CUDA Tools for Profiling and Debugging

Markus Hrywniak, Senior DevTech Compute, April 2024

Session outline

Goals

- Use `compute-sanitizer` to automatically detect correctness issues (invalid memory accesses)
- Use `cuda-gdb` to manually and interactively debug a CUDA program
- Use `Nsight Systems` to learn the basic workflow to optimize performance of GPU programs

Debugging Correctness, then **Debugging Performance**

Debugging Correctness: Best Practices

Before you start

- Crashes are "nice" – the stacktrace often points to the bug
- Prerequisite: Compile flags
  - While developing, always use `-g -lineinfo`
  - Use `-g -G` for manual debugging
  - Specific flags for compilers/languages (e.g. gfortran): `-fcheck=bounds`
- Memory corruption: Out-of-bounds accesses may or may not crash
  - *compute-sanitizer*: Automate finding these errrors
- Other issues: Manual debugging
  - *cuda-gdb*: Command-line debugger, GPU extensions
  - `CUDA_LAUNCH_BLOCKING=1` forces synchronous kernel launches

NVCC compile flags for debugging	
`-g`	Embed symbol info for ***host*** code
`-lineinfo`	Generate line correlation info for ***device*** code
`-G`	Device debug – slow

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
- Main usage: Auto-detect invalid GPU code and shortcut debugging effort
  - Directly pinpoint source code line/addresses, access size
- Leak-checking for device allocations - forgot to call `cudaFree()`?
  - `--leak-check full`
- Filtering and other capabilities. Two commonly useful switches:
  - `--log-file output.log`
    - Separates (potentially verbose) output into separate file
  - `--kernel-regex kns=some_substring`
    - Only checks kernels containing "some_substring"

compute-sanitizer

Example launch

- Run it: `srun --pty compute-sanitizer ./set_vector`
- Abbreviated output:

```
========= COMPUTE-SANITIZER

========= Invalid __global__ write of size 4 bytes
========= at 0xc0 in
/p/home/jusers/hrywniak1/juwels/GPU-Course/task1/set_vector.cu:20:set(int,float*,float)
========= by thread (0,0,0) in block (0,0,0)
========= Address 0x2afe49a02000 is out of bounds
========= Saved host backtrace up to driver entry point at kernel launch time
[....]

========= Target application returned an error
========= ERROR SUMMARY: 1025 errors
```

- Actual output can be very long, if many GPU threads produce (similar) errors.

Task 1

Use `compute-sanitizer` to automatically identify an error

- Location of code: `2-Tools/exercises/tasks/task1`
- Steps (see also Instructions.ipynb)
  - Fix set-vector.cu!
  - Use compute-sanitizer to locate error in set-vector.cu, and fix it
  - compute-sanitizer should run without errors!
  - Build: `make`
  - Run: `make run / make memcheck`

cuda-gdb

Extends GDB for CUDA applications
https://docs.nvidia.com/cuda/cuda-gdb/index.html

- "Symbolic Debugger" – leaverage debug symbols to correlate execution issues with original source code
- Interactive/manual tool, with useful shortcuts
  - https://docs.nvidia.com/cuda/cuda-gdb/index.html#automatic-error-checking
- Textual, like a shell for debugging – Not the easiest to master, but very powerful, and works everywhere
- Basic workflow for segfaults
  - Crashing app invoked via
    - `./my_app_name my_app_arg another_arg`
  - becomes
    - **`cuda-gdb --args`** `./my_app_name my_app_arg another_arg`
  - Shows you the debugger shell prompt: (cuda-gdb)
    - Launch program with "`run`"
  - Identify the segfault – Done ☺
- Advanced workflow to step through execution, understand program flow, inspect and modify variables,…

cuda-gdb Cheat Sheet

(doubles as a GDB cheat sheet)

- Most commands have abbreviations
  - continue → cont, break → b, info → i, backtrace → bt, ...
  - cuda thread 4 → cu th 4
- Use TAB completion to help you remember command names
- Use help and apropos to avoid a round-trip to the browser (try: apropos cuda.*api)

run	Begin progam execution under debugger
backtrace	Print call stack (e.g. after an exception)
list	List source code around current location
print <var>	Print contents of <var>, e.g. *"print i"* to print the loop counter *i*
set var <var>=<value>	Set value of <var> to <value>, e.g. *"set var i=42"*
break 10 break foo.cpp:10 break my_func	Set breakpoint (suspend execution) on: line 10 in current file ... line 10 in file foo.cpp ... function my_func in any file
set cuda api_failures **stop**	Break on any CUDA API failures (e.g. launch errors)
continue / next / step	Resume execution (after hitting breakpoint) until next: break / line / instruction
info locals	Print all local variables in current scope
info cuda threads	Print current thread configuration
cuda thread 15	Switch focus to thread (here: 15)

cuda-gdb Examples

Launch

- Launching the application inside the debugger – like a shell

```
$ cuda-gdb --args ./gpu-print # The same works on pure CPU using plain gdb.
[...]
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./gpu_print...
(cuda-gdb)
```

- Type run to actually launch the program itself

```
(cuda-gdb) run
Starting program: ./gpu_print
[Detaching after fork from child process 7437]
[New Thread 0x15554ca60000 (LWP 7449)]
[New Thread 0x15554c85f000 (LWP 7450)]
blockIdx.x = 1, threadIdx.x = 0, i = 0
[...]
(cuda-gdb) # program finished running, debugger waiting for new instructions
```

The Most Essential Command

In case of segfault, remember the backtrace

- If your app crashes or terminates unexpectedly, the debugger can very often tell you the exact location of the issue
  - Both in CPU and GPU code

```
$ cuda-gdb --args ./gpu-print

(cuda-gdb) run

[...]

CUDA Exception: Warp Illegal Address
The exception was triggered at PC 0xacbc90 (gpu_print.cu:19)

Thread 1 "gpu_print" received signal CUDA_EXCEPTION_14, Warp Illegal Address.
[Switching focus to CUDA kernel 0, grid 1, block (0,0,0), thread (0,0,0), device 0,sm 0,warp 0,lane 0]
0x0000000000acbca0 in print_test<<<(2,1,1),(32,1,1)>>> () at gpu_print.cu:19
19                double x = *(double*)nullptr;
(cuda-gdb) bt # "backtrace"
#0  0x0000000000acbca0 in print_test<<<(2,1,1),(32,1,1)>>> () at gpu_print.cu:19
```

- Backtrace tries to print all stack frames (i.e. function calls) with line information up to the current location
  - Equally useful when manually debugging or using breakpoints
  - Some errors can corrupt the stack, making the backtrace less useful

Breakpoints

Interrupting execution to inspect program state

- Retry, but before launch, set a breakpoint that will pause execution
- Reminder: You need –G for meaningful kernel debugging

```
(cuda-gdb) l print_test # show source of function
[...]
(cuda-gdb) break 18
Breakpoint 1 at 0x403fe6: file .../exercises/tasks/task2/gpu_print.cu, line 20.
(cuda-gdb) run
Starting program: ./gpu_print
[...]
[Switching focus to CUDA kernel 0, grid 1, block (0,0,0), thread (0,0,0), device 0,sm 0,warp 0,lane 0]
Thread 1 "gpu_print" hit Breakpoint 1, print_test<<<(2,1,1),(32,1,1)>>> () at gpu_print.cu:18
18              int i = 0;
(cuda-gdb) print i
$1 = <optimized out>
(cuda-gdb) next
19              printf("blockIdx.x = %d, threadIdx.x = %d, i = %d\n", blockIdx.x, threadIdx.x, i);
(cuda-gdb) print i
$2 = 0
(cuda-gdb) continue # resume execution
```

