Analysis Report

kernel_extend(HitInfo*, int)

Duration	11.32815 ms (11,328,152 ns)
Grid Size	[4801,1,1]
Block Size	[64,1,1]
Registers/Thread	61
Shared Memory/Block	0 B
Shared Memory Executed	0 B
Shared Memory Bank Size	4 B

[0] GeForce GTX 960M

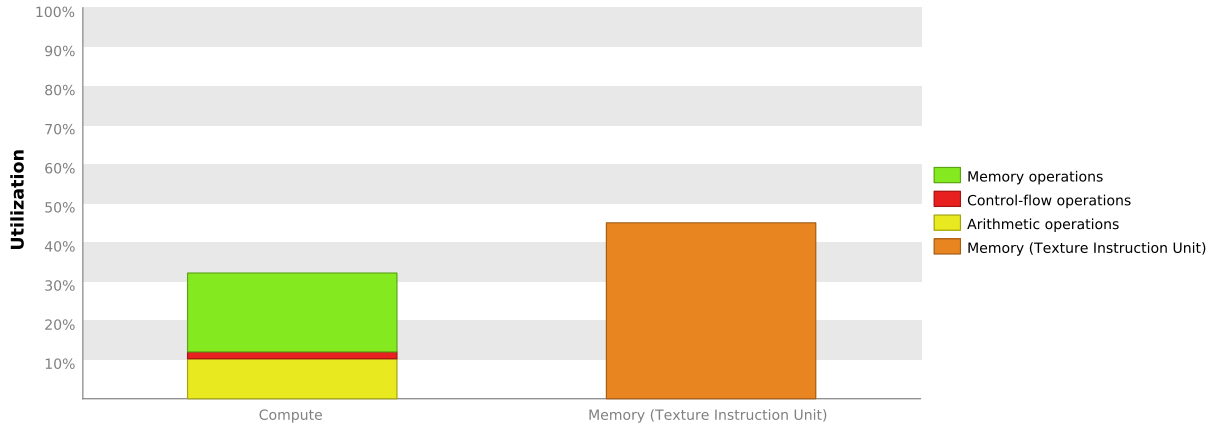
GPU UUID	GPU-d464bc85-cc44-fb1c-cd67-523a9a36c6a0
Compute Capability	5.0
Max. Threads per Block	1024
Max. Threads per Multiprocessor	2048
Max. Shared Memory per Block	48 KiB
Max. Shared Memory per Multiprocessor	64 KiB
Max. Registers per Block	65536
Max. Registers per Multiprocessor	65536
Max. Grid Dimensions	[2147483647, 65535, 65535]
Max. Block Dimensions	[1024, 1024, 64]
Max. Warps per Multiprocessor	64
Max. Blocks per Multiprocessor	32
Single Precision FLOP/s	1.505 TeraFLOP/s
Double Precision FLOP/s	47.04 GigaFLOP/s
Number of Multiprocessors	5
Multiprocessor Clock Rate	1.176 GHz
Concurrent Kernel	true
Max IPC	6
Threads per Warp	32
Global Memory Bandwidth	80.16 GB/s
Global Memory Size	1.958 GiB
Constant Memory Size	64 KiB
L2 Cache Size	2 MiB
Memcpy Engines	1
PCIe Generation	3
PCIe Link Rate	8 Gbit/s
PCIe Link Width	16

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 "kernel_extend" 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 "GeForce GTX 960M". 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. Instruction Latencies May Be Limiting Performance

Instruction stall reasons indicate the condition that prevents warps from executing on any given cycle. The following chart shows the break-down of stalls reasons averaged over the entire execution of the kernel. The kernel has good theoretical and achieved occupancy indicating that there are likely sufficient warps executing on each SM. Since occupancy is not an issue it is likely that performance is limited by the instruction stall reasons described below.

