Performance and Debugging Tools

David Appelhans, NVIDIA Devtech Compute | SC25 / November 16th 2025

Goals for this session

Why you should use tools, and what they do

- All code has bugs. The more code, the more bugs.

- Different classes of “bugs“
  - Crashes, undefined behavior, deadlocks, correctness issues, …
  - Time to solution, parallel efficiency, energy efficiency
- Treat performance issues as bugs, especially at scale

- Manual debugging: Mimic program execution, print state, time code regions
- Tool-assisted debugging: Automate “tedious“ part
  - More information, less effort
  - Otherwise unavailable information (hardware counters!)
- Fixing bugs not (yet ☺) automated: Tools simplify and enable analysis

- Introduce workflow and representative tools, focus on distributed GPU applications

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 in GPU code
- 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

Focus on Performance

...without neglecting correctness

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

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 theme: workload is distributed; correctness/performance bug similar for all processes
- Run tool N times in parallel, have N output files, only look at 1 (or 2, ...)
- Even works for load imbalance (profile the rank spending least amount of time waiting in barriers).
  - For parallel race conditions parallel tools are best.
- How to get unique tool output?
- `%q{ENV_VAR}` supported by all the NVIDIA tools discussed here, embed environme
  - `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`

https://www.open-mpi.org/faq/?category=running#mpi-environmental-variables
http://mvapich.cse.ohio-state.edu/static/media/mvapich/mvapich2-userguide.html#x1-34600013
https://slurm.schedmd.com/srun.html#SECTION_OUTPUT-ENVIRONMENT-VARIABLES
Nsight Systems stitch multiple reports: https://docs.nvidia.com/nsight-systems/UserGuide/index.html#multi-report-mpi-example

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`

https://docs.nvidia.com/cuda/cuda-gdb/index.html#gpu-core-dump-support

Using cuda-gdb with MPI

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

```
#include <unistd.h>
if (rank == 0) {
    char name[255]; gethostname(name, sizeof(name)); bool attached;
    printf("rank %d: pid %d on %s ready to attach\n", rank, getpid(), name);
    while (!attached) { sleep(5); }
}
```

- Launch process, sleep on particular rank

```
$ srun -n 4 ./jacobi -niter 10
rank 0: pid 28920 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 28920
```

- Wake up sleeping process and continue debugging normally

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

JSC system shortcut: `sgoto --help`

Debugging Performance

Why you *must* use profilers

- Donald Knuth paraphrase: Don't optimize prematurely.
- Do not trust your gut instinct – very often misleading
  - Easy to waste a lot of time chasing the "perceived" issue
- Timers are not sufficient metrics. Need profilers.
- Optimization will not monotonically decrease runtime.
- Understand the goal of an optimization: use profiler to verify goal was accomplished, investigate new blocker.
- Don't give up prematurely based on runtime.

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

compute-sanitizer

Anatomy of an error

- Look into log file, or use `compute-sanitizer --read <save file>`
- Actual output can be very long: usually many GPU threads produce (similar) errors.

```
========= COMPUTE-SANITIZER
[....]
========= Invalid __global__ write of size 4 bytes
=========     at 0x6d0 in mpi/jacobi_kernels.cu:60:initialize_boundaries(float*, float*, float,
                                                                         int, int, int, int)
=========     by thread (1,0,0) in block (32,0,0)
=========     Address 0x14fb88020000 is out of bounds
=========     Saved host backtrace up to driver entry point at kernel launch time
=========     Host Frame: [0x20d6ea]
=========                 in libcuda.so.1
=========     Host Frame: [0x115ab]
[....]
=========
========= ERROR SUMMARY: 10 errors
```

- We introduced an off-by-one error in line **60** ourselves:

```
a_new[iy * nx + (nx - 1) + 1] = y0;
```

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 theme: workload is distributed; correctness/performance bug similar for all processes
- Run tool N times in parallel, have N output files, only look at 1 (or 2, …)
- Even works for load imbalance (profile the rank spending least amount of time waiting in barriers).
  - For parallel race conditions parallel tools are best.
- How to get unique tool output?
- `%q{ENV_VAR}` supported by all the NVIDIA tools discussed here, embed environme
  - `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
```

https://www.open-mpi.org/faq/?category=running#mpi-environmental-variables
http://mvapich.cse.ohio-state.edu/static/media/mvapich/mvapich2-userguide.html#x1-34600013
https://slurm.schedmd.com/srun.html#SECTION_OUTPUT-ENVIRONMENT-VARIABLES
Nsight Systems stitch multiple reports: https://docs.nvidia.com/nsight-systems/UserGuide/index.html#multi-report-mpi-example

Using cuda-gdb with MPI

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

```
#include <unistd.h>
if (rank == 0) {
    char name[255]; gethostname(name, sizeof(name)); bool attached;
    printf("rank %d: pid %d on %s ready to attach\n", rank, getpid(), name);
    while (!attached) { sleep(5); }
}
```

- Launch process, sleep on particular rank

```
$ srun –n 4 ./jacobi -niter 10
rank 0: pid 28920 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 28920
```

- Wake up sleeping process and continue debugging normally

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

JSC system shortcut:
`sgoto --help`

Using cuda-gdb with MPI

Environment variables for easier debugging

- Automatically wait for attach on exception without code changes:

```
$ CUDA_DEVICE_WAITS_ON_EXCEPTION=1 srun ./jacobi -niter 10
Single GPU jacobi relaxation: 10 iterations on 16384 x 16384 mesh with norm check every 1
iterations
jwb0129.juwels: The application encountered a device error and CUDA_DEVICE_WAITS_ON_EXCEPTION is
set. You can now attach a debugger to the application (PID 31562) for inspection.
```

- Same as before, go to node (see previous slide), then attach cuda-gdb:

```
$ cuda-gdb --pid 31562
CUDA Exception: Warp Illegal Address
The exception was triggered at PC 0x508ca70 (jacobi_kernels.cu:88)

Thread 1 "jacobi" received signal CUDA_EXCEPTION_14, Warp Illegal Address.
[Switching focus to CUDA kernel 0, grid 4, block (0,0,0), thread (0,20,0), device 0, sm 0, warp 21,
lane 0]
0x000000000508ca80 in jacobi_kernel<32, 32><<<(512,512,1),(32,32,1)>>> (/*...*/) at jacobi_kernels.c
u:88
88              real foo = *((real*)nullptr);
```

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`

https://docs.nvidia.com/cuda/cuda-gdb/index.html#gpu-core-dump-support

Example: Opening a Core Dump

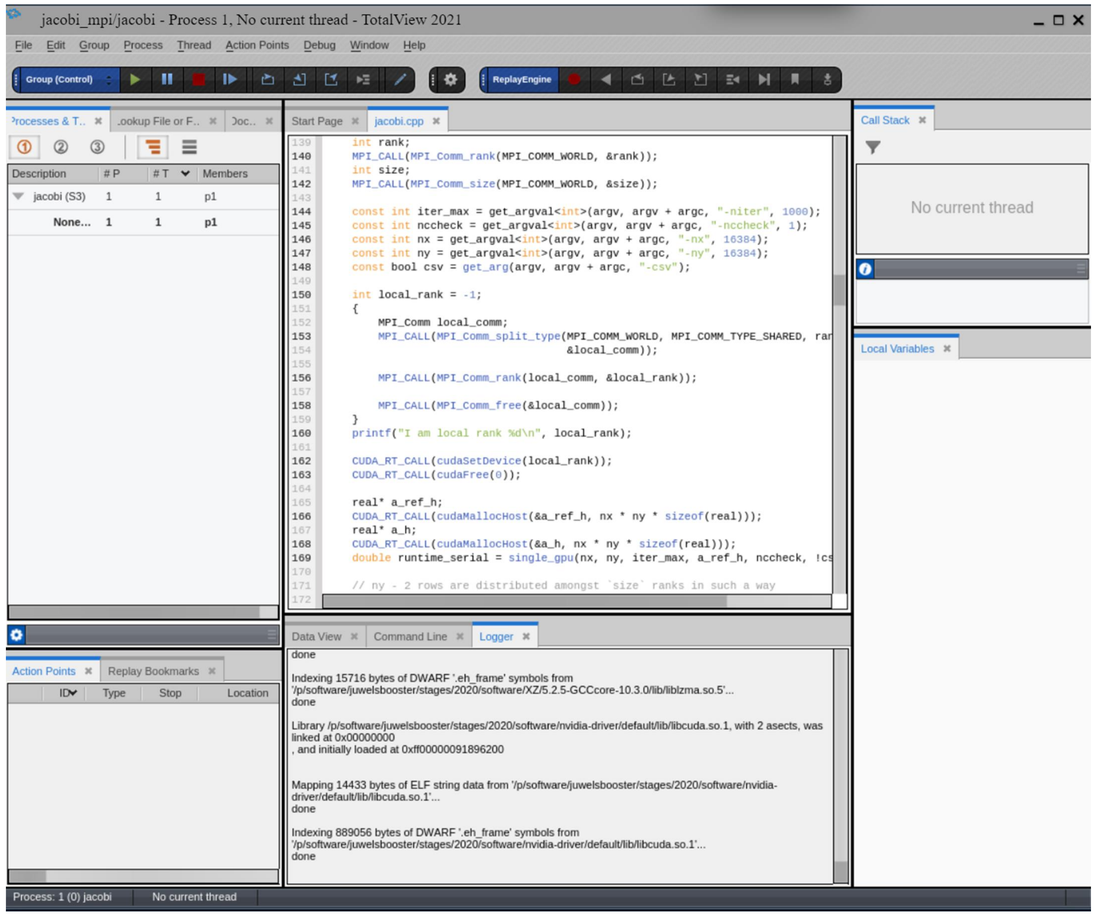
- Running and generating the core file

