Process and thread distribution and binding

Kurt Lust
LUMI User Support Team (LUST)
University of Antwerp

Evolving version, last update December 2024

What are we talking about?

- Distribute processes and threads across the available resources for the job
- and bind them to the resources to ensure they stay there and only use the assigned resources
  - Across nodes: Only distribution
  - Within a node: Binding necessary
- System software level (Linux/ROCm/Slurm):
  - Control groups used at the job and job step level, sometimes at the task level
  - Affinity mask to control where a thread can get scheduled
  - ROCm runtime also has a mechanism to control access to GPUs
- Tools for verification in the lumi-CPEtools modules

When/where is it done?

- Slurm level
  - Creation of allocation: Slurm reserves resources at the node level using control groups
  - Creation of job step:
    - Distributes tasks across nodes and cores/hardware threads on nodes
    - Default in most cases: Binds tasks to CPUs (affinity mask) and GPUs (control groups unfortunately)
- Application library level
  - Cray MPICH can renumber the ranks
  - OpenMP runtime: select number of CPU threads and thread binning within the resources of a task using affinity masks
  - ROCm runtime: Select GPUs using `ROCR_VISIBLE_DEVICES`
- Does not always make sense on nodes that are not job-exlusive!

Why do I need this?

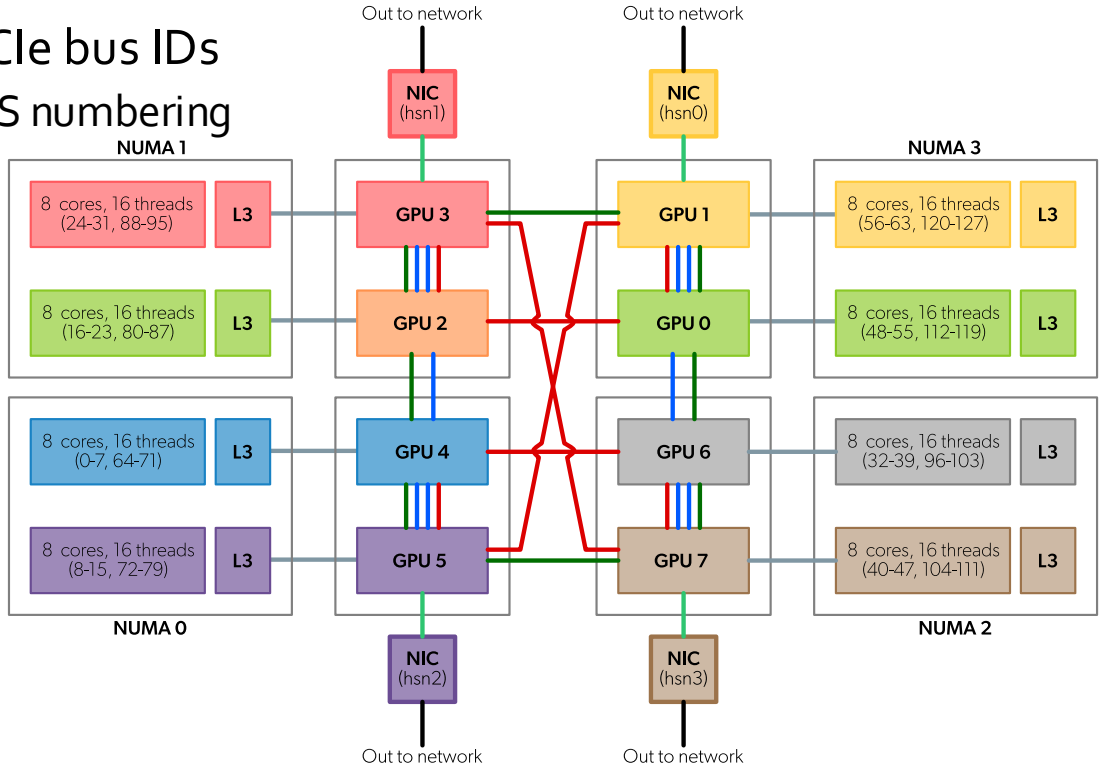
- Importance of memory locality at all levels (cache and main memory)
  - E.g.: MPI application with 14 GB/rank so 16 ranks on node: Spread out across CCDs…
  - Shared memory with lack of memory locality: Maybe need to bundle threads if the application fits in a socket
  - No solution that's always optimal!
- Short connection between CPU and GPU sometimes essential for fast communication between both
  - Cache-coherent accesses to GPU memory by the CPU
- Mapping of MPI ranks to reduce inter-node traffic and maximise intra-node traffic which is much faster
  - Also on the GPU: Map communication pattern on the topology of a node

