Analysis Report

rt_gpu_absorption(int*, float*, char*, int*, int*, float*, float*, int, int, int, float, float, float, float, float, float, float, int, float*, int, int)

Duration	1.34302 ms (1,343,025 ns)
Grid Size	[10062,1,1]
Block Size	[256,1,1]
Registers/Thread	29
Shared Memory/Block	1 KiB
Shared Memory Executed	8 KiB
Shared Memory Bank Size	4 B

[0] NVIDIA TITAN X (Pascal)

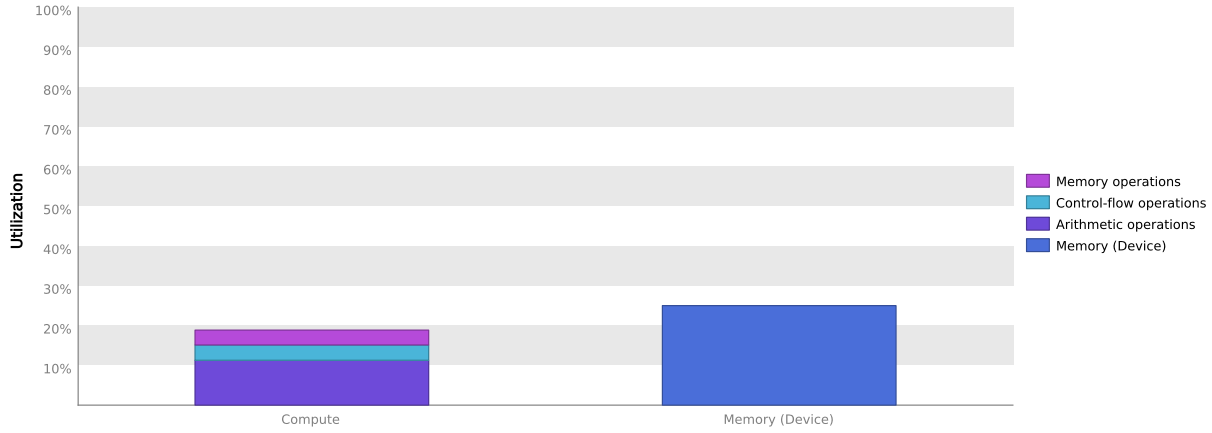
GPU UUID	GPU-49ea360c-a143-941c-a568-8f392fc23a9d
Compute Capability	6.1
Max. Threads per Block	1024
Max. Threads per Multiprocessor	2048
Max. Shared Memory per Block	48 KiB
Max. Shared Memory per Multiprocessor	96 KiB
Max. Registers per Block	65536
Max. Registers per Multiprocessor	65536
Max. Grid Dimensions	[2147483647, 65535, 65535]
Max. Block Dimensions	[1024, 1024, 64]
Max. Warps per Multiprocessor	64
Max. Blocks per Multiprocessor	32
Half Precision FLOP/s	85.736 GigaFLOP/s
Single Precision FLOP/s	10.974 TeraFLOP/s
Double Precision FLOP/s	342.944 GigaFLOP/s
Number of Multiprocessors	28
Multiprocessor Clock Rate	1.531 GHz
Concurrent Kernel	true
Max IPC	6
Threads per Warp	32
Global Memory Bandwidth	480.48 GB/s
Global Memory Size	11.904 GiB
Constant Memory Size	64 KiB
L2 Cache Size	3 MiB
Memcpy Engines	2
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 "rt_gpu_absorption" 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 "NVIDIA TITAN X (Pascal)". 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 256 [2^8] 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.

