KERNEL PROFILING USING NSIGHT

GPA - 2

FRANK, HERMAN, HASAN,

ANSWER:

How do you know your code is running fast?

How do you know your code is memory efficient?

How do you know you’re fully utilizing your SMs?

ANSWER:

HOW DO WE USE NSIGHT

OVERVIE

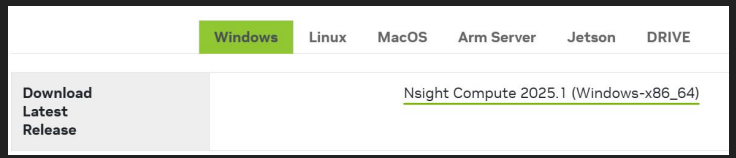
- Installation & Running
- Analysis
- Matrix Multiplication Example
- Performance Metrics Deep Dive

INSTALLATIO

INSTALLATION

1) Install Nsight Compute

   a) Go to: https://developer.nvidia.com/tools-overview/nsight-compute/get-started



RUNNING

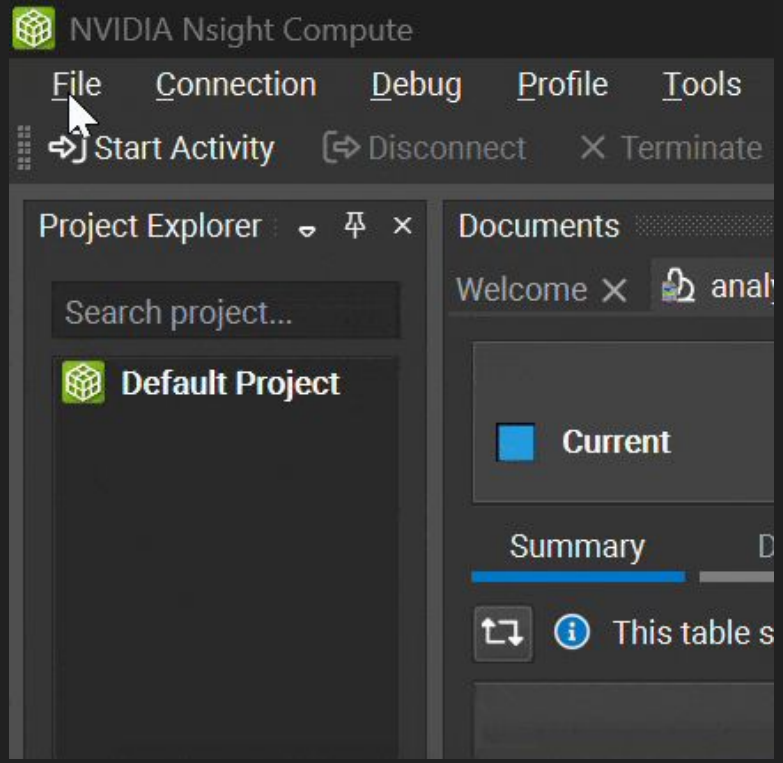
- Modify your slurm file
  - Change --constraint from --constraint="projects" to --constraint="projects,perf,nvperf"
  - Add to file: ncu --set full -f -o outputFileName ./test_file > test_file.out

ANALYSI S

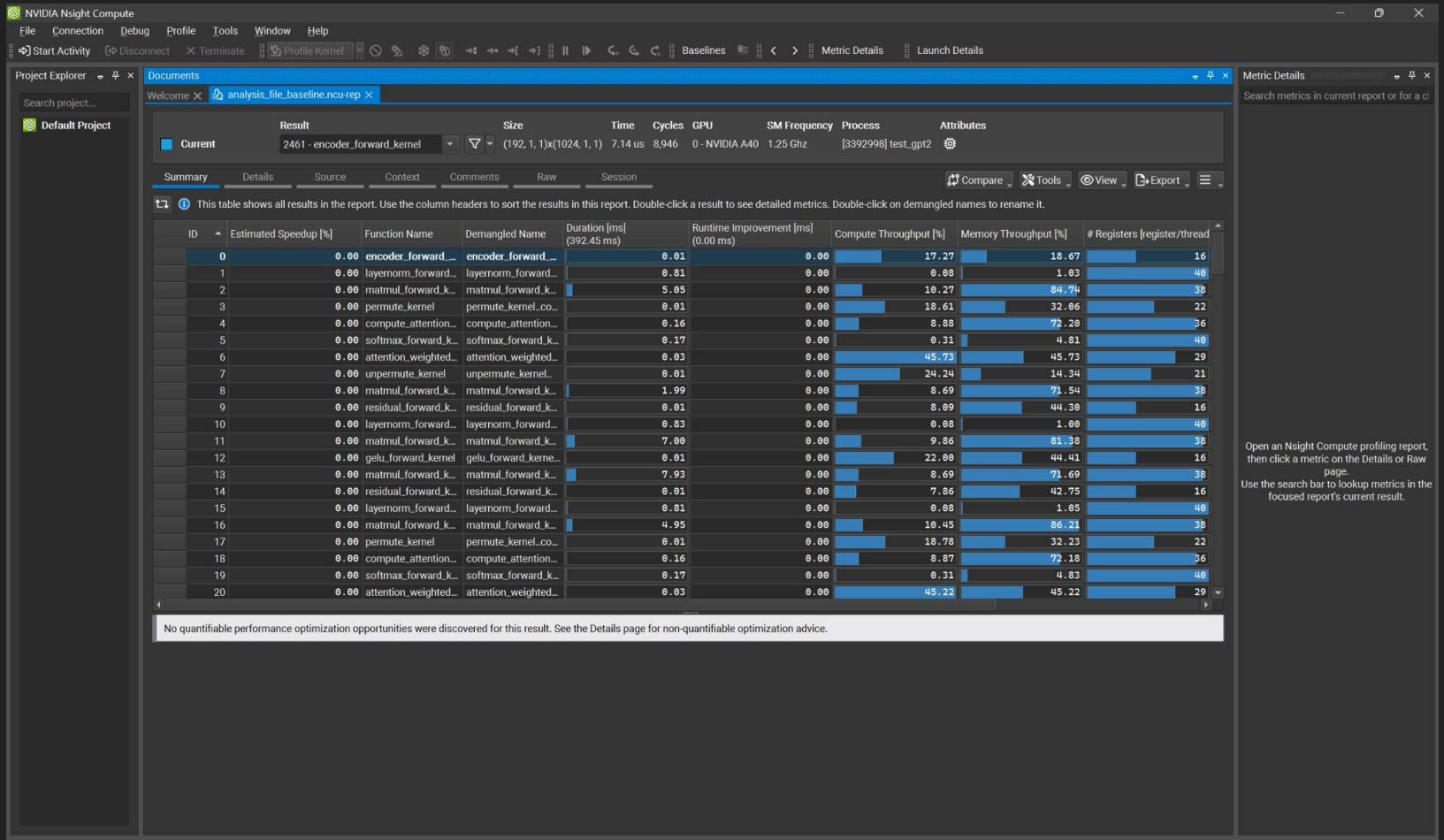
ANALYSI

To analyze:

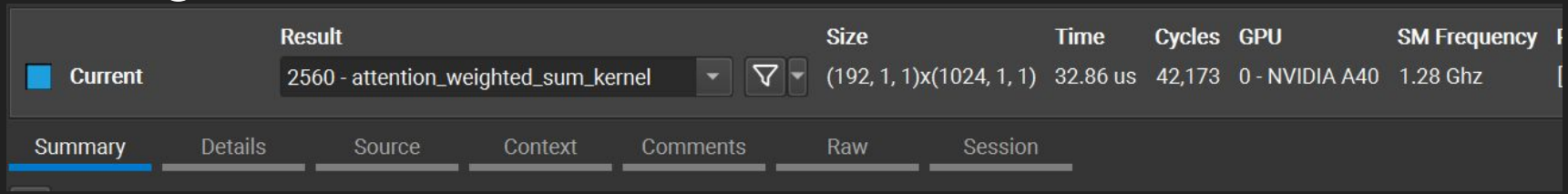
- Download the .ncu-rep file to your machine
- Load the file into Nsight Compute!



Here’s how it should look like!



Starting info:



SUMMAR Shows all kernels and executions, alongside basic runtime info.

DETAIL Detailed information on all metrics collected by Nsight Compute.

SOURC Source code of selected kernel in Summary page.

CONTEX (optional) shows CPU call stack at the time of kernel launch.

COMMENT Shows comments written in profiling file.

RAW Shows ALL metrics collected for the selected kernel and its data.

SUMMARY

ID	Estimated Speedup [%]	Function Name	Demangled Name	Duration [ms] (392.45 ms)	Runtime Improvement [ms] (0.00 ms)	Compute Throughput [%]	Memory Throughput [%]	# Registers [register/thread
0	0.00	**encoder_forward_...**	**encoder_forward_...**	0.01	0.00	17.27	18.67	16
1	0.00	layernorm_forward...	layernorm_forward...	0.81	0.00	0.08	1.03	40
2	0.00	matmul_forward_k...	matmul_forward_k...	5.05	0.00	10.27	84.74	38
3	0.00	permute_kernel	permute_kernel..co...	0.01	0.00	18.61	32.06	22
4	0.00	compute_attention...	compute_attention...	0.16	0.00	8.88	72.20	36
5	0.00	softmax_forward_k...	softmax_forward_k...	0.17	0.00	0.31	4.81	40
6	0.00	attention_weighted...	attention_weighted...	0.03	0.00	45.73	45.73	29
7	0.00	unpermute_kernel	unpermute_kernel..	0.01	0.00	24.24	14.34	21
8	0.00	matmul_forward_k...	matmul_forward_k...	1.99	0.00	8.69	71.54	38
9	0.00	residual_forward_k...	residual_forward_k...	0.01	0.00	8.09	44.30	16
10	0.00	layernorm_forward...	layernorm_forward...	0.83	0.00	0.08	1.00	40
11	0.00	matmul_forward_k...	matmul_forward_k...	7.00	0.00	9.86	81.38	38
12	0.00	gelu_forward_kernel	gelu_forward_kerne...	0.01	0.00	22.00	44.41	16
13	0.00	matmul_forward_k...	matmul_forward_k...	7.93	0.00	8.69	71.69	38
14	0.00	residual_forward_k...	residual_forward_k...	0.01	0.00	7.86	42.75	16
15	0.00	layernorm_forward...	layernorm_forward...	0.81	0.00	0.08	1.05	40
16	0.00	matmul_forward_k...	matmul_forward_k...	4.95	0.00	10.45	86.21	38
17	0.00	permute_kernel	permute_kernel..co...	0.01	0.00	18.78	32.23	22
18	0.00	compute_attention...	compute_attention...	0.16	0.00	8.87	72.18	36
19	0.00	softmax_forward_k...	softmax_forward_k...	0.17	0.00	0.31	4.83	40
20	0.00	attention_weighted...	attention_weighted...	0.03	0.00	45.22	45.22	29

