CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich

Executing and debugging Multi-GPU Codes on Alps

CSCS Webinar 2025

Sebastian Keller, Fabian Bösch, Jean-Guillaume Piccinali and Daniel Ganellari

Table of contents

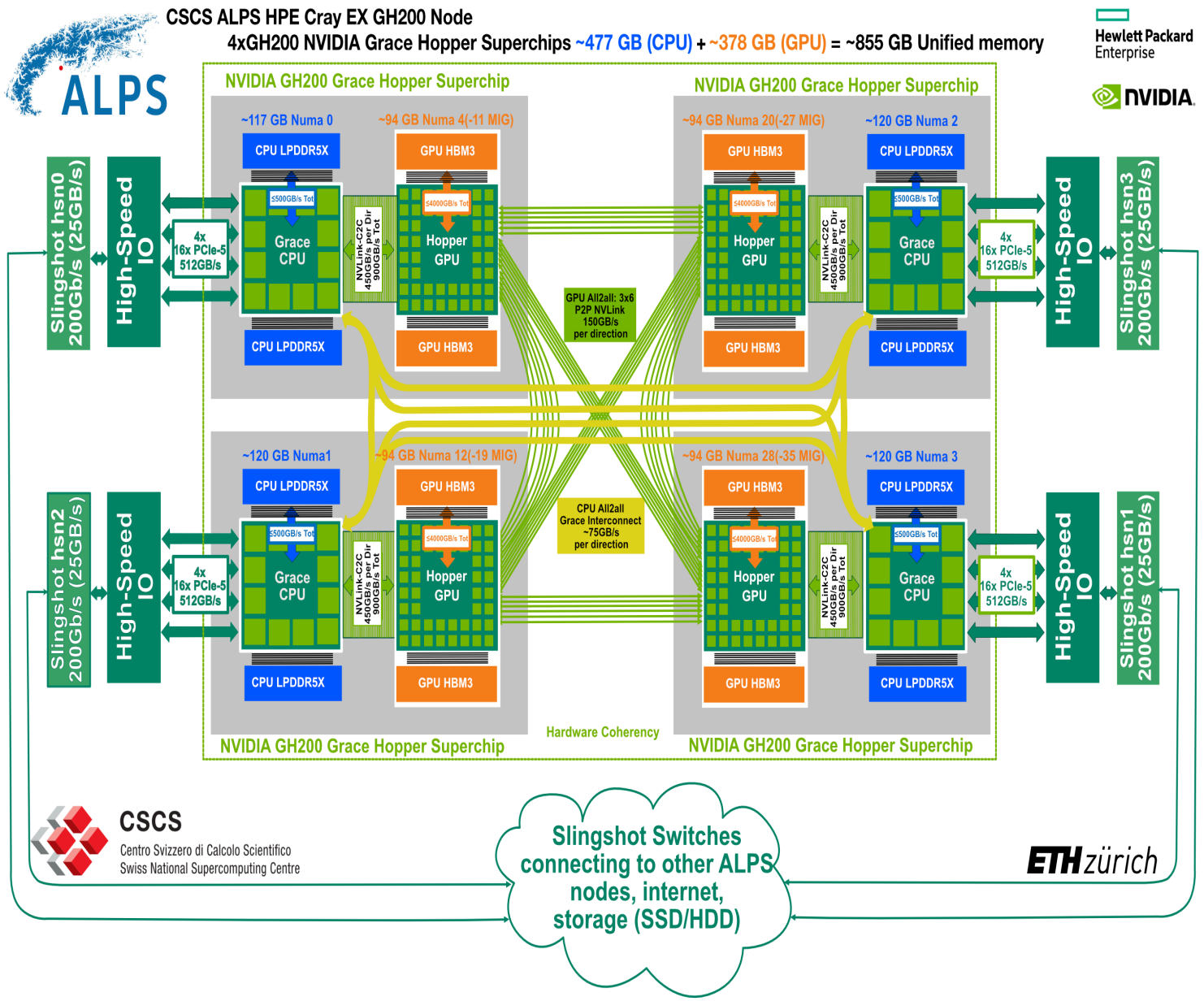
CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich

Hardware overview

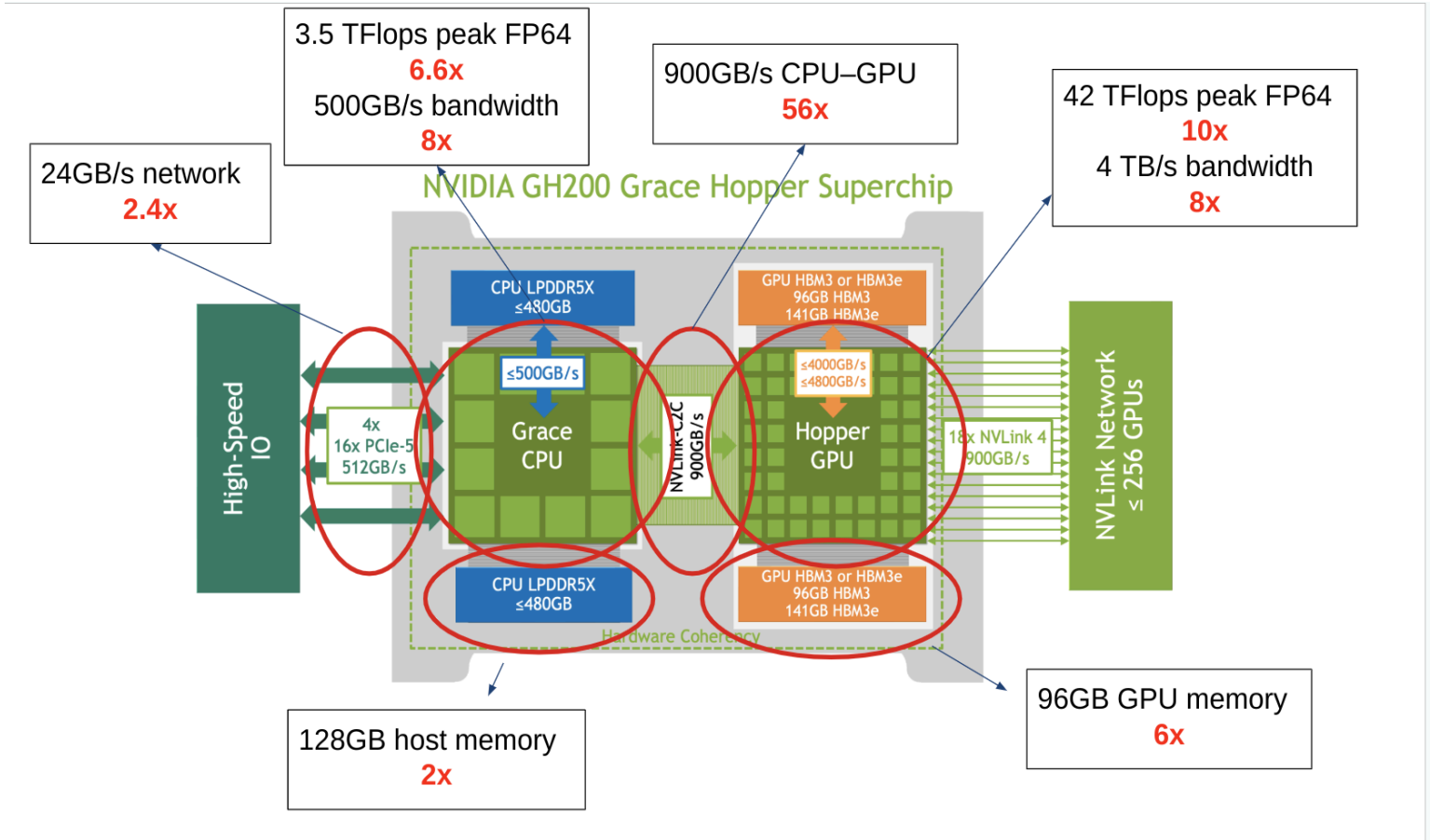
Grace-Hopper GH200

ALPS Grace-Hopper node



- 4 CPU-GPU modules
- Fast NVLink connections with 150 GB/s between the modules
- 4 network cards with Slingshot-11 technology providing 25 GB/s per card
- Total node-to-network bandwidth of 100 GB/s

Grace-Hopper (GH200) module



- 72 ARM Grace CPU cores and one H100 GPU
- 128GB host memory and 96GB GPU memory
- Fast NVLink chip-to-chip (C2C) between CPU and GPU with 900GB/s

Reference: A Case Study with GH200 Superchip (arXiv:2408.11556v1)

Four modules make a node

- All four modules form a single machine with 4x72 = 288 CPU cores
- All memory visible and allocatable by a single process:

```
daint-nid:~$ free -m
              total        used        free      shared  buff/cache   available
Mem:         874968       70156      841573        5484       18773      804811
```

- CPU and GPU memory is unified into a single address space
- **Both** CPU and GPU memory are allocatable and accessible from the CPU
- Each node has 4x128GB + 4x96GB = 896GB memory
- CPU and GPU memory from different modules are organized into different **NUMA** (non-uniform memory access) domains

Moving from Piz Daint to GH200

Machine	New feature	Code requirements
Piz Daint XC50	GPUs	manage GPU memory write GPU kernels
ALPS GH200	CPU-GPU C2C and unified address space	*None*

- **Piz Daint XC50 added GPUs as a new feature**
  - existing software needed to adapt
  - added code complexity
- **ALPS GH200 adds a fast CPU-GPU connection and a unified address space**
  - no changes to existing software needed
  - reduced complexity, memory accessible on CPU and GPU

NUMA domains of GH200

NUMA

What is Non-Uniform Memory Access (NUMA)?

- NUMA provides information about physical distances between domains and associated hardware resources

Example:

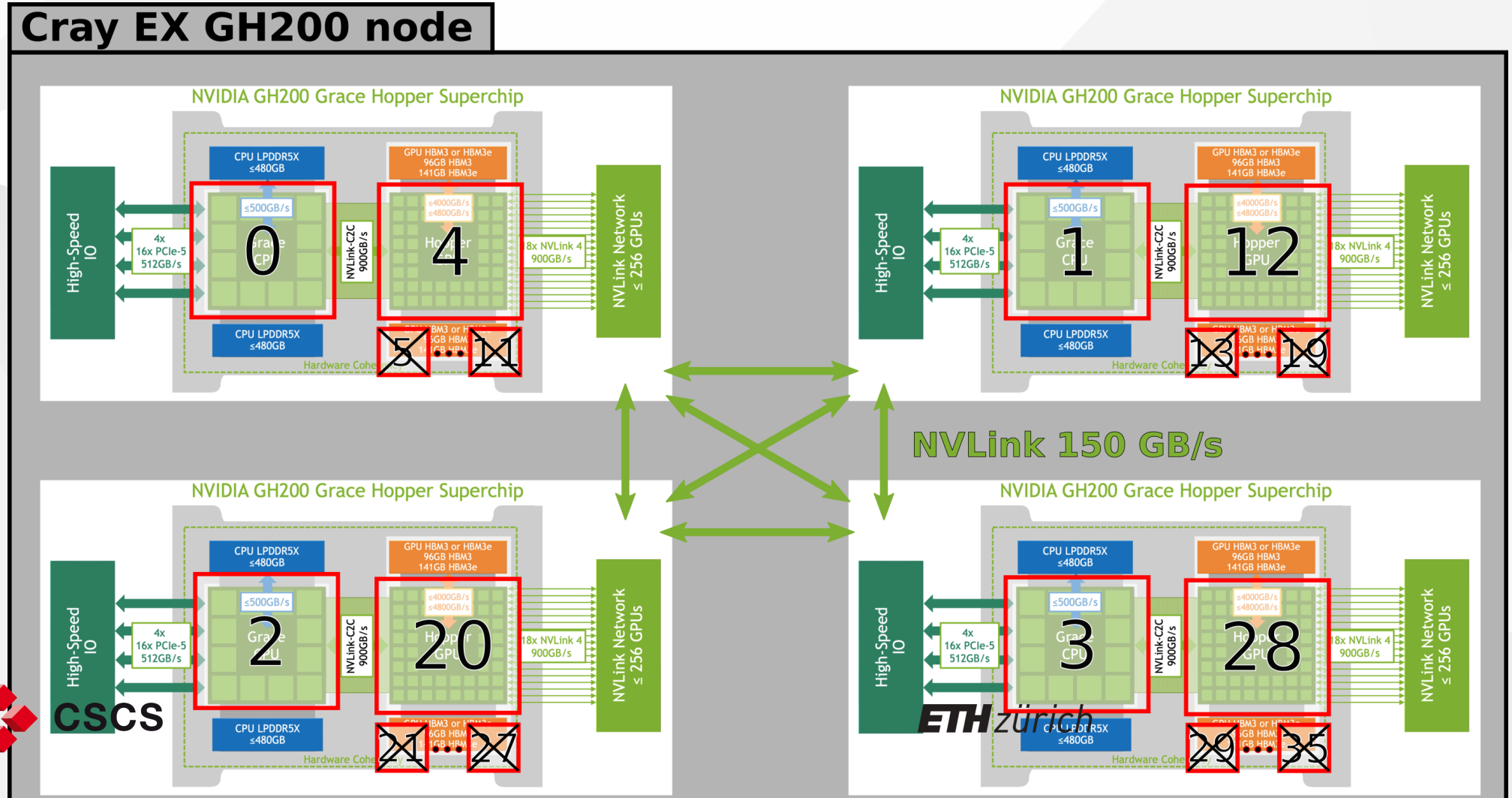
- Systems with **two CPU sockets** such as Piz Daint MC:
  - **two NUMA nodes:** memory associated with either of the sockets
  - accessing memory of the other socket is slower

Impact on developing high-performance multithreaded code:

- Either write NUMA-aware code (observing first touch-policy)
- or launch at least 1 MPI rank for each NUMA node

NUMA nodes of GH200

- 1 node per CPU
- 1 node per GPU (+7 empty, non-used nodes for multi-instance GPU, MiG)



NUMA nodes of GH200

```
daint-nid:~$ numactl -H
available: 36 nodes (0-35)
# CPU modules (0-3), each with 72 cores and 128GB RAM
node 0 cpus: 0-71
node 1 cpus: 72-143
node 2 cpus: 144-215
node 3 cpus: 216-287
# GPU modules (4,12,20,28), each with 96GB RAM
node 4 cpus:                    # GPU of module 0
node 4 size: 96768 MB
node 5 cpus:                    #empty node module 0
node 5 size: 0 MB
...
```

- Only 8 of 36 NUMA domains are used: 4 CPUs (0-3) and 4 GPUs (4,12,20,28)
- Remaining domains are reserved for Multi-Instance GPU (MIG) technology

NUMA nodes of GH200

`numactl -H` also shows connectivity information between domains.

```
daint-nid:~$ numactl -H
available: 36 nodes (0-35)
[...]
node   0    1    2    3    4   12   20   28
  0:  10   40   40   40   80  120  120  120
  1:  40   10   40   40  120   80  120  120
  2:  40   40   10   40  120  120   80  120
  3:  40   40   40   10  120  120   20   80
  4:  80  120  120  120   10  255  255  255
 12: 120   80  120  120  255   10  255  255
 20: 120  120   80  120  255  255   10  255
 28: 120  120  120   80  255  255  255   10
```

NUMA Distance Values: 40: CPU→CPU (different modules) • **80**: CPU→GPU (same module) • **120**: CPU→GPU (different modules) • **255**: GPU→GPU (different modules)

Wait... NUMA nodes on GPUs?

Yes, that's right! GH200 is the first hardware at CSCS with shared memory addresses between CPUs and GPUs.

- Memory allocated with host APIs like `malloc`, `new` and `mmap` can be **accessed by all CPUs and GPUs** in the compute node
- Physical memory placement decided by **first touch principle**
- **Automatic page migration**: repeated accesses to memory on a different NUMA node can cause memory to be migrated to the closest NUMA node
    - The feature is **disabled on Daint and Clariden** to address performance issues affecting NCCL-based workloads (e.g. LLM training)
- Memory allocated with `cudaMalloc` still **cannot** be accessed from the CPU and by other GPUs on the compute node

Passing system memory to GPU kernels

Memory allocated through host APIs can be passed to GPU kernels:

```
__global__ void add(const int* x, const int* y, int* z, size_t n)
{
    size_t tid = blockDim.x * blockIdx.x + threadIdx.x;
    if (tid < n) { z[tid] = x[tid] + y[tid]; }
}

size_t n = 1024;
std::vector<int> a(n), b(n), c(n); // uses `new` to allocate memory

// OK, pass system memory to
add<<<n/128, 128>>>(a.data(), b.data(), c.data(), n);
```

⚠️ The constructor of `std::vector` initializes elements with a single CPU thread, memory in this example will be placed in host-LPDDR5

Controlling NUMA memory placement

Placement of a memory page is decided by the NUMA node of the thread that first ***writes*** to it, not by the thread that allocates it.

```
__global__ void add(const int* x, const int* y, int* z, size_t n)
{
    size_t tid = blockDim.x * blockIdx.x + threadIdx.x;
    if (tid < n) { z[tid] = x[tid] + y[tid]; }
}

size_t n = 1024;
int* a = new int[n];
int* b = new int[n];
int* c = new int[n];

// or, see [1]: std::vector<int, DefaultInitAdaptor<int>> a(n), b(n), c(n);

cudaMemset(a, 0, n * sizeof(int)); // first touch on GPU, memory will be placed in H100-HBM3
cudaMemset(b, 0, n * sizeof(int));
cudaMemset(c, 0, n * sizeof(int));

add<<<n/128, 128>>>(a.data(), b.data(), c.data(), n);
```

[1]: https://github.com/sekelle/cornerstone-octree/blob/master/include/cstone/util/noinit_alloc.hpp

Summary: NUMA on GH200

- GPU memory allocated with `cudaMalloc` behaves the same as on previous machines;
  the **CPU and other GPUs can't access it.**
- System memory (`malloc`, `new`, ...) now **also accessible by GPUs**
- NUMA memory placement decided by the **first touch policy**, which can be tricky to get right
- Designing code to run with 1 rank per module still a reasonable choice in most cases
  - can still take advantage of unified memory between CPU and GPU
  - no need to manage multiple GPUs per process...
  - ... or deal with NUMA effects

CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich

SLURM best practices

Moving to GH200: SLURM and affinity

- Affinity indicates which CPU cores the threads of a process/rank may run on
- CPU cores and GPUs are both assigned sequentially by SLURM to the 4 modules:

GH200 node	
Module 0: GPU `0`, CPUs `0–71`	**Module 1**: GPU `1`, CPUs `72–143`
Module 2: GPU `2`, CPUs `144–215`	**Module 3**: GPU `3`, CPUs `216–287`

SLURM affinity is configured with sensible defaults: each MPI rank within a node will be placed on a different module.

