AFFINITY – Placement, Order and Binding

CASTIEL Training on LUMI
May 2-5, 2023

Gina Sitaraman, Bob Robey

AMD
together we advance_

[Public]

Authors and Contributors

- Tom Papatheodore, ORNL
- Marcus Wagner, HPE
- Alfio Lazzaro, HPE
- Georgios Markomanolis, AMD
- Bill Brantley, AMD
- Noel Chalmers, AMD
- Kjetil Haugen, AMD

AMD
together we advance_

[Public]

Agenda

- A look at Modern Heterogenous Architectures
- What is Affinity? Why is it important?
- Understanding Node Topology
- Placement Considerations on LUMI nodes
- Case Studies: Affinity Settings for Different Types of Applications

AMD
together we advance_

[Public]

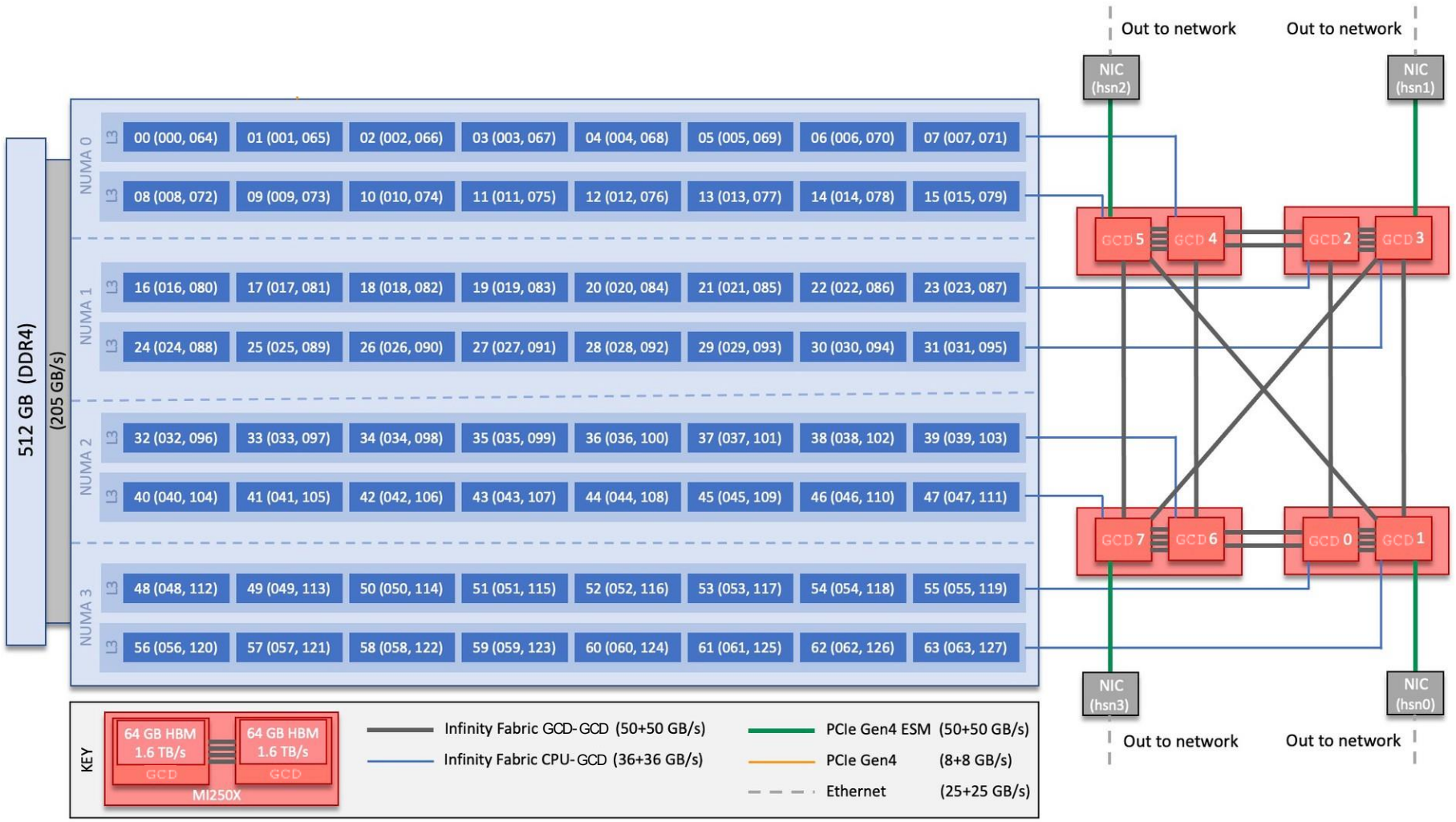
Modern Hardware Architectures

- Increasingly complex with multiple resources
  - sockets
  - cores
  - GPUs
  - memory controllers
  - NICs (Network Interface Cards)
- Peripherals such as GPUs and memory controllers are local to a CPU socket
- Operating System (OS) controls process scheduling but is not designed for parallel and high-performance computing jobs
  - Processes may be preempted
  - When rescheduled on a new core, cached data has to be moved to the caches close to the new core
  - OS is unaware of parallel processes or their threads

AMD
together we advance_

[Public]

LUMI Node Architecture



- 64 cores on a single socket CPU
- 4 MI250X GPUs, each with 2 GCDs
  - Each GCD is presented as a GPU device to `rocm-smi`
- 512 GB of DDR4 RAM
- Infinity Fabric™ links between GCDs and between GCDs and CPU cores
- 4 NICs attached to odd numbered GCDs

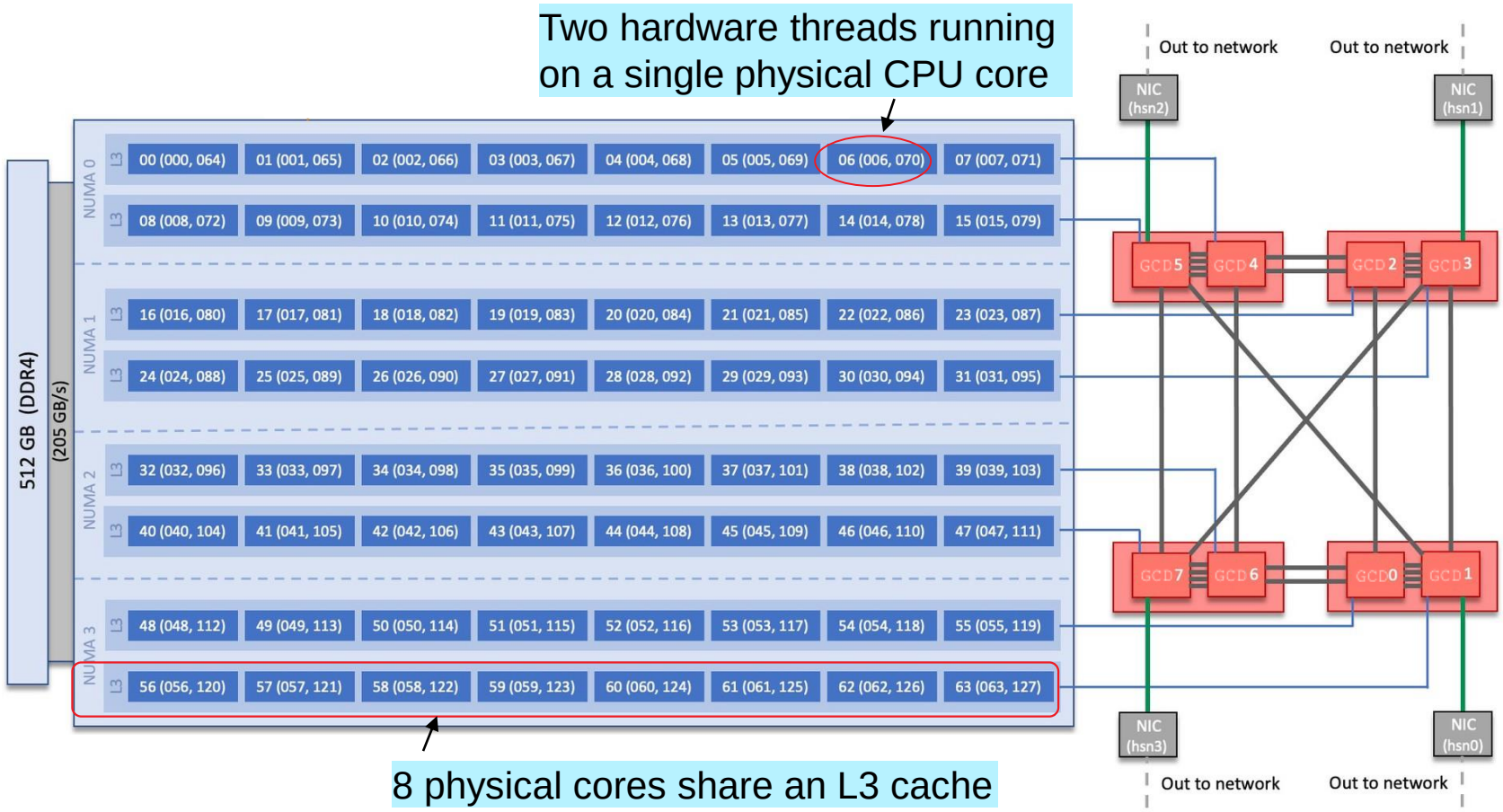
Courtesy: https://docs.olcf.ornl.gov/systems/frontier_user_guide.html#frontier-compute-nodes