Core numbering

- Linux core (actually hardware thread/virtual core) numbering does not reflect the hierarchy
  - Numbers 0-127 on LUMI-C are the first hardware thread on each physical core, 128-255 then the second one, so *i* and *i+128* map onto the same physical core
  - On LUMI-G: Core 0-63 first hardware thread, core 64-127 second, so *i* and *i+64* map onto the same physical core
- Hardware threading on LUMI is turned on when booting a node
  - Slurm does not turn hardware threading off, but doesn't include the second hardware thread in the affinity mask when multithreading is off
  - Slurm only does so at the regular job step level
  - The Slurm batch step will always see both hardware threads for each core!
- Technical discussion in the notes if you're interested

GPU Numbering (1)

- Very tricky
- Numbering based on the PCIe bus IDs
  - Global numbering or bare-OS numbering



GPU Numbering (2)

- Very tricky
- Numbering based on the PCIe bus IDs
  - *Global numbering* or *bare-OS numbering* (0-7)
- Job-level control group
  - New numbering starting from 0: *job-local numbering*
  - Same order though
- Task-level control group
  - Yet another numbering starting from 0: *task-local numbering*
  - And a headache for MPI and RCCL applications
- Further restricting access via `ROCR_VISIBLE_DEVICES` will start yet another numbering in, e.g., the HIP runtime

GPU Numbering - Remarks

- Very technical demonstrations in the notes
- Slurm works differently with CPUs and GPUs on LUMI
  - CPUs: Control groups at the job level, after that affinity masks
  - GPUs: Control groups at the job and task level, even though `ROCR_VISIBLE_DEVICES` plays a bit the role of an affinity mask
- Affinity masks work differently from `ROCR_VISIBLE_DEVICES`
  - Affinity masks always refer to the global / bare OS numbering of the hardware threads
  - `ROCR_VISIBLE_DEVICES` numbering is based on the local numbering in the context where the variable is used
  - Affinity masks can only shrink as you go deeper in a hierarchy
  - `ROCR_VISIBILE_DEVICES`, being just an environment variable, can be abused to gain access to extra resources (within the confines of the control group)

Task distribution with Slurm (1)

- `srun --distribution={block|cyclic|plane=<s>}[:{block|cyclic|fcyclic}[,{Pack|NoPack}]`
- Level 1: Distribution of tasks across nodes
  - `block`: Fill first node in allocation, then fill second, etc.
    - `Pack`: Fill completely before moving to the next node
    - `NoPack`: More ballanced, trying to fill all nodes as equally as possible
  - `cyclic`: First assign one task to each node, then from the first node again assign a second task, …
  - `plane=<s>`: As cyclic, but assigning `s` tasks at a time before moving on
  - More options that we do not discuss

Task distribution with Slurm (2)

- `srun --distribution={block|cyclic|plane=<s>}[:{block|cyclic|fcyclic}[,{Pack|NoPack}]`
- Level 2: Distribution of tasks across cores
  - `block`: Consecutive sets of cores for each task
  - `cyclic`: First assign one task to each socket on the first set of consecutive cores/virtual cores of each socket, then assign a second task on each socket on the next set of cores, …
  - `fcyclic`: Will spread tasks out across sockets
    - Not clear where this is useful on an AMD system except for cases with one task per node and a lot of memory for that task
- Level 3 not shown in this simplified version
- Default: `block:block:nopack` but `block:*` results in `block:cyclic`
- L2 and L3 distribution conflicts with the CPU binding mechanism that we will discuss
  - But usefull with `--cpus-per-task`

Task-to-CPU binding with Slurm

- Works with affinity masks
- `srun --cpu-bind=[{quiet|verbose},]<type>`
- Some `<type>` options are for automatic binding
  - `--cpu-bind=threads` is the default behaviour on LUMI
  - Other options: See the manual
- Other `<type>` options define a list of slots to be used
  - Combination with `--distribution` L2/L3 options does not make sense
  - `--cpu-bind=map_cpu:<cpu_id_for_task_0>,<cpu_id_for_task_1>,...` : Specify a single hardware thread for each task on the node
    - For MPI programs
  - `--cpu-bind=mask_cpu:<mask_for_task_0>,<mask_for_task_1>,...` : Specify afinity mask for each task on the node.
    - For OpenMP or hybrid programs