BREAKDOWN

ID

BREAKDOWN

ESTIMATED SPEEDUP

Estimated Speedup [%]
0.00
0.00
0.00
0.00
0.00
0.00
0.00
0.00
0.00
0.00
0.00
0.00
0.00
0.00
0.00
0.00
0.00

BREAKDOWN

FUNCTION DEMANGLED NAME

Function Name	Demangled Name
encoder_forward_kernel	encoder_forward_kernel(float *, const int *, const float *, const float *, int, int, int)
layernorm_forward_kernel	layernorm_forward_kernel(float *, float *, float *, const float *, const float *, const float *, int, int, int)
matmul_forward_kernel	matmul_forward_kernel(float *, const float *, const float *, const float *, int, int)
permute_kernel	permute_kernel(float *, float *, float *, const float *, int, int, int, int)
compute_attention_scores_kernel	compute_attention_scores_kernel(float *, const float *, const float *, int, int, int, int)
softmax_forward_kernel	softmax_forward_kernel(float *, float, const float *, int, int)
attention_weighted_sum_kernel	attention_weighted_sum_kernel(float *, const float *, const float *, int, int, int, int)
unpermute_kernel	unpermute_kernel(float *, float *, int, int, int, int)
matmul_forward_kernel	matmul_forward_kernel(float *, const float *, const float *, const float *, int, int)
residual_forward_kernel	residual_forward_kernel(float *, float *, float *, int)
layernorm_forward_kernel	layernorm_forward_kernel(float *, float *, float *, const float *, const float *, const float *, int, int, int)
matmul_forward_kernel	matmul_forward_kernel(float *, const float *, const float *, const float *, int, int)

BREAKDOWN

Duration [ms] (392.45 ms)	Runtime Improvement [ms] (0.00 ms)
0.01	0.00
0.81	0.00
5.05	0.00
0.01	0.00
0.16	0.00
0.17	0.00
0.03	0.00
0.01	0.00
1.99	0.00
0.01	0.00
0.83	0.00
7.00	0.00
0.01	0.00
7.93	0.00
0.01	0.00
0.81	0.00
4.95	0.00
0.01	0.00
0.16	0.00

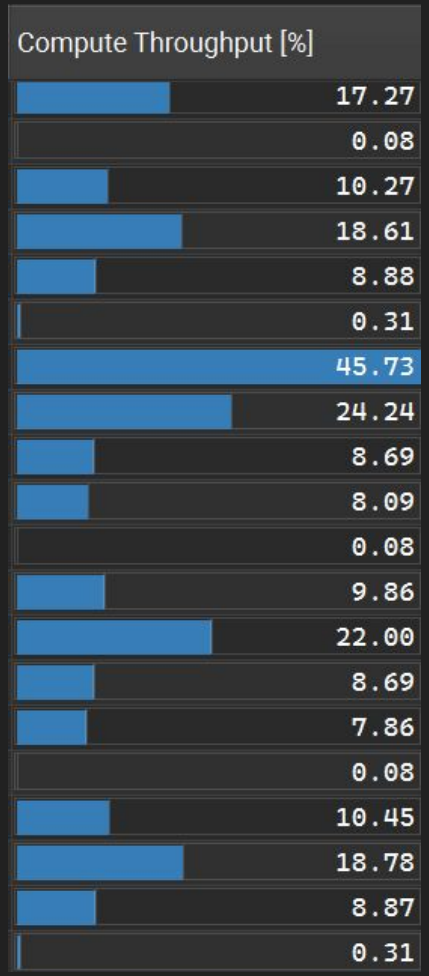
DURATION [MS] (TOTAL

RUNTIME IMPROVEMENT [MS] (TOTAL

TIME

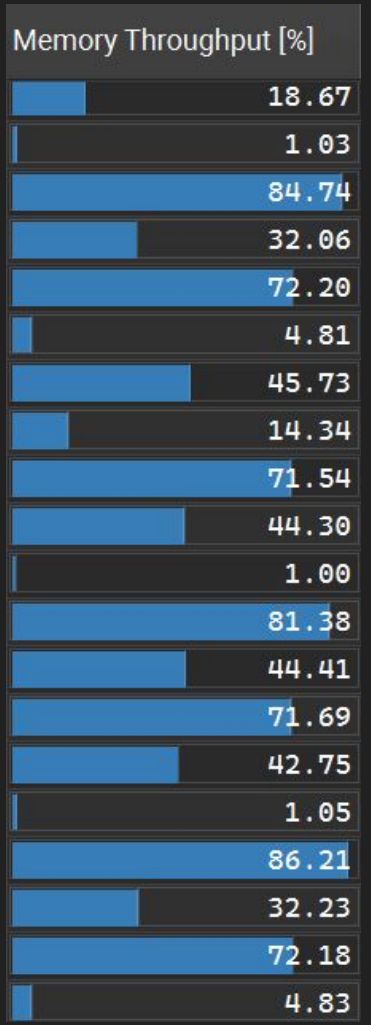
BREAKDOWN

COMPUTE THROUGHPUT



BREAKDOWN

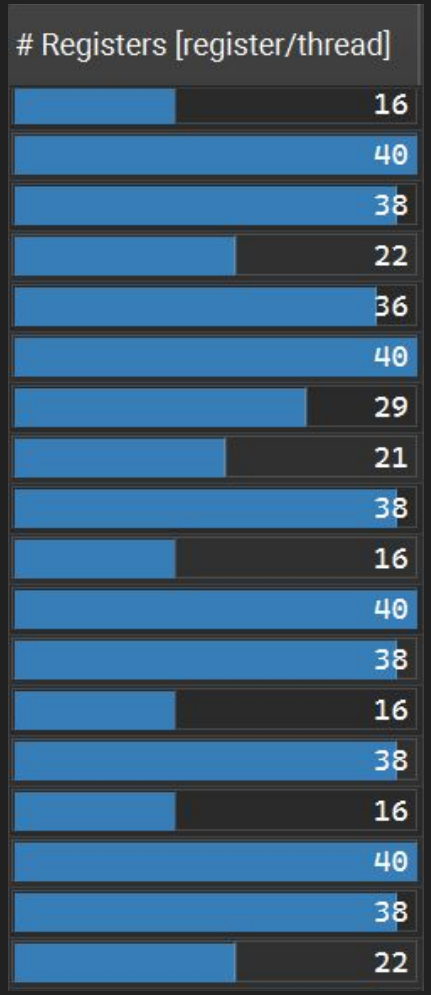
MEMORY THROUGHPUT [%]



BREAKDOWN

REGISTERS

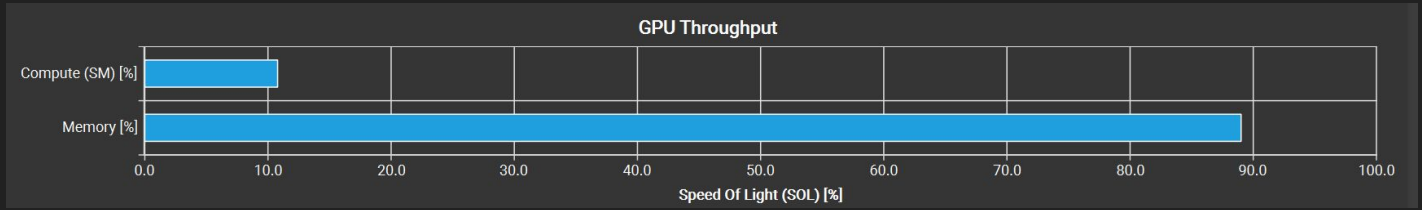
[REGISTER/THREAD]

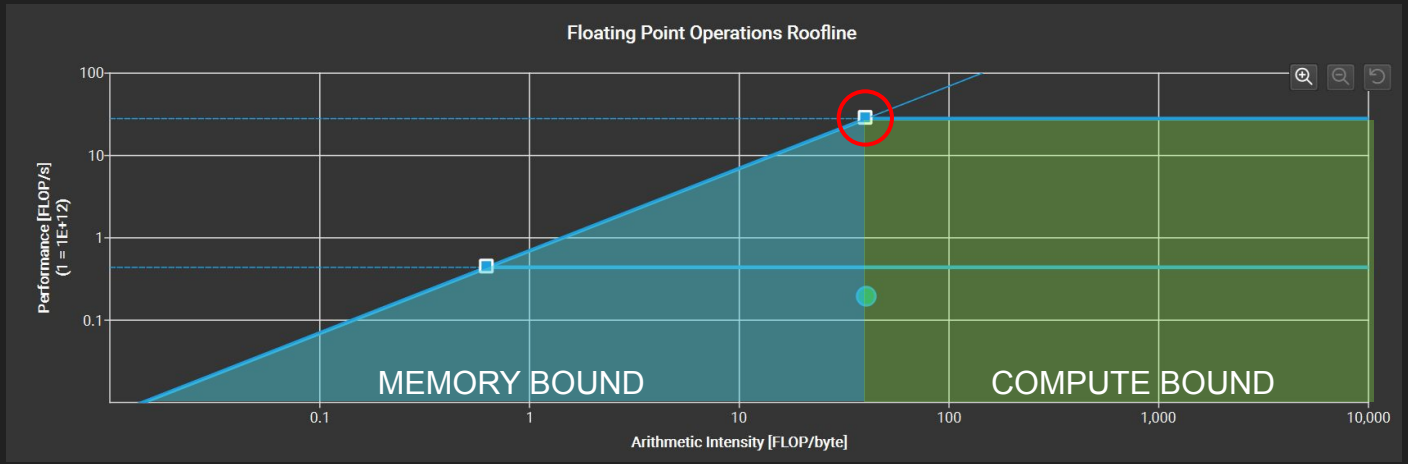


DETAILS OVERVIEW

Speed of Light

- High-level performance at a glance
- Achieved throughput relative to theoretical maximum