AMD
together we advance_

[Public]

NUMA (Non-Uniform Memory Access)

- Multi-processor systems where resources are divided into multiple nodes or domains
- A NUMA domain is a grouping of cores, memory and other peripherals
- Each CPU core is attached to its own local memory while being able to access memory attached to other processors
- Local memory accesses are fast while remote memory accesses have a higher latency, especially those that cross a socket-to-socket interconnect
- With local accesses, memory contention from CPUs is reduced resulting in increased bandwidth



AMD
together we advance_

[Public]

NUMA configuration (NPS)

- LUMI nodes may be configured at boot time with 1 or 4 NUMA domains Per Socket (NPS)
  - Site administers this setting, users cannot change it
- **NPS1**:
  - 1 NUMA domain per socket
  - Memory accesses interleaved across all 8 memory channels
  - More uniform bandwidth but slightly higher latency than NPS4 case
  - More tolerant of hot spots in memory channels
  - For example, if you are running only 1 MPI rank, you may benefit from a higher CPU memory bandwidth
- **NPS4**:
  - 4 NUMA domains per socket
  - Memory accesses in a domain interleaved across 2 memory channels
  - Potential for higher memory bandwidth due to reduced contention and lower latency
  - May be vulnerable to hot spots
  - With NPS4, affinity is really important – need to spread processes across the NUMA domains

- LUMI nodes are currently configured with NPS4

AMD
together we advance_

[Public]

What is Affinity?

- Affinity is a way for processes to indicate preference for hardware components (memory, cores, NICs, caches)
  - Processes can be pinned to resources typically belonging to the same NUMA domain
- Why is Affinity important?
  - Improves cache reuse
  - Improves NUMA memory locality
  - Reduces contention for resources
  - Lowers latency
  - Reduces variability from run to run
- Where is Affinity needed?
  - Extremely important for processes running on CPU cores and the resulting placement of their data in CPU memory
  - When running on GPUs, affinity is less critical unless there is a bottleneck with the location of data in host memory
    - Memory copies between host and device, page migration and direct memory access may be affected if data in host memory is not in same NUMA domain
  - Within a GPU, affinity is far less important
- For parallel processes, Affinity is more than binding:
  - Placement
  - Order

AMD
together we advance_

[Public]

Process Placement

- **Placement** indicates where a process is placed
- **Motivation**: maximize available resources for a particular application/workload
  - We want to use all resources (cores, caches, GPUs, NICs, memory controllers, etc...)
  - Processes may have multiple threads (OpenMP®) and require separate cores for each thread
  - We may want to use only hardware/physical cores and not virtual cores
  - We may not have enough memory per process, we may want to skip some cores
  - We may want to reserve some cores for system operations to reduce jitter for timing purposes
  - MPI prefers "gang scheduling" whereas the OS doesn't know the processes are connected
    - When a process waits to be scheduled by the OS, it may cause all other processes to wait longer at a synchronization barrier
- Until the last decade, placement was not that important
  - Only 2-8 cores on a CPU, uniform architectures, no GPUs
  - Distributed or Shared memory systems
  - The OS controlled placement of processes, and that was okay
- On hardware today, controlling placement may help
  - Avoid oversubscription of compute resources and unnecessary contention for common resources
  - Avoid non-uniform use of compute resources where some processors are used, and some are idle
  - Avoid sub-optimal communication performance when processes are placed too widely apart
  - Prevent migration of processes
- Affinity controls in the OS and MPI have greatly improved and changed

AMD
together we advance_

[Public]

Order of Processes

- Order defines how processes of a parallel job are distributed across the sockets of the node
- Why is order important?
  - Processes communicating with each other are close together for lower latency and higher bandwidth
  - Load balancing heavy workloads by scattering across compute resources
- **Round-robin** or **Cyclic**:
  - Processes are distributed in a round-robin fashion across sockets.
  - For example, if there are 8 MPI ranks and 2 sockets, rank 0 is scheduled on socket 0, rank 1 on socket 1, rank 2 on socket 0, rank 3 on socket 1 and so on.
  - Maximizes available cache for each process, and evenly utilizes the resources of a node
- **Packed** or **Close**:
  - Consecutive MPI ranks are assigned to processors in the same socket until it is filled before scheduling a rank on a different socket
  - For example, if there are 8 MPI ranks and 2 sockets each with a 4 core CPU, ranks 0-3 are scheduled on socket 0, and ranks 4-7 are scheduled on socket 1
  - Improved performance due to data locality if ranks that communicate the most are accessing data in the same memory node and sharing cache