Task-to-CPU binding with Slurm: Masks

- 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 its mask
- Single mask for 7 cores out of 8 (disabling core `#0`)
  - Core numbers: `76543210`
  - Binary mask: `1111111`0
  - Hexadecimal value: `0xfe`
  - Leading zeros can be omitted, but each element can still be very long

Task-to-CPU binding with Slurm: Examples

- `salloc --nodes=1 --partition=standard-g`
  `module load LUMI/24.03 partition/G lumi-CPEtools/1.1-cpeGNU-24.03`
  `srun --ntasks=8 --cpu-bind=map_cpu:49,57,17,25,1,9,33,41 mpi_check -r`
  - Example will be relevant for LUMI-G
- `srun --ntasks=8 --cpu-bind=mask_cpu:\`
  `7e000000000000,7e00000000000000,7e0000,7e000000,7e,7e00,7e00000000,7e0000000000 \`
  `hybrid_check -r`
  - Like the above but now enabling 6 cores per CCD (1-6).
  - Masks with use of both hardware threads can become extremely long, certainly on LUMI-C…
    - Playing with `--cpus-per-task` and then further restricting with OpenMP environment variables may be the easier way on LUMI-C
- Do not combine with `-c`/`--cpus-per-task`!

Task-to-GPU binding with Slurm

- Currently not recommended on LUMI
  - The control groups mechanism that Slurm uses breaks Peer2Peer IPC for GPU-aware MPI
- `srun --gpu-bind=[{quiet|verbose},]<type>`
- Some `<type>` options are for automatic binding
  - `--gpu-bind=none` is the most useful variant on LUMI: Turns off Slurm binding
  - `--gpu-bind=closest` is broken on LUMI
  - Other options: See the manual
- Other `<type>` options for fully manual distribution
  - `--gpu-bind=map_cpu:<gpu_id_for_task_0>,<gpu_id_for_task_1>,...` : Specify a single GPU for each task on the node
  - `--gpu-bind=mask_cpu:<mask_for_task_0>,<mask_for_task_1>,...` : Specify multiple GPUs via a mask (but only 2 hexadecimal digits as there are only 8 GPUs per node)

MPI rank redistribution with Cray MPICH

- Default behaviour: MPI rank *i* on task *i*
- Cray MPICH has its own mechanism to reorder MPI ranks on Slurm tasks that is more powerful than Slurm's
  - Best to use block distribution in Slurm for this.
  - `export MPICH_RANK_REORDER_METHOD=0` : Round-robin (like Slurm cyclic ordering)
  - `export MPICH_RANK_REORDER_METHOD=1` : Default, preserve the ordering from Slurm
  - `export MPICH_RANK_REORDER_METHOD=2` : Folded rank placement: First assign ranks on first task slot of each node from 0 till …, then assign a rank on the second task slot but now from … till 0, and so on.
  - `export MPICH_RANK_REORDER_METHOD=3` : Custom ordering set by the file `MPICH_RANK_ORDER` (or `$MPICH_RANK_REORDER_FILE`)
- The CPE has profiling tools that help you determine the optimal rank ordering
- See the 4/5-day Advanced LUMI course for more details

Refining core binding in OpenMP

- Slurm will assign cores up to the task/process level
  - Special case: Batch job step: All hardware threads of all cores of the first node of the job
- Thread-level control in OpenMP through library functions or environment variables
  - Debug: `export OMP_DISPLAY_AFFINITY=true`
  - `export OMP_NUM_THREADS=<num>` : Set number of threads
    - Multiple comma-separated numbers possible for multi-level parallelism
  - `OMP_PLACES` to define the places to use for binding: hardware thread level, core level or socket level, or an explicit list
  - `OMP_PROC_BIND` to set distribution and binding strategy over places
- Single level parallelism: Experiment with `omp_check` and `hybrid_check` in `lumi-CPEtools`

Refining core binding in OpenMP: OMP_PLACES

- Defines the places to use for binding
  - `OMP_PLACES=threads` : OpenMP threads restricted to a single hardware thread (default)
  - `OMP_PLACES=cores` : OpenMP threads restricted to both hardware threads of a core
  - `OMP_PLACES=socket` : OpenMP threads restricted to all hardware threads of a single socket
  - Or define a set of locations (very technical)
    ```
    export OMP_PLACES="{0,1,2,3},{8,9,10,11},{16,17,18,19}”
    export OMP_PLACES="{0:4}:3:8”
    ```
    - Core numbers here are relative to those available to the process and not physical numbers

