INTRODUCTION TO NVIDIA PROFILING TOOLS

Chandler Zhou, 20191219

AGENDA

Overview of Profilers

Nsight Systems

Nsight Compute

Case Studies

Summary

OVERVIEW OF PROFILERS

NVVP Visual Profiler

nvprof the command-line profiler

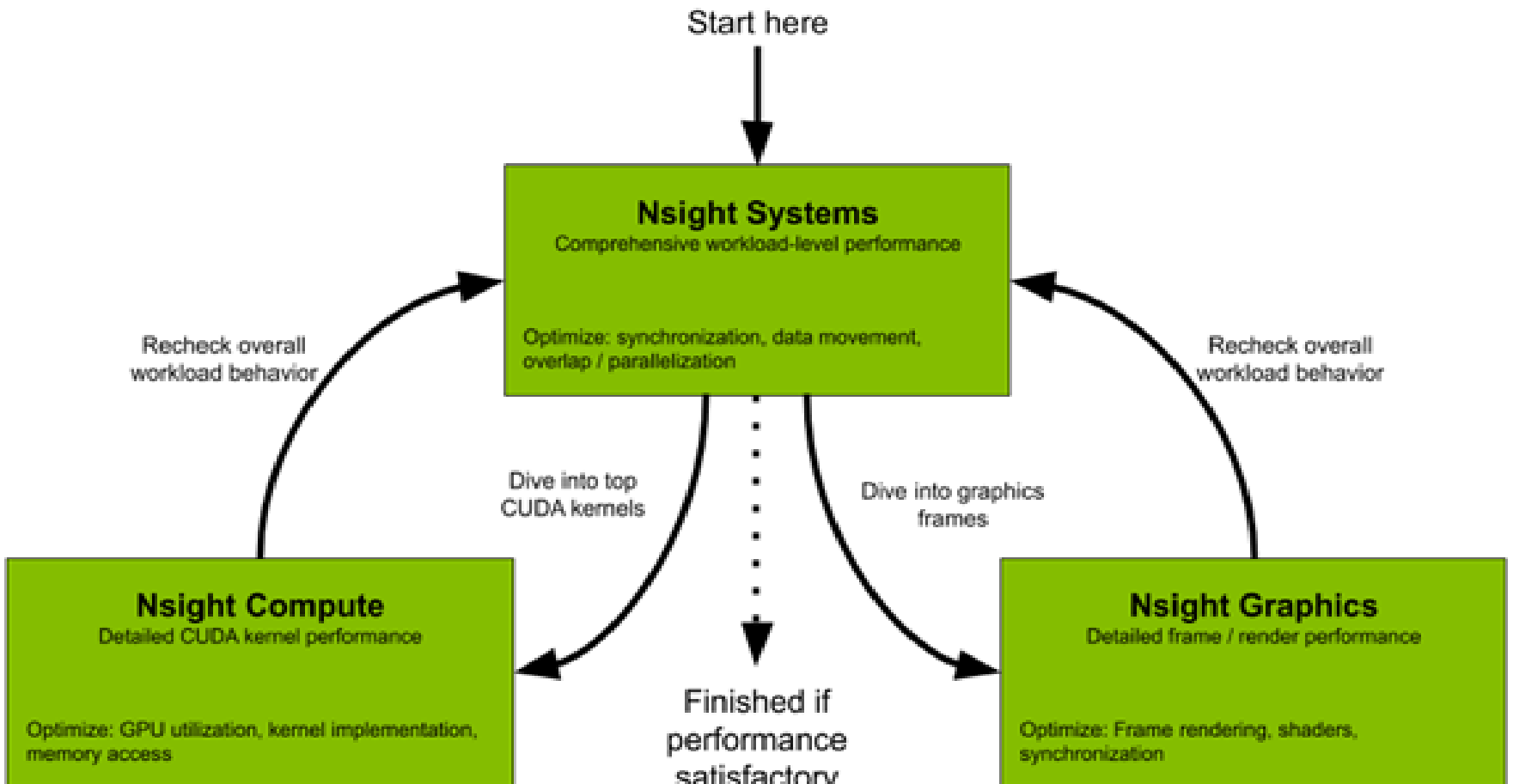
Nsight Systems A system-wide performance analysis tool

Nsight Compute An interactive kernel profiler for CUDA applications

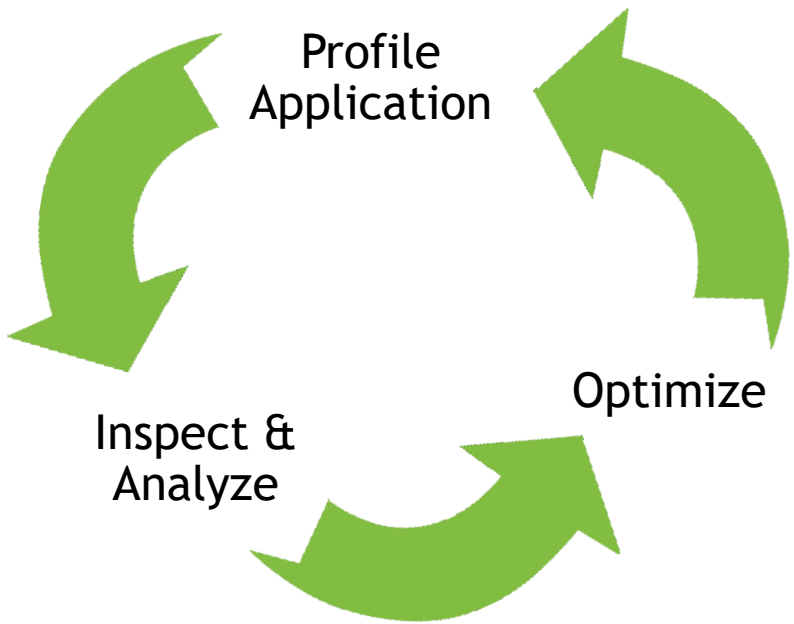
Note that Visual Profiler and nvprof will be **deprecated** in a future CUDA release

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

NSIGHT SYSTEMS

Key Features

Compute

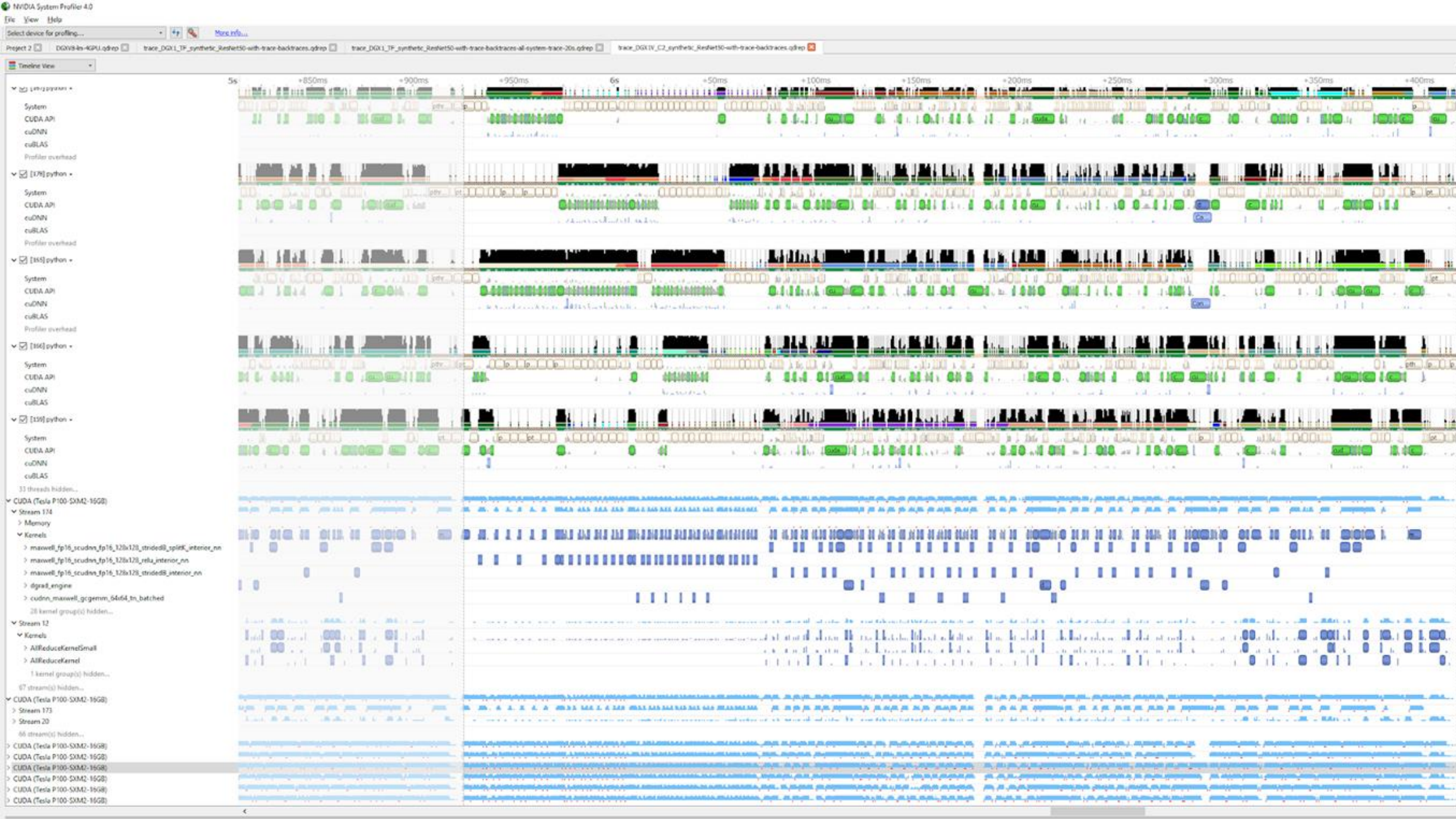
- CUDA API. Kernel launch and execution correlation
- Libraries: cuBLAS, cuDNN, TensorRT
- OpenACC

Graphics

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

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

User annotations API (NVTX)


NVIDIA System Profiler 4.0
File View Help
Select device for profiling...
More info...
Project 2
DGX1-8x-4GPU.qdrep
trace_DGX1_TF_synthetic_ResNet50-with-trace-backtraces.qdrep
trace_DGX1_TF_synthetic_ResNet50-with-trace-backtraces-all-system-trace-20s.qdrep
trace_DGX1V_C2_synthetic_ResNet50-with-trace-backtraces.qdrep
Timeline View
5s
+850ms
+900ms
+950ms
6s
+50ms
+100ms
+150ms
+200ms
+250ms
+300ms
+350ms
+400ms
System
CUDA API
cuDNN
cuBLAS
Profiler overhead
[178] python
[165] python
[166] python
[159] python
33 threads hidden...
CUDA (Tesla P100-SXM2-16GB)
Stream 174
Memory
Kernels
maxwell_fp16_scudnn_fp16_128x128_strided8_splitK_interior_nn
maxwell_fp16_scudnn_fp16_128x128_relu_interior_nn
maxwell_fp16_scudnn_fp16_128x128_strided8_interior_nn
dgrad_engine
cudnn_maxwell_gcgemm_64x64_tn_batched
28 kernel group(s) hidden...
Stream 12
Kernels
AllReduceKernelSmall
AllReduceKernel
1 kernel group(s) hidden...
67 stream(s) hidden...
CUDA (Tesla P100-SXM2-16GB)
Stream 173
Stream 20
66 stream(s) hidden...
CUDA (Tesla P100-SXM2-16GB)
CUDA (Tesla P100-SXM2-16GB)
CUDA (Tesla P100-SXM2-16GB)
CUDA (Tesla P100-SXM2-16GB)
CUDA (Tesla P100-SXM2-16GB)
CUDA (Tesla P100-SXM2-16GB)

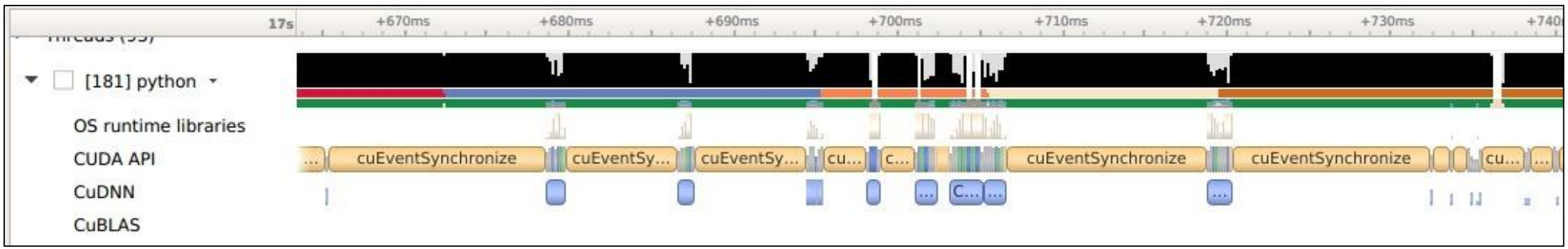
```
Detected compositing window manager.
Reverting to full screen capture at every frame.
To disable this check run with --no-wm-check
(though that is not advised, since it will probably produce faulty results).

Initializing...
Buffer size adjusted to 4096 from 4096 frames.
Opened PCM device default
Recording on device default is set to:
1 channels at 22050Hz
Capturing!
X Error: BadAccess (attempt to access private resource denied)
Bad Access on XGrabKey.
Shortcut already assigned.
X Error: BadAccess (attempt to access private resource denied)
Bad Access on XGrabKey.
Shortcut already assigned.
X Error: BadAccess (attempt to access private resource denied)
Bad Access on XGrabKey.
Shortcut already assigned.
X Error: BadAccess (attempt to access private resource denied)
Bad Access on XGrabKey.
Shortcut already assigned.

^C
*****************************************

Cached 5 MB, from 1774 MB that were received.
Average cache compression ratio: 99.7 %

*****************************************
Saved 226 frames in a total of 225 requests
Shutting down.....
STATE:ENCODING
Encoding started!
This may take several minutes.
Pressing Ctrl-C will cancel the procedure (resuming will not be possible, but
any portion of the video, which is already encoded won't be deleted).
Please wait...
Output file: out.ogv
[100%]
Encoding finished!
Wait a moment please...

Done.
Written 1883381 bytes
(1716075 of which were video data and 167306 audio data)

Cleanning up cache...
Done!!!
Goodbye!
chandler@nvidia:~$ ls
chandlerz_nvidia_8371_p4BJ        demo       intern      MLLibraries              NVIDIA_CUDA-8.0_Samples  output    taobao_model_with_cudaGraph    tools      网上提交材料
cuda_10.2.89_440.33.01_linux.run  Desktop    JDCloud     Music                    NVIDIA_CUDA-9.0_Samples  out.txt   Templates                      wenx
cudaLaunchOverhead                detect     libvpx      mylib                    nvvp_workspace           Paper     TENSORRT_INT8_BATCH_DIRECTORY  work
cuda-workspace                    Documents  log         NVIDIA_CUDA-10.0_Samples openSource              Pictures  test                           workspace
data                              Downloads  miniconda3  NVIDIA_CUDA-10.2_Samples out.ogv                  Public    tool                           x264
chandler@nvidia:~$ recordmydesktop
```

