Parallel Processing - Tutorials

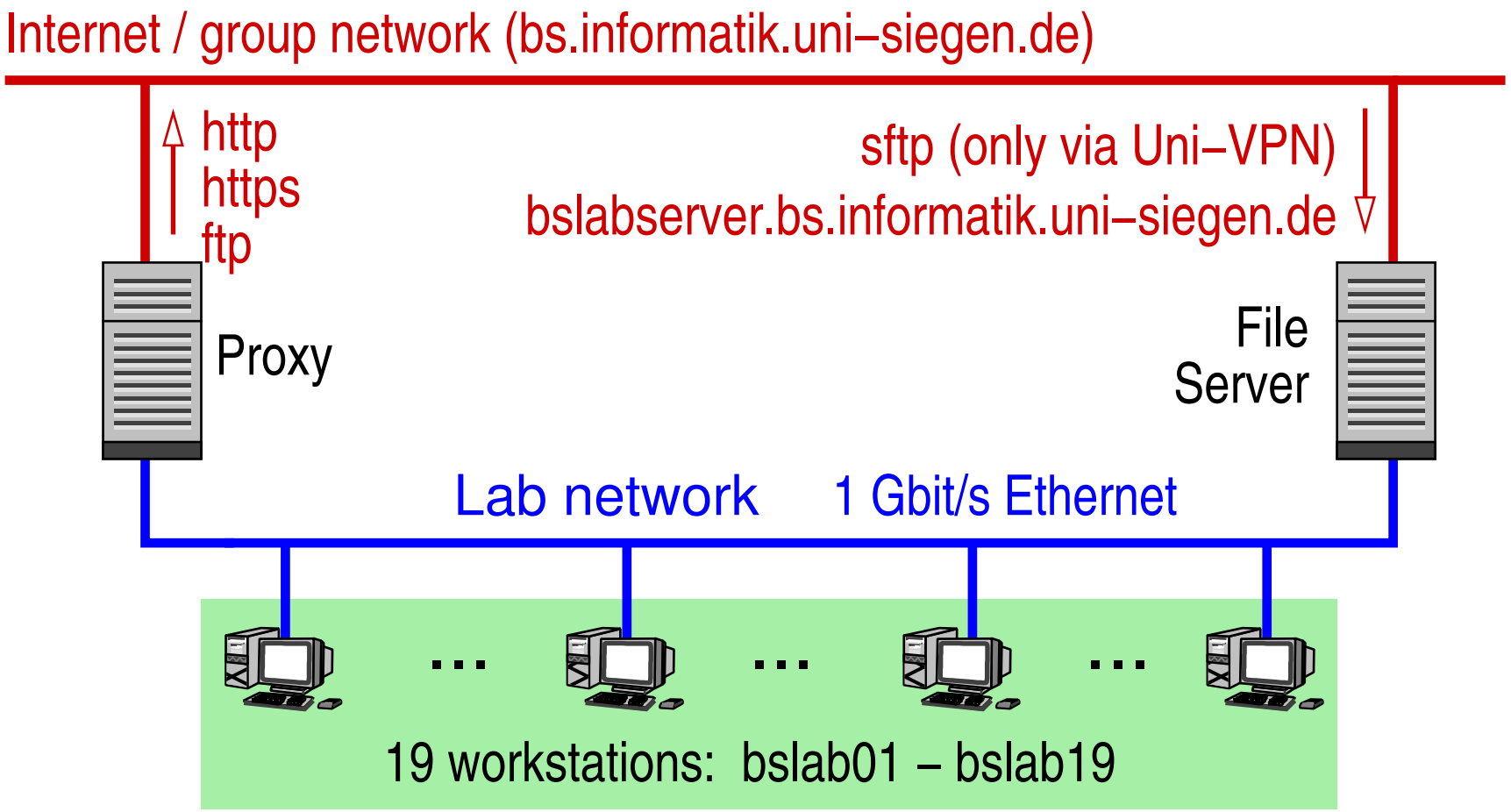
Roland Wismüller
Universität Siegen
roland.wismueller@uni-siegen.de
Tel.: 0271/740-4050, Büro: H-B 8404

Stand: October 31, 2025

Parallel Processing - Tutorials

1 Computing Environment (H-A 4111)

Overview of the Lab



Intel Core Ultra 7, 8+12 cores, 5.2/4.6 GHz, 32 GiB RAM

+ RTX 4060, 3072 cores, 1.8 GHz, 8 GiB RAM

Software Environment

- OS: Linux Debian 12, Kernel 6.1.0
- Compiler:
  - GNU `g++`, version 12.2
    - fully supports OpenMP 4.5, partial support for OpenMP 5.0
  - NVIDIA `nvc++`, version 25.7
    - supports most of OpenMP 4.5, plus subset of OpenMP 5.0
- Debugging: `gdb` (only for code compiled with `g++`)
- Performance analysis:
  - `scalasca`, version 2.6.2 (for OpenMP and MPI)
  - NVIDIA Nsight Compute, `ncu-ui`, version 2025.2.1 (for GPU)

Software Environment ...

