**Milestone 3 Report**

1. **Baseline:**

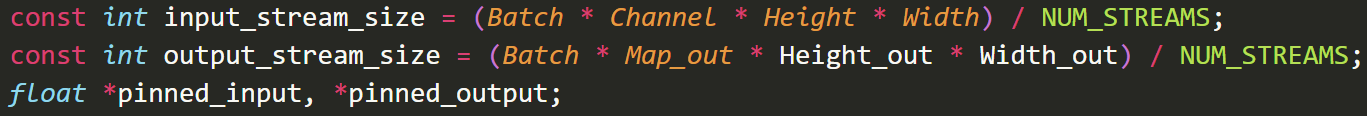
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Batch Size | Op Time 1 | Op Time 2 | Total Execution Time | Accuracy |
| 100 | *0.182773 ms* | *0.469821 ms* | *0m1.351s* | *0.86* |
| 1000 | *1.19884 ms* | *4.51816 ms* | *0m9.315s* | *0.886* |
| 10000 | *11.4459 ms* | *44.9767 ms* | *1m30.283s* | *0.8714* |

1. **Optimization Number 0: \_\_Streams\_\_**
   1. How does this optimization work in theory? Expected behavior?

CUDA streams facilitate asynchronous execution of operations on the GPU, enabling concurrent execution of kernels and memory transfers. They allow for better resource utilization by overlapping computation and memory operations, organized within streams, and provide flexibility in scheduling and controlling the execution flow. While operations within a stream are executed in the order they are submitted, across streams, there's no guaranteed order of execution without explicit synchronization. Streams play a crucial role in maximizing GPU performance by enabling parallelism, independence of operations, and efficient resource management, ultimately contributing to accelerated computation for a wide range of applications.

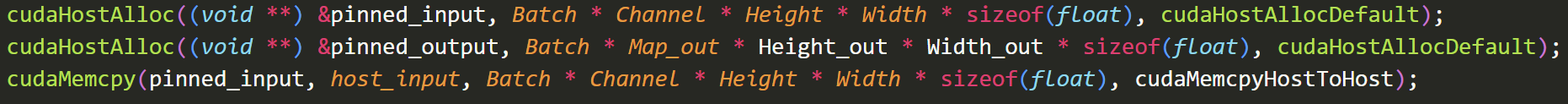
* 1. How did you implement your code? Explain thoroughly and show code snippets.

I began implementing my code by migrating the conv\_forward\_gpu and conv\_forward\_gpu\_epilog portions to the conv\_forward\_prolog function. While initially the host steps were separated in milestone 2, I knew streams demanded memory copying and kernel invocation side by side. First, I created variables for the sizes of an input stream and output stream to be used as offsets. I also initialized pointers to pinned memory and globally defined an arbitrary number of 20 streams to be optimized later.

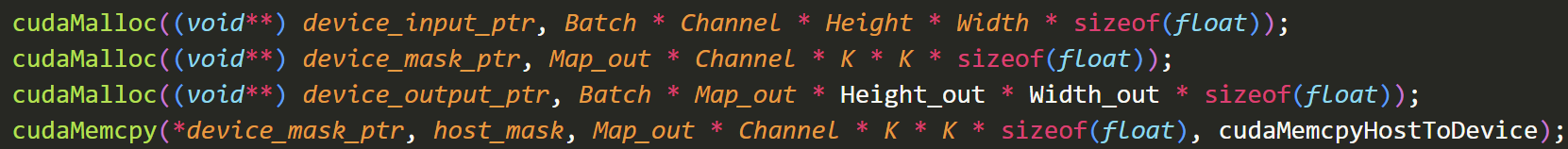




Then, I allocated pinned memory on the host for the input and output data, and pinned the input memory via memcpy.



I allocated memory on the device for the input, mask, and output, and copied the mask from the host to the device.

