Omkar Gurjar (ogurjar2)

Milestone 3 Report

Contents

0. Baseline:

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*1.3747 ms*	*0.483686 ms*	*0m1.459s*	*0.86*
1000	*2.36643 ms*	*4.56208 ms*	*0m9.300s*	*0.886*
10000	*13.0767 ms*	*45.3058 ms*	*1m30.229s*	*0.8714*

1. Optimization Number #: 0 Optimization Name: Streams

a. How does this optimization work in theory? Expected behavior?

Cuda streams functionality allows improved concurrency by executing the api calls through asynchronous queues. Transferring the whole data from host to device, then executing the kernel and then copying the data back only keeps one part of the GPU busy at a time. Instead, we can have multiple queues or Streams so that the data transfer and kernel execution can happen in parallel.

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

As explained in the previous section, the main idea is to divide our data into multiple batches so that they can be processed in an overlapping manner by different streams. First, we declared two hyper-parameters to decide the number of streams and size of each batch of data. Consequently, we calculated the number of batches each stream needs to process.

```
const int num_streams = 4;
int mini_batch_size = 250;
int mini_batch_per_stream = ceil(Batch*1.0/num_streams*mini_batch_size);
```

Next, we initialize an array of streams. We also pin the host memory to ensure fast access using cudaHostRegister() function.

```
cudaStream_t streams[num_streams];
for(int i=0; i<num_streams; i++)
  cudaStreamCreate(&streams[i]);
```

Finally, we write nested loops to iterate through all the batches for each stream. We keep track of an index variable to track the batches which have already been queued. We use the strategy 1 from the slides (queueing the H2D, kernel and D2H for each stream) as it gives us the required overlap as shown in the next section. Note that all the code was moved to the prolog function to get access to the host data pointers.

```
int start_idx = 0;
for(int mini_batch_idx=0; mini_batch_idx < mini_batch_per_stream; mini_batch_idx++)
{
  for(int stream_idx=0; stream_idx < num_streams; stream_idx++)
  {
    if(start_idx>=Batch)
      break;
    // printf("%d %d\n", start_idx, stream_idx);
    int mem_start_input = start_idx*Channel*Height*Width;
    int mem_start_output = start_idx*Map_out*Height_out*Width_out;
    int copy_size = min(mini_batch_size, Batch-start_idx);
    int mini_batch_in_size = copy_size*Channel*Height*Width;
    int mini_batch_out_size = copy_size*Map_out*Height_out*Width_out;

    cudaMemcpyAsync(*device_input_ptr+mem_start_input, &host_input[mem_start_input], mini_batch_in_size*sizeof(float), cudaMemc
    conv_forward_kernel<<<gridDim, blockDim, 0, streams[stream_idx]>>>(*device_output_ptr, *device_input_ptr, *device_mask_ptr,
    cudaMemcpyAsync(&host_output_temp[mem_start_output], *device_output_ptr+mem_start_output, mini_batch_out_size*sizeof(float)
    start_idx += mini_batch_size;
  }
}
```

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

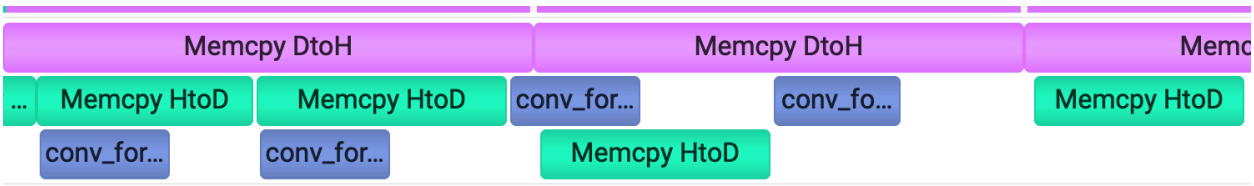
In this optimization, the OP times are irrelevant as we have moved the whole code to the prolog function. We report the total execution times for each batch size.

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	-	-	*0m6.527s*	*0.86*
1000	-	-	*0m10.419s*	*0.886*
10000	-	-	*1m32.069s*	*0.8714*

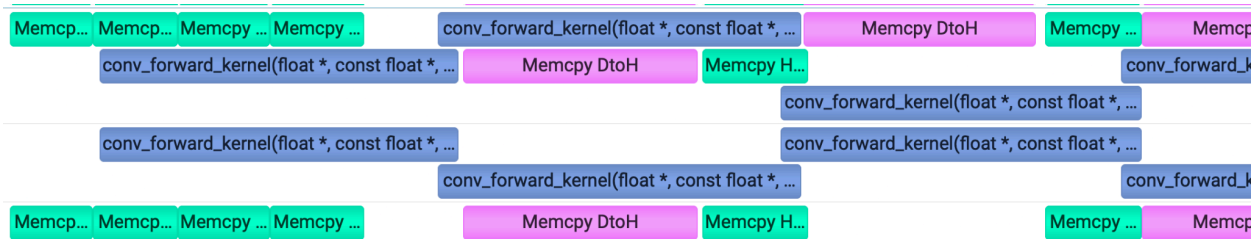
We observe a slight increase in the total execution times as compared to our baselines, but the difference isn't significant. Overall, in terms of execution time this optimization achieves similar performance.

Next, we look at the overlap between different streams by analyzing the nsys-rep files using Nvidia Nsight Systems. The results are shown for the first and second convolution layers respectively.

Layer 1:



Layer 2:



- We observe an overlap between the memory copies (in both directions) and the kernel execution for both the layers, showing the improved concurrency.

- The time taken by DtoH memory transfer is higher due to the larger size of output as compared to the inputs.
- We also observe that cudaHostRegister and cudaHostUnreguster takes up a lot of time which might induce overhead nullifying any speed improvements.

Time (%)	Total Time (ns)	Num Calls	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	Name
40.8	195,791,451	4	48,947,862.8	42,068,310.5	8,303,070	103,351,760	48,113,273.6	cudaHostRegister
32.4	155,210,923	8	19,401,365.4	130,906.0	7,414	154,451,557	54,568,544.9	cudaMalloc
26.2	125,769,552	4	31,442,388.0	32,199,985.5	11,522,315	49,847,266	20,707,285.8	cudaHostUnregister

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

This optimization can be paired with OP4 by reading the kernel mask into constant memory which can improve efficiency due to reduced global memory accesses. We can also unroll the convolution loop in the kernel using OP5 which can prevent some overhead. Additionally, fp16 arithmetic can be utilized like OP10 to reduce memory usage.

