**Milestone 3 Report**

[https://drive.google.com/drive/folders/1hsnHKc5wXTTQZxJgnMXj5QMI25pihC1p]

1. **Baseline:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Batch Size | Op Time 1 | Op Time 2 | Total Execution Time | Accuracy |
| 100 | *1.4376 ms* | *1.30287 ms* | *0m1.438s* | *0.86* |
| 1000 | *9.03263 ms* | *8.5722 ms* | *0m10.556s* | *0.886* |
| 10000 | *66.5099 ms* | *67.4866 ms* | *1m41.466s* | *0.8714* |

1. **Req\_1: Using Tensor Cores to speed up matrix multiplication (required)**

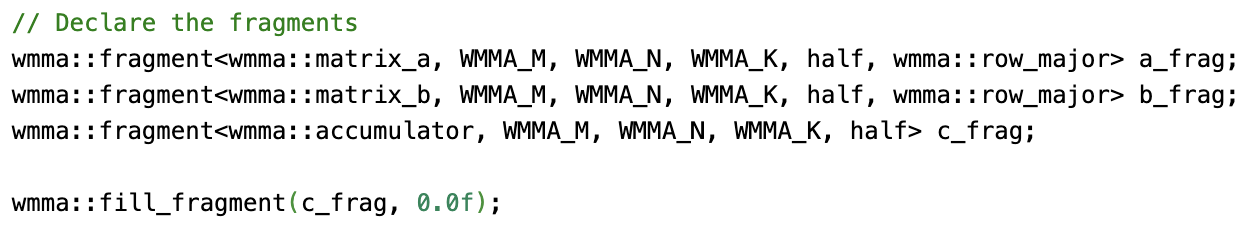
[https://drive.google.com/drive/folders/1qkQT22paNTdRCoMb6X13jo8Ulu8xEcxQ]

* 1. How does this optimization theoretically optimize your convolution kernel? Expected behavior?

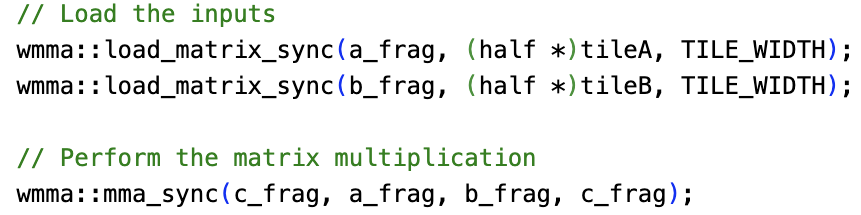
Using Tensor Cores to speed up matrix multiplication optimizes the convolution kernel by leveraging the hardware acceleration capabilities specifically designed for dense matrix operations. Tensor Cores in NVIDIA GPUs are specialized units that perform mixed-precision matrix-matrix multiplications at extremely high throughput. Using tensor cores will speed up tiled matrix multiplication in my implementation. Expected behavior includes faster execution time and faster op time since it measures the time spent working (performing matrix computations) on a convolution layer.

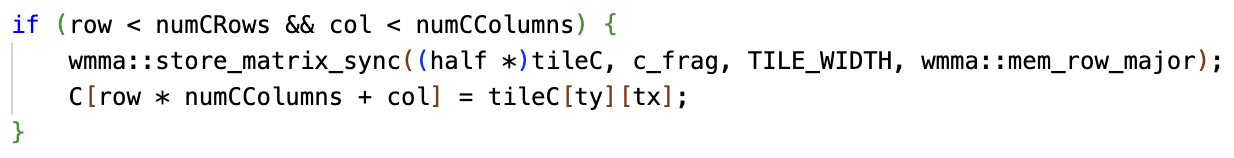
* 1. How did you implement your code? Explain thoroughly and show code snippets. Justify the correctness of your implementation with proper profiling results.

The implementation leverages CUDA's wmma (Warp Matrix Multiply and Accumulate) API, specifically optimized for Tensor Core operations with FP16 matrix multiplication. The input matrices are divided into 16x16x16 tiles to meet Tensor Core requirements.



Input matrix tiles are loaded into registers using wmma::load\_matrix\_sync to minimize global memory access latency. The actual multiplication and accumulation are executed on Tensor Cores using wmma::mma\_sync.