AMD
together we advance_

Understanding Node Topology

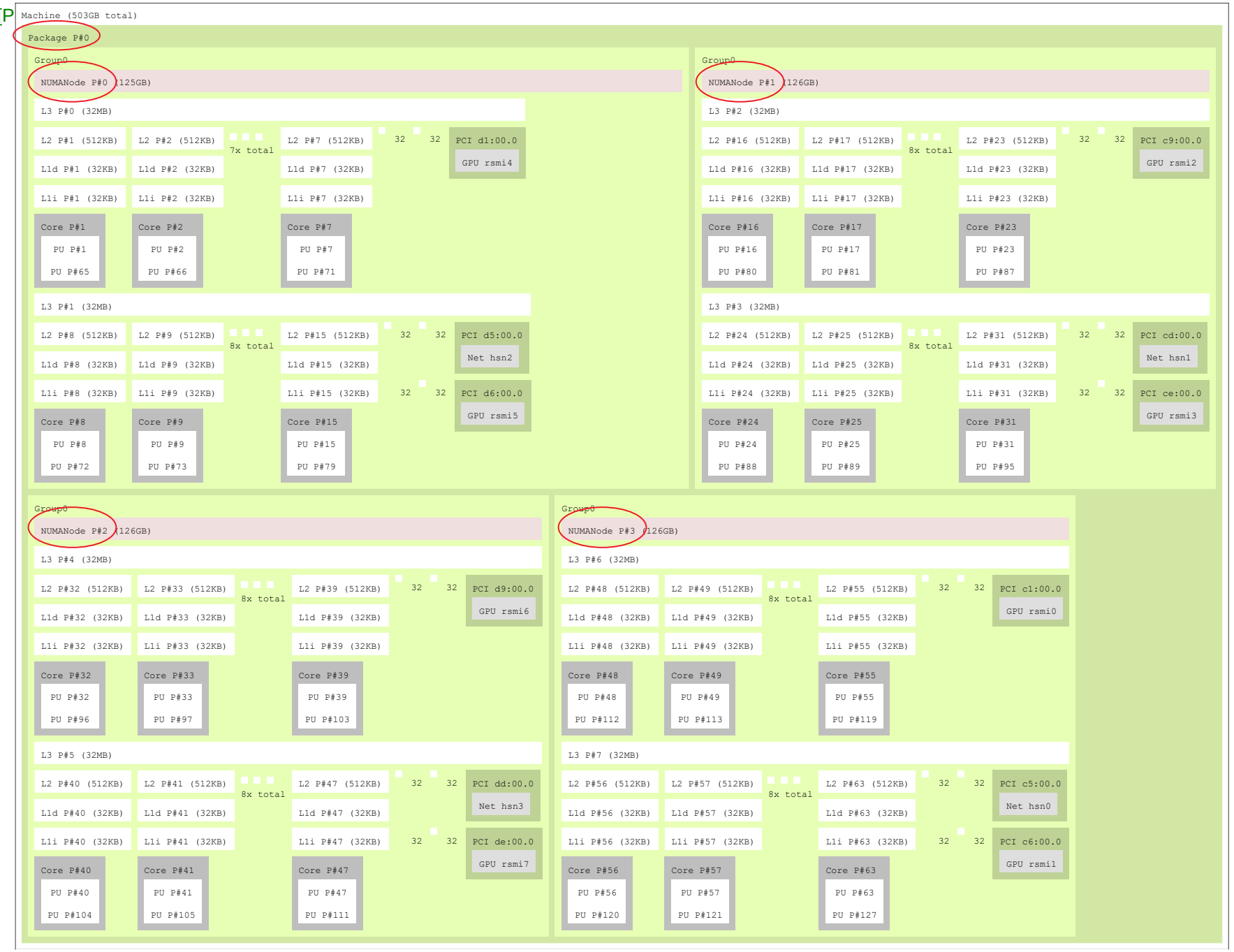
[Public]

Understanding Node Topology

- Even on a LUMI type system, the configuration may be different
  - Number of NUMA domains per socket may change at boot time
  - Some physical cores may be reserved
  - Virtual cores may be enabled or disabled

- Some tools can help understand your system better
  - **`lstopo`**: from `hwloc` package to visualize node architecture
  - **`lscpu`**: gathers and displays CPU architecture information
  - **`numactl -H`**: shows available NUMA nodes in the system and CPU core affinity for each node
  - **`rocm-smi --showtopo`**: Displays the NUMA node and the CPU affinity associated with every GPU device.

AMD
together we advance_



LUMI Node Topology

- `lstopo -p out.svg`
- 1 socket = 1 package
- 4 NUMA nodes in socket

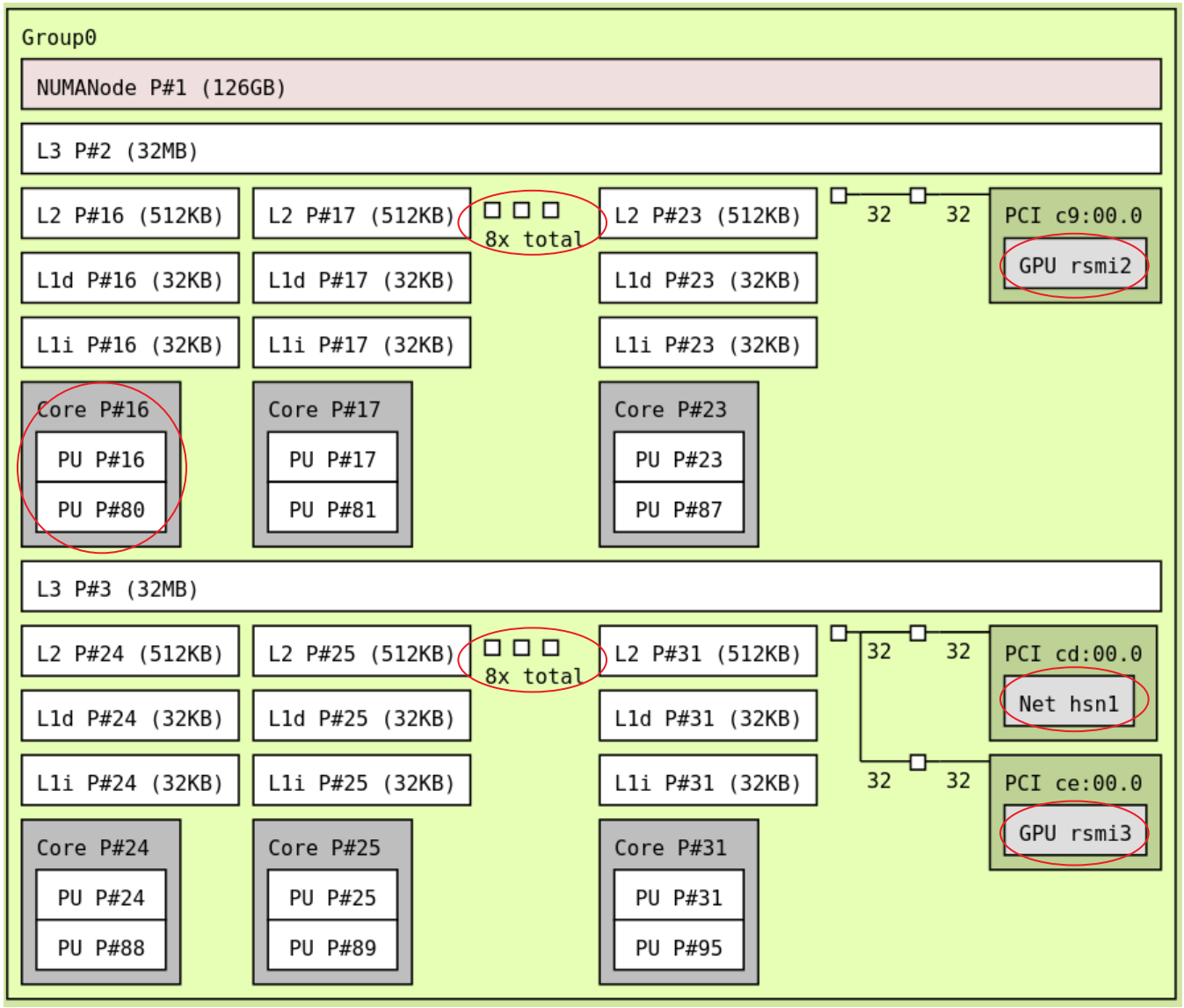
If you can't read this, it proves how complex the architecture is :)

Host: nid005103
Indexes: physical
Date: Tue Nov 15 01:44:25 2022

AMD
together we advance_

[Public]

Understanding Node Topology – lstopo NUMA domain #1



- 8 physical cores + 8 virtual cores share an L3 cache
- Two sets of 8 physical cores in a NUMA domain
- Two GCDs in a NUMA domain
- One high-speed NIC per NUMA domain

AMD
together we advance_

[Public]