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

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

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

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

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

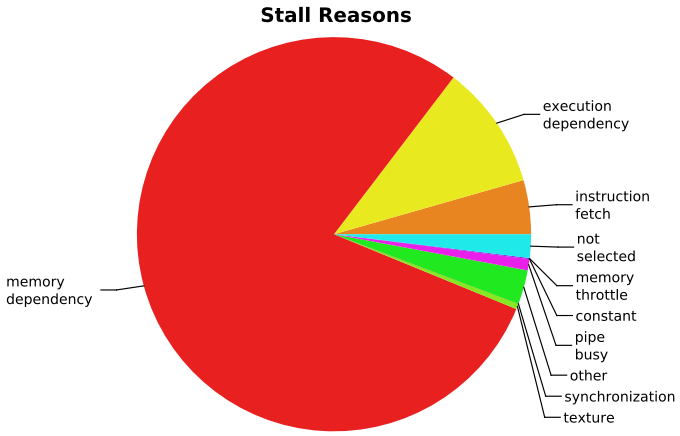
Not Selected - Warp was ready to issue, but some other warp issued instead. You may be able to sacrifice occupancy without impacting latency hiding and doing so may help improve cache hit rates.

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.

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

Optimization: Resolve the primary stall issue; memory dependency.

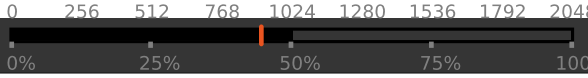
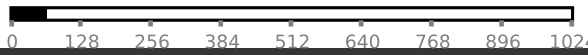
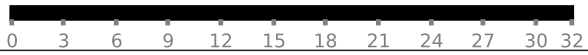
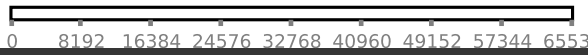
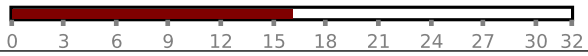


2.2. GPU Utilization May Be Limited By Register Usage

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.

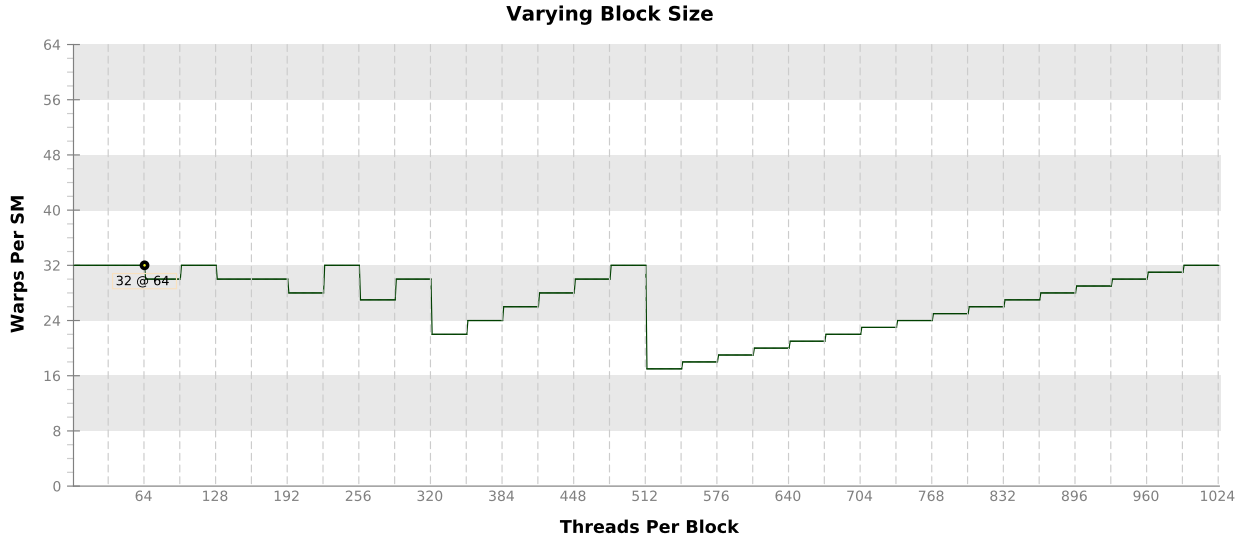
The kernel uses 61 registers for each thread (3904 registers for each block). This register usage is likely preventing the kernel from fully utilizing the GPU. Device "GeForce GTX 960M" provides up to 65536 registers for each block. Because the kernel uses 3904 registers for each block each SM is limited to simultaneously executing 16 blocks (32 warps). Chart "Varying Register Count" below shows how changing register usage will change the number of blocks that can execute on each SM.

