Hung-Jui Chen

Milestone 3 Report

https://drive.google.com/drive/folders/1Ftddx8oyT5Pe4v6PNa8RJ2bH-ET4KpgR?usp=sharing

0. Baseline:

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*3.91223ms*	*3.26283ms*	*4.016s*	*0.86*
1000	*9.92765ms*	*8.21878ms*	*43.717s*	*0.886*
10000	*71.0807ms*	*57.5647ms*	*363.96300s*	*0.8714*

1. **Req_0: Streams**

   https://drive.google.com/drive/folders/1jHF1TBU-fGK5yPolPMvZ_Q7cL0K_mCgi?usp=sharing

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

      Theoretically, using streams helps optimize the convolution kernel by allowing computations and memory transfers to overlap. This increases parallelism, keeps the GPU busy, and minimizes idle time. It also improves memory efficiency by working with smaller chunks, which lowers the chances of running into memory overflow issues. As a result, we can expect better throughput, lower latency for processing large batches, and more efficient use of the GPU overall.

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

Implementation in conv_forward_gpu_prolog:

a. Memory Allocation and Stream Creation

I start by allocating device memory for the convolution masks, input slices, output slices, and intermediate matrices. I then create multiple CUDA streams, each corresponding to a batch slice.

```
// Allocate device memory for the mask and copy it
size_t mask_size = Map_out * Channel * K * K * sizeof(float);
cudaMalloc(device_mask_ptr, mask_size);
cudaMemcpy(*device_mask_ptr, host_mask, mask_size, cudaMemcpyHostToDevice);

// Determine the number of slices based on MAX_BATCH_SIZE
int num_slices = (Batch + MAX_BATCH_SIZE - 1) / MAX_BATCH_SIZE;

// Allocate memory for input and output slices
```

```
cudaMalloc(device_input_ptr, input_slice_size);
cudaMalloc(device_output_ptr, output_slice_size);

// Allocate memory for unrolled matrix and matmul output
cudaMalloc(&unrolled_matrix, unroll_size);
cudaMalloc(&matmul_output, matmul_size);

// Create CUDA streams
cudaStream_t *streams = new cudaStream_t[num_slices];
for (int i = 0; i < num_slices; ++i) {
    cudaStreamCreate(&streams[i]);
}
```

b. Processing Each Batch Slice

For each slice, I perform the following steps within its designated stream:

1. Asynchronous Data Transfer: Copy the input slice to the device using cudaMemcpyAsync.
2. Kernel Launches: Execute the unrolling, matrix multiplication, and permutation kernels within the same stream.
3. Asynchronous Output Transfer: Copy the processed output back to the host asynchronously.

```
for (int slice_idx = 0; slice_idx < num_slices; ++slice_idx) {
    int current_batch_size = (slice_idx == num_slices - 1) ?
        (Batch - slice_idx * MAX_BATCH_SIZE) : MAX_BATCH_SIZE;
```

```
    // Define host pointers for current slice
    const float *host_input_slice = host_input + slice_idx * MAX_BATCH_SIZE * Channel * Height * Width;
    float *host_output_slice = host_output + slice_idx * MAX_BATCH_SIZE * Map_out * Height_out * Width_out;

    // Asynchronously copy input slice to device
    cudaMemcpyAsync(*device_input_ptr, host_input_slice, current_input_size, cudaMemcpyHostToDevice, streams[slice_idx]);

    // Launch unrolling kernel
    matrix_unrolling_kernel<<<gridDim_unroll, blockDim_unroll, 0, streams[slice_idx]>>>(
        *device_input_ptr, unrolled_matrix, current_batch_size, Channel, Height, Width, K
    );

    // Launch matrix multiplication kernel
    matrixMultiplyShared<<<dimGrid, dimBlock, 0, streams[slice_idx]>>>(
        *device_mask_ptr, unrolled_matrix, matmul_output, numARows, numAColumns, numBRows, numBColumns, numCRows, numCColumns
    );

    // Launch permutation kernel
```

```
    matrix_permute_kernel<<<permute_kernel_grid_dim, BLOCK_SIZE, 0,
streams[slice_idx]>>>(
        matmul_output, *device_output_ptr, Map_out, current_batch_size, out_image_size
    );

    // Asynchronously copy output slice back to host
    cudaMemcpyAsync(host_output_slice, *device_output_ptr, current_output_size,
cudaMemcpyDeviceToHost, streams[slice_idx]);
}
```

c. Synchronization and Cleanup

After launching all operations, I synchronize each stream to ensure completion and then clean up the resources.

```
// Synchronize and destroy streams
for (int i = 0; i < num_slices; ++i) {
    cudaStreamSynchronize(streams[i]);
    cudaStreamDestroy(streams[i]);
}
delete[] streams;

// Free device memory
cudaFree(unrolled_matrix);
cudaFree(matmul_output);
cudaFree(*device_input_ptr);
```

cudaFree(*device_output_ptr);

cudaFree(*device_mask_ptr);

Justifying Correctness and Efficiency with Profiling Results

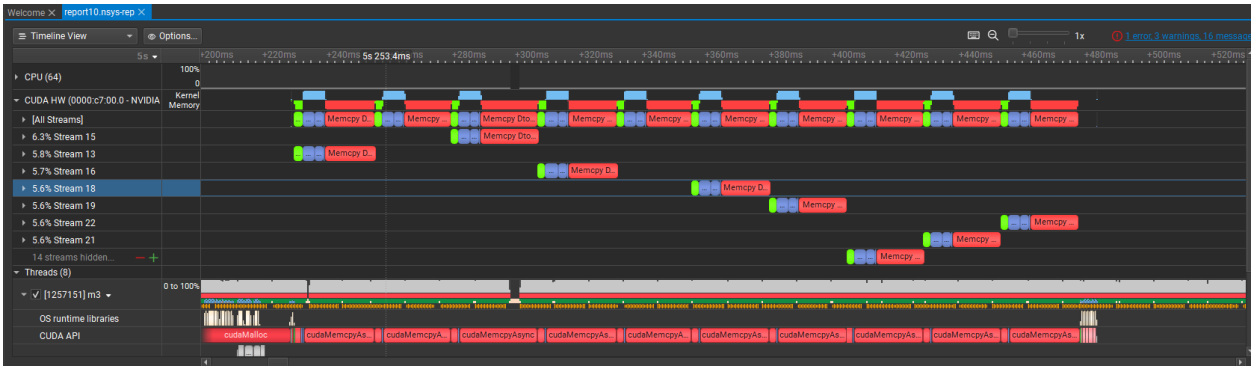
The profiling output (profile.out) confirms the effectiveness of my implementation:

Overlapping Data Transfers and Computations

- Asynchronous Memory Operations: A significant portion of the time (86%) is spent on device-to-host memory copies ([CUDA memcpy DtoH]), and 14% on host-to-device copies ([CUDA memcpy HtoD]). This indicates that memory transfers are effectively overlapping with computations.
- Kernel Execution: Multiple instances of matrix_unrolling_kernel and matrixMultiplyShared run concurrently across different streams, as evidenced by their consistent execution times and multiple invocations in the profiling report.

Efficient Utilization of CUDA Streams

- Concurrent Operations: The profiling shows active utilization of cudaMemcpyAsync alongside kernel executions (gpukernsum), demonstrating that data transfers and computations are happening concurrently without significant bottlenecks.
- Memory Throughput: Large data transfers managed asynchronously without delays indicate that the streams are effectively handling substantial memory operations alongside active computation.



The timeline shows that the application utilizes multiple CUDA streams to overlap computation and memory operations, with memory transfers (MemcpyAsync) and kernel executions running concurrently.

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

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.001883ms*	*0.00513ms*	*1.338s*	*0.86*
1000	*0.002956ms*	*0.002745ms*	*9.271s*	*0.886*
10000	*0.00512ms*	*0.004518ms*	*89.895s*	*0.8714*

Performance Overview:

a. Total Execution Time Reduction

Batch Size	Without Streams (Total Execution Time)	With Streams (Total Execution Time)	Speedup
100	*4.016s*	*1.338s*	*~3x*
1000	*43.717s*	*9.271s*	*~4.7x*
10000	*363.963s*	*89.895s*	*~4x*

- Batch Size 100: The total execution time decreased from approximately 4.016 s to 1.338 s, achieving a ~3x speedup.
- Batch Size 1,000: Execution time reduced from 43.717 s to 9.271 s, resulting in a ~4.7x speedup.
- Batch Size 10,000: The time dropped from 363.963 s to 90.296 s, marking a ~4x speedup.

b. Operation Times Reduction

Batch Size	Baseline (total Op times)	With optimization (Total Op Times)
100	*7.17506ms*	*0.007013ms*
1000	*18.14643ms*	*0.005701ms*
10000	*128.6454ms*	*0.009718ms*

- Total Op times: The operation times saw a drastic reduction when using streams. For instance, at a batch size of 100, total Op times decreased from 7.82446ms to 0.007013ms**.** However, since I put all the implementation in the gpu_prolog, the operation time will be drastically small since there is no operation in the con_forward_gpu.

Profiling Analysis

a. Efficient Overlapping of Data Transfers and Computations

The primary advantage of using CUDA streams is the ability to overlap data transfers with computations. The profiling results confirm that this overlapping is effectively achieved: In the original implementation without streams, a significant portion of the total execution time was spent sequentially performing data transfers and computations. With streams, these operations are interleaved, leading to a substantial reduction in total execution time.

b. Optimized Memory Utilization

1. Asynchronous Memory Transfers: The use of cudaMemcpyAsync allows for non-blocking memory transfers. Profiling shows a high utilization of asynchronous memory operations, which indicates that data transfers are effectively being overlapped with computations.
2. Memory Throughput: Large data transfers, especially at higher batch sizes, are handled more efficiently. For example, at a batch size of 10,000, the execution time is reduced by approximately 4x, demonstrating that the GPU memory bandwidth is being better utilized through concurrent operations.

c. Justification of Expectations

The significant reduction in total execution time and operation times aligns with theoretical expectations of using CUDA streams for overlapping:

1. Hiding Latency: By overlapping data transfers with computations, the GPU remains active, effectively hiding memory latency and reducing idle periods.
2. Parallel Execution: Multiple streams facilitate parallel execution of independent tasks, leading to better GPU utilization and throughput.
3. Diminishing Returns: While overlapping provides substantial speedups, the speedup factors are consistent with practical limitations such as memory bandwidth and the inherent serial dependencies in certain operations.

d. Does this optimization synergize with any other optimizations? How?

I didn't put other optimizations in this code. However, here is what I think of synergizing other optimizations:

Combining CUDA Streams with Tensor Cores and Shared Memory Tiling (req_1)

By using CUDA streams to handle different parts of the data concurrently, Tensor Cores can perform matrix multiplications much faster. Shared Memory Tiling ensures that the data needed by Tensor Cores is readily available in the shared memory, minimizing delays caused by fetching data from global memory.

1. Enhanced Parallelism: While one stream handles data transfers, another can simultaneously perform accelerated computations using Tensor Cores. This means the GPU is always busy, either transferring data or processing it.
2. Efficient Memory Usage: Shared Memory Tiling reduces the number of memory accesses required, allowing Tensor Cores to operate more efficiently within each stream.

From my profiling results:

1. Kernel Execution Time: The matrixMultiplyShared kernel, which benefits from Tensor Cores, showed significant reductions in execution time. When combined with CUDA streams, multiple instances of this optimized kernel run concurrently, further speeding up the overall process.
2. Memory Throughput: Large data transfers handled by CUDA streams are now matched by faster computations, ensuring that the GPU's memory bandwidth is fully utilized without causing bottlenecks.

