ECE 408 Applied Parallel Programming
Fall 2019 Report

1 Milestone 2

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

Solution
[*CUDA memcpy HtoD*] 30.05%
volta_scudnn_128x64_relu_interior_nn_v1 17.81%
volta_gcgemm_64x32_nt 17.10%
void fft2d_c2r_32x32 8.56%
volta_sgemm_128x128_tn 7.79%
void op_generic_tensor_kernel 6.50%
void fft2d_r2c_32x32 5.72%

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

Solution
cudaStreamCreateWithFlags 43.17%
cudaMemGetInfo 31.31%
cudaFree 21.65%

(c) Include an explanation of the difference between kernels and API calls.

Solution
Kernel calls use less time than API calls. API calls in this piece of code will call kernel. So they will use up more time.

(d) Show output of rai running MXNet on the CPU.

Solution
Loading fashion-mnist data... done
Loading model... done
New Inference
EvalMetric: 'accuracy': 0.8154
17.40user 4.58system 0:09.00elapsed 244%CPU (0avgtext+0avgdata 6046604maxresident)k
0inputs+2824outputs (0major+1603908minor)pagefaults 0swaps

(e) List program run time

Solution
user 17.46
system 4.65
real 0:09.07 elapsed

(f) Show output of rai running MXNet on the CPU.

Solution
Loading fashion-mnist data... done
Loading model... done
New Inference
EvalMetric: 'accuracy': 0.8154
5.11user 3.30system 0:04.68elapsed 179%CPU (0avgtext+0avgdata 2970344maxresident)k
0inputs+1712outputs (0major+732652minor)pagefaults 0swaps

(g) List program run time

Solution
user 5.11
system 3.30
real 0:04.68 elapsed

(h) List whole program execution time

Solution
user 100.89
system 8.12
real 1:31.66 elapsed

(i) List Op Times

Solution
Op Time: 12.051926
Op Time: 75.994165

2 Milestone 3

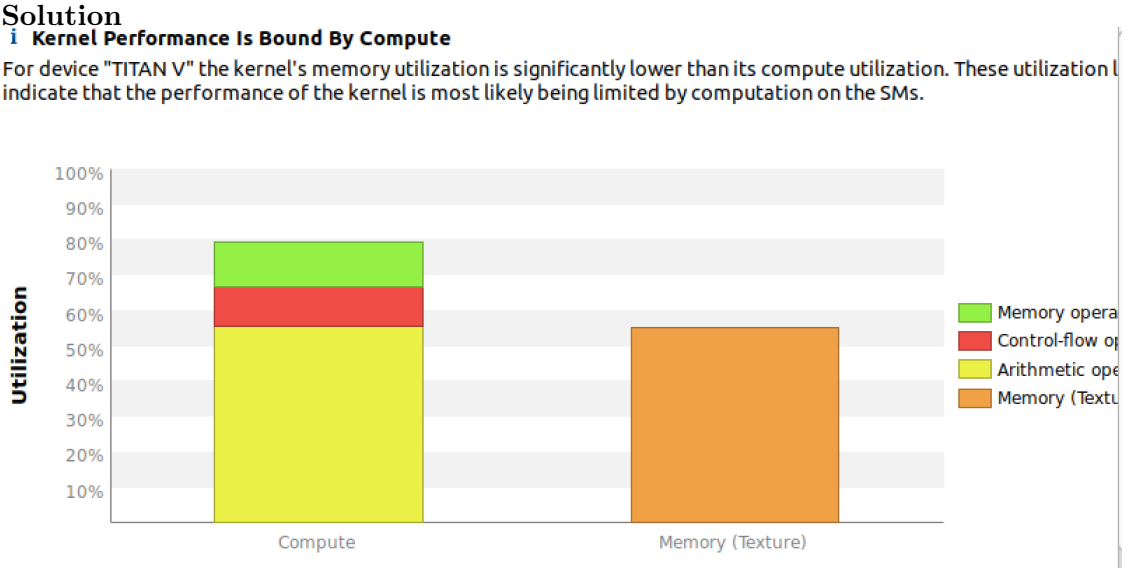
(j) Correctness and time with 3 different dataset sizes

Solution

Size	*User*	*System*	*Real*	*Correctness*
100	4.94	3.11	0 : 04.36	0.76
1000	7.14	3.68	0 : 07.61	0.767
10000	5.35	3.03	0 : 05.07	0.7653

(k) nvprof profiling

Solution



This kernel exhibits low compute throughput and memory bandwidth utilization, which are below 60%. As we can see from the graph, arithmetic and memory operation take up most of utilization. So its performance is most likely limited by the latency of arithmetic or memory operations.

Occupancy Is Not Limiting Kernel Performance

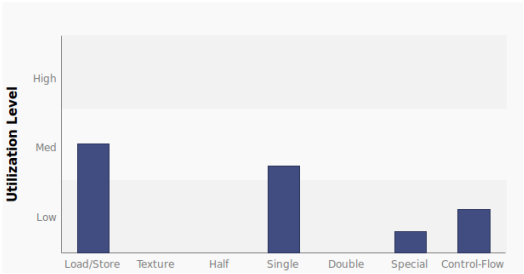
The kernel's block size, register usage, and shared memory usage allow it to fully utilize all warps on the GPU. More...

Variable	Achieved	Theoretical	Device Limit	Grid Size: [100,12,25] (30000 blocks)Block Size: [16,16,1] (256 threads)
Occupancy Per SM				
Active Blocks		8	32	0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32
Active Warps	53.8	64	64	0 4 8 12 16 20 24 28 32 36 40 44 48 52 56 60 64
Active Threads		2048	2048	0 256 512 768 1024 1280 1536 1792 2048
Occupancy	84.1%	100%	100%	0% 15% 30% 45% 60% 75% 90% 100%
Warps				
Threads/Block		256	1024	0 128 256 384 512 640 768 896 1024
Warps/Block		8	32	0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32
Block Limit		8	32	0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32
Registers				
Registers/Thread		32	65536	0 8192 16384 24576 32768 40960 49152 57344 65536
Registers/Block		8192	65536	0 8k 16k 24k 32k 40k 48k 56k 64k
Block Limit		8	32	0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32
Shared Memory				
Shared Memory/Block		0	98304	0 16k 32k 48k 64k 80k 96k
Block Limit		0	32	0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32

It shows that the kernel's block size, register usage, and shared memory usage allow it to fully utilize all warps on the GPU. But we are only using part of them due to control divergence.

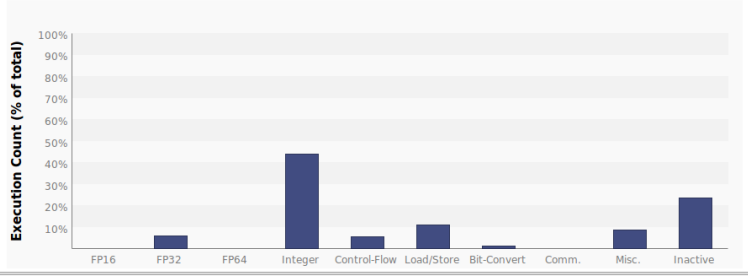
Results

Control-Flow - Direct and indirect branches, jumps, and calls.



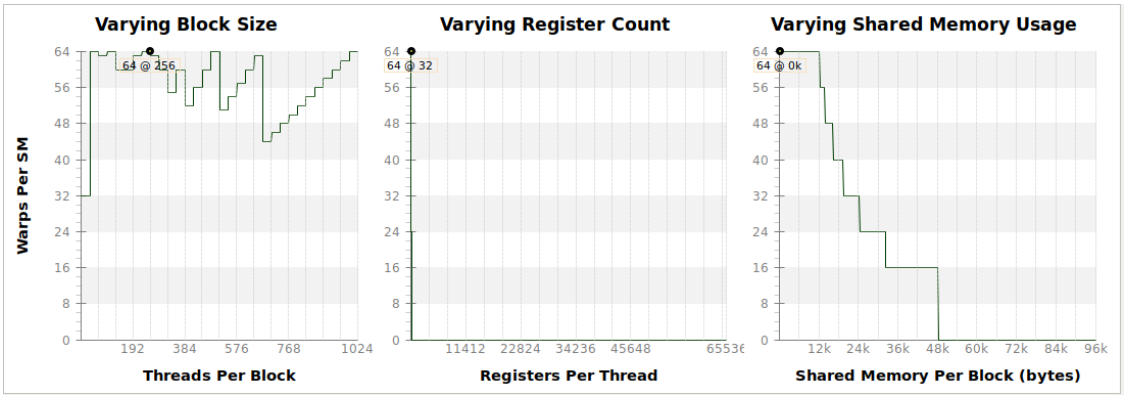
Instruction Execution Counts

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.



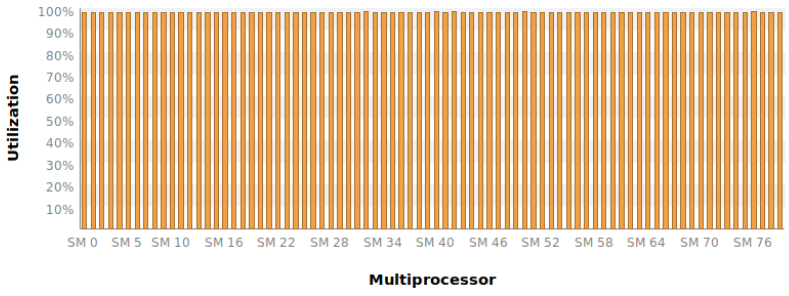
Floating-Point Operation Counts

The following chart shows the mix of floating-point operations executed by the kernel. The operations are grouped into classes and for each class the chart shows the percentage of thread execution cycles that were devoted to executing operations in that class. The results do not sum to 100% because non-floating-point operations executed by the kernel are not shown in this chart.



Multiprocessor Utilization

The kernel's blocks are distributed across the GPU's multiprocessors for execution. Depending on the number of blocks and the execution duration of each block some multiprocessors may be more highly utilized than others during execution of the kernel. The following chart shows the utilization of each multiprocessor during execution of the kernel. More...



Global Memory Alignment and Access Pattern

Memory bandwidth is used most efficiently when each global memory load and store has proper alignment and access pattern. The analysis is per assembly instruction.

