithmetic inside the conv_forward_kernel. After that we launch another kernel to convert FP16 back to FP32 using "__half2float()" function, and get the output.*

*I think this optimization will increase performance, since we cut the amount of Memcpy to the half.*

*I developed two versions, one version synergizes with Optimization 4, and another version synergizes with Optimization 4 (GPU version).*

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).
Based on Optimization 4 (Kernel Fusion):

| Batch Size | Op Time 1 | Op Time 2 | Total Execution Time | Accuracy |
|---|---|---|---|---|
| 100 | *0.857424ms* | *0.947429ms* | *0m1.686s* | *0.86* |
| 1000 | *6.20048ms* | *7.20274ms* | *0m10.866s* | *0.887* |
| 5000 | *29.0414ms* | *34.3087ms* | *0m49.551s* | *0.8712* |

Based on Optimization 3 (GPU Version: 2-Kernel GPU Unrolling):

| Batch Size | Op Time 1 | Op Time 2 | Total Execution Time | Accuracy |
|---|---|---|---|---|
| 100 | *1.02267ms* | *1.27717ms* | *0m1.580s* | *0.86* |
| 1000 | *6.35008ms* | *9.1874ms* | *0m10.781s* | *0.887* |
| 5000 | *30.8617ms* | *44.9375ms* | *1m4.854s* | *0.8712* |

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).

*For the version based on Optimization 4, there is no significant change in performance, I think that is because we launched two more kernels, which takes up the time we saved in data transfer.*
*For the version based on Optimization 3 (GPU version), there is an improvement in performance compared with Optimization 3 (GPU version). I think that is because Memcpy takes more proportion of total time, and using FP16 will cut down this time significantly, even more than the time increase due to the two kernel launches.*

*Nsys: The version based on Optimization 4 (Kernel Fusion):*

```
Generating CUDA API Statistics...
CUDA API Statistics (nanoseconds)

Time(%)      Total Time       Calls         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  --------------------------------
   66.3       540910081          20      27045504.1           35215       284711877  cudaMemcpy
   23.1       188765375          38       4967509.9            1958       184254643  cudaMalloc
    7.7        62513592          28       2232628.3             777        32351672  cudaDeviceSynchronize
    2.5        20022346          28        715083.8            4542        19645238  cudaLaunchKernel
    0.5         4024324          38        105903.3            2072          833880  cudaFree
    0.0           76432           6         12738.7           11913           14060  cudaMemcpyToSymbol



Generating CUDA Kernel Statistics...

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

Time(%)      Total Time   Instances         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  --------------------------------
   95.1        59418669           6       9903111.5            9408        32345808  conv_forward_kernel
    3.7         2320557           6        386759.5            1568         1341845  convertFP16toFP32
    1.2          721050          12         60087.5            1248          371037  convertFP32toFP16
    0.0            2880           2          1440.0            1344            1536  do_not_remove_this_kernel
    0.0            2496           2          1248.0            1216            1280  prefn_marker_kernel


CUDA Memory Operation Statistics (nanoseconds)

Time(%)      Total Time  Operations         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  --------------------------------
   91.3       483848833           6      80641472.2           23519       284010193  [CUDA memcpy DtoH]
    8.7        45890176          14       3277869.7            1120        23990679  [CUDA memcpy HtoD]
    0.0            9600           6          1600.0            1184            2208  [CUDA memcpy DtoD]


CUDA Memory Operation Statistics (KiB)

            Total      Operations            Average            Minimum            Maximum  Name
----------------  --------------  -----------------  -----------------  -----------------  ------------------------
         862672.0               6           143778.7            148.535           500000.0  [CUDA memcpy DtoH]
         276206.0              14            19729.0              0.004           144453.0  [CUDA memcpy HtoD]
              7.0               6                1.2              0.158                6.0  [CUDA memcpy DtoD]
```