- MPI: MPICH version 4.2.3
  - implements MPI version 4.1

Enabling / Switching Environments

- Default environment: `g++` with OpenMP
- NVIDIA environment: `nvc++` with OpenMP and GPU offloading
  - to enable: `module load nvhpc`
  - to return to default: `module unload nvhpc`
- PV environment: MPI and/or Scalasca
  - not compatible with NVIDIA environment!
  - to enable: `module load pv`
  - to disable: `module unload pv`

Some Useful Tools

- `lstopo`
  - shows CPU cores plus cache size for each cache level
- `taskset`
  - allows to define the core(s) where a program should start
  - `taskset -c 0-7 prog` # execute *prog* on performance cores
  - `taskset -c 8-19 prog` # execute *prog* on efficiency cores
- `nvtop`
  - shows GPU usage and processes using the GPU

CPU: Intel Core Ultra 7 265

- 8 Performance Cores (0-7):
  - max. 5.2 GHz
  - 48 KB L1 Data Cache, 3 MB L2 Cache
- 12 Efficiency Cores (8-20):
  - max. 4.6 GHz
  - 32 KB L1 Data Cache, 4 cores share 4 MB L2 Cache
- 30 MB shared L3 Cache
- Theoretical peak performance of one performance core:
  83 GFlop/s (64-bit)
- Maximum memory bandwitdh: 89 GB/s

GPU: NVidia RTX 4060, ADA architecture, AD107

- 24 Streaming Multiprocessors (SM)
- 3072 FP32 cores, 48 FP64 cores
- max. 1.8 GHz
- 65536 Registers per SM
- Theoretical peak performance:
  - 15.11 TFlop/s (FP32)
  - 236.2 GFlop/s (FP64)
- Theoretical memory throughput: 272 GB/s

Parallel Processing - Tutorials

2 Tools for OpenMP

2.1 Compiling

- Two alternative Compilers in the lab H-A 4111: `g++` and `nvc++`
- To use `nvc++`, load the corresponding module:
  - `module load nvhpc`
- In order to return to using `g++`, unload the module:
  - `module unload nvhpc`
- Compilation:
  - `g++ -fopenmp myProg.cpp -o myProg`

  or
  - `nvc++ -mp myProg.cpp -o myProg`
- Add option '`-O`' or '`-O3`' for optimization

2.2 Executing

- Identical to sequential programs, e.g.:
  - `./myProg`
- (Maximum) number of threads can be defined in environment variable `OMP_NUM_THREADS`:
  - `export OMP_NUM_THREADS=4`
  - applies to all programs started in the same shell
- Temporary (re-)definition of `OMP_NUM_THREADS`:
  - `OMP_NUM_THREADS=2 ./myProg`
  - useful, e.g., for speedup measurements

2.3 Debugging

- There are only few debuggers that fully support OpenMP
  - e.g., Totalview
  - requires tight cooperation between compiler and debugger
  - not available in the lab H-A 4111
- In the lab H-A 4111 (and other Linux PCs):
  - `gdb` allows halfway reasonable debugging
    - as it supports multiple threads
  - `gdb` is the standard LINUX debugger
    - text based, no GUI

2.3 Debugging ...

- Prerequisite: compilation with debugging information
  - sequential: `g++ -g -o myProg myProg.cpp`
  - with OpenMP: `g++ -g -fopenmp ...`
- Limited(!) debugging is also possible in combination with optimization
  - however, the debugger may show unexpected behavior
  - if possible: switch off the optimization
    - `g++ -g -O0 ...`

Important functions of a debugger (Examples for `gdb`):

- Start the programm: `run arg1 arg2`
- Set breakpoints on code lines: `break file.cpp:35`
- Set breakpoints on functions: `break myFunc`
- Show the procedure call stack: `where`
- Navigate in the procedure call stack: `up` bzw. `down`
- Show the contents of variables: `print i`
- Change the contents of variables: `set variable i=i*15`
- Continue the program (after a breakpoint): `continue`
- Single-step execution: `step` bzw. `next`

Important functions of a debugger (Examples for `gdb`): ...

- Show all threads: `info threads`
- Select a thread: `thread 2`
  - subsequent commands typically only affect the selected thread
- Source code listing: `list`
- Help: `help`
- Exit the debugger: `quit`

- All commands can also be abbreviated in `gdb`

Sample session with gdb (sequential)

```
bsclk01> g++ -g -O0 -o ross ross.cpp   ← Option -g for debugging
bsclk01> gdb ./ross
GNU gdb 6.6
Copyright 2006 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public ...
(gdb) b main   ← Set breakpoint on function main
Breakpoint 1 at 0x400d00: file ross.cpp, line 289.
(gdb) run 5 5 0   ← Start program with given arguments
Starting program: /home/wismueller/LEHRE/pv/ross 5 5 0
Breakpoint 1, main (argc=4, argv=0x7fff0a131488) at ross.cpp:289
289     if (argc != 4) {
(gdb) list   ← Listing around the current line
284
285     /*
286     ** Get and check the command line arguments
```

2.3 Debugging ...

```
287     */
288
289     if (argc != 4) {
290         cerr << "Usage: ross <size_x> <size_y> ...
291         cerr << "       <size_x> <size_y>: size...
292         cerr << "       <all>: 0 = compute one ...
293         cerr << "              1 = compute all ...
(gdb) b 315   ← Set breakpoint on line 315
Breakpoint 2 at 0x400e59: file ross.cpp, line 315.
(gdb) c   ← Continue the program
Continuing.
Breakpoint 2, main (argc=4, argv=0x7fff0a131488) at ross.cpp:315
315         num_moves = Find_Route(size_x, size_y, moves);
(gdb) n   ← Execute next source line (here: 315)
320         if (num_moves >= 0) {
(gdb) p num_moves   ← Print contents of num_moves
$1 = 24
```

2.3 Debugging ...

```
(gdb) where   ← Where is the program currently stopped?
#0  main (argc=4, argv=0x7fff0a131488) at ross.cpp:320
(gdb) c   ← Continue program
Continuing.
Solution:
  ...
Program exited normally.
(gdb) q   ← exit gdb
bsclk01>
```