Optimization: Select each entry below to open the source code to a global load or store within the kernel with an inefficient alignment or access pattern. For each load or store improve the alignment and access pattern of the memory access. More...

Line / File	new-forward.cuh - /mxnet/src/operator/custom
45	Global Load L2 Transactions/Access = 4.8, Ideal Transactions/Access = 3.4 [4752000 L2 transactions for 990000 total executions]
45	Global Load L2 Transactions/Access = 5.3, Ideal Transactions/Access = 3.4 [5247000 L2 transactions for 990000 total executions]
45	Global Load L2 Transactions/Access = 5.3, Ideal Transactions/Access = 3.4 [5247000 L2 transactions for 990000 total executions]
45	Global Load L2 Transactions/Access = 4.8, Ideal Transactions/Access = 3.4 [4752000 L2 transactions for 990000 total executions]
45	Global Load L2 Transactions/Access = 4.8, Ideal Transactions/Access = 3.4 [4752000 L2 transactions for 990000 total executions]
47	Global Store L2 Transactions/Access = 4.8, Ideal Transactions/Access = 3.4 [950400 L2 transactions for 198000 total executions]

Memory Bandwidth And Utilization

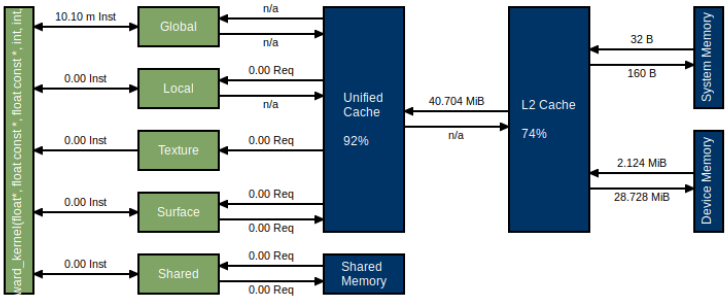
The following table shows the memory bandwidth used by this kernel for the various types of memory on the device. The table also shows the utilization of each memory type relative to the maximum throughput supported by the memory. More...

	Transactions	Bandwidth	Utilization
Shared Memory			
Shared Loads	0	0 B/s	
Shared Stores	0	0 B/s	
Shared Total	0	0 B/s	Idle Low Medium High Max
L2 Cache			
Reads	1348082	191.289 GB/s	
Writes	950429	134.863 GB/s	
Total	2298511	326.153 GB/s	Idle Low Medium High Max
Unified Cache			
Local Loads	0	0 B/s	
Local Stores	0	0 B/s	
Global Loads	29686667	4,212.462 GB/s	
Global Stores	950400	134.859 GB/s	
Texture Reads	12630813	7,169.12 GB/s	
Unified Total	43267880	11,516.441 GB/s	Idle Low Medium High Max
Device Memory			
Reads	69584	9.874 GB/s	
Writes	941356	133.576 GB/s	
Total	1010940	143.45 GB/s	Idle Low Medium High Max
System Memory [PCIe configuration: Gen3 x16, 8 Gbit/s]			
Reads	1	141.897 kB/s	Idle Low Medium High Max
Writes	5	709.487 kB/s	Idle Low Medium High Max

As the table shows, our kernel do not use any shared memory. The utilization of L2 cache and device memory is low relative to the maximum throughput supported by the corresponding memory.

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.



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 84.8% if predicated instructions are not taken into account. The kernel's not predicated off warp execution efficiency of 76.6% 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
41	Divergence = 16.5% [39600 divergent executions out of 240000 total executions]

The report indicates that divergent executions in our kernel account for 16.5% of total excecutions. Therefore, divergent branches lower warp execution efficiency, which leads to inefficient use of the GPU's compute resources.

3 Milestone 4

(a) Unroll + shared-memory Matrix multiply

Solution

In this optimization, we prepared an unrolled input feature map (X_unrolled) before performing Matrix multiplication. During the unrolling, we mapped elements in feature map to the unrolled matrix, this would duplicate elements in feature map and occupy more memory. However, the design utilizes a memory write coalescing pattern as every output is derived from the input feature map elements. In the Matrix multiply part we

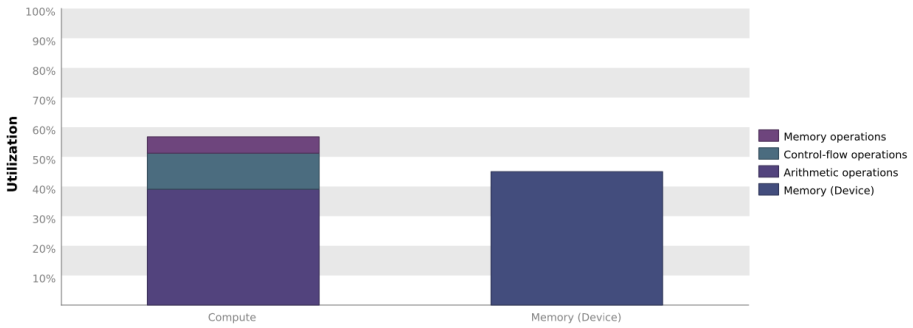
used shared-memory to optimization.
After the optimization, the Op time increased, one major contributor to the Op time is that the unroll kernel copy from original feature matrix.
The Op times for size 10000 dataset are: 0.129585 and 0.249899.
There are two kernels, the first one copy feature matrix to unroll matrix.

1. Compute, Bandwidth, or Latency Bound

The first step in analyzing an individual kernel is to determine if the performance of the kernel is bounded by computation, memory bandwidth, or instruction/memory latency. The results below indicate that the performance of kernel "mxnet::op::forward_kernel" is most likely limited by instruction and memory latency. You should first examine the information in the "Instruction And Memory Latency" section to determine how it is limiting performance.

1.1. Kernel Performance Is Bound By Instruction And Memory Latency

This kernel exhibits low compute throughput and memory bandwidth utilization relative to the peak performance of "TITAN V". These utilization levels indicate that the performance of the kernel is most likely limited by the latency of arithmetic or memory operations. Achieved compute throughput and/or memory bandwidth below 60% of peak typically indicates latency issues.



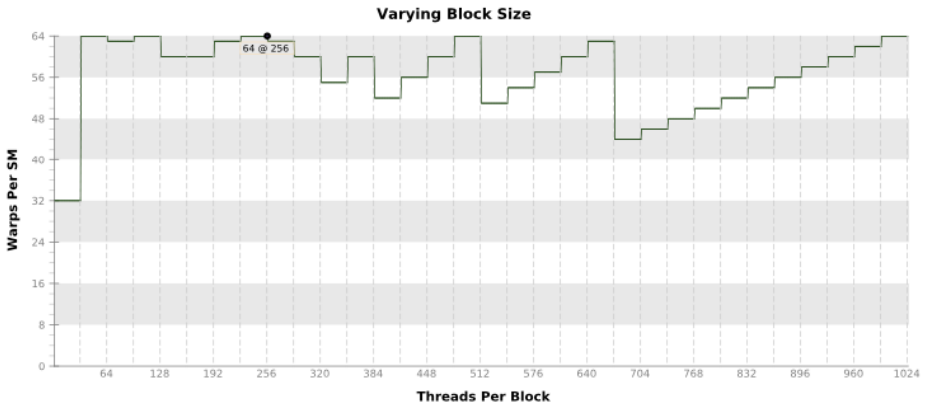
2. Instruction and Memory Latency

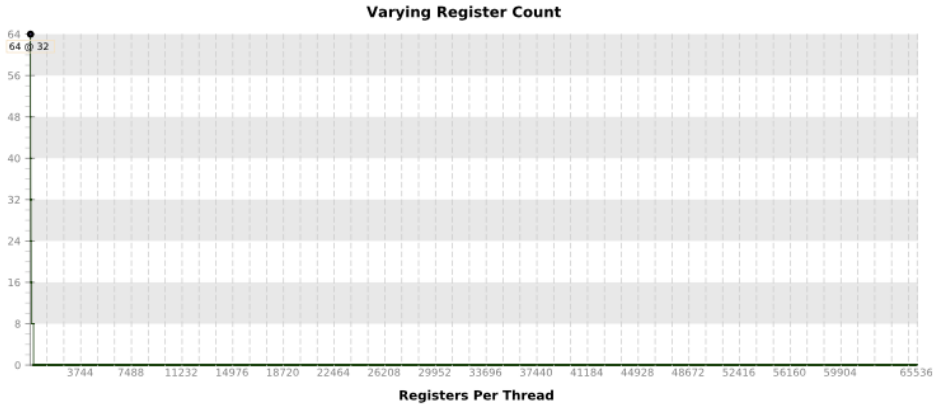
Instruction and memory latency limit the performance of a kernel when the GPU does not have enough work to keep busy. The performance of latency-limited kernels can often be improved by increasing occupancy. Occupancy is a measure of how many warps the kernel has active on the GPU, relative to the maximum number of warps supported by the GPU. Theoretical occupancy provides an upper bound while achieved occupancy indicates the kernel's actual occupancy.

