DEEP DIVE INTO NSIGHT SYSTEMS & NSIGHT COMPUTE

Bing Liu, 202012

AGENDA

Overview of Profilers

Nsight Systems

Nsight Compute

Case Studies

Summary

OVERVIEW OF PROFILERS

NVVP Visual Profiler

nvprof the command-line profiler

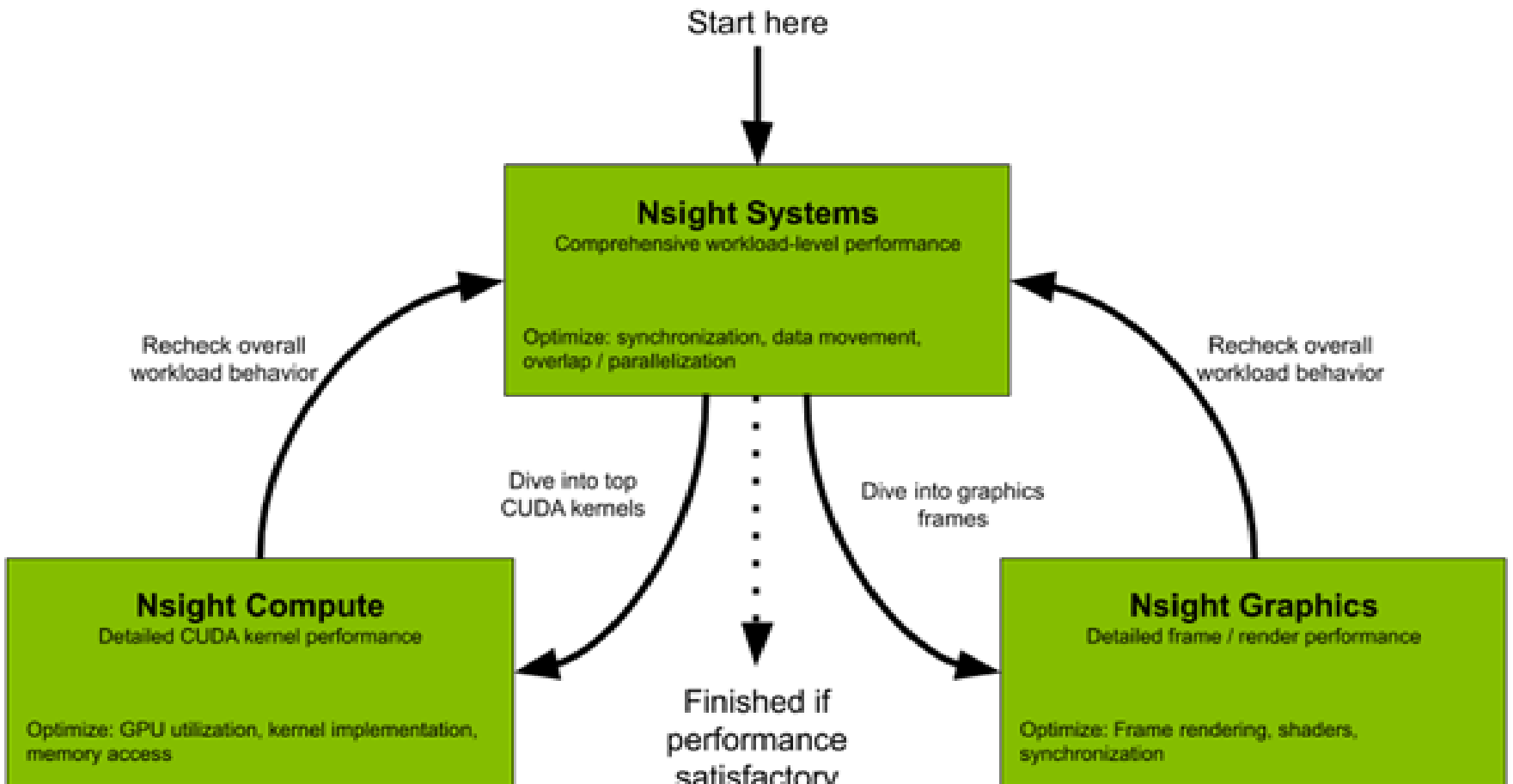
Nsight Systems A system-wide performance analysis tool

Nsight Compute An interactive kernel profiler for CUDA applications

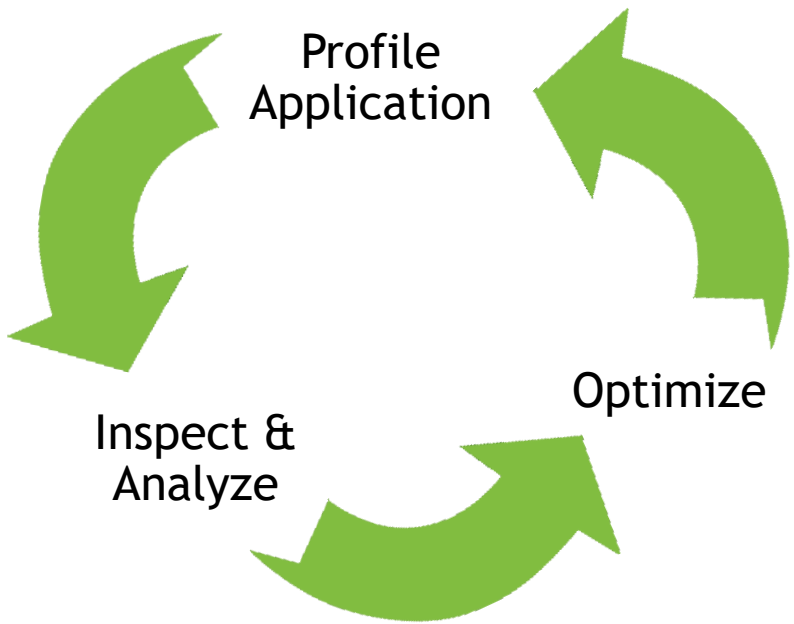
Note that Visual Profiler and nvprof will be **deprecated** in a future CUDA release

We strongly recommend you transfer to Nsight Systems and Nsight Compute

NSIGHT PRODUCT FAMILY



OVERVIEW OF OPTIMIZATION WORKFLOW



Iterative process continues until desired performance is achieved

NSIGHT SYSTEMS

Overview

System-wide application algorithm tuning

- Focus on the application's algorithm – a unique perspective

Locate optimization opportunities

- See gaps of unused CPU and GPU time

Balance your workload across multiple CPUs and GPUs

- CPU algorithms, utilization, and thread state
- GPU streams, kernels, memory transfers, etc

Support for Linux & Windows, x86-64 & Tegra. Host only for Mac

NSIGHT SYSTEMS

Key Features

Compute

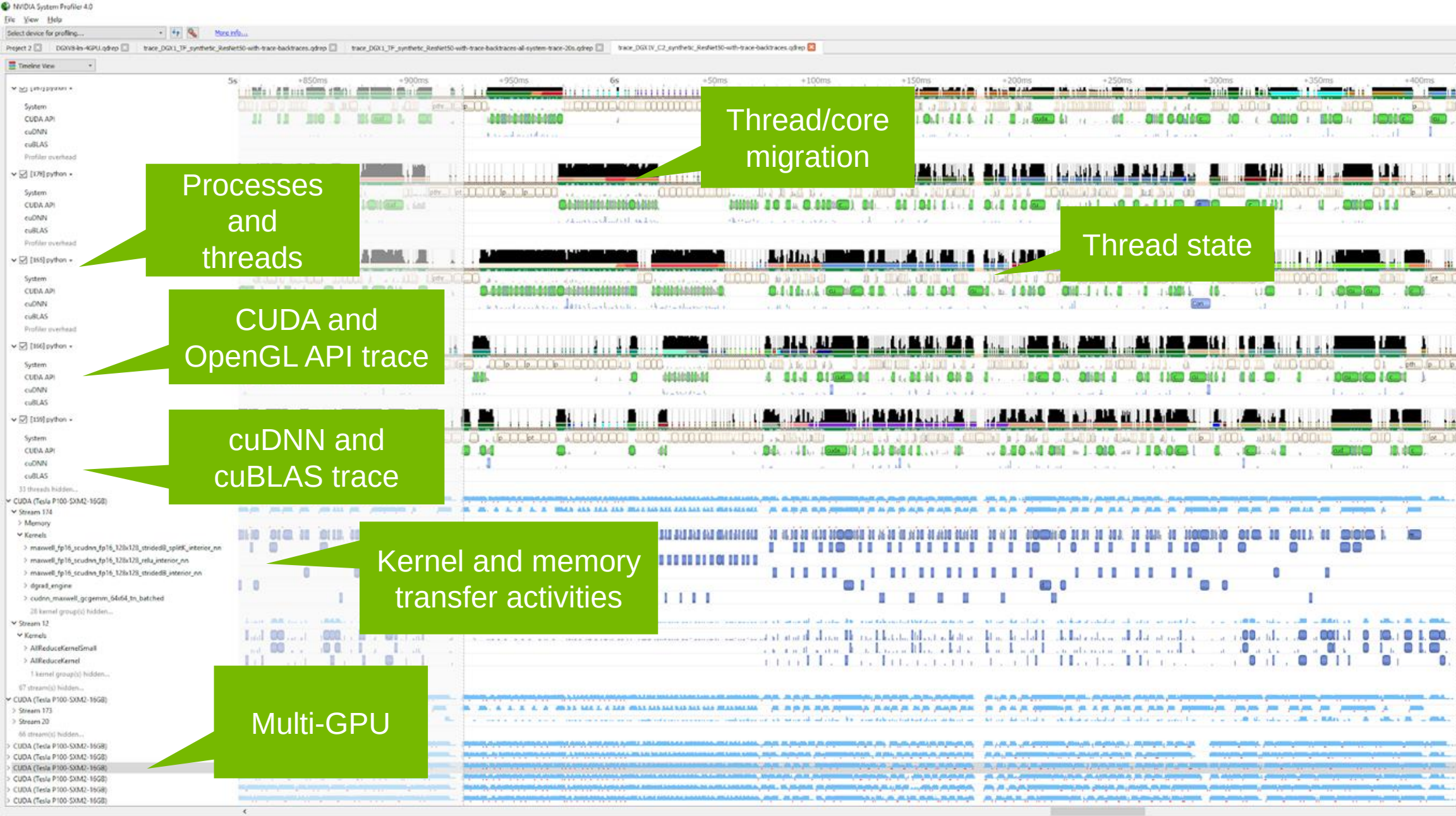
- CUDA API. Kernel launch and execution correlation
- Libraries: cuBLAS, cuDNN, TensorRT
- OpenACC

Graphics

- Vulkan, OpenGL, DX11, DX12, DXR, V-sync

OS Thread state and CPU utilization, pthread, file I/O, etc.

User annotations API (NVTX)

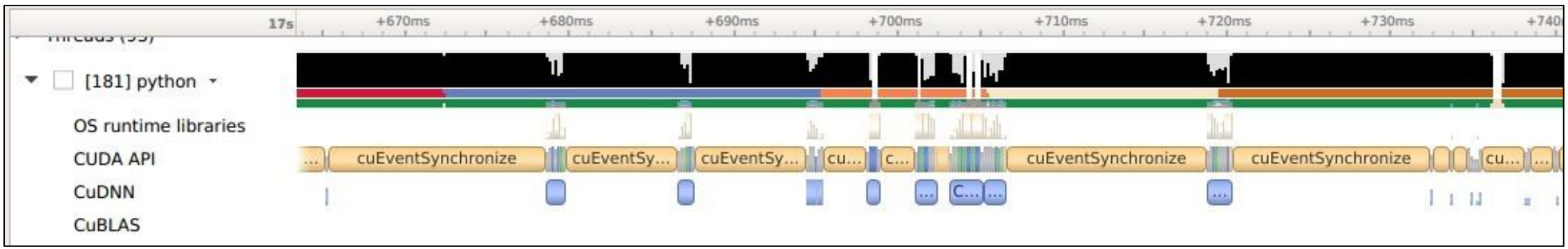

Thread/core migration
Processes and threads
Thread state
CUDA and OpenGL API trace
cuDNN and cuBLAS trace
Kernel and memory transfer activities
Multi-GPU

CPU THREADS

Thread Activities

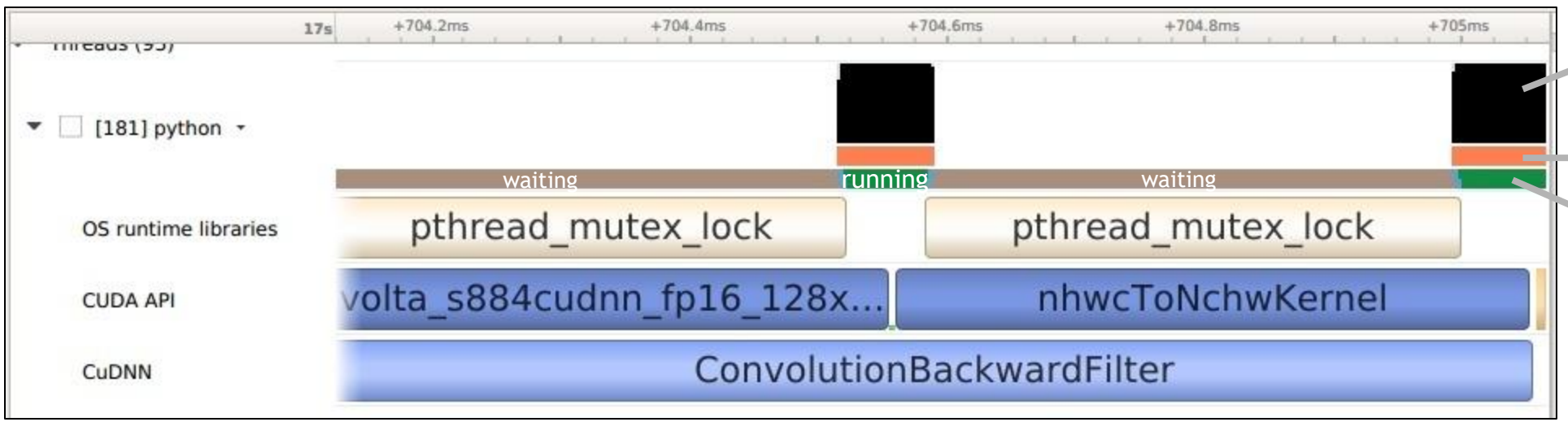
Get an overview of each thread's activities

- Which core the thread is running and the utilization
- CPU state and transition
- OS runtime libraries usage: pthread, file I/O, etc.
- API usage: CUDA, cuDNN, cuBLAS, TensorRT, …



CPU THREADS

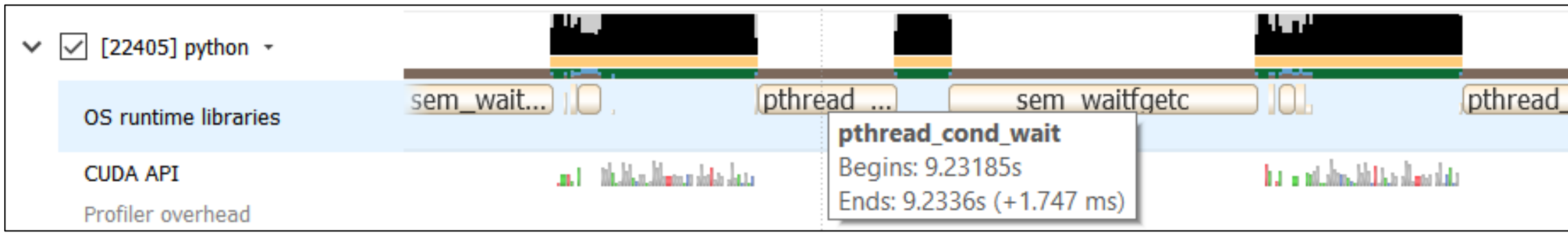
Thread Activities



OS RUNTIME LIBRARIES

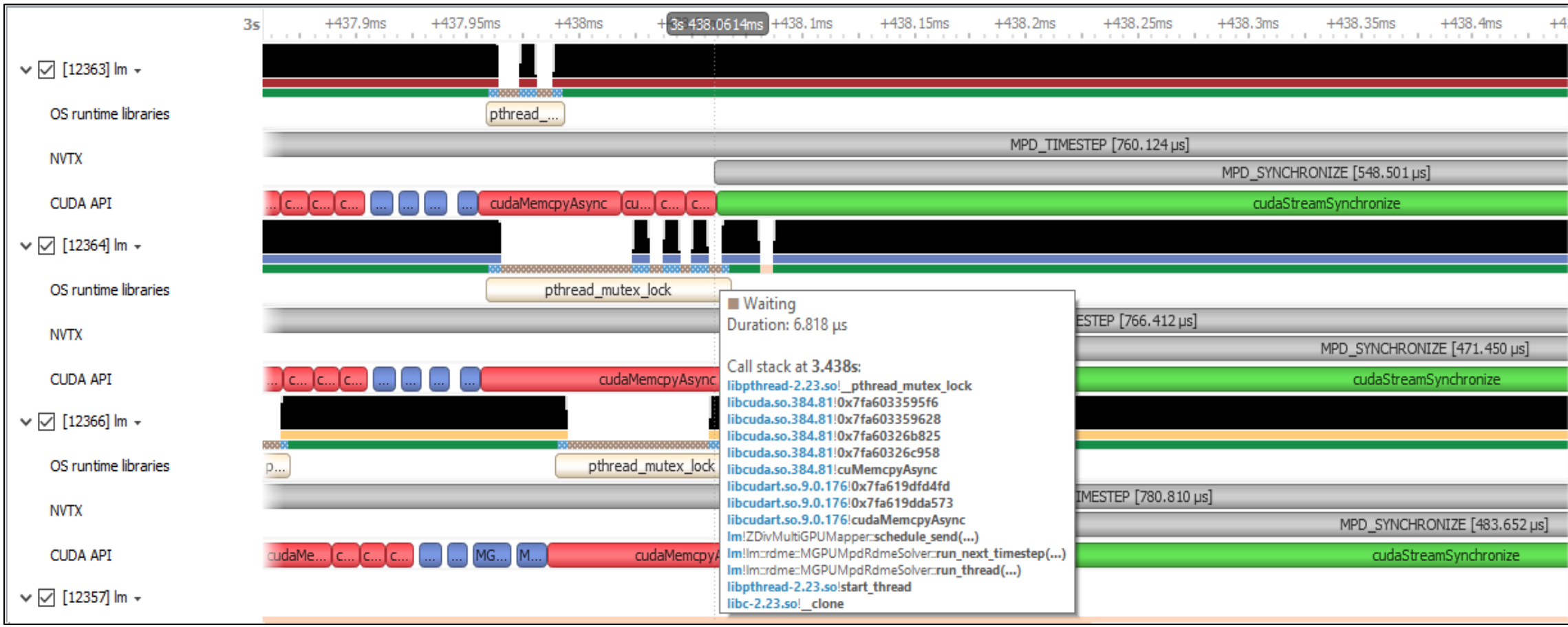
Identify time periods where threads are blocked and the reason

Locate potentially redundant synchronizations



OS RUNTIME LIBRARIES

Backtrace for time-consuming calls to OS runtime libs



CUDA API

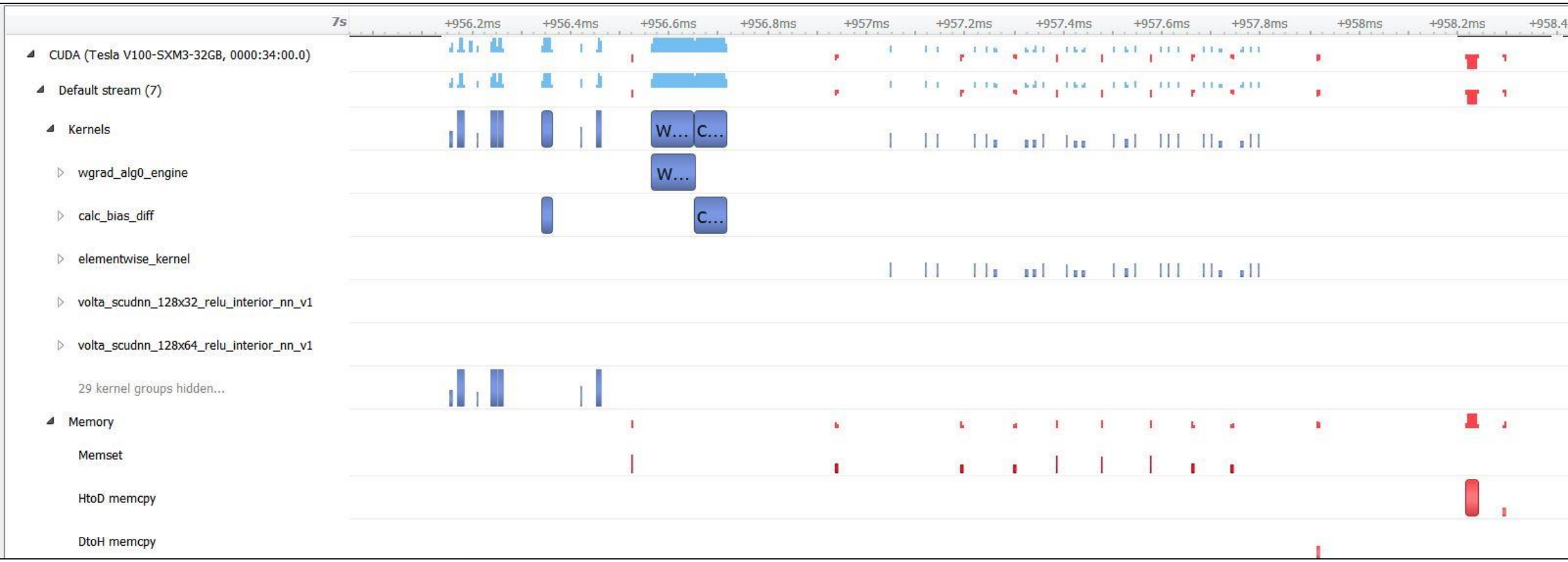
Trace CUDA API Calls on OS thread

- See when kernels are dispatched
- See when memory operations are initiated
- Locate the corresponding CUDA workload on GPU

