
Parallel Processing - Tutorials

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

Stand: December 9, 2025

Contents

1 Computing Environment (H-A 4111)	2
2 Tools for OpenMP	9
2.1 Compiling	10
2.2 Executing	11
2.3 Debugging	12
2.4 Performance Analysis	20
3 GPU Programming with OpenMP	24
3.1 Compilation	25
3.2 NVidia Nsight Compute	26

1-2

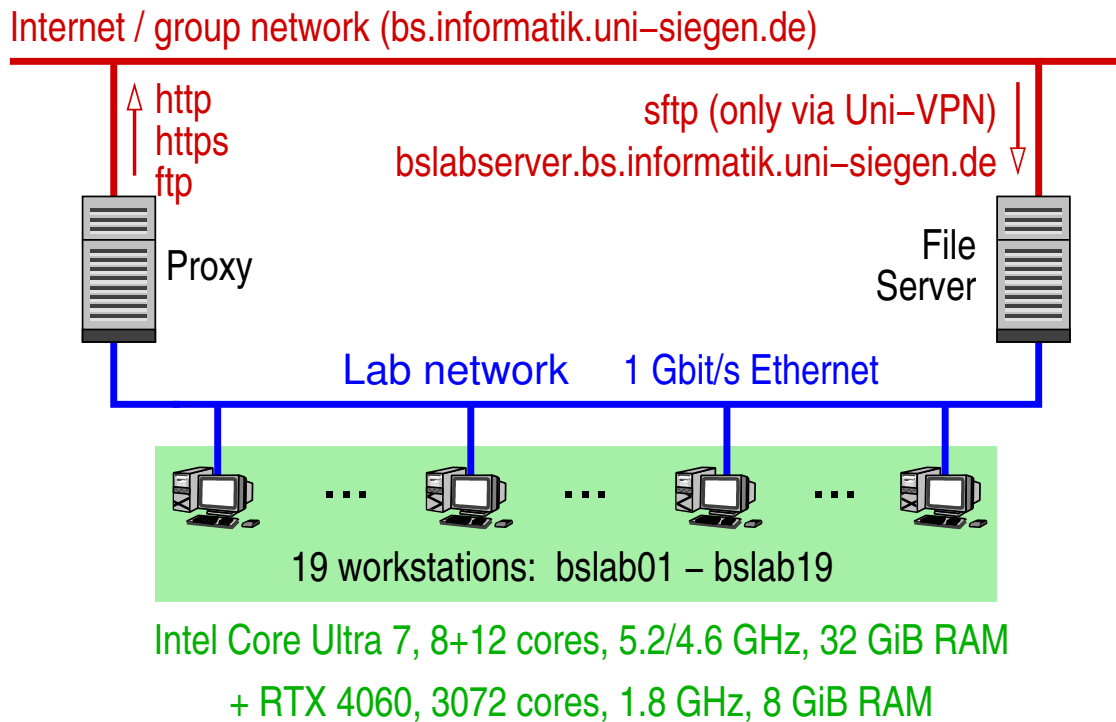


Parallel Processing - Tutorials

1 Computing Environment (H-A 4111)



Overview of the Lab



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 bandwidth: 89 GB/s

Notes for slide 7:

Concerning the peak performance of a performance core:

- ➔ AVX-2 allows registers to hold four 64-bit FP numbers.
- ➔ The core can issue up to two fused-multiply-add instructions per cycle.
- ➔ In theory, this allows up to $5.2 * 4 * 2 * 2 = 83.2$ GFlop/s.

Concerning memory throughput:

- ➔ The Intel Core Ultra 7 265 supports two memory channels (64 bit wide).
- ➔ The memory in our PCs is DDR5 with 5600 MT/s.
- ➔ Thus, max. bandwidth is $5.6 * 2 * 64 / 8 = 89.6$ 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 ...`