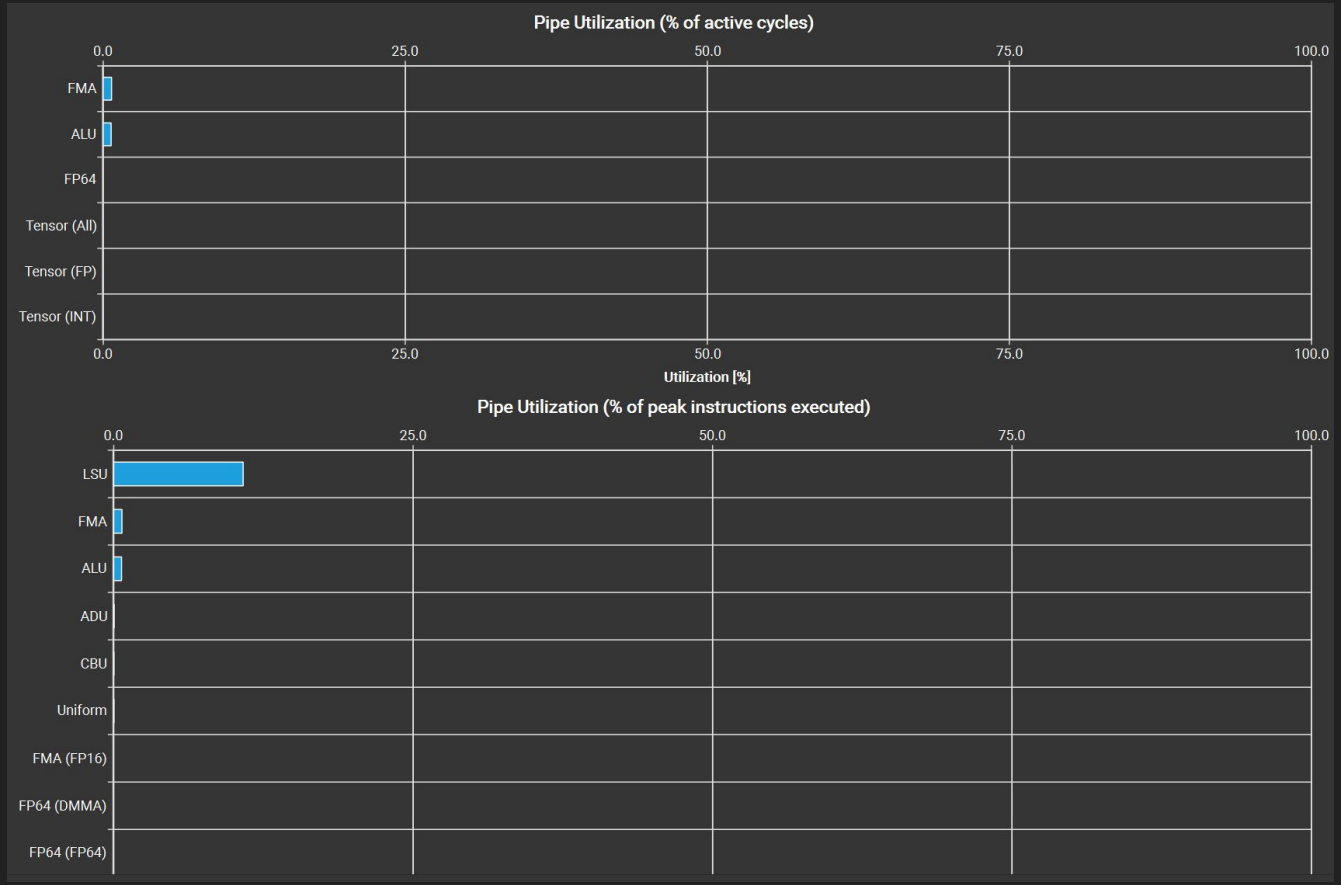




Arithmetic Intensity = FLOP/byte

Closer to Roofline is better!

Compute Workload Analysis



Pipe: Hardware that carries out specific instructions

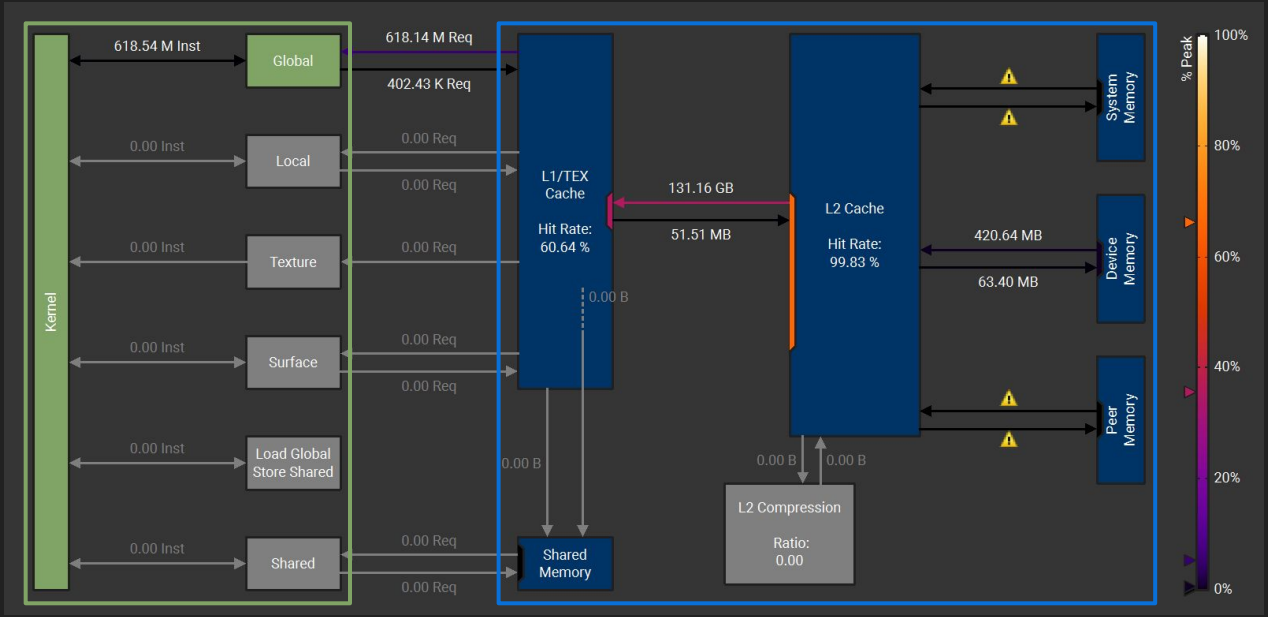
1. % of Cycles when the **SM** was **active**

Low = pipe is idle, even when SM is *active*
High = SMs are working when they're *active*

2. % Of theoretical peak ("Speed of light" of pipes)

Low = Memory stalls, branch divergence, etc.
High = Utilizing hardware as much as possible