Understanding CPU Architecture

lscpu

```
Architecture:                    x86_64
CPU(s):                          128
On-line CPU(s) list:             0-127
Thread(s) per core:              2
Core(s) per socket:              64
Socket(s):                       1
NUMA node(s):                    4
Model name:                      AMD EPYC 7A53 64-Core Processor
Frequency boost:                 enabled
CPU MHz:                         3488.045
L1d cache:                       2 MiB
L1i cache:                       2 MiB
L2 cache:                        32 MiB
L3 cache:                        256 MiB
NUMA node0 CPU(s):               0-15,64-79
NUMA node1 CPU(s):               16-31,80-95
NUMA node2 CPU(s):               32-47,96-111
NUMA node3 CPU(s):               48-63,112-127
```

OS sees 128 cores or hardware threads (HWT)

Hyperthreading is enabled

Hardware thread affinity to NUMA domains

AMD
together we advance_

[Public]

Understanding NUMA Configuration

numactl -H

Here, hardware threads 0-15 and 64-79 belong to NUMA domain 0

```
available: 4 nodes (0-3)
node 0 cpus: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79
node 0 size: 128411 MB
node 0 free: 119892 MB
node 1 cpus: 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95
node 1 size: 129015 MB
node 1 free: 124248 MB
node 2 cpus: 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 96 97 98 99 100 101 102 103 104 105 106 107 108
109 110 111
node 2 size: 129015 MB
node 2 free: 124702 MB
node 3 cpus: 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 112 113 114 115 116 117 118 119 120 121 122 123
124 125 126 127
node 3 size: 128998 MB
node 3 free: 124737 MB
node distances:
node   0   1   2   3
  0:  10  12  12  12
  1:  12  10  12  12
  2:  12  12  10  12
  3:  12  12  12  10
```

More obvious on multiple socket nodes

AMD
together we advance_

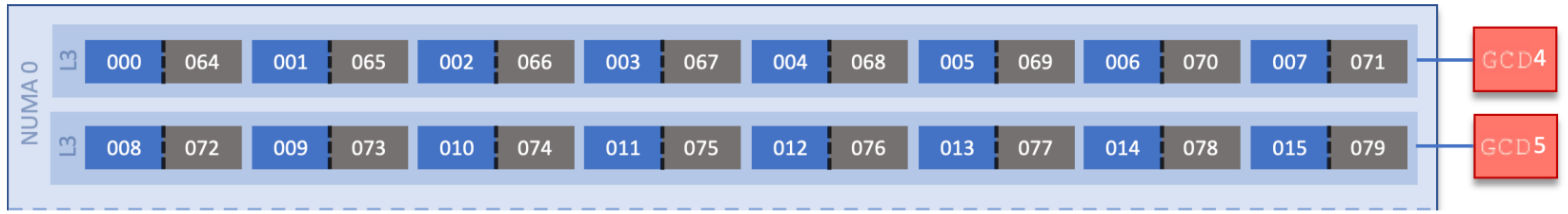
[Public]

Understanding NUMA Configuration for GPUs

rocm-smi --showtopo

```
================================== Numa Nodes ==================================
GPU[0]          : (Topology) Numa Node: 3
GPU[0]          : (Topology) Numa Affinity: 3
GPU[1]          : (Topology) Numa Node: 3
GPU[1]          : (Topology) Numa Affinity: 3
GPU[2]          : (Topology) Numa Node: 1
GPU[2]          : (Topology) Numa Affinity: 1
GPU[3]          : (Topology) Numa Node: 1
GPU[3]          : (Topology) Numa Affinity: 1
GPU[4]          : (Topology) Numa Node: 0
GPU[4]          : (Topology) Numa Affinity: 0
GPU[5]          : (Topology) Numa Node: 0
GPU[5]          : (Topology) Numa Affinity: 0
GPU[6]          : (Topology) Numa Node: 2
GPU[6]          : (Topology) Numa Affinity: 2
GPU[7]          : (Topology) Numa Node: 2
GPU[7]          : (Topology) Numa Affinity: 2
============================== End of ROCm SMI Log ==============================
```

GCDs 4 and 5 are located in NUMA domain 0



AMD
together we advance_

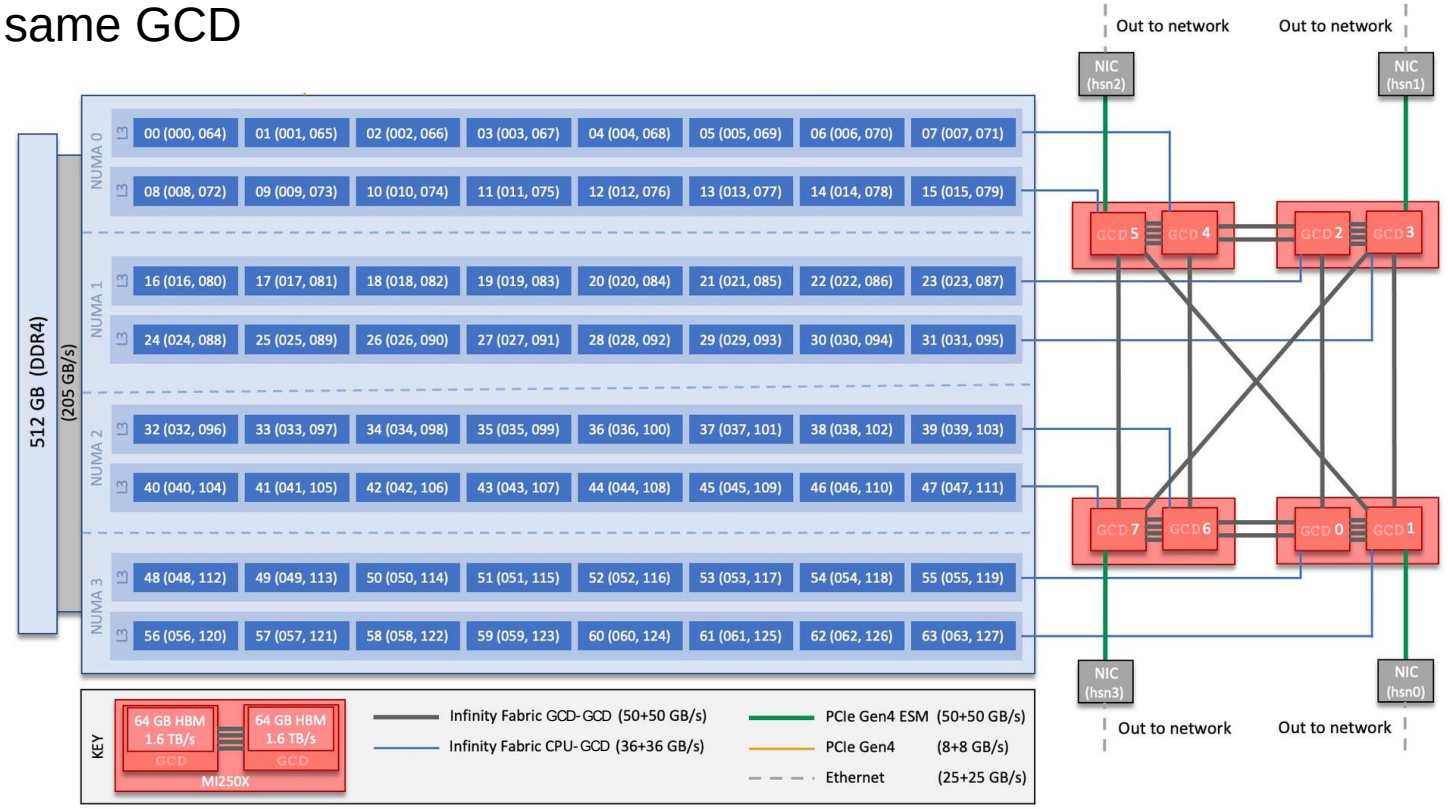
Placement Considerations on LUMI nodes

[Public]

Placement Considerations on LUMI nodes

- Each GCD is connected to one of the NUMA domains via a high-speed Infinity Fabric™ link
- Memory bandwidth is highest between GCDs of the same MI250X GPU
- NICs are directly connected to odd numbered GCDs
- Multiple processes can run on the same GCD