4 MPI ranks per node with up to 72 cores

This case automatically ends up with ideal affinity placement:

- No NUMA effects
- All threads of a rank bound to the same module
- Each rank gets a different GPU by default
- Use up to 72 threads per rank. Many applications will be faster with fewer.
- Sensible default programming model for new code on ALPS-GH200

```
#SBATCH --nodes=<numNodes>
#SBATCH --ntasks-per-node=4
#SBATCH --cpus-per-task=72
#SBATCH --gpus-per-task=1          # Otherwise make sure to export CUDA_VISIBLE_DEVICES=$SLURM_LOCAL_ID

export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK

#export MPICH_GPU_SUPPORT_ENABLED=1   # Set as default on Daint Alps. Breaking change for non GPU-aware MPI code!

srun ./the-app
```

4 MPI ranks per node (interactively)

affinity.sh

```
#!/bin/bash
export LOCAL_RANK=$SLURM_LOCALID
export GLOBAL_RANK=$SLURM_PROCID
export GPUS=(0 1 2 3)
export NUMA_NODE=$(echo "$LOCAL_RANK % 4" | bc)
export CUDA_VISIBLE_DEVICES=${GPUS[$NUMA_NODE]}

echo Rank: $GLOBAL_RANK/$LOCAL_RANK sees GPU: $CUDA_VISIBLE_DEVICES on numa node: $NUMA_NODE

numactl --cpunodebind=$NUMA_NODE --membind=$NUMA_NODE "$@"
```

```
srun -N1 -n4 -c72 ./affinity.sh ./the-app
```

Multiple ranks per GPU with MPS

- On Old Piz Daint, GPUs could be over-subscribed with multiple ranks by setting the `CRAY_CUDA_MPS` environment variable.
- On GH200 the *CUDA Multi-process Service (MPS)* needs to be called manually instead:

```
#SBATCH --nodes=2
#SBATCH --ntasks-per-node=32 # 32 MPI ranks per node
#SBATCH --cpus-per-task=8
#SBATCH --account=<account>

export OMP_NUM_THREADS=8

# Assigns closest GPUs to ranks according to their CPU affinity.
srun ./mps-wrapper.sh <code> <args>
```

The `mps-wrapper.sh` script is available on our knowledge base at

https://confluence.cscs.ch/display/KB -> User Guides -> Running jobs

Summary: SLURM and affinity

Node configuration	for optimal performance, remember to
4 ranks x 72 cores `--ntasks-per-node=4 -c72`	• use up to 72 threads • check if fewer threads are faster
>4 ranks	• use the mps-wrapper script
1 rank	• write NUMA-aware code • manage multiple GPUs in code

Refer to the CSCS knowledge base at https://confluence.cscs.ch/display/KB
`-> Scientific Computing/Scientific Applications` for

- example batch submission scripts, wrapper scripts

CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich

Benchmarks

Benchmarks, and why to run them

As a developer, running benchmarks allows you to validate your software environment, i.e.

- Compiler flags and CMake settings
- Environment variables
- SLURM and affinity settings
- Software stack and dependencies
- How you call APIs like MPI, CUDA-runtime

Measuring compute flops

Matrix-Matrix multiplication on CPUs and GPUs: https://github.com/eth-cscs/node-burn

Multiplying 10000^2 matrices on 4 GPUs using `cublas` :

```
daint-ln001:~/build/node-burn$ srun -N1 -n4 -c72 ./burn -ggemm,10000
```

```
nid006672:gpu    214 iterations, 42622.84 GFlops,     10.0 seconds,    2.400 Gbytes
nid006672:gpu    203 iterations, 40582.83 GFlops,     10.0 seconds,    2.400 Gbytes
nid006672:gpu    201 iterations, 40118.36 GFlops,     10.0 seconds,    2.400 Gbytes
nid006672:gpu    206 iterations, 41009.39 GFlops,     10.0 seconds,    2.400 Gbytes
```

- ✅ peak GPU performance
- ✅ all ranks on a different GPU

Measuring compute flops: TF32 and tensor cores

`cublas` supports matrix-matrix multiplication in TF32, a reduced precision format.

To test this, switch to single precision (FP32/ `float`) and enable tensor cores in `cublas` :

```
using value_type = float;
```

```
cublasCreate(&cublas_handle);
cublasSetMathMode(cublas_handle, CUBLAS_TF32_TENSOR_OP_MATH);
```

```
CC=gcc CXX=g++ cmake .. -DDOUBLE=OFF -DTF32=ON
```

```
daint-ln001:~/build/node-burn$ srun -N1 -n4 -c72 ./burn -ggemm,10000
```

```
nid006649:gpu    1328 iterations, 265526.72 GFlops,     10.0 seconds,    3.221 Gbytes
nid006649:gpu    1261 iterations, 252183.61 GFlops,     10.0 seconds,    3.221 Gbytes
nid006649:gpu    1315 iterations, 262859.93 GFlops,     10.0 seconds,    3.221 Gbytes
nid006649:gpu    1323 iterations, 264556.08 GFlops,     10.0 seconds,    3.221 Gbytes
```

Measuring compute flops (CPU)

Multiplying 3000^2 matrices on CPUs with `OpenBlas`

```
OMP_NUM_THREADS=71 srun -N1 -n4 -c72 ./burn -cgemm,3000
```

```
nid006672:cpu    364 iterations,  1964.63 GFlops,     10.0 seconds,    0.216 Gbytes
nid006672:cpu    358 iterations,  1930.91 GFlops,     10.0 seconds,    0.216 Gbytes
nid006672:cpu    350 iterations,  1887.75 GFlops,     10.0 seconds,    0.216 Gbytes
nid006672:cpu    342 iterations,  1843.75 GFlops,     10.0 seconds,    0.216 Gbytes
```

✅ Decent performance on 4 CPU sockets

- Benchmarking revealed a quirk of `OpenBlas` : it uses one extra thread, hence `OMP_NUM_THREADS=71`

Example scenario:

- Imagine you'd like to evaluate the NVIDIA performance libraries (NVPL) for your application.
- Use this benchmark (`node-burn`), link it against NVPL and test the performance.

Measuring memory bandwidth

- How much bandwidth can we achieve with a simple OpenMP for-loop?
- How many threads are needed to achieve maximum bandwidth?

`node-burn` can run a STREAM benchmark:

```
OMP_NUM_THREADS=1 srun -N1 -n1 -c1 ./burn -cstream,500000000
```

```
nid006646:cpu     25 iterations,    14.78 GB/s,     10.2 seconds
```

```
OMP_NUM_THREADS=16 srun -N1 -n1 -c16 ./burn -cstream,500000000
```

```
nid006646:cpu    353 iterations,   211.62 GB/s,     10.0 seconds
```

```
OMP_NUM_THREADS=64 srun -N1 -n1 -c72 ./burn -cstream,500000000
```

```
nid006650:cpu    751 iterations,   450.53 GB/s,     10.0 seconds
```

✅ Performance close to hardware limit of 500 GB/s.

Network bandwidth between CPUs, same node

- MPI_Send/Recv bandwidth between 2 CPUs on different modules (in the same node):

```
daint-ln001:~/build/node-burn$ OMP_NUM_THREADS=64 srun -N1 --ntasks-per-node=2 -c72 osu_bw -m 134217728 H H
# OSU MPI Bandwidth Test v5.9
# Size      Bandwidth (MB/s)
...
16384               11185.31
32768               16997.28
65536               28699.76
131072              40667.73
262144              52561.34
524288              52273.78
1048576             25893.22  # Performance drop for message > 1MB, protocol switch (eager -> rendezvous)
2097152             24912.75
4194304             20132.55
8388608             17849.81
16777216            14977.46  # matching the single-threaded memory bandwidth (./burn -cstream)
```

❓ Why is the bandwidth dropping down to 15 GB/s?

💡 **Answer**: For large messages (>1MB), MPI internally uses a single-threaded shared-memory copy operation, matching the single-thread STREAM benchmark of 14.78 GB/s.

Network bandwidth between GPUs, same node

The provided `prgenv-gnu` software stacks on ALPS provides OSU benchmarks:

- MPI_Send/Recv bandwidth between 2 GPUs in a node (NVLink)

```
daint-ln001:~/build/node-burn$ OMP_NUM_THREADS=64 srun -N1 --ntasks-per-node=2 -c72 ~/affinity.sh osu_bw D D
# OSU MPI-CUDA Bandwidth Test v5.9
# Send Buffer on DEVICE (D) and Receive Buffer on DEVICE (D)
# Size      Bandwidth (MB/s)

...
16384                3995.67
32768                8042.68
65536               16022.34
131072              31954.64
262144              59236.19
524288              82370.28
1048576             99679.24
2097152            112164.52
4194304            120592.17
```

✅ Achieves 120 GB/s with a message size of 4MB (theoretical limit: 150 GB/s)

Network bandwidth between GPUs, different nodes

- MPI_Send/Recv bandwidth between 2 GPUs in different nodes:

```
daint-ln003:~/build/node-burn$ OMP_NUM_THREADS=64 srun -N2 --ntasks-per-node=1 -c72 ~/affinity.sh osu_bw D D
# OSU MPI-CUDA Bandwidth Test v5.9
# Send Buffer on DEVICE (D) and Receive Buffer on DEVICE (D)
# Size      Bandwidth (MB/s)
...
16384                20389.44
32768                19104.43
65536                22400.52
131072               23368.16
262144               23705.47
524288               23770.06
1048576              23981.94
2097152              24029.10
4194304              24052.61
```

✅ Achieves 24 GB/s with a message size of 4MB (theoretical limit: 25 GB/s)

CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

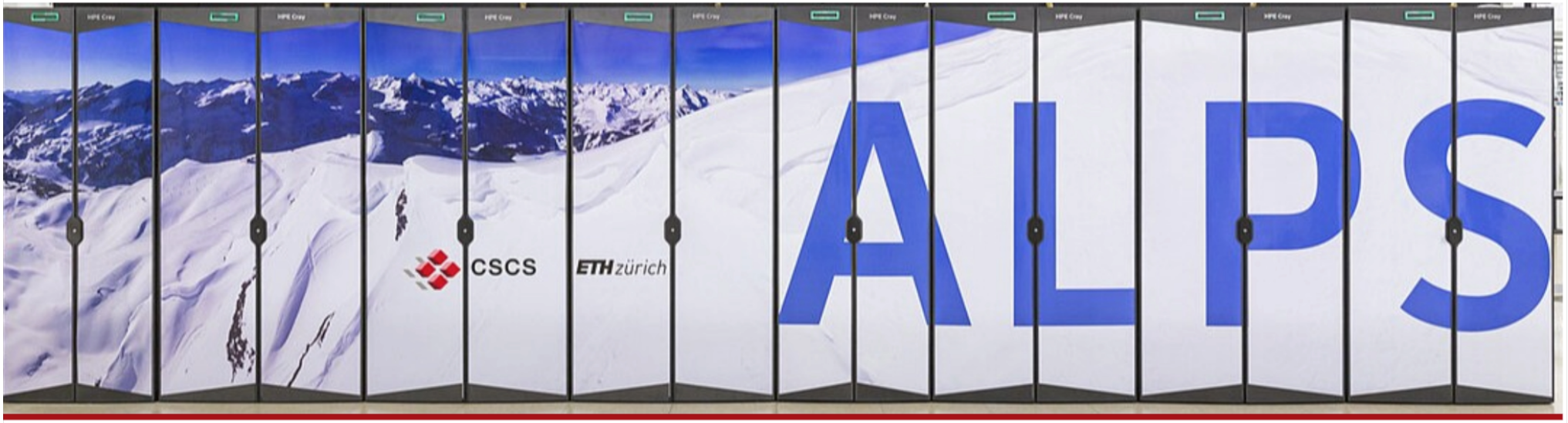
ETH zürich

Questions?

We hope this helps you get the most out of ALPS!

CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich



Executing and debugging Multi-GPU Codes on Alps

Swiss National Supercomputing Centre (CSCS)
May 15th, 2025

CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich

Getting Started with NVIDIA Nsight

NVIDIA Nsight: Setup your environment with the tools

Nsight is a toolkit for debugging and profiling applications on GPUs

Nsight Systems (nsys) is a high-level performance analysis tool, designed to identify bottlenecks

Nsight Compute (ncu) is a low-level performance analysis tool, designed for detailed analysis of CUDA kernels

- Nsight is available with any uenv that comes with a CUDA install

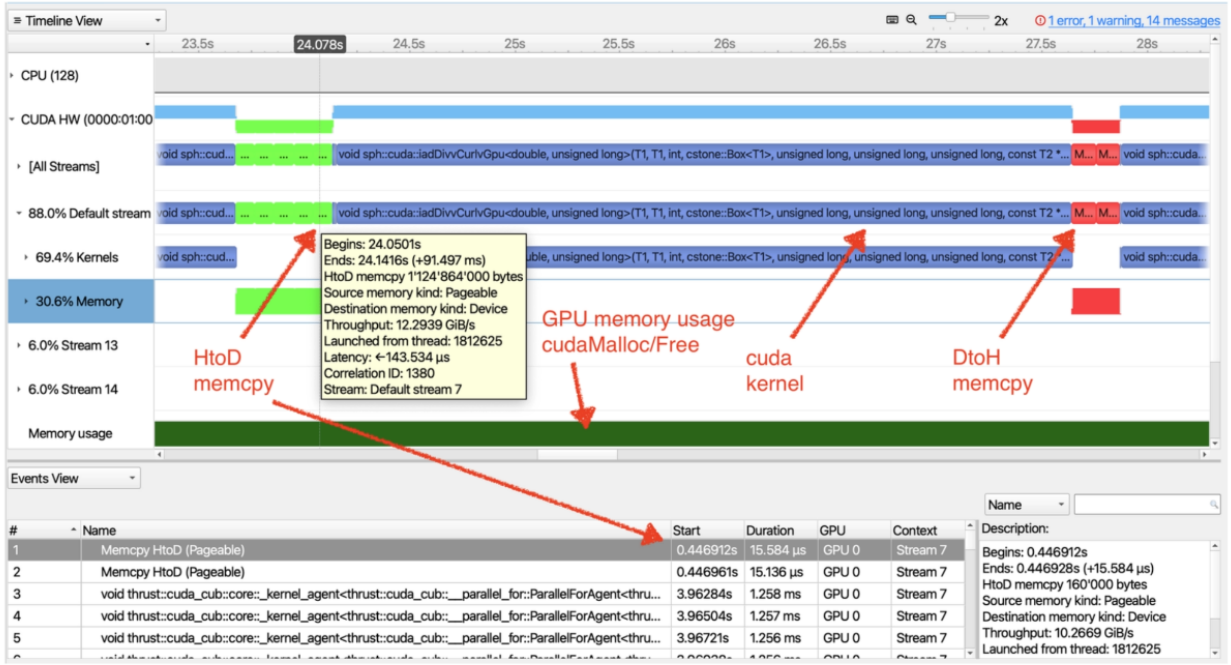
```
uenv image pull prgenv-gnu/24.11:v2
uenv start prgenv-gnu/24.11:v2 --view=default
nsys --version # version 2024.4.2.133-244234382004v0
ncu --version  # Version 2024.3.0.0
```

- No recompilation is needed
- Submit your job with sbatch on Alps

```
srun ~/cuda_visible_devices.sh nsys profile "nsys_args" ./myexe
                               ##########################
# "nsys_args": --trace=cuda,cublas,mpi,nvtx --mpi-impl=mpich
#              --cuda-memory-usage=true --stats=false
#              --output %h.%q{SLURM_NODEID}.%q{SLURM_PROCID}
```

Nsight Systems: User interface

- A succesful job will create 1 or more `*.nsys-rep` report file(s)



- Install the client and open the `nsys-rep` file; GPU performance is often limited by data transfers

Nsight Systems: Interact with the tool

- Measuring compute flops with node-burn

```
gpu 207 iterations,  41 229.97 GFlops, 10.0 seconds, 2.400 Gbytes # node-burn FP64 precision
gpu 303 iterations, 265 990.14 GFlops, 10.0 seconds, 3.221 Gbytes # node-burn FP32 reduced precision
```

