ECE408 Project Report

Ayushi Patel Jayati Singh Kartikeya Sharma Rishabh Goyal

Team details

Team name: cudashouldawoulda
School affiliation: On campus students
Ayushi Patel (**ayuship2**)
Jayati Singh (**jayati**)
Kartikeya Sharma (**ksharma**)
Rishabh Goyal (**rgoyal6**)

Milestone 2

- **Include a list of all kernels that collectively consume more than 90% of the program time:**

Time(%)	Name
30.23%	[CUDA memcpy HtoD]
18.00%	volta_scudnn_128x64_relu_interior_nn_v1
17.31%	volta_gcgemm_64x32_nt
8.82%	fft2d_c2r_32x32
7.86%	volta_sgemm_128x128_tn
6.62%	op_generic_tensor_kernel
6.57%	fft2d_r2c_32x32
3.97%	cudnn::detail::pooling_fw_4d_kernel
0.42%	mshadow::cuda::MapPlanLargeKernel

- **Include a list of all CUDA API calls that collectively consume more than 90% of the program time.**

Time(%)	Name
42.85%	cudaStreamCreateWithFlags
33.41%	cudaMemGetInfo
20.90%	cudaFree

- **Include an explanation of the difference between kernels and API calls**

  Kernels (GPU Activities) in the *nvprof* output represent actual usage of the GPU for any kind of task. The time taken for GPU Activities represents the difference between the times the task actually started executing on the GPU and finished executing on the GPU.

  API calls are made by the host code (or by other API calls made by the code) that access the CUDA runtime. A GPU Activity is performed by initiating it with some form of API call. However since API calls are asynchronous, their finishing time is not related to the GPU activity that it launches, it may even finish executing before the kernel code is done using the GPU.

- **Show output of rai running MXNet on the CPU**

```
Loading fashion-mnist data... done
Loading model... done
New Inference
EvalMetric: {áccuracy:́ 0.8154}
```

- **List program run time**
  19.70 seconds user
  6.46 seconds system

- **Show output of rai running MXNet on the GPU**

  ```
  Loading fashion-mnist data... done
  Loading model... done
  New Inference
  EvalMetric: {accuracy: 0.8154}
  ```

- **List program run time**
  5.05 seconds user
  3.40 seconds system

CPU implementation

- **List whole program execution time**
  87.40 seconds user
  10.34 seconds system

- **List Op Times**
  Op Time 1: 11.223992 seconds
  Op Time 2: 60.508100 seconds

Milestone 3

Implemented a GPU Convolution

GPU implementation

Number of images 10000

- **Correctness**
  Correctness: 0.7653

- **List whole program execution time**
  7.83 seconds user
  4.07 seconds system

- **List Op Times**
  Op Time 1: 0.060051 seconds
  Op Time 2: 0.223534 seconds

Number of images 100

- **Correctness**
  Correctness: 0.76

- **List whole program execution time**
  7.41 seconds user
  3.49 seconds system

- **List Op Times**
  Op Time 1: 0.002697 seconds
  Op Time 2: 0.003387 seconds

Number of images 1000

- **Correctness**
  Correctness: 0.767

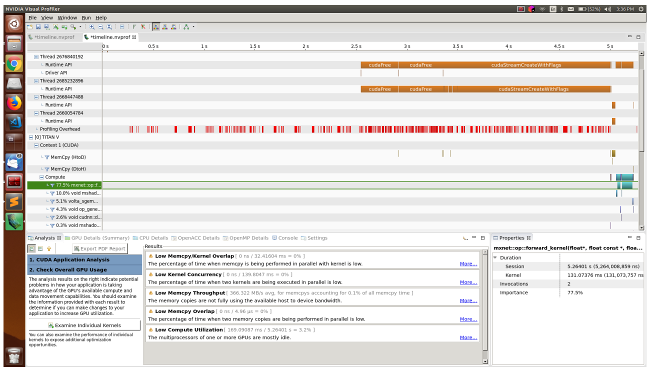
- **List whole program execution time**
  7.55 seconds user
  3.52 seconds system

- **List Op Times**
  Op Time 1: 0.007680 seconds
  Op Time 2: 0.022720 seconds

Nvprof profiling the execution

The following image shows nvvp profiling the overall GPU usage. As shown, there are a few issues with the kernel, for example low compute utilization, since most of the time is spent on memory copying and profiling overhead.

The next image shows nvvp identifying control divergence in the code. There is 50% control divergence at line 38, which is the line that checks to see if the output indices are within bounds. This condition is necessary because some threads would not be computing the output.



Results

Low Memcpy/Kernel Overlap [0 ns / 32.41604 ms = 0%]
The percentage of time when memcpy is being performed in parallel with kernel is low. More...

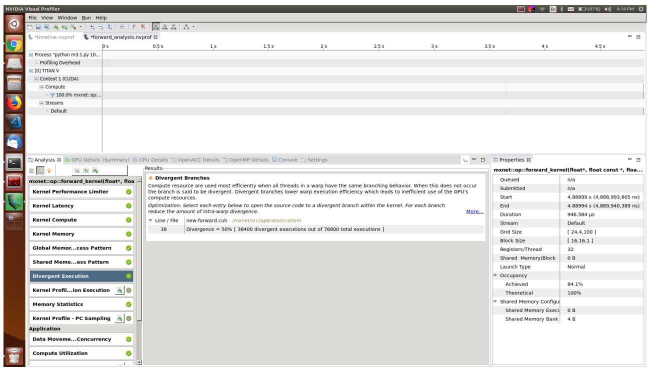
Low Kernel Concurrency [0 ns / 139.8047 ms = 0%]
The percentage of time when two kernels are being executed in parallel is low. More...

