Analysis Report

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

Duration	1.20233 ms (1,202,326 ns)
Grid Size	[3,3,1000]
Block Size	[16,16,1]
Registers/Thread	40
Shared Memory/Block	4 KiB
Shared Memory Executed	0 B
Shared Memory Bank Size	4 B

[0] TITAN V

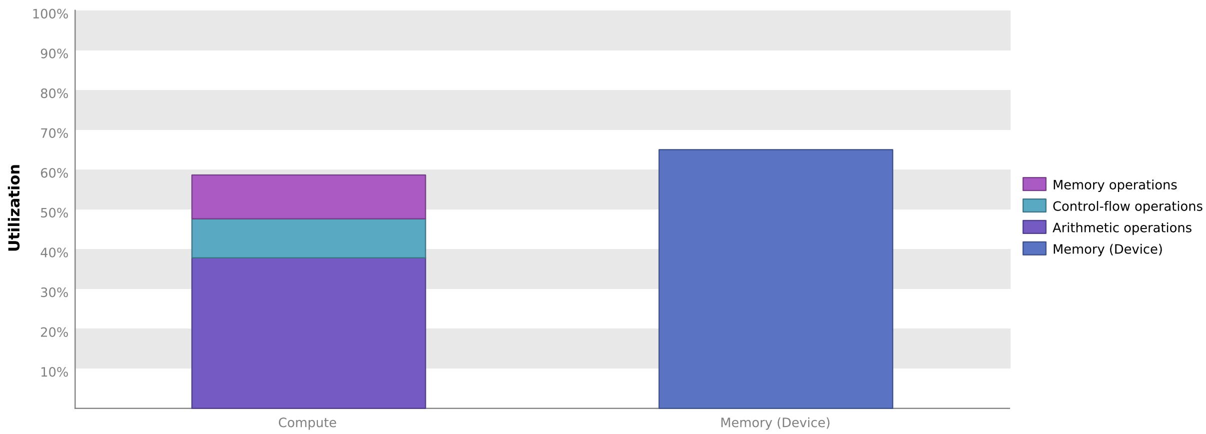
GPU UUID	GPU-01e2078d-caf5-3bc2-aeb7-80d9b3d6e701
Compute Capability	7.0
Max. Threads per Block	1024
Max. Threads per Multiprocessor	2048
Max. Shared Memory per Block	48 KiB
Max. Shared Memory per Multiprocessor	96 KiB
Max. Registers per Block	65536
Max. Registers per Multiprocessor	65536
Max. Grid Dimensions	[2147483647, 65535, 65535]
Max. Block Dimensions	[1024, 1024, 64]
Max. Warps per Multiprocessor	64
Max. Blocks per Multiprocessor	32
Half Precision FLOP/s	29.798 TeraFLOP/s
Single Precision FLOP/s	14.899 TeraFLOP/s
Double Precision FLOP/s	7.45 TeraFLOP/s
Number of Multiprocessors	80
Multiprocessor Clock Rate	1.455 GHz
Concurrent Kernel	true
Max IPC	4
Threads per Warp	32
Global Memory Bandwidth	652.8 GB/s
Global Memory Size	11.755 GiB
Constant Memory Size	64 KiB
L2 Cache Size	4.5 MiB
Memcpy Engines	7
PCIe Generation	3
PCIe Link Rate	8 Gbit/s
PCIe Link Width	8

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 memory bandwidth. You should first examine the information in the "Memory Bandwidth" section to determine how it is limiting performance.

1.1. 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 Device memory.



2. 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.

2.1. 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: Each entry below points 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.

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

Line 47	Global Load L2 Transactions/Access = 8.1, Ideal Transactions/Access = 3.6 [537500 L2 transactions for 66000 total executions]
Line 47	Global Load L2 Transactions/Access = 8.8, Ideal Transactions/Access = 3.6 [580500 L2 transactions for 66000 total executions]
Line 47	Global Load L2 Transactions/Access = 8.1, Ideal Transactions/Access = 3.6 [537500 L2 transactions for 66000 total executions]
Line 47	Global Load L2 Transactions/Access = 8.8, Ideal Transactions/Access = 3.6 [580500 L2 transactions for 66000 total executions]