Memory Workload Analysis



Port: Incoming and Outgoing traffic

Link: Instructions, Requests, or Data between Units

%Peak = Percentage of peak utilization of each Link/Port

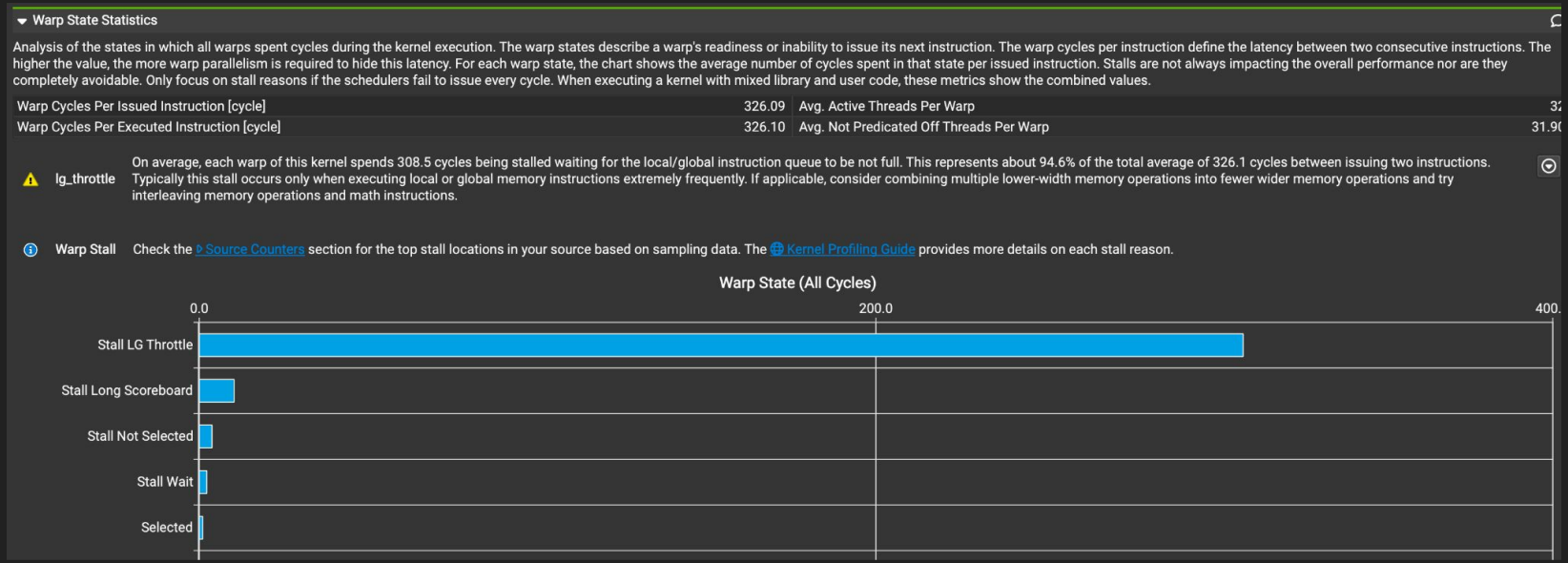
Logical Units

Physical Units

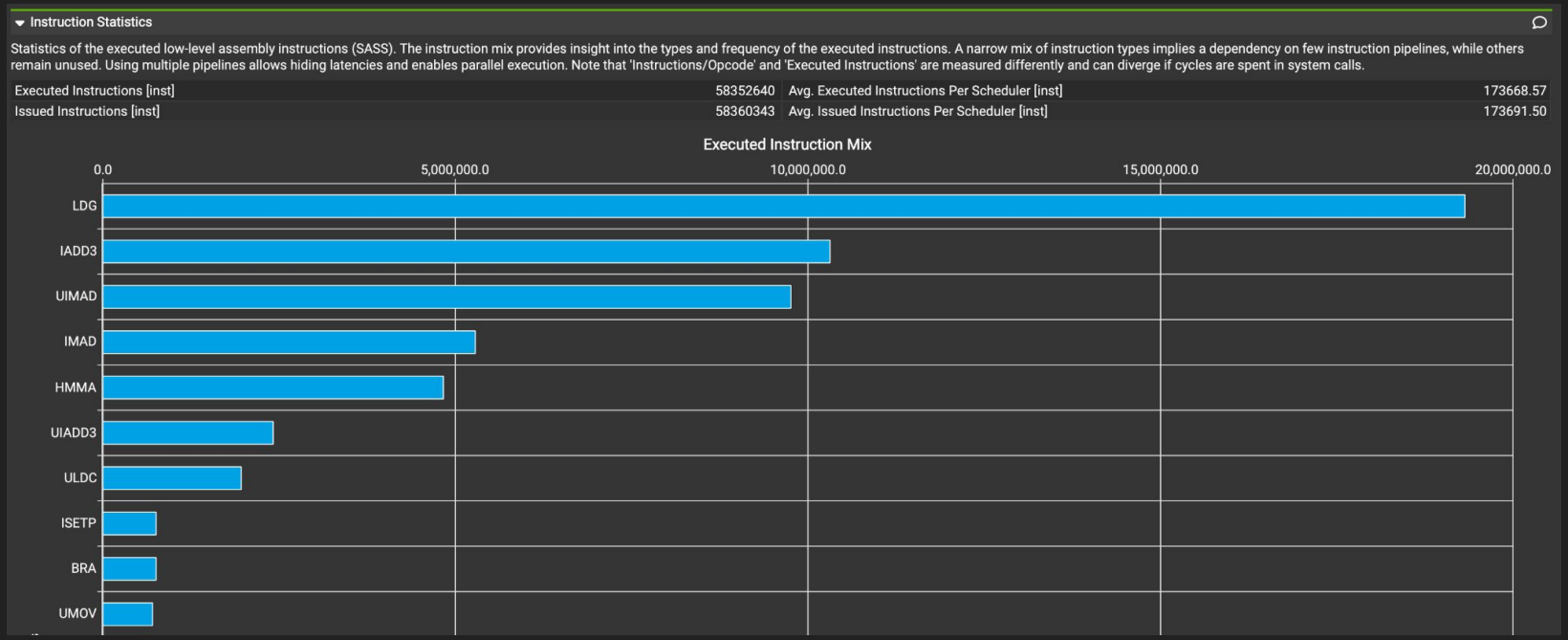
Almost all instructions go through L1 → L2 → DRAM

Warp State Statistics

- Warp cycle
- eligible/non-eligible warp



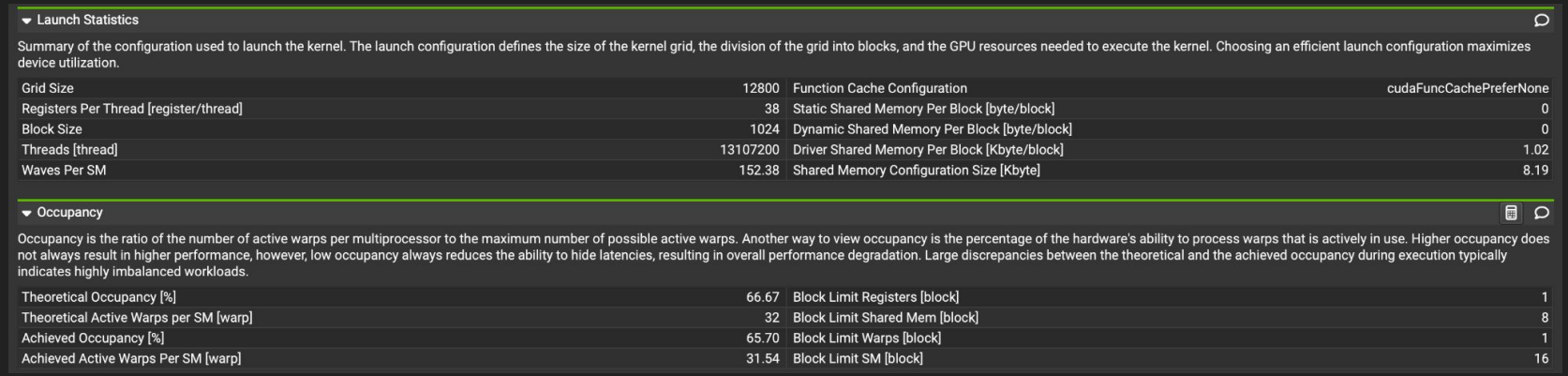
Instructions Statistics



The register file size is 64K 32-bit registers per SM.
Warps per SM 48
https://docs.nvidia.com/cuda/ampere-tuning-guide/index.html

Launch Statistics & Occupancy

Actual warps active per SM / Max warps active per SM



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	12800	Function Cache Configuration	cudaFuncCachePreferNone
Registers Per Thread [register/thread]	38	Static Shared Memory Per Block [byte/block]	0
Block Size	1024	Dynamic Shared Memory Per Block [byte/block]	0
Threads [thread]	13107200	Driver Shared Memory Per Block [Kbyte/block]	1.02
Waves Per SM	152.38	Shared Memory Configuration Size [Kbyte]	8.19

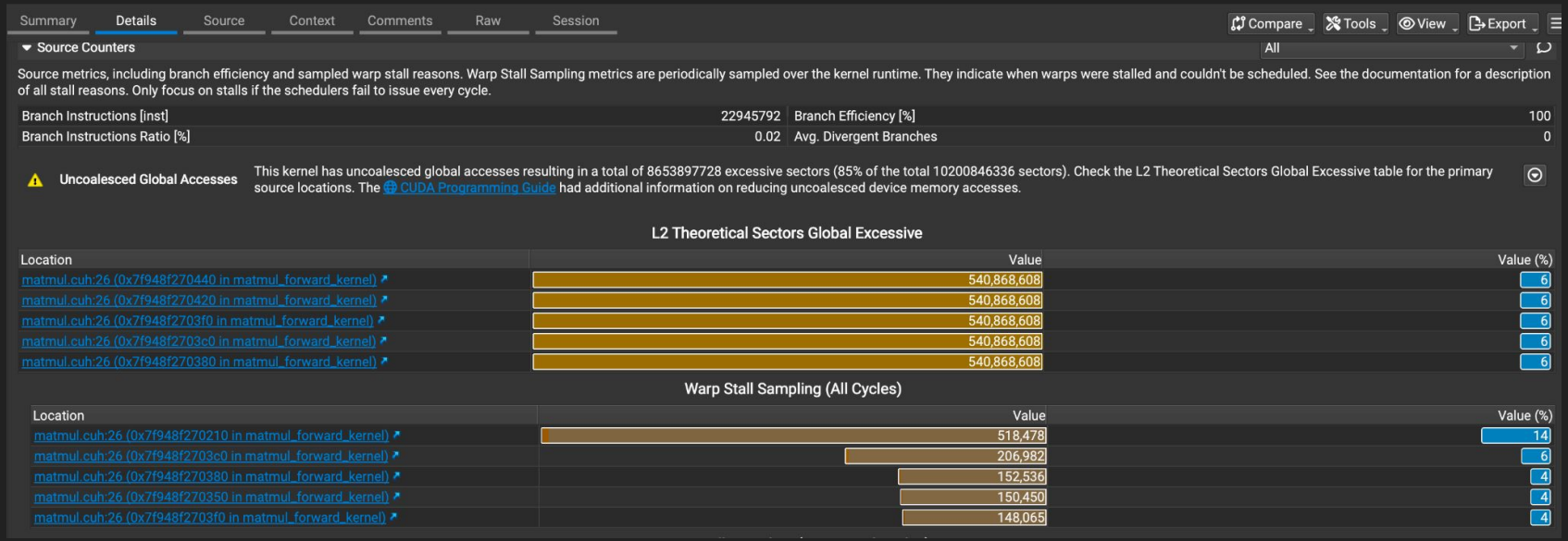
Occupancy

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 [%]	66.67	Block Limit Registers [block]	1
Theoretical Active Warps per SM [warp]	32	Block Limit Shared Mem [block]	8
Achieved Occupancy [%]	65.70	Block Limit Warps [block]	1
Achieved Active Warps Per SM [warp]	31.54	Block Limit SM [block]	16

Block Limit Registers: total registers per SM / (registers per thread×threads per block)
Block Limit Shared Mem: Shared mem per SM / Shared mem per block
Block Limit Warps: Max warps per SM / Warps per block
Block Limit SM: hard architecture limit 16 blocks per SM

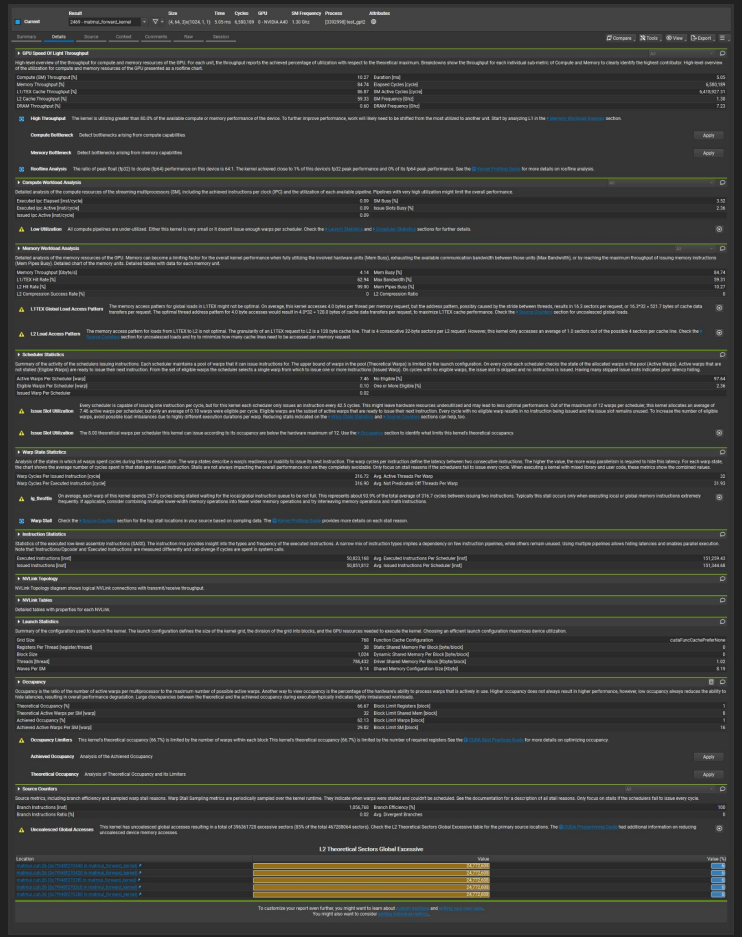
Source Counters



EXAMPLE

Simple example: Matmul

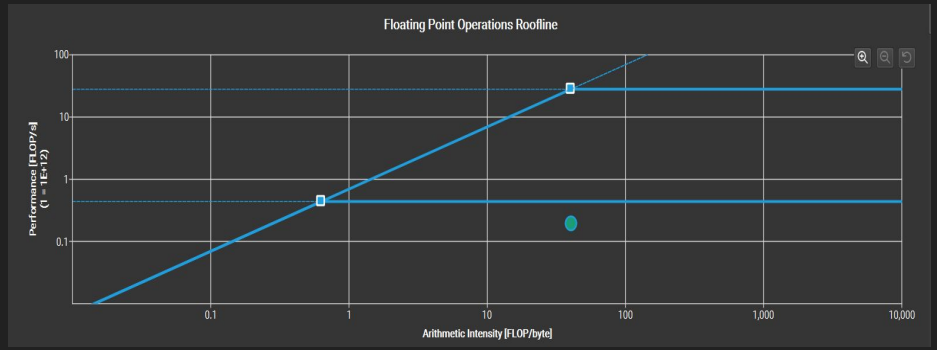
- (4,64,3) * (1024,1,1)
- Batched matrix multiplication kernel for GPT-2's forward propagation
- It performs linear transformation for each token embedding.
- The whole ncu report looks like this
- It's a lot, but no need to panic