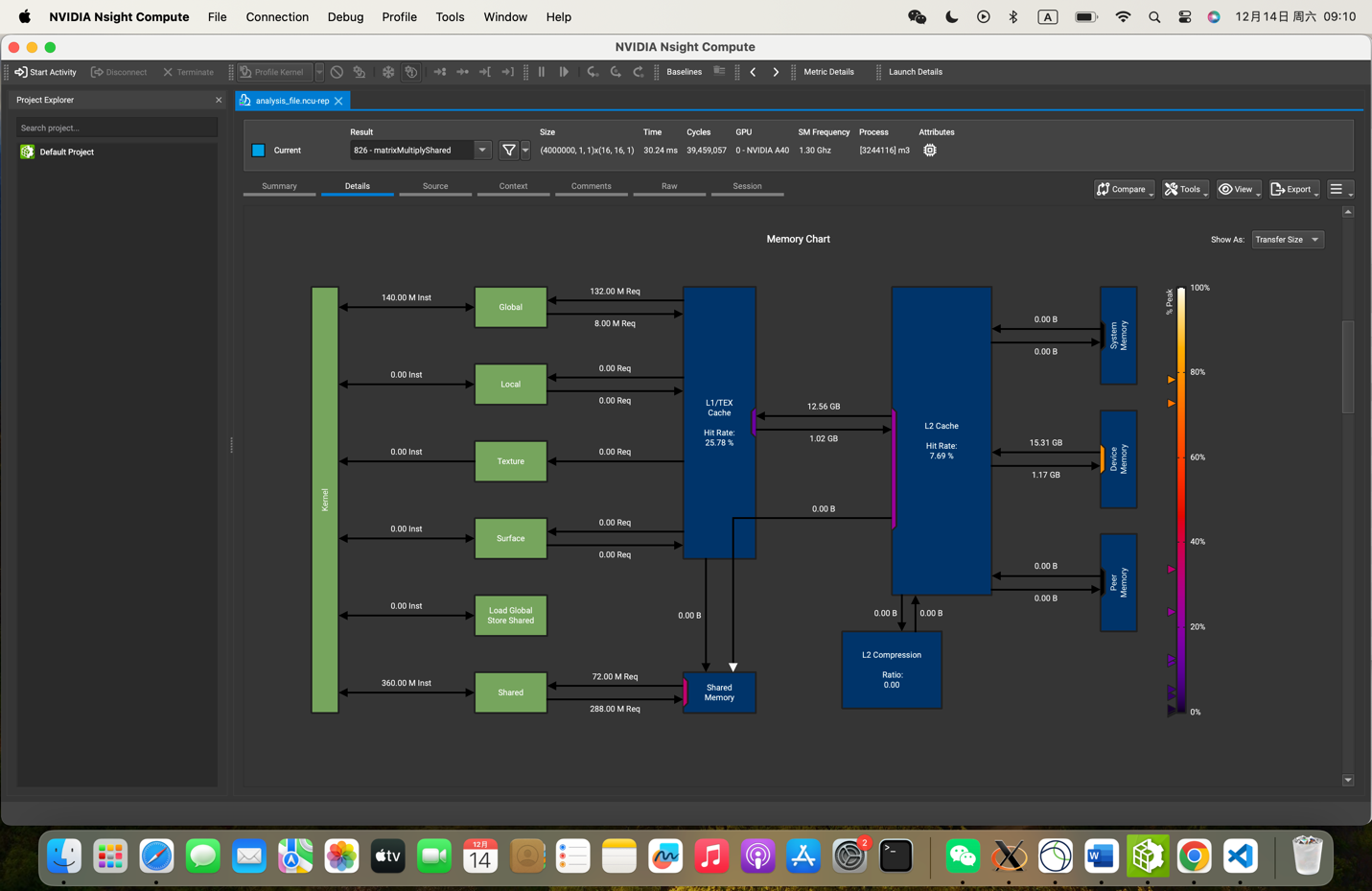
The results are written back to global memory in row-major format using wmma::store\_matrix\_sync. 

The implementation is correct because the accuracy of three batches aligns well with the baseline, and the matrix multiplication kernel has a reasonable memory usage. (See the image and table in part c.)

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

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Batch Size | Op Time 1 | Op Time 2 | Total Execution Time | Accuracy |
| 100 | *1.37048 ms* | *1.20863 ms* | *0m1.382s* | *0.86* |
| 1000 | *8.9006 ms* | *7.75943 ms* | *0m9.724s* | *0.886* |
| 10000 | *65.4857 ms* | *59.7481* | *1m33.444s* | *0.8712* |

Yes, it did. Because both Op Time and Total Execution Time are improved compared to the baseline, and the accuracy accuracy is well within acceptable limits for most applications, balancing performance and precision effectively. The profiling results confirm that the implementation effectively leverages Tensor Cores for matrix multiplication. The scaling behavior and performance gains match expectations, although further optimizations in memory movement and thread utilization could enhance performance, especially for smaller batch sizes.



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

Yes, it does. Using tensor cores and fused kernel (fusing unrolling, matrix multiplication, and permutation kernels) can once again improved the Op Time, as shown below.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Batch Size | Op Time 1 | Op Time 2 | Total Execution Time | Accuracy |
| 10000 | *50.6113 ms* | *28.4013 ms* | *1m31.503s* | *0.8714* |

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

Textbook Chapter 13.

https://developer.nvidia.com/blog/programming-tensor-cores-cuda-9

1. **Req\_2: Kernel fusion for unrolling and matrix-multiplication (required)**

[https://drive.google.com/drive/folders/1laWSRzlYa35vjCjypg5qroF-0-Jfsng\_]

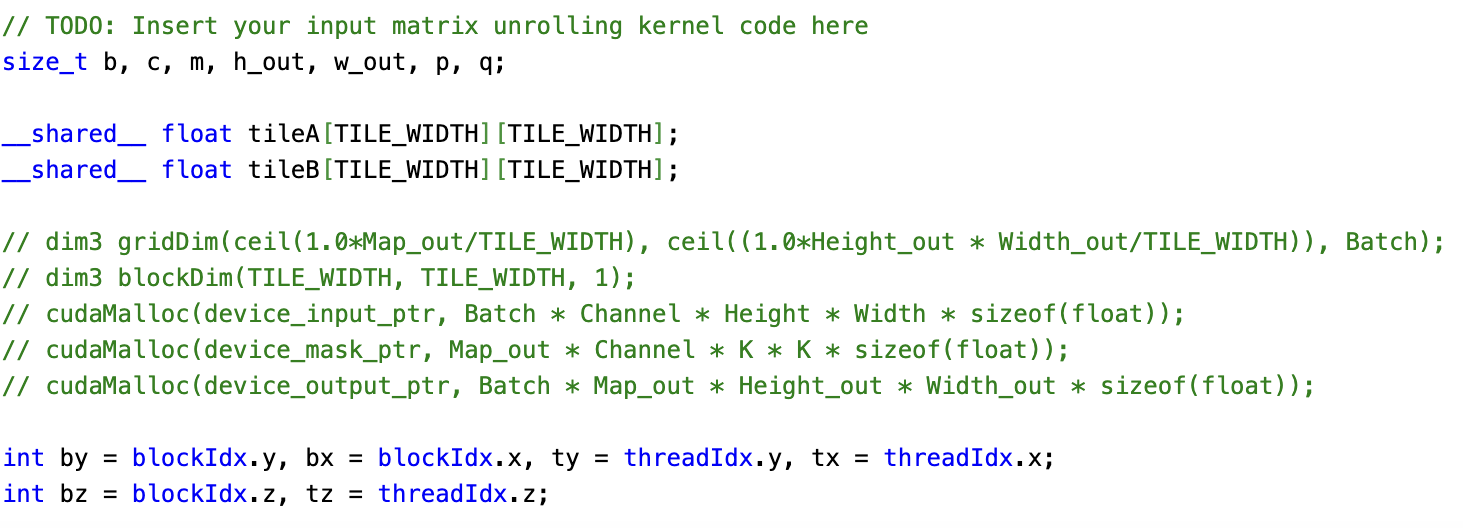
* 1. How does this optimization theoretically optimize your convolution kernel? Expected behavior?

