

LUMI

A white wolf is the central focus, standing in a futuristic, blue-toned digital environment. The background is filled with vertical data streams, grid patterns, and glowing particles, creating a high-tech, cybernetic atmosphere. The wolf is looking slightly to the right of the camera.

Running Jobs on LUMI

Maciej Szpindler
LUMI User Support Team
Cyfronet

8 February 2024

Running Jobs

- Slurm intro
- Slurm partitions
- Interactive and batch jobs
- Job arrays
- Binding tasks to resources
- Running containers
- *Hands-on excercises*

Slurm intro

- **Slurm** is an open source cluster management and job scheduling system which provides:
 - exclusive and/or non-exclusive allocation of resources (compute nodes)
 - infrastructure for starting, executing, and monitoring jobs
 - fair share queue of pending jobs

Slurm version

Note Slurm version on LUMI is **22.05.10** (as of February 2024). Default documentation on the web is for versions 23.11. Please use specific version:

<https://slurm.schedmd.com/archive/slurm-22.05.10/>

Slurm partition

- Slurm partitions are groups of nodes with similar resources or associated limits
- Logical concept to manage access to LUMI HW partitions (GPU, CPU nodes)
- List of available partitions
 - `sinfo -s`
- Partition details
 - `scontrol show partition <partition-name>`
- *Different context of partition: hardware (name), access (Slurm directive), target architecture (environment module)*

Partitions (subject to change on 12 February 2024)



Partition name	Max walltime	Max jobs	Max resources/job	HW partition	
Slurm partitions allocatable by node (exclusively)					
standard-g	2 days	210 /200 running (105/100)	1024 (512) nodes	LUMI-G	} LUMI Consortium projects only
standard	2 days	120 /100 running (110/50)	512 (256) nodes	LUMI-C	
ju-standard-g	2 days	105/100 running	512 nodes	LUMI-G	} EuroHPC JU projects only
ju-standard	2 days	110/50 running	256 nodes	LUMI-C	
Slurm partitions allocatable by resources (shared)					
small-g	3 days	210 (200 running)	4 nodes	LUMI-G	
small	3 days	220 (200 running)	4 nodes	LUMI-C	
dev-g	3hours	2 (1 running)	32 nodes	LUMI-G	
debug	30 minutes	2 (1 running)	4 nodes	LUMI-C	
largemem	1 day	30 (20 running)	1 nodes	LUMI-D	

Project account

- Running jobs requires a project account
 - It is created when you are granted project allocation
 - Your project account ID is required to submit a job
 - Account ID has a name **project_XXXXXXXXXX** (9 digits)
- You can use the **lumi-allocations** command to list your projects
 - Alternative is to use the **groups** command to see account IDs
 - Your allocation portal should also show your project's account IDs

```
Data updated: 2023-09-19 11:31:15
```

Project	CPU (used/allocated)	GPU (used/allocated)	Storage (used/allocated)
project_465000688	0/10000000 (0.0%) core/hours	0/1000 (0.0%) gpu/hours	0/10 (0.0%) TB/hours

Allocation budget

- Do not use **sreport** command
 - It shows usage with different metrics than actual billing units
 - It creates unnecessary load on the Slurm controller

Slurm Job

- Job describes compute work unit and resources it is executed with
- Resources are nodes (including CPUs, GPUs and memory) and time
 - These are specified by either slurm command options or directives in a script
- Work unit is a user environment and a set of commands to execute
 - Posted as remote execute command or within a job script
 - Slurm controls remote execution via parallel launcher
 - Each parallel execute command results in a job step with multiple processes (tasks)

Parallel launcher

- **srun** is the only parallel launcher on LUMI
 - Starts parallel processes on (remote) compute nodes
 - returns the highest exit code of all tasks or the highest signal
 - there is no mpirun nor mpiexec commands available

Interactive jobs

- Using **salloc**
 - creates pool of resources reserved for interactive execution (job steps and its tasks)
 - the command will start a new shell **on the login node**
 - you can start parallel execution on the allocated nodes with **srun**
 - to obtain a shell on the first allocated compute node you can use **srun --pty**
 - the allocation can be terminated by exiting the shell with **exit**

```
~> salloc --nodes=2 --account=<project_id> --partition=<partition_name> --time=15
salloc: Granted job allocation 123456
salloc: Waiting for resource configuration

~> srun --ntasks=32 --cpus-per-task=8 ./mpi_openmp_application
~> exit
```