Low Memcpy Throughput [366.322 MB/s avg, for memcpys accounting for 0.1% of all memcpy time]
The memory copies are not fully using the available host to device bandwidth. More...

Low Memcpy Overlap [0 ns / 4.96 µs = 0%]
The percentage of time when two memory copies are being performed in parallel is low. More...

Low Compute Utilization [169.09087 ms / 5.26401 s = 3.2%]
The multiprocessors of one or more GPUs are mostly idle. More...

Duration	
Session	5.26401 s (5,264,008,859 ns)
Kernel	131.07376 ms (131,073,757 ns)
Invocations	2
Importance	77.5%

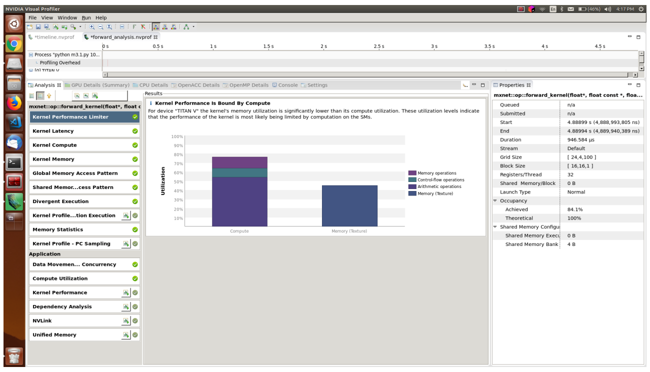


Results

Divergent Branches
Compute resource are used most efficiently when all threads in a warp have the same branching behavior. When this does not occur the branch is said to be divergent. Divergent branches lower warp execution efficiency which leads to inefficient use of the GPU's compute resources.
Optimization: Select each entry below to open the source code to a divergent branch within the kernel. For each branch reduce the amount of intra-warp divergence. More...

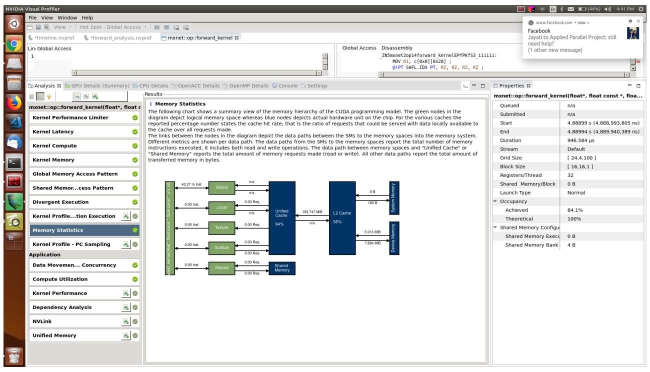
Line / File	new-forward.cuh - /mxnet/src/operator/custom
38	Divergence = 50% [38400 divergent executions out of 76800 total executions]

Next, we show the compute unit utilization of the kernel. NVVP shows that the code is bound by compute, even though we don't use shared memory. When we look at the global memory access pattern, we notice that the L2 cache has a 95% hit-rate, which makes the memory access more efficient.



Results

Kernel Performance Is Bound By Compute
For device "TITAN V" the kernel's memory utilization is significantly lower than its compute utilization. These utilization levels indicate that the performance of the kernel is most likely being limited by computation on the SMs.



Global Access Disassembly

```
_ZN5mxnet2op14forward_kernelEPfPKfS3_iiiiii:
    MOV R1, c[0x0][0x28] ;
    @!PT SHFL.IDX PT, RZ, RZ, RZ, RZ ;
```

Results

Memory Statistics
The following chart shows a summary view of the memory hierarchy of the CUDA programming model. The green nodes in the diagram depict logical memory space whereas blue nodes depicts actual hardware unit on the chip. For the various caches the reported percentage number states the cache hit rate; that is the ratio of requests that could be served with data locally available to the cache over all requests made.
The links between the nodes in the diagram depict the data paths between the SMs to the memory spaces into the memory system. Different metrics are shown per data path. The data paths from the SMs to the memory spaces report the total number of memory instructions executed, it includes both read and write operations. The data path between memory spaces and "Unified Cache" or "Shared Memory" reports the total amount of memory requests made (read or write). All other data paths report the total amount of transferred memory in bytes.

mxnet::op::forward_kernel(float*, float const *, floa...

Queued	n/a
Submitted	n/a
Start	4.88899 s (4,888,993,805 ns)
End	4.88994 s (4,889,940,389 ns)
Duration	946.584 µs
Stream	Default
Grid Size	[24,4,100]
Block Size	[16,16,1]
Registers/Thread	32
Shared Memory/Block	0 B
Launch Type	Normal
Occupancy	
Achieved	84.1%
Theoretical	100%
Shared Memory Configu	
Shared Memory Execu	0 B
Shared Memory Bank	4 B

Milestone 4

Optimization 1 (Unroll + Shared Memory Matrix Multiply):

Here, we implement the convolution operation as a matrix multiply, which is further optimized by using shared memory. The following image illustrates the unrolling process. Essentially each convolution is a dot product between the kernel and the corresponding portion of the image, which is turned into a multiplication between a row and a column in the matrix.