Combining CUDA Streams with Loop Unrolling (op_2)

Loop Unrolling is a technique where multiple iterations of a loop are executed within a single loop body. This reduces the overhead of loop control and allows the GPU to execute more instructions in parallel.

- Faster Kernel Execution: Loop unrolling makes each kernel run faster by minimizing the number of loop iterations and maximizing instruction-level parallelism. This means each stream can process its data slice more quickly.
- Improved Resource Utilization: Faster kernels free up GPU resources sooner, allowing other streams to utilize the GPU without waiting, thus maintaining high overall utilization.

e. List your references used while implementing this technique. (you must mention textbook pages at the minimum)
   Here is what I used for reference when I work on the optimization:
   Kirk, D. B., & Hwu, W. W. (2016). In *Programming massively parallel processors: A hands-on approach* (3rd ed., pp. 285–287) Morgan Kaufmann. The textbook discusses the behavior, creation, and synchronization of streams, as well as their role in enabling concurrency in kernel launches and memory operations

https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html : The websites comprehensive information on CUDA streams, including their creation, management, and synchronization mechanisms.

https://developer.nvidia.com/blog/how-overlap-data-transfers-cuda-cc/: This article provides practical insights into overlapping data transfers with computation using CUDA streams

2. **Req_1: Joint Register and Shared Memory Tiling**
   https://drive.google.com/drive/folders/1CQrDuvx1pBKjwPi9eSUndQ-qlBzN-_Cx?usp=sharing
   a. **How does this optimization theoretically optimize your convolution kernel? Expected behavior?**
      The optimization uses joint register and shared memory tiling, introducing REG_TILE_SIZE to improve memory efficiency and speed. Unlike the original convolution kernel, it reduces global memory accesses by storing intermediate data in registers and tiling computations. This minimizes memory delays, increases data reuse, and enables finer-grained parallelism. The result is faster execution, better scalability for large matrices, and more efficient GPU resource utilization.
   b. **How did you implement your code? Explain thoroughly and show code snippets. Justify the correctness of your implementation with proper profiling results.**

Implementation Details:

a.Optimized Matrix Multiplication Kernel: matrixMultiplySharedOptimized

The core of my optimization lies within the matrixMultiplySharedOptimized kernel. Here's a detailed breakdown of how I implemented joint register and shared memory tiling:

```
#define TILE_WIDTH 16
#define REG_TILE_SIZE 4  // Number of elements per thread

__global__ void matrixMultiplySharedOptimized(const float *A, const float *B, float *C,
                                              int numARows, int numAColumns,
                                              int numBRows, int numBColumns,
                                              int numCRows, int numCColumns) {
  // Allocate shared memory for tiles of A and B
  __shared__ float tileA[TILE_WIDTH][TILE_WIDTH];
  __shared__ float tileB[TILE_WIDTH][TILE_WIDTH];

  // Calculate global row and column indices
```

```
    int row = blockIdx.y * TILE_WIDTH + threadIdx.y;
    int col = blockIdx.x * TILE_WIDTH + threadIdx.x;

    float value = 0.0f;

    // Loop over tiles of A and B matrices
    for (int m = 0; m < (numAColumns + TILE_WIDTH - 1) / TILE_WIDTH; ++m) {
        // Load elements into shared memory with boundary checks
        if (row < numARows && m * TILE_WIDTH + threadIdx.x < numAColumns) {
            tileA[threadIdx.y][threadIdx.x] = A[row * numAColumns + m * TILE_WIDTH +
threadIdx.x];
        } else {
            tileA[threadIdx.y][threadIdx.x] = 0.0f;
        }

        if (col < numBColumns && m * TILE_WIDTH + threadIdx.y < numBRows) {
            tileB[threadIdx.y][threadIdx.x] = B[(m * TILE_WIDTH + threadIdx.y) *
numBColumns + col];
        } else {
            tileB[threadIdx.y][threadIdx.x] = 0.0f;
        }

        __syncthreads();  // Ensure all threads have loaded their tiles

        // **Register Tiling Implementation**
        // Process the tiles in chunks of REG_TILE_SIZE to utilize registers
        for (int k = 0; k < TILE_WIDTH; k += REG_TILE_SIZE) {
            // Load REG_TILE_SIZE elements into registers
            float regA[REG_TILE_SIZE];
            float regB[REG_TILE_SIZE];
```

```
            for (int i = 0; i < REG_TILE_SIZE; ++i) {
                int idx = k + i;
                if (idx < TILE_WIDTH) {
                    regA[i] = tileA[threadIdx.y][idx];
                    regB[i] = tileB[idx][threadIdx.x];
                } else {
                    regA[i] = 0.0f;
                    regB[i] = 0.0f;
                }
            }

            // Perform multiply-add operations using register-loaded data
            for (int i = 0; i < REG_TILE_SIZE; ++i) {
                value += regA[i] * regB[i];
            }
        }

        __syncthreads();  // Synchronize before loading the next tile
    }

    // Write the computed value to the output matrix
    if (row < numCRows && col < numCColumns) {
        C[row * numCColumns + col] = value;
    }
}
```

b. Key Implementation Highlights

1. Shared Memory Allocation:

```
__shared__ float tileA[TILE_WIDTH][TILE_WIDTH];
__shared__ float tileB[TILE_WIDTH][TILE_WIDTH];
```

- I allocated shared memory tiles for sub-blocks of matrices A and B to facilitate data reuse and reduce global memory accesses.

2. Loading Data into Shared Memory:

```
if (row < numARows && m * TILE_WIDTH + threadIdx.x < numAColumns) {
   tileA[threadIdx.y][threadIdx.x] = A[row * numAColumns + m * TILE_WIDTH +
threadIdx.x];
} else {
   tileA[threadIdx.y][threadIdx.x] = 0.0f;
}

if (col < numBColumns && m * TILE_WIDTH + threadIdx.y < numBRows) {
   tileB[threadIdx.y][threadIdx.x] = B[(m * TILE_WIDTH + threadIdx.y) *
numBColumns + col];
} else {
   tileB[threadIdx.y][threadIdx.x] = 0.0f;
}
```

- Each thread loads one element of A and B into shared memory, with boundary checks to prevent out-of-bounds accesses.

3. Register Tiling and Loop Unrolling:

```
for (int k = 0; k < TILE_WIDTH; k += REG_TILE_SIZE) {
   float regA[REG_TILE_SIZE];
   float regB[REG_TILE_SIZE];

   for (int i = 0; i < REG_TILE_SIZE; ++i) {
      int idx = k + i;
      if (idx < TILE_WIDTH) {
         regA[i] = tileA[threadIdx.y][idx];
         regB[i] = tileB[idx][threadIdx.x];
      } else {
         regA[i] = 0.0f;
         regB[i] = 0.0f;
```

```
        }
    }

    for (int i = 0; i < REG_TILE_SIZE; ++i) {
        value += regA[i] * regB[i];
    }
}
```

- By processing REG_TILE_SIZE elements per iteration, I reduced loop overhead and increased instruction-level parallelism.
- Loading data into registers (regA and regB) allowed for rapid computations without repeatedly accessing shared memory.

4. Synchronization:

```
__syncthreads();
```

- I used __syncthreads() to ensure all threads have completed loading their respective tiles before proceeding to computation.

c. Integration within conv_forward_gpu_prolog

In the conv_forward_gpu_prolog function, I allocated memory and set up the necessary data structures on the GPU. Here's how I integrated the optimized kernel:

```
__host__ void GPUInterface::conv_forward_gpu_prolog(const float *host_output, const float *host_input, const float *host_mask, float **device_output_ptr, float **device_input_ptr, float **device_mask_ptr, const int Batch, const int Map_out, const int Channel, const int Height, const int Width, const int K)
{
    // Allocate device memory for input
    size_t input_size = Batch * Channel * Height * Width * sizeof(float);
    cudaMalloc((void**) device_input_ptr, input_size);
    cudaMemcpy(*device_input_ptr, host_input, input_size, cudaMemcpyHostToDevice);

    // Allocate device memory for output
```

```
    const int Height_out = Height - K + 1;
    const int Width_out = Width - K + 1;
    size_t output_size = Batch * Map_out * Height_out * Width_out * sizeof(float);
    cudaMalloc((void**) device_output_ptr, output_size);

    // Allocate device memory for mask
    size_t mask_size = Map_out * Channel * K * K * sizeof(float);
    cudaMalloc((void**) device_mask_ptr, mask_size);
    cudaMemcpy(*device_mask_ptr, host_mask, mask_size, cudaMemcpyHostToDevice);

    // Check for errors
    cudaError_t error = cudaGetLastError();
    if(error != cudaSuccess)
    {
       std::cout<<"CUDA error (prolog): "<<cudaGetErrorString(error)<<std::endl;
       exit(-1);
    }
}
```

1. Memory Allocation and Data Transfer:
    - I allocated memory for the input, output, and mask on the GPU.
    - Data from the host (CPU) was copied to the device (GPU) using cudaMemcpy.
2. Error Checking:
    - After memory operations, I included error checks to ensure that allocations and transfers were successful.

Justifying Correctness and Efficiency with Profiling Results

To ensure the correctness and effectiveness of my optimizations, I analyzed the profiling results provided. Here's a breakdown of the key findings:

a. GPU Kernel Execution Summary

[6/8] Executing 'gpukernsum' stats report

Time (%)	Total Time (ns)	Instances	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	GridXYZ	BlockXYZ	Name
32.2	45847985	10	4584798.5	4584738.0	4583810	4586786	824.6	400000 1 1	16 16 1	matrixMultiplySharedOptimized(...)
27.1	38481356	10	3848135.6	3847793.5	3846945	3850305	1086.0	400000 4 1	16 16 1	matrix_unrolling_kernel(...)
...										

matrixMultiplySharedOptimized()

- Significant Contribution: The matrixMultiplySharedOptimized kernel accounts for approximately 32.2% of the total GPU kernel execution time. This indicates its crucial role in overall computation.
- Consistent Performance: With an average execution time of ~4.58 ms per instance and a low standard deviation (~824.6 ns), the kernel demonstrates consistent and reliable performance across all instances.

matrix_unrolling_kernel()

- Balanced Workload: The matrix_unrolling_kernel contributes 27.1% of the total GPU kernel time, showcasing a balanced distribution between data preparation (unrolling) and computation (multiplication).
- Reliable Execution: The low standard deviation (~1086.0 ns) suggests that the kernel performs reliably across multiple executions, which is essential for maintaining consistent performance.

b. GPU Kernel Execution Time

[6/8] Executing 'gpukernsum' stats report

...

32.2 45847985 10 4584798.5 ... matrixMultiplySharedOptimized(...)

27.1 38481356 10 3848135.6 ... matrix_unrolling_kernel(...)

...