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

- https://developer.nvidia.com/nsight-systems
- https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__MEMORY.html#group__CUDART__MEMORY_1ge8d5c17670f16ac4fc8fcb4181cb490c
- Chapter 13 “CUDA dynamic parallelism”. D. Kirk and W. Hwu, “Programming Massively Parallel Processors – A Hands-on Approach,” Morgan Kaufman Publisher, 3rd edition, 2016

2. Optimization Number #: 1 Optimization Name Shared Memory Convolution

a. How does this optimization work in theory? Expected behavior?

In the naive implementation of the convolution kernel from Milestone 2, each thread reads all the elements required to compute its output value directly from the global memory. This leads to a bandwidth issue and a memory bottleneck. To solve this, we leverage shared memory which is a type of GPU memory shared between all the threads in a block and has much faster access time compared to global memory. The idea is that threads in a block share the loading of data from the global memory into a shared memory array called tile which is then shared by other threads in the block preventing repeated memory accesses.

Owning to the data reuse, the optimization is expected to improve the GPU throughput leading to faster run time for the kernel. However, the benefits may not be very significant if the data reuse isn't large within a block.

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

To implement this, we opt for strategy 2 discussed in the slides. In this, each block has one thread corresponding to each input matrix element. Since our output matrix is smaller in size than the input, we turn off the extra threads while performing the computation and data writes. The first change we make is to change the block dimension to include the extra elements read from the input matrix which are required for the convolution.

Since, we have different input and output sizes for the first and second convolution layer, we utilize the **extern** keyword to declare a dynamically sized shared memory. We set the amount of shared memory needed during the kernel launch.

```
int radius = (K-1)/2;
int block_size = TILE_WIDTH + 2*radius;
int shared_memory_size = (Channel*block_size*block_size)*sizeof(float);
dim3 gridDim(grid_x, grid_y, grid_z);
dim3 blockDim(block_size, block_size, 1);
conv_forward_kernel<<<gridDim, blockDim, shared_memory_size>>>(device_output,
```

Next, inside the convolution kernel, for each thread, we read the corresponding element from the input matrix for each channel into the shared memory.

```
extern __shared__  float tile[];
int tile_size = TILE_WIDTH + 2*(K-1)/2;
bool valid_thread = (x_out<Width) && (y_out < Height);
for(int c=0; c<Channel; c++)
{
    if(valid_thread)
        tile[c*(tile_size*tile_size) + ty*tile_size + tx] = in_4d(batch, c, y_out, x_out);
    else
        tile[c*(tile_size*tile_size) + ty*tile_size + tx] = 0.0f;
}
__syncthreads();
```

We also include basic boundary check conditions to avoid unauthorized memory accesses and we turn off the extra threads which are beyond the output dimensions for each block

```
if (tx>=TILE_WIDTH||ty>=TILE_WIDTH)
    return;
if(x_out>=Width_out || y_out >= Height_out)
    return;
```

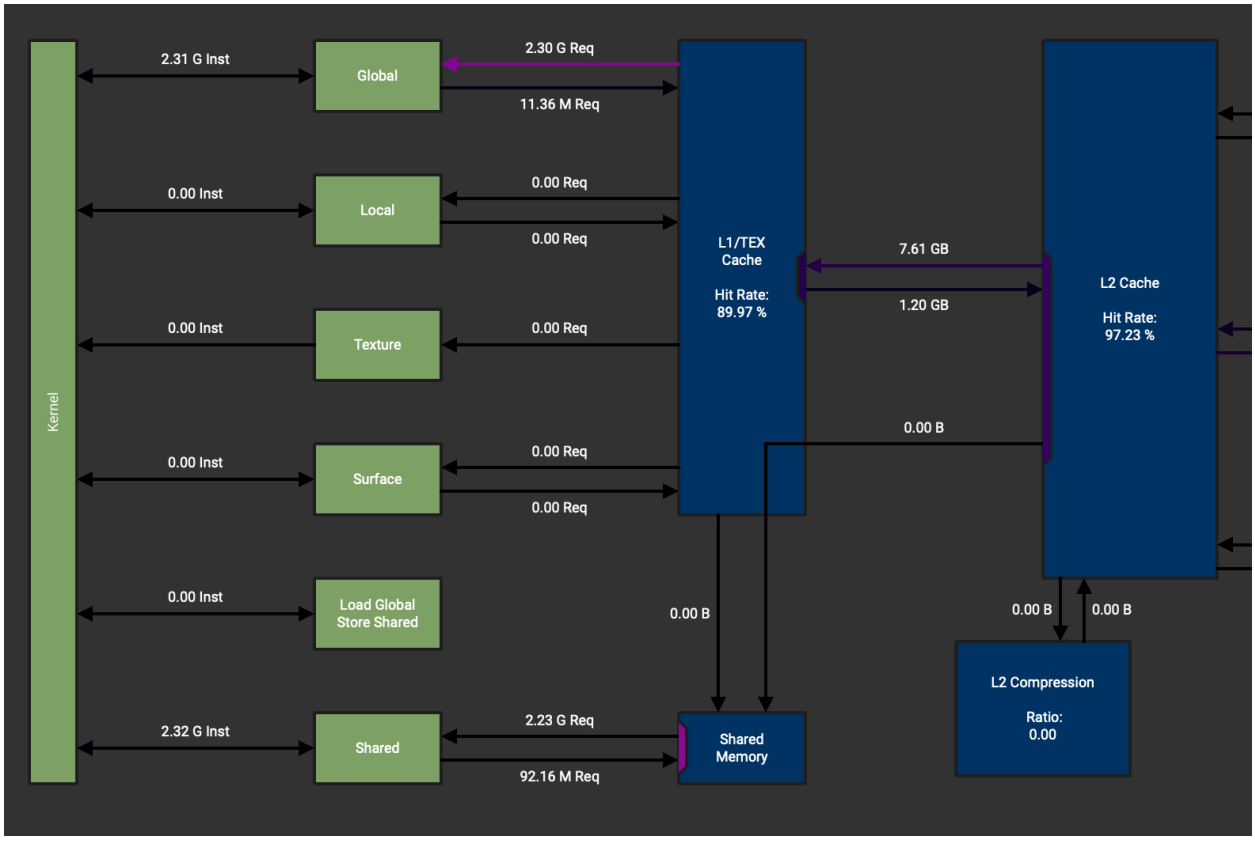
Finally, we simply replace the input memory access with shared memory access in the main convolution loop.

```
result += tile[c*(tile_size*tile_size) + (ty+h)*tile_size + tx+w]*mask_4d(m, c, h, w);
```

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

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.259508 ms*	*0.719621 ms*	*0m2.554s*	*0.86*
1000	*1.82123 ms*	*6.97226 ms*	*0m9.753s*	*0.886*
10000	*17.3458 ms*	*69.3812 ms*	*1m32.600s*	*0.8714*

- Total Op Time: 86.72 ms (Baseline: 58.3 ms)
- The optimization increased the OP times for both the layers for the batch size 10000, while it did improve OP Time 1 marginally for batch sizes 100, 1000. Thus, overall it does not lead to expected results.
- First, we validate that shared memory is actually being utilized by the Kernel by looking at the memory chart. We see that around 2.32G of data is loaded into the shared memory.