*The version based on Optimization 3 (GPU Version, 2-Kernel GPU Unrolling):*

```
Generating CUDA API Statistics...
CUDA API Statistics (nanoseconds)

Time(%)      Total Time       Calls         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ----------------------------
   65.0       536942309          20      26847115.4           30054       285669425  cudaMemcpy
   23.8       196845716          44       4473766.3            2148       188552650  cudaMalloc
    7.7        63959016          34       1881147.5            1196        32394245  cudaDeviceSynchronize
    2.1        17186381          34        505481.8            3774        16843255  cudaLaunchKernel
    1.3        10736203          44        244004.6            2393         2453740  cudaFree
    0.0           66823           6         11137.2            8963           13632  cudaMemcpyToSymbol



Generating CUDA Kernel Statistics...

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

Time(%)      Total Time   Instances         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ----------------------------
   75.5        48220852           6       8036808.7            5952        32387435  conv_forward_kernel
   19.7        12610624           6       2101770.7            7295         7128074  unroll_Kernel
    3.6         2316847           6        386141.2            1632         1340310  convertFP16toFP32
    1.1          722363          12         60196.9            1472          370397  convertFP32toFP16
    0.0            2880           2          1440.0            1440            1440  do_not_remove_this_kernel
    0.0            2624           2          1312.0            1280            1344  prefn_marker_kernel


CUDA Memory Operation Statistics (nanoseconds)

Time(%)      Total Time  Operations         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ----------------------------
   92.5        490556518           6      81759419.7           13023       284938298  [CUDA memcpy DtoH]
    7.5         39812658          14       2843761.3            1152        19899274  [CUDA memcpy HtoD]
    0.0            10368           6          1728.0            1312            2464  [CUDA memcpy DtoD]


CUDA Memory Operation Statistics (KiB)

            Total      Operations            Average            Minimum            Maximum  Name
----------------  --------------  -----------------  -----------------  -----------------  ----------------------------
         862672.0               6           143778.7            148.535           500000.0  [CUDA memcpy DtoH]
         276206.0              14            19729.0              0.004           144453.0  [CUDA memcpy HtoD]
              7.0               6                1.2              0.158                6.0  [CUDA memcpy DtoD]
```

*Nsight-Compute:*

*The version based on Optimization 4 (Kernel Fusion):*

*The version based on Optimization 3 (GPU Version, 2-Kernel GPU Unrolling):*


*From nsys, for the version based on Optimization 4, we can see a large cut down in cudaMalloc time, and although the kernel time for conv_forward_kernel decreases, the extra convert kernels take extra time, so that the sum of kernel times for kernel "conv_forward_kernel," kernels "convertFP32toFP16" and "convertFP16toFP32" are approximately the same as the previous conv_forward_kernel time for Optimization 4.*

*In Nsight-Compute, we can see increase in both SM and Memory utilization, which can explain why we have no improvement in Op-time.*

*From nsys, for the version based on Optimization 3 (GPU version), we can see that cudaMalloc time are halved, and the kernel time for both "conv_forward_kernel" and "unroll_Kernel" are halved. With the two extra kernels for converting between FP16 and FP32, the total kernel time is still much shorter than the original one. This explains the improvement in Op-time.*

*In Nsight-Compute, the SM utilization is decreased, but the Memory utilization is increased for conv_forward_kernel.*

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

*CUDA C++ Programming Guide: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#warp-matrix-functions*
*Github – NVIDIA Developer Blog Tensor Core Code Samples: Simple Tensor Core GEMM: https://github.com/NVIDIA-developer-blog/code-samples/blob/master/posts/tensor-cores/simpleTensorCoreGEMM.cu*
*Programming Tensor Cores in CUDA 9: https://developer.nvidia.com/blog/programming-tensor-cores-cuda-9/*

6. **Optimization 6: Tuning with restrict and loop unrolling (3 points)**

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

*I chose to do Tuning with restrict and loop unrolling, because our baseline situation is very suitable for including restricted pointers and do loop unrolling for the inner loops related with K.*

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 need to first change all the data related pointers into restricted pointers by adding "__restrict__" keyword before their names upon their definition, and then we can unroll the double "for" loop of p and q (K*K elements needed to be multiplied with the mask) for small K values. Here I have done a manual loop unrolling, we can also do the unrolling using "#pragma unroll <times to unroll>."*

*I think this optimization would increase performance, because of the following reasons:*
*By tuning with restrict, will solve the aliasing problem in C-type languages, announcing that these pointers cannot be the alias of any other pointers, so that the compiler will not spend time to infer whether there is an aliasing issue.*
*By unrolling loops, we can reduce the branch divergence caused by the "if" condition implied in the "for" loop. So that the number of compiled instructions will also be reduced, thus we can improve the performance.*

*This optimization is directly synergized with the M2 baseline. But in order to reach better performance, I changed TILE_WIDTH from 16 to 8.*