Refining core binding in OpenMP: OMP_PROC_BIND

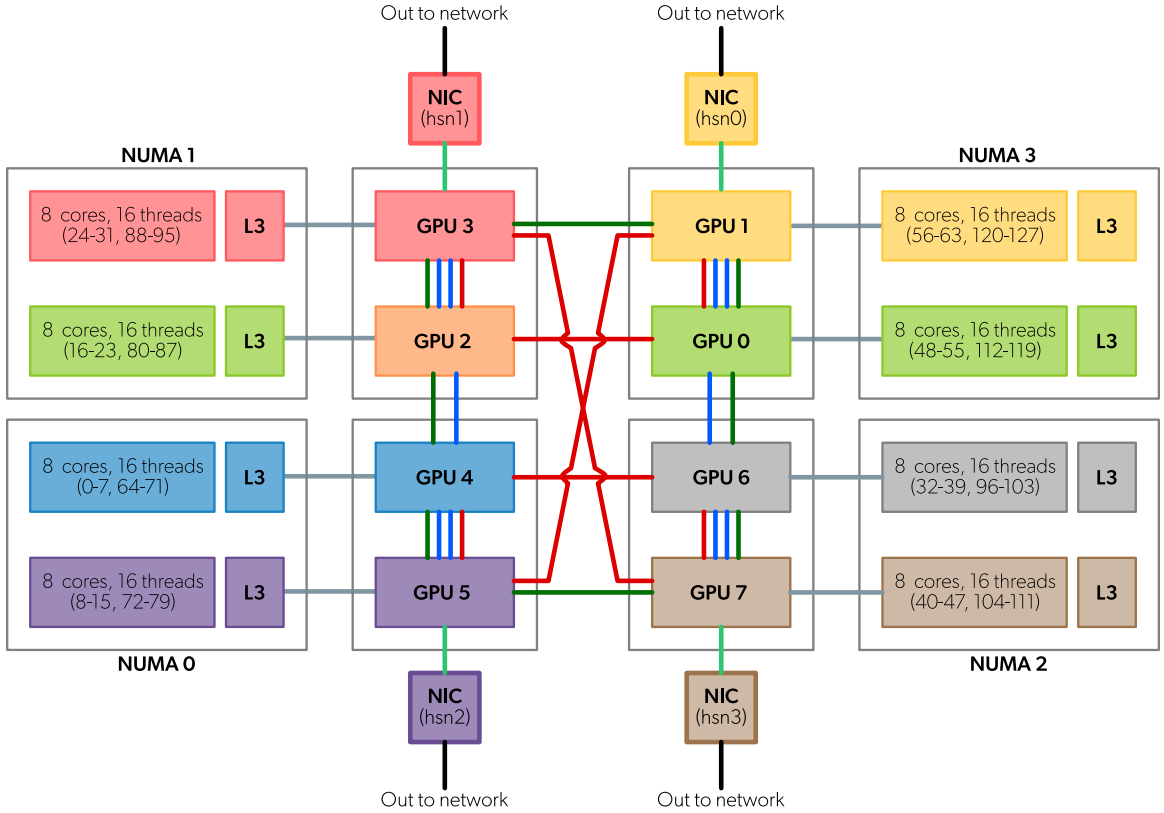
- Distribution over the places and binding selection:
  - `OMP_PROC_BIND=false` : Turn off OpenMP thread binding, use the task affinity mask
  - `OMP_PROC_BIND=close` : Try to keep the OpenMP threads as close as possible with one in each place (unless oversubscribed)
  - `OMP_PROC_BIND=spread` : Try to spread the OpenMP threads out as much as possible
  - `OMP_PROC_BIND=master` : Keep threads in the same place as the master thread.
    - Mostly useful if the place is a socket
- Multiple comma-separated options possible for nested parallelism
- Non-standard option in CCE: `auto` which is the default (other compilers: `false`)
  - CCE does a very reasonable job in many cases
- Many implementations have additional environment variables to tune the distribution

GPU binding with ROCR_VISIBLE_DEVICES

- Works at a very low level of the ROCm software stack
- Limits visibility to certain GPUs for all applications using the ROCm runtime
  - So also covers HIP and OpenCL
- Value: Comma-separated list of all device indices exposed to the application
  - Uses the local numbering in the control group
- Differences with affinity masks for CPUs
  - Affinity masks are OS-controled
  - Therefore the OS can ensure you can only make masks more restrictive than the parent
  - Affinity masks always use the global numbering of hardware threads while `ROCR_VISIBLE_DEVICES` uses the local numbering in the control group