- We also see that the memory throughput has decreased as compared to the baseline implementation.

OP 1:

Memory Throughput [Gbyte/second]	13.80
L1/TEX Hit Rate [%]	89.97
L2 Hit Rate [%]	97.23
L2 Compression Success Rate [%]	0

Baseline:

Memory Throughput [Gbyte/second]	21.37
L1/TEX Hit Rate [%]	97.11
L2 Hit Rate [%]	97.09
L2 Compression Success Rate [%]	0

- As our **kernel size is small** (7x7) there isn't a very high potential for data reuse and given that declaring the dynamic shared memory has its own overhead, the overall optimization isn't effective. This leads to an increase in OP times. With a larger kernel size, the results might improve.
- Another thing to note is that the L1, L2 cache hit rate is very high which might be the reason why shared memory doesn't improve performance as the data is being read from caches most of the time.

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

Yes, this optimization can be used in combination with OP 4 as loading the kernel into the constant memory can further decrease the memory requirements for the kernel. Also, the same concept of using tiles is used in OP2, OP3 to implement the tiled matrix multiplication.

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

- D. Kirk and W. Hwu, "Programming Massively Parallel Processors – A Hands-on Approach," Morgan Kaufman Publisher, 3rd edition, 2016. Chapter 7, Pages 19-24.
- https://developer.nvidia.com/blog/using-shared-memory-cuda-cc/ – Using Dynamic Shared Memory.

3. Optimization Number #: 2 Optimization Name Input Matrix Unrolling & Tiled Matrix Multiplication

a. How does this optimization work in theory? Expected behavior?

The optimization aims to convert the convolution operation into a matrix multiplication. The motivation is that the generalized matrix matrix multiplication (GEMM) is a highly optimized operation due to its widespread use in various algorithms. This is possible since both convolution and gemm perform dot products to get output elements. To achieve this we first unroll the input matrix such that all the elements required to compute a given output element are stored in each column sequentially (in row wise order), following which we multiply the kernel matrix with the unrolled input matrix to generate our output.

Although GEMM is optimized, unrolling the input matrix is expected to be the bottleneck step as it requires additional computation, kernel invocation and more memory requirements due to repeating in the unrolled matrix.

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

The implementation for this optimization involved writing two separate kernels for unrolling the input matrix and multiplying kernel matrix to the unrolled input.

For the unrolled kernel. we launch a grid configuration such that each thread maps to a single output element and unrolls the elements required to compute its corresponding output element. We set the grid x dimension to be the number of input channels so each thread processes a single channel.

```
dim3 gridDim(Channel, width_tiles*height_tiles, small_batch);
```

```
int w_base = c*K*K;
for(int p=0; p<K; p++)
{
  for(int q=0; q<K; q++)
    {
        int h_unroll = w_base + p*K + q;
        int w_unroll = h*Width_out + w;
        device_input_unroll[batch*(Height_unroll*Width_unroll) + h_unroll*Width_unroll + w_unroll] = in_4d(global_batch, c, h+p, w+q);
    }
}
```

For GEMM, we utilize the standard tiled matrix multiplication code described in the textbook.

Now, as the size of unrolled matrix is K*K times larger than the input matrix, in order to limit the memory requirements, we process the batches as multiple smaller batches by looping over them and calling the kernel for just that small-batch.

```
cudaMalloc((void **)&device_input_unroll, unroll_size*sizeof(float));

for(int batch_idx=0; batch_idx < ceil(1.0*Batch/small_batch); batch_idx++)
{
    int batch_start = batch_idx*small_batch;
    unrollKernel<<<gridDim, blockDim>>>(Batch, Channel, Height, Width, K, device_input, device_input_unroll, batch_start);
    matrixMultiplyShared<<<gridDim1, blockDim>>>(device_mask, device_input_unroll, device_output, Height_kernel, Width_kernel, Height_unroll, Widt
}
```

To account for this, we pass an additional parameter to the kernels which keeps track of the batch offset.

```
int batch = blockIdx.z;
int global_batch = batch_start + batch;
```

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

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.94302 ms*	*0.8422 ms*	*1m42.042s*	*0.86*
1000	*7.68069 ms*	*7.15365 ms*	*0m9.550s*	*0.886*
10000	*74.9683 ms*	*70.3473 ms*	*1m31.890s*	*0.8714*

- Total Op Time: 145.3 ms (Baseline: 58.3 ms)
- As expected, this optimization increases both the OP times. Also, we see that the OP1 and OP2 times become almost the same in all the cases, whereas in the baseline OP time 1 was much smaller. As the size of the unrolled image is larger for the second layer, we can note that this technique works better for larger matrix sizes.
- We also observe that the cudaLaunchKernel time has increased significantly compared to the baseline which might have led to an increase in OP times. This is due to the multiple kernel launches due to batch processing.

OP2:

Time (%)	Total Time (ns)	Num Calls	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	Name
49.9	304,345,079	8	38,043,134.9	9,799,359.5	25,347	153,861,716	59,578,787.4	cudaMemcpy
25.0	152,845,399	10	15,284,539.9	150,147.5	79,058	150,996,617	47,684,640.7	cudaMalloc
23.5	143,260,979	6	23,876,829.8	7,875.5	3,767	73,907,252	37,008,146.2	cudaDeviceSynchronize
1.3	8,238,885	8	1,029,860.6	852,926.0	105,408	2,571,537	952,310.9	cudaFree
0.3	1,661,017	404	4,111.4	3,196.0	2,995	124,544	7,786.7	cudaLaunchKernel
0.0	4,569	1	4,569.0	4,569.0	4,569	4,569	0.0	cuModuleGetLoadingMode

Baseline:

Time (%)	Total Time (ns)	Num Calls	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	Name
58.5	311,308,687	8	38,913,585.9	9,721,169.5	36,629	155,348,379	60,947,789.6	cudaMemcpy
29.2	155,084,716	8	19,385,589.5	137,914.0	72,005	153,771,021	54,300,240.4	cudaMalloc
10.7	56,898,749	6	9,483,124.8	7,764.5	3,967	45,269,145	18,134,934.7	cudaDeviceSynchronize
1.6	8,322,200	8	1,040,275.0	471,369.0	96,310	2,574,028	1,069,884.4	cudaFree
0.1	399,329	6	66,554.8	36,288.0	13,636	178,114	67,076.7	cudaLaunchKernel
0.0	4,689	1	4,689.0	4,689.0	4,689	4,689	0.0	cuModuleGetLoadingMode

- We also see that the added time due to the unrolling, which is quite close to the time taken by matrix multiplication. This overhead increases the OP time.

