Add Baseline
Apply Rules
Occupancy Calculator
Save as PDF

Page: Details   Result: 0 - 536 - kernel

	Result	Time	Cycles	Regs	GPU	SM Frequency	CC	Process
Current	536 - kernel (512, 288, 1)x(16, 16, 1)	110.25 msecond	165,314,067	29	0 - NVIDIA GeForce RTX 3070	1.50 cycle/nsecond	8.6	[13932] MandelbrotSet.exe

Command line profiler metrics

l1tex__t_requests_pipe_lsu_mem_global_op_ld.sum [request]	0	l1tex__t_sectors_pipe_lsu_mem_global_op_ld.sum [sector]	0

GPU Speed Of Light Throughput

High-level overview of the throughput for compute and memory resources of the GPU. For each unit, the throughput reports the achieved percentage of utilization with respect to the theoretical maximum. Breakdowns show the throughput for each individual sub-metric of Compute and Memory to clearly identify the highest contributor. High-level overview of the utilization for compute and memory resources of the GPU presented as a roofline chart.

Compute (SM) Throughput [%]	87.78	Duration [msecond]	110.25
Memory Throughput [%]	0.67	Elapsed Cycles [cycle]	165,314,067
L1/TEX Cache Throughput [%]	0.81	SM Active Cycles [cycle]	165,260,508.28
L2 Cache Throughput [%]	0.67	SM Frequency [cycle/nsecond]	1.50
DRAM Throughput [%]	0.49	DRAM Frequency [cycle/nsecond]	6.79

High Throughput The kernel is utilizing greater than 80.0% of the available compute or memory performance of the device. To further improve performance, work will likely need to be shifted from the most utilized to another unit. Start by analyzing workloads in the Compute Workload Analysis section.

FP64/32 Utilization The ratio of peak float (fp32) to double (fp64) performance on this device is 64:1. The kernel achieved close to 0% of this device's fp32 peak performance and 18% of its fp64 peak performance. If Compute Workload Analysis determines that this kernel is fp64 bound, consider using 32-bit precision floating point operations to improve its performance. See the Kernel Profiling Guide for more details on roofline analysis.

FP64/32 Utilization The achieved fp64 performance is 70% lower than the fp64 pipeline utilization. Check the Instruction Statistics section to see if using fused instructions can benefit this kernel.

Compute Workload Analysis

Detailed analysis of the compute resources of the streaming multiprocessors (SM), including the achieved instructions per clock (IPC) and the utilization of each available pipeline. Pipelines with very high utilization might limit the overall performance.

Executed Ipc Elapsed [inst/cycle]	0.47	SM Busy [%]	87.81
Executed Ipc Active [inst/cycle]	0.47	Issue Slots Busy [%]	11.81
Issued Ipc Active [inst/cycle]	0.47		

Very High Utilization FP64 is the highest-utilized pipeline (87.8%). It executes 64-bit floating point operations. The pipeline is over-utilized and likely a performance bottleneck. See the Kernel Profiling Guide or hover over the pipeline name to understand the workloads handled by each pipeline. The Instruction Statistics section shows the mix of executed instructions in this kernel. Check the Warp State Statistics section for which reasons cause warps to stall.

Memory Workload Analysis

Detailed analysis of the memory resources of the GPU. Memory can become a limiting factor for the overall kernel performance when fully utilizing the involved hardware units (Mem Busy), exhausting the available communication bandwidth between those units (Max Bandwidth), or by reaching the maximum throughput of issuing memory instructions (Mem Pipes Busy). Detailed chart of the memory units. Detailed tables with data for each memory unit.

Memory Throughput [Gbyte/second]	2.11	Mem Busy [%]	0.54
L1/TEX Hit Rate [%]	83.26	Max Bandwidth [%]	0.67
L2 Hit Rate [%]	89.61	Mem Pipes Busy [%]	12.10
L2 Compression Success Rate [%]	0	L2 Compression Ratio	0

L1TEX Local Load Access Pattern The memory access pattern for local loads in L1TEX might not be optimal. On average, this kernel accesses 2.0 bytes per thread per memory request; but the address pattern, possibly caused by the stride between threads, results in 4.0 sectors per request, or 4.0*32 = 127.3 bytes of cache data transfers per request. The optimal thread address pattern for 2.0 byte accesses would result in 2.0*32 = 64.0 bytes of cache data transfers per request, to maximize L1TEX cache performance. Check the Source Counters section for uncoalesced local loads.