Choose rank order and placement carefully to optimize communication

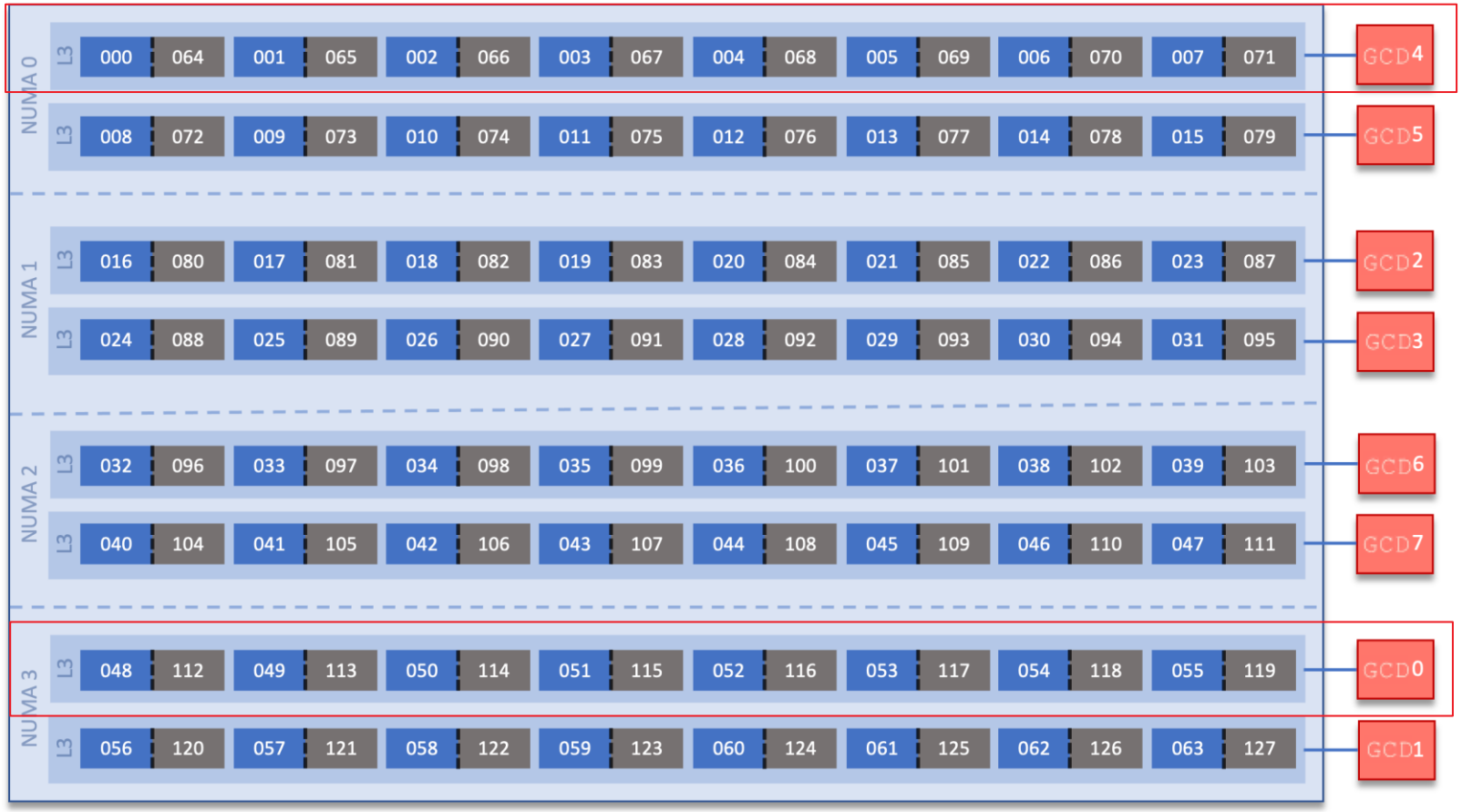


AMD
together we advance_

[Public]

Placement Considerations on LUMI nodes

- Each GCD is connected to a set of 8 CPU cores via a high-speed Infinity Fabric™ link
  - Pinning a process and its threads on cores closest to the GCD it uses improves the efficiency of H2D and D2H transfers



Cores 0-7 are closest to GCD 4

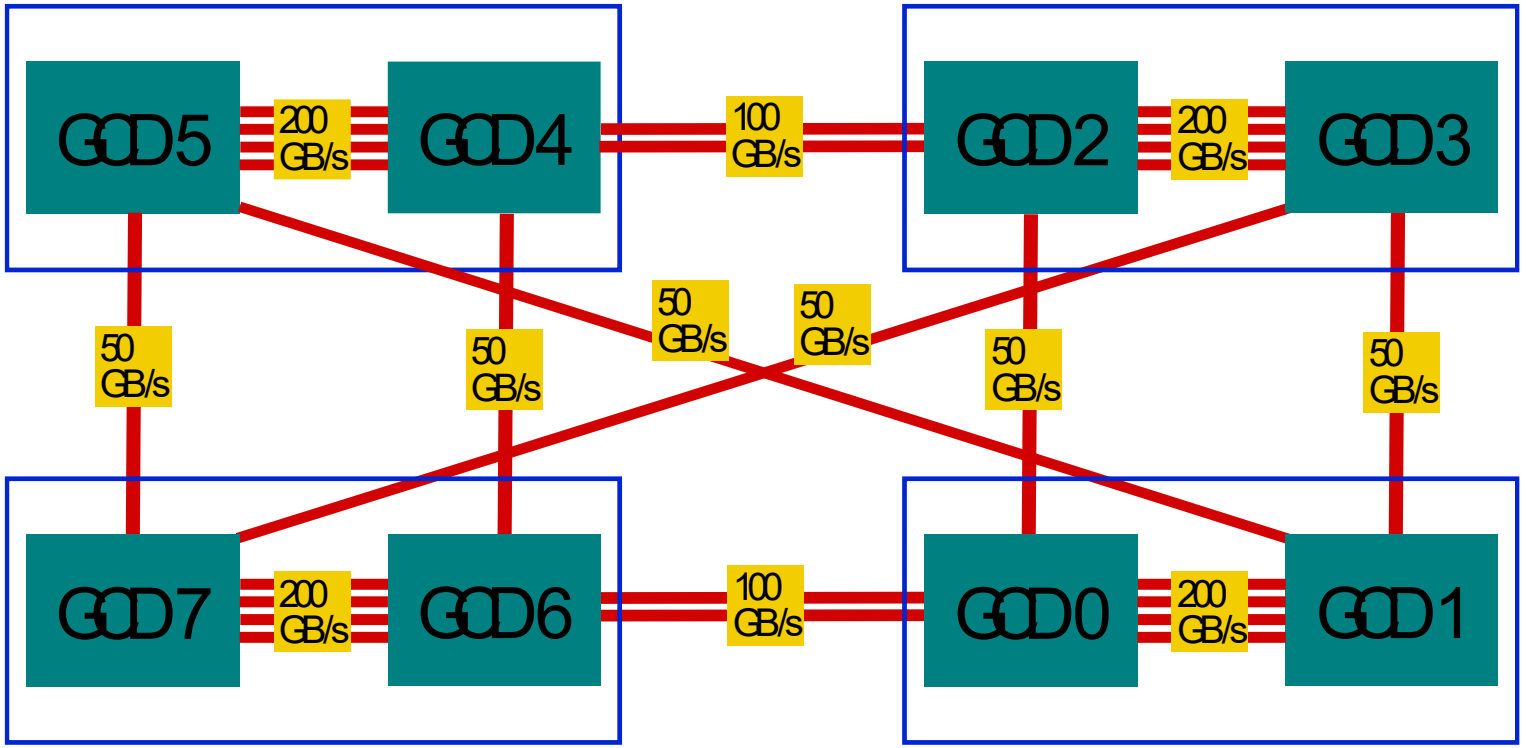
Cores 48-55 are closest to GCD 0

AMD
together we advance_

[Public]