OP 2:

Time (%)	Total Time (ns)	Instances	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	GridXYZ			BlockXYZ			
31.8	45,871,138	100	458,711.4	458,720.0	458,240	459,233	188.0	400	1	100	16	16	1	matrixMultiplyShared(const f
29.5	42,620,613	100	426,206.1	425,937.0	415,809	434,944	4,405.9	4	9	100	16	16	1	unrollKernel(int, int, int,
19.8	28,599,498	100	285,995.0	285,952.0	284,161	292,160	973.3	1	25	100	16	16	1	unrollKernel(int, int, int,
18.9	27,281,326	100	272,813.3	272,800.0	272,001	273,473	309.5	73	1	100	16	16	1	matrixMultiplyShared(const f

Baseline:

Time (%)	Total Time (ns)	Instances	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	GridXYZ			BlockXYZ			
79.6	45,266,735	1	45,266,735.0	45,266,735.0	45,266,735	45,266,735	0.0	16	9	10000	16	16	1	conv_forward_kern
20.4	11,601,896	1	11,601,896.0	11,601,896.0	11,601,896	11,601,896	0.0	4	25	10000	16	16	1	conv_forward_kern
0.0	4,800	2	2,400.0	2,400.0	2,368	2,432	45.3	1	1	1	1	1	1	do_not_remove_thi
0.0	4,704	2	2,352.0	2,352.0	2,304	2,400	67.9	1	1	1	1	1	1	prefn_marker_kern

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

This optimization can be utilized in combination with OP4 by storing the mask matrix in constant memory. Further, we can also use it with OP10 by utilizing the fp16 operations which reduces memory usage, this will allow us to increase our mini-batch size leading to less kernel overhead.

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

- D. Kirk and W. Hwu, “Programming Massively Parallel Processors – A Hands-on Approach,” Morgan Kaufman Publisher, 3rd edition, 2016. Chapter 16, Section 16.4 Pages 16-22.
- D. Kirk and W. Hwu, “Programming Massively Parallel Processors – A Hands-on Approach,” Morgan Kaufman Publisher, 3rd edition, 2016. Chapter 4, Section 4.5 Pages 23-32.

4. **Optimization Number #: 3 Optimization Name: Kernel Fusion for unrolling and matrix multiplication**

    a. **How does this optimization work in theory? Expected behavior?**

Implementing input matrix unrolling and matrix multiplication as separate kernels poses the challenge of memory constraints which forces a mini-batch wise implementation. This also increases the kernel launch overhead due to repeated kernel invocations. We can solve this issue by implementing a single kernel to perform both the operations. Instead of explicitly creating the unroll matrix, we can use the index mapping to read the corresponding input elements directly from the input matrix.

This optimization is expected to improve the runtime compared to OP 2 (separate kernels), however it may have some inefficiencies since the data access pattern from the input matrix isn't coalesced.

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

This can be easily implemented by modifying the code developed during OP2. We modify the tiled matrix multiplication kernel to create a mapping between the theoretical unrolled matrix and the input matrix and use this mapping to load the values into the shared memory.

```
if ((col < numBColumns) && (q*TILE_WIDTH+ty < numBRows))
{
  int h_unroll = (q*TILE_WIDTH+ty); // 6
  int w_unroll = col; // 1
  int channel = h_unroll/(K*K); // 1
  int Width_out = Width - K + 1;
  int h = w_unroll/Width_out; // 0
  int w = w_unroll%Width_out; // 1
  int p = (h_unroll - channel*K*K)/K; // (6-4)/2 = 1
  int _q = (h_unroll - channel*K*K)%K; // (6-4)%2 = 0
  subTileB[ty][tx] = in_4d(global_batch, channel, h+p, w+_q);
}
else
  subTileB[ty][tx] = 0.0f;
```

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

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.54858 ms*	*0.301677 ms*	*0m3.196s*	*0.86*
1000	*4.78558 ms*	*4.78558 ms*	*0m9.549s*	*0.886*
10000	*47.111 ms*	*27.849 ms*	*1m30.168 s*	*0.8714*

- Total Op Time: 74.95 ms (Baseline: 58.3 ms, OP2 (separate kernel): 145.3 ms)
- As expected, we observe an almost 2x speed up compared to the separate kernel implementation.

- The cudaLauchKernel calls and time has been decreased significantly compared to OP2 and is even lesser compared to the baseline.

OP3:

Time (%)	Total Time (ns)	Num Calls	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	Name
55.3	288,541,440	8	36,067,680.0	9,326,380.0	25,498	146,646,450	56,321,751.5	cudaMemcpy
28.7	149,846,186	8	18,730,773.3	140,699.5	82,485	148,559,831	52,459,101.8	cudaMalloc
14.4	74,892,536	6	12,482,089.3	7,569.5	5,660	47,038,241	20,258,942.2	cudaDeviceSynchronize
1.6	8,309,769	8	1,038,721.1	440,542.0	101,510	2,578,591	1,087,515.0	cudaFree
0.0	244,780	6	40,796.7	29,781.0	15,820	80,511	27,604.2	cudaLaunchKernel
0.0	3,707	1	3,707.0	3,707.0	3,707	3,707	0.0	cuModuleGetLoadingMode

OP2:

Time (%)	Total Time (ns)	Num Calls	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	Name
49.9	304,345,079	8	38,043,134.9	9,799,359.5	25,347	153,861,716	59,578,787.4	cudaMemcpy
25.0	152,845,399	10	15,284,539.9	150,147.5	79,058	150,996,617	47,684,640.7	cudaMalloc
23.5	143,260,979	6	23,876,829.8	7,875.5	3,767	73,907,252	37,008,146.2	cudaDeviceSynchronize
1.3	8,238,885	8	1,029,860.6	852,926.0	105,408	2,571,537	952,310.9	cudaFree
0.3	1,661,017	404	4,111.4	3,196.0	2,995	124,544	7,786.7	cudaLaunchKernel
0.0	4,569	1	4,569.0	4,569.0	4,569	4,569	0.0	cuModuleGetLoadingMode