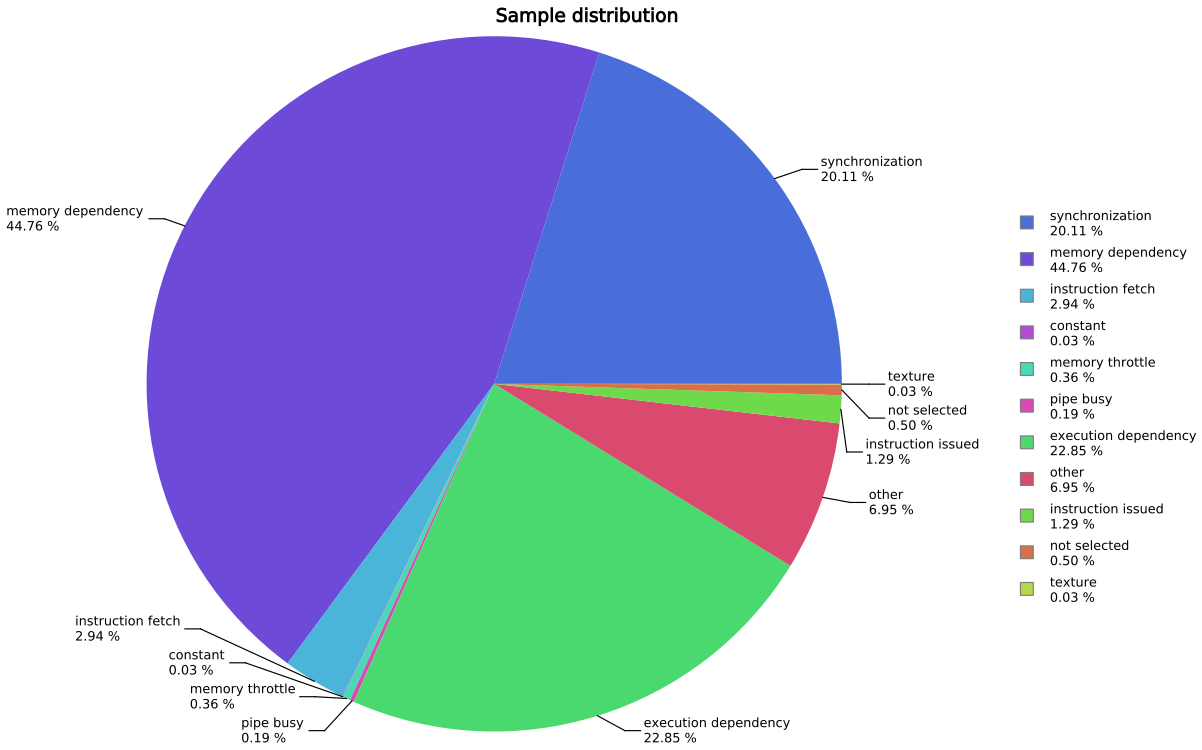
Sleeping -The warp is blocked, yielded or sleeping.

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
rt_gpu_absorption(int*, float*, char*, int*, int*, float*, float*, int, int, int, float, float, float, float, float, float, float, int, float*, int, int)	195823	100.0

Source Files :

/usr/local/cuda/targets/x86_64-linux/include/sm_30_intrinsics.hpp
/home/yishun/projectcode/Cuda/AnACor_public_fork/AnACor/float/ray_tracing_gpu_f.cu
/home/yishun/projectcode/Cuda/AnACor_public_fork/AnACor/float/GPU_reduction.cuh