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 | *0.215484ms* | *0.570131ms* | *0m1.597s* | *0.86* |
| 1000 | *2.02813ms* | *5.58366ms* | *0m10.747s* | *0.886* |
| 5000 | *10.0617ms* | *27.8089ms* | *0m50.969s* | *0.871* |

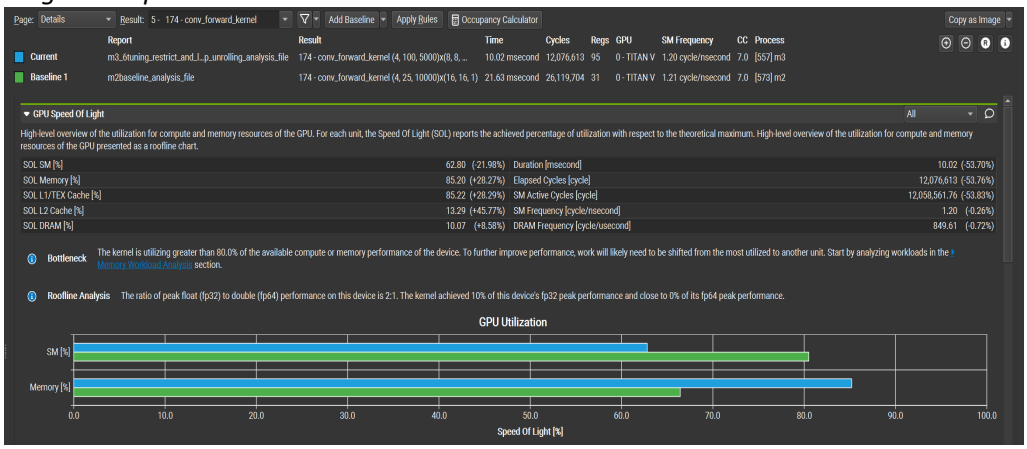
d. Was implementing this optimization successful in improving performance? Why or why not? Include profiling results from *nsys* and *Nsight-Compute* to justify your answer, directly comparing to your baseline (or the previous optimization this one is built off of).

*Implementing this optimization is successful in improving performance. Because by tuning restrict and loop unrolling, the device kernel execution time should be cut down, since both compilation time and compiled instruction numbers are reduced.*

*Nsys:*

```
Generating CUDA API Statistics...
CUDA API Statistics (nanoseconds)

Time(%)      Total Time       Calls         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ------------------------------
   70.7       555116411          20      27755820.6           30724       303961579  cudaMemcpy
   24.1       188880107          20       9444005.3            2320       185706365  cudaMalloc
    4.9        38398438          16       2399902.4             850        28081022  cudaDeviceSynchronize
    0.3         2565965          20        128298.2            2386          672018  cudaFree
    0.0          389723          10         38972.3           24681          126847  cudaLaunchKernel



Generating CUDA Kernel Statistics...

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

Time(%)      Total Time   Instances         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ------------------------------
  100.0        38362846           6       6393807.7            6944        28077845  conv_forward_kernel
    0.0            2752           2          1376.0            1344            1408  do_not_remove_this_kernel
    0.0            2464           2          1232.0            1216            1248  prefn_marker_kernel


CUDA Memory Operation Statistics (nanoseconds)

Time(%)      Total Time  Operations         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ------------------------------
   91.7       505103568           6      84183928.0           23200       303178274  [CUDA memcpy DtoH]
    8.3        45916034          14       3279716.7            1152        23994712  [CUDA memcpy HtoD]


CUDA Memory Operation Statistics (KiB)

            Total      Operations            Average            Minimum            Maximum  Name
----------------  --------------  -----------------  -----------------  -----------------  ------------------------
        862672.0               6           143778.7            148.535           500000.0  [CUDA memcpy DtoH]
        276206.0              14            19729.0              0.004           144453.0  [CUDA memcpy HtoD]
```

*Nsight-Compute:*



*From nsys, we can see that both cudaMemcpy and cudaMalloc time are halved. And for the kernel time, the kernel time consumption of "conv_forward_kernel" is reduced to the 40% of its original time, which indicates that our optimization truly improves the performance.*

*From Nsight-Compute, we can see that SM utilization is decreased for conv_forward_kernel, which means either we need less SM resources to complete the tasks, or we have less tasks to complete. Thus, we have the shorter Op-time.*
*The increase of Memory utilization is probably because we need more space to store the instructions from the unrolled loops.*

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

*CUDA C++ Programming Guide: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#restrict*
*Lecture Slides: Lecture 22 Accelerating Matrix.*