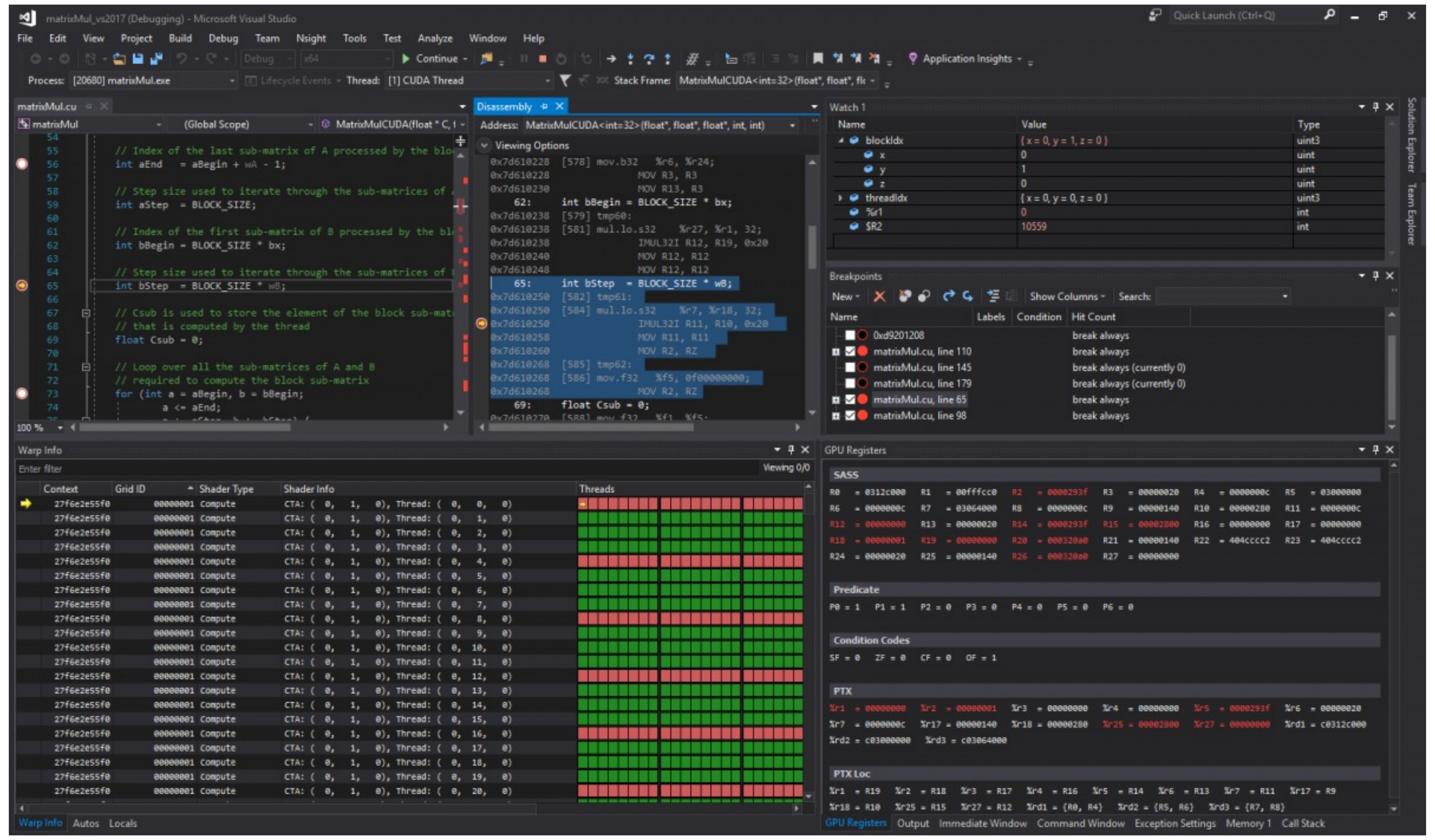
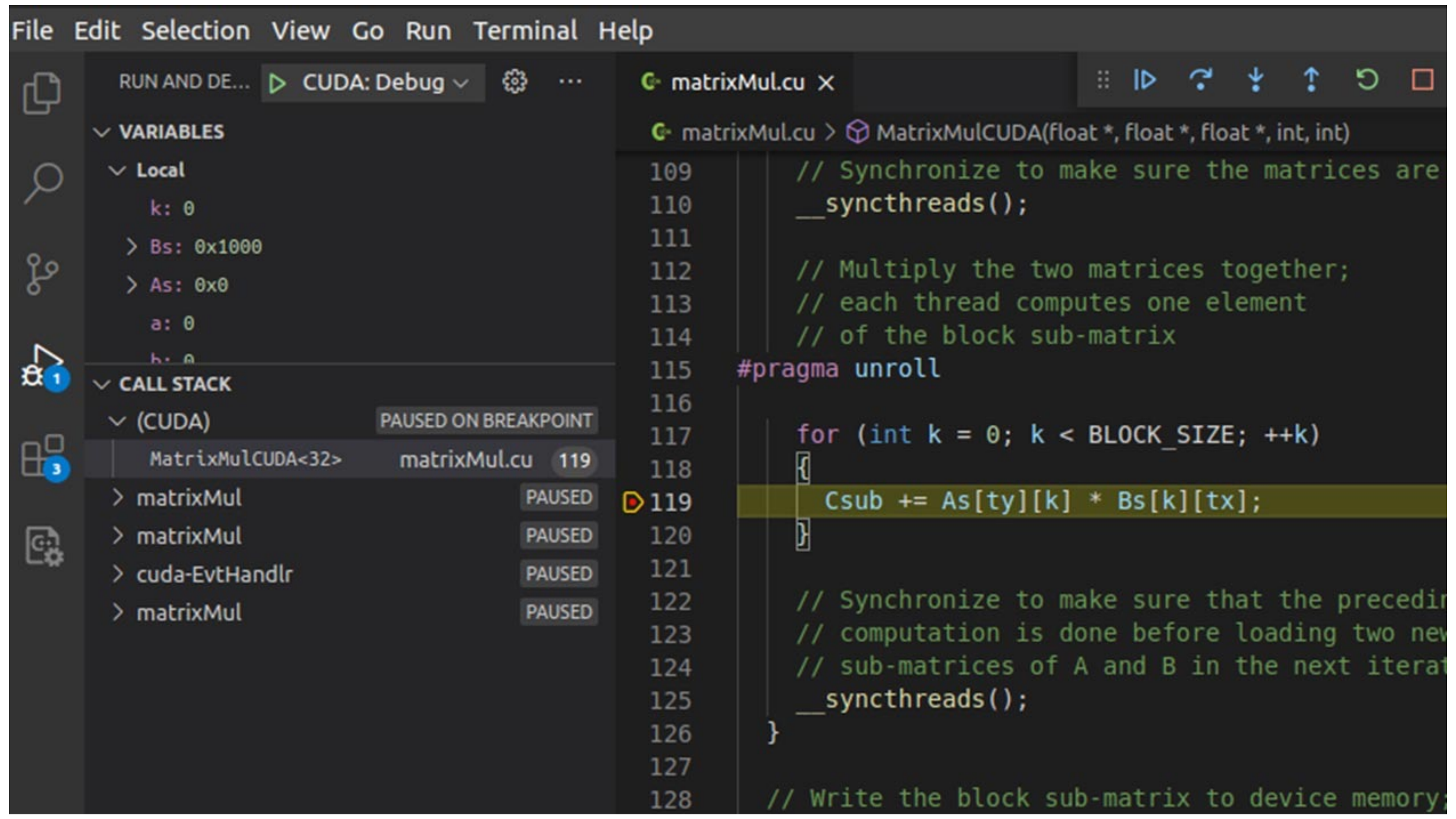
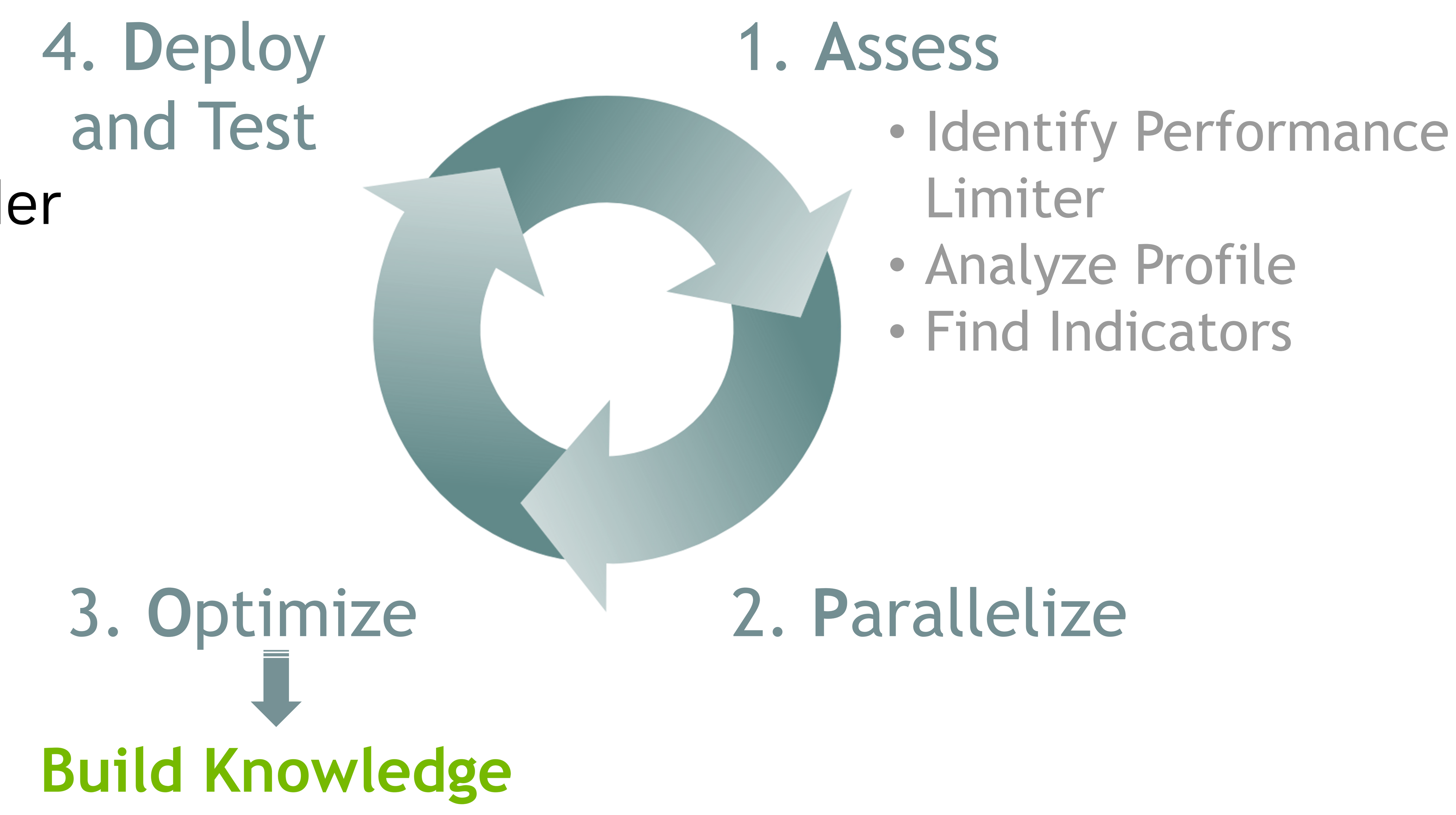
