

Analysis Report

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

Duration	5.1965 ms (5,196,500 ns)
Grid Size	[27,1,1000]
Block Size	[32,32,1]
Registers/Thread	31
Shared Memory/Block	8 KiB
Shared Memory Executed	0 B
Shared Memory Bank Size	4 B

[0] TITAN V

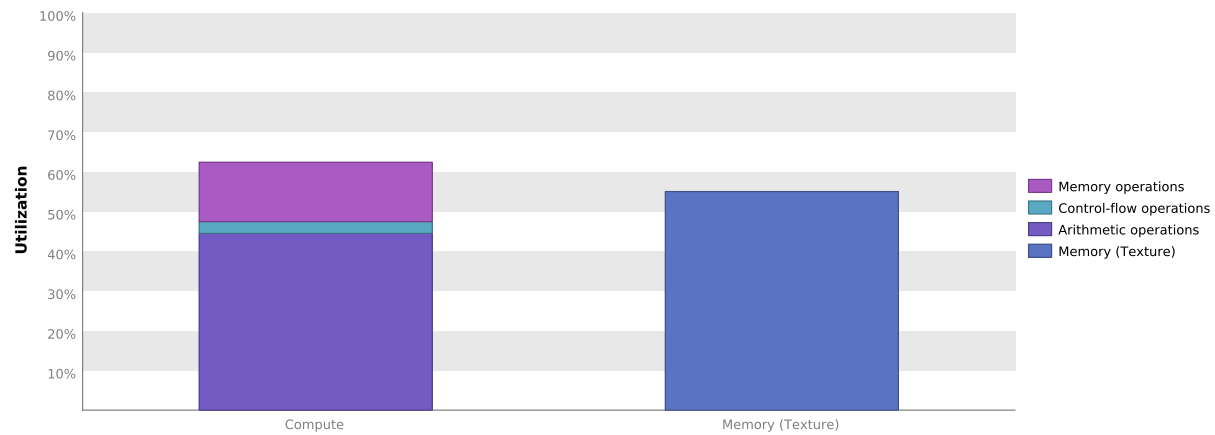
GPU UUID	GPU-9910c808-3335-47c7-980a-bf909aedc5aa
Compute Capability	7.0
Max. Threads per Block	1024
Max. Threads per Multiprocessor	2048
Max. Shared Memory per Block	48 KiB
Max. Shared Memory per Multiprocessor	96 KiB
Max. Registers per Block	65536
Max. Registers per Multiprocessor	65536
Max. Grid Dimensions	[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::matrixMultiplySh..." 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 results below indicate that the GPU does not have enough work because instruction execution is stalling excessively.

2.1. Kernel Profile - PC Sampling

The Kernel Profile - PC Sampling gives the number of samples for each source and assembly line with various stall reasons. The samples are collected at a period of 2048 [2^11] cycles. You can change the period under Settings->Analysis tab. The allowed values are from 5 to 31. Increasing the period would reduce the number of samples collected.

Using this information you can pinpoint portions of your kernel that are introducing latencies and the reason for the latency. Samples are taken in round robin order for all active warps at a fixed number of cycles regardless of whether the warp is issuing an instruction or not.

Instruction Issued - Warp was issued

Instruction Fetch - The next assembly instruction has not yet been fetched.

Execution Dependency - An input required by the instruction is not yet available. Execution dependency stalls can potentially be reduced by increasing instruction-level parallelism.

Memory Dependency - A load/store cannot be made because the required resources are not available or are fully utilized, or too many requests of a given type are outstanding. Data request stalls can potentially be reduced by optimizing memory alignment and access patterns.

Texture - The texture sub-system is fully utilized or has too many outstanding requests.

Synchronization - The warp is blocked at a __syncthreads() call.

Constant - A constant load is blocked due to a miss in the constants cache.

Pipe Busy - The compute resource(s) required by the instruction is not yet available.

Memory Throttle - Large number of pending memory operations prevent further forward progress. These can be reduced by combining several memory transactions into one.

Not Selected - Warp was ready to issue, but some other warp issued instead. You may be able to sacrifice occupancy without impacting latency hiding and doing so may help improve cache hit rates.