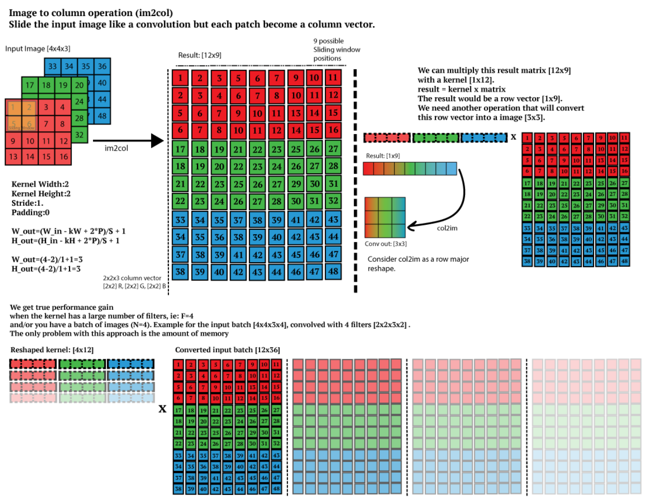
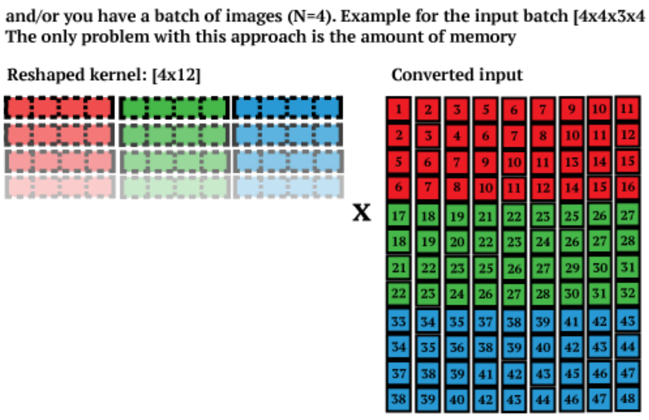


Image to column operation (im2col)
Slide the input image like a convolution but each patch become a column vector.
Input Image [4x4x3]
Result: [12x9]
9 possible Sliding window positions
im2col
Kernel Width:2
Kernel Height:2
Stride:1.
Padding:0
W_out=(W_in - kW + 2*P)/S + 1
H_out=(H_in - kH + 2*P)/S + 1
W_out=(4-2)/1+1=3
H_out=(4-2)/1+1=3
2x2x3 column vector [2x2] R, [2x2] G, [2x2] B
We can multiply this result matrix [12x9] with a kernel [1x12].
result = kernel x matrix
The result would be a row vector [1x9].
We need another operation that will convert this row vector into a image [3x3].
X
Result: [1x9]
col2im
Conv out: [3x3]
Consider col2im as a row major reshape.
We get true performance gain when the kernel has a large number of filters, ie: F=4
and/or you have a batch of images (N=4). Example for the input batch [4x4x3x4], convolved with 4 filters [2x2x3x2] .
The only problem with this approach is the amount of memory
Reshaped kernel: [4x12]
Converted input batch [12x36]
X

The next image illustrates the matrix multiply. Here the kernel is unrolled into a row vector and is multiplied with the unrolled input matrix. Since the kernel is already provided in this unrolled form as the layout is in row-major order, this also has the advantage of skipping the unrolling step with the kernel, and only performing it on the input.


and/or you have a batch of images (N=4). Example for the input batch [4x4x3x4
The only problem with this approach is the amount of memory
Reshaped kernel: [4x12]
Converted input
X

The following are the op-times for different dataset sizes:

- **Dataset Size 10000**:
  Op Time: 0.144436
  Op Time: 0.210640
- **Dataset Size 1000**:
  Op Time: 0.022911
  Op Time: 0.028153
- **Dataset Size 100**:
  Op Time: 0.003470
  Op Time: 0.003790

See file ece408_src/new-forward_shmem.cuh for implementation

Optimization 2 (Restrict + Loop unroll):

We use the __restrict__ keyword to explicitly tell the compiler that there is no pointer aliasing. This additional information allows the compiler to optimize its global memory accesses. GPUs with Compute Capability 3.5 and beyond have special read-only caches for data that the is read only for the lifetime of the kernel. When one uses the const keyword in addition to __restrict__ to qualify the pointer, it gives the compiler enough information to be sure that the data is read-only.

In addition, we also use the unroll pragma that parallelizes loops within the kernel. This has the potential to improve performance due to two possible reasons:

1. By removing the overhead associated with implementation of loops
2. By increasing the amount of parallelism

- **Dataset Size 10000:**
  Op Time: 0.133988
  Op Time: 0.249085
- **Dataset Size 1000:**
  Op Time: 0.027697
  Op Time: 0.028324
- **Dataset Size 100:**
  Op Time: 0.002797
  Op Time: 0.003542

This optimization results in slight improvements in kernel runtime for some cases and makes performance worse in others. One possible reason for the degradation might be that #pragma unroll requires the size of the loop to be known beforehand. Having variable number of iterations that are determined during runtime may hurt performance.

See file ece408_src/new-forward_constmem.cuh for implementation.

Optimization 3 (Kernel in constant memory):

Here, we transfer the kernel to constant memory, so that it isn't loaded into shared memory from global memory each time and accessed directly from the cached constant memory. The results of this optimization are:

- **Dataset Size 10000:**
  Op Time: 0.135243
  Op Time: 0.471975
- **Dataset Size 1000:**
  Op Time: 0.027884
  Op Time: 0.052545
- **Dataset size 100:**
  Op Time: 0.002029
  Op Time: 0.005090

This optimization seems to make the kernel perform worse, as can be observed from the increased optimes.