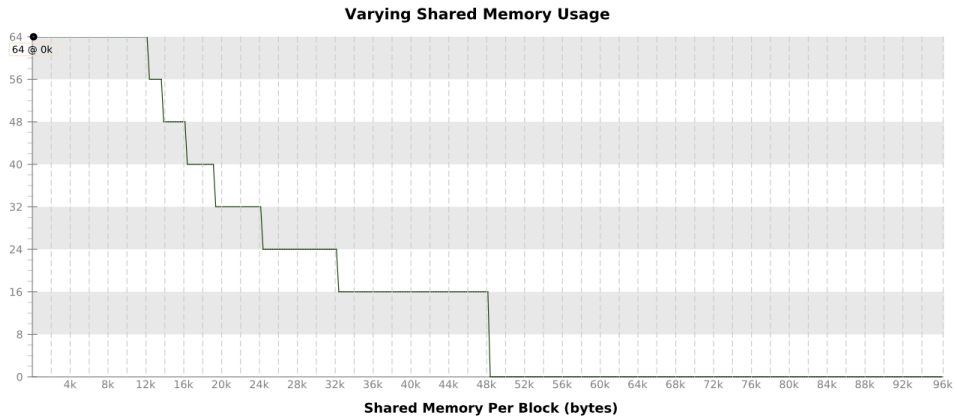
2.1. Occupancy Is Not Limiting Kernel Performance

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

Variable	Achieved	Theoretical	Device Limit	Grid Size: [10000,12,25] (3000000 blocks) Block Size:
Occupancy Per SM				
Active Blocks		8	32	0 4 8 12 16 20 24 28 32
Active Warps	51.71	64	64	0 9 18 27 36 45 54 64
Active Threads		2048	2048	0 512 1024 1536 2048
Occupancy	80.8%	100%	100%	0% 25% 50% 75% 100%
Warps				
Threads/Block		256	1024	0 256 512 768 1024
Warps/Block		8	32	0 4 8 12 16 20 24 28 32
Block Limit		8	32	0 4 8 12 16 20 24 28 32
Registers				
Registers/Thread		32	65536	0 16384 32768 49152 65536
Registers/Block		8192	65536	0 16k 32k 48k 64k
Block Limit		8	32	0 4 8 12 16 20 24 28 32
Shared Memory				
Shared Memory/Block		0	98304	0 32k 64k 96k
Block Limit		0	32	0 4 8 12 16 20 24 28 32

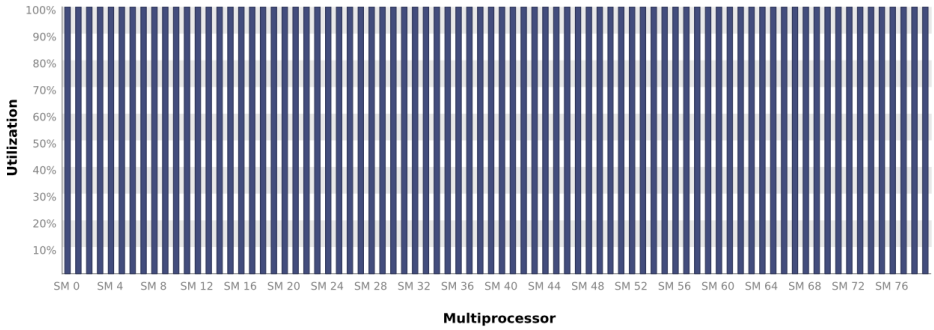

Varying Block Size
Warps Per SM
64 @ 256
0
8
16
24
32
40
48
56
64
64
128
192
256
320
384
448
512
576
640
704
768
832
896
960
1024
Threads Per Block


Varying Register Count
64 @ 32
0
8
16
24
32
40
48
56
64
3744
7488
11232
14976
18720
22464
26208
29952
33696
37440
41184
44928
48672
52416
56160
59904
65536
Registers Per Thread


Varying Shared Memory Usage
64 @ 0k
0
8
16
24
32
40
48
56
64
4k
8k
12k
16k
20k
24k
28k
32k
36k
40k
44k
48k
52k
56k
60k
64k
68k
72k
76k
80k
84k
88k
92k
96k
Shared Memory Per Block (bytes)

2.3. Multiprocessor Utilization

The kernel's blocks are distributed across the GPU's multiprocessors for execution. Depending on the number of blocks and the execution duration of each block some multiprocessors may be more highly utilized than others during execution of the kernel. The following chart shows the utilization of each multiprocessor during execution of the kernel.



3. Compute Resources

GPU compute resources limit the performance of a kernel when those resources are insufficient or poorly utilized. Compute resources are used most efficiently when all threads in a warp have the same branching and predication behavior. The results below indicate that a significant fraction of the available compute performance is being wasted because branch and predication behavior is differing for threads within a warp.

3.1. 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 84.9% if predicated instructions are not taken into account. The kernel's not predicated off warp execution efficiency of 76.4% is less than 100% due to divergent branches and predicated instructions.

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

3.2. 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: Each entry below points to a divergent branch within the kernel. For each branch reduce the amount of intra-warp divergence.

/mxnet/src/operator/custom/./new-forward.cuh

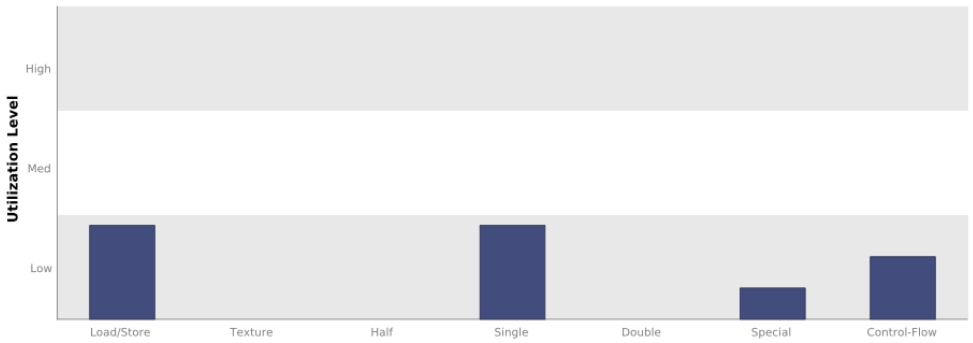
Line 32	Divergence = 16.5% [3960000 divergent executions out of 24000000 total executions]
Line 33	Divergence = 0% [0 divergent executions out of 19800000 total executions]
Line 33	Divergence = 0% [0 divergent executions out of 19800000 total executions]
Line 34	Divergence = 0% [0 divergent executions out of 99000000 total executions]
Line 34	Divergence = 0% [0 divergent executions out of 19800000 total executions]
Line 35	Divergence = 0% [0 divergent executions out of 99000000 total executions]
Line 35	Divergence = 0% [0 divergent executions out of 99000000 total executions]
Line 35	Divergence = 0% [0 divergent executions out of 99000000 total executions]
Line 35	Divergence = 0% [0 divergent executions out of 99000000 total executions]
Line 35	Divergence = 0% [0 divergent executions out of 99000000 total executions]
Line 35	Divergence = 0% [0 divergent executions out of 99000000 total executions]
Line 36	Divergence = 0% [0 divergent executions out of 99000000 total executions]
Line 36	Divergence = 0% [0 divergent executions out of 99000000 total executions]
Line 44	Divergence = 0% [0 divergent executions out of 19800000 total executions]

3.3. Function Unit Utilization

Different types of instructions are executed on different function units within each SM. Performance can be limited if a function unit is over-used by the instructions executed by the kernel. The following results show that the kernel's performance is not limited by overuse of any function unit.

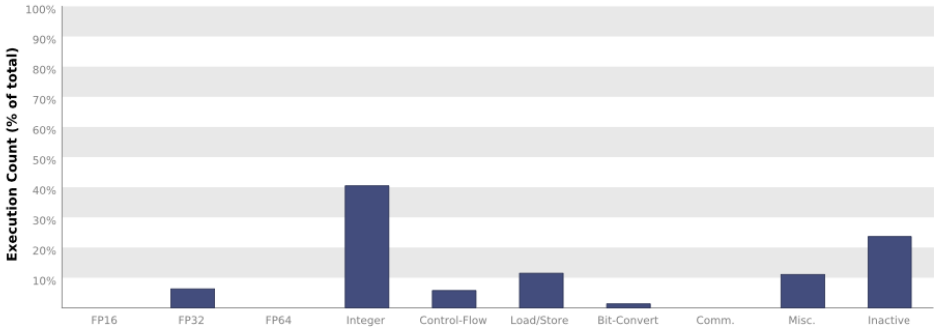
Load/Store - Load and store instructions for shared and constant memory.
Texture - Load and store instructions for local, global, and texture memory.
Half - Half-precision floating-point arithmetic instructions.
Single - Single-precision integer and floating-point arithmetic instructions.
Double - Double-precision floating-point arithmetic instructions.

Special - Special arithmetic instructions such as sin, cos, popc, etc.
Control-Flow - Direct and indirect branches, jumps, and calls.



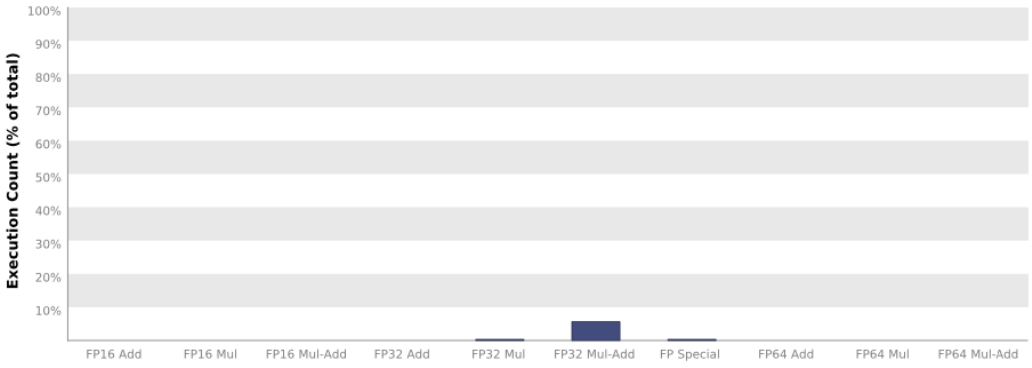
3.4. Instruction Execution Counts

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.



3.5. Floating-Point Operation Counts

The following chart shows the mix of floating-point operations executed by the kernel. The operations are grouped into classes and for each class the chart shows the percentage of thread execution cycles that were devoted to executing operations in that class. The results do not sum to 100% because non-floating-point operations executed by the kernel are not shown in this chart.



4. Memory Bandwidth

Memory bandwidth limits the performance of a kernel when one or more memories in the GPU cannot provide data at the rate requested by the kernel.

4.1. 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.

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	151466239	159.673 GB/s	
Writes	95040038	100.19 GB/s	
Total	246506277	259.863 GB/s	Idle Low Medium High Max
Unified Cache			
Local Loads	0	0 B/s	
Local Stores	0	0 B/s	
Global Loads	2474928955	2,609.027 GB/s	
Global Stores	95040000	100.19 GB/s	
Texture Reads	791165061	3,336.129 GB/s	
Unified Total	3361134016	6,045.346 GB/s	Idle Low Medium High Max
Device Memory			
Reads	201776702	212.709 GB/s	
Writes	95094655	100.247 GB/s	
Total	296871357	312.957 GB/s	Idle Low Medium High Max
System Memory			
[PCIe configuration: Gen3 x16, 8 Gbit/s]			
Reads	0	0 B/s	Idle Low Medium High Max
Writes	5	5.27 kB/s	Idle Low Medium High Max

4.2. 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.