Without kernel fusion, each step would require separate kernel launches, incurring significant overhead due to repeated kernel initialization and synchronization. Kernel fusion combines these steps into a single kernel, eliminating the overhead of multiple launches. A reduction in kernel launch time and synchronization delays, leading to faster execution, especially where launch overhead is significant relative to computation time.

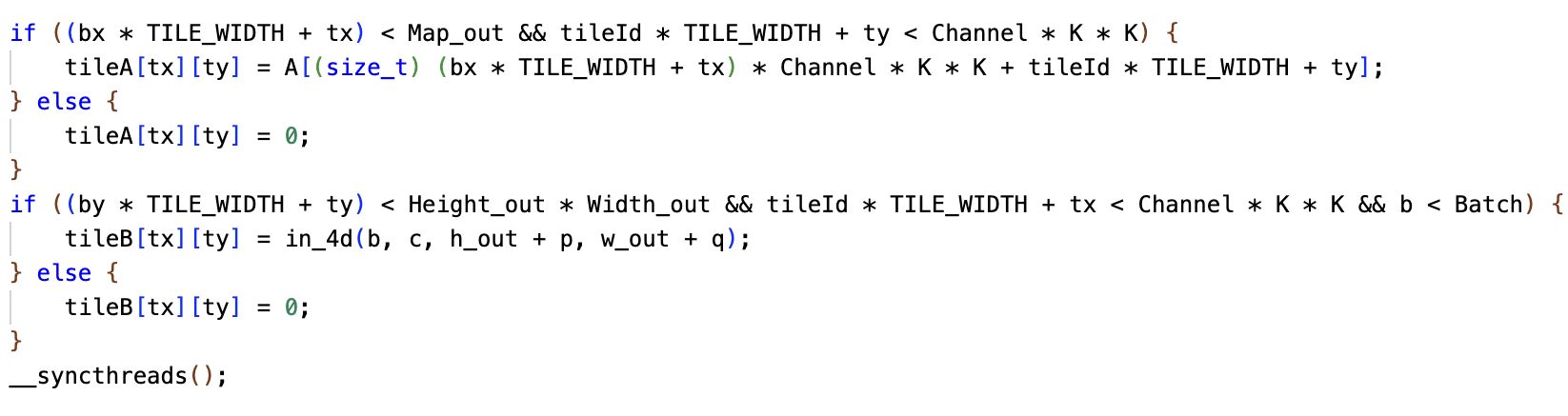
When unrolling and matrix multiplication are fused, intermediate data remains in faster shared memory or registers, avoiding costly global memory reads and writes. Reduced global memory bandwidth usage and improved memory efficiency. This leads to faster access times and increased throughput for memory-bound workloads.

* 1. How did you implement your code? Explain thoroughly and show code snippets. Justify the correctness of your implementation with proper profiling results.

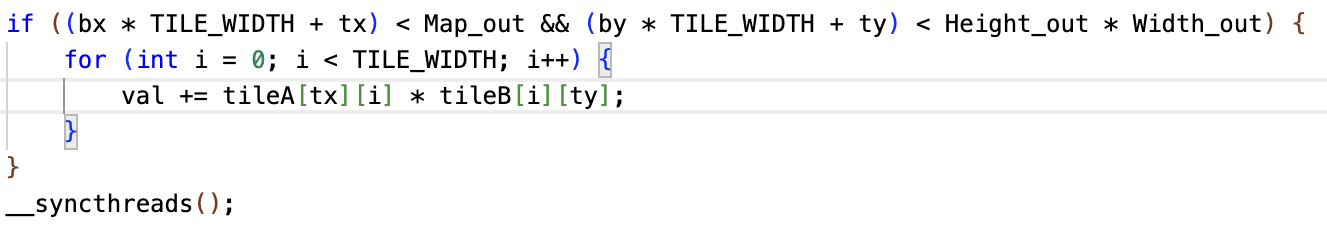
The kernel uses a 3D grid where each block processes a portion of the output matrix. Each thread computes a small part of the final result. Two shared memory arrays (tileA and tileB) are used to hold portions of matrices A and B that each block will process.



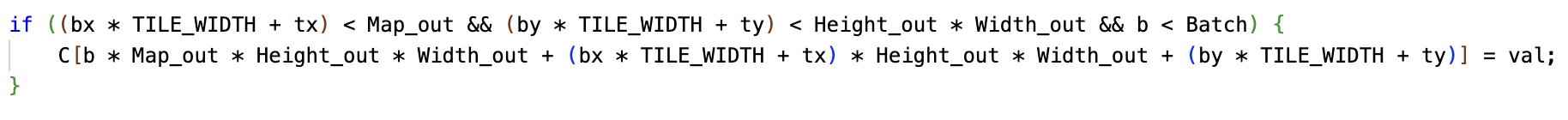
The loop iterates over tiles of the matrix. The tileId variable helps in unrolling the A matrix for computation. For every tile of the filter, corresponding values are loaded into the shared memory.



After loading the tiles into shared memory, the threads compute the convolution by multiplying corresponding elements of tileA and tileB, then summing the products into the val variable.



Finally, after all iterations, the computed value is stored in the output matrix C, which holds the result of the convolution.