GPU WORKLOAD

See CUDA workloads execution time

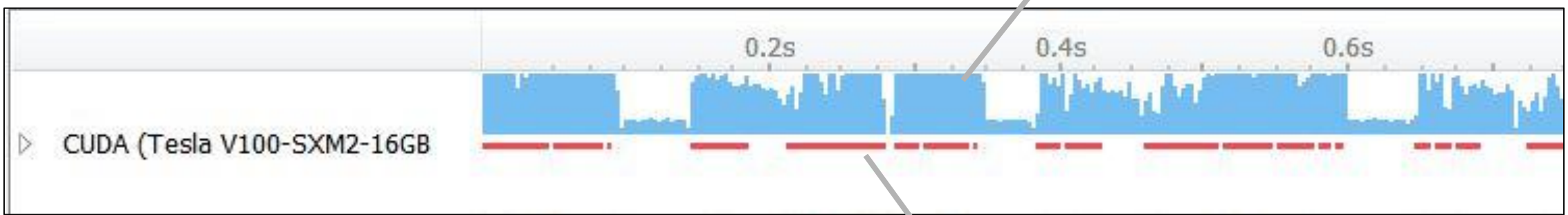
Locate idle GPU times



GPU WORKLOAD

See trace of GPU activity

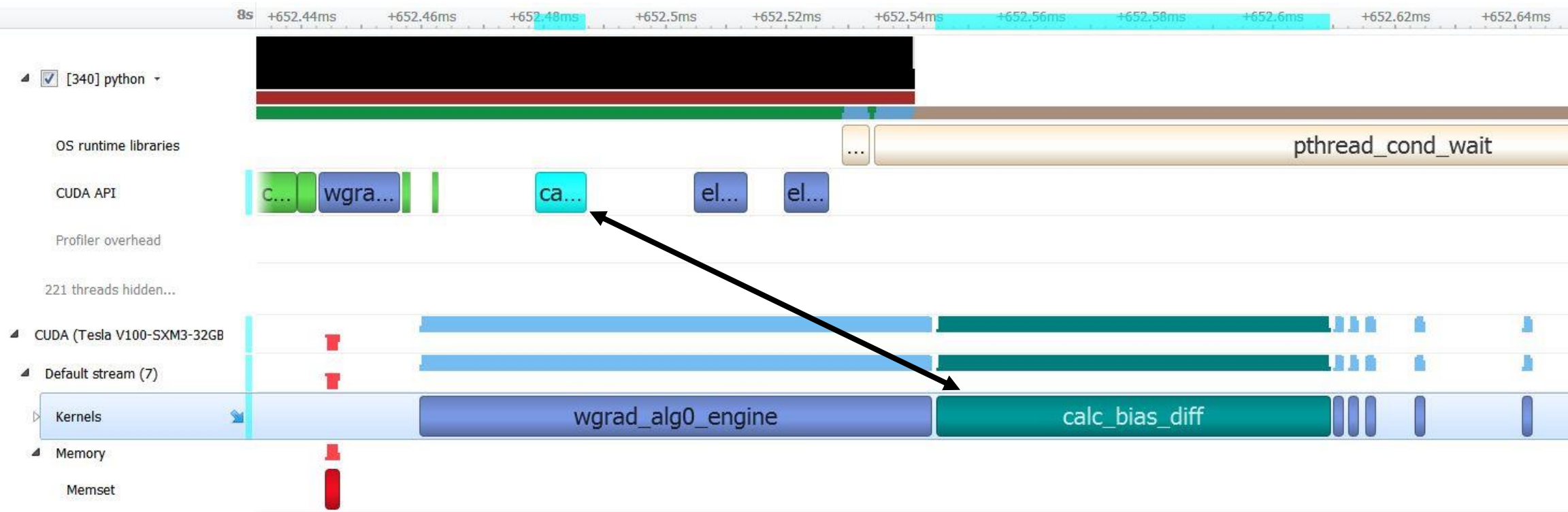
Locate idle GPU times



% Chart for
Avg. CUDA kernel coverage
(Not SM occupancy)

% Chart for
Avg. no. of memory operations

CORRELATION TIES API TO GPU WORKLOAD

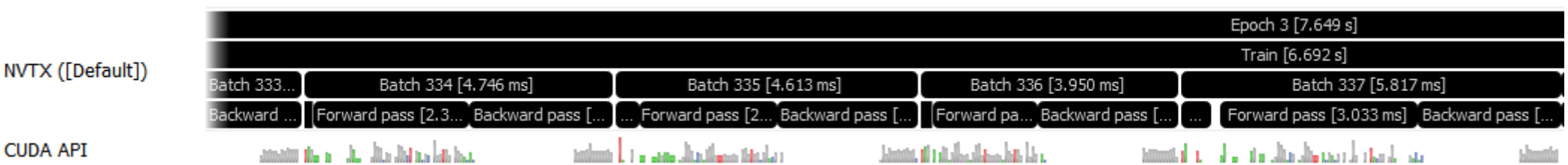


Selecting one highlights both cause and effect, i.e. dependency analysis

NVTX INSTRUMENTATION

NVIDIA Tools Extension (NVTX) to annotate the timeline with application's logic

Helps understand the profiler's output in app's algorithmic context



NVTX INSTRUMENTATION

Usage

Include the header “**nvToolsExt.h**”

Call the API functions from your source

Link the NVTX library on the compiler command line with **-lnvToolsExt**

Also supports Python

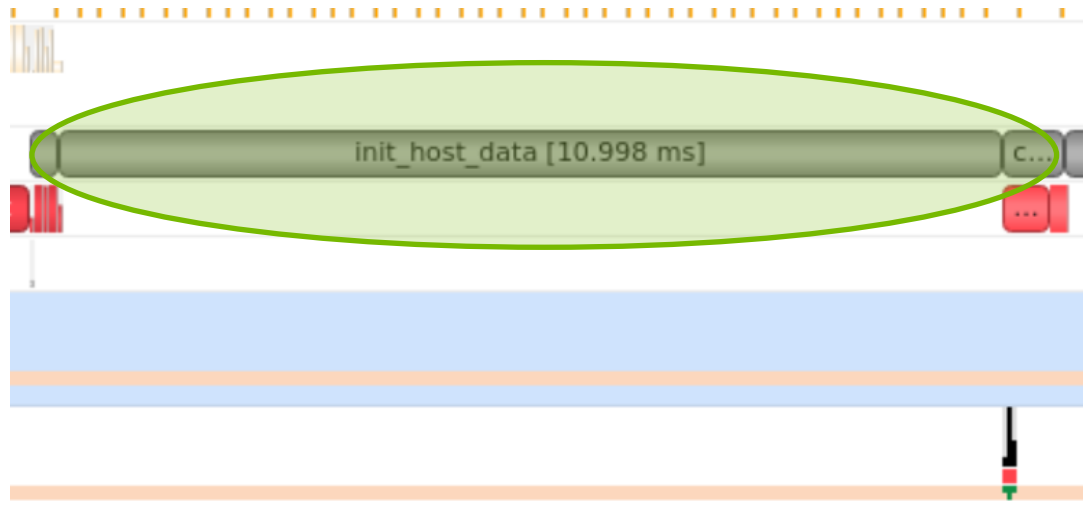
CuPy -> https://docs.cupy.dev/en/v9.0.0a1/reference/cuda.html#profiler

TF -> https://developer.nvidia.com/blog/tensorflow-performance-logging-plugin-nvtx-plugins-tf-public/

NVTX INSTRUMENTATION

Example

```
#include "nvToolsExt.h"
...
void myfunction( int n, double * x )
{
    nvtxRangePushA("init_host_data");
    //initialize x on host
    init_host_data(n,x,x_d,y_d);
    nvtxRangePop();
}
...
```



NSIGHT COMPUTE

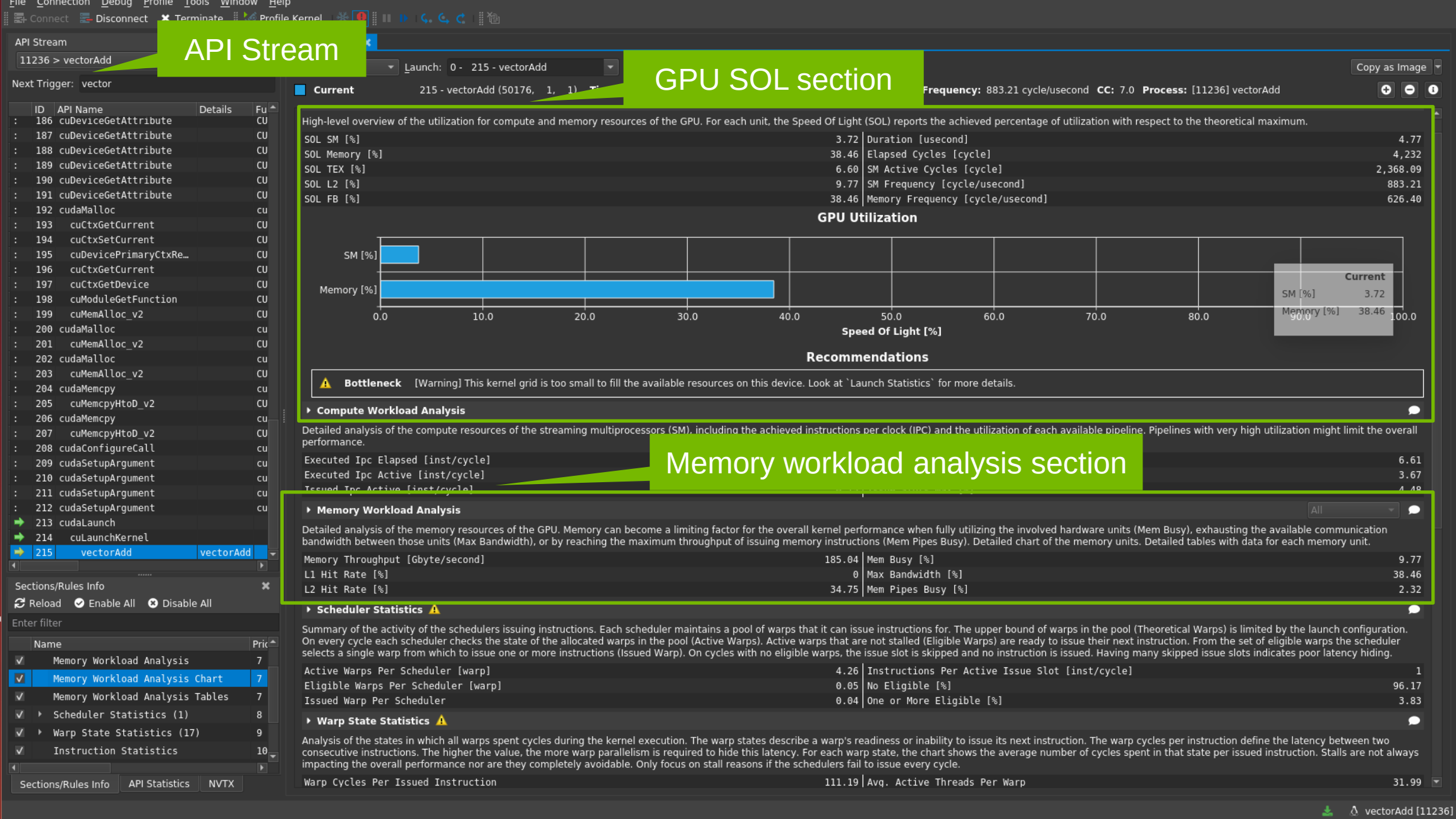
Next-Gen Kernel Profiling Tool

Interactive kernel profiler

- Graphical profile report. For example, the SOL and Memory Chart
- Differentiating results across one or multiple reports using baselines
- Fast Data Collection

The UI executable is called **nv-nsight-cu**, and the command-line one is **nv-nsight-cu-cli**

GPUs: Pascal, Volta, **Turing, Ampere**


File Connection Debug Profile Tools Window Help
Connect Disconnect Terminate Profile Kernel
API Stream
API Stream
11236 > vectorAdd
Next Trigger: vector
ID API Name Details Fu
186 cuDeviceGetAttribute CU
187 cuDeviceGetAttribute CU
188 cuDeviceGetAttribute CU
189 cuDeviceGetAttribute CU
190 cuDeviceGetAttribute CU
191 cuDeviceGetAttribute CU
192 cudaMalloc cu
193 cuCtxGetCurrent CU
194 cuCtxSetCurrent CU
195 cuDevicePrimaryCtxRe… CU
196 cuCtxGetCurrent CU
197 cuCtxGetDevice CU
198 cuModuleGetFunction CU
199 cuMemAlloc_v2 CU
200 cudaMalloc cu
201 cuMemAlloc_v2 CU
202 cudaMalloc cu
203 cuMemAlloc_v2 CU
204 cudaMemcpy cu
205 cuMemcpyHtoD_v2 CU
206 cudaMemcpy cu
207 cuMemcpyHtoD_v2 CU
208 cudaConfigureCall cu
209 cudaSetupArgument cu
210 cudaSetupArgument cu
211 cudaSetupArgument cu
212 cudaSetupArgument cu
213 cudaLaunch
214 cuLaunchKernel
215 vectorAdd vectorAdd
Sections/Rules Info
Reload Enable All Disable All
Enter filter
Name Pric
Memory Workload Analysis 7
Memory Workload Analysis Chart 7
Memory Workload Analysis Tables 7
Scheduler Statistics (1) 8
Warp State Statistics (17) 9
Instruction Statistics 10
Sections/Rules Info
API Statistics
NVTX
Launch: 0 - 215 - vectorAdd
Copy as Image
Current 215 - vectorAdd (50176, 1, 1)
Frequency: 883.21 cycle/usecond CC: 7.0 Process: [11236] vectorAdd
GPU SOL section
High-level overview of the utilization for compute and memory resources of the GPU. For each unit, the Speed Of Light (SOL) reports the achieved percentage of utilization with respect to the theoretical maximum.
SOL SM [%] 3.72
SOL Memory [%] 38.46
SOL TEX [%] 6.60
SOL L2 [%] 9.77
SOL FB [%] 38.46
Duration [usecond] 4.77
Elapsed Cycles [cycle] 4,232
SM Active Cycles [cycle] 2,368.09
SM Frequency [cycle/usecond] 883.21
Memory Frequency [cycle/usecond] 626.40
GPU Utilization
SM [%]
Memory [%]
0.0 10.0 20.0 30.0 40.0 50.0 60.0 70.0 80.0 90.0 100.0
Speed Of Light [%]
Current
SM [%] 3.72
Memory [%] 38.46
Recommendations
Bottleneck [Warning] This kernel grid is too small to fill the available resources on this device. Look at `Launch Statistics` for more details.
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] 6.61
Executed Ipc Active [inst/cycle] 3.67
Memory workload analysis section
Memory Workload Analysis
All
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] 185.04
L1 Hit Rate [%] 0
L2 Hit Rate [%] 34.75
Mem Busy [%] 9.77
Max Bandwidth [%] 38.46
Mem Pipes Busy [%] 2.32
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] 4.26
Eligible Warps Per Scheduler [warp] 0.05
Issued Warp Per Scheduler 0.04
Instructions Per Active Issue Slot [inst/cycle] 1
No Eligible [%] 96.17
One or More Eligible [%] 3.83
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.
Warp Cycles Per Issued Instruction 111.19
Avg. Active Threads Per Warp 31.99
vectorAdd [11236]

KEY FEATURES

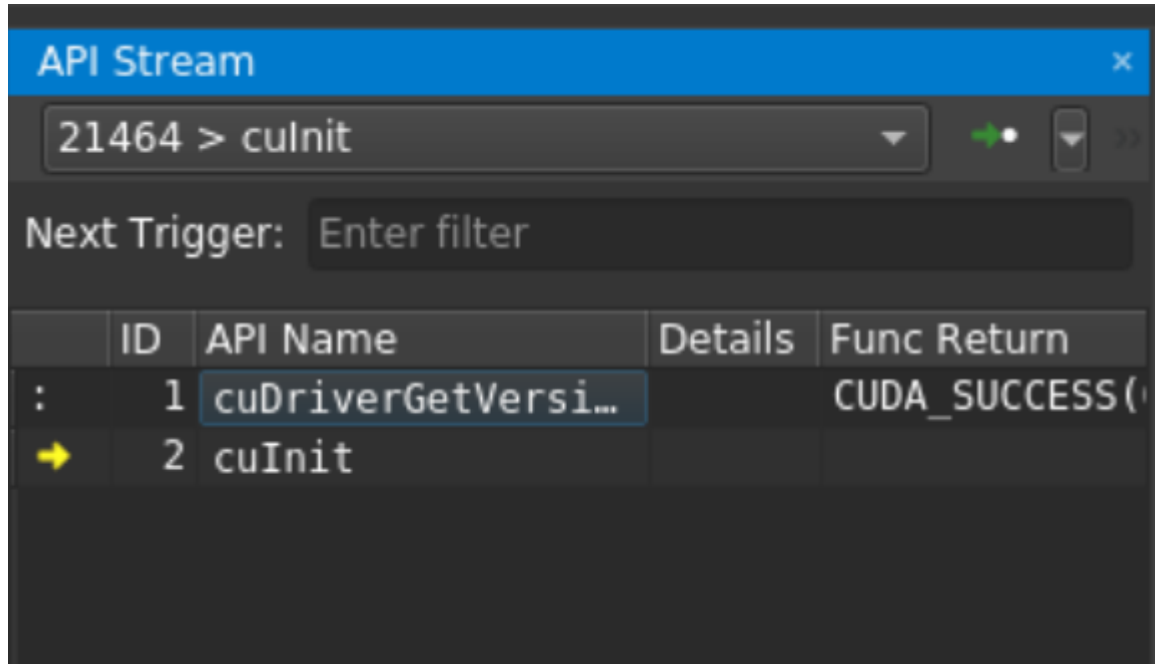
API Stream

Interactive profiling with API Stream

- Run to the next (CUDA) kernel
- Run to the next (CUDA) API call
- Run to the next range start
- Run to the next range stop

Next Trigger. The filter of API and kernel

- “foo” the next kernel launch/API call matching reg exp ‘foo’



KEY FEATURES

Sections

An **event** is a countable activity, action, or occurrence on a device

A **metric** is a characteristic of an application that is calculated from one or more event values

$$gld\_efficiency = \frac{gld_{128} * 16 + gld_{64} * 8 + gld_{32} * 4 + gld_{16} * 2 + gld_{8}}{(sm7xMioGlobalLdHit + sm7xMioGlobalLdMiss) * 32}$$

A **section** is a group of some metrics. Aim to help developers to group metrics and find optimization opportunities quickly

KEY FEATURES

“Events”

Table 10. Events Mapping Table from CUPTI Events to PerfWorks Metrics for Compute Capability >= 7.0

nvprof Event	PerfWorks Metric or Formula (>= SM 7.0)
active_cycles	sm__cycles_active.sum
active_cycles_pm	sm__cycles_active.sum
active_cycles_sys	sys__cycles_active.sum
active_warps	sm__warps_active.sum
active_warps_pm	sm__warps_active.sum
atom_count	smsp__inst_executed_op_generic_atom_dot_alu.sum
elapsed_cycles_pm	sm__cycles_elapsed.sum
elapsed_cycles_sm	sm__cycles_elapsed.sum
elapsed_cycles_sys	sys__cycles_elapsed.sum
fb_subp0_read_sectors	dram__sectors_read.sum
fb_subp1_read_sectors	dram__sectors_read.sum
fb_subp0_write_sectors	dram__sectors_write.sum
fb_subp1_write_sectors	dram__sectors_write.sum
global_atom_cas	smsp__inst_executed_op_generic_atom_dot_cas.sum
gred_count	smsp__inst_executed_op_global_red.sum
inst_executed	sm__inst_executed.sum
inst_executed_fma_pipe_s0	smsp__inst_executed_pipe_fma.sum
inst_executed_fma_pipe_s1	smsp__inst_executed_pipe_fma.sum
inst_executed_fma_pipe_s2	smsp__inst_executed_pipe_fma.sum
inst_executed_fma_pipe_s3	smsp__inst_executed_pipe_fma.sum
inst_executed_fp16_pipe_s0	smsp__inst_executed_pipe_fp16.sum
inst_executed_fp16_pipe_s1	smsp__inst_executed_pipe_fp16.sum
inst_executed_fp16_pipe_s2	smsp__inst_executed_pipe_fp16.sum
inst_executed_fp16_pipe_s3	smsp__inst_executed_pipe_fp16.sum

Events in nvprof are equivalent in NCU.

https://docs.nvidia.com/nsight-compute/NsightComputeCli/index.html#command-line-options-profile

KEY FEATURES

Metrics

Table 9. Metrics Mapping Table from CUPTI to PerfWorks

nvprof Metric	PerfWorks Metric or Formula (>= SM 7.0)
achieved_occupancy	sm__warps_active.avg.pct_of_peak_sustained_active
atomic_transactions	l1tex__t_set_accesses_pipe_lsu_mem_global_op_atom.sum + l1tex__t_set_accesses_pipe_lsu_mem_global_op_red.sum
atomic_transactions_per_request	(l1tex__t_sectors_pipe_lsu_mem_global_op_atom.sum + l1tex__t_sectors_pipe_lsu_mem_global_op_red.sum) / (l1tex__t_requests_pipe_lsu_mem_global_op_atom.sum + l1tex__t_requests_pipe_lsu_mem_global_op_red.sum)
branch_efficiency	n/a
cf_executed	smsp__inst_executed_pipe_cbu.sum + smsp__inst_executed_pipe_adu.sum
cf_fu_utilization	n/a
cf_issued	n/a
double_precision_fu_utilization	smsp__inst_executed_pipe_fp64.avg.pct_of_peak_sustained_active
dram_read_bytes	dram__bytes_read.sum
dram_read_throughput	dram__bytes_read.sum.per_second
dram_read_transactions	dram__sectors_read.sum
dram_utilization	dram__throughput.avg.pct_of_peak_sustained_elapsed
dram_write_bytes	dram__bytes_write.sum
dram_write_throughput	dram__bytes_write.sum.per_second
dram_write_transactions	dram__sectors_write.sum
eligible_warps_per_cycle	smsp__warps_eligible.sum.per_cycle_active
flop_count_dp	smsp__sass_thread_inst_executed_op_dadd_pred_on.sum + smsp__sass_thread_inst_executed_op_dmul_pred_on.sum + smsp__sass_thread_inst_executed_op_dfma_pred_on.sum * 2
flop_count_dp_add	smsp__sass_thread_inst_executed_op_dadd_pred_on.sum
flop_count_dp_fma	smsp__sass_thread_inst_executed_op_dfma_pred_on.sum
flop_count_dp_mul	smsp__sass_thread_inst_executed_op_dmul_pred_on.sum
flop_count_hp	smsp__sass_thread_inst_executed_op_hadd_pred_on.sum +

Metrics are much different from nvprof and more related to HW.