Lecture 4 Slide 11 figures

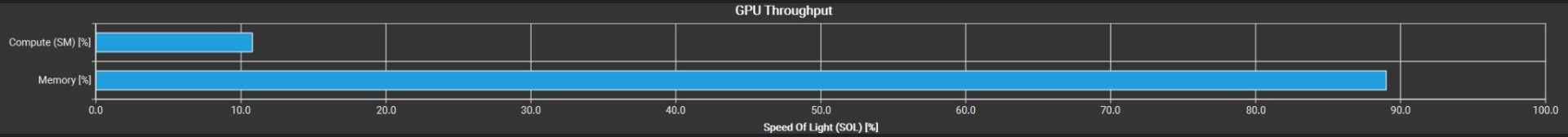
Why is that kernel “BAD”

- **Memory Bottleneck**
- Very poor memory coalescing
- Global memory inefficiency/overuse
- Abysmally low compute throughput
- Warps stall constantly



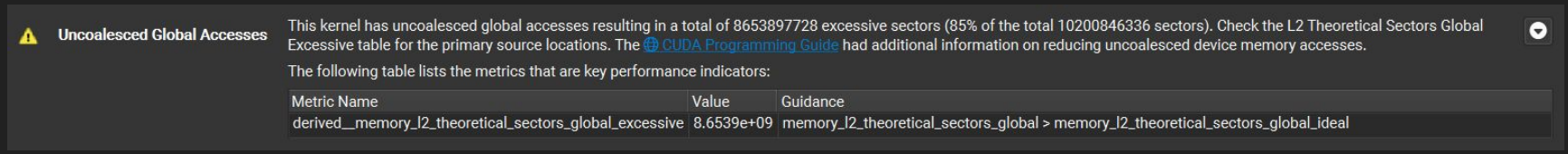
High-level overview of the throughput for compute and memory resources of the GPU. For each unit, the throughput reports the achieved percentage of utilization with respe
level overview of the utilization for compute and memory resources of the GPU presented as a roofline chart.

Compute (SM) Throughput [%]	10.79
Memory Throughput [%]	89.01
L1/TEX Cache Throughput [%]	89.19
L2 Cache Throughput [%]	66.30
DRAM Throughput [%]	0.66



Why is that kernel “BAD”

- Memory Bottleneck
- **Very poor memory coalescing**
- Global memory inefficiency/overuse
- Abysmally low compute throughput
- Warps stall constantly



Uncoalesced Global Accesses

This kernel has uncoalesced global accesses resulting in a total of 8653897728 excessive sectors (85% of the total 10200846336 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.

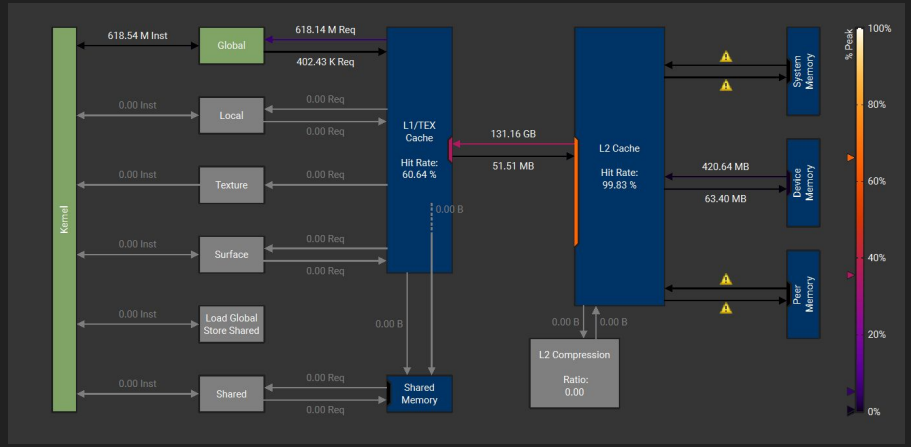
The following table lists the metrics that are key performance indicators:

Metric Name	Value	Guidance
derived__memory_l2_theoretical_sectors_global_excessive	8.6539e+09	memory_l2_theoretical_sectors_global > memory_l2_theoretical_sectors_global_ideal

`derived__memory_l2_theoretical_sectors_global_excessive`	Excessive theoretical number of sectors requested in L2 from global memory instructions, because not all not predicated-off threads performed the operation.

Why is that kernel “BAD”

- Memory Bottleneck
- Very poor memory coalescing
- **Global memory inefficiency/overuse**
- Abysmally low compute throughput
- Warps stall constantly



Memory Workload Analysis

Detailed analysis of the memory resources of the GPU. Memory can become a limiting factor for the overall kernel pe
bandwidth between those units (Max Bandwidth), or by reaching the maximum throughput of issuing memory instruc

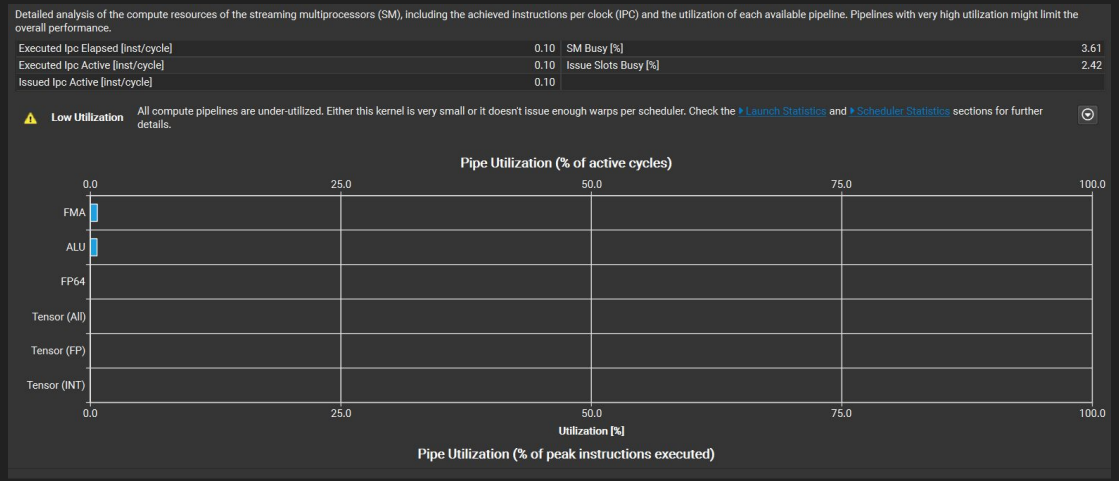
Memory Throughput [Gbyte/s]	4.62
L1/TEX Hit Rate [%]	60.64
L2 Hit Rate [%]	99.83
L2 Compression Success Rate [%]	0

SPECIFICATIONS

GPU architecture	**NVIDIA Ampere architecture**
GPU memory	**48 GB GDDR6 with ECC**
Memory bandwidth	**696 GB/s**
Interconnect interface	**NVIDIA® NVLink® 112.5 GB/s (bidirectional)³ PCIe Gen4: 64GB/s**
NVIDIA Ampere architecture-based CUDA Cores	**10,752**
NVIDIA second-generation RT Cores	**84**
NVIDIA third-generation Tensor Cores	**336**
Peak FP32 TFLOPS (non-Tensor)	**37.4**

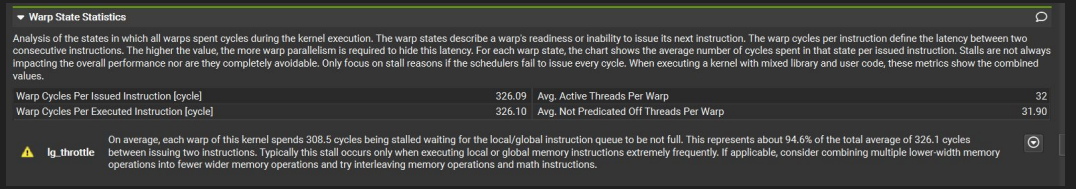
Why is that kernel “BAD”

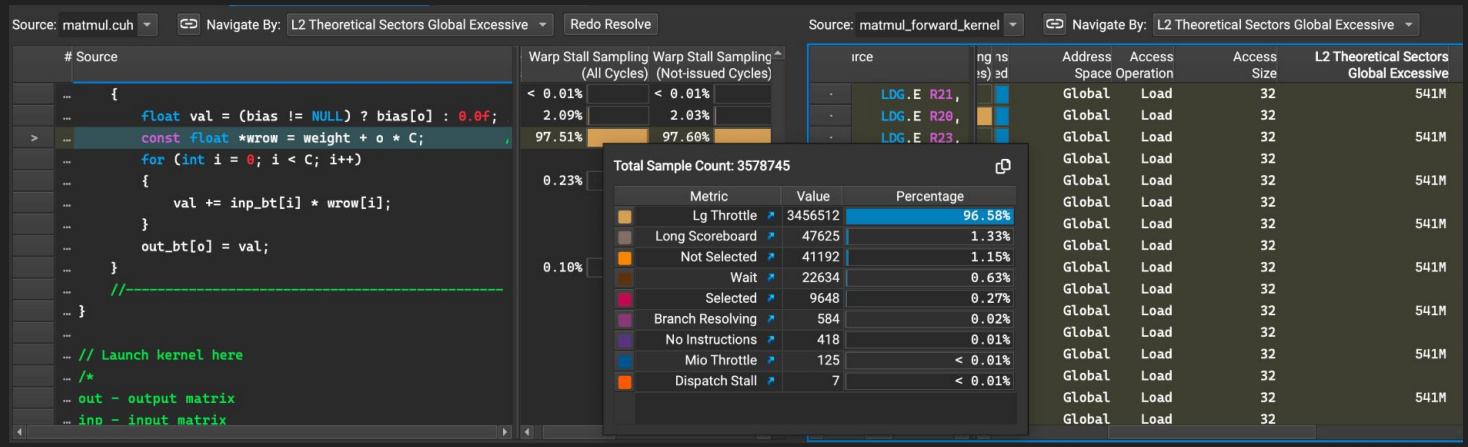
- Memory Bottleneck
- Very poor memory coalescing
- Global memory inefficiency/overuse
- **Abysmally low compute throughput**
- Warps stall constantly