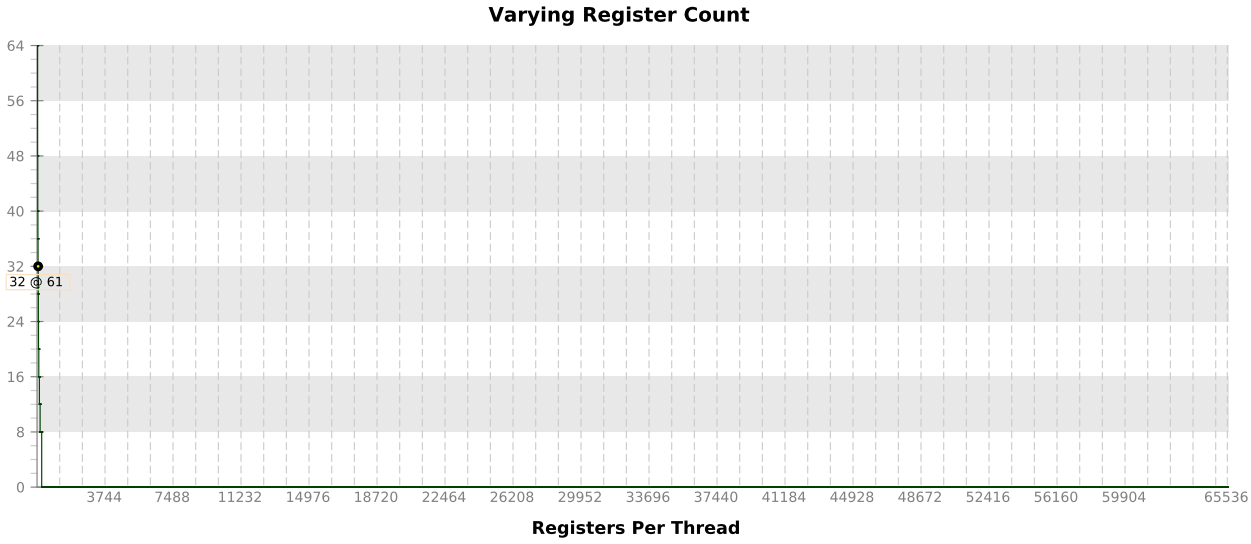
Optimization: Use the -maxrregcount flag or the __launch_bounds__ qualifier to decrease the number of registers used by each thread. This will increase the number of blocks that can execute on each SM. On devices with Compute Capability 5.2 turning global cache off can increase the occupancy limited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