2.2. GPU Utilization Is Limited By Shared Memory Usage

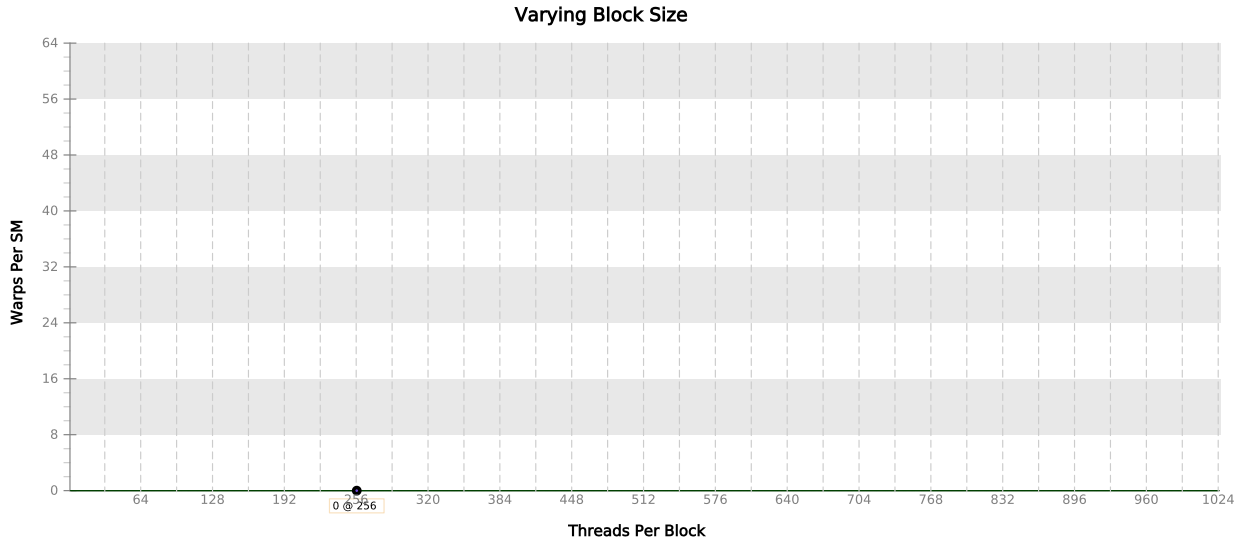
The kernel uses 1 KiB of shared memory for each block. This shared memory usage is likely preventing the kernel from fully utilizing the GPU. Device "NVIDIA TITAN X (Pascal)" is configured to have 96 KiB of shared memory for each SM. Because the kernel uses 1 KiB of shared memory for each block each SM is limited to simultaneously executing 8 blocks (64 warps). Chart "Varying Shared Memory Usage" below shows how changing shared memory usage will change the number of blocks that can execute on each SM.

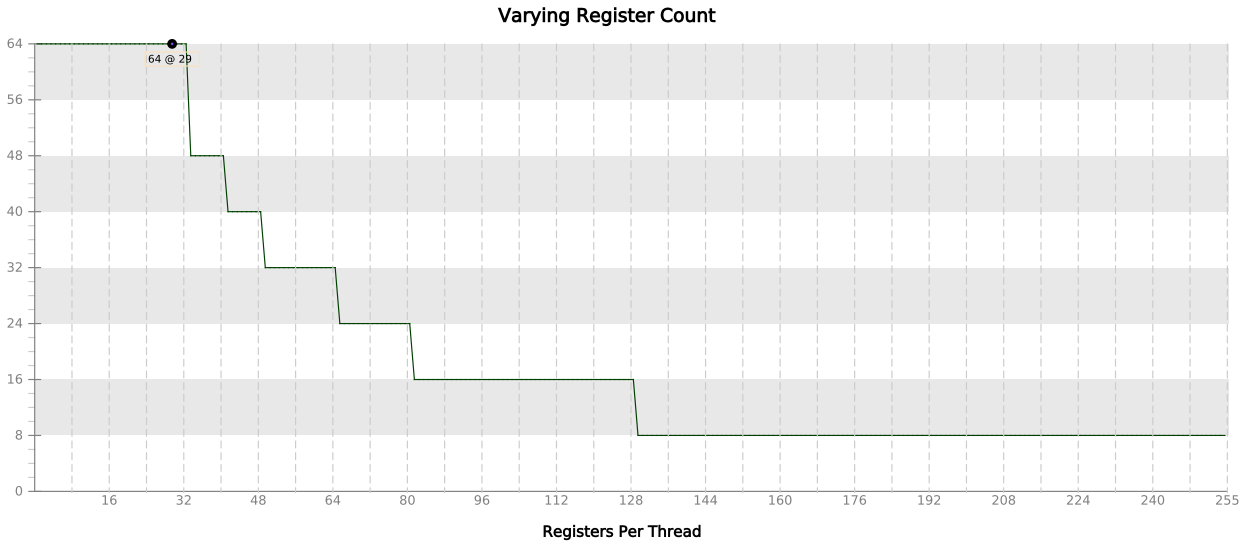
Optimization: Reduce shared memory usage to increase the number of blocks that can execute on each SM. You can also increase the number of blocks that can execute on each SM by increasing the amount of shared memory available to your kernel. You do this by setting the preferred cache configuration to "prefer shared".