2.3 Debugging ...



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`

2.3 Debugging ...



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`

2.3 Debugging ...



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             <size_x> <size_y>: size...
292             <all>: 0 = compute one ...
293             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>
```

2.3 Debugging ...



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.
0x000000000401711 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>

2.4 Performance Analysis ...



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 ./barrier`
 - ➔ 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`

Notes for slide 21:

If you want to use Scalasca at home, you can download an appliance for Oracle VirtualBox, which includes Linux, g++ compilers, OpenMP, MPI, Scalasca and Visual Studio Code with g++ plugins.

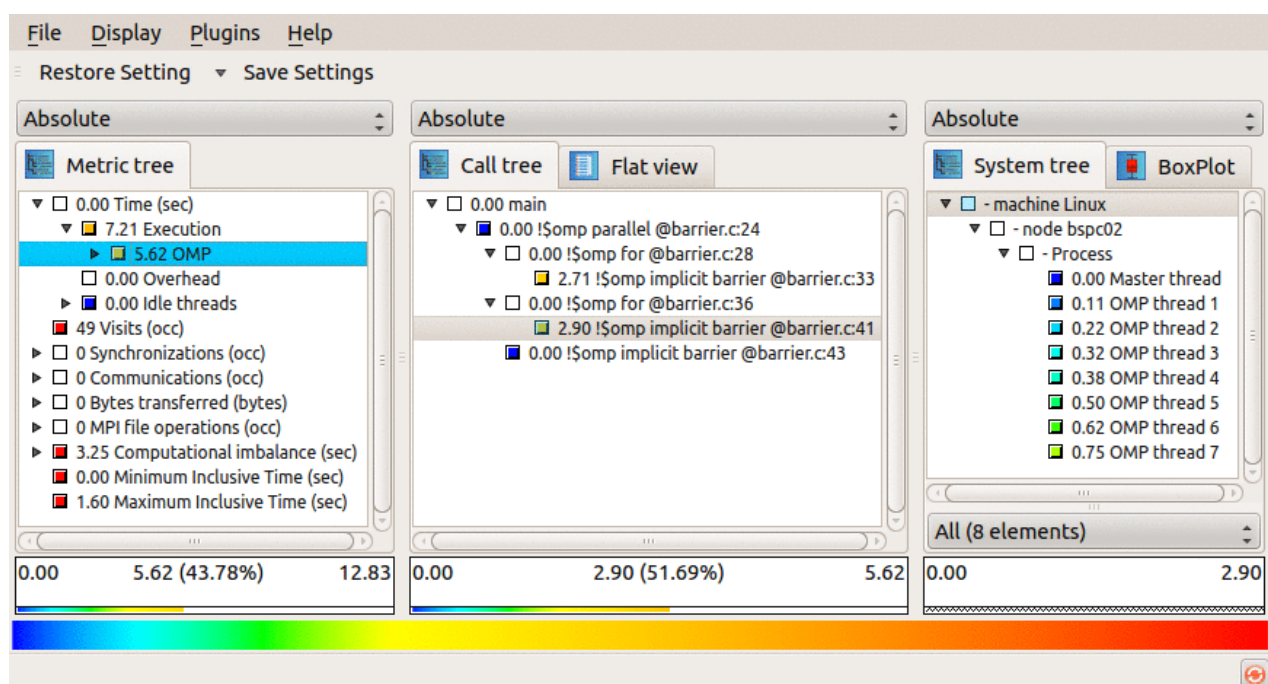
See <https://moodle.uni-siegen.de/mod/url/view.php?id=884597>.

21-1

2.4 Performance Analysis ...

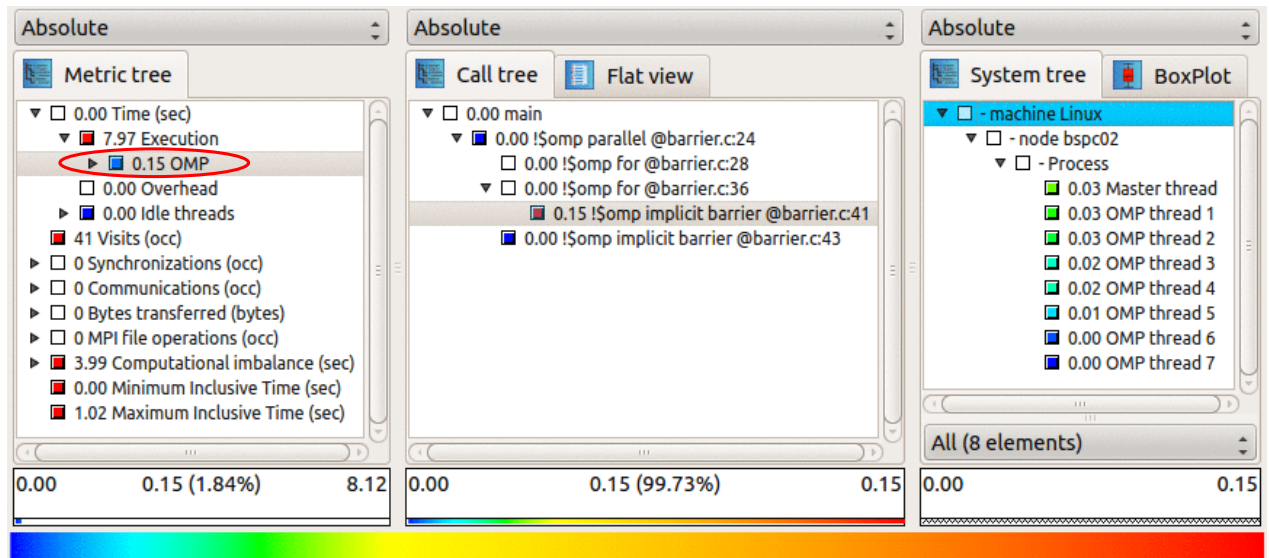


Scalasca: Example from Sect. 3.3.5



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`:



Notes for slide 23:

When interpreting the times indicated by Scalasca, the following must be observed:

- ➔ The metric displayed for an entry (here: time) always excludes the visible sub-entries. When, e.g., the item “7.97 Execution” in the *Metric tree* shown in the screen dump is folded (i.e., no longer visible), Scalasca displays “8.12 Execution” (0.15s execution time for OMP + 7.97s for the remaining execution).
