Project Milestone 3 Report, Jack Tipping

(Apologize fro grammar was rushing at end).

**OP 0, Streams:**

**How does the optimization work? How does this method theoretically optimize your convolution kernel? Expected behavior?**

Streams work by overlapping mem copies with kernel execution. Due to the run time of memcpy being a limiting factor, this works to theoretically greatly decrease op time.

**How did you implement your code? You must thoroughly explain the thought process behind your implementation with code snippets and justify the correctness of your implementation with proper profiling results.**

Layer Time: 155.561 ms

Op Time: 0.00586 ms

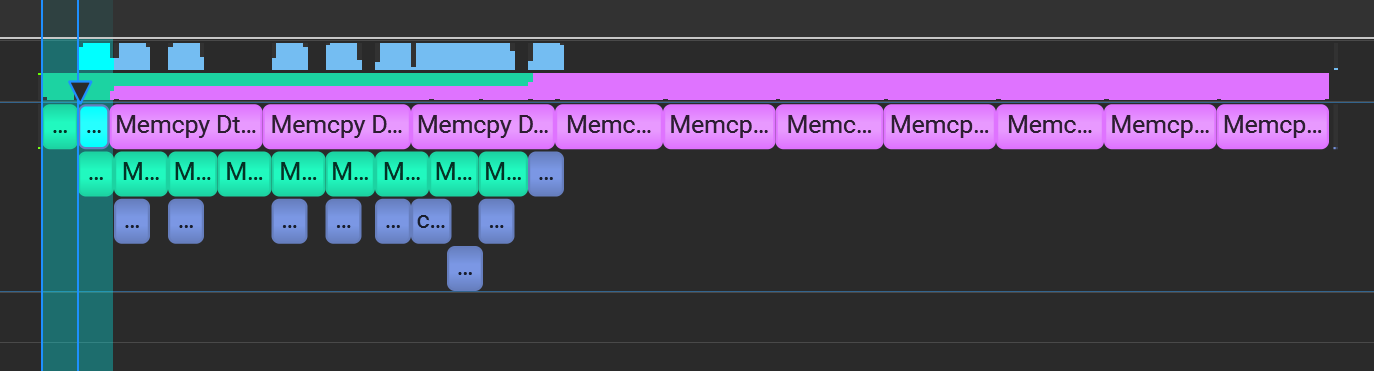
Conv-GPU==

prolog complete

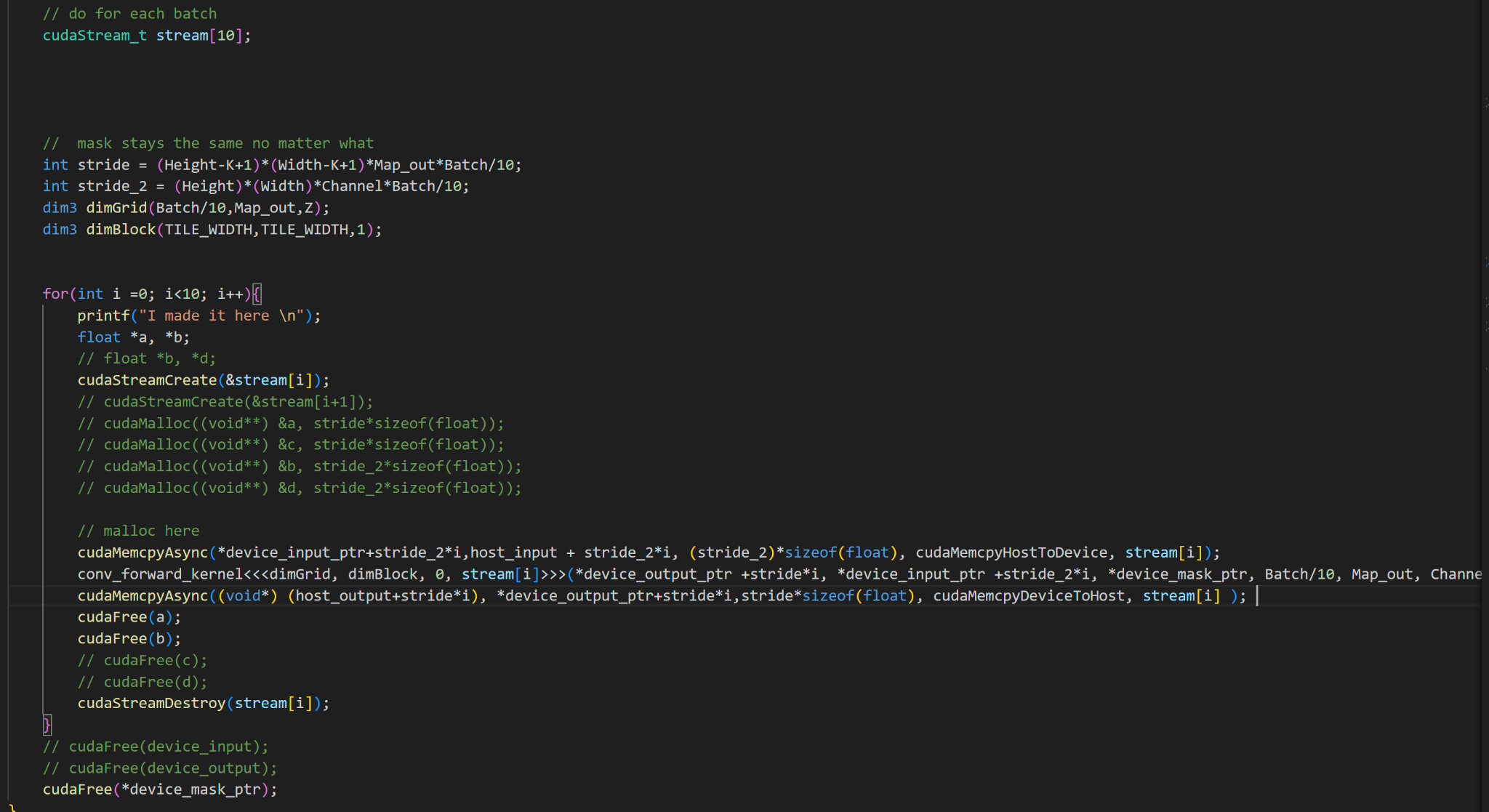
Layer Time: 134.771 ms

Op Time: 0.004709 ms

Test Accuracy: 0.8714

****

As you can see there is overlap, I would have hoped to get a bit more of a pipeline, but this shows how mem copies are happening during kernel calls/mem copies to devices. This means that our intended functionality is occurring. Additionally, the accuracy matches milestone 2 so functionality is as expected for overlapping and doesn’t create inaccuracies.



Explaining the thought process further, looking at this code 10 streams are created in an array. Then a for loop is hit looping through input sizes divided by 10. Each time a cuda stream is created and then memcpys happen asynchronously in addition to kernel calls creating the overlapping.

**Did the performance match your expectations? Analyze the profiling results as a scientist.**

Looking at the structure of calls, op times apparently is broken and I assume that is because the streams mess with how the length of a kernel call is seen. Overall the profiling results match.

**Does the optimization synergize with any of the other optimizations? How?**

Because streams involve memcpy\_s it synergizes with all operations technically, as all require a memcpy. The one it synergizes the best with is probably unroll since I loop through batches and while I loop I could start streams to optimize. Although, due to the batch size we would have to be working with larger data sets to make it worth it.

**What references did you use when implementing this technique?**

I referenced lecture slides, which gave general structures and some syntax for using streams.

**OP 2, Unrolling:**

**How does the optimization work? How does this method theoretically optimize your convolution kernel? Expected behavior?**

The idea is to change the forward aspect of the CNN to one single matrix multiply kernel. This is done by unrolling the given matrix into a matrix which can be multiplied with the mask to get the desired input. With a speedy matrix multiplication we should be able to reduce times.

**How did you implement your code? You must thoroughly explain the thought process behind your implementation with code snippets and justify the correctness of your implementation with proper profiling results.**