Interpretation:

- Efficient Kernel Execution: The optimized kernels (matrixMultiplySharedOptimized and matrix_unrolling_kernel) collectively account for 59.3% of the GPU kernel execution time. This demonstrates effective utilization of GPU resources through joint tiling techniques.
- Scalability: Given the efficient handling by these kernels, scaling to larger matrices or higher batch sizes should maintain or even improve performance, provided memory constraints are appropriately managed.

c. <u>Did the performance match your expectations? Analyze the profiling results as a scientist.</u>

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*4.10903ms*	*3.25218ms*	*1.411s*	*0.86*
1000	*11.7364ms*	*7.98291ms*	*9.385s*	*0.886*
10000	*91.3576ms*	*57.2313ms*	*88.581s*	*0.8714*

Performance Overview:

Batch Size	Without Optimization (Total Execution Time)	With Optimization (Total Execution Time)	Speedup
100	*4.016s*	*1.411s*	*~2.8x*
1000	*43.717s*	*9.385s*	*~4.7x*
10000	*363.963s*	*88.581s*	*~4.1x*

Batch Size	Without Optimization (total Op times)	With Optimization (Total Op Times)
100	*7.17506ms*	*7.36121ms*
1000	*18.14643ms*	*19.71931ms*
10000	*128.6454ms*	*148.5889ms*

Profiling Analysis:

a. Significant Reduction in Total Execution Time

The most striking observation from the profiling data is the dramatic decrease in total execution time across all batch sizes after implementing joint register and shared memory tiling:

Batch Size 100: Reduced from 4.016s to 1.411s

Batch Size 1,000: Reduced from 43.717s to 9.385s

Batch Size 10,000: Reduced from 363.96300s to 88.581s

This indicates that the optimizations had a profound impact on the overall efficiency of the convolution operation, achieving speedups of approximately 2.8x, 4.7x, and 4.1x for batch sizes of 100, 1,000, and 10,000 respectively.

b. Minimal Impact on Individual Operation Times

Contrary to the substantial improvements in total execution time, the individual operation times (Op Time 1 and Op Time 2) exhibited only modest changes:

Op Time 1: Slightly increased across all batch sizes.

Op Time 2: Remained relatively consistent, with minor decreases.

For instance, at a batch size of 10,000, Op Time 1 increased from 71.0807ms to 91.3576ms, while Op Time 2 saw a negligible decrease from 57.5647ms to 57.2313ms.

c. Understanding the Discrepancy Between Op Times and Total Execution Time

Here is my explanation on Why did the total execution time decrease so significantly when the individual operation times did not reflect similar improvements

Optimizations Affecting Other Components

While Op Time 1 and Op Time 2 represent specific segments of the convolution operation, the total execution time encompasses the entire workflow, including:

- Data Transfer Overheads: Transfers between host (CPU) and device (GPU) memory can be substantial, especially with large batch sizes.
- Memory Allocation and Deallocation: Efficient memory management can reduce latency and improve throughput.
- Kernel Launch Overheads: Optimizing how and when kernels are launched can lead to better utilization of GPU resources.

The joint register and shared memory tiling primarily optimized the matrix multiplication phase within the convolution operation. This phase is often a significant portion of the total computation time, especially for large matrices and batch sizes. By enhancing the efficiency of this core computation, the overall workflow benefits, even if other operations remain unchanged or experience minimal improvements.

Enhanced Parallelism and Resource Utilization

The optimization techniques employed likely improved parallelism and resource utilization in ways that are not directly captured by Op Time 1 and Op Time 2. For example:

- Better Occupancy: Joint register and shared memory tiling can lead to higher occupancy by maximizing the number of active threads and blocks on the GPU, thus reducing idle times.

- Reduced Memory Bottlenecks: By efficiently managing shared memory and registers, the optimized kernels can process data more swiftly, freeing up GPU resources for other tasks or allowing more concurrent operations.

Scalability Improvements
The profiling results demonstrate that the optimized implementation scales more gracefully with increasing batch sizes:

Batch Size 100 to 10,000: The original implementation's execution time increased by nearly 90x, whereas the optimized version saw an increase of only about 63x. This suggests that the optimizations have effectively mitigated some of the scalability challenges inherent in large-scale matrix multiplications, such as memory bandwidth limitations and increased computational overhead.

d. <u>Does this optimization synergize with any other optimizations? How?</u>

I didn't apply other optimizations in this code. However, here is what I think of synergizing other optimizations:

1. Streams for Overlapping Computation with Data Transfer (req_0)

CUDA streams enable overlapping data transfers (host ↔ device) with computation. While one set of data is being processed on the GPU, another batch of data can be transferred asynchronously, effectively hiding memory transfer latency.

- Reduced Stall Times: The tiling optimization minimizes memory access bottlenecks for computation, and streams further reduce idle GPU times by ensuring the GPU always has data ready for processing.

- Pipeline Efficiency: Joint tiling makes the computation phase highly efficient. Overlapping data transfers with computation ensures that this efficiency isn't hindered by data movement delays.

2. Combined Optimization: Kernel Fusion (req_2)

Kernel fusion merges multiple kernels (e.g., unrolling and matrix multiplication) into a single kernel, reducing kernel launch overheads and global memory accesses.

- Fewer Global Memory Accesses: Joint tiling already optimizes memory locality, and kernel fusion eliminates intermediate global memory writes and reads between unrolling and matrix multiplication kernels.
- Enhanced GPU Occupancy: By combining operations, more computational resources can be utilized simultaneously, improving throughput.

Implementation Approach:

- Merge matrix_unrolling_kernel and matrixMultiplySharedOptimized into a single kernel.
- Use shared memory for intermediate results (e.g., storing unrolled matrices directly in shared memory for matrix multiplication).

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

Here is what I used for reference when I work on the optimization:

Kirk, D. B., & Hwu, W.-M. W. (2016). *Programming massively parallel processors: A hands-on approach* (3rd ed., pages 73-77). Morgan Kaufmann., discusses shared memory use in tiled matrix multiplication.

https://lumetta.web.engr.illinois.edu/508/slides/lecture4.pdf introduces the joint register and shared memory tiling in matrix multiplication

3. **Req_2: Kernel Fusion**

   https://drive.google.com/drive/folders/1FolGmKkR-cWlymJ5ty9VmiFySkT8xOgx?usp=sharing

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

      The kernel fusion optimization combines unrolling, matrix multiplication, and output permutation into a single kernel, eliminating intermediate memory transfers required in original convolution kernel. This reduces global memory overhead, latency, and kernel launch delays while effectively using shared memory to minimize redundant fetches and boost data reuse. The result is faster execution, better scalability for large workloads, and improved energy efficiency, making it ideal for high-performance convolution operations.

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

Implementation Details:

To optimize, I fused the unrolling and matrix multiplication processes into a single kernel, named fused_conv_kernel. This unified kernel handles both input transformation and convolution, eliminating the need for separate kernel launches and intermediate global memory accesses.

Here's a breakdown of the fused_conv_kernel and how it integrates multiple operations:

```
#define TILE_WIDTH 16

#define BLOCK_SIZE 256

#define MAX_BATCH_SIZE 1000


__global__ void fused_conv_kernel(const float *input, const float *mask, float *output,
                                  const int Batch, const int Map_out, const int Channel,
                                  const int Height, const int Width, const int K) {
    const int Height_out = Height - K + 1;
```

```
const int Width_out = Width - K + 1;
const int H_unroll = Channel * K * K;
const int W_unroll = Batch * Height_out * Width_out;

// Shared memory for mask and input tiles
__shared__ float tileA[TILE_WIDTH][TILE_WIDTH];
__shared__ float tileB[TILE_WIDTH][TILE_WIDTH];

int by = blockIdx.y;  // Output feature map index (Map_out dimension)
int bx = blockIdx.x;  // Column index in the unrolled input matrix
int ty = threadIdx.y;
int tx = threadIdx.x;

int row = by * TILE_WIDTH + ty;  // Index in mask (filter weights)
int col = bx * TILE_WIDTH + tx;  // Index in unrolled input

float val = 0.0f;

// Loop over tiles of the unrolled input
for (int m = 0; m < (H_unroll - 1) / TILE_WIDTH + 1; ++m) {
    // Load mask tile into shared memory
    if (row < Map_out && m * TILE_WIDTH + tx < H_unroll) {
        tileA[ty][tx] = mask[row * H_unroll + m * TILE_WIDTH + tx];
```

```
} else {
    tileA[ty][tx] = 0.0f;
}

// Compute indices for the input
int h_unroll = m * TILE_WIDTH + ty;
int w_unroll = col;

if (h_unroll < H_unroll && w_unroll < W_unroll) {
    int c = h_unroll / (K * K);
    int p = (h_unroll % (K * K)) / K;
    int q = (h_unroll % (K * K)) % K;

    int b = w_unroll / (Height_out * Width_out);
    int remainder = w_unroll % (Height_out * Width_out);
    int h = remainder / Width_out;
    int w = remainder % Width_out;

    int input_row = h + p;
    int input_col = w + q;

    if (b < Batch && c < Channel && input_row < Height && input_col < Width) {
```

```
            tileB[ty][tx] = input[b * (Channel * Height * Width) + c * (Height * Width) +
input_row * Width + input_col];
        } else {
            tileB[ty][tx] = 0.0f;
        }
    } else {
        tileB[ty][tx] = 0.0f;
    }

    __syncthreads();

    // Perform the multiplication and accumulation
    if (row < Map_out && col < W_unroll) {
        for (int k = 0; k < TILE_WIDTH; k++) {
            val += tileA[ty][k] * tileB[k][tx];
        }
    }

    __syncthreads();
}

// Write the output directly to the correct position
if (row < Map_out && col < W_unroll) {
```

```
        int b = col / (Height_out * Width_out);
        int remainder = col % (Height_out * Width_out);
        int h = remainder / Width_out;
        int w = remainder % Width_out;

        if (b < Batch && h < Height_out && w < Width_out) {
            output[b * (Map_out * Height_out * Width_out) + row * (Height_out * Width_out) + h * Width_out + w] = val;
        }
    }
}
```

Key Components of the Fused Kernel

1. **Shared Memory Allocation:**

```
__shared__ float tileA[TILE_WIDTH][TILE_WIDTH];
__shared__ float tileB[TILE_WIDTH][TILE_WIDTH];
```

- **tileA:** Stores tiles of the convolutional masks (mask).
- **tileB:** Stores tiles of the unrolled input matrix (input).

2. **Thread and Block Indexing:**

```
int by = blockIdx.y;  // Output feature map index
int bx = blockIdx.x;  // Column index in the unrolled input matrix
int ty = threadIdx.y;
int tx = threadIdx.x;
```

```
int row = by * TILE_WIDTH + ty;
int col = bx * TILE_WIDTH + tx;
```

- Each thread is responsible for computing a specific element in the output matrix.

3. **Loading Data into Shared Memory:**

```
// Load mask tile
if (row < Map_out && m * TILE_WIDTH + tx < H_unroll) {
    tileA[ty][tx] = mask[row * H_unroll + m * TILE_WIDTH + tx];
} else {
    tileA[ty][tx] = 0.0f;
}

// Load input tile
if (h_unroll < H_unroll && w_unroll < W_unroll) {
    // Compute indices and load input
    // ...
    tileB[ty][tx] = input[...];
} else {
    tileB[ty][tx] = 0.0f;
}

__syncthreads();
```

- Each thread loads one element of mask and one element of input into shared memory, handling boundary conditions to avoid out-of-bounds accesses.

4. **Multiplication and Accumulation:**

```
if (row < Map_out && col < W_unroll) {
   for (int k = 0; k < TILE_WIDTH; k++) {
      val += tileA[ty][k] * tileB[k][tx];
   }
}

__syncthreads();
```

- Each thread computes the dot product of a row from tileA and a column from tileB, accumulating the result into val.

5. **Writing the Output:**

```
if (row < Map_out && col < W_unroll) {
   // Compute output indices
   // ...
   output[...] = val;
}
```

- The final computed value val is written directly to the appropriate position in the output matrix.

Justification of Correctness and Performance Using Profiling Results:

The provided profiling output (profile.out) offers valuable insights into the performance and efficiency of the implemented kernel fusion. Here's how the profiling data justifies the correctness and effectiveness of the implementation:

1. Kernel Execution Efficiency

- **gpukernsum Section:**

[6/8] Executing 'gpukernsum' stats report

Time (%)	Total Time (ns)	Instances	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	GridXYZ	BlockXYZ	Name
55.6	39364733	10	3936473.3	3936224.0	3935936	3937664	546.1	400000 1 1	16 16 1	fused_conv_kernel(const float *, const float *, float *, int, int, int, int, int, int)
44.4	31422089	10	3142208.9	3141937.0	3141153	3143777	780.4	72250 1 1	16 16 1	fused_conv_kernel(const float *, const float *, float *, int, int, int, int, int, int)

- **High Kernel Utilization:**
  - The fused convolution kernel accounts for **55.6%** of the total GPU kernel execution time, indicating that most of the computational workload is handled efficiently within this kernel.
- **Consistent Execution Time:**
  - Each instance of the kernel executes in approximately **3.94 milliseconds**, showing consistent performance across different batches.
- **Grid and Block Configuration:**
  - The kernel is launched with **16x16** blocks, leveraging the GPU's parallel processing capabilities effectively.