Placement Considerations on LUMI nodes

- Memory bandwidth is highest between GCDs of the same MI250X GPU
  - 4 Infinity Fabric™ links connect the two GCDs for a combined 200 GB/s peak bandwidth in each direction
  - Place pairs of ranks that communicate the most on GCDs of the same MI250X GPU



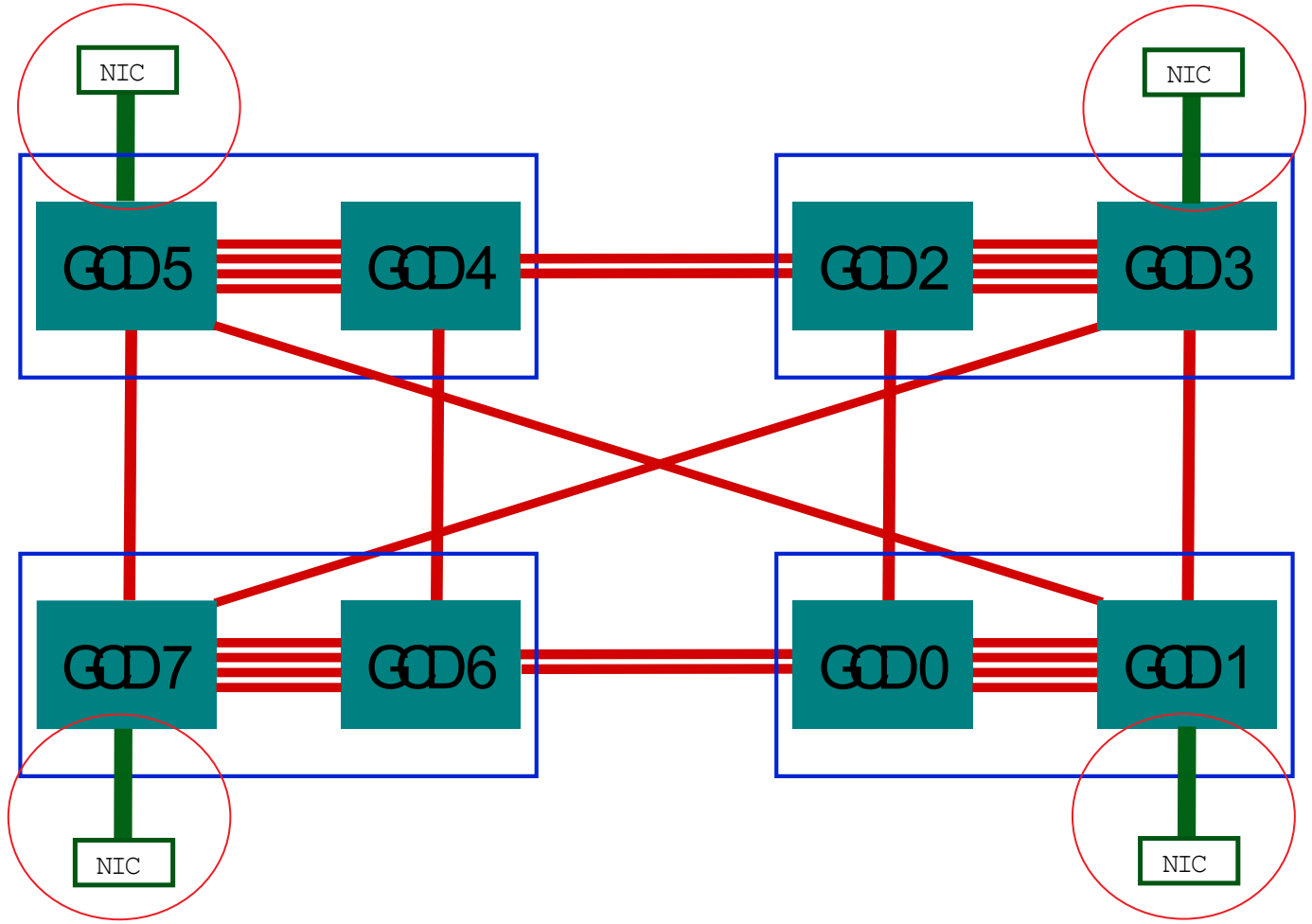
- Peak Bandwidth in each direction of Infinity Fabric™ link shown
- Even though bandwidths are different between GCDs, communication using device buffers will be at least as fast as communication using host buffers

AMD
together we advance_

[Public]

Placement Considerations on LUMI nodes

- On a LUMI node, there are 4 NICs
- NICs are directly connected to odd numbered GCDs
- Inter-node MPI Communication using device buffers is expected to be faster (GPU Aware MPI)
- Cray provides environment variables for mapping processes to the NIC in the same NUMA domain

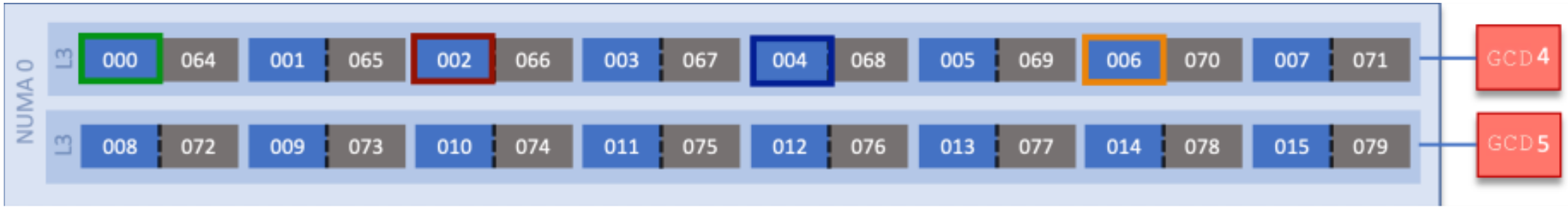


AMD
together we advance_

[Public]

Placement Considerations on LUMI nodes

- Multiple processes on the same GCD
  - AMD GPUs natively support running multiple MPI ranks on the same device where all processes share the available resources and improve utilization
  - Depending on the application's communication pattern, pack ranks that communicate most on the same device



Here, 4 MPI ranks are running on GCD 4, and are pinned to cores 0, 2, 4 and 6 respectively

AMD
together we advance_

[Public]

Choose Rank Order Carefully to Optimize Communication

- Intra-node communication is faster than inter-node communication
- Application expert may know the best placement
  - For example, stencil near neighbors should be placed next to each other
- HPE's CrayPat profiler may be used to detect communication pattern between MPI ranks and generate a rank order file that can then be supplied to Cray MPICH
- HPE's `grid_order` utility may also be used to determine optimal rank order, check with HPE for more details
- Slurm binding options

AMD
together we advance_

[Public]

How do I verify if I got the right Affinity?