Conv-GPU==

prolog complete

Layer Time: 332.169 ms

Op Time: 159.463 ms

Conv-GPU==

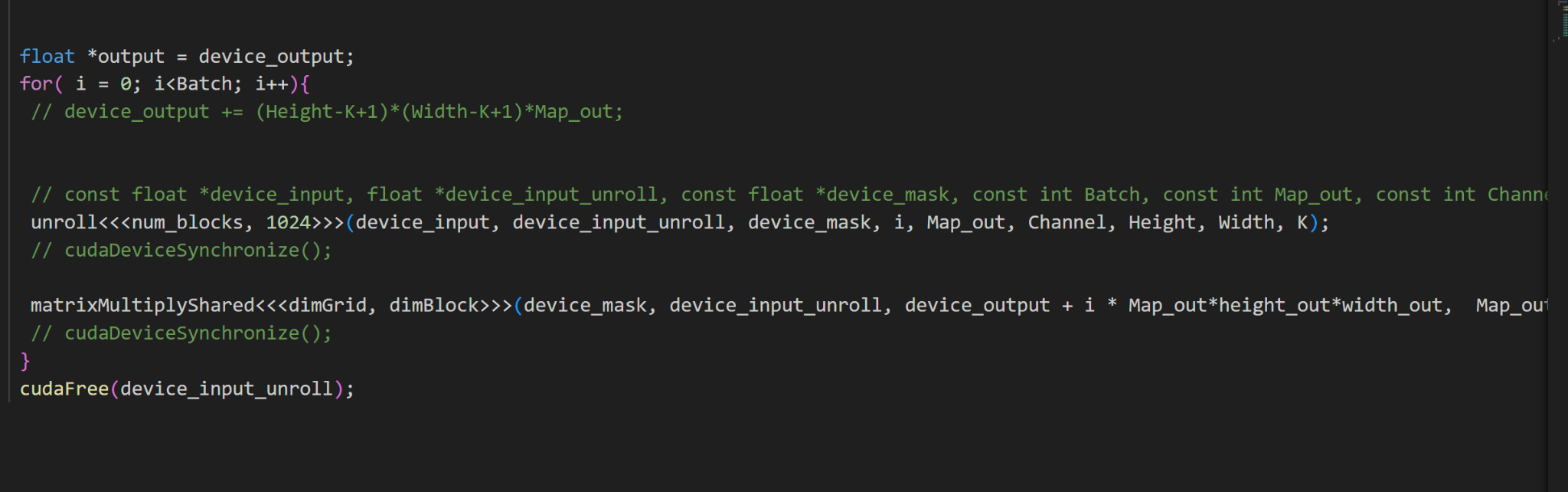
prolog complete

Layer Time: 305.851 ms

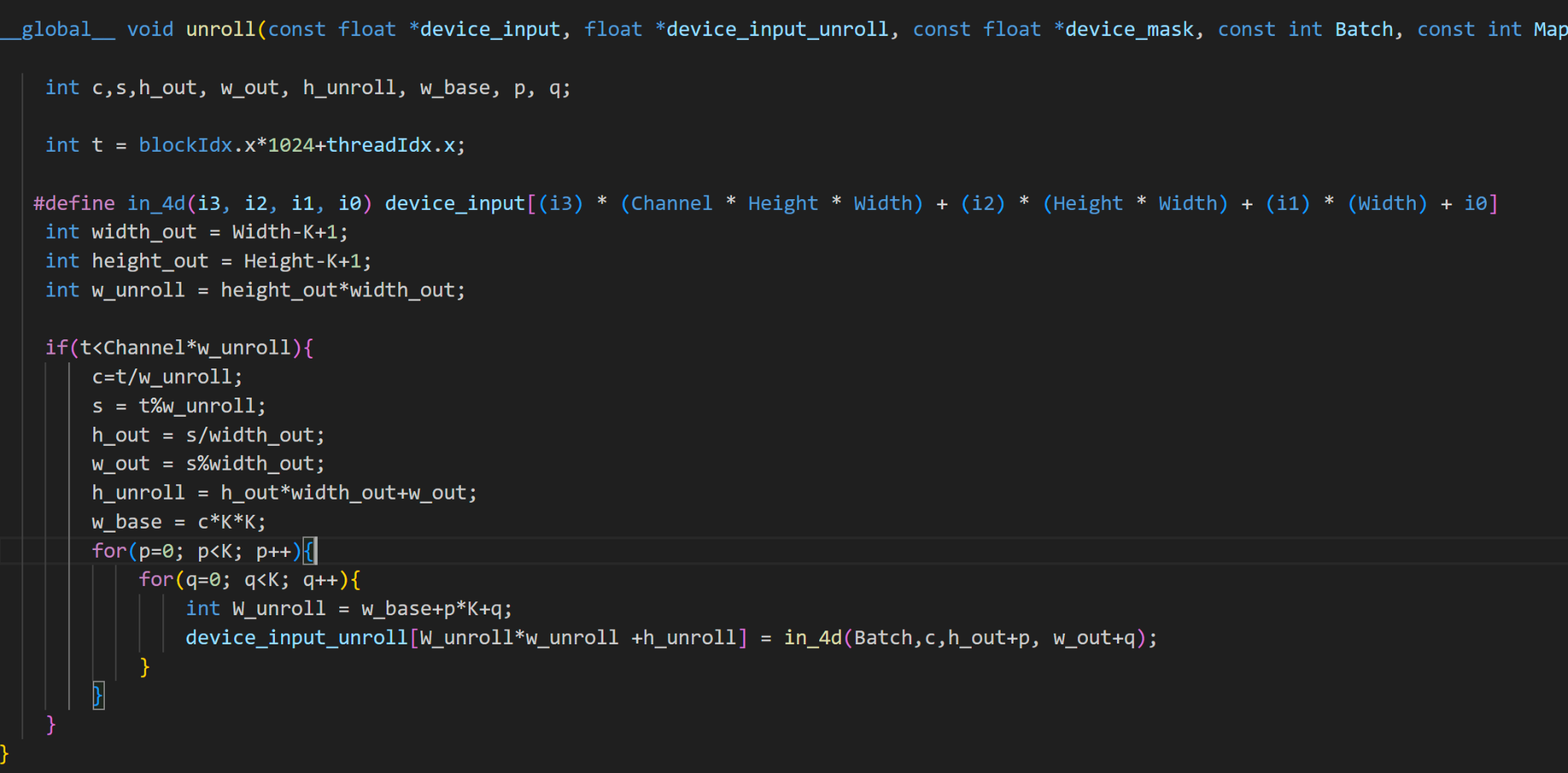
Op Time: 178.177 ms

Test Accuracy: 0.8714

My optimes show that accuracy was met but times were way off. I will discuss more below on why that may be the case.



For code I start with the host code. The for loop exists because the unroll covers unrolling a matrix per batch, hence we must loop over batches to be able to perform the unroll. Num\_blocks in this case is equal to our targeted size which is width out times by height out times by channels. Now looking at unroll code,



Here we make a matrix of H\_unroll and w\_unroll, we do this through various index parameters which essentially index one batch of the input array correctly. Where we loop through mask size (without map out) and adjust to output size width out and height out.

Afterwards reference the shared memory matrix multiplication kernel where I multiply the mask with the unrolled matrix to get the appropriate output matrix values and size.

**Did the performance match your expectation? Analyze the profiling results as a scientist.**

The unroll did not work as intended, as it did lead to an increase in times. I think this is because in the textbook they mentioned the main point was that the unroll matrix would take advantage of the very fast gemm in the textbook (which is included in the standard library.) This gemm most likely takes advantage of things like tensor cores, so my feeble matrix multiply kernel simply isn’t fast enough to take advantage of the unroll.

**Does the optimization synergize with any of the other optimizations? How?**

It would synergize well with streams as I loop over batches, if I reduced the amount I looped and integrated with streams it would be easily integrated.

**What references did you use when implementing this technique?**

I referenced the textbook where they went in depth into doing unrolling. I also used my matrix multiply kernel from another lab.

**OP 1, Shared Tiling:**

**How does the optimization work? How does this method theoretically optimize your convolution kernel? Expected behavior?**

Streams work by overlapping mem copies with kernel execution. Due to the run time of memcpy being a limiting factor, this works to theoretically greatly decrease op time.

**How did you implement your code? You must thoroughly explain the thought process behind your implementation with code snippets and justify the correctness of your implementation with proper profiling results.**