Interactive jobs

- Using **srun** directly (single job step)
 - Executes single job step on remote compute nodes
 - To start a shell on the first allocated node in a specific job/allocation use
`srun --interactive --pty --jobid=<jobid> $SHELL`
 - The `-w nid00XXXX` option selects a specific compute node
`srun --interactive --pty --jobid=<jobid> -w nid002217 ...`
 - Use `--overlap` option to share resources already allocated by your other job

```
~> srun --interactive --pty --jobid=<jobid> top
```

Batch jobs

- Batch jobs are submitted with **sbatch job.sh** command
 - **job.sh** file is your job script
 - Job script is regular shell script with **#SBATCH** directives and execute command(s)
- You can set Slurm job options with (in the order of precedence)
 - command line (sbatch options)
 - via environmental variables (**\$SBATCH_** and **\$SLURM_**)
 - Job script directives (lines preceded with **#SBATCH**)
- sbatch exits immediately after the Slurm controller gets the job script and assigns a job ID
- Slurm runs a single instance of the job script on the *first compute node* in the allocation
- Both standard output and error are directed to a file **slurm-<job_id>.out** by default

Batch job script

- Remember to include the sheebang in the first line of your job script
 - `#!/bin/bash` is recommended
 - Skipping the sheebang line or using fancy interpreters may result in module failures
- Directive line is **#SBATCH** followed by sbatch option and value
- Directives are interpreted until new line without **#SBATCH** pragma
- Remember environmental variables precedes job script directives (if you use them in shell profile/bashrc)
- You can define dependencies between batch jobs

How to list my jobs

- **squeue** command shows all current jobs (running and pending)
 - **--me** option alias shows only jobs owned by you
 - **--start** option gives estimation on actual start time of the job

Sbatch options

- **--time** Set a limit on the total run time of the job allocation
- **--account** Charge resources used by this job to specified project
- **--partition** Request a specific partition for the resource allocation
- **--job-name** Specify a name for the job allocation
- **--mail-user** Used to specify the email that should receive notification
- **--mail-type** When to send an email: BEGIN, END, FAIL, ALL
- **--nodes** Number of nodes to be allocated
- **--ntasks** Maximum number of tasks (MPI ranks)
- **--ntasks-per-node** Number of tasks per node
- **--cpus-per-task** Number of cores per tasks
- **--cpus-per-gpu** Number of CPUs per allocated GPU
- **--gpus** Total number of GPUs to be allocated for the job
- **--gpus-per-node** Number of GPUs per node
- **--gpus-per-task** Number of GPUs per task
- **--mem** Set the memory per node
- **--mem-per-cpu** Memory per allocated CPU cores
- **--mem-per-gpu** Memory per allocated GPU

Other Slurm options

- **--exclusive** the job is allocated all CPUs and GRES on all nodes in the allocation, but is only allocated as much memory as it requested
- **--mem=0** requests all the memory on a node (be careful, "all" but also "any")
- **--export** propagates environment variables from the submission environment to the launched application, **ALL** by default
- **--time** accepts time formats "minutes", "minutes:seconds", "hours:minutes:seconds", "days-hours", "days-hours:minutes" and "days-hours:minutes:seconds"
- **--reservation=<reservation_names>** allocates resources for the job from the named reservation
- **--dependency=<type:job_id[:job_id]>** defines the condition that the job with ID `job_id` must fulfil before the job which depends on it can start; type includes `after`, `afterany`, `afterok`, `afternotok`

Job vs step allocations

- Job allocation is the set of resources created with **sbatch** or **salloc**
- Step allocation is the **srun** (or other serial) command executed within an existing job allocation
- Step allocations usually inherit job options but can overwrite them (with possibly slightly different meaning), e.g.
 - **--exclusive** prevents other jobs from using the same node/dedicates separate cpus to multiple job steps in a single job
 - **--cpus-per-task** sets steps' specific cpu binding
 - **--exact** allows a step access to only the resources requested for the step

Example

```
~> salloc --partition=small-g --exclusive --nodes=1 --tasks=1 --gpus-per-node=1 --cpus-per-task=6 --time=5
~> srun gpu_check -l
MPI 000 - OMP 000 - HWT 001 (CCD0) - Node nid005066 - RT_GPU_ID 0 - GPU_ID 0 - Bus_ID d1(GCD4/CCD0)
MPI 000 - OMP 001 - HWT 002 (CCD0) - Node nid005066 - RT_GPU_ID 0 - GPU_ID 0 - Bus_ID d1(GCD4/CCD0)
MPI 000 - OMP 002 - HWT 003 (CCD0) - Node nid005066 - RT_GPU_ID 0 - GPU_ID 0 - Bus_ID d1(GCD4/CCD0)
MPI 000 - OMP 003 - HWT 004 (CCD0) - Node nid005066 - RT_GPU_ID 0 - GPU_ID 0 - Bus_ID d1(GCD4/CCD0)
MPI 000 - OMP 004 - HWT 005 (CCD0) - Node nid005066 - RT_GPU_ID 0 - GPU_ID 0 - Bus_ID d1(GCD4/CCD0)
MPI 000 - OMP 005 - HWT 006 (CCD0) - Node nid005066 - RT_GPU_ID 0 - GPU_ID 0 - Bus_ID d1(GCD4/CCD0)
```