CPU THREADS

Thread Activities

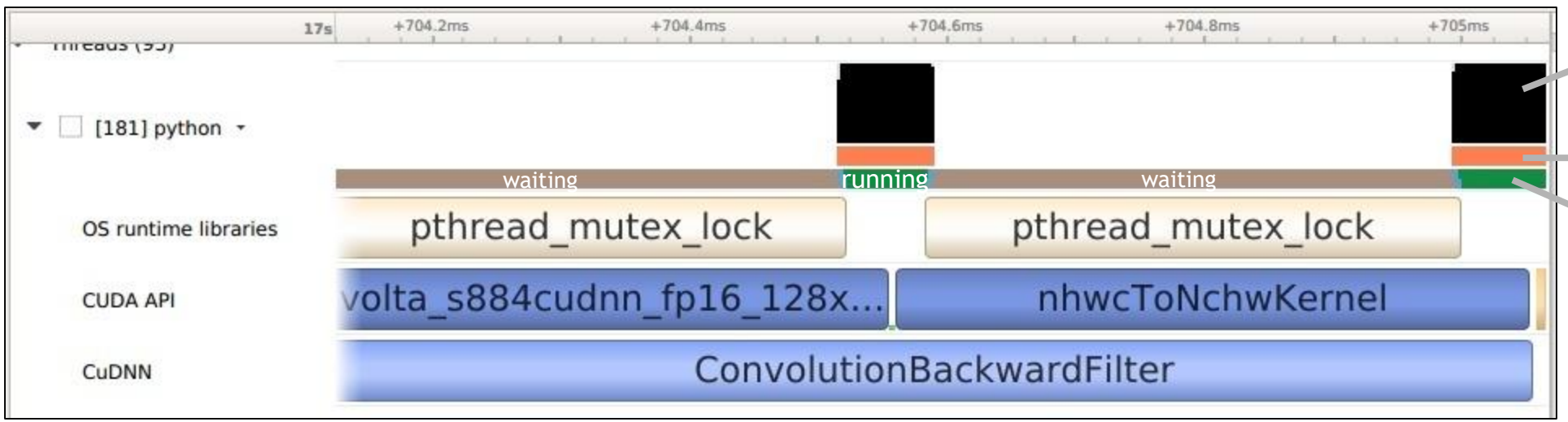
Get an overview of each thread's activities

- 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

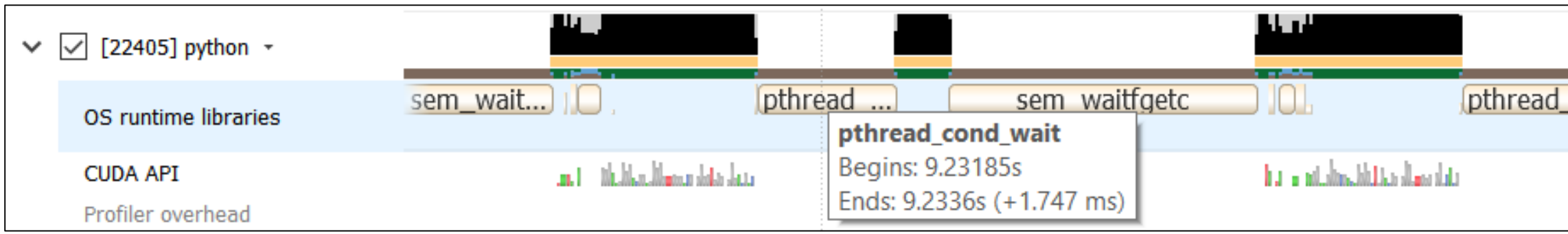
Thread Activities



OS RUNTIME LIBRARIES

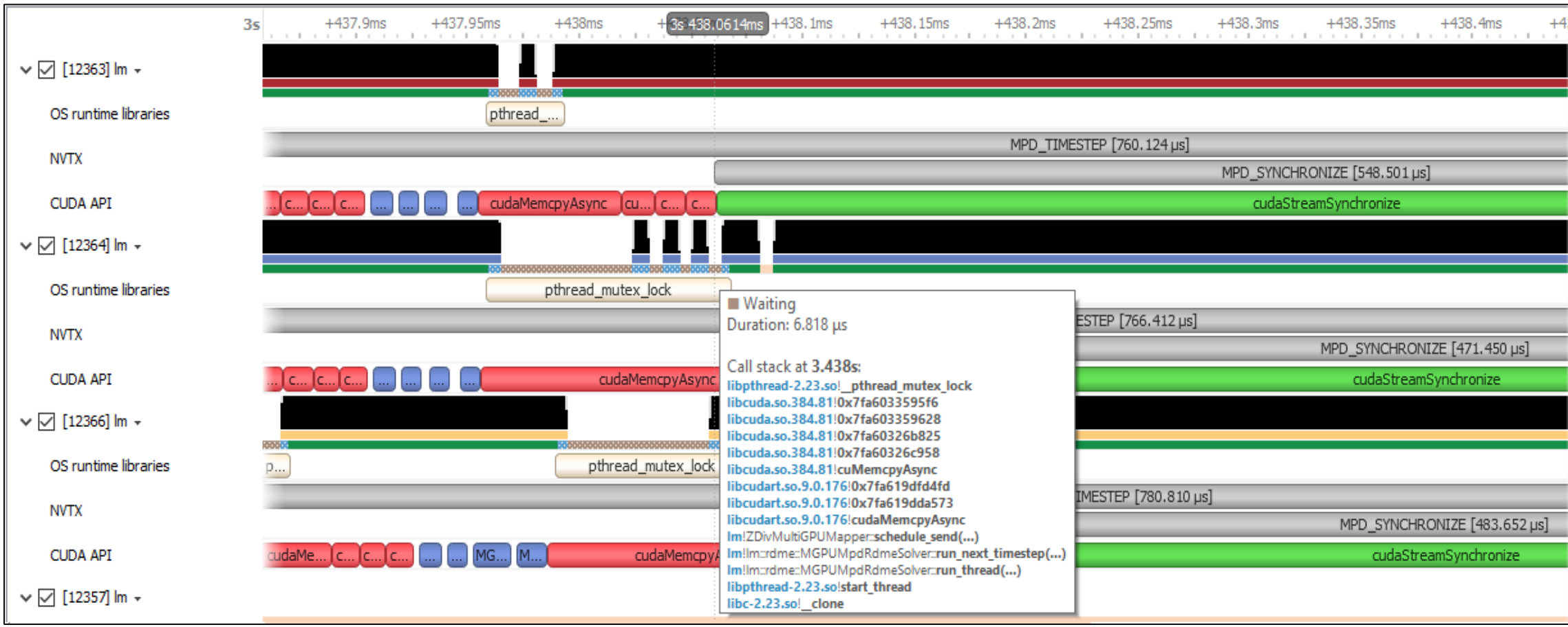
Identify time periods where threads are blocked and the reason