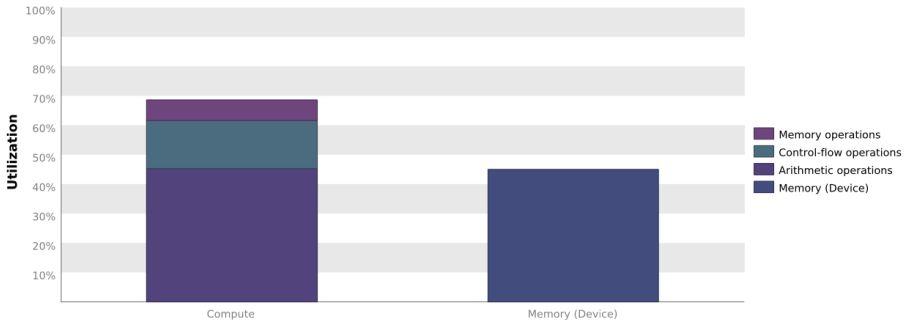
The NVVP analysed matrix multiplication kernel.

1. Compute, Bandwidth, or Latency Bound

The first step in analyzing an individual kernel is to determine if the performance of the kernel is bounded by computation, memory bandwidth, or instruction/memory latency. The results below indicate that the performance of kernel "mxnet::op::forward_kernel" is most likely limited by compute. You should first examine the information in the "Compute Resources" section to determine how it is limiting performance.

1.1. 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.



2. Compute Resources

GPU compute resources limit the performance of a kernel when those resources are insufficient or poorly utilized. Compute resources are used most efficiently when all threads in a warp have the same branching and predication behavior. The results below indicate that a significant fraction of the available compute performance is being wasted because branch and predication behavior is differing for threads within a warp.

2.1. 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 87.8% if predicated instructions are not taken into account. The kernel's not predicated off warp execution efficiency of 79.5% is less than 100% due to divergent branches and predicated instructions.

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

2.2. 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: Each entry below points to a divergent branch within the kernel. For each branch reduce the amount of intra-warp divergence.

/mxnet/src/operator/custom/./new-forward.cuh

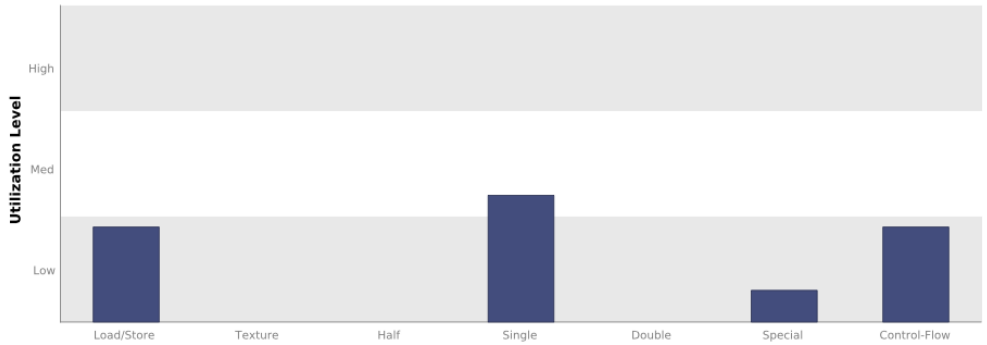
Line 32	Divergence = 50% [3840000 divergent executions out of 7680000 total executions]
Line 33	Divergence = 0% [0 divergent executions out of 86400000 total executions]
Line 33	Divergence = 0% [0 divergent executions out of 7200000 total executions]
Line 34	Divergence = 0% [0 divergent executions out of 432000000 total executions]
Line 34	Divergence = 0% [0 divergent executions out of 86400000 total executions]
Line 35	Divergence = 0% [0 divergent executions out of 432000000 total executions]
Line 35	Divergence = 0% [0 divergent executions out of 432000000 total executions]
Line 35	Divergence = 0% [0 divergent executions out of 432000000 total executions]
Line 35	Divergence = 0% [0 divergent executions out of 432000000 total executions]
Line 35	Divergence = 0% [0 divergent executions out of 432000000 total executions]
Line 35	Divergence = 0% [0 divergent executions out of 432000000 total executions]
Line 36	Divergence = 0% [0 divergent executions out of 432000000 total executions]
Line 36	Divergence = 0% [0 divergent executions out of 432000000 total executions]
Line 44	Divergence = 0% [0 divergent executions out of 7200000 total executions]

2.3. Function Unit Utilization

Different types of instructions are executed on different function units within each SM. Performance can be limited if a function unit is over-used by the instructions executed by the kernel. The following results show that the kernel's performance is not limited by overuse of any function unit.

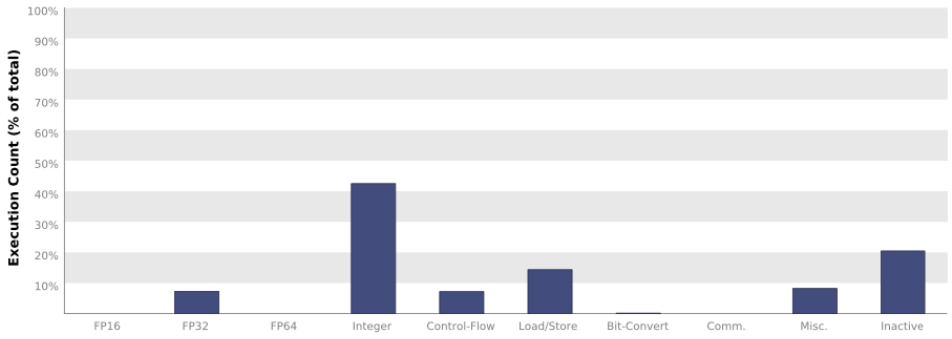
Load/Store - Load and store instructions for shared and constant memory.
Texture - Load and store instructions for local, global, and texture memory.
Half - Half-precision floating-point arithmetic instructions.
Single - Single-precision integer and floating-point arithmetic instructions.
Double - Double-precision floating-point arithmetic instructions.

Special - Special arithmetic instructions such as sin, cos, popc, etc.
Control-Flow - Direct and indirect branches, jumps, and calls.



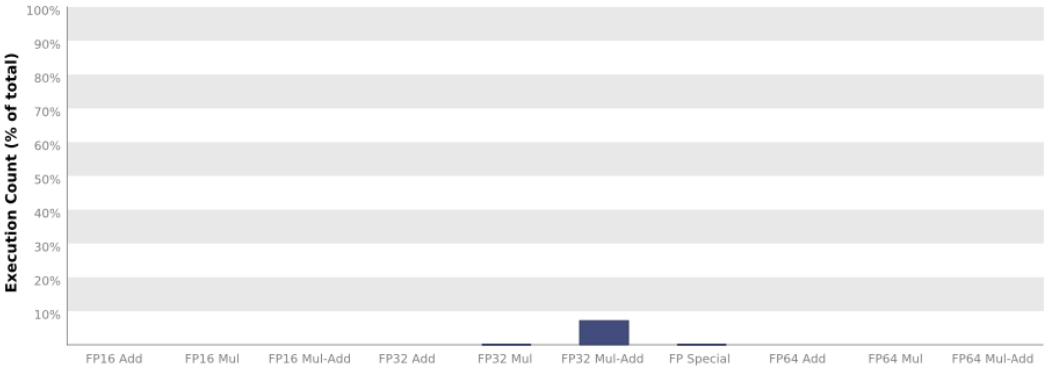
2.4. Instruction Execution Counts

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.



2.5. Floating-Point Operation Counts

The following chart shows the mix of floating-point operations executed by the kernel. The operations are grouped into classes and for each class the chart shows the percentage of thread execution cycles that were devoted to executing operations in that class. The results do not sum to 100% because non-floating-point operations executed by the kernel are not shown in this chart.



3. Memory Bandwidth

Memory bandwidth limits the performance of a kernel when one or more memories in the GPU cannot provide data at the rate requested by the kernel.

3.1. 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.

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	679654349	241.281 GB/s	
Writes	37410516	13.281 GB/s	
Total	717064865	254.562 GB/s	Idle Low Medium High Max
Unified Cache			
Local Loads	0	0 B/s	
Local Stores	0	0 B/s	
Global Loads	11205387000	3,977.969 GB/s	
Global Stores	37410000	13.281 GB/s	
Texture Reads	3010830067	4,275.439 GB/s	
Unified Total	14253627067	8,266.688 GB/s	Idle Low Medium High Max
Device Memory			
Reads	867244282	307.876 GB/s	
Writes	37459609	13.298 GB/s	
Total	904703891	321.174 GB/s	Idle Low Medium High Max
System Memory			
[PCIe configuration: Gen3 x16, 8 Gbit/s]			
Reads	0	0 B/s	Idle Low Medium High Max
Writes	5	1.775 kB/s	Idle Low Medium High Max

3.2. 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.

