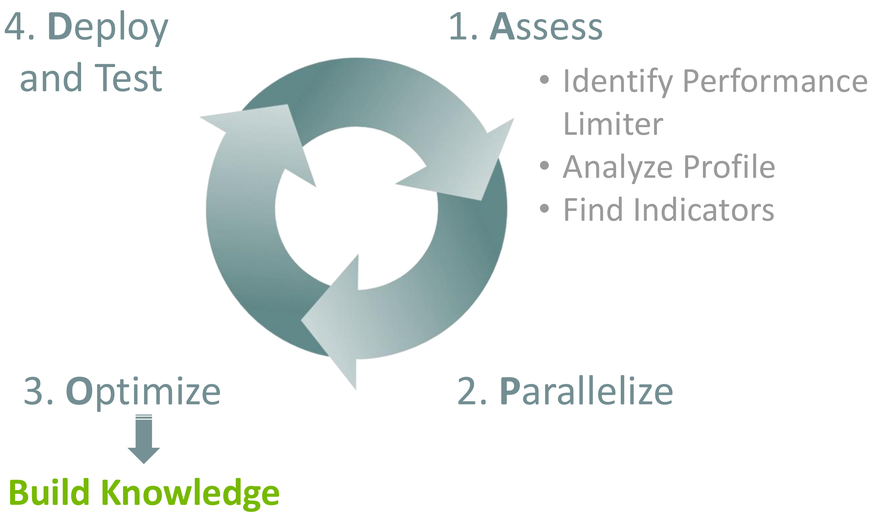
NVIDIA Nsight Tools Introduction

Filippo Spiga | fspiga@nvidia.com

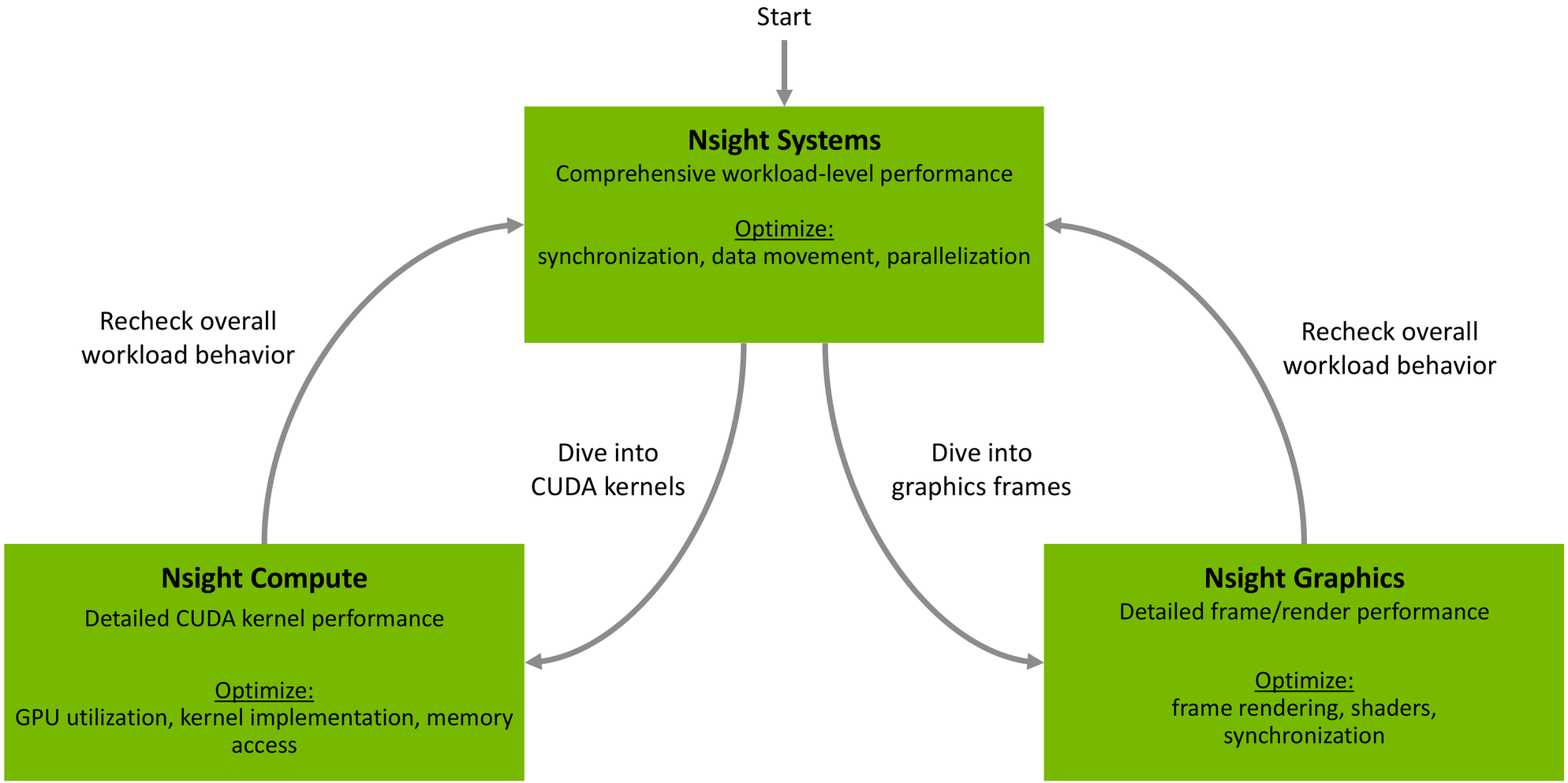
Performance Optimization: where to start?

Why you *must* use profilers

- Paraphrasing Donald Knuth:
  - Don‘t overoptimize - optimize your own time by using tools to focus on relevant parts
- Do not trust your gut instinct – very often *very* misleading
  - Easy to waste a lot of time chasing the "perceived" issue
- Getting the same information, you end up reimplementing your own profiler
- Iterative workflow
- Different kinds of measurement tools, different tradeoffs
  - Instrumenting/Sampling
  - Profiling/Tracing
  - multi-process, single-process, kernel-level
- Here: Focus on GPU and system-level: Nsight Systems

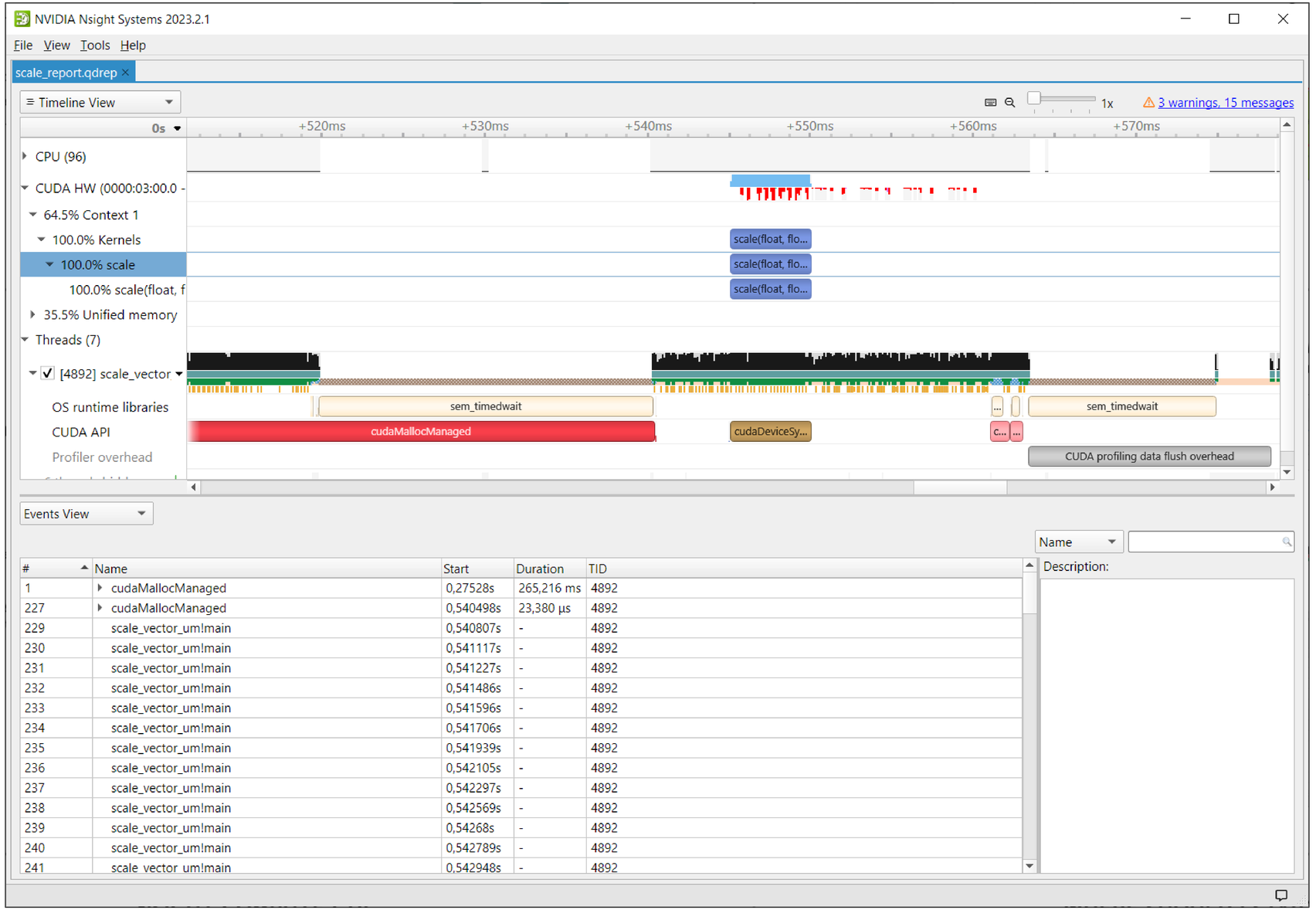


NVIDIA Nsight Performance Analysis Tools



Nsight Systems GUI

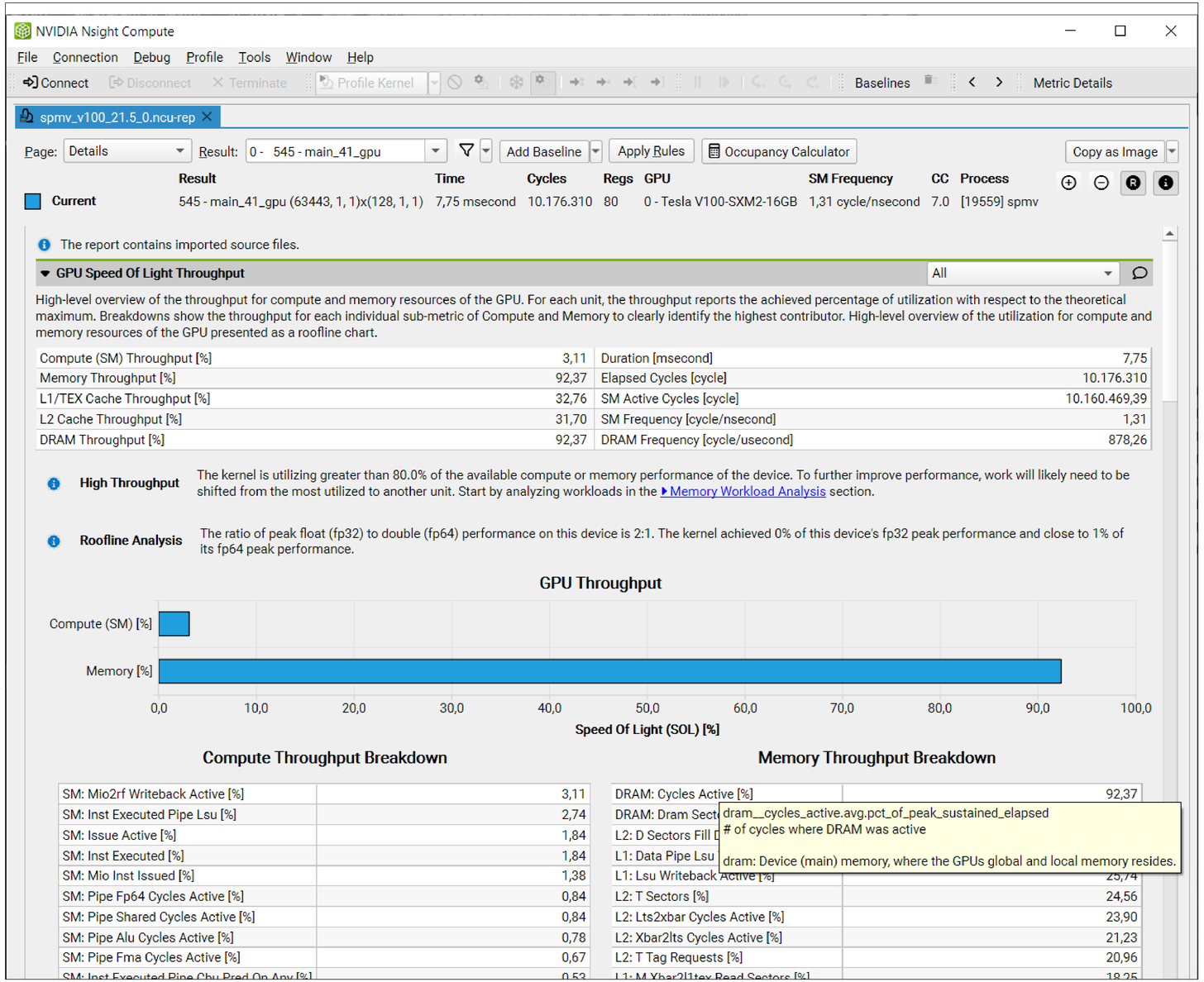
Main timeline view, Events View



Nsight Compute GUI

First steps in kernel analysis - Understanding the initial limiter

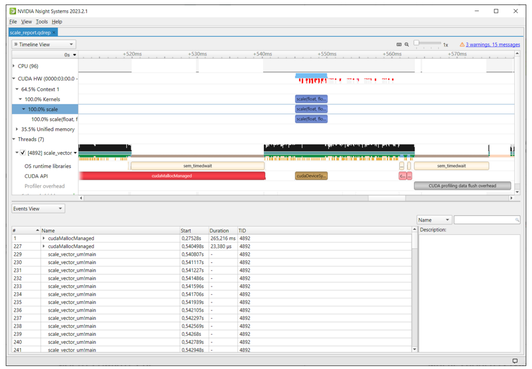
- GPU "Speed of Light Throughput"
  - SOL = theoretical peak
- "Breakdown" tables
  - DRAM: Cycles Active
- Tooltips
- Rules point to next steps

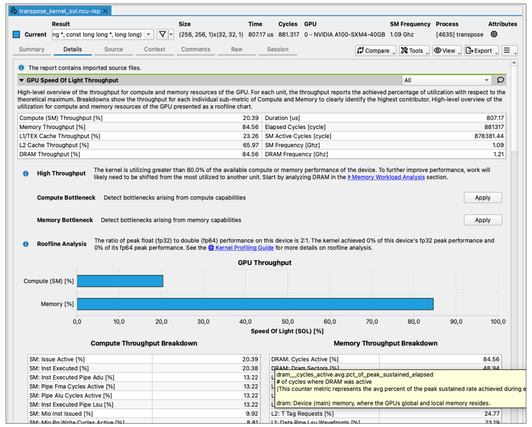


Where Should I Start Profiling?

And which tool to use?