Test batch size: 10000

Loading fashion-mnist data...Done

Loading model...Done

Conv-GPU==

prolog complete

Layer Time: 198.737 ms

Op Time: 12.5463 ms

Conv-GPU==

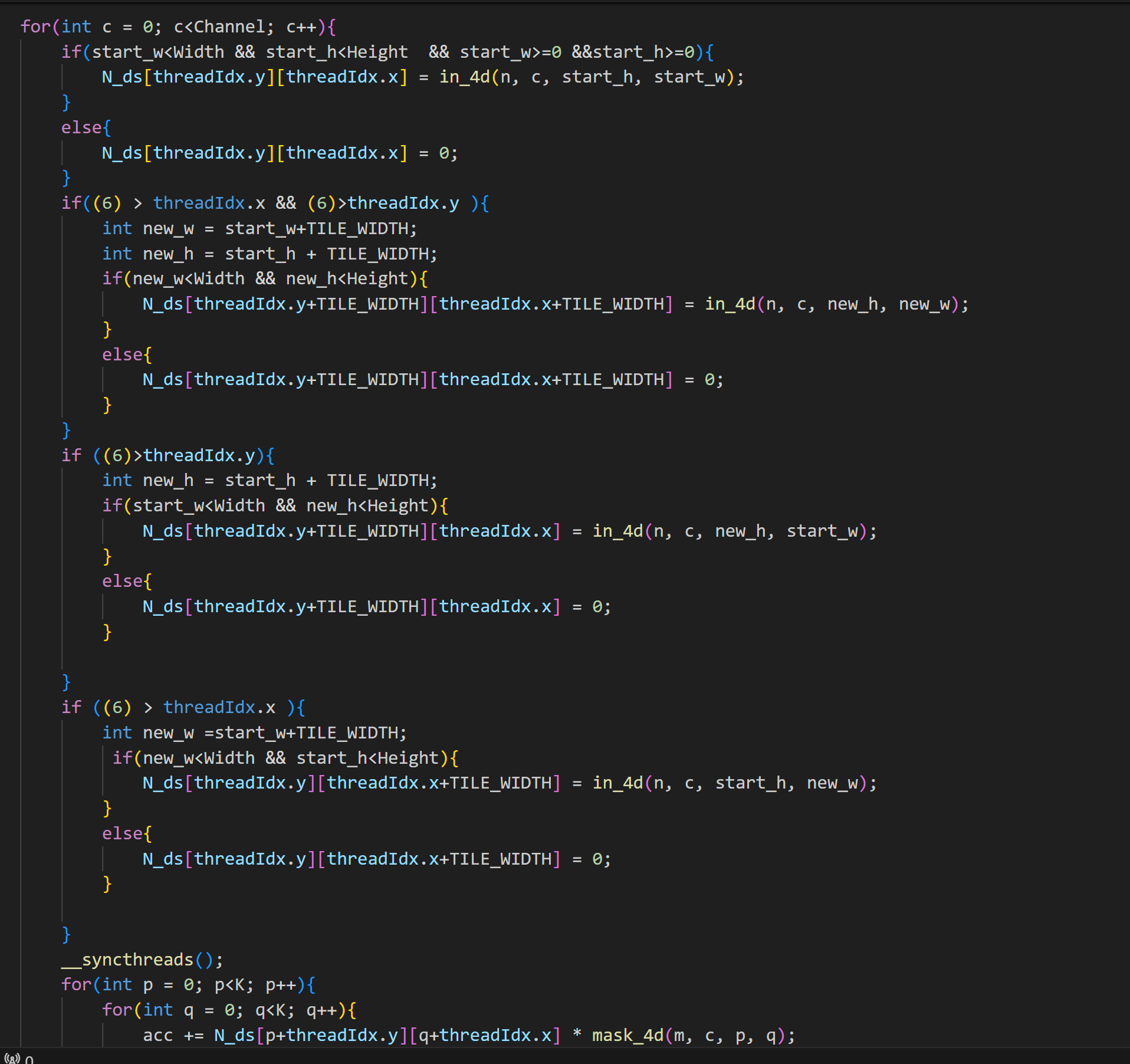
prolog complete

Layer Time: 195.905 ms

Op Time: 67.9961 ms

Test Accuracy: 0.8714

And this is my code



Above this is a tile of size [TILEWIDTH+6][TILEWIDTH+6], the if statements fill out the tile shared among threads and checks if the thread should be filling in two spaces. Once the tile is loaded I simply start from the thread location where it is convoluting over and loop over mask width/height. Below is nsys which shows shared memory is used.

**Did the performance match your expectation? Analyze the profiling results as a scientist.**

I expected to get more benefits from the tiling, but perhaps the tile size simply isn’t optimal for this problem. Additionally, I used strategy 2 and my thread reuse is low so I don’t get as good optimes as strategy 3 for example.

**Does the optimization synergize with any of the other optimizations? How?**

This synergizes with any optimizations that involve the original kernel as it simply tiles the input with shared mem to help increase speeds.

**What references did you use when implementing this technique?**

I referenced lecture slides, which gave general structures and some syntax for using strategy 2.

**OP 4, Weight Matrix Constant Mem:**

**How does the optimization work? How does this method theoretically optimize your convolution kernel? Expected behavior?**

Use constant memory to reduce memory access time. So looking at what's constant and small enough in size, the mask fits this.

**How did you implement your code? You must thoroughly explain the thought process behind your implementation with code snippets and justify the correctness of your implementation with proper profiling results.**

