Running Jobs on LUMI

Maciej Szpindler
LUMI User Support Team
Cyfronet

LUMI

Running Jobs

- Slurm intro
- Slurm partitions
- Interactive jobs
- Batch jobs
- Job arrays
- Running containers
- GPU/CPU/thread binding, NUMA
- *Hands on*

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 **22.05.8** (as of May 2023). Default documentation on the web is for versions 23.02. Please use specific version:

https://slurm.schedmd.com/archive/slurm-22.05.8/

Slurm partition

- Slurm partitions are (possibly overlapping) groups of nodes with similar resources or associated limits
- Logical concept to manage access to LUMI HW partitions (GPU, CPU nodes)
- Different context of partition: hardware, access, target architecture
- List of available partitions

```
sinfo -s
```

- Partition details

```
scontrol show partition <partition-name>
```

Available partitions

Partition name	Max walltime	Max jobs	Max resources/job	HW partition
Slurm partitions allocatable by node (exclusively)				
standard-g	2 days	210 (200 running)	1024 nodes	LUMI-G
standard	2 days	120 (100 running)	512 nodes	LUMI-C
bench	1 day	n/a	All node	LUMI-C
Slurm partitions allocatable by resources (shared)				
dev-g	6 hours	2 (1 running)	16 nodes	LUMI-G
small-g	3 days	210 (200 running)	4 nodes	LUMI-G
small	3 days	220 (200 running)	4 nodes	LUMI-C
debug	30 minutes	2 (1 running)	4 nodes	LUMI-C
largemem	1 day	30 (20 running)	1 nodes	LUMI-D

Fairness

- The Slurm partition setup of LUMI prioritizes jobs that aim to scale out
  - most nodes are reserved for jobs that use them exclusively (standard partitions)
- Your job (or allocation request) is queued until resource time-window is available
- Examine the queue of jobs

  `squeue`
  - **`--me`** option is an alias for list of your jobs
  - **`--start`** shows when your pending job will start
- Factors that decides on your job's queue priority

  `sprio`
  - **`Fairshare`** is a factor responsible for a "fair" access to all users/accounts

Project account

- Running jobs requires a project account
  - It is created when you are granted project allocation
  - You need to specify your project account ID in your job script (or with the command option)
  - This is mandatory
  - Account ID has a name **`project_xxxxxxxxx`** (9 digits)
- You can use the **`lumi-allocations`** command to list the projects of which you are a member.
  - Alternative is to use the `groups` command to see account IDs
  - Your allocation portal should also show your project's account IDs

Interactive jobs

- Using **`salloc`**
  - creates pool of resources reserved for your interactive execution (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
  - You can execute single parallel task with **`srun`** command
  - 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 used by your other job step (task)

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

Job launcher

- **`srun`** is the only parallel launcher on LUMI
  - there is no mpirun nor mpiexec commands
  - returns the highest exit code of all tasks or the highest signal

Batch jobs

- Batch jobs are submitted with **`sbatch job.sh`** command
  - File job.sh is your job script
  - Job script is regular shell script with **`#SBATCH`** directives and execute command
  - You can use Slurm options with directives , from the command line or via environmental variables
- sbatch exits immediately after the script is successfully transferred to the Slurm controller and assigned a Slurm job ID
- Slurm runs a single copy of the batch script on the first node in the set of allocated nodes
- 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
  - #SBATCH" directive lines before any executable commands
  - #SBATCH directives are interpreted once the first non-comment non-whitespace line is reached
- Command line options overrides any environment variables and environment variables overrides any options set in a batch job script
- You can enable e-mail notifications in the job script
- You can define dependencies between batch jobs

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
- **`--export`** propagates environment variables from the submission environment to the launched application, **`ALL`** by default
- **`--time`** accepts time formats include "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

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 (queue) name
#SBATCH --nodes=10                     # Total number of nodes
#SBATCH --ntasks=640                   # Total number of mpi tasks
#SBATCH --mem=0                        # Allocate all the memory on the 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
#SBATCH --mail-user=username@domain.com
# Any other commands must follow the #SBATCH directives