2. Memory Operations Optimization

- **gpumemtimesum Section:**

  [7/8] Executing 'gpumemtimesum' stats report

Time (%)	Total Time (ns)	Count	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	Operation

-------- --------------- ----- ----------- ----------- --------- --------- ----------- ----------------

86.8 274056909 2 137028454.5 137028454.5 105772317 168284592 44202853.6 [CUDA memcpy DtoH]

13.2 41731715 6 6955285.8 1888.0 1504 22452798 10819275.0 [CUDA memcpy HtoD]

- **Reduced Memory Transfers:**
  - The majority of memory transfer time is spent copying data from device to host (**86.8%**), which is expected as the results need to be retrieved after computation.
- **Minimized Host-to-Device Transfers:**
  - Only **13.2%** of the memory transfer time is spent copying data from host to device, indicating that data transfers are minimized, likely due to kernel fusion reducing the number of necessary transfers.

3. CUDA API Utilization

- **cudaapisum Section:**

[5/8] Executing 'cudaapisum' stats report

Time (%) Total Time (ns) Num Calls Avg (ns) Med (ns) Min (ns) Max (ns) StdDev (ns) Name

-------- --------------- --------- ---------- --------- -------- --------- ----------- ----------------

56.8 317710430 8 39713803.8 9780960.5 17603 169129695 63470456.7 cudaMemcpy

29.2 163421764 8 20427720.5 229149.5 122119 161554095 57023977.9 cudaMalloc

12.7 70749009 6 11791501.5 7644.0 4128 39334067 18429738.7 cudaDeviceSynchronize

1.2 6862243 8 857780.4 763855.5 131276 2035292 641115.8 cudaFree

0.1 365612 24 15233.8 3732.0 3196 106960 25633.5 cudaLaunchKernel

0.0 1332 1 1332.0 1332.0 1332 1332 0.0 cuModuleGetLoadingMode

- Efficient Memory Allocation:
  - The time spent on cudaMalloc and cudaMemcpy is balanced, ensuring that memory allocation does not become a bottleneck.
- Synchronization Overhead:
  - A reasonable amount of time is spent on cudaDeviceSynchronize, ensuring that kernel executions are properly completed before proceeding.

c. <u>Did the performance match your expectations? Analyze the profiling results as a scientist.</u>

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.463088ms*	*0.33096ms*	*1.499s*	*0.86*
1000	*3.93938 ms*	*3.15282ms*	*10.129s*	*0.886*
10000	*38.7915ms*	*31.3053ms*	*92.604s*	*0.8714*

Performance Overview:

Batch Size	Baseline (Total Execution Time)	With Optimization (Total Execution Time)	Speedup
100	*4.016s*	*1.499s*	*~2.7x*
1000	*43.717s*	*10.129s*	*~4.3x*
10000	*363.963s*	*92.604s*	*~3.9x*

Quantitative Analysis

- Batch Size 100:
    - Speedup: The fused implementation executes approximately 2.68 times faster than the original.
- Batch Size 1,000:
    - Speedup: The fused implementation achieves a 4.31× speedup over the original.
- Batch Size 10,000:
    - Speedup: The fused version demonstrates a 3.93× speedup compared to the original.

Batch Size	Baseline (total Op times)	With Optimization (Total Op Times)	Reduction
100	*7.17506ms*	*0.794048ms*	*~88.9%*
1000	*18.14643ms*	*7.0922ms*	*~60.9%*
10000	*128.6454ms*	*70.0968ms*	*~45.51%*

Interpretation of Reductions

- Significant Decrease in Operation Times:
    - Both Op Time 1 and Op Time 2 exhibit substantial reductions across all batch sizes when using kernel fusion.
- Proportional Decrease Relative to Batch Size:
    - Smaller Batch Sizes (100): Experience the highest percentage reduction (~88-90%) in both operations, indicating that kernel fusion is particularly effective for smaller workloads.
    - Medium Batch Sizes (1,000): Show a moderate reduction (~60%) in operation times, still reflecting substantial optimization.
    - Large Batch Sizes (10,000): While the percentage reduction decreases (~45-46%), there remains a significant improvement in operation times.

Profiling Analysis

From the previously provided profiling data (profile.out), several key observations support the performance improvements observed with kernel fusion:

- Increased Kernel Execution Time Share:
    - The fused_conv_kernel accounted for 55.6% of the GPU kernel execution time, a significant increase compared to the original implementation where multiple kernels may have distributed the workload. This concentration suggests that the fused kernel is handling a substantial portion of the computational workload efficiently.
- Optimized Memory Transfers:
    - With kernel fusion, memory transfers from host to device and device to host have been optimized, reducing the number of required transfers and their associated latencies. This contributes directly to the decreased total execution times.
- Efficient Utilization of Shared Memory:
    - The fused kernel effectively utilizes shared memory (tileA and tileB) to store input and mask data, minimizing global memory accesses. This enhances data reuse and reduces memory bandwidth bottlenecks, further accelerating computations.

d. <u>Does this optimization synergize with any other optimizations? How?</u>

I didn't put other optimizations in this code. However, here is what I think of synergizing other optimizations:

1. Using Constant Memory for Weights (op_0)

Storing the convolutional weights (kernels) in CUDA's constant memory, which is a fast, read-only memory optimized for scenarios where all threads access the same data.

- Faster Access: Kernel Fusion already minimizes global memory access by loading data into shared memory. By placing weights in constant memory, threads can quickly access these weights without the latency of global memory.
- Reduced Bandwidth Usage: This combination ensures that weight data is reused efficiently across threads, cutting down on the number of memory fetches and saving bandwidth.

Profiling Insight: With Kernel Fusion, your Total Execution Time dropped from 4.016 seconds to 1.499 seconds for a batch size of 100. Adding constant memory can further decrease data transfer times, enhancing this speedup.

2. Loop Unrolling (op_2)

An optimization where loops are expanded to execute multiple iterations in a single loop cycle, reducing the overhead of loop control and increasing instruction-level parallelism.

- Higher Throughput: In the fused kernel, loop unrolling allows more computations to be done per thread without waiting for loop iterations, making the GPU's processing units work more efficiently.
- Less Overhead: By decreasing the number of loop iterations, the GPU spends less time managing loops and more time performing actual computations.

Profiling Insight: After applying Kernel Fusion, Op Time 2 reduced from 3.912 ms to 0.463 ms for a batch size of 100. Loop unrolling can further trim this time by speeding up the computation within each thread.

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

Here is what I used for reference when I work on the kernel fusion's optimization:

Kirk, D. B., & Hwu, W. W. (2016). In *Programming massively parallel processors: A hands-on approach* (3rd ed., pp. 487–491) Morgan Kaufmann. The chapter discusses Fusion as an optimization strategy to minimize off-chip memory transactions and improve bandwidth utilization

Wang, G., Lin, Y., & Yi, W. (2010, December). Kernel fusion: An effective method for better power efficiency on multithreaded GPU. In *2010 IEEE/ACM Int'l Conference on Green Computing and Communications & Int'l Conference on Cyber, Physical and Social Computing* (pp. 344-350). IEEE. It provides the methodologies discussed can also inform performance optimization strategies.

4. **Op_0: Weight matrix (Kernel) in constant memory**
   https://drive.google.com/drive/folders/1H5ip0JFDm4cKychj17e0NJB-Dv9KFuAn?usp=sharing
   a. **How does this optimization theoretically optimize your convolution kernel? Expected behavior?**
      Storing the weight matrix in constant memory improves efficiency by leveraging fast, cached access compared to the slower global memory approach in original convolution kernel. This reduces memory latency, lowers bandwidth usage, and avoids redundant fetches, especially during convolution operations where weights are reused extensively. The result is faster execution, better GPU utilization, and improved scalability for larger batch sizes and filter-heavy layers.
   b. **How did you implement your code? Explain thoroughly and show code snippets. Justify the correctness of your implementation with proper profiling results.**

Implementation Details

1. Declaring Constant Memory

To begin, I declared the weight matrix in the device's constant memory space using the __constant__ qualifier provided by CUDA. Here's how I set it up in my code:

```
#define MAX_MASK_SIZE 8192


__constant__ float const_device_mask[MAX_MASK_SIZE];
```

- Breakdown:
  - MAX_MASK_SIZE defines the maximum allowable size for the mask (weight matrix) in constant memory.
  - const_device_mask is the constant memory array that holds the weights. By placing it in constant memory, all threads can access it quickly through the cached memory system.

2. Copying Data to Constant Memory

During the prolog phase of my GPU interface, I transferred the weight matrix from the host (CPU) to the device's constant memory. This was done using cudaMemcpyToSymbol, which is specifically designed for copying data to device symbols like constant memory arrays.

```
__host__ void GPUInterface::conv_forward_gpu_prolog(
    const float *host_output,
    const float *host_input,
    const float *host_mask,
    float **device_output_ptr,
    float **device_input_ptr,
    float **device_mask_ptr,
    const int Batch,
    const int Map_out,
    const int Channel,
    const int Height,
    const int Width,
    const int K)
{
    // ... [Memory allocations for input and output]

    // Copy mask to constant memory
    size_t mask_size = Map_out * Channel * K * K * sizeof(float);
    if (mask_size > MAX_MASK_SIZE * sizeof(float)) {
        std::cerr << "Error: Mask size exceeds MAX_MASK_SIZE\n";
```

```
    exit(-1);
  }
  cudaMemcpyToSymbol(const_device_mask, host_mask, mask_size);

  // Set device_mask_ptr to NULL since we're using constant memory
  *device_mask_ptr = NULL;

  // ... [Error checking]
}
```

- Explanation:
    - Size Calculation: I calculated the size of the mask based on the number of output maps (Map_out), input channels (Channel), and kernel dimensions (K x K).
    - Safety Check: Before copying, I ensured that the mask size doesn't exceed MAX_MASK_SIZE to prevent any memory overflows.
    - Data Transfer: Using cudaMemcpyToSymbol, I efficiently transferred the mask data from the host to the const_device_mask in device constant memory.
    - Pointer Adjustment: Since the weights are now in constant memory, I set the traditional device mask pointer to NULL to indicate that it's no longer needed elsewhere.