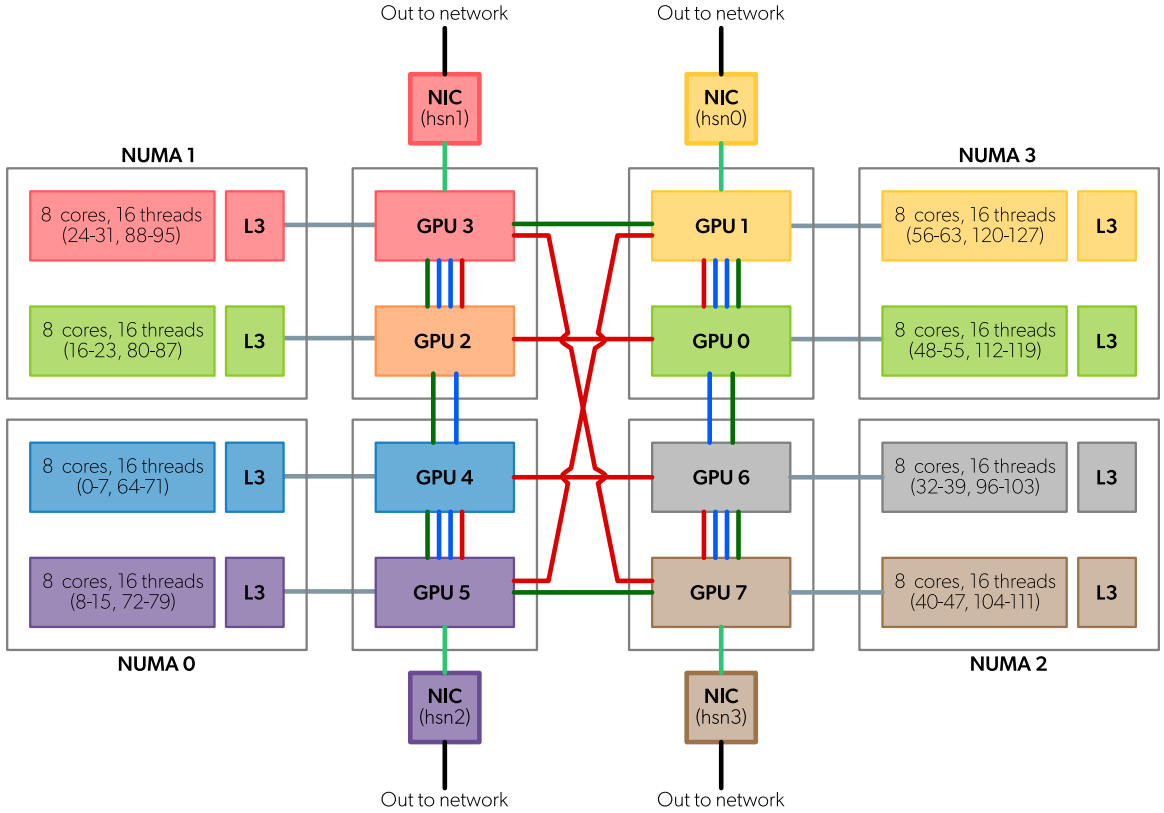
GPU binding: Optimal mapping (1)

CCD	Available HWTs	GCD
0	1-7, 65-71	4
1	9-15, 73-79	5
2	17-23, 81-87	2
3	25-32, 89-95	3
4	33-39, 97-103	6
5	41-47, 105-111	7
6	49-55, 113-119	0
7	57-63, 121-127	1