Other - The warp is blocked for an uncommon reason.

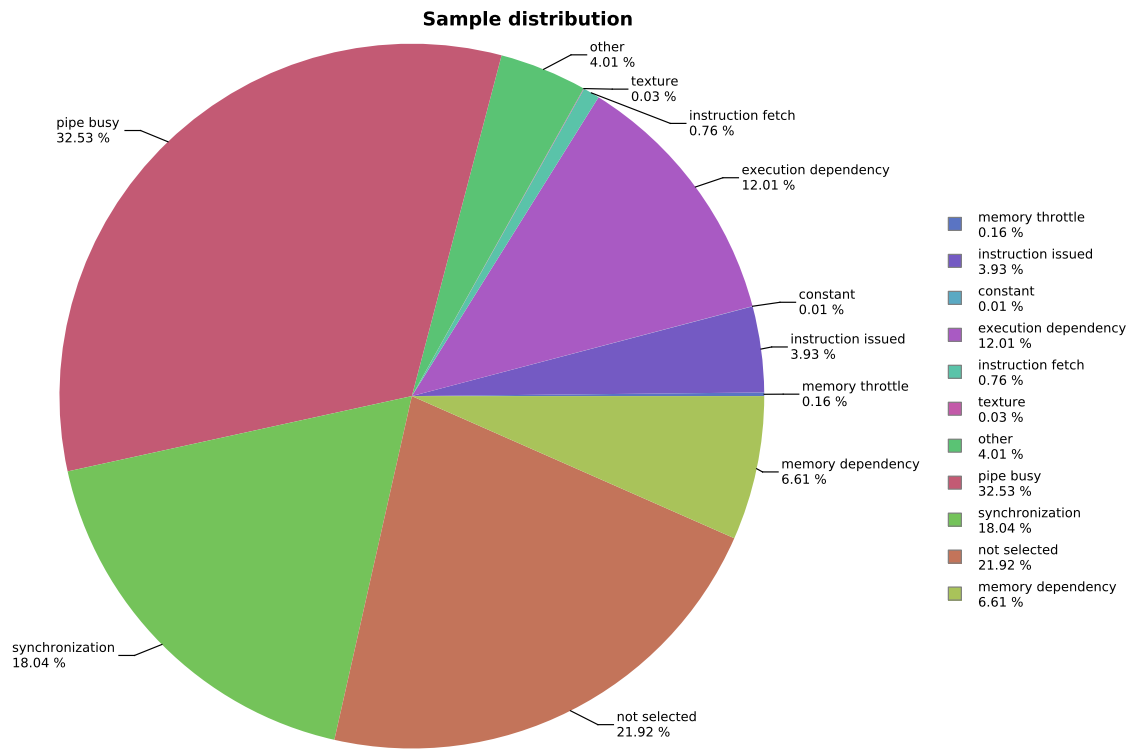
Sleeping -The warp is blocked, yielded or sleeping.

Examine portions of the kernel that have high number of samples to know where the maximum time was spent and observe the latency reasons for those samples to identify optimization opportunities.

Cuda Functions	Sample Count	% of Kernel Samples
mxnet::op::matrixMultiplyShared(float*, float*, float*, int, int, int, int, int, int, int, int, int, int)	270329	100.0

Source Files :