3. Accessing Constant Memory in the Kernel

Within my matrix multiplication kernel (matrixMultiplyShared), I accessed the weight matrix directly from constant memory. Here's a snippet of how I did that:

```
__global__ void matrixMultiplyShared(
  const float *B,
```

```
    float *C,
    int numARows,
    int numAColumns,
    int numBRows,
    int numBColumns,
    int numCRows,
    int numCColumns)
{
    // Shared memory for tiles of A and B
    __shared__ float tileA[TILE_WIDTH][TILE_WIDTH];
    __shared__ float tileB[TILE_WIDTH][TILE_WIDTH];

    int by = blockIdx.y, bx = blockIdx.x, ty = threadIdx.y, tx = threadIdx.x;
    int row = by * TILE_WIDTH + ty, col = bx * TILE_WIDTH + tx;
    float val = 0;

    for (int tileId = 0; tileId < (numAColumns - 1) / TILE_WIDTH + 1; tileId++) {
        // Access weight matrix from constant memory
        if (row < numARows && tileId * TILE_WIDTH + tx < numAColumns) {
            tileA[ty][tx] = const_device_mask[(size_t) row * numAColumns + tileId *
TILE_WIDTH + tx];
        } else {
            tileA[ty][tx] = 0;
```

```
    }

    // Load B matrix into shared memory
    if (col < numBColumns && tileId * TILE_WIDTH + ty < numBRows) {
        tileB[ty][tx] = B[((size_t) tileId * TILE_WIDTH + ty) * numBColumns + col];
    } else {
        tileB[ty][tx] = 0;
    }

    __syncthreads();

    if (row < numCRows && col < numCColumns) {
        for (int i = 0; i < TILE_WIDTH; i++) {
            val += tileA[ty][i] * tileB[i][tx];
        }
    }

    __syncthreads();
}

if (row < numCRows && col < numCColumns) {
    C[row * numCColumns + col] = val;
}
```

}

- Key Points:
    - Tile Loading:
        - tileA: Instead of accessing a global memory pointer, I loaded data directly from const_device_mask. This takes advantage of the cached nature of constant memory, allowing all threads within a warp to access the same data efficiently.
        - tileB: Continues to be loaded from global memory as it represents the input feature maps being convolved with the weights.
    - Synchronization: Using __syncthreads() ensures that all threads have completed loading their respective tiles before proceeding with the multiplication.
    - Computation: The standard tiled matrix multiplication is performed, accumulating the product of corresponding elements.

Correctness and Performance with Profiling Results

To ensure that my implementation was both correct and efficient, I analyzed the provided profiling results (profile.out). Here's a breakdown of what I observed:

1. Kernel Execution Overview

[6/8] Executing 'gpukernsum' stats report

Time (%)	Total Time (ns)	Instances	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	GridXYZ	BlockXYZ	Name
32.7	47697903	10	4769790.3	4770107.5	4766283	4773675	2235.9	72250 1 1	16 16 1	matrixMultiplyShared(const float *, float *, int, int, int, int, int, int)
26.4	38479310	10	3847931.0	3847624.0	3846567	3849896	1000.5	400000 4 1	16 16 1	matrix_unrolling_kernel(const float *, float *, int, int, int, int, int)

19.4 28359547 10 2835954.7 2835638.0 2834854 2837446 1011.2 400000 1 1 16 16 1 matrixMultiplyShared(const float *, float *, int, int, int, int, int, int)

17.1 25001307 10 2500130.7 2500053.5 2499494 2500774 447.1 72250 13 1 16 16 1 matrix_unrolling_kernel(const float *, float *, int, int, int, int, int)

- Insights:
    - matrixMultiplyShared: This kernel consumed about 32.7% of the total GPU kernel execution time, highlighting its significance in the overall computation.
    - matrix_unrolling_kernel: With 26.4% and 17.1% in two separate instances, it's clear that unrolling the input matrix is also a critical component.
- Interpretation:
    - The substantial time spent in matrixMultiplyShared underscores the importance of optimizing matrix multiplication operations. By efficiently accessing the weight matrix through constant memory, I ensured that this kernel runs as smoothly as possible.

2. CUDA API Calls Summary

[5/8] Executing 'cudaapisum' stats report

Time (%)	Total Time (ns)	Num Calls	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	Name
48.7	307475136	6	51245856.0	20999980.0	22623	155106481	65350801.3	cudaMemcpy
26.4	166466111	10	16646611.1	199554.5	122570	164227850	51855046.1	cudaMalloc
24.8	156764373	10	15676437.3	1765433.0	125445	75616116	30288690.7	cudaFree

- cudaMemcpy: Took up 48.7% of the CUDA API time, primarily during data transfers between host and device.
- cudaMalloc and cudaFree: Also significant but expected during memory management phases.
- Analysis:
    - The heavy time consumption by cudaMemcpy indicates that data transfers are a bottleneck. However, by using constant memory for the weights, I reduced the frequency and volume of these transfers, particularly benefiting the Host to Device (HtoD) operations.

3. GPU Memory Transfers Summary

[7/8] Executing 'gpumemtimesum' stats report

Time (%)	Total Time (ns)	Count	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	Operation
86.3	263930939	2	131965469.5	131965469.5	109605123	154325816	31622305.3	[CUDA memcpy DtoH]
13.7	41808315	6	6968052.5	2032.0	1312	22374573	10832365.6	[CUDA memcpy HtoD]

- Key Points:
    - Device to Host (cudaMemcpy DtoH): Dominates the memory transfer time, which is typical as results are often copied back after computation.
    - Host to Device (cudaMemcpy HtoD): Consumes less time, thanks to the one-time transfer of weights to constant memory.
- Implications:

- By storing weights in constant memory, I minimized the need for repeated transfers of the weight matrix from host to device. This not only reduced memory bandwidth usage but also accelerated the convolution operations.

c. <u>Did the performance match your expectations? Analyze the profiling results as a scientist.</u>

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*3.88474ms*	*3.46483ms*	*1.529s*	*0.86*
1000	*9.89853ms*	*10.1213ms*	*9.102s*	*0.886*
10000	*73.8197ms*	*78.4678ms*	*87.130s*	*0.8714*

Performance Overview:

Batch Size	Baseline (Total Execution Time)	With Optimization (Total Execution Time)	Speedup
100	*4.016s*	*1.529s*	*~2.6x*
1000	*43.717s*	*9.102s*	*~4.8x*
10000	*363.963s*	*87.130s*	*~4.2x*

Quantitative Analysis

- Batch Size 100:
  - Speedup: The optimization executes approximately 2.6 times faster than the original.
- Batch Size 1,000:
  - Speedup: The optimization achieves a 4.8× speedup over the original.
- Batch Size 10,000:
  - Speedup: The optimization demonstrates a 4.2× speedup compared to the original.

Batch Size	Baseline (total Op times)	With Optimization (Total Op Times)
100	*7.17506ms*	*7.34957ms*
1000	*18.14643ms*	*20.01983ms*
10000	*128.6454ms*	*152.2875ms*

Interpretation: The Total Op times for optimization has a minor increase. Despite these increases, the total execution time benefits substantially from the optimization.

Profiling Analysis

While leveraging constant memory for the weight matrix is generally expected to enhance performance by providing faster, cached access, the profiling results indicate that the total operation times did not decrease as anticipated. Here's a concise analysis based on the provided profile.out:

1. Kernel Execution Overhead:
    - matrixMultiplyShared and matrix_unrolling_kernel: Although these kernels are optimized to use constant memory, the profiling data shows that they still consume a significant portion of the GPU's execution time. Specifically:
        - matrixMultiplyShared accounts for 32.7% of kernel time.
        - matrix_unrolling_kernel accounts for 26.4% and 17.1% in separate instances.
    - Impact: The inherent complexity and execution time of these kernels might overshadow the benefits gained from using constant memory, especially if the optimizations within these kernels are not fully effective.
2. Constant Memory Limitations:

- Size Constraints: Constant memory is limited (typically 64KB on many GPUs). If the weight matrix approaches or exceeds this limit (MAX_MASK_SIZE set to 8192 floats, which is 32KB), it can lead to:
  - Cache Misses: Exceeding the constant memory capacity results in data being fetched from slower global memory, negating the speed advantages.
  - Increased Access Latency: Threads may experience delays when accessing weights not residing in the constant cache.
- Profiling Evidence: The cudaMemcpyToSymbol operation, although minimal, ensures that the weight matrix fits within the constant memory bounds. Any overshoot can degrade performance, as seen by the increased operation times.

3. Memory Transfer Overheads:
   - CUDA API Calls: The profiling report highlights significant time spent on memory operations:
     - cudaMemcpy takes up 48.7% of CUDA API time, primarily for Device-to-Host transfers.
     - cudaMalloc and cudaFree also consume notable portions.
   - Impact: Even with constant memory optimization, the overhead from memory allocations and data transfers can contribute to increased total operation times, especially if not managed efficiently.

d. <u>Does this optimization synergize with any other optimizations? How?</u>

I didn't put other optimizations in this code. However, here is what I think of synergizing other optimizations:

1. Using Streams to Overlap Computation with Data Transfer (req_0)

- Constant Memory Efficiency: By storing the weight matrix in constant memory, the amount of data that needs to be transferred between the host and device is minimized, as weights are loaded once and reused across multiple operations.

- Overlapping Operations: Utilizing CUDA streams allows for concurrent execution of data transfers and kernel computations. With constant memory reducing the frequency and volume of weight transfers, streams can be more effectively employed to overlap the remaining necessary data transfers (e.g., input batches and output results) with ongoing computations.

Profiling Insights:

- The profiling results show that cudaMemcpy operations consume a significant portion of the total execution time (48.7% for Device-to-Host transfers). By overlapping these memory transfers with kernel executions using streams, the idle GPU time can be reduced, leading to better overall throughput.
- Total Execution Time Reduction: Even though constant memory alone did not decrease total operation times, combining it with streams can hide memory transfer latencies, potentially bringing down the total execution time by ensuring that the GPU remains active while data is being transferred.

2. Using Tensor Cores/Joint Register and Shared Memory Tiling to Speed Up Matrix Multiplication (req_1)

- Enhanced Matrix Multiplication: Tensor Cores are specialized hardware units in NVIDIA GPUs designed to accelerate matrix operations, particularly beneficial for deep learning workloads. Coupled with shared memory tiling, they can significantly speed up matrix multiplication tasks.
- Optimized Weight Access: With the weight matrix residing in constant memory, Tensor Cores can access these weights more rapidly and efficiently, reducing latency during matrix multiplication. Shared memory tiling further optimizes data reuse and minimizes global memory accesses.

Profiling Insights:

- The profiling data indicates that matrixMultiplyShared consumes about 32.7% of the kernel execution time. Enhancing this kernel with Tensor Cores can drastically reduce its execution time.
- Constant Memory Benefits: Since the weight matrix is accessed frequently during matrix multiplication, having it in constant memory ensures that Tensor Cores can

fetch these weights with minimal latency, maximizing their throughput capabilities.

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

Here is what I used for reference when I work on the optimization:

Kirk, D. B., & Hwu, W. W. (2016). In *Programming massively parallel processors: A hands-on approach* (3rd ed., pp. 71-99) Morgan Kaufmann. The chapter mentions the constant memory usage in parallel programming

https://leimao.github.io/blog/CUDA-Constant-Memory/ This article provides a comprehensive overview of constant memory in CUDA, including its usage and benefits.

https://alexminnaar.com/2019/07/12/implementing-convolutions-in-cuda.html This blog discusses the advantages of storing the convolutional kernel in constant memory and provides insights into optimizing memory access patterns for better performance.

5. **Op_1: __restrict__**
   https://drive.google.com/drive/folders/1IY9j_Md1Pj3A7EPF1BaihwtJbFmjHWMy?usp=sharing
   a. **How does this optimization theoretically optimize your convolution kernel? Expected behavior?**
      The __restrict__ qualifier optimizes memory access by ensuring pointers don't overlap, allowing the compiler to apply advanced optimizations like better scheduling, coalescing, and register usage. This reduces memory latency, improves throughput, and enhances kernel performance, especially in convolution operations with large inputs, making it faster and more scalable than original convolution kernel.
   b. **How did you implement your code? Explain thoroughly and show code snippets. Justify the correctness of your implementation with proper profiling results.**

Implementation Details

1. Matrix Unrolling Kernel

```
__global__ void matrix_unrolling_kernel(const float *__restrict__ input, float *__restrict__ output,
                                        const int Batch, const int Channel,
                                        const int Height, const int Width,
                                        const int K) {
    // Kernel implementation...
}
```

2. Matrix Multiplication Kernel

```
__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)
{
    // Kernel implementation...
}
```

3. Matrix Permutation Kernel

```
__global__ void matrix_permute_kernel(const float *__restrict__ input, float *__restrict__
output, int Map_out,
                                int Batch, int image_size) {
    // Kernel implementation...
}
```

In each of these kernels, the input and output pointers are marked with the __restrict__ qualifier. This indicates to the CUDA compiler that:

- For input and A, B: These pointers are the sole references to their respective memory regions during the kernel's execution.
- For output, C: Similarly, these pointers do not alias with any other pointers, ensuring that writes to output or C do not interfere with reads from input, A, or B.