2.2. 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 47	Shared Store Transactions/Access = 2, Ideal Transactions/Access = 1 [129000 transactions for 66000 total executions]
Line 47	Shared Store Transactions/Access = 2, Ideal Transactions/Access = 1 [129000 transactions for 66000 total executions]
Line 47	Shared Store Transactions/Access = 2, Ideal Transactions/Access = 1 [129000 transactions for 66000 total executions]
Line 47	Shared Store Transactions/Access = 2, Ideal Transactions/Access = 1 [129000 transactions for 66000 total executions]
Line 67	Shared Load Transactions/Access = 1.9, Ideal Transactions/Access = 1 [5940000 transactions for 3060000 total executions]
Line 67	Shared Load Transactions/Access = 1.9, Ideal Transactions/Access = 1 [5940000 transactions for 3060000 total executions]
Line 67	Shared Load Transactions/Access = 1.9, Ideal Transactions/Access = 1 [5940000 transactions for 3060000 total executions]
Line 67	Shared Load Transactions/Access = 1.9, Ideal Transactions/Access = 1 [5940000 transactions for 3060000 total executions]
Line 67	Shared Load Transactions/Access = 1.9, Ideal Transactions/Access = 1 [5940000 transactions for 3060000 total executions]

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

Line 67	Shared Load Transactions/Access = 1.9, Ideal Transactions/Access = 1 [5940000 transactions for 3060000 total executions]
Line 67	Shared Load Transactions/Access = 1.9, Ideal Transactions/Access = 1 [5940000 transactions for 3060000 total executions]
Line 67	Shared Load Transactions/Access = 1.9, Ideal Transactions/Access = 1 [5940000 transactions for 3060000 total executions]
Line 67	Shared Load Transactions/Access = 1.9, Ideal Transactions/Access = 1 [5940000 transactions for 3060000 total executions]
Line 67	Shared Load Transactions/Access = 1.9, Ideal Transactions/Access = 1 [5940000 transactions for 3060000 total executions]
Line 67	Shared Load Transactions/Access = 1.9, Ideal Transactions/Access = 1 [5940000 transactions for 3060000 total executions]
Line 67	Shared Load Transactions/Access = 1.9, Ideal Transactions/Access = 1 [5940000 transactions for 3060000 total executions]
Line 67	Shared Load Transactions/Access = 1.9, Ideal Transactions/Access = 1 [5940000 transactions for 3060000 total executions]
Line 67	Shared Load Transactions/Access = 1.9, Ideal Transactions/Access = 1 [5940000 transactions for 3060000 total executions]

2.3. GPU Utilization Is Limited By Memory Bandwidth

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. The results show that the kernel's performance is potentially limited by the bandwidth available from one or more of the memories on the device.

Optimization: Try the following optimizations for the memories with high bandwidth utilization.
Shared Memory - If possible use 64-bit accesses to shared memory and 8-byte bank mode to achieved 2x throughput.
L2 Cache - Align and block kernel data to maximize L2 cache efficiency.
Unified Cache - Reallocate texture data to shared or global memory. Resolve alignment and access pattern issues for global loads and stores.
Device Memory - Resolve alignment and access pattern issues for global loads and stores.
System Memory (via PCIe) - Make sure performance critical data is placed in device or shared memory.

Transactions	Bandwidth	Utilization	
Shared Memory			
Shared Loads	87247866	9,288.435 GB/s	
Shared Stores	517512	55.094 GB/s	
Shared Total	87765378	9,343.529 GB/s	Idle Low Medium High Max
L2 Cache			
Reads	17736644	472.062 GB/s	
Writes	16632016	442.662 GB/s	
Total	34368660	914.725 GB/s	Idle Low Medium High Max
Unified Cache			
Local Loads	0	0 B/s	
Local Stores	0	0 B/s	
Global Loads	2232769	59.425 GB/s	
Global Stores	0	0 B/s	
Texture Reads	43510945	4,632.189 GB/s	
Unified Total	45743714	4,691.614 GB/s	Idle Low Medium High Max
Device Memory			
Reads	7793310	207.42 GB/s	
Writes	7460020	198.549 GB/s	
Total	15253330	405.969 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	133.075 kB/s	Idle Low Medium High Max

2.4. 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.

3. 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.

3.1. 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.