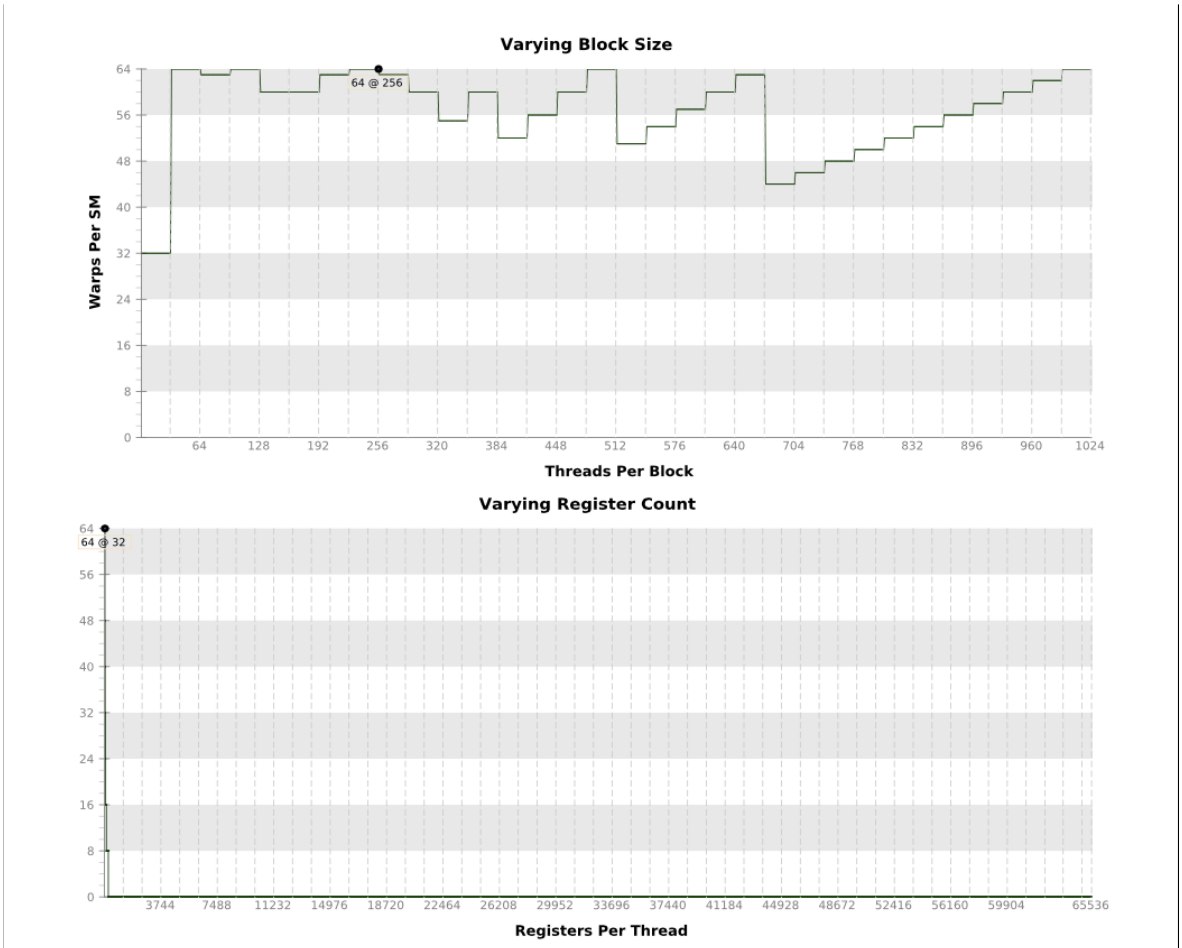
4. Instruction and Memory Latency

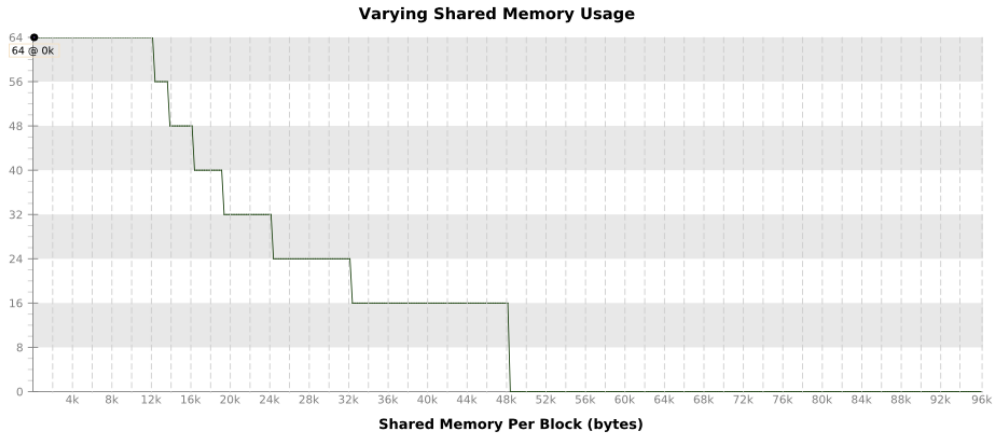
Instruction and memory latency limit the performance of a kernel when the GPU does not have enough work to keep busy. The performance of latency-limited kernels can often be improved by increasing occupancy. Occupancy is a measure of how many warps the kernel has active on the GPU, relative to the maximum number of warps supported by the GPU. Theoretical occupancy provides an upper bound while achieved occupancy indicates the kernel's actual occupancy.

4.1. 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.

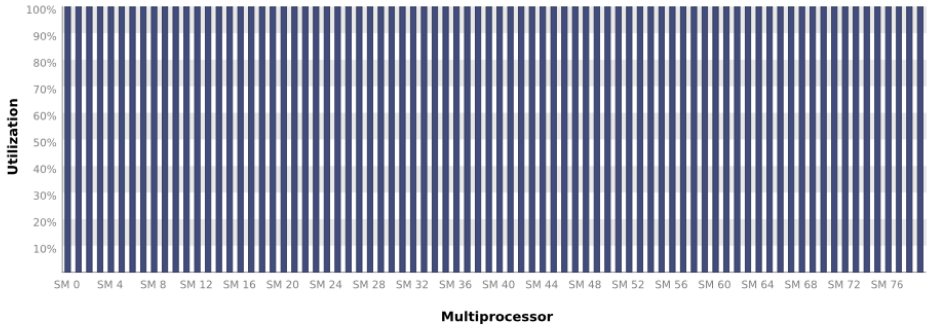
Variable	Achieved	Theoretical	Device Limit	Grid Size: [10000,24,4] (960000 blocks) Block Size: [1
Occupancy Per SM				
Active Blocks		8	32	0 4 8 12 16 20 24 28 32
Active Warps	57.48	64	64	0 9 18 27 36 45 54 64
Active Threads		2048	2048	0 512 1024 1536 2048
Occupancy	89.8%	100%	100%	0% 25% 50% 75% 100%
Warps				
Threads/Block		256	1024	0 256 512 768 1024
Warps/Block		8	32	0 4 8 12 16 20 24 28 32
Block Limit		8	32	0 4 8 12 16 20 24 28 32
Registers				
Registers/Thread		32	65536	0 16384 32768 49152 65536
Registers/Block		8192	65536	0 16k 32k 48k 64k
Block Limit		8	32	0 4 8 12 16 20 24 28 32
Shared Memory				
Shared Memory/Block		0	98304	0 32k 64k 96k
Block Limit		0	32	0 4 8 12 16 20 24 28 32


Varying Block Size
64 @ 256
Warps Per SM
0
8
16
24
32
40
48
56
64
64
128
192
256
320
384
448
512
576
640
704
768
832
896
960
1024
Threads Per Block
Varying Register Count
64 @ 32
0
8
16
24
32
40
48
56
64
3744
7488
11232
14976
18720
22464
26208
29952
33696
37440
41184
44928
48672
52416
56160
59904
65536
Registers Per Thread



4.3. Multiprocessor Utilization

The kernel's blocks are distributed across the GPU's multiprocessors for execution. Depending on the number of blocks and the execution duration of each block some multiprocessors may be more highly utilized than others during execution of the kernel. The following chart shows the utilization of each multiprocessor during execution of the kernel.



(b) Shared Memory convolution

Solution

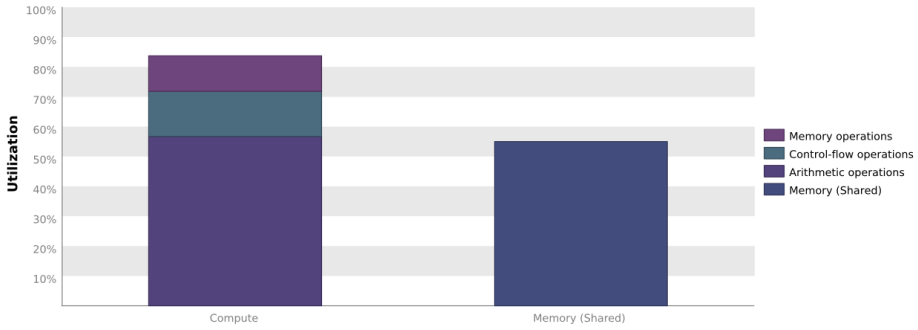
In this optimization we use shared memory to do the convolution. It took longer time than doing convolution without this optimization. One reason could because the kernel needed extra time to load data into shared memory. Again in NVVP output we can see that the most time is used for copy all that data. However, it is significantly less than the copying for the constant memory optimization for some reason. For some reason though the algorithm took longer than the original, which maybe is because of extra floating point computation that had to be used in order to use shared memory and tiling.
The Op times for size 10000 dataset are: 0.037272 and 0.127970.

1. Compute, Bandwidth, or Latency Bound

The first step in analyzing an individual kernel is to determine if the performance of the kernel is bounded by computation, memory bandwidth, or instruction/memory latency. The results below indicate that the performance of kernel "mxnet::op::forward_kernel" is most likely limited by compute. You should first examine the information in the "Compute Resources" section to determine how it is limiting performance.

1.1. Kernel Performance Is Bound By Compute

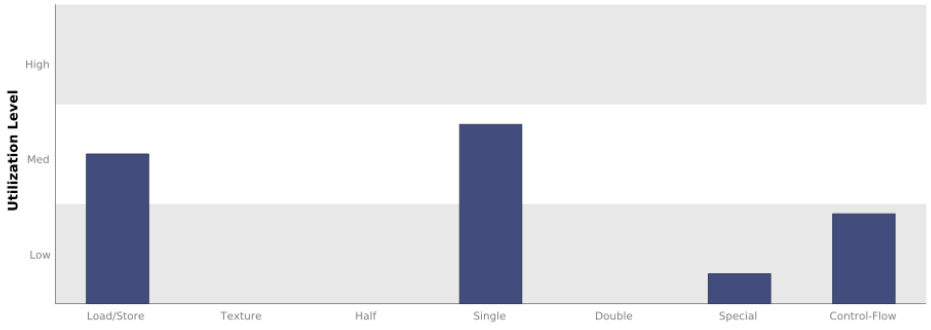
For device "TITAN V" the kernel's memory utilization is significantly lower than its compute utilization. These utilization levels indicate that the performance of the kernel is most likely being limited by computation on the SMs.