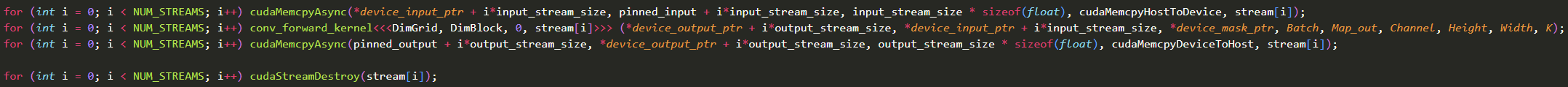


I changed the z-axis of DimGrid to account for partitioning the batches into streams and created the steam using CUDA APIs.

A screen shot of a computer code

Description automatically generated

I asynchronously copied the pinned input memory to the device input memory, invoked the kernel, and asynchronously copied the device output memory to the pinned input memory. Each of these steps were performed in separate for loops to maximize performance gain, and memory locations for these were all offset by iteration i \* respective stream size. The streams were subsequently destroyed.



I copied the pinned output memory back to the regular host memory and freed the device and pinned memory.

A black screen with colorful text

Description automatically generated

* 1. Did the performance match your expectation? Show your analysis results using profiling tools.

Op times are not relevant since my implementation was not using conv\_forward\_gpu. The nsys report showed many kernels being invoked, all with much shorter durations and elapsed cycles compared to the baseline report.

A screenshot of a computer

Description automatically generated

The green is memory copy from host to device, the blue is the kernel, and the purple is memory copy from device to host. One can clearly see computation and memory operations are being overlapped to improve performance, matching my expectation.

* 1. Does this optimization synergize with any other optimizations? How and why?

Implementing streams to overlap computation with data transfer can synergize well with several other optimizations, particularly those aimed at reducing computation and memory access latency. For example, when combined with tiled convolution and tiled matrix multiplication using shared memory, streams can help hide the latency of memory transfers by allowing data to be transferred asynchronously while computation on other streams proceeds concurrently. This overlap of computation and data transfer maximizes the utilization of GPU resources, leading to improved overall performance. Additionally, when coupled with input channel reduction using atomics or FP16 arithmetic implementation, streams can further enhance efficiency by minimizing the time spent waiting for data transfers, thereby keeping the GPU cores busy with meaningful computation tasks.

* 1. List your references used while implementing this technique. (you must mention textbook pages at the minimum)
* While Chapter 13 was not available on the course site, here was some useful NVIDIA documentation: <https://docs.nvidia.com/cuda/archive/10.1/cuda-c-programming-guide/#streams>
* Course slides Lecture 20

1. **Optimization Number 1: \_\_Tiled Convolution\_\_**
   1. How does this optimization work in theory? Expected behavior?

Tiled convolution using shared memory in CUDA involves partitioning the input data into tiles that fit within the shared memory of each CUDA block. Each block loads a tile of input data and a corresponding tile of the convolution kernel into shared memory, allowing for efficient reuse of data due to shared memory's low latency and high bandwidth. Threads within a block then perform convolutions by accessing the shared memory tiles, minimizing global memory accesses. The expected behavior is that each thread block independently computes a portion of the output, exploiting data locality and reducing memory traffic, ultimately accelerating the convolution operation compared to a naive implementation.

* 1. How did you implement your code? Explain thoroughly and show code snippets.

I implemented my code using strategy 2, where each thread loads on input elements and only some threads compute an output. I began by calculating block\_width using the TILE\_WIDTH and mask width, K, and adjusted the DimBlock and shared mem size accordingly.

A black screen with yellow and orange text

Description automatically generated

Transitioning to the kernel, I instantiated my shared memory to grab the size defined from the host’s kernel invocation. I also defined tx, ty, and idx variables to be used in the tiled convolution.

A screen shot of a computer code

Description automatically generated

I looped through all of the channels for loading shared memory and performing multiplication, but not writing the final sum back to the output. I loaded the tile from the input if h and w were within the input’s height and width and synchronized the threads.

A black screen with colorful text

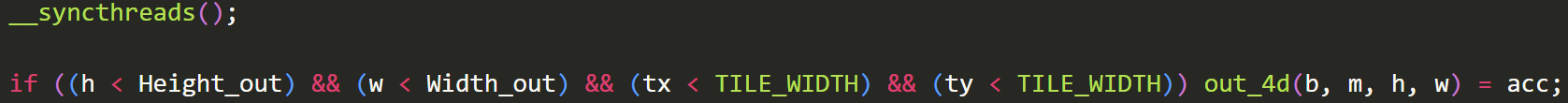
Description automatically generated

If the thread was within the tile, I looped through the mask using a double for loop and multiplied the tile element by the mask element. The tile was indexed in row-major order by using the thread indexes in summation with the respective for loop index. This was added back to an accumulator variable to be written back to the output later. Threads were then synchronized.