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



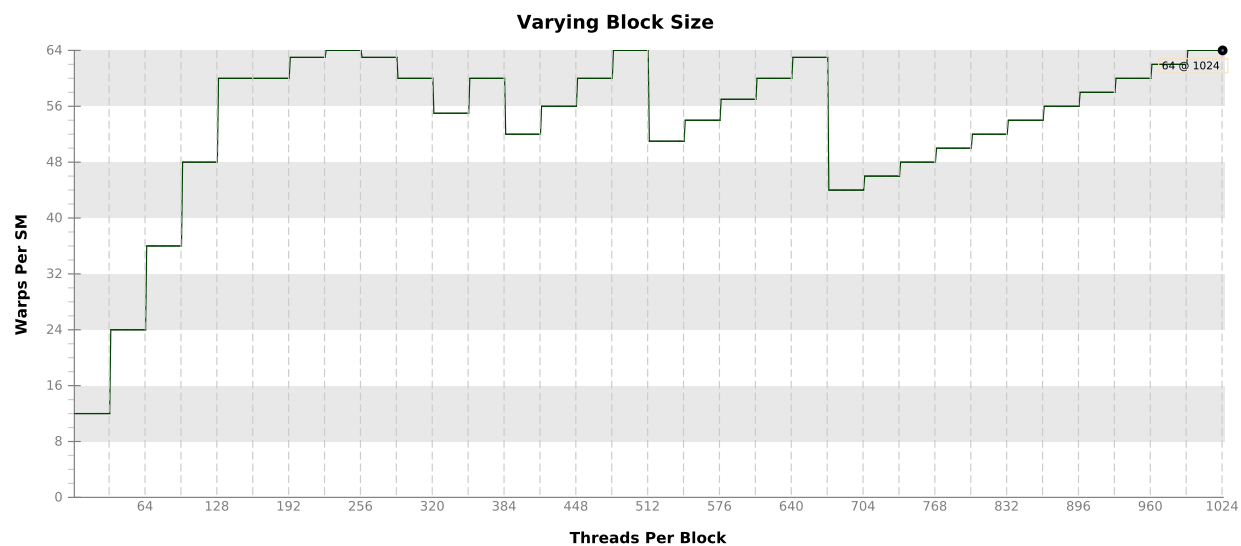
2.2. Occupancy Is Not Limiting Kernel Performance

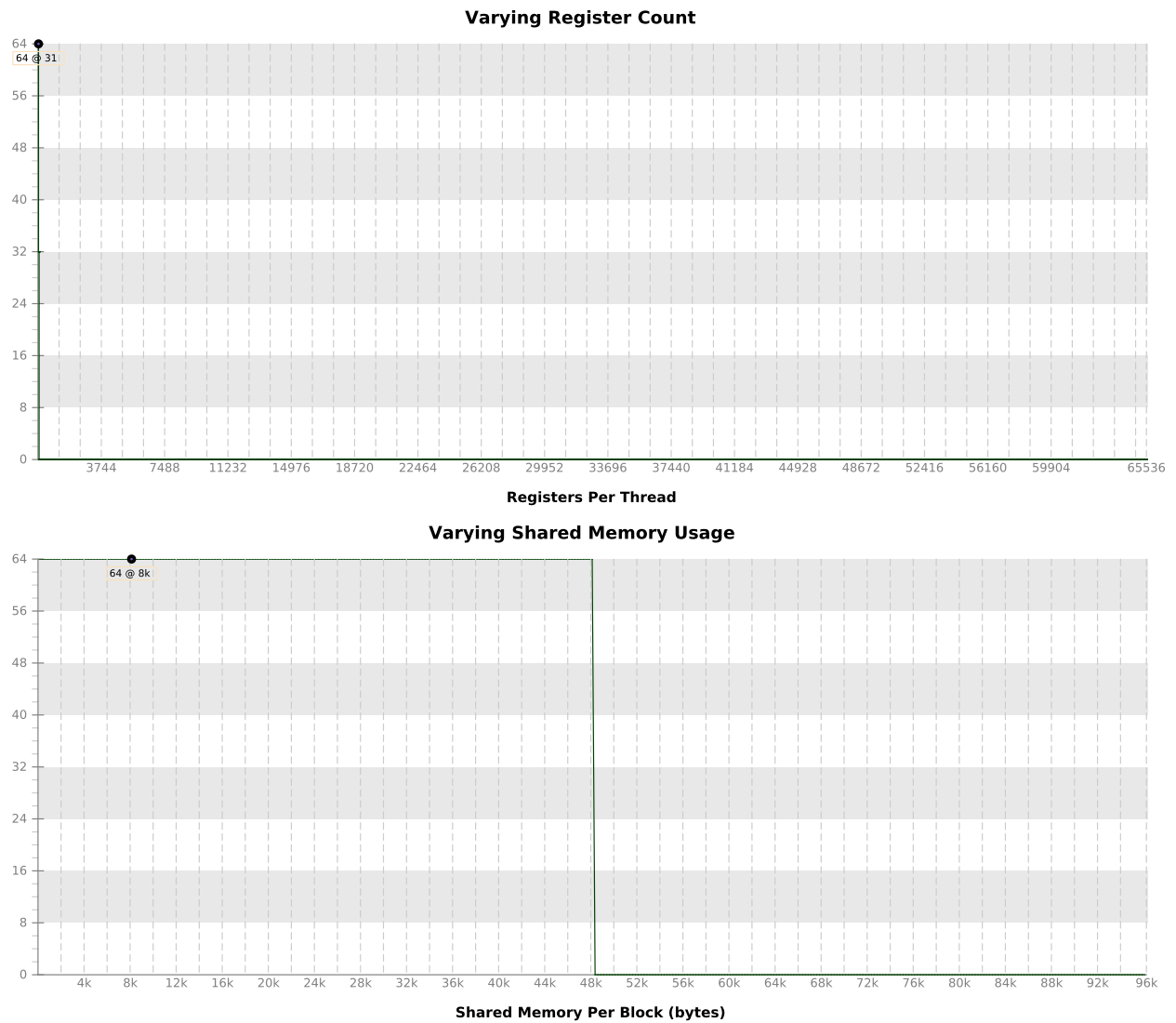
The kernel's block size, register usage, and shared memory usage allow it to fully utilize all warps on the GPU.