Here your task runs on a single CCD and closest GPU but a job has entire node exclusively allocated

```
~> srun --partition=small-g --exclusive --nodes=1 --tasks=1 --gpus-per-node=1 --cpus-per-task=6 --time=5 \
gpu_check -l
```

Single step allocation may result with the same binding but if you are lucky; it depends on other jobs already sharing the node!

Automatic requeuing

- LUMI is using Slurm **automatic requeuing** of jobs upon **node** failure
 - your job is automatically resubmitted if any of it's nodes allocated fails
 - **identical job ID is used and the previous output truncated**
- Disable automatic requeuing with **--no-requeue** option
- Avoid your output file being truncated with **--open-mode=append** option
- Use the value of the **SLURM_RESTART_COUNT** variable
 - The value of this variable is 0 for first time the job is run
 - If the job has been restarted then the value is incremented

Generic Job script

```
#!/bin/bash -l
#SBATCH --job-name=examplejob      # Job name
#SBATCH --output=examplejob.o%j    # Name of stdout output file
#SBATCH --error=examplejob.e%j     # Name of stderr error file
#SBATCH --partition=standard       # Partition name, mandatory for job to be scheduled
#SBATCH --nodes=10                 # Total number of nodes
#SBATCH --ntasks=640               # Total number of mpi tasks
#SBATCH --mem=100G                  # Allocate 100 gigabytes memory per node
#SBATCH --time=1-12:00:00          # Run time (d-hh:mm:ss)
#SBATCH --mail-type=all            # Send email at begin and end of job
#SBATCH --account=project_<id>    # Project for billing, mandatory for job to be scheduled

# Any other commands must follow the #SBATCH directives

# Launch MPI parallel code with srun
srun ./your_application # Use srun instead of mpirun or mpiexec
```

Examples

```
~> sbatch --account=dummy-invalid-project job.sh
```

```
sbatch: error: Batch job submission failed: Invalid account or  
account/partition combination specified
```

Missing or
wrong
project id

```
~> sbatch job.sh
```

```
sbatch: error: Batch job submission failed: No partition  
specified or system default partition
```

Missing or
wrong
project id

```
#SBATCH --ntasks-per-node=8
```

```
#SBATCH --cpus-per-task=8
```

```
~> sbatch job.sh
```

```
sbatch: error: Batch job submission failed: Requested node  
configuration is not available
```

Not all 64
cores of a
node are
available to
users' jobs

Job scripts and modules

- You need to load modules to set specific environment for your job
 - Otherwise it is propagated from your current shell
 - Remember the sheebang line
- **LUMI** software stack modules
 - Use partition modules **partition/C/G/L** to set target architecture for the runtime environment
 - **partition/container** is a specific case
- **CrayEnv** (native) software environment
 - Use **craype-*** module family to set target architecture
- Hardcoding modules in the `bashrc` profile may cause troubles

Job arrays

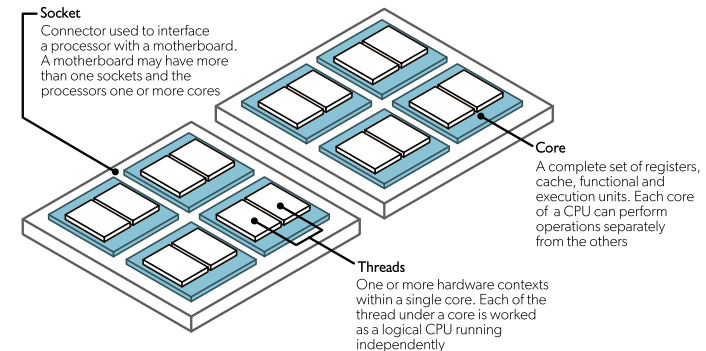
- Slurm job array submits a given number of independent jobs
- Use `--array` option to define the number of array tasks
- The `SLURM_ARRAY_TASK_ID` environment variable identifies each array task uniquely
- Job arrays use IDs of the form `<jobid>_<arrayindex>`
 - pending array task are shown as one entry with it's IDs combined
 - running ones are shown as individual jobs
- Job array is subject to limits same as single job (see partition's limits)

```
#!/bin/bash
#SBATCH --array=1-16
#SBATCH --output=array_%A_%a.out
#SBATCH --error=array_%A_%a.err
#SBATCH --time=01:00:00
#SBATCH --ntasks=1
#SBATCH --mem=4G

# Print the task index. echo "My SLURM_ARRAY_TASK_ID: " $SLURM_ARRAY_TASK_ID
srun ./myapp --input input_data_${SLURM_ARRAY_TASK_ID}.inp
```