- Always tradeoff between slightly conflicting goals
  - Performance; Maintainability; Effort
- Start with a system-level view → Nsight Systems
- Ensure you understand your timeline, and where the GPU is active/inactive
  - where initialization happens
  - how the time-% shifts for different relevant workloads
- Take the low-hanging fruit!
- Don‘t shy away from kernel-level optimization, but ensure you understand impact
  - Again, Amdahl‘s: Hypothetically, optimized kernel takes 0 s, how large is whole-program speedup?
- General guidelines – if your whole timeline is a single kernel, by all means start optimizing it first!
  - Second half of session has more detail on Nsight Compute

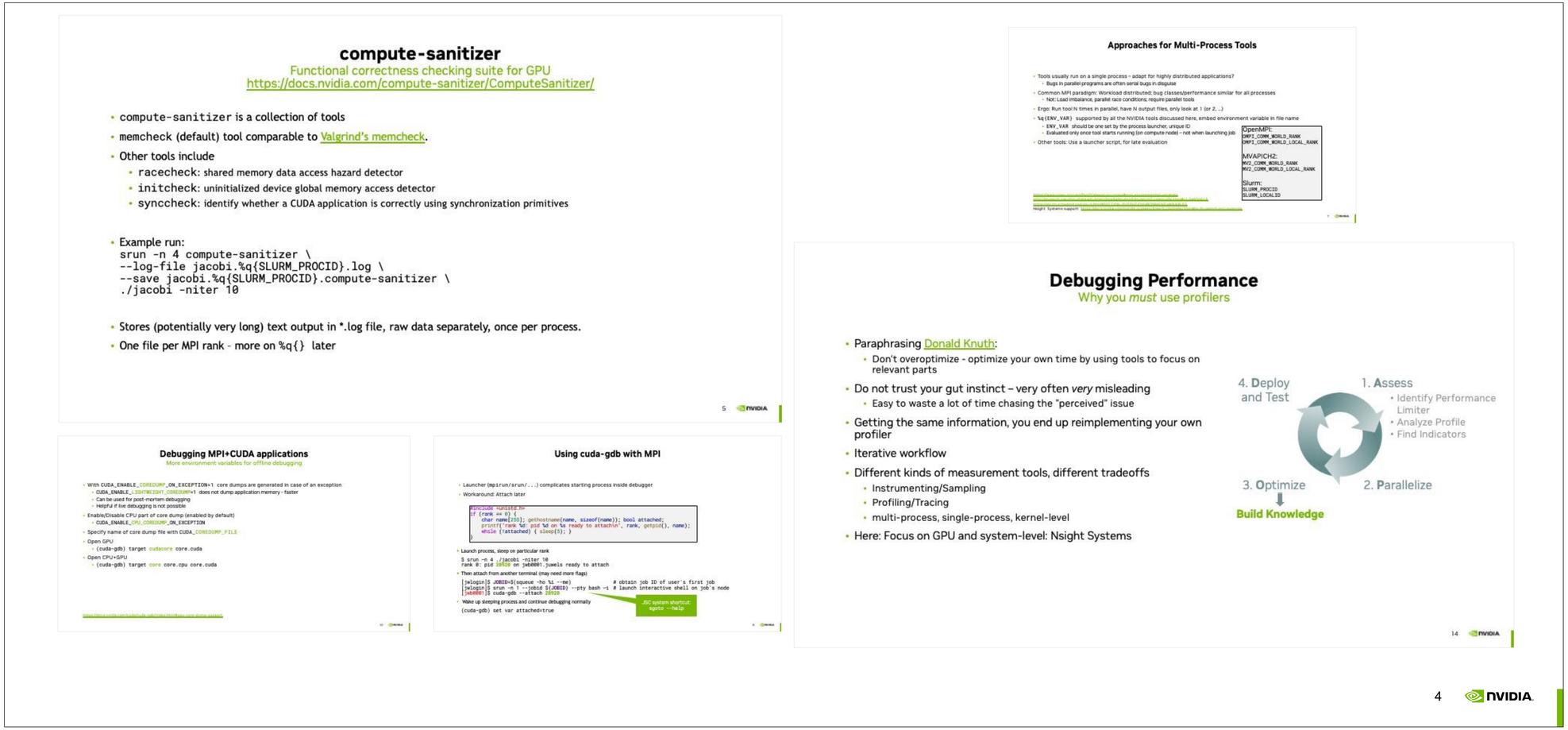




Not today: Correctness debugging

(but worth knowing about!)

- More in 04-L_Performance_and_debugging_tools from Multi-GPU tutorial
  - compute-sanitizer, cuda-gdb, …



compute-sanitizer

Functional correctness checking suite for GPU
https://docs.nvidia.com/compute-sanitizer/ComputeSanitizer/

- `compute-sanitizer` is a collection of tools
- `memcheck` (default) tool comparable to Valgrind's memcheck.
- Other tools include
  - `racecheck`: shared memory data access hazard detector
  - `initcheck`: uninitialized device global memory access detector
  - `synccheck`: identify whether a CUDA application is correctly using synchronization primitives
- Example run:

```
srun -n 4 compute-sanitizer \
--log-file jacobi.%q{SLURM_PROCID}.log \
--save jacobi.%q{SLURM_PROCID}.compute-sanitizer \
./jacobi -niter 10
```

- Stores (potentially very long) text output in *.log file, raw data separately, once per process.
- One file per MPI rank - more on `%q{}` later

5 NVIDIA

Approaches for Multi-Process Tools

- Tools usually run on a single process – adapt for highly distributed applications?
  - Bugs in parallel programs are often serial bugs in disguise
- Common MPI paradigm: Workload distributed; bug classes/performance similar for all processes
  - Not: Load imbalance, parallel race conditions; require parallel tools
- Ergo: Run tool N times in parallel, have N output files, only look at 1 (or 2, …)
- `%q{ENV_VAR}` supported by all the NVIDIA tools discussed here, embed environment variable in file name
  - `ENV_VAR` should be one set by the process launcher, unique ID
  - Evaluated only once tool starts running (on compute node) – not when launching job
- Other tools: Use a launcher script, for late evaluation

OpenMPI:
`OMPI_COMM_WORLD_RANK`
`OMPI_COMM_WORLD_LOCAL_RANK`

MVAPICH2:
`MV2_COMM_WORLD_RANK`
`MV2_COMM_WORLD_LOCAL_RANK`

Slurm:
`SLURM_PROCID`
`SLURM_LOCALID`

Nsight Systems support

1 NVIDIA

Debugging Performance

Why you *must* use profilers

- Paraphrasing Donald Knuth:
  - Don't overoptimize - optimize your own time by using tools to focus on relevant parts
- Do not trust your gut instinct – very often *very* misleading
  - Easy to waste a lot of time chasing the "perceived" issue
- Getting the same information, you end up reimplementing your own profiler
- Iterative workflow
- Different kinds of measurement tools, different tradeoffs
  - Instrumenting/Sampling
  - Profiling/Tracing
  - multi-process, single-process, kernel-level
- Here: Focus on GPU and system-level: Nsight Systems

14 NVIDIA

Debugging MPI+CUDA applications

More environment variables for offline debugging

- With `CUDA_ENABLE_COREDUMP_ON_EXCEPTION=1` core dumps are generated in case of an exception
  - `CUDA_ENABLE_LIGHTWEIGHT_COREDUMP=1` does not dump application memory - faster
  - Can be used for post-mortem debugging
  - Helpful if live debugging is not possible
- Enable/Disable CPU part of core dump (enabled by default)
  - `CUDA_ENABLE_CPU_COREDUMP_ON_EXCEPTION`
- Specify name of core dump file with `CUDA_COREDUMP_FILE`
- Open GPU
  - `(cuda-gdb) target cudacore core.cuda`
- Open CPU+GPU
  - `(cuda-gdb) target core core.cpu core.cuda`

Using cuda-gdb with MPI

- Launcher (`mpirun/srun/...`) complicates starting process inside debugger
- Workaround: Attach later

```
#include <unistd.h>
if (rank == 0) {
  char name[256]; gethostname(name, sizeof(name)); bool attached;
  printf("rank %d: pid %d on %s ready to attach\n", rank, getpid(), name);
  while (!attached) { sleep(1); }
}
```

- Launch process, sleep on particular rank

```
$ srun -n 4 ./jacobi -niter 10
rank 0: pid 20928 on jwb0001.juwels ready to attach
```

- Then attach from another terminal (may need more flags)

```
[jwlogin]$ JOBID=$(squeue -ho %i --me)                  # obtain job ID of user's first job
[jwlogin]$ srun -n 1 --jobid ${JOBID} --pty bash -i # launch interactive shell on job's node
[jwb0001]$ cuda-gdb --attach 20928
```

- Wake up sleeping process and continue debugging normally

```
(cuda-gdb) set var attached=true
```

JSC system shortcut: `sgoto --help`

4 NVIDIA

NVIDIA

NVTX

NVIDIA Tools Extensions (NVTX)

C, C++, Python

Instrument application source code

- Header-only library

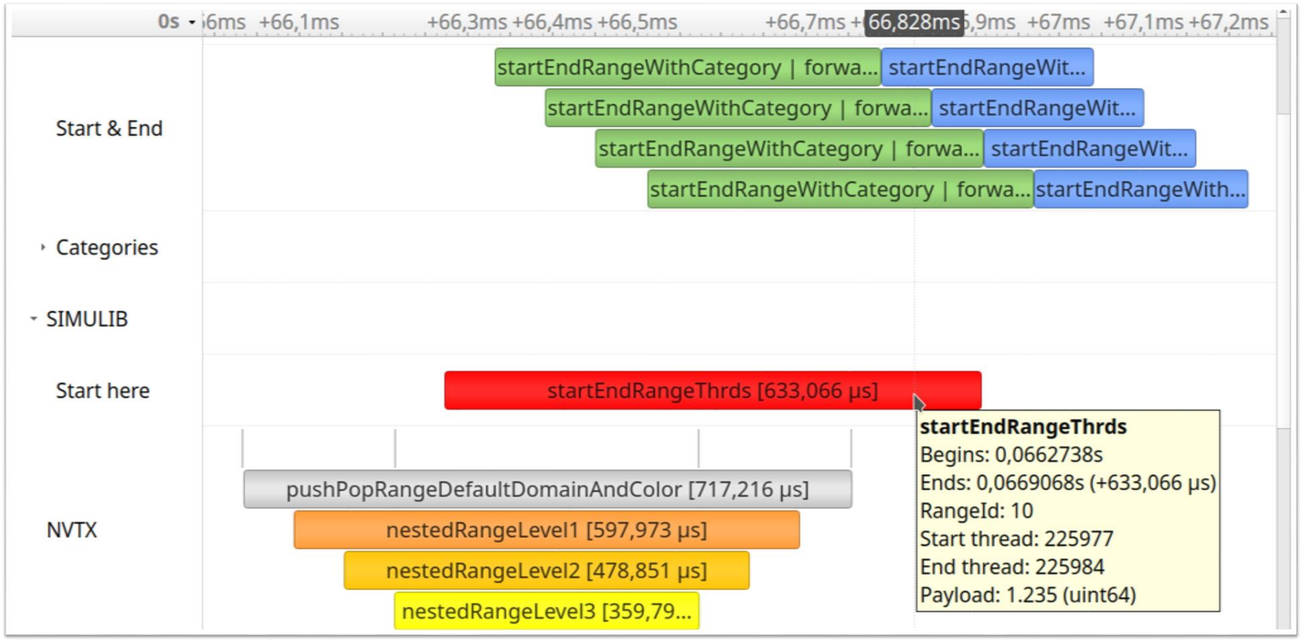
```
#include <nvtx3/nvToolsExt.h>
```

- **Marker**

```
nvtxMark("Point in time");
```

- **Push-Pop range**

```
nvtxRangePush(“Perfectly nested range");
// Do something interesting in the range.
nvtxRangePop();
```



- **Start-End range**

```
nvtxRangeId_t rangeId = nvtxRangeStart("This is a start/end range");
// Somewhere else in the code, not necessarily same thread as range start call
nvtxRangeEnd(rangeId);
```

- Provides concepts like domains, categories, colors, registered strings, and extended payloads (with user-defined layout)
- Sources: https://github.com/NVIDIA/NVTX/tree/release-v3/
- Enable Nsight Systems NVTX collection with --**trace nvtx** (enabled by default)

NVTX

Python and Fortran Bindings

- Python decorators `@nvtx.annotate()` or context manager "`with nvtx.annotate(...)`"

```
import nvtx

@nvtx.annotate("f()", color="purple")
def f():
    for i in range(5):
        with nvtx.annotate("loop", color="red"):
            # Python code goes here
```

- Get NVTX Python module: `python -m pip install nvtx`
- PyTorch CUDA provides NVTX bindings

```
from torch.cuda import nvtx

nvtx.range_push("YourCode")
# your Python code
nvtx.range_pop()
```

- Also see this blog or the docs: https://nvtx.readthedocs.io/en/latest/index.html

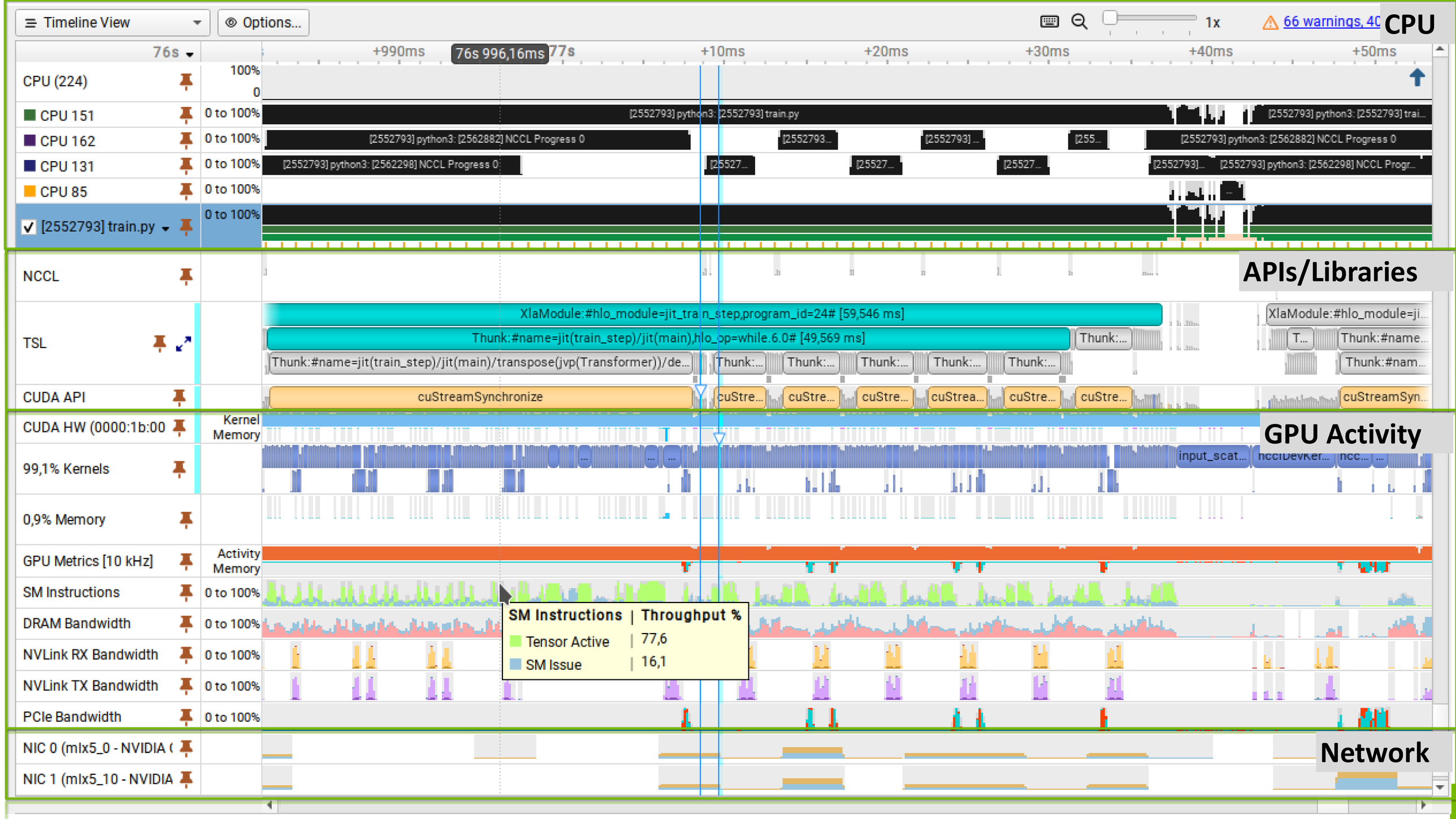
The NVIDIA HPC SDK Fortran compiler provides NVTX bindings.

- **`libnvhpcwrapnvtx.[a|so]`** must be linked
- Docs: https://docs.nvidia.com/hpc-sdk/compilers/fortran-cuda-interfaces/index.html#cfnvtx-runtime

```
use nvtx
…
call nvtxStartRange("YourRange")
! some Fortran code
call nvtxEndRange
```

- For other compilers, write your own Fortran NVTX bindings or use existing ones, e.g. https://raw.githubusercontent.com/maxcuda/NVTX_example/master/nvtx.f90

Nsight System
(from the GUI)


Timeline View
Options...
1x
66 warnings, 40
CPU
76s
+990ms
76s 996,16ms
77s
+10ms
+20ms
+30ms
+40ms
+50ms
CPU (224)
100%
0
CPU 151
0 to 100%
[2552793] python3: [2552793] train.py
CPU 162
[2552793] python3: [2562882] NCCL Progress 0
CPU 131
[2552793] python3: [2562298] NCCL Progress 0
CPU 85
[2552793] train.py
APIs/Libraries
NCCL
TSL
XlaModule:#hlo_module=jit_train_step,program_id=24# [59,546 ms]
Thunk:#name=jit(train_step)/jit(main),hlo_op=while.6.0# [49,569 ms]
Thunk:#name=jit(train_step)/jit(main)/transpose(jvp(Transformer))/de...
CUDA API
cuStreamSynchronize
GPU Activity
CUDA HW (0000:1b:00
Kernel
Memory
99,1% Kernels
0,9% Memory
GPU Metrics [10 kHz]
Activity
Memory
SM Instructions
DRAM Bandwidth
NVLink RX Bandwidth
NVLink TX Bandwidth
PCIe Bandwidth
SM Instructions | Throughput %
Tensor Active | 77,6
SM Issue | 16,1
Network
NIC 0 (mlx5_0 - NVIDIA
NIC 1 (mlx5_10 - NVIDIA

A First (I)Nsight

Recording with the CLI

- Use the command line
  - `srun nsys profile --trace=cuda,nvtx,mpi --force-overwrite=true --output=my_report.%q{SLURM_PROCID} \ ./jacobi -niter 10`
- Inspect results: Open the report file in the GUI
  - Also possible to get details on command line
  - Either add `--stats` to profile command line, or: `nsys stats --help`

- Runs set of reports on command line, customizable (**sqlite** + **Python**):
  - Useful to check validity of profile, identify important kernels

```
Running [.../reports/gpukernsum.py jacobi_metrics_more-nvtx.0.sqlite]...

 Time(%)  Total Time (ns)  Instances  Avg (ns)   Med (ns)   Min (ns)  Max (ns)  StdDev (ns)          Name
 -------  ---------------  ---------  ---------  ---------  --------  --------  -----------  --------------------
    99.9         36750359         20  1837518.0  1838466.5    622945   3055044    1245121.7  void jacobi_kernel
     0.1            22816          2    11408.0    11408.0      7520     15296       5498.5  initialize_boundaries
```