```
$ CUDA_ENABLE_COREDUMP_ON_EXCEPTION=1 CUDA_ENABLE_LIGHTWEIGHT_COREDUMP=1 srun ./jacobi -niter 10
Single GPU jacobi relaxation: 10 iterations on 16384 x 16384 mesh with norm check every 1 iterations
srun: error: jwb0021: tasks 0-3: Aborted (core dumped)

$ ls core*
core.jwb0021.juwels.23959  core_1633801834_jwb0021.juwels_23959.nvcudmp ...
```

- And opening the core dump in `cuda-gdb`

```
(cuda-gdb) target cudacore core_1633801834_jwb0021.juwels_23959.nvcudmp
Opening GPU coredump: core_1633801834_jwb0021.juwels_23959.nvcudmp

[New Thread 23979]

warning: No exception was found on the device

[Current focus set to CUDA kernel 0, grid 4, block (0,0,0), thread (0,2,0), device 0, sm 0, warp 0, lane 0]

#0  0x00000000057e1ae0 in void jacobi_kernel<32, 32>(float*, float const*, float*, int, int, int, bool)
   <<<(512,512,1),(32,32,1)>>> ()
    at /p/project/cexalab/hrywniak1/code/multi-gpu-programming-models/mpi/jacobi_kernels.cu:87

87              real foo = *((real*)nullptr);