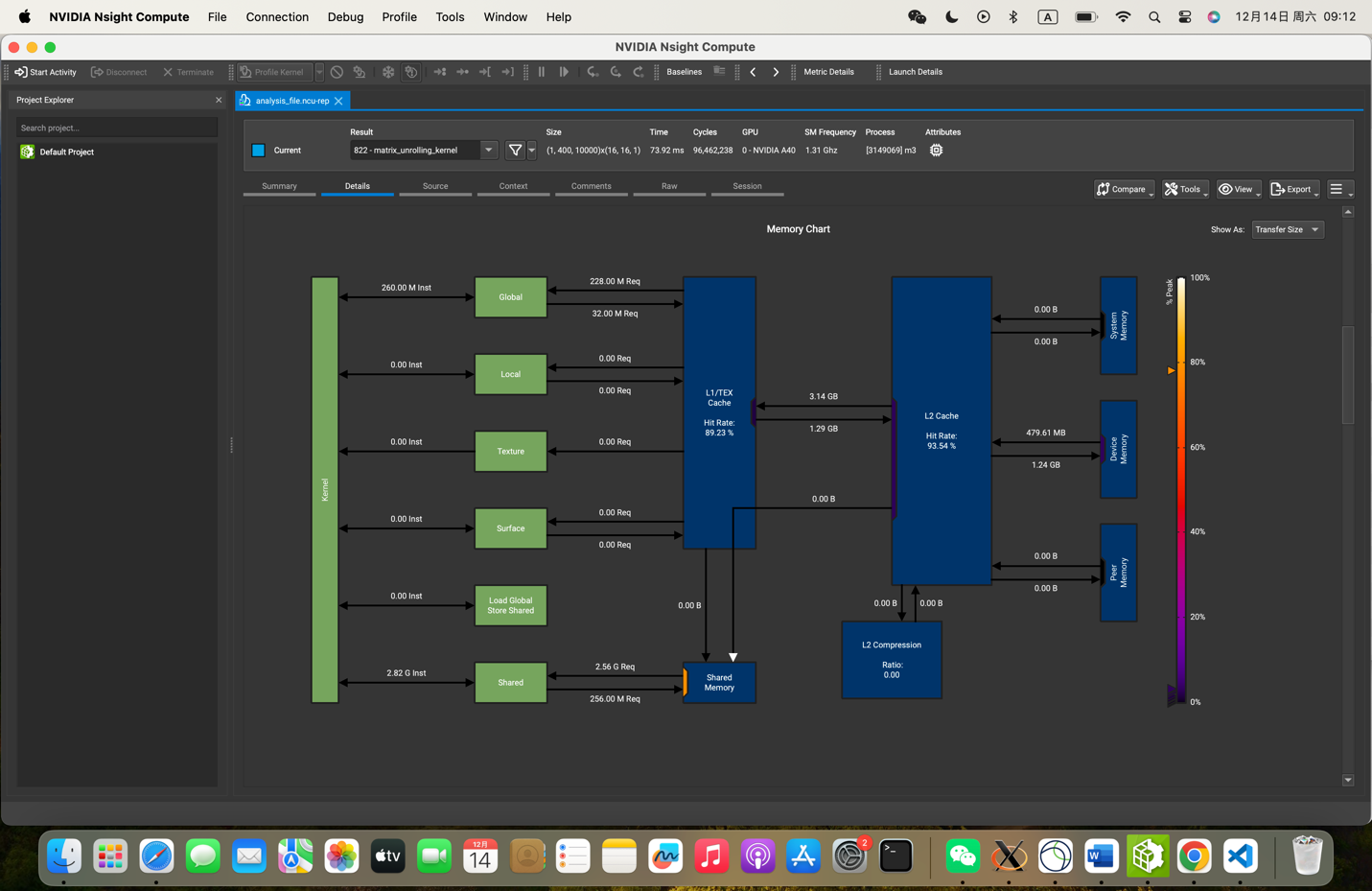


The implementation is correct because the accuracy of three batches aligns well with the baseline, and the matrix multiplication kernel has a reasonable memory usage. (See the image and table in part c.)

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

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Batch Size | Op Time 1 | Op Time 2 | Total Execution Time | Accuracy |
| 100 | *1.32939 ms* | *1.26382 ms* | *0m1.574s* | *0.86* |
| 1000 | *8.73796 ms* | *8.21206 ms* | *0m10.621s* | *0.886* |
| 10000 | *64.4946 ms* | *62.2252 ms* | *1m31.010s* | *0.8714* |

The performance matches expectations, as larger batch sizes naturally result in higher execution times due to more data to process. The total execution time scales predictably with batch size. The accuracy is stable across batch sizes, with only slight fluctuations. The drop in accuracy at larger batch sizes (from 0.886 to 0.8714) is minor and does not indicate a significant issue with the kernel. The memory usage is reasonable given the matrix sizes and the use of shared memory for tile storage. The kernel efficiently uses the shared memory for each block, minimizing global memory access and reducing overhead since the hit rate of L2 cache is higher than the baseline as well as many other optimizations.



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

Yes, it does. Using tensor cores and fused kernel (fusing unrolling, matrix multiplication, and permutation kernels) can once again improved the Op Time, as shown below.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Batch Size | Op Time 1 | Op Time 2 | Total Execution Time | Accuracy |
| 10000 | *50.6113 ms* | *28.4013 ms* | *1m31.503s* | *0.8714* |

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

The codes in milestone 1 and 2 since no textbook chapters/content available for kernel fusion.

1. **Op\_0: Weight matrix (Kernel) in constant memory**

[https://drive.google.com/drive/folders/1vpeBSZJODn1h7Mv54pwzAiBYqShZW7qn]

* 1. How does this optimization theoretically optimize your convolution kernel? Expected behavior?

In CUDA, constant memory is a read-only memory space on the device that is cached and optimized for values that do not change during kernel execution. The constant memory is cached, so when one thread accesses a particular value, that value is cached and made available to other threads in the warp, reducing global memory traffic.

With the kernel stored in constant memory, the convolution operation is expected to execute faster, as it minimizes global memory accesses for the kernel values, reducing memory bandwidth bottlenecks. This optimization will scale better as the batch size increases or as the kernel size grows, since the constant memory access time remains constant, irrespective of the size of the input data or the batch.

* 1. How did you implement your code? Explain thoroughly and show code snippets. Justify the correctness of your implementation with proper profiling results.