A black screen with colorful text

Description automatically generated

After looping through the channels, I synchronized the threads one final time. Then, if h and w were less than the output height and output width respectively, and the thread indices were still within the tile, acc was written back to its output.



* 1. Did the performance match your expectation? Show your analysis results using profiling tools.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Batch Size | Op Time 1 | Op Time 2 | Total Execution Time | Accuracy |
| 100 | *0.259146 ms* | *0.946283 ms* | *0m2.550s* | *0.86* |
| 1000 | *1.74135 ms* | *9.29663 ms* | *0m9.261s* | *0.886* |
| 10000 | *16.8 ms* | *92.804 ms* | *1m29.997s* | *0.8714* |

A computer screen shot of a diagram

Description automatically generated

I did not expect op time to increase, but this may be due to the additional overhead from shared memory and tiled convolution. Firstly, the overhead of managing shared memory itself can add latency. While shared memory is faster than global memory, there are additional instructions required to load and store data into shared memory, as well as synchronization overhead when multiple threads access the same shared memory bank simultaneously. Secondly, the process of tiling and reorganizing data to fit into shared memory and to perform tiled convolution introduces additional computational overhead. This overhead includes the need for additional indexing calculations to access the tiled data, as well as potential inefficiencies in the tiling process itself, such as padding or boundary conditions.

* 1. Does this optimization synergize with any other optimizations? How and why?

Tiled (shared memory) convolution can synergize with input matrix unrolling & tiled matrix multiplication using shared memory. By utilizing shared memory for both convolution and matrix multiplication, you ensure that data is efficiently reused, minimizing memory access latency. Additionally, if you're overlapping computation with data transfer using streams, tiled convolution allows for more efficient data reuse within the limited shared memory space, enabling better overlap and overall performance gains.

* 1. List your references used while implementing this technique. (you must mention textbook pages at the minimum)
* Textbook Chapter 7
* Course slide lecture 8

1. **Optimization Number 2: \_\_Unrolling & Matrix Multiplication\_\_**
   1. How does this optimization work in theory? Expected behavior?

Input matrix unrolling is a preprocessing step in convolutional neural networks (CNNs) where the input feature map is transformed into a 2D matrix, enabling efficient convolution operations. This transformation rearranges the spatial dimensions of the input into a flattened form that aligns with the convolutional kernel. Tiled matrix multiplication using shared memory in CUDA involves partitioning the input, kernel, and output matrices into smaller tiles that fit within the available shared memory of a CUDA block. Each block is responsible for computing a subset of the output matrix by loading tiles of the input and kernel matrices into shared memory, performing the matrix multiplication within the block, and storing the results. The expected behavior is parallelized matrix multiplication, leveraging shared memory to minimize data movement and maximize computational throughput, leading to faster convolutional operations on GPUs.

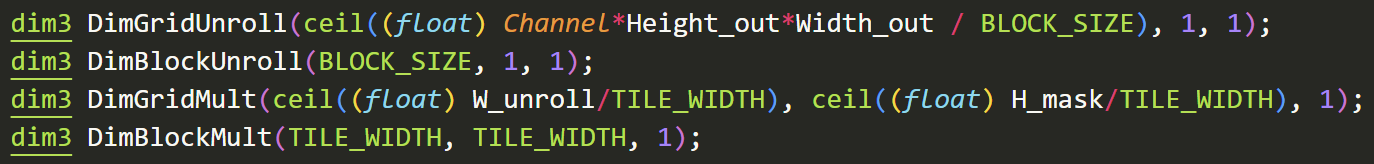
* 1. How did you implement your code? Explain thoroughly and show code snippets.

After reviewing the course slides, I began by instantiating relevant dimensions and allocating memory for the unrolled input matrix.