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

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
- With the **`LUMI`** software stack
  - Use partition modules **`partition/C/G/L`** to choose target architecture for the application or the library
  - Do not confuse with Slurm **`--partition`** selection
- 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
```

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
  - Docker container can be run without manual conversion
  - 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 registry

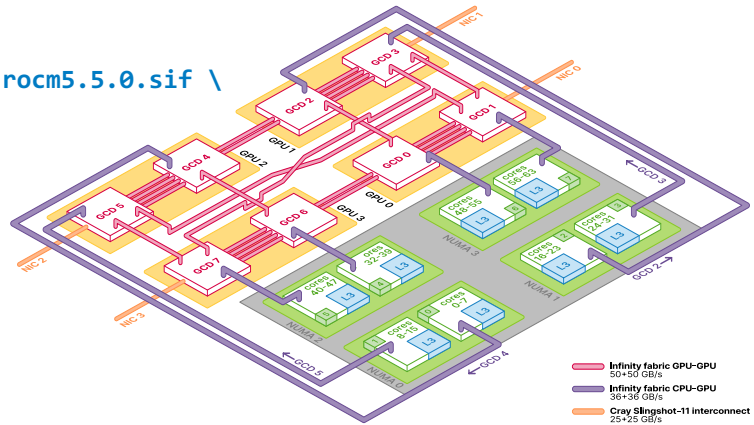
- Pulling container from the DockerHub

```
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
```

```
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
```



Binding file systems 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 containers in parallel

- For MPI containers, the image must use MPICH ABI-compatible MPI version
- Using the host MPI
  - Install singularity bindings from the LUMI Software Stack

    ```
    module load LUMI partition/<lumi-partition> EasyBuild-user
    eb singularity-bindings-system-cpeGNU-<toolchain-version>.eb -r
    ```

  - Run the container with the specific environment

    ```
    module load singularity-bindings
    srun --partition=<partition> --account=<account> --nodes=2 singularity run <mpi_container>.sif
    ```

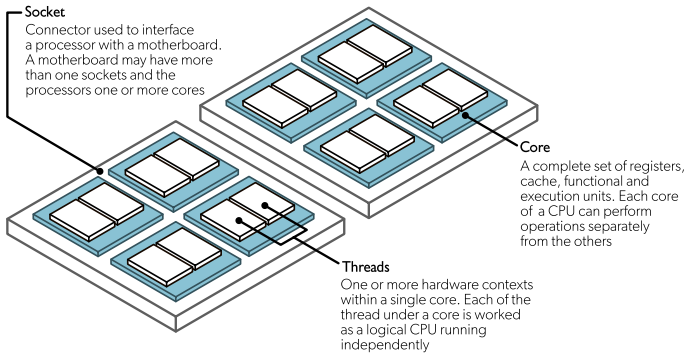
- Using the container MPI
  - Run with Slurm generic PMI mode

    ```
    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

Concept of socket, core and threads

- LUMI compute nodes use Non-Uniform Memory Access design
  - 2 threads per core, 64 cores, 4 NUMA domains per socket
  - 1 socket for LUMI-G nodes
  - 2 sockets for LUMI-C nodes
- Memory in the local NUMA node can be accessed faster
- Binding of a process or thread to a specific core can improve the performance by increasing memory locality
- Binding only makes sense for exclusive node access, this is the default for the `standard` and `standard-g` partitions



Tasks distribution

- Slurm can use different policies to distribute tasks (MPI ranks)
  - **`--distribution=<dist>`** srun option sets the policy
  - **`<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

- Hyperthreads
  - Control hardware multithreading with **`--hint=nomultithread`** (default) Slurm option
  - Hardware threads are visible as cores 64-127 (LUMI-G) 128-255 (LUMI-C)
- Software multi-threading
  - OpenMP provides control over a thread affinity
  - Display binding with **`OMP_DISPLAY_AFFINITY=TRUE`** environmental variable
  - Use **`OMP_PLACES`** to define where the threads should be pinned on with values threads, cores, sockets
  - Use **`OMP_PROC_BIND`** to define 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

- Slurm can bind tasks to specific 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>`**
  - **map_gpu:<list>** custom bindings of tasks with <list> a comma-separated list of GPUIDs
  - **mask_gpu:<list>** custom bindings of tasks with <list> a comma-separated hexadecimal values of mask for GPUs
- Memory binding is also possible

Combining tasks and threads

- For a hybrid MPI+OpenMP jobs use --cpus-per-task srun option
  - Allocates multiple cores per process (MPI rank)
  - Allows spawned threads bind to allocated cores
  - It still requires **OMP_NUM_THREADS** for explicit control
  - ***NOTE***: Beginning with 22.05, srun will not inherit the --cpus-per-task value requested by salloc or sbatch. It must be requested again with the call to srun or set with the **SRUN_CPUS_PER_TASK** environment variable if desired for the task(s).
- Inspect actual task/thread affinity with
  - **MPICH_CPUMASK_DISPLAY=1**

Multi GPU runs

- GPU mapping (automatic assignment)
  - Slurm GPU binding
  - Slurm CPU mapping
- GPU masking (explicit mapping)
  - Using ROCm environment variable **`ROCR_VISIBLE_DEVICES`**
  - Custom **`select_gpu`** wrapper script
- GPU-aware MPI
  - Turn on with **`MPICH_GPU_SUPPORT_ENABLED=1`** MPI variable
  - Allows to use device pointers (buffers)
  - Map tasks to network interfaces with **`MPICH_OFI_NIC_POLICY=GPU`**

Low-noise mode

- LUMI-G nodes have the *low-noise* mode activated
  - One core (`#0`) is restricted for the operating system
  - Only 63 cores are available to the jobs
  - Jobs requesting 64 cores/node will never run
- 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
  - Works only with exclusive allocation (mind `small-g` and `dev-g` partitions)

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: `1111111`0
  - 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
```