Locate potentially redundant synchronizations



OS RUNTIME LIBRARIES

Backtrace for time-consuming calls to OS runtime libs



CUDA API

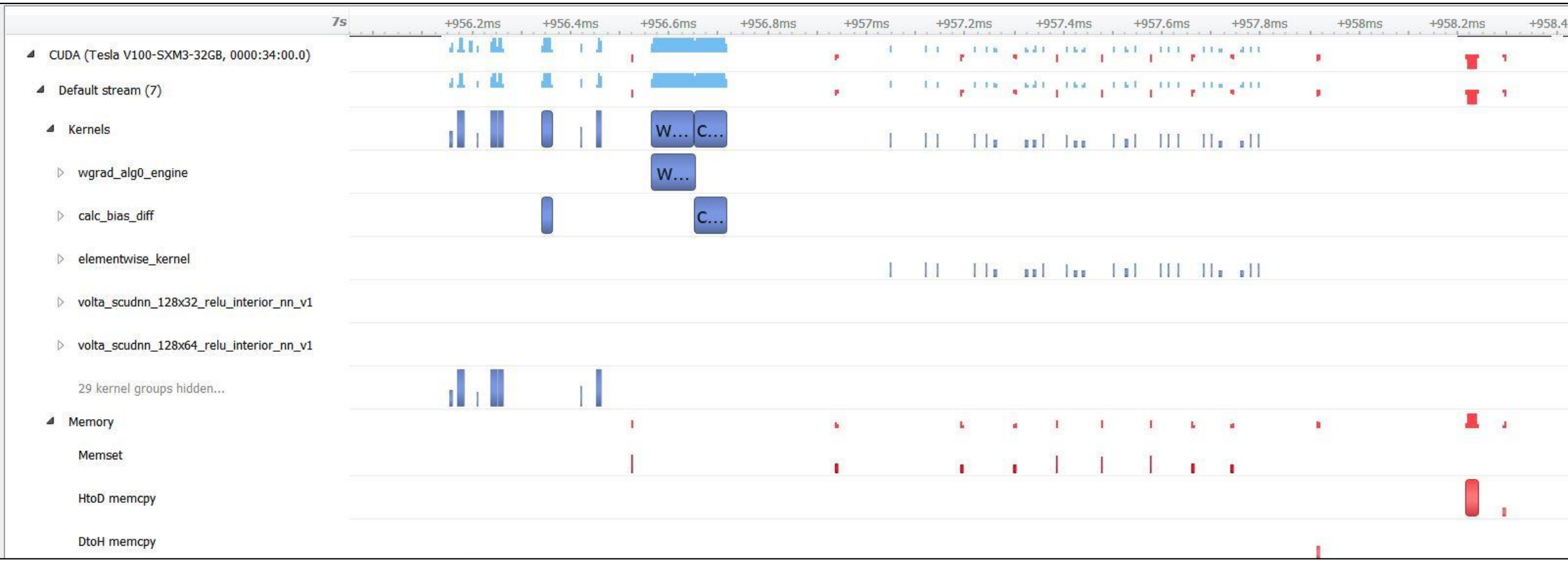
Trace CUDA API Calls on OS thread

- 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

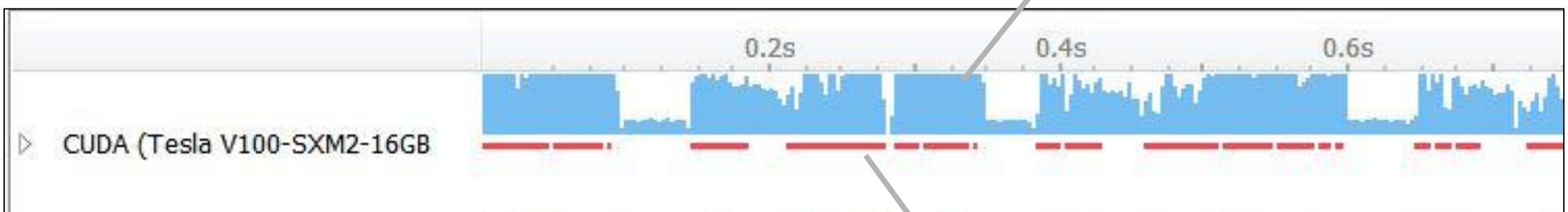
Locate idle GPU times



GPU WORKLOAD

See trace of GPU activity

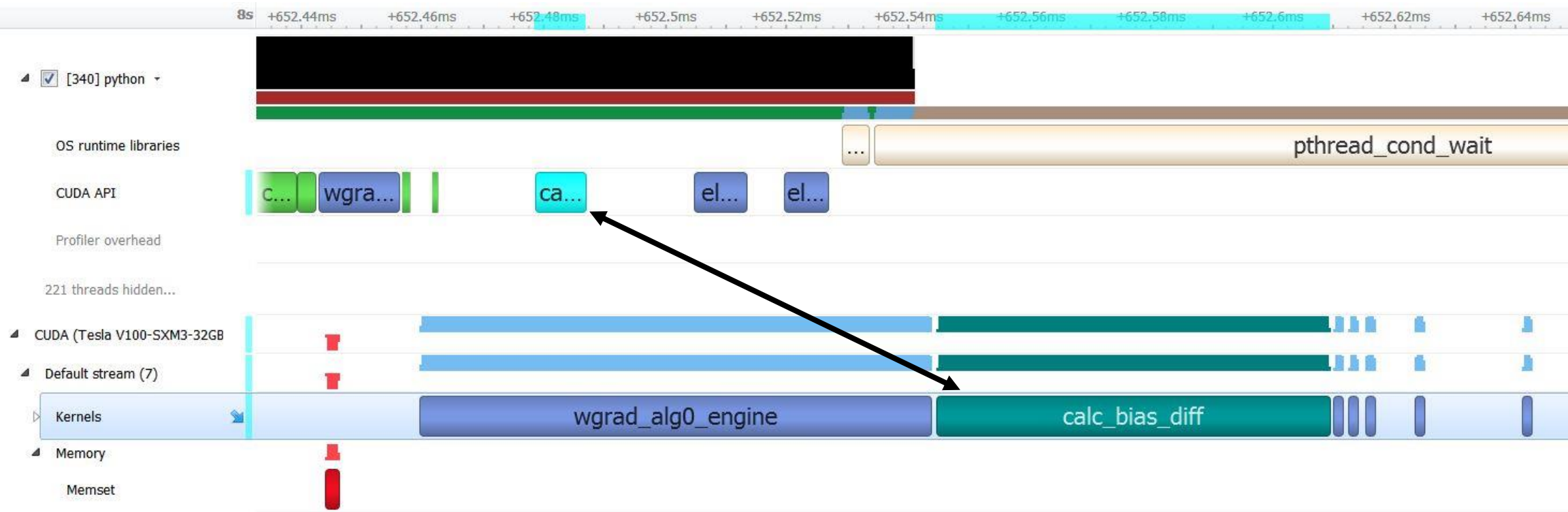
Locate idle GPU times



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

% Chart for
Avg. no. of memory operations

CORRELATION TIES API TO GPU WORKLOAD

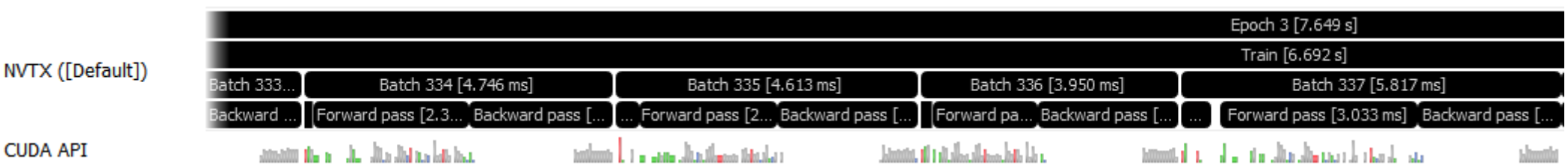


Selecting one highlights both cause and effect, i.e. dependency analysis

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

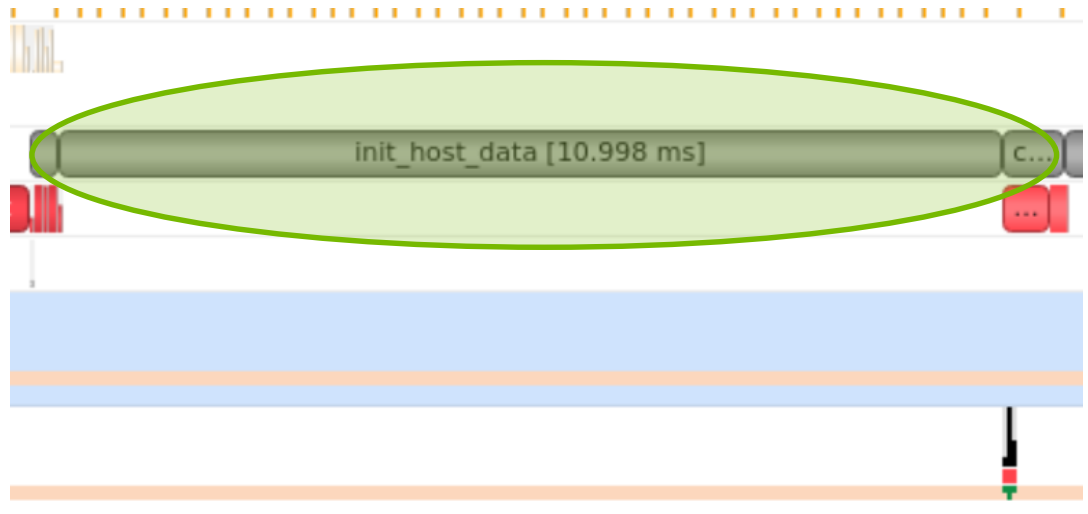
Link the NVTX library on the compiler command line with **-lnvToolsExt**

Also supports Python

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**

File Connection Debug Profile Tools Window Help

Connect Disconnect Terminate Profile Kernel

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) Ti… **Frequency:** 883.21 cycle/usecond **CC:** 7.0 **Process:** [11236] vectorAdd

API Stream

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	Duration [usecond]	4.77
SOL Memory [%]	38.46	Elapsed Cycles [cycle]	4,232
SOL TEX [%]	6.60	SM Active Cycles [cycle]	2,368.09
SOL L2 [%]	9.77	SM Frequency [cycle/usecond]	883.21
SOL FB [%]	38.46	Memory Frequency [cycle/usecond]	626.40

GPU Utilization

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
Issued Ipc Active [inst/cycle]	4.48

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	Mem Busy [%]	9.77
L1 Hit Rate [%]	0	Max Bandwidth [%]	38.46
L2 Hit Rate [%]	34.75	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	Instructions Per Active Issue Slot [inst/cycle]	1
Eligible Warps Per Scheduler [warp]	0.05	No Eligible [%]	96.17
Issued Warp Per Scheduler	0.04	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

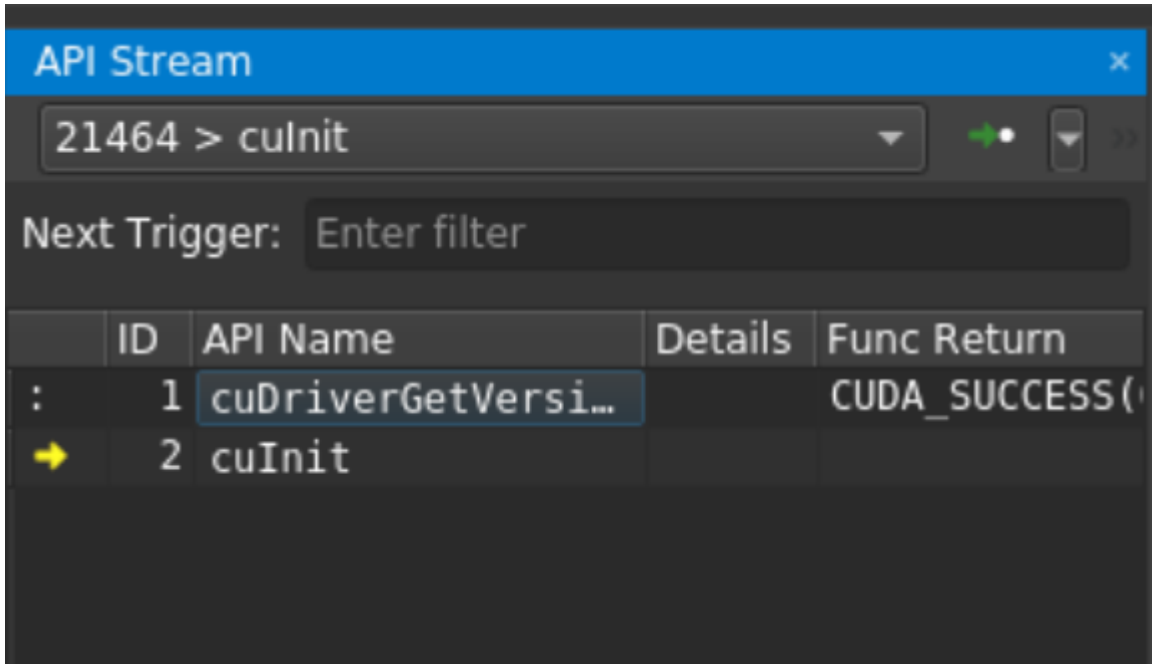
API Stream

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

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

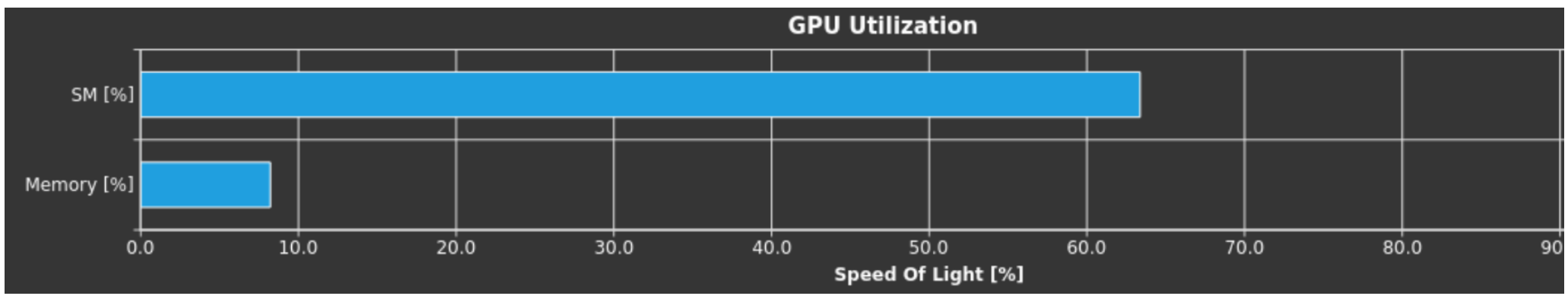
A **section** is a group of some metrics. Aim to help developers to group metrics and find optimization opportunities quickly

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

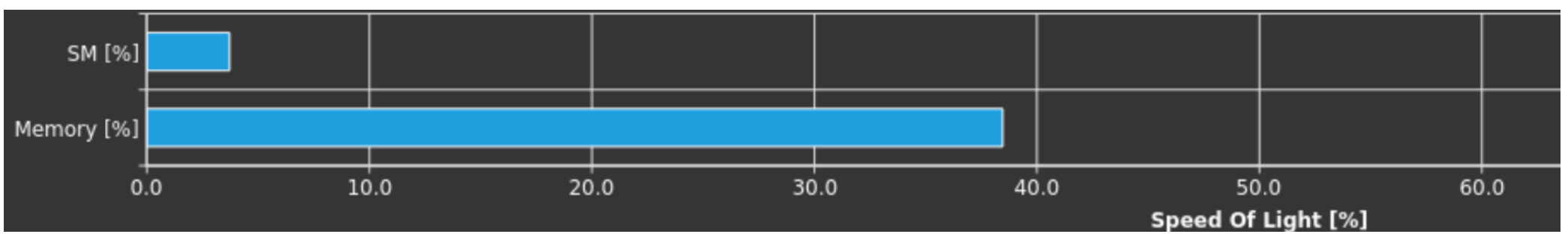


SOL SECTION

Sections

SOL Section (case 2: 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

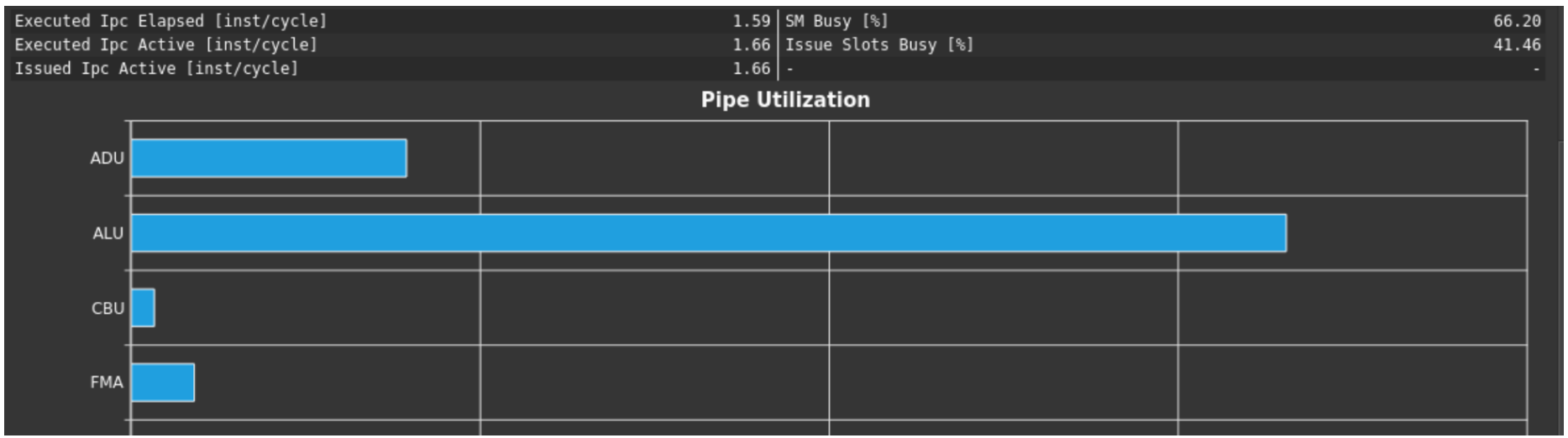


COMPUTE WORKLOAD ANALYSIS

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



SCHEDULER STATISTICS

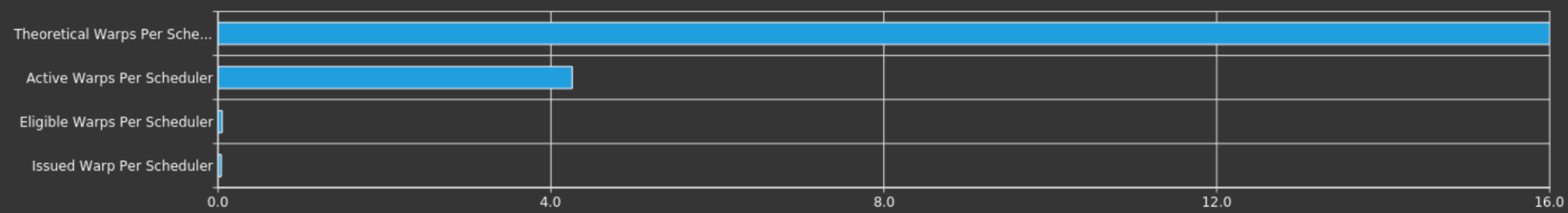
Sections

Scheduler Statistics(case 2)