A screen shot of a computer

Description automatically generated

I created grid and block dimensions for the unroll kernel working in one (x) direction using a block size of 32 threads. I also created grid and block dimensions for the tiled matrix multiplication similar to lab 3. However, this time, the rows and columns of the output matrix were height of the mask and width of the unrolled input matrix respectively.



Looping through each batch, I invoked both kernels, passing the unrolled input matrix between the two. It was also important to offset the output pointer based on which batch was running.

**A black screen with colorful text

Description automatically generated**

In the unroll kernel, I instantiated my index and performed a boundary check to ensure it was within range for unrolling. After studying the textbook, I was able to calculate h\_out and w\_out indices. Then, I looped through the mask using p and q and calculated h\_unroll and w\_unroll based on the unrolling layout. Using all of these indices, I was able to index into the original input and assign it to the unrolled input with its own index.

A computer screen with text

Description automatically generated

In the multiplication kernel, I instantiated the shared memory for the mask and (unrolled) input, column and row indices, and accumulator p.

A computer screen shot of text

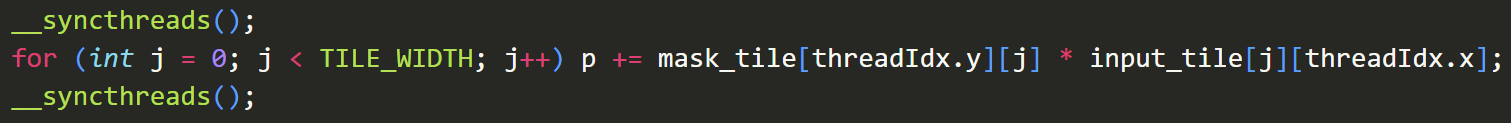
Description automatically generated

Looping through each tile, I first loaded the mask in shared memory tiles. This was performed horizontally by checking that each row was within the height of the mask and current thread (with the correct tile offset) was within the width. Similarly, the input was loaded into shared memory tiles. However, this was done vertically, checking that the col was less than the width of the input and the current thread was within the height. For either mask or input, if the boundary checking was not met, zeroes were loaded into the tiles.

A computer code on a black background

Description automatically generated

After syncing threads to make sure all shared tiles were loaded, I looped through each tile and calculated the partial sum with proper indexing. Once the sum was complete, I synced threads another time to make sure the product was accurate.



Finally, since the output had the dimensions of the height of the mask and width of the unrolled input matrix, if the row and column were within these bounds respectively, the product was written back to the output at the proper location.



* 1. Did the performance match your expectation? Show your analysis results using profiling tools.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Batch Size | Op Time 1 | Op Time 2 | Total Execution Time | Accuracy |
| 100 | *2.5253 ms* | *2.62184 ms* | *0m1.256s* | *0.86* |
| 1000 | *24.1917 ms* | *24.3383 ms* | *0m9.323s* | *0.886* |
| 10000 | *242.497 ms* | *245.431 ms* | *1m30.713s* | *0.8714* |

A screenshot of a computer

Description automatically generated

I did expect op times to considerably increase for this optimization. Input unrolling expands the input data into a larger matrix format, which increases memory usage and can lead to more frequent memory accesses, impacting overall performance. Similarly, tiled matrix multiplication using shared memory involves additional steps such as data movement between global and shared memory, thread synchronization, and managing shared memory bank conflicts, all of which introduce overhead. While these optimizations aim to enhance data reuse and computation efficiency, the added complexity and overhead can sometimes outweigh the performance gains.

* 1. Does this optimization synergize with any other optimizations? How and why?

This optimization does not synergize well with other optimizations because of its unique logical complexity. Kernel fusing may have helped this optimization, although I chose to not implementation that optimization. FP16 may be a synergetic choice, however, as FP16 arithmetic inherently requires less memory bandwidth and storage compared to FP32 arithmetic.

* 1. List your references used while implementing this technique. (you must mention textbook pages at the minimum)
* Textbook Chapter 16 pages 16-23
* PM3 tips lecture slides

1. **Optimization Number 4: \_\_Kernel in Constant Memory\_\_**
   1. How does this optimization work in theory? Expected behavior?