2.2. Function Unit Utilization

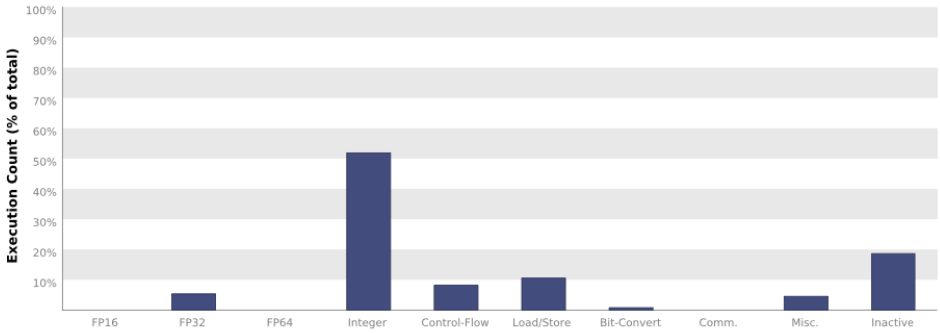
Different types of instructions are executed on different function units within each SM. Performance can be limited if a function unit is over-used by the instructions executed by the kernel. The following results show that the kernel's performance is not limited by overuse of any function unit.

Load/Store - Load and store instructions for shared and constant memory.
Texture - Load and store instructions for local, global, and texture memory.
Half - Half-precision floating-point arithmetic instructions.
Single - Single-precision integer and floating-point arithmetic instructions.
Double - Double-precision floating-point arithmetic instructions.
Special - Special arithmetic instructions such as sin, cos, popc, etc.
Control-Flow - Direct and indirect branches, jumps, and calls.



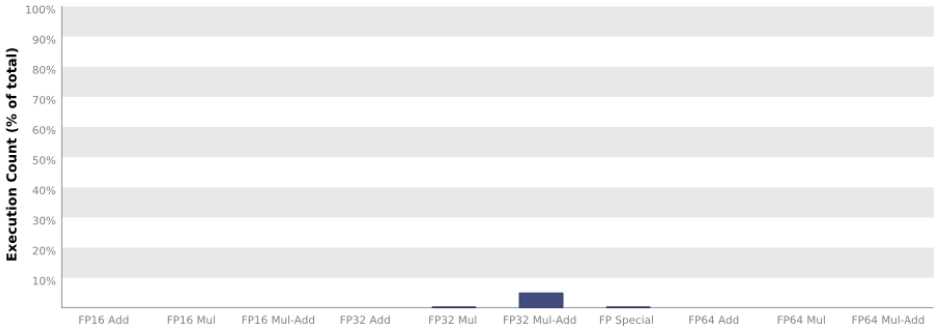
2.3. Instruction Execution Counts

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.



2.4. Floating-Point Operation Counts

The following chart shows the mix of floating-point operations executed by the kernel. The operations are grouped into classes and for each class the chart shows the percentage of thread execution cycles that were devoted to executing operations in that class. The results do not sum to 100% because non-floating-point operations executed by the kernel are not shown in this chart.



3. Memory Bandwidth

Memory bandwidth limits the performance of a kernel when one or more memories in the GPU cannot provide data at the rate requested by the kernel. The results below indicate that the kernel is limited by the bandwidth available to the shared memory.

3.1. Shared Memory Alignment and Access Pattern

Memory bandwidth is used most efficiently when each shared memory load and store has proper alignment and access pattern.

Optimization: Select each entry below to open the source code to a shared load or store within the kernel with an inefficient alignment or access pattern. For each load or store improve the alignment and access pattern of the memory access.

/mxnet/src/operator/custom/./new-forward.cuh

Line 41	Shared Store Transactions/Access = 1, Ideal Transactions/Access = 1 [9000000 transactions for 9000000 total executions]
Line 49	Shared Store Transactions/Access = 1.4, Ideal Transactions/Access = 1 [15600000 transactions for 10800000 total executions]
Line 49	Shared Store Transactions/Access = 1.4, Ideal Transactions/Access = 1 [39000000 transactions for 27000000 total executions]
Line 49	Shared Store Transactions/Access = 1.4, Ideal Transactions/Access = 1 [12480000 transactions for 8640000 total executions]
Line 57	Shared Load Transactions/Access = 1, Ideal Transactions/Access = 1 [120000000 transactions for 120000000 total executions]
Line 57	Shared Load Transactions/Access = 1, Ideal Transactions/Access = 1 [120000000 transactions for 120000000 total executions]
Line 57	Shared Load Transactions/Access = 1, Ideal Transactions/Access = 1 [120000000 transactions for 120000000 total executions]
Line 57	Shared Load Transactions/Access = 1, Ideal Transactions/Access = 1 [120000000 transactions for 120000000 total executions]
Line 57	Shared Load Transactions/Access = 1, Ideal Transactions/Access = 1 [120000000 transactions for 120000000 total executions]
Line 57	Shared Load Transactions/Access = 2, Ideal Transactions/Access = 1 [240000000 transactions for 120000000 total executions]
Line 57	Shared Load Transactions/Access = 2, Ideal Transactions/Access = 1 [240000000 transactions for 120000000 total executions]
Line 57	Shared Load Transactions/Access = 2, Ideal Transactions/Access = 1 [240000000 transactions for 120000000 total executions]
Line 57	Shared Load Transactions/Access = 2, Ideal Transactions/Access = 1 [240000000 transactions for 120000000 total executions]
Line 57	Shared Load Transactions/Access = 2, Ideal Transactions/Access = 1 [240000000 transactions for 120000000 total executions]

3.2. 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.

Transactions	Bandwidth	Utilization	
Shared Memory			
Shared Loads	1858463575	7,022.704 GB/s	
Shared Stores	77148421	291.526 GB/s	
Shared Total	1935611996	7,314.23 GB/s	Idle Low Medium High Max
L2 Cache			
Reads	149712863	141.433 GB/s	
Writes	261360498	246.905 GB/s	
Total	411073361	388.338 GB/s	Idle Low Medium High Max
Unified Cache			
Local Loads	0	0 B/s	
Local Stores	0	0 B/s	
Global Loads	465239798	439.508 GB/s	
Global Stores	261360000	246.905 GB/s	
Texture Reads	1418999492	5,362.071 GB/s	
Unified Total	2145599290	6,048.484 GB/s	Idle Low Medium High Max
Device Memory			
Reads	201226974	190.098 GB/s	
Writes	95095551	89.836 GB/s	
Total	296322525	279.934 GB/s	Idle Low Medium High Max
System Memory			
[PCIe configuration: Gen3 x16, 8 Gbit/s]			
Reads	0	0 B/s	Idle Low Medium High Max
Writes	5	4.723 kB/s	Idle Low Medium High Max

3.3. 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.

4. Instruction and Memory Latency

Instruction and memory latency limit the performance of a kernel when the GPU does not have enough work to keep busy. The performance of latency-limited kernels can often be improved by increasing occupancy. Occupancy is a measure of how many warps the kernel has active on the GPU, relative to the maximum number of warps supported by the GPU. Theoretical occupancy provides an upper bound while achieved occupancy indicates the kernel's actual occupancy. The results below indicate that occupancy can be improved by reducing the number of registers used by the kernel.

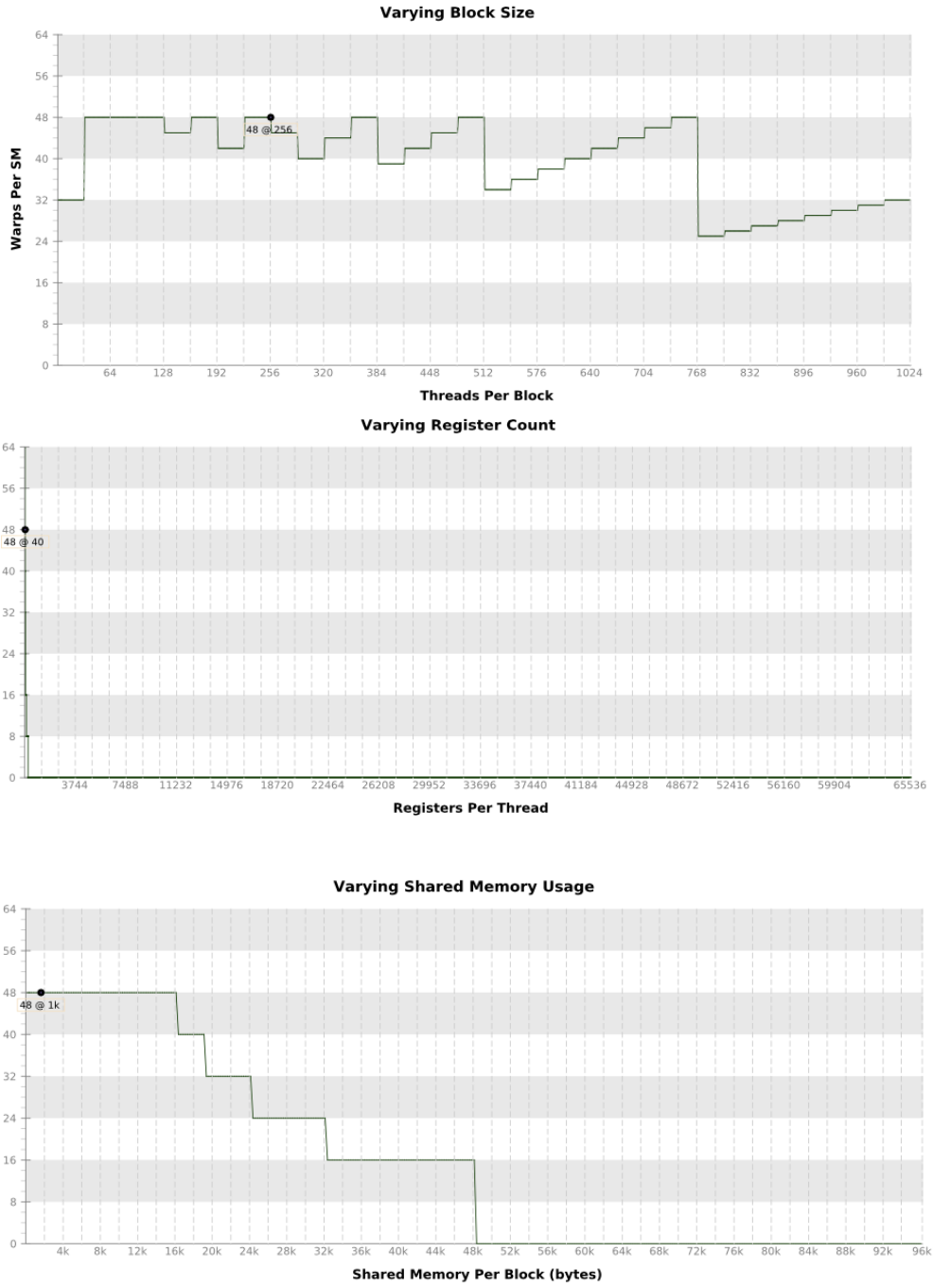
4.1. GPU Utilization May Be Limited By Register Usage

Theoretical occupancy is less than 100% but is large enough that increasing occupancy may not improve performance. You can attempt the following optimization to increase the number of warps on each SM but it may not lead to increased performance.