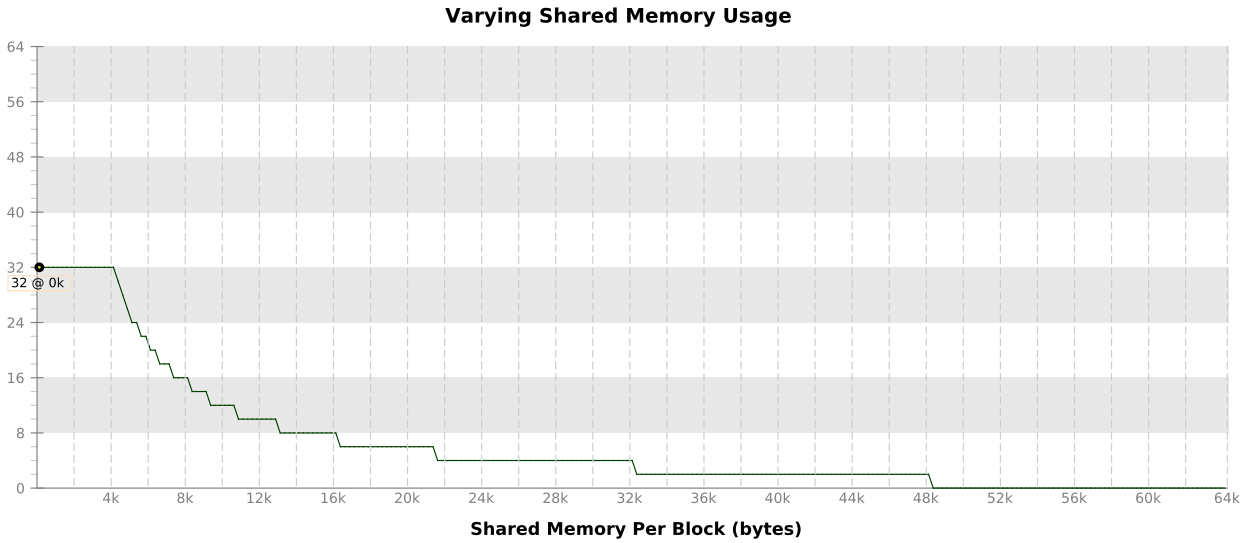
Variable	Achieved	Theoretical	Device Limit	Grid Size: [4801,1,1] (4801 blocks) Block Size: [64,1,1] (64 threa
Occupancy Per SM				
Active Blocks		16	32	
Active Warps	28.19	32	64	
Active Threads		1024	2048	
Occupancy	44.1%	50%	100%	
Warps				
Threads/Block		64	1024	
Warps/Block		2	32	
Block Limit		32	32	
Registers				
Registers/Thread		61	65536	
Registers/Block		4096	65536	
Block Limit		16	32	
Shared Memory				
Shared Memory/Block		0	65536	
Block Limit		0	32	

2.3. Occupancy Charts

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

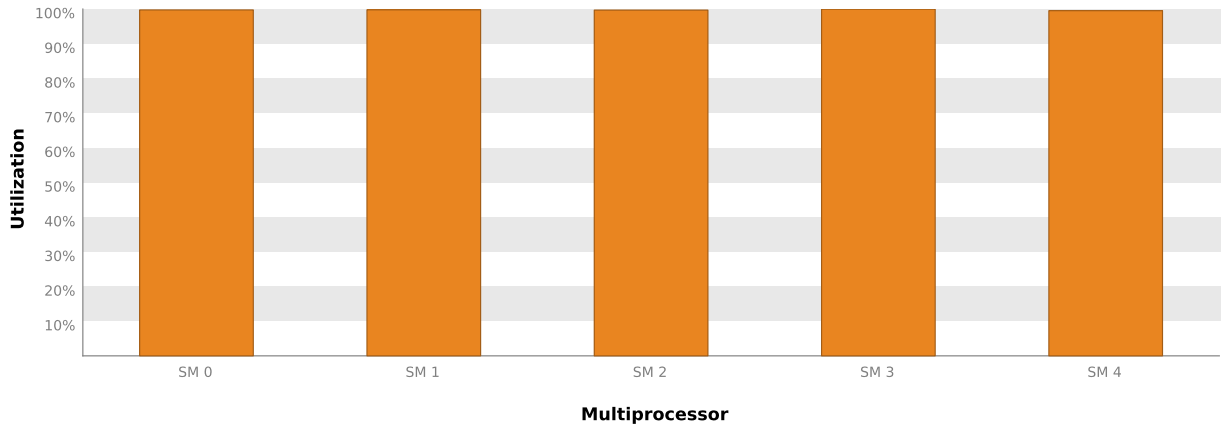

Varying Block Size
Warps Per SM
64
56
48
40
32
24
16
8
0
32 @ 64
64
128
192
256
320
384
448
512
576
640
704
768
832
896
960
1024
Threads Per Block


Varying Register Count
64
56
48
40
32
24
16
8
0
32 @ 61
3744
7488
11232
14976
18720
22464
26208
29952
33696
37440
41184
44928
48672
52416
56160
59904
65536
Registers Per Thread



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. 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 51.8% if predicated instructions are not taken into account. The kernel's not predicated off warp execution efficiency of 50.5% is less than 100% due to divergent branches and predicated instructions.

