ECE408 Project Report

Team Name: TensorJammed
Team Members:

Name	NetID	UIN
Jingning Tang	jtang10	664436670
Monil Pathak	mpathak2	672052183
Xiang Li	xiangl12	667244119

Final Report:

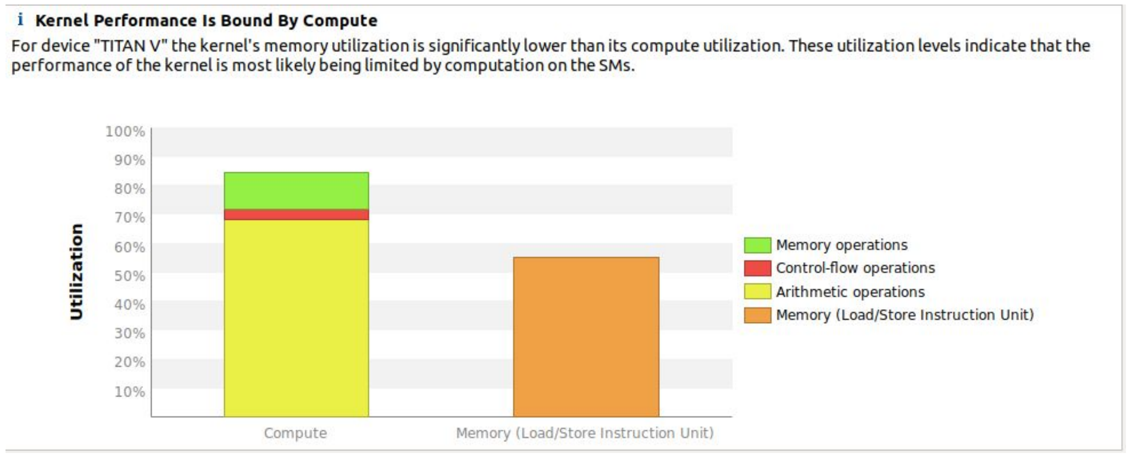
Optimization 1: Kernel fusion for unrolling and matrix-multiplication

- **How you identified the optimization opportunity?**
  We realized the combining two kernels will save the trip back to global memory and eventually got the code worked out from the inspiration of exam 2.

- **Why you thought the approach would be fruitful?**
  If we launch separate kernels, we need to store the intermediate results back to the global memory and read them back for the second kernel. Since the operations on the global memory consumes a lot of cycles, fusing two kernels together to use either register or shared memory would reduce the global memory usage.

- **The effect of the optimization. was it fruitful, and why or why not. Use nvprof and NVVP to justify your explanation.**
  We used the exam 2 code where x is conceptually unrolled while being loaded into shared memory, and a tiled matrix multiplication is performed with the loaded shared memory.
  The performance is comparable to the shared memory convolution kernel in milestone 4. We achieve an op time of 51 + 134 ms for fused unroll and matrix

multiplication, but both kernel bounded by compute. From nvvp we see all 32 blocks per SM is active and the occupancy is 97%. Problems we have discovered include a low bandwidth of device memory reads (4.042GB/s). This shows our loading is not well coalesced and not using L2 cache effectively. Since we are launching with a block of 8 * 8, the control divergence is larger than desired (7.2%).


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.

- **Any external references used during identification or development of the optimization**
  Exam 2 code.

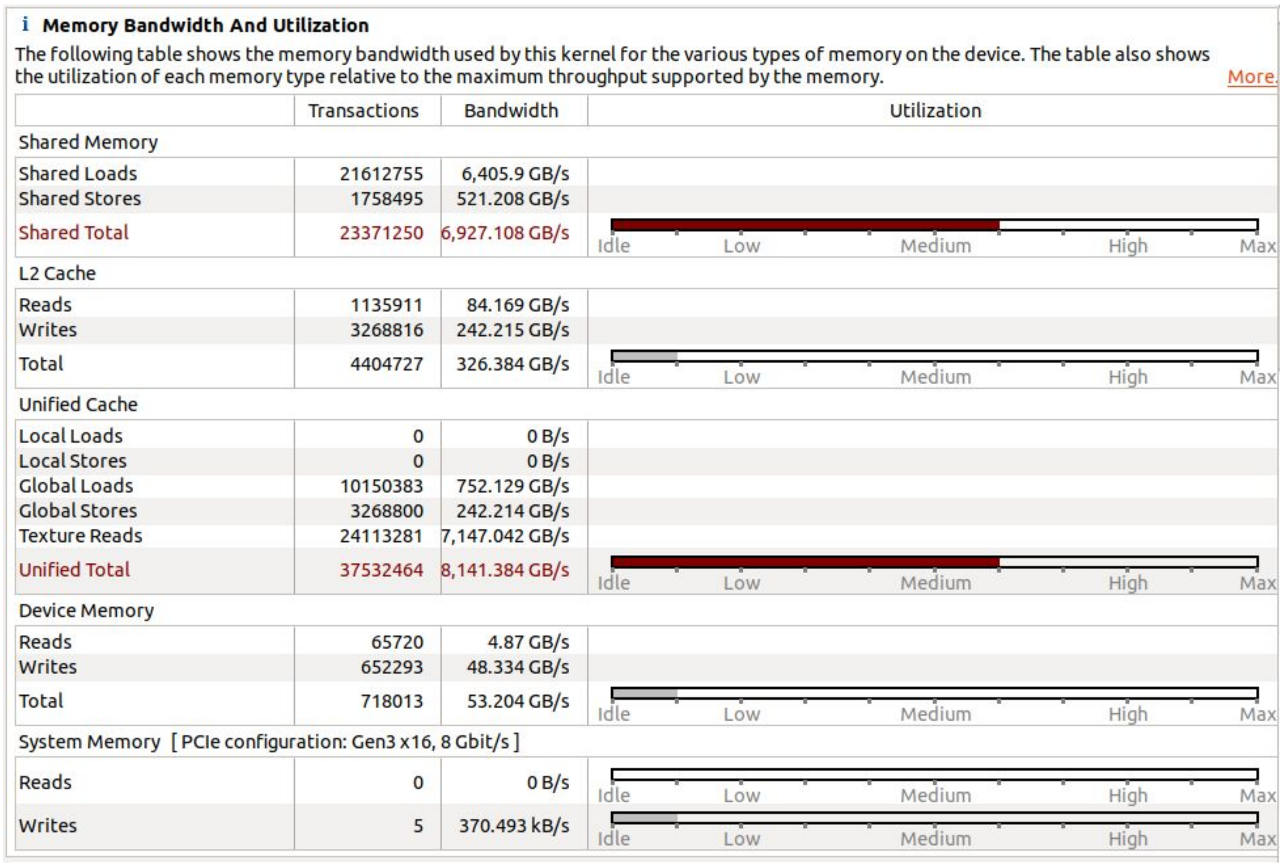
Optimization 2: Data prefetching & Coalescing for better Global Memory Usage

- **How you identified the optimization opportunity?**
  Realizing that the global memory into shared memory is really time-consuming, we thought about how we can increase the bandwidth.

- **Why you thought the approach would be fruitful?**
  1. By improving coalescing, it will naturally boost the performance by utilizing the RAM burst.
  2. For the data prefetching, the idea is in each iteration of the loop, we can pre-load the global memory data to the register and then in the next iteration let shared memory read from these registers. This mixes the computation with the global memory loading and could improve the task parallelism of loading and computing. Since the computation part takes very long time, we put the pre-loading code right before computation so the computation and global memory read will greatly overlap and save some time.