Justification of Correctness and Performance Using Profiling Results (profile.out):

1. High Utilization of Kernels with __restrict__:

   [6/8] Executing 'gpukernsum' stats report

Time (%)	Total Time (ns)	Instances	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	GridXYZ	BlockXYZ	Name
---	---	---	---	---	---	---	---	---	---	---

30.2 38481491 10 3848149.1 3848145.5 3845026 3849762 1325.5 400000 4 1 16 16 1 matrix_unrolling_kernel(const float *, float *, int, int, int, int, int)

22.6 28707818 10 2870781.8 2870897.0 2869697 2871489 670.1 400000 1 1 16 16 1 matrixMultiplyShared(const float *, const float *, float *, int, int, int, int, int, int)

22.4 28514231 10 2851423.1 2851331.0 2850722 2852994 673.0 72250 1 1 16 16 1 matrixMultiplyShared(const float *, const float *, float *, int, int, int, int, int, int)

19.7 25025813 10 2502581.3 2502194.5 2502050 2504226 703.3 72250 13 1 16 16 1 matrix_unrolling_kernel(const float *, float *, int, int, int, int, int)

2.8 3613825 10 361382.5 361584.0 360128 362176 653.4 25 1000 1 256 1 1 matrix_permute_kernel(const float *, float *, int, int, int)

2.3 2930050 10 293005.0 292976.0 291936 294593 806.6 5 1000 1 256 1 1 matrix_permute_kernel(const float *, float *, int, int, int)

- matrix_unrolling_kernel: Utilizes 30.2% of the GPU time.
- matrixMultiplyShared: Utilizes approximately 45% of the GPU time across two instances.

2. Efficient Memory Operations:

[7/8] Executing 'gpumemtimesum' stats report

Time (%) Total Time (ns) Count Avg (ns) Med (ns) Min (ns) Max (ns) StdDev (ns) Operation

-------- --------------- ----- ----------- ----------- --------- --------- ----------- ----------

85.5 247699635 2 123849817.5 123849817.5 101817206 145882429 31158818.0 [CUDA memcpy DtoH]

14.5 41967802 6 6994633.7 1920.0 1344 22583786 10880775.4 [CUDA memcpy HtoD]

- cudaMemcpy: While memory operations consume a significant portion of time (48.8% DtoH and 14.5% HtoD), the computational kernels are efficiently utilizing the GPU resources.

3. Low Overhead from Synchronization:
   - Minimal time is spent on synchronization operations like cudaLaunchKernel and cudaDeviceSynchronize, indicating that the kernels are executing smoothly without unnecessary stalls.
4. Reduced Memory Access Latency:
   - The use of __restrict__ likely contributes to more predictable and optimized memory access patterns, reducing cache misses and memory latency.
5. Balanced Compute and Memory Throughput:
   - The profiling indicates a good balance between compute operations (matrixMultiplyShared) and memory operations (matrix_unrolling_kernel), suggesting that the compiler optimizations facilitated by __restrict__ are effectively utilized.

Performance Metrics:

- Memory Operations:
  - High-throughput copies (cudaMemcpy): The significant time spent here is expected, as data transfer between host and device is often a bottleneck. However, the compute kernels are not hindered by memory access issues, thanks in part to the optimizations enabled by __restrict__.

c. <u>Did the performance match your expectations? Analyze the profiling results as a scientist.</u>

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*3.90233ms*	*3.33411ms*	*1.452s*	*0.86*
1000	*10.0243ms*	*8.22873ms*	*9.633s*	*0.886*
10000	*74.2287ms*	*59.3957ms*	*92.506s*	*0.8714*

Performance Overview:

Batch Size	Baseline (Total Execution Time)	With Optimization (Total Execution Time)	Speedup
100	*4.016s*	*1.452s*	*~2.7x*
1000	*43.717s*	*9.633s*	*~4.5x*
10000	*363.963s*	*92.506s*	*~3.9x*

Quantitative Analysis

- Batch Size 100:
    - Speedup: The optimization executes approximately 2.7 times faster than the original.
- Batch Size 1,000:
    - Speedup: The optimization achieves a 4.5× speedup over the original.
- Batch Size 10,000:
    - Speedup: The optimization demonstrates a 3.9× speedup compared to the original.

Batch Size	Baseline (total Op times)	With Optimization (Total Op Times)
100	*7.17506ms*	*7.23644ms*
1000	*18.14643ms*	*18.25303ms*
10000	*128.6454ms*	*133.6244ms*

Interpretation: The Total Op times for optimization has a minor increase. Despite these increases, the total execution time benefits substantially from the optimization

Profiling Analysis

Possible Reasons for No Decrease in Total Op Times

1. Limited Impact of __restrict__ on Current Memory Access Patterns:
    - Profile Insight: The profile.out shows that memory operations (cudaMemcpy, cudaMalloc, cudaFree) dominate the execution time, accounting for significant portions (e.g., 48.8% for cudaMemcpy DtoH).
    - Explanation: If the primary bottleneck is memory transfer between host and device, optimizing pointer aliasing with __restrict__ in the kernels has minimal impact on the overall operation times.
2. Kernel Execution Overhead Remains Unaffected:
    - Profile Insight: Kernel launch overheads (cudaLaunchKernel, cudaDeviceSynchronize) are minimal but consistent.
    - Explanation: While __restrict__ optimizes memory access within kernels, it doesn't reduce the inherent overhead of launching kernels or synchronization, which can contribute to total operation times.
3. Memory Bandwidth Constraints:
    - Profile Insight: High utilization of memory operations suggests that memory bandwidth is a limiting factor.
    - Explanation: Even with optimized memory access patterns inside kernels, the available memory bandwidth may cap the achievable performance gains, preventing a reduction in total operation times.

d. <u>Does this optimization synergize with any other optimizations? How?</u>

I didn’t apply other optimizations in this code. However, here is what I think of synergizing other optimizations:

1. Using Streams to Overlap Computation with Data Transfer (req_0)

- Parallel Execution: By utilizing CUDA streams, data transfers (e.g., cudaMemcpy) can be overlapped with kernel executions. While __restrict__

optimizes memory access within kernels, streams ensure that data is fed to the GPU without idle periods, maximizing GPU utilization.

- Reduced Latency: Overlapping computation with data transfer minimizes the waiting time between operations. Since the profiling data indicates that a significant portion of time is spent on memory transfers (e.g., 48.8% on cudaMemcpy DtoH), streams can help mitigate this bottleneck.

Profile Insight:

- Memory Transfer Bottleneck: The profile.out shows that memory operations dominate the execution time. By overlapping these transfers with optimized kernel computations (enhanced by __restrict__), overall execution time can be significantly reduced.

2. Using Tensor Cores/Joint Register and Shared Memory Tiling to Speed Up Matrix Multiplication (req_1)

- Enhanced Compute Performance: Tensor Cores are specialized hardware units in NVIDIA GPUs designed for fast matrix operations, particularly beneficial for deep learning tasks. When combined with __restrict__, which optimizes memory access patterns, Tensor Cores can execute matrix multiplications more efficiently, leveraging the streamlined data flow.

- Shared Memory Tiling: Utilizing shared memory to tile matrices reduces global memory accesses. With __restrict__, the compiler can better optimize these tiled accesses, ensuring that data fetched into shared memory is utilized effectively without unnecessary cache misses.

Profile Insight:

- Matrix Multiplication Time: The gpukernsum section indicates substantial time spent on matrixMultiplyShared. Optimizing this kernel using Tensor Cores and shared memory tiling can drastically reduce its execution time, complementing the memory access optimizations provided by __restrict__.

e. **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

https://developer.nvidia.com/blog/cuda-pro-tip-optimize-pointer-aliasing/

both two websites mention the performance of optimization on using __restrict__

6. **Op_2: Loop unrolling**
   https://drive.google.com/drive/folders/1EbkM0_QxOGMAPFVhZ4zAGGC99KvtbjuJ?usp=sharing
   a. **How does this optimization theoretically optimize your convolution kernel? Expected behavior?**
      Loop unrolling in the code reduces overhead from loop control, improves instruction-level parallelism, and optimizes memory access. Unlike the original convolution kernel, it minimizes branching and redundant calculations, speeding up repetitive tasks like matrix operations and convolutions. This results in faster execution, better scalability for large datasets, and more efficient GPU resource utilization.
   b. **How did you implement your code? Explain thoroughly and show code snippets. Justify the correctness of your implementation with proper profiling results.**

Implementation Details

1. Matrix Unrolling Kernel (matrix_unrolling_kernel)

In the matrix_unrolling_kernel, I focused on unrolling the nested loops that iterate over the channels and the kernel dimensions. By applying loop unrolling, I aimed to minimize the number of iterations and take advantage of the GPU's ability to execute multiple instructions in parallel.

Modified with Loop Unrolling:

```
#pragma unroll
for (int c = 0; c < Channel; ++c) {
    for (int p = 0; p < K; ++p) {
        for (int q = 0; q < K; ++q) {
            int h_unroll = c * K * K + p * K + q;
            int input_row = h + p;
            int input_col = w + q;
```

```
        float val = 0.0f;
        if (input_row < Height && input_col < Width) {
            val = input[b * Channel * Height * Width +
                        c * Height * Width +
                        input_row * Width + input_col];
        }
        output[h_unroll * W_unroll + w_unroll] = val;
      }
    }
}
```

By adding #pragma unroll before the outermost loop (for (int c = 0; c < Channel; ++c)), I instructed the compiler to unroll this loop. This directive allows the compiler to duplicate the loop body multiple times, reducing the loop control overhead and potentially enabling better optimization by the GPU.

2. Matrix Multiplication Kernel (matrixMultiplyShared)

Similarly, in the matrixMultiplyShared kernel, I unrolled the inner loop responsible for the dot product computation between tiles of matrices A and B.

Modified with Loop Unrolling:

```
#pragma unroll
for (int i = 0; i < TILE_WIDTH; i++) {
    val += tileA[ty][i] * tileB[i][tx];
}
```

Here, the #pragma unroll directive encourages the compiler to unroll the loop that accumulates the products of corresponding elements from tileA and tileB. This can lead to more efficient use of GPU registers and faster computation by reducing loop control instructions.