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	Instructions Per Active Issue Slot [inst/cycle]	1
Eligible Warps Per Scheduler [warp]	0.05	No Eligible [%]	96.17
Issued Warp Per Scheduler	0.04	One or More Eligible [%]	3.83

Warps Per Scheduler



Recommendations

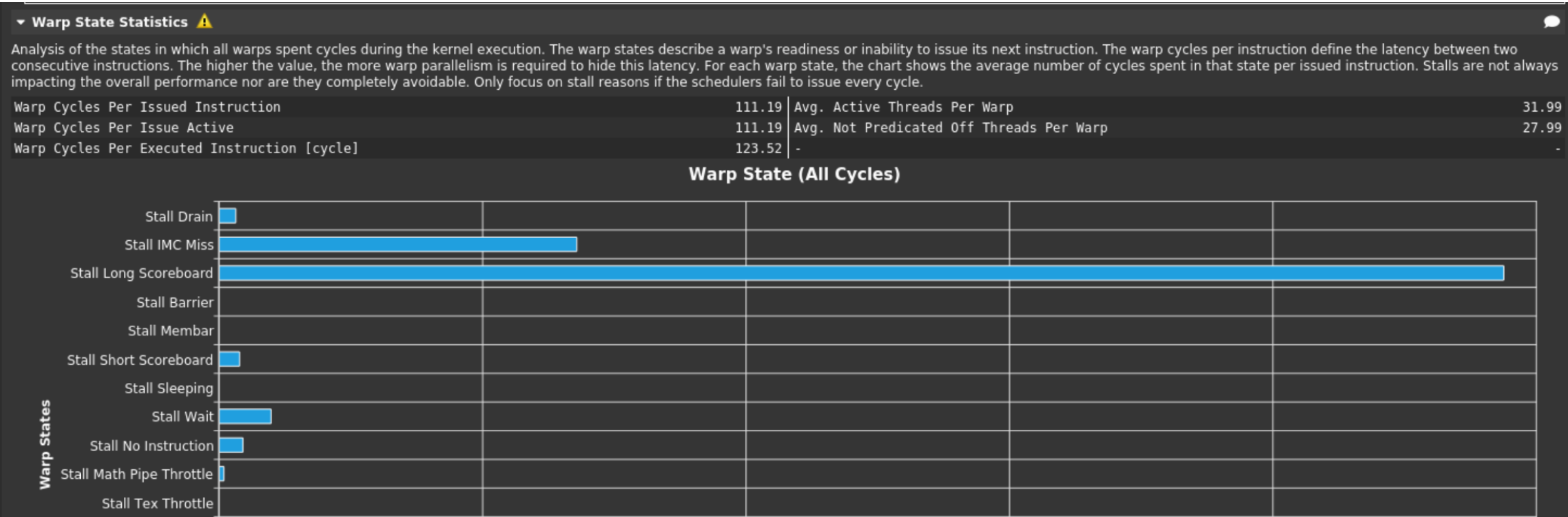
Issue Slot Utilization

[Warning] Every scheduler is capable of issuing one instruction per cycle, but for this kernel each scheduler only issues an instruction every 26.1 cycles. This might leave hardware resources underutilized and may lead to less optimal performance. Out of the maximum of 16 warps per scheduler, this kernel allocates an average of 4.26 active warps per scheduler, but only an average of 0.05 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 either increase the number of active warps or reduce the time the active warps are stalled.

WARP STATE STATISTICS

Sections

Warp State Statistics (case 2)

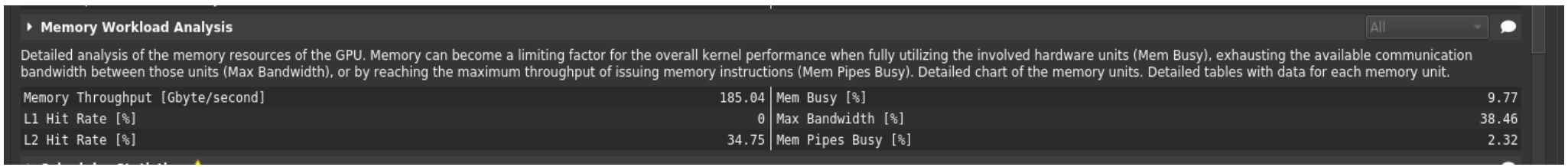


MEMORY WORKLOAD ANALYSIS

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.



WARP SCHEDULER

Volta Architecture

4 Warp Scheduler per SM

Manages a pool of warps:

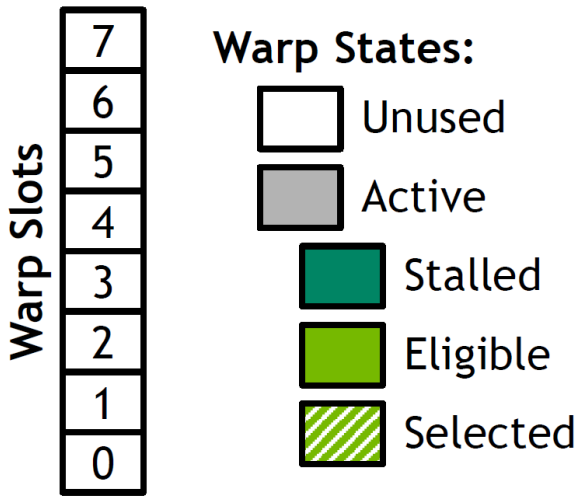
- Volta: 16 warp slots
- Turing: 8 warp slots