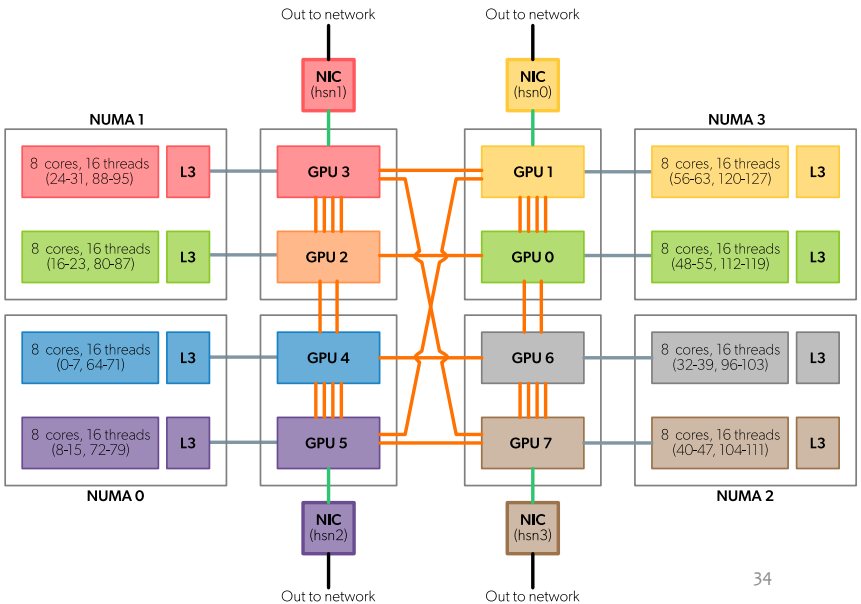
Inspecting binding with MPI

- Use specific **`MPICH_CPUMASK_DISPLAY=1`** variable to print actual bitmask for MPI ranks

```
[PE_0]: cpumask set to 7 cpus on nid005301, cpumask = 00000000111111100000000000000000000000000000000000000000000000
[PE_1]: cpumask set to 7 cpus on nid005301, cpumask = 11111110000000000000000000000000000000000000000000000000000000
[PE_2]: cpumask set to 7 cpus on nid005301, cpumask = 00000000000000000000000000000000000001111111000000000000000000
[PE_3]: cpumask set to 7 cpus on nid005301, cpumask = 00000000000000000000000000001111111000000000000000000000000000
[PE_4]: cpumask set to 7 cpus on nid005301, cpumask = 00000000000000000000000000000000000000000000000000011111110
[PE_5]: cpumask set to 7 cpus on nid005301, cpumask = 00000000000000000000000000000000000000000000011111110000000
[PE_6]: cpumask set to 7 cpus on nid005301, cpumask = 00000000000000000000001111111000000000000000000000000000000000
[PE_7]: cpumask set to 7 cpus on nid005301, cpumask = 00000000000000011111110000000000000000000000000000000000000000
```