Exporting data

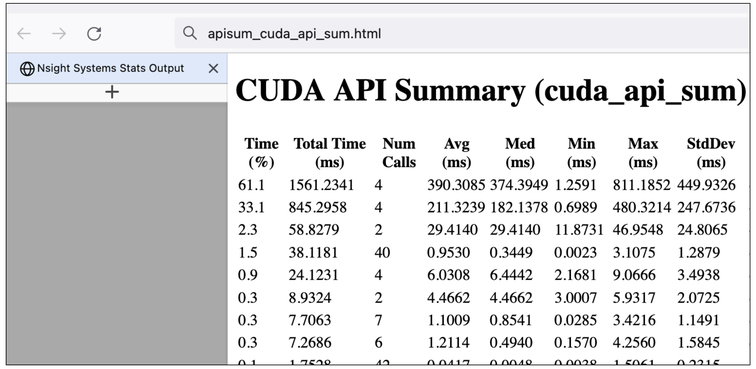
nsys stats, reports and export

- To sqlite, and directly into many human-readable formats

```
$ nsys stats –h
  .....
  Available formats (and file extensions):

  column     Human readable columns (.txt)
  table      Human readable table (.txt)
  csv        Comma Separated Values (.csv)
  tsv        Tab Separated Values (.tsv)
  json       JavaScript Object Notation (.json)
  hdoc       HTML5 document with <table> (.html)
  htable     Raw HTML <table> (.html)

  This option may be used multiple times. Multiple formats may also be specified using a comma-separated list.
```



Time (%)	Total Time (ms)	Num Calls	Avg (ms)	Med (ms)	Min (ms)	Max (ms)	StdDev (ms)
61.1	1561.2341	4	390.3085	374.3949	1.2591	811.1852	449.9326
33.1	845.2958	4	211.3239	182.1378	0.6989	480.3214	247.6736
2.3	58.8279	2	29.4140	29.4140	11.8731	46.9548	24.8065
1.5	38.1181	40	0.9530	0.3449	0.0023	3.1075	1.2879
0.9	24.1231	4	6.0308	6.4442	2.1681	9.0666	3.4938
0.3	8.9324	2	4.4662	4.4662	3.0007	5.9317	2.0725
0.3	7.7063	7	1.1009	0.8541	0.0285	3.4216	1.1491
0.3	7.2686	6	1.2114	0.4940	0.1570	4.2560	1.5845

```
nsys stats -r cuda_api_sum --timeunit ms
-f hdoc -o apisum jacobi.nsys-rep
```

- For other formats, e.g. HDF5:
  - `nsys export –t hdf my_file.nsys-rep`
  - Supports more (arrow, json, …): `nsys export -h`
- Full (extensive) documentation: https://docs.nvidia.com/nsight-systems/UserGuide/index.html#post-collection-analysis

Example: Available reports

Extending and customizing

- `nsys stats --help-reports`
  - list all available reports
- Example: CUDA API sum, customize to show only “Stream” APIs
  - `cp /opt/nvidia/nsight-systems/2024.1.1/host-linux-x64/reports/cuda_api_sum.py my_cuda_sum.py`
  - Editing `my_cuda_sum.py`, for example:

```
…
80 LEFT JOIN
81     StringIds AS ids
82     ON ids.id == summary.nameId
83 WHERE Name LIKE "%Stream%”
84 ORDER BY 2 DESC
…
```

- Running in the same way (reports from current folder picked up):
  - `nsys stats -r my_cuda_sum --timeunit ms jacobi_2024.1-0.nsys-rep`

```
Processing [jacobi_2024.1-0.sqlite] with [./my_cuda_sum.py]...

 ** CUDA API Summary (my_cuda_sum):

 Time (%)  Total Time (ms)  Num Calls  Avg (ms)  Med (ms)  Min (ms)  Max (ms)  StdDev (ms)          Name
 --------  ---------------  ---------  --------  --------  --------  --------  -----------  --------------------
      1.5          38.1181         40    0.9530    0.3449    0.0023    3.1075       1.2879  cudaStreamSynchronize
      0.0           0.4594         18    0.0255    0.0020    0.0018    0.4024       0.0941  cuStreamCreate
      0.0           0.0925         80    0.0012    0.0009    0.0006    0.0097       0.0012  cudaStreamWaitEvent
      0.0           0.0646          6    0.0108    0.0064    0.0043    0.0294       0.0098  cudaStreamDestroy
      0.0           0.0492          6    0.0082    0.0026    0.0024    0.0343       0.0128  cudaStreamCreate
      0.0           0.0457          2    0.0229    0.0229    0.0113    0.0344       0.0163  cuStreamDestroy_v2
      0.0           0.0101          4    0.0025    0.0025    0.0022    0.0029       0.0004  cuStreamSynchronize
```

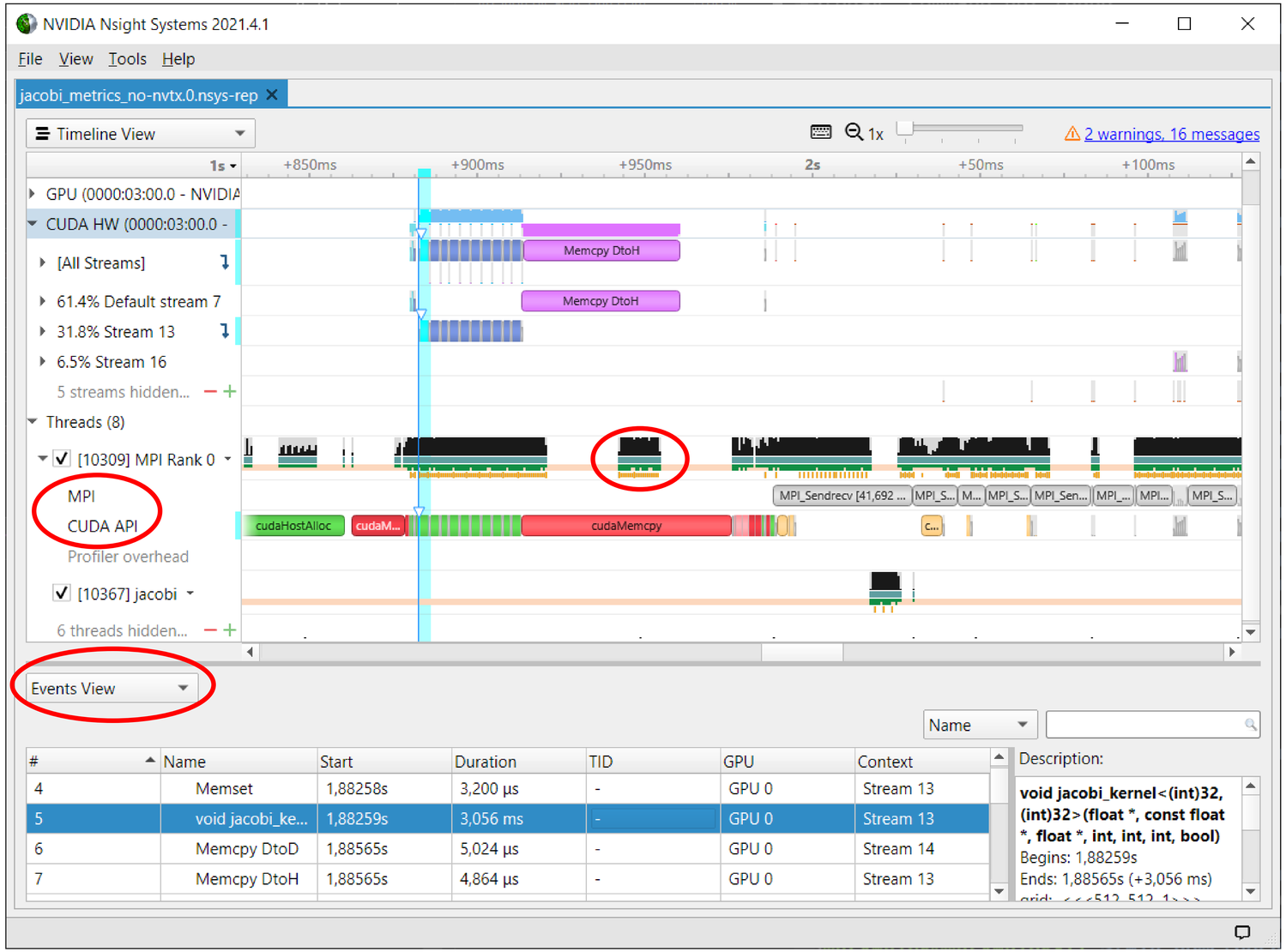
```
The following built-in reports are available:

  cuda_api_gpu_sum[:nvtx-name][:base|:mangled] -- CUDA Summary (API/Kernels/MemOps)
  cuda_api_sum -- CUDA API Summary
  cuda_api_trace -- CUDA API Trace
  cuda_gpu_kern_gb_sum[:nvtx-name][:base|:mangled] -- CUDA GPU Kernel/Grid/Block Summary
  cuda_gpu_kern_sum[:nvtx-name][:base|:mangled] -- CUDA GPU Kernel Summary
  cuda_gpu_mem_size_sum -- CUDA GPU MemOps Summary (by Size)
  cuda_gpu_mem_time_sum -- CUDA GPU MemOps Summary (by Time)
  cuda_gpu_sum[:nvtx-name][:base|:mangled] -- CUDA GPU Summary (Kernels/MemOps)
  cuda_gpu_trace[:nvtx-name][:base|:mangled] -- CUDA GPU Trace
  cuda_kern_exec_sum[:nvtx-name][:base|:mangled] -- CUDA Kernel Launch & Exec Time Summary
  cuda_kern_exec_trace[:nvtx-name][:base|:mangled] -- CUDA Kernel Launch & Exec Time Trace
  dx11_pix_sum -- DX11 PIX Range Summary
  dx12_gpu_marker_sum -- DX12 GPU Command List PIX Ranges Summary
  dx12_pix_sum -- DX12 PIX Range Summary
  mpi_event_sum -- MPI Event Summary
  mpi_event_trace -- MPI Event Trace
  network_congestion[:ticks_threshold=<ticks_per_ms>] -- Network Devices Congestion
  nvtx_gpu_proj_sum -- NVTX GPU Projection Summary
  nvtx_gpu_proj_trace -- NVTX GPU Projection Trace
  nvtx_kern_sum[:base|:mangled] -- NVTX Range Kernel Summary
  nvtx_pushpop_sum -- NVTX Push/Pop Range Summary
  nvtx_pushpop_trace -- NVTX Push/Pop Range Trace
  nvtx_startend_sum -- NVTX Start/End Range Summary
  nvtx_sum -- NVTX Range Summary
  nvvideo_api_sum -- NvVideo API Summary
  openacc_sum -- OpenACC Summary
  opengl_khr_gpu_range_sum -- OpenGL KHR_debug GPU Range Summary
  opengl_khr_range_sum -- OpenGL KHR_debug Range Summary
  openmp_sum -- OpenMP Summary
  osrt_sum -- OS Runtime Summary
  um_cpu_page_faults_sum -- Unified Memory CPU Page Faults Summary
  um_sum[:rows=<limit>] -- Unified Memory Analysis Summary
  um_total_sum -- Unified Memory Totals Summary
  vulkan_api_sum -- Vulkan API Summary
  vulkan_api_trace -- Vulkan API Trace
  vulkan_gpu_marker_sum -- Vulkan GPU Range Summary
  vulkan_marker_sum -- Vulkan Range Summary
  wddm_queue_sum -- WDDM Queue Utilization Summary

For more information, use '--help-reports <report_name>'
```

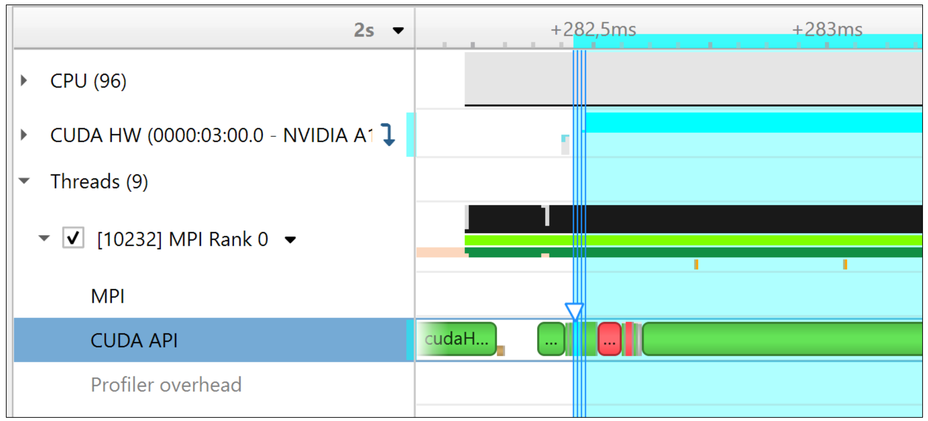
System-level Profiling with Nsight Systems

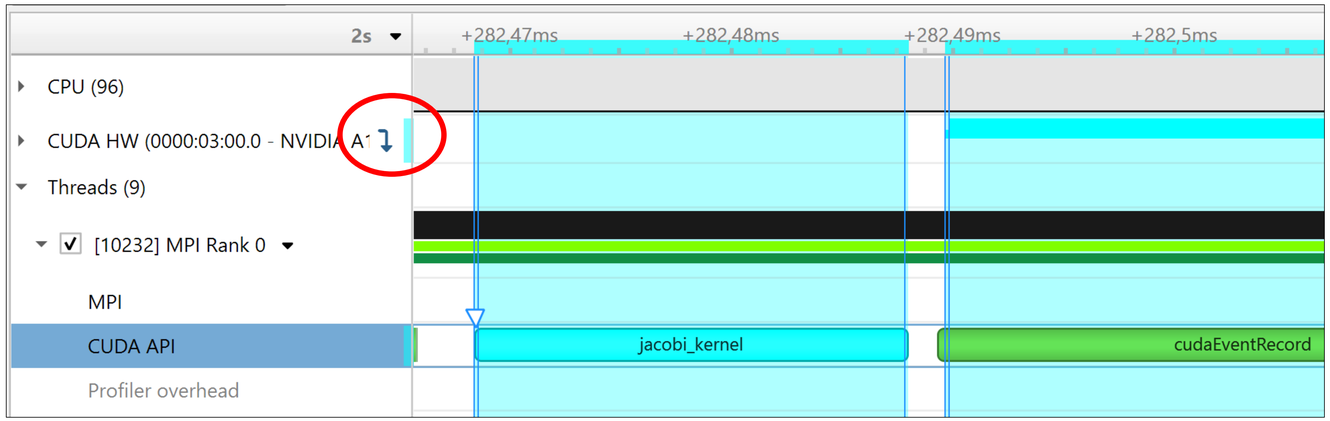
- Global timeline view
  - CUDA HW: streams, kernels, memory
- Different traces, e.g. CUDA, MPI
  - correlations API <-> HW
- Stack samples
  - bottom-up, top-down for CPU code
- GPU metrics
- Events View
  - Expert Systems
- looks at single process (tree)
  - correlate multi-process reports in single timeline