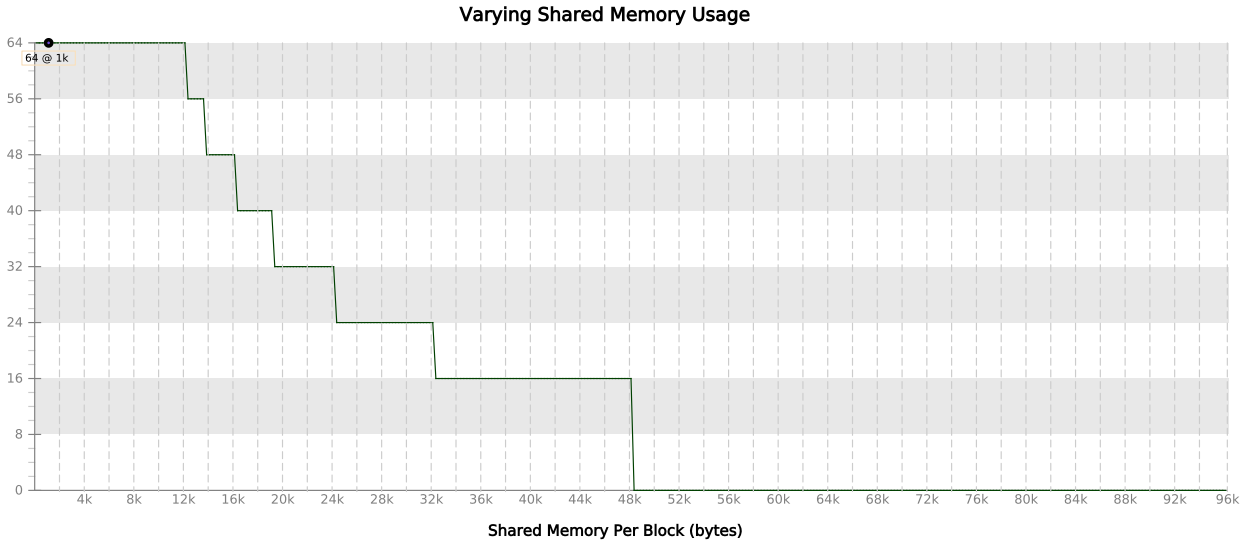
Variable	Achieved	Theoretical	Device Limit	Grid Size: [10062,1,1] (10062 blocks) Block Size: [256,1,1] (256
Occupancy Per SM				
Active Blocks		0	32	0 3 6 9 12 15 18 21 24 27 30 32
Active Warps	62.27	0	64	0 7 14 21 28 35 42 49 56 64
Active Threads		0	2048	0 256 512 768 1024 1280 1536 1792 2048
Occupancy	97.3%	0%	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		29	255	0 32 64 96 128 160 192 224 255
Registers/Block		8192	65536	0 16k 32k 48k 64k
Block Limit		8	32	0 3 6 9 12 15 18 21 24 27 30 32
Shared Memory				
Shared Memory/Block		1024	98304	0 32k 64k 96k
Block Limit		0	32	0 3 6 9 12 15 18 21 24 27 30 32