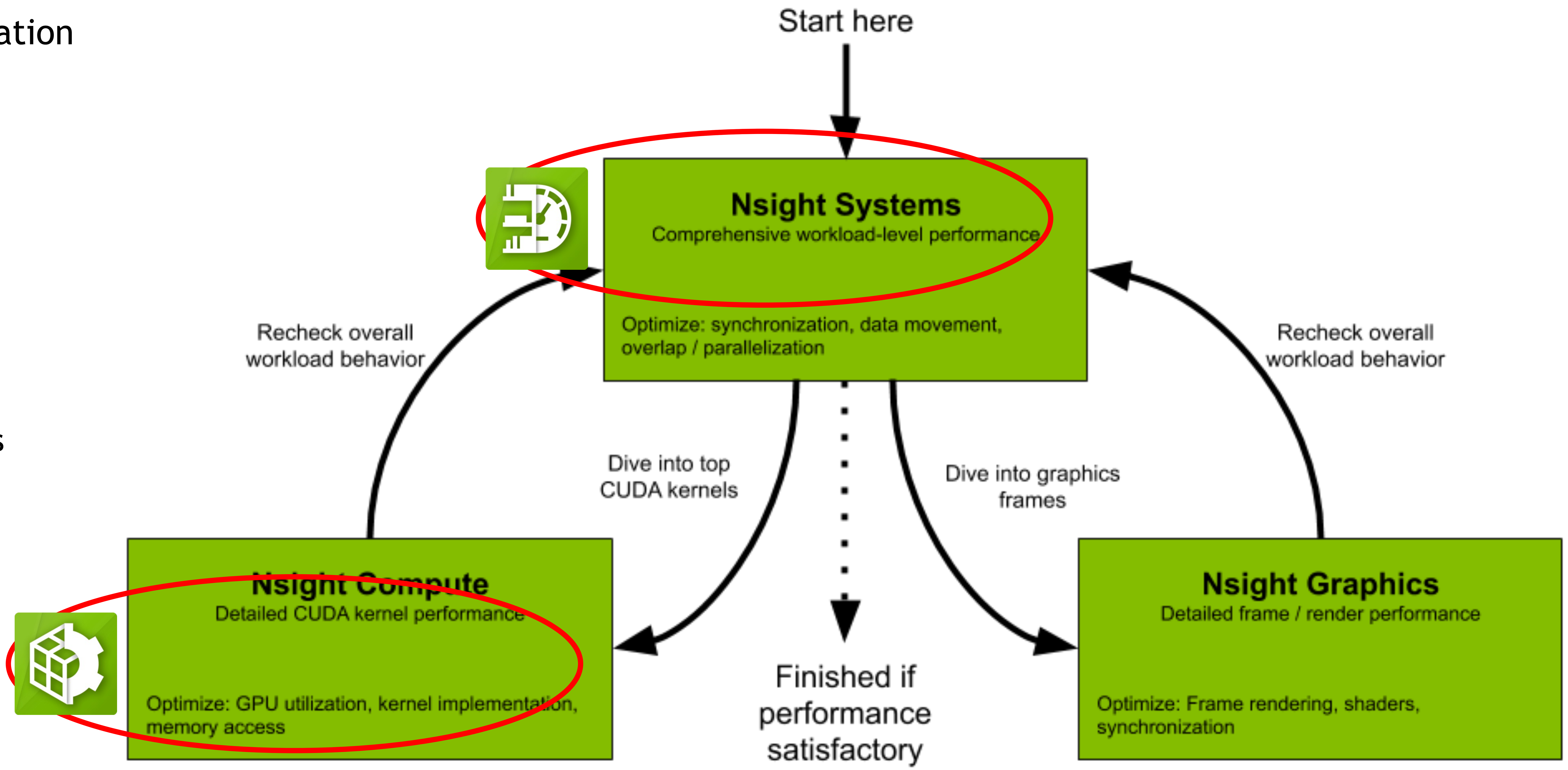
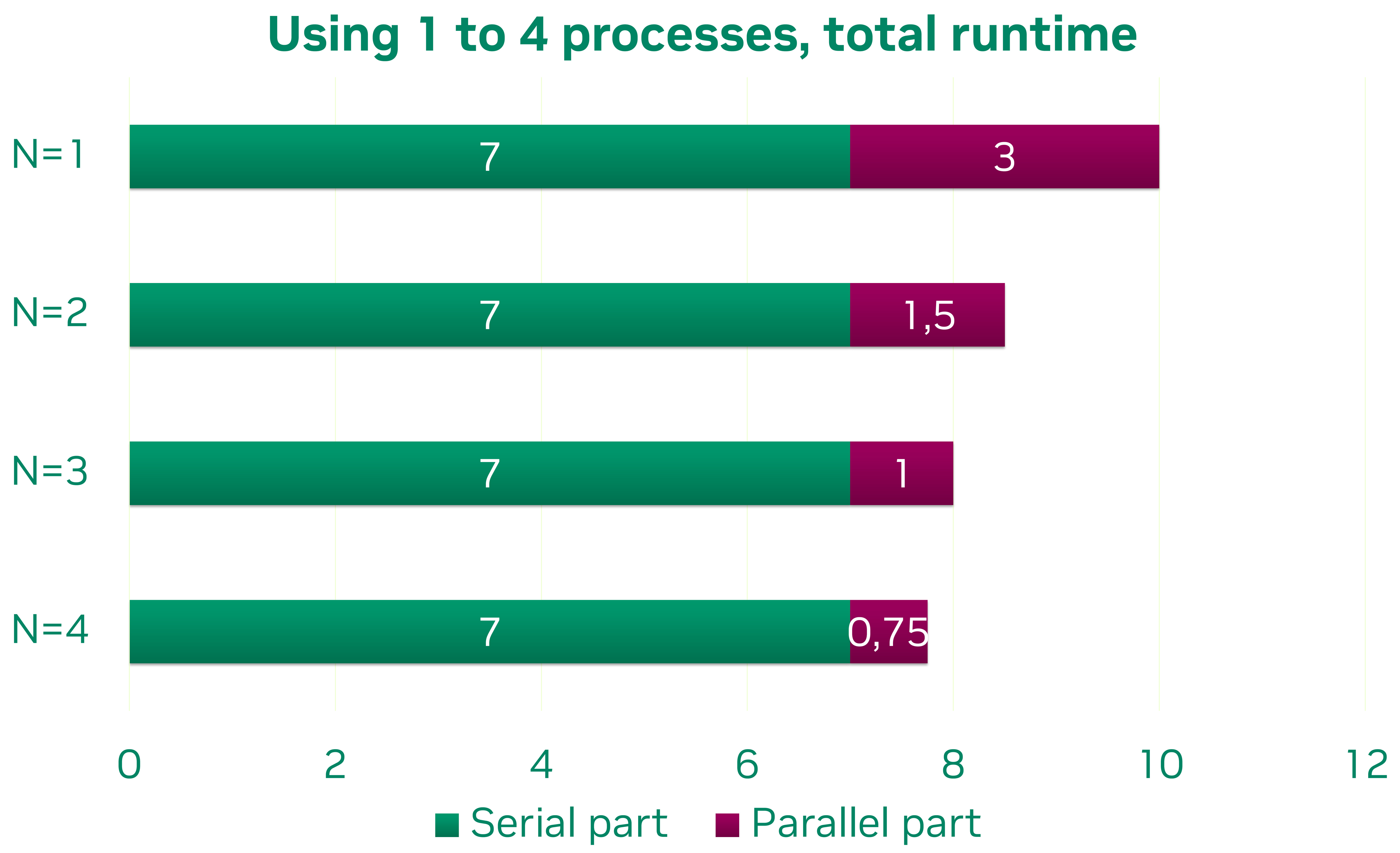
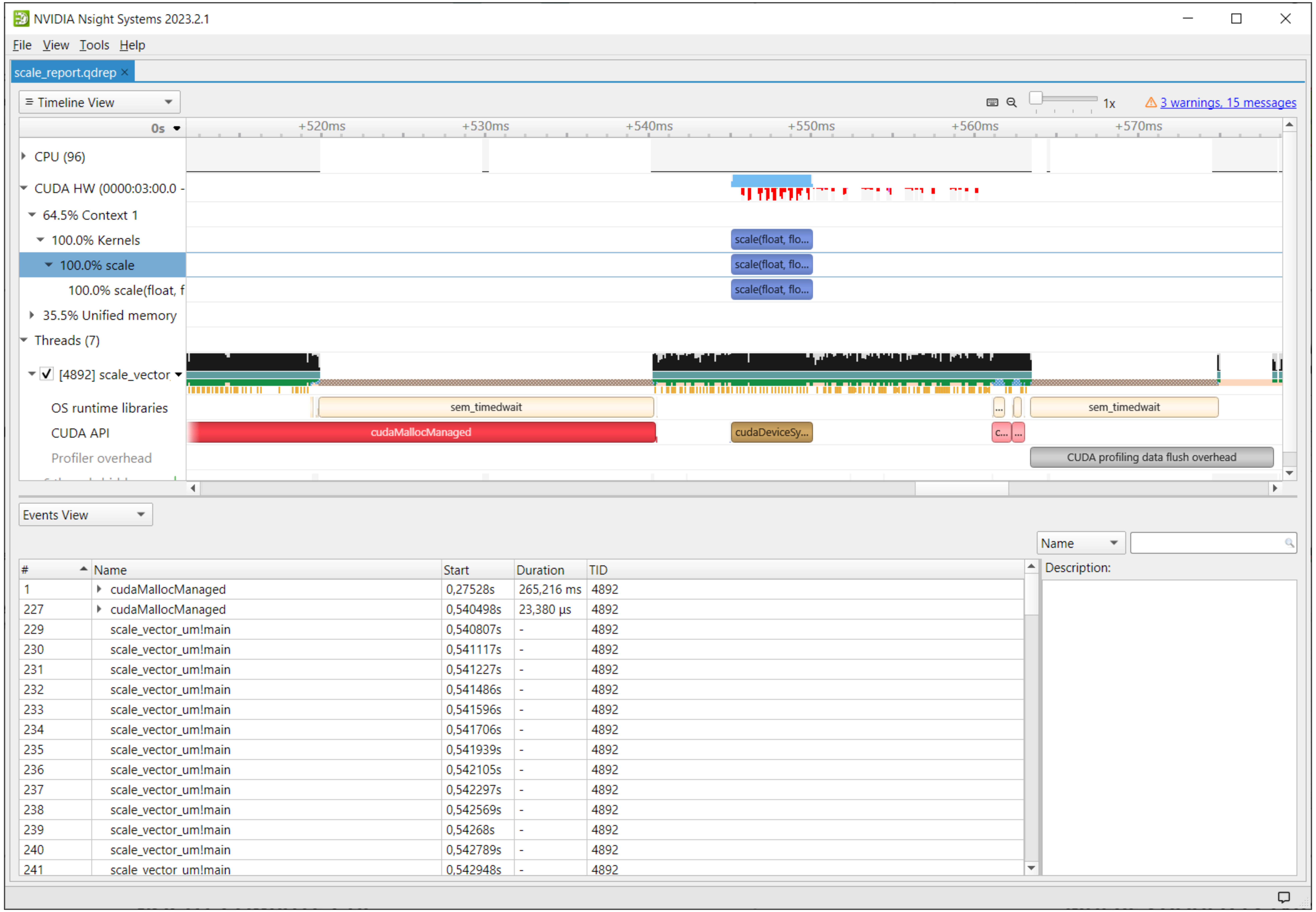