(cuda-gdb)
```

Specialized parallel debuggers



- `cuda-gdb` can debug multiple processes (`add-inferior`), although...
- For truly parallel bugs (e.g. multi-node, multi-process race conditions), third-party tools offer more convenience
  - Or enable „live“ analysis in the first place
- Linaro DDT (formerly ARM)
- Perforce TotalView (screenshot)
  - Available as module on JSC systems

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

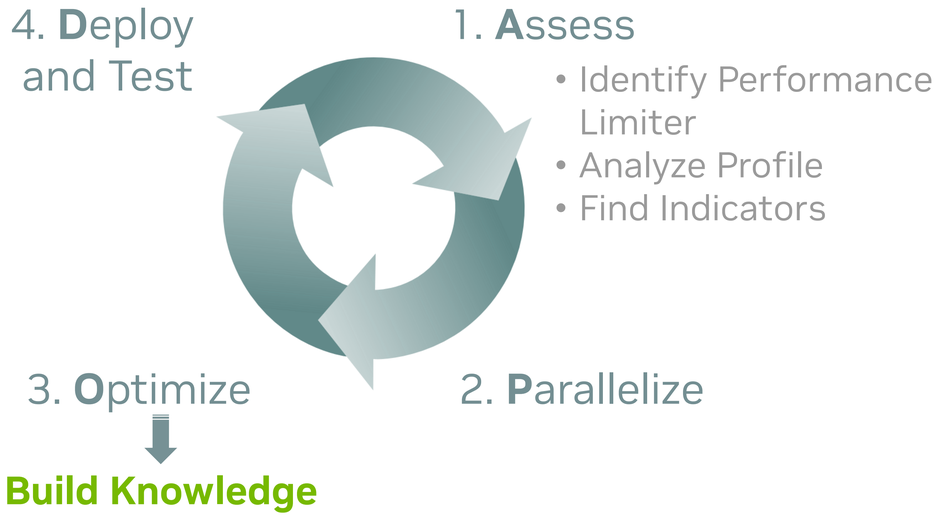
Why you *must* use profilers

- Donald Knuth paraphrase: Don‘t optimize prematurely.
- Do not trust your gut instinct – very often misleading
  - Easy to waste a lot of time chasing the "perceived" issue
- Timers are not sufficient metrics. Need profilers.
- Optimization will not monotonically decrease runtime.
- Understand the goal of an optimization: use profiler to verify goal was accomplished, investigate new blocker.
- Don’t give up prematurely based on runtime.

Debugging Performance

Why you *must* use profilers

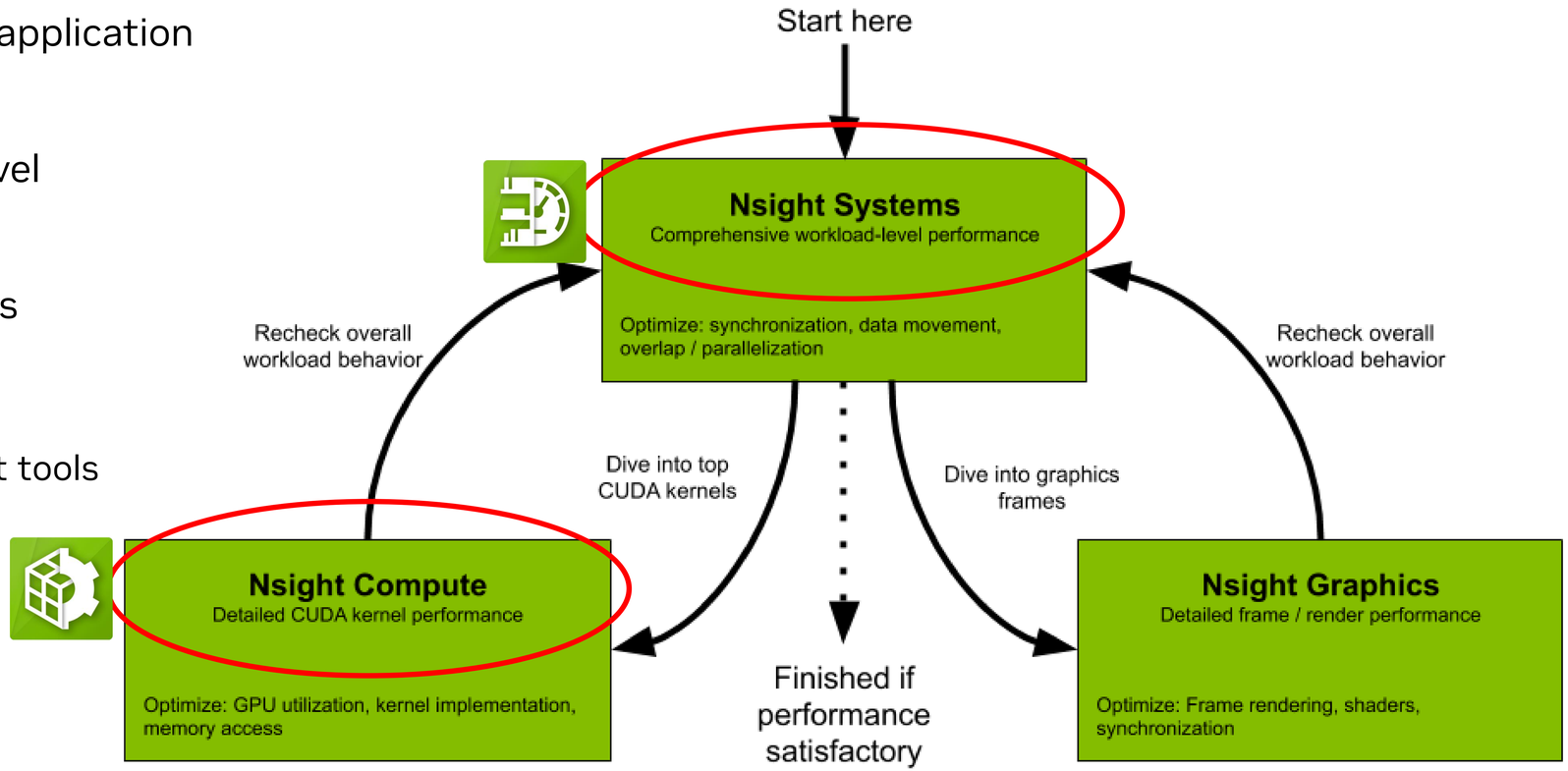
- Iterative workflow
- Different kinds of measurement tools, different tradeoffs
  - Instrumenting/Sampling
  - Profiling/Tracing
  - multi-process, single-process, kernel-level
- Here: Focus on GPU and system-level: Nsight Systems



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



A First (I)Nsight

Recording with the CLI

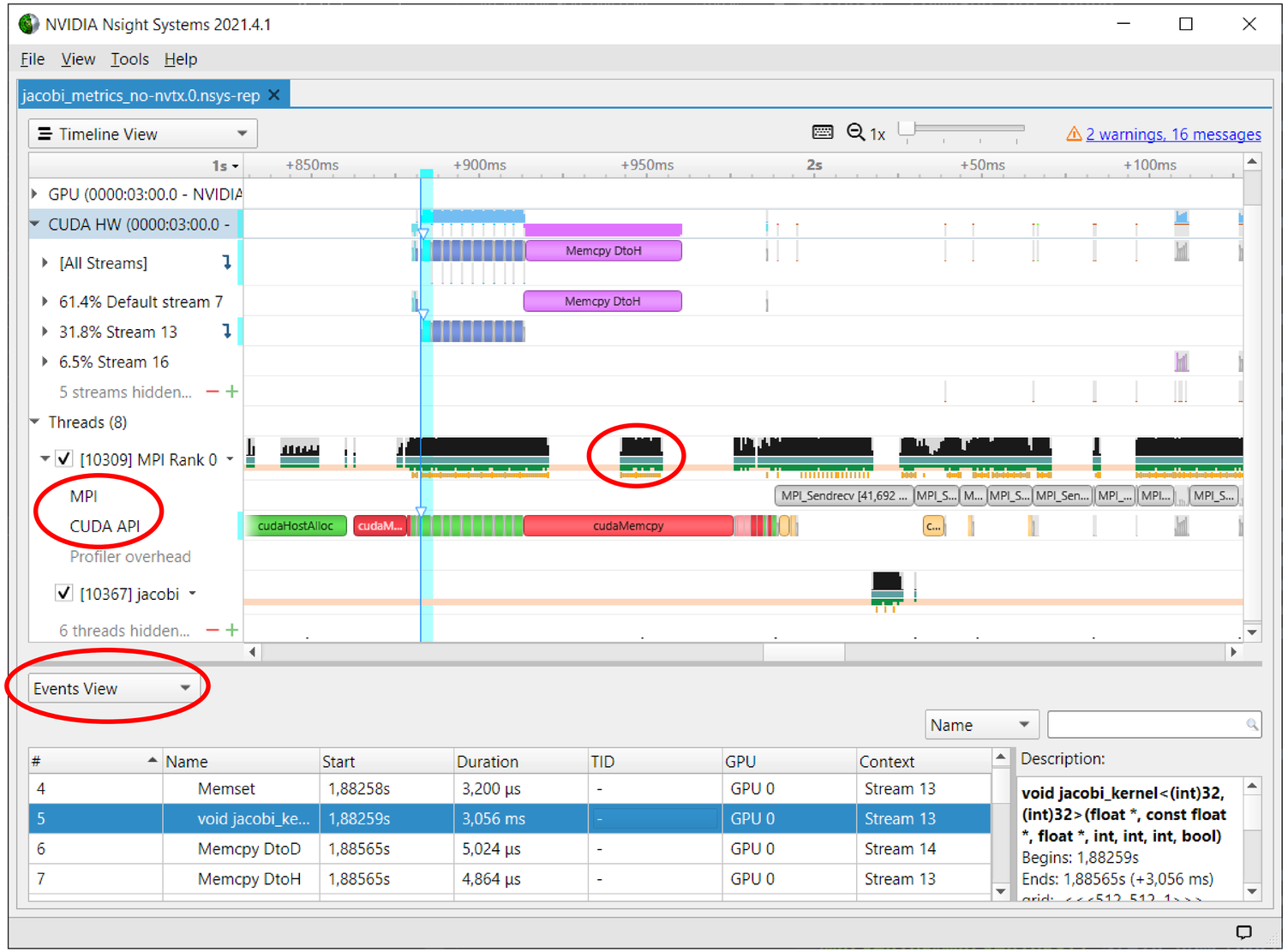
- Use the command line
  - `srun nsys profile --trace=cuda,nvtx,mpi --output=my_report.%q{SLURM_PROCID} ./jacobi -niter 10`
- Inspect results: Open the report file in the GUI
  - Also possible to get details on command line
  - Either add `--stats` to profile command line, or: `nsys stats --help`

- 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
```

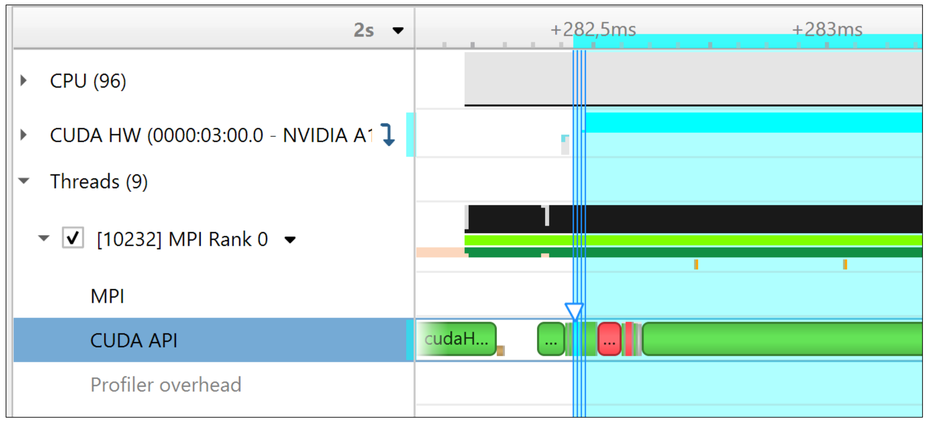
System-Level Profiling with Nsight Systems

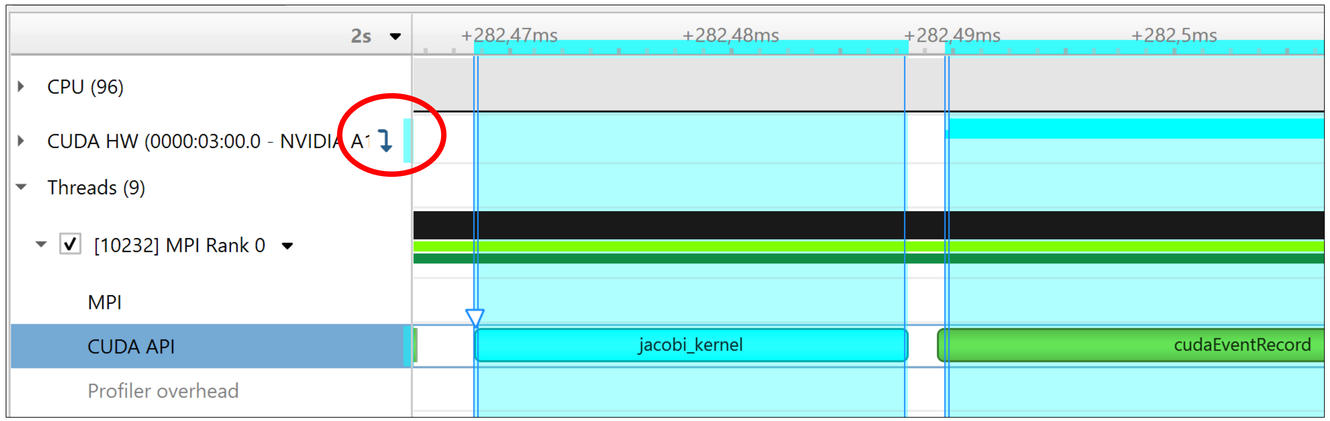
- Global timeline view
  - CUDA HW: streams, kernels, memory
- Different traces, e.g. CUDA, MPI
  - correlations API <-> HW
- Stack samples
  - bottom-up, top-down for CPU code
- GPU metrics
- Events View
  - Expert Systems
- looks at single process (tree)
  - correlate multi-process reports in single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