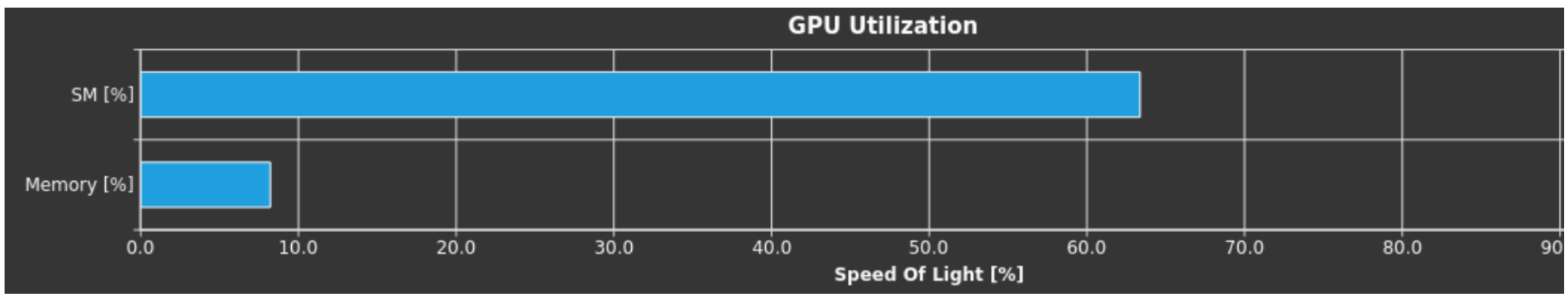
https://docs.nvidia.com/nsight-compute/NsightComputeCli/index.html#command-line-options-profile

SOL SECTION

Sections

SOL Section (case 1: Compute Bound)

- High-level overview of the utilization for compute and memory resources of the GPU. For each unit, the Speed Of Light (SOL) reports the achieved percentage of utilization with respect to the theoretical maximum

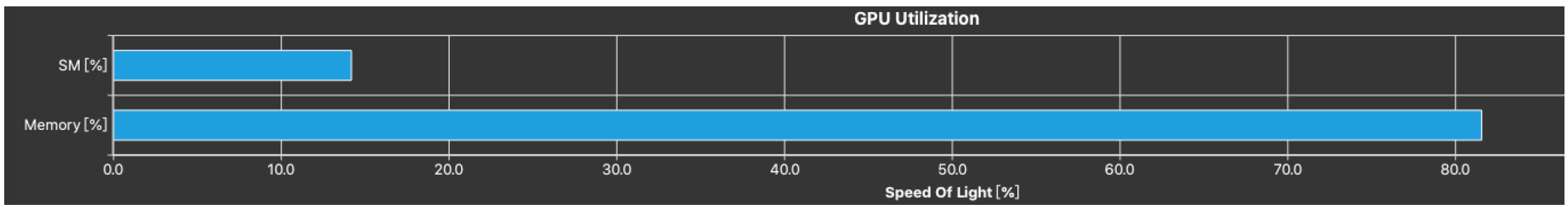


SOL SECTION

Sections

SOL Section (case 2: Memory Bound)

- High-level overview of the utilization for compute and memory resources of the GPU. For each unit, the Speed Of Light (SOL) reports the achieved percentage of utilization with respect to the theoretical maximum

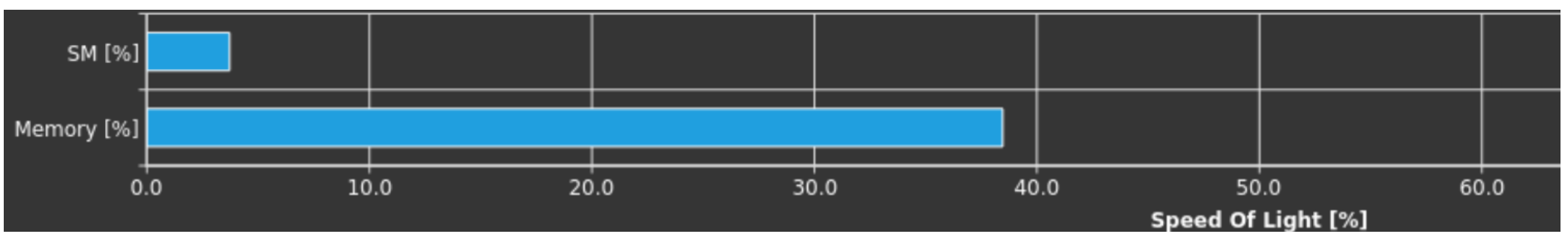


SOL SECTION

Sections

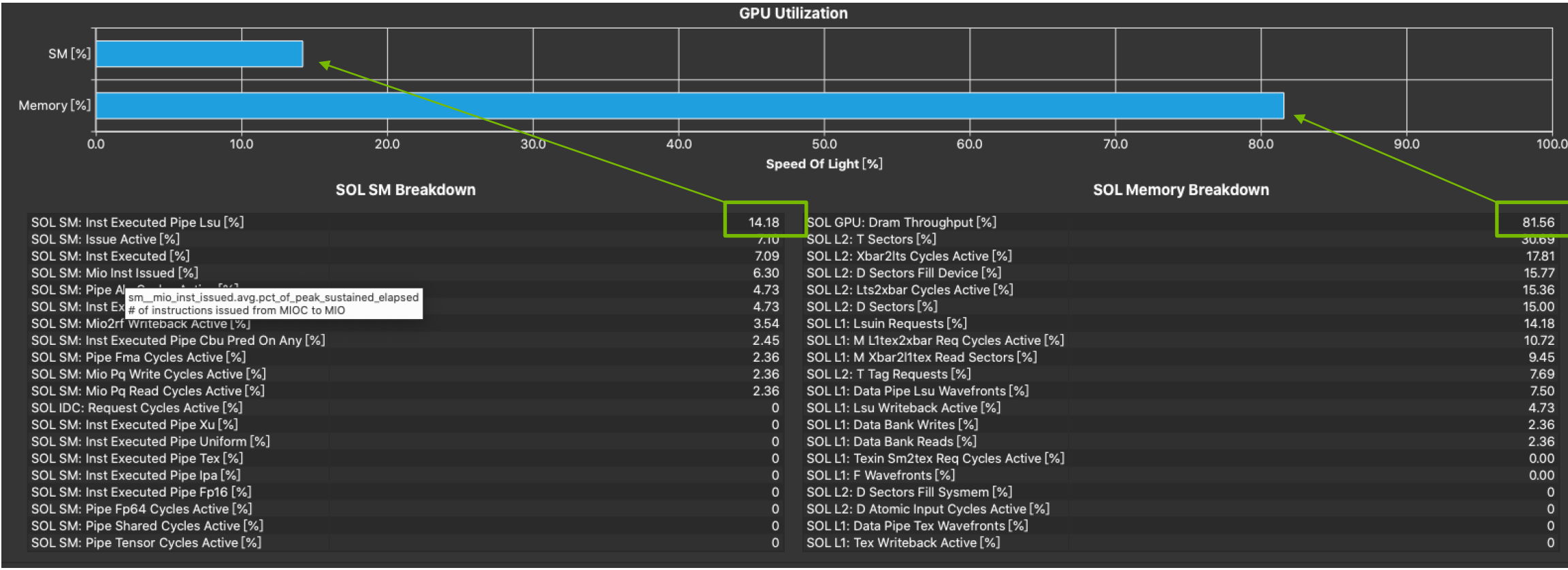
SOL Section (case 3: Latency Bound)

- High-level overview of the utilization for compute and memory resources of the GPU. For each unit, the Speed Of Light (SOL) reports the achieved percentage of utilization with respect to the theoretical maximum



SOL SECTION

Sections



SOL SM Breakdown

SOL SM: Inst Executed Pipe Lsu [%]	14.18
SOL SM: Issue Active [%]	7.10
SOL SM: Inst Executed [%]	7.09
SOL SM: Mio Inst Issued [%]	6.30
SOL SM: Pipe Al	4.73
SOL SM: Inst Ex	4.73
SOL SM: Mio2rf Writeback Active [%]	3.54
SOL SM: Inst Executed Pipe Cbu Pred On Any [%]	2.45
SOL SM: Pipe Fma Cycles Active [%]	2.36
SOL SM: Mio Pq Write Cycles Active [%]	2.36
SOL SM: Mio Pq Read Cycles Active [%]	2.36
SOL IDC: Request Cycles Active [%]	0
SOL SM: Inst Executed Pipe Xu [%]	0
SOL SM: Inst Executed Pipe Uniform [%]	0
SOL SM: Inst Executed Pipe Tex [%]	0
SOL SM: Inst Executed Pipe Ipa [%]	0
SOL SM: Inst Executed Pipe Fp16 [%]	0
SOL SM: Pipe Fp64 Cycles Active [%]	0
SOL SM: Pipe Shared Cycles Active [%]	0
SOL SM: Pipe Tensor Cycles Active [%]	0

sm__mio_inst_issued.avg.pct_of_peak_sustained_elapsed
# of instructions issued from MIOC to MIO

SOL Memory Breakdown

SOL GPU: Dram Throughput [%]	81.56
SOL L2: T Sectors [%]	30.69
SOL L2: Xbar2lts Cycles Active [%]	17.81
SOL L2: D Sectors Fill Device [%]	15.77
SOL L2: Lts2xbar Cycles Active [%]	15.36
SOL L2: D Sectors [%]	15.00
SOL L1: Lsuin Requests [%]	14.18
SOL L1: M L1tex2xbar Req Cycles Active [%]	10.72
SOL L1: M Xbar2l1tex Read Sectors [%]	9.45
SOL L2: T Tag Requests [%]	7.69
SOL L1: Data Pipe Lsu Wavefronts [%]	7.50
SOL L1: Lsu Writeback Active [%]	4.73
SOL L1: Data Bank Writes [%]	2.36
SOL L1: Data Bank Reads [%]	2.36
SOL L1: Texin Sm2tex Req Cycles Active [%]	0.00
SOL L1: F Wavefronts [%]	0.00
SOL L2: D Sectors Fill Sysmem [%]	0
SOL L2: D Atomic Input Cycles Active [%]	0
SOL L1: Data Pipe Tex Wavefronts [%]	0
SOL L1: Tex Writeback Active [%]	0

SOL SECTION

Unit details

Units

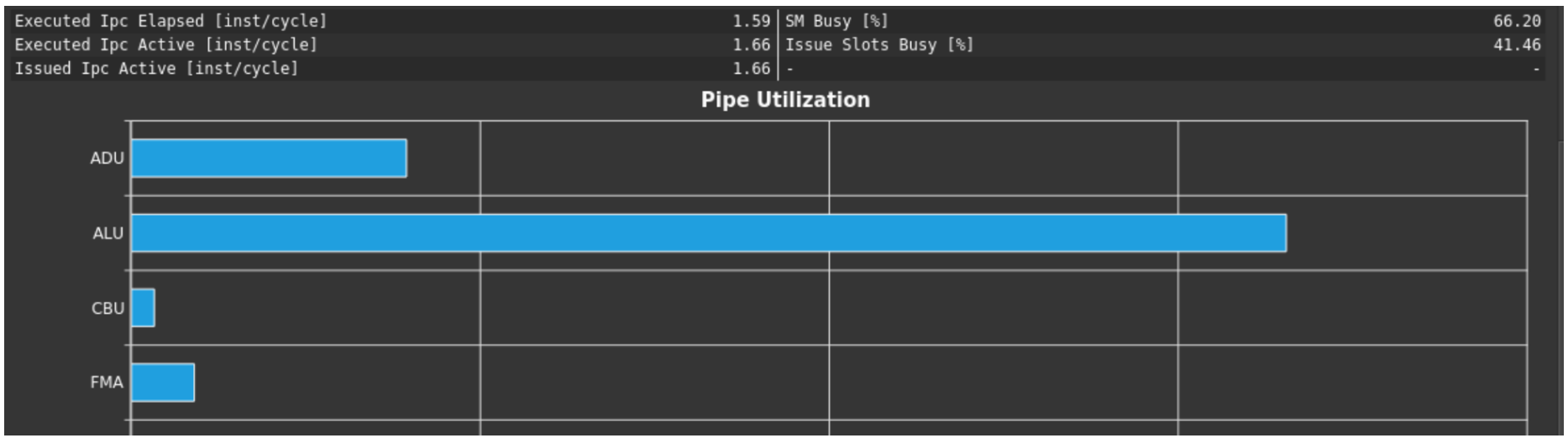
`dram`	Device (main) memory, where the GPUs global and local memory resides.
`fbpa`	The FrameBuffer Partition is a memory controller which sits between the level 2 cache (LTC) and the DRAM. The number of FBPAs varies across GPUs.
`fe`	The Frontend unit is responsible for the overall flow of workloads sent by the driver. FE also facilitates a number of synchronization operations.
`gpc`	The General Processing Cluster contains SM, Texture and L1 in the form of TPC(s). It is replicated several times across a chip.
`gpu`	The entire Graphics Processing Unit.
`gr`	Graphics Engine is responsible for all 2D and 3D graphics, compute work, and synchronous graphics copying work.
`idc`	The InDexed Constant Cache is a subunit of the SM responsible for caching constants that are indexed with a register.
`l1tex`	The Level 1 (L1)/Texture Cache is located within the GPC. It can be used as directed-mapped shared memory and/or store global, local and texture data in its cache portion.
`lts`	A Level 2 (L2) Cache Slice is a sub-partition of the Level 2 cache.
`sm`	The Streaming Multiprocessor handles execution of a kernel as groups of 32 threads, called warps. Warps are further grouped into cooperative thread arrays (CTA), called blocks in CUDA. All warps of a CTA execute on the same SM. CTAs share various resources across their threads, e.g. the shared memory.
`smsp`	SMSP is a sub-partition of the SM.
`sys`	Logical grouping of several units
`tpc`	Thread Processing Clusters are units in the GPC. They contain one or more SM, Texture and L1 units, the Instruction Cache (ICC) and the Indexed Constant Cache (IDC).

COMPUTE WORKLOAD ANALYSIS

Sections

Compute Workload Analysis (case 1)

- 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



COMPUTE WORKLOAD ANALYSIS

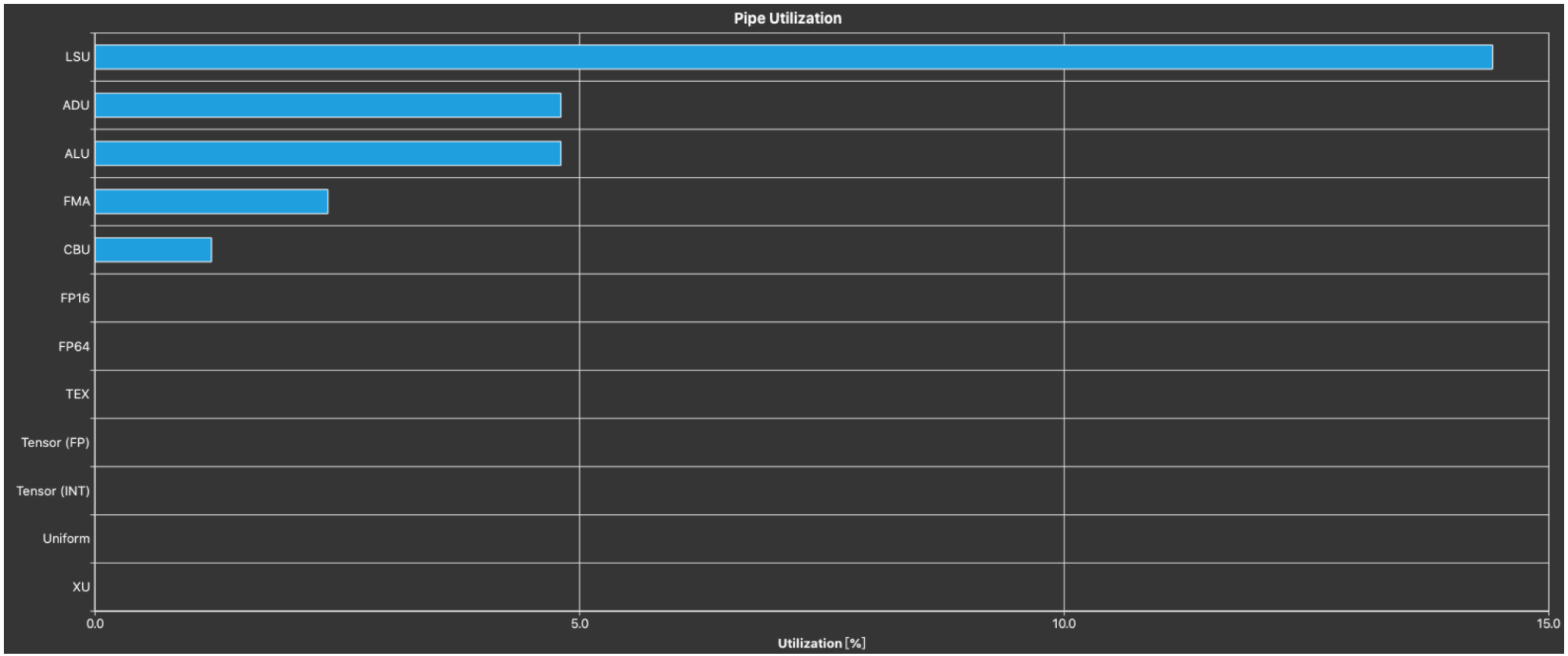
Compute Pipelines Details

Pipelines

ADU	Address Divergence Unit. The ADU is responsible for address divergence handling for branches/jumps. It also provides support for constant loads and block-level barrier instructions.
ALU	Arithmetic Logic Unit. The ALU is responsible for execution of most bit manipulation and logic instructions. It also executes integer instructions, excluding IMAD and IMUL. On NVIDIA Ampere architecture chips, the ALU pipeline performs fast FP32-to-FP16 conversion.
CBU	Convergence Barrier Unit. The CBU is responsible for warp-level convergence, barrier, and branch instructions.
FMA	Fused Multiply Add/Accumulate. The FMA pipeline processes most FP32 arithmetic (FADD, FMUL, FMAD). It also performs integer multiplication operations (IMUL, IMAD), as well as integer dot products. On GA10x, FMA is a logical pipeline that indicates peak FP32 and FP16x2 performance. It is composed of the FMAHeavy and FMALite physical pipelines.
FMA (FP16)	FMA (FP16) represents FP16x2 instruction execution within the logical FMA pipeline. It also contains a fast FP16-to-FP32 converter.
FMALite	FMALite performs FP32 arithmetic (FADD, FMUL, FMA) and FP16 arithmetic (HADD2, HMUL2, HFMA2).
FMAHeavy	FMAHeavy performs FP32 arithmetic (FADD, FMUL, FMAD), FP16 arithmetic (HADD2, HMUL2, HFMA2), and integer dot products.
FP16	Half-precision floating-point unit. On Volta, Turing and NVIDIA GA100, the FP16 pipeline performs paired FP16 instructions (FP16x2). It also contains a fast FP32-to-FP16 and FP16-to-FP32 converter. Starting with GA10x chips, this functionality is part of the FMA pipeline.
FP64	Double-precision floating-point unit. The FP64 unit is responsible for most FP64 instructions (DADD, DMUL, DMAD, ...). The implementation of FP64 varies greatly per chip. Consequently, its throughput can differ significantly.
LSU	Load Store Unit. The LSU pipeline issues load, store, atomic, and reduction instructions to the L1TEX unit for global, local, and shared memory. It also issues special register reads (S2R), shuffles, and CTA-level arrive/wait barrier instructions to the L1TEX unit.
Tensor (DP)/Tensor (FP64)	Double-precision floating-point matrix-multiply and accumulate unit.
Tensor (FP)/Tensor (FP16/TF32)	Mixed-precision (FP16/TF32 and FP32) floating-point matrix-multiply and accumulate unit.
Tensor (INT)	Integer matrix-multiply and accumulate unit.
TEX	Texture Unit. The SM texture pipeline forwards texture and surface instructions to the L1TEX unit's TEXIN stage. On GPUs where FP64 or Tensor pipelines are decoupled, the texture pipeline forwards those types of instructions, too.
Uniform	Uniform Data Path. This scalar unit executes instructions where all threads use the same input and generate the same output.
XU	Transcendental and Data Type Conversion Unit. The XU pipeline is responsible for special functions such as sin, cos, and reciprocal square root. It is also responsible for int-to-float, and float-to-int type conversions.

COMPUTE WORKLOAD ANALYSIS

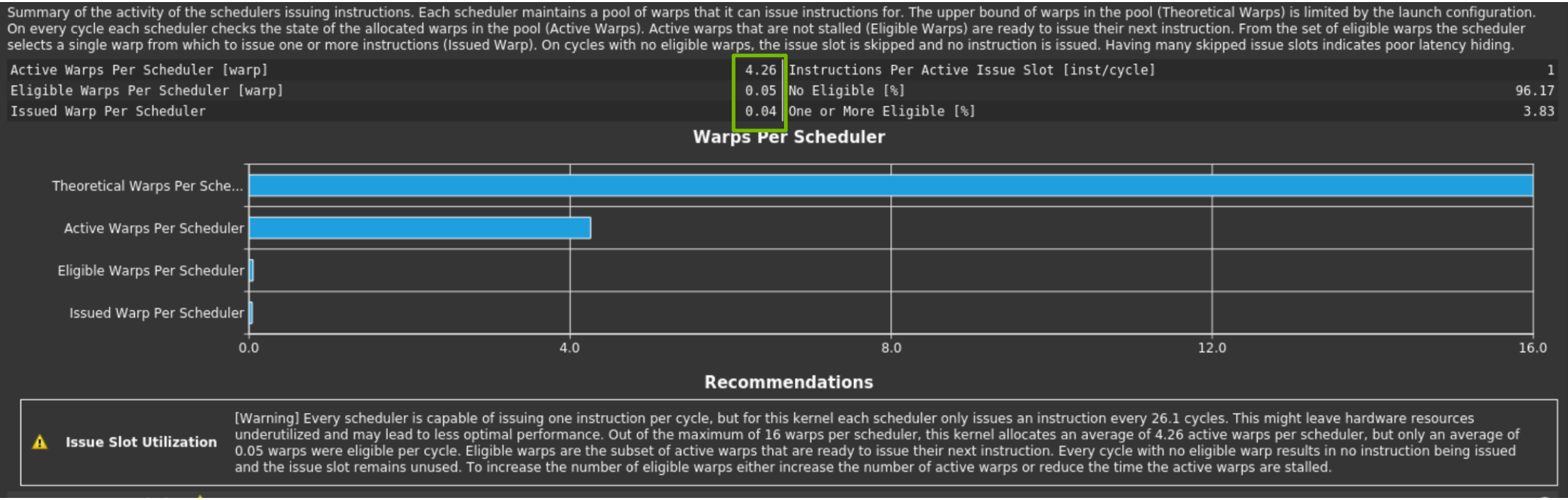
Sections



SCHEDULER STATISTICS

Sections

Scheduler Statistics(case 2)



WARP STATE STATISTICS

Sections

Warp State Statistics (case 2)

major reasons cause stall:

- an instruction fetch,
- a memory dependency (result of memory instruction),
- an execution dependency (result of previous instruction),
- a pipeline is busy,
- a synchronization barrier.