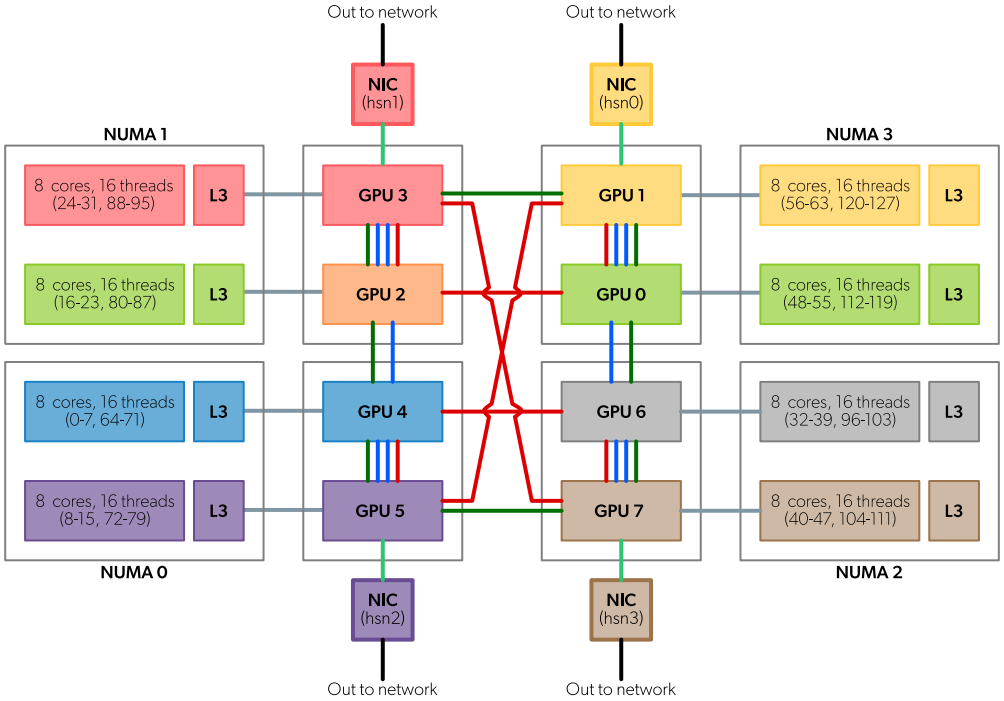
GPU binding: Optimal mapping (2)

GCD	CCD	Available HWTs
0	6	49-55, 113-119
1	7	57-63, 121-127
2	2	17-23, 81-87
3	3	25-32, 89-95
4	0	1-7, 65-71
5	1	9-15, 73-79
6	4	33-39, 97-103
7	5	41-47, 105-111



GPU binding: Embedded rings

- Green ring: 0 – 1 – 3 – 2 – 4 – 5 – 7 – 6 – 0
- Red ring: 0 – 1 – 5 – 4 – 6 – 7 – 3 – 2 – 0



GPU binding: Implementation

- Combination of two mechanisms:
  - CPU side: Use `--cpu-bind`, or in some cases simply `--cpus-per-task`
  - GPU side: Manual binding required by setting `ROCR_VISIBLE_DEVICES` because Slurm uses a mechanism with unwanted side effects.
    - Use a wrapper script that computes the proper GPU(s) from the Slurm local task id, sets `ROCR_VISIBLE_DEVICES` and then starts the application

GPU binding: Linear GCD, match cores (1)

```
…
#SBATCH --partition=standard-g
#SBATCH --gpus-per-node=8
…
cat << EOF > select_gpu_$SLURM_JOB_ID
#!/bin/bash
export ROCR_VISIBLE_DEVICES=\$SLURM_LOCALID
exec \$*
EOF
chmod +x select_gpu_$SLURM_JOB_ID
…
CPU_BIND1="map_cpu:49,57,17,25,1,9,33,41"
srun --ntasks=$((SLURM_NNODES*8)) --cpu-bind=$CPU_BIND1 \
    ./select_gpu_$SLURM_JOB_ID gpu_check -l
```

GPU binding: Linear GCD, match cores (2)

```
…
#SBATCH --partition=standard-g
#SBATCH --gpus-per-node=8
…
cat << EOF > select_gpu_$SLURM_JOB_ID
#!/bin/bash
export ROCR_VISIBLE_DEVICES=\$SLURM_LOCALID
exec \$*
EOF
chmod +x select_gpu_$SLURM_JOB_ID
…
CPU_BIND2="mask_cpu:0xfe000000000000,0xfe00000000000000"
CPU_BIND2="$CPU_BIND2,0xfe0000,0xfe000000"
CPU_BIND2="$CPU_BIND2,0xfe,0xfe00"
CPU_BIND2="$CPU_BIND2,0xfe00000000,0xfe0000000000"
srun --ntasks=$((SLURM_NNODES*8)) --cpu-bind=$CPU_BIND2 \
    ./select_gpu_$SLURM_JOB_ID gpu_check -l
```