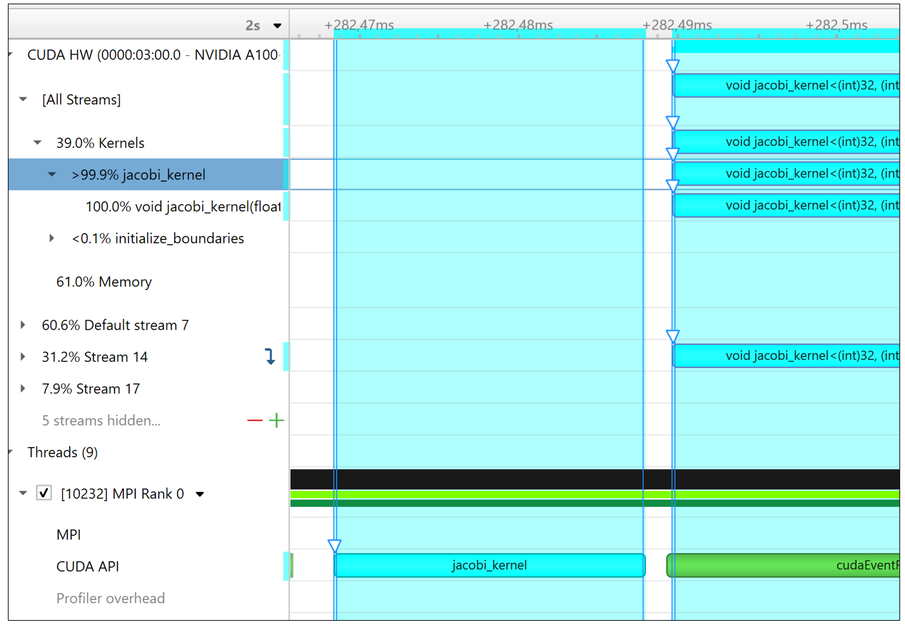
Correlating Events on the Timeline

Selecting events in one row highlights related events



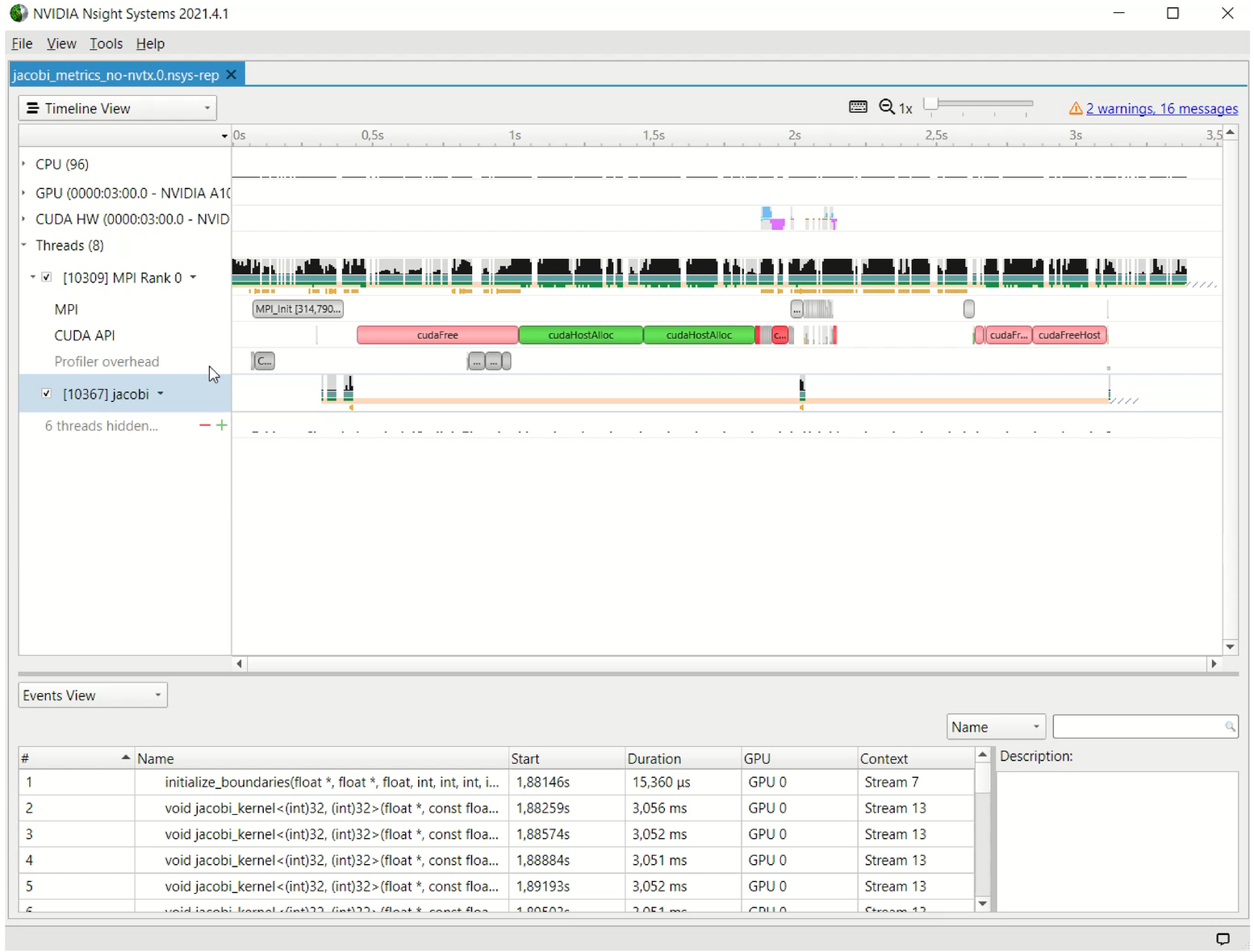


- CUDA's execution model is asynchronous
  - Kernel launch on host returns
  - Kernel runs on GPU
- Visualized in profiler



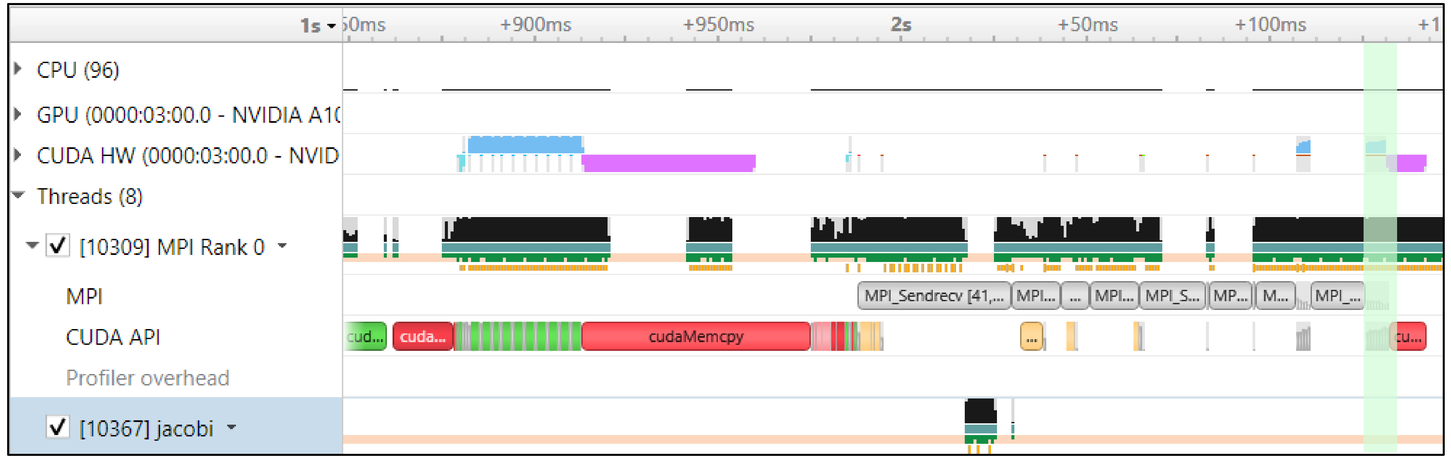
Nsight Systems Basic Workflow

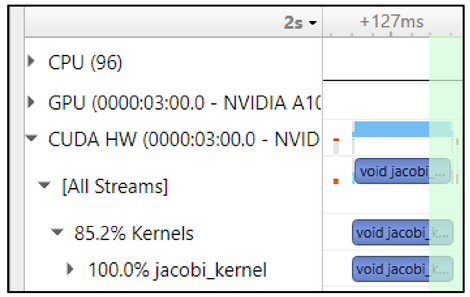
Navigating the timeline and finding interesting areas

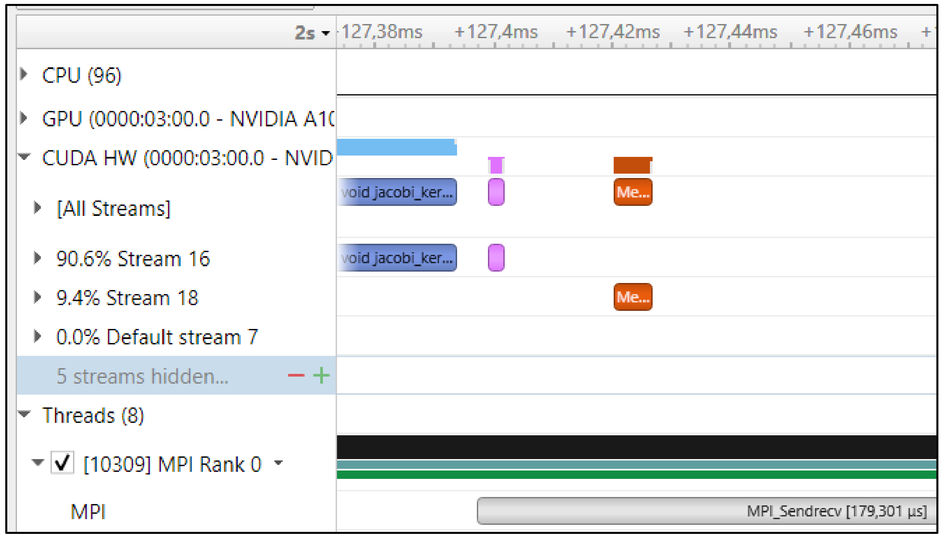


Discovering Optimization Potential

- Using Jacobi solver example*
- Spot kernels – lots of whitespace
  - Which part is „bad“?
  - Enhance!
- MPI calls
  - Memory copies
  - We know: This is CUDA-aware MPI
- Even without knowing source, insight
- Too complicated for repeated/reliable usage
  - How to simplify navigating and comparing reports?







*See https://github.com/NVIDIA/multi-gpu-programming-models/

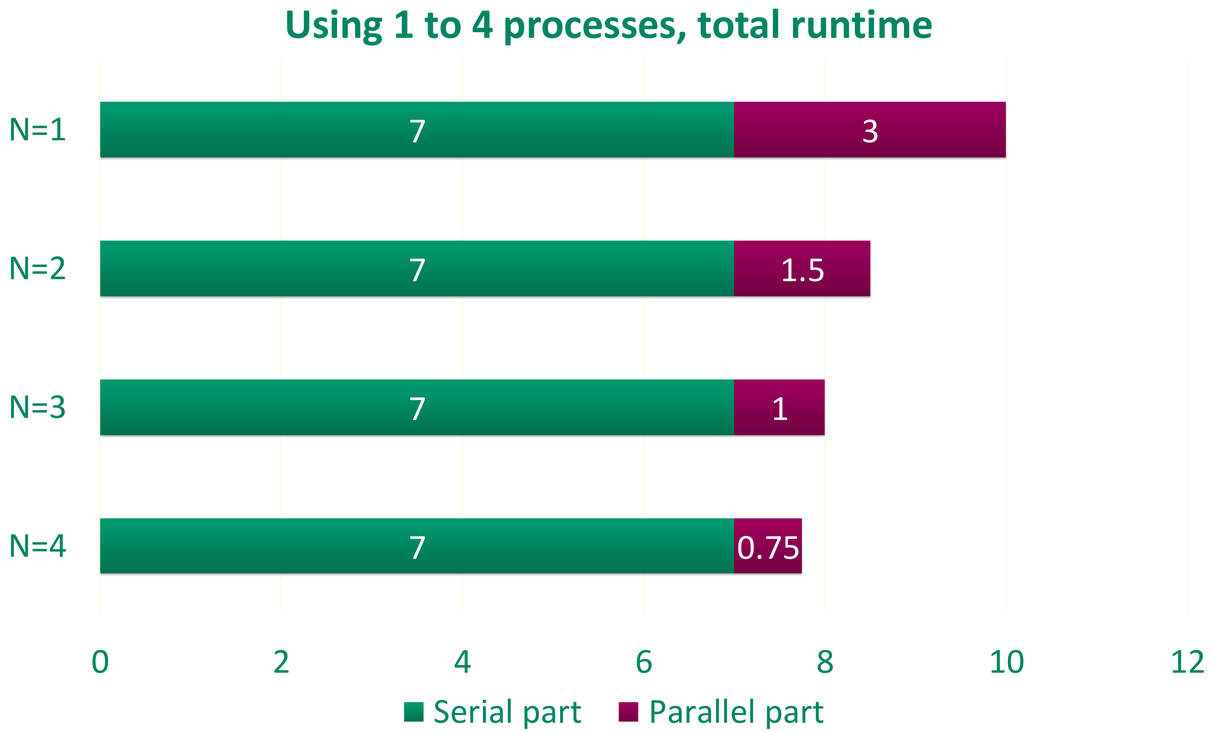
Interlude - Maximum Achievable Speedup

Amdahl‘s law

- Amdahl‘s law states overall speedup s given the parallel fraction p of code and number of processes N

$$\boldsymbol{s} = \frac{1}{1 - p + \frac{p}{N}} < \frac{1}{1 - p}$$

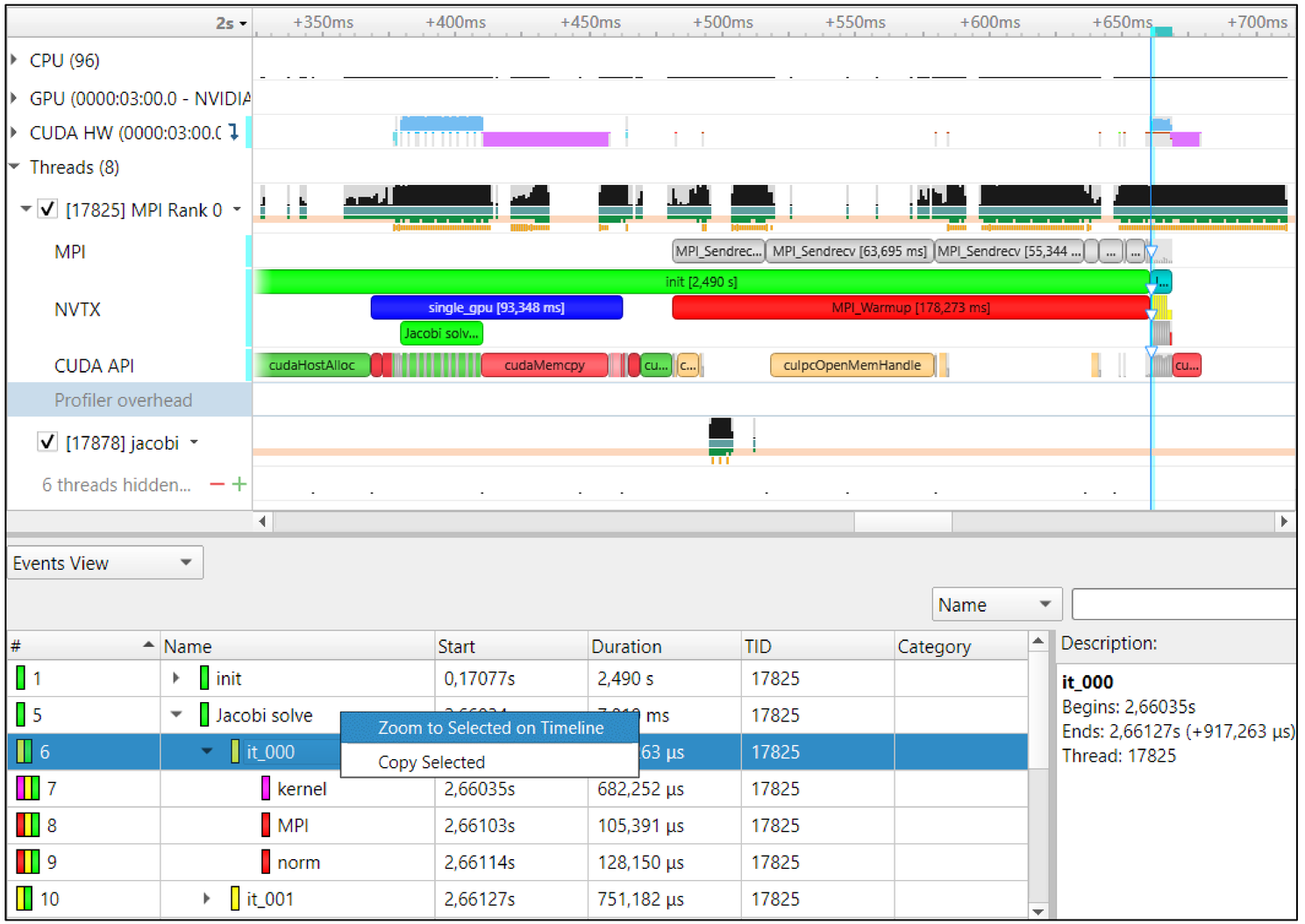
- Limited by serial fraction, even for $N \to \infty$
- Example for $p = 30\%$
- Generally applicable on any level
  - e.g. also valid for per-method speedups



Adding some Color

Code annotation with NVTX

- Same section of timeline as before
  - Events view: Quick navigation
- Like manual timing, only less work
- Nesting
- Correlation, filtering



Adding NVTX

Simple range-based API

- `#include "nvtx3/nvToolsExt.h"`
  - NVTX v3 is header-only, needs just `-ldl`
  - C++ and Python APIs
- Fortran: NVHPC compilers include module
  - Just `use nvtx` and `-lnvhpcwrapnvtx`
  - Other compilers: See blog posts linked below
- Definitely: Include `PUSH/POP` macros (see links below)

  `PUSH_RANGE(`*`name, color_idx`*`)`
- Sprinkle them strategically through code
  - Use hierarchically: Nest ranges
- Not shown: Advanced usage (domains, …)
- Similar range-based annotations exist for other tools
  - e.g. SCOREP_USER_REGION_BEGIN

```
int main(int argc, char** argv) {
    PUSH_RANGE("main", 0)
    PUSH_RANGE("init", 1)
    do_initialization();
    POP_RANGE
    /* ... */
    PUSH_RANGE("computation", 2)
    jacobi_kernel<<</* ... */, compute_stream>>>(...);
    cudaStreamSynchronize(compute_stream);
    POP_RANGE
    /* ... */
    POP_RANGE
}
```