The kernel uses 40 registers for each thread (10240 registers for each block). This register usage is likely preventing the kernel from fully utilizing the GPU. Device "TITAN V" provides up to 65536 registers for each block. Because the kernel uses 10240 registers for each block each SM is limited to simultaneously executing 6 blocks (48 warps). Chart "Varying Register Count" below shows how changing register usage will change the number of blocks that can execute on each SM.

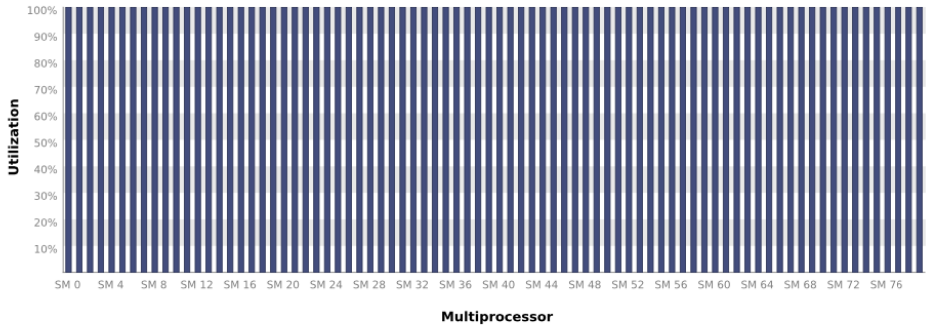
Optimization: Use the -maxrregcount flag or the __launch_bounds__ qualifier to decrease the number of registers used by each thread. This will increase the number of blocks that can execute on each SM. On devices with Compute Capability 5.2 turning global cache off can increase the occupancy limited by register usage.

Variable	Achieved	Theoretical	Device Limit	Grid Size: [10000,12,25] (3000000 blocks) Block Size:
Occupancy Per SM				
Active Blocks		6	32	0 4 8 12 16 20 24 28 32
Active Warps	46.09	48	64	0 9 18 27 36 45 54 64
Active Threads		1536	2048	0 512 1024 1536 2048
Occupancy	72%	75%	100%	0% 25% 50% 75% 100%
Warps				
Threads/Block		256	1024	0 256 512 768 1024
Warps/Block		8	32	0 4 8 12 16 20 24 28 32
Block Limit		8	32	0 4 8 12 16 20 24 28 32
Registers				
Registers/Thread		40	65536	0 16384 32768 49152 65536
Registers/Block		10240	65536	0 16k 32k 48k 64k
Block Limit		6	32	0 4 8 12 16 20 24 28 32
Shared Memory				
Shared Memory/Block		1700	98304	0 32k 64k 96k
Block Limit		54	32	0 4 8 12 16 20 24 28 32


Varying Block Size
Warps Per SM
48 @ 256
Threads Per Block
Varying Register Count
48 @ 40
Registers Per Thread
Varying Shared Memory Usage
48 @ 1k
Shared Memory Per Block (bytes)

4.3. Multiprocessor Utilization

The kernel's blocks are distributed across the GPU's multiprocessors for execution. Depending on the number of blocks and the execution duration of each block some multiprocessors may be more highly utilized than others during execution of the kernel. The following chart shows the utilization of each multiprocessor during execution of the kernel.



(c) Weight matrix (kernel values) in constant memory

Solution

In this optimization we put all kernels in constant memory instead of fetching them from global memory each time. This speed up execution by a small amount because GPU no longer needs to wait for global memory each time and could get from the faster constant memory. In the nvvp output we can see that a very large portion of the time is spent copying memory from the host to the device. This is because there is a lot of data to copy like the input, output, and const kernel memory. Everything else is fairly insignificant in terms of time consuming. Calculating the convolution took less time than the copying, this coincident with what we learn during the class.

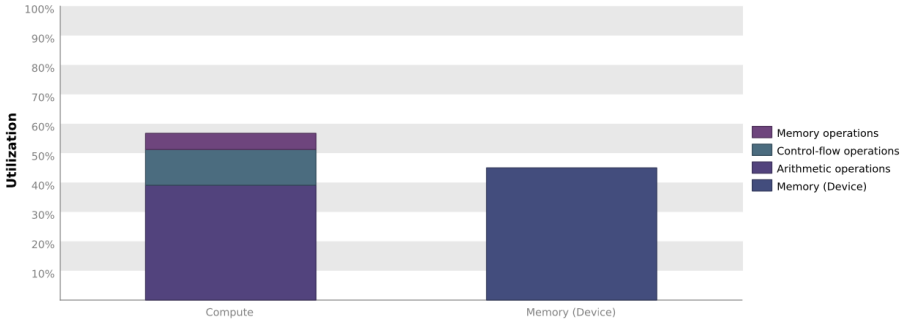
The Op times for size 10000 dataset are: 0.030465 and 0.092395.

1. Compute, Bandwidth, or Latency Bound

The first step in analyzing an individual kernel is to determine if the performance of the kernel is bounded by computation, memory bandwidth, or instruction/memory latency. The results below indicate that the performance of kernel "mxnet::op::forward_kernel" is most likely limited by instruction and memory latency. You should first examine the information in the "Instruction And Memory Latency" section to determine how it is limiting performance.

1.1. Kernel Performance Is Bound By Instruction And Memory Latency

This kernel exhibits low compute throughput and memory bandwidth utilization relative to the peak performance of "TITAN V". These utilization levels indicate that the performance of the kernel is most likely limited by the latency of arithmetic or memory operations. Achieved compute throughput and/or memory bandwidth below 60% of peak typically indicates latency issues.



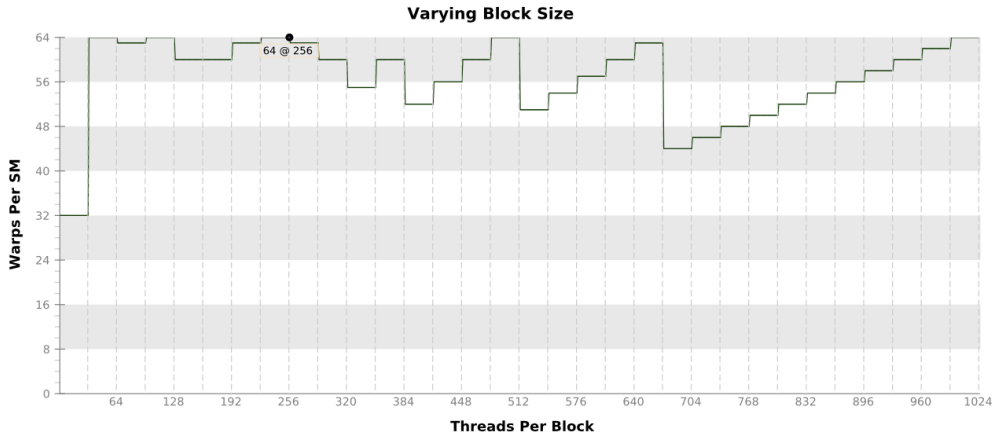
2. Instruction and Memory Latency

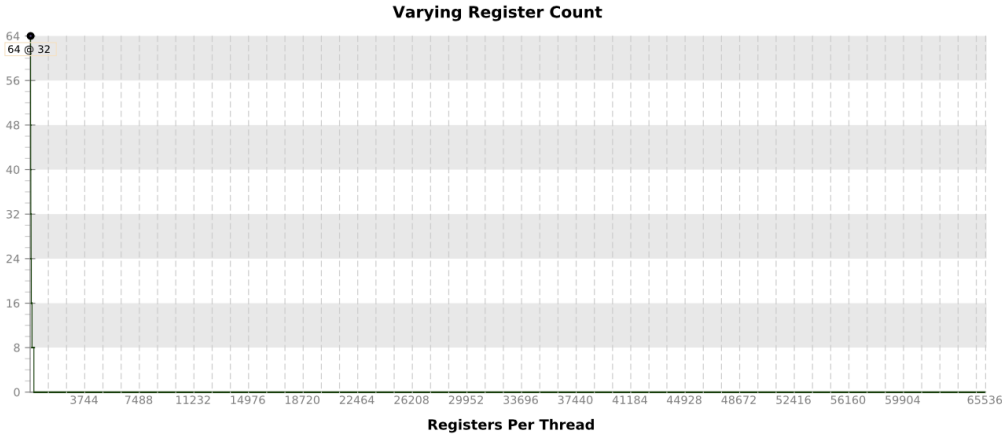
Instruction and memory latency limit the performance of a kernel when the GPU does not have enough work to keep busy. The performance of latency-limited kernels can often be improved by increasing occupancy. Occupancy is a measure of how many warps the kernel has active on the GPU, relative to the maximum number of warps supported by the GPU. Theoretical occupancy provides an upper bound while achieved occupancy indicates the kernel's actual occupancy.