GPU binding: Linear CCD, match GCD (1)

```
…
#SBATCH --partition=standard-g
#SBATCH --gpus-per-node=8
…
cat << EOF > select_gpu_$SLURM_JOB_ID
#!/bin/bash
GPU_ORDER=(4 5 2 3 6 7 0 1)
export ROCR_VISIBLE_DEVICES=\${GPU_ORDER[\$SLURM_LOCALID]}
exec \$*
EOF
chmod +x select_gpu_$SLURM_JOB_ID
…
CPU_BIND1="map_cpu:1,9,17,25,33,41,49,57"
srun --ntasks=$((SLURM_NNODES*8)) --cpu-bind=$CPU_BIND1 \
    ./select_gpu_$SLURM_JOB_ID gpu_check -l
```

GPU binding: Linear CCD, match GCD (2)

```
…
#SBATCH --partition=standard-g
#SBATCH --gpus-per-node=8
…
cat << EOF > select_gpu_$SLURM_JOB_ID
#!/bin/bash
GPU_ORDER=(4 5 2 3 6 7 0 1)
export ROCR_VISIBLE_DEVICES=\${GPU_ORDER[\$SLURM_LOCALID]}
exec \$*
EOF
chmod +x select_gpu_$SLURM_JOB_ID
…
CPU_BIND2="mask_cpu"
CPU_BIND2="$CPU_BIND2:0x00000000000000fe,0x00000000000fe00"
CPU_BIND2="$CPU_BIND2,0x0000000000fe0000,0x0000000fe000000"
CPU_BIND2="$CPU_BIND2,0x000000fe00000000,0x0000fe0000000000"
CPU_BIND2="$CPU_BIND2,0x00fe000000000000,0xfe00000000000000"
srun --ntasks=$((SLURM_NNODES*8)) --cpu-bind=$CPU_BIND2 \
    ./select_gpu_$SLURM_JOB_ID gpu_check -l
```

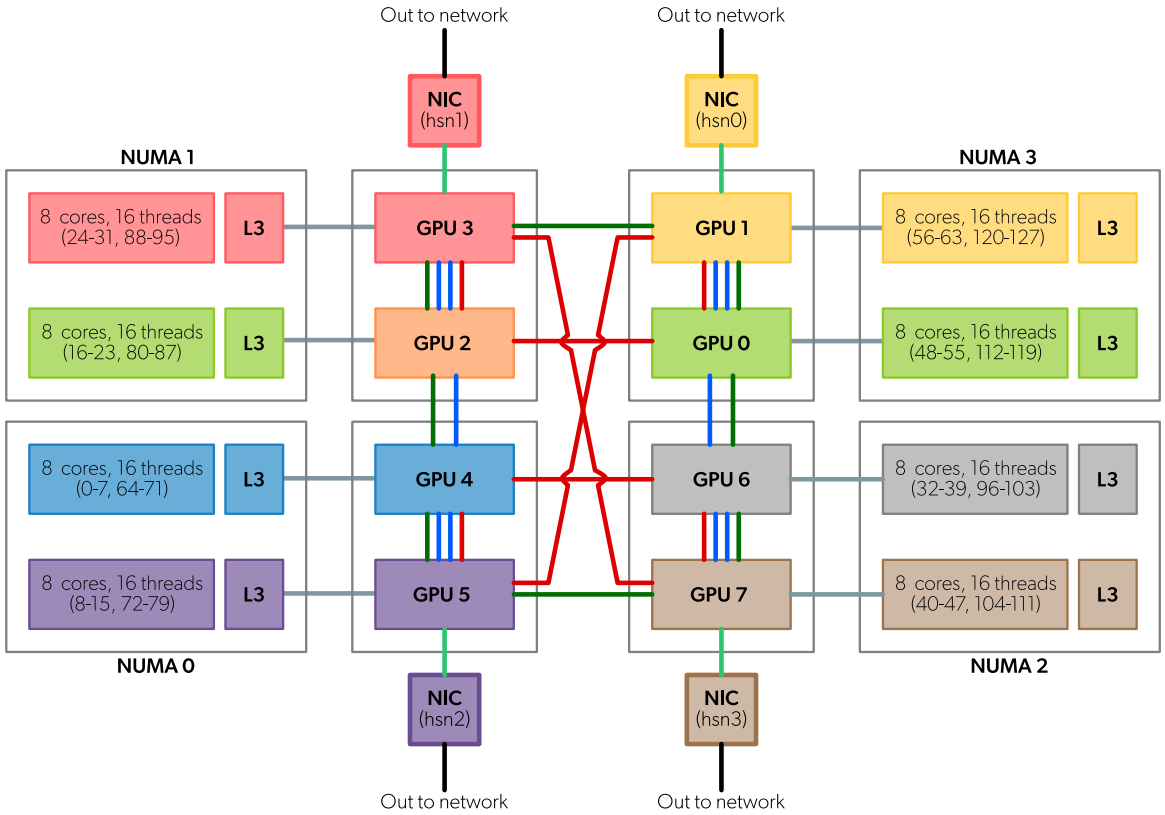
GPU binding: Linear CCD, match GCD (3)

```
…
#SBATCH --partition=standard-g
#SBATCH --gpus-per-node=8
…
cat << EOF > select_gpu_$SLURM_JOB_ID
#!/bin/bash
GPU_ORDER=(4 5 2 3 6 7 0 1)
export ROCR_VISIBLE_DEVICES=\${GPU_ORDER[\$SLURM_LOCALID]}
exec \$*
EOF
chmod +x select_gpu_$SLURM_JOB_ID
…
srun --ntasks=$((SLURM_NNODES*8)) -cpus-per-task=7 \
    ./select_gpu_$SLURM_JOB_ID gpu_check -l

…
export OMP_NUM_THREADS=6
srun --ntasks=$((SLURM_NNODES*8)) -cpus-per-task=7 \
    ./select_gpu_$SLURM_JOB_ID gpu_check -l
```