Sample session with gdb (OpenMP)

```
bslab03> g++ -fopenmp -O0 -g -o heat heat.cpp solver-jacobi.cpp
bslab03> gdb ./heat
GNU gdb (GDB) SUSE (7.5.1-2.1.1)
...
(gdb) run 500
...
Program received signal SIGFPE, Arithmetic exception.
0x0000000000401711 in solver._omp_fn.0 () at solver-jacobi.cpp:58
58                                  b[i][j] = i/(i-100);
(gdb) info threads
  Id   Target Id         Frame
  4    Thread ... (LWP 6429) ... in ... at solver-jacobi.cpp:59
  3    Thread ... (LWP 6428) ... in ... at solver-jacobi.cpp:59
  2    Thread ... (LWP 6427) ... in ... at solver-jacobi.cpp:63
* 1    Thread ... (LWP 6423) ... in ... at solver-jacobi.cpp:58
(gdb) q
```

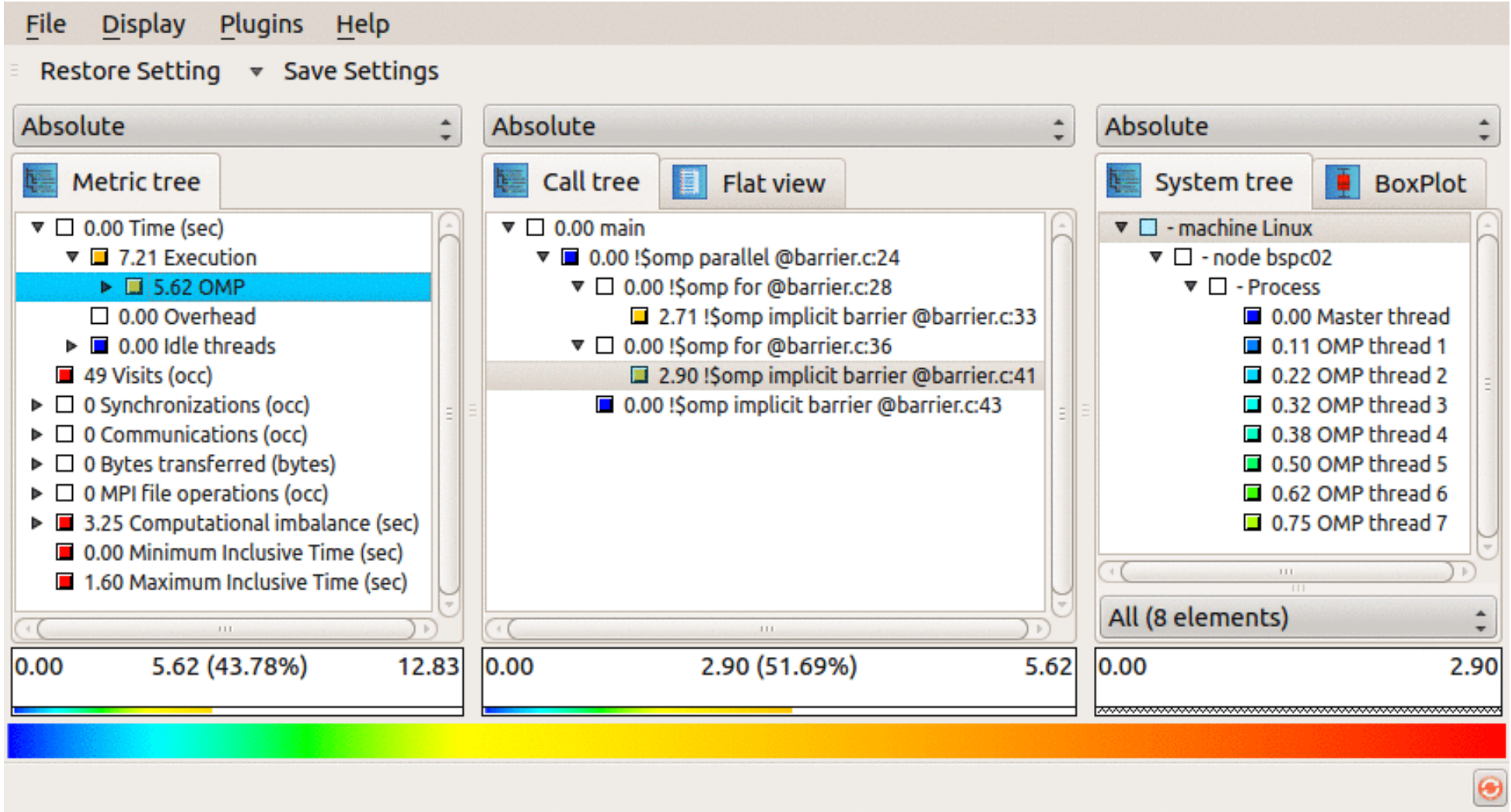
2.4 Performance Analysis

- Typically: **instrumentation** of the generated executable code during/after the compilation
  - insertion of code at important places in the program
    - in order monitor relevant events
    - e.g., at the beginning/end of parallel regions, barriers, ...
  - during the execution, the events will be
    - individually logged in a **trace file** (***Spurdatei***)
    - or already summarized into a **profile**
  - evaluation is done after the program terminates
  - c.f. Section 2.9.6 of the lecture
- Example: Scalasca
  - see `https://www.scalasca.org/scalasca/software`