- `nsys stats` allows to extract data from `.nsys-rep` file created with `nsys profile`

```
** CUDA Summary (API/Kernels/MemOps) (cuda_api_gpu_sum): FP64.nsys-rep
 Time (%)  Total Time (ns)  Instances   Category    Operation
 --------  --------------  ---------  ----------  ----------------------
     49.4   10,166,101,888         67  CUDA_API    cudaDeviceSynchronize
     49.3   10,165,330,044         64  CUDA_KERNEL sm90_xmma_gemm_f64f64_ <----
      1.0      203,181,888          9  CUDA_API    cudaMalloc
      0.2       32,718,144         19  CUDA_API    cudaFree

 ** CUDA Summary (API/Kernels/MemOps) (cuda_api_gpu_sum): TF32.nsys-rep
 Time (%)  Total Time (ns)  Instances   Category    Operation
 --------  --------------  ---------  ----------  ----------------------
     49.4   10,008,663,584        438  CUDA_API    cudaDeviceSynchronize
     49.4   10,007,302,253        435  CUDA_KERNEL sm90_xmma_gemm_f32f32_ <----
      1.0      198,221,120          9  CUDA_API    cudaMalloc
      0.1       15,499,904         19  CUDA_API    cudaFree

# filtered out Avg (ns), Med (ns), Min (ns), Max (ns), StdDev (ns)
```

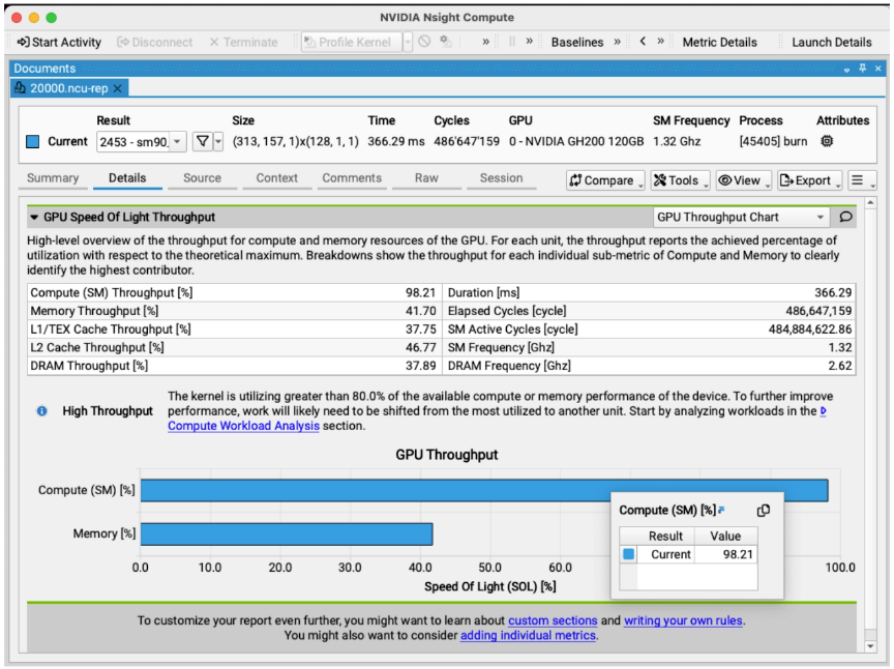
- nsys stats `--help-reports` for more details

Nsight Compute: User interface

- Submit your job on Alps (overhead)

```
srun ~/cuda_visible_devices.sh ncu <ncu args> ./build_fp64/burn -ggemm,10000 -d 10
                               ###############
# <ncu args>: --kernel-name regex:^sm90_xmma_gemm_f64f64_f64f64_f64 -f -o %q{SLURM_JOBID}.%q{SLURM_PROCID}
```

- A succesful job will create 1 or more `*.ncu-rep` report file(s), open them with the ncu client (remotely)



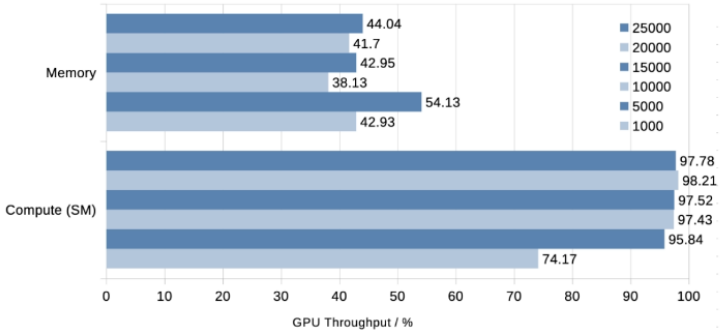
Nsight Compute: Interact with the tool

- `ncu --csv` allows to extract data from `.ncu-rep` file(s)
- ncu collects performance metrics organised in performance sections, by default `SpeedOfLight.section`

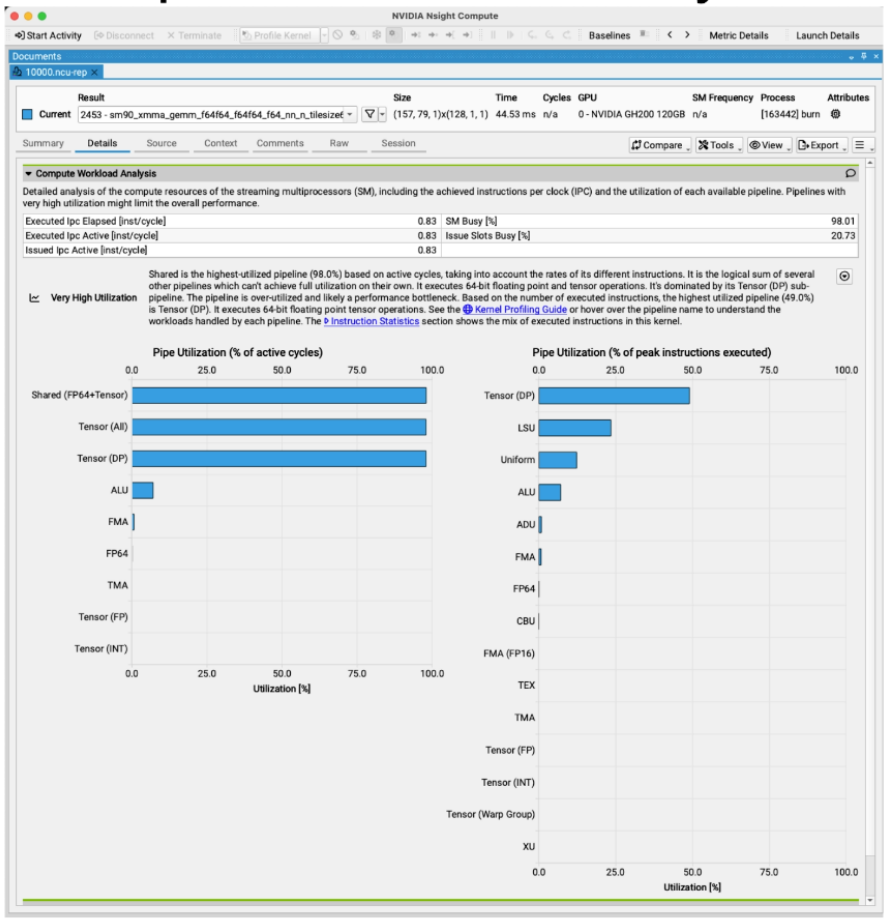
```
ncu --list-sections

C2CLink                        Nvlink_Tables            SpeedOfLight*  WarpStateStatistics
ComputeWorkloadAnalysis<---    Nvlink_Topology          PmSampling*    SchedulerStatistics
InstructionStatistics          MemoryWorkloadAnalysis*  Nvlink         SourceCounters
LaunchStatistics               NumaAffinity             Occupancy      WorkloadDistribution
```

- Rerun with a metric section or metric name for advanced analysis (`ComputeWorkloadAnalysis`)

```
srun ncu --section ComputeWorkloadAnalysis ...
srun ncu --metrics 'sm__throughput.avg.pct_of_peak_sustained_elapsed' ...
```



Nsight Compute: Compute Workload Analysis

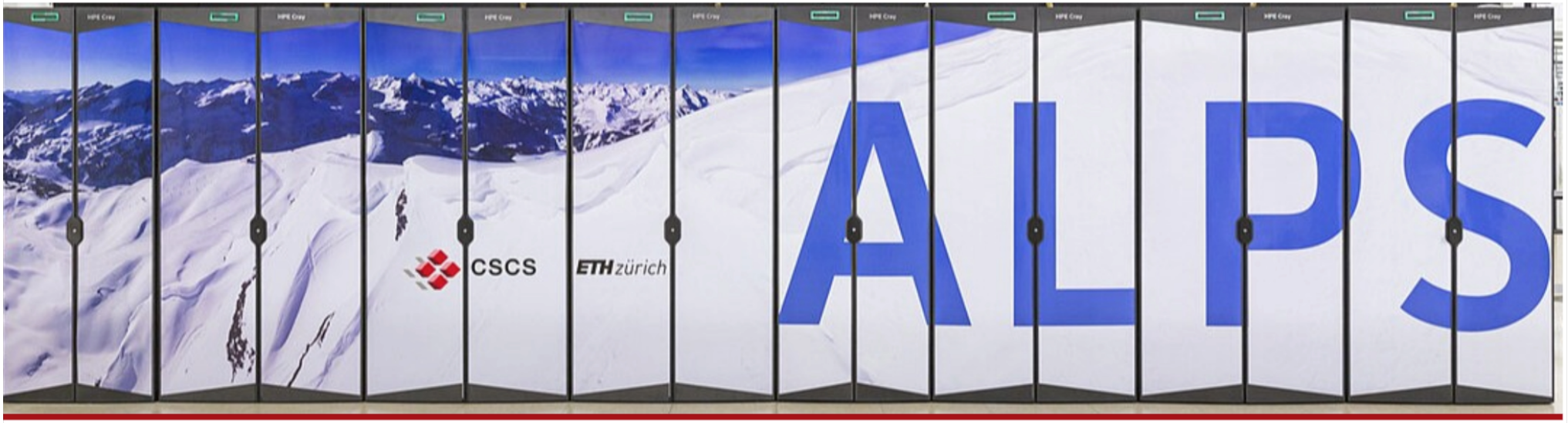


References

- Nsight: https://developer.nvidia.com/tools-overview
  - Nsight Systems: https://developer.nvidia.com/nsight-systems
  - Nsight Compute: https://developer.nvidia.com/nsight-compute

CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich



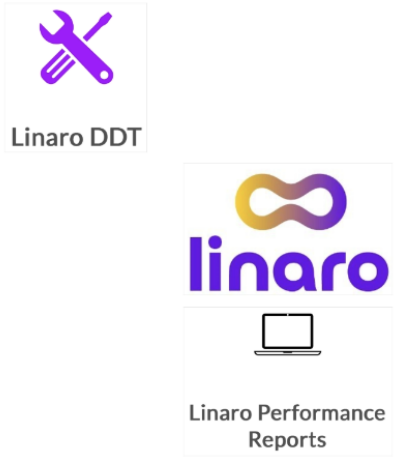
Executing and debugging Multi-GPU Codes on Alps

Swiss National Supercomputing Centre (CSCS)
May 15th, 2025

CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich

Getting Started with Linaro Forge DDT



SOS27_Rudy-Shand_Linaro_2025-03-19.pdf

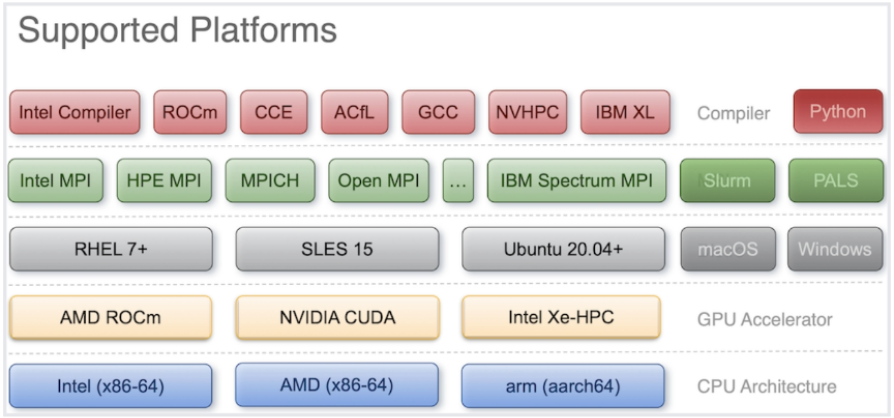
Linaro Forge

Linaro Forge is a toolkit for debugging and profiling parallel applications on CPUs and GPUs

Linaro DDT is a user friendly scalable parallel debugger for C/C++/Fortran, Python applications

Linaro MAP is a performance analysis tool for experts and novices alike

Linaro Performance Reports gives an overview of an application's performance



Linaro DDT: Setup your environment with the tool

- Start a session on Alps with 2 uenvs

```
# uenv image find linaro-forge
uenv image pull prgenv-gnu/24.11:v2
uenv image pull linaro-forge/24.1.2:v1
uenv start prgenv-gnu/24.11:v2,linaro-forge/24.1.2:v1 --view=prgenv-gnu:default
# https://eth-cscs.github.io/cscs-docs/software/devtools/
```

- Test that both uenvs have been mounted correctly

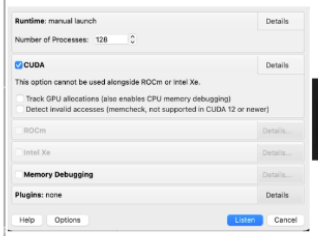
```
> uenv status
linaro:/user-tools    prgenv-gnu:/user-environment
  views:                views:
    forge:                default (loaded):

> source /user-tools/activate



> which mpicxx
/user-environment/env/default/bin/mpicxx

> which ddt-client
/user-tools/env/forge/bin/ddt-client
# find /user-tools/ -name userguide-forge.pdf
```

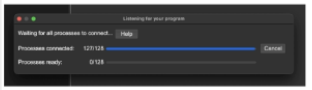
- prgenv-gnu will be mounted at `/user-environment`
- linaro-forge will be mounted at `/user-tools`
- Source the `/user-tools/activate` script instead of the forge view…

- to use the compilers from `/user-environment`
- and use the tools from `/user-tools`

Linaro DDT: Debug your application with the tool

- 🖥️ On your laptop
  - Install and configure the Linaro forge client: https://eth-cscs.github.io/cscs-docs/software/devtools/linaro-uenv
  - Connect the client to Alps using the Remote Launch menu
  - Set the number of processes to listen to in Manual launch

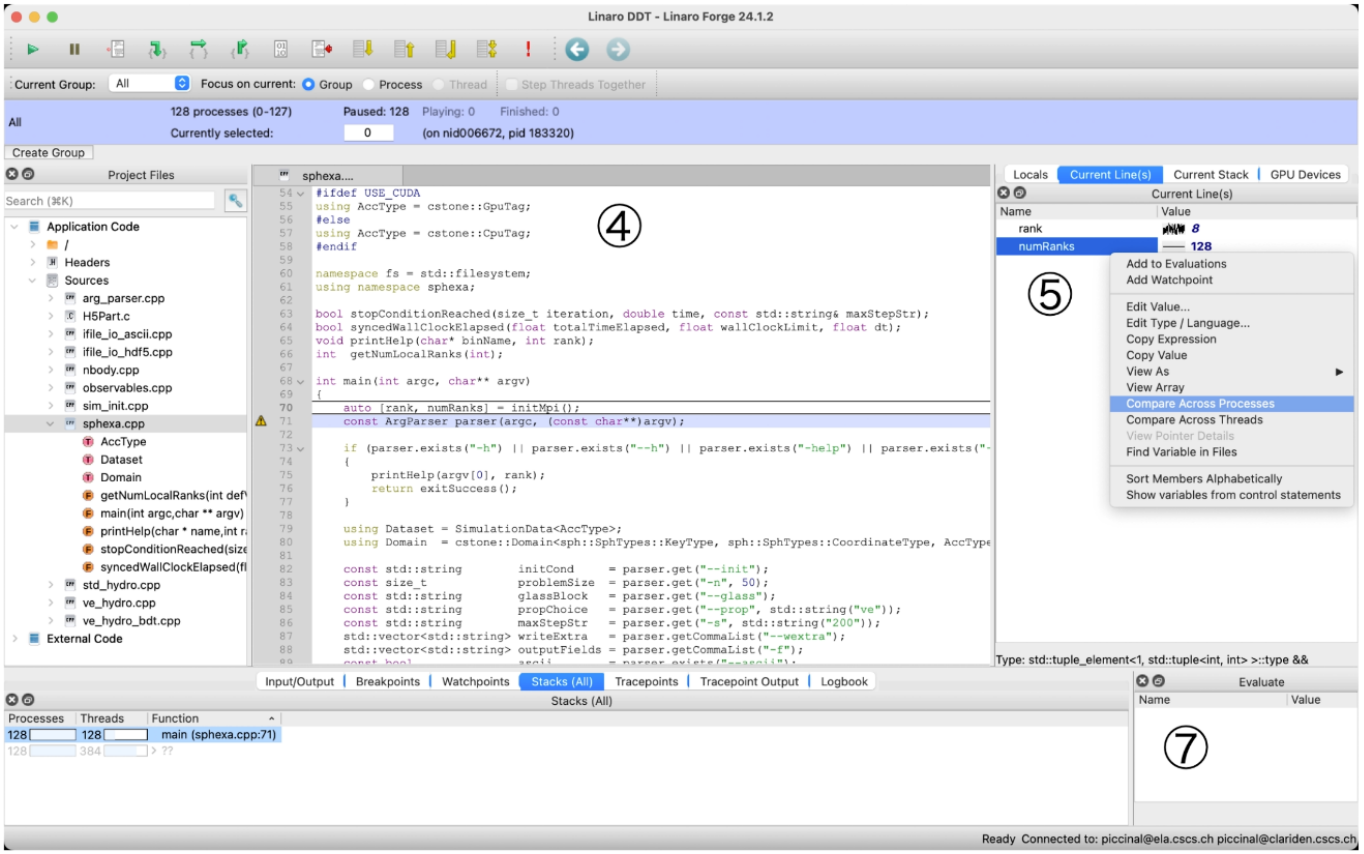




- ⛰️ Then on Alps
  - Compile your code with the `-g` (for CPU) and `nvcc -G` (for GPU) debugging flags
  - Add `ddt-client` to your job:

```
srun -pdebug -t10 -n128 -N32 ./cuda_visible_devices.sh ddt-client ./myexe
                                                       ##########
```