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

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.

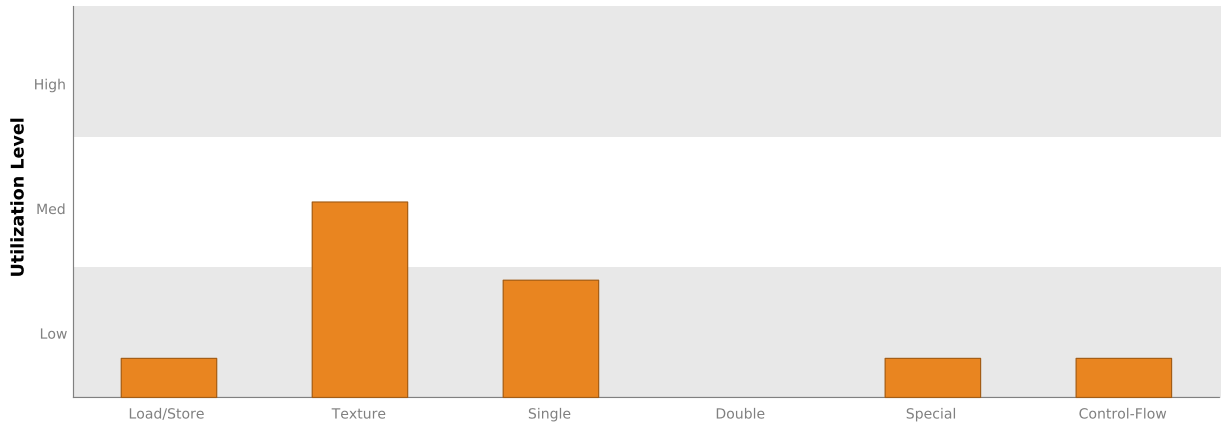
/home/hugo/repos/cuda_pathtracer/src/kernels.h

Line 46	Divergence = 4% [382 divergent executions out of 9600 total executions]
Line 65	Divergence = 0% [0 divergent executions out of 9600 total executions]
Line 65	Divergence = 0% [0 divergent executions out of 9600 total executions]
Line 169	Divergence = 45.5% [276095 divergent executions out of 607184 total executions]
Line 178	Divergence = 0% [0 divergent executions out of 9600 total executions]
Line 178	Divergence = 0% [0 divergent executions out of 9600 total executions]
Line 204	Divergence = 18.9% [82861 divergent executions out of 438287 total executions]
Line 208	Divergence = 6.2% [17877 divergent executions out of 289066 total executions]
Line 233	Divergence = 8% [48715 divergent executions out of 607184 total executions]
Line 288	Divergence = 0% [0 divergent executions out of 9602 total executions]
Line 292	Divergence = 0% [0 divergent executions out of 9600 total executions]