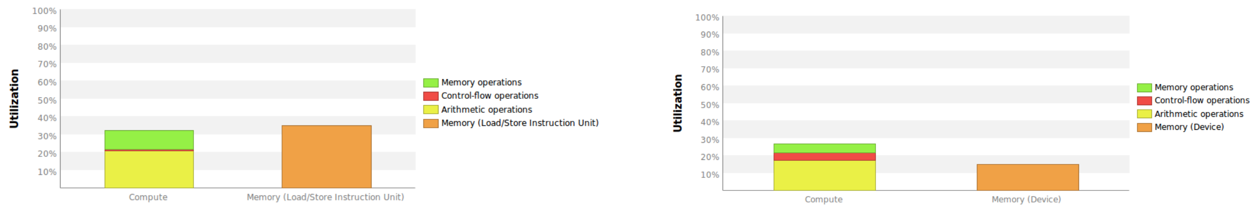
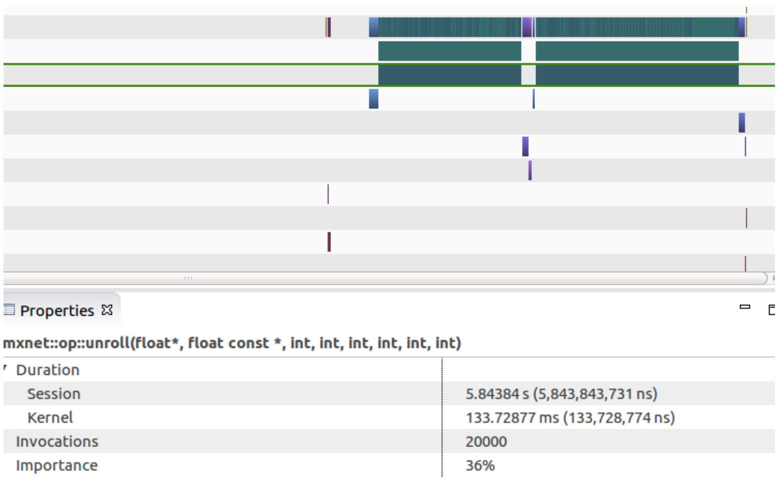


Figure 1: MatrixMultiply kernel Performance after Optimisation 1 (left) and Optimization 3 (right)

Moving kernel weights to constant memory does not cause much improvement in performance over moving kernel weights to shared memory. In fact, it results in a slight degradation in performance.

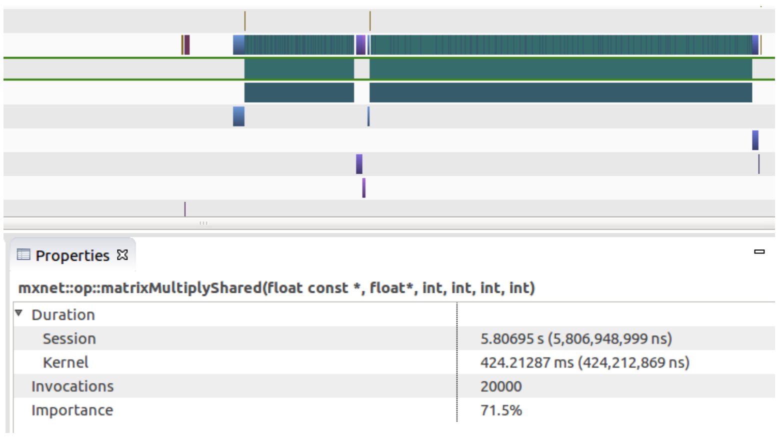
See file ece408_src/new-forward_constmem.cuh for implementation.


Properties

mxnet::op::unroll(float*, float const *, int, int, int, int, int, int)

Duration	
Session	5.84384 s (5,843,843,731 ns)
Kernel	133.72877 ms (133,728,774 ns)
Invocations	20000
Importance	36%

Figure 2: Unroll kernel time for optimization 3


Properties

mxnet::op::matrixMultiplyShared(float const *, float*, int, int, int, int)

Duration	
Session	5.80695 s (5,806,948,999 ns)
Kernel	424.21287 ms (424,212,869 ns)
Invocations	20000
Importance	71.5%

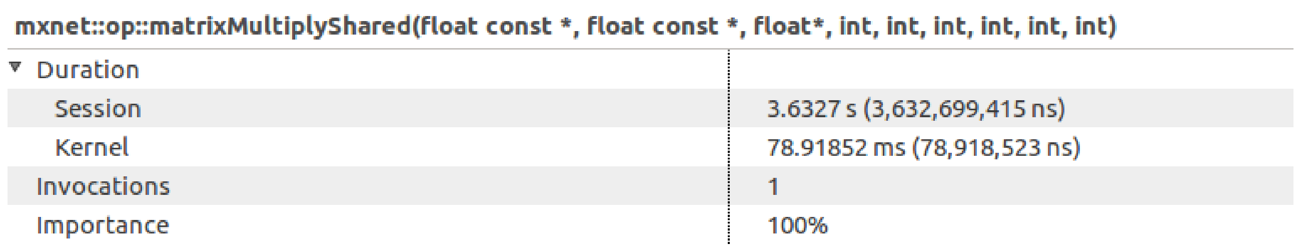
Figure 3: MatrixMultiply kernel time for optimization 3

Optimization 4 (Kernel Fusion for Unrolling and Matrix Multiplication)

We see that invoking 20000 kernels (one per image in the batch) increases the time significantly and thus, necessitates the need for kernel fusion. Hence, we also implemented a single kernel to do the unrolling and shared matrix multiplication together, instead of doing it separately in two kernels. Instead of unrolling the entire matrix, now we just unroll a tiled part of the matrix as and when it is required and move it straight to shared memory.