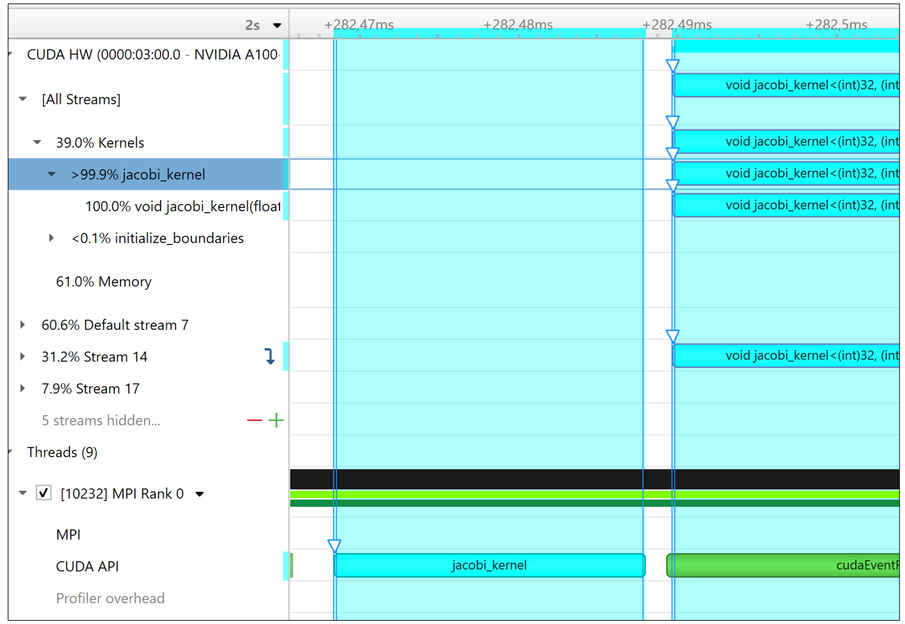
Correlating Events on the Timeline

Selecting events in one row highlights related events



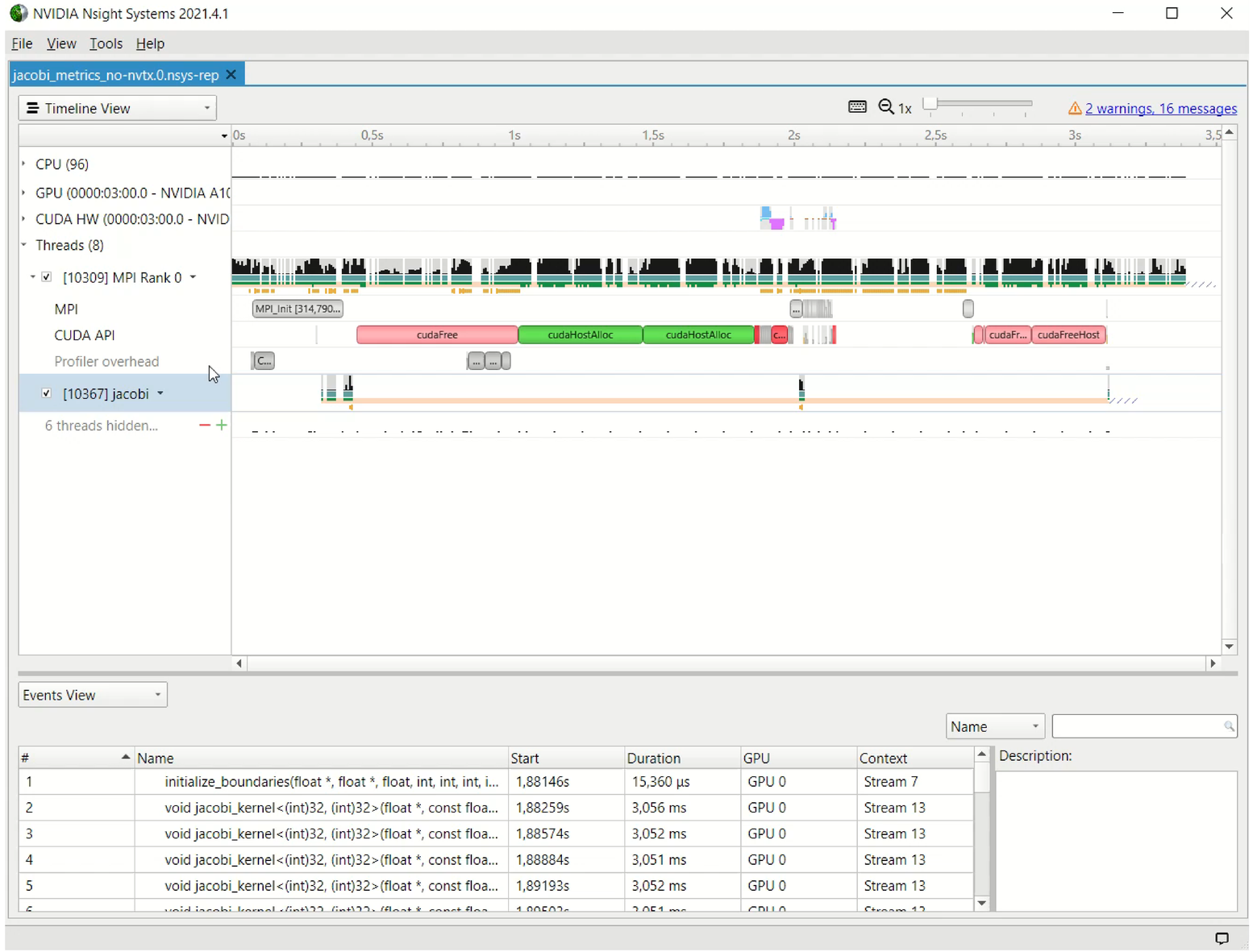


- CUDA's execution model is asynchronous
  - Kernel launch on host returns
  - Kernel runs on GPU
- Visualized in profiler