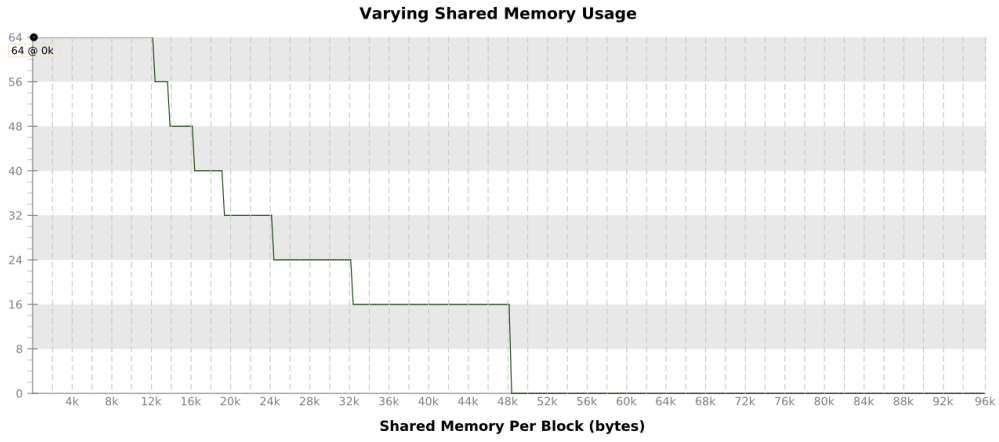
2.1. Occupancy Is Not Limiting Kernel Performance

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

Variable	Achieved	Theoretical	Device Limit	Grid Size: [10000,12,25] (3000000 blocks) Block Size:
Occupancy Per SM				
Active Blocks		8	32	0 4 8 12 16 20 24 28 32
Active Warps	51.71	64	64	0 9 18 27 36 45 54 63 64
Active Threads		2048	2048	0 512 1024 1536 2048
Occupancy	80.8%	100%	100%	0% 25% 50% 75% 100%
Warps				
Threads/Block		256	1024	0 256 512 768 1024
Warps/Block		8	32	0 4 8 12 16 20 24 28 32
Block Limit		8	32	0 4 8 12 16 20 24 28 32
Registers				
Registers/Thread		32	65536	0 16384 32768 49152 65536
Registers/Block		8192	65536	0 16k 32k 48k 64k
Block Limit		8	32	0 4 8 12 16 20 24 28 32
Shared Memory				
Shared Memory/Block		0	98304	0 32k 64k 96k
Block Limit		0	32	0 4 8 12 16 20 24 28 32

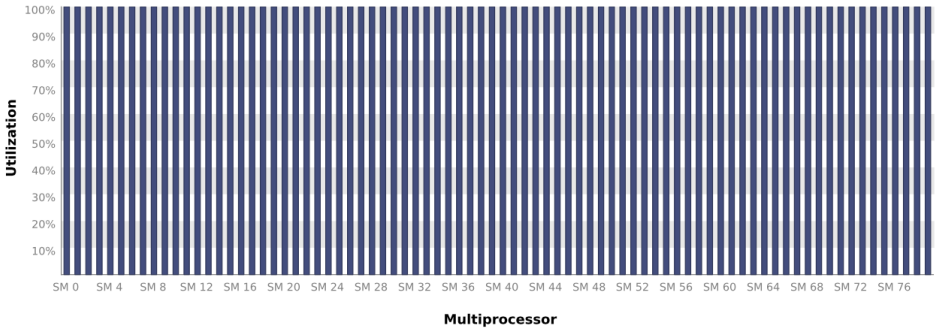

Varying Block Size
Warps Per SM
64 @ 256
Threads Per Block


Varying Register Count
64 @ 32
Registers Per Thread


Varying Shared Memory Usage
64 @ 0k
Shared Memory Per Block (bytes)

2.3. Multiprocessor Utilization

The kernel's blocks are distributed across the GPU's multiprocessors for execution. Depending on the number of blocks and the execution duration of each block some multiprocessors may be more highly utilized than others during execution of the kernel. The following chart shows the utilization of each multiprocessor during execution of the kernel.



3. Compute Resources

GPU compute resources limit the performance of a kernel when those resources are insufficient or poorly utilized. Compute resources are used most efficiently when all threads in a warp have the same branching and predication behavior. The results below indicate that a significant fraction of the available compute performance is being wasted because branch and predication behavior is differing for threads within a warp.

3.1. 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 84.9% if predicated instructions are not taken into account. The kernel's not predicated off warp execution efficiency of 76.4% is less than 100% due to divergent branches and predicated instructions.

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

3.2. 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: Each entry below points to a divergent branch within the kernel. For each branch reduce the amount of intra-warp divergence.

/mxnet/src/operator/custom/./new-forward.cuh

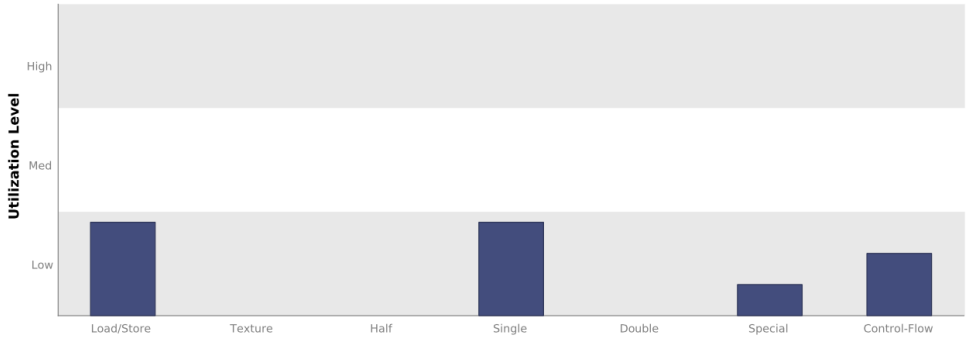
Line 32	Divergence = 16.5% [3960000 divergent executions out of 24000000 total executions]
Line 33	Divergence = 0% [0 divergent executions out of 19800000 total executions]
Line 33	Divergence = 0% [0 divergent executions out of 19800000 total executions]
Line 34	Divergence = 0% [0 divergent executions out of 19800000 total executions]
Line 34	Divergence = 0% [0 divergent executions out of 99000000 total executions]
Line 35	Divergence = 0% [0 divergent executions out of 99000000 total executions]
Line 35	Divergence = 0% [0 divergent executions out of 99000000 total executions]
Line 35	Divergence = 0% [0 divergent executions out of 99000000 total executions]
Line 35	Divergence = 0% [0 divergent executions out of 99000000 total executions]
Line 35	Divergence = 0% [0 divergent executions out of 99000000 total executions]
Line 35	Divergence = 0% [0 divergent executions out of 99000000 total executions]
Line 36	Divergence = 0% [0 divergent executions out of 99000000 total executions]
Line 36	Divergence = 0% [0 divergent executions out of 99000000 total executions]
Line 44	Divergence = 0% [0 divergent executions out of 19800000 total executions]

3.3. Function Unit Utilization

Different types of instructions are executed on different function units within each SM. Performance can be limited if a function unit is over-used by the instructions executed by the kernel. The following results show that the kernel's performance is not limited by overuse of any function unit.

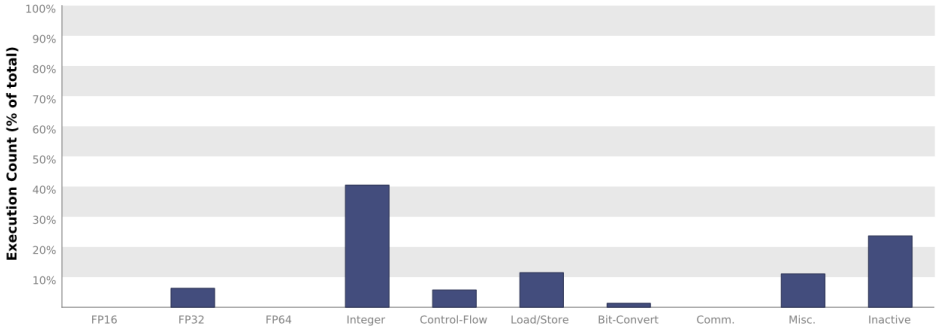
Load/Store - Load and store instructions for shared and constant memory.
Texture - Load and store instructions for local, global, and texture memory.
Half - Half-precision floating-point arithmetic instructions.
Single - Single-precision integer and floating-point arithmetic instructions.
Double - Double-precision floating-point arithmetic instructions.

Special - Special arithmetic instructions such as sin, cos, popc, etc.
Control-Flow - Direct and indirect branches, jumps, and calls.



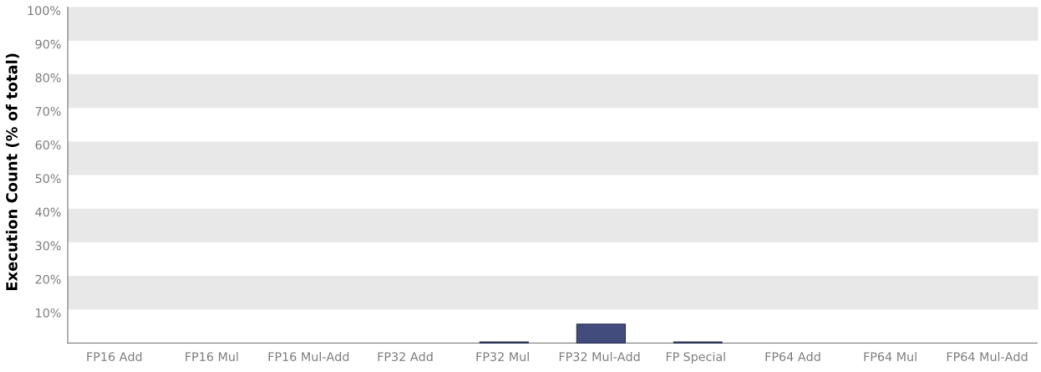
3.4. Instruction Execution Counts

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.



3.5. Floating-Point Operation Counts

The following chart shows the mix of floating-point operations executed by the kernel. The operations are grouped into classes and for each class the chart shows the percentage of thread execution cycles that were devoted to executing operations in that class. The results do not sum to 100% because non-floating-point operations executed by the kernel are not shown in this chart.



4. Memory Bandwidth

Memory bandwidth limits the performance of a kernel when one or more memories in the GPU cannot provide data at the rate requested by the kernel.

4.1. 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.

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	151466239	159.673 GB/s	
Writes	95040038	100.19 GB/s	
Total	246506277	259.863 GB/s	Idle Low Medium High Max
Unified Cache			
Local Loads	0	0 B/s	
Local Stores	0	0 B/s	
Global Loads	2474928955	2,609.027 GB/s	
Global Stores	95040000	100.19 GB/s	
Texture Reads	791165061	3,336.129 GB/s	
Unified Total	3361134016	6,045.346 GB/s	Idle Low Medium High Max
Device Memory			
Reads	201776702	212.709 GB/s	
Writes	95094655	100.247 GB/s	
Total	296871357	312.957 GB/s	Idle Low Medium High Max
System Memory			
[PCIe configuration: Gen3 x16, 8 Gbit/s]			
Reads	0	0 B/s	Idle Low Medium High Max
Writes	5	5.27 kB/s	Idle Low Medium High Max

4.2. 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.