Concept of socket, core and threads

- Compute nodes use Non-Uniform Memory Access (NUMA) design
 - 2 threads per core, 64 cores, 4 NUMA domains per socket
 - 8 cores per CCD, 2 CCDs per NUMA domain
 - 1 socket for LUMI-G nodes
 - 2 sockets for LUMI-C nodes
- Memory in the local NUMA node can be accessed faster
- Slurm can bind process or thread to a specific core
 - For improved memory access performance
 - It works only for exclusive node access,
 - Default for the standard and standard-g partitions



Slurm tasks

- Task is a single application process executing as a part of a job step
 - Single step can execute multiple tasks (e.g. MPI application parallel processes)
 - Each task can spawn its threads (with a hybrid MPI-OpenMP approach)
- For a job (step) with multiple tasks Slurm can
 - Distribute tasks across nodes, sockets, cores and multi-threads
 - Bind tasks to physical CPU cores or NUMA domains
 - Bind tasks to GPUs
 - Allocate multiple CPU cores for task's thread execution

Tasks distribution

- Slurm can use different policies to distribute tasks (MPI ranks, logical processes)
 - `--distribution=<dist>` option
 - `<dist>` can be subdivided in multiple levels for nodes, sockets and cores
 - Requires exclusive access
- Node level
 - **block (default)** distributes tasks to a node such that consecutive tasks share a node
 - **cyclic** consecutive tasks are distributed over consecutive nodes (in a round-robin fashion)
- Socket level
 - **block** consecutive tasks are distributed on the same socket
 - **cyclic (default)** tasks are distributed in a round-robin fashion across sockets
- Core level
 - inherits from second distribution method
- Combine distribution levels with semicolon, for instance `--distribution=block:block`

Multi-threading

- Hypertreads
 - `--hint=nomultithread` (default) Slurm option disables use of hyperthreads
 - Hardware threads are visible as cores 64-127 (LUMI-G) 128-255 (LUMI-C)
- Software multi-threading
 - OpenMP runtime controls thread affinity and pinning
 - Display binding with `OMP_DISPLAY_AFFINITY=TRUE` environmental variable
 - `OMP_PLACES` defines where to pin threads on, values: threads, cores, sockets
 - `OMP_PROC_BIND` defines how threads are mapped to the places
 - `spread` distributes (spread) the threads as evenly as possible
 - `close` binds threads close to the master thread
 - `master` binds threads to the same place as the master thread
 - `false` allows threads to be moved between places and disables thread affinity

Binding tasks to resources

- Requires exclusive access
- CPU binding (srun only) **--cpu-bind=<bind>**
 - **threads** tasks are pinned to the logical threads
 - **cores** tasks are pinned to the cores
 - **sockets** tasks are pinned to the sockets
 - **map_cpu:<list>** custom bindings of tasks with <list> a comma-separated list of CPUIDs
 - **mask_cpu:<list>** custom bindings of tasks with <list> a comma-separated hexadecimal values of mask for cores
- GPU binding with **--gpu-bind=<bind>**
 - **closest** binds each task to the closest GPU (can bind to multiple GPUs)
 - With exclusive allocation provides correct binding but prevents direct GPU to GPU communication (see slide on *Running with multiple GPUs*)
 - Other gpu-bind options are not working properly on LUMI
- Memory binding is also possible

Combining tasks and threads

- For a hybrid MPI+OpenMP jobs use **--cpus-per-task** option
 - Allocates multiple cores per task (MPI rank)
 - It still requires **OMP_NUM_THREADS** for explicit control

NOTE: Beginning with version 22.05, srun is not inheriting the --cpus-per-task value requested by salloc or sbatch. On LUMI the behavior is patched but in some cases still needs to be requested again with the call to srun or set with the **SRUN_CPUS_PER_TASK** environment variable if desired for the task(s)