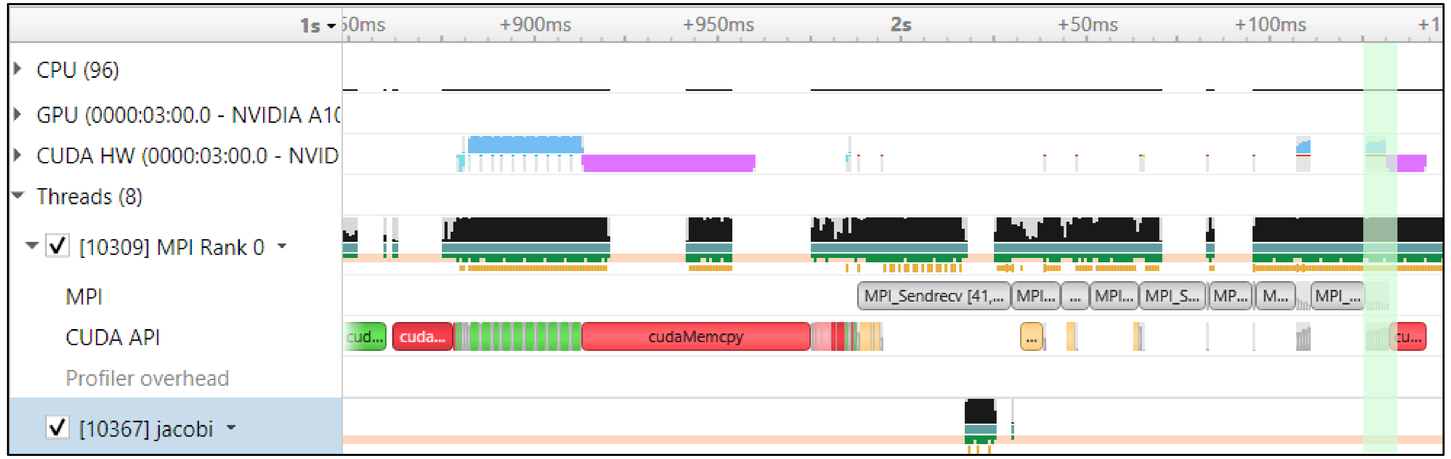
Nsight Systems Basic Workflow

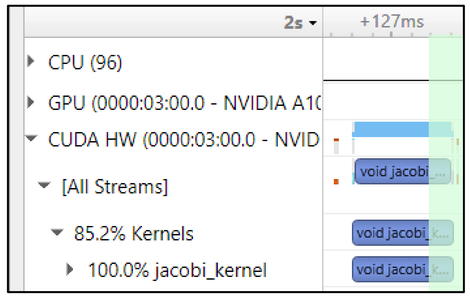
Navigating the timeline and finding interesting areas

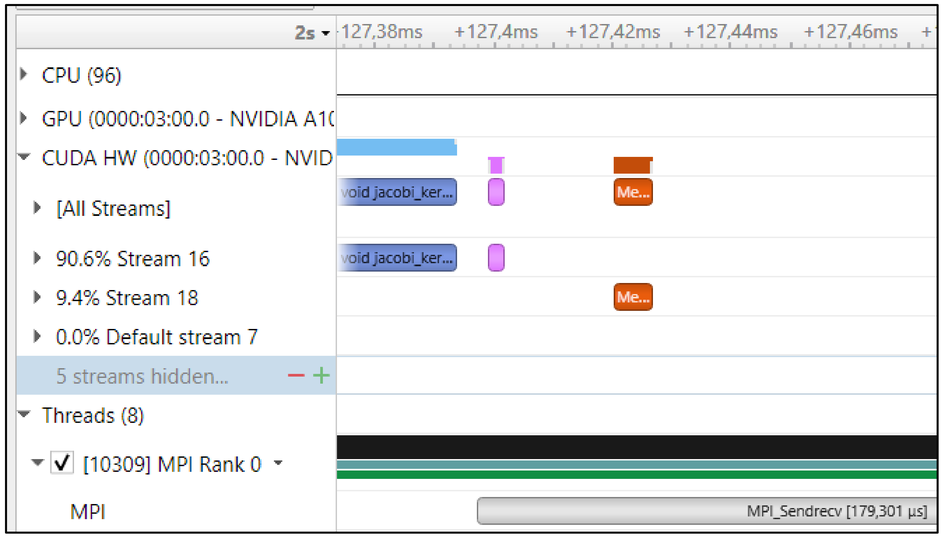


Discovering Optimization Potential

- Using our Jacobi example (see exercise)
- 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?