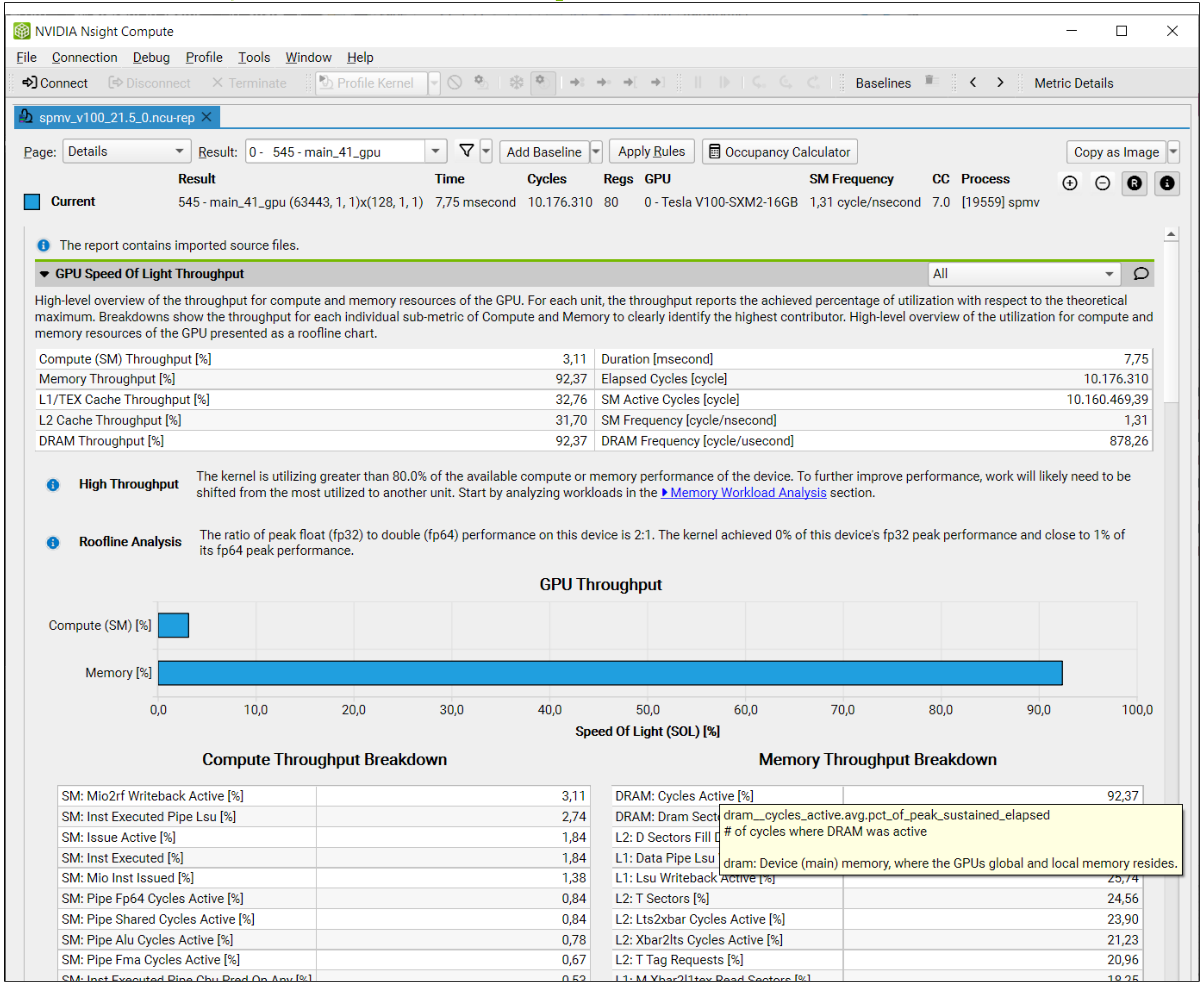
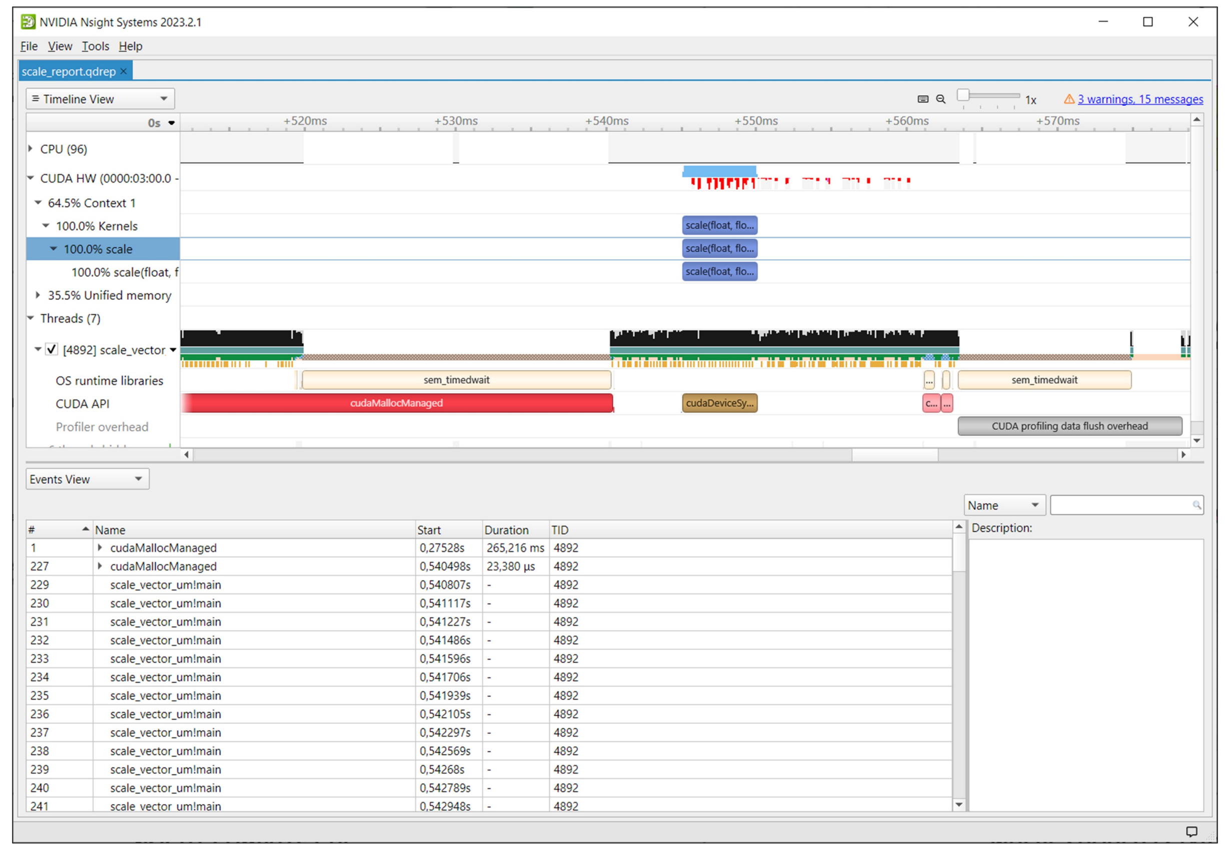
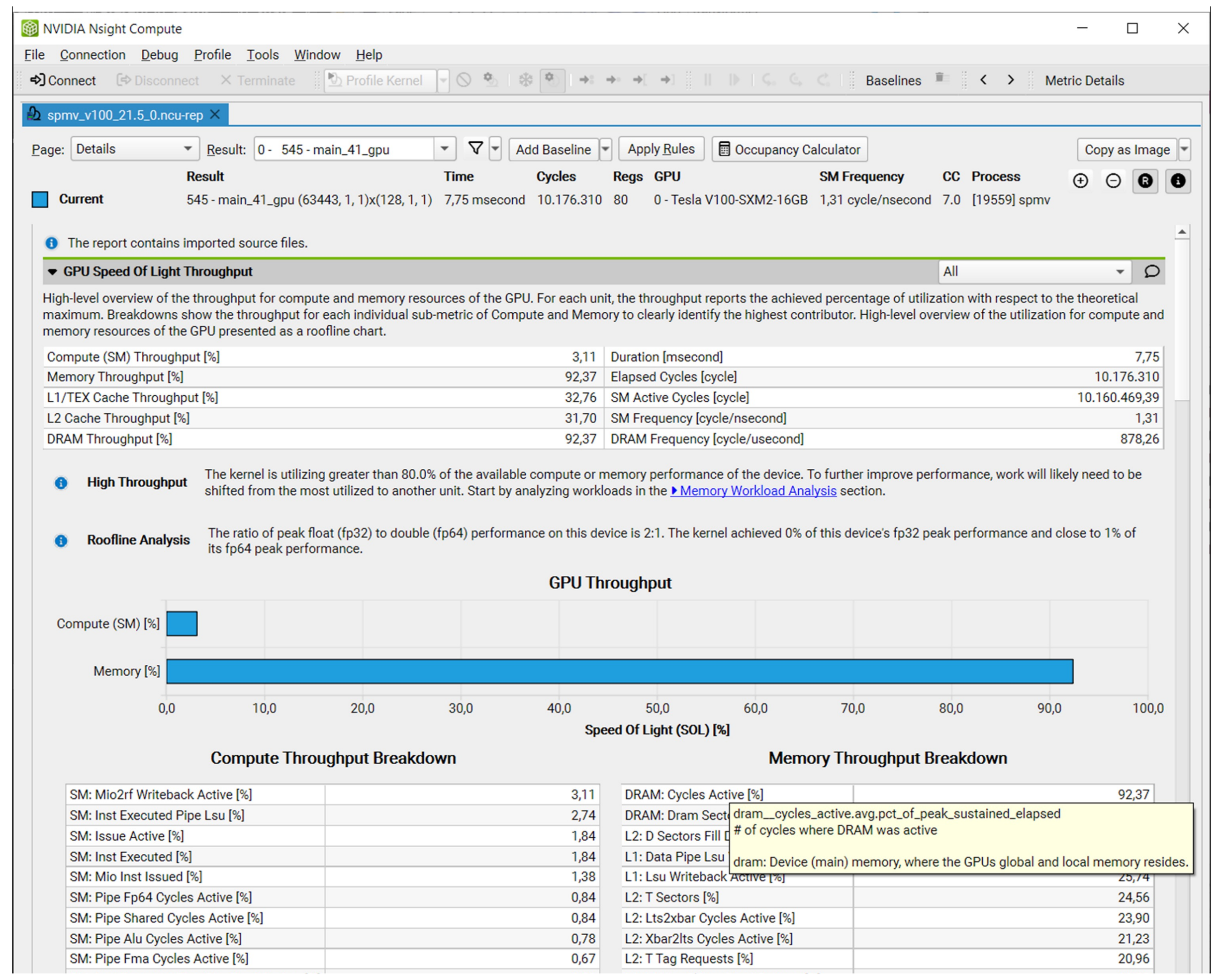