- **Dataset size 10000:**
  Op Time: 0.040706
  Op Time: 0.105419
- **Dataset size 1000:**
  Op Time: 0.004092
  Op Time: 0.011269
- **Dataset size 100:**
  Op Time: 0.000435
  Op Time: 0.001162

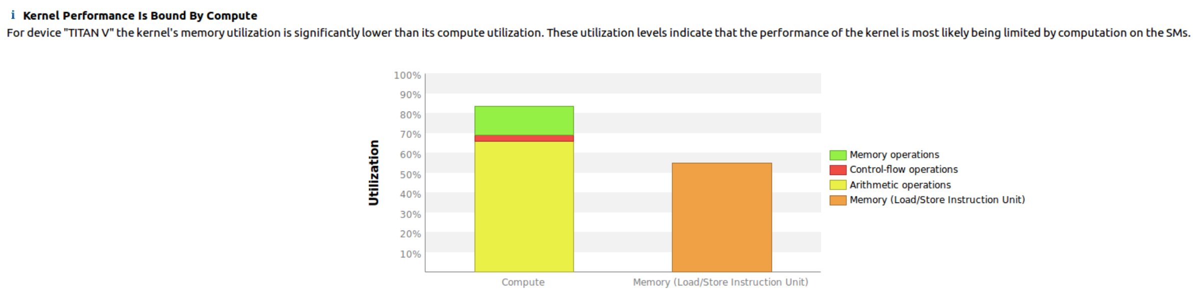
This optimization results in a great performance jump as it subsumes all kernel invocations and replaces them with one invocation. It also prevents multiple writebacks to the global memory for the entire unrolled matrix. *See file ece408_src/new-forward-fusion.cuh for implementation.*



mxnet::op::matrixMultiplyShared(float const *, float const *, float*, int, int, int, int, int, int)

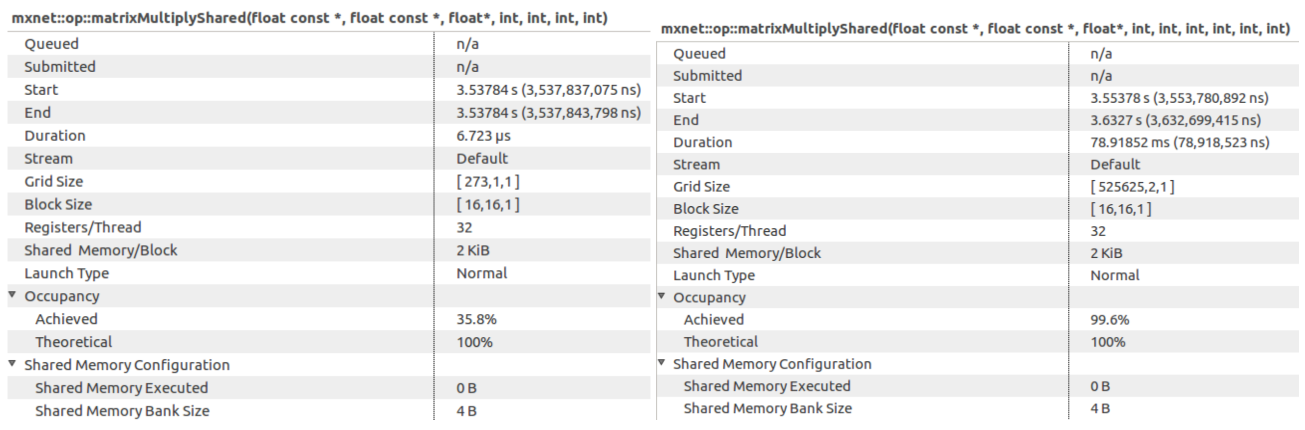
▼ Duration	
Session	3.6327 s (3,632,699,415 ns)
Kernel	78.91852 ms (78,918,523 ns)
Invocations	1
Importance	100%

Figure 4: Indicates significant improvement in kernel time for the fused kernel



i Kernel Performance Is Bound By Compute
For device "TITAN V" the kernel's memory utilization is significantly lower than its compute utilization. These utilization levels indicate that the performance of the kernel is most likely being limited by computation on the SMs.

Figure 5: Increased compute and memory utilisation



mxnet::op::matrixMultiplyShared(float const *, float const *, float*, int, int, int, int)

Queued	n/a
Submitted	n/a
Start	3.53784 s (3,537,837,075 ns)
End	3.53784 s (3,537,843,798 ns)
Duration	6.723 μs
Stream	Default
Grid Size	[273,1,1]
Block Size	[16,16,1]
Registers/Thread	32
Shared Memory/Block	2 KiB
Launch Type	Normal
▼ Occupancy	
Achieved	35.8%
Theoretical	100%
▼ Shared Memory Configuration	
Shared Memory Executed	0 B
Shared Memory Bank Size	4 B

mxnet::op::matrixMultiplyShared(float const *, float const *, float*, int, int, int, int, int, int)

Queued	n/a
Submitted	n/a
Start	3.55378 s (3,553,780,892 ns)
End	3.6327 s (3,632,699,415 ns)
Duration	78.91852 ms (78,918,523 ns)
Stream	Default
Grid Size	[525625,2,1]
Block Size	[16,16,1]
Registers/Thread	32
Shared Memory/Block	2 KiB
Launch Type	Normal
▼ Occupancy	
Achieved	99.6%
Theoretical	100%
▼ Shared Memory Configuration	
Shared Memory Executed	0 B
Shared Memory Bank Size	4 B

Figure 6: Compares one invocation of optimisation 3 (left) with optimisation 4 (right)

Milestone 5

Optimization 5 (Joint Register and Shared-Memory Tiling):

Since Registers are accessed at extremely high throughput being private to each thread and shared memory is accessed at lower throughput than registers but is visible to all threads in a block, we typically use both for tiling different dimensions of a multidimensional data.