L1TEX Global Store Access Pattern The memory access pattern for global stores in L1TEX might not be optimal. On average, this kernel accesses 2.0 bytes per thread per memory request; but the address pattern, possibly caused by the stride between threads, results in 4.0 sectors per request, or 4.0*32 = 127.3 bytes of cache data transfers per request. The optimal thread address pattern for 2.0 byte accesses would result in 2.0*32 = 64.0 bytes of cache data transfers per request, to maximize L1TEX cache performance. Check the Source Counters section for uncoalesced global stores.

L1TEX Local Store Access Pattern The memory access pattern for local stores in L1TEX might not be optimal. On average, this kernel accesses 2.0 bytes per thread per memory request; but the address pattern, possibly caused by the stride between threads, results in 4.0 sectors per request, or 4.0*32 = 127.3 bytes of cache data transfers per request. The optimal thread address pattern for 2.0 byte accesses would result in 2.0*32 = 64.0 bytes of cache data transfers per request, to maximize L1TEX cache performance. Check the Source Counters section for uncoalesced local stores.

L2 Store Access Pattern The memory access pattern for stores from L1TEX to L2 is not optimal. The granularity of an L1TEX request to L2 is a 128 byte cache line. That is 4 consecutive 32-byte sectors per L2 request. However, this kernel only accesses an average of 3.2 sectors out of the possible 4 sectors per cache line. Check the Source Counters section for uncoalesced stores and try to minimize how many cache lines need to be accessed per memory request.

Scheduler Statistics

Summary of the activity of the schedulers issuing instructions. Each scheduler maintains a pool of warps that it can issue instructions for. The upper bound of warps in the pool (Theoretical Warps) is limited by the launch configuration. On every cycle each scheduler checks the state of the allocated warps in the pool (Active Warps). Active warps that are not stalled (Eligible Warps) are ready to issue their next instruction. From the set of eligible warps the scheduler selects a single warp from which to issue one or more instructions (Issued Warp). On cycles with no eligible warps, the issue slot is skipped and no instruction is issued. Having many skipped issue slots indicates poor latency hiding.

Active Warps Per Scheduler [warp]	10.51	No Eligible [%]	88.19
Eligible Warps Per Scheduler [warp]	0.18	One or More Eligible [%]	11.81
Issued Warp Per Scheduler	0.12		

Issue Slot Utilization Every scheduler is capable of issuing one instruction per cycle, but for this kernel each scheduler only issues an instruction every 8.5 cycles. This might leave hardware resources underutilized and may lead to less optimal performance. Out of the maximum of 12 warps per scheduler, this kernel allocates an average of 10.51 active warps per scheduler, but only an average of 0.18 warps were eligible per cycle. Eligible warps are the subset of active warps that are ready to issue their next instruction. Every cycle with no eligible warp results in no instruction being issued and the issue slot remains unused. To increase the number of eligible warps, avoid possible load imbalances due to highly different execution durations per warp. Reducing stalls indicated on the Warp State Statistics and Source Counters sections can help, too.

Warp State Statistics

Analysis of the states in which all warps spent cycles during the kernel execution. The warp states describe a warp's readiness or inability to issue its next instruction. The warp cycles per instruction define the latency between two consecutive instructions. The higher the value, the more warp parallelism is required to hide this latency. For each warp state, the chart shows the average number of cycles spent in that state per issued instruction. Stalls are not always impacting the overall performance nor are they completely avoidable. Only focus on stall reasons if the schedulers fail to issue every cycle. When executing a kernel with mixed library and user code, these metrics show the combined values.

Warp Cycles Per Issued Instruction [cycle]	89.03	Avg. Active Threads Per Warp	30.57
Warp Cycles Per Executed Instruction [cycle]	89.03	Avg. Not Predicated Off Threads Per Warp	29.55

short_scoreboard On average, each warp of this kernel spends 38.1 cycles being stalled waiting for a scoreboard dependency on an MIO operation (not to TEX or L1). This represents about 42.8% of the total average of 89.0 cycles between issuing two instructions. The primary reason for a high number of stalls due to short scoreboards is typically memory operations to shared memory, but other contributors include frequent execution of special math instructions (e.g. MUFU) or dynamic branching (e.g. BRX, JMX). Consult the Memory Workload Analysis section to verify if there are shared memory operations and reduce bank conflicts, if reported. Assigning frequently accessed values to variables can assist the compiler in using low-latency registers instead of direct memory accesses.

tex_throttle On average, each warp of this kernel spends 35.3 cycles being stalled waiting for the L1TEX instruction queue to be not full. This represents about 39.7% of the total average of 89.0 cycles between issuing two instructions. This stall reason is high in cases of utilization of the L1TEX pipeline. If applicable, consider combining multiple lower-width memory operations into fewer wider memory operations and try interleaving memory operations and math instructions.