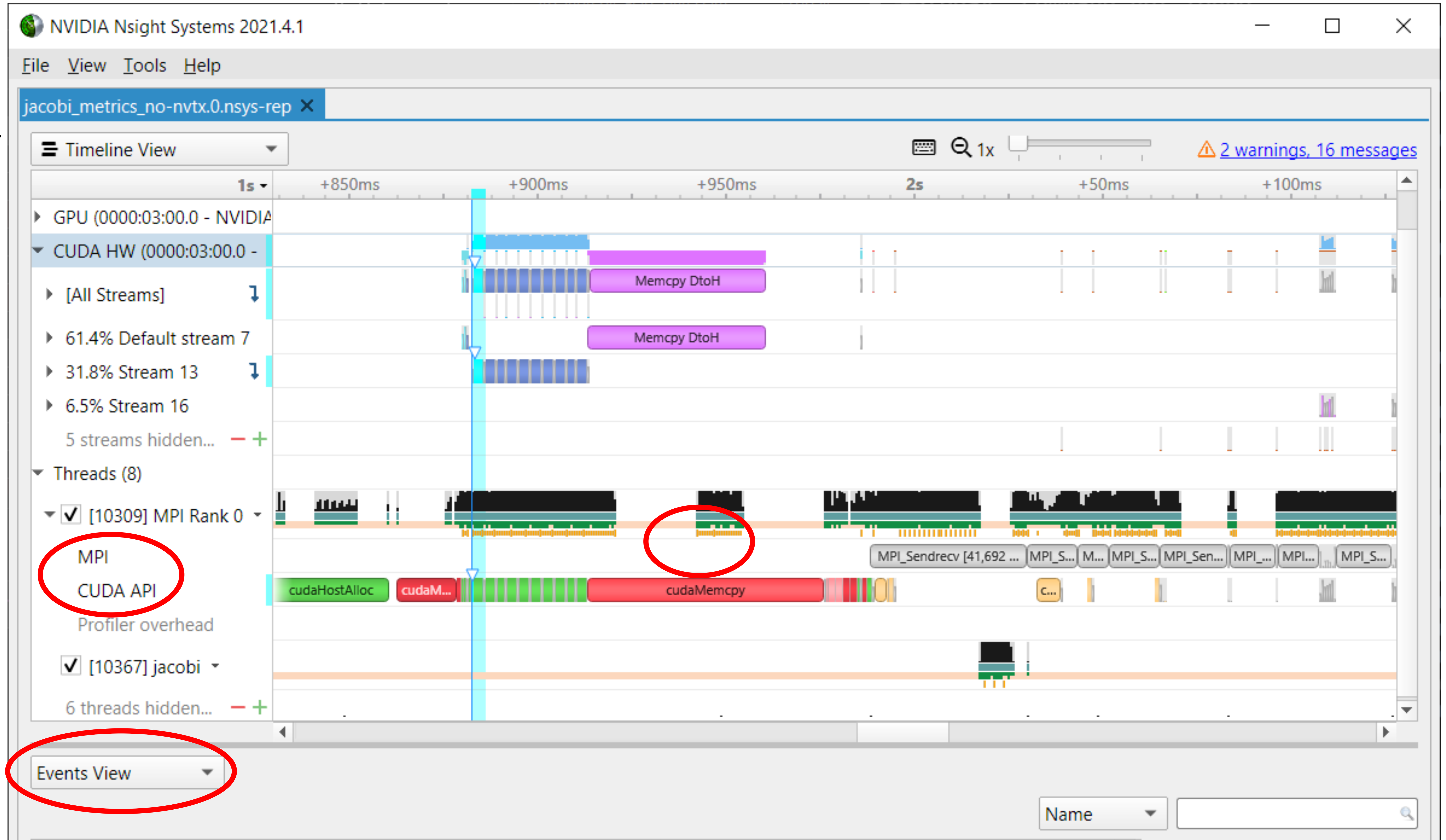
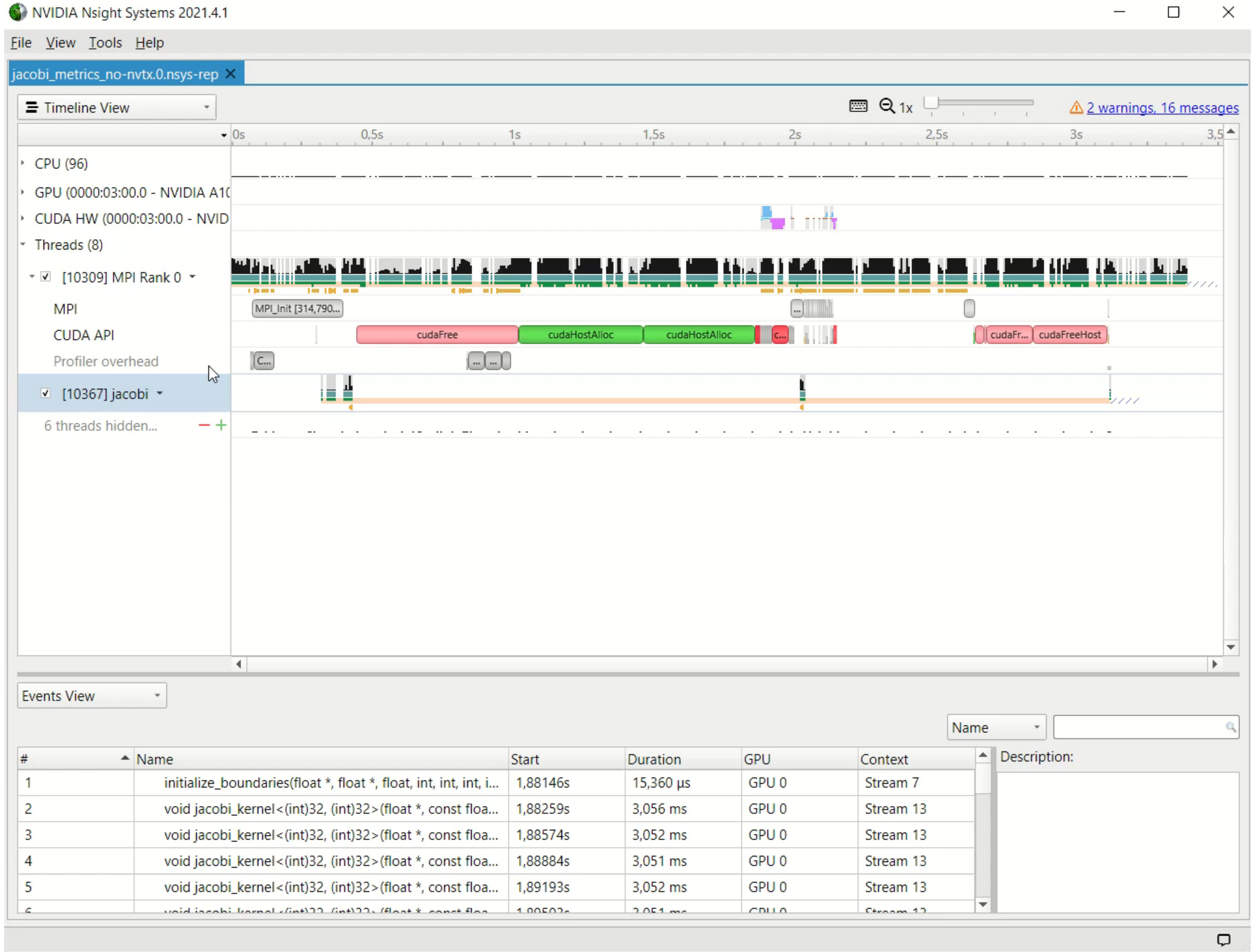
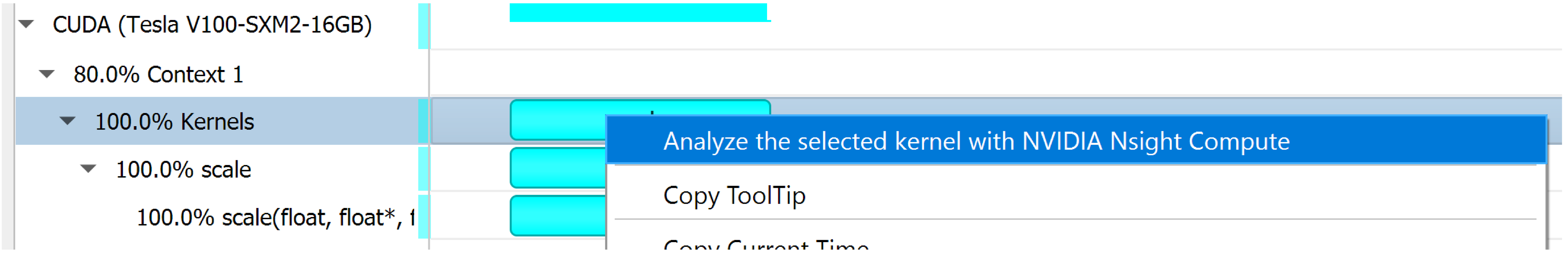
- Why "optimized out"?