Putting the weight matrix (kernel) in constant memory is an optimization technique in CUDA that leverages the read-only nature of the kernel during execution. Constant memory provides fast, read-only access with caching capabilities, which can significantly reduce memory access latency and bandwidth usage. By storing the kernel in constant memory, CUDA ensures that the kernel elements are efficiently cached and accessed by all threads in the thread block, leading to improved memory throughput and overall performance during kernel execution. This optimization is particularly effective when the kernel remains constant throughout the execution of the CUDA kernel, as it maximizes memory access efficiency and minimizes redundant memory transfers.

* 1. How did you implement your code? Explain thoroughly and show code snippets.

I began by defining a mask size that provides enough memory for the code to perform properly without exceeding the maximum hardware constraints of constant memory and instantiated the mask as constant memory.

A black background with text

Description automatically generated

I removed the device mask as an input to the kernel and copied over the host\_mask memory to constant memory instead.



I removed any remaining references to device mask memory since it was not being used anymore.

* 1. Did the performance match your expectation? Show your analysis results using profiling tools.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Batch Size | Op Time 1 | Op Time 2 | Total Execution Time | Accuracy |
| 100 | *0.122469 ms* | *0.395161 ms* | *0m1.394s* | *0.86* |
| 1000 | *0.88601 ms* | *3.76023 ms* | *0m9.239s* | *0.886* |
| 10000 | *8.54579 ms* | *37.2947 ms* | *1m29.758s* | *0.8714* |

A screenshot of a computer

Description automatically generated

The performance did match my expectations, decreasing op times and significantly improving memory throughput. This is because constant memory is leveraged with much shorter cycle time, alleviating the memory bottleneck and improving overall performance.

* 1. Does this optimization synergize with any other optimizations? How and why?

This optimization synergizes well with practically any other optimizations. In this project, the kernel is guaranteed to be read-only, so this is a simple optimization to get a performance gain that will work alongside any other optimization.

* 1. List your references used while implementing this technique. (you must mention textbook pages at the minimum)
* Textbook Chapter 7 pages 8-11
* Course slides lecture 7

1. **Optimization Number 5: \_\_Tuning\_\_**
   1. How does this optimization work in theory? Expected behavior?

The pragma unroll optimization, combined with the use of the restrict keyword, aims to enhance performance by instructing the compiler to unroll loops at compile time, reducing loop overhead and improving instruction-level parallelism. The restrict keyword informs the compiler that pointer accesses through a particular pointer do not overlap with other pointer accesses, enabling more aggressive optimizations. Loop unrolling replicates loop bodies, allowing for better utilization of processor resources such as instruction pipelining and instruction cache. Together, these optimizations can lead to faster execution times, particularly in scenarios where loop iterations are known at compile time, as the compiler can generate more efficient code.

* 1. How did you implement your code? Explain thoroughly and show code snippets.

I used #pragma unroll before each for loop to unroll them, effectively replicating the loop bodies. While pragma handled the tuning autonomically, tuning the unrolling to perform 3\*7\*7 lines would work best since we know Channel = 3 and K = 7.

A screen shot of a computer code

Description automatically generated

Then, I included the \_\_restrict\_\_ keyword for the input, mask, and output kernel pointers in the arguments of the kernel for more aggressive optimization.



* 1. Did the performance match your expectation? Show your analysis results using profiling tools.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Batch Size | Op Time 1 | Op Time 2 | Total Execution Time | Accuracy |
| 100 | *0.560971 ms* | *0.471073 ms* | *0m2.699s* | *0.86* |
| 1000 | *1.67883 ms* | *4.52436 ms* | *0m9.752s* | *0.886* |
| 10000 | *11.9963 ms* | *44.9884 ms* | *1m33.682s* | *0.8714* |

A screen shot of a black screen

Description automatically generated

While op times are about the same, the ALU usage significantly decreased (~55% to ~30%). This is because the branch instruction is reduced due to effectively tuning, alleviating ALU utilization and improving overall performance.

* 1. Does this optimization synergize with any other optimizations? How and why?