Variable	Achieved	Theoretical	Device Limit	Grid Size: [27,1,1000] (27000 blocks) Block Size: [32,32,1] (1024 threads)
Occupancy Per SM				
Active Blocks		2	32	
Active Warps	63.45	64	64	
Active Threads		2048	2048	
Occupancy	99.1%	100%	100%	
Warps				
Threads/Block		1024	1024	
Warps/Block		32	32	
Block Limit		2	32	
Registers				
Registers/Thread		31	65536	
Registers/Block		32768	65536	
Block Limit		2	32	
Shared Memory				
Shared Memory/Block		8192	98304	
Block Limit		12	32	

2.3. Occupancy Charts

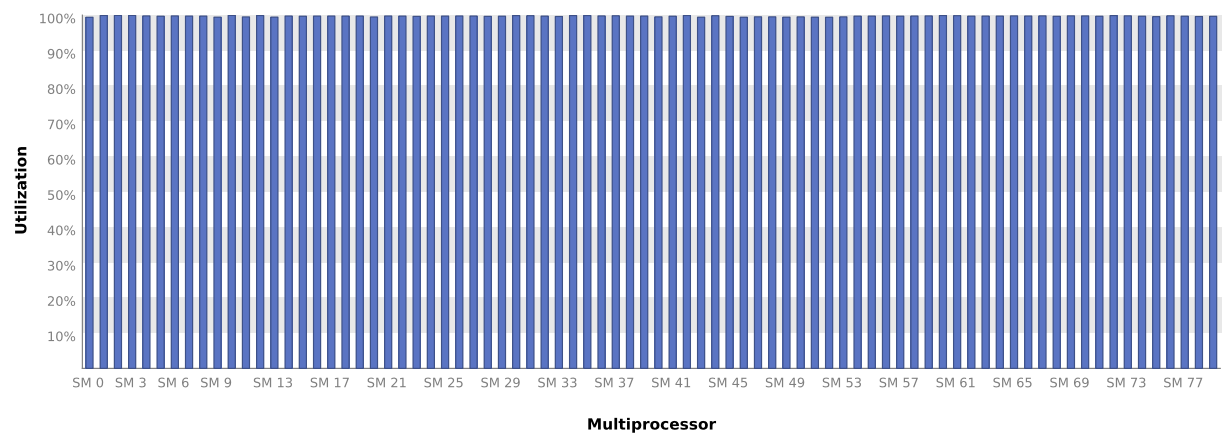
The following charts show how varying different components of the kernel will impact theoretical occupancy.