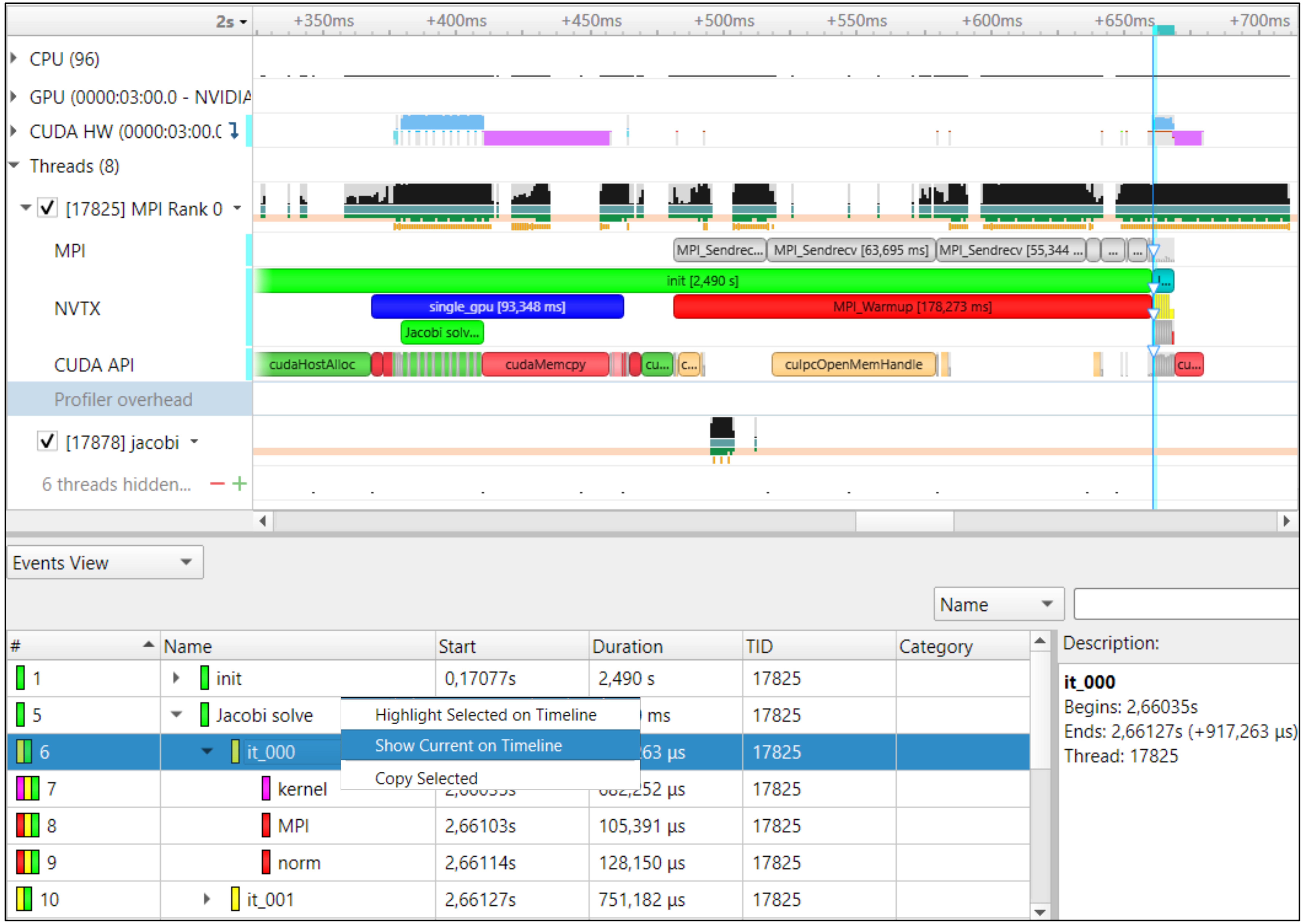
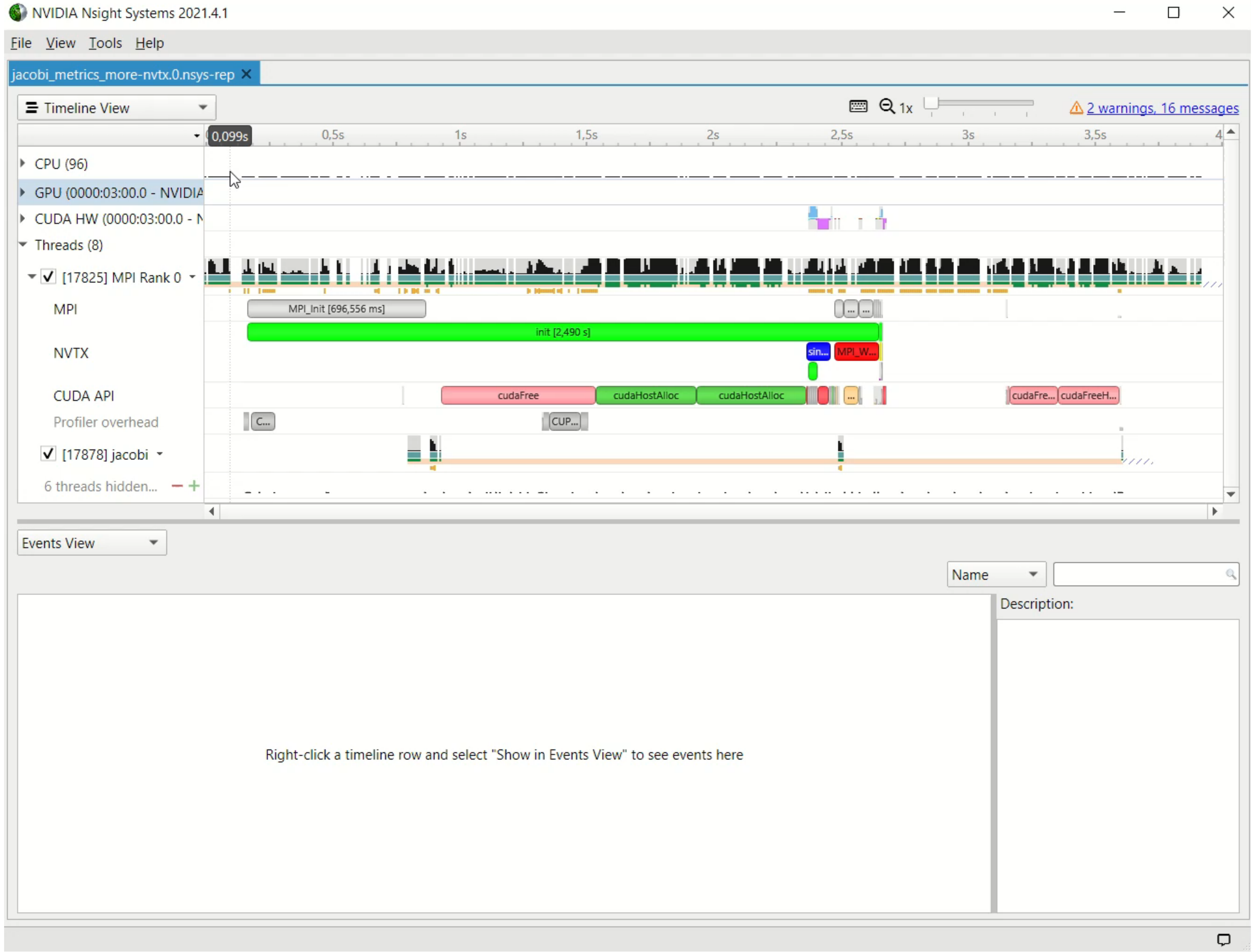
Breakpoints and Program State