Linaro DDT: User interface



① Navigation controls

② Process controls

③ Files/Functions explorer

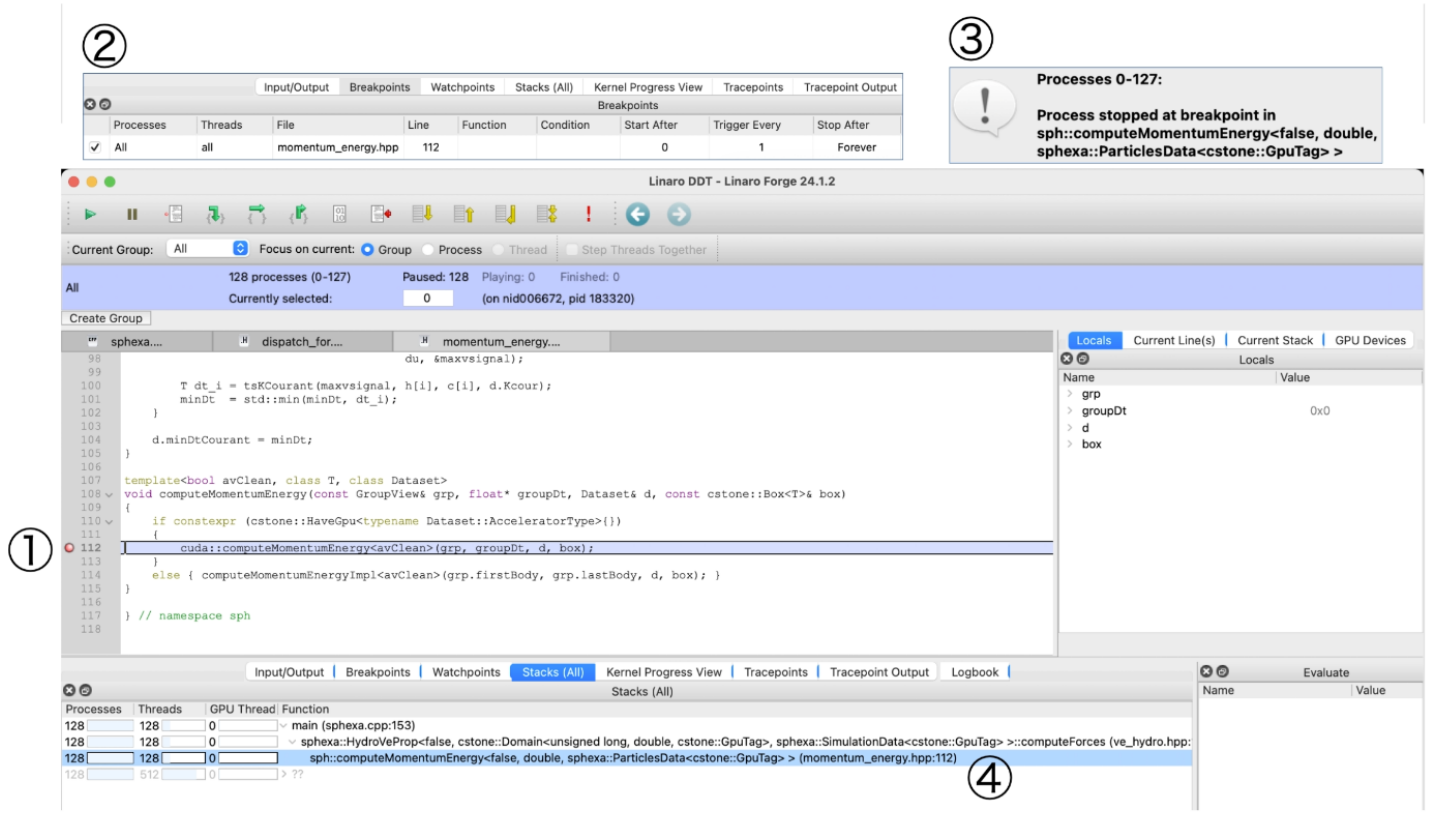
④ Source code view

⑤ Variables explorer

⑥ Parallel callstack view

⑦ Evaluate view

Linaro DDT: Controlling CPU execution (breakpoints)



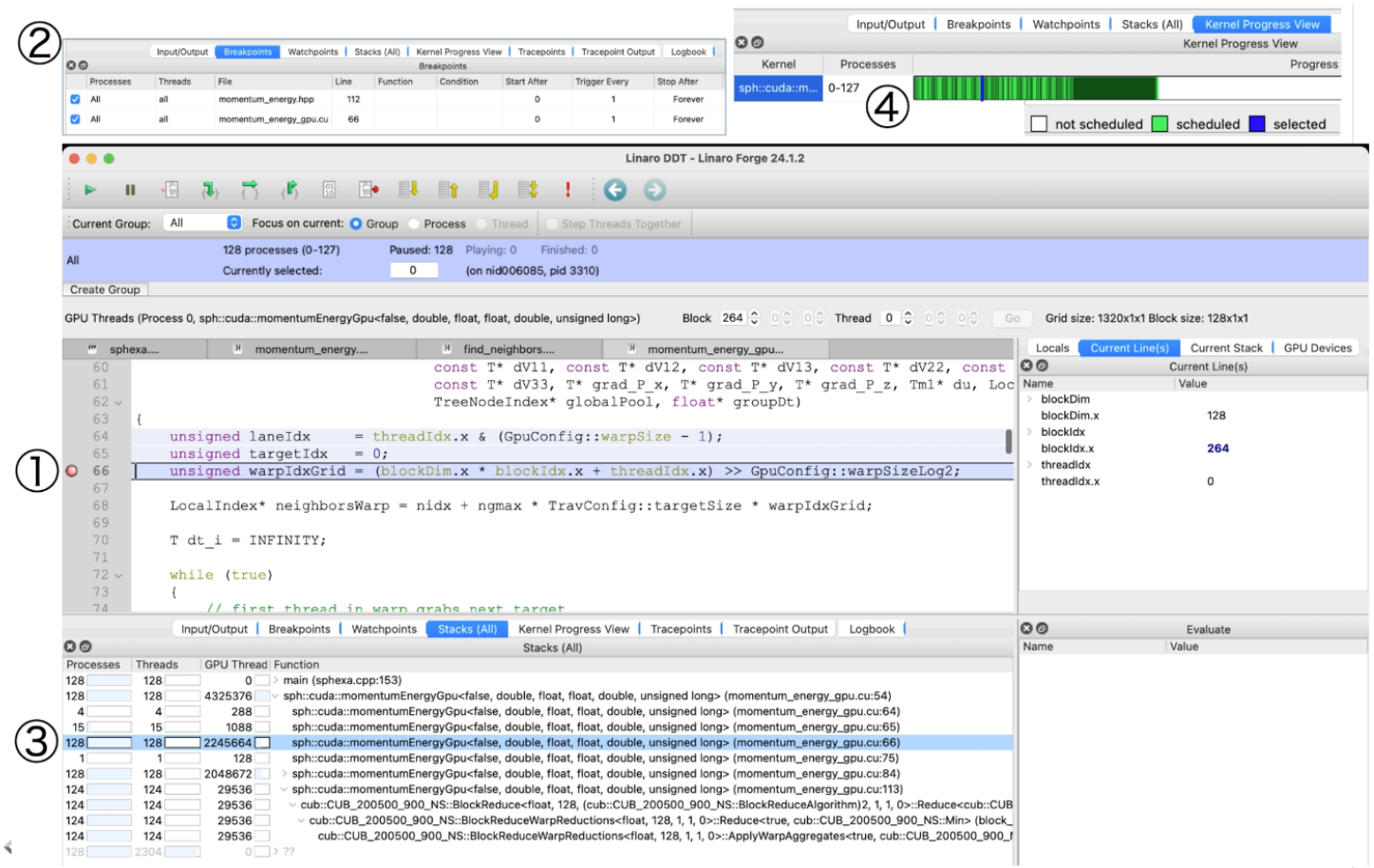
① Source code view: add breakpoint(s) to pause CPU execution

② Breakpoint view: enable/disable/delete (user/conditional/watch/trace) breakpoints

③ Control message: DDT will pause execution when reaching a breakpoint

④ Parallel Stack view: displays CPU functions progress per process

Linaro DDT: Controlling GPU execution (breakpoints)



① Source code view: add breakpoint(s) to pause GPU threads execution

② Breakpoint view: enable/disable/delete breakpoints

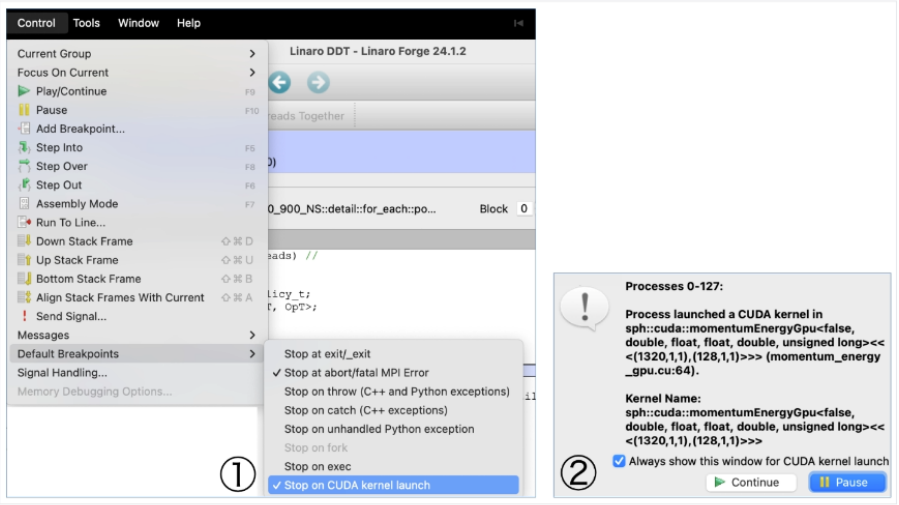
③ Parallel Stack view: display the location and number of GPU threads

④ Kernel Progress View: displays kernel progress, including scheduled and inactive GPU threads

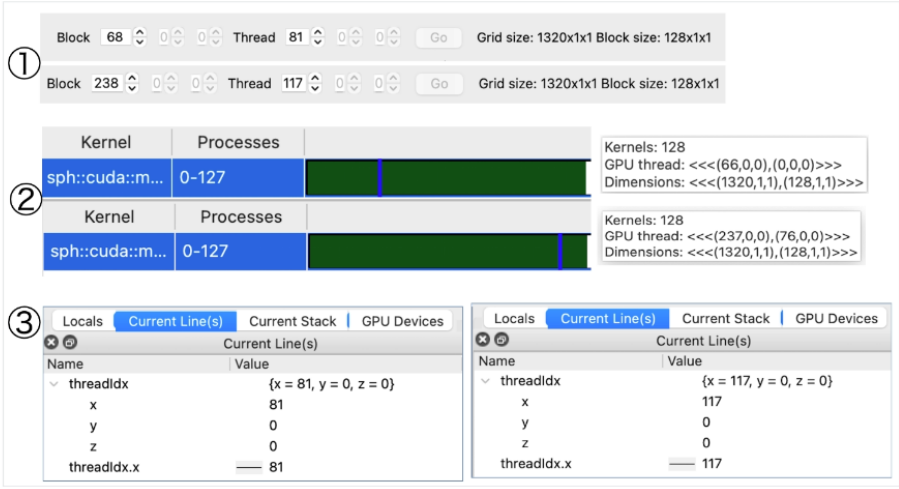
Linaro DDT: Controlling GPU execution (Kernel Launch)

- Stop on CUDA kernel launch:

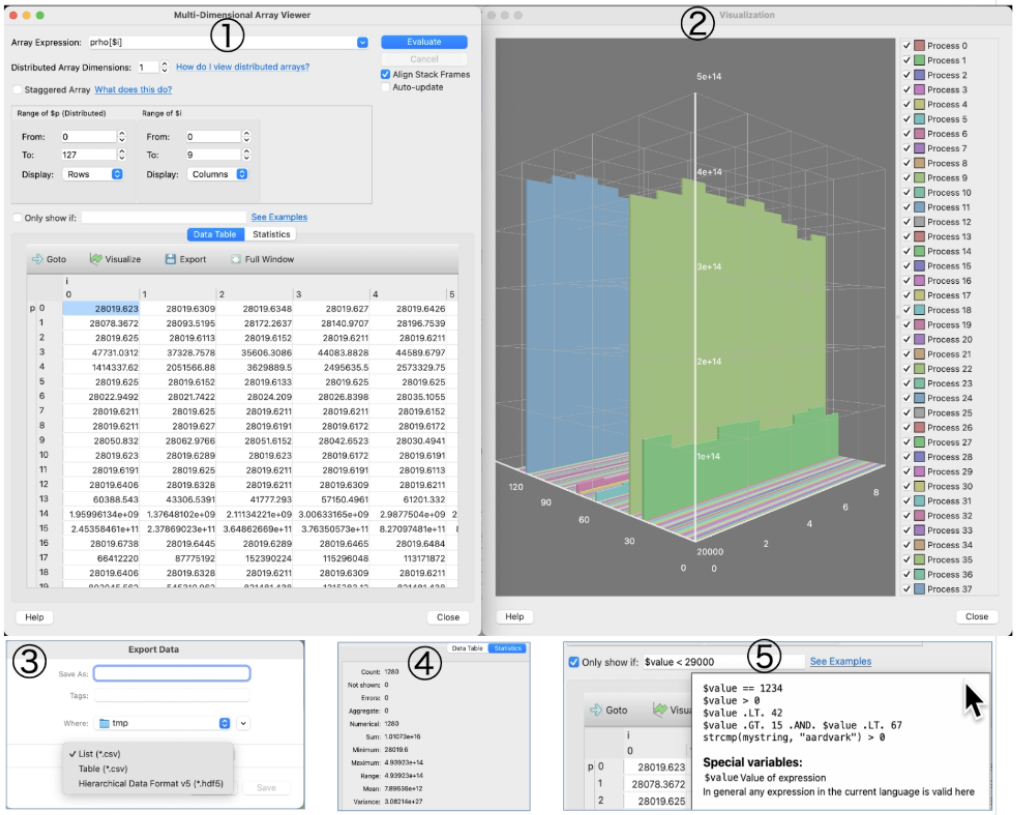
  DDT will pause execution at the entry point of each kernel launch



- Navigate between CUDA threads



Linaro DDT: Multi-dimensional Array Viewer



① Array Viewer: inspect distributed data

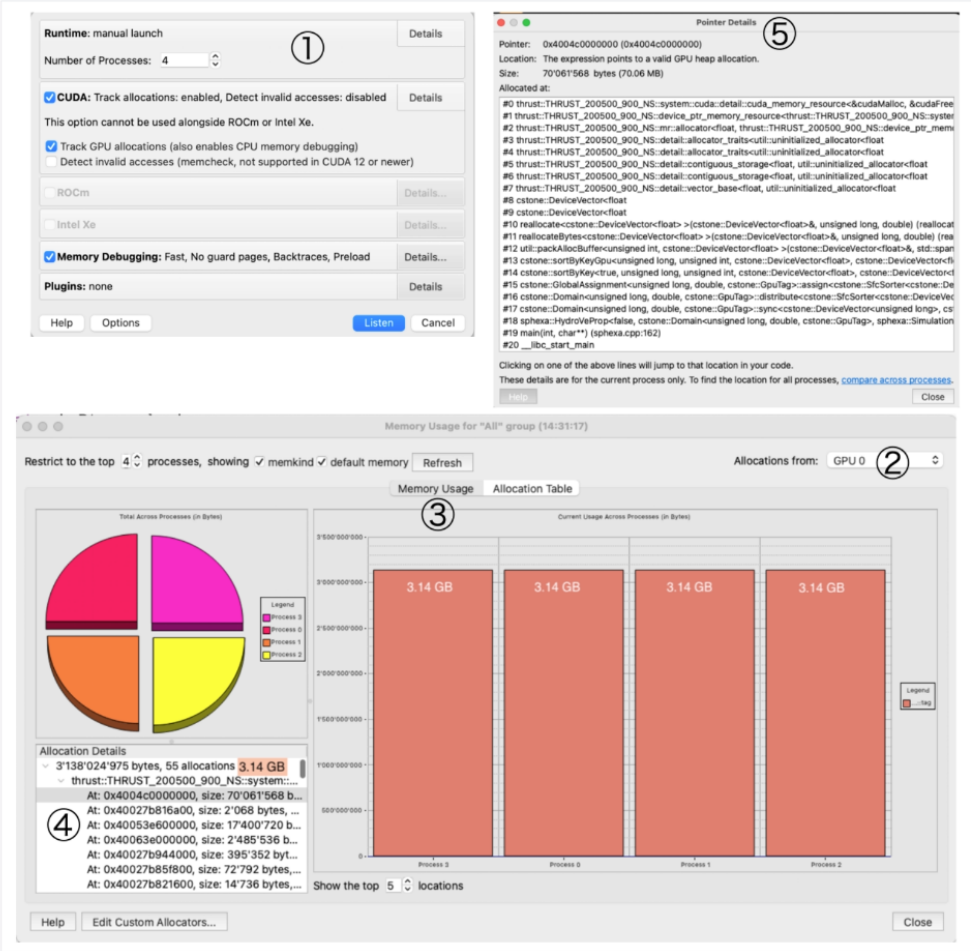
② Visualize: data with plots

③ Export: data in csv or hdf5 format

④ Statistics: interpret your data

⑤ Filter: data values

Linaro DDT: Memory debugging



① Enable memory debugging in the Run window, start the job, pause execution and select `Current Memory Usage` from the Tools menu