WARP SCHEDULER

Volta Architecture

4 Warp Scheduler per SM

Manages a pool of warps:

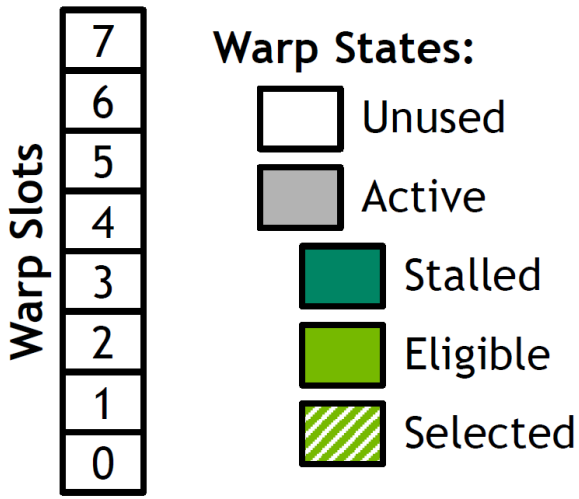
- Volta: 16 warp slots
- Turing: 8 warp slots

Each scheduler can issue 1 warp/cycle

Offers simplified mental model for profiling and SM metrics

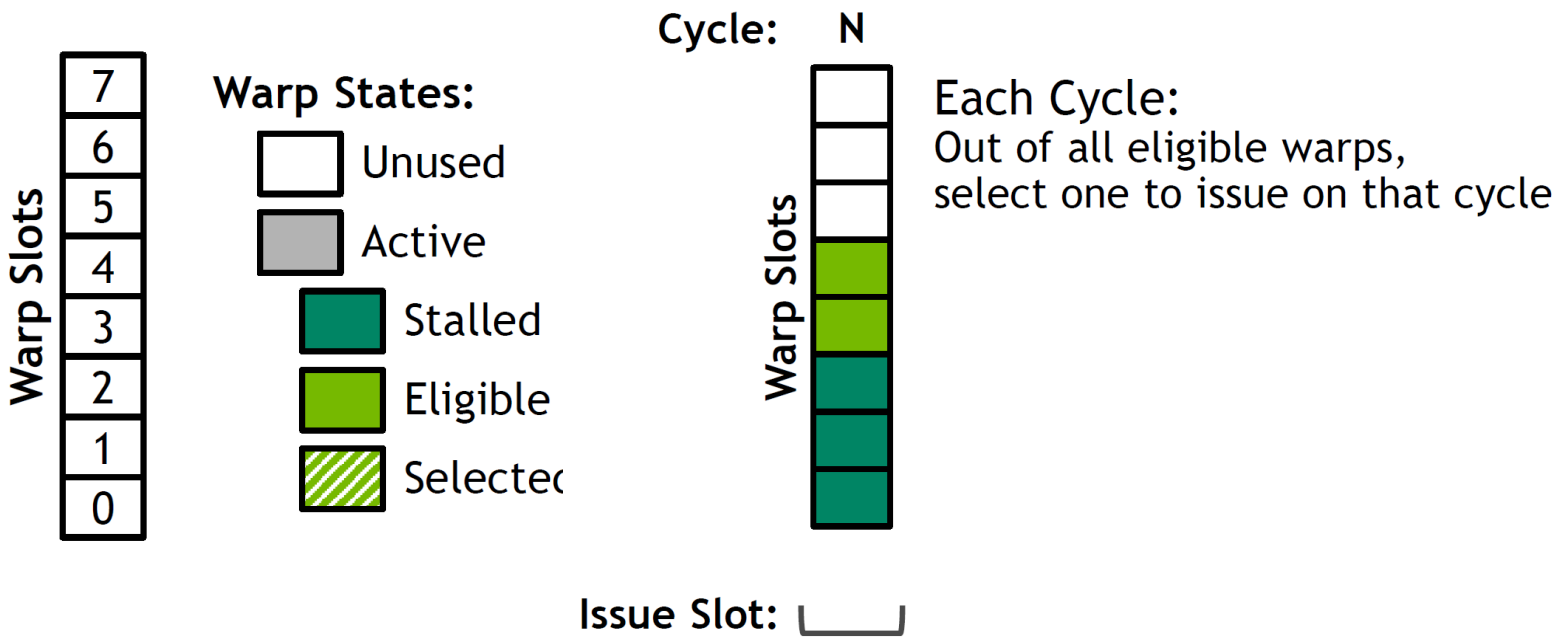
WARP SCHEDULER

Mental Model for Profiling



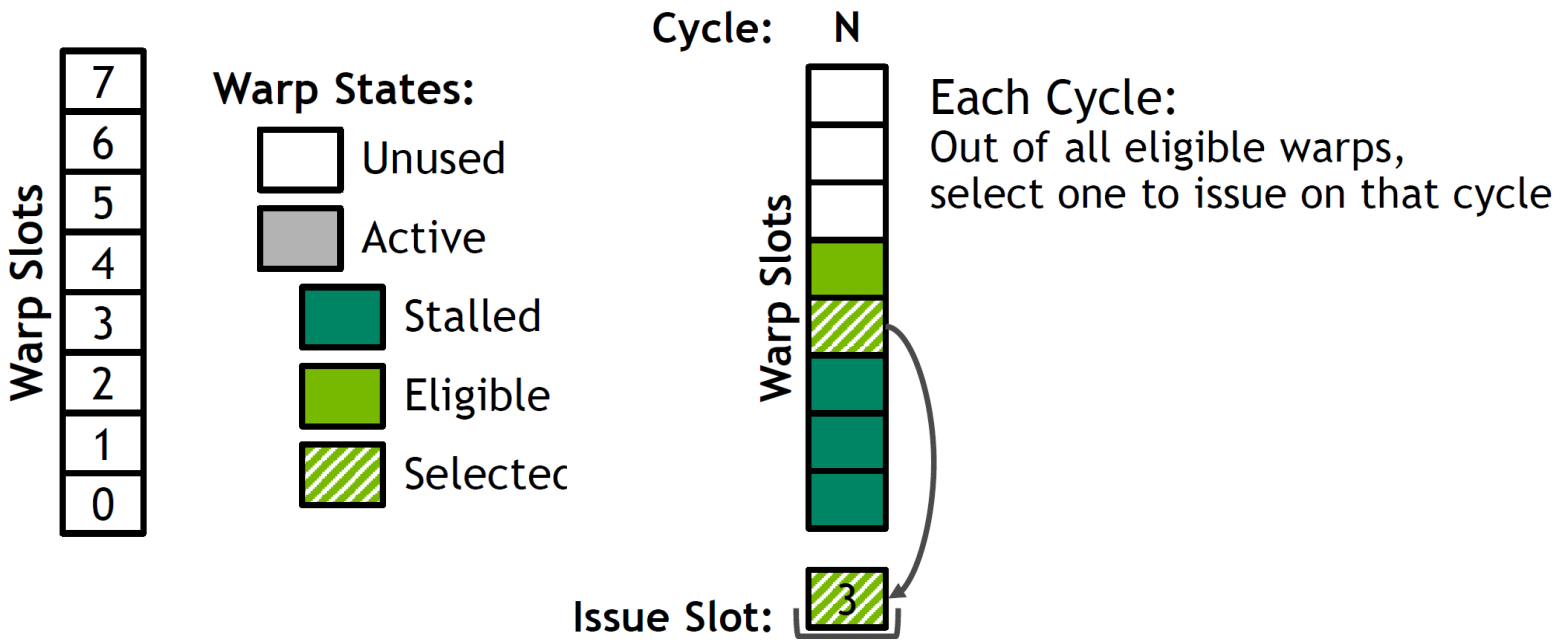
WARP SCHEDULER

Mental Model for Profiling



WARP SCHEDULER

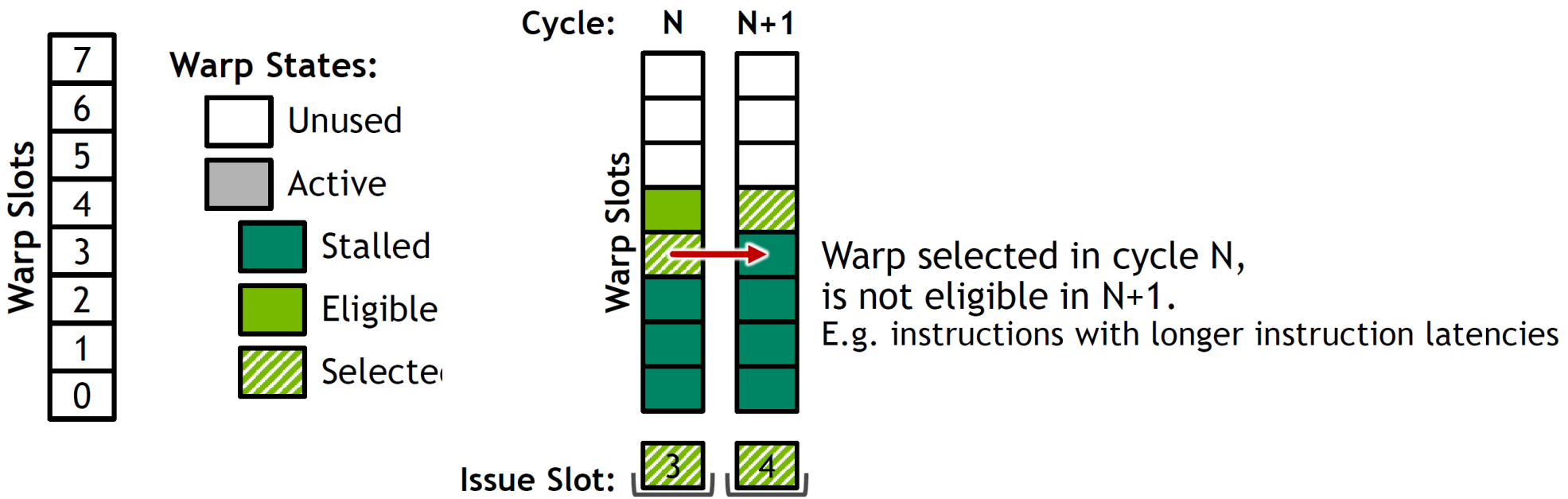
Mental Model for Profiling



Each Cycle:
Out of all eligible warps,
select one to issue on that cycle

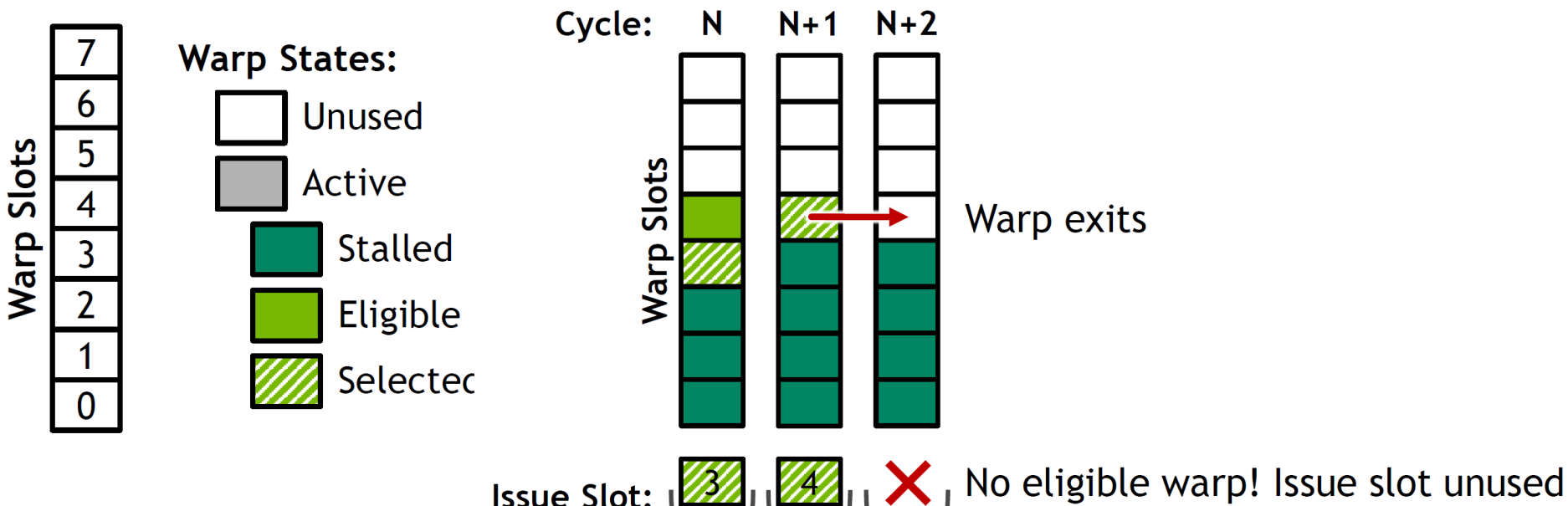
WARP SCHEDULER

Mental Model for Profiling



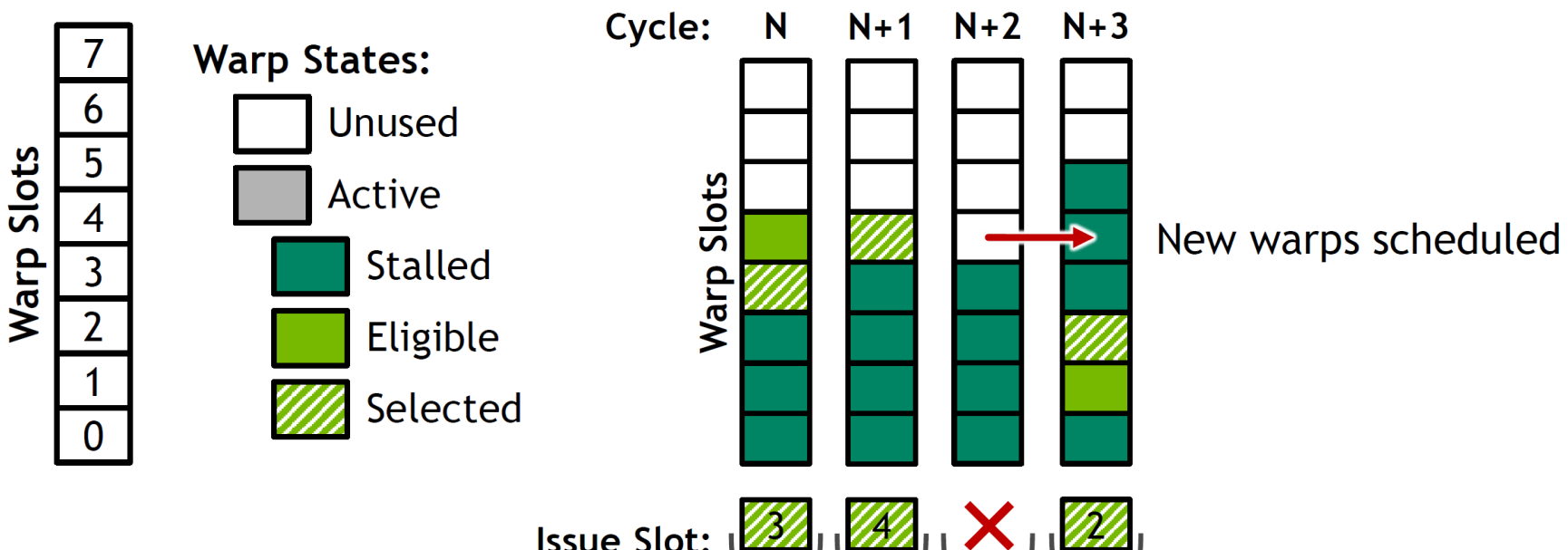
WARP SCHEDULER

Mental Model for Profiling



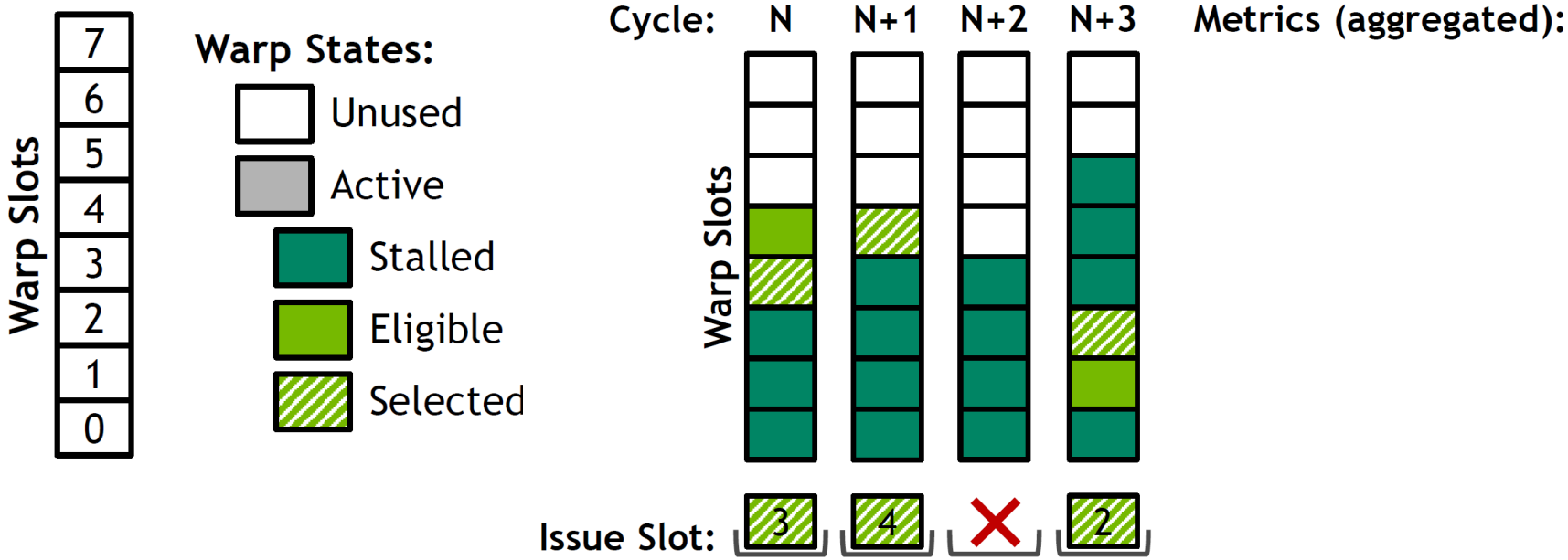
WARP SCHEDULER

Mental Model for Profiling



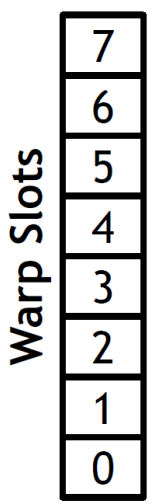
WARP SCHEDULER

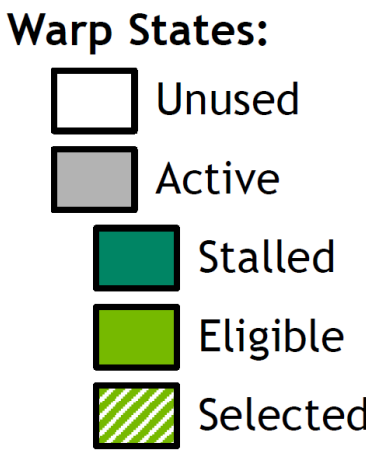
Mental Model for Profiling

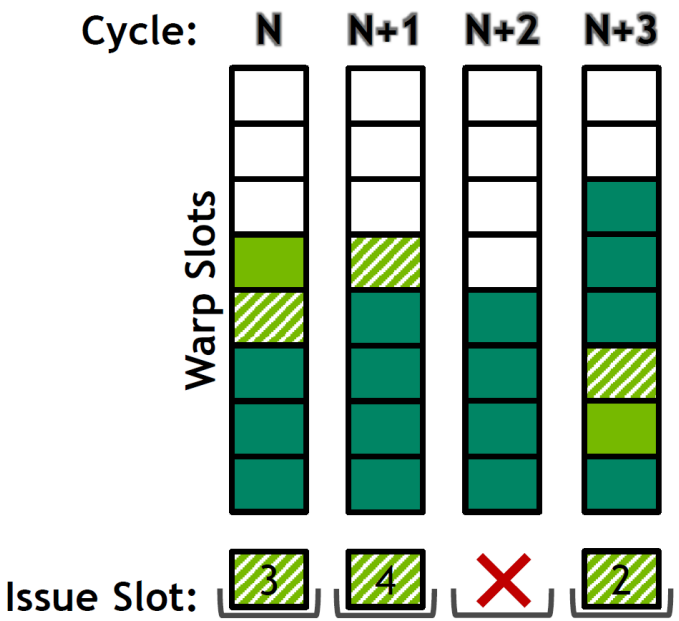


WARP SCHEDULER

Mental Model for Profiling





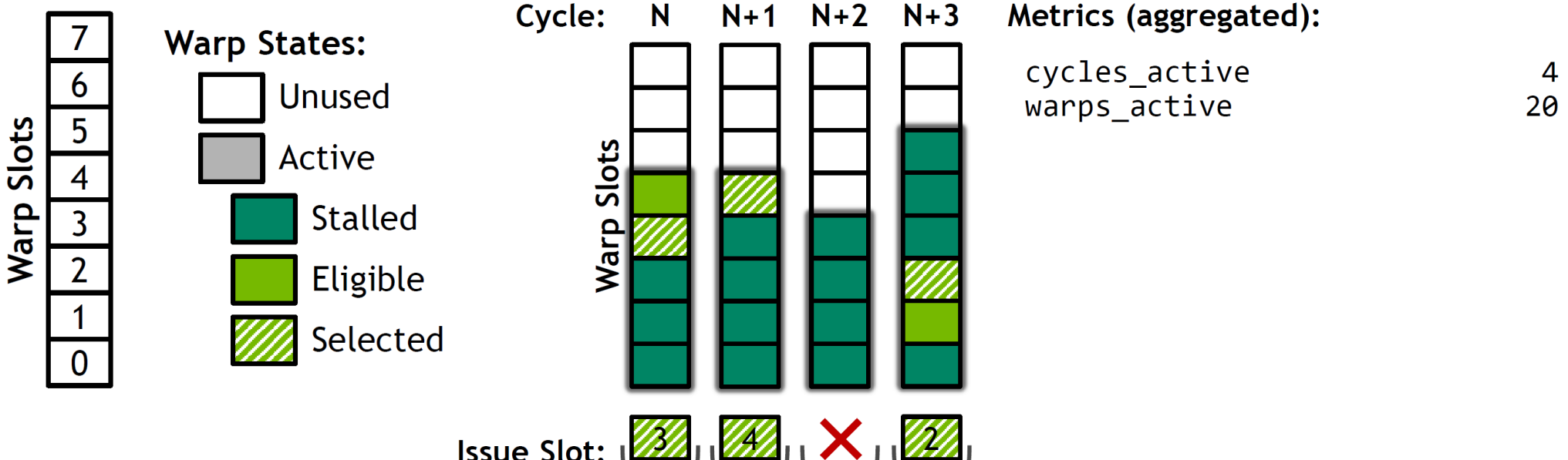


Metrics (aggregated):

`cycles_active`	4

WARP SCHEDULER

Mental Model for Profiling

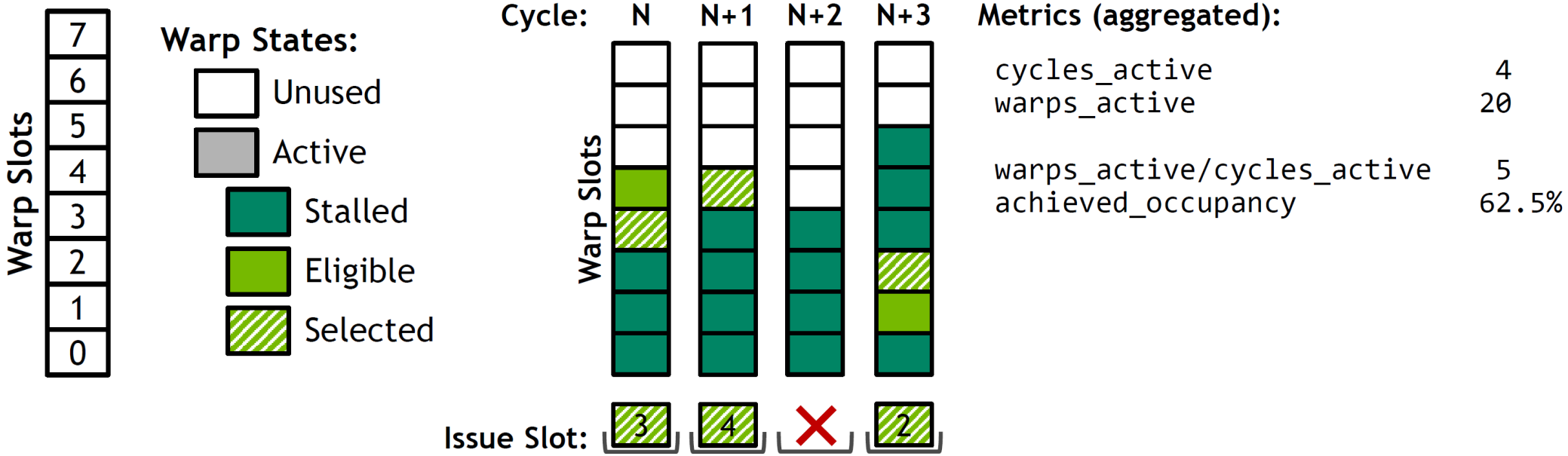


Metrics (aggregated):