Changing the course of execution

- Breakpoints can be deleted again

```
(cuda-gdb) i brea # "info breakpoints"
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x0000000000acbf60 in print_test() at gpu_print.cu:18
        breakpoint already hit 1 time
(cuda-gdb) d 1 # "delete 1"
(cuda-gdb) i brea
No breakpoints or watchpoints.
```

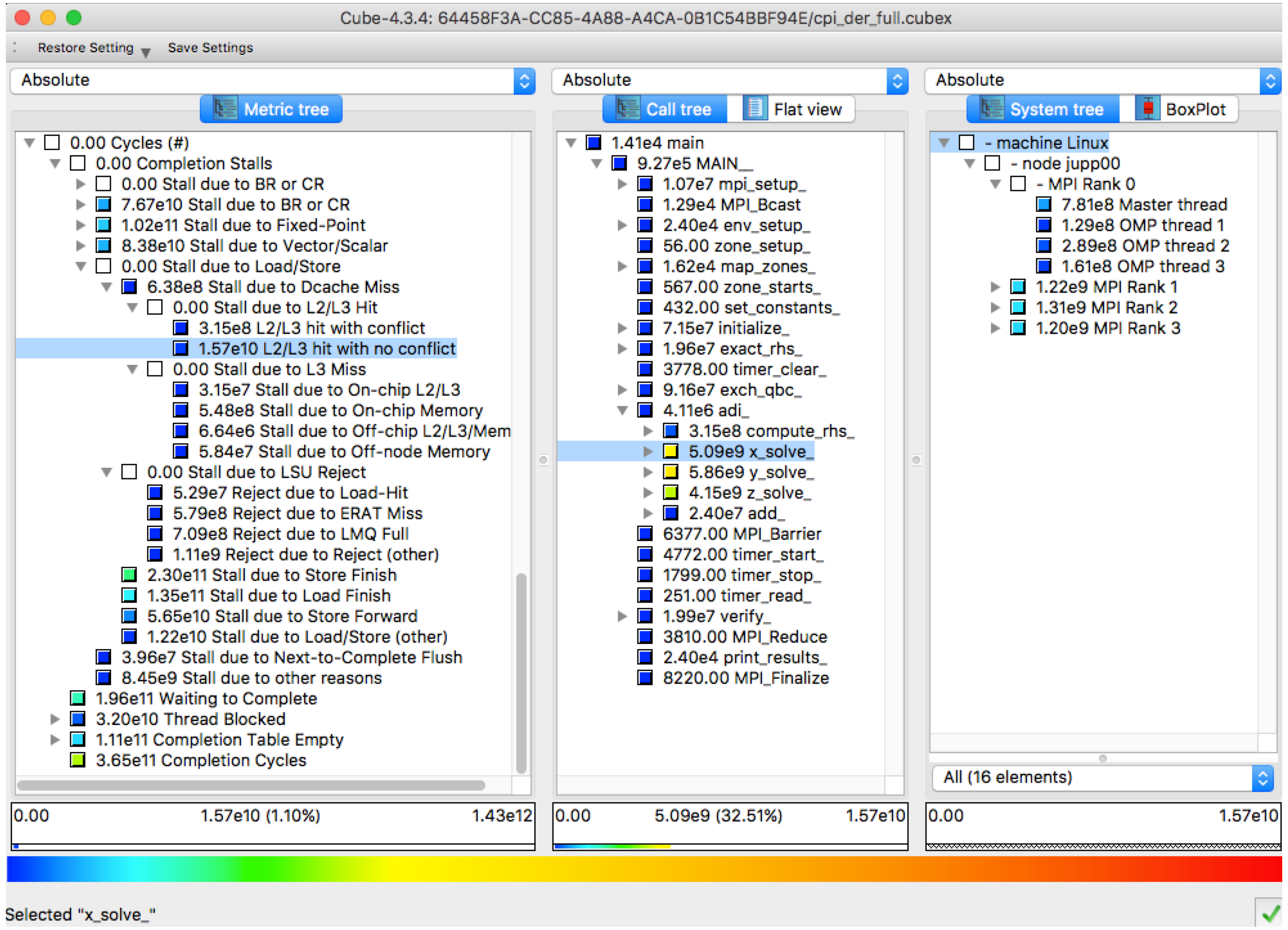
- Breakpoints can be conditional, also: watchpoints (see help)
- Actively change state by setting variables
  - `(cuda-gdb) set var my_variable = 11`
- Actively change control flow by calling functions
  - `(cuda-gdb) call my_print_func("debugging message")`
- Inspect memory and variables. Assume we have `const char* s = "my_str"`
  - ```
    (cuda-gdb) print s # prints "my_str"
    (cuda-gdb) print s[0]@3 # prints "my_"
    (cuda-gdb) x/5c s  # prints next 5 values following address s interpreted as chars (check help)
    ```
  - `0x4c54f0:      109 'm' 121 'y' 95 '_'  115 's' 116 't'`
    ```

GPU-Specifics

New commands in cuda-gdb

- GPU-specifics: Setting the *focus*