The kernel (deviceMask) is stored in constant memory using the \_\_constant\_\_ qualifier:

*\_\_constant\_\_ float deviceMask[6556];*

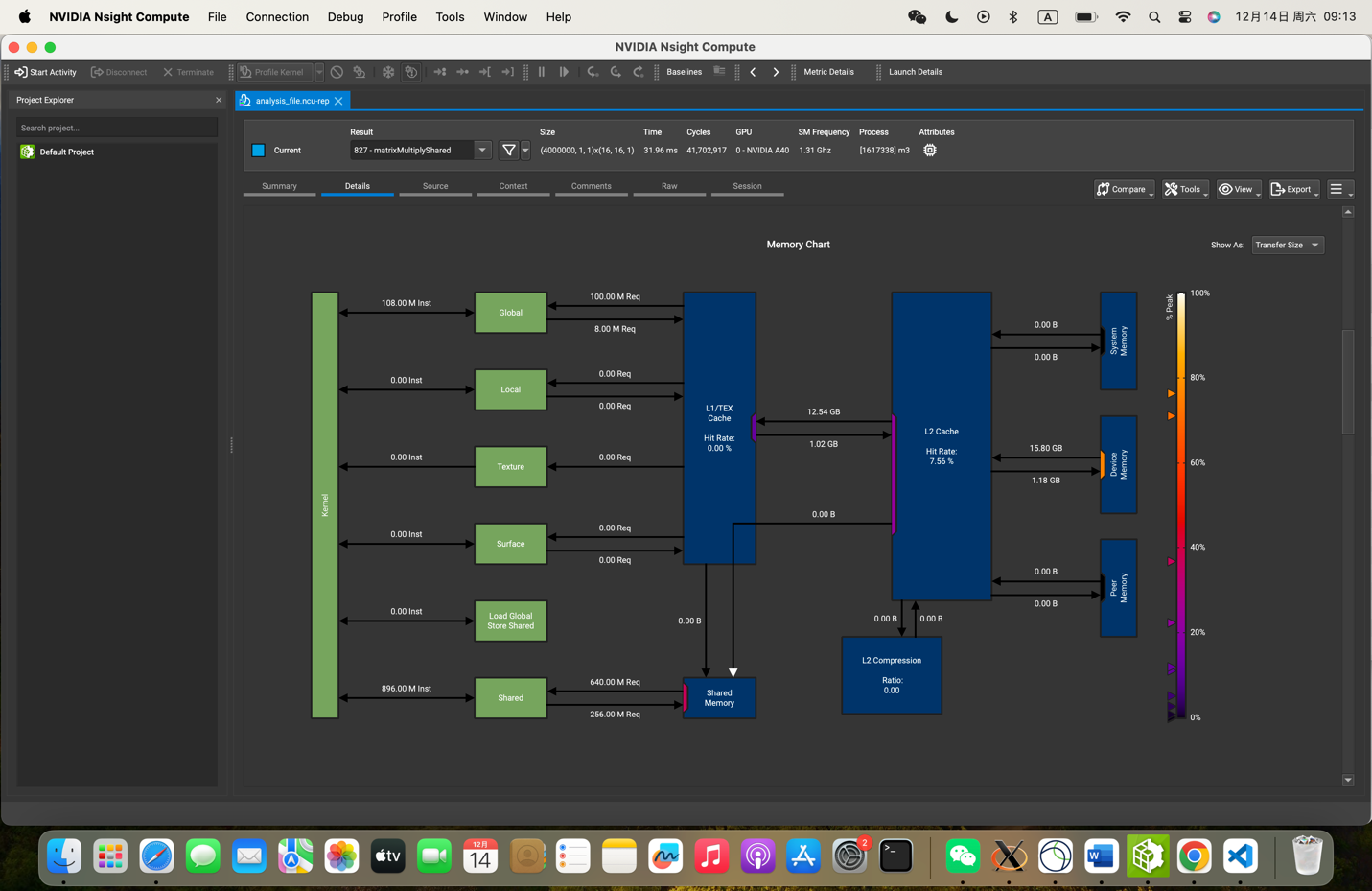
Copy the data from the host array host\_mask into the device's constant memory variable deviceMask:

*cudaMemcpyToSymbol(deviceMask, host\_mask, Map\_out \* Channel \* K \* K \* sizeof(float));*

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

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Batch Size | Op Time 1 | Op Time 2 | Total Execution Time | Accuracy |
| 100 | *1.29764 ms* | *1.35984 ms* | *0m1.424s* | *0.86* |
| 1000 | *8.90353 ms* | *10.445 ms* | *0m9.485s* | *0.886* |
| 10000 | *65.8705 ms* | *86.3546 ms* | *1m31.162s* | *0.8714* |

Not entirely. Op Time 1 and Total Execution Time have some improvements across batch sizes. However, the most significant change occurs in Op Time 2, which increases substantially as the batch size grows. This suggests that the second operation in the pipeline (likely the second convolution layer) becomes a bottleneck with larger batches. Even though constant memory is designed for read-only access by multiple threads, it doesn't inherently guarantee coalesced access. If the threads in the second convolution layer do not access deviceMask in a coalesced manner, it could result in inefficient memory access, causing the second operation time to increase.



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

No, since it slows down other optimizations’ speed up.

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

ECE 408 Fall 2024 Lab 4 Manual.

Textbook chapter 4&5.

1. **Op\_1: \_\_restrict\_\_ keyword**

[https://drive.google.com/drive/folders/195JnJQbElrsDhk0OjuXGlHFe1rGhLFh0]

* 1. How does this optimization theoretically optimize your convolution kernel? Expected behavior?

The \_\_restrict\_\_ keyword is a pointer qualifier in CUDA programming that tells the compiler that the pointer to a memory region is not aliased, meaning that no other pointer will point to the same memory region. The compiler can safely assume that the data being accessed by the \_\_restrict\_\_ pointers are not shared between different variables, allowing for more aggressive caching. This reduces memory traffic and improves cache utilization, especially in operations like convolution where data is repeatedly accessed in a pattern.

The expected behavior is improved performance in terms of reduced execution time, particularly in memory-bound operations like matrix multiplications and convolutions, as long as the code adheres to the no-aliasing rule.

* 1. How did you implement your code? Explain thoroughly and show code snippets. Justify the correctness of your implementation with proper profiling results.

Code:

*\_\_global\_\_ void matrixMultiplyShared(const float \*\_\_restrict\_\_ A, const float \*\_\_restrict\_\_ B, float \*\_\_restrict\_\_ C,*

*int numARows, int numAColumns,*

*int numBRows, int numBColumns,*

*int numCRows, int numCColumns)*

The \_\_restrict\_\_ keyword is applied to the pointers A, B, and C. This ensures the compiler knows that these pointers do not alias with each other, allowing it to optimize the code more aggressively.