Performance analysis using Scalasca

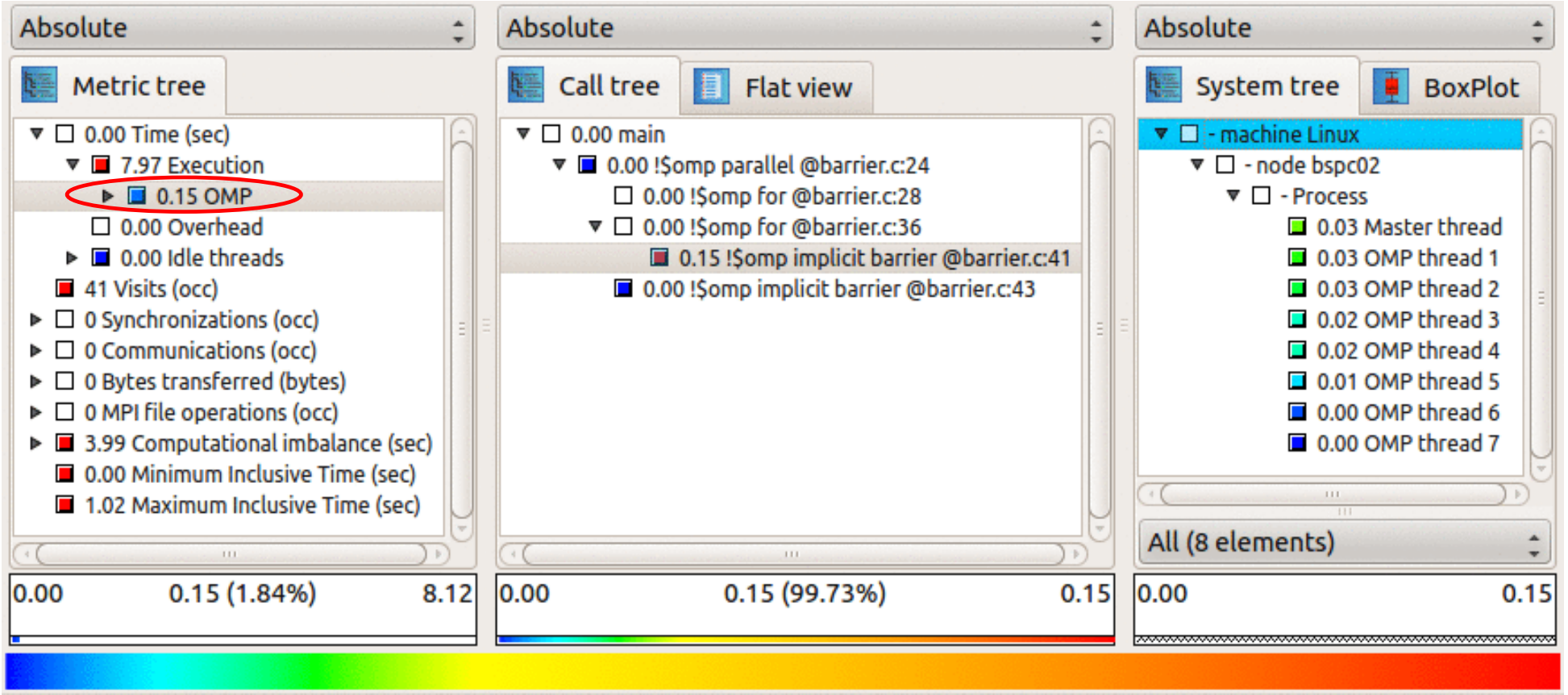
- Load the `pv` module to set the paths etc.:
  - `module load pv`
- Compile the program:
  - `scalasca -instrument g++ -fopenmp ...  barrier.cpp`
- Execute the program:
  - `scalasca -analyze ./barrrier`
  - stores data in a directory `scorep_barrier_0x0_sum`
    - `0x0` indicates the number of threads (0 = default)
    - directory must not yet exist; remove it, if necessary
- Interactive analysis of the recorded data:
  - `scalasca -examine scorep_barrier_0x0_sum`

Scalasca: Example from Sect. 3.3.5



Scalasca: Example from Sect. 3.3.5 ...

- In the example, the waiting time at barriers in the first loop can be reduced drastically by using the option `nowait`:



Parallel Processing - Tutorials

3 GPU Programming with OpenMP

3.1 Compilation

- In the lab H-A 4111: use `nvc++`
- To do so, first load the corresponding module:
  - `module load nvhpc`
- Compilation:
  - `nvc++ -O3 -mp=gpu -gpu=cc89 myProg.cpp -o myProg`
  - `cc89` specifies *Compute Capability 8.9* (for RTX 4060)
- For optimization: try the option '`-gpu=cc89,maxregcount:`*N*'
  - where *N* specifies the maximum number of registers that a kernel will use

3.2 NVidia Nsight Compute

- Performance analysis tool available in the lab room H-A 4111
- Graphical tool for profiling CUDA kernels on an NVIDIA GPUs
  - CUDA kernel: function that is executed by multiple threads on the cores of the GPU
  - when using OpenMP, CUDA kernels are generated by the OpenMP compiler from the code sections marked with a `target` directive