② Navigate between GPUs from the drop-down menu

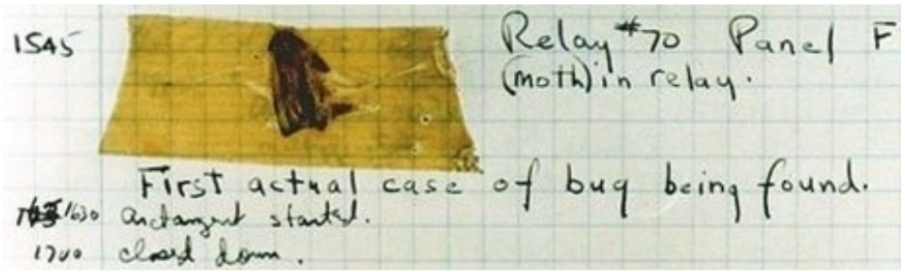
③ Find out how much memory is allocated

④ Track the amount of memory allocated to the pointer(s)

⑤ Inspect which part of the code is using memory

Linaro Forge: Constraints

- ALPS has a license for a maximum of 128 processes
  - Currently, CSCS only has a license for NVIDIA GPUs
  - Get in touch if you need help
  - LUMI has a license for a maximum of 512 processes



```
Fun fact: On September 9th, 1947, computer scientists and engineers in Cambridge,
found a moth caught between the relay contacts of the Harvard Mark II computer.
They recorded the incident as the "first actual case of a bug being found."
```

CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich

Getting Started with Linaro Forge MAP

Linaro MAP: Compile and run your code with the tool

- Load the uenvs and check that the tool is loaded

```
uenv start prgenv-gnu/24.11:v2,linaro-forge/24.1.2:v1 --view=prgenv-gnu:default
source /user-tools/activate

ls -l /opt/cray/pe/cti/default/ # the tool requires cray-cti ⚠

which map # /user-tools/env/forge/bin/map
# find /user-tools/ -name userguide-forge.pdf
```

- Build your code with line-level profiling (`-lineinfo`) and optimization flags

```
cmake -S src -B build -DCMAKE_CUDA_FLAGS="-lineinfo" -DCMAKE_CXX_FLAGS="-g1 -Ofast" -DCMAKE_CUDA_ARCHITECTURES=90
cmake --build build
```

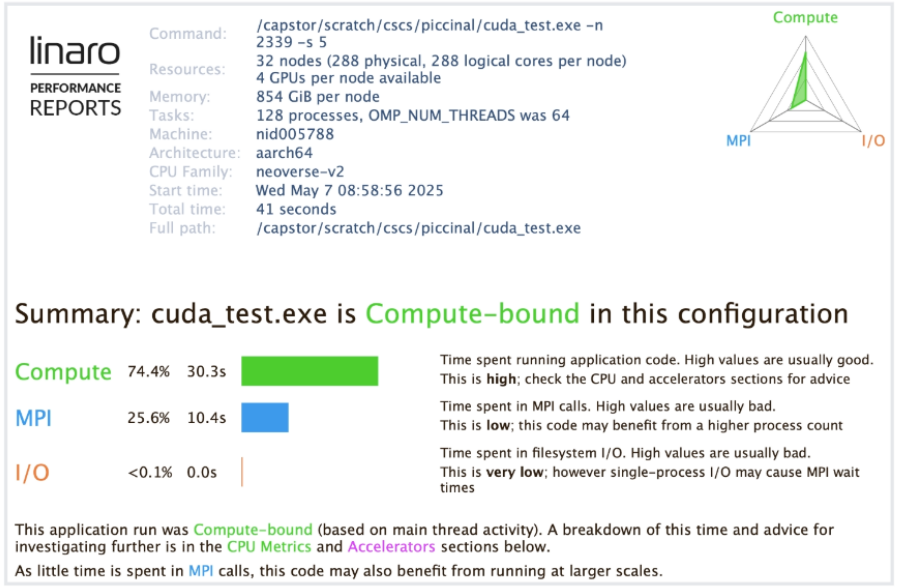
- Submit your job with sbatch on Alps (`map` instead of `srun`)

```
export FORGE_DEBUGGER_WRAPPER=$HOME/cuda_visible_devices.sh
map -n $SLURM_NTASKS --mpi=slurm --mpiargs="--ntasks-per-node=4" --profile  ./myexe
###                                                              #########
# Add --cuda-kernel-analysis and --cuda-transfer-analysis for detailed analysis
```

Linaro MAP: Analyze the performance

```
perf-report myexe_128p_32n_64t_2025-05-07_08-58.map
```

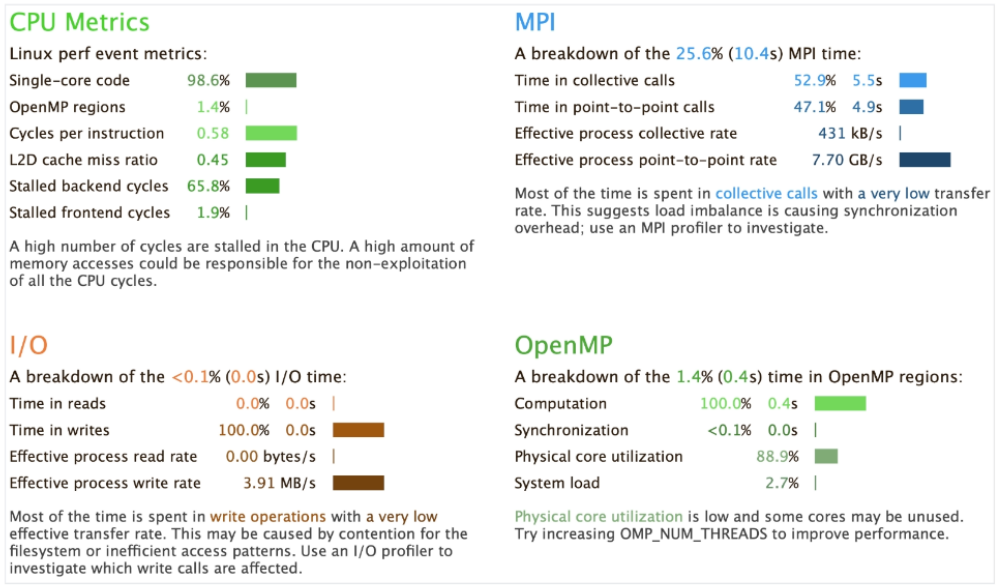
- Run `perf-report` to summarize performance from a profile generated by `MAP`
  - reports in .html and .txt formats by default ➡️
  - add `perf-report -o rpt.csv` for .csv format
  - or `map --report=<summary|txt|csv|html>` directly
- Report summary shows distribution of elapsed time
  - in Compute, MPI, I/O (and Python) regions
  - plus a performance radar chart
  - and helpful pieces of advices

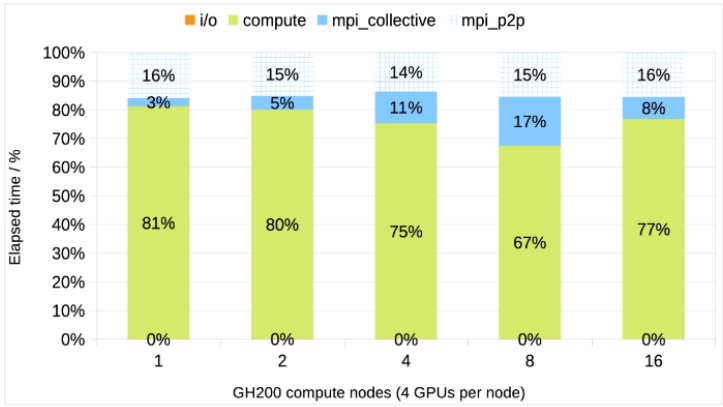


https://docs.linaroforge.com/24.1.3/html/forge/performance_reports/interpreting_performance_reports/index.html

Linaro MAP: Analyze the host performance

- `perf-report` gives a high-level overview of application performance (Compute, MPI, I/O)

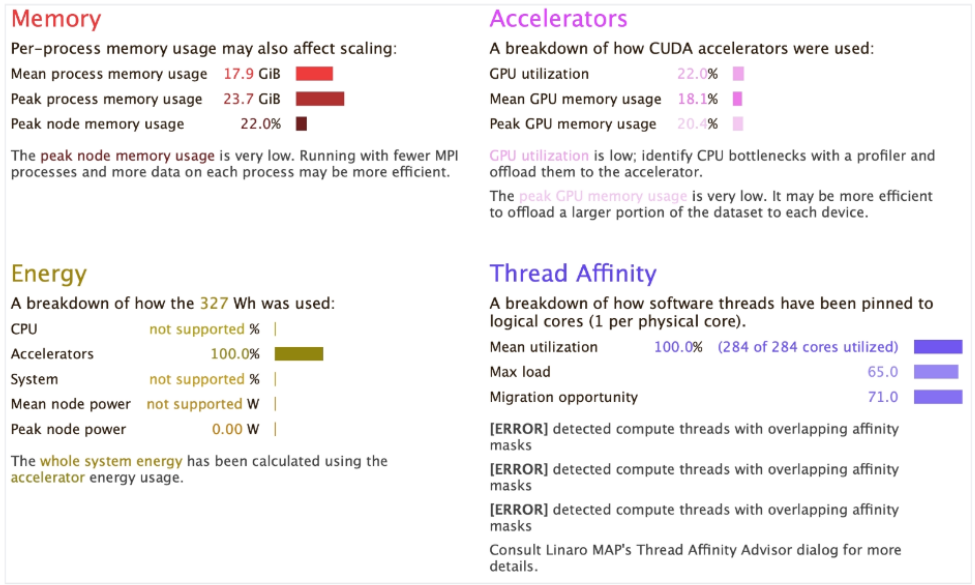




- CPU time spent in application code, MPI time and transfer rates spent in MPI calls ⬆️
- I/O time spent in open/close and read/write filesystem operations
- OpenMP time spent in OpenMP regions (computation and synchronization)
  - All % are relative to the breadown region time, not the wallclock time; make custom charts using the exported data (`--csv`)

Linaro MAP: Analyze the device performance

- `perf-report` gives a high-level overview of application performance (Memory, GPU, Energy, CPU Affinity)



Memory

Per-process memory usage may also affect scaling:

Mean process memory usage	17.9 GiB
Peak process memory usage	23.7 GiB
Peak node memory usage	22.0%

The peak node memory usage is very low. Running with fewer MPI processes and more data on each process may be more efficient.

Accelerators

A breakdown of how CUDA accelerators were used:

GPU utilization	22.0%
Mean GPU memory usage	18.1%
Peak GPU memory usage	20.4%

GPU utilization is low; identify CPU bottlenecks with a profiler and offload them to the accelerator.

The peak GPU memory usage is very low. It may be more efficient to offload a larger portion of the dataset to each device.

Energy

A breakdown of how the 327 Wh was used:

CPU	not supported %
Accelerators	100.0%
System	not supported %
Mean node power	not supported W
Peak node power	0.00 W

The whole system energy has been calculated using the accelerator energy usage.

Thread Affinity

A breakdown of how software threads have been pinned to logical cores (1 per physical core).

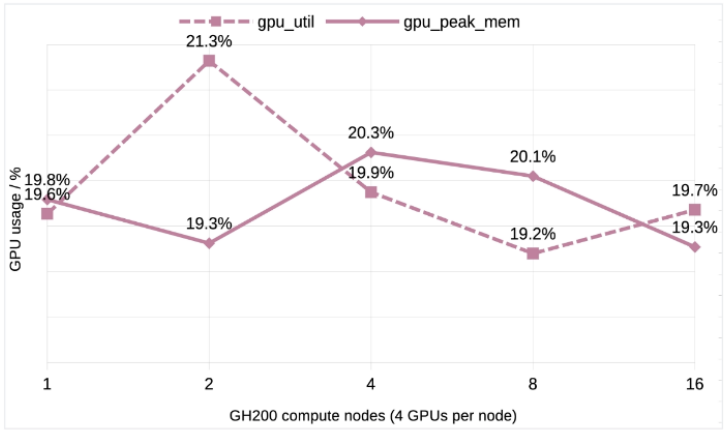
Mean utilization	100.0% (284 of 284 cores utilized)
Max load	65.0
Migration opportunity	71.0

[ERROR] detected compute threads with overlapping affinity masks

[ERROR] detected compute threads with overlapping affinity masks

[ERROR] detected compute threads with overlapping affinity masks

Consult Linaro MAP's Thread Affinity Advisor dialog for more details.



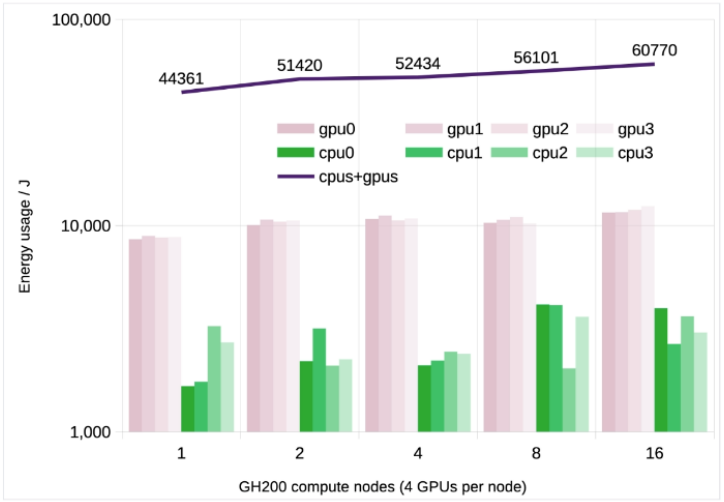
- Memory (RSS) `unified` memory usage across all processes and nodes over the entire job duration ⬆️
- Accelerators GPU usage (CUDA kernels and memory)
- Energy used by the job
- Thread affinity cpu bindings (licensed feature)

Linaro MAP: Energy usage

- Collecting Consumed Energy Data on Alps is possible
  - Energy (and Power) `pm_counters` can be read from the `/sys/cray/pm_counters/` files

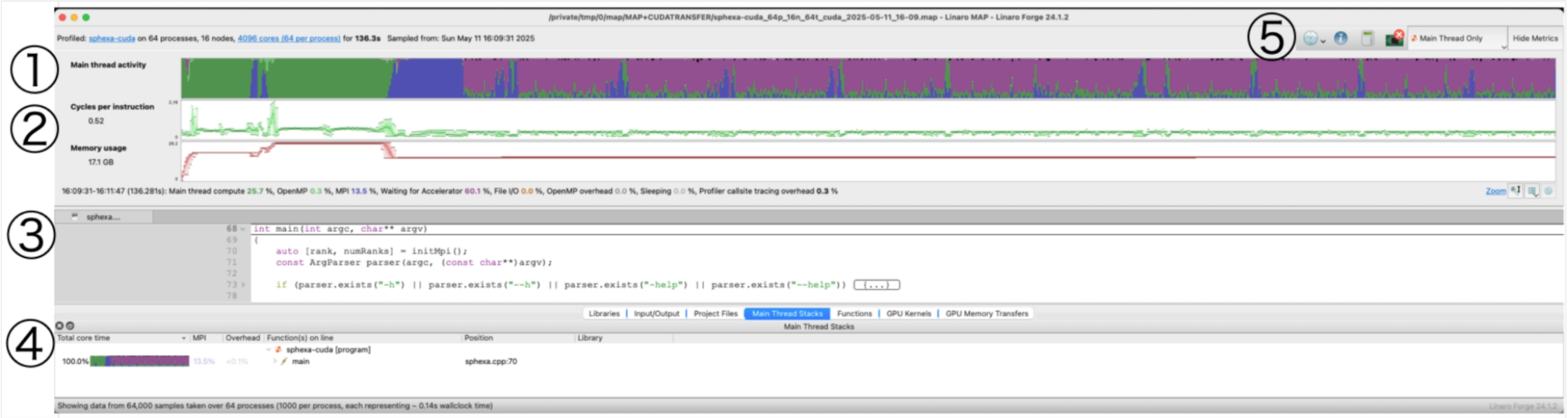
```
# cat /sys/cray/pm_counters/energy
2267329144 J 1746883535815882 us
```

- Consumed Energy data
  - at Node, CPU and GPU levels
  - Energy of a single Node job is $E_{t1} - E_{t0}$ (in Joules)
  - Default collection rate is 10 Hz
  - ➡️