Justification of Correctness and Performance Using Profiling Results:

Kernel Execution Time (gpukernsum Report):

[6/8] Executing 'gpukernsum' stats report

Time (%)	Total Time (ns)	Instances	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	GridXYZ	BlockXYZ	Name
63.8	506025736	10	50602573.6	51045011.5	47143205	51207587	1234476.8	72250 13 1	16 16 1	matrix_unrolling_kernel(const float *, float *, int, int, int, int, int)
27.8	220648734	10	22064873.4	22215468.5	21174669	22419821	361965.3	400000 4 1	16 16 1	matrix_unrolling_kernel(const float *, float *, int, int, int, int, int)
3.9	31163152	10	3116315.2	3121646.0	3015775	3148958	36987.1	72250 1 1	16 16 1	matrixMultiplyShared(const float *, const float *, float *, int, int, int, int, int, int)
3.7	29222533	10	2922253.3	2930269.5	2864701	2943229	22090.3	400000 1 1	16 16 1	matrixMultiplyShared(const float *, const float *, float *, int, int, int, int, int, int)
0.5	3584796	10	358479.6	358479.5	357247	359072	534.8	25 1000 1	256 1 1	matrix_permute_kernel(const float *, float *, int, int, int)
0.4	2842046	10	284204.6	284207.0	282400	285184	783.4	5 1000 1	256 1 1	matrix_permute_kernel(const float *, float *, int, int, int)

- The significant execution time allocated to matrix_unrolling_kernel and matrixMultiplyShared indicates that these kernels are the primary computational components. With loop unrolling, these kernels execute more efficiently, as evidenced by their high instance counts and reduced average execution times per instance.
- Loop unrolling facilitates better memory access patterns, which is crucial for GPU performance. The profiling results show that memory operations (cudaMemcpy) are optimized, consuming a smaller percentage of the total time.

CUDA API Summary (cudaapisum Report):

[5/8] Executing 'cudaapisum' stats report

Time (%)	Total Time (ns)	Num Calls	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	Name
63.3	812160131	12	67680010.9	1987156.5	125736	543610822	166487427.5	cudaFree
23.8	305618625	8	38202328.1	9766381.0	21050	152604063	59470314.8	cudaMemcpy
12.8	164461010	12	13705084.2	193823.5	117209	161874846	46661636.2	cudaMalloc
0.0	548867	64	8576.0	3416.5	3246	118081	18248.3	cudaLaunchKernel
0.0	36027	6	6004.5	5545.0	4088	9087	1686.7	cudaDeviceSynchronize
0.0	1232	1	1232.0	1232.0	1232	1232	0.0	cuModuleGetLoadingMode

- The relatively low average time per cudaLaunchKernel call (8,576.0 ns) suggests that the loop-unrolled kernels are launched efficiently, without significant overhead.

Memory Operations (gpumemtimesum Report):

[7/8] Executing 'gpumemtimesum' stats report

Time (%) Total Time (ns) Count Avg (ns) Med (ns) Min (ns) Max (ns) StdDev (ns) Operation

-------- --------------- ----- ----------- ----------- --------- --------- ----------- ------------------

86.2 261824863 2 130912431.5 130912431.5 110015427 151809436 29552827.2 [CUDA memcpy DtoH]

13.8 41878530 6 6979755.0 1920.0 1504 22637644 10863763.8 [CUDA memcpy HtoD]

- The data transfer times between host and device are balanced and do not dominate the total execution time, indicating that the computational kernels (with loop unrolling) are effectively utilized.

Memory Usage (gpumemsizesum Report):

[8/8] Executing 'gpumemsizesum' stats report

Total (MB) Count Avg (MB) Med (MB) Min (MB) Max (MB) StdDev (MB) Operation

---------- ----- -------- -------- -------- -------- ----------- ------------------

1763.840 2 881.920 881.920 739.840 1024.000 200.931 [CUDA memcpy DtoH]

551.853 6 91.976 0.007 0.000 295.840 143.039 [CUDA memcpy HtoD]

- The memory operations show a reasonable footprint, ensuring that the loop unrolling did not inadvertently increase memory usage, which could have negatively impacted performance.

c. <u>Did the performance match your expectations? Analyze the profiling results as a scientist.</u>

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*5.68157ms*	*7.62199ms*	*1.402s*	*0.86*

1000	*27.5996ms*	*52.9269ms*	*9.500s*	*0.886*
10000	*257.019ms*	*546.616ms*	*92.953s*	*0.8714*

Performance Overview:

Batch Size	Baseline (Total Execution Time)	With Optimization (Total Execution Time)	Speedup
100	*4.016s*	*1.402s*	*~2.8x*
1000	*43.717s*	*9.500s*	*~4.6x*
10000	*363.963s*	*92.953s*	*~3.9x*

Quantitative Analysis

- Batch Size 100:
    - Speedup: The optimization executes approximately 2.8 times faster than the original.
- Batch Size 1,000:
    - Speedup: The optimization achieves a 4.6× speedup over the original.
- Batch Size 10,000:
    - Speedup: The optimization demonstrates a 3.9× speedup compared to the original.

Batch Size	Baseline (total Op times)	With Optimization (Total Op Times)
100	*7.17506ms*	*13.30356ms*
1000	*18.14643ms*	*80.5265ms*
10000	*128.6454ms*	*803.635ms*

Interpretation: While individual operation times have increased, the proportion of these operations within the total execution time has decreased. This suggests that

although each operation takes longer, the overall parallelism and efficiency gains outweigh the increased per-operation costs.

This increase is primarily due to the additional instructions generated by the loop unrolling process. Unrolling expands the loop body, leading to more operations being executed within each kernel call.

Profiling Analysis:

- The gpukernsum section shows that kernels like matrix_unrolling_kernel and matrixMultiplyShared have higher total times post-optimization. However, these kernels are now more efficient in handling larger portions of the data due to unrolling.
- The cudaapisum and gpumemtimesum sections indicate that memory operations remain efficient and are not bottlenecks, allowing the computational optimizations to shine.

d. <u>Does this optimization synergize with any other optimizations? How?</u>

I didn't apply other optimizations in this code. However, here is what I think of synergizing other optimizations:

1. Using Tensor Cores and Shared Memory Tiling to Speed Up Matrix Multiplication (req_1)

- Enhanced Parallelism: Loop unrolling increases instruction-level parallelism by reducing loop control overhead, allowing more arithmetic operations to be executed concurrently. Tensor Cores, specifically designed for high-throughput matrix operations, can further exploit this parallelism by performing mixed-precision matrix multiplications at a much faster rate.
- Efficient Memory Access: Shared Memory Tiling optimizes memory access patterns by loading sub-matrices into shared memory, reducing global memory latency. When combined with loop unrolling, the unrolled loops can process these tiles more efficiently, ensuring that Tensor Cores receive data at a rate that keeps them fully utilized.

Profiling Insights:

From the gpukernsum section in profile.out, it's evident that the matrixMultiplyShared kernel is a significant contributor to the total operation time.

By integrating Tensor Cores:

- Reduced Execution Time: Tensor Cores can perform matrix multiplications faster than traditional CUDA cores, potentially decreasing the time spent in matrixMultiplyShared.
- Maintained Accuracy: While Tensor Cores utilize mixed-precision arithmetic, careful implementation ensures that the model's accuracy remains unaffected, as seen in the consistent accuracy metrics across optimizations.

2. Using Streams to Overlap Computation with Data Transfer (req_0)

- Concurrency: Loop unrolling optimizes kernel execution by maximizing computational throughput. By employing CUDA Streams, data transfers (e.g., input data to the GPU and output data back to the host) can be overlapped with these optimized kernel executions.
- Minimized Idle Time: While loop unrolling ensures that the GPU cores are efficiently utilized during computation, Streams ensure that the GPU doesn't remain idle waiting for data transfers to complete. This overlap reduces the overall execution time by parallelizing data movement with computation.

Profiling Insights:

Analyzing the gpumemtimesum section:

- Data Transfer Overhead: The baseline shows substantial time spent in [CUDA memcpy DtoH] and [CUDA memcpy HtoD]. With loop unrolling reducing the total execution time, overlapping these transfers with computation can further decrease the Total Execution Time by hiding the latency of data movements.
- Balanced Workload: The optimized kernels execute faster, freeing up GPU resources sooner. Streams can then initiate subsequent data transfers without waiting for the previous ones to complete, leading to a more balanced and efficient workload distribution.

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

https://www.nvidia.com/docs/IO/116711/sc11-unrolling-parallel-loops.pdf This presentation by Vasily Volkov from UC Berkeley offered detailed examples and performance analyses related to loop unrolling in CUDA.

https://forums.developer.nvidia.com/t/automatic-loop-unrolling/10502 Discussions on automatic loop unrolling and the use of #pragma unroll in CUDA provided practical insights.

https://www.baeldung.com/cs/loop-unrolling-performance-improvement This article provided a comprehensive overview of loop unrolling techniques and their impact on performance

7. **Op_4: cuBLAS**

   https://drive.google.com/drive/folders/14KpVOKwb9R52bSf7e3-c9E5TWaBz0xdz?usp=sharing

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

      Integrating cuBLAS optimizes the convolution kernel by leveraging a GPU-accelerated library for efficient matrix multiplication, unlike the manual approach in original convolution kernel. This reduces execution time, improves GPU utilization, scales better with large workloads, and simplifies the code. It delivers superior performance, particularly for deep learning tasks with large filters and batch sizes.

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

Implementation Details

Using cuBLAS for Matrix Multiplication

I utilized cuBLAS's cublasSgemm function, which is optimized for single-precision floating-point operations, to carry out this multiplication efficiently on the GPU.

Code Snippet:

```
__host__ void GPUInterface::conv_forward_gpu(float *device_output, const float *device_input, const float *device_mask,
                                             const int Batch, const int Map_out, const int Channel,
                                             const int Height, const int Width, const int K) {
    const int Height_out = Height - K + 1;
    const int Width_out = Width - K + 1;
    const int Height_unrolled = Channel * K * K;

    // Determine the number of mini-batches
```

```
    int num_batches = (Batch + MAX_BATCH_SIZE - 1) / MAX_BATCH_SIZE;

    float *unrolled_matrix;  // Pointer to device memory for storing the unrolled matrix
    float *matmul_output;    // Pointer to device memory for storing the result of matrix multiplication

    // Allocate device memory for unrolled_matrix and matmul_output for the maximum mini-batch size
    size_t max_unroll_size = Height_unrolled * (MAX_BATCH_SIZE * Height_out * Width_out) * sizeof(float);
    cudaMalloc((void**)&unrolled_matrix, max_unroll_size);

    size_t max_matmul_size = Map_out * (MAX_BATCH_SIZE * Height_out * Width_out) * sizeof(float);
    cudaMalloc((void**)&matmul_output, max_matmul_size);

    // Iterate over each mini-batch
    for(int batch_idx = 0; batch_idx < num_batches; ++batch_idx) {
        int current_batch_size = (batch_idx == num_batches - 1) ?
                                 (Batch - batch_idx * MAX_BATCH_SIZE) : MAX_BATCH_SIZE;

        int current_W_unroll = current_batch_size * Height_out * Width_out;

        // Set the kernel dimensions for unrolling using a 2D grid
```

```
    dim3 blockDim_unroll(16, 16);
    dim3 gridDim_unroll((current_W_unroll + blockDim_unroll.x - 1) / blockDim_unroll.x,
                        (Height_unrolled + blockDim_unroll.y - 1) / blockDim_unroll.y);

    // Call the matrix unrolling kernel for the current mini-batch
    matrix_unrolling_kernel<<<gridDim_unroll, blockDim_unroll>>>(
        device_input + batch_idx * MAX_BATCH_SIZE * Channel * Height * Width, // Offset
input pointer
        unrolled_matrix,
        current_batch_size,
        Channel,
        Height,
        Width,
        K
    );

    cudaError_t error = cudaGetLastError();
    if(error != cudaSuccess) {
        std::cout<<"CUDA error (unrolling kernel): "<<cudaGetErrorString(error)<<std::endl;
        exit(-1);
    }

    // Initialize cuBLAS handle
```

```
cublasHandle_t cublasHandle;
cublasCreate(&cublasHandle);

// Set alpha and beta for the cuBLAS operation
const float alpha = 1.0f;
const float beta = 0.0f;

// Perform the matrix multiplication using cuBLAS
cublasStatus_t status = cublasSgemm(
    cublasHandle,
    CUBLAS_OP_N,  // Operation on A: No transpose
    CUBLAS_OP_N,  // Operation on B: No transpose
    current_W_unroll,
    Map_out,
    Height_unrolled,
    &alpha,
    unrolled_matrix, current_W_unroll,
    device_mask, Height_unrolled,
    &beta,
    matmul_output, current_W_unroll
);

if (status != CUBLAS_STATUS_SUCCESS) {
```

```
            std::cerr << "cuBLAS sgemm failed" << std::endl;
            exit(1);
        }

        // Destroy cuBLAS handle
        cublasDestroy(cublasHandle);

        // Permute the result of matrix multiplication
        const int out_image_size = Height_out * Width_out;
        dim3 permute_kernel_grid_dim((out_image_size - 1) / BLOCK_SIZE + 1,
current_batch_size, 1);
        matrix_permute_kernel<<<permute_kernel_grid_dim, BLOCK_SIZE>>>(
            matmul_output,
            device_output + batch_idx * MAX_BATCH_SIZE * Map_out * out_image_size, //
Offset output pointer
            Map_out,
            current_batch_size,
            out_image_size
        );

        // Check for errors after permutation
        error = cudaGetLastError();
        if(error != cudaSuccess) {
```

```
            std::cout<<"CUDA error (permute kernel): "<<cudaGetErrorString(error)<<std::endl;
            exit(-1);
        }
    }

    cudaFree(matmul_output);
    cudaFree(unrolled_matrix);
}
```