```
(cuda-gdb) i cuda threads
  BlockIdx ThreadIdx To BlockIdx ThreadIdx Count         Virtual PC     Filename  Line
Kernel 0
*  (0,0,0)   (0,0,0)     (0,0,0)  (31,0,0)    32 0x0000000000acbf90 gpu_print.cu    19
   (1,0,0)   (0,0,0)     (1,0,0)  (31,0,0)    32 0x0000000000acbf60 gpu_print.cu    18
(cuda-gdb) cuda thread
thread (0,0,0)
(cuda-gdb) cuda thread 10
[Switching focus to CUDA kernel 0, grid 1, block (0,0,0), thread (10,0,0), device 0,sm 0,warp 0,lane
10]
19                printf("blockIdx.x = %d, threadIdx.x = %d, i = %d\n", blockIdx.x, threadIdx.x, i);
```

- Focus can be set to specific blocks, SMs, devices, … – **help cuda**
  - Hardware and software abstractions (e.g. blocks vs. SMs)
- Options: Try `(cuda-gdb) set cuda`<ENTER> for a list
  - Two commonly-used options: **api_failures** and **launch_blocking**

IDE Integration

Beyond shells and text-based user interfaces

- Why use an integrated development environment (IDE)?
  - Source code editor with CUDA C/C++ highlighting
  - Project / file management with integration of version control
  - Build system
  - Graphical interface for debugging heterogeneous applications
- On Windows: Nsight Visual Studio Edition
  - https://developer.nvidia.com/nsight-visual-studio-edition/
- Nsight **Visual Studio Code** Edition
  - https://developer.nvidia.com/nsight-visual-studio-code-edition/
- Eclipse : https://docs.nvidia.com/cuda/nsight-eclipse-plugins-guide
- Recommended: https://github.com/NVIDIA/nsight-training





Task 2

Change program execution on-the-fly with cuda-gdb

- Location of code: 2-Tools/exercises/tasks/task2
- Steps (see also Instructions.ipynb)
  - Let thread 4 from the first block (block 0) print 42 instead of 0.
    Do not change the source code!
    Do use cuda-gdb commands and breakpoints.
  - Build and run once to see the standard output:
    `make run`
  - Run and debug interactively on a compute node:
    1. `eval $JSC_SUBMIT_CMD bash -i`
    2. `cuda-gdb --args ...`

This gets you an interactive shell on the compute node

- Hints:
  - Use the cheat sheet: breakpoints, listing source, *set*ting variable values, changing the active *cuda thread*…
  - If you get stuck, see the solutions directory for the commands to feed into *cuda-gdb*
    - The Makefile has *debug-cuda-gdb* and *debug-cuda-gdb-solution* commands you can also try

Write Debuggable Software

A case for modularity, and proper test cases

- Think about interfaces in your code: Which parts must depend on each other, etc.
  - Example: BLAS, linear algebra routines
- Think about structure and architecture („the big picture“)
- Don‘t go overboard: „I read this book, we need 100% test coverage“, etc.
  - For many research codes that would be overkill
- **“Everything should be made as simple as possible, but no simpler.“**
- Badly structured legacy code slows you down as well, as it resists change
  - Today‘s code is tomorrow‘s legacy
  - Strike a balance, avoid full rewrites. Code encapsulates hard-earned bug fixes and knowledge

- Representative test cases
  - Contain the correct science, walk the code paths
  - But run quickly, best on a single process, should run on a single node
  - Some (but not all) tests at full scale

Debugging Performance

Why you must use profilers

- Paraphrasing Donald Knuth:
  - Don‘t overoptimize, but meta-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
- Focus on GPU and system-level: Nsight Systems
  - Continue with kernel analysis in Nsight Compute (tomorrow)



The Nsight Suite Components

How the pieces fit together

- Nsight **Systems**: Coarse-grained, whole-application
- Nsight **Compute**: Fine-grained, kernel-level
- NVTX: Support and structure across tools
- Main purpose: Performance optimization
  - But at their core, advanced measurement tools



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

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

- 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



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!
  - Performance Optimization session has more detail on Nsight Compute



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 into single timeline



#	Name	Start	Duration	TID	GPU	Context
4	Memset	1,88258s	3,200 µs	-	GPU 0	Stream 13
5	void jacobi_ke...	1,88259s	3,056 ms	-	GPU 0	Stream 13
6	Memcpy DtoD	1,88565s	5,024 µs	-	GPU 0	Stream 14
7	Memcpy DtoH	1,88565s	4,864 µs	-	GPU 0	Stream 13

Nsight Systems Basic Workflow

Navigating the timeline and finding interesting areas



Launching the Profilers

How-to on the JSC systems

- `module load GCC Nsight-Systems Nsight-Compute`
- Nsight Systems
  - *nsys* (CLI) and *nsys-ui* (GUI)
  - Record timeline:
    `nsys profile -o scale_um_baseline ./scale_vector_um`
  - Always specify a meaningful output file name. Auto-timestamping: `-o $(date +%Y%m%d_%H-%M-%S)__my_app`
- Nsight Compute
  - *ncu* (CLI) and *ncu-ui* (GUI)
  - Record all kernels, or (here) select specific instance:
    `ncu --set full -k scale -s 0 -c 1 -f -o scale_kernel_baseline ./scale_vector_um`
  - Nsight Systems can help generate the –s/-c arguments:



Task 3

Analyze and profile `scale_vector_um`

- Location of code: 2-Tools/exercises/tasks/task3
- See Instructions.ipynb
- Use the command line tools to gather a profile
  - Then use the GUI to view it: X-Forwarding, or Xpra (described in the .ipynb)
- Objective: Get to know the tools and basic workflow. Check the .ipynb and the Makefile:
  - Main Goal: Use Nsight Systems to write *scale_vector_um*'s timeline to file and open the result in the GUI
  - Try to determine:
    - Kernel runtime
    - CUDA API operations and their duration
  - Optional Goal: Use Nsight Compute to profile a specific kernel on the command line, then write the output to a file and open it in the GUI
    - What are the limiters of the kernel?

A first (I)Nsight

Recording with the CLI

- Use the command line
  - `srun nsys profile --trace=cuda,nvtx,mpi --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
```

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/

Nsight Systems Workflow With NVTX

Repeating the analysis



NVIDIA

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
  ```

NVIDIA
  ```

Other Profilers

Large-scale MPI profiling, custom tooling, and other uses

- Performance counters available via CUPTI (CUDA Profiling Tools Interface)
  - Build your own profiler (integration): https://docs.nvidia.com/cupti/index.html
- Score-P: Measurement infrastructure, can record CPU/GPU
- Cube: Display hierarchical info collected via Score-P
- Vampir: Analyze application traces, discover MPI issues
- … and many more



Summary

- Overview of GPU tools
  - Debugging with compute-sanitizer and cuda-gdb
  - Whole-program optimization with Nsight Systems
  - Individual kernels with Nsight Compute
- Profiler usage a "must" for performance optimization
  - ...puts the P in HPC
- Workflow is equally important
  - Increase GPU utilization („fill whitespace“)
  - Focus on top kernels, find their limiters, fix them
  - Implement and repeat

Questions?

mhrywniak@nvidia.com

Further Material

- GTC on-demand talks
  - What, Where, and Why? Use CUDA Developer Tools to Detect, Locate, and Explain Bugs and Bottlenecks (s41493, GTC 2022)
  - Tuning GPU Network and Memory Usage in Apache Spark (s31566, GTC 2022)
- Documentation for cuda-gdb, compute-sanitizer, Nsight Systems and Nsight Compute
  - In particular, the Kernel Profiling guide (installed with Nsight Compute, or online): https://docs.nvidia.com/nsight-compute/ProfilingGuide/index.html
- GTC labs from Nsight teams: https://github.com/NVIDIA/nsight-training
- Open Hackathons material, e.g., https://github.com/openhackathons-org/nways_accelerated_programming

NVIDIA