- Use `top` or `htop` to visualize where processes and their threads are running
- If using OpenMPI, `mpirun --report-bindings` can be used to show the binding of each process as a mask
- For MPI + OpenMP® programs, you can use the following simple "Hello, World" program to check mappings: https://code.ornl.gov/olcf/hello_mpi_omp
- For MPI + OpenMP® + HIP programs, a simple "Hello, World" program with HIP can be used to verify mappings: https://code.ornl.gov/olcf/hello_jobstep
- HPE's `xthi` script: https://github.com/olcf/XC30-Training/blob/master/affinity/Xthi.c
- Example code from Essentials of Parallel Computing, Chapter 14 can be used to verify mappings for OpenMP®, MPI and MPI+OpenMP cases: https://github.com/essentialsofparallelcomputing/Chapter14

Best Practice: Run script or “hello world” program prior to your application in the same Slurm batch job to confirm affinity setting

AMD
together we advance_

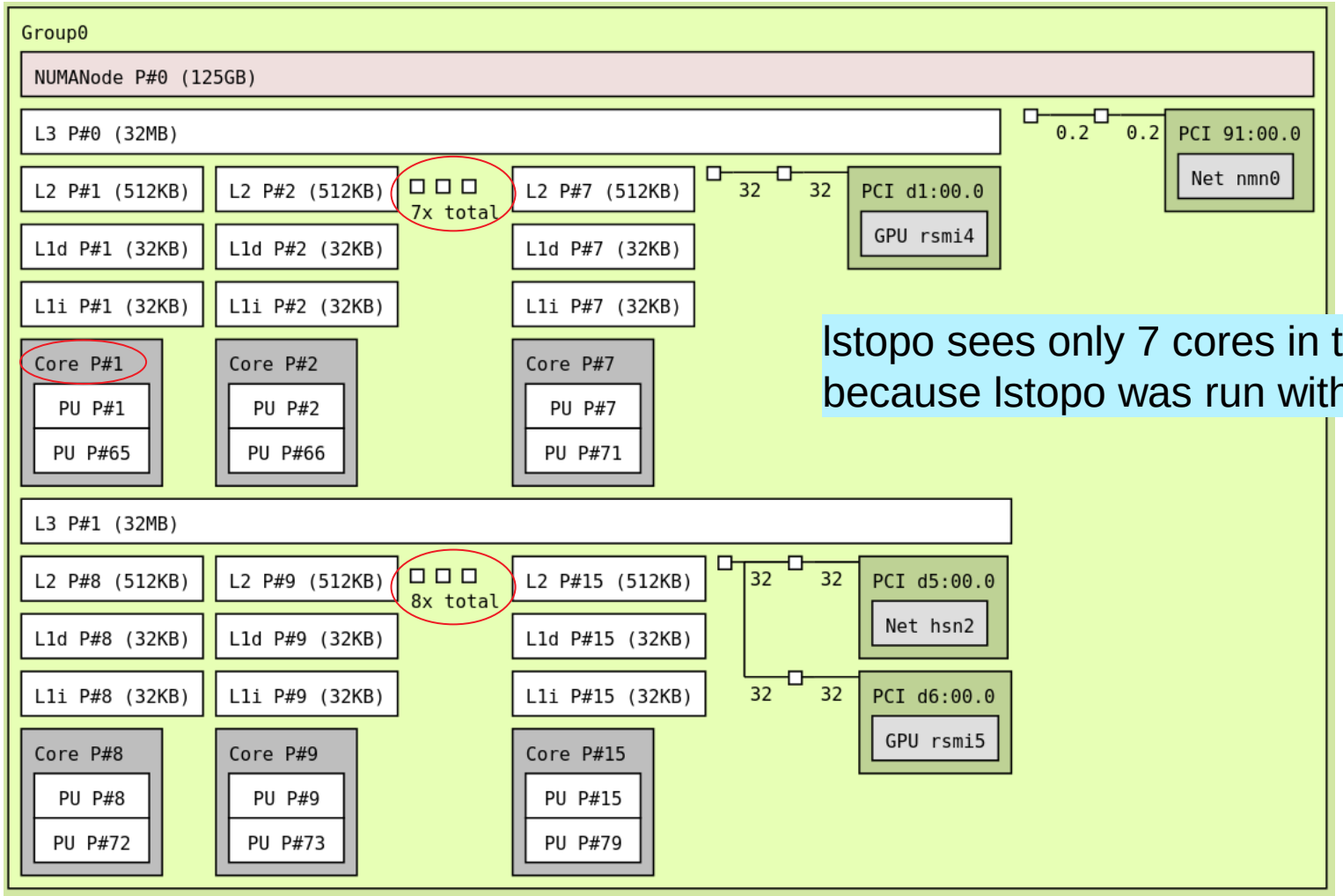
[Public]

Low noise mode on LUMI – A Small Detour

Where is Core 0?

Slurm setting in LUMI reserves Core 0 for system operations

Helps reduce jitter and variability from run to run



lstopo sees only 7 cores in the first CPU set because lstopo was run with Slurm

For applications that are bandwidth bound, GPU bound or not multi-threaded, losing one core may not be a big deal. Losing a core in CPU compute bound applications will hurt performance.

AMD
together we advance_

[Public]

Case Studies for Setting Affinity

- **Serial Applications with OpenMP®**
  - Using `numactl`
  - Using OpenMP® settings, `OMP_PLACES, OMP_PROC_BIND`
  - Using GNU OpenMP® environment variables, `GOMP_CPU_AFFINITY`
- **MPI Applications + OpenMP® + HIP**
  - Using Slurm binding options
    - 1 MPI rank per GCD
    - 1 MPI rank per GCD, 8 OpenMP threads per rank
    - 2 MPI ranks per GCD

AMD
together we advance_

Case Studies: Serial Application + OpenMP®

Setting CPU Affinity

[Public]

Controlling Affinity for Serial Applications – numactl

- Use **`numactl`** from libnuma-dev Linux® package to control NUMA policy for processes and shared memory

```
numactl –C 2,3 –m 0 ./exe
        ^--  Run exe on CPU cores 2 or 3 and allocate mem on NUMA node 0
numactl –C 1-7 -i 0,1 ./exe
        ^--  Run exe on cores 1-7 and interleave memory allocations on NUMA nodes 0 and 1
```

- More detailed documentation can be found in the `numactl` manpage

- To verify bindings, run `htop` or `top`

AMD
together we advance_

[Public]

Controlling Affinity for Serial Applications – OpenMP® settings

- OpenMP® 5.2 standard specifies environment variables to control affinity settings
- **OMP_PLACES** indicates hardware resources
  - Can be an abstract name: `cores`, `threads`, `sockets`, `l1_caches` or `numa_domains` (definitions are implementation specific)
  - Can be an explicit list of places described by non-negative numbers

```
export OMP_PLACES=threads                    # each place is a single hardware thread
export OMP_PLACES={0,1},{2,3},{4,5},{6,7}    # Run process and its threads on given cores
export OMP_PLACES={0:$OMP_NUM_THREADS:2}
```

- **OMP_PROC_BIND** indicates how OpenMP® threads are bound to resources
  - Can be a comma separated list of `primary`, `close` or `spread`, indicating policies for nested levels of parallelism
  - Can be `false` to disable thread affinity

```
export OMP_PROC_BIND=close      # Bind threads close to primary thread on given places
export OMP_PROC_BIND=spread     # Spread threads evenly on given places
export OMP_PROC_BIND=primary    # Bind threads on the same place as the primary thread
```

- **OMP_DISPLAY_AFFINITY=TRUE** helps verify bindings
- **OMP_AFFINITY_FORMAT** helps define the format when displaying OpenMP affinity information

```
export OMP_AFFINITY_FORMAT="Thread Affinity: %0.3L %.8n %.15{thread_affinity} %.12H"
```

- More details can be found in the OpenMP® Specification: https://www.openmp.org/spec-html/5.0/openmpch6.htm

AMD
together we advance_

[Public]

Controlling Affinity for Serial Applications – GOMP_CPU_AFFINITY

- If using GNU OpenMP® implementation, we can set up CPU core affinity for a process and its threads using the environment variable, GOMP_CPU_AFFINITY

```
export GOMP_CPU_AFFINITY=0-64:4
export OMP_NUM_THREADS=16
./exe
```

In the above example, we expect the 16 threads of the process to be bound to cores 0, 4, 8, 12, 16, ... 60