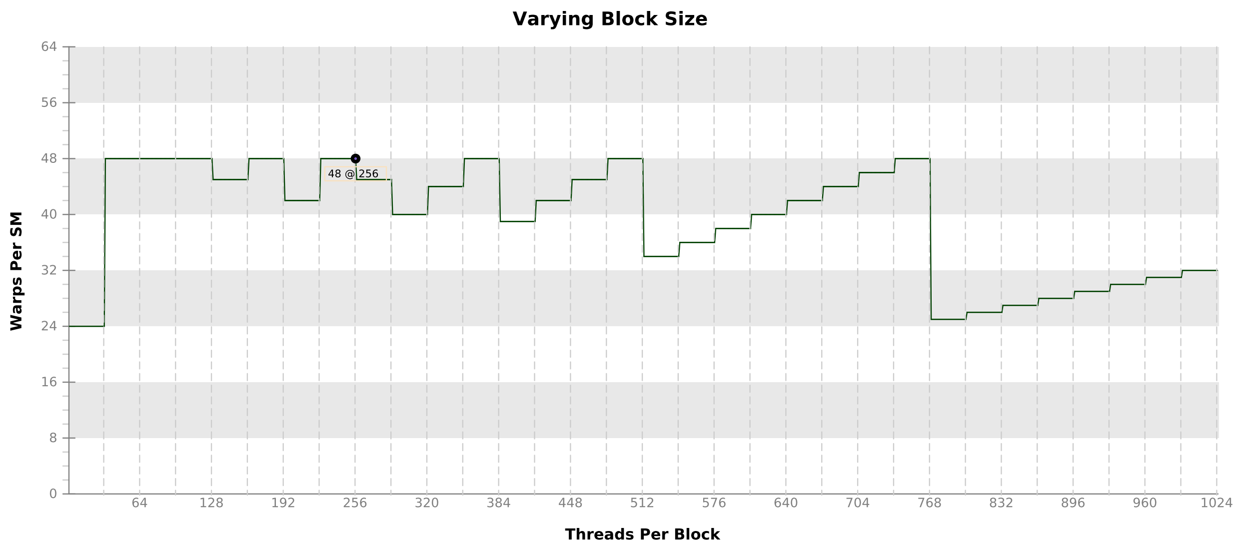
The kernel uses 40 registers for each thread (10240 registers for each block). This register usage is likely preventing the kernel from fully utilizing the GPU. Device "TITAN V" provides up to 65536 registers for each block. Because the kernel uses 10240 registers for each block each SM is limited to 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.

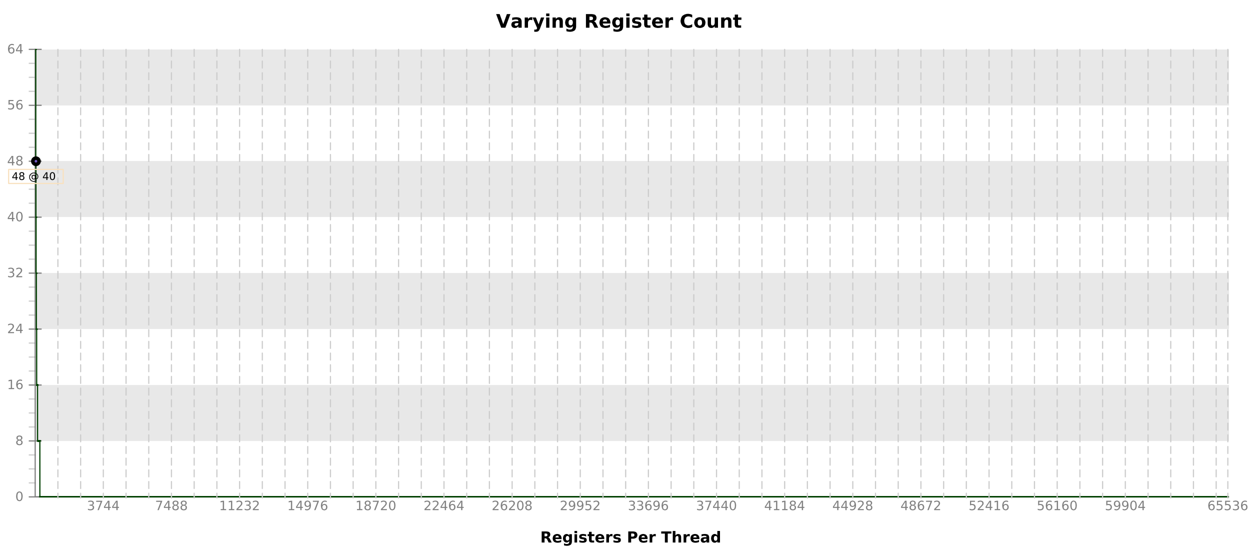
Optimization: Use the -maxrregcount flag or the __launch_bounds__ qualifier to decrease the number of registers used by each thread. This will increase the number of blocks that can execute on each SM. On devices with Compute Capability 5.2 turning global cache off can increase the occupancy limited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