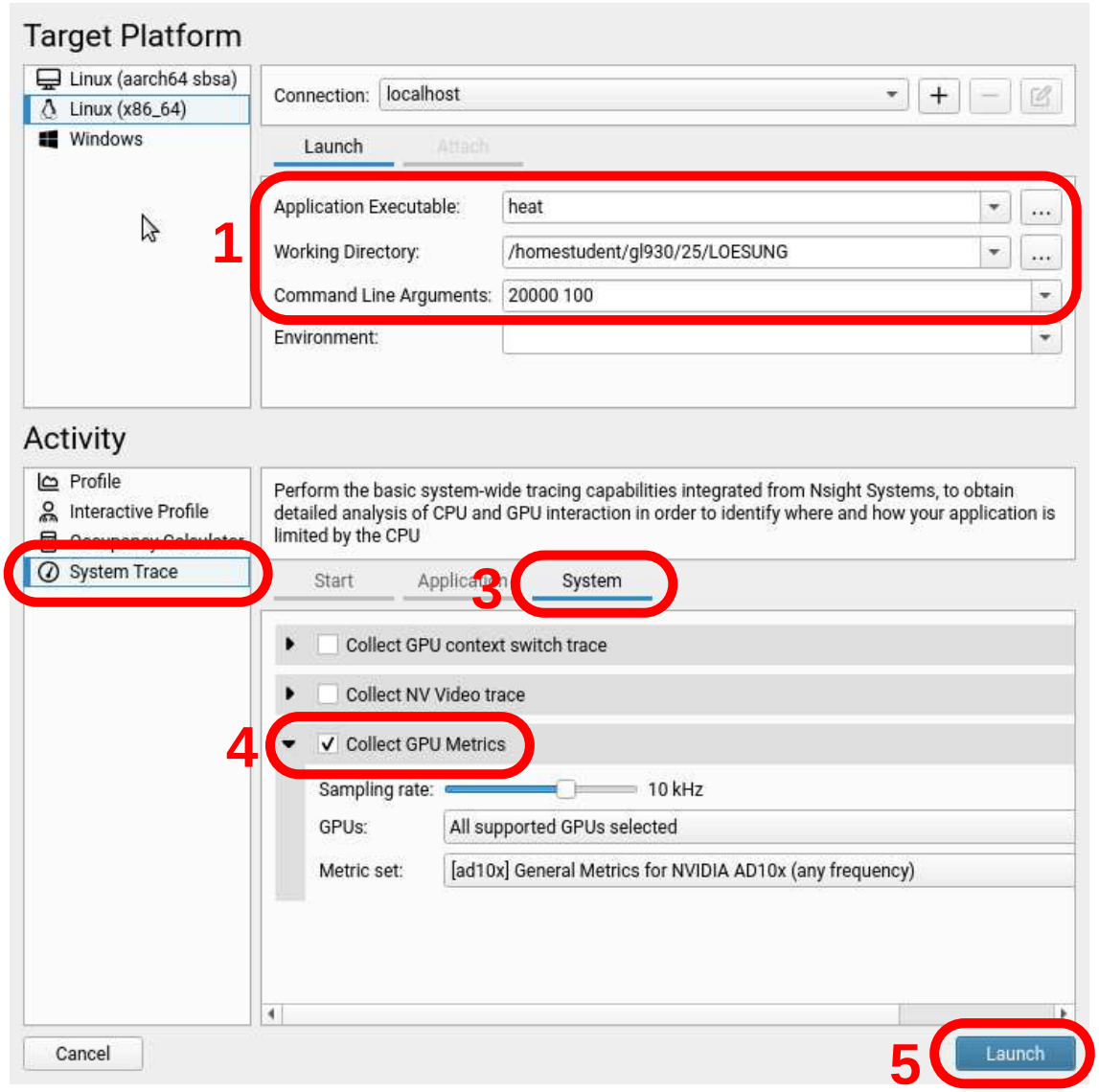
Starting Nsight Compute

- Load the `nvhpc` module:
  - `module load nvhpc`
- Compile your program, e.g.:
  - `nvc++ -O3 -mp=gpu -gpu=cc89 -o heat heat.cpp ...`
- Start the Nsight Compute GUI:
  - `ncu-ui`