2.3. Occupancy Charts

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

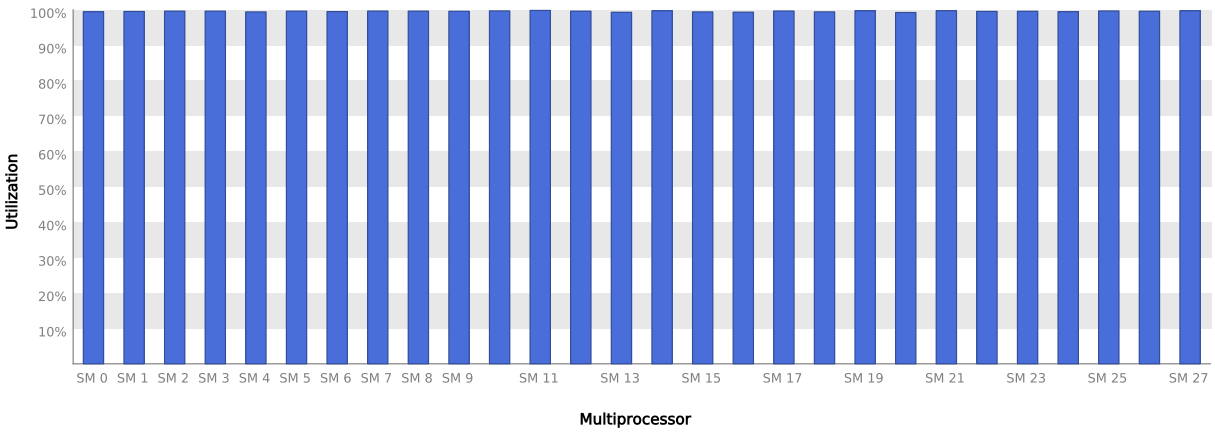






2.4. Multiprocessor Utilization

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.


100%
90%
80%
70%
60%
50%
40%
30%
20%
10%
Utilization
SM 0
SM 1
SM 2
SM 3
SM 4
SM 5
SM 6
SM 7
SM 8
SM 9
SM 11
SM 13
SM 15
SM 17
SM 19
SM 21
SM 23
SM 25
SM 27
Multiprocessor

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 Fuctions :

rt_gpu_absorption(int*, float*, char*, int*, int*, float*, float*, int, int, int, float, float, float, float, float, float, float, int, float*, int, int)

Maximum instruction execution count in assembly: 563472
Average instruction execution count in assembly: 83905
Instructions executed for the kernel: 57894701
Thread instructions executed for the kernel: 1827922642
Non-predicated thread instructions executed for the kernel: 1635516137
Warp non-predicated execution efficiency of the kernel: 88.3%
Warp execution efficiency of the kernel: 98.7%

Source files :

/usr/local/cuda/targets/x86_64-linux/include/sm_30_intrinsics.hpp
/home/yishun/projectcode/Cuda/AnACor_public_fork/AnACor/float/ray_tracing_gpu_f.cu
/home/yishun/projectcode/Cuda/AnACor_public_fork/AnACor/float/GPU_reduction.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.

/home/yishun/projectcode/Cuda/AnACor_public_fork/AnACor/float/GPU_reduction.cuh

Line 14	Divergence = 0% [0 divergent executions out of 241488 total executions]
Line 14	Divergence = 0% [0 divergent executions out of 241488 total executions]
Line 14	Divergence = 0% [0 divergent executions out of 80496 total executions]
Line 14	Divergence = 0% [0 divergent executions out of 80496 total executions]

/home/yishun/projectcode/Cuda/AnACor_public_fork/AnACor/float/ray_tracing_gpu_f.cu

Line 474	Divergence = 0% [0 divergent executions out of 236096 total executions]
Line 474	Divergence = 0% [0 divergent executions out of 517832 total executions]
Line 474	Divergence = 0% [0 divergent executions out of 563472 total executions]
Line 474	Divergence = 0% [0 divergent executions out of 45640 total executions]
Line 528	Divergence = 0% [0 divergent executions out of 45640 total executions]