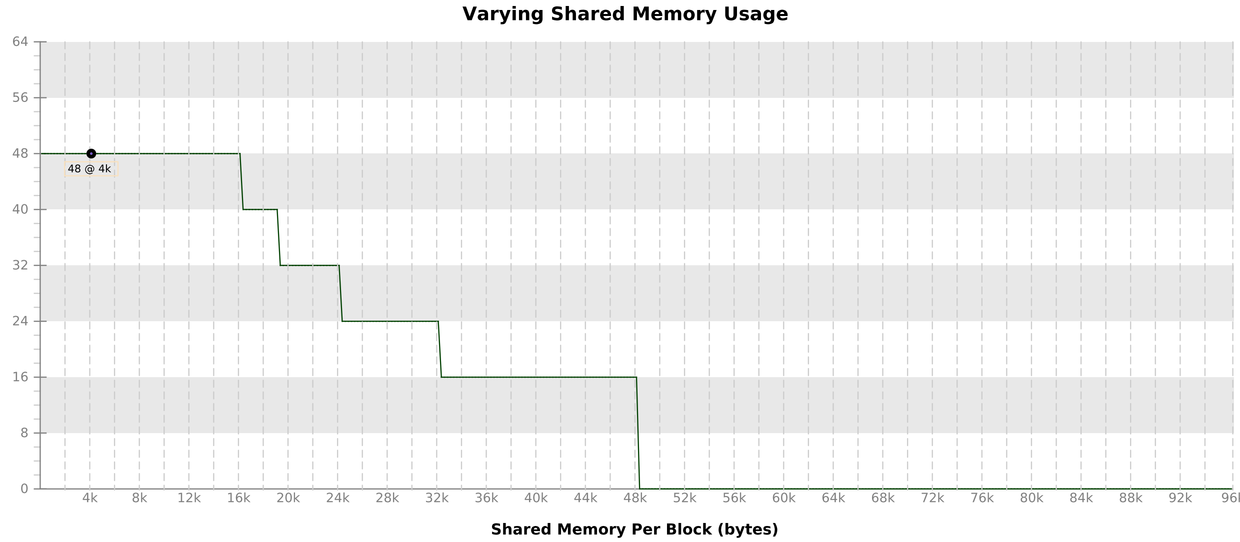
Variable	Achieved	Theoretical	Device Limit	Grid Size: [3,3,1000] (9000 blocks) Block Size: [16,16,1] (256 th
Occupancy Per SM				
Active Blocks		6	32	0 3 6 9 12 15 18 21 24 27 30 32
Active Warps	38.41	48	64	0 7 14 21 28 35 42 49 56 6364
Active Threads		1536	2048	0 256 512 768 1024 1280 1536 1792 2048
Occupancy	60%	75%	100%	0% 25% 50% 75% 100%
Warps				
Threads/Block		256	1024	0 128 256 384 512 640 768 896 1024
Warps/Block		8	32	0 3 6 9 12 15 18 21 24 27 30 32
Block Limit		8	32	0 3 6 9 12 15 18 21 24 27 30 32
Registers				
Registers/Thread		40	65536	0 8192 16384 24576 32768 40960 49152 57344 65536
Registers/Block		10240	65536	0 16k 32k 48k 64k
Block Limit		6	32	0 3 6 9 12 15 18 21 24 27 30 32
Shared Memory				
Shared Memory/Block		4096	98304	0 32k 64k 96k
Block Limit		24	32	0 3 6 9 12 15 18 21 24 27 30 32

3.2. Occupancy Charts

The following charts show how varying different components of the kernel will impact theoretical occupancy.