Why is that kernel “BAD”

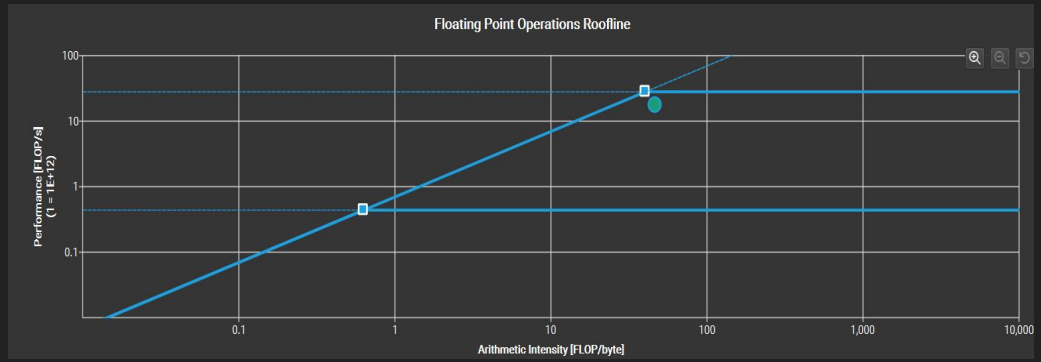
- Memory Bottleneck
- Very poor memory coalescing
- Global memory inefficiency/overuse
- Abysmally low compute throughput
- **Warps stall constantly**

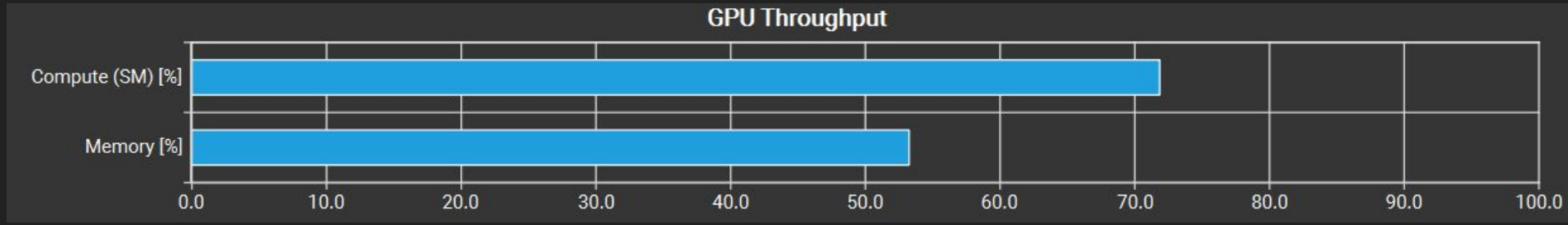




Improvement: cuBLAS optimization

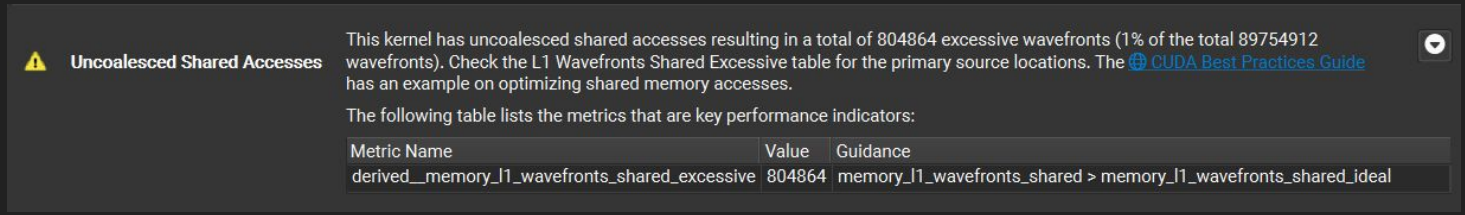
- Faster: 104.77ms -> 2.29ms
- **No More Memory Bottleneck**
- Improved memory coalescing
- Shared Memory Utilization
- Satisfying compute throughput
- Warps issue more instructions





Improvement: cuBLAS optimization

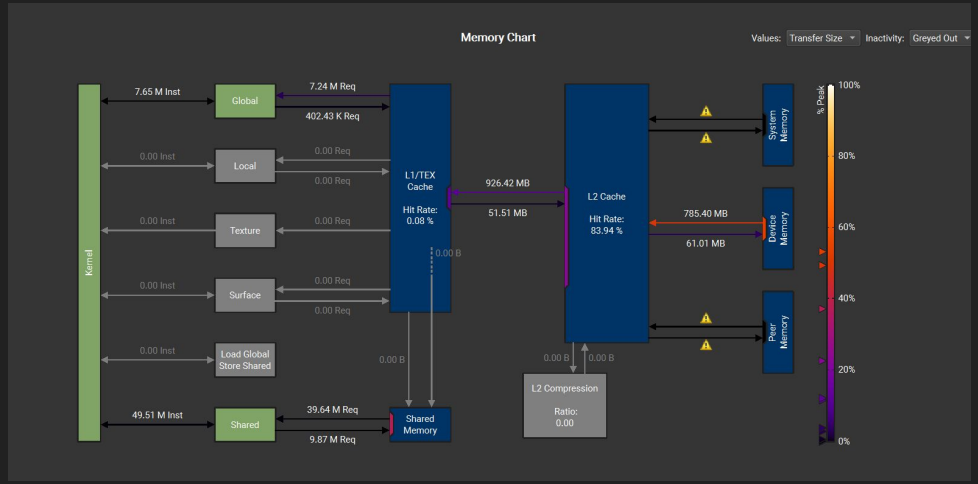
- Faster: 104.77ms -> 2.29ms
- No More Memory Bottleneck
- **Improved memory coalescing**
- Shared Memory Utilization
- Satisfying compute throughput
- Warps issue more instructions



Still minor issues, but overall much better

Improvement: cuBLAS optimization

- Faster: 104.77ms -> 2.29ms
- No More Memory Bottleneck
- Improved memory coalescing
- **Shared Memory Utilization**
- Satisfying compute throughput
- Warps issue more instructions



Improvement: cuBLAS optimization

- Faster: 104.77ms -> 2.29ms
- No More Memory Bottleneck
- Improved memory coalescing
- Shared Memory Utilization
- **Satisfying compute throughput**
- Warps issue more instructions

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]	0.10	SM Busy [%]	3.61
Executed Ipc Active [inst/cycle]	0.10	Issue Slots Busy [%]	2.42
Issued Ipc Active [inst/cycle]	0.10		

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]	2.87	SM Busy [%]	73.02
Executed Ipc Active [inst/cycle]	2.92	Issue Slots Busy [%]	73.02
Issued Ipc Active [inst/cycle]	2.92		

Improvement: cuBLAS optimization

- Faster: 104.77ms -> 2.29ms
- No More Memory Bottleneck
- Improved memory coalescing
- Shared Memory Utilization
- Satisfying compute throughput
- **Warps issue more instructions**

Overall: 18,780 (7.80597%)

Contributors	Value	Contribution (%)
smsp__pcsamp_warps_issue_stalled_barrier	13,337	71.02%
smsp__pcsamp_warps_issue_stalled_not_selected	3,912	20.83%
smsp__pcsamp_warps_issue_stalled_dispatch_stall	1,424	7.58%
smsp__pcsamp_warps_issue_stalled_selected	69	0.37%
smsp__pcsamp_warps_issue_stalled_mio_throttle	38	0.20%
smsp__pcsamp_warps_issue_stalled_branch_resolving	0	0%
smsp__pcsamp_warps_issue_stalled_drain	0	0%
smsp__pcsamp_warps_issue_stalled_imc_miss	0	0%
smsp__pcsamp_warps_issue_stalled_lg_throttle	0	0%
smsp__pcsamp_warps_issue_stalled_long_scoreboard	0	0%
smsp__pcsamp_warps_issue_stalled_math_pipe_throttle	0	0%
smsp__pcsamp_warps_issue_stalled_membar	0	0%
smsp__pcsamp_warps_issue_stalled_misc	0	0%
smsp__pcsamp_warps_issue_stalled_no_instructions	0	0%
smsp__pcsamp_warps_issue_stalled_short_scoreboard	0	0%
smsp__pcsamp_warps_issue_stalled_sleeping	0	0%
smsp__pcsamp_warps_issue_stalled_tex_throttle	0	0%
smsp__pcsamp_warps_issue_stalled_wait	0	0%

Warp Cycles Per Issued Instruction [cycle]	326.09	Avg. Active Threads Per Warp	32
Warp Cycles Per Executed Instruction [cycle]	326.10	Avg. Not Predicated Off Threads Per Warp	31.90

Warp Cycles Per Issued Instruction [cycle]	5.36	Avg. Active Threads Per Warp	32
Warp Cycles Per Executed Instruction [cycle]	5.36	Avg. Not Predicated Off Threads Per Warp	31.76

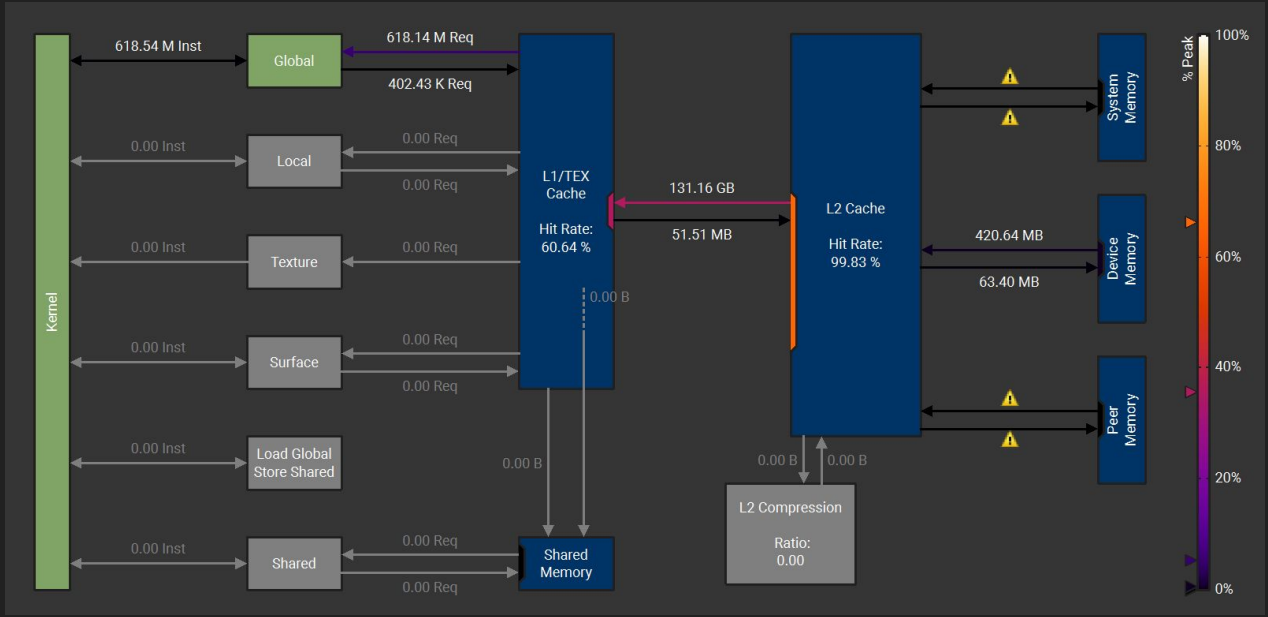
How does Nsight Compute do this?