In the example, you can see that the `nowait` option has made the time for OpenMP (synchronization) significantly smaller (0.15s instead of 5.62s), but the pure execution time has slightly increased (from 7.21s to 7.97s), possibly because of competition for the memory.
- ➔ The time that Scalasca displays is the **summed execution time of all threads**, including waiting times. In the example, the program actually terminated after 1.3s.
- ➔ Scalasca still shows a load imbalance (*Computational imbalance*), since, e.g., thread 7 still calculates much more in the first loop than thread 1. Scalasca is not able to recognize that this imbalance exactly cancels the corresponding imbalance in the second OMP loop.

Parallel Processing - Tutorials

3 GPU Programming with OpenMP

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

Notes for slide 25:

Background on the `regcount` option:

- ➔ In the Ada architecture (RTX 4060), each SM has 65536 registers, which are shared by all threads on one SM.
- ➔ This means that the number of registers used by a kernel limits the number of warps on an SM. E.g., if the kernel uses 64 registers, there cannot be more than 32 warps per SM (a warp has 32 threads).
- ➔ More warps on an SM increase the *occupancy*. A higher occupancy is beneficial, because when the threads in the current warp are waiting for, e.g., memory, the scheduler can choose threads from another warp. This allows better latency hiding and can increase the performance.
- ➔ On the other hand, a smaller register number may require the compiler to spill registers to memory, which decreases performance.
- ➔ Thus, an optimal number of registers must be found experimentally.

The NVidia Nsight Compute tool provides some help in finding the optimal number of registers (Section 'Occupancy' in the 'Details' tab of the Kernel profile).

25-1

3 GPU Programming with OpenMP ...



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`



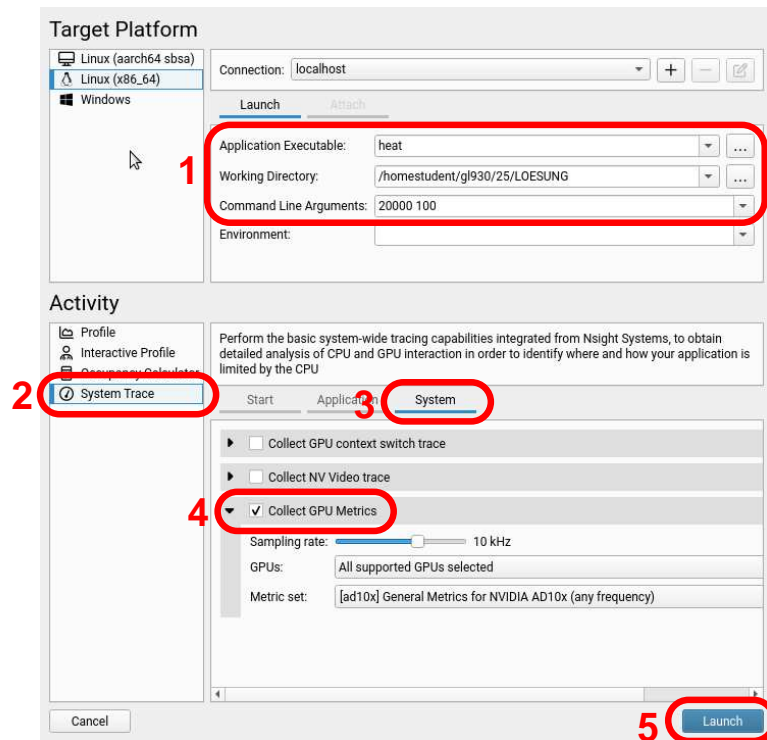
System Level Profiling

- ➔ 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

3.2 NVidia Nsight Compute ...



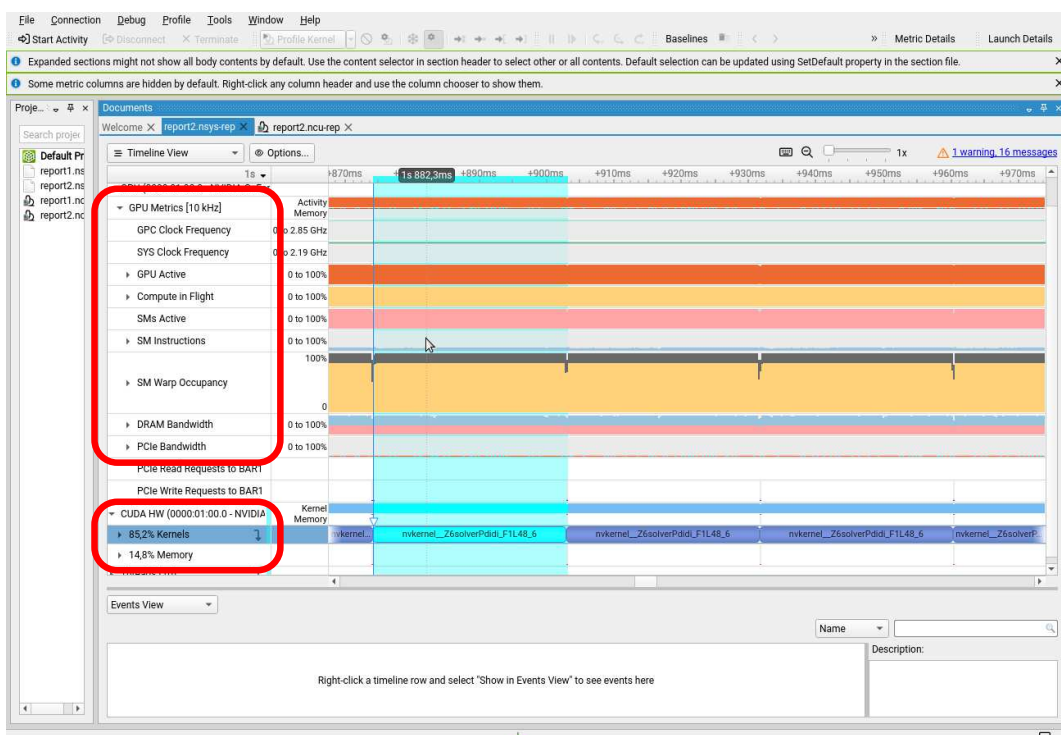
System Level Profiling ...



3.2 NVidia Nsight Compute ...



System Level Profiling ...



Notes for slide 30:

In the timeline diagram, among others, you can examine various GPU metrics (e.g. instruction throughput and used DRAM bandwidth). If you open the 'CUDA HW' timeline, you can see when kernels are executing or memory transfers between host and target take place.

30-1

3.2 NVidia Nsight Compute ...



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

Notes for slide 31:

As the current version of Nsight Compute seems to have a bug, you may have to fix the kernel's name, before you can do profiling for that kernel. The kernel name is a function name for the code contained in an OpenMP `target` section that is automatically generated by the `nvc++` compiler.

In the field 'Kernel Name' in the 'Filter' tab, you may see something like