Test batch size: 10000

Loading fashion-mnist data...Done

Loading model...Done

Conv-GPU==

this is k 7

this is map\_out 4

prolog complete

Layer Time: 178.974 ms

Op Time: 10.0524 ms

Conv-GPU==

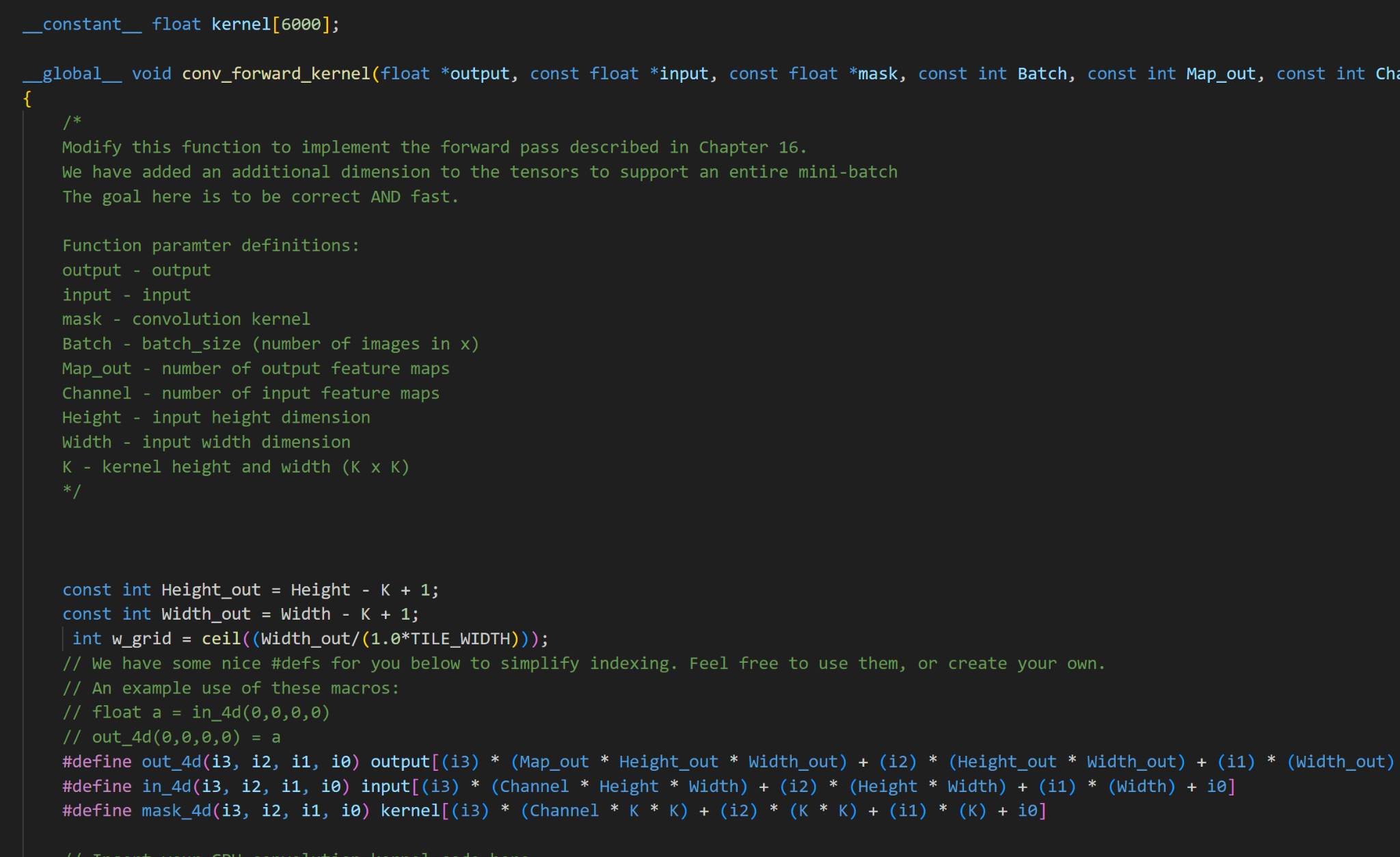
this is k 7

this is map\_out 16

prolog complete

Layer Time: 170.303 ms

Op Time: 43.3167 ms



Basically three changes were necessary, adding constant mem kernel, replacing mask\_4d with kernel and then using cuda memcpy symbol. Since all we need to do is use constant mem and the mask is small enough to fit into constant mem (about 2700 floats) we replaced the mask with the kernel. Below shows the constant mem usage.

**Did the performance match your expectation? Analyze the profiling results as a scientist.**

Constant memory is extremely fast, but lacks size, so the gains in speed are noticeable, but due to the inability to have a large size they aren’t super impactful. Due to this, the results are as expected.

**Does the optimization synergize with any of the other optimizations? How?**

Can synergize with every optimization since it just declares the mask in constant memory.

**What references did you use when implementing this technique?**

I referenced Lab 4 where we used constant memory.

**OP 5, Tuning with Restrict and Unrolling:**

**How does the optimization work? How does this method theoretically optimize your convolution kernel? Expected behavior?**

Restrict simply says two pointers do not overlap in memory, a consideration most people take but in this case it acts simply as a promise to the compiler. The unrolling acts to literally unroll a for loop, basically get rid of the loop to speed up op times.