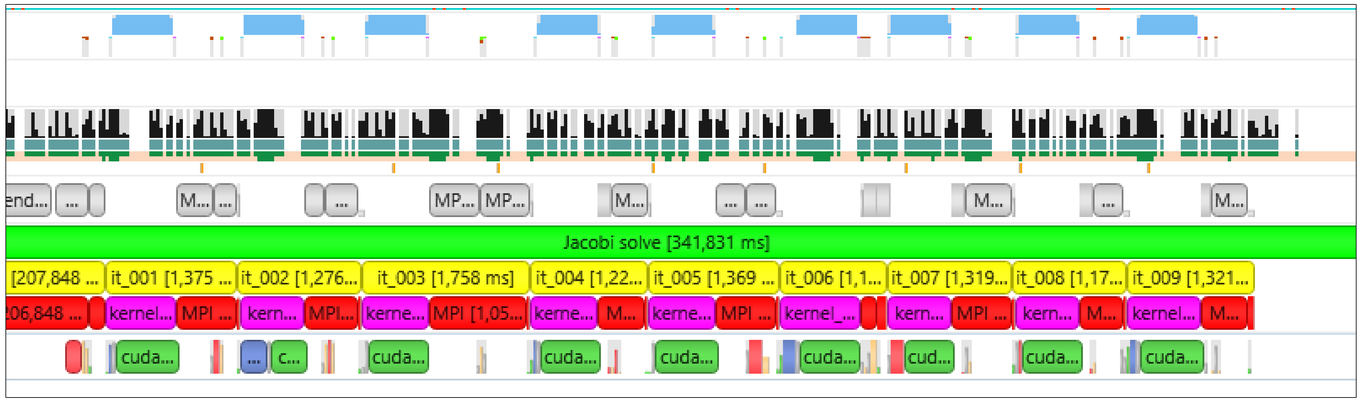
https://github.com/NVIDIA/NVTX and https://nvidia.github.io/NVTX/#how-do-i-use-nvtx-in-my-code

https://developer.nvidia.com/blog/cuda-pro-tip-generate-custom-application-profile-timelines-nvtx/
https://developer.nvidia.com/blog/customize-cuda-fortran-profiling-nvtx/

Minimizing Profile Size

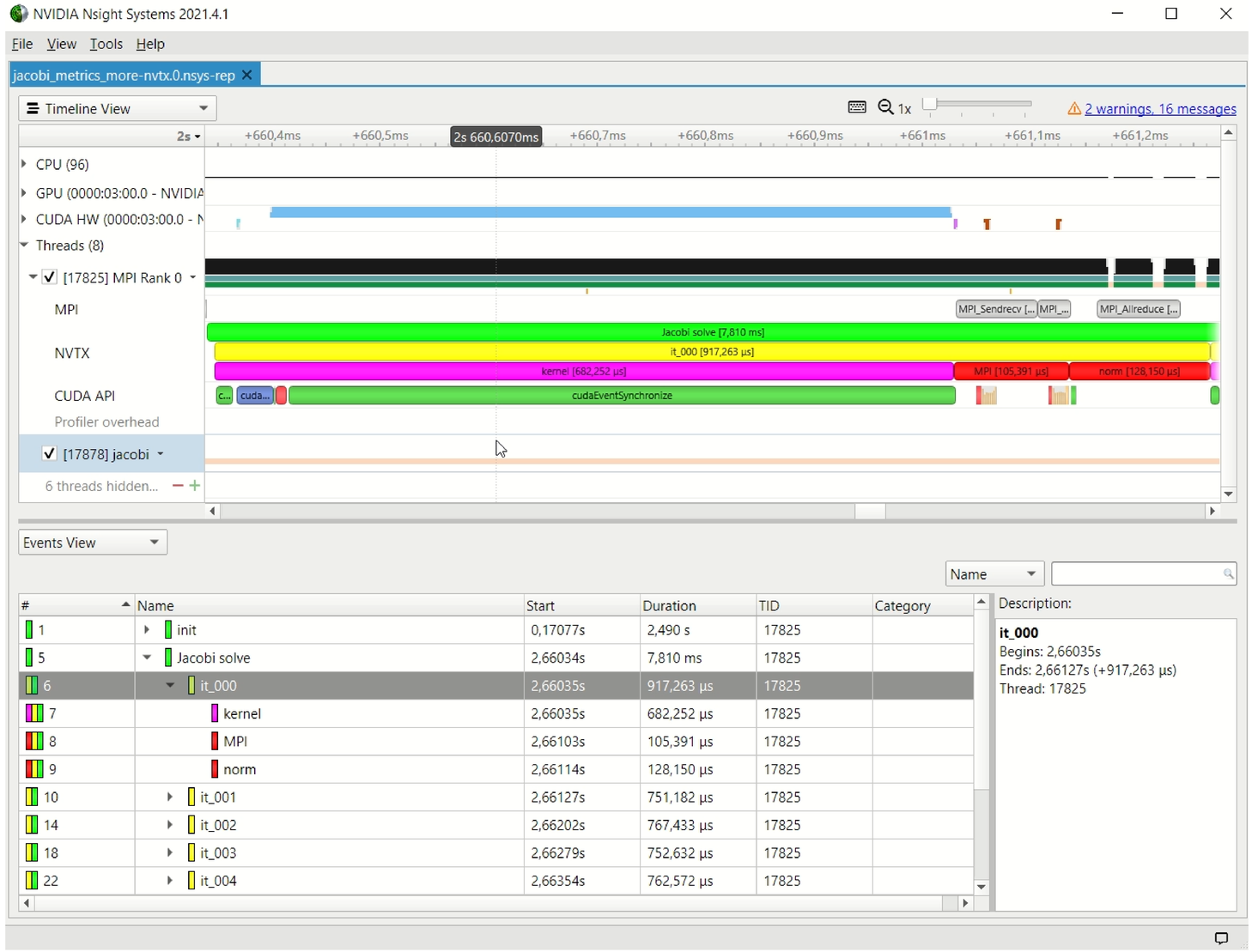
Shorter time, smaller files = quicker progress

- Only profile what you need – all profilers have some overhead
  - Example: Event that occurs after long-running setup phase
- Bonus: lower number of events leads to smaller file size
- Add to nsys command line:
  - `--capture-range=nvtx --nvtx-capture=any_nvtx_marker_name \`
    `--env-var=NSYS_NVTX_PROFILER_REGISTER_ONLY=0 --kill none`
    - Use NVTX registered strings for best performance
- Alternatively: `cudaProfilerStart()` and `-Stop()`
  - `--capture-range=cudaProfilerApi`



Nsight Systems Workflow with NVTX

Repeating the analysis



#	Name	Start	Duration	TID	Category
1	init	0,17077s	2,490 s	17825	
5	Jacobi solve	2,66034s	7,810 ms	17825	
6	it_000	2,66035s	917,263 µs	17825	
7	kernel	2,66035s	682,252 µs	17825	
8	MPI	2,66103s	105,391 µs	17825	
9	norm	2,66114s	128,150 µs	17825	
10	it_001	2,66127s	751,182 µs	17825	
14	it_002	2,66202s	767,433 µs	17825	
18	it_003	2,66279s	752,632 µs	17825	
22	it_004	2,66354s	762,572 µs	17825	

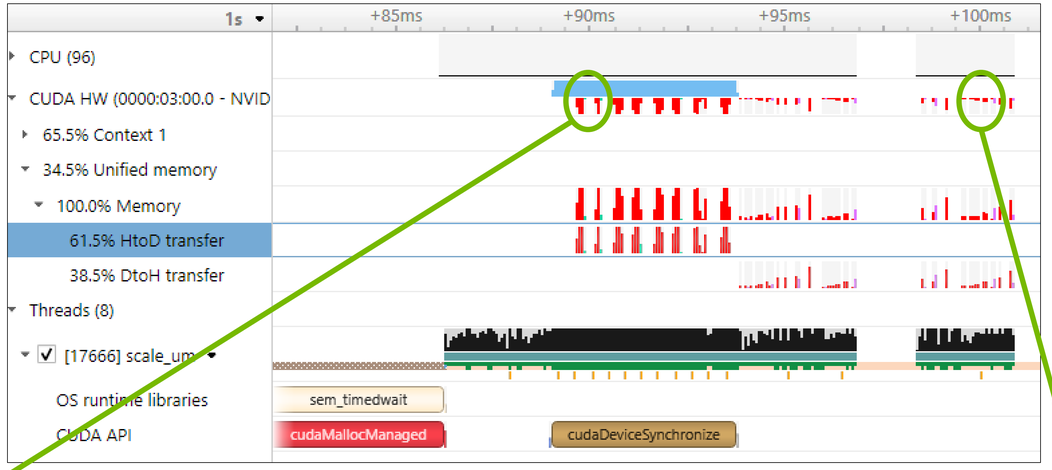
GPU Metrics in Nsight Systems

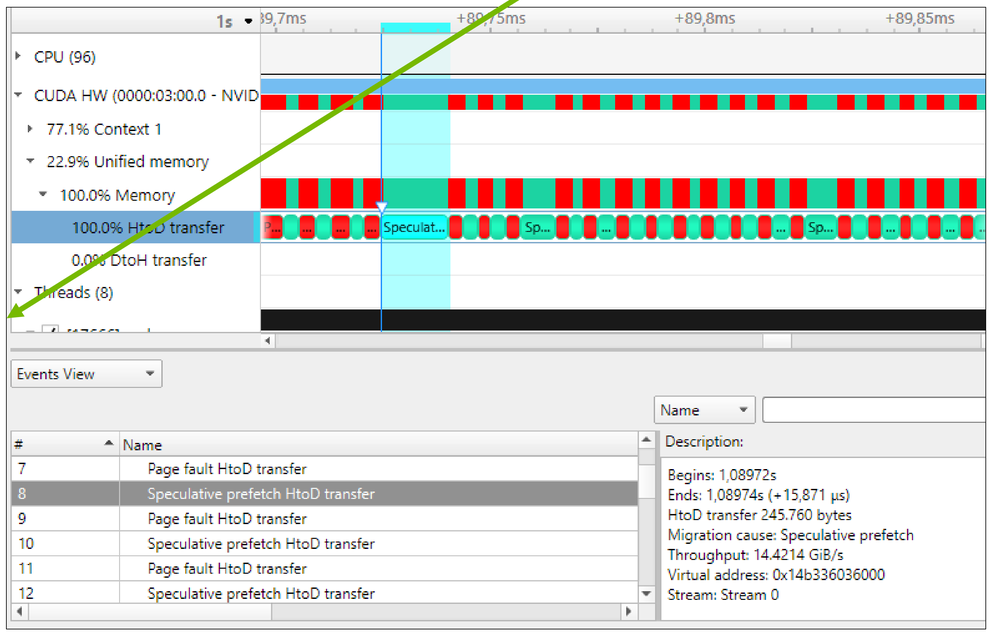
...and other traces you can activate

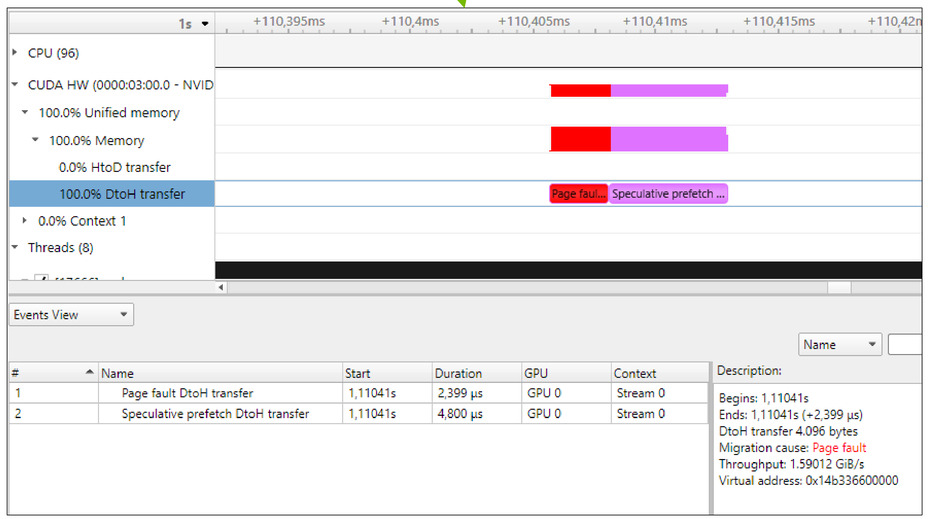
- Valuable low-overhead insight into HW usage:
  - SM instructions
  - DRAM Bandwidth, PCIe Bandwith (GPUDirect)
- Also: Memory usage, Page Faults (higher overhead)
  - CUDA Programming guide: **Unified Memory Programming**
- Can save kernel-level profiling effort!
- `nsys profile --gpu-metrics-device=0 --cuda-memory-usage=true --cuda-um-cpu-page-faults=true --cuda-um-gpu-page-faults=true ./app`

Unified Memory movement

Observing transfers in Nsight Systems



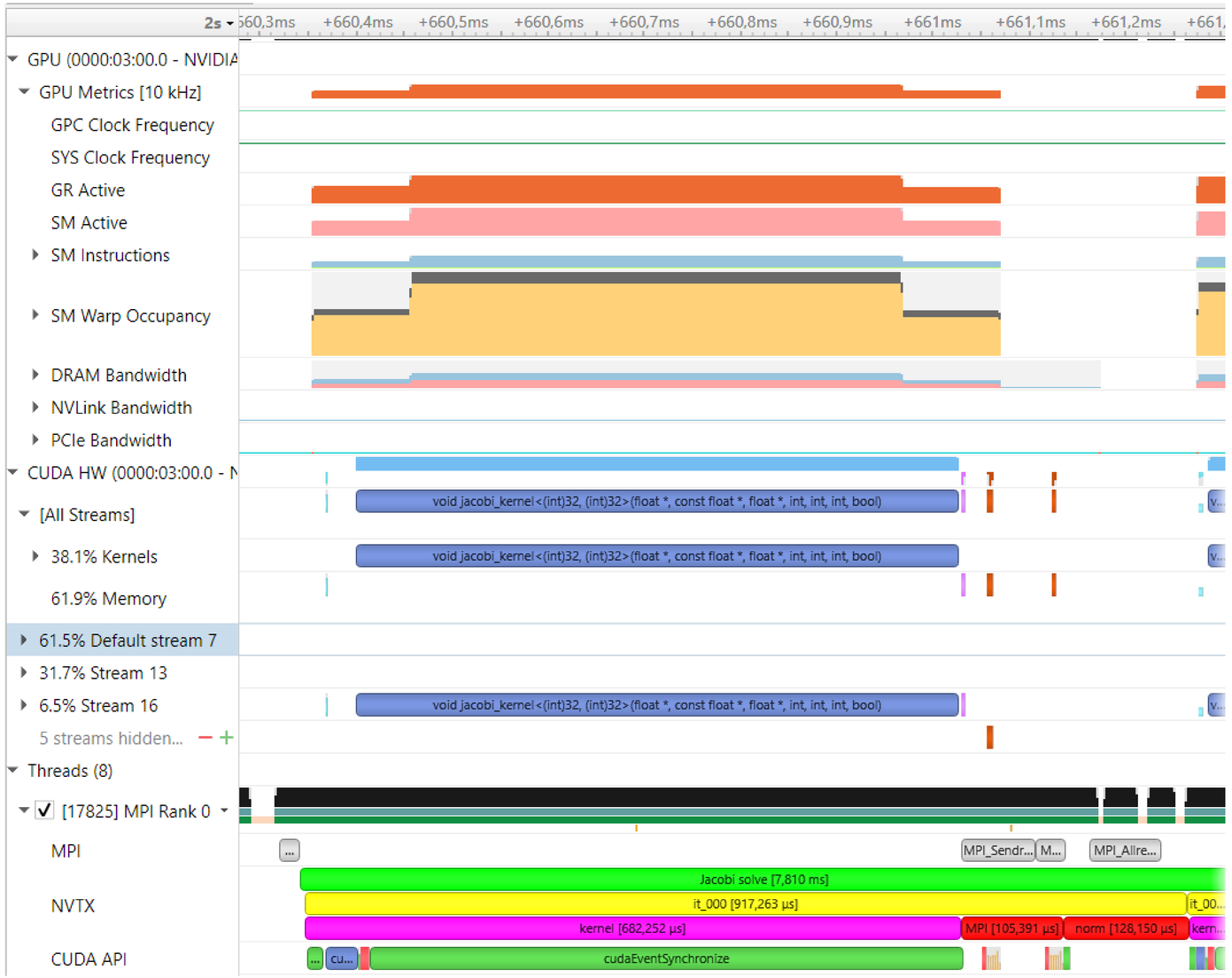




Focusing the Analysis

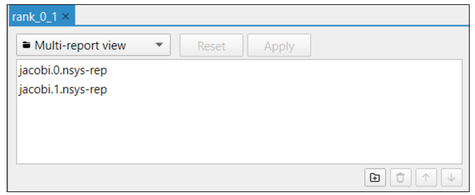
Introducing GPU metrics sampling

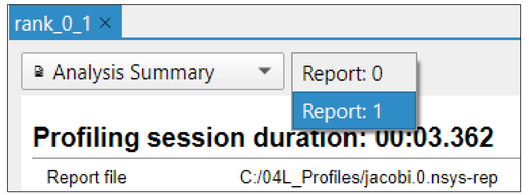
- Discover the „unit cell“ of performance
  - in our case: single iteration
- Other blank spots during setup can be ignored (amortized, many more iterations)
- Maybe: Too small for proper comms profiling
- Kernel itself adequately using GPU
  - Remaning blank spots?
- Norm calculation
  - Can be turned off
- But still: Overlap potential? Can we run kernel during MPI?