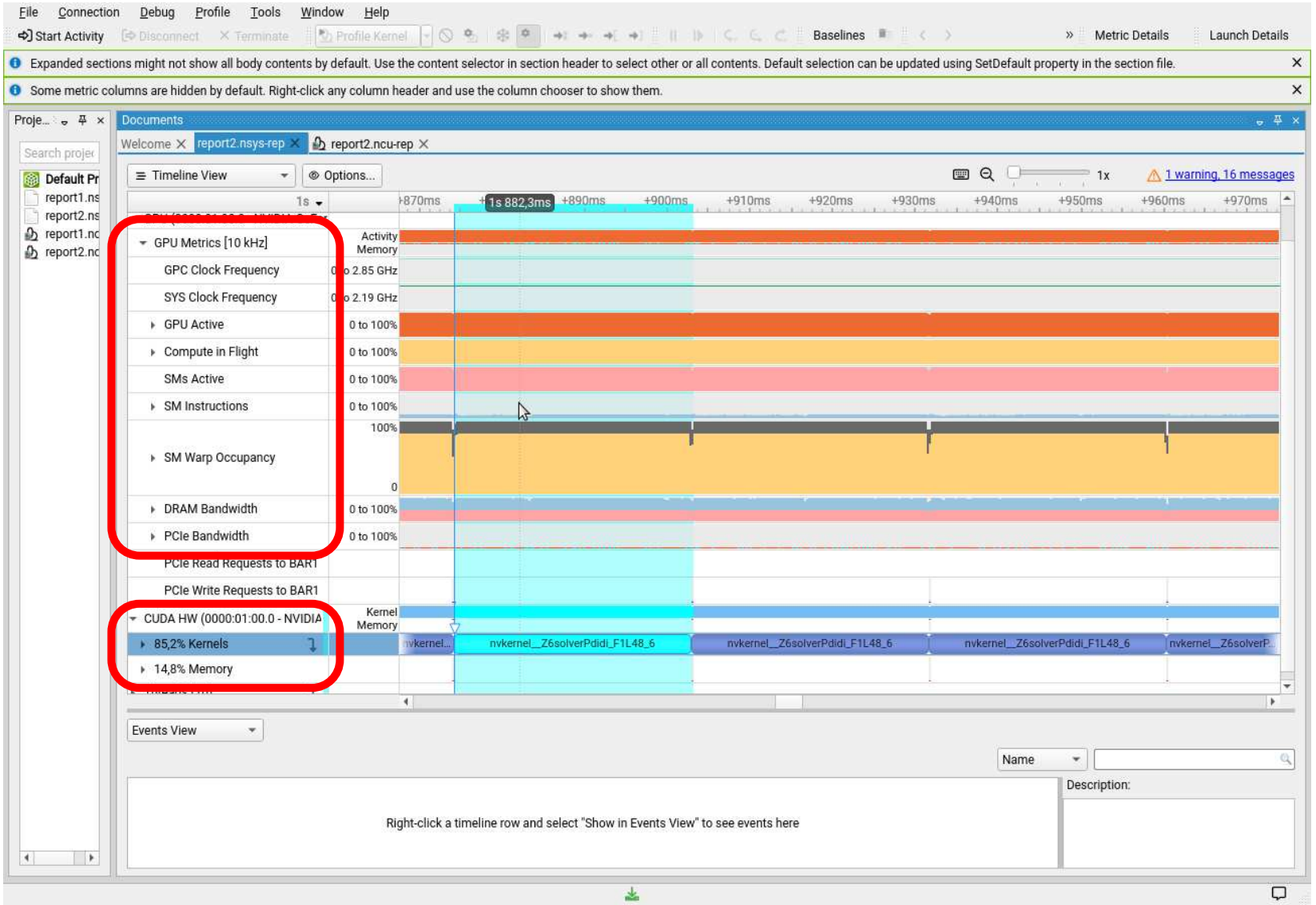
System Level Profiling

- Provides an overview on the utilization of the GPU
- To start the profiling process:
  - press ’Start Activity’ in upper left corner of main window
  - a dialog window opens
  - in the ’Target Platform’ section, fill in fields
    - ’Application Executable’
    - ’Working Directory’
    - ’Command Line Arguments’
  - in the ’Activity’ section, select ’System Trace’ on the left side
  - in the ’System’ tab, tick ’Collect GPU Metrics’.
  - click the ’Launch’ button in the lower right corner

System Level Profiling ...



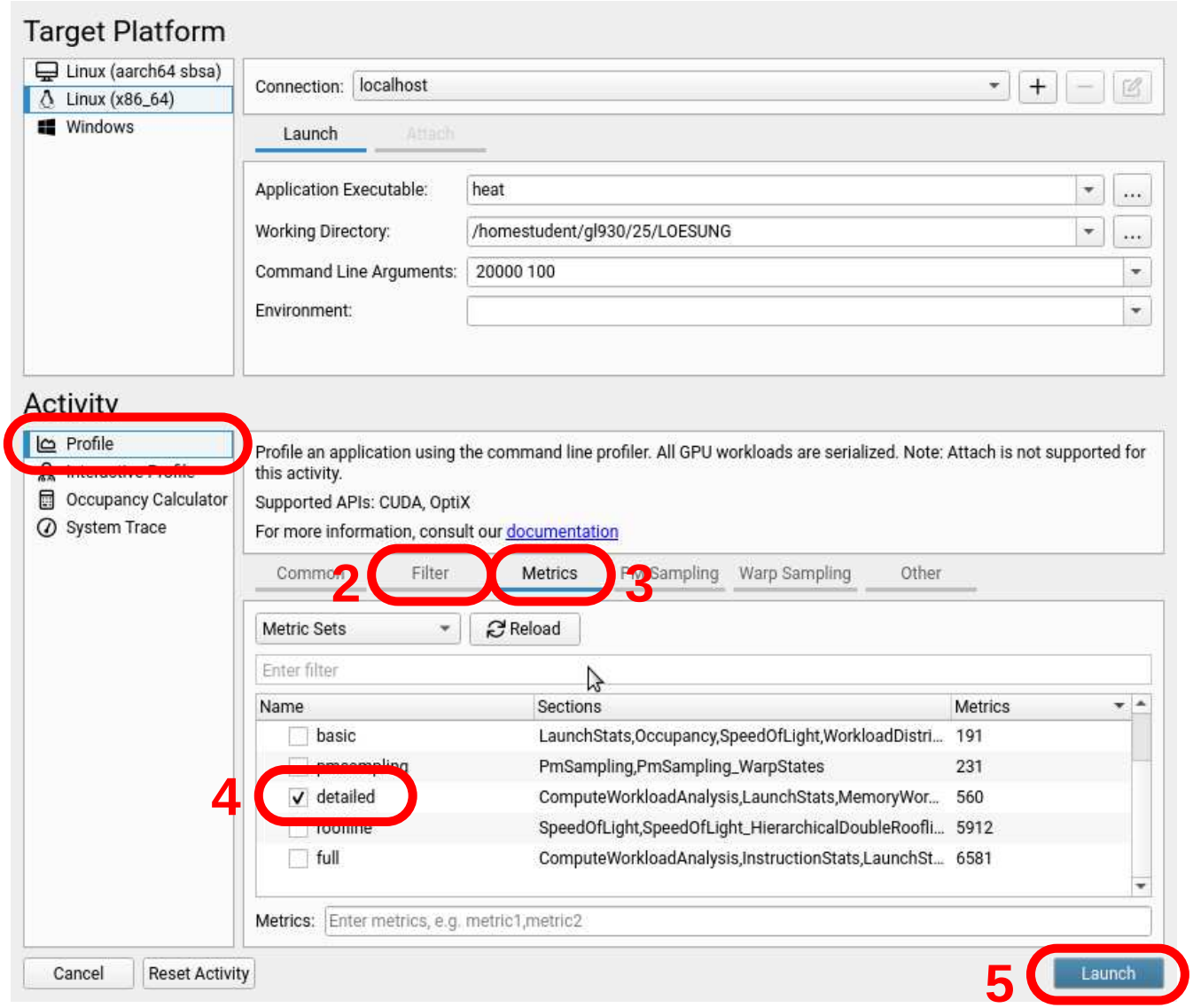
System Level Profiling ...