- **The effect of the optimization. was it fruitful, and why or why not. Use nvprof and NVVP to justify your explanation.**
    1. The data prefetching is a successful optimization. We achieved an op time of 47 + 118 ms, an improve of 4ms and 16ms for the two layers respectively. The first kernel is now bounded by both compute and memory bandwidth, while the second still bounded by compute. Since this method used more registers, the active block per SM decreased to 8 but the occupancy is still above 90 percent.
    2. The memory access issues persists. The good strategy is to consecutively load the global memory into the shared memory to achieve coalescing and then calculates the unrolled indices to fetch data from shared memory. This is definitely faster than our current implementation but could be affected by shared memory bank conflict. We should have looked into the profiling data much earlier so that we could start fixing the coalescing issue much earlier.

Below are the snapshots from nvprof showing that the device memory utilization is very low (53.2 GB/s).



Memory Bandwidth And Utilization

The following table shows the memory bandwidth used by this kernel for the various types of memory on the device. The table also shows the utilization of each memory type relative to the maximum throughput supported by the memory. More...

	Transactions	Bandwidth	Utilization
Shared Memory			
Shared Loads	21612755	6,405.9 GB/s	
Shared Stores	1758495	521.208 GB/s	
Shared Total	23371250	6,927.108 GB/s	Medium
L2 Cache			
Reads	1135911	84.169 GB/s	
Writes	3268816	242.215 GB/s	
Total	4404727	326.384 GB/s	Idle
Unified Cache			
Local Loads	0	0 B/s	
Local Stores	0	0 B/s	
Global Loads	10150383	752.129 GB/s	
Global Stores	3268800	242.214 GB/s	
Texture Reads	24113281	7,147.042 GB/s	
Unified Total	37532464	8,141.384 GB/s	Medium
Device Memory			
Reads	65720	4.87 GB/s	
Writes	652293	48.334 GB/s	
Total	718013	53.204 GB/s	Idle
System Memory [PCIe configuration: Gen3 x16, 8 Gbit/s]			
Reads	0	0 B/s	Idle
Writes	5	370.493 kB/s	Idle

- **Any external references used during identification or development of the optimization**
  Slide from UPENN

Optimization 3: Thread coarsening

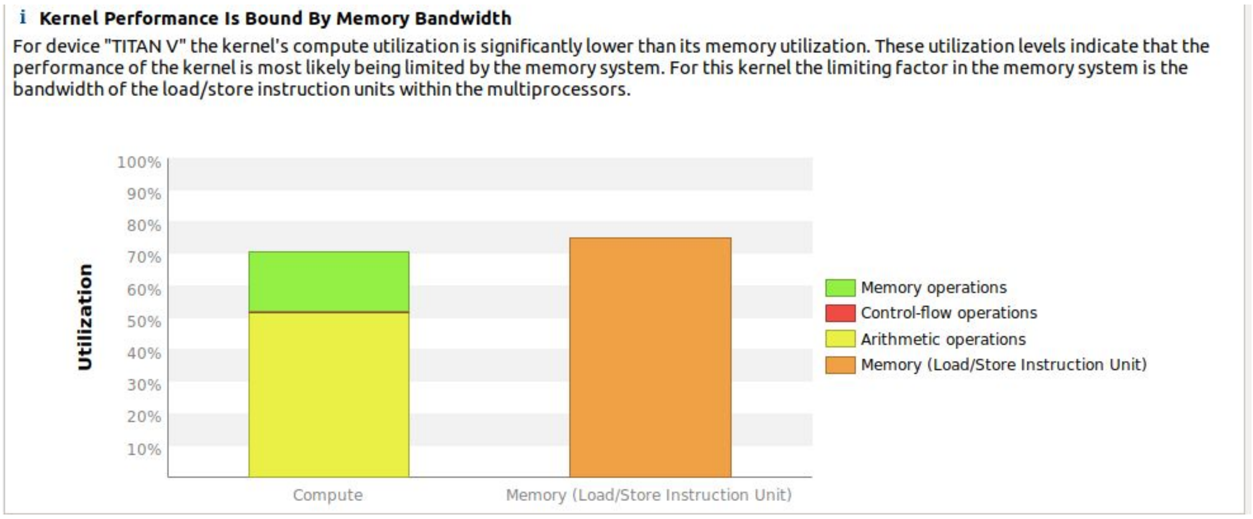
- **How you identified the optimization opportunity?**
  We found this from the slides in ECE 508 and textbook that allocating more work to each thread to speed things up. After all, we are primarily bounded by compute and if one thread computes more than one output we may see a decrease in the overhead and improve compute capacity.

- **Why you thought the approach would be fruitful?**
  If one thread only calculate one output, chances are many thread-private resources, especially registers will not be effectively reused. By allowing each thread to calculate multiple outputs, the total amount of threads required to finish the job will be heavily reduced and therefore launching more blocks at the same time will be possible.

- **The effect of the optimization. was it fruitful, and why or why not. Use nvprof and NVVP to justify your explanation.**
  The optimization is successful. For instance, If we set the thread granularity to 4 (each thread computes 4 output locations),it would decrease the op time to 26 + 78 ms, 21ms and 40ms improvement respectively compared to granularity of 1.