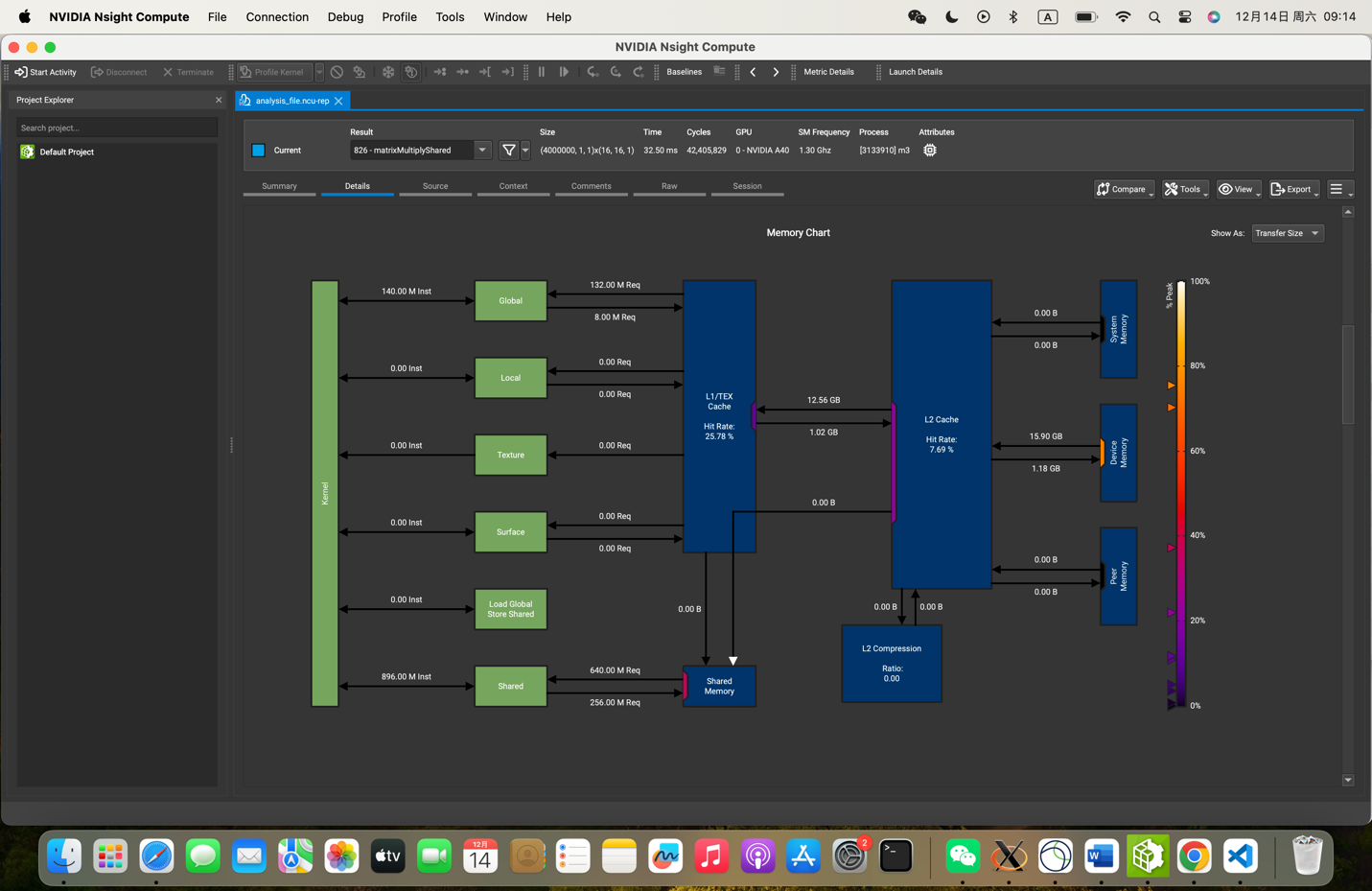
* 1. Did the performance match your expectation? Analyze the profiling results as a scientist.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Batch Size | Op Time 1 | Op Time 2 | Total Execution Time | Accuracy |
| 100 | *1.46247 ms* | *1.19085 ms* | *0m1.458s* | *0.86* |
| 1000 | *8.99299 ms* | *8.56971 ms* | *0m9.999s* | *0.886* |
| 10000 | *66.6917 ms* | *67.7639 ms* | *1m34.353s* | *0.8714* |

The performance improvement from the \_\_restrict\_\_ keyword did meet expectations to some extent, but there are nuanced aspects to consider in the profiling results:

A small performance gain is observed in Op Time 1, but Op Time **2** slightly increases, likely due to other bottlenecks in the kernel. For the largest batch size, the total execution time decreases slightly, but Op Time 2 is still significantly higher. This indicates that while \_\_restrict\_\_ helped in improving the speed of memory accesses, the operation still becomes memory-bound due to the large size of the input.

The \_\_restrict\_\_ keyword helps the compiler generate more efficient code for memory access, which works hand-in-hand with the proper alignment of memory accesses since the hit rate of L2 cache is higher than the baseline.



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

Yes, the \_\_restrict\_\_ optimization synergizes well with several other optimizations, including using tensor cores and fused kernel.

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

https://github.com/illinois-cs-coursework/fa24\_ece408\_wendiw2/blob/main/Project/README\_m3.md

1. **Op\_2: Loop unrolling** [https://drive.google.com/drive/folders/1VbUdgu39Bhm9OTFSSh5Hb0ZmltnhG6zt]
   1. How does this optimization theoretically optimize your convolution kernel? Expected behavior?

Loop unrolling is a compiler optimization technique where the loop's iterations are expanded into multiple copies of the loop body. This reduces the overhead of loop control (e.g., incrementing the loop variable, checking loop conditions) and can enhance performance by enabling better instruction-level parallelism and better use of the processor's pipeline.

The expected behavior is that the convolution kernel will become faster due to less overhead and better parallelism. Specifically, the kernel should run with reduced execution time for each operation (Op Time 1 and Op Time 2), leading to faster overall execution.

* 1. How did you implement your code? Explain thoroughly and show code snippets. Justify the correctness of your implementation with proper profiling results.