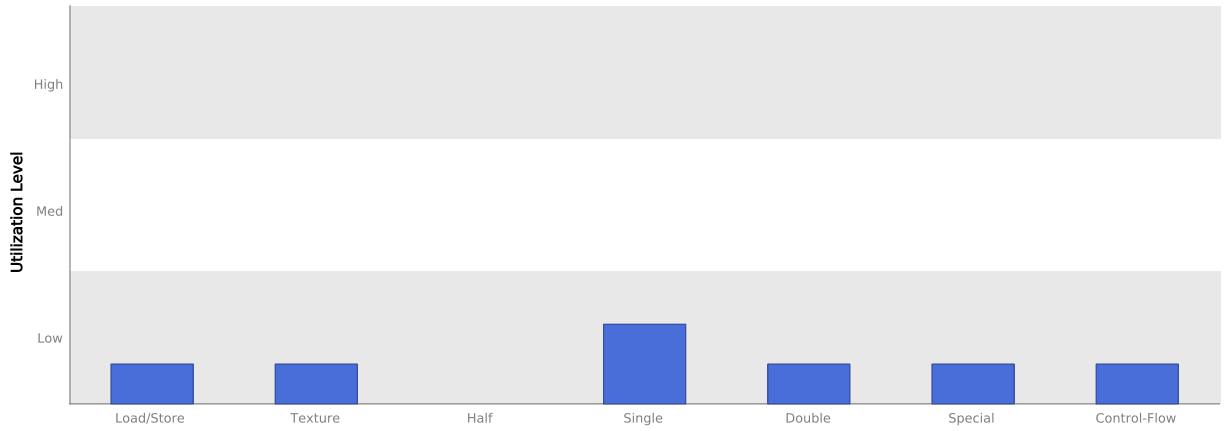
/home/yishun/projectcode/Cuda/AnACor_public_fork/AnACor/float/ray_tracing_gpu_f.cu

Line 530	Divergence = 0% [0 divergent executions out of 45640 total executions]
Line 534	Divergence = 0% [0 divergent executions out of 45640 total executions]
Line 561	Divergence = 0% [0 divergent executions out of 281736 total executions]
Line 571	Divergence = 0% [0 divergent executions out of 281736 total executions]
Line 591	Divergence = 0% [0 divergent executions out of 236096 total executions]
Line 595	Divergence = 0% [0 divergent executions out of 236096 total executions]
Line 610	Divergence = 0% [0 divergent executions out of 80496 total executions]
Line 610	Divergence = 0% [0 divergent executions out of 80496 total executions]
Line 626	Divergence = 0% [0 divergent executions out of 6520 total executions]
Line 638	Divergence = 0% [0 divergent executions out of 33728 total executions]
Line 640	Divergence = 0% [0 divergent executions out of 33728 total executions]
Line 691	Divergence = 0% [0 divergent executions out of 80496 total executions]
Line 743	Divergence = 0% [0 divergent executions out of 563472 total executions]
Line 743	Divergence = 0% [0 divergent executions out of 80496 total executions]
Line 776	Divergence = 6% [9925 divergent executions out of 165838 total executions]
Line 851	Divergence = 12.5% [10062 divergent executions out of 80496 total executions]
Line 860	Divergence = 0% [0 divergent executions out of 294814 total executions]
Line 860	Divergence = 0% [0 divergent executions out of 294814 total executions]
Line 860	Divergence = 0% [0 divergent executions out of 294814 total executions]
Line 860	Divergence = 0% [0 divergent executions out of 294814 total executions]
Line 860	Divergence = 0% [0 divergent executions out of 10062 total executions]
Line 860	Divergence = 0% [0 divergent executions out of 294814 total executions]
Line 860	Divergence = 0% [0 divergent executions out of 294814 total executions]