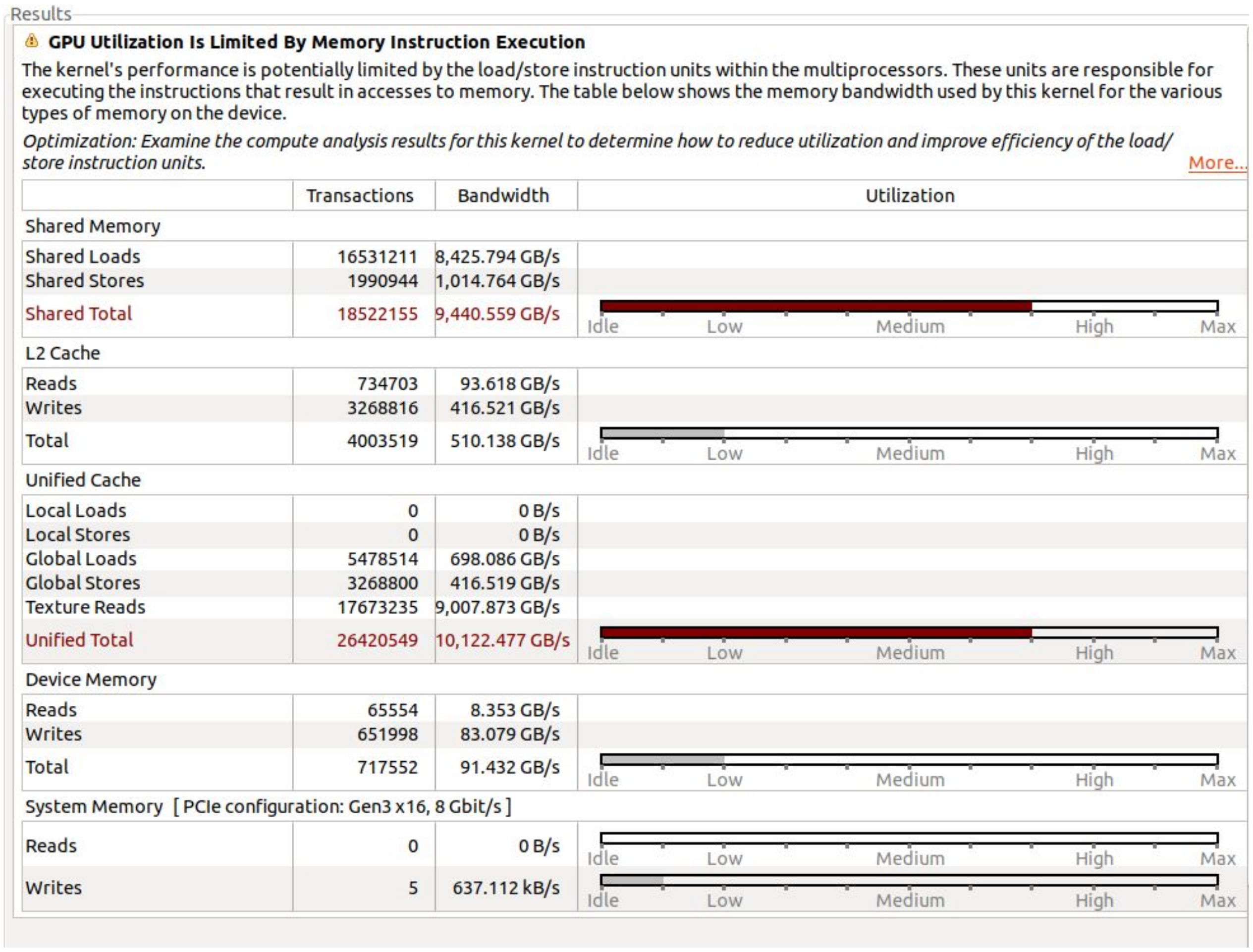
  In the process, we also tried different thread granularities. Starting from 1, we used 2, 4, and later 8 and 10. We found the bigger granularity gets, the faster out computation will be. Granularity of 10 does not have any improvement upon 8 so we eventually used 8 for the final version. While thread coarsening does speed up the computation, more registers are used to store the computation results locally, causing the kernel to be bounded by memory bandwidth in both layers. The shared memory usage and global memory bandwidth almost doubled due to the improvement in compute, but it is still problematic . At the same time 62 registers are used per thread. It limits the number of block per SM to 4 in the first kernel and 2 in the second kernel. The occupancy consequently is decreased to around 50%.

Below are the performance analysis from granularity of 4. Note that there are significant device memory utilization compared to the lower granularity but register usage is also skyrocketed that only 4 kernels are launched per SM.



Kernel Performance Is Bound By Memory Bandwidth

For device "TITAN V" the kernel's compute utilization is significantly lower than its memory utilization. These utilization levels indicate that the performance of the kernel is most likely being limited by the memory system. For this kernel the limiting factor in the memory system is the bandwidth of the load/store instruction units within the multiprocessors.



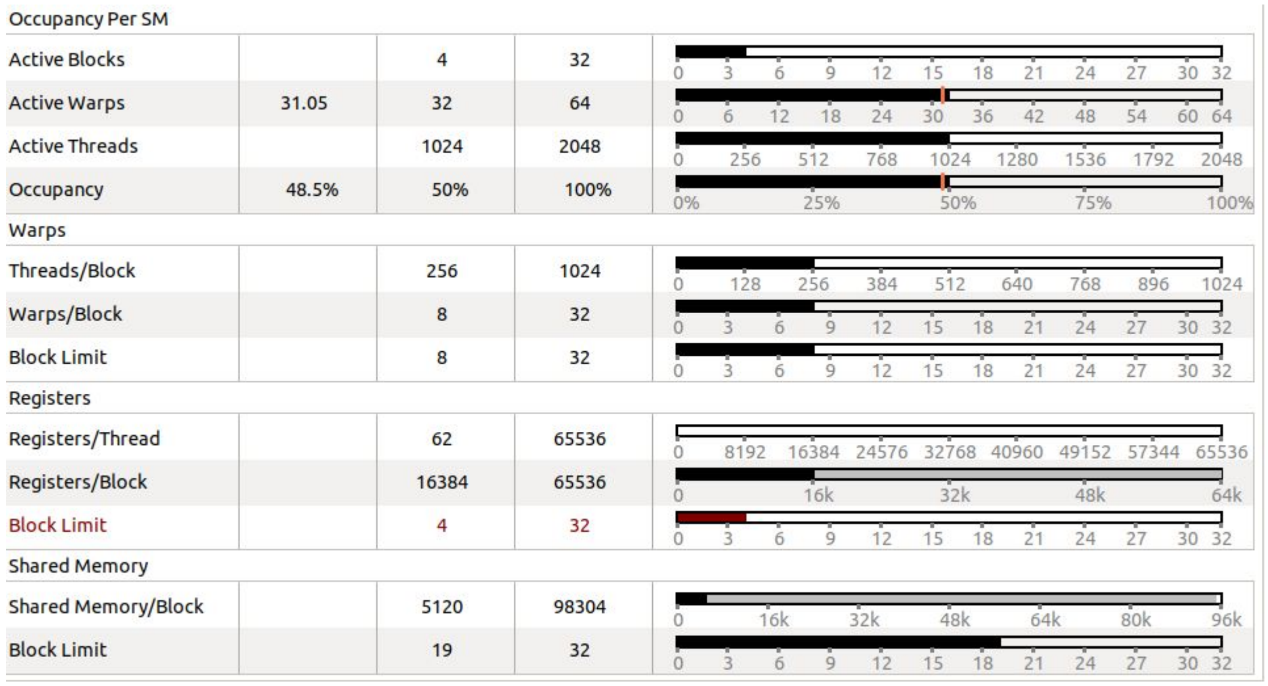
Results

GPU Utilization Is Limited By Memory Instruction Execution

The kernel's performance is potentially limited by the load/store instruction units within the multiprocessors. These units are responsible for executing the instructions that result in accesses to memory. The table below shows the memory bandwidth used by this kernel for the various types of memory on the device.

Optimization: Examine the compute analysis results for this kernel to determine how to reduce utilization and improve efficiency of the load/store instruction units. More...

	Transactions	Bandwidth	Utilization
Shared Memory			
Shared Loads	16531211	8,425.794 GB/s	
Shared Stores	1990944	1,014.764 GB/s	
Shared Total	18522155	9,440.559 GB/s	
L2 Cache			
Reads	734703	93.618 GB/s	
Writes	3268816	416.521 GB/s	
Total	4003519	510.138 GB/s	
Unified Cache			
Local Loads	0	0 B/s	
Local Stores	0	0 B/s	
Global Loads	5478514	698.086 GB/s	
Global Stores	3268800	416.519 GB/s	
Texture Reads	17673235	9,007.873 GB/s	
Unified Total	26420549	10,122.477 GB/s	
Device Memory			
Reads	65554	8.353 GB/s	
Writes	651998	83.079 GB/s	
Total	717552	91.432 GB/s	
System Memory [PCIe configuration: Gen3 x16, 8 Gbit/s]			
Reads	0	0 B/s	
Writes	5	637.112 kB/s	



Occupancy Per SM			
Active Blocks		4	32
Active Warps	31.05	32	64
Active Threads		1024	2048
Occupancy	48.5%	50%	100%
Warps			
Threads/Block		256	1024
Warps/Block		8	32
Block Limit		8	32
Registers			
Registers/Thread		62	65536
Registers/Block		16384	65536
Block Limit		4	32
Shared Memory			
Shared Memory/Block		5120	98304
Block Limit		19	32

- **Any external references used during identification or development of the optimization**
  ECE 508 slides, textbook chapter 6.