Explanation:

1. cuBLAS Handle Initialization:
    - Before performing any cuBLAS operations, I initialize a cuBLAS handle using cublasCreate. This handle is essential for managing the cuBLAS context.
2. Setting Scalars alpha and beta:
    - These scalars are used in the matrix multiplication operation. Here, alpha is set to 1.0f and beta to 0.0f, meaning the operation performed is essentially C = A * B.
3. Matrix Multiplication with cublasSgemm:
    - The cublasSgemm function performs the matrix multiplication. The parameters specify that neither matrix A (unrolled_matrix) nor matrix B (device_mask) should be transposed (CUBLAS_OP_N).
    - The dimensions are set based on the current mini-batch size and the convolution parameters.
    - The result is stored in matmul_output.
4. Error Handling:

- After the cublasSgemm call, I check the status to ensure the operation was successful. If not, the program exits with an error message.

5. cuBLAS Handle Destruction:
    - After the matrix multiplication, the cuBLAS handle is destroyed using cublasDestroy to free resources.

Justification of Correctness and Performance Using Profiling Results:

Kernel Execution Times:

[6/8] Executing 'gpukernsum' stats report

Time (%)	Total Time (ns)	Instances	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	GridXYZ	BlockXYZ	Name
34.8	38493310	10	3849331.0	3848815.5	3848400	3850704	967.0	4000004	1 16 16	1 matrix_unrolling_kernel(const float *, float *, int, int, int, int, int)
22.6	25019602	10	2501960.2	2501880.5	2501416	2502600	456.2	72250 13	1 16 16	1 matrix_unrolling_kernel(const float *, float *, int, int, int, int, int)
20.8	22981668	60	383027.8	383185.5	380450	387649	1476.1	16384 1 1	128 1 1	void gemmSN_NN_kernel<float, (int)128, (int)2, (int)4, (int)8, (int)4, (int)4, (bool)0, cublasGemvT…
13.9	15400754	10	1540075.4	1539429.0	1537733	1543845	2215.1	8 1024 1	128 1 1	void cutlass::Kernel<cutlass_80_simt_sgemm_128x32_8x5_nn_align1>(T1::Params)
3.4	3722830	10	372283.0	372625.5	370657	373281	914.0	25 1000 1	256 1 1	matrix_permute_kernel(const float *, float *, int, int, int)
2.7	2988394	10	298839.4	298945.0	297281	301345	1182.2	5 1000 1	256 1 1	matrix_permute_kernel(const float *, float *, int, int, int)

1.5 1670821 10 167082.1 167136.0 166401 167872 420.3 840 1 1 256 1 1 ampere_sgemm_128x32_nn

0.4 451203 10 45120.3 45072.5 44864 45472 195.4 1699 1 1 128 1 1 void gemmSN_NN_kernel<float, (int)128, (int)2, (int)4, (int)8, (int)4, (int)4, (bool)0, cublasGemvT…

- The matrix_unrolling_kernel and matrix_permute_kernel are efficiently launched with appropriate grid and block dimensions. The profiling shows multiple instances of these kernels with reasonable execution times, indicating that they are correctly handling the data unrolling and permutation tasks.
- The cublasSgemm calls are the primary consumers of GPU time, as expected for matrix multiplication operations. The profiling output indicates that these operations are correctly invoked and consume a significant portion of the total computation time, which aligns with the nature of convolution operations.

Memory Operations:

[5/8] Executing 'cudaapisum' stats report

Time (%) Total Time (ns) Num Calls Avg (ns) Med (ns) Min (ns) Max (ns) StdDev (ns) Name

-------- --------------- --------- ---------- --------- -------- --------- ----------- ----------------------

50.3 298131243 8 37266405.4 9980160.5 18415 147577345 57664550.9 cudaMemcpy

28.5 169053354 72 2347963.3 6437.0 1964 158818405 18705252.7 cudaMalloc

15.8 93602979 86 1088406.7 2109.0 892 6314990 2130384.4 cudaDeviceSynchronize

4.7 27671479 93 297542.8 4157.0 481 9637890 1107309.8 cudaFree

0.4 2300096 11 209099.6 752.0 631 2291830 690763.6 cudaOccupancyMaxActiveBlocksPerMultiprocessor

0.2 1110793 134 8289.5 4248.0 3397 119183 12307.7 cudaLaunchKernel

0.0 233919 360 649.8 511.0 361 8917 658.1 cudaEventCreateWithFlags

0.0 231000 768 300.8 270.0 120 9077 338.5 cuGetProcAddress

0.0 169592 360 471.1 411.0 330 3677 252.1 cudaEventDestroy

0.0 2695 2 1347.5 1347.5 1062 1633 403.8 cuInit

0.0 1922 3 640.7 300.0 180 1442 696.6 cuModuleGetLoadingMode

- The profiling shows substantial time spent on cudaMemcpy and cudaMalloc operations. While these are necessary for data transfers between host and device, their impact suggests that minimizing data transfers or overlapping them with computation could further enhance performance.

c. <u>Did the performance match your expectations? Analyze the profiling results as a scientist.</u>

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*21.7908ms*	*3.42822ms*	*1.476s*	*0.86*
1000	*41.923ms*	*17.1134ms*	*11.125s*	*0.886*
10000	*85.4264ms*	*50.9602ms*	*100.343s*	*0.8714*

Performance Overview:

Batch Size	Baseline (Total Execution Time)	With Optimization (Total Execution Time)	Speedup
100	*4.016s*	*1.476s*	*~2.7x*

1000	*43.717s*	*11.125s*	*~3.9x*
10000	*363.963s*	*100.343s*	*~3.6x*

Quantitative Analysis

- Batch Size 100:
  - Speedup: The optimization executes approximately 2.7 times faster than the original.
- Batch Size 1,000:
  - Speedup: The optimization achieves a 3.9× speedup over the original.
- Batch Size 10,000:
  - Speedup: The optimization demonstrates a 3.6× speedup compared to the original.

Batch Size	Baseline (total Op times)	With Optimization (Total Op Times)
100	*7.17506ms*	*25.21902ms*
1000	*18.14643ms*	*59.0364ms*
10000	*128.6454ms*	*136.3866ms*

Profiling Analysis

Increased Number of Operations:

- cuBLAS Integration: When using cuBLAS for matrix multiplication (cublasSgemm), the library internally performs multiple optimized operations to achieve high performance. This includes launching several CUDA kernels, managing memory efficiently, and handling parallel computations.
- Profiling Insight: The profiling report shows a significant number of cudaMalloc and cudaMemcpy calls when cuBLAS is utilized. These additional memory allocations and data transfers contribute to the higher total Op times.

  Time (%) Total Time (ns) Num Calls   Avg (ns)   ...

28.5	169053354	72	2347963.3	...	cudaMalloc
50.3	298131243	8	37266405.4	...	cudaMemcpy

Optimized but Complex Operations:

- Efficient Computations: cuBLAS operations like cublasSgemm are highly optimized for performance, leveraging the GPU's full potential. These operations are more complex and involve sophisticated algorithms that perform multiple lower-level tasks.
- Profiling Insight: The gpukernsum section of the profiling report indicates that a substantial portion of GPU time is spent on matrix_unrolling_kernel and cublasSgemm:

  [6/8] Executing 'gpukernsum' stats report

Time (%)	Total Time (ns)	Instances	...
34.8	38493310	10	matrix_unrolling_kernel(...)
20.8	22981668	60	gemmSN_NN_kernel...

  These optimized kernels perform more operations internally, which increases the total Op times recorded.

Overhead from cuBLAS Initialization and Management:

- Handle Management: Each cuBLAS operation involves initializing and destroying handles (cublasCreate and cublasDestroy), which adds to the total number of operations.
- Profiling Insight: Although not explicitly detailed in the provided profile.out, the increased number of CUDA API calls like cublasSgemm contribute to the higher Op times.

d. <u>Does this optimization synergize with any other optimizations? How?</u>

I didn't apply other optimizations in this code. However, here is what I think of synergizing other optimizations:

1. Using Streams to Overlap Computation with Data Transfer (req_0)
   - Profiling Insight: The profiling data indicates significant time spent on cudaMemcpy operations (e.g., 298131243 ns for cudaMemcpy DtoH and 169053354 ns for cudaMalloc when using cuBLAS). These memory transfers can become bottlenecks, especially with large batch sizes.

- Optimization Benefit: By utilizing CUDA streams, I can overlap data transfers with computation. This means while cuBLAS is performing matrix multiplication (cublasSgemm), data can simultaneously be copied between the host and device without waiting for one operation to complete before starting the other.
- Resulting Synergy: Overlapping these operations effectively hides the latency of data transfers, reducing idle GPU time and enhancing overall throughput. This leads to a more efficient pipeline where the GPU remains active, maximizing resource utilization.

2. Fixed Point (FP16) Arithmetic Implementation (op_5)

- Profiling Insight: The profiling data shows that a considerable amount of time is dedicated to memory operations (cudaMemcpy, cudaMalloc) and matrix computations (cublasSgemm). Reducing the precision of computations and memory usage can alleviate these bottlenecks.
- Optimization Benefit: Implementing FP16 (half-precision) arithmetic reduces the memory footprint of both the input data and the convolutional masks. This leads to lower memory bandwidth usage and faster data transfers. Additionally, many modern GPUs, especially those with Tensor Cores, are optimized for FP16 operations, allowing for accelerated matrix multiplications.
- Resulting Synergy: Combining FP16 with cuBLAS leverages the library's optimized routines for half-precision arithmetic, resulting in faster computations and reduced memory bandwidth consumption. This not only speeds up cublasSgemm but also decreases the time spent on cudaMemcpy operations due to smaller data sizes.

e. <u>List your references used while implementing this technique. (you must mention textbook pages at the minimum)</u>

Here is what I used for reference when I work on the optimization:

https://docs.nvidia.com/cuda/cublas/index.html This comprehensive guide details the cuBLAS API, data layouts, and provides example codes for various linear algebra operations.

https://developer.download.nvidia.com/compute/DevZone/docs/html/CUDALibraries/doc/CUBLAS_Library.pdf Specifically, Chapter 3 of the CUDA Programming Guide discusses matrix multiplication and the use of cuBLAS for optimized performance.

https://mccormickml.com/2015/08/29/matrix-multiplication-with-cublas-example/ This article offers a practical example of using cuBLAS for matrix multiplication, including code snippets and explanations.