GPU binding: Green ring (1)

Task	GCD	CCD	HWTs
0	0	6	49-55, 113-119
1	1	7	57-63, 121-127
2	3	3	25-32, 89-95
3	2	2	17-23, 81-87
4	4	0	1-7, 65-71
5	5	1	9-15, 73-79
6	7	5	41-47, 105-111
7	6	4	33-39, 97-103



GPU binding: Green ring (2)

```
…
#SBATCH --partition=standard-g
#SBATCH --gpus-per-node=8
…
cat << EOF > select_gpu_$SLURM_JOB_ID
#!/bin/bash
GPU_ORDER=(0 1 3 2 4 5 7 6)
export ROCR_VISIBLE_DEVICES=\${GPU_ORDER[\$SLURM_LOCALID]}
exec \$*
EOF
chmod +x select_gpu_$SLURM_JOB_ID
…
CPU_BIND1="map_cpu:49,57,25,17,1,9,41,33"
srun --ntasks=$((SLURM_NNODES*8)) --cpu-bind=$CPU_BIND1 \
    ./select_gpu_$SLURM_JOB_ID gpu_check -l
```

GPU binding: Green ring (3)

```
cat << EOF > select_gpu_$SLURM_JOB_ID
#!/bin/bash
GPU_ORDER=(0 1 3 2 4 5 7 6)
export ROCR_VISIBLE_DEVICES=\${GPU_ORDER[\$SLURM_LOCALID]}
exec \$*
EOF
chmod +x select_gpu_$SLURM_JOB_ID
…
CCD_MASK=( 0x00000000000000fe \
           0x000000000000fe00 \
           0x0000000000fe0000 \
           0x00000000fe000000 \
           0x000000fe00000000 \
           0x0000fe0000000000 \
           0x00fe000000000000 \
           0xfe00000000000000 )
CPU_BIND2="mask_cpu"
CPU_BIND2="$CPU_BIND2:${CCD_MASK[6]},${CCD_MASK[7]}"
CPU_BIND2="$CPU_BIND2,${CCD_MASK[3]},${CCD_MASK[2]}"
CPU_BIND2="$CPU_BIND2,${CCD_MASK[0]},${CCD_MASK[1]}"
CPU_BIND2="$CPU_BIND2,${CCD_MASK[5]},${CCD_MASK[4]}"
srun --ntasks=$((SLURM_NNODES*8)) --cpu-bind=$CPU_BIND2 \
    ./select_gpu_$SLURM_JOB_ID gpu_check -l
```

“Allocate by resources” partitions

- Proper binding not possible unless exclusively allocating entire nodes only
- Slurm will use a control group per task for the GPUs
  - You almost have to use `--gpus-per-task` to ensure that GPUs and tasks are on the same nodes (unless you use just a single node)
  - Problems with Peer2Peer IPC
  - Solution:
    - Turn off with `--gpu-bind=none`
    - This will number visible GPUs for the job on each node from 0,
    - and we can then again use the local task ID to assign a GPU to each task via `ROCR_VISIBLE_DEVICES` via the `select_gpu` script trick.
- Optimal mapping is not possible


Questions?