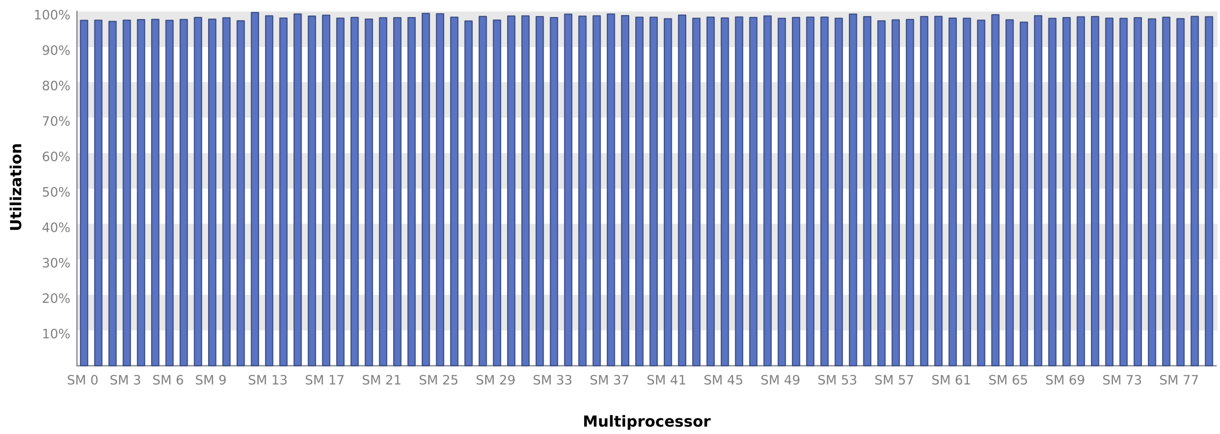

Varying Block Size
Warps Per SM
Threads Per Block
48 @ 256
0
8
16
24
32
40
48
56
64
64
128
192
256
320
384
448
512
576
640
704
768
832
896
960
1024


Varying Register Count
Registers Per Thread
48 @ 40
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



3.3. 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.



4. 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.

4.1. Kernel Profile - Instruction Execution

The Kernel Profile - Instruction Execution shows the execution count, inactive threads, and predicated threads for each source and assembly line of the kernel. Using this information you can pinpoint portions of your kernel that are making inefficient use of compute resource due to divergence and predication.

Examine portions of the kernel that have high execution counts and inactive or predicated threads to identify optimization opportunities.

Cuda Fuctions :

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

Maximum instruction execution count in assembly: 3060000
Average instruction execution count in assembly: 1287594
Instructions executed for the kernel: 266532000
Thread instructions executed for the kernel: 5764068000
Non-predicated thread instructions executed for the kernel: 5376218000
Warp non-predicated execution efficiency of the kernel: 63.0%
Warp execution efficiency of the kernel: 67.6%

Source files :

/mxnet/src/operator/custom/./new-forward.cuh
/usr/local/cuda/include/sm_20_atomic_functions.hpp

4.2. 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 67.6% if predicated instructions are not taken into account. The kernel's not predicated off warp execution efficiency of 63% is less than 100% due to divergent branches and predicated instructions.

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

4.3. 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 23	Divergence = 0% [0 divergent executions out of 72000 total executions]
Line 52	Divergence = 70.8% [51000 divergent executions out of 72000 total executions]
Line 53	Divergence = 0% [0 divergent executions out of 612000 total executions]
Line 61	Divergence = 0% [0 divergent executions out of 612000 total executions]
Line 61	Divergence = 0% [0 divergent executions out of 3060000 total executions]