`cycles_active`	4
`warps_active`	20

WARP SCHEDULER

Mental Model for Profiling

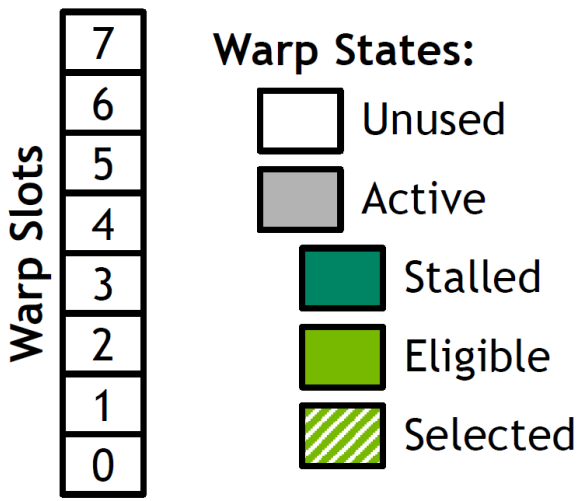


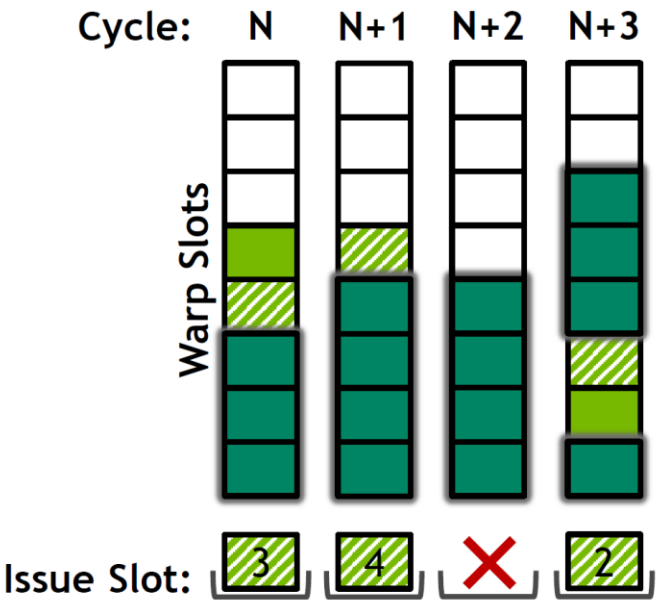
Metrics (aggregated):

cycles_active	4
warps_active	20
warps_active/cycles_active	5
achieved_occupancy	62.5%

WARP SCHEDULER

Mental Model for Profiling



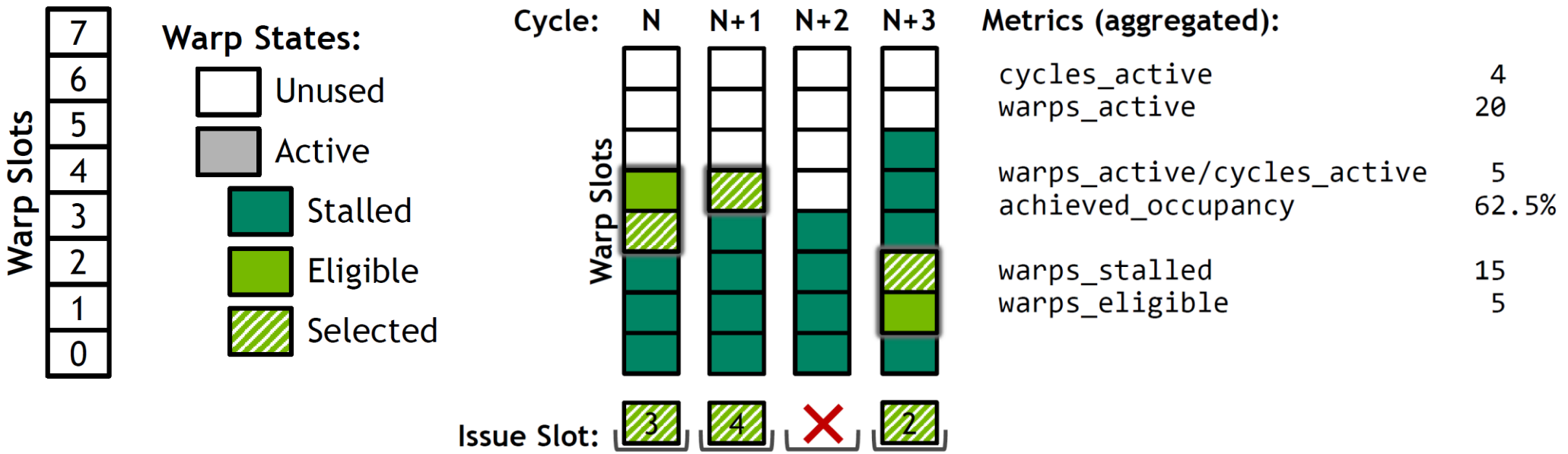


Metrics (aggregated):

cycles_active	4
warps_active	20
warps_active/cycles_active	5
achieved_occupancy	62.5%
warps_stalled	15

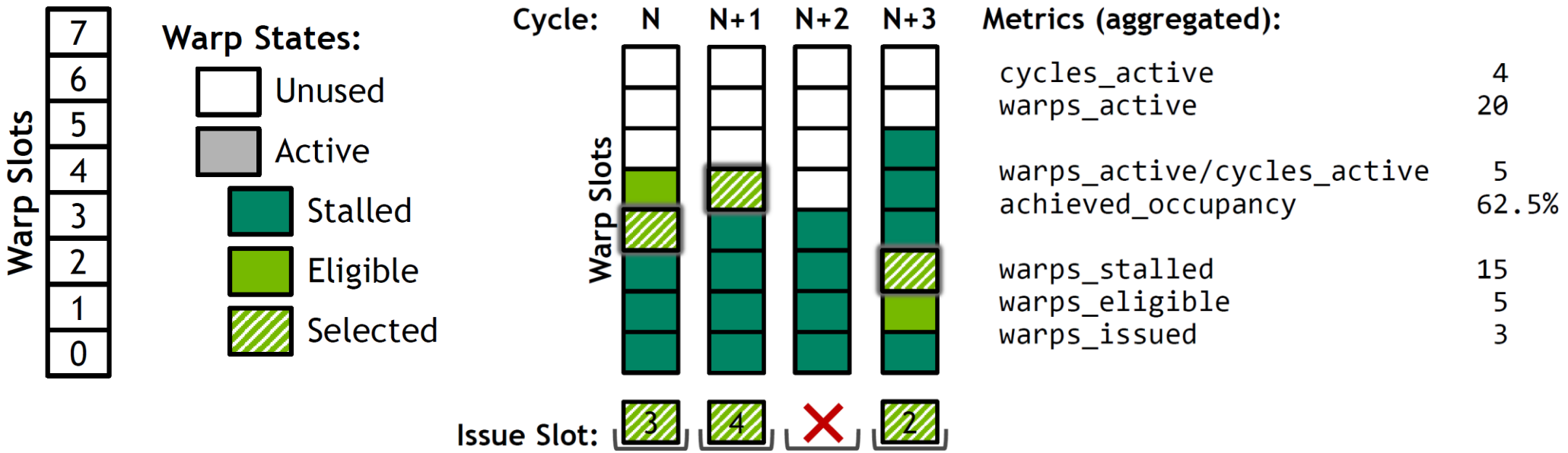
WARP SCHEDULER

Mental Model for Profiling



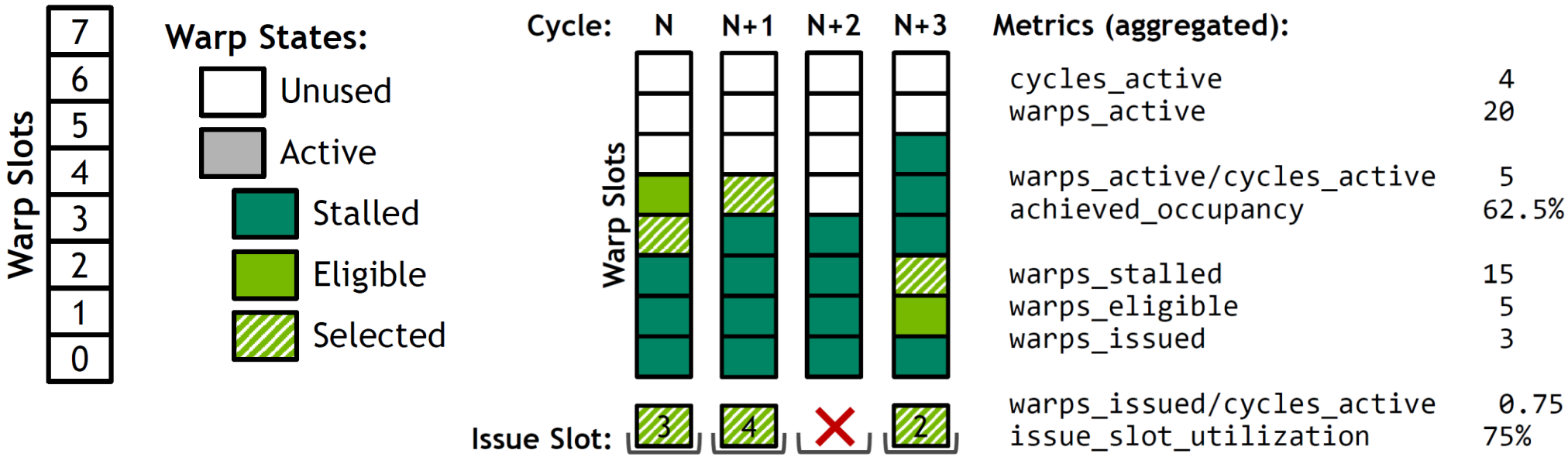
WARP SCHEDULER

Mental Model for Profiling



WARP SCHEDULER

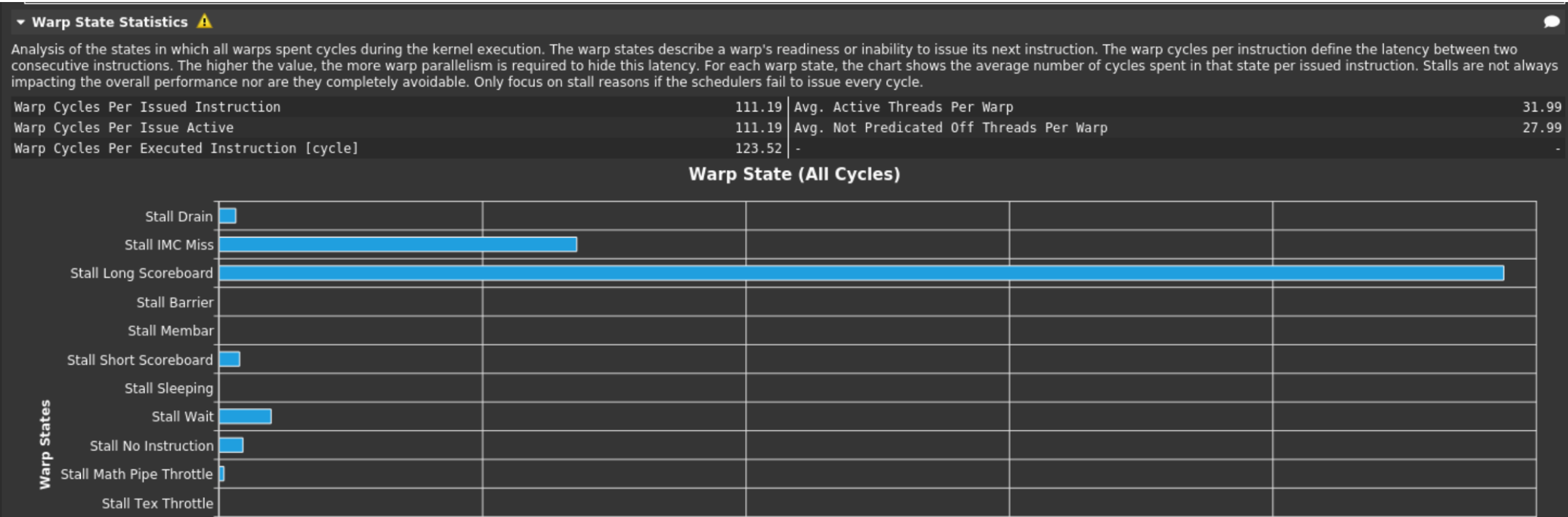
Mental Model for Profiling



WARP STATE STATISTICS

Sections

Warp State Statistics (case 2)

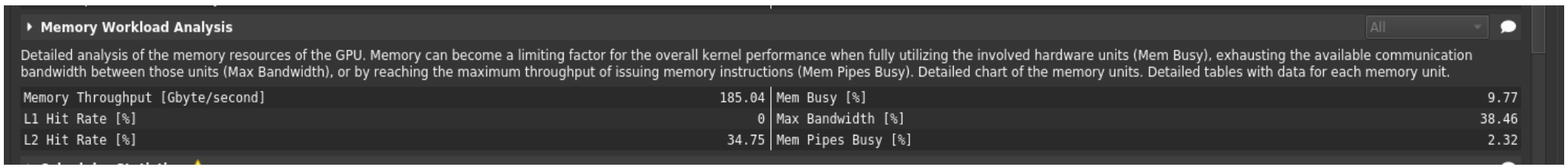


MEMORY WORKLOAD ANALYSIS

Sections

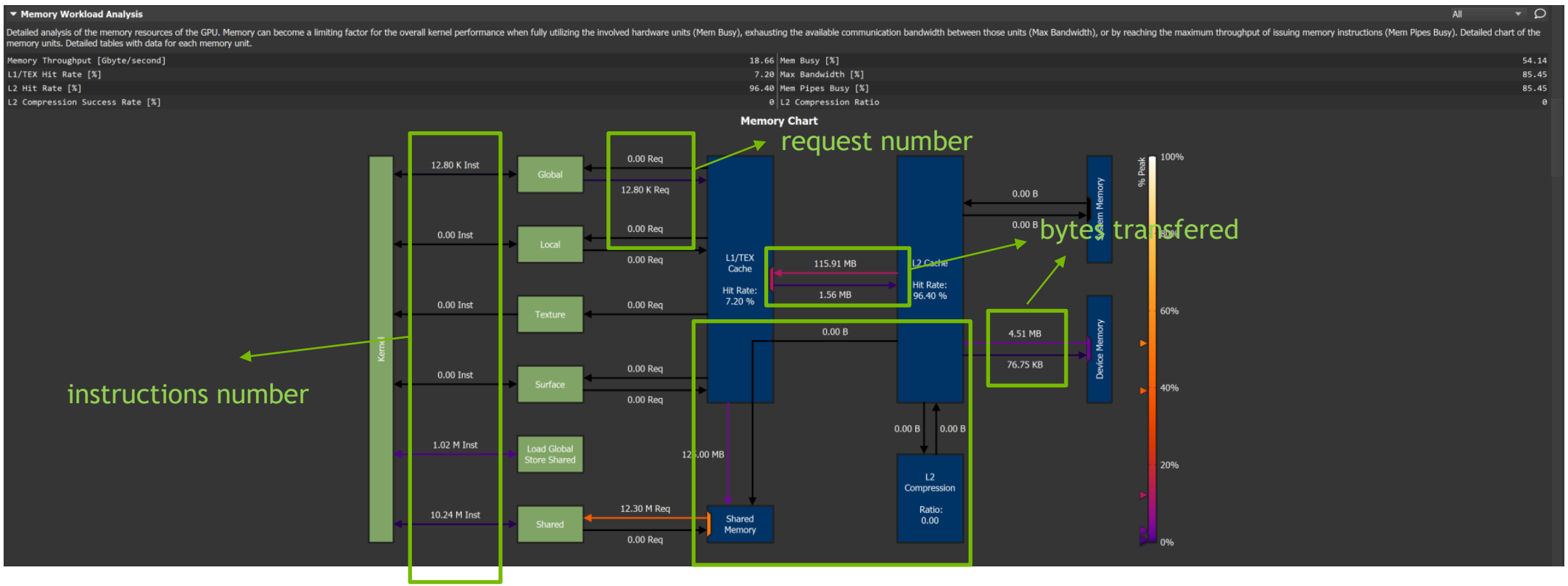
Memory Workload Analysis

- 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). Depending on the limiting factor, the memory chart and tables allow to identify the exact bottleneck in the memory system.



MEMORY WORKLOAD ANALYSIS

Memory Hierarchy



For Ampere LDGSTS and L2 Compression

MEMORY WORKLOAD ANALYSIS

Memory Report

Shared Memory

	Instructions	Wavefronts	% Peak	Bank Conflicts
Shared Load	10,240,000	12,299,255	39.32	7,161
Shared Store	0	0	0	84,208
Shared Store From Global Load	1,024,000	2,048,000	1.64	1,024,000
Shared Atomic	0	0	0	0
Other	-	3,841,234	12.28	0
Total	11,264,000	18,188,489	53.24	1,115,369

L1/TEX Cache

	Instructions	Requests	Sectors	Sectors/Req	Wavefront % Peak	Hit Rate	Bytes	Sector Misses to L2	% Peak to L2	Returns to SM	% Peak to SM
Global Load To Shared Store (access)	1,024,000	1,024,000	4,096,000	4	4.67	7.12	0			-	-
Global Load To Shared Store (bypass)	0	0	0	0		-	0	0	0	-	-
Surface Load	0	0	0	0	0	0	0	0	0		
Texture Load	0	0	0	0	0	0	0	0	0	0	0
Global Store	12,800	12,800	51,200	4	0.06	3.13	1,638,400			-	-
Local Store	0	0	0	0	0	0	0	51,200	0.16	-	-
Surface Store	0	0	0	0	0	0	0	0	0	-	-
Global Reduction	0	0	0	0	0	0	0	0	0	-	-
Surface Reduction	0	0	0	0	0	0	0	0	0	-	-
Global Atomic ALU	0	0	0	0	0	0	0	0	0	see above	see above
Global Atomic CAS	0										
Surface Atomic ALU	0	0	0	0	0	0	0	0	0	see above	see above
Surface Atomic CAS	0										
Loads	1,024,000	1,024,000	4,096,000	4	4.67	7.12	0	3,812,620	12.19	33,304	0.11
Stores	12,800	12,800	51,200	4	0.06	3.13	1,638,400	51,200	0.16	-	-
Total	1,036,800	1,036,800	4,147,200	4	4.73	7.07	1,638,400	3,863,820	12.35	33,304	0.11

L2 Cache

	Requests	Sectors	Sectors/Req	% Peak	Hit Rate	Bytes	Throughput	Sector Misses to Device	Sector Misses to System	Sector Misses to Peer
L1/TEX Load	1,900,338	3,812,814	2.01	19.30	97.31	122,010,0...	473,935,860,783.10	102,400	0	0
L1/TEX Store	25,600	51,200	2	0.26	100	1,638,400	6,364,201,367.31	0	0	0
L1/TEX Atomic ALU	0	0	0	0	0	0	0	0	0	0
L1/TEX Atomic CAS	0	0	0	0	0	0	0	0	0	0
L1/TEX Reduction	0	0	0	0	0	0	0	0	0	0
L1/TEX Total	1,929,653	3,864,448	2.00	19.56	97.35	123,662,3...	480,354,008,701.06	102,400	0	0

Device Memory

	Sectors	% Peak	Bytes	Throughput
Load	147,672	2.65	4,725,504	18,355,748,912.37
Store	2,456	0.04	78,592	305,282,784.34
Total	150,128	2.69	4,804,096	18,661,031,696.71

LAUNCH STATISTICS

Instructions distribution in program

Launch Statistics Analysis

- Detailed analysis of the resources of the GPU.
- It helps us understand the GPU workload and how shared memory and registers affect the occupancy.

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.