Restrict and loop unrolling can synergize effectively with other optimizations such as tiled convolution and tiled matrix multiplication using shared memory. The restrict keyword informs the compiler that pointers are not aliased, allowing for more aggressive optimization. Loop unrolling reduces loop overhead, enabling better utilization of shared memory and potentially improving memory access patterns. When combined with tiled convolution and matrix multiplication, which heavily rely on shared memory access patterns for data reuse and parallelism, the reduced overhead from loop unrolling and the compiler's ability to optimize memory accesses with restrict can lead to further performance gains. These optimizations work hand in hand to maximize memory bandwidth utilization and parallelism.

* 1. List your references used while implementing this technique. (you must mention textbook pages at the minimum)
* <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#restrict>
* [https://www.nvidia.com/docs/IO/116711/sc11-unrolling-parallel-loops.pdf](https://www.nvidia.com/docs/IO/116711/sc11-unrolling-parallel-loops.pdf\)

1. **Optimization Number 6: \_\_Sweeping\_\_**
   1. How does this optimization work in theory? Expected behavior?

Sweeping various parameters allows me to find the best values to maximize performance. By analyzing statistics, I can also identify trends to better future programming.

* 1. How did you implement your code? Explain thoroughly and show code snippets.

I changed the #define statements for TILE\_WIDTH and BLOCK\_SIZE, and generated nsys profile. I used 5 different TILE\_WIDTH values with a BLOCK\_SIZE of 32 and 5 different BLOCK\_SIZE values with a TILE\_WIDTH of 16.

* 1. Did the performance match your expectation? Show your analysis results using profiling tools.

|  |  |  |  |
| --- | --- | --- | --- |
| TILE\_WIDTH | Op Time 1 | Op Time 2 | Accuracy |
| 4 | *325.049 ms* | *381.255 ms* | *0.8714* |
| 8 | *289.324 ms* | *324.772 ms* | *0.8714* |
| 16 | *307.679 ms* | *308.467 ms* | *0.8714* |
| 32 | *383.315 ms* | *385.266 ms* | *0.8714* |
| 64 | *161.914 ms* | *159.385 ms* | *0.1* |

|  |  |  |  |
| --- | --- | --- | --- |
| BLOCK\_SIZE | Op Time 1 | Op Time 2 | Accuracy |
| 8 | *310.438 ms* | *307.988 ms* | *0.8714* |
| 16 | *311.189 ms* | *307.574 ms* | *0.8714* |
| 32 | *307.693 ms* | *308.778 ms* | *0.8714* |
| 64 | *310.625 ms* | *306.5 ms* | *0.8714* |
| 128 | *307.404 ms* | *307.99 ms* | *0.8714* |

The TILE\_WIDTH results matched my expectations. Best performance sat around 8 – 16, whereas the network broke at a width of 64. This makes sense because it is larger than the fixed BLOCK\_SIZE of 32. On the contrary, I was surprised to see changes in BLOCK\_SIZE not affecting the op times. This may be because the GPU resources are not being fully utilized and the bottleneck exists elsewhere, like memory.

* 1. Does this optimization synergize with any other optimizations? How and why?

This optimization synergizes with input unrolling and matrix multiplication by identifying the most optimal TILE\_WIDTH and BLOCK\_SIZE that maximize performance.

* 1. List your references used while implementing this technique. (you must mention textbook pages at the minimum)

N/A

1. **Optimization Number 10: \_\_FP16\_\_**
   1. How does this optimization work in theory? Expected behavior?

The optimization of FP16 CUDA Fixed Arithmetic utilizing the \_\_half data type aims to leverage the benefits of reduced precision while performing calculations on CUDA-enabled GPUs. By using \_\_half, which represents a 16-bit floating-point value, instead of the standard 32-bit float data type, the memory bandwidth and computation requirements are halved, potentially resulting in significant performance improvements. However, this reduction in precision may lead to some loss of accuracy, especially in scenarios where high precision is required, but it often proves acceptable in applications where the emphasis is on computational efficiency rather than exact numerical precision.

* 1. How did you implement your code? Explain thoroughly and show code snippets.

I began by including the CUDA header file that enabled FP16.



I changed the data type of the accumulator variable from float to half, also making sure to convert the value of which it was being assigned.