Under non-instrumented mode of NCU:

1. Hardware performance monitoring unit
   - Tracks metrics like number and types of instructions executed, cache hits/misses, memory transactions, etc.
2. Sampling + Counters
   - Samples statistics like warps stalling, pipeline utilization
   - Counts number and types of instructions executed, number of branched instructions

PERFORMANCE METRICS DEEP DIVE

Memory Workload - Baseline Mat. Mul. (No Tiling)



618M Total Instructions

<u>L1 Cache Under-utilized</u>
40% (250M) Miss L1
L2 Reads cost **130 GB!**
~200 cycles per L2 load*

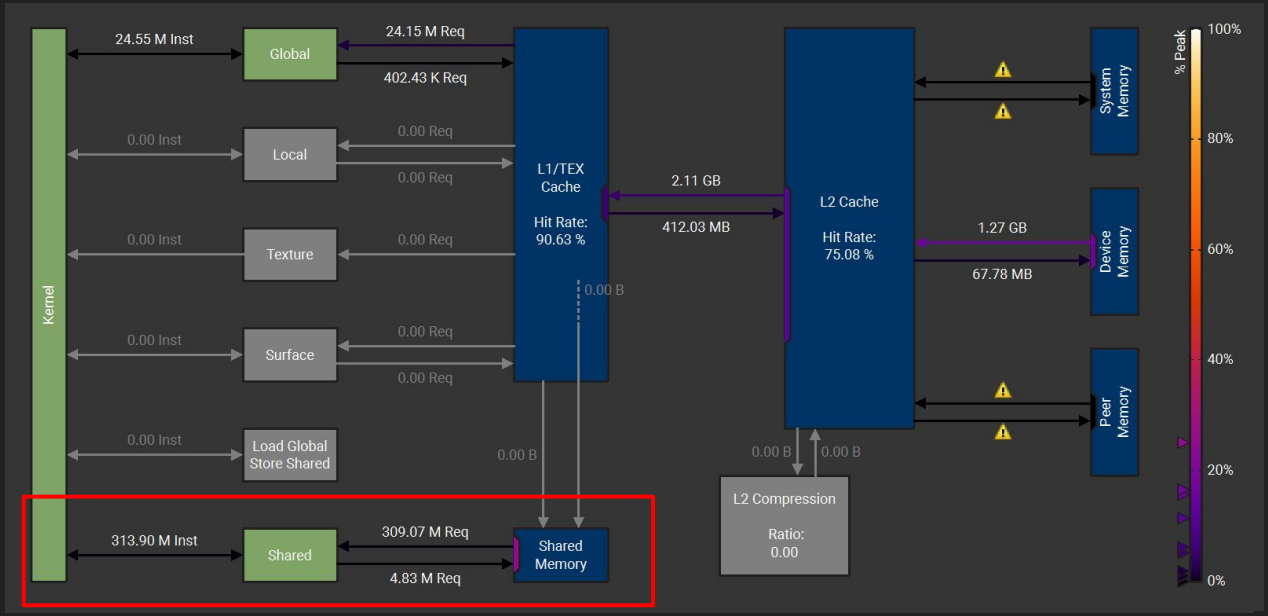
<u>Cache bound</u>
Matrices too big for cache

Repeatedly gets streamed from DRAM → L2 → L1

For Reference:
Input Matrix: (4 x 64) x 768 = 196,608 total elements
Weight Matrix: 50304 x 768 = 38,633,472 total elements
Total Input Size: ~155 MB
Total Available L2 Cache: 6 MB

*based on A100 GPU

Memory Workload - Register & Shared Memory Tiling



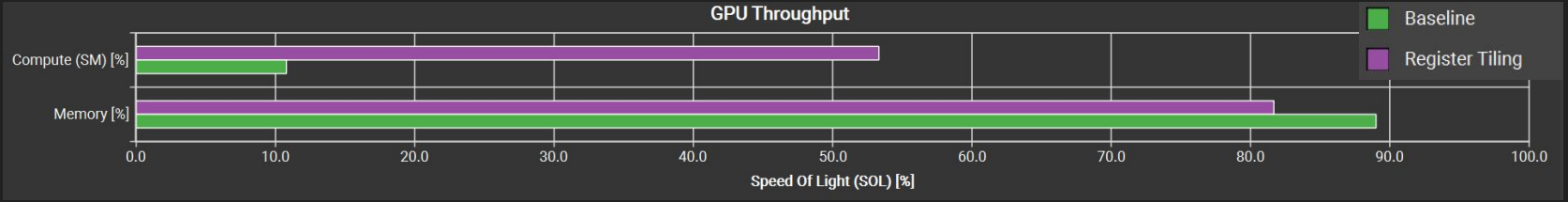
24M Global (-96%)
+ 313M Shared
= 337M (-45%) Total Instr.

<u>L2 Cache Not Stressed</u>
-96% less Global reads
90% L1 Hit Rate
→ L2 bandwidth freed up

T = 64: 64x Reuse with shared memory:
309.9 load /4.8 store = 64!

For Reference:
Input Matrix: (4 x 64) x 768 = 196,608 total elements
Weight Matrix: 50304 x 768 = 38,633,472 total elements
Total Input Size: ~155 MB
Total Available L2 Cache: 6 MB

Memory Workload - Register & Shared Memory Tiling



Results

- Compute: 10.8% → 53.3%
- Memory: 89.6% → 81%
- Runtime: 104.7ms → 11.58ms

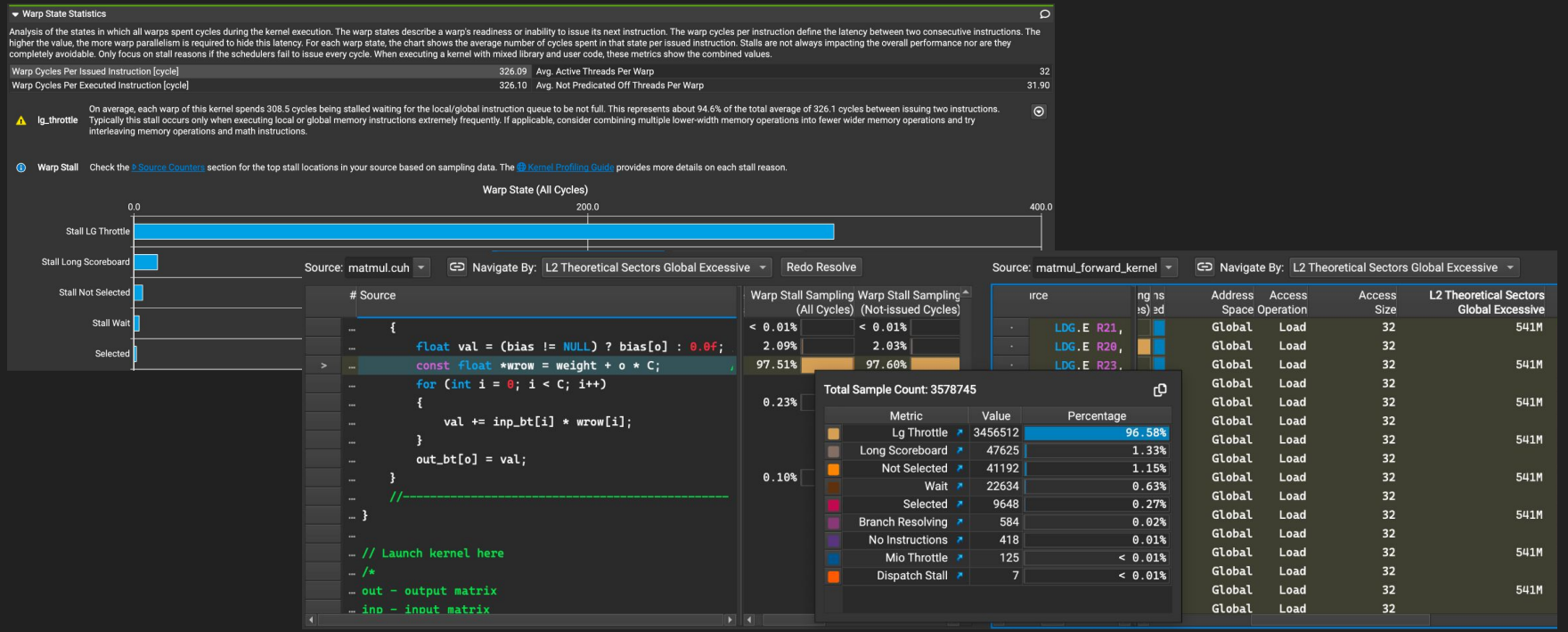
Profiling Summary

- Less cycles stalled for loading memory → increased FLOPS
  - Less Global instructions and lower L2 bandwidth
- Use of on-chip registers and shared memory → lower memory latency
  - ~5-10 cycles v.s. ~200