Code: if (row < numCRows && col < numCColumns) {

for (int i = 0; i < TILE\_WIDTH; i += 16) {

val += tileA[ty][i] \* tileB[i][tx];

val += tileA[ty][i+1] \* tileB[i+1][tx];

val += tileA[ty][i+2] \* tileB[i+2][tx];

val += tileA[ty][i+3] \* tileB[i+3][tx];

val += tileA[ty][i+4] \* tileB[i+4][tx];

val += tileA[ty][i+5] \* tileB[i+5][tx];

val += tileA[ty][i+6] \* tileB[i+6][tx];

val += tileA[ty][i+7] \* tileB[i+7][tx];

val += tileA[ty][i+8] \* tileB[i+8][tx];

val += tileA[ty][i+9] \* tileB[i+9][tx];

val += tileA[ty][i+10] \* tileB[i+10][tx];

val += tileA[ty][i+11] \* tileB[i+11][tx];

val += tileA[ty][i+12] \* tileB[i+12][tx];

val += tileA[ty][i+13] \* tileB[i+13][tx];

val += tileA[ty][i+14] \* tileB[i+14][tx];

val += tileA[ty][i+15] \* tileB[i+15][tx];

}

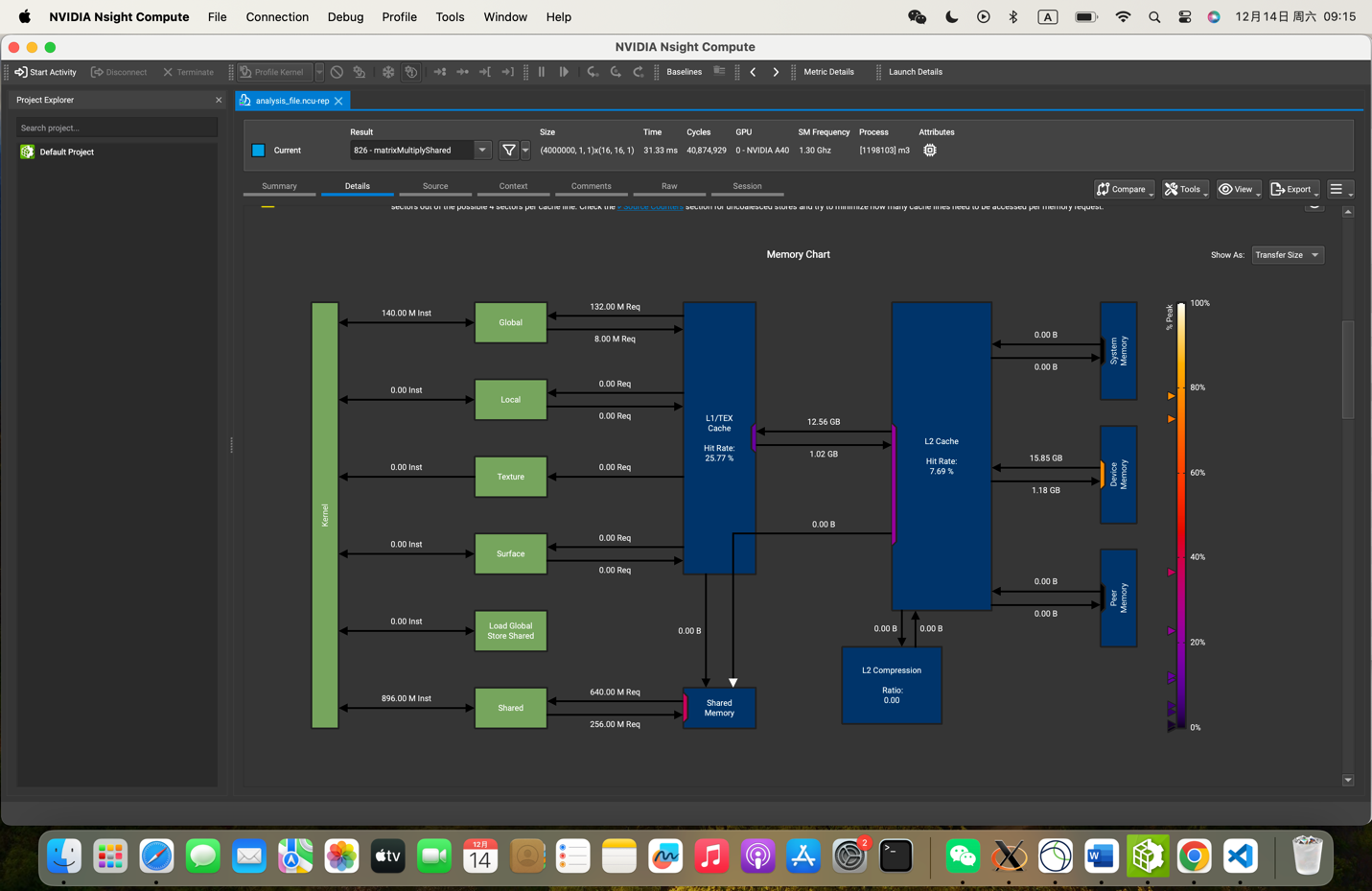
}

The implementation is correct because the accuracy of three batches aligns well with the baseline, and the matrix multiplication kernel has a reasonable memory usage. (See the image and table in part c.)

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

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Batch Size | Op Time 1 | Op Time 2 | Total Execution Time | Accuracy |
| 100 | *1.3509 ms* | *1.153 ms* | *0m1.377s* | *0.86* |
| 1000 | *8.91886 ms* | *8.32214 ms* | *0m9.826s* | *0.886* |
| 10000 | *66.0162 ms* | *65.1761 ms* | *1m34.308s* | *0.8714* |

The performance matches expectations in that loop unrolling does reduce the operation times slightly, particularly for Op Time 1, which indicates that the overhead from the loop control has been effectively minimized. However, the increase in Op Time 2 for larger batch sizes suggests that the kernel is still bottlenecked by other factors, possibly related to memory access or shared memory utilization. The unrolling helps the compiler generate more efficient code for memory access, which works hand-in-hand with the proper alignment of memory accesses since the hit rate of L2 cache is higher than the baseline.



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

Yes, loop unrolling works well with several other optimizations, like fused kernel.

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

https://github.com/illinois-cs-coursework/fa24\_ece408\_wendiw2/blob/main/Project/README\_m3.md

1. **Op\_5: Fixed point (FP16) arithmetic implementation (this can modify model accuracy slightly)**

[https://drive.google.com/drive/folders/1eS2-UUecboc6pfa8Fr5Fo0igxCpDb2td]

* 1. How does this optimization theoretically optimize your convolution kernel? Expected behavior?