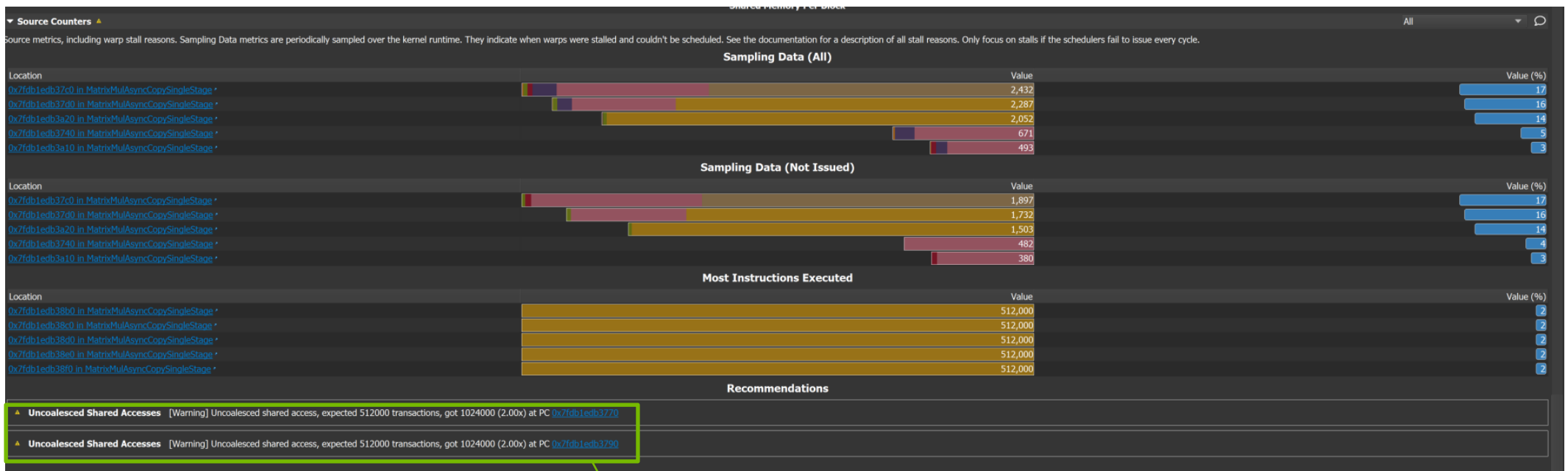
Grid Size	1,600	Registers Per Thread [register/thread]	35
Block Size	256	Static Shared Memory Per Block [Kbyte/block]	2.05
Threads [thread]	409,600	Dynamic Shared Memory Per Block [byte/block]	0
Waves Per SM	3.70	Driver Shared Memory Per Block [Kbyte/block]	1.02
		Shared Memory Configuration Size [Kbyte]	32.77

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]	6
Theoretical Active Warps per SM [warp]	48	Block Limit Shared Mem [block]	33
Achieved Occupancy [%]	90.47	Block Limit Warps [block]	6
Achieved Active Warps Per SM [warp]	43.42	Block Limit SM [block]	16

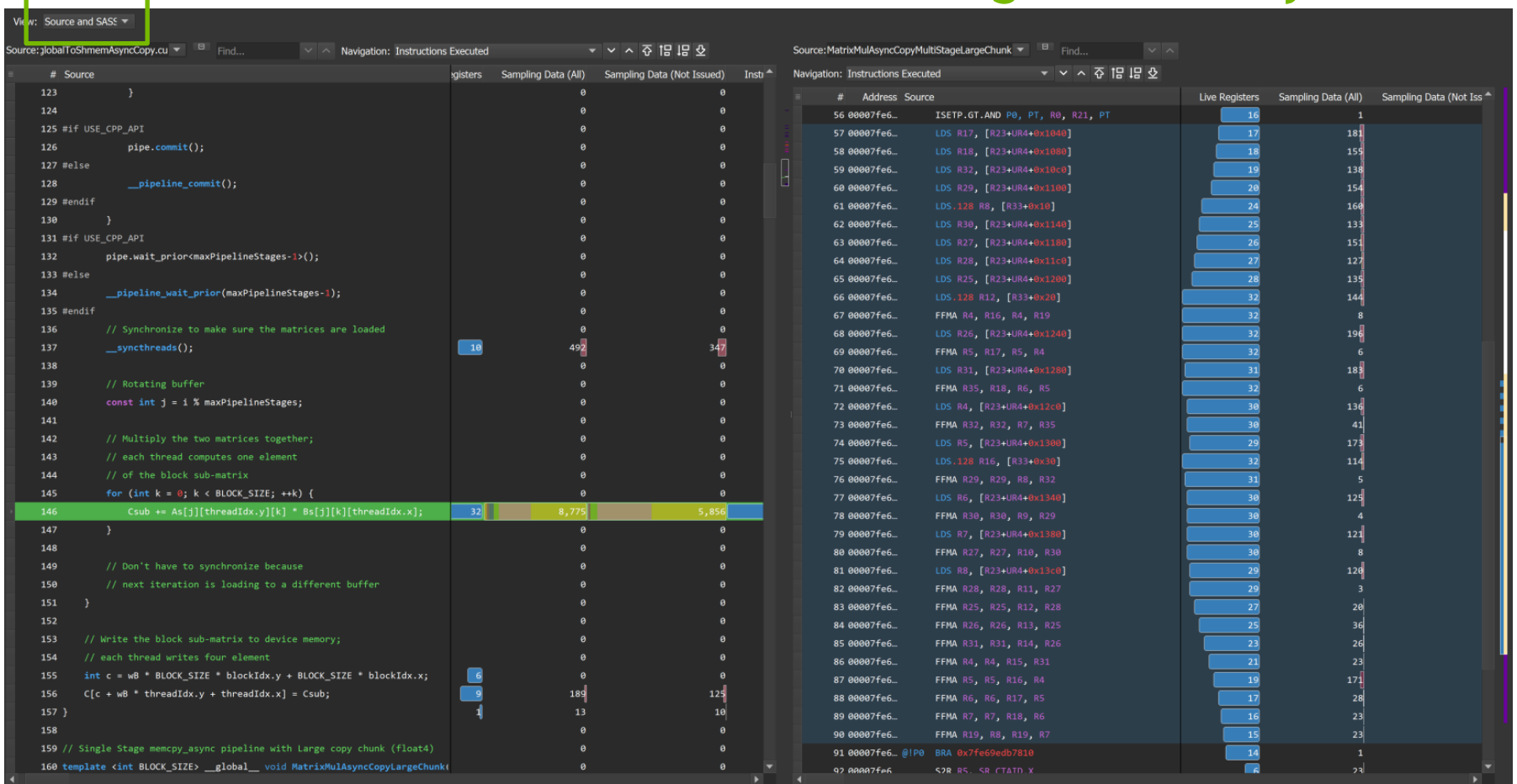
SOURCE CONTERS STATISTICS



Indicates inefficient memory access instructions, including shared memory and global memory access.

SOURCE PAGE

Understand the code generated by NVCC



Add -lineinfo to NVCC Flag

CUDA C/C++ <-> PTX
CUDA C/C++ <-> SASS
PTX <-> SASS

UNDERSTAND STALL REASON

Sections

Warp State Statistics (case 2)

major reasons cause stall:

- an instruction fetch,
- a memory dependency (result of memory instruction),
- an execution dependency (result of previous instruction),
- a pipeline is busy,
- a synchronization barrier.

UNDERSTAND STALL REASON

Typical Stall Reason

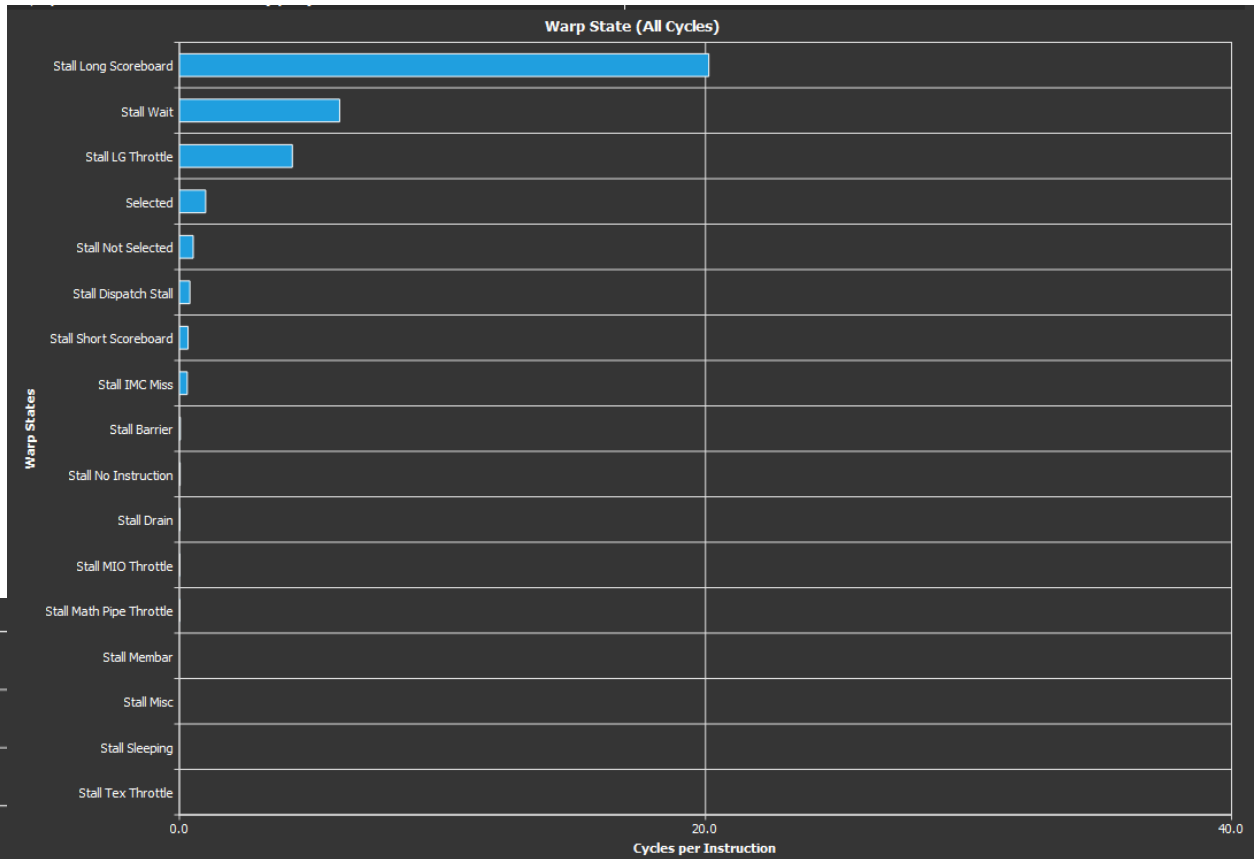
- Common Stall Reasons:
- Long Scoreboard → L1Tex (Global, Local, Suface, Tex) result dependency.
- Short Scoreboard → Shared memory result dependency or frequent MUFU or Dynamic branching.
- LG Throttle → Waiting for the L1 instruction queue for local and global (LG) memory operations to be not full. Stall occurs only when executing local or global memory instructions extremely frequently.
- MIO Throttle → Was stalled waiting for the MIO (memory input/output) instruction queue to be not full. Stall occurs when executing LDS, MUFU or Dynamic Branching extremely frequently.
- Math Pipe Throttle → Was stalled waiting for the execution pipe to be available.

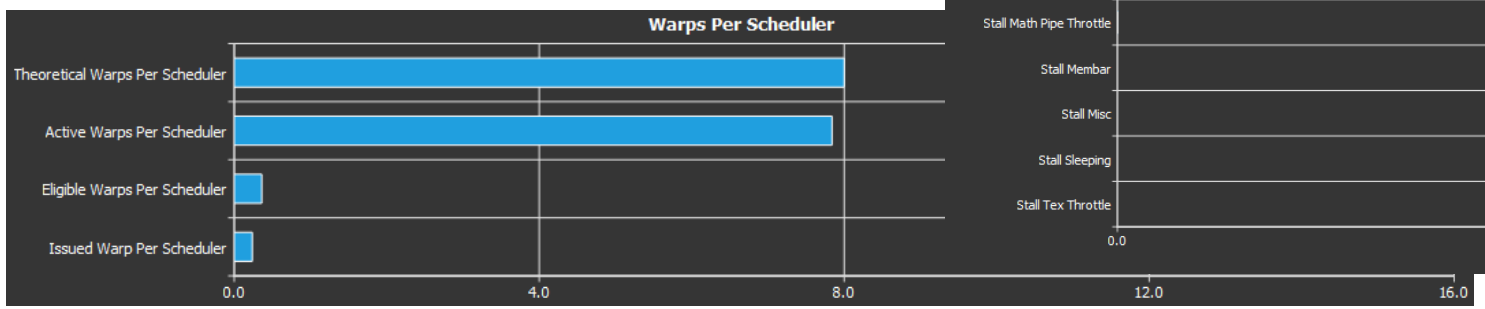
UNDERSTAND STALL REASON

Long Scoreboard

Long score board -> global memory latency

```
__global__ void stall_reason_lsb(int * dramptr){

    int tid = threadIdx.x;
    int laneid = tid % 32;
    dramptr[laneid] = laneid;
    __syncthreads();

    int idx = laneid;
#pragma unroll
    for(int i = 0; i < 1000; i++)
        idx = dramptr[idx];
    dramptr[idx] = idx;


}
```



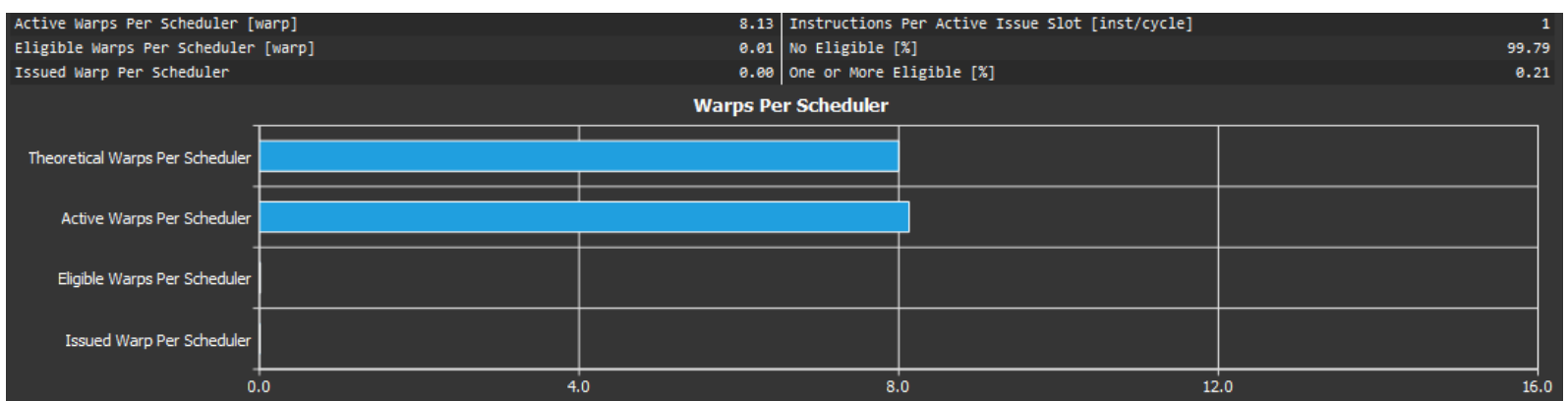


UNDERSTAND STALL REASON

LG Throttle

LG Throttle -> Too many & frequent global access

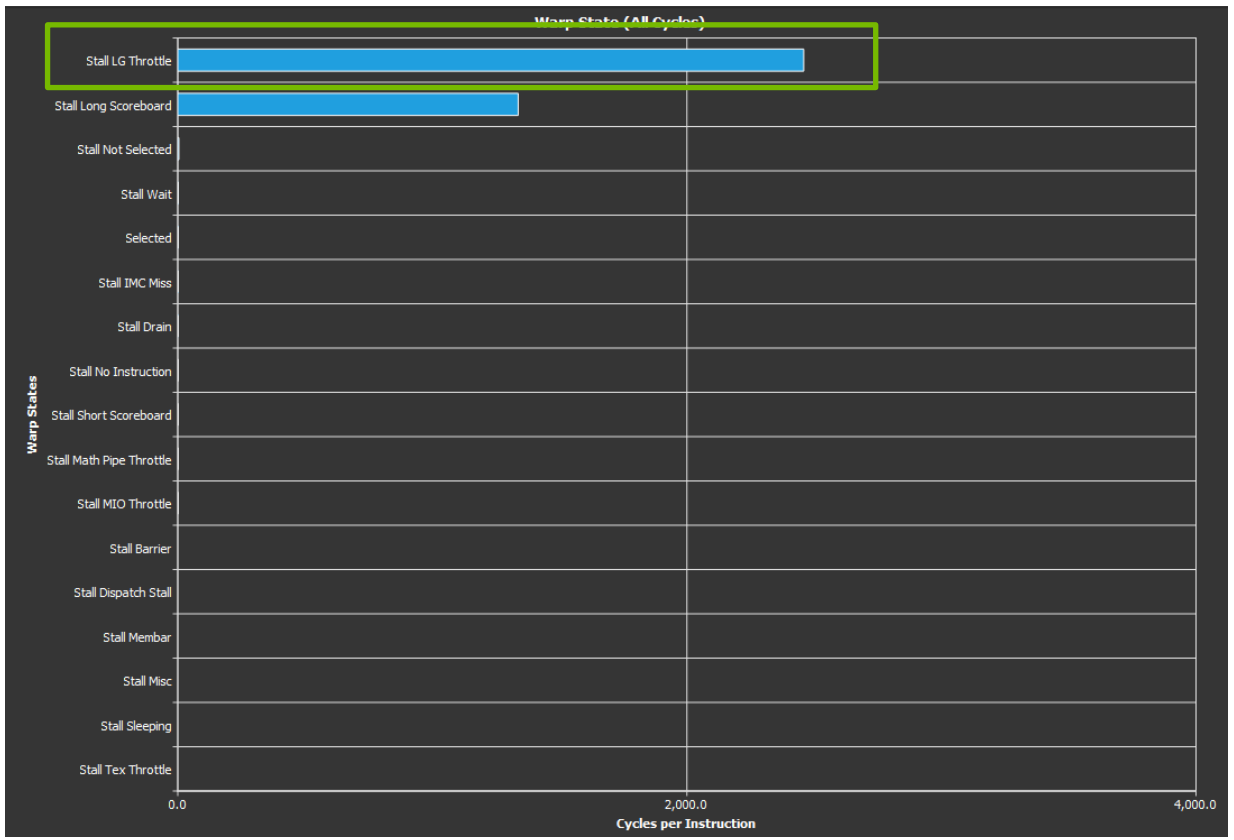
```
__global__ void stall_reason_lg_worst(int8_t * dramptr, int8_t * dramptr2){

    int tid = threadIdx.x;
    int offset = tid * 1000;
#pragma unroll
    for(int i = 0; i < 2000; i++)
        dramptr2[offset + i] = dramptr[offset + i];

}
```



UNDERSTAND STALL REASON

LG Throttle

LG Throttle -> Too many & frequent global access



Un**coalesced** global memory access caused a large number of requests to block the queue.

UNDERSTAND STALL REASON

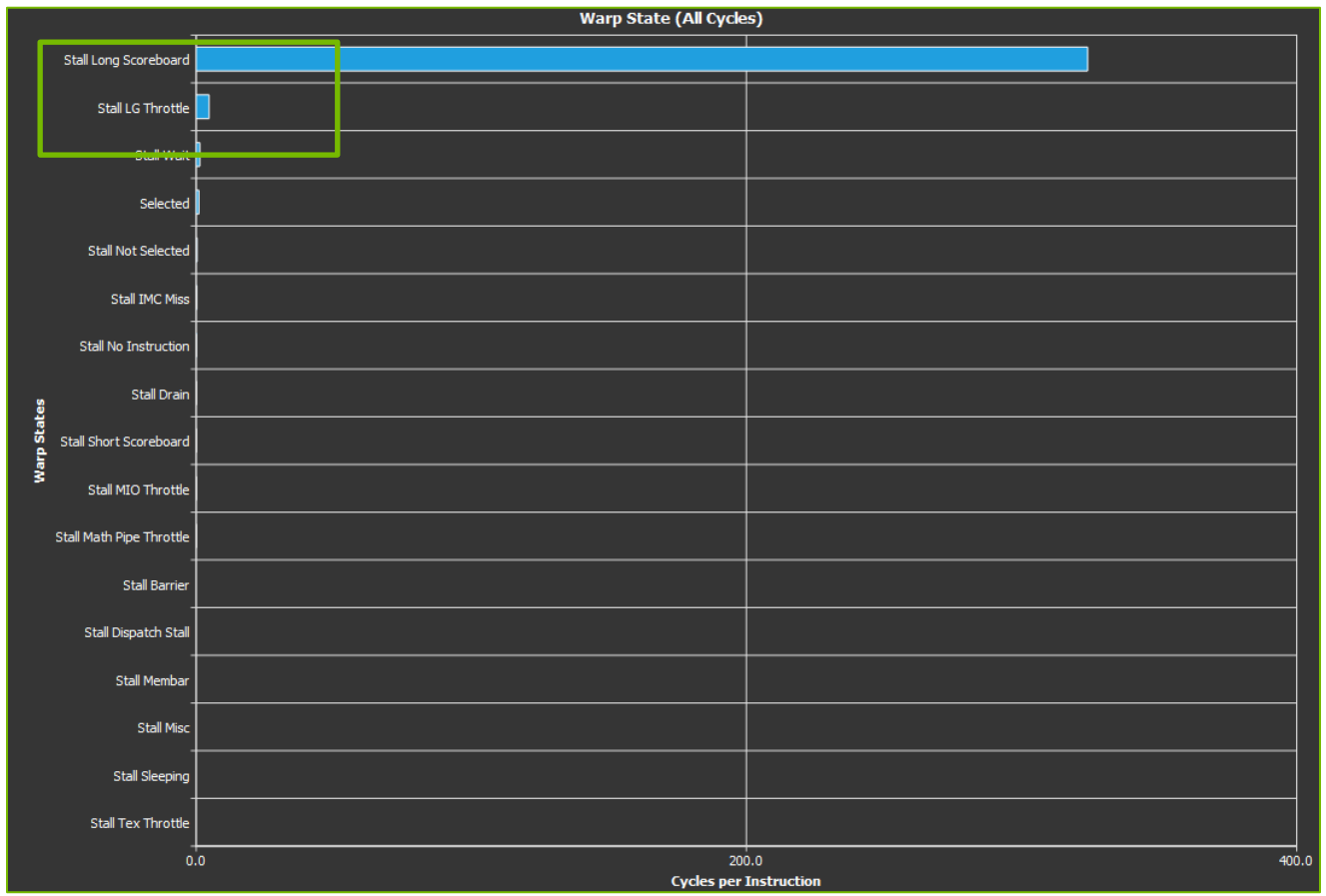
LG Throttle

Coalesced global memory access reduce a large number of requests.

```
__global__ void stall_reason_lg_worse(int8_t * dramptr, int8_t * dramptr2){

    int tid = threadIdx.x;
    int total_thread = 1024;//blockDim.x;
#pragma unroll
    for(int i = 0; i < 2000; i++)
        dramptr2[i * total_thread + tid] = dramptr[i * total_thread + tid];

}
```

UNDERSTAND STALL REASON

LG Throttle



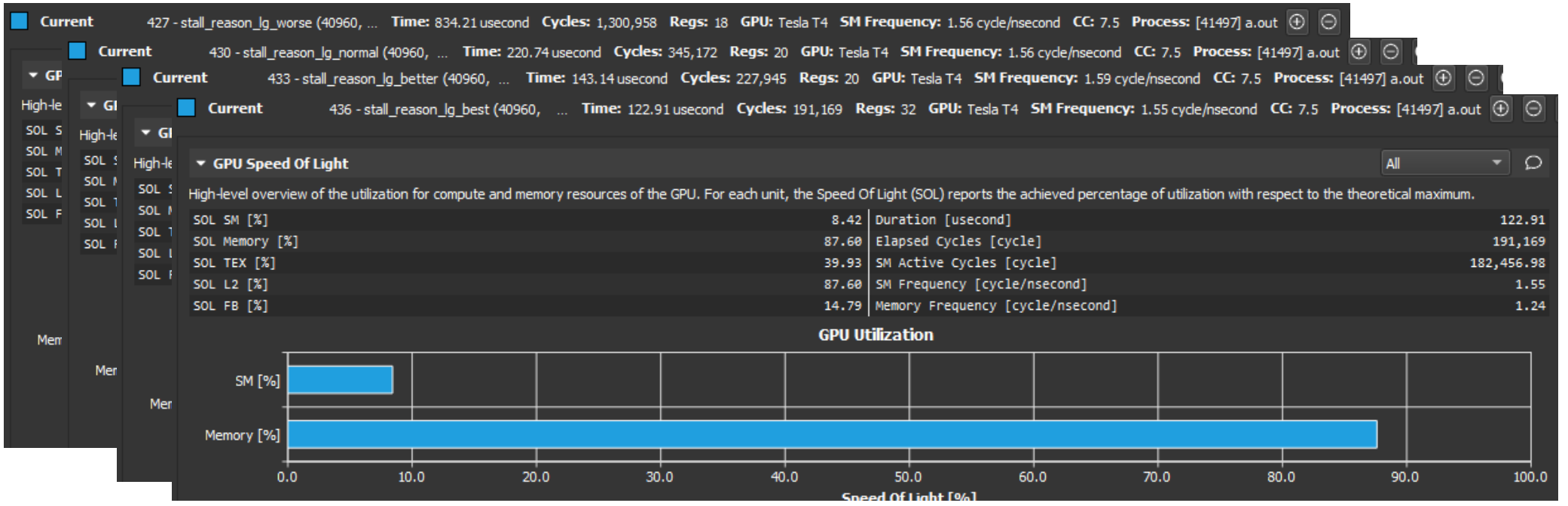
UNDERSTAND STALL REASON

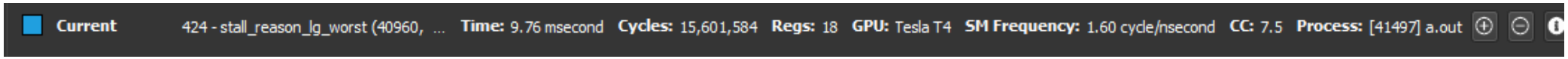
LG Throttle

```
__global__ void stall_reason_lg_worse(int8_t * dramptr, int8_t * dramptr2){

    int tid = threadIdx.x;
    int total_thread = 1024;//blockDim.x;
#pragma unroll
    for(int i = 0; i < 2000; i++)
        dramptr2[i * total_thread + tid] = dramptr[i * total_thread + tid];

}
__global__ void stall_reason_lg_normal(int8_t * dramptr, int8_t * dramptr2){

    int tid = threadIdx.x;
    int total_thread = 1024;//blockDim.x;
    int * ptr = (int*)dramptr;
    int * ptr2 = (int*)dramptr2;
#pragma unroll
    for(int i = 0; i < 500; i++)
        ptr2[i * total_thread + tid] = ptr[i * total_thread + tid];

}
__global__ void stall_reason_lg_best(int8_t * dramptr, int8_t * dramptr2){

    int tid = threadIdx.x;
    int total_thread = 1024;//blockDim.x;
    int4 * ptr = (int4*)dramptr;
    int4 * ptr2 = (int4*)dramptr2;
#pragma unroll
    for(int i = 0; i < 125; i++)
        ptr2[i * total_thread + tid] = ptr[i * total_thread + tid];
```