Optimization 4: Half precision arithmetic

- **How you identified the optimization opportunity?**
  Someone mentioned this on Piazza. With the prior knowledge that CUDA has some advanced FP16 arithmetic modules (tensor core) or algorithms that can speed things up, we decided to try it out.

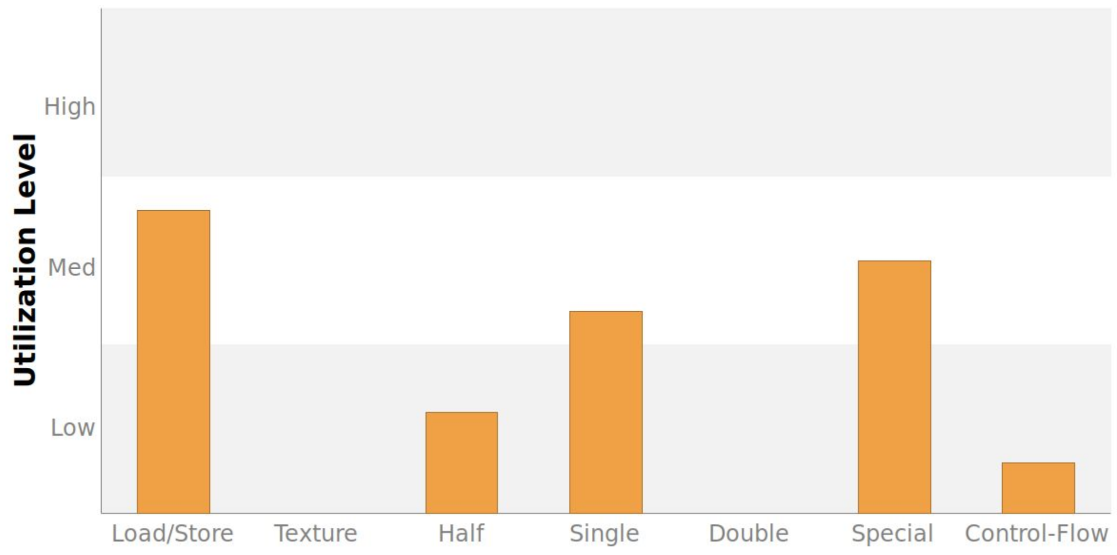
  Special message for Vikram and Zaid:
  While I don't know the exact probe or command to check if my half precision is done on tensor cores, based on my reading, I don't think it is. The fp16 computation relies on the library "cuda_fp16.h" while tensor core utilizations are defined in "mma.h". Also, if you go through the samples provided by CUDA, you can see that tensor core coding is much more complicated than half precision. I would definitely give it a try after the project is due or given more time.

- **Why you thought the approach would be fruitful?**
  The half-precision data format requires less memory usage. There are also vectorized forms of half precision variable such as half2. It combines two half

together and now we can use a 32-bit to represent two numbers. Also there are special arithmetic functions for half precision.

- **The effect of the optimization. was it fruitful, and why or why not. Use nvprof and NVVP to justify your explanation.**
  Half-precision optimization is very successful that we incorporated it into the final code. The following comparison is based on granularity of 2, one with FP16 and one without. Full precision achieves the Op time of 62 + 159 ms while half-precision has 27.4 + 68.2 ms. With half-precision computation involved, the load/store utilization level is dropped from high to medium. This is because with granularity of 2, there are a TILE_WIDTH x TILE_WIDTH x 2 amount of shared memory needed to store X value for full-precision, but with half-precision, two FP16 value is combined as a half2 datatype variable, reducing required shared memory usage by half. As for the memory bandwidth, the half-precision method achieved 18 GB/s, more than double of the full-precision, 7.73 GB/s. Note that all these methods are built upon the uncoalesced access so the number here is much smaller than expected.

  Below is a compute analysis utilization level analysis. Note the Half-precision part was not existed at all if fully using FP32. The usage increases as the granularity increases.



- **Any external references used during identification or development of the optimization**
  NVIDIA blog

Optimization 5: Sweeping various parameters to find best values

- **How you identified the optimization opportunity?**
  In an attempt to improve the memory access pattern, decrease control divergence and let the kernel fit each input layer better, we decided to assign different block size to different layers. After some experiments, 12 * 12 for the first layer and 24 * 24 for the second layer gives the best result.
  We have also tried different thread coarsening granularity and concluded with a granularity of 8.

- **Why you thought the approach would be fruitful?**
  If the input size is a multiple of the block size then it is clear that a lot of control divergence could be eliminated and the memory access in a warp could potentially be more coalesced.

- **The effect of the optimization. was it fruitful, and why or why not. Use nvprof and NVVP to justify your explanation.**
  We identify the tile width early on and added it to different versions, and we can generally see a 30 ms improvement in total. Later we swept the tile width for both kernels after we finalized all the optimization techniques. 12 and 24 is best best values for the two kernels. Values not divided by 4 can cause more control divergence while larger values, especially for the second stage, will make the register usage situation even worse, causing even less blocks per SM. For example, TILE_WIDTH of 28 for the second stage will increase the Op time from 40ms to 100ms, mainly from less blocks per SM.

Optimization 6: Register Tiling

- **How you identified the optimization opportunity?**
  Register tiling matrix multiplication is mentioned in the github page so we dived in for some research.

- **Why you thought the approach would be fruitful?**
  While double shared memory for both input matrices is fast, one can obviously see that there are many redundant loadings in the process. Consider a simple

case of matrix multiplication C = A * B. For each column in C, it requires the same column from B. However, for the shared memory method, we load the column of B once for each entry in the column of C. We can mitigate this problem by storing it directly in the registers and let the thread to take care the entire column of the C (assume that the matrix is not so big that data will fit nicely). Also, it is another way to increase the granularity of each thread. We can used so much less threads to finish the same job.

- **The effect of the optimization. was it fruitful, and why or why not. Use nvprof and NVVP to justify your explanation.**
  While this is the holy grail of this project, we stuck in the memory coalescing for too long and didn't have time to try this method.

- **Any external references used during identification or development of the optimization**

How your team organized and divided up this work.
Jingning wrote the code for milestone 1 to 4. For final round, he worked on the baseline fused baseline unrolling and matrix multiplication, data prefetching, half-precision and researched on register tiling. He also worked on generating profiling data and wrote the report.

Xiang worked with Jingning in milestone 1 to 4. He implemented separate strategies for milestone 3 and 4. As for the final project, he worked on the baseline and register tiling as well. He also spent time fixing the coalescing, parameter sweeping and worked on most profiling work and report.

Monil worked on some occasions for the first 3 milestones and initial exploration on shared memory convolution back in milestone 4 but did not work on final project.