Baseline:

Time (%)	Total Time (ns)	Num Calls	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	Name
58.5	311,308,687	8	38,913,585.9	9,721,169.5	36,629	155,348,379	60,947,789.6	cudaMemcpy
29.2	155,084,716	8	19,385,589.5	137,914.0	72,005	153,771,021	54,300,240.4	cudaMalloc
10.7	56,898,749	6	9,483,124.8	7,764.5	3,967	45,269,145	18,134,934.7	cudaDeviceSynchronize
1.6	8,322,200	8	1,040,275.0	471,369.0	96,310	2,574,028	1,069,884.4	cudaFree
0.1	399,329	6	66,554.8	36,288.0	13,636	178,114	67,076.7	cudaLaunchKernel
0.0	4,689	1	4,689.0	4,689.0	4,689	4,689	0.0	cuModuleGetLoadingMode

- We observe that both OP times are smaller compared to OP2 (separate kernels) as there is no overhead due to the unrolling kernel. More importantly, the OP Time2 is smaller compared to Baseline (27.849 ms vs 45.3ms). This shows that matrix matrix multiplication is effective for larger image sizes.

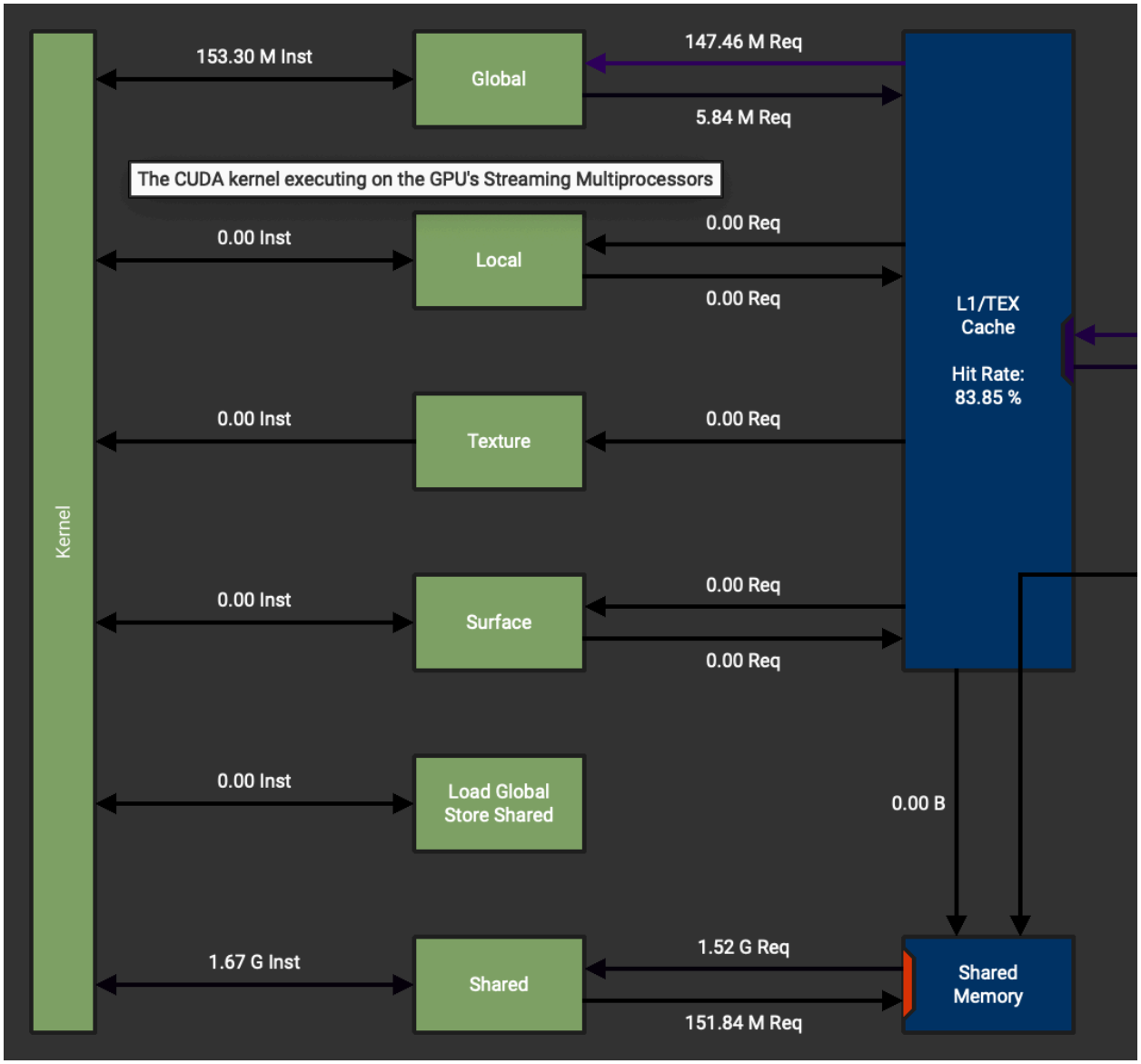
OP3:

Time (%)	Total Time (ns)	Instances	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	GridXYZ			BlockXYZ			
62.8	47,034,843	1	47,034,843.0	47,034,843.0	47,034,843	47,034,843	0.0	400	1	10000	16	16	1	unrollAndMatrixM
37.2	27,823,197	1	27,823,197.0	27,823,197.0	27,823,197	27,823,197	0.0	73	1	10000	16	16	1	unrollAndMatrixM
0.0	4,768	2	2,384.0	2,384.0	2,368	2,400	22.6	1	1	1	1	1	1	do_not_remove_th
0.0	4,672	2	2,336.0	2,336.0	2,304	2,368	45.3	1	1	1	1	1	1	prefn_marker_ker

OP2:

Time (%)	Total Time (ns)	Instances	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	GridXYZ			BlockXYZ			
31.8	45,871,138	100	458,711.4	458,720.0	458,240	459,233	188.0	400	1	100	16	16	1	matrixMultiplyShared(const f
29.5	42,620,613	100	426,206.1	425,937.0	415,809	434,944	4,405.9	4	9	100	16	16	1	unrollKernel(int, int, int,
19.8	28,599,498	100	285,995.0	285,952.0	284,161	292,160	973.3	1	25	100	16	16	1	unrollKernel(int, int, int,
18.9	27,281,326	100	272,813.3	272,800.0	272,001	273,473	309.5	73	1	100	16	16	1	matrixMultiplyShared(const f

- As shown below, we can see that shared memory is being utilized to perform the matrix multiplication



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

This optimization can be utilized in combination with OP4 by storing the mask matrix in constant memory. Given that it improves the OP Time for the second convolution layer but not the first, we can use this kernel only for layer 2 while using the OP1 tiled convolution kernel for layer 1. We can also use OP10 with this optimization to leverage fp16 arithmetic to improve speed and save memory.