Each scheduler can issue 1 warp/cycle

Offers simplified mental model for profiling and SM metrics

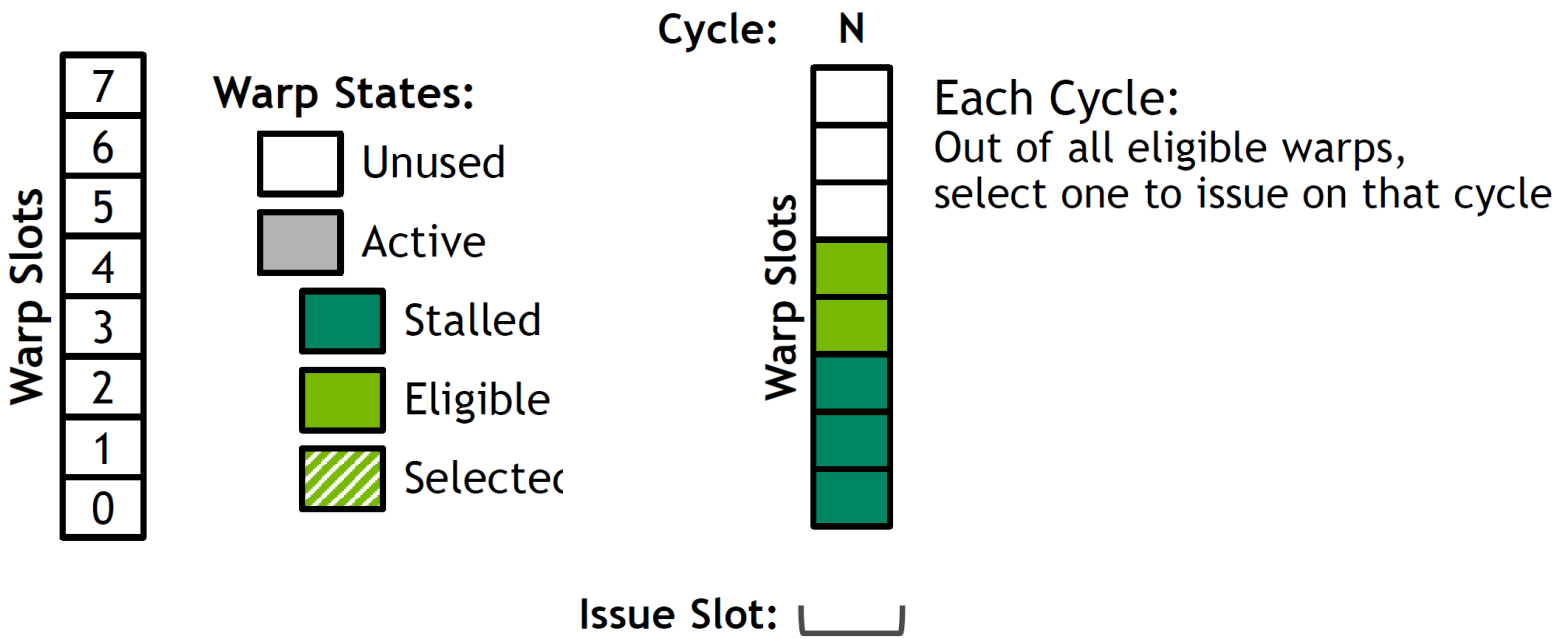
WARP SCHEDULER

Mental Model for Profiling



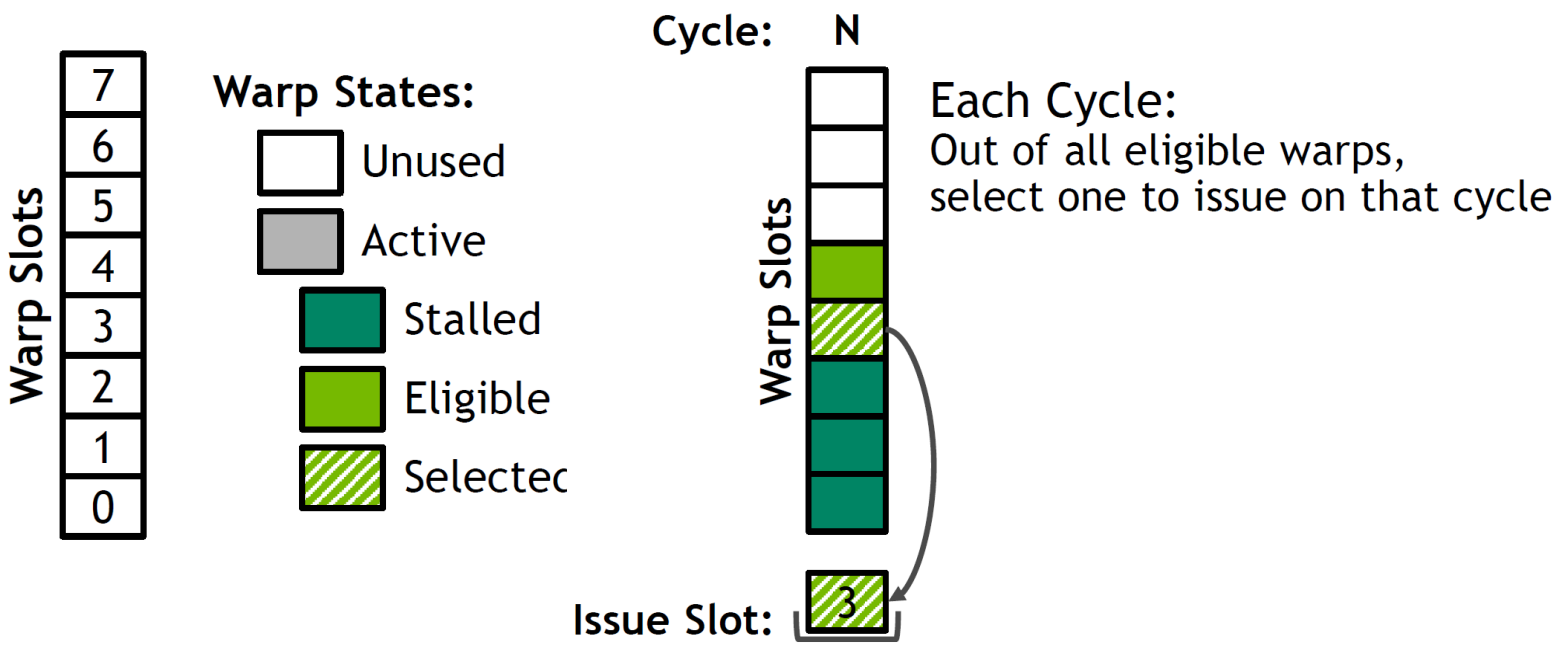
WARP SCHEDULER

Mental Model for Profiling



WARP SCHEDULER

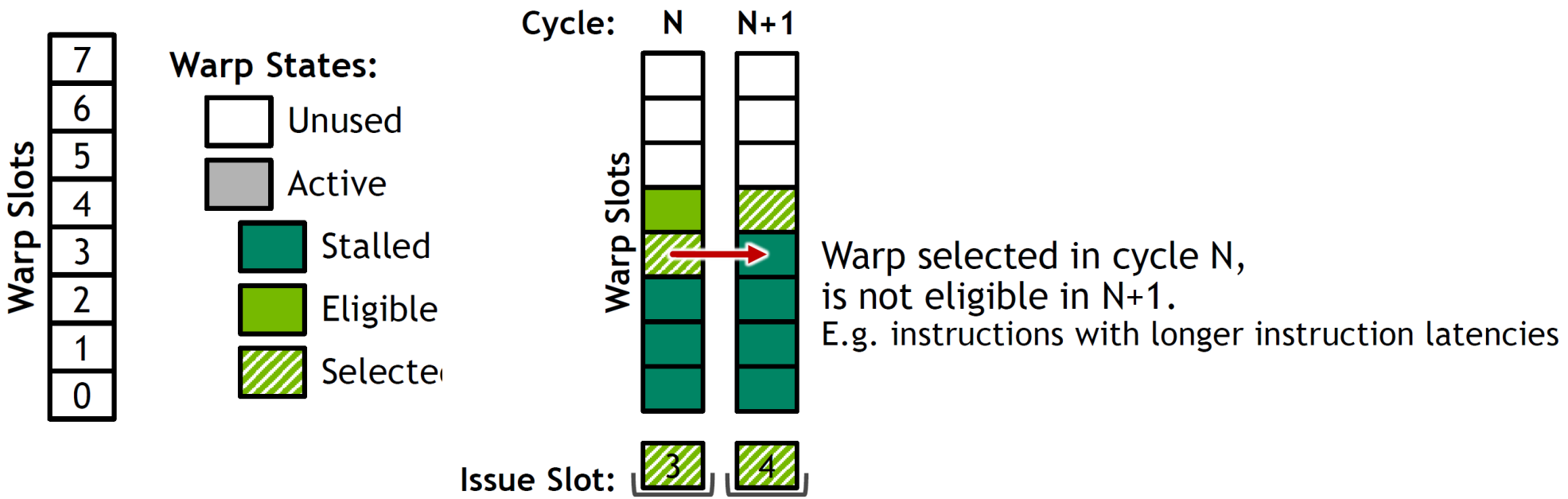
Mental Model for Profiling



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

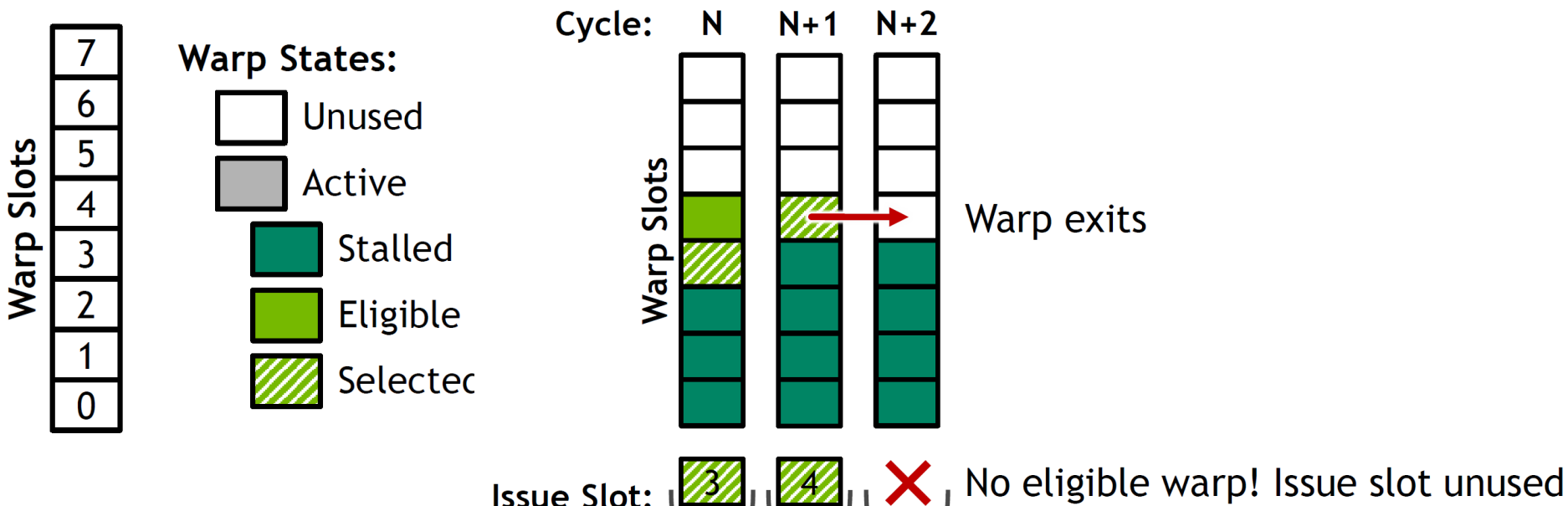
WARP SCHEDULER

Mental Model for Profiling



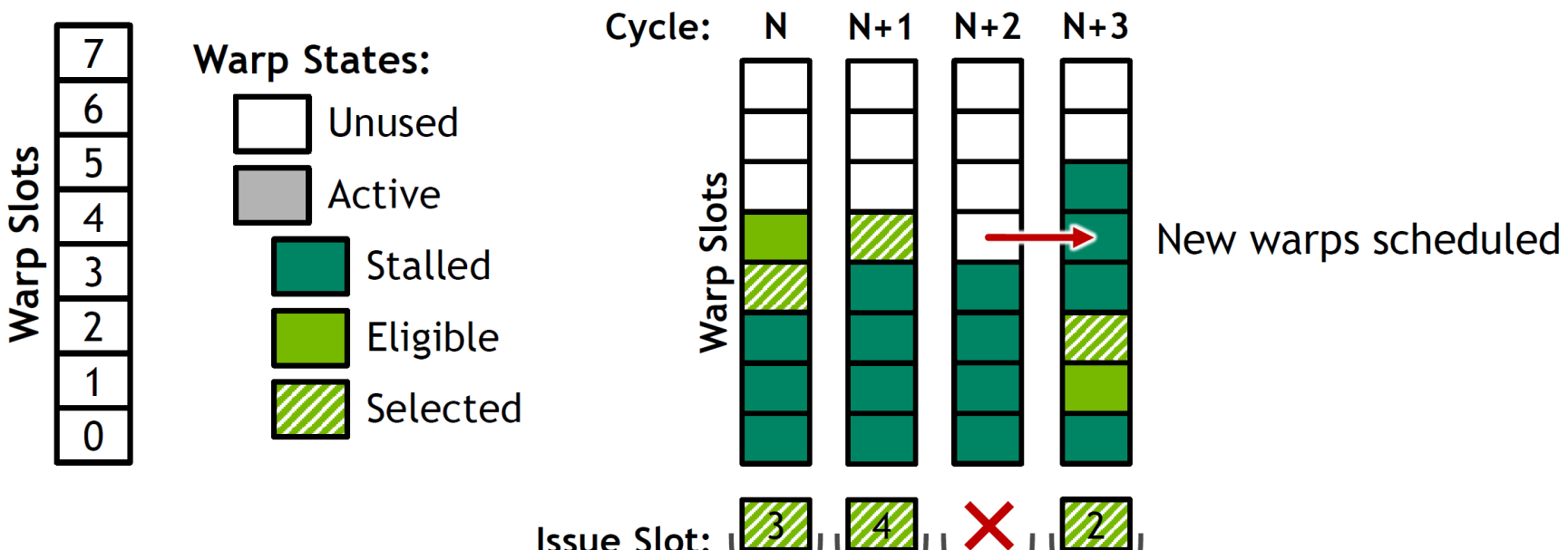
WARP SCHEDULER

Mental Model for Profiling



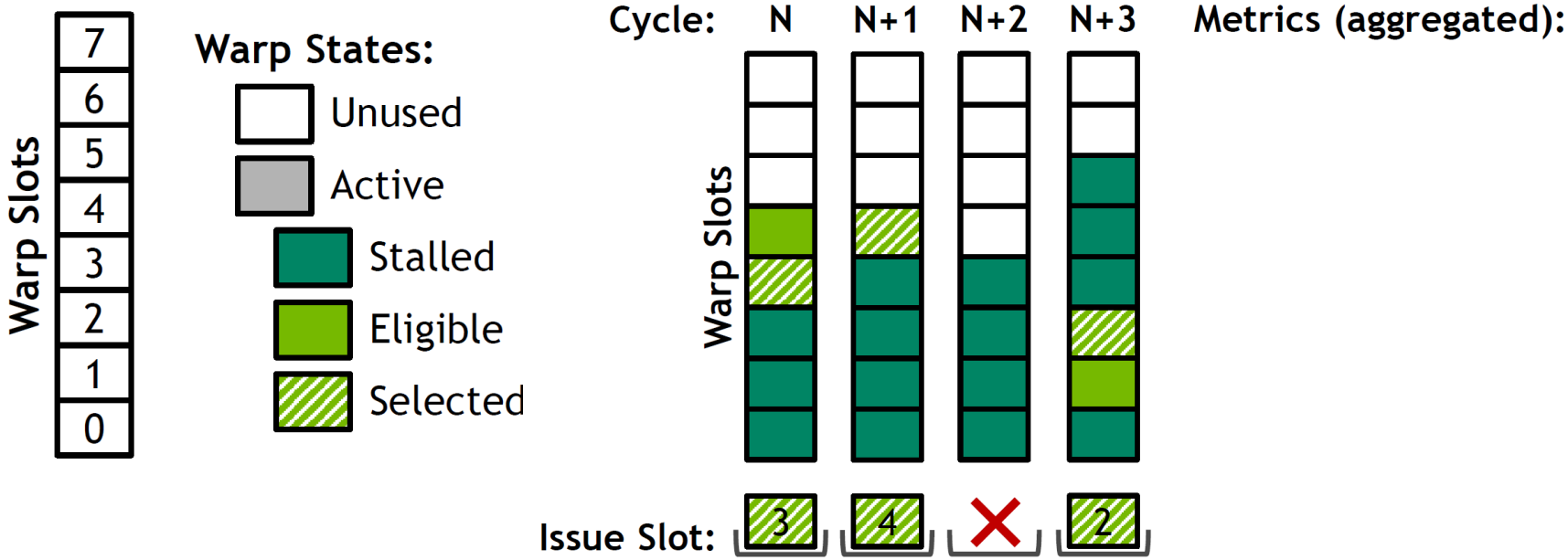
WARP SCHEDULER

Mental Model for Profiling



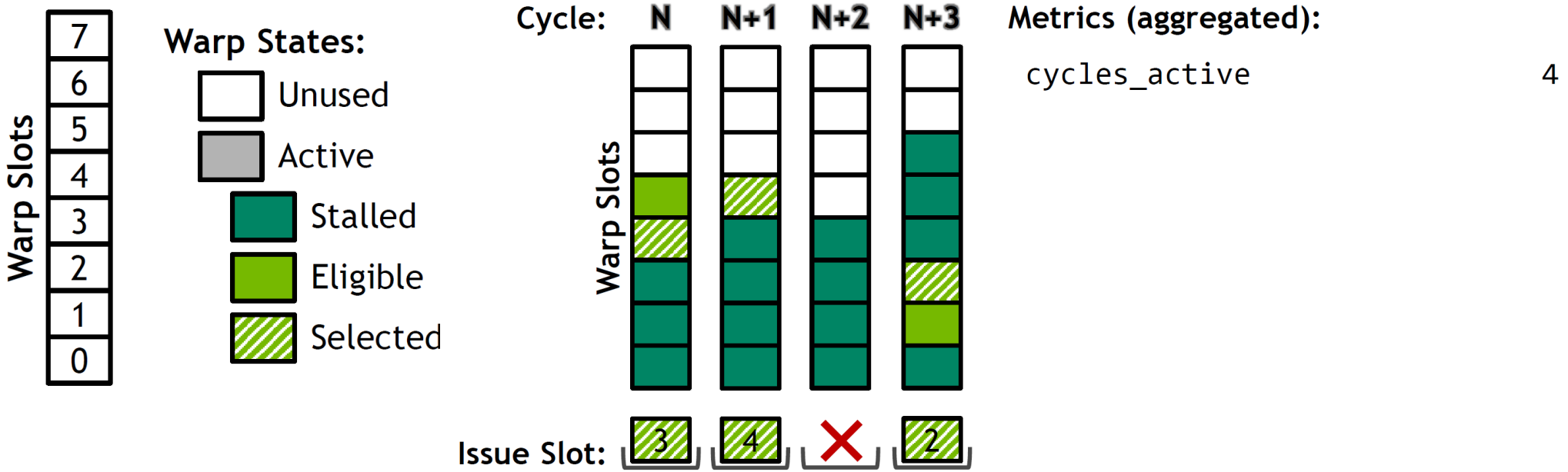
WARP SCHEDULER

Mental Model for Profiling



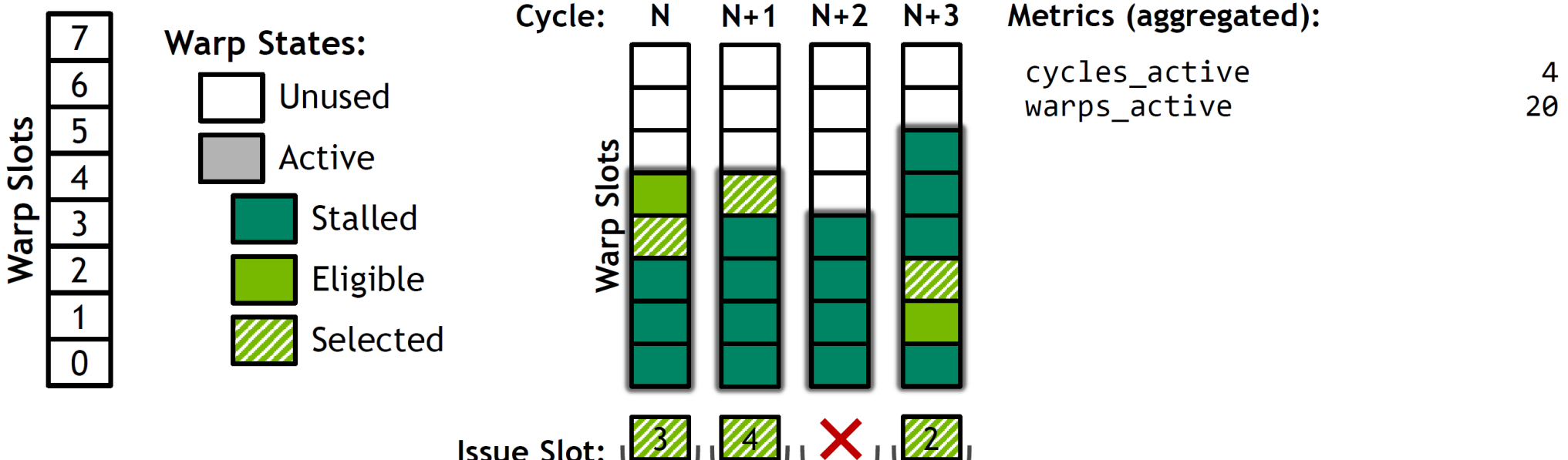
WARP SCHEDULER

Mental Model for Profiling



WARP SCHEDULER

Mental Model for Profiling

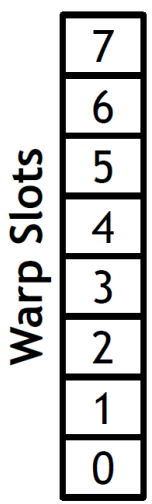


Metrics (aggregated):