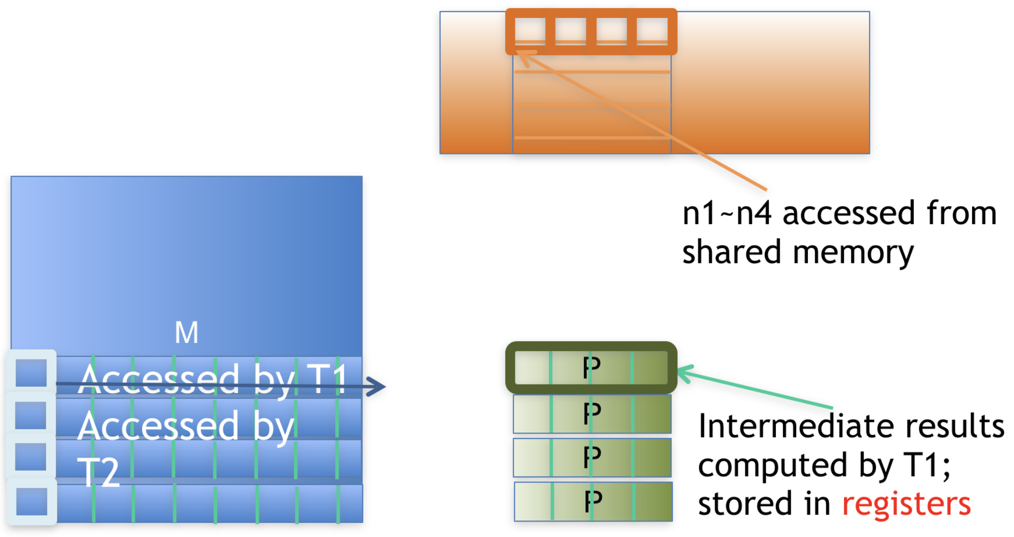
Algorithm: Here M,N are input matrices and P = output matrix = MN

Each block has (TILE_WIDTH_M) threads. A tile of the N matrix is loaded into shared memory, one element by each thread, for a total of TILE_WIDTH_N * K elements per block, where K = TILE_WIDTH_M / TILE_WIDTH_N .

Each thread loads K elements from M, into thread-local registers. These K elements are in the same row from M. The value of TILE_WIDTH_N is decided based on the number of registers available per thread.
Each thread then computes TILE_WIDTH_N output elements, and writes them to thread-local registers. After the algorithm is done, it writes them to global memory.

In one iteration, each thread accesses one M element from register, accesses TILE_WIDTH_N elements from shared memory and calculates one step for TILE_WIDTH_N P elements. K such steps are performed per iteration, each iteration handles a different tile from M and N. Total number of iterations is thus (number of columns in A)/K.

This is better illustrated in the below figure:


n1~n4 accessed from shared memory
M
Accessed by T1
Accessed by T2
P
P
P
P
Intermediate results computed by T1; stored in registers

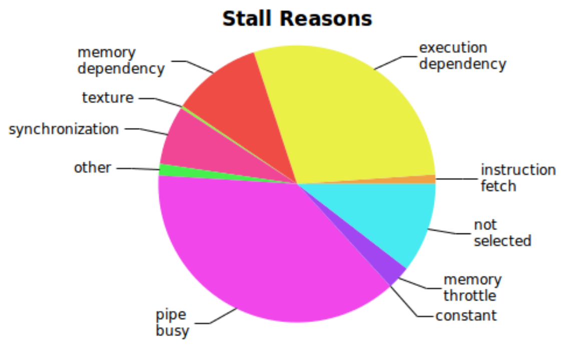
-
  ```
  Dataset Size 10000:
  Op Time:  0.033027
  Op Time:  0.054981
  Correctness:  0.7653
  ```

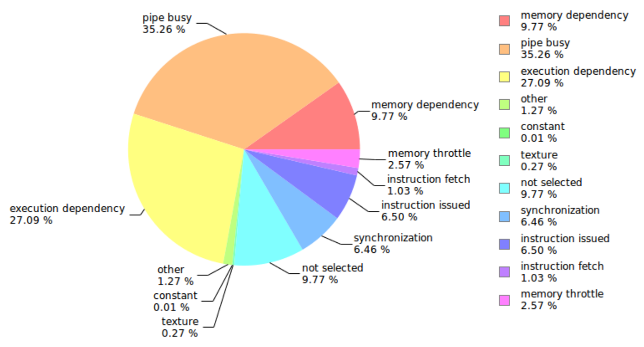
-
  ```
  Dataset Size 1000:
  Op Time:  0.003000
  Op Time:  0.005001
  Correctness:  0.767
  ```

-
  ```
  Dataset size 100:
  Op Time:  0.000366
  Op Time:  0.000610
  Correctness:  0.76
  ```

We then ran nvprof on this kernel, and observed the following:

- We first looked at what was limiting kernel latency, called instruction stall reasons, which indicate conditions that prevent warps from executing on any given cycle. As can be observed from the pie chart, most of the time the stalls are caused due to the pipe being busy. This means that most of the time, the compute resources required are not available.