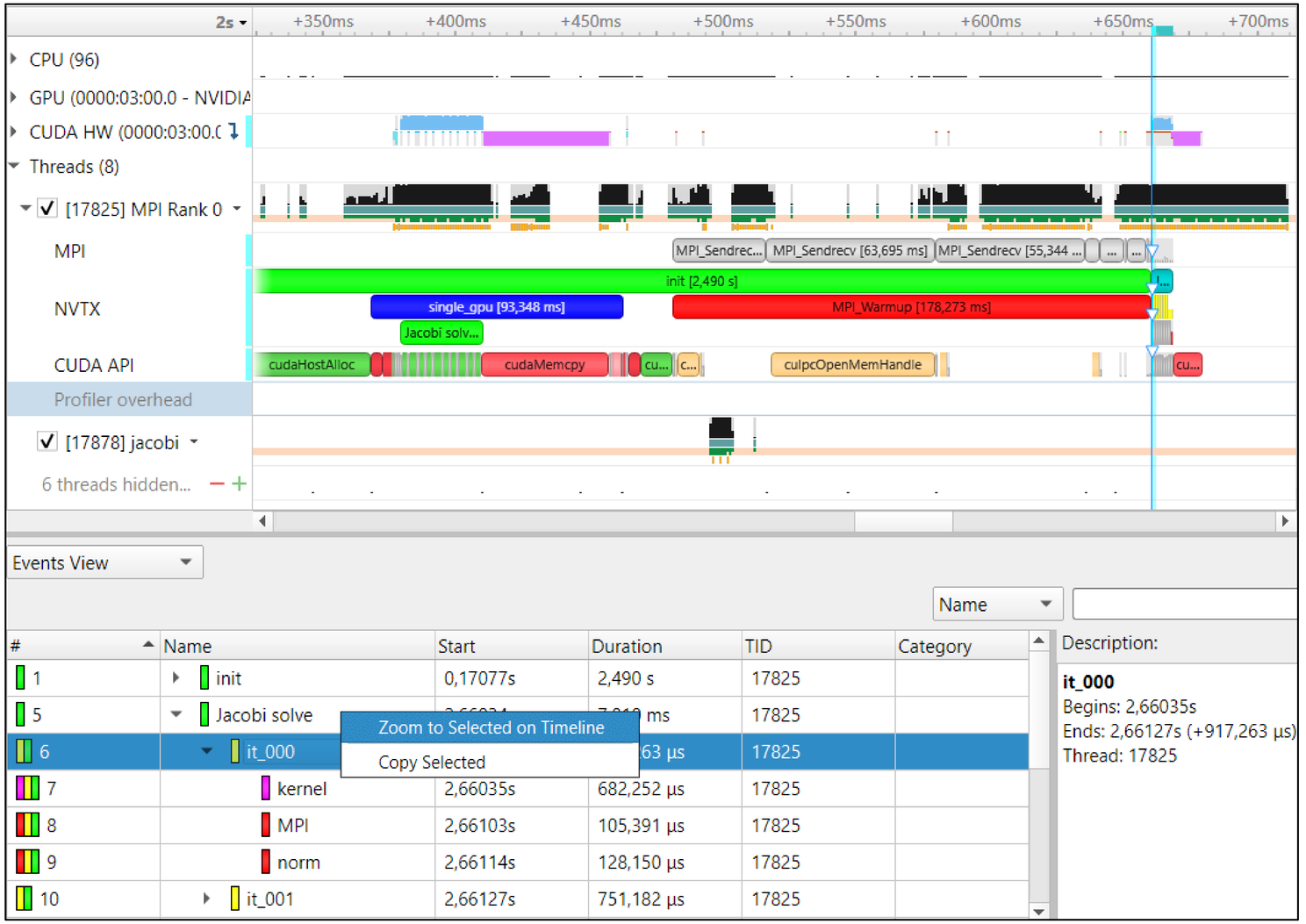




Adding some Color

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 `-cudalib=nvtx`
  - 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
}
```

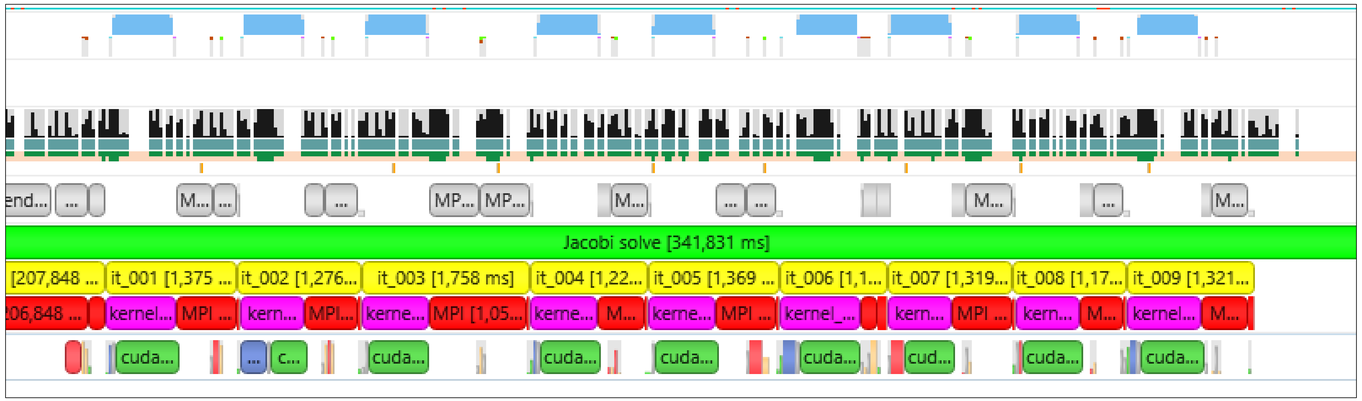
https://github.com/NVIDIA/NVTX and https://nvidia.github.io/NVTX/#how-do-i-use-nvtx-in-my-code

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

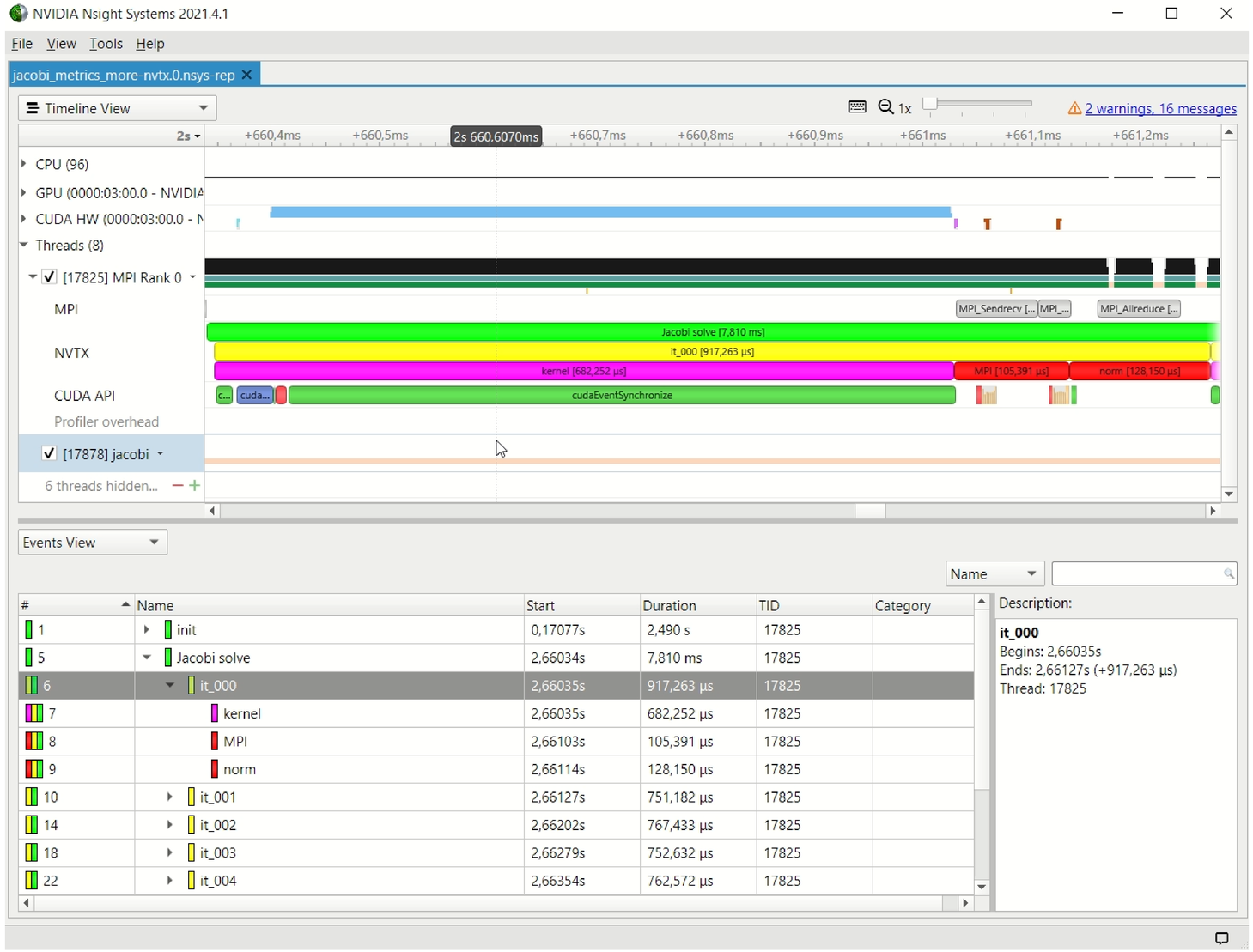
Shorter time, smaller files = quicker progress

- 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

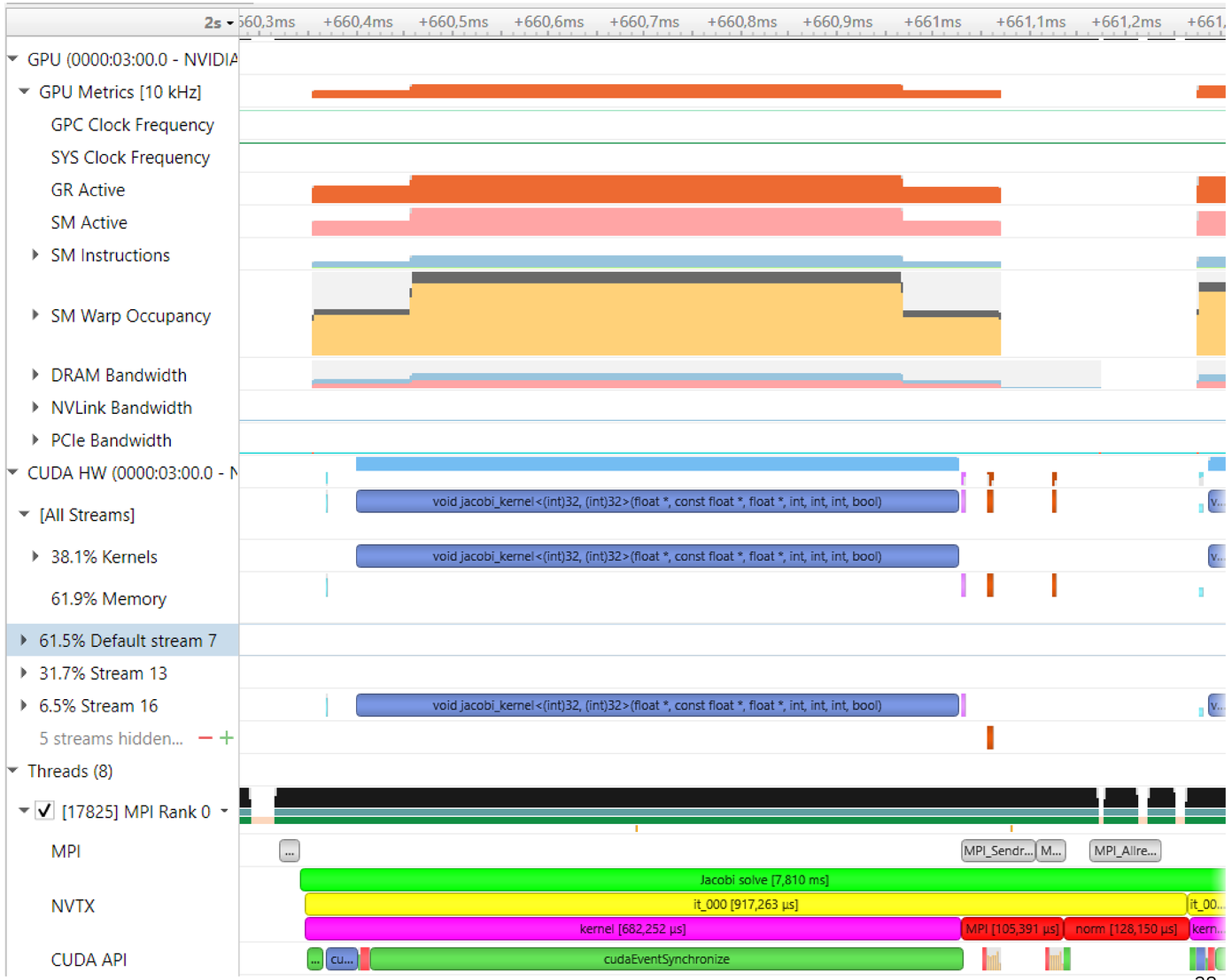
...and other traces you can activate

- 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`

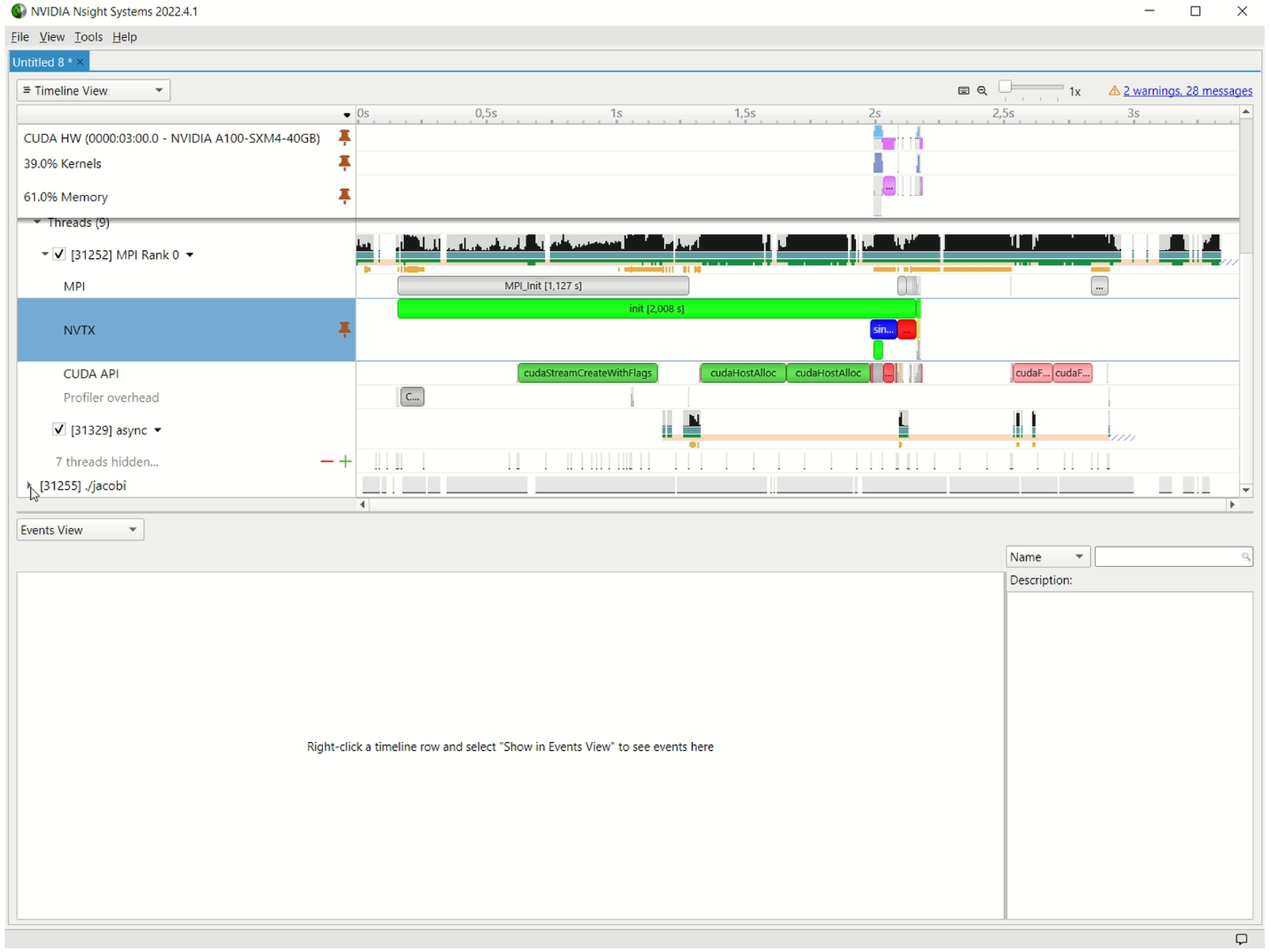
Focusing the Analysis

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?
  - later lectures

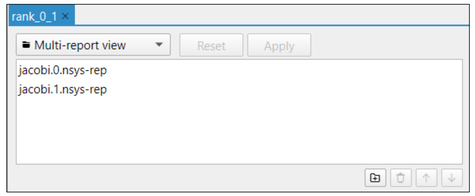


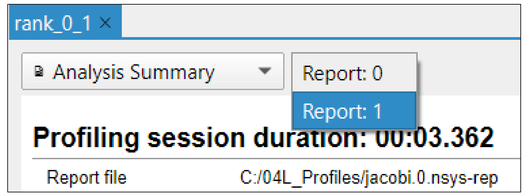
Using Multiple Reports in Nsight Systems

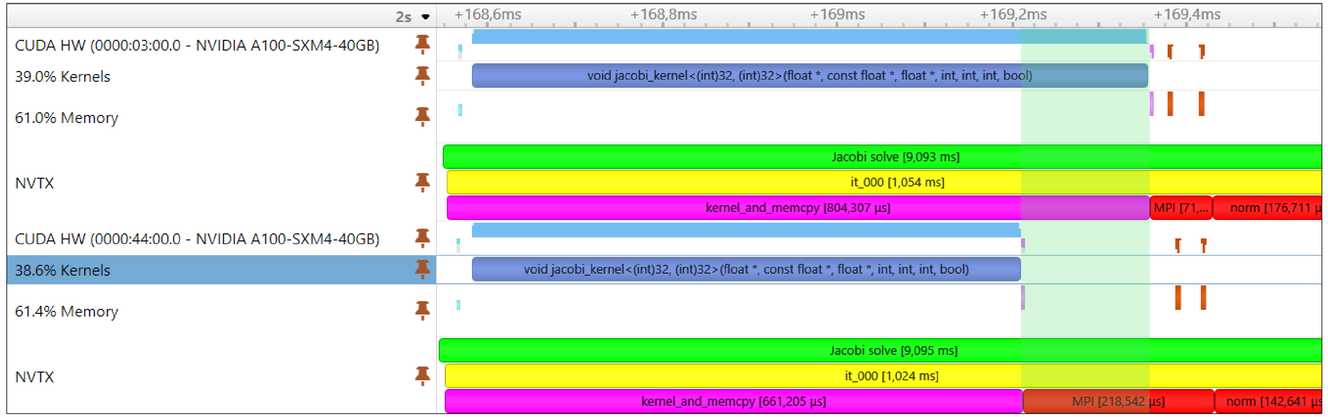


Multi-Process GPU Analysis

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







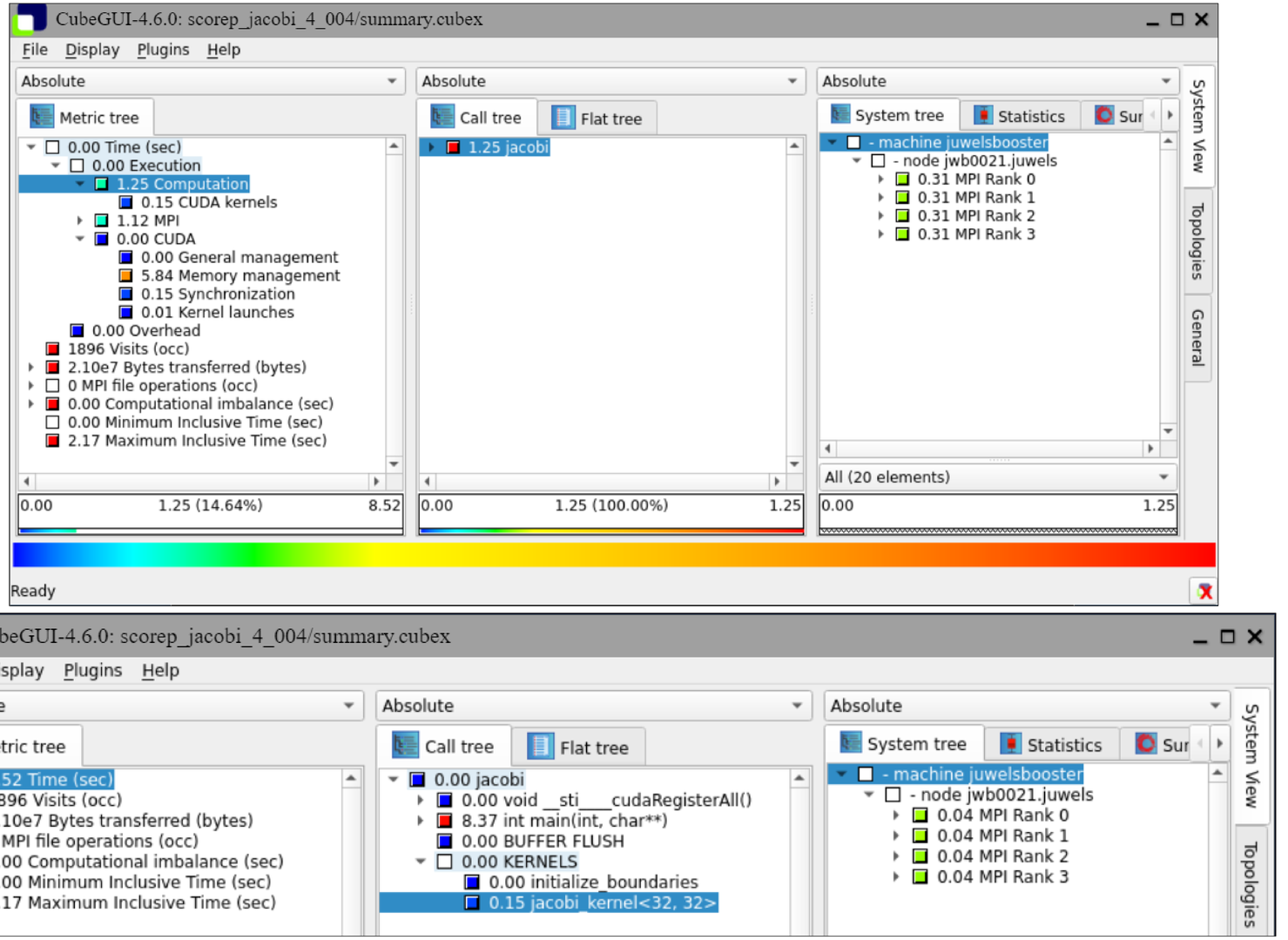