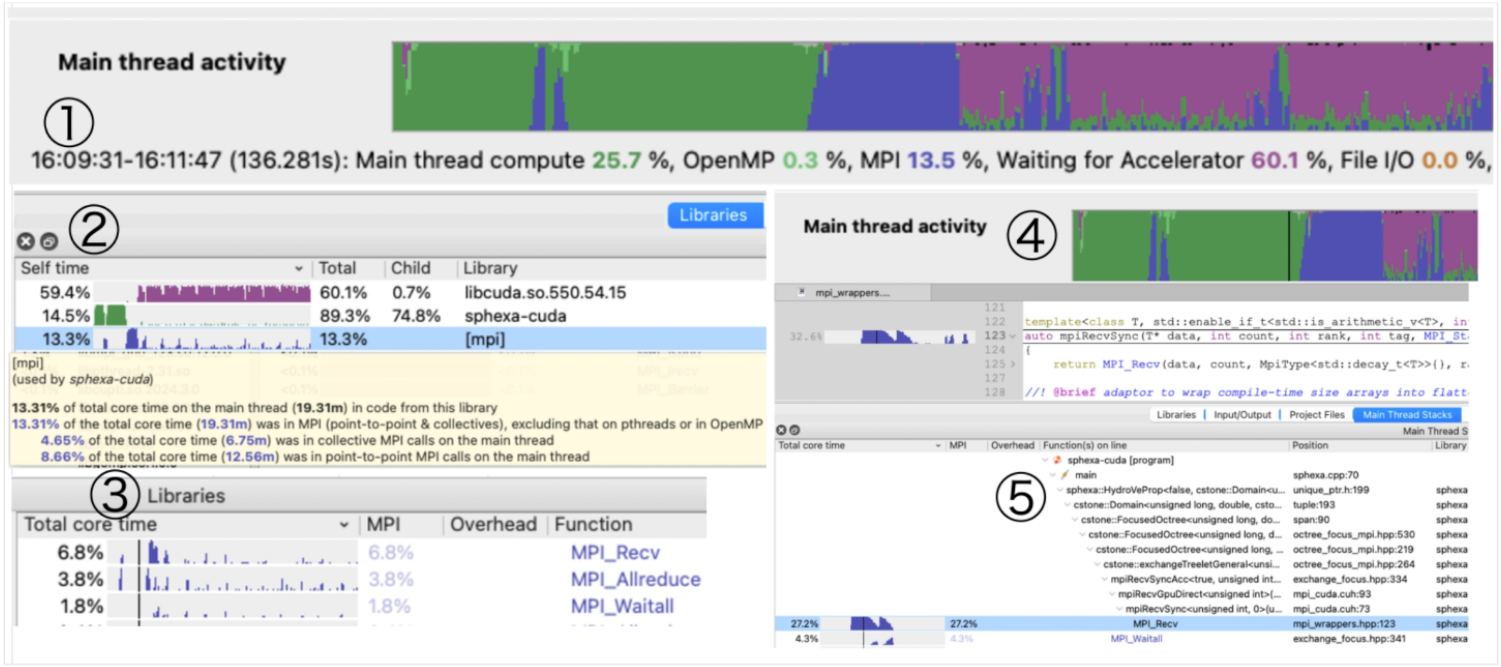
- Energy usage at Node level can also be accessed with the Slurm `sacct` command

Linaro MAP: MAP User Interface



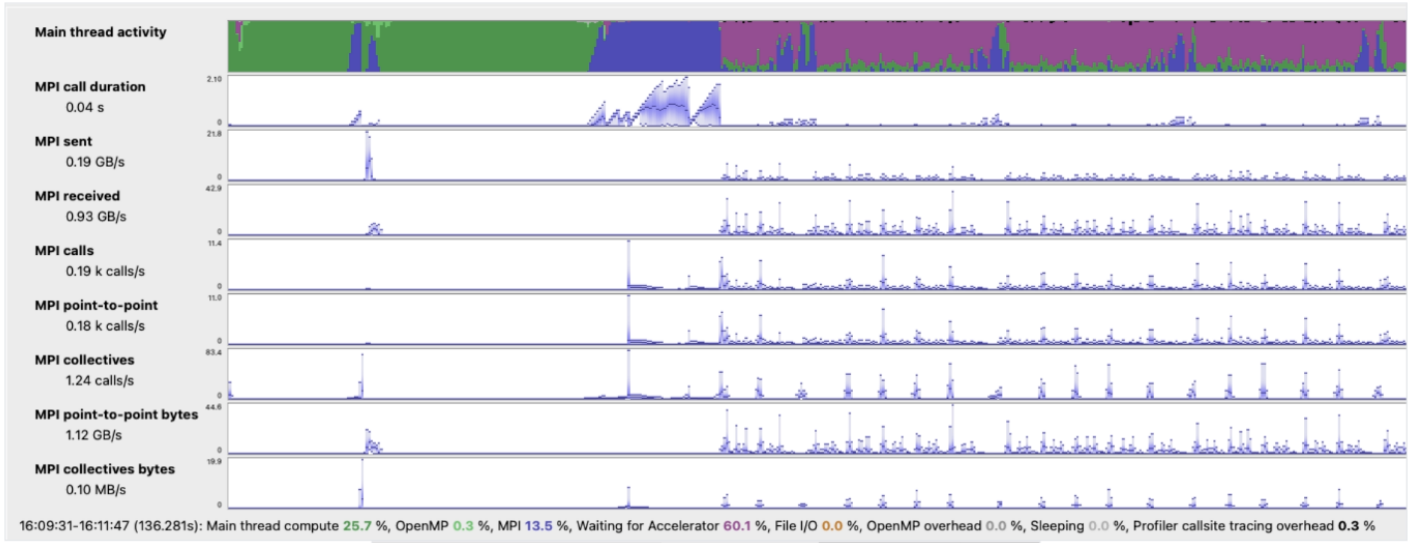
① Activity Timeline view: CPU, MPI, OpenMP, Accelerators, I/O, Memory
② Metrics view
③ Source Code view including editing and version control
④ Sparkline charts and Stack view and tabs
⑤ Time switch (% of runtime or total core-time) and Program details

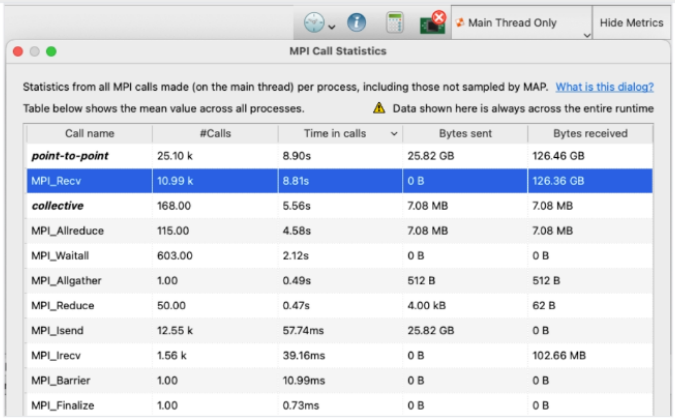
Linaro MAP: MPI functions



① 3 regions: CPU (25%), MPI (13%) and GPU (60%)
② Library view: 4.6% of the time in MPI collectives, 8.7% in MPI point-to-point
③ In P2P, top function is `MPI_Recv` (6.8%) (|)
④ Zoom in the MPI region of interest
⑤ Use the Stack view to locate the MPI call in the code source

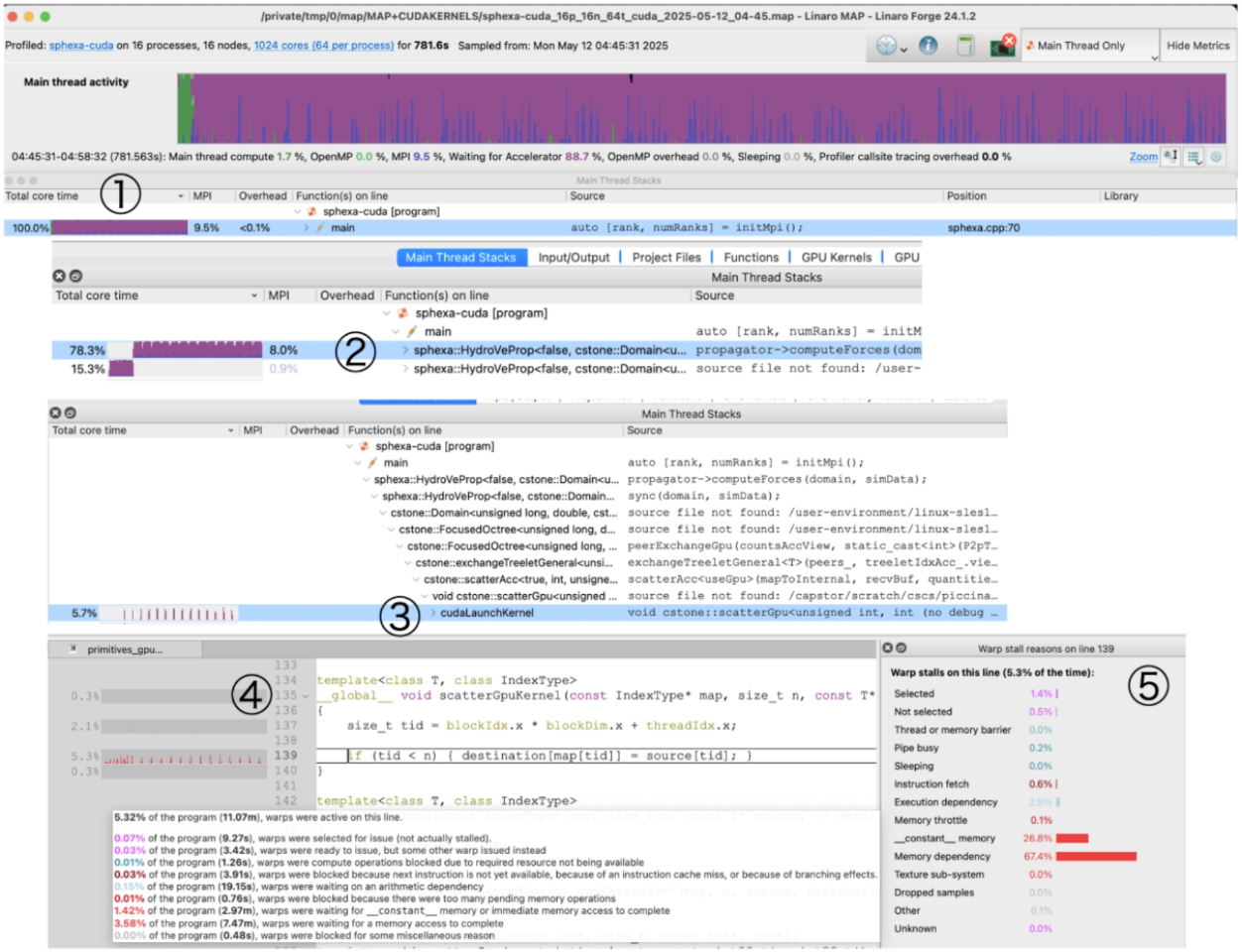
Linaro MAP: MPI metrics and statistics





Call name	#Calls	Time in calls	Bytes sent	Bytes received
point-to-point	25.10 k	8.90s	25.82 GB	126.46 GB
MPI_Recv	10.99 k	8.81s	0 B	126.36 GB
collective	168.00	5.56s	7.08 MB	7.08 MB
MPI_Allreduce	115.00	4.58s	7.08 MB	7.08 MB
MPI_Waitall	603.00	2.12s	0 B	0 B
MPI_Allgather	1.00	0.49s	512 B	512 B
MPI_Reduce	50.00	0.47s	4.00 kB	62 B
MPI_Isend	12.55 k	57.74ms	25.82 GB	0 B
MPI_Irecv	1.56 k	39.16ms	0 B	102.66 MB
MPI_Barrier	1.00	10.99ms	0 B	0 B
MPI_Finalize	1.00	0.73ms	0 B	0 B

Linaro MAP: CUDA kernel analysis



① GPU activity dominates most of the total core time

② `computeForces` function uses a significant share (78% of which 70% waiting for the GPU)

③ `scatterGpuKernel` CUDA kernel is launched iteratively

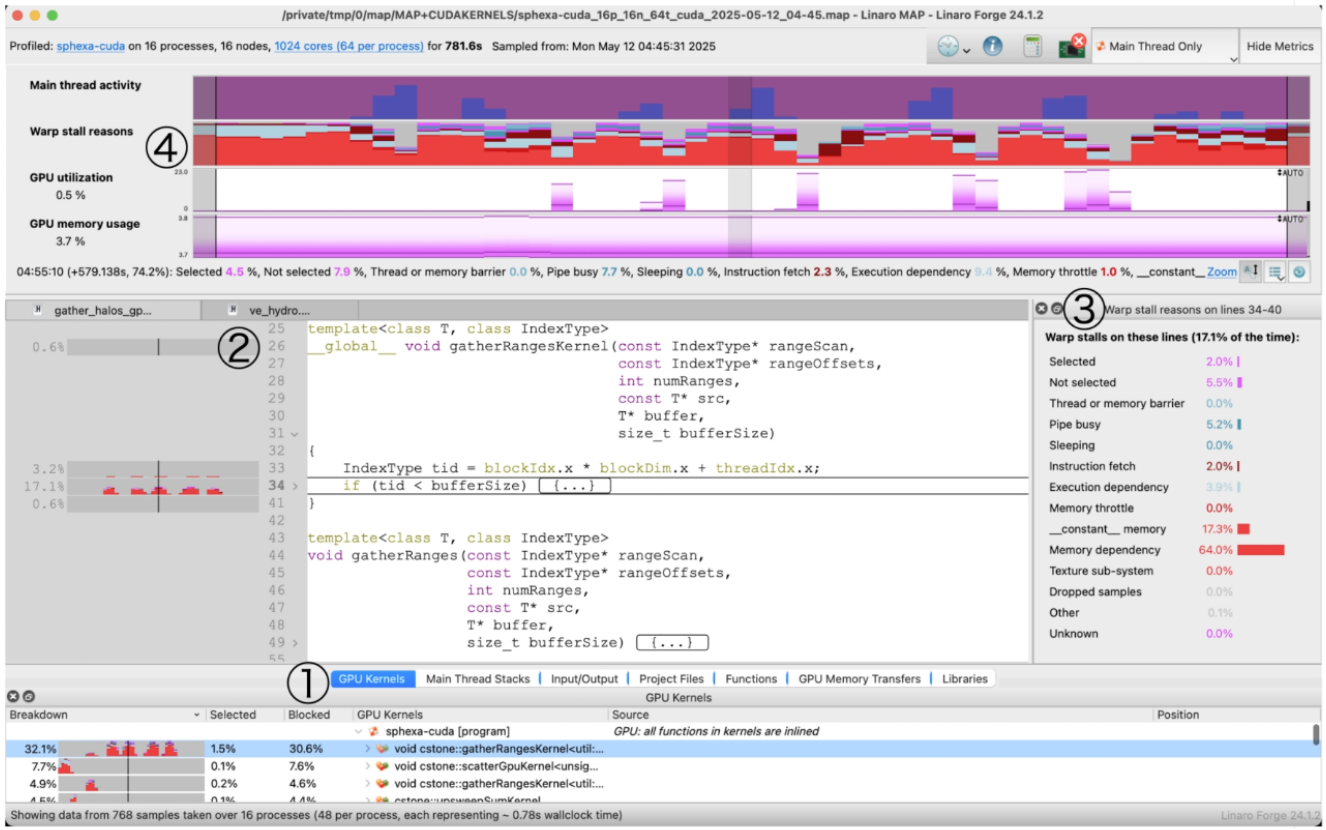
④ Source Code view gives some reasons to warp stalls impacting performance

⑤ Primary reason for warp stalls is Memory Dependency, followed by Memory Accesses

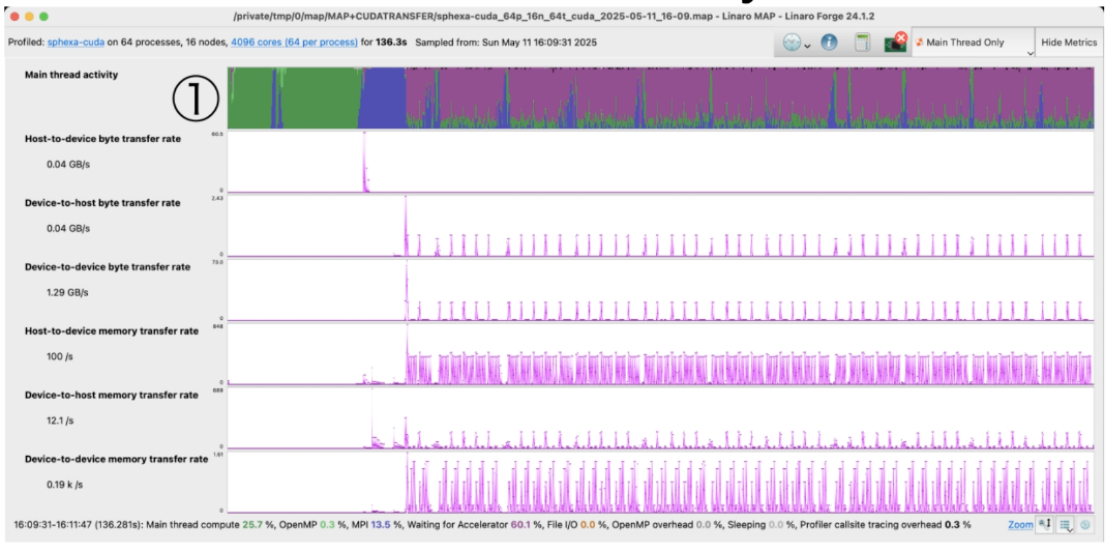
Linaro MAP: CUDA Warp Stall Reasons

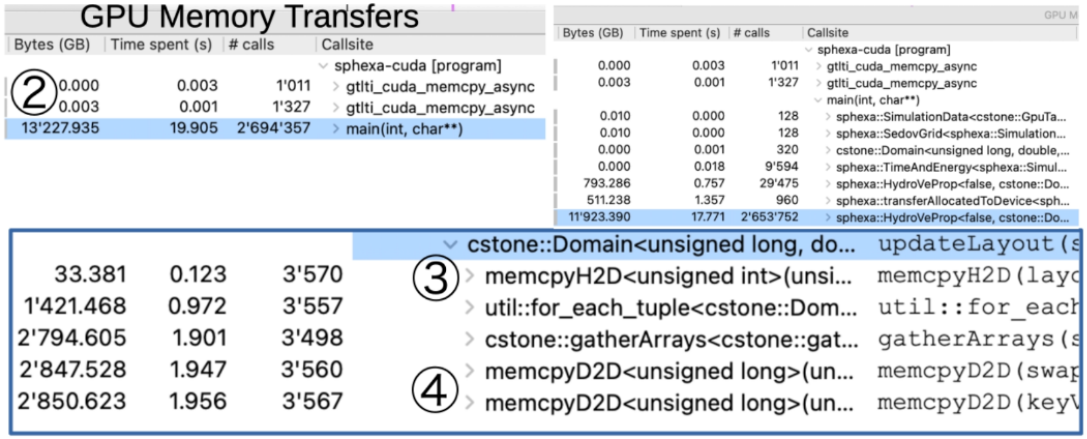
- Instruction fetch: Warp is blocked because next instruction is not yet available, because of an instruction cache miss, or because of branching effects.
- Execution dependency: Instruction is waiting on an arithmetic dependency.
- Memory dependency: Warp is blocked because it is waiting for a memory access to complete.
- Texture sub-system: Texture sub-system is fully utilized or has too many outstanding requests.
- Thread or memory barrier: Warp is blocked as it is waiting at __syncthreads or at a memory barrier.
- __constant__ memory: Warp is blocked waiting for __constant__ memory and immediate memory access to complete.
- Pipe busy: Compute operation cannot be performed due to required resource not being available.
- Memory throttle: Warp is blocked because there are too many pending memory operations.
- Not selected: Warp was ready to issue, but some other warp issued instead.
- Dropped samples: Samples dropped (not collected) by hardware due to backpressure or overflow.