`cycles_active`	4
`warps_active`	20

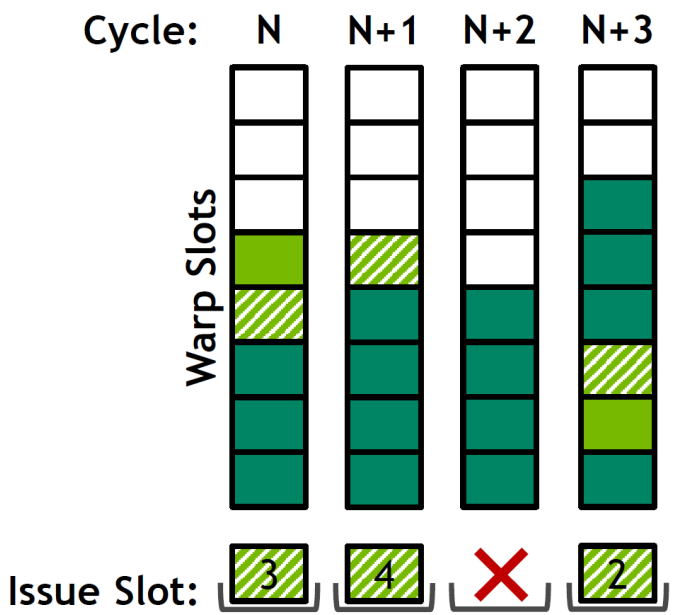
WARP SCHEDULER

Mental Model for Profiling



Warp States:

- Unused
- Active
- Stalled
- Eligible
- Selected

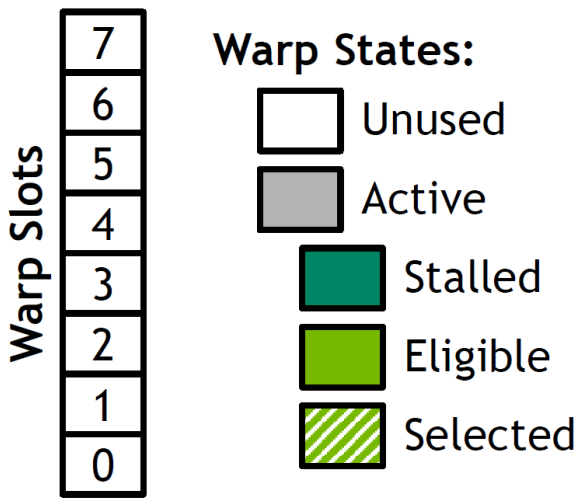


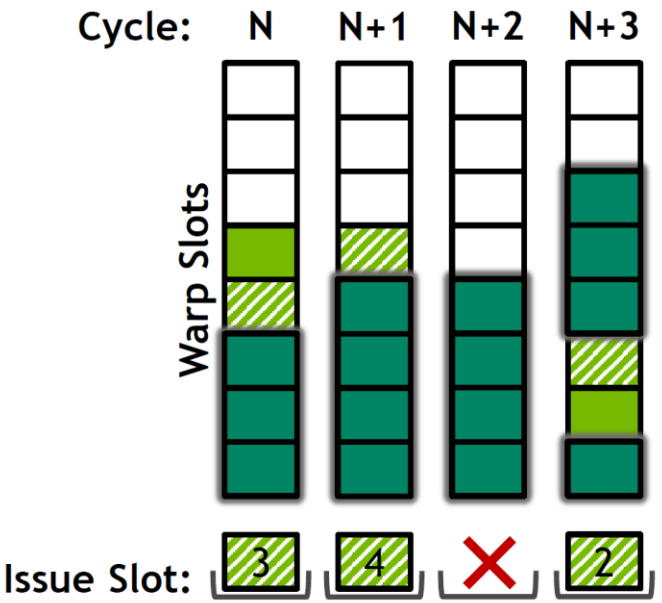
Metrics (aggregated):

```
cycles_active                    4
warps_active                    20

warps_active/cycles_active       5
achieved_occupancy           62.5%
```

WARP SCHEDULER

Mental Model for Profiling



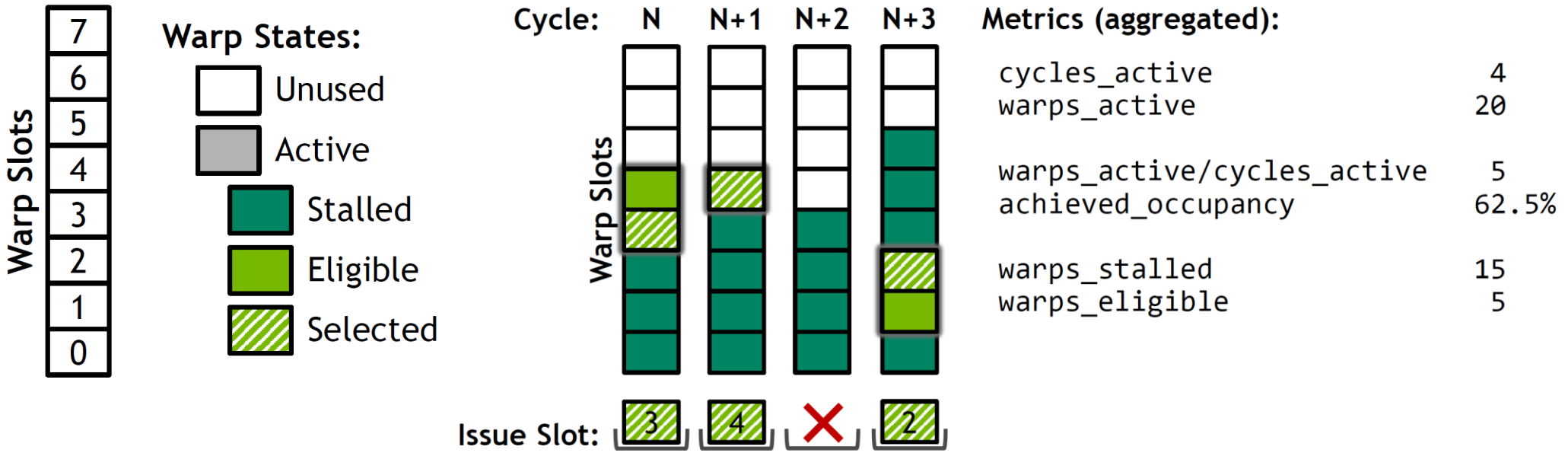


Metrics (aggregated):

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

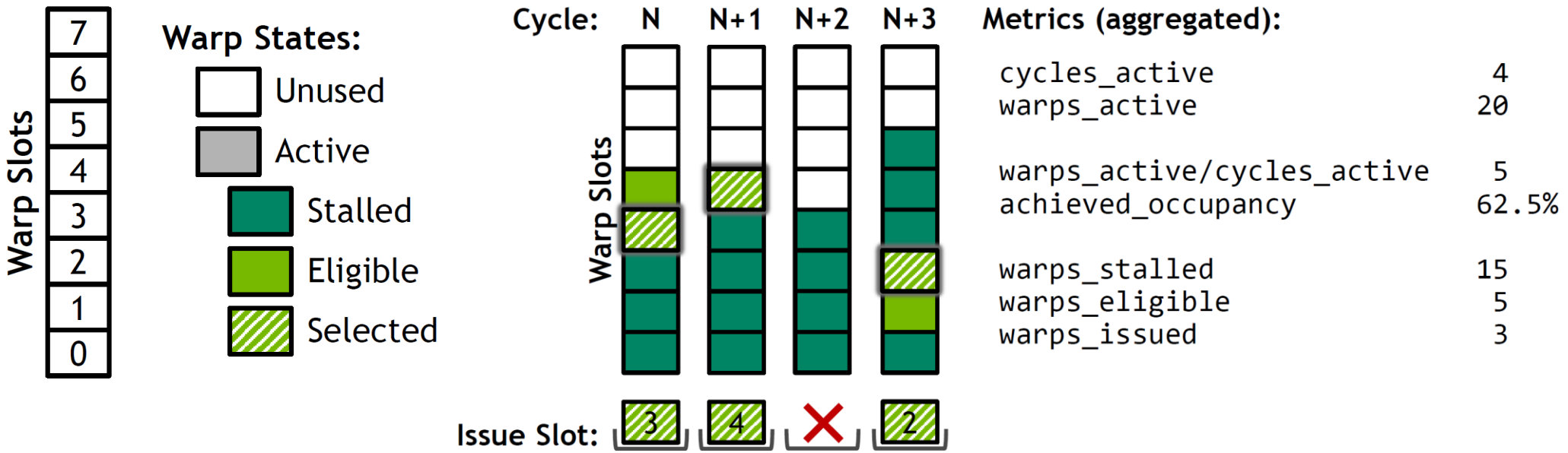
WARP SCHEDULER

Mental Model for Profiling



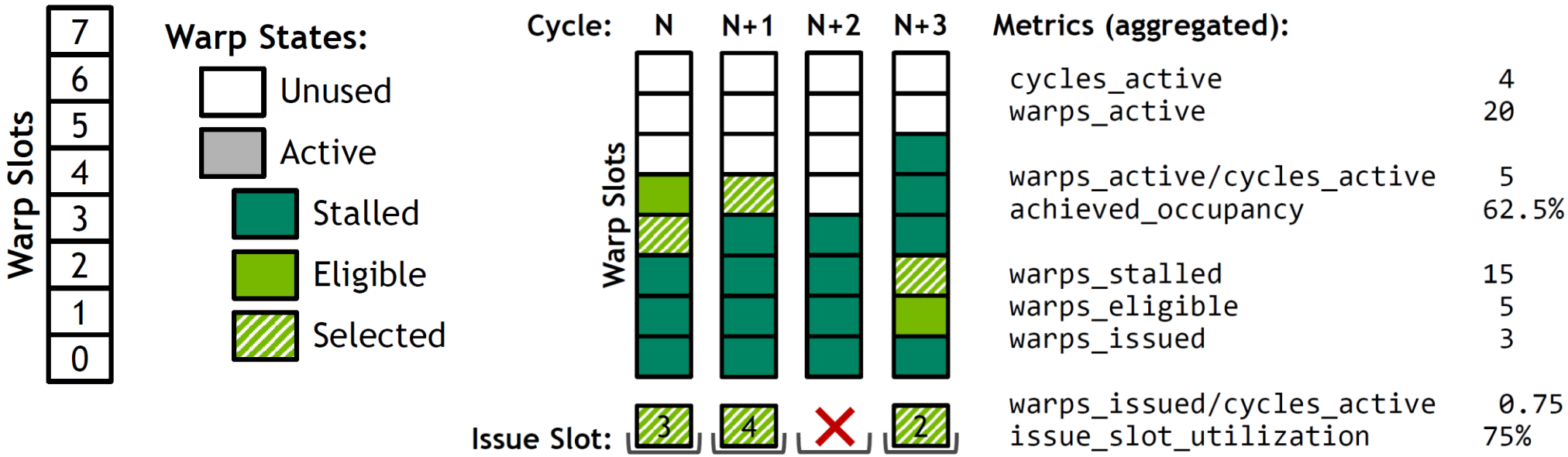
WARP SCHEDULER

Mental Model for Profiling



WARP SCHEDULER

Mental Model for Profiling



CASE STUDY 1: SIMPLE DNN TRAINING

DATASET

mnist

The **MNIST database**

A database of handwritten digits

Will be used for training a DNN

that recognizes handwritten digits

SIMPLE TRAINING PROGRAM

mnist

A simple DNN training program from
https://github.com/pytorch/examples/tree/master/mnist

Uses PyTorch, accelerated using a Volta GPU

Training is done in batches and epochs

- Load data from disk
- Data is copied to the device
- Forward pass
- Backward pass

```
def train(args, model, device, train_loader, optimizer, epoch):
    model.train()

    for batch_idx, (data, target) in enumerate(train_loader):

        data, target = data.to(device), target.to(device)

        optimizer.zero_grad()
        output = model(data)
        loss = F.nll_loss(output, target)

        loss.backward()
        optimizer.step()

        if batch_idx % args.log_interval == 0:
            print('Train Epoch: {} [{}/{} ({:.0f}%)]\tLoss: {:.6f}'.format(
                epoch, batch_idx * len(data), len(train_loader.dataset),
                100. * batch_idx / len(train_loader), loss.item()))
```