UNDERSTAND STALL REASON

LG Throttle





UNDERSTAND STALL REASON

Short Scoreboard

Short score board -> shared memory latency

```
__global__ void stall_reason_ssb(int* dramptr){

    __shared__ int smm[32];
    int tid = threadIdx.x;
    int laneid = tid % 32;
    smm[laneid] = laneid;
    __syncthreads();

    int idx = laneid;
#pragma unroll
    for(int i = 0; i < 1000; i++)
        idx = smm[idx];
    dramptr[idx] = idx;
}


void stall_reason_ssb_launcher(int* dramptr){
    stall_reason_ssb<<<40, 1024>>>(dramptr);
}
```

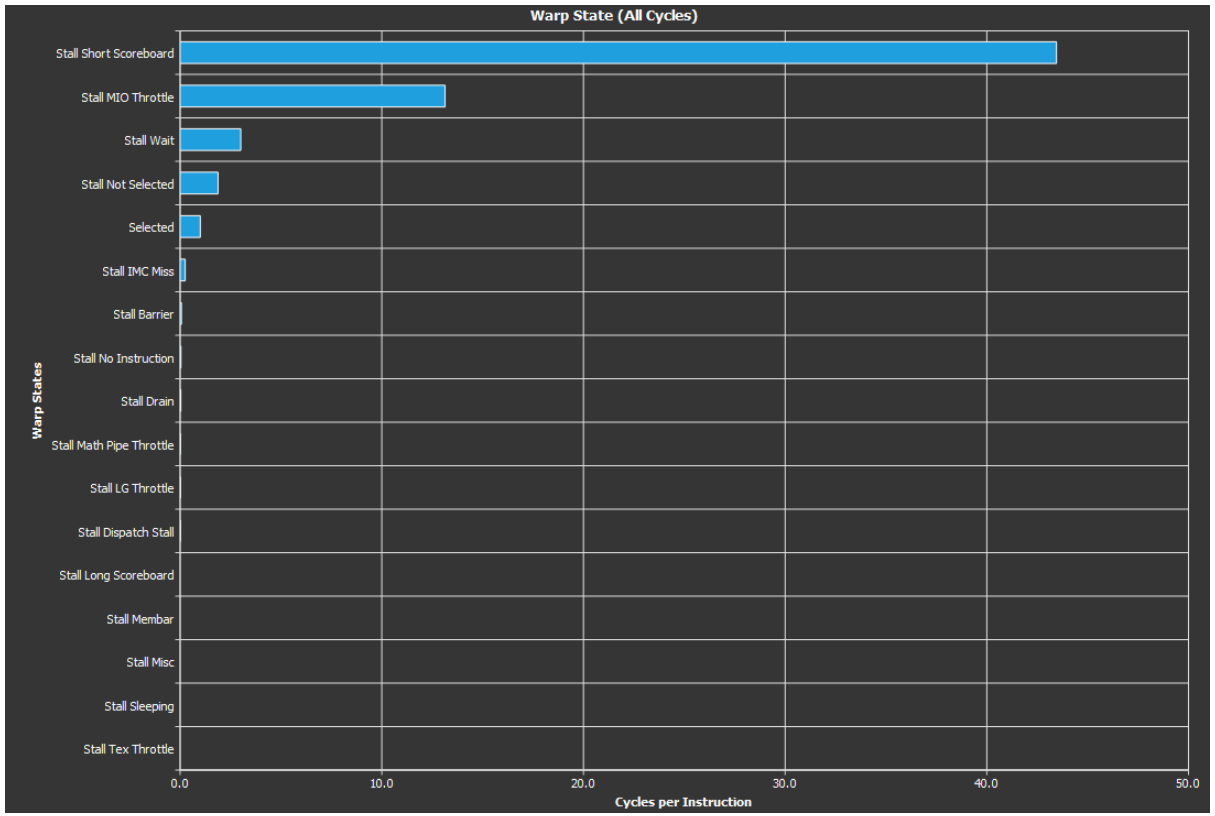
	Instructions	Requests	% Peak	Bank Conflicts
Shared Load	1,280,000	1,280,000	49.56	0
Shared Store	1,280	1,280	0.05	0
Shared Atomic	0	-	-	-
Total	1,281,280	1,281,280	49.61	0

Instructions = total_threads / WARP_SZ * instructions_num

UNDERSTAND STALL REASON

Short Scoreboard

Short scoreboard -> shared memory latency



UNDERSTAND STALL REASON

MIO Throttle

MIO Throttle -> extreme utilization of the MIO pipelines. Egs: shared memory, special math

```
__global__ void stall_reason_mio_bad(int* dramptr){

    __shared__ int smm[32][32];
    __shared__ int smm2[32][32];
    int tid = threadIdx.x;
    int laneid = tid % 32;
#pragma unroll
    for(int i = 0; i < 32; i++)
        smm2[laneid][i] = smm[laneid][i];

    __syncthreads();

}
```

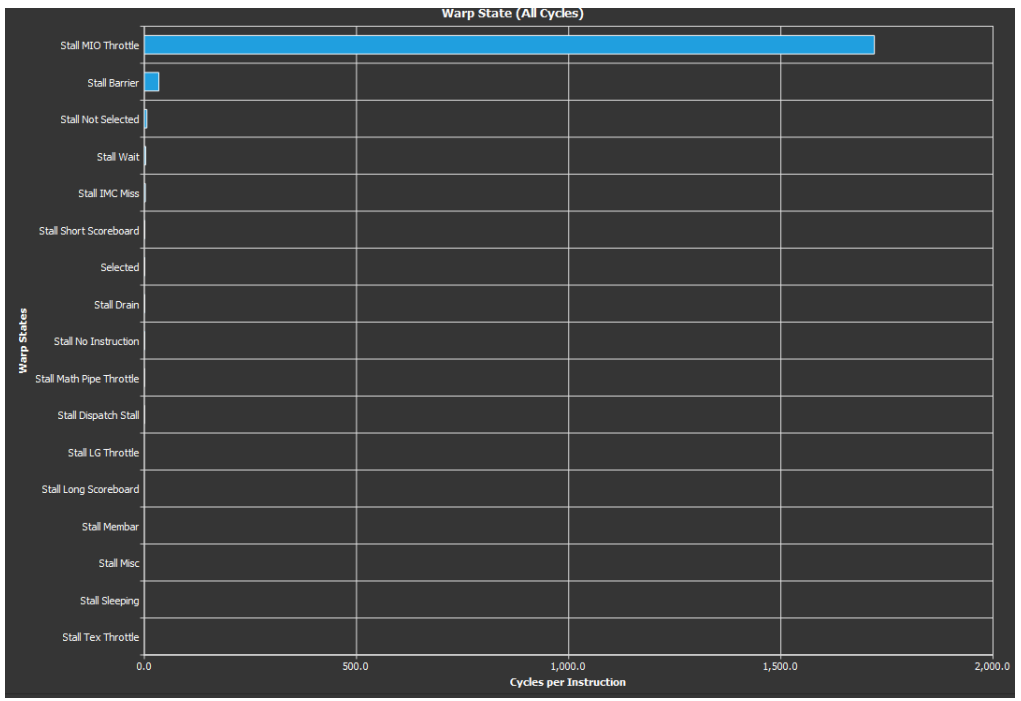
```
__global__ void stall_reason_mio_good(int* dramptr){

    __shared__ int smm[32][32];
    __shared__ int smm2[32][32];
    int tid = threadIdx.x;
    int laneid = tid % 32;
#pragma unroll
    for(int i = 0; i < 32; i++)
        smm2[i][laneid] = smm[i][laneid];

    __syncthreads();

}
```

```
void stall_reason_mio_launcher(int* dramptr){
    stall_reason_mio_bad<<<40,1024>>>(dramptr);
    stall_reason_mio_good<<<40,1024>>>(dramptr);

}
```

UNDERSTAND STALL REASON

MIO Throttle

MIO Bad Func



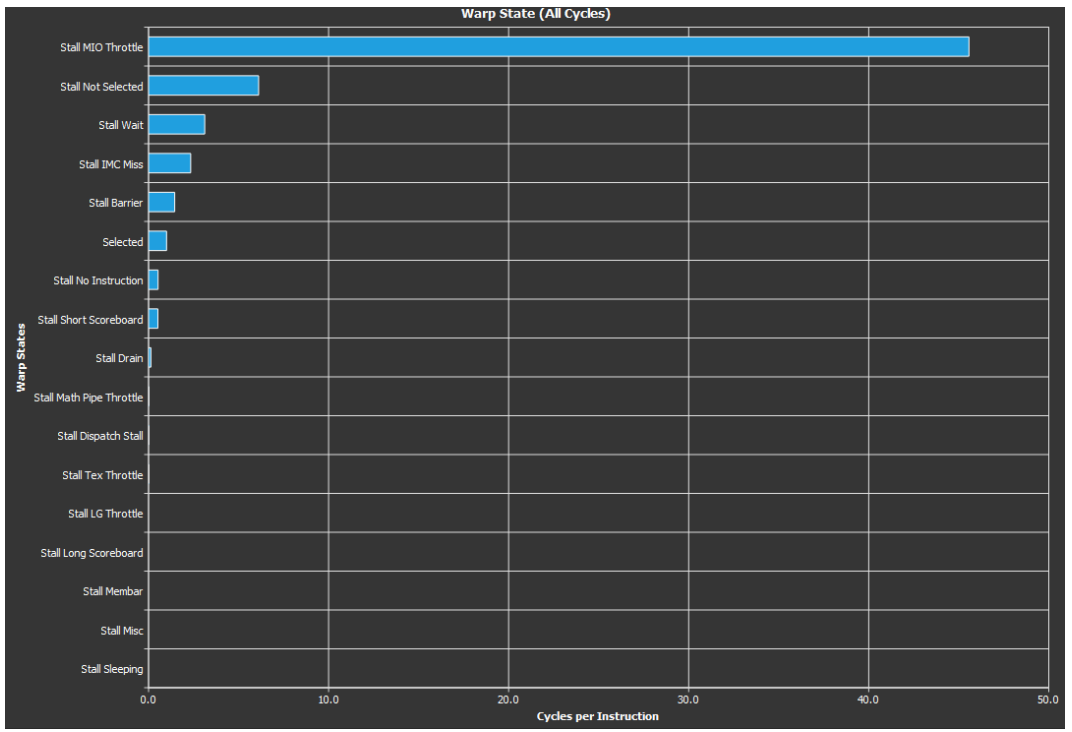
	Instructions	Requests	% Peak	Bank Conflicts
Shared Load	40,960	1,310,720	24.94	1,269,760
Shared Store	40,960	1,310,720	24.94	1,269,760
Shared Atomic	0	-	-	-
Total	81,920	2,621,440	49.87	2,539,520

Requests = 32(bank conflict) * instructions.

UNDERSTAND STALL REASON

MIO Throttle

MIO Good Func



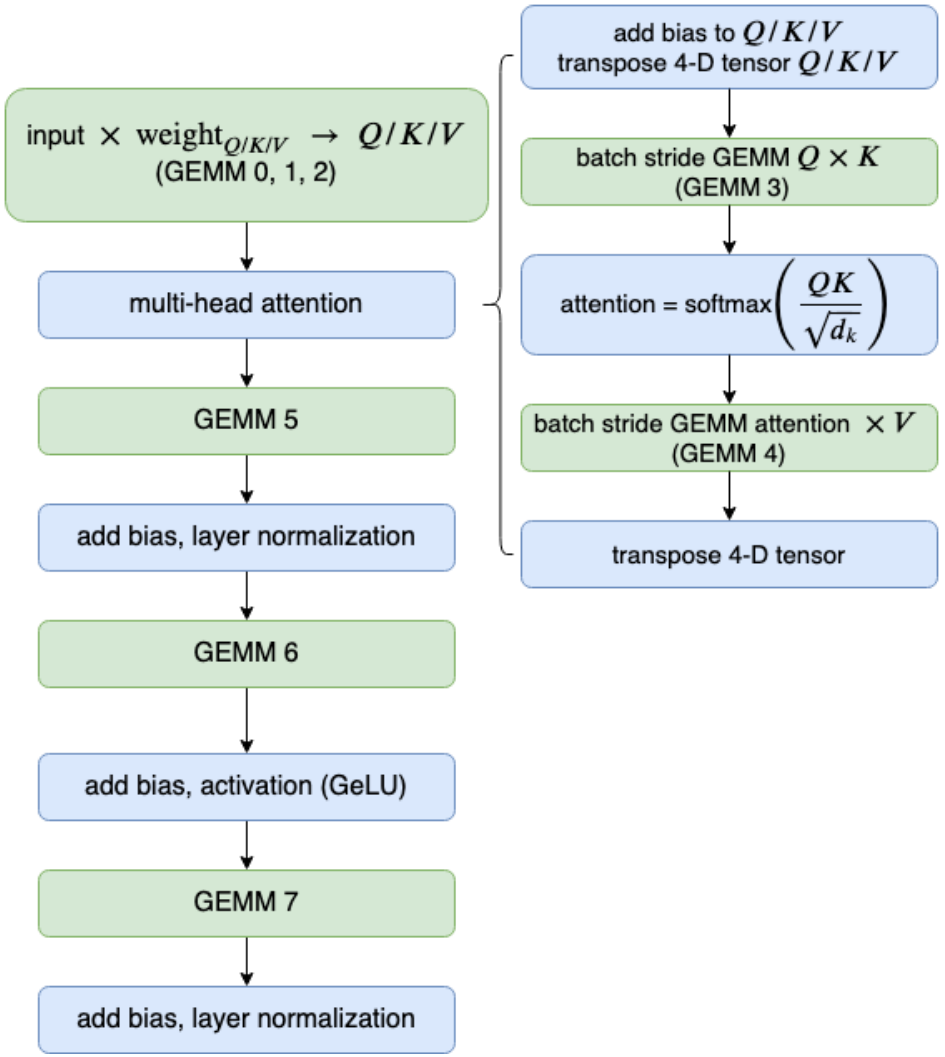
	Instructions	Requests	% Peak	Bank Conflicts
Shared Load	40,960	40,960	22.64	0
Shared Store	40,960	40,960	22.64	0
Shared Atomic	0	-	-	-
Total	81,920	81,920	45.29	0

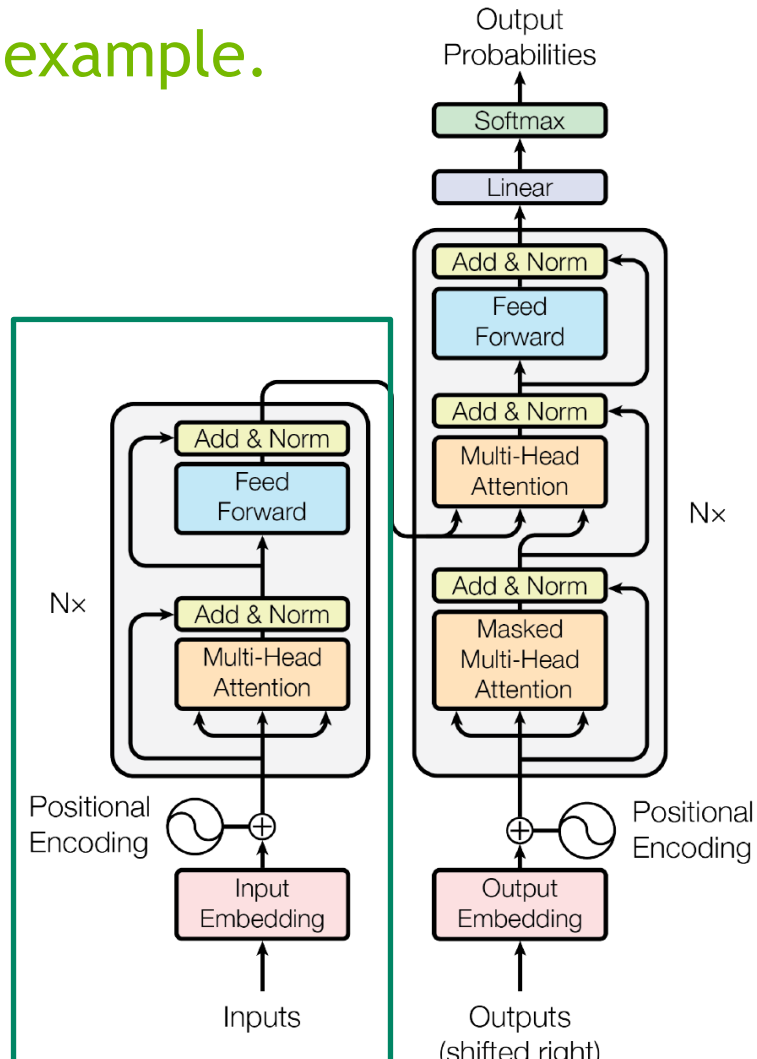
Requests = 1(no bank conflict) * instructions.

CASE STUDY 1: NSYS USAGE

FASTER TRANSFORMER

Take the encoder as an example.





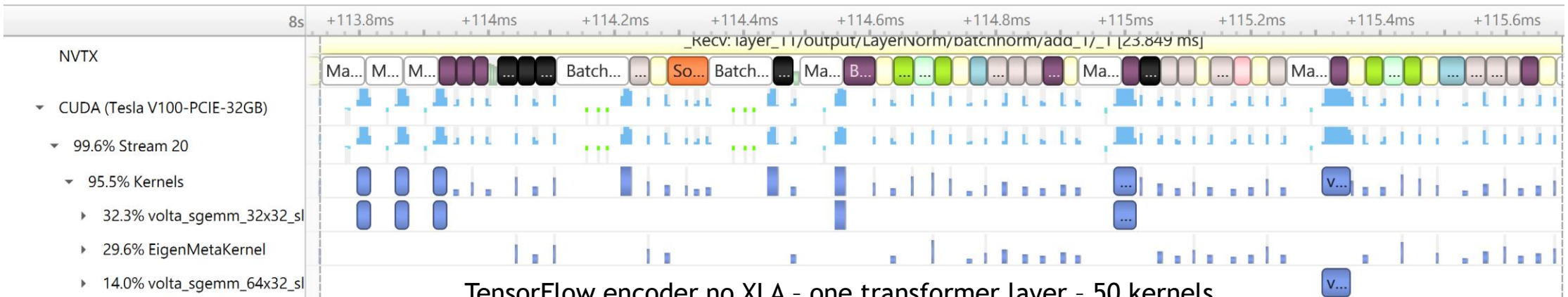
Transformer architecture [1]

OPTIMIZE FASTER TRANSFORMER

Where is the bottleneck for Encoder?

GPU is idle in many time

Reason: kernels are too small -> kernel launch bound



TensorFlow encoder no XLA – one transformer layer – 50 kernels
batch size 1, 12 heads, size per head 64, FP32

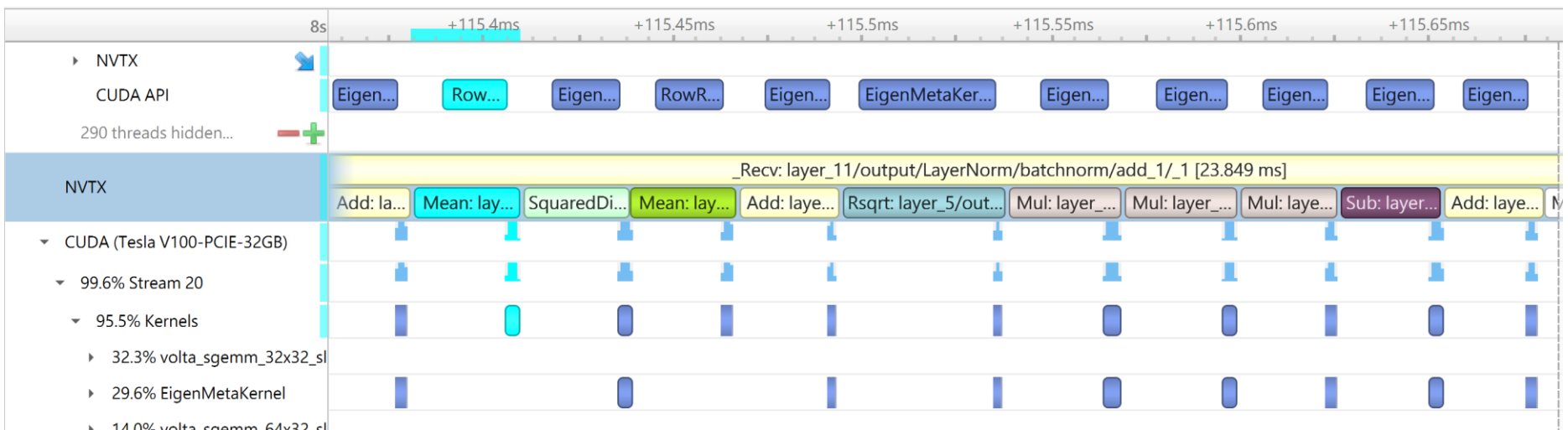
OPTIMIZE FASTER TRANSFORMER

Where is the bottleneck for Encoder?