Fixed-point (FP16) arithmetic optimization utilizes half-precision floating point (FP16) numbers instead of the standard single-precision (FP32) floating point numbers. FP16 uses 16 bits for each number instead of 32 bits, which can significantly reduce memory usage and increase throughput, especially on GPUs that are optimized for low-precision arithmetic.

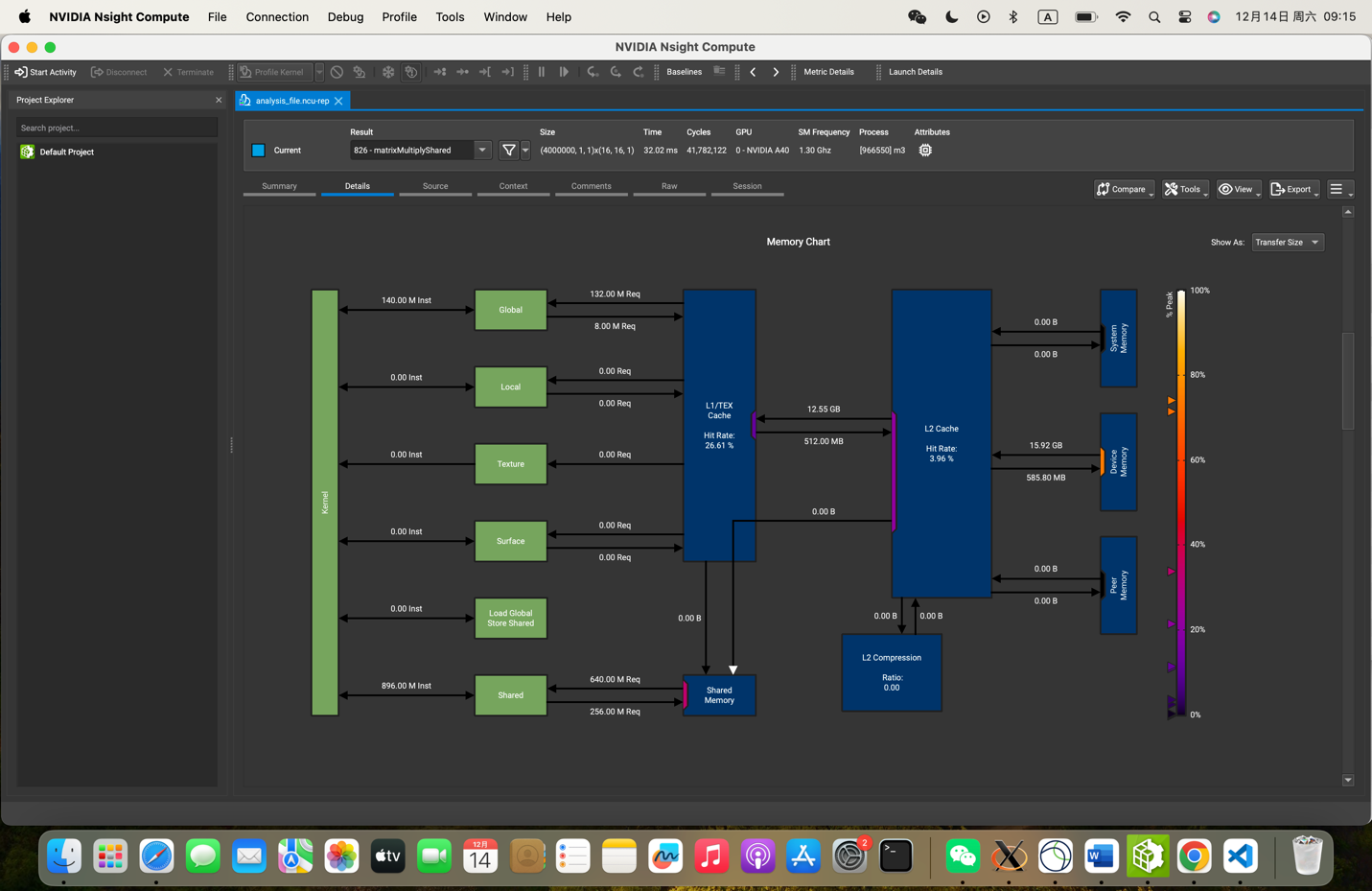
The expected behavior is that, by switching to FP16, the kernel should see a reduction in execution time, especially in terms of memory accesses and floating-point operations, leading to faster overall performance. However, there may be a slight drop in model accuracy due to the lower precision, which is typical when switching to lower-precision arithmetic.

* 1. How did you implement your code? Explain thoroughly and show code snippets. Justify the correctness of your implementation with proper profiling results.

1. \_\_global\_\_ void matrixMultiplyShared(const float \*A, const float \*B, \_\_half \*C,
2. int numARows, int numAColumns,
3. int numBRows, int numBColumns,
4. int numCRows, int numCColumns)
5. {
6. \_\_shared\_\_ \_\_half tileA[TILE\_WIDTH][TILE\_WIDTH];
7. \_\_shared\_\_ \_\_half tileB[TILE\_WIDTH][TILE\_WIDTH];
8. int by = blockIdx.y, bx = blockIdx.x, ty = threadIdx.y, tx = threadIdx.x;
9. int row = by \* TILE\_WIDTH + ty, col = bx \* TILE\_WIDTH + tx;
10. \_\_half val = 0;
    1. Did the performance match your expectation? Analyze the profiling results as a scientist.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Batch Size | Op Time 1 | Op Time 2 | Total Execution Time | Accuracy |
| 100 | *2.16008 ms* | *2.13327 ms* | *0m1.595s* | *0.86* |
| 1000 | *8.82454 ms* | *8.67405 ms* | *0m10.053s* | *0.887* |
| 10000 | *64.7495 ms* | *66.3851 ms* | *1m36.189s* | *0.8716* |

Yes, since Op Time, Execution Time and Accucary all look good.



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

Yes, like fused kernel and tensor cores.

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

https://developer.nvidia.com/blog/mixed-precision-programming-cuda-8/