Milestone 4

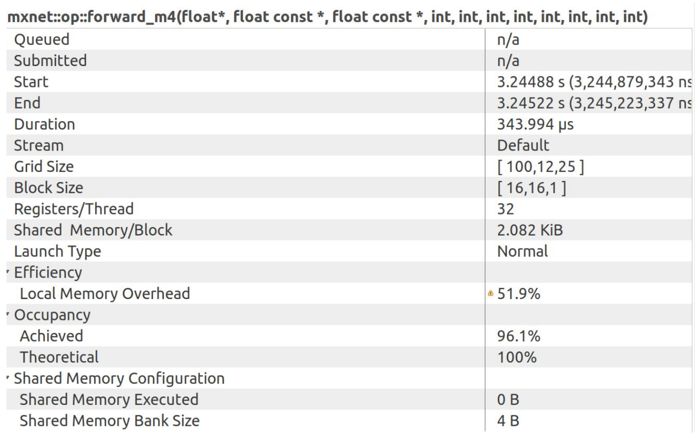
Optimizations we have tried:

1. Unroll the input matrix and use shared memory matrix multiplication to directly compute the convolution.
2. Constant memory for convolution kernel.
3. Unroll the for loop.
4. Shared memory to speed up the convolution

Note: In the following analysis the op time refers to that of running with 100 samples.

Result:
We first want to directly go for unroll and matrix multiplication because we believe that will be the fastest way to compute convolution if implemented correctly. However, the current unrolling implementation only unrolls one image at a time and will write back to global memory each time. For each input image x [C, H, W], we will unroll it to a matrix x_unroll [C, K*K, H_out*W_out], where H_out and W_out are the dimensions of output matrix y. Then we will perform the standard shared memory matrix multiplication as we did in the lab. As a result, unrolling and matrix multiplication uses separate kernels and we have to loop for each image to compute, and the op time is around 0.213 s and 0.468 s for two layers.

Later, we decided to try out the straightforward optimizations for the convolution to finish this milestone first. It involves constant memory for convolution kernel, unrolling the for loop and shared memory to compute the convolution. The constant memory and shared memory utilizations are highly similar to the lab and we also referred to the textbook heavily. For the unrolling for loop, simple #pragma unroll should do the trick. Combining these 3 optimizations, the op time is reduced by 20 us on the first forward layer with sample size 100.



mxnet::op::forward_m4(float*, float const *, float const *, int, int, int, int, int, int, int, int)

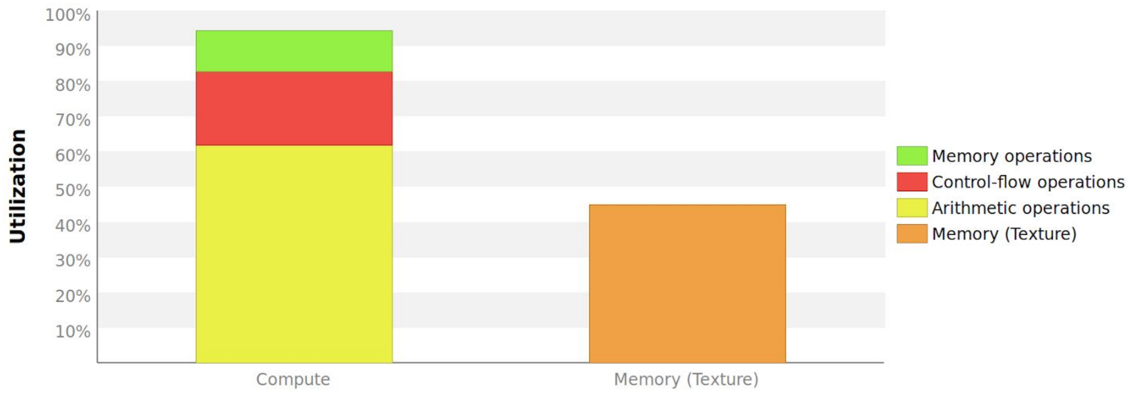
Queued	n/a
Submitted	n/a
Start	3.24488 s (3,244,879,343 ns
End	3.24522 s (3,245,223,337 ns
Duration	343.994 μs
Stream	Default
Grid Size	[100,12,25]
Block Size	[16,16,1]
Registers/Thread	32
Shared Memory/Block	2.082 KiB
Launch Type	Normal
Efficiency	
Local Memory Overhead	51.9%
Occupancy	
Achieved	96.1%
Theoretical	100%
Shared Memory Configuration	
Shared Memory Executed	0 B
Shared Memory Bank Size	4 B

Profiling:

- Shared memory
    a. We tried this optimization to increase memory utilization.
    b. For the shared memory, we only utilized about 2 kB, much less than the total allowed amount of 96 kB. We use only 256 threads per block instead of 1024 as last time. And we found more threads per block will actually decrease the performance.
    c. Divergence is a lot more (50% + 16.5% vs. 4%). This is no surprise because we have one more conditional line and also the program is not carefully calibrated.
    d. The compute and memory utilization become balanced.
    e. Shared memory utilization increased, which is aligned with the optimization.

    f. The final result is that execution time increased by about 10%. This is likely due to the overhead of syncthreads and the increase in divergence.

- Constant memory
    a. While the book decide to put both the filter bank and the input image into the shared memory, we put the input image in shared memory and put the filter bank in constant memory instead, hoping this would give better performance.
    b. However, due to the overhead of constant memory copy, the performance suffer a decrease. The execution time increased by about 35%.
- Loop unrolling
    a. With loop unrolling we get a major performance boost. The op time decreased by about half from 0.07 and 0.24 to 0.03 and 0.11.
    b. This could be because the shared memory usage and L2 cache usage both increased by 30 to 50 percent, although still lower than desired.



Compared to the milestone 3 implementation, the compute utilization is boosted in all three operations and by 17% (absolute difference) with constant memory, shared memory and loop unrolling combined. However, we are still bounded by computation.

- Unroll + matrix multiplication
    a. Due to the loss of parallelism mentioned above, there are too many kernel invocations and the overhead is considerable. The performance takes a major hit despite the result is correct.
    b. The op time become 0.19 and 0.36 from the original 0.03 and 0.11.
    c. Nvprof shows the GPU resources are underused because we are processing one input image at a time. Also, the unroll operation is taking 40% longer than the matrix multiplication.
    d. We used tiled matrix multiplication from the lab. Despite using shared memory it is still bound by memory latency. We hope the fusion of unroll and gemm kernel will solve this problem.

Next Step:

We are definitely not going to use the convolution for the final challenge. The problem right now for the unroll and matmul are looping through batches and separate kernel, so we can improve in the following ways:

1. Fuse the two kernels to bypass the device memory usage. We can either carefully compute the desired index for w and x or do the unrolling the store them into the shared memory. The shared memory can only store all the input for the first stage but not for the second and potential competition test. So maybe we can separately store them into register and shared memory and manage them, or just go with indexing.
2. Looping through batches. This is not correct since we should assign multiple grid to work on them instead of iterating. The current unrolling algorithm has to be expanded to include the batch number.
3. Explore the possibility to improve the matrix multiplication.