Running with multiple GPUs

- Automatic GPU assignment
 - `--gpus-per-node`, `--gpu-bind=closest`
 - May assign multiple GPUs to a single task
- Explicit GPU mapping
 - Using ROCm environment variable `ROCR_VISIBLE_DEVICES` or `HIP_VISIBLE_DEVICES`
 - Custom `select_gpu` wrapper script
 - Caution with `--gpus-per-task` option, it can brake direct GPU communication

CAUTION: Slurm's `--gpu-bind=closest` provides correct assignment but brakes GPU IPC (direct communiation). If you need your MPI application to use direct communication between task's assigned GPUs then use proper CPU binding with GPU visibility masking (`ROCR_VISIBLE_DEVICES`).

GPU-aware MPI

- Only Cray MPICH provides GPU awareness on LUMI
 - You can possibly run some (old) OpenMPI versions on LUMI but without GPU awareness
- Enable device pointers (buffers) within MPI calls
 - Link application against GTL (GPU Transfer Library) library
 - Implicitly via modules: `craype-accel-amd-gfx90a` (CrayEnv) or `partition/G` (LUMI/x.y)
 - Explicitly with link option: `PE_MPICH_GTL_DIR_amd_gfx90a -lmpi_gtl_hsa`
 - On runtime with `MPICH_GPU_SUPPORT_ENABLED=1` environmental variable
- Map MPI tasks (ranks) to network interfaces
 - On runtime with `MPICH_OFI_NIC_POLICY=GPU`
 - No Slurm binding feature available

Example

See

<https://docs.csc.fi/apps/gromacs/#example-batch-script-for-lumi-full-gpu-node>

```
#!/bin/bash
#SBATCH --partition=standard-g
#SBATCH --account=<project>
#SBATCH --time=00:15:00
#SBATCH --nodes=1
#SBATCH --gpus-per-node=8
#SBATCH --ntasks-per-node=8
```

```
cat << EOF > select_gpu
#!/bin/bash
export ROCR_VISIBLE_DEVICES=${SLURM_LOCALID}
exec \${*}
EOF
chmod +x ./select_gpu
```

```
export MPICH_GPU_SUPPORT_ENABLED=1
export GMX_FORCE_GPU_AWARE_MPI=1
export GMX_ENABLE_DIRECT_GPU_COMM=1
```

```
module load GROMACS/2023.3-cpeAMD-22.12-HeFFTe-GPU
```

Assignment

```
export OMP_NUM_THREADS=7
CPU_BIND="mask_cpu:fe000000000000, fe00000000000000"
CPU_BIND="${CPU_BIND}, fe0000, fe000000"
CPU_BIND="${CPU_BIND}, fe, fe00"
CPU_BIND="${CPU_BIND}, fe00000000, fe0000000000"
```

App runtime

```
srunk \
--cpu-bind=${CPU_BIND} \
./select_gpu \
gmx_mpi mdrun \
-nb gpu -bonded gpu -pme gpu -update gpu \
<gromacs_opts>
```

Simultaneous job steps

- Slurm allows simultaneous job steps in a single allocation (job)
- Consider example from the srun manual

```
#!/bin/bash  
srun -n4 prog1 &  
srun -n3 prog2 &  
srun -n1 prog3 &  
srun -n1 prog4 &  
wait
```

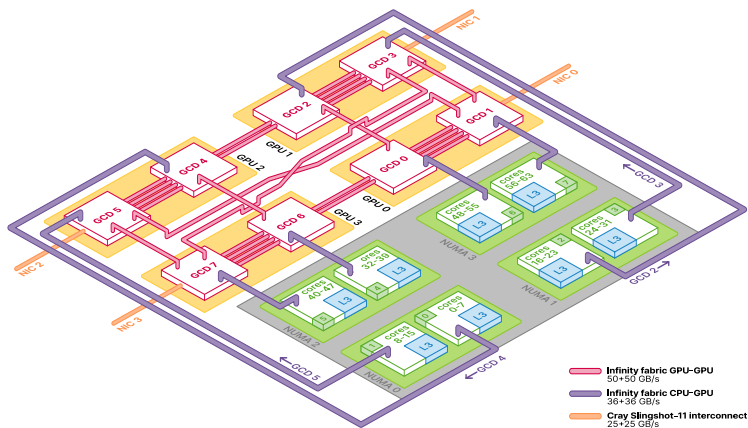
- Simultaneous steps may help to fully utilize standard partition nodes with jobs that not scale
- You may consider combining **--exclusive** sbatch and **--exact** srun options to distribute step tasks within the node(s)
- There are issues with simultaneous job steps on multi-node LUMI-C w/o job exclusive access (partition **small**)