Multi-process GPU Analysis

- Load multiple reports into timeline
  - analyze differences in execution, GPU utilization
- Pin rows for comparison
- Example: End time of kernel execution

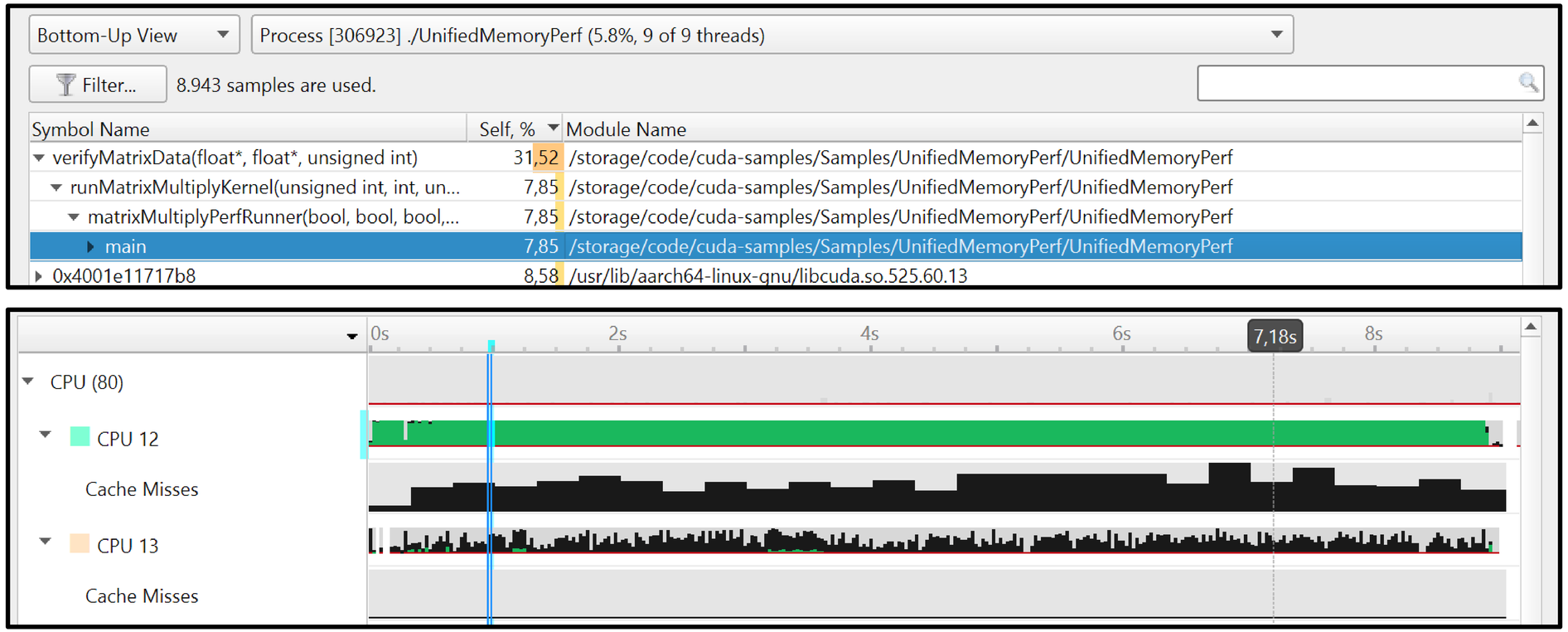




CPU Profiling

Analyzing host-side call trees

- CPU bottom-up (or top-down) profiling based on samples
- Linux perf events, e.g. cache misses
  - https://docs.nvidia.com/nsight-systems/UserGuide/index.html#cpu-linuxperf



Nsight System
(from the Command Line)

CLI Profiling Switches

Tracing and output

- API tracing

  `-t, --trace=`**`cuda,nvtx,osrt,opengl`**

  `(cublas,cusparse,cudnn,mpi,oshmem,ucx,openacc,openmp,vulkan,nvvideo,python-gil,none)`

- Summary statistics (profile output on command line)

  `--stats=[true|`**`false`**`]`

- Report file name

  `-o, --output=report#`
  (patterns for hostname, PID `%p` and environment variables `%q{ENV_VAR}`)

- Overwrite existing report

  `-f, --force-overwrite=[true|`**`false`**`]`

(Use **`nsys profile --help`** for a list of available options.)

CLI Profiling Switches

Limit data recording

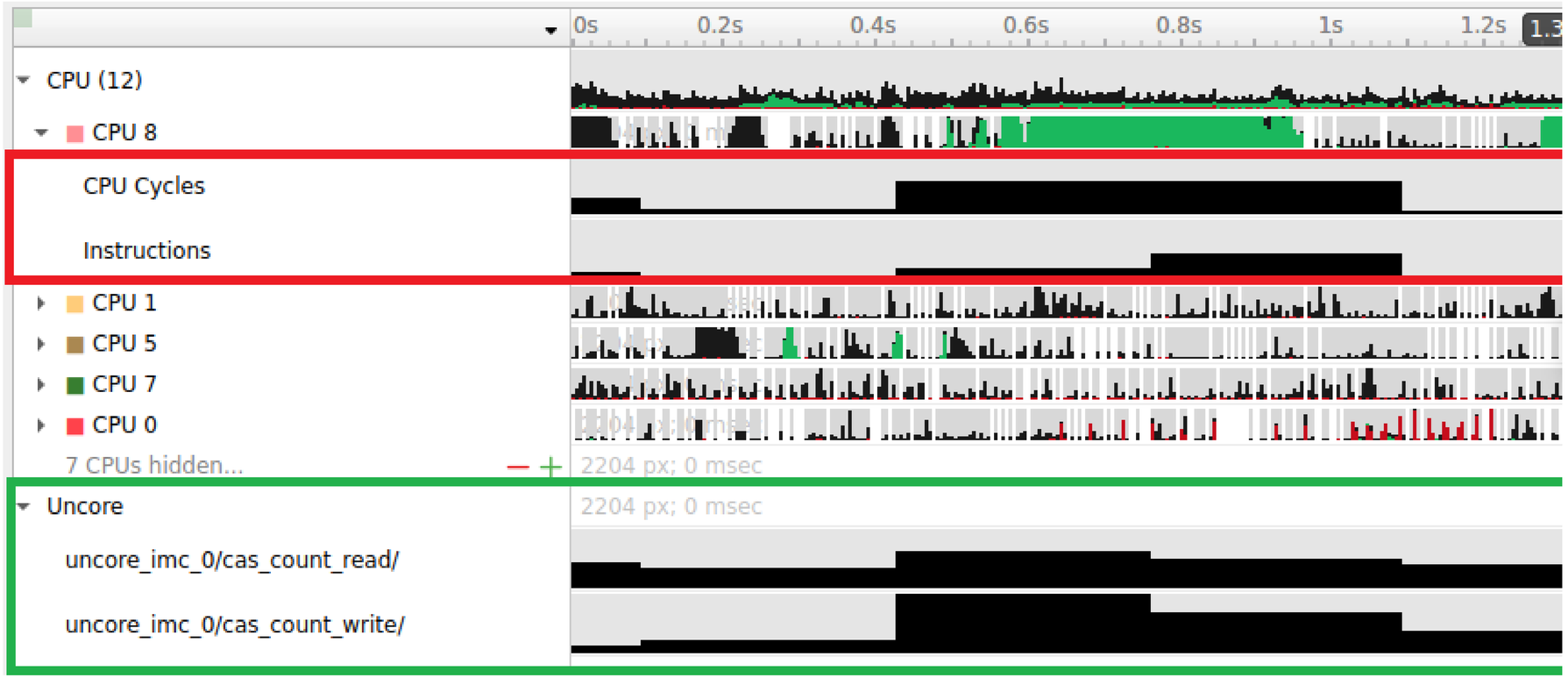
- Capture range trigger
  `-c, --capture-range={`**`none`**`,cudaProfilerApi,nvtx}`
  `-p, --nvtx-capture=`range@domain' (only applicable for `-c nvtx`)
- Capture range end behavior
  `--capture-range-end={none,stop,`**`stop-shutdown`**`,repeat[:N],repeat-shutdown:N}`
- Profiling session end
  `--wait={primary|`**`all`**`}` (wait for primary process or also on re-parented processes)
  `--kill={none|sigkill|`**`sigterm`**`|<signal number>}` (send signal to target app)
- Collection start delay and duration
  `-y, --delay=<seconds>` (default: `0`)
  `-d, --duration<seconds>` (default: `0`)
- NVTX domain filtering
  `--nvtx-domain-[include|exclude]=<comma-separated list of NVTX domains>`
- Reduce the sampling rate
  `--sampling-period=<number of reference cycles events>` (default: `4500000`)
  `--event-sampling-interval=<milliseconds>` (default: `10`)
  `--gpu-metrics-frequency=<Hz>` (default: `10000`)
  `--python-sampling-frequency=<Hz>` (default: `1000`)

CPU Counter Sampling

- Grace, ARM, x86-64
- CPU/SoC uncore
  - LLC
  - DRAM
  - PCIe
  - C2C
  - SCF
- Kernel tracepoints
- System-wide
- CPU core and socket events and metrics

```
--event-sample=<none|system-wide>
--event-sampling-interval=<10|100|…>  [ms]
--cpu-{core,socket}-{events,metrics}=<2,3,…>
                                     =help
```

for a full list of events



Demo

How to use nsys

DAY6/nsys-wrapper

Option1: profile everything in a node

```
srun nsys profile -t mpi,nvtx,cuda --stats=true gmx_mpi mdrun -ntomp ${OMP_NUM_THREADS} -noconfout -nsteps 16000 -nstlist 300 -nb gpu -update gpu -pme gpu -npme 1 -dlb no -v
```

Option 2: profile a specific rank in a multi-node multi-process execution

```
#!/bin/bash

if [[ $SLURM_PROCID == 0 ]]; then
        nsys profile -t mpi,nvtx,cuda --stats=true $*
else
        $*
fi
```

```
srun ./wrap_nsys.sh gmx_mpi mdrun -ntomp ${OMP_NUM_THREADS} -noconfout -nsteps 16000 -nstlist 300 -nb gpu -update gpu -pme gpu -npme 1 -dlb no -v
```

THANK YOU!

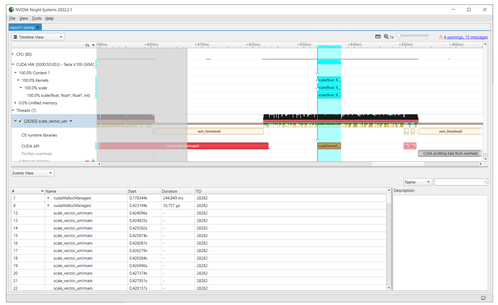
Download	https://developer.nvidia.com/nsight-systems **NOTE**: website version is newer than CUDA Toolkit version
Docs	https://docs.nvidia.com/nsight-systems/index.html
Forums	https://devtalk.nvidia.com/default/board/308/nsight-systems/
Email	nsight-systems@nvidia.com
Blogs	https://developer.nvidia.com/blog/nvidia-nsight-systems-containers-cloud https://developer.nvidia.com/blog/nsight-systems-exposes-gpu-optimization https://developer.nvidia.com/blog/understanding-the-visualization-of-overhead-and-latency-in-nsight-systems https://developer.nvidia.com/blog/nvidia-tools-extension-api-nvtx-annotation-tool-for-profiling-code-in-python-and-c-c/

Nsight Compute

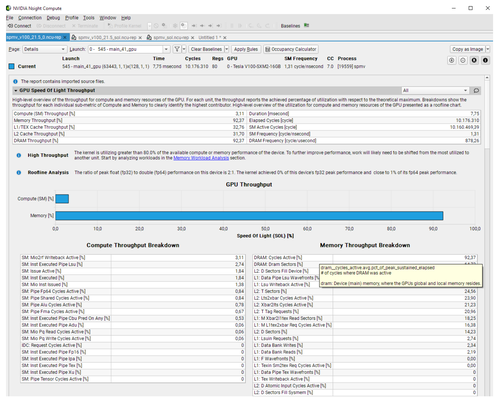
Switching the Analysis Focus

Diving into a kernel

- Optimization: Always tradeoff between slightly conflicting goals
  - Performance; Maintainability; Effort
- Start with a system-level view → Nsight Systems
- Ensure understanding of timeline, and where the GPU is active/inactive
  - where initialization happens
  - how the time-% shifts for different relevant workloads



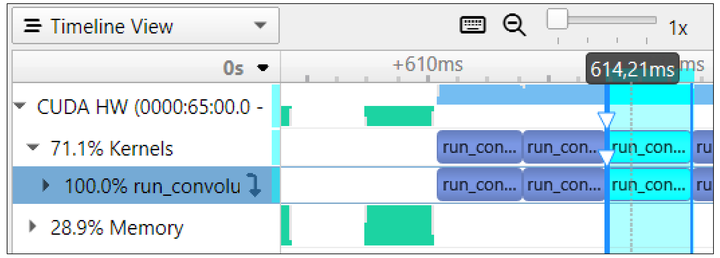
- Don‘t shy away from kernel-level analysis, but ensure impact is understood
  - Again, Amdahl‘s: Hypothetically, optimized kernel takes 0 s, how large is whole-program speedup?
- General guidelines – if whole timeline *is* a single kernel, by all means start with it first!
  - We will now look at **Nsight Compute** usage

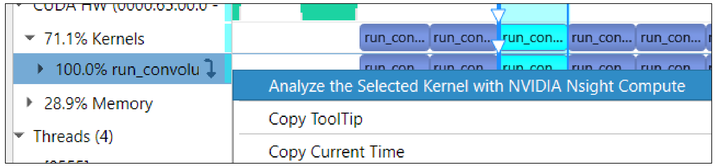


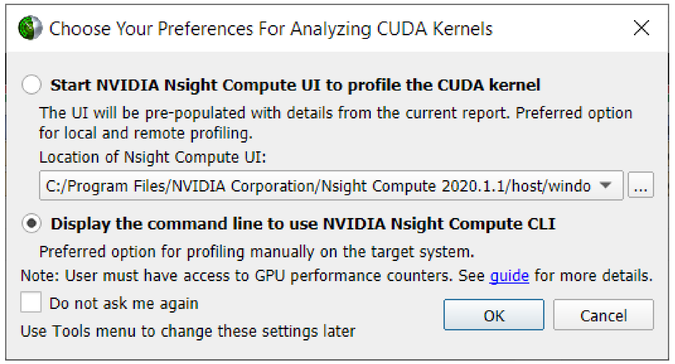
Drilling Down on a Kernel

Analysis with Nsight Compute

- Simple shortcut: Right-click menu in Nsight Systems







- Run command line
  - `ncu --set full --import-source on --kernel-name regex:PartialName --launch-skip 11 --launch-count 1 -f -o my_report ./my_application ...`
  - `regex:PartialName` important for complicated names (templates), or selecting multiple kernels: `Kernel(A|B)`
  - `--kernel-name-base demangled` for templates
- Important switches for metrics collection, pre-selected sets
- Fully customizable, `ncu --help`. Check `--list-metrics` and `--query-metrics`
- We use GUI for analysis and load report file
  - Alternatively, interactively run and analyze directly through GUI

Example: Generational Changes

Know your Hardware

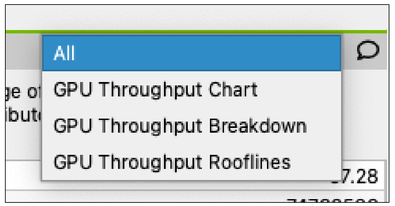
	V100	A100
SMs	80	108
Tensor Core Precision	FP16	FP64, TF32, BF16, FP16, I8, I4, B1
Unified L1/Shared Memory per SM	128 KB	192 KB
L2 Cache Size	6144 KB	40960 KB
Memory Bandwidth	900 GB/sec	1555 GB/sec (40GB) 2039 GB/sec (80GB)
NVLink Interconnect	300 GB/sec	600 GB/sec

https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/a100/pdf/nvidia-a100-datasheet-us-nvidia-1758950-r4-web.pdf

First Steps in Kernel Analysis

Understanding initial limiter

- GPU „Speed of Light Throughput“
  - SOL = theoretical peak
- „Breakdown“ tables
  - DRAM: Cycles Active
  - Switch to „All“ to see tables:



- Tooltips
- Rules point to next steps

transpose_kernel_sol.ncu-rep

	Result	Size	Time	Cycles	GPU	SM Frequency	Process	Attributes
Current	ng *, const long long *, long long)	(256, 256, 1)x(32, 32, 1)	807.17 us	881.317	0 - NVIDIA A100-SXM4-40GB	1.09 Ghz	[4635] transpose	

Summary | Details | Source | Context | Comments | Raw | Session

Compare | Tools | View | Export

The report contains imported source files.

GPU Speed Of Light Throughput — All

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

Compute (SM) Throughput [%]	20.39	Duration [us]	807.17
Memory Throughput [%]	84.56	Elapsed Cycles [cycle]	881317
L1/TEX Cache Throughput [%]	23.26	SM Active Cycles [cycle]	878381.44
L2 Cache Throughput [%]	65.97	SM Frequency [Ghz]	1.09
DRAM Throughput [%]	84.56	DRAM Frequency [Ghz]	1.21

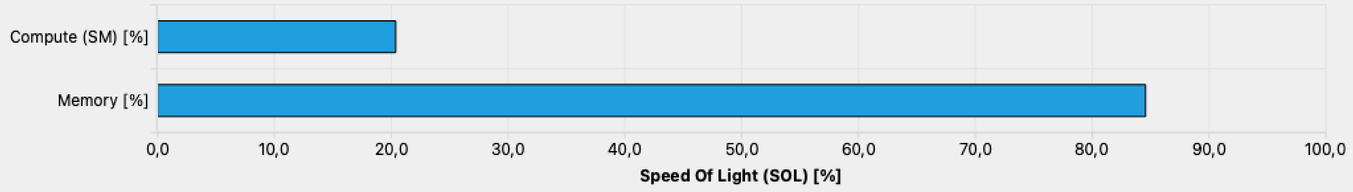
High Throughput The kernel is utilizing greater than 80.0% of the available compute or memory performance of the device. To further improve performance, work will likely need to be shifted from the most utilized to another unit. Start by analyzing DRAM in the Memory Workload Analysis section.