3.3. Function Unit Utilization

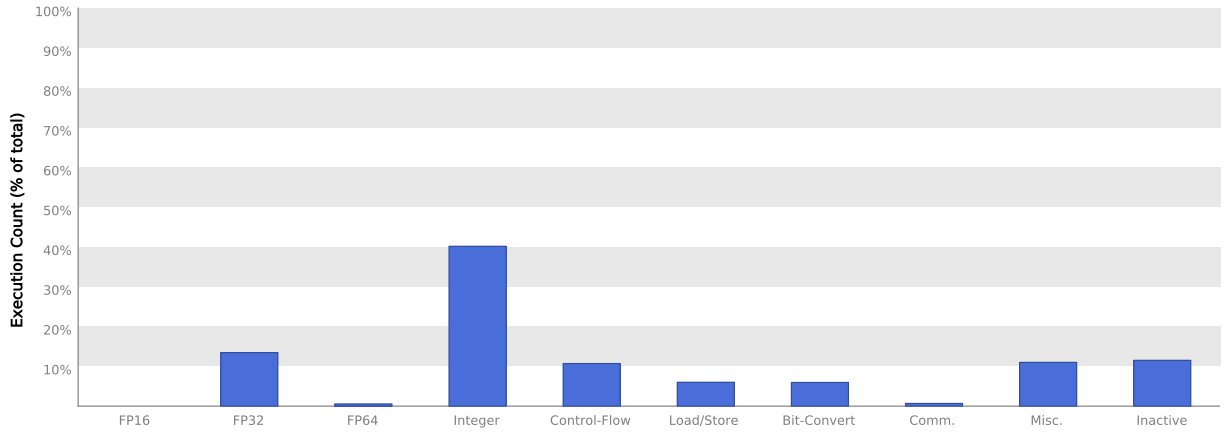
Different types of instructions are executed on different function units within each SM. Performance can be limited if a function unit is over-used by the instructions executed by the kernel. The following results show that the kernel's performance is not limited by overuse of any function unit.

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.



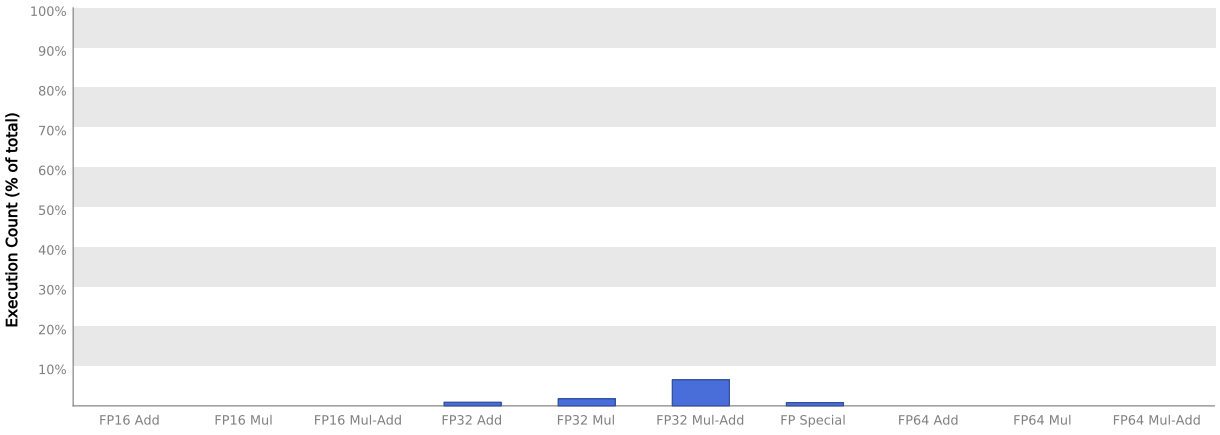
3.4. Instruction Execution Counts

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.


Execution Count (% of total)
100%
90%
80%
70%
60%
50%
40%
30%
20%
10%
FP16 Add
FP16 Mul
FP16 Mul-Add
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	362232	34.523 GB/s	
Shared Stores	301860	28.769 GB/s	
Shared Total	664092	63.293 GB/s	Idle Low Medium High Max
L2 Cache			
Reads	5130047	122.233 GB/s	
Writes	2684040	63.952 GB/s	
Total	7814087	186.185 GB/s	Idle Low Medium High Max
Unified Cache			
Local Loads	0	0 B/s	
Local Stores	0	0 B/s	
Global Loads	10954211	122.194 GB/s	
Global Stores	2684025	63.952 GB/s	
Texture Reads	3226593	76.879 GB/s	
Unified Total	16864829	263.025 GB/s	Idle Low Medium High Max
Device Memory			
Reads	2772406	66.058 GB/s	
Writes	2226642	53.054 GB/s	
Total	4999048	119.111 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	119.134 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.