GPU is idle in many time

Reason: kernels are too small

E.g., using 11 kernels (mean, add, …) to compute the LayerNorm



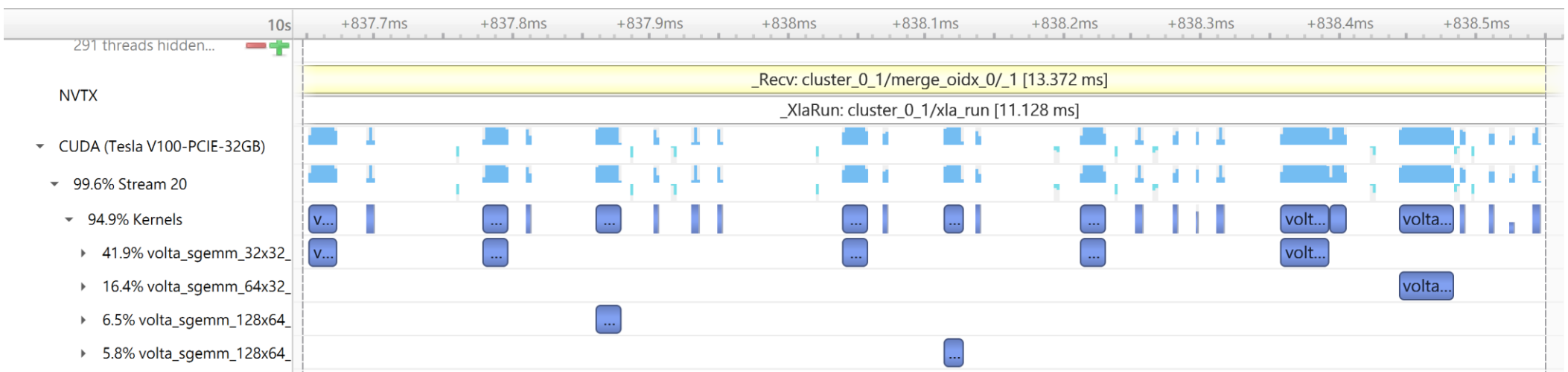
TensorFlow LayerNorm no XLA – 11 kernels
batch size 1, 12 heads, size per head 64, FP32

OPTIMIZE FASTER TRANSFORMER

Where is the bottleneck for Encoder?

A simple solution: Using TensorFlow XLA to fuse kernel automatically

Become better, but still many idle fraction

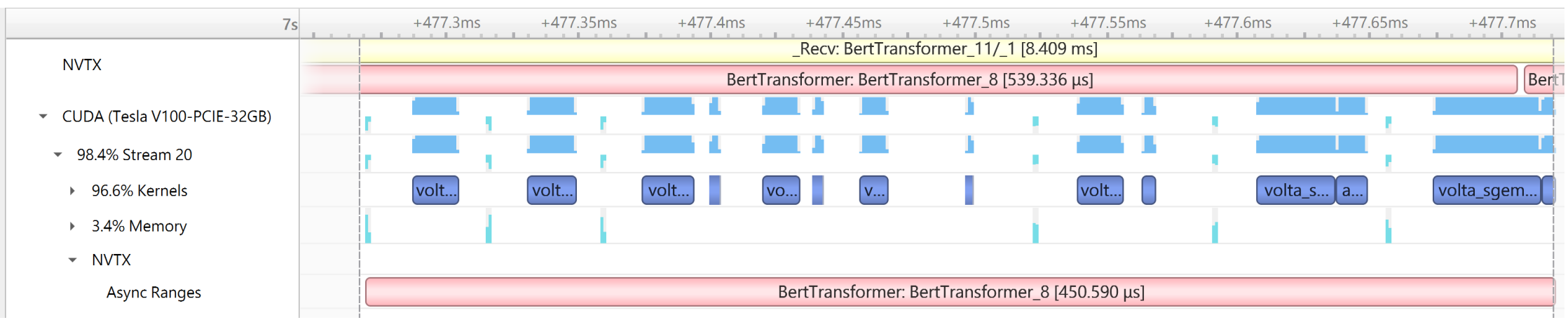


TensorFlow encoder with XLA – one transformer layer – 24 kernels
batch size 1, 12 heads, size per head 64, FP32

OPTIMIZE FASTER TRANSFORMER

Fused Encoder

Timeline after fusion



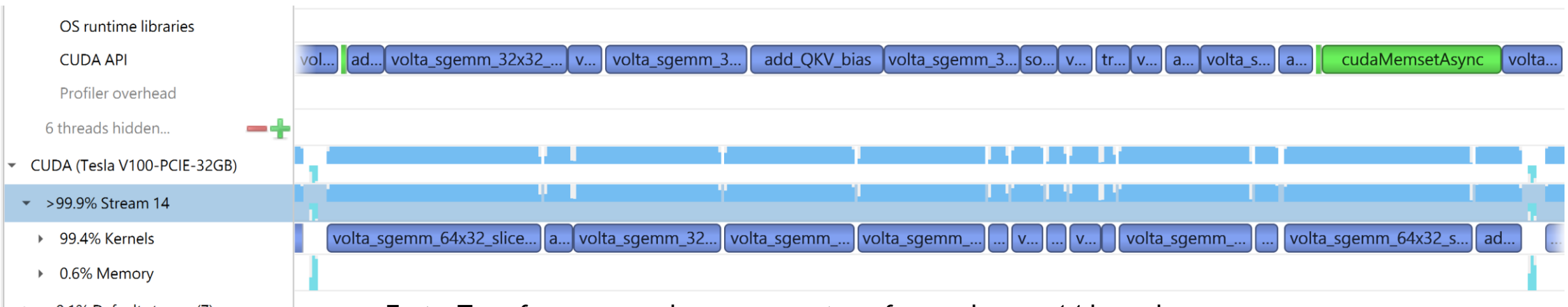
FasterTransformer encoder op – one transformer layer – 14 kernels
batch size 1, 12 heads, size per head 64, FP32

OPTIMIZE FASTER TRANSFORMER

Fused Encoder

Timeline after fusion

The pure cpp API can provide better performance



FasterTransformer encoder cpp – one transformer layer – 14 kernels
batch size 1, 12 heads, size per head 64, FP32

CASE STUDY 2: MATRIX TRANSPOSITION

MATRIX TRANSPOSITION

m = 8192 n = 4096. Some theoretical metrics

total bytes read = 8192 * 4096 * 4 = 134,217,728 B =128 MB

total bytes write = 8192 * 4096 * 4 = 134,217,728 B = 128 MB

total read transactions (32B) = 134,217,728 / 32 = 4,194,304

total write transactions (32B) = 134,217,728 / 32 = 4,194,304

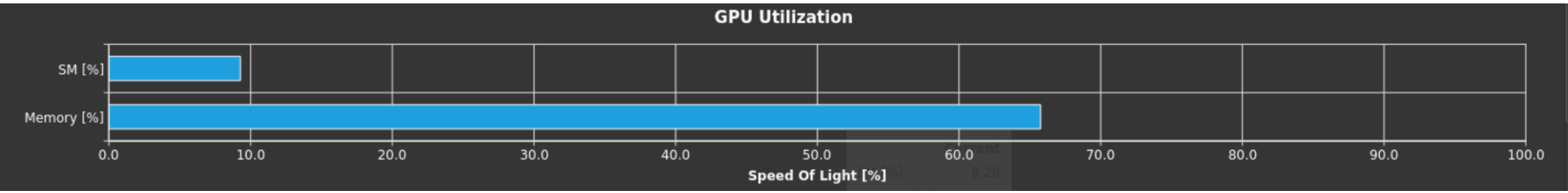
MATRIX TRANSPOSITION

Naïve Implementation

```
// m the number of rows of input matrix
// n the number of cols of input matrix
__global__ void transposeNative(float *input, float *output, int m, int n)
{
    int colID_input = threadIdx.x + blockDim.x*blockIdx.x;
    int rowID_input = threadIdx.y + blockDim.y*blockIdx.y;

    if (rowID_input < m && colID_input < n)
    {
        int index_input  = colID_input + rowID_input*n;
        int index_output = rowID_input + colID_input*m;

        output[index_output] = input[index_input];
    }
}
```

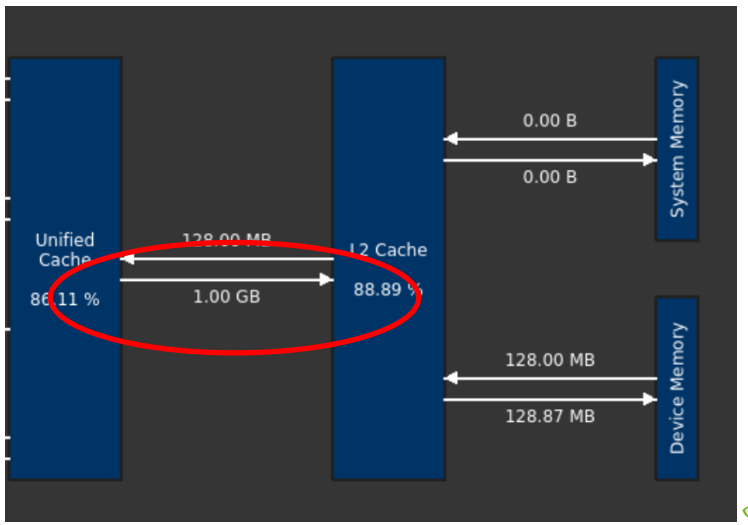
MATRIX TRANSPOSITION

Naïve Implementation



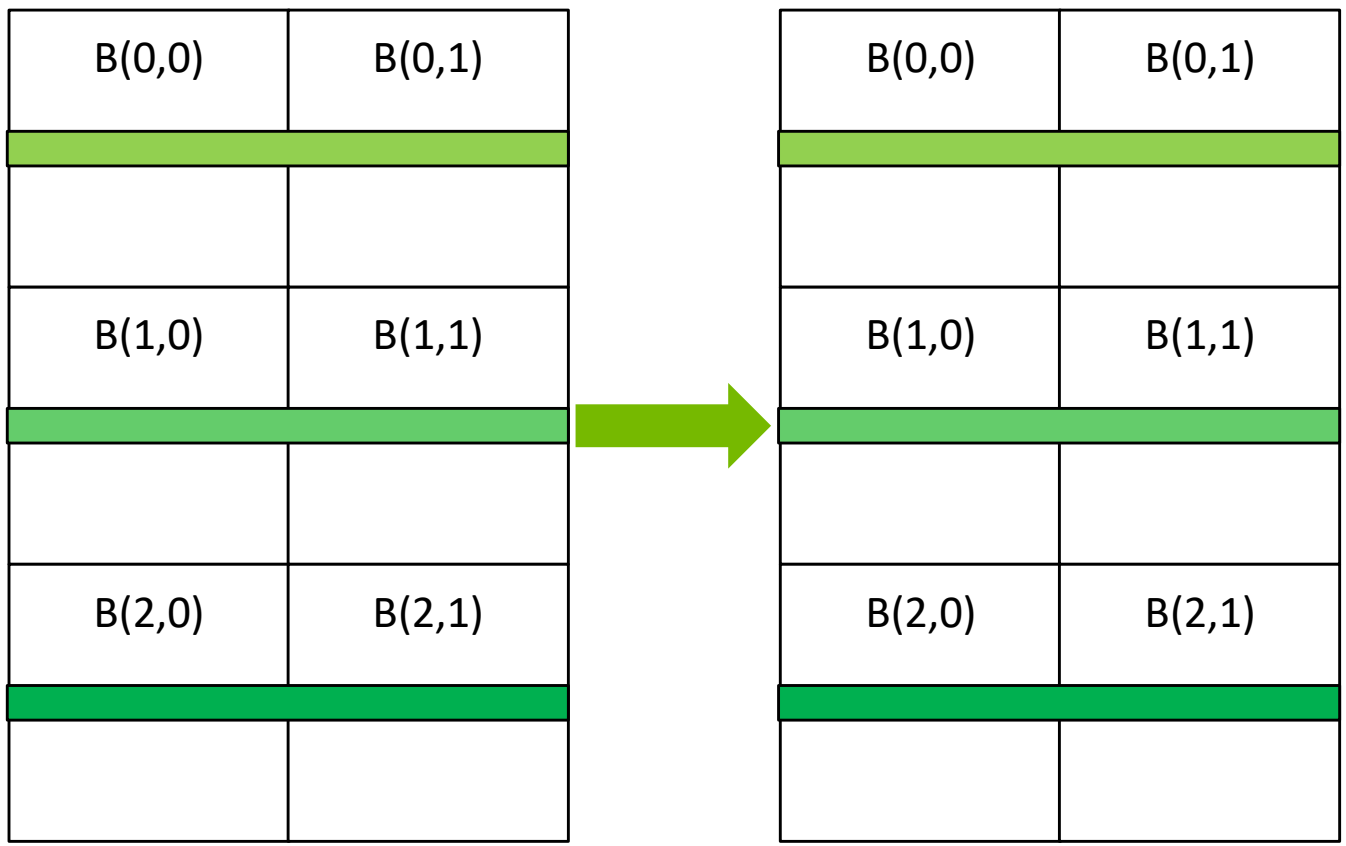
	TEX->L2 Requests (32B)	L2->Tex Returns (32B)
global load		4,194,394
global store	33,554,432	
time (us)	1890	

33,554,432 / 4,194,304 = 8, Utilization 12.5%



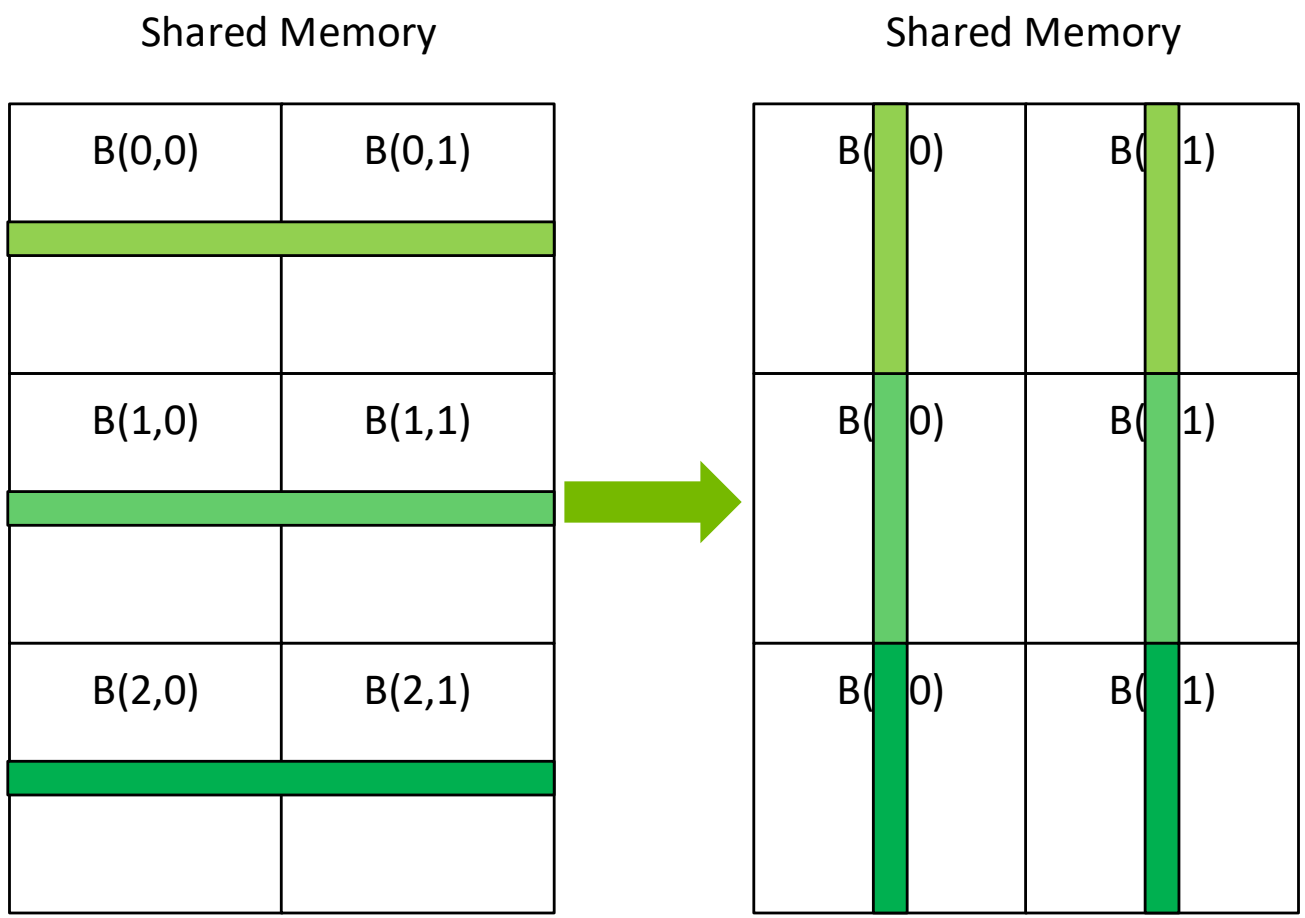
OPTIMIZATION WITH SHARED MEMORY

Load Data to Shared Memory



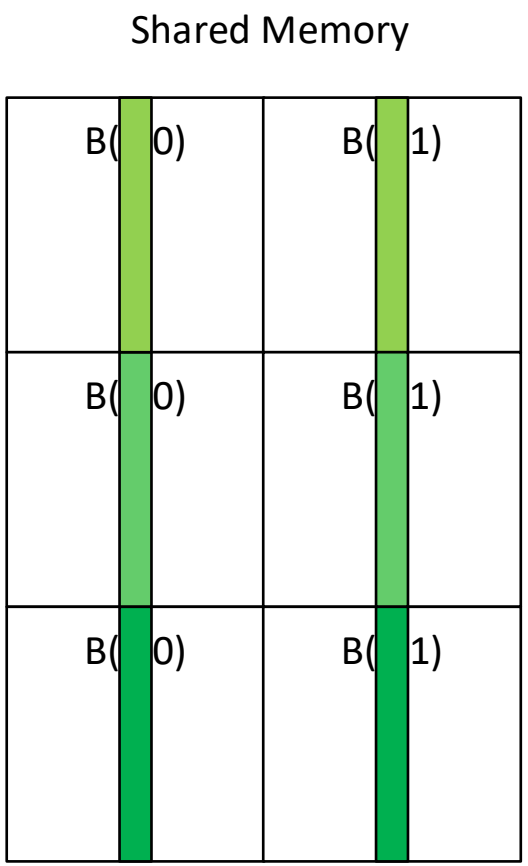
OPTIMIZATION WITH SHARED MEMORY

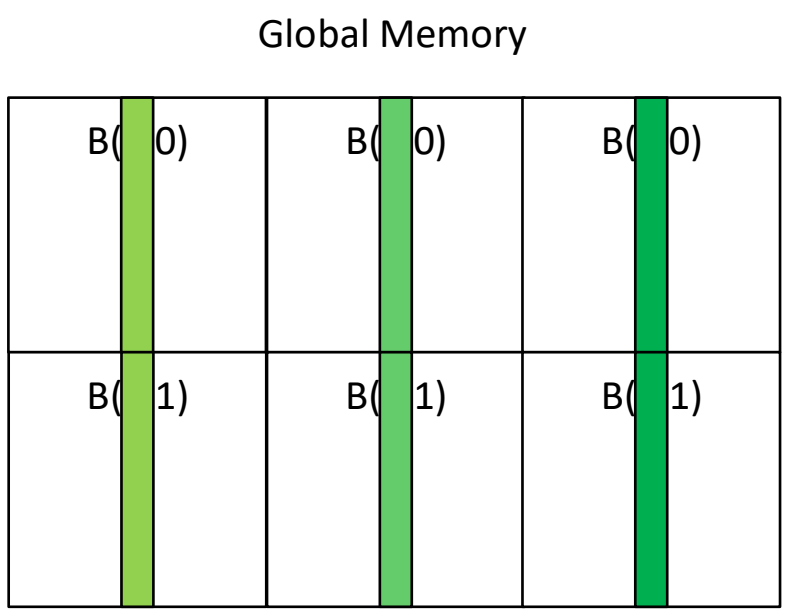
Local Transposition in Shared Memory



OPTIMIZATION WITH SHARED MEMORY

Block Transposition When Writing to Global Memory






```
dst_col = threadIdx.x + blockDim.y*blockIdx.y;

dst_row = threadIdx.y + blockDim.x*blockIdx.x;
```

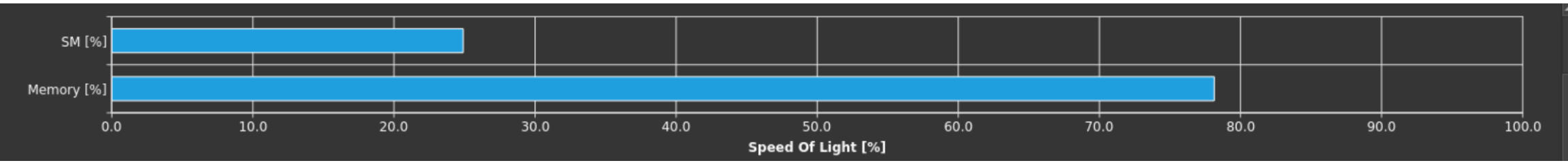
MATRIX TRANSPOSITION

Optimized Implementation

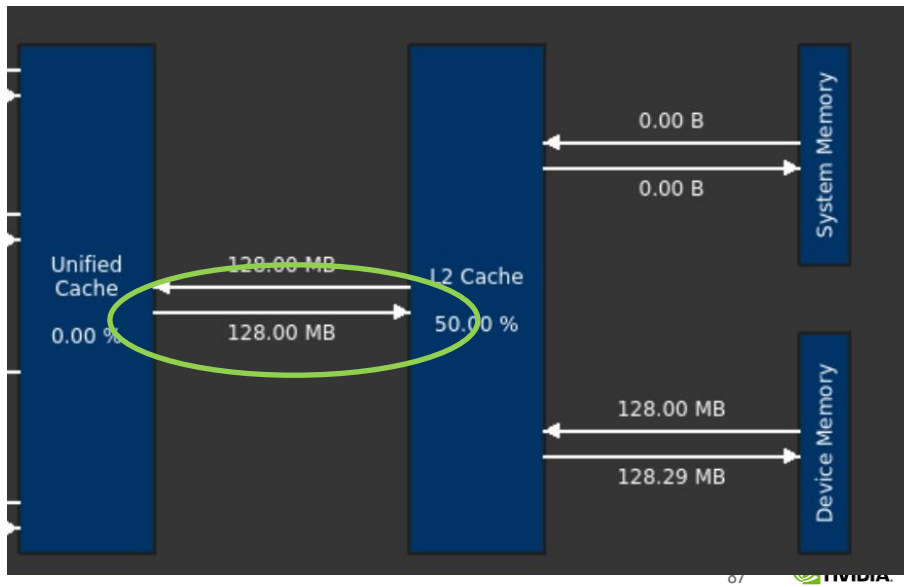
```
__global__ void transposeOptimized(float *input, float *output, int m, int n){
    int colID_input = threadIdx.x + blockDim.x*blockIdx.x;
    int rowID_input = threadIdx.y + blockDim.y*blockIdx.y;

    __shared__ float sdata[32][33];
    if (rowID_input < m && colID_input < n)
    {
        int index_input  = colID_input + rowID_input*n;
        sdata[threadIdx.y][threadIdx.x] = input[index_input];

        __syncthreads();

        int dst_col = threadIdx.x + blockIdx.y * blockDim.y;
        int dst_row = threadIdx.y + blockIdx.x * blockDim.x;
        output[dst_col + dst_row*m] = sdata[threadIdx.x][threadIdx.y];
    }
}
```

MATRIX TRANSPOSITION

Optimized Implementation



	TEX->L2 Requests (32B)	L2->Tex Returns (32B)
global load		4,194,394
global store	4,194,394	
time (us)	525	



SUMMARY

Nsight Systems is a system-level profiler

Nsight Compute is for kernel profiling tool

Basic knowledge of CUDA programming and GPU architecture is needed for profiling

Encourage developers to use Nsight Systems & Nsight Compute instead of NVVP & nvprof

Use profiler tools whenever possible to locate the optimization opportunities to avoid premature optimization

Use top-down approach; no need to jump directly into SASS code


NVIDIA