**How did you implement your code? You must thoroughly explain the thought process behind your implementation with code snippets and justify the correctness of your implementation with proper profiling results. (ignore the print statements that are excess)**

Test batch size: 10000

Loading fashion-mnist data...Done

Loading model...Done

Conv-GPU==

this is k 7

this is map\_out 4

prolog complete

Layer Time: 186.151 ms

Op Time: 11.4559 ms

Conv-GPU==

this is k 7

this is map\_out 16

prolog complete

Layer Time: 176.694 ms

Op Time: 50.9788 ms

[jtipping@dt-login01 Project]$ cat m2.out

Test batch size: 10000

Loading fashion-mnist data...Done

Loading model...Done

Conv-GPU==

this is k 7

this is map\_out 4

prolog complete

Layer Time: 186.151 ms

Op Time: 11.4559 ms

Conv-GPU==

this is k 7

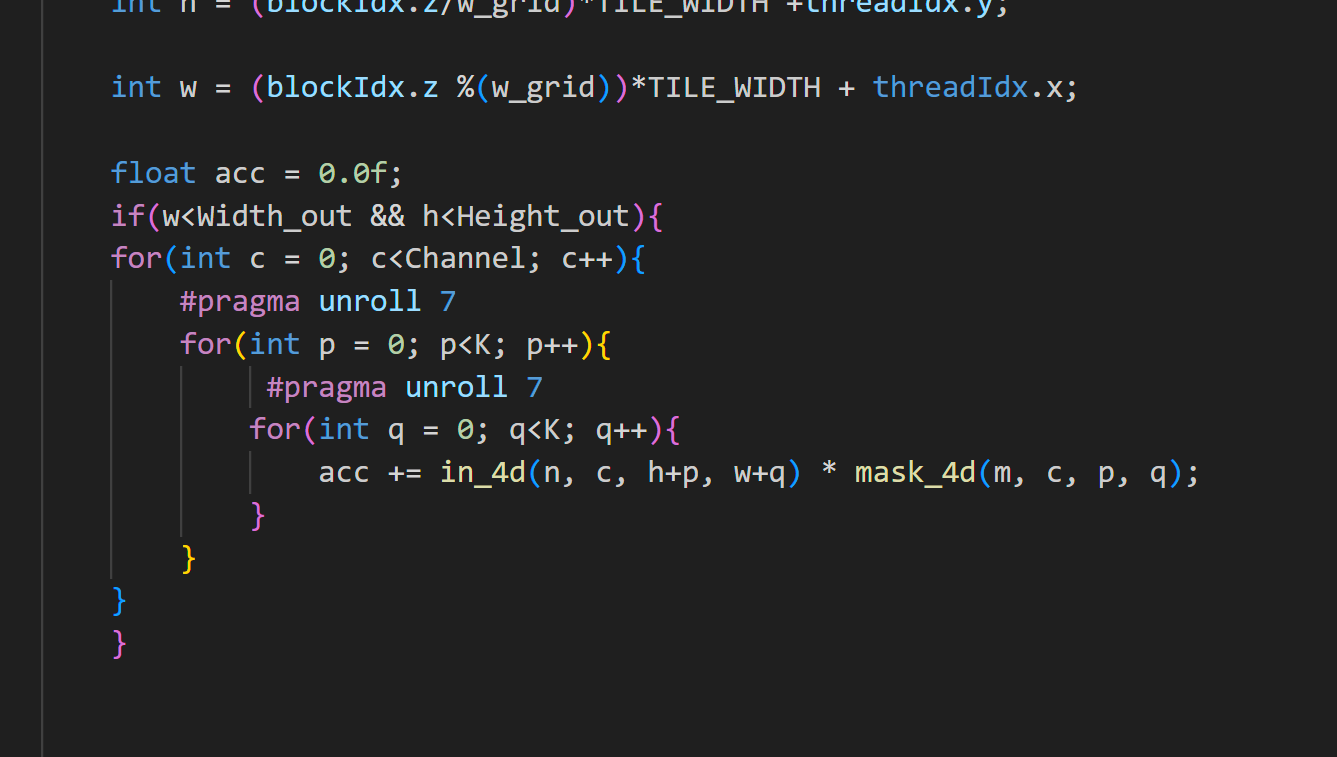
this is map\_out 16

prolog complete

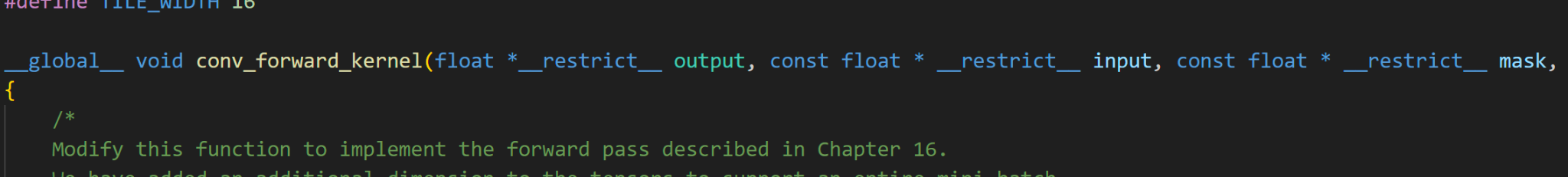
Layer Time: 176.694 ms

Op Time: 50.9788 ms

Test Accuracy: 0.8714



And



I chose 7 for the pragma unroll, since K = 7 so then pragma unroll does the entire for loop. Restrict is placed on all dynamically allocated memory which tells the compiler as I said above. Looking at nsys there isn’t too much to show, just understand that op times increased from base and remained somewhat similar to the other op times.

**Did the performance match your expectation? Analyze the profiling results as a scientist.**

Like constant mem, this speeds a small area unrolling the for loop, which technically doesn’t change too much other than making all code seen sequentially. So over all batches there are small incremental improvements which cause it to have a small noticeable speed up.

**Does the optimization synergize with any of the other optimizations? How?**

Restrict works with any of the optimizations as all use pointers. Pragam unroll works with all for loops, so hence all optimizations unless one gets rid of for loops which is unlikely.

**What references did you use when implementing this technique?**

I referenced <https://forums.developer.nvidia.com/t/pragma-unroll/3042>.

**OP 10, FP16 only \_\_half:**

**How does the optimization work? How does this method theoretically optimize your convolution kernel? Expected behavior?**

FP 16 moves 32-bit floating point into 16 bit, 16 bit operations have a faster run time. Due to this we can expect faster run times.