Community Profiling Tools

Specialized for large-scale distributed analysis

- Detecting issues at scale of thousands of GPUs (and processes)
  - Need to slice and dice data, too much to make sense of raw data
- Common measurement/instrumentation infrastructure: *Score-P*
  - Prefix all compilation/linker commands with `scorep –cuda`
- GPU data integration
  - CUDA profiling tools interface (CUPTI)
- Run the application to collect…
  - profiling data, for Scalasca
  - tracing data, for Vampir
  - (selection of tools not exhaustive)
- Tracing in particular: Careful tuning to keep overhead low (filtering)

Scalasca / CUBE

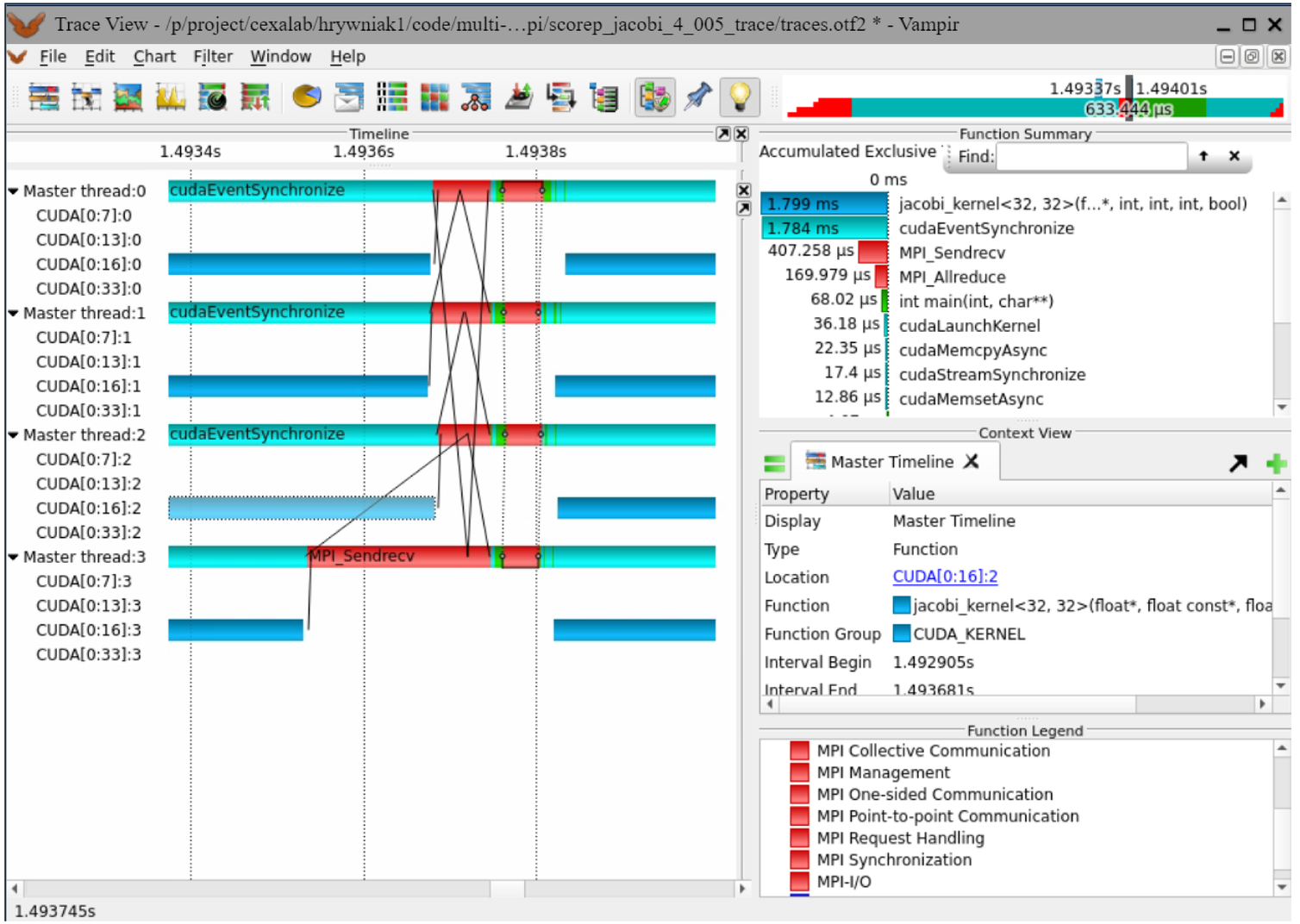
- Breakdown of different metrics across functions and processes
- Left-to-right: Selection influences breakdown
- Expanding changes inclusive/exclusive
- Example analysis:
  - Detect computational imbalance
- https://scalasca.org/



Vampir trace

- Analyze multi-process patterns
- What you can see in screenshot
  - Main timeline
  - Function summary
- Example analysis: Pinpoint MPI message relationships
  - e.g. late sender issues

- https://vampir.eu/



Summary

- Looked at a wide selection of different tools
  - compute-sanitizer
  - cuda-gdb
  - Nsight-Systems
  - Score-P: Scalasca, Vampir, …
  - and don‘t forget compiler flags and checks
- Correctness is paramount, but so is optimal resource usage
- Pick right tool for the job – and take the time to learn it thoroughly
  - Do not trust your gut when analyzing performance, easy to be misled
  - How to adapt serial (or small-scale) tooling to highly distributed applications
- Meant as guideline, not gospel
  - Especially performance issues often require creativity to solve
- Workflow is equally important

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 and Nsight Systems and blog posts:
  - Efficient CUDA Debugging: Memory Initialization and Thread Synchronization with NVIDIA Compute Sanitizer
  - Optimizing CUDA Memory Transfers with NVIDIA Nsight Systems
  - …and other posts: https://developer.nvidia.com/blog/tag/nsight/
- GTC labs from Nsight teams: https://github.com/NVIDIA/nsight-training
- GPU bootcamp material, e.g., https://github.com/openhackathons-org/nways_accelerated_programming