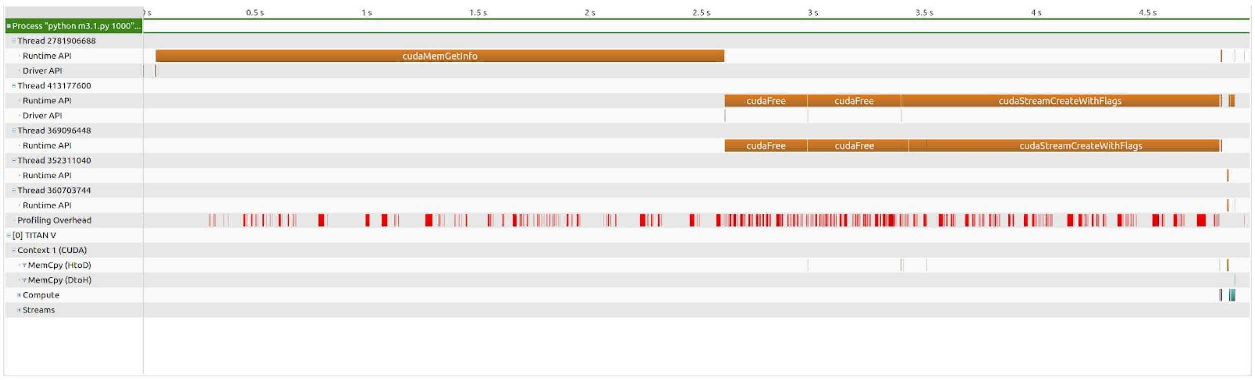
Milestone 3

1. Correctness and timing with 3 different dataset sizes

Dataset size	Correctness	Op time 1	Op time 2
100	0.85	0.000452	0.001633
1000	0.827	0.004298	0.016266
10000	0.8171	0.042824	0.148790

2. **nvprof**: profiling details
   Note that all the information are gathered based on the dataset size of 100. Both kernels should share similar timelines, performance patterns, while the properties in the analysis differ, as shown in the last section.

   Timeline:

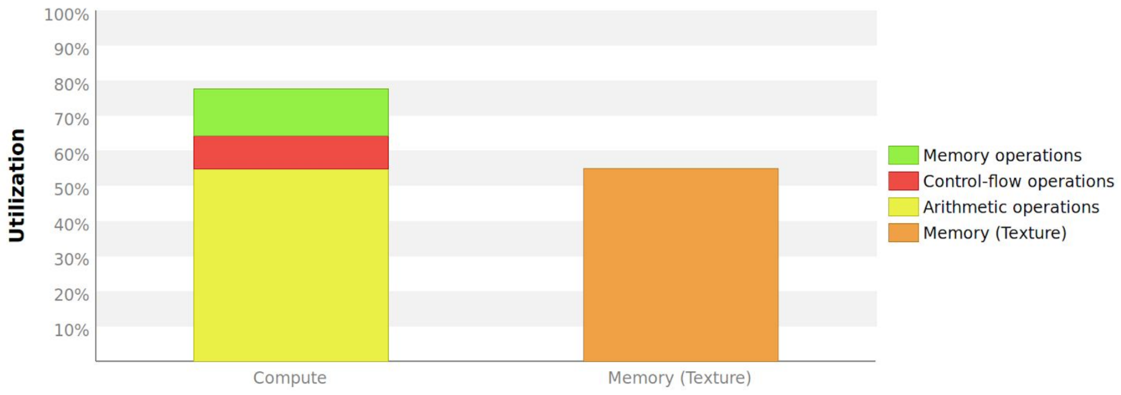


The timeline contains the time allocation for different API calls and kernel computation times. For the cleanness of the timeline, we didn't expand the compute tab above but instead briefly describe it below. The timeline is mostly occupied by cudaMemGetInfo, cudaFree and cudaStreamWithFlags, with some profiling overhead overlapping with them. The entire process takes 4.93s, but the computation only takes 23.79ms, roughly 0.5%.

Metrics:

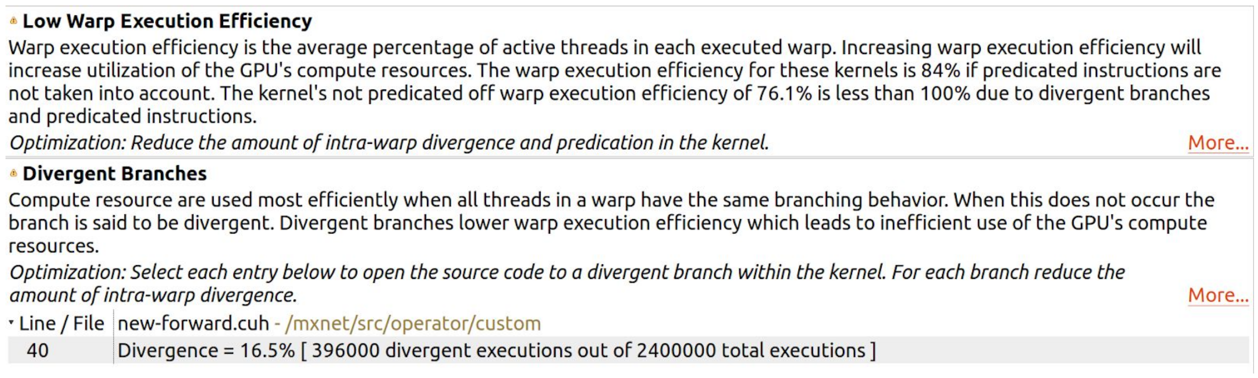
Kernel Performance Is Bound By Compute And Memory Bandwidth

For device "TITAN V" compute and memory utilization are balanced. These utilization levels indicate that kernel performance is good, but that additional performance improvement may be possible if either of both of compute and memory utilization levels are increased.



The overall analysis above suggests that compute and memory utilization are balanced. However, the detailed analysis, listed below, showed more information and improvement recommendations.

Compute Analysis



Low Warp Execution Efficiency

Warp execution efficiency is the average percentage of active threads in each executed warp. Increasing warp execution efficiency will increase utilization of the GPU's compute resources. The warp execution efficiency for these kernels is 84% if predicated instructions are not taken into account. The kernel's not predicated off warp execution efficiency of 76.1% is less than 100% due to divergent branches and predicated instructions.

Optimization: Reduce the amount of intra-warp divergence and predication in the kernel. More...

Divergent Branches

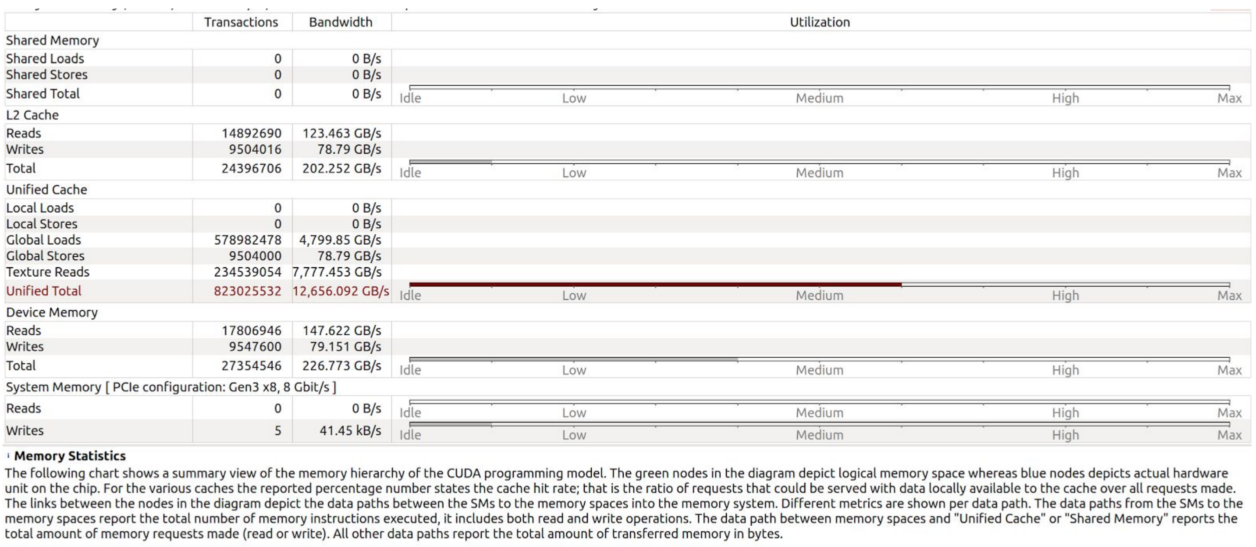
Compute resource are used most efficiently when all threads in a warp have the same branching behavior. When this does not occur the branch is said to be divergent. Divergent branches lower warp execution efficiency which leads to inefficient use of the GPU's compute resources.