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

- D. Kirk and W. Hwu, "Programming Massively Parallel Processors – A Hands-on Approach," Morgan Kaufman Publisher, 3rd edition, 2016. Chapter 16, Section 16.4 Pages 16-22.
- D. Kirk and W. Hwu, "Programming Massively Parallel Processors – A Hands-on Approach," Morgan Kaufman Publisher, 3rd edition, 2016. Chapter 4, Section 4.5 Pages 23-32.

5. Optimization Number #: 4 Optimization Name: Weight Matrix (Kernel) in constant memory

a. How does this optimization work in theory? Expected behavior?

Constant memory is a memory allocated on the device which is small in size (usually 64KB), it is set by the host and can not be modified by the kernel. The constant memory is accessible to all the threads in a grid with a very low latency as it is typically cached. Due to these properties, it can be leveraged in cases where all the threads need access to a constant value or values. The mask or weight matrix in case of convolution can be stored in the constant memory since it satisfies all the mentioned requirements. In our case the mask size is Channels*Map_Out*K*K which isn;t very large and under the prescribed limit.

With this optimization, we expect the OP times to decrease as the kernel won't be read from the global memory. Additionally, we also expect a decrease in memory bandwidth.

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

To implement, we declare the mask as a constant memory global variable. As we have different memory requirements for the two layers, we use the maximum mask size.

```
#include <cmath>
#include <iostream>
#include "gpu-new-forward.h"
#define TILE_WIDTH 16
__constant__ float Mask[16*4*7*7];
```

Next, we copy the mask from the host memory to constant memory

```
cudaMemcpyToSymbol(Mask, host_mask, mask_size*sizeof(float));
```

Finally, we make changes to the convolution kernel by removing the mask parameter and reading the mask values from the globally defined mask variable.

```
#define mask_4d(i3, i2, i1, i0) Mask[(i3) * (Channel * K * K) + (i2) * (K * K) + (i1) * (K) + i0]
```

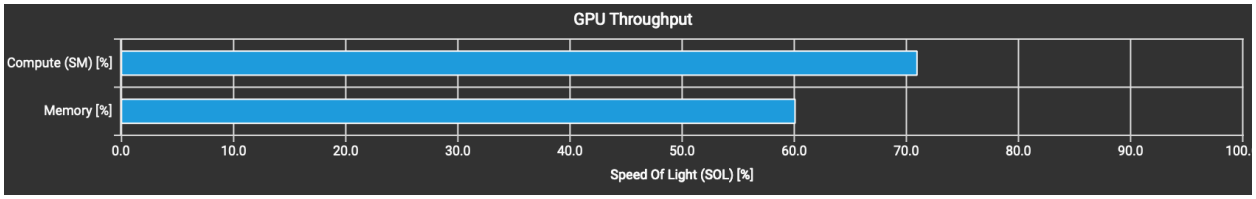
The remaining implementation remains the same as the baseline.

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

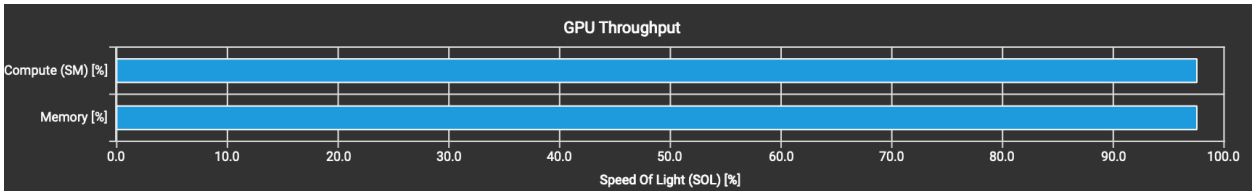
Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*4.0539 mss*	*0.397726 ms*	*0m1.461s*	*0.86*
1000	*21.0056 ms*	*18.4517 ms*	*0m10.086 s*	*0.886*
10000	*8.96649 ms*	*37.6022 ms*	*1m38.148 s*	*0.8714*

- Total Op Time: 46.29 ms (Baseline: 58.3 ms)
- We observe that both OP Time 1 and Op Time 2 decrease compared to the baseline.
- The speed of light memory throughput decreases from 97.55% to 60.97% compared to the baseline, showing the decreased memory requirements.

OP4:

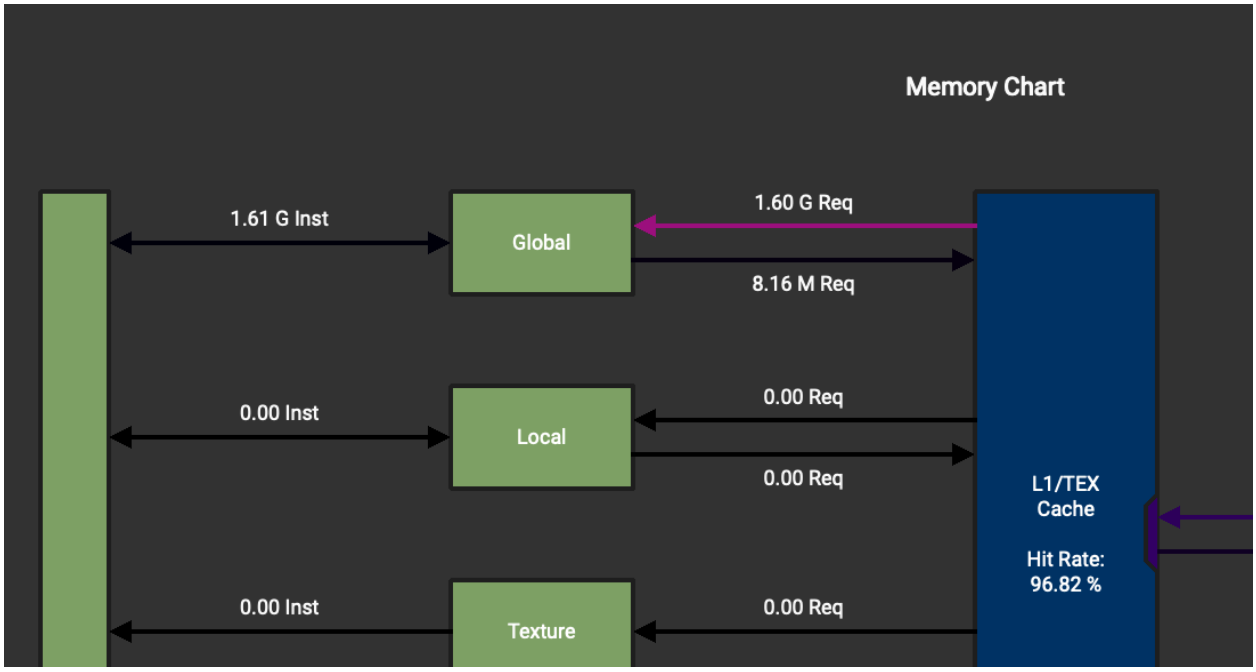


Baseline:

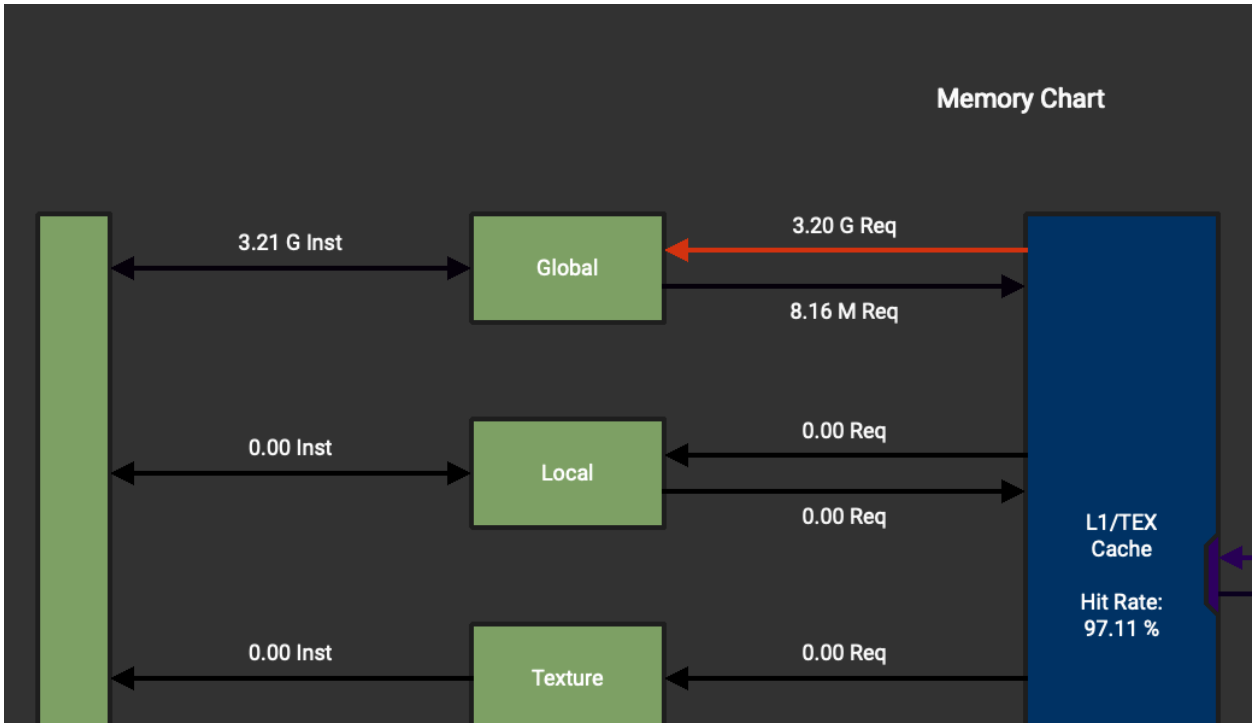


- By looking at the memory chart, we see that the data access to the global memory becomes exactly half compared to the baseline. This is because for computing each output element, we are now making half as many accesses.

OP4:



Baseline:



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

In theory, this optimization can be used with any of the other optimizations as all of them use the mask and storing it in constant memory will improve performance.

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

- D. Kirk and W. Hwu, "Programming Massively Parallel Processors – A Hands-on Approach," Morgan Kaufman Publisher, 3rd edition, 2016. Chapter 7, Section 7.3 Pages 8-11.
- https://leimao.github.io/blog/CUDA-Constant-Memory/

6. **Optimization Number #: 5 Optimization Name: Tuning with restrict and loop unrolling**

   a. **How does this optimization work in theory? Expected behavior?**

In case of a for loop, the loop body can be unrolled in order to reduce some overhead instructions. This happens due to instruction level parallelism which allows the compiler to execute multiple independent instructions simultaneously. However, too much unrolling can lead to increased register usage.

The use of restrict keywords while declaring arguments prevents pointer aliasing (multiple pointer variables pointing to the same address). Pointer aliasing leads to inefficiency as the compiler needs to make sure that the data corresponding to a pointer hasn't been modified by its alias. This can be solved by adding a __restrict__ keyword to function parameters, which ensures that the data pointed to by the pointer isn't referenced by any other pointer. This optimization is expected to improve efficiency.

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

To prevent pointer aliasing, we simply change the parameters of our function and add the restrict keyword.

```
__global__ void conv_forward_kernel(float * __restrict__ output, const float* __restrict__ input, const float * __restrict__ mask,
```

As our mask size is 7, we unroll the loop 2, 4 and 7 times.

```
for(int c=0; c<Channel; c++)
{
    for(int h=0; h<K; h++)
    {
        int w = 0;
        result += in_4d(batch, c, y_out+h, x_out+w)*mask_4d(m, c, h, w);
        result += in_4d(batch, c, y_out+h, x_out+w+1)*mask_4d(m, c, h, w+1);
        result += in_4d(batch, c, y_out+h, x_out+w+2)*mask_4d(m, c, h, w+2);
        result += in_4d(batch, c, y_out+h, x_out+w+3)*mask_4d(m, c, h, w+3);
        result += in_4d(batch, c, y_out+h, x_out+w+4)*mask_4d(m, c, h, w+4);
        result += in_4d(batch, c, y_out+h, x_out+w+5)*mask_4d(m, c, h, w+5);
        result += in_4d(batch, c, y_out+h, x_out+w+6)*mask_4d(m, c, h, w+6);
    }
}
```

```
float result = 0.0f;
for(int c=0; c<Channel; c++)
{
    for(int h=0; h<K; h++)
    {
        for(int w=0; w<K; w+=2)
        {
            result += in_4d(batch, c, y_out+h, x_out+w)*mask_4d(m, c, h, w);
            if(w+1 < K)
                result += in_4d(batch, c, y_out+h, x_out+w+1)*mask_4d(m, c, h, w+1);
        }
    }
}
```