**How did you implement your code? You must thoroughly explain the thought process behind your implementation with code snippets and justify the correctness of your implementation with proper profiling results.**

Test batch size: 10000

Loading fashion-mnist data...Done

Loading model...Done

Conv-GPU==

this is k 7

this is map\_out 4

prolog complete

Layer Time: 185.63 ms

Op Time: 11.767 ms

Conv-GPU==

this is k 7

this is map\_out 16

prolog complete

Layer Time: 182.597 ms

Op Time: 54.0189 ms

Test Accuracy: 0.8716



The idea here is to change all arithmetic to half arithmetic, and then convert back at the to upload out value as float. I would have preferred to use hfma, but unfortunately could not get it to work.

**Did the performance match your expectation? Analyze the profiling results as a scientist.**

I expected a bit more noticeable of an improvement, but at least improvement is seen. Since there is 7x7 multiply and add in one kernel call I thought it would add up more than it did. I assume using \_\_half2 would have the most noticeable gains.

**Does the optimization synergize with any of the other optimizations? How?**

Synergizes with all that include operations such as add subtract or multiply.

**What references did you use when implementing this technique?**

I referenced the mp3 slides and the nVidia website that contains APIs.

**OP 6, sweeping tile width:**

Taking a difference approach here laying out data and then answering the questions.

| TILE WIDTH | OPTIME (ms) |
| --- | --- |
| 13 | 399.15 |
| 14 | 399.942 |
| 15 | 401.015 |
| 16 | 337.64 |
| 17 | 399.546 |
| 18 | 399.97 |
| 19 | 399.765 |
| 20 | 400.394 |
| 21 | 396.732 |
| 22 | 399.942 |
|  |  |
|  | Sum off both op times I see |
|  |  |

**How does the optimization work? How does this method theoretically optimize your convolution kernel? Expected behavior?**

My goal is to test different tile widths and compare the op times. I have no idea what to expect as I am simply testing.

**Did the performance match your expectation? Analyze the profiling results as a scientist.**

I remembered tile width being a power of 2 being helpful, but I couldn’t remember why, but then I remembered for sharing memory a TILE WIDTH which is a power of 2 will be able to better align with memory. Due to the better memory alignment we see an improvement in the performance at 16 as 16 is a power of 2.

**Does the optimization synergize with any of the other optimizations? How?**

Synergizes with all operations which do shared memory tiling.

**What references did you use when implementing this technique?**

Nothing, just collected data.