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
40	Divergence = 16.5% [396000 divergent executions out of 2400000 total executions]

Above shows that control divergence happens at line 40, where the boundary condition check occurred. This is unavoidable from the algorithm side, but we can set the TILE_WIDTH to be the multiple of image size to avoid that, if possible.

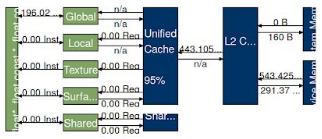
Memory Analysis



	Transactions	Bandwidth	Utilization
Shared Memory			
Shared Loads	0	0 B/s	
Shared Stores	0	0 B/s	
Shared Total	0	0 B/s	Idle Low Medium High Max
L2 Cache			
Reads	14892690	123.463 GB/s	
Writes	9504016	78.79 GB/s	
Total	24396706	202.252 GB/s	Idle Low Medium High Max
Unified Cache			
Local Loads	0	0 B/s	
Local Stores	0	0 B/s	
Global Loads	578982478	4,799.85 GB/s	
Global Stores	9504000	78.79 GB/s	
Texture Reads	234539054	7,777.453 GB/s	
Unified Total	823025532	12,656.092 GB/s	Idle Low Medium High Max
Device Memory			
Reads	17806946	147.622 GB/s	
Writes	9547600	79.151 GB/s	
Total	27354546	226.773 GB/s	Idle Low Medium High Max
System Memory [PCIe configuration: Gen3 x8, 8 Gbit/s]			
Reads	0	0 B/s	Idle Low Medium High Max
Writes	5	41.45 kB/s	Idle Low Medium High Max

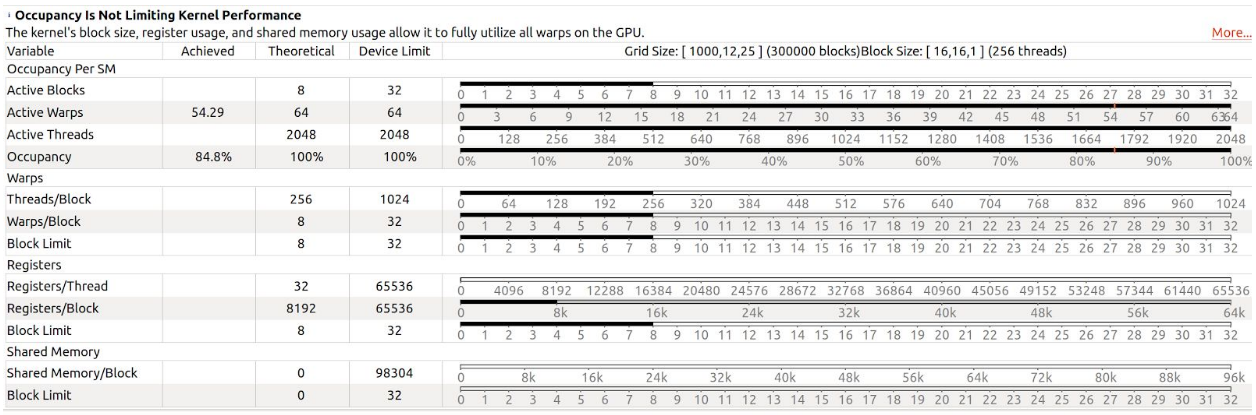
Memory Statistics

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.



The memory analysis shows that no utilization of shared memory and L2 cache, as showing above, and also points out the lines (45 and 49) where global memory access occurred. This is no surprise because we haven’t optimized anything yet.

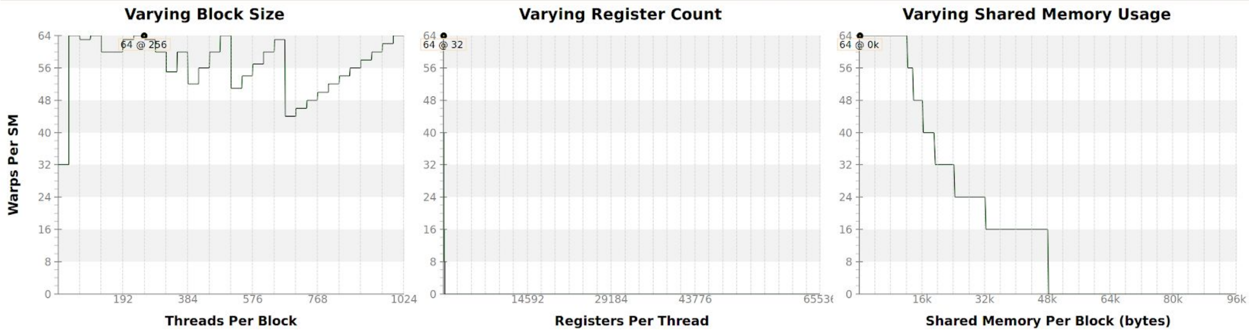
Latency Analysis



Occupancy Is Not Limiting Kernel Performance

The kernel's block size, register usage, and shared memory usage allow it to fully utilize all warps on the GPU. More...

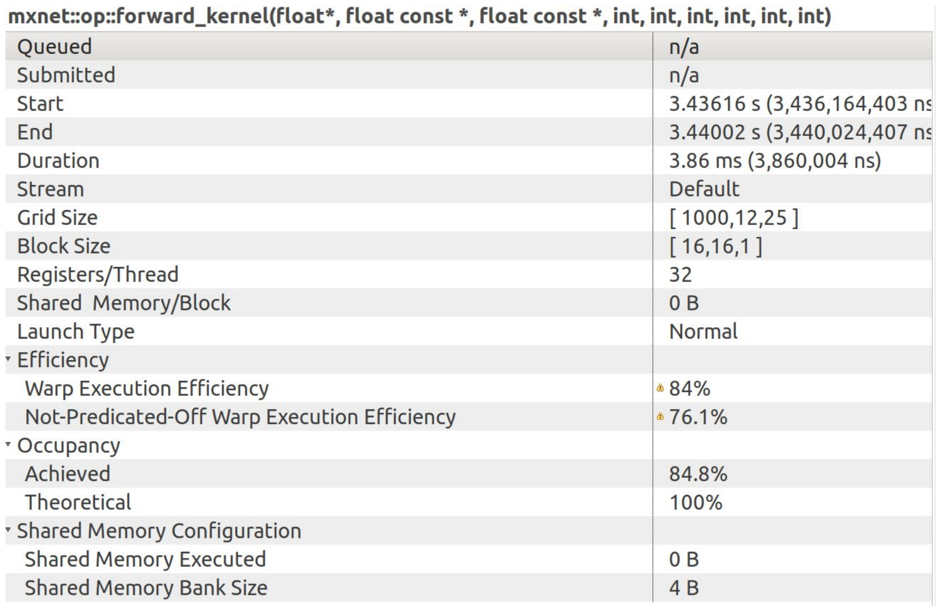
Variable	Achieved	Theoretical	Device Limit	Grid Size: [1000,12,25] (300000 blocks)Block Size: [16,16,1] (256 threads)
Occupancy Per SM				
Active Blocks		8	32	
Active Warps	54.29	64	64	
Active Threads		2048	2048	
Occupancy	84.8%	100%	100%	
Warps				
Threads/Block		256	1024	
Warps/Block		8	32	
Block Limit		8	32	
Registers				
Registers/Thread		32	65536	
Registers/Block		8192	65536	
Block Limit		8	32	
Shared Memory				
Shared Memory/Block		0	98304	
Block Limit		0	32	



The latency analysis shows we have not fully used the thread/block, registers/block and shared memory yet.