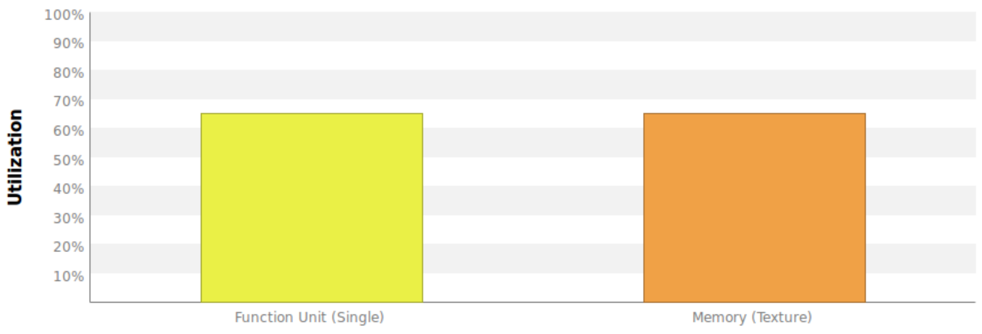


To investigate further, we looked at the register usage, since this kernel uses a lot of registers to store output and load one of the input matrices. As expected, nvvp suggested that the GPU utilization may be limited by register usage. The occupancy per SM was only 50%, whereas the device limit is 100%.

So we then tried reducing the register usage of the kernel, and replaced some of the integer variable declarations with `#defines`. This improved performance, as can be observed in the new optimes for the dataset of size 10000:

Op Time: 0.015302
Op Time: 0.044840

- **Compute v/s bandwidth**: As can be observed in the below figure, the compute and memory bandwidth utilization of our kernel are both pretty balanced, which was a good sign.



Optimization 6 (Multiple kernel implementations for different layers)

From Table 1, we see that the number of input and output channels for the first layer is significantly less than the second layer. This is one of the reasons that justify the need for different layer implementations.

Constant Memory Usage for Kernel 1

The filter matrices are not modified and can be read-only. If these matrices are transferred to the constant memory from the host-code just once then the cost to read from global memory is amortized amongst all the threads. Using `cudaMemcpyToSymbol(w_con_dptr, w.dptr_, M*C*ks*sizeof(float))` we transfer the kernel matrices to the constant memory for the first kernel. Table 2 shows that the use of constant memory for the second kernel invocation increases the execution time significantly. This is because the transfer of the filter matrix for the second kernel is too expensive as the number of input and output channels is very high. Thus, the benefits of using constant memory are less than the cost of the transfer. For the first kernel, however, constant memory use significantly improves performance. Thus, we decided to use different implementations

Register-Tiling for Kernel 2

Since the cost of transfer to the constant memory is very high for the second invocation, we implemented register-tiled matrix multiply here. We did not implement register-tiled matrix multiply for the first one as constant memory access where each thread in a warp accesses the same location is as fast as register access and hence both give similar performance as mentioned in the CUDA Programming Guide Performance Considerations.

Optimization 7 (Sweeping various parameters to find best values of block sizes)

Kernel 1: To get input fmaps to shared memory

Table 2 shows the improvement in the execution time for the first kernel at 16x16 and 32x32. It is important to note that *only* subtileN is loaded to the shared memory and the filter matrix is read directly from the constant memory. The output channels M=12, output fmaps have a dimension of 66x66 and input filter has dimension of 5x5. Hence, to maximize thread coalescing while loading the input fmap elements, the width 32 was chosen. To select the height of the tile width, two points need to be noted. (i) If the height is a multiple of 12, there will many warps that do not do any work while loading from the input fmaps since 25 is not a multiple of 12. (ii) If the height is 25, there will be many warps doing no work while writing the output to the output fmaps since output channels are only 12. Thus we select a height of 13 to get an optimal trade-off. Since `blockDim.x` is 32, there will be no control divergence. We get an optime of 0.024386s for 10000 images. This is further improved to 0.015302s with the introduction of `defines` for variables of filter width = 5, since this is common to both invocations and can be resolved at compile-time. This is also mentioned in Optimization 5.

Kernel 2: Optimal tile-size for Register-Tiling

For register tiling, the tile width of the second matrix is decided based on the number of registers, and the tile width of the first matrix has to be an integer multiple of this value. We experimented with various tile sizes

Kernel Invocation	**K, C, M**
Layer 1	5, 1, 12
Layer	5, 12, 24

Table 1: Number of input channels (C), output channels (M), and filter width (K) for the two convolution layers

Unrolled MM with SM	**8x8**	**12x12**	**16x16**	**24x24**	**32x32**
Layer 1	0.036034	0.028974	0.027741	0.052693	0.052346
Layer 2	0.090382	0.086505	0.083616	0.05732	0.079713
Unrolled MM with CM	**8x8**	**12x12**	**16x16**	**24x24**	**32x32**
Layer 1	0.045674	0.032482	0.024596	0.052966	0.042679
Layer 2	0.677344	0.617081	0.406107	0.296773	0.116721

Table 2: Unrolled MM (Matrix Multiply) using Shared Memory(SM) and Constant Memory(CM) for filter matrices for the two convolution layers

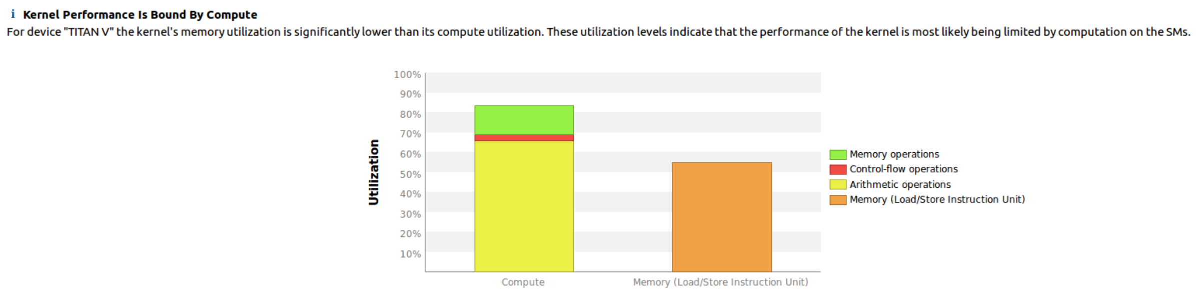
(x,y) where x and y are the tile widths of the first and second matrices respectively. We tried values (64,16), (32,16), (16,16), (64,32), (32,32) and so on. We observed the best optimes for the tile widths (32,32) so we decided to stick with this for our final kernel. The obtained optime (for dataset size 10000) was 0.032433s.