Low-noise mode

- LUMI-G nodes have the *low-noise* mode activated
 - There are eight cores reserved (#0, 8, 16, 24, 32, 40, 48 and 56)
 - Only 56 cores are available for users' jobs
 - **Jobs requesting 64 cores/node will never run**
 - This eliminates OS jitter and allows symmetric NUMA task bindings
- Default core bindings may be sub-optimal
 - Thread team belonging to one task (MPI rank) may spread on multiple NUMA domains
 - Symmetric distribution requires 7 cores per GPU
 - Use custom binding with CPU masks to allow direct GPU to GPU communication
 - Works only with exclusive allocation (mind `small-g` and `dev-g` partitions)

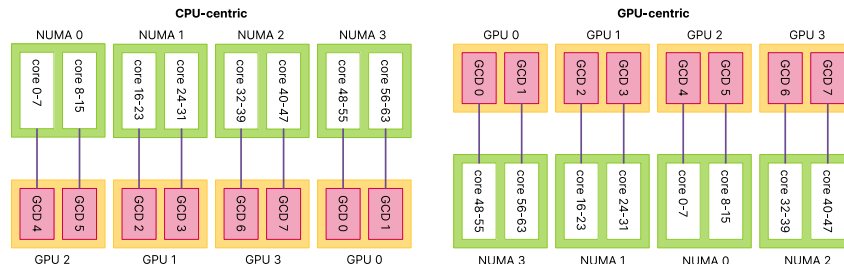
Adding GPUs to equation

- Note no direct correspondence between the NUMA region order and GPU numbering



```
~> rocm-smi --showtopo
```

```
GPU[0]      : (Topology) Numa Node: 3
GPU[1]      : (Topology) Numa Node: 3
GPU[2]      : (Topology) Numa Node: 1
GPU[3]      : (Topology) Numa Node: 1
GPU[4]      : (Topology) Numa Node: 0
GPU[5]      : (Topology) Numa Node: 0
GPU[6]      : (Topology) Numa Node: 2
GPU[7]      : (Topology) Numa Node: 2
```



Inspecting GPU and CPU bindings

- **lumi-CPEtools** module provides
 - **hybrid_check** tool program showing masks for CPU binding
 - **gpu_check** combines CPU binding with GPU assignment
 - See hands-on examples to experiment with these tools
- Taskset simple linux utility for checking cpu masks

Binding with gpu-bind

```
#!/bin/bash -l
#SBATCH --partition=standard-g
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=8
#SBATCH --cpus-per-task=7
#SBATCH --gpus-per-node=8
#SSBATCH --gpu-bind=closest
#SBATCH --time=5
#SBATCH --account=<project_account>

export SRUN_CPUS_PER_TASK=7
export OMP_NUM_THREADS=7
export OMP_PROC_BIND=close
export OMP_PLACES=cores

module load LUMI/23.09 partition/G lumi-CPETools
srun gpu_check -l
```

This will expose
single GPU to
each task
but no direct GPU
communication is
available

Use unless your
application relies on
direct GPU comm.

Binding with gpu-bind version 1

Not working

```
#!/bin/bash -l
#SBATCH --partition=standard-g
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=8
#SBATCH --cpus-per-task=7
#SBATCH --gpus-per-node=8
#SBATCH --gpu-bind="map_gpu:4,5,2,3,6,7,0,1"
#SBATCH --time=5
#SBATCH --account=<project_account>

GPU_BIND="map_gpu:4,5,2,3,6,7,0,1"
export SRUN_CPUS_PER_TASK=7
export OMP_NUM_THREADS=7
export OMP_PROC_BIND=close
export OMP_PLACES=cores
srun --gpu-bind=${GPU_BIND} gpu_check-1
```

Binding with `gpu-bind` and `cpu-bind`

```
#!/bin/bash -l
#SBATCH --partition=standard-g      # Partition (queue) name
#SBATCH --nodes=1                   # Total number of nodes
#SBATCH --ntasks-per-node=8        # 8 MPI ranks per node
#SBATCH --gpus-per-node=8           # Allocate one gpu / MPI rank
#SBATCH --time=5                     # Run time (d-hh:mm:ss)
#SBATCH --account=<project_account> # Project for billing

CPU_BIND="mask_cpu:0xfe,0xfe00,"
CPU_BIND="${CPU_BIND}0xfe0000,0xfe000000,"
CPU_BIND="${CPU_BIND}0xfe00000000,0xfe0000000000,"
CPU_BIND="${CPU_BIND}0xfe000000000000,0xfe00000000000000"

GPU_BIND="map_gpu:4,5,2,3,6,7,0,1"
```

```
export OMP_NUM_THREADS=7
export OMP_PROC_BIND=close
export OMP_PLACES=cores

export MPICH_GPU_SUPPORT_ENABLED=1

srun --cpu-bind=${CPU_BIND} --gpu-bind=${GPU_BIND}
    gpu_check
```