Also, we can see the metrics' difference between forward 1 and forward 2. Forward 2 takes fewer grids but a longer time to finish.

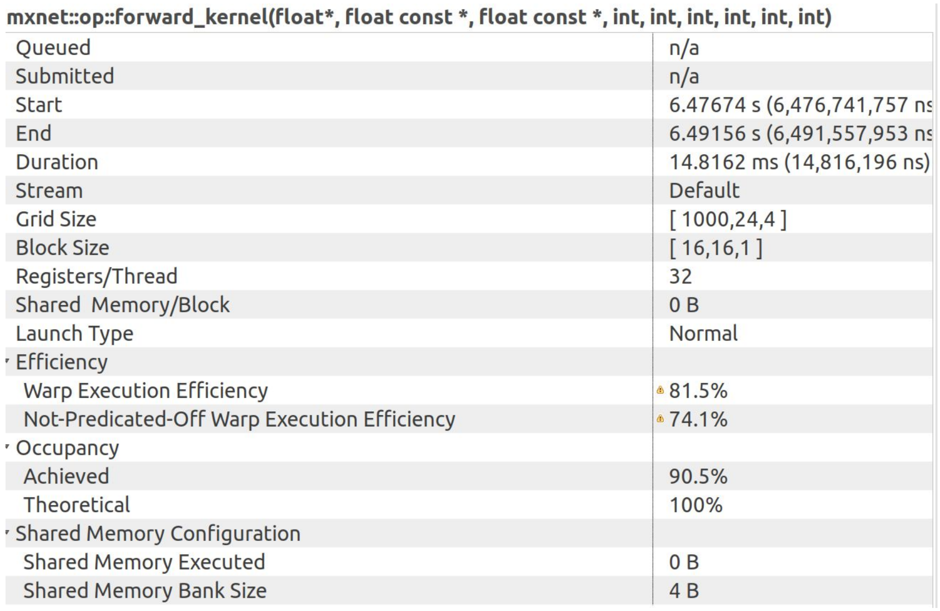
Forward 1



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

Queued	n/a
Submitted	n/a
Start	3.43616 s (3,436,164,403 ns
End	3.44002 s (3,440,024,407 ns
Duration	3.86 ms (3,860,004 ns)
Stream	Default
Grid Size	[1000,12,25]
Block Size	[16,16,1]
Registers/Thread	32
Shared Memory/Block	0 B
Launch Type	Normal
Efficiency	
Warp Execution Efficiency	84%
Not-Predicated-Off Warp Execution Efficiency	76.1%
Occupancy	
Achieved	84.8%
Theoretical	100%
Shared Memory Configuration	
Shared Memory Executed	0 B
Shared Memory Bank Size	4 B

Forward 2



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

Queued	n/a
Submitted	n/a
Start	6.47674 s (6,476,741,757 ns
End	6.49156 s (6,491,557,953 ns
Duration	14.8162 ms (14,816,196 ns)
Stream	Default
Grid Size	[1000,24,4]
Block Size	[16,16,1]
Registers/Thread	32
Shared Memory/Block	0 B
Launch Type	Normal
Efficiency	
Warp Execution Efficiency	81.5%
Not-Predicated-Off Warp Execution Efficiency	74.1%
Occupancy	
Achieved	90.5%
Theoretical	100%
Shared Memory Configuration	
Shared Memory Executed	0 B
Shared Memory Bank Size	4 B

Milestone 2

1. **List whole program execution time**

user: 132.44s, system: 4.41s, elapsed: 2min 6.72s

2. **List op time**

Op time: 21.280921

Op time: 101.222768

Milestone 1

Team Name: tensorjammed
Team Members:

Name	NetID	School
Jingning Tang	jtang10	UIUC
Monil Pathak	mpathak2	UIUC
Xiang Li	xiangl12	UIUC

1. Include a list of all kernels that collectively consume more than 90% of the program time.

Time (%)	Time (ms)	Name
~~36.84%~~	~~37.628~~	~~CUDA memcpy HtoD~~
22.70%	23.185	volta_scudnn_128x32_relu_interior_nn_v1
20.80%	21.248	cudnn::detail::implicit_convolve_sgemm
7.39%	7.546	volta_sgemm_128x128_tn
7.24%	7.391	cudnn::detail::activation_fw_4d_kernel
4.31%	4.404	cudnn::detail::pooling_fw_4d_kernel
99.28% (in total)	101.402 (in total)	

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

Time (%)	Time (ms)	Name
40.47%	37.628	cudaStreamCreateWithFlags
33.77%	23.185	cudaMemGetInfo
22.04%	21.248	cudaFree
96.28% (in total)	82.061 (in total)	

The memcpy operation appears in both GPU activities and API calls. From our understanding, it is obviously an API call provided by CUDA. It can be part of the GPU activities but might not be part of the kernel actions as we commonly define. As described in the following question, kernel is usually defined as the programmer-defined function running on different threads on GPU.

3. Include an explanation of the difference between kernels and API calls

Kernels are programmer-defined functions called by the host running on multiple GPU threads in parallel, whereas API calls are CUDA-defined functions called by the host (CPU).

To be more specific, kernel is the code defined by the programmer and loaded onto N different threads GPU. In the baseline model, they include convolution, non-linear activations (ReLU and tanh), pooling and matrix multiplications (sgemm).

API calls are the functions provided by CUDA and called on the host side for operations including device query, cudaMalloc, cudaMemcpy, and cudaLaunchKernel, etc.

4. Show output of rai running MXNet on the CPU

```
✱ Running /usr/bin/time python m1.1.py
Loading fashion-mnist data... done
Loading model... done
New Inference
EvalMetric: {'accuracy': 0.8177}
20.09user 4.03system 0:13.44elapsed 179%CPU (0avgtext+0avgdata
5955964maxresident)k
0
inputs+2856outputs (0major+1585730minor)pagefaults 0swaps
```

5. List program run time

user: 20.09s, system: 4.03s, elapsed: 13.44s

6. Show output of rai running MXNet on the GPU

```
✱ Running /usr/bin/time python m1.2.py
Loading fashion-mnist data... done
Loading model... done
New Inference
EvalMetric: {'accuracy': 0.8177}
4.30user 2.70system 0:04.58elapsed 153%CPU (0avgtext+0avgdata
2849680maxresident)k
0inputs+
4568outputs (0major+708272minor)pagefaults 0swaps
```

7. List program run time

user: 4.30s, system: 0.04s, elapsed: 4.58s