```
nvkernel__Z6solverPddidi_F1L48__3...
```

If the name ends with some dots, it is not the correct name. The correct kernel name is shown in the timeline diagram, when you zoom in:



Typically, you will have to replace the trailing `'__3...'` with `'_6'` or some other number.

31-1

3.2 NVidia Nsight Compute ...



Kernel Profiling ...

The screenshot shows the Nsight Compute interface. The 'Target Platform' section is at the top, showing 'Linux (aarch64 sbsa)' and 'Linux (x86_64)'. The 'Activity' section is below, showing 'Profile' selected. The 'Metrics' tab is active, showing a table of metric sets. The 'Launch' button is at the bottom right.

Annotations:

- 1: Profile button in the Activity section.
- 2: Filter tab in the Metrics section.
- 3: Metrics tab in the Metrics section.
- 4: Detailed metric set checkbox in the Metrics section.
- 5: Launch button in the Metrics section.

Name	Sections	Metrics
<input type="checkbox"/> basic	LaunchStats,Occupancy,SpeedOfLight,WorkloadDistri...	191
<input type="checkbox"/> pmsampling	PmSampling,PmSampling_WarpStates	231
<input checked="" type="checkbox"/> detailed	ComputeWorkloadAnalysis,LaunchStats,MemoryWor...	560
<input type="checkbox"/> rooime	SpeedOfLight,SpeedOfLight_HierarchicalDoubleRoofi...	5912
<input type="checkbox"/> full	ComputeWorkloadAnalysis,InstructionStats,LaunchSt...	6581

3.2 NVidia Nsight Compute ...



Kernel Profiling ...

The screenshot shows the NVidia Nsight Compute interface. The main window displays a table of kernel results. The 'Current' kernel is selected, showing a result of 241, a size of (19998, 1, 1)x(128, 1, 1), a time of 33,66 ms, and a GPU of 0 - NVIDIA GeForce RTX 4060. Below the table, a section titled 'Performance Optimization Opportunities' is highlighted with a red box. This section contains two main items:

- FP64/32 Utilization**: The ratio of peak float (FP32) to double (FP64) performance on this device is 64.1. The workload achieved 0% of this device's FP32 peak performance and 34% of its FP64 peak performance. Est. Speedup: 80.47%. This workload is FP64 bound, consider using 32-bit precision floating point operations to improve its performance.
- Uncoalesced Global Accesses**: This kernel has uncoalesced global accesses resulting in a total of 62433756 excessive sectors (9% of the total 662753729 sectors). Check the L2 Theoretical Sectors Global Excessive table for the primary source locations. Est. Speedup: 9.41%.

3.2 NVidia Nsight Compute ...



Kernel Profiling ...

The screenshot shows the NVidia Nsight Compute interface with detailed profiling results. The 'Current' kernel is selected, showing a result of 241, a size of (19998, 1, 1)x(128, 1, 1), a time of 33,66 ms, and a GPU of 0 - NVIDIA GeForce RTX 4060. The 'Details' tab is active, showing a table of performance metrics:

Metric	Value	SM Busy [%]
Executed Ipc Elapsed [inst/cycle]	0,64	81,74
Executed Ipc Active [inst/cycle]	0,64	20,57
Issued Ipc Active [inst/cycle]	0,82	

Below the table, the 'Compute Workload Analysis' section is expanded, showing a table of frequently executed FP64 instructions:

Location	Opcode	Executed Instructions
@source:solver-jacobi.cpp:50:solver-jacobi.cpp, line 51@	DADD	3,74963E+07
@source:solver-jacobi.cpp:52:solver-jacobi.cpp, line 53@	DADD	1,24988E+07
@source:solver-jacobi.cpp:50:solver-jacobi.cpp, line 51@	DMUL	1,24988E+07

Notes for slide 34:

Once the measurement has finished, the tool shows a detailed analysis of the profiled kernel. The 'Summary' tab shows the most important performance issues, while the 'Details' tab provides detailed metrics for, e.g., computational throughput, memory usage, and occupancy (number of warps per SM).

34-1

3.2 NVidia Nsight Compute ...



More Information

- ➔ [NVIDIA Nsight Compute home page](#)
- ➔ [NVIDIA Nsight Compute documentation](#)

Parallel Processing - Tutorials

4 Working with MPI

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
 - in MPICH, `mpiexec` uses `ssh` to start processes on remote nodes
 - avoid an entry `localhost` in the configuration file, as 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 programs:
 - `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`