Compute Bottleneck Detect bottlenecks arising from compute capabilities — Apply

Memory Bottleneck Detect bottlenecks arising from memory capabilities — Apply

Roofline Analysis The ratio of peak float (fp32) to double (fp64) performance on this device is 2:1. The kernel achieved 0% of this device's fp32 peak performance and 0% of its fp64 peak performance. See the Kernel Profiling Guide for more details on roofline analysis.

GPU Throughput



Compute Throughput Breakdown

SM: Issue Active [%]	20.39
SM: Inst Executed [%]	20.38
SM: Inst Executed Pipe Adu [%]	13.22
SM: Pipe Fma Cycles Active [%]	13.22
SM: Pipe Alu Cycles Active [%]	13.22
SM: Inst Executed Pipe Lsu [%]	13.22
SM: Mio Inst Issued [%]	9.92
SM: Mio Rq Write Cycles Active [%]	8.81

Memory Throughput Breakdown

DRAM: Cycles Active [%]	84.56
DRAM: Dram Sectors [%]	48.94
L2: T Tag Requests [%]	24.77
L1: Data Pipe Lsu Wavefronts [%]	23.19

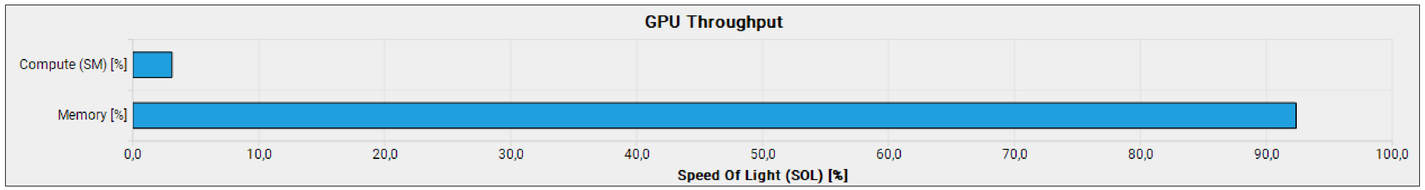
dram__cycles_active.avg.pct_of_peak_sustained_elapsed
# of cycles where DRAM was active
(This counter metric represents the avg percent of the peak sustained rate achieved during e

dram: Device (main) memory, where the GPUs global and local memory resides.

First Steps in Kernel Analysis

Understanding initial limiter

- GPU „Speed of Light Throughput“
- „Breakdown“ tables
  - DRAM: Cycles Active
- Tooltips
- Rules point to next steps



Memory Throughput Breakdown

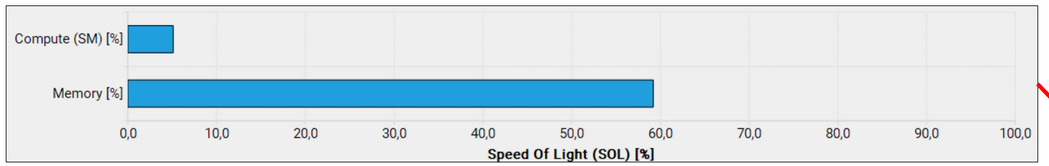
DRAM: Cycles Active [%]	92,37
DRAM: Dram Sectors [%]	64,72
L2: D Sectors Fill Device [%]	31,70
L1: Data Pipe Lsu Wavefronts [%]	26,11

dram__cycles_active.avg.pct_of_peak_sustained_elapsed
# of cycles where DRAM was active
(This counter metric represents the avg percent of the peak sustained rate achieved during elapsed cycles across all sub-unit instances)

dram: Device (main) memory, where the GPUs global and local memory resides.

Kernel-level Profiling

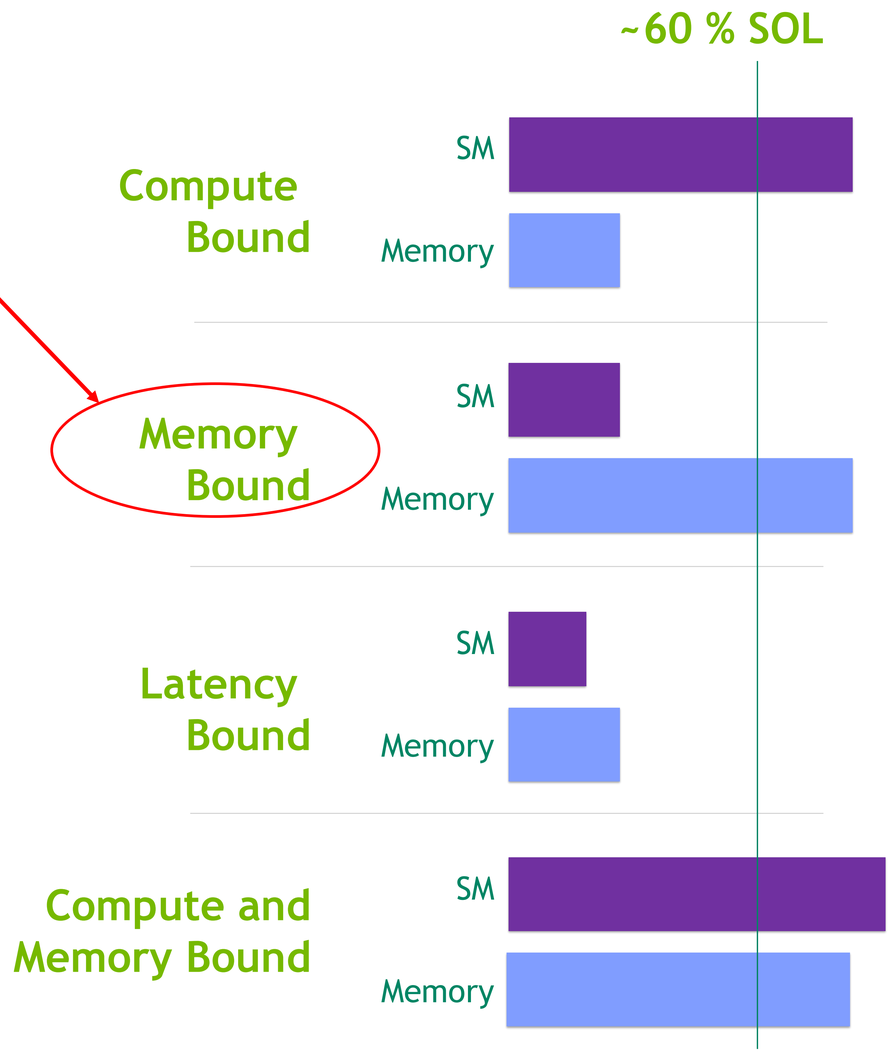
Performance limiter categories



- Four possible combinations of high/low...
- ...memory utilization
- ...compute utilization

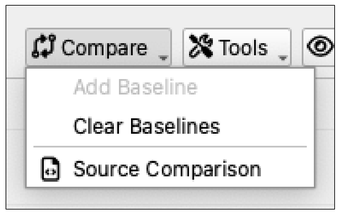
- Good? Bad?

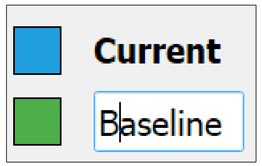
→ Depends on problem and its implementation



Comparing Performance with Baselines

- Part of workflow: Repeat analysis – compare with previous version





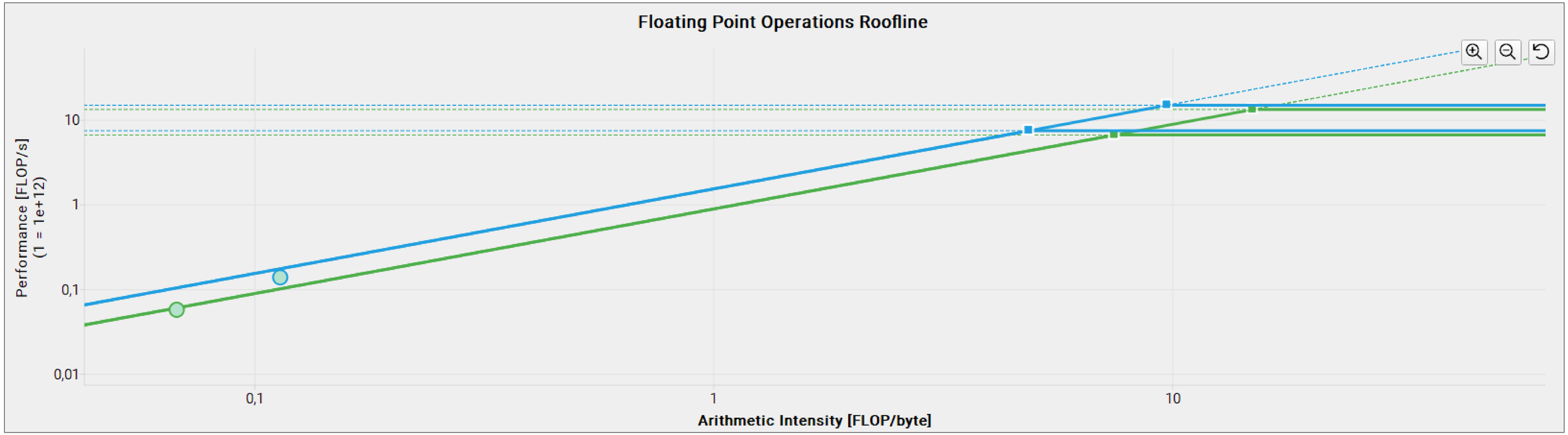
- Baselines window to keep them organized: Turn on/off, save and load

Baselines

		Name	Launch	Report	Time	Cycles	Registers	GPU	SM Frequency	CC
☑		SPMV on V100	545 - main_41_gpu (63443, 1, 1)x(12...	C:/Users/mhrywniak/On...	7,75 msecond	10.176.310	80	0 - Tesla V100-S...	1,31 cycle/nsec...	7.0
☐		SPMV_v2 on V100	545 - main_41_gpu (65535, 1, 1)x(12...	C:/Users/mhrywniak/On...	17,93 msecond	23.481.264	32	0 - Tesla V100-S...	1,31 cycle/nsec...	7.0

Builtin Roofline Chart

- Comparisons between code versions, hardware versions
  - Increase AI (reuse), shift to right
  - Below roofline? Find and fix limiter, latency issues, ...
- From hardware counter metrics
  - Here: Same code on V100 and A100



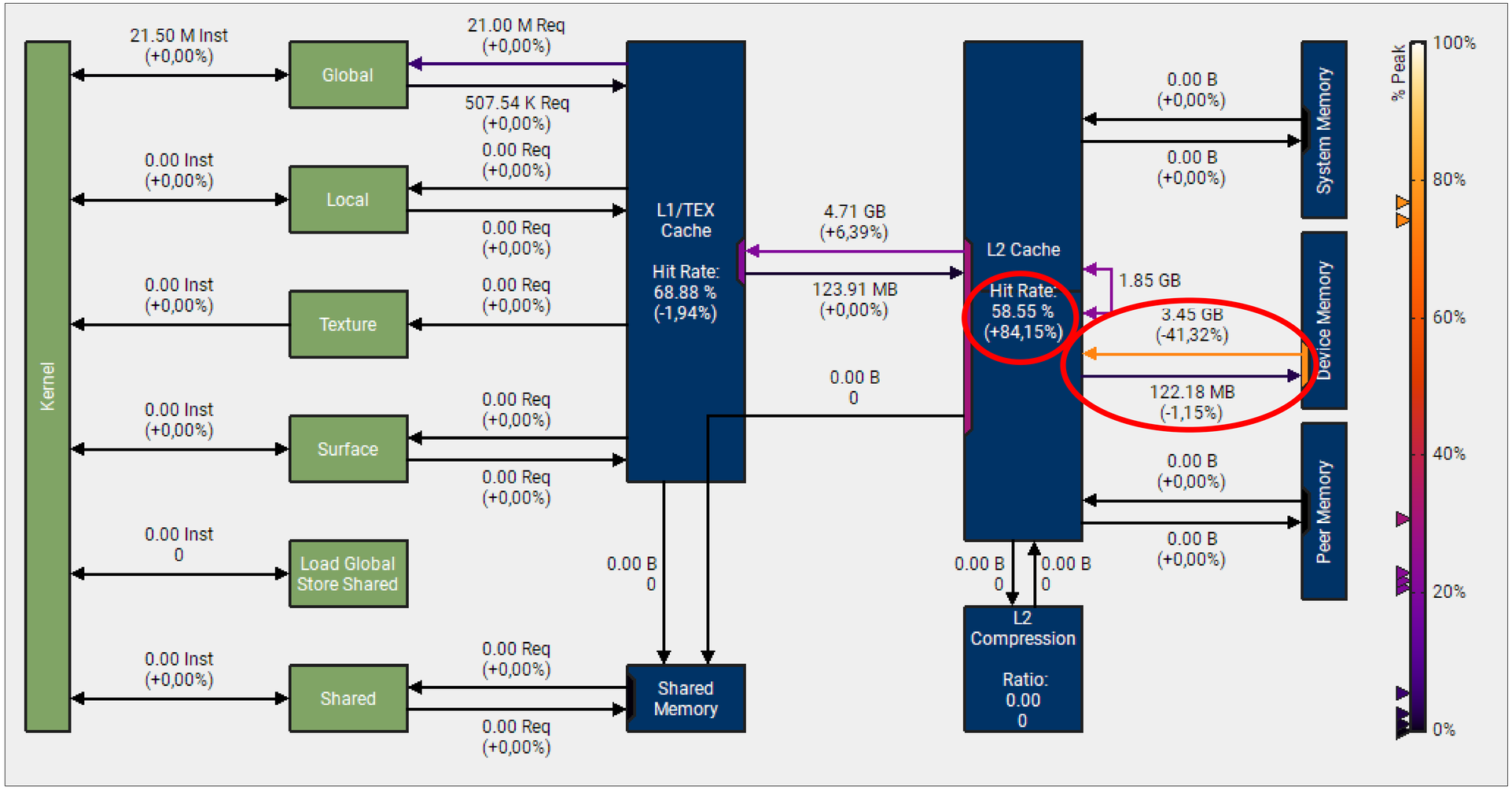
Memory Workload Analysis

- Let Profiler hints guide you: ⚠ **High Memory Throughput**
- Overview
- Memory chart
- Tables for details on all memory subsystem parts
  - Shared/L1/L2/Device Memory

Using the Memory Chart for Comparisons

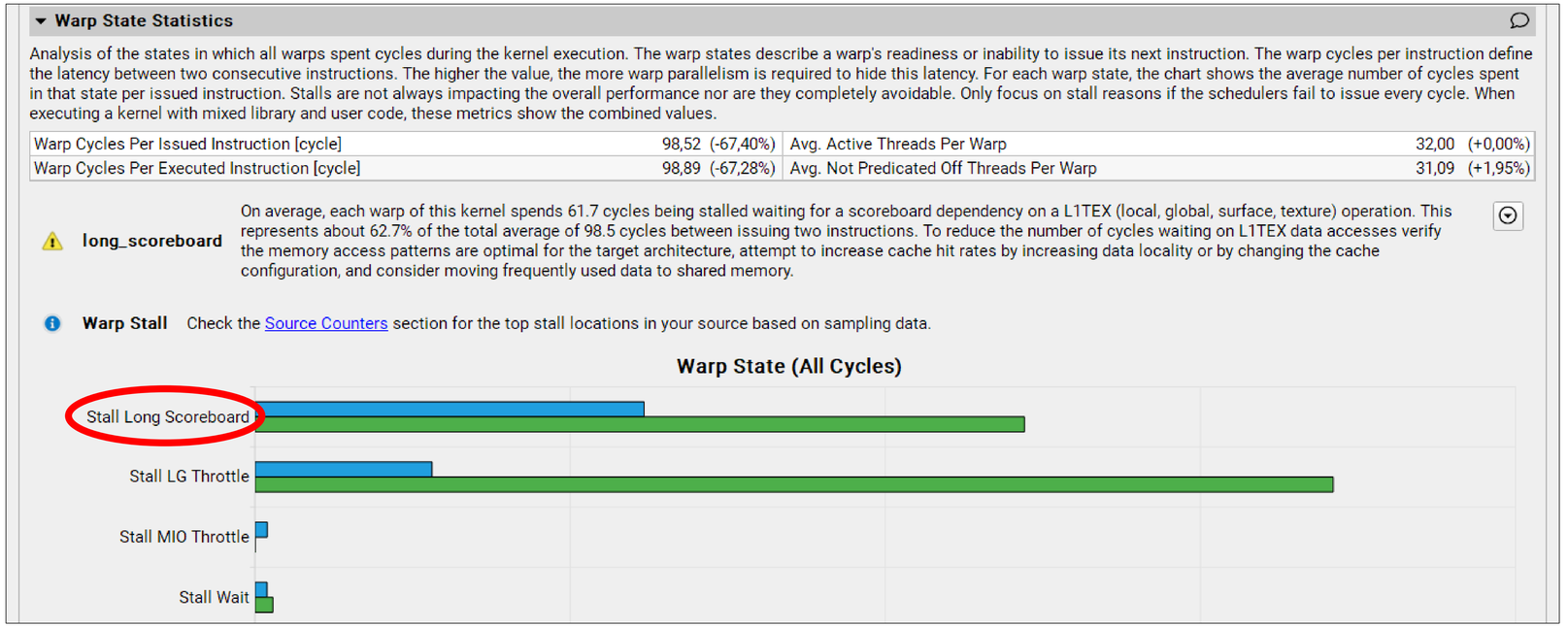
From V100 to A100

- Using baseline
  - Different code versions
  - Hardware versions
- Left to right
  - Kernel/Instructions on SM
  - Caches
  - Device/system memory
- Throughput peak %
  - color-coded
- Effect of much larger L2 cache on A100