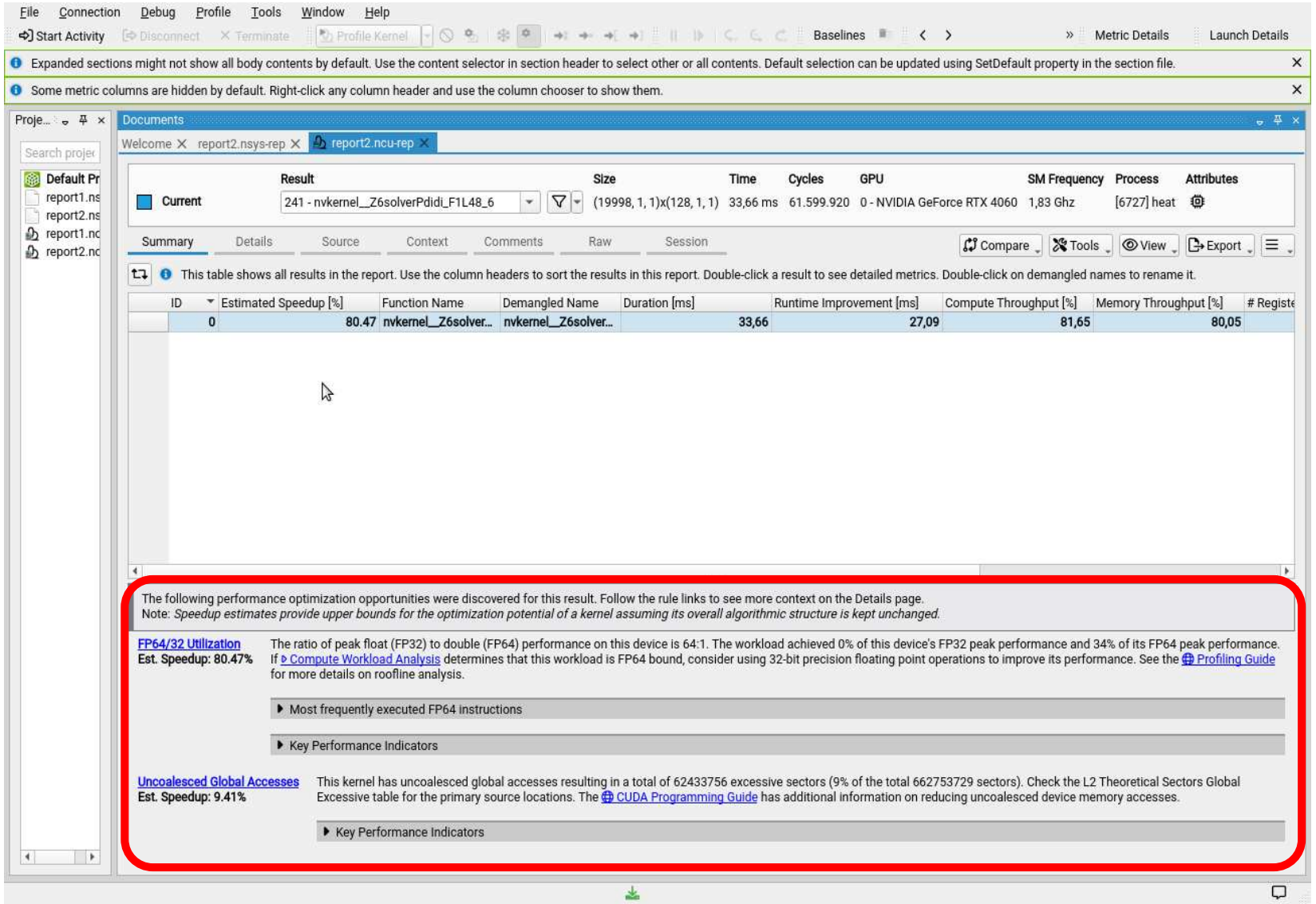
Kernel Profiling

- Provides more insight into the execution of a kernel executing on the GPU
- To start a kernel profile:
  - in the timeline view, right-click on the box representing the kernel's execution
  - in the popup menu, select 'Profile Kernel'
  - you will see the 'Start Activity' window, where 'Profile' is already selected
  - if necessary: fix the kernel's name in the 'Kernel Name' field of the 'Filter' tab
  - select the 'Metrics' tab and tick 'detailed'
  - click the 'Launch' button in the lower right corner