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

Batch Size	# Unroll	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*7*	*0.189355 ms*	*3.41453 ms*	*0m1.390s*	*0.86*
1000	*7*	*1.21972 ms*	*4.49501 ms*	*0m9.603s*	*0.886*
10000	*7*	*11.5614 ms*	*44.6916 ms*	*1m33.261s*	*0.8714*
10000	*4*	*13.0616 ms*	*51.1124 ms*		*0.8714*
10000	*2*	*13.0256 ms*	*51.0081 ms*		*0.8714*

- Op Time Total (unrolling 7 times): 56.25 ms (Baseline: 58.3 ms)
- We observe that unrolling the entire inner loop (unrolling 7 times) is slightly faster than the baseline implementation.
- Further, on tuning the number of unrolls, we observe that unrolling 7 times achieves a better performance (56.25 ms) compared to both 2 times (64.17 ms) and 4 times (64.02

ms). This slight increase might be due to the overhead caused by the if statements to prevent violating the loop condition.

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

Yes, given the inner loop of the convolution is present in most of the implementations, we can pair this optimization with any of the other optimizations which use a convolution kernel. Similarly, the input image and mask can be added with the __restrict__ keyword. Specifically, we can use it with OP0, OP1, OP4.

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

- D. Kirk and W. Hwu, "Programming Massively Parallel Processors – A Hands-on Approach," Morgan Kaufman Publisher, 3rd edition, 2016. Chapter 14, Section 14.3 Page 327.
- https://developer.nvidia.com/blog/cuda-pro-tip-optimize-pointer-aliasing/

7. Optimization Number #: 7 Optimization Name: Multiple Kernel Implementation for different layer sizes

a. How does this optimization work in theory? Expected behavior?

As observed from previous implementations, the efficiency of kernels is dependent on the size of the inputs. Thus we can strategically select which kernel to utilize depending on the layer size. By this we will be utilizing the strengths of the different techniques.

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

Inside the conv_forward_gpu function, we check for the channel and map_out dimensions to decide which kernel to use. For the first layer which has a smaller input we utilize the baseline kernel implementation, whereas for layer 2 we utilize the fused kernel which combines unrolling and matrix multiplication.

```
if(Channel == 1 && Map_out==4)
{
    // printf("kernel1");
    int grid_z = Batch;
    int grid_y = width_tiles*height_tiles;
    int grid_x = Map_out;
    dim3 gridDim(grid_x, grid_y, grid_z);
    dim3 blockDim(TILE_WIDTH, TILE_WIDTH, 1);
    conv_forward_kernel<<<gridDim, blockDim>>>(device_output, device_input, device_mask, Batch, Map_out, Channel, Height, Width, K);
}
else{
    // printf("kernel2");
    int Width_unroll = Height_out*Width_out;
    int Height_unroll = K*K*Channel;
    int Height_kernel = Map_out;
    int Width_kernel = K*K*Channel;

    int Height_output = Height_kernel;
    int Width_output = Width_unroll;

    int unroll_size = Batch*Height_unroll*Width_unroll;

    dim3 gridDim(ceil(1.0*Width_output/TILE_WIDTH), ceil(1.0*Height_output/TILE_WIDTH), Batch);
    dim3 blockDim(TILE_WIDTH, TILE_WIDTH, 1);

    unrollAndMatrixMultiplyShared<<<gridDim, blockDim>>>(device_mask, device_input, device_output, Height_kernel, Width_kernel, Height_unro
}
```

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

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.199063 ms*	*0.317135 ms*	*0m1.371s*	*0.86*
1000	*1.23995 ms*	*2.85295 ms*	*0m9.434s*	*0.886*
10000	*11.7696 ms*	*27.8881 ms*	*1m31.568 s*	*0.8714*

- Total OP Time: 39.64 ms (baseline: 58.3 ms, OP3(fused kernel): 74.95 ms)
- We observe that we see a clear improvement over both the baseline and op3 kernels. It achieves a faster op time 1 compared to fused kernel (op3) and a faster op time 2 compared to the baseline, thus giving us the best performance of both worlds.

OP7:

Time (%)	Total Time (ns)	Instances	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	GridXYZ	BlockXYZ	
70.5	27,801,626	1	27,801,626.0	27,801,626.0	27,801,626	27,801,626	0.0	73 1 10000	16 16 1	unrollAndMatrixMu
29.5	11,626,743	1	11,626,743.0	11,626,743.0	11,626,743	11,626,743	0.0	4 25 10000	16 16 1	conv_forward_kerr

OP3:

Time (%)	Total Time (ns)	Instances	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	GridXYZ			BlockXYZ			
62.8	47,034,843	1	47,034,843.0	47,034,843.0	47,034,843	47,034,843	0.0	400	1	10000	16	16	1	unrollAndMatrixMu
37.2	27,823,197	1	27,823,197.0	27,823,197.0	27,823,197	27,823,197	0.0	73	1	10000	16	16	1	unrollAndMatrixMu

Baseline:

Time (%)	Total Time (ns)	Instances	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	GridXYZ			BlockXYZ			
79.6	45,266,735	1	45,266,735.0	45,266,735.0	45,266,735	45,266,735	0.0	16	9	10000	16	16	1	conv_forward_kern
20.4	11,601,896	1	11,601,896.0	11,601,896.0	11,601,896	11,601,896	0.0	4	25	10000	16	16	1	conv_forward_kern
0.0	4,800	2	2,400.0	2,400.0	2,368	2,432	45.3	1	1	1	1	1	1	do_not_remove_thi
0.0	4,704	2	2,352.0	2,352.0	2,304	2,400	67.9	1	1	1	1	1	1	prefn_marker_kern

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

Yes, in order to further improve the baseline kernel which is used for layer 1, we can combine OP4 kernel in constant memory and OP5 loop unrolling which will improve the performance for layer 1 as well as layer 2.

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

- D. Kirk and W. Hwu, “Programming Massively Parallel Processors – A Hands-on Approach,” Morgan Kaufman Publisher, 3rd edition, 2016. Chapter 16, Section 16.4 Pages 16-22.