Not working

Binding with cpu-bind only

```
#!/bin/bash -l
#SBATCH --partition=standard-g
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=8
#SBATCH --gpus-per-node=8
#SBATCH --time=5
#SBATCH --account=<project_account>
```

```
CPU_BIND="mask_cpu:0xfe0000000000,0xfe00000000000000,"
```

```
CPU_BIND="${CPU_BIND}0xfe0000,0xfe000000,"
```

```
CPU_BIND="${CPU_BIND}0xfe,0xfe00,"
```

```
CPU_BIND="${CPU_BIND}0xfe00000000,0xfe0000000000"
```

```
export OMP_NUM_THREADS=7
```

```
export OMP_PROC_BIND=close
```

```
export OMP_PLACES=cores
```

```
module load LUMI/23.09 partition/G lumi-CPEtools
```

```
srun --cpu-bind=${CPU_BIND} gpu_check -l
```

This will expose all available GPUs to each task

Use if your application is able to manage GPUs internally

Binding with cpu-bind and wrapper

This will expose
again single GPUs
to each task

```
#!/bin/bash -l
#SBATCH --partition=standard-g
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=8
#SBATCH --gpus-per-node=8
#SBATCH --time=5
#SBATCH --account=<project_account>
```

```
cat << EOF > select_gpu
```

```
#!/bin/bash
```

```
export ROCR_VISIBLE_DEVICES=\${SLURM_LOCALID}
```

```
exec \${*}
```

```
EOF
```

```
chmod +x ./select_gpu
```

```
CPU_BIND="mask_cpu:0xfe000000000000,0xfe000000000000,"
CPU_BIND="${CPU_BIND}0xfe0000,0xfe000000,"
CPU_BIND="${CPU_BIND}0xfe,0xfe00,"
CPU_BIND="${CPU_BIND}0xfe00000000,0xfe0000000000"
```

```
export OMP_NUM_THREADS=7
```

```
export OMP_PROC_BIND=close
```

```
export OMP_PLACES=cores
```

```
module load LUMI/23.09 partition/G lumi-CPEtools
```

```
srunch --cpu-bind=${CPU_BIND} ./select_gpu gpu_check -l
```

Container jobs

- LUMI provides the **singularity** runtime included in the HPE Cray OS
 - No modules need to be loaded
 - No custom versions are supported
- No container build service is provided on LUMI currently
 - Bring your own container policy
 - Use native .sif file or Docker repository/registry
- You can run containers with `srun` directly

```
srun --partition=<partition> --account=<account_id> singularity exec ubuntu_21.04.sif \  
cat /etc/os-release
```

Running container from the DockerHub

- Docker container can be run without additional conversion
- Pulling container from the DockerHub

```
singularity pull docker://rocm/tensorflow-build:latest-focal-python3.8-rocm5.5.0
```

- Running the container in the interactive mode

```
srun --pty \  
  --ntasks=1 --gpus=8 --partition=dev-g \  
  --account=<account_id> --time=10 \  
singularity exec tensorflow-build_latest-focal-python3.8-rocm5.5.0.sif \  
rocm-smi --showtopo
```

Binding host files in the container

- LUMI filesystem (`/scratch` or `/project`) are not accessible from within the container
- They need to be explicitly bound by passing the `-B/--bind` command line option to the `singularity` command
- Simply binding `/scratch` or `/project` will not work
 - These paths are symlinks on LUMI, you must bind full paths to make them available in the container

Running MPI containers

- Using the host MPI (Cray MPICH)
 - Use specific partition/container architecture module and easybuild to generate helper modules to bind the Cray programming environment
 - There are a couple of base images for host MPI preinstalled on the system
- For containerized MPI, the **image must use MPICH ABI-compatible** version
 - Run with Slurm generic PMI mode

```
    srun --partition=<partition> --account=<account> --nodes=2 \  
        --mpi=pmi2 \  
        singularity run <mpi_container>.sif
```
 - No support for OpenMPI at this stage, although second approach may work for specific builds