Data Loading

Copy to Device

Forward Pass

Backward Pass

TRAINING PERFORMANCE

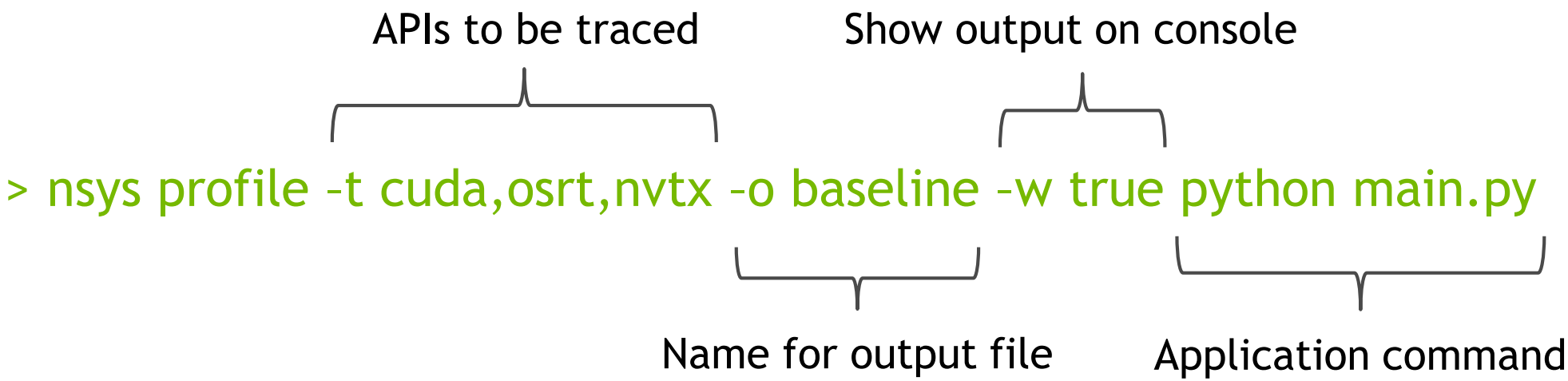
mnist

Execution time

> python main.py

Takes **89** seconds on a Volta GPU

STEP 1: PROFILE

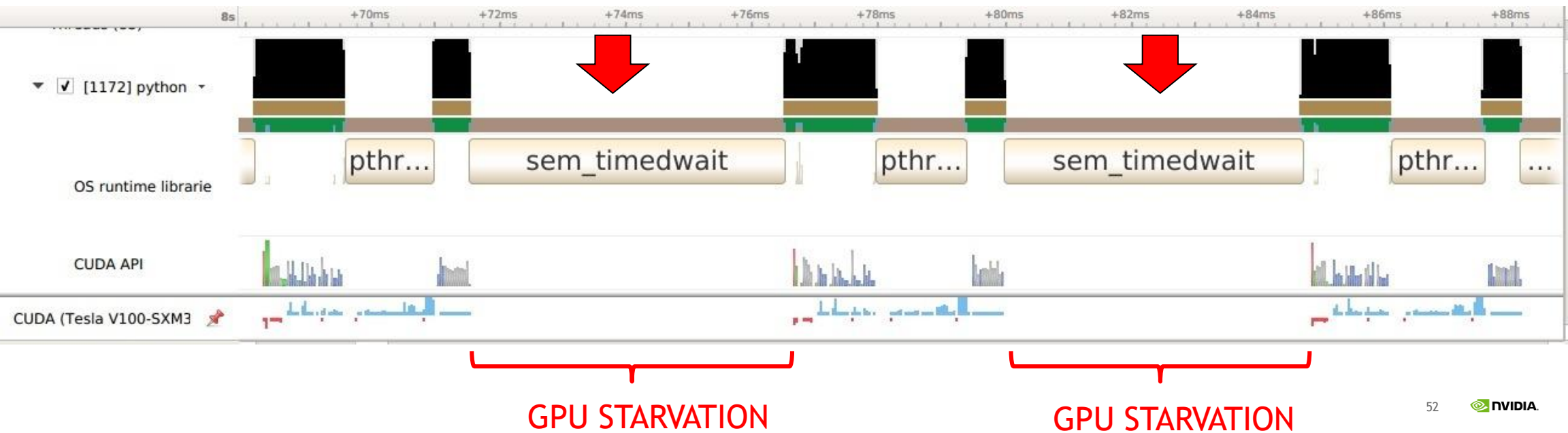


BASELINE PROFILE

GPU is Starving

Training time = 89 seconds

CPU waits on a semaphore and starves the GPU!



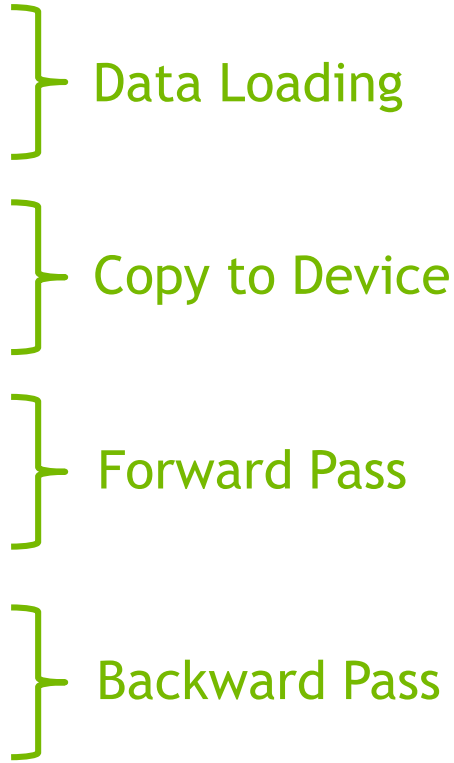
STEP2: INSPECT THE TIMELINE

From the View of Application

Add NVTX flags to understand the timeline from the view of application

```
nvtxRangePushA("different train passes");
 LoadData()/CopyToDevice()/Forward()/Backward()
nvtxRangePop();
```

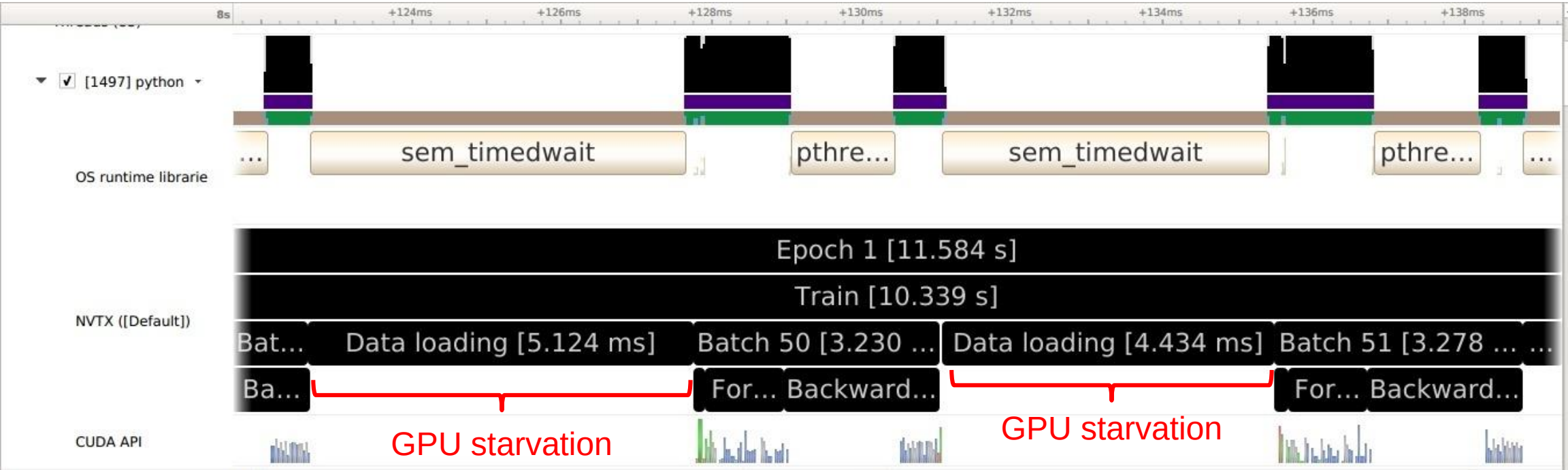
```
def train(args, model, device, train_loader, optimizer, epoch):
    model.train()
    nvtx.range_push("Data loading");
    for batch_idx, (data, target) in enumerate(train_loader):
        nvtx.range_pop();
        nvtx.range_push("Batch " + str(batch_idx))

        nvtx.range_push("Copy to device")
        data, target = data.to(device), target.to(device)
        nvtx.range_pop()

        nvtx.range_push("Forward pass")
        optimizer.zero_grad()
        output = model(data)
        loss = F.nll_loss(output, target)
        nvtx.range_pop()

        nvtx.range_push("Backward pass")
        loss.backward()
        optimizer.step()
        nvtx.range_pop()

        nvtx.range_pop()

        if batch_idx % args.log_interval == 0:
            print('Train Epoch: {} [{}/{} ({:.0f}%)]\tLoss: {:.6f}'.format(
                epoch, batch_idx * len(data), len(train_loader.dataset),
                100. * batch_idx / len(train_loader), loss.item()))
        nvtx.range_push("Data loading");
    nvtx.range_pop();
```



PROFILE WITH NVTX

GPU is Starving

The GPU starvation is caused by data loading



STEP3: OPTIMIZE SOURCE CODE

Data loader was configured to use 1 worker thread:

```
kwargs = {'num_workers': 1, 'pin_memory': True} if use_cuda else {}
```

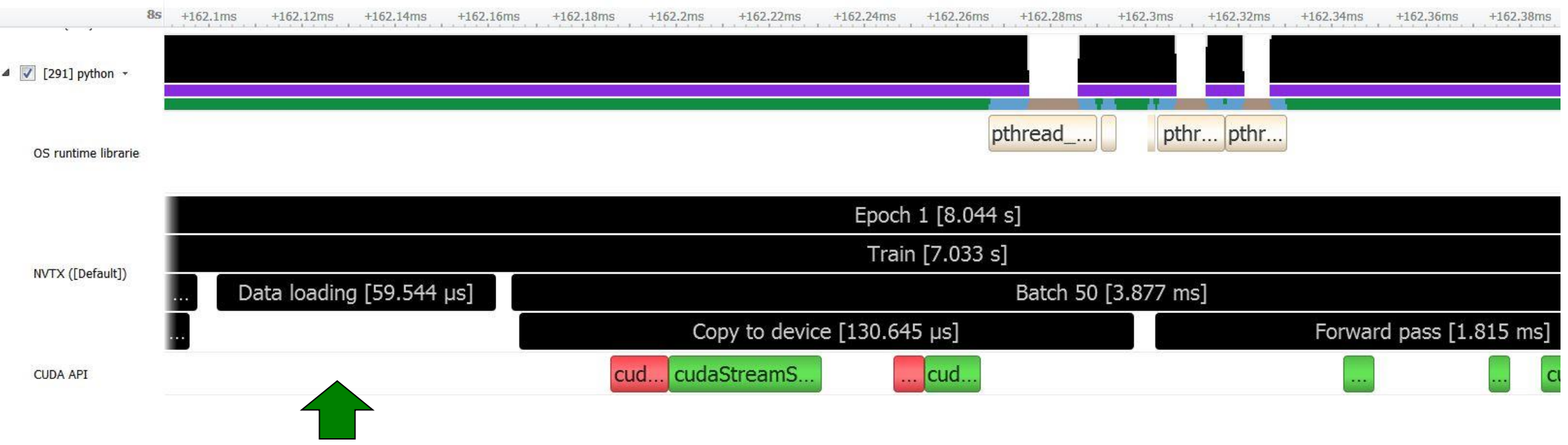
Let’s switch to using 8 worker threads:

```
kwargs = {'num_workers': 8, 'pin_memory': True} if use_cuda else {}
```

AFTER OPTIMIZATION

GPU is Starving

Time for data loading reduced for each batch



Reduced from 5.1ms to 60us for each batch

AFTER OPTIMIZATION

4.2x speedup on Tesla V100 GPU!