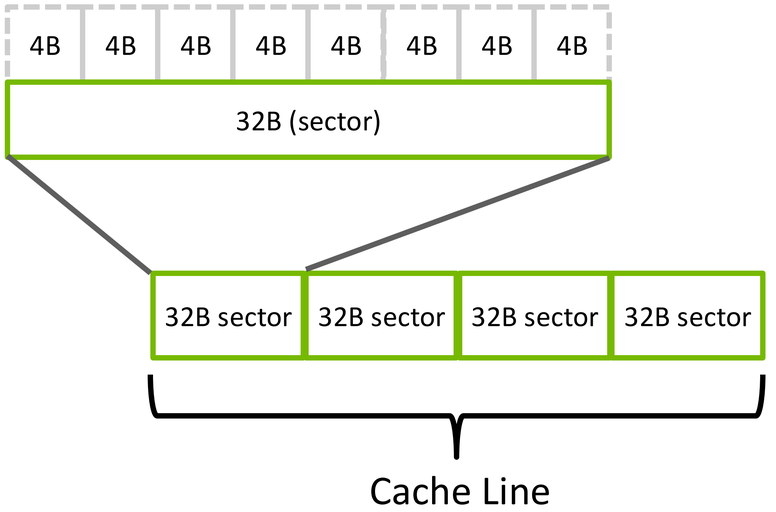
Pinpointing Bottlenecks in Source

- Warp state: Long Scoreboard stalls
  - Common for memory-bound codes
  - Waiting on device mem access to complete
- Less than on V100?
  - Plausible, since we saw reduction in device memory traffic
- Throttle vs. Stall: Queue full



Memory Transactions and Coalescing

- Access to global memory triggers transactions (Device Mem Access)
- Memory access granularity = 32 bytes = 1 sector
- Cache line = 128 bytes = 4 consecutive sectors
  - Example: 4 byte per thread → 4B * 32 threads (1 warp) = 128B
- Data goes from device memory through L2 cache
- Granularity*: Sector for L2, Cache line for L1



*The full picture:
S32089: Understanding and Optimizing Memory-Bound Kernels with Nsight Compute

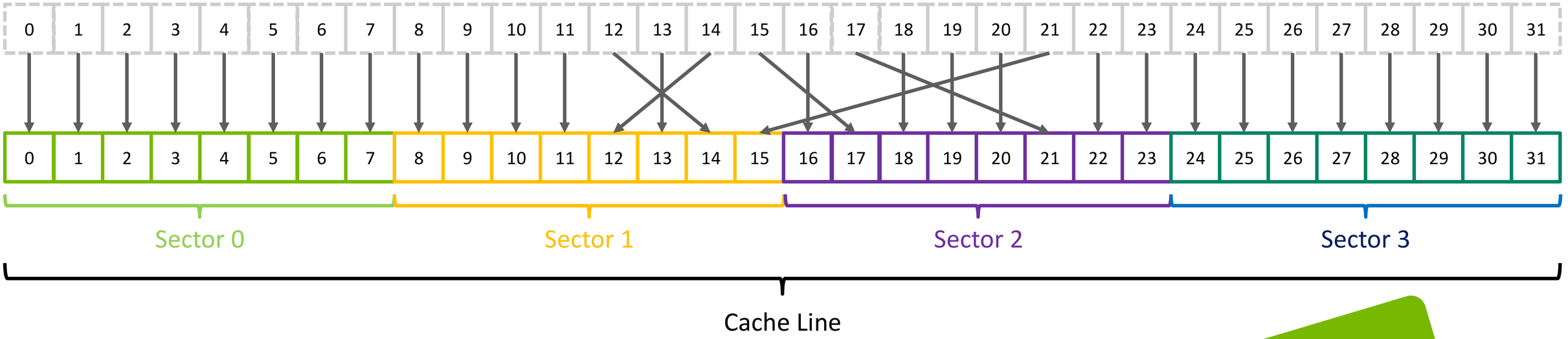
Accessing Global Memory

optimal access pattern (4byte words) – fully coalesced

```
int x_val = x[threadIdx.x];
```

All addresses fall within 4 sectors

Bus utilization: 100%



Permutations are also fine

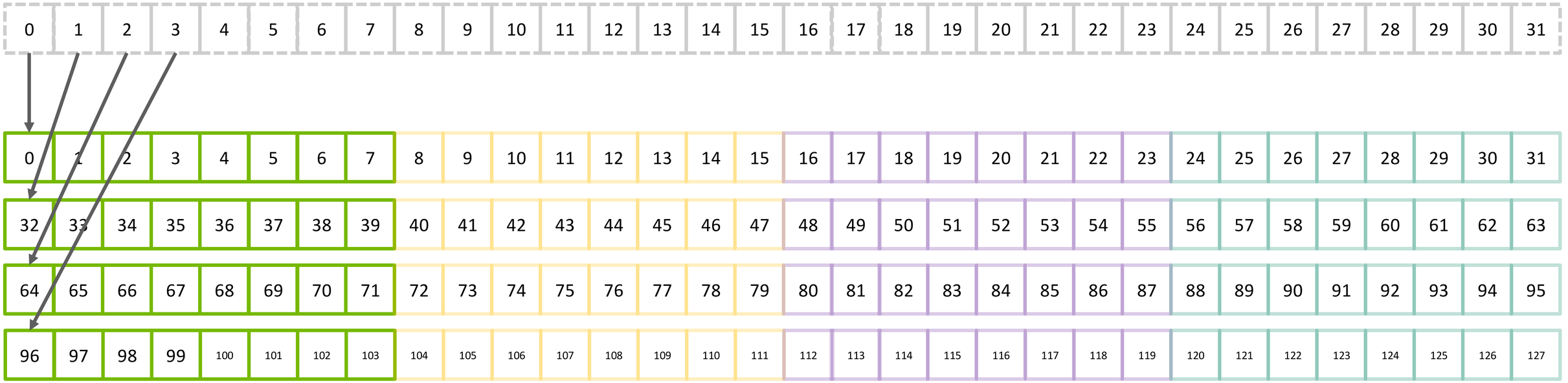
Accessing Global Memory

worst case access pattern (4byte words) – fully uncoalesced

```
// stride 32
int x_val = x[32*threadIdx.x];
// „random“ (pointer chasing, lists, tree, ...)
int x_val = x[lookup[threadIdx.x]];
```

All addresses fall in 32 different sectors

Bus utilization: 12.5%



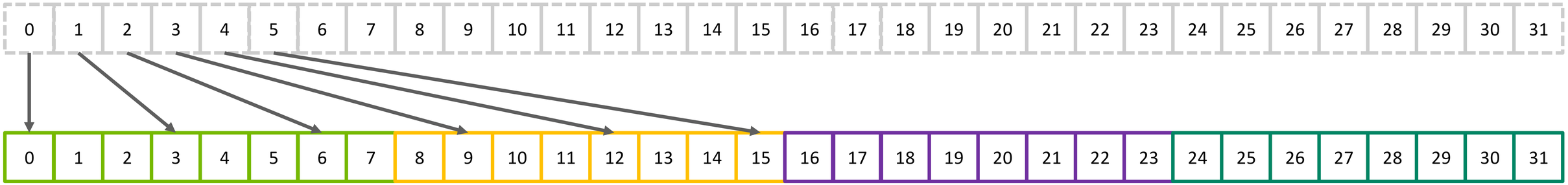
Accessing Global Memory – SoA explained

common access pattern: stride 3

```
int x_val = x[3*threadIdx.x];
```

All addresses fall within 12 sectors (4 byte words)

Bus utilization: 33%



`struct {float x,y,z;} a; ... a[tid].x`

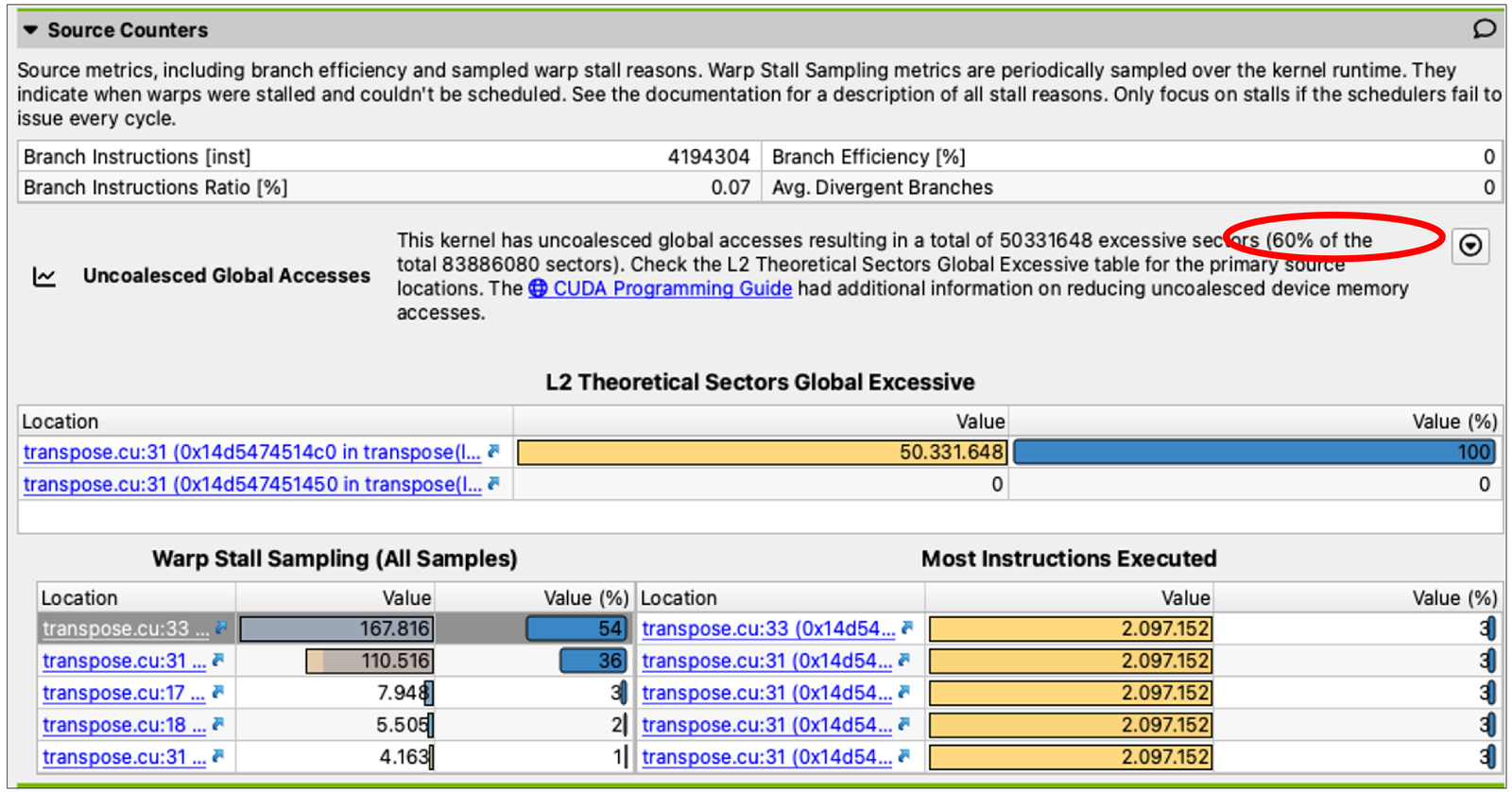
→ use structure-of-arrays (SoA): `a.x[tid]`

`float a[M][N]; ... a[tid][42]`

→ multi-dimensional arrays: pay attention to coalescing (row-major, column-major?)

Pinpointing Bottlenecks in Source

- Source counters: Sampling
  - Warp state, per-instruction
  - Locate stalls via „Not issued“ data
- Warp stalls when data (register) not ready
- In general: Minimize uncoalesced accesses (here: 60% of all)
  - Large caches lessen impact
- **Links will take you to Source/SASS view**



Source Counters

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

Branch Instructions [inst]	4194304	Branch Efficiency [%]	0
Branch Instructions Ratio [%]	0.07	Avg. Divergent Branches	0

Uncoalesced Global Accesses

This kernel has uncoalesced global accesses resulting in a total of 50331648 excessive sectors (60% of the total 83886080 sectors). Check the L2 Theoretical Sectors Global Excessive table for the primary source locations. The CUDA Programming Guide had additional information on reducing uncoalesced device memory accesses.

L2 Theoretical Sectors Global Excessive

Location	Value	Value (%)
transpose.cu:31 (0x14d5474514c0 in transpose(I...	50.331.648	100
transpose.cu:31 (0x14d547451450 in transpose(I...	0	0

Warp Stall Sampling (All Samples)

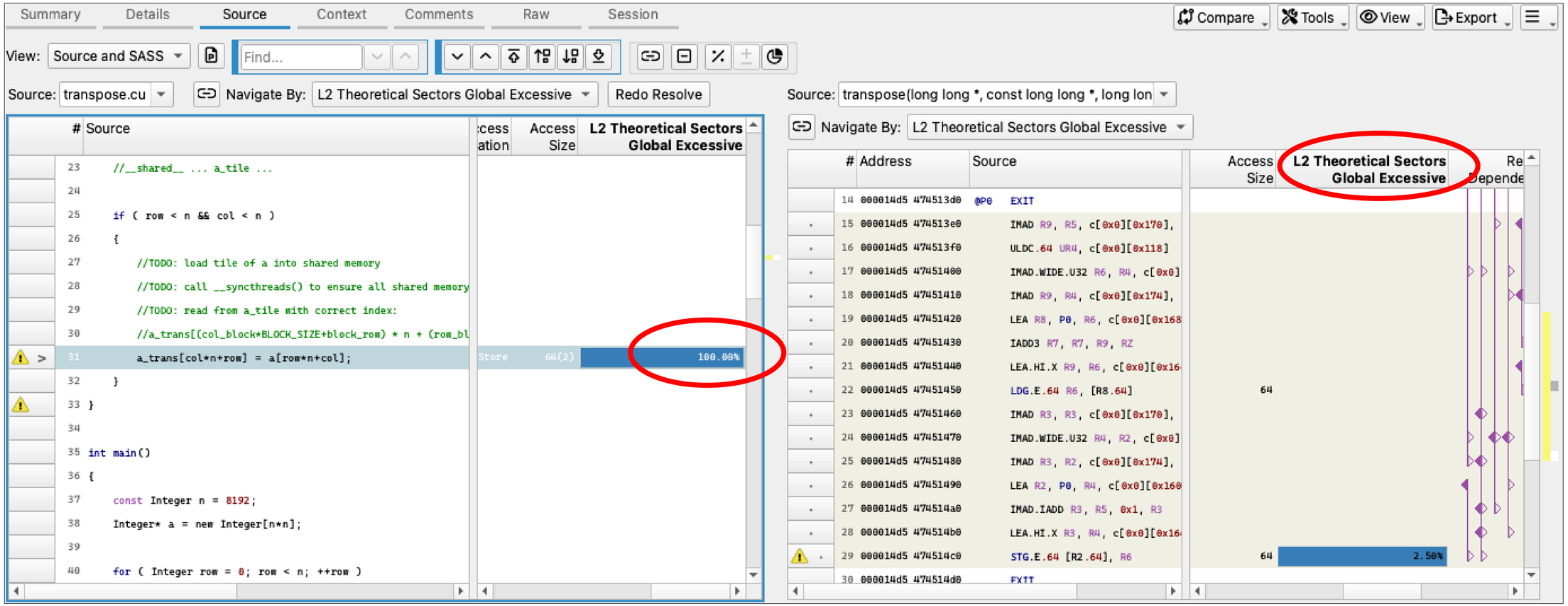
Location	Value	Value (%)
transpose.cu:33 ...	167.816	54
transpose.cu:31 ...	110.516	36
transpose.cu:17 ...	7.948	3
transpose.cu:18 ...	5.505	2
transpose.cu:31 ...	4.163	1

Most Instructions Executed

Location	Value	Value (%)
transpose.cu:33 (0x14d54...	2.097.152	3
transpose.cu:31 (0x14d54...	2.097.152	3
transpose.cu:31 (0x14d54...	2.097.152	3
transpose.cu:31 (0x14d54...	2.097.152	3
transpose.cu:31 (0x14d54...	2.097.152	3

Source and SASS

- Uses lineinfo/debug data for correlation (embedded source very useful)
- Instruction-level counters on SASS
  - LDG? Check tooltips
- Bidirectional, associated lines highlighted on selection



Occupancy

Calculator now integrated into Nsight Compute

- Roughly: Amount of GPU hardware 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”
  - caching effects, more/less registers, …
- Still: simple mechanism to tweak and experiment with
  - Use builtin occupancy calculator for “what if” analysis
  - How to fit one block more/less on the SM?
- In CUDA C++, try `__launch_bounds__` to influence compiler heuristics
  - https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#launch-bounds