LUMI

Maciej Szpindler

Cyfronet

Poland

m.szpindler@cyfronet.pl

Follow us

Twitter: @LUMIhpc

LinkedIn: LUMI supercomputer

YouTube: LUMI supercomputer

www.lumi-supercomputer.eu

contact@lumi-supercomputer.eu



EuroHPC
Joint Undertaking



The acquisition and operation of the EuroHPC supercomputer is funded jointly by the EuroHPC Joint Undertaking, through the European Union's Connecting Europe Facility and the Horizon 2020 research and innovation programme, as well as the of Participating States FI, BE, CH, CZ, DK, EE, IS, NO, PL, SE.

Leverage from
the EU
2014–2020



European Union
European Regional
Development Fund



Kainuun liitto

Understanding bitmasks

- Slurm uses hexadecimal masks to select which CPU cores tasks should bind to
 - Bits ordered right to left
 - First bit masks core #0
 - Each task need it's mask
- Single mask for 7 cores out of 8 (disabling core #0)
 - Core numbers: 76543210
 - Binary mask: 11111110
 - Hexadecimal value: 0xfe
- Slurm expression
 - Allocation (salloc/sbatch)
 - `--nodes=1 --ntasks-per-node=1 --partition=small-g --exclusive`
 - `--nodes=1 --ntasks-per-node=1 --partition=standard-g`
 - Execution (srun)
 - `--cpu-bind=mask_cpu:0xfe bash -c 'taskset -cp $$'`

More bitmasks

- More tasks to allocate full node symmetrically with 7 tasks per each CCD
 - First CCD:
 - Binary mask: 11111110 (8 bits, zero at first), hexadecimal value: **0xfe** (2 digits)
 - Second CCD:
 - 1111111000000000 (16 bits, zeros at first 9 bits), hexadecimal value: **0xfe00** (4 digits)
 - Third CCD:
 - 111111100000000000000000 (24 bits), hexadecimal value: **0xfe0000** (6 digits)
 - ...
- Complete masks
 - sbatch/salloc: **--ntasks-per-node=8 --exclusive**
 - srun: **--cpu-bind=mask_cpu:0xfe,0xfe00,** #cores 1-7, 9-15
0xfe0000,0xfe000000, #cores 17-23, 25-31
0xfe00000000,0xfe0000000000, #cores 33-39, 41-47
0xfe000000000000,0xfe00000000000000 #cores 49-55, 57-63

Another binding diagnose

- `hybrid_check` tool from the `lumi-CPEtools` module (LUMI Software Stack)

```
~> srun --cpus-per-task=7 --hint=nomultithread hybrid_check -r

++ hybrid_check: MPI rank 0/8 OpenMP thread 0/7 on cpu 1/128 of nid007916 mask 1-7
++ hybrid_check: MPI rank 0/8 OpenMP thread 1/7 on cpu 2/128 of nid007916 mask 1-7
++ hybrid_check: MPI rank 0/8 OpenMP thread 2/7 on cpu 3/128 of nid007916 mask 1-7
++ hybrid_check: MPI rank 0/8 OpenMP thread 3/7 on cpu 4/128 of nid007916 mask 1-7
++ hybrid_check: MPI rank 0/8 OpenMP thread 4/7 on cpu 5/128 of nid007916 mask 1-7
++ hybrid_check: MPI rank 0/8 OpenMP thread 5/7 on cpu 6/128 of nid007916 mask 1-7
++ hybrid_check: MPI rank 0/8 OpenMP thread 6/7 on cpu 7/128 of nid007916 mask 1-7
++ hybrid_check: MPI rank 1/8 OpenMP thread 0/7 on cpu 9/128 of nid007916 mask 9-15
++ hybrid_check: MPI rank 1/8 OpenMP thread 1/7 on cpu 10/128 of nid007916 mask 9-15
++ hybrid_check: MPI rank 1/8 OpenMP thread 2/7 on cpu 11/128 of nid007916 mask 9-15
++ hybrid_check: MPI rank 1/8 OpenMP thread 3/7 on cpu 12/128 of nid007916 mask 9-15
++ hybrid_check: MPI rank 1/8 OpenMP thread 4/7 on cpu 13/128 of nid007916 mask 9-15
++ hybrid_check: MPI rank 1/8 OpenMP thread 5/7 on cpu 14/128 of nid007916 mask 9-15
++ hybrid_check: MPI rank 1/8 OpenMP thread 6/7 on cpu 15/128 of nid007916 mask 9-15
```