Linaro MAP: CUDA kernel analysis



Linaro MAP: CUDA Memory Transfers





⓪ Enable CUDA memory transfer profiling with the `--cuda-transfer-analysis` flag in MAP

① The CUDA metrics view displays GPU memory transfer rates: Host-to-Device (H2D), Device-to-Host (D2H), Device-to-Device (D2D), measured in Bytes and calls per process

② The GPU Memory Transfers tab provides a detailed breakdown for memory transfers

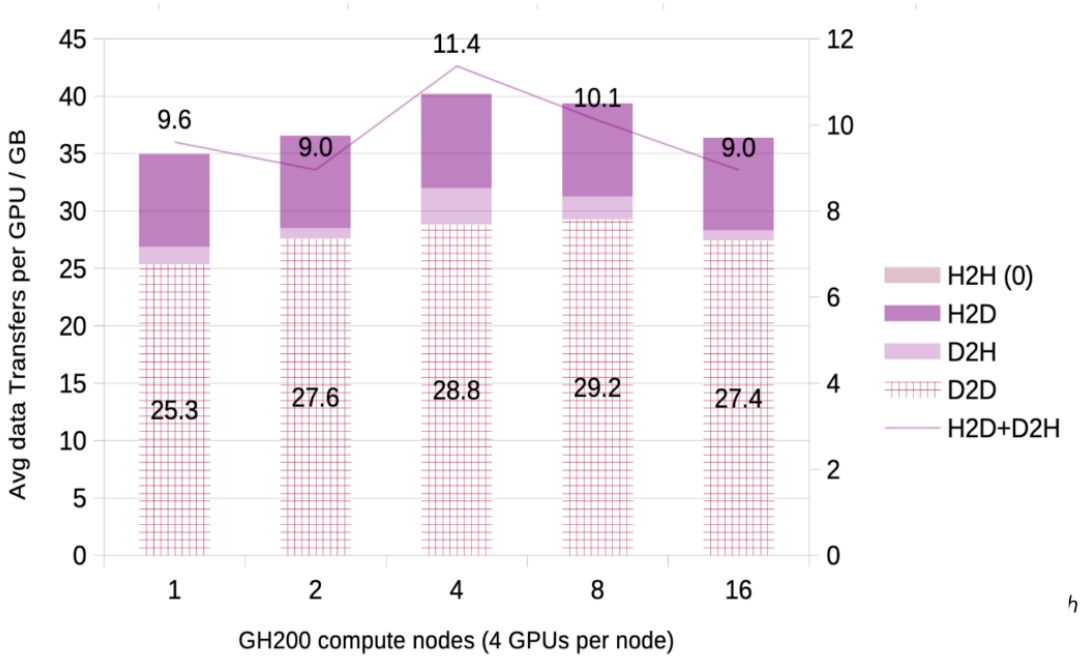
③ ④ Host-to-Device and Device-to-Device memory copies occur in the `updateLayout` function

Linaro MAP: Interact with the tool

- MAP can export profiling data in JSON format
  - open a .map file in the Linaro MAP GUI

    then export the data in the `File/Export Profile Data as JSON` menu
  - or from the command line `map --export=myfile.json myfile.map`

```
jq '.samples.metrics | keys' myfile.json \
  |grep dtod
  "gpu_dtod_bytes_transferred_per_second",
  "gpu_dtod_bytes_transferred_total",
  "gpu_dtod_memory_transfers_per_second",
  "gpu_dtod_memory_transfers_total",
  "gpu_dtod_transfer_time_percentage",
  "gpu_dtod_transfer_time_total",
```

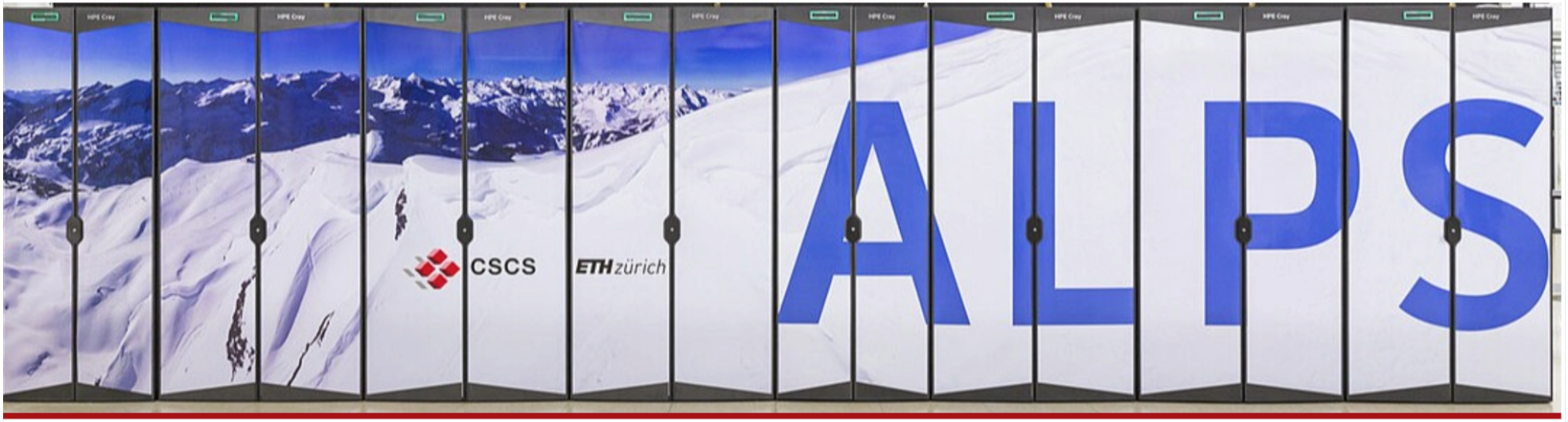


References

- Rudy-Shand: https://sos27.cscs.ch/wp-content/uploads/2025/04/SOS27_Rudy-Shand_Linaro_2025-03-19.pdf
- Linaro: https://www.linaroforge.com
  - DDT: https://www.linaroforge.com/linaro-ddt
  - MAP: https://www.linaroforge.com/linaro-map
  - PERF-REPORT: https://www.linaroforge.com/linaro-performance-reports
  - UENV: https://eth-cscs.github.io/cscs-docs/software/devtools/linaro-uenv
  - LUMI: https://lumi-supercomputer.github.io/LUMI-EasyBuild-docs/a/ARMForge/
  - CUPTI: https://docs.nvidia.com/cupti/api/group__CUPTI__ACTIVITY__API.html

CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich



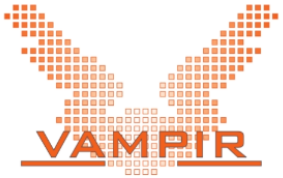
Executing and debugging Multi-GPU Codes on Alps

Swiss National Supercomputing Centre (CSCS)
May 15th, 2025

CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich

Getting Started with Score-P/Scalasca



- Performance analysis is the process of measuring the performance of an application with a tool
- The `Virtual Institute for High Productivity Supercomputing` (www.vi-hps.org) develops an open source Scalable Performance Measurement Infrastructure for Parallel Codes

- `Score-P` is one of the performance analysis tool supported by VI-HPS
  - Profiling gives a statistical overview of the measured performance
    - It can help to quickly identify hotspots or performance bottlenecks
  - Tracing records a detailed timeline of events
    - It can help to understand performance issues with a higher level of details (and overhead)

Score-P: Compiling your code with the tool

- Load the uenv (in /user-environment):

```
uenv image pull scorep/9.0-gcc13:v1  # uenv image find scorep
uenv start scorep/9.0-gcc13:v1 --view default

scorep --version # 9.0
scalasca --version # 2.6.2
cubelib-config --version # 4.9
find /user-environment/ -name scorep.pdf
```

- Invoke `cmake` without instrumentation:

```
SCOREP_WRAPPER=OFF cmake -S src -B build
-DCMAKE_CXX_COMPILER=scorep-mpic++ \
-DCMAKE_C_COMPILER=scorep-mpicc \
-DCMAKE_CUDA_COMPILER=scorep-nvcc \
-DCMAKE_CUDA_ARCHITECTURES=90
```

then start building with instrumentation:

```
SCOREP_WRAPPER=ON cmake --build build
```

- Select the report type before running the executable:

```
export SCOREP_ENABLE_PROFILING=true # Call-path profiling: CUBE4 data format (profile.cubex)
export SCOREP_ENABLE_TRACING=true # Event-based tracing: OTF2 data format (traces.otf2)
```

Score-P: Profiling

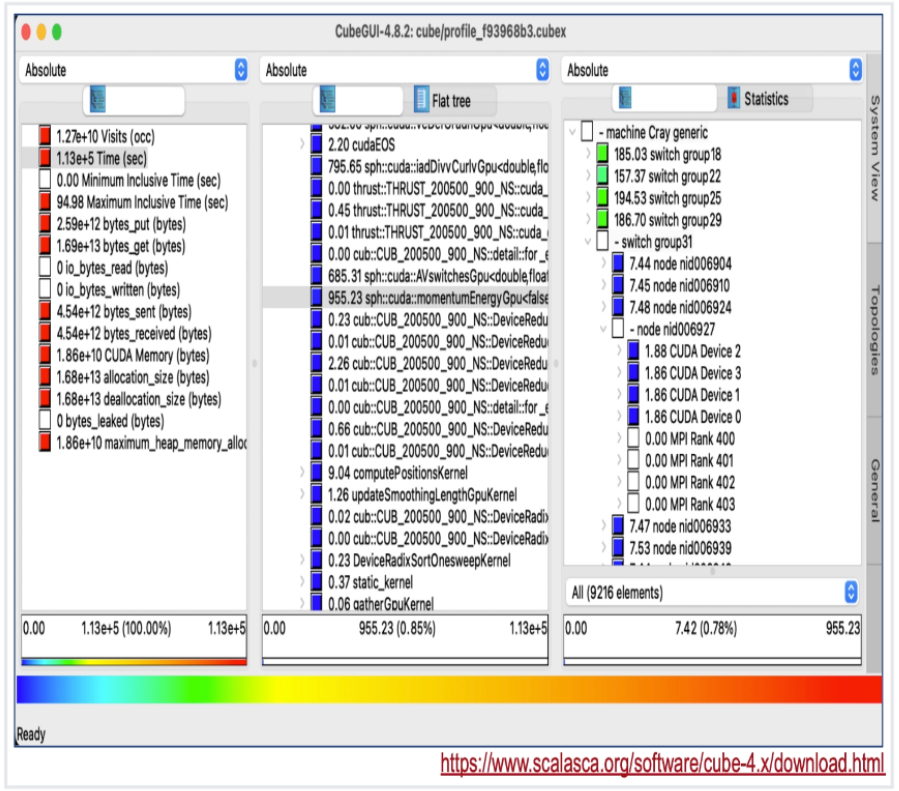
- Activate profiling with:

```
export SCOREP_ENABLE_PROFILING=true
```

- Run your job with srun or sbatch on Alps
- Copy the profile (`profile.cubex`) to your laptop
- Install Cube from scalasca.org

- Analyze the file with Cube:
  - performance metric (left panel)
  - call path (middle panel)
  - system resource (right panel)

```
/Applications/Cube/<...>/maccubegui.sh ./profile.cubex
```



https://www.scalasca.org/software/cube-4.x/download.html

Score-P: Tracing

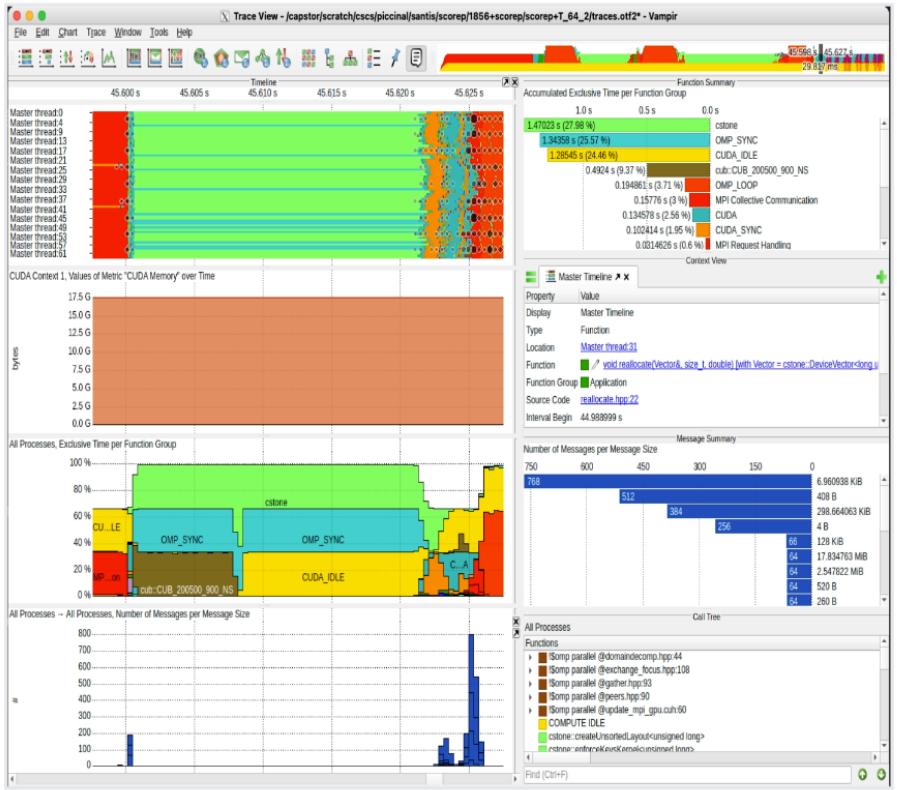
- Activate tracing with:

```
export SCOREP_ENABLE_TRACING=true
export SCOREP_FILTERING_FILE='initial_scorep.filter'
# more about this in the next slides
```

- Run your job with srun or sbatch on Alps

- Analyze the tracefile with `Vampir`:

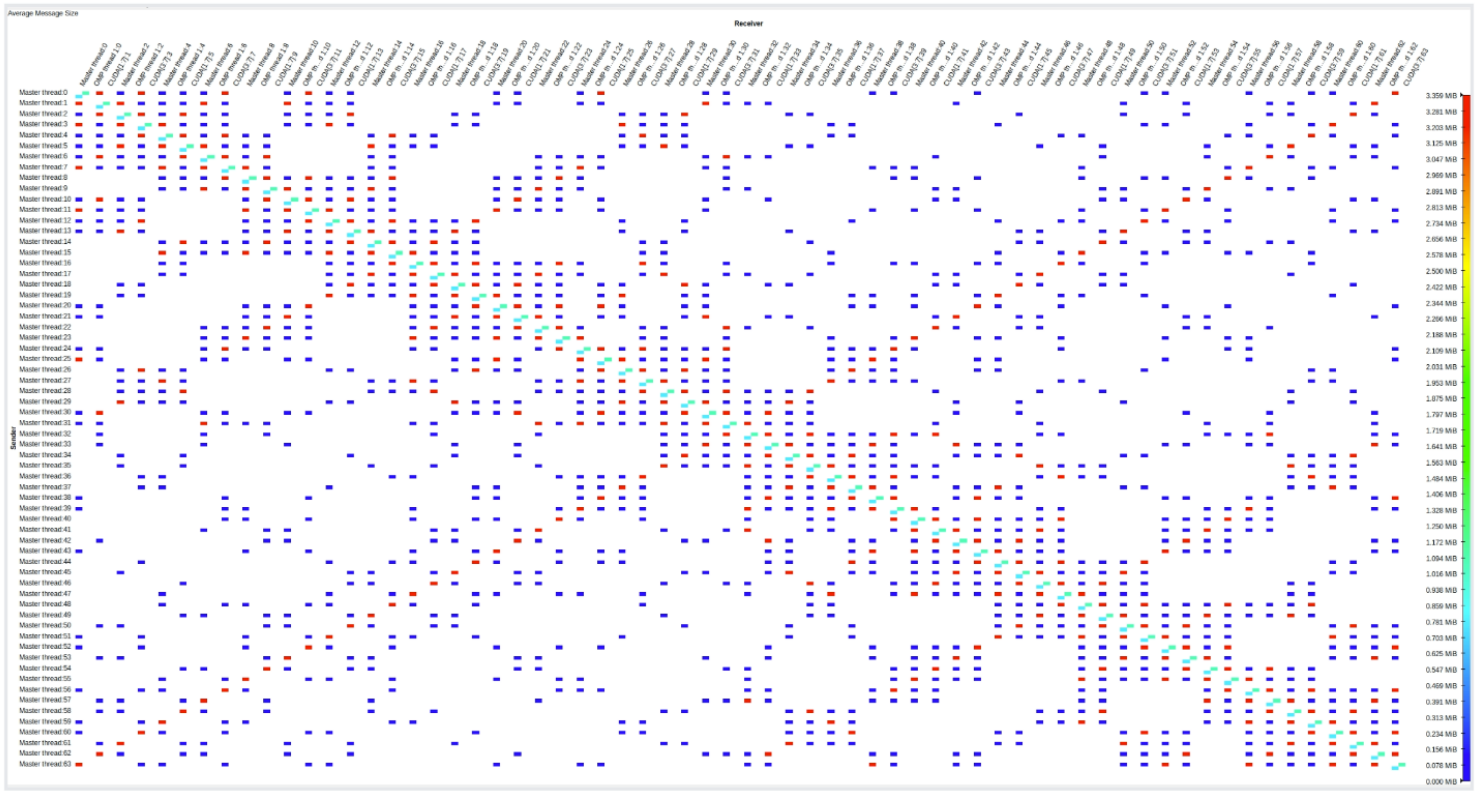
```
ssh -X eiger.cscs.ch  # gui requires x86_64 ⚠
ln -s /capstor/store/cscs/userlab/vampir/10.6.1/bin/vampir
./vampir ./traces.otf2
```



https://vampir.eu

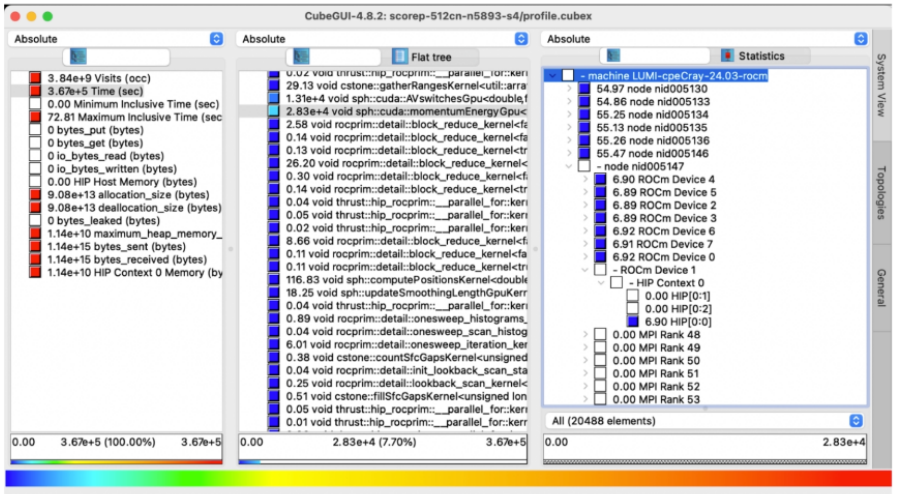
Score-P: Communication Matrix View

- It shows information about messages sent between processes and communication imbalances
- The rows represent the sending processes, the columns represent the receivers



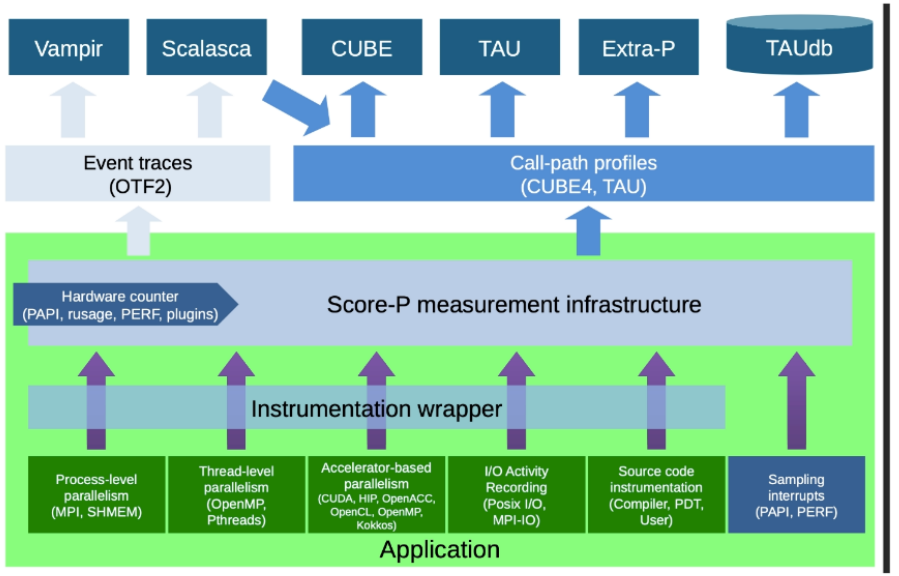
LUMI

Score-P supports AMD gpus too



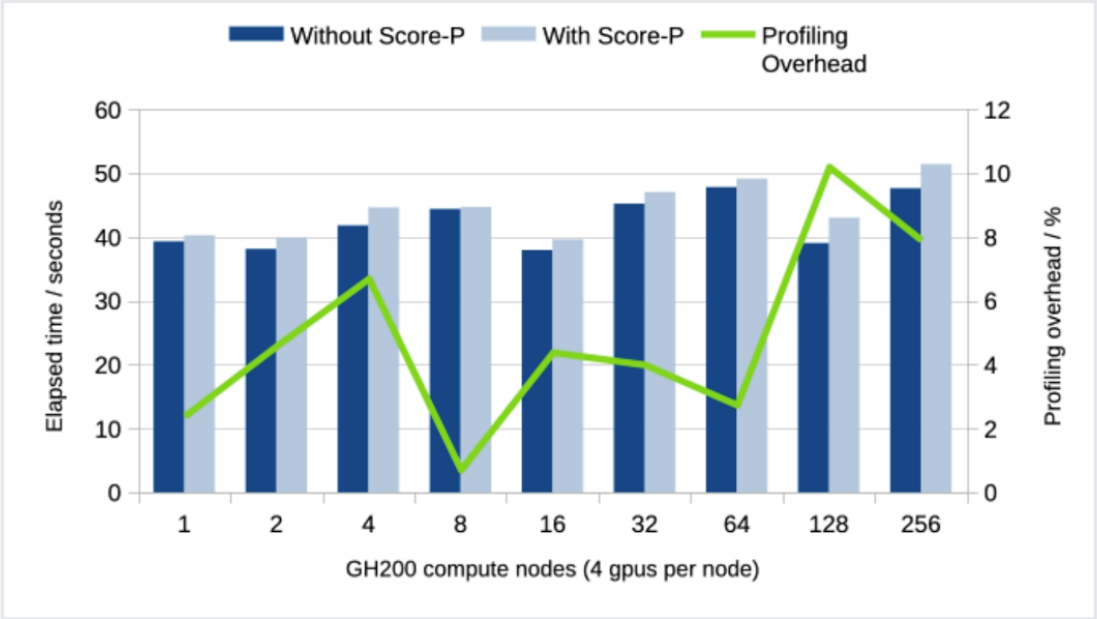
https://www.scalasca.org/software/cube-4.x/download.html

We have only scratched the surface, there is more



Score-P: Profiling Overhead

- Profiling will make your simulation run longer
- The overhead will vary between codes and job sizes (weak scaling results from 4 to 1024 gpus)



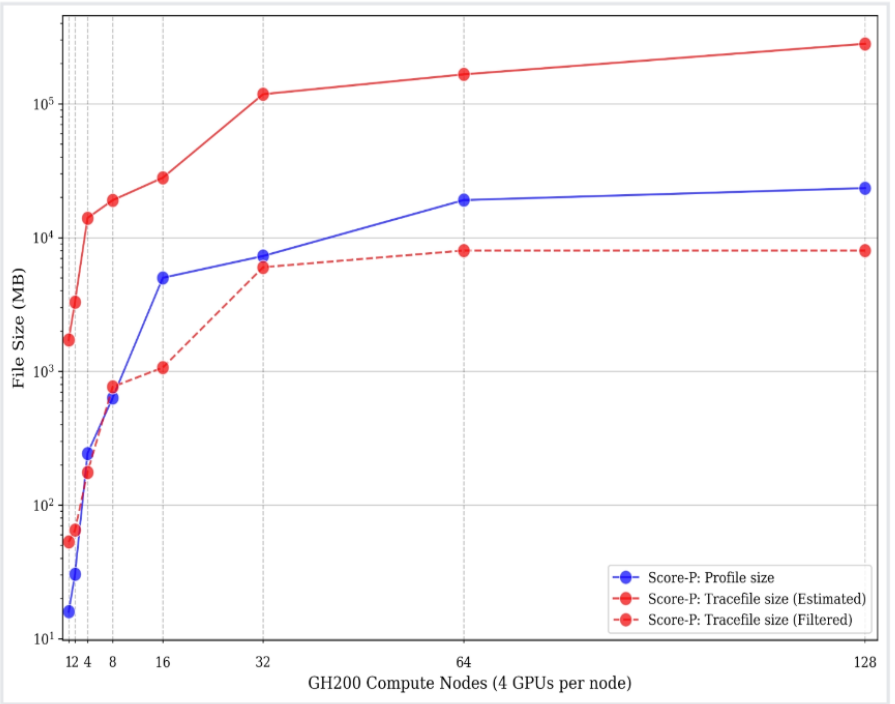
Adapt your jobscript for the additional compute time; the overhead is usually small (less than 10% here)

Score-P: Tracing Overhead

- Tracing will make your simulation run longer
- Tracing will also create larger files than profiling
- `scorep-score` allows to estimate the size of an OTF2 tracefile from a CUBE profile

```
> scorep-score profile.cubex

Estimated aggregate size of event trace: 3301MB <---
Estimated requirements for largest trace buffer: 413MB
Estimated memory requirements (SCOREP_TOTAL_MEMORY): 543MB
```

- `scorep-score -g` allows to reduce the overhead via filtering, for example:
  - from 280 GB to less than 10 GB with 512 gpus.
  - There are other ways to reduce overhead (API): `SCOREP_USER_REGION_BEGIN / END`




```
scorep-score -g profile.cubex # generate filter file
scorep-score -f initial_scorep.filter profile.cubex
export SCOREP_FILTERING_FILE='initial_scorep.filter'
```

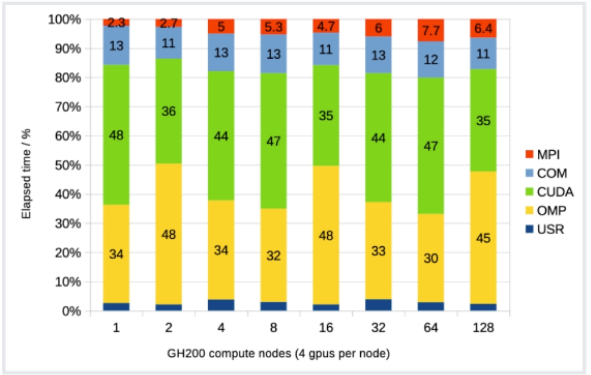
Score-P: Constraints and Workarounds

- The Vampir GUI is currently available only on x86-64 cpu based systems
  - Alps has aarch64 (Grace) CPU nodes
  - You can use Vampir on x86_64 CPU nodes (at CSCS or LUMI)
    - CSCS standard license allows to read trace files with up to 256 concurrent threads of execution
    - The Scalasca Trace Tool is your friend

Score-P: Interact with the tool

- Score-P provides tools to read profile and tracefiles from the cli
- scorep-score can report runtime distribution among function groups:

```
> scorep-score profile.cubex                    *******
flt     type  max_buf[B]      visits time[s] time[%] time/visit[us]  region
         ALL 432,816,718 132,510,706 1293.78   100.0           9.76  ALL
         USR 428,336,402 131,737,694   27.12     2.1           0.21  USR
         OMP   3,450,496     595,712  624.56    48.3        1048.42  OMP
        CUDA     812,380     121,828  465.62    36.0        3821.94  CUDA
         COM     144,144      44,308  140.95    10.9        3181.12  COM
         MPI      87,440      11,164   35.54     2.7        3183.21  MPI
```

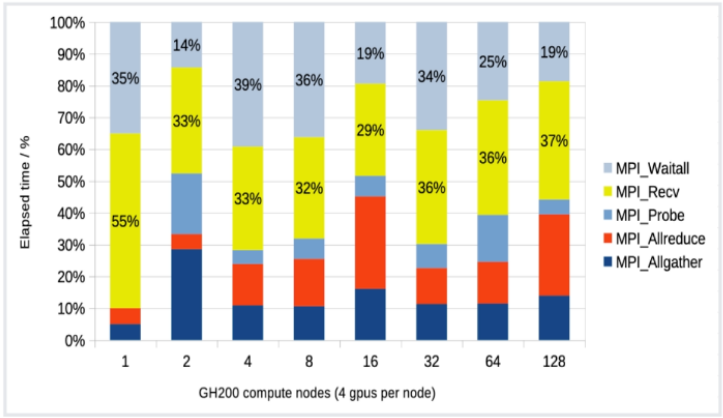


- where:
  - USR: all user functions (except those in the COM group)
  - OMP: all OpenMP regions/construct
  - CUDA: all CUDA API functions and kernels
  - MPI: all MPI functions
  - COM: all user functions that appear on a call-path to any functions from the above groups

Score-P: Interact with the tool

- scorep-score `-r` can report runtime distribution of every call in a function group, here MPI:

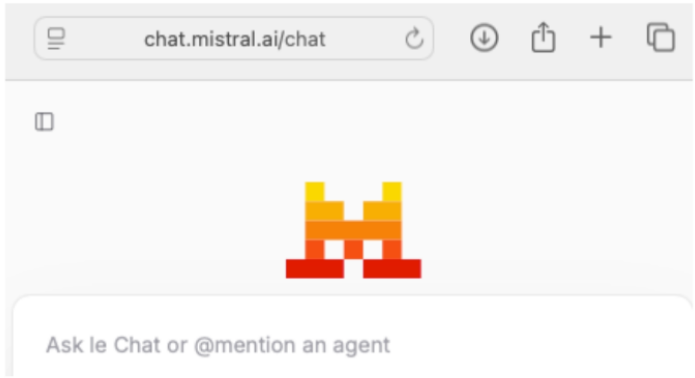
```
> scorep-score  -r  profile.cubex
                                    *******
type  max_buf[B]  visits  time[s] time[%] time/visit[us]  region
 MPI      89,548 751,552  1452.58     1.6        1932.77  MPI_Recv
 MPI      30,056 591,808   214.10     0.2         361.77  MPI_Probe
 MPI       2,522  49,664   718.21     0.8       14461.42  MPI_Waitall
 MPI       1,564  11,776  1032.04     1.1       87639.23  MPI_Allreduce
 MPI         272   2,048    63.17     0.1       30846.76  MPI_Reduce
 MPI          68     512   536.47     0.6     1047791.09  MPI_Allgather
 MPI          68     512    52.36     0.1      102263.07  MPI_Barrier
```



- `cube_calltree` can report aggregated values in a call tree for a given metric, here exclusive time:

```
> cube_calltree -m time -p -t 1 profile.cubex # add -i for inclusive time

0.368645 (0.06562%)         + int main(int, char**)
1.61967 (0.2883%)           |   + void sphexa::syncCoords(size_t, size_t, size_t, Vector&, ...
0.0135334 (0.002409%)       |   |   + void cstone::computeSfcKeys(const T*, const T*, const T*, KeyType*, ...
0.000199776 (3.556e-05%)    |   |   |   + !$omp parallel @sfc.hpp:286
66.0603 (11.759%)           |   |   |   |   + !$omp for @sfc.hpp:286
12.9532 (2.306%)            |   |   |   |   |   + !$omp implicit barrier @sfc.hpp:290
...
```

Questions?



References

- Virtual Institute for High Productivity Supercomputing: https://www.vi-hps.org
- Score-P: https://www.vi-hps.org/tools/score-p.html
- Cube: https://www.scalasca.org/scalasca/software
- Vampir: https://vampir.eu
- Score-P API: https://perftools.pages.jsc.fz-juelich.de/cicd/scorep/tags/scorep-9.0/html/group__SCOREP__User.html#gaab4b3ccc2b169320c1d3bf7fe19165f9
- Scalasca Trace Tool: https://apps.fz-juelich.de/scalasca/releases/scalasca/2.6/docs/manual/
- LUMI: https://lumi-supercomputer.github.io/LUMI-EasyBuild-docs/s/Score-P