2.4. 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. 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 Functions :

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

Maximum instruction execution count in assembly: 8940000

Average instruction execution count in assembly: 5713735

Instructions executed for the kernel: 1382724000

Thread instructions executed for the kernel: 43321848000

Non-predicated thread instructions executed for the kernel: 39788227000

Warp non-predicated execution efficiency of the kernel: 89.9%

Warp execution efficiency of the kernel: 97.9%

Source files :

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

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 35	Divergence = 0% [0 divergent executions out of 8640000 total executions]
Line 35	Divergence = 0% [0 divergent executions out of 864000 total executions]
Line 43	Divergence = 3.5% [300000 divergent executions out of 8640000 total executions]
Line 45	Divergence = 2.8% [240000 divergent executions out of 8640000 total executions]
Line 54	Divergence = 0% [0 divergent executions out of 864000 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.

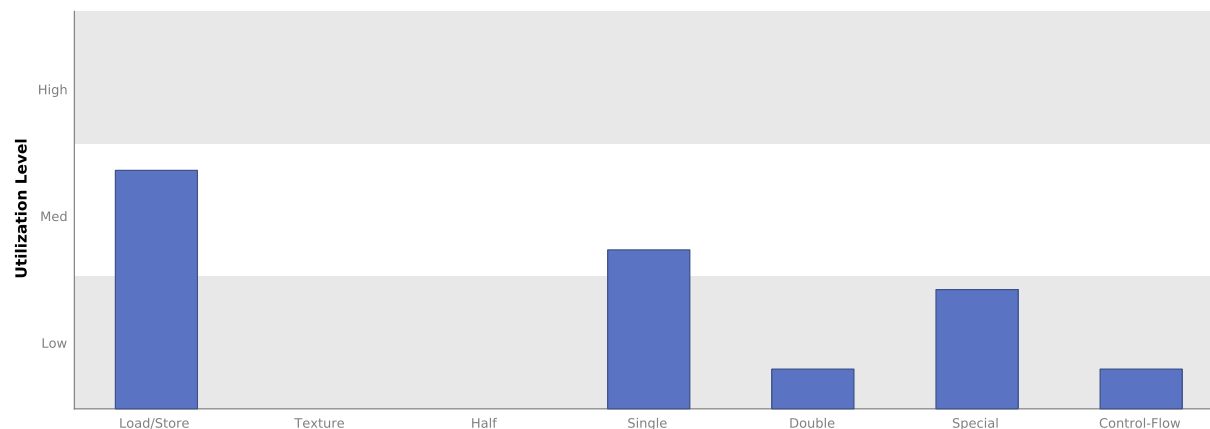
Texture - Load and store instructions for local, global, and texture memory.

Half - Half-precision floating-point arithmetic instructions.

Single - Single-precision integer and floating-point arithmetic instructions.

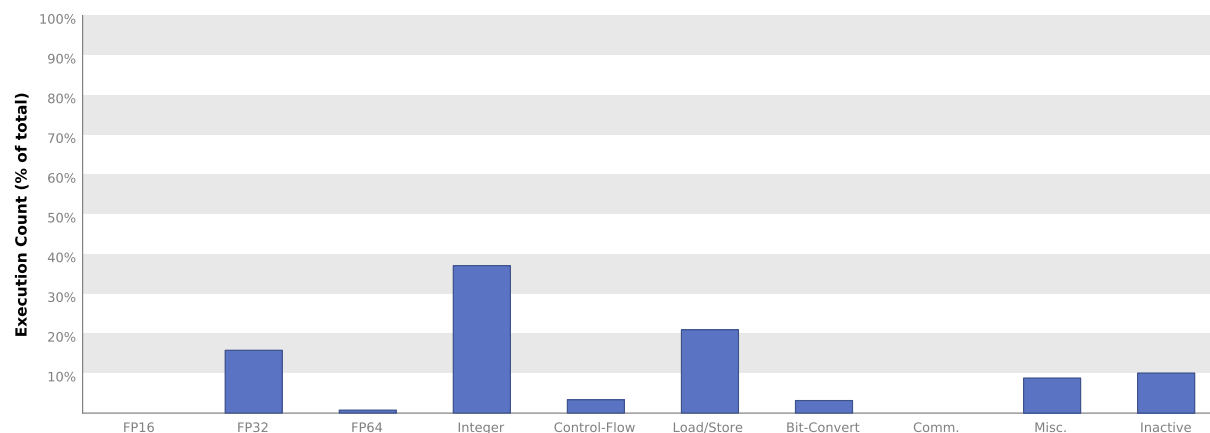
Double - Double-precision floating-point arithmetic instructions.