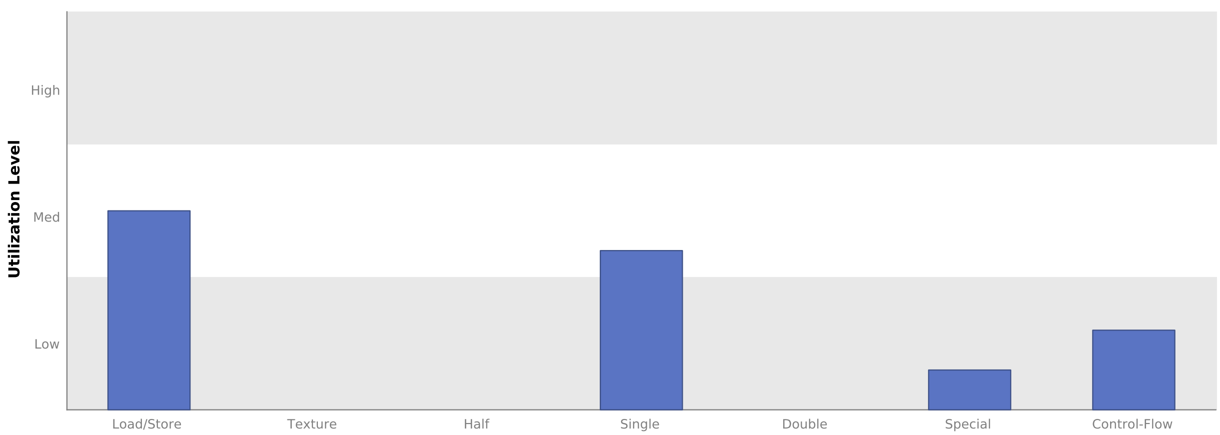
/mxnet/src/operator/custom/./new-forward.cuh

Line 62	Divergence = 0% [0 divergent executions out of 3060000 total executions]
Line 62	Divergence = 0% [0 divergent executions out of 3060000 total executions]
Line 62	Divergence = 0% [0 divergent executions out of 3060000 total executions]
Line 62	Divergence = 0% [0 divergent executions out of 3060000 total executions]
Line 62	Divergence = 0% [0 divergent executions out of 3060000 total executions]
Line 67	Divergence = 0% [0 divergent executions out of 3060000 total executions]
Line 67	Divergence = 0% [0 divergent executions out of 3060000 total executions]
Line 67	Divergence = 0% [0 divergent executions out of 3060000 total executions]
Line 85	Divergence = 0% [0 divergent executions out of 72000 total executions]
Line 85	Divergence = 0% [0 divergent executions out of 51000 total executions]
Line 85	Divergence = 0% [0 divergent executions out of 72000 total executions]

4.4. Function Unit Utilization

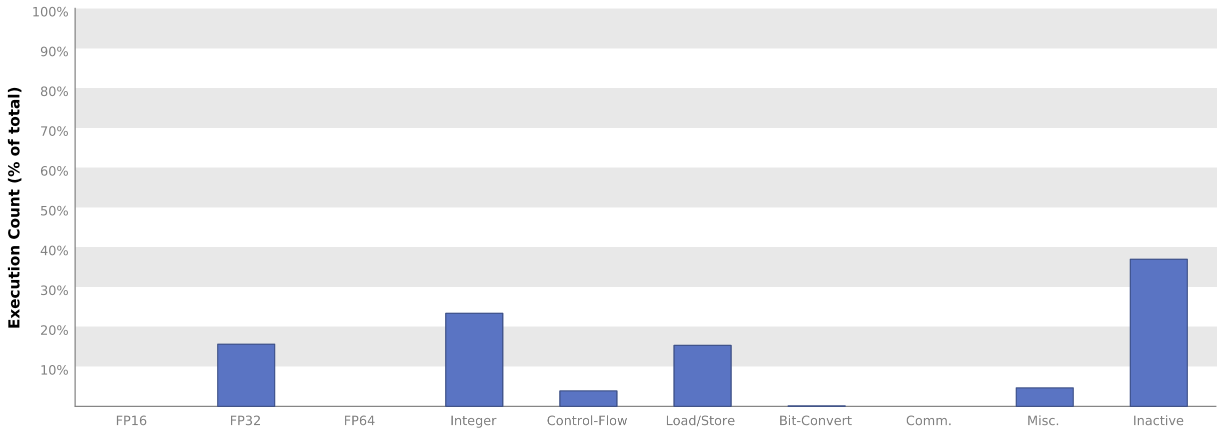
Different types of instructions are executed on different function units within each SM. Performance can be limited if a function unit is over-used by the instructions executed by the kernel. The following results show that the kernel's performance is not limited by overuse of any function unit.

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.



4.5. 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.



4.6. 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.