Optimization 8 (Parallelization Across Input Channels)

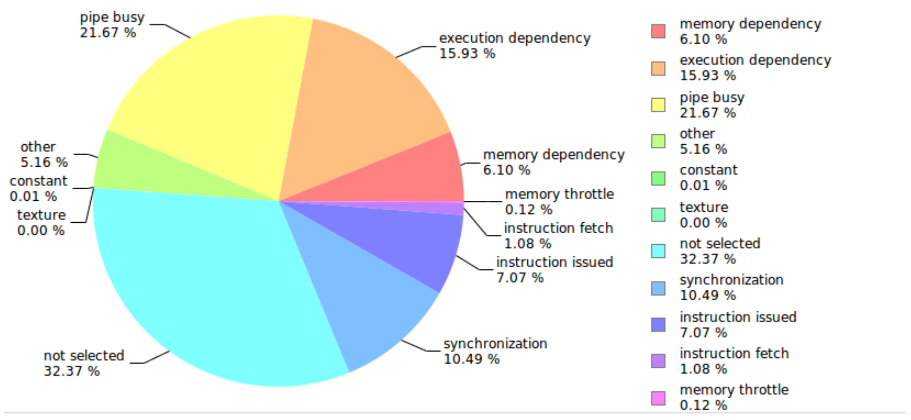
Within the fused shared matrix multiply - unroll kernel we map multiple threads to a single output element. Lets say that the input was $B \times C \times H \times W$ and and kernel size is $M \times C \times K \times K$. Then the output dimensions are $B \times M \times (H - K + 1) \times (W - K + 1)$. The number of threads used per output element is T, then each thread unrolls and multiplies $\frac{C}{T}$ channels. We then use an atomic add operation to add the result of each of these T threads to their corresponding output elements. We did not use reduction instead of atomicAdd as the number of channels per thread was fairly small and would not have provided much speed up.
Our observations were that this optimization did not result in any improvement over the vanilla fusion kernel. Hence we did not use it as part of our optimized kernel. An NVVP analysis showed that the kernel was compute bound. We believe that the number of threads spawned in the kernel without parallelization across input channels are enough to occupy all resources at once, hence introducing more parallelization does not result in any improvements. The introduction of atomic adds further harms the performance.

T	1	2	4	6	12
10000	0.071207	0.081504	0.113376	0.122382	0.153217
1000	0.007167	0.008686	0.012358	0.012112	0.015321
100	0.000748	0.000911	0.001258	0.001239	0.001563

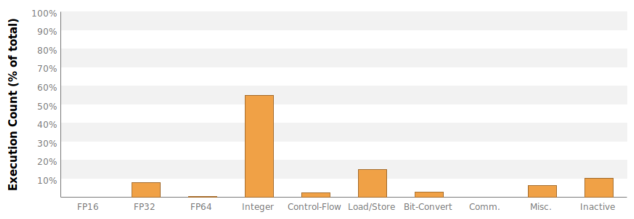
Table 3: Kernel run time (in seconds) for the second invocation for different values of T


Kernel Performance Is Bound By Compute

For device "TITAN V" the kernel's memory utilization is significantly lower than its compute utilization. These utilization levels indicate that the performance of the kernel is most likely being limited by computation on the SMs.



The following chart shows the mix of instructions executed by the kernel. The instructions are grouped into classes and for each class the chart shows the percentage of thread execution cycles that were devoted to executing instructions in that class. The "Inactive" result shows the thread executions that did not execute any instruction because the thread was predicated or inactive due to divergence.

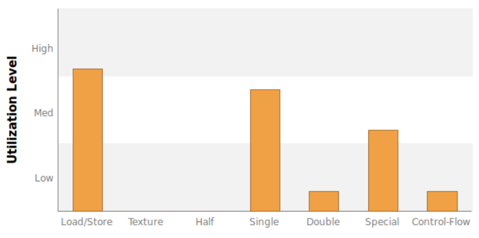


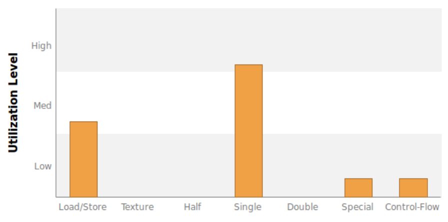
To improve this kernel we tried reducing the number of integer operations in the kernel. But that did not cause significant improvement in the performance.

Miscellaneous Optimizations

We also tried some miscellaneous optmizations which yielded small improvements in optimes:

- According to CUDA documentation, *"Integer division and modulo operations are particularly costly and should be avoided or replaced with bitwise operations whenever possible: If n is a power of 2, (i / n) is equivalent to (i >>log2 n) and (i % n) is equivalent to (i & n - 1)"*. Thus we replaced division with right-bitshift and modulo with a bitwise-and wherever possible.
- Double precision floating point arithmetic can lead to additional instruction cycles. Thus we append *f* whenever defining floating point constants to avoid them being interpreted as doubles. This reduced the usage of double precision floating point unit, as shown in the below nvvp figure.


Utilization Level
High
Med
Low
Load/Store
Texture
Half
Single
Double
Special
Control-Flow


Utilization Level
High
Med
Low
Load/Store
Texture
Half
Single
Double
Special
Control-Flow