After multiplying the input and mask, I converted the product from a float to a half and preformed a half add of the converted product with acc, reassigning the sum to acc.



* 1. Did the performance match your expectation? Show your analysis results using profiling tools.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Batch Size | Op Time 1 | Op Time 2 | Total Execution Time | Accuracy |
| 100 | 0.210354 ms | *0.473376 ms* | *0m1.216s* | *0.86* |
| 1000 | *1.19923 ms* | *4.52899 ms* | *0m9.295s* | *0.887* |
| 10000 | *11.3918 ms* | *45.0217 ms* | *1m30.437s* | *0.8714* |

A screenshot of a computer

Description automatically generated

While op times did not vary much from baseline, pipe utilization shifted from the ALU to FMA (FP16), redistributing hardware use to potentially improve performance. This matched my expectation.

* 1. Does this optimization synergize with any other optimizations? How and why?

FP16 arithmetic implementation synergizes well with several other optimizations. Firstly, when combined with tiled (shared memory) convolution and input matrix unrolling, FP16 arithmetic can significantly reduce the memory bandwidth requirements, as FP16 data types consume half the memory compared to FP32. This reduction in memory traffic allows for more efficient use of shared memory, enhancing the performance of tiled convolution and matrix multiplication. Additionally, when paired with input channel reduction using atomics, FP16 arithmetic can further alleviate memory bandwidth constraints by reducing the amount of data processed per channel, enhancing overall parallelism and throughput.

* 1. List your references used while implementing this technique. (you must mention textbook pages at the minimum)
* <https://docs.nvidia.com/cuda/cuda-math-api/group__CUDA__MATH____HALF__MISC.html>

1. **Optimization Number 9: \_\_Atomics\_\_**
   1. How does this optimization work in theory? Expected behavior?

The optimization of input channel reduction using atomics works by allowing multiple threads to independently compute partial sums for each output element while aggregating them atomically into the output array. Each thread computes the convolution result for a specific output element, utilizing atomics to ensure that concurrent writes to the same output element do not cause data races. This approach effectively parallelizes the computation across threads within a block while ensuring correctness by synchronizing updates to shared memory locations. The expected behavior is that each thread contributes its computed value to the output element atomically, avoiding conflicts and producing the correct result for each element of the output array.

* 1. How did you implement your code? Explain thoroughly and show code snippets.

I began implementing by code by changing the z-axis of the grid to the number of channels.



Instead of assigning the blockIdx.z to b, I assigned it to c and changed the for loop to loop through the batch.

A screenshot of a computer code

Description automatically generated

Finally, I changed the writeback to the output to an atomic add.



* 1. Did the performance match your expectation? Show your analysis results using profiling tools.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Batch Size | Op Time 1 | Op Time 2 | Total Execution Time | Accuracy |
| 100 | *0.450243 ms* | *0.656298 ms* | *0m2.699s* | *0.86* |
| 1000 | *3.63788 ms* | *7.42878 ms* | *0m9.752s* | *0.886* |
| 10000 | *35.7735 ms* | *77.8427 ms* | *1m33.682s* | *0.8714* |

This performance matched my expectation since atomic add operations increase operation times with synchronization overhead. When multiple threads contend for access to the same memory location, atomic operations ensure data integrity by allowing only one thread to modify the data at a time, which can lead to serialization and increased latency, especially in highly parallel environments like GPUs.

* 1. Does this optimization synergize with any other optimizations? How and why?

Input channel reduction using atomics can also synergize effectively with FP16 arithmetic implementation. When using atomics to reduce input channels, you're essentially reducing the amount of data processed during each computation step. This reduction in data size aligns well with the benefits offered by FP16 arithmetic, which operates on half-precision floating-point numbers. FP16 arithmetic inherently requires less memory bandwidth and storage compared to FP32 arithmetic. By combining input channel reduction with FP16 arithmetic, you further reduce memory bandwidth requirements, allowing for more efficient use of available resources, especially in memory-bound operations like convolutional neural networks.

* 1. List your references used while implementing this technique. (you must mention textbook pages at the minimum)
* Textbook Chapter 9
* <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html?highlight=atomic%20add#atomicadd>