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



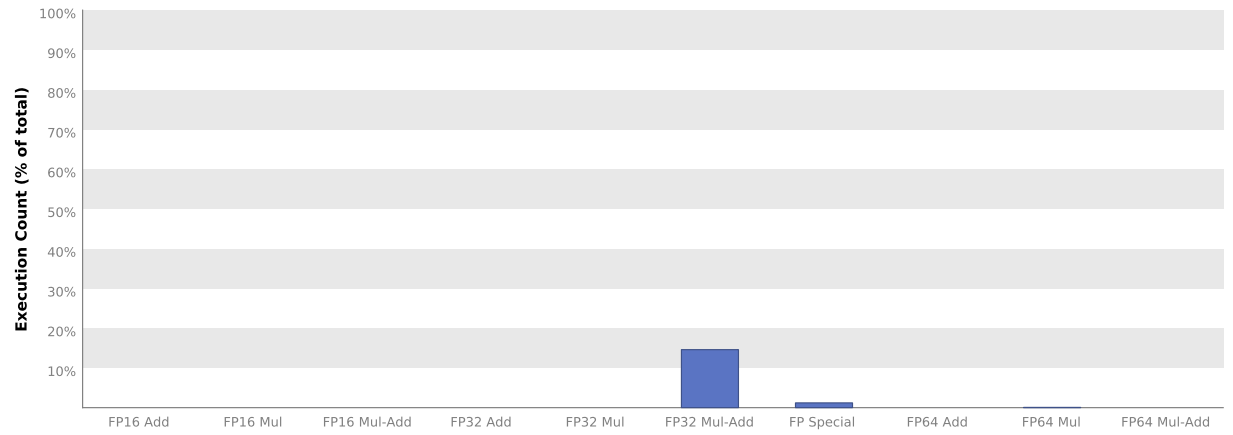
3.4. Instruction Execution Counts

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. The results below indicate that the kernel is limited by the bandwidth available to the shared memory.

4.1. 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 37	Global Load L2 Transactions/Access = 4.2, Ideal Transactions/Access = 3.8 [27540000 L2 transactions for 6480000 total executions]
Line 43	Global Load L2 Transactions/Access = 5.3, Ideal Transactions/Access = 3.9 [42825000 L2 transactions for 8100000 total executions]
Line 51	Global Store L2 Transactions/Access = 4.8, Ideal Transactions/Access = 3.9 [30900000 L2 transactions for 6480000 total executions]



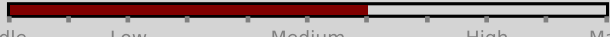


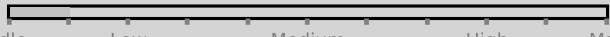
4.2. High Local Memory Overhead

Local memory loads and stores account for 74% of total memory traffic. High local memory traffic typically indicates excessive register spilling.

Optimization: Use the `-maxrregcount` flag or the `__launch_bounds__` qualifier to increase the number of registers available to `nvcc` when compiling the kernel.

4.3. 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	310628339	7,651.386 GB/s	
Shared Stores	17421742	429.132 GB/s	
Shared Total	328050081	8,080.518 GB/s	
L2 Cache			
Reads	7496784	46.165 GB/s	
Writes	30900016	190.282 GB/s	
Total	38396800	236.447 GB/s	
Unified Cache			
Local Loads	0	0 B/s	
Local Stores	0	0 B/s	
Global Loads	69658681	428.958 GB/s	
Global Stores	30900000	190.282 GB/s	
Texture Reads	328114467	8,082.104 GB/s	
Unified Total	428673148	8,701.343 GB/s	
Device Memory			
Reads	1635185	10.069 GB/s	
Writes	2523198	15.538 GB/s	
Total	4158383	25.607 GB/s	
System Memory			
[PCIe configuration: Gen3 x8, 8 Gbit/s]			
Reads	0	0 B/s	
Writes	5	30.789 kB/s	

4.4. 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.