CASE STUDY 2: MATRIX TRANSPOSITION

MATRIX TRANSPOSITION

m = 8192 n = 4096. Some theoretical metrics

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

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

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

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

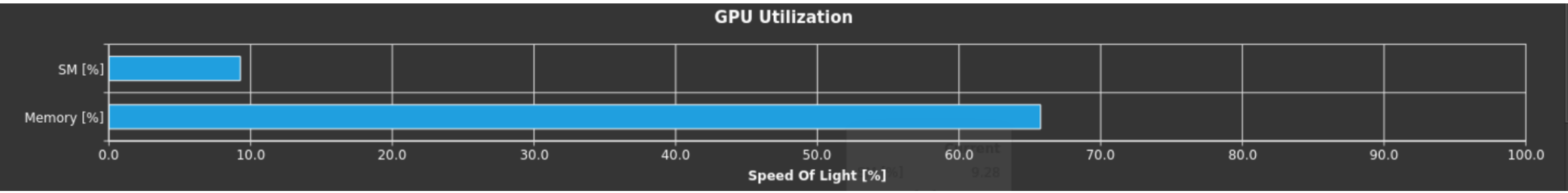
MATRIX TRANSPOSITION

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];
    }
}
```

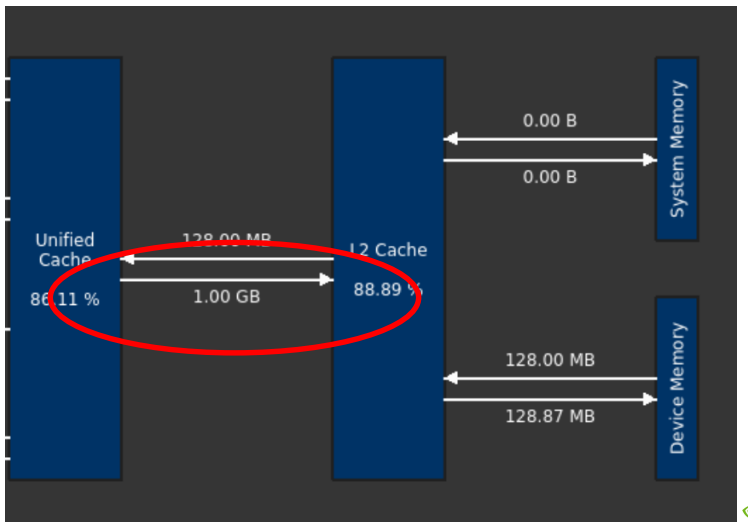
MATRIX TRANSPOSITION

Naïve Implementation



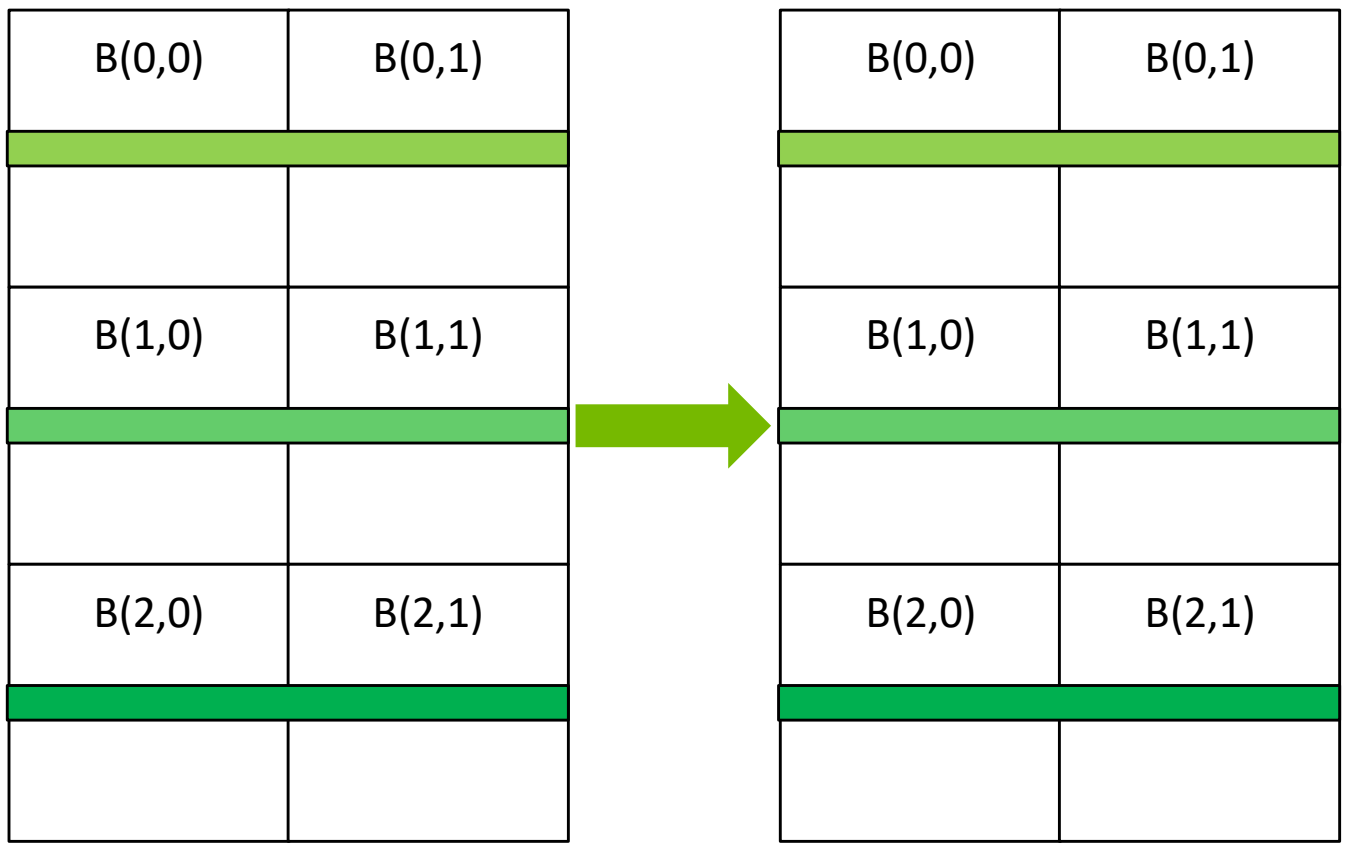
	TEX->L2 Requests (32B)	L2->Tex Returns (32B)
global load		4,194,394
global store	33,554,432	
time (us)	1890	

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



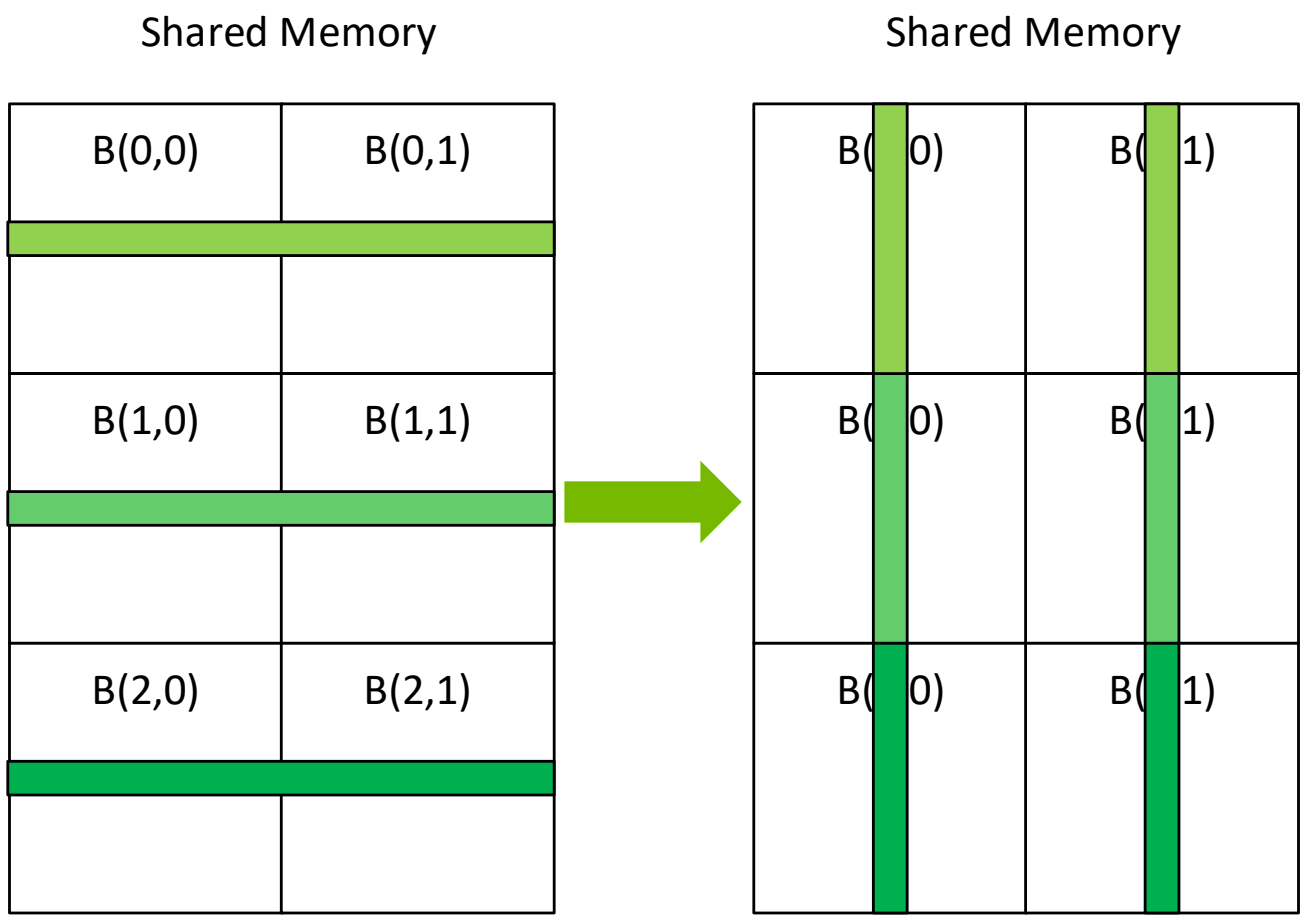
OPTIMIZATION WITH SHARED MEMORY

Load Data to Shared Memory



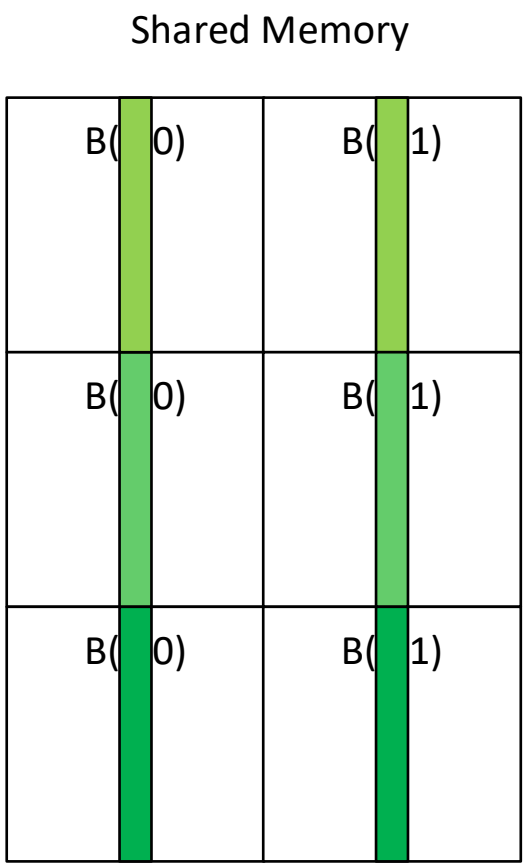
OPTIMIZATION WITH SHARED MEMORY

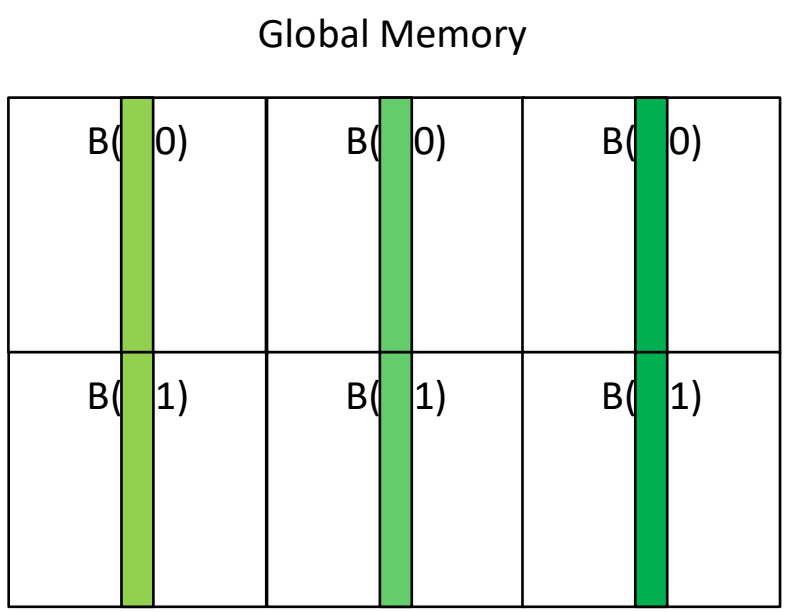
Local Transposition in Shared Memory



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;
```

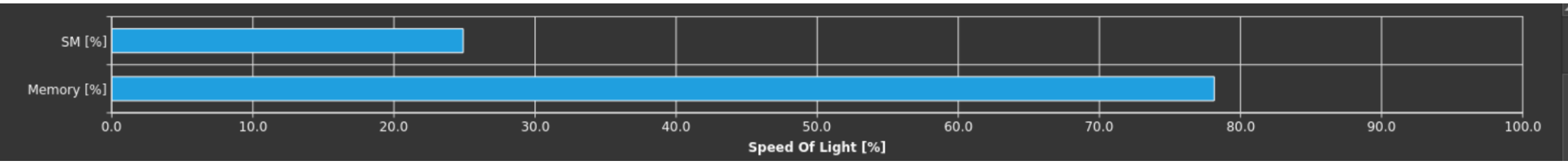
MATRIX TRANSPOSITION

Optimized Implementation

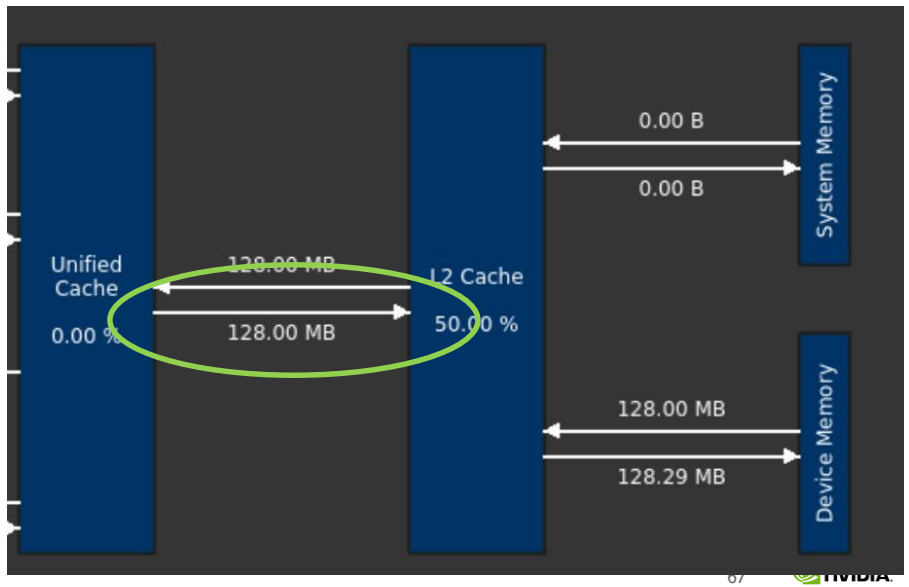
```
__global__ void transposeOptimized(float *input, float *output, int m, int n){
    int colID_input = threadIdx.x + blockDim.x*blockIdx.x;
    int rowID_input = threadIdx.y + blockDim.y*blockIdx.y;

    __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