Warp Stall Check the Source Counters section for the top stall locations in your source based on sampling data. The Kernel Profiling Guide provides more details on each stall reason.

Instruction Statistics

Statistics of the executed low-level assembly instructions (SASS). The instruction mix provides insight into the types and frequency of the executed instructions. A narrow mix of instruction types implies a dependency on few instruction pipelines, while others remain unused. Using multiple pipelines allows hiding latencies and enables parallel execution. Note that 'Instructions/Opcode' and 'Executed Instructions' are measured differently and can diverge if cycles are spent in system calls.

Executed Instructions [inst]	3,589,923,116	Avg. Executed Instructions Per Scheduler [inst]	19,510,451.72
Issued Instructions [inst]	3,589,939,286	Avg. Issued Instructions Per Scheduler [inst]	19,510,539.60

FP32/64 Instructions This kernel executes 0 fused and 306265766 non-fused FP32 instructions. By converting pairs of non-fused instructions to their fused, higher-throughput equivalent, the achieved FP32 performance could be increased by up to 50% (relative to its current performance). Check the Source page to identify where this kernel executes FP32 instructions.

FP32/64 Instructions This kernel executes 0 fused and 178804938 non-fused FP64 instructions. By converting pairs of non-fused instructions to their fused, higher-throughput equivalent, the achieved FP64 performance could be increased by up to 50% (relative to its current performance). Check the Source page to identify where this kernel executes FP64 instructions.

NVLink Topology

NVLink Topology diagram shows logical NVLink connections with transmit/receive throughput.

NVLink Tables

Detailed tables with properties for each NVLink.

Launch Statistics

Summary of the configuration used to launch the kernel. The launch configuration defines the size of the kernel grid, the division of the grid into blocks, and the GPU resources needed to execute the kernel. Choosing an efficient launch configuration maximizes device utilization.

Grid Size	147,456	Registers Per Thread [register/thread]	29
Block Size	256	Static Shared Memory Per Block [byte/block]	0
Threads [thread]	37,748,736	Dynamic Shared Memory Per Block [byte/block]	0
Waves Per SM	534.26	Driver Shared Memory Per Block [Kbyte/block]	1.02
Function Cache Configuration	cudaFuncCachePreferNone	Shared Memory Configuration Size [Kbyte]	8.19

Occupancy

Occupancy is the ratio of the number of active warps per multiprocessor to the maximum number of possible active warps. Another way to view occupancy is the percentage of the hardware's ability to process warps that is actively in use. Higher occupancy does not always result in higher performance, however, low occupancy always reduces the ability to hide latencies, resulting in overall performance degradation. Large discrepancies between the theoretical and the achieved occupancy during execution typically indicates highly imbalanced workloads.

Theoretical Occupancy [%]	100	Block Limit Registers [block]	8
Theoretical Active Warps per SM [warp]	48	Block Limit Shared Mem [block]	8
Achieved Occupancy [%]	87.13	Block Limit Warps [block]	6
Achieved Active Warps Per SM [warp]	41.82	Block Limit SM [block]	16

Occupancy Limiters This kernel's theoretical occupancy is not impacted by any block limit. The difference between calculated theoretical (100.0%) and measured achieved occupancy (87.1%) can be the result of warp scheduling overheads or workload imbalances during the kernel execution. Load imbalances can occur between warps within a block as well as across blocks of the same kernel. See the CUDA Best Practices Guide for more details on optimizing occupancy.

Source Counters

Source metrics, including branch efficiency and sampled warp stall reasons. Warp Stall Sampling metrics are periodically sampled over the kernel runtime. They indicate when warps were stalled and couldn't be scheduled. See the documentation for a description of all stall reasons. Only focus on stalls if the schedulers fail to issue every cycle.

Branch Instructions [inst]	431,072,264	Branch Efficiency [%]	99.61
Branch Instructions Ratio [%]	0.12	Avg. Divergent Branches	7,761.27

Uncoalesced Global Accesses This kernel has uncoalesced global accesses resulting in a total of 17363 excessive sectors (0% of the total 4749409 sectors). Check the L2 Theoretical Sectors Global Excessive table for the primary source locations. The CUDA Programming Guide had additional information on reducing uncoalesced device memory accesses.

L2 Theoretical Sectors Global Excessive

Location	Value	Value (%)
kernel.cu:95 (0x700bbc6f0 in kernel)	9,632	55
kernel.cu:105 (0x700bbcb90 in kernel)	7,731	45