3.3. Function Unit Utilization

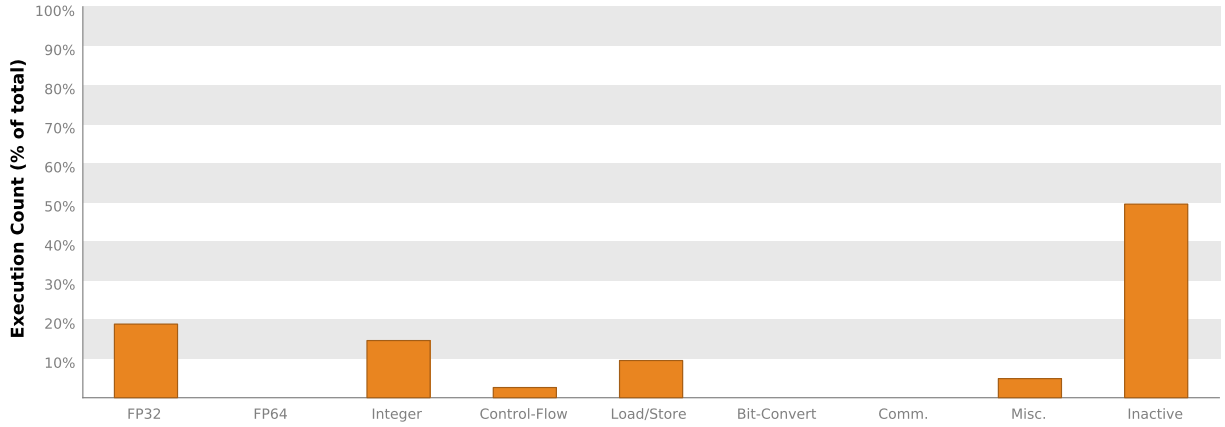
Different types of instructions are executed on different function units within each SM. Performance can be limited if a function unit is over-used by the instructions executed by the kernel. The following results show that the kernel's performance is not limited by overuse of any function unit.

Load/Store - Load and store instructions for shared and constant memory.
Texture - Load and store instructions for local, global, and texture memory.
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.



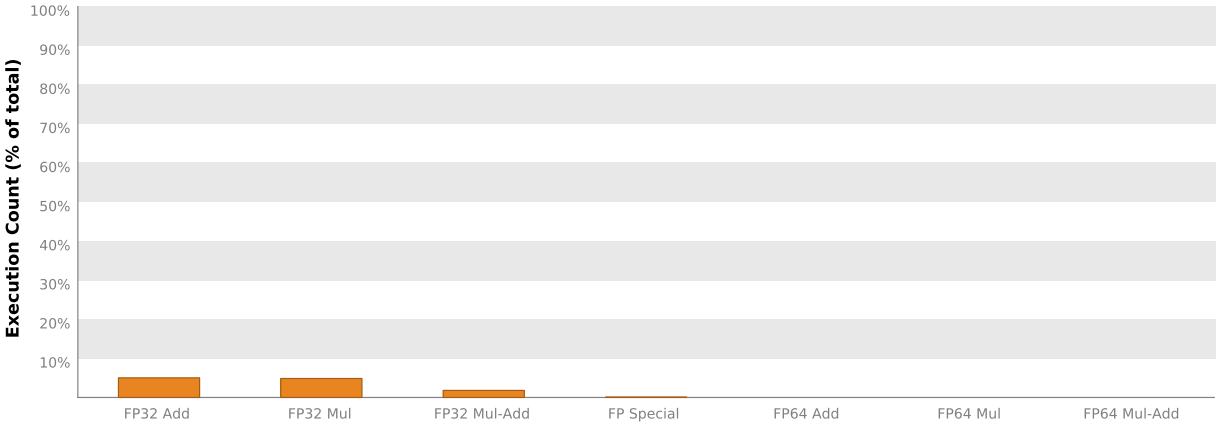
3.4. Instruction Execution Counts

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.


Execution Count (% of total)
100%
90%
80%
70%
60%
50%
40%
30%
20%
10%
FP32 Add
FP32 Mul
FP32 Mul-Add
FP Special
FP64 Add
FP64 Mul
FP64 Mul-Add

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	53388325	175.093 GB/s	
Writes	16970904	55.658 GB/s	
Total	70359229	230.751 GB/s	Idle Low Medium High Max
Unified Cache			
Local Loads	28855685	94.636 GB/s	
Local Stores	15434775	50.62 GB/s	
Global Loads	44445184	124.679 GB/s	
Global Stores	1536000	5.037 GB/s	
Texture Reads	45197167	148.229 GB/s	
Unified Total	135468811	423.201 GB/s	Idle Low Medium High Max
Device Memory			
Reads	355995	1.168 GB/s	
Writes	328840	1.078 GB/s	
Total	684835	2.246 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	16.398 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.