No Eligible 97.58%
One or More Eligible 2.42%

Baseline warp stall

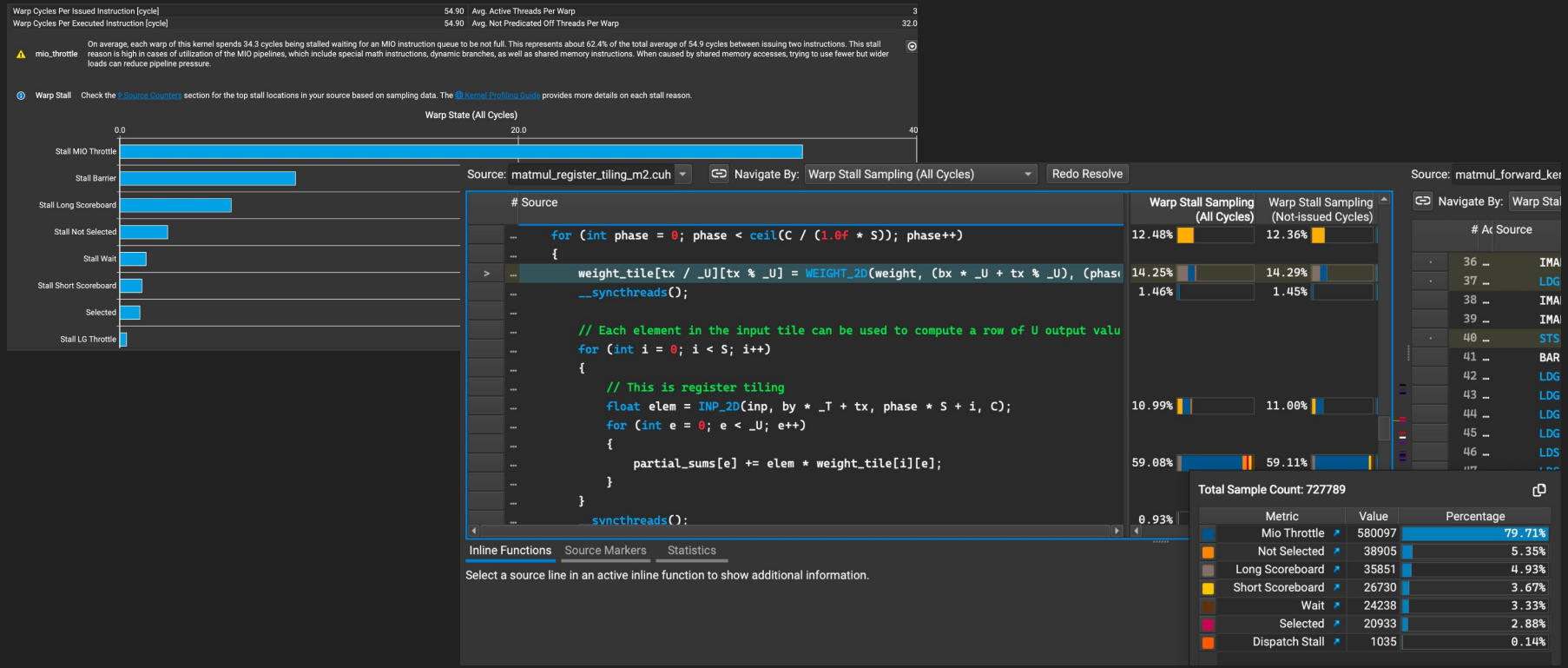
Lg Throttle – executing local or global memory instructions extremely frequently



No Eligible 85.97%
One or More Eligible 14.03%

Register tiling warp stall

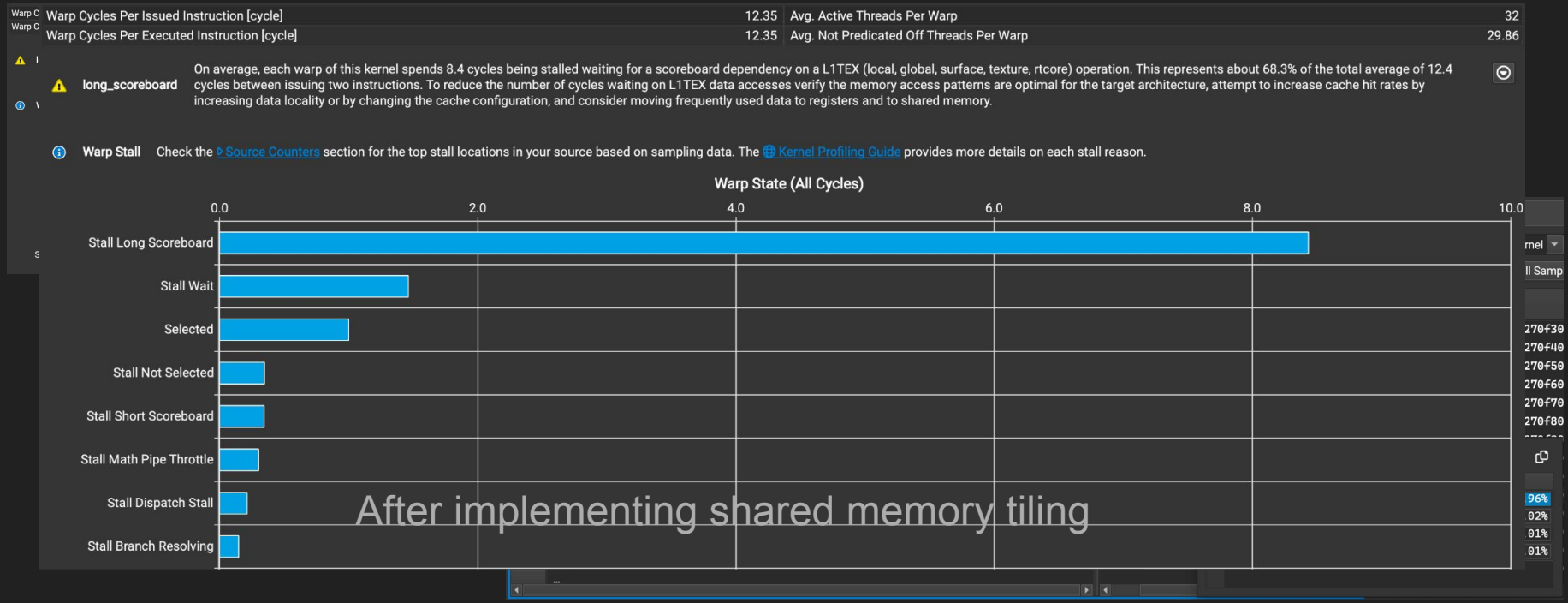
Mio_throttle – special math instructions, dynamic branches, shared memory instructions



No Eligible 87.68%
One or More Eligible 12.32%

Tensor core warp stall

Long scoreboard – waiting on memory operations from L1TEX (global, local, or surface memory).



No Eligible 26.93%
One or More Eligible 73.07%

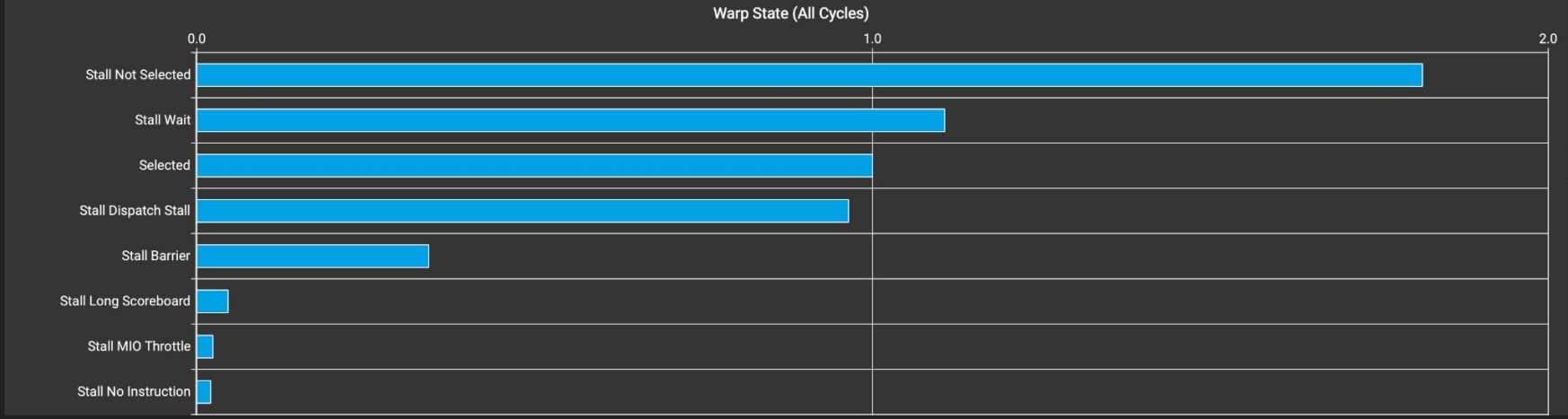
cuBLAS warp stall

Not selected: eligible warps that were not picked by the scheduler to issue that cycle as another warp was selected

Warp Cycles Per Issued Instruction [cycle]	5.36	Avg. Active Threads Per Warp	32
Warp Cycles Per Executed Instruction [cycle]	5.36	Avg. Not Predicated Off Threads Per Warp	31.76

not_selected On average, each warp of this kernel spends 1.8 cycles being stalled due to not being selected by the scheduler. This represents about 33.9% of the total average of 5.4 cycles between issuing two instructions. Not selected warps are eligible warps that were not picked by the scheduler to issue that cycle as another warp was selected. A high number of not selected warps typically means you have sufficient warps to cover warp latencies and you may consider reducing the number of active warps to possibly increase cache coherence and data locality.

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.



The register file size is 64K 32-bit registers per SM.
Warps per SM 48
https://docs.nvidia.com/cuda/ampere-tuning-guide/index.html

6. Launch Statistics & Occupancy

Actual warps active per SM / Max warps active per SM

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	12800	Function Cache Configuration	cudaFuncCachePreferNone
Registers Per Thread [register/thread]	38	Static Shared Memory Per Block [byte/block]	0
Block Size	1024	Dynamic Shared Memory Per Block [byte/block]	0
Threads [thread]	13107200	Driver Shared Memory Per Block [Kbyte/block]	1.02
Waves Per SM	152.38	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 [%]	66.67	Block Limit Registers [block]	1
Theoretical Active Warps per SM [warp]	32	Block Limit Shared Mem [block]	8
Achieved Occupancy [%]	65.70	Block Limit Warps [block]	1
Achieved Active Warps Per SM [warp]	31.54	Block Limit SM [block]	16

Block Limit Registers: total registers per SM / (registers per thread×threads per block)
Block Limit Shared Mem: Shared mem per SM / Shared mem per block
Block Limit Warps: Max warps per SM / Warps per block
Block Limit SM: hard architecture limit 16 blocks per SM

6. Launch Statistics & Occupancy

The register file size is 64K 32-bit registers per SM.
Warps per SM 48
https://docs.nvidia.com/cuda/ampere-tuning-guide/index.html