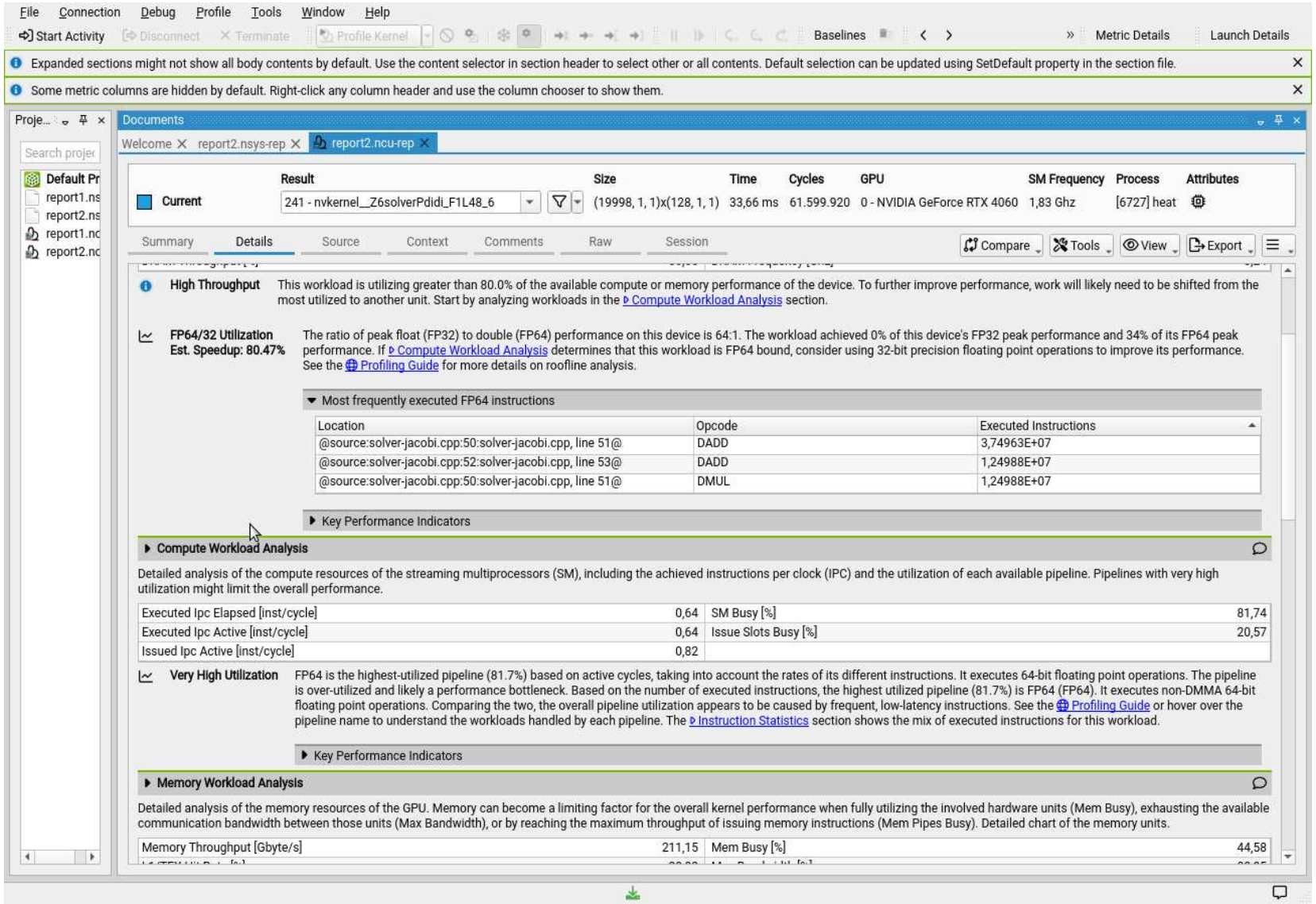
Kernel Profiling ...



Kernel Profiling ...



Kernel Profiling ...



More Information

- NVIDIA Nsight Compute home page
- NVIDIA Nsight Compute documentation

Parallel Processing - Tutorials

4 Working with MPI

Available MPI implementations

- E.g., MPICH, OpenMPI
  - portable implementations of the MPI-2 standard
- To use MPICH in the lab H-A 4111:
  - `module load pv`

Compiling MPI programs: `mpic++`

- `mpic++ -o myProg myProg.cpp`
- Not a separate compiler for MPI, but just a script that defines additional compiler options:
  - include und linker paths, MPI libraries, ...
  - option `-show` shows the invocations of the compiler

Running MPI programs

- Standardized start command:
  - `mpiexec -n 3 myProg args`
  - starts `myProg args` with 3 processes
  - `myProg` must be on the command search path or must be specified with (absolute or relative) path name
- On which nodes do the processes start?
  - depends on the implementation and the platform
  - in MPICH: specification is possible via a configuration file:
    - `mpiexec -n 3 -machinefile machines myProg args`
    - configuration file contains a list of node names, e.g.:

```
bslab01      ← start one process on bslab03
bslab05:2    ← start two processes on bslab05
```

Running MPI programs ...

- Executing on specific cores:
  - `mpiexec -n 3 ...  taskset -c 0-7 myProg args`
- Remarks on `mpiexec`:
  - on the remote nodes, the MPI programs start in the same directory, in which `mpiexec` was invoked
  - `mpiexec` uses `ssh` to start an auxiliary process on each node that is listed in the configuration fiile
    - even when less MPI processes should be started
  - avoid an entry `localhost` in the configuration file
    - it results in problems with `ssh`

Debugging

- MPICH and OpenMPI support `gdb` and `totalview`
- Using `gdb`:
  - `mpiexec -enable-x -n ... xterm -e gdb myProg`
    - instead of `xterm`, you may (have to) use other console programs, e.g., `konsole` or `gnome-terminal`
  - for each process, a `gdb` starts in its own console window
  - in `gdb`, start the process with `run` *`args...`*
- Prerequisite: compilation with debugging information
  - `mpic++ -g -o myProg myProg.cpp`

Performance Analysis using Scalasca

- In principle, in the same way as for OpenMP
- Compiling the program:
  - `scalasca -instrument mpic++ -o myprog myprog.cpp`
- Running the programms:
  - `scalasca -analyze mpiexec -n 4 ...  ./myprog`
  - creates a directory `scorep_myprog_4_sum`
    - `4` indicates the number of processes
    - directory must not previously exist; delete it, if necessary
- Interactive analysis of the recorded data:
  - `scalasca -examine scorep_myprog_4_sum`