Adding GPUs to equation

- Note no direct correspondence between the the NUMA region order and GPU numbering
  - Recall `rocm-smi` topology output
- CPU-centric approach (1 task/GPU)
  - Use masks from previous slide
  - Use **`select_gpu`** wrapper
  - Or try **--gpu-bind=map_gpu:<map>**
- GPU-centric approach
  - Reorder task cpu masking



Complete script (CPU centric binding)

```
#!/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} \
./hello_jobstep/hello_jobstep
```

This will expose single GPU to one task

Complete script (GPU centric binding)

```
#!/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:0xfe000000000000,0xfe00000000000000,"
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

export MPICH_GPU_SUPPORT_ENABLED=1

srun --cpu-bind=${CPU_BIND} ./hello_jobstep/hello_jobstep
```

This will expose all GPUs to every task

Complete script (using wrapper)

```
#!/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

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

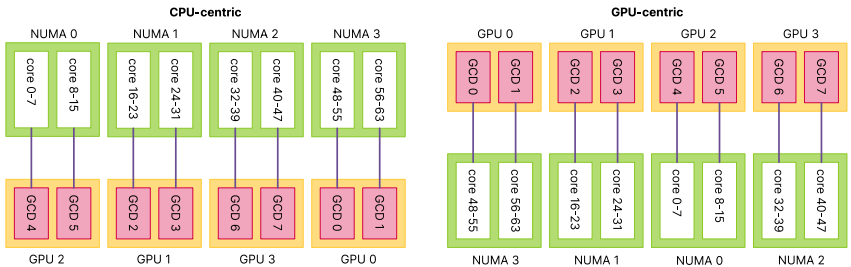
```
CPU_BIND="mask_cpu:0xfe000000000000,0xfe00000000000000,"
CPU_BIND="${CPU_BIND}0xfe0000,0xfe000000,"
CPU_BIND="${CPU_BIND}0xfe,0xfe00,"
CPU_BIND="${CPU_BIND}0xfe00000000,0xfe0000000000"

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

export MPICH_GPU_SUPPORT_ENABLED=1

srun --cpu-bind=${CPU_BIND} ./select_gpu \
 ./hello_jobstep/hello_jobstep
```

This will expose again single GPUs to each task

More advanced wrapper

- You can also start with CPU centric binding and refine wrapper to reorder GPUs visibility

```
#!/bin/bash

GPUSID="4 5 2 3 6 7 0 1"

GPUSID=(${GPUSID})

if [ ${#GPUSID[@]} -gt 0 -a -n "${SLURM_NTASKS_PER_NODE} ]; then

if [ ${#GPUSID[@]} -gt $SLURM_NTASKS_PER_NODE ]; then

   export ROCR_VISIBLE_DEVICES=${GPUSID[$(($SLURM_LOCALID))]}

 else

   export ROCR_VISIBLE_DEVICES=${GPUSID[$((SLURM_LOCALID /
   ($SLURM_NTASKS_PER_NODE / ${#GPUSID[@]})))]}

 fi

fi

exec $*
```



Know issues

- Identifying optimal task binding for a multi GPU performance is complex
- Support for heterogeneous jobs in Slurm is currently broken
  - You cannot execute mixed CPU/GPU jobs

```
sbatch --partition=standard-g : --partition=standard job.sh
```

  - Regular MPMD jobs should still work with **--multi-prog** option

Large-scale runs

- Opportunity to perform runs on the entirety of LUMI
- Provide at most a 1 page description of what you are intending to do
- Deadline for the application is Wednesday 11 days before the last Sunday every month
- Access on the last Sunday every month (subject to change over time)
- Resource usage for any runs during this window will be billed as usual
- The applications are submitted via the Helpdesk, using contact form with the *"large-scale runs"* category