- Note: Same setting can be used to define affinity of threads for each process in an MPI job as well

AMD
together we advance_

Case Studies: MPI + OpenMP® + HIP

Setting CPU + GPU affinity

[Public]

Controlling Affinity of MPI Applications

- OpenMPI
  - mpirun offers several options for process placement, order and binding
  - See manpage for **mpirun** for extensive documentation of all affinity related options

- Slurm
  - Slurm offers a rich set of options to control binding of tasks to hardware resources
  - See manpages for **srun** or **slurm.conf** for documentation of all affinity related options

- MPICH does not have many affinity control options
  - Use native process manager, `mpiexec.hydra`
  - Slurm integration using compile time option `"--with-pmi=slurm --with-pm=no"`

- Be ready to read man pages as options may change

AMD
together we advance_

[Public]

MPI with OpenMP® Example

See full code at: https://code.ornl.gov/olcf/hello_mpi_omp

```
/* ----------------------------------------------------------
MPI + OpenMP Hello, World program to help understand process
and thread mapping to physical CPU cores and hardware threads
---------------------------------------------------------- */
int main(int argc, char *argv[]){
    MPI_Init(&argc, &argv);
    int size;
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    char name[MPI_MAX_PROCESSOR_NAME];
    int resultlength;
    MPI_Get_processor_name(name, &resultlength);

    int hwthread;
    int thread_id = 0;
    #pragma omp parallel default(shared) private(hwthread, thread_id)
    {
        thread_id = omp_get_thread_num();
        hwthread  = sched_getcpu();
        printf("MPI %03d - OMP %03d - HWT %03d - Node %s\n", rank, thread_id, hwthread, name);
    }
    MPI_Finalize();
    return 0;
}
```

Sample output:

```
MPI 001 - OMP 000 - HWT 003 - Node nid007564
MPI 001 - OMP 001 - HWT 004 - Node nid007564
```


AMD
together we advance_

[Public]

MPI + OpenMP + HIP Example

See full code at: https://code.ornl.gov/olcf/hello_jobstep

```
printf("MPI %03d - OMP %03d - HWT %03d - Node %s - RT_GPU_ID %s - GPU_ID %s - Bus_ID %s\n",
                   rank, thread_id, hwthread, name, rt_gpu_id_list.c_str(), gpu_id_list, busid_list.c_str());
```

rank	MPI_Comm_rank
thread_id	omp_get_thread_num()
hwthread	sched_getcpu()
name	MPI_Get_processor_name
gpu_id	ROCR_VISIBLE_DEVICES
busid	hipDeviceGetPCIBusId
rt_gpu_id	HIP runtime GPU ID i.e, 0, 1, .. 7

Sample output:

```
MPI 000 - OMP 000 - HWT 001 - Node nid005116 - RT_GPU_ID 0 - GPU_ID 4 - Bus_ID d1
MPI 001 - OMP 000 - HWT 002 - Node nid005116 - RT_GPU_ID 0 - GPU_ID 5 - Bus_ID d6
```

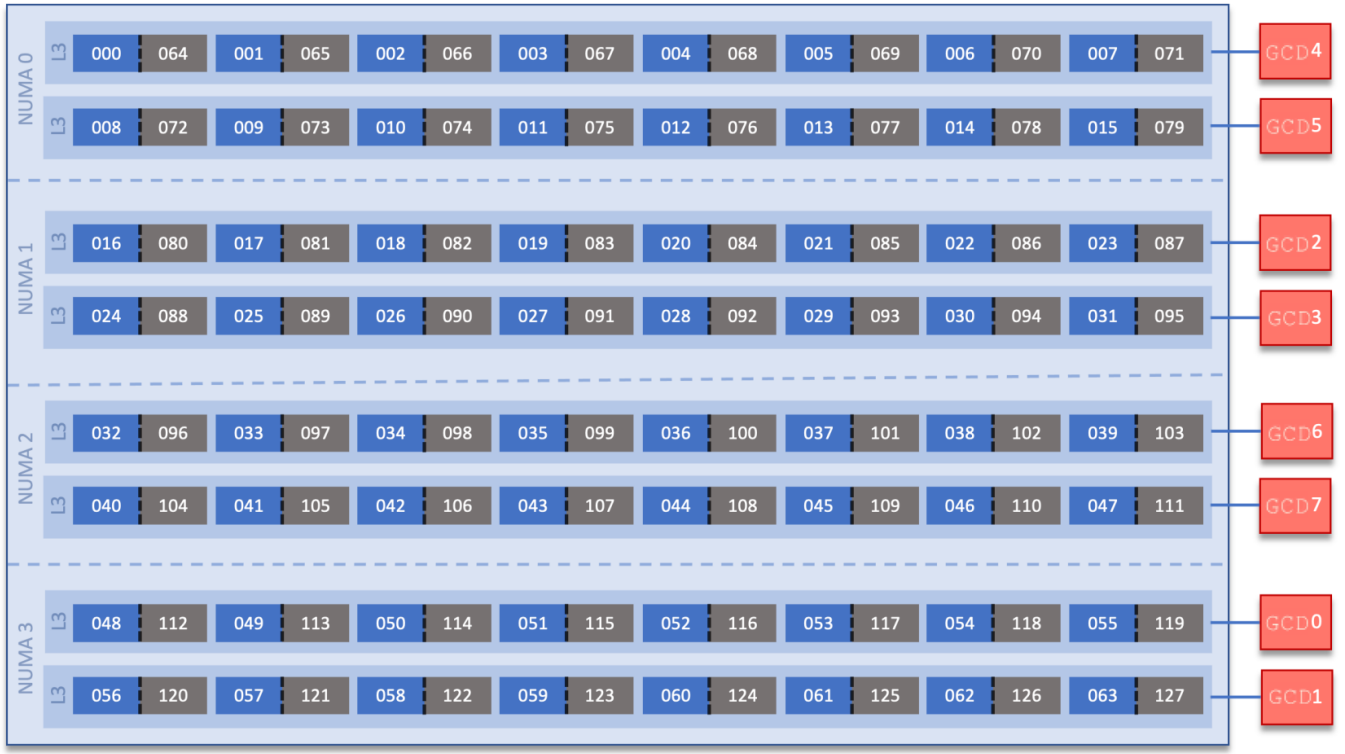

AMD
together we advance_

[Public]

Mapping Processes to GCDs on LUMI – Expected Mapping

- We need the following GCD to core mapping for optimal performance on LUMI, and we want to see a core picked from each set for each rank

GCD ID	0	1	2	3	4	5	6	7
CPU set	48-55	56-63	16-23	24-31	0-7	8-15	32-39	40-47



AMD
together we advance_

[Public]

Setting GPU Device Visibility on LUMI nodes

- By default, processes see all GPU devices. So, device visibility needs to be restricted for each process.
- May be able to allocate only some GPUs using Slurm – this sets `ROCR_VISIBLE_DEVICES` or `HIP_VISIBLE_DEVICES` to the set of GPUs requested depending on the site's Slurm configuration

- **`HIP_VISIBLE_DEVICES`** restricts GPU devices visible to the HIP runtime
- **`ROCR_VISIBLE_DEVICES`** restricts GPU devices visible to ROCr runtime
  - The HIP runtime depends on the ROCr runtime, so the HIP layer can only see the subset of devices selected by ROCR_VISIBLE_DEVICES

AMD
together we advance_

[Public]

Mapping Processes to GCDs on LUMI

A simple way: Initialize **ROCR_VISIBLE_DEVICES** using **SLURM_LOCALID**

Example script from **man mpi** on LUMI:

```
$ cat set_gpu_device.sh
#!/bin/bash
export ROCR_VISIBLE_DEVICES=$SLURM_LOCALID
exec $*
```

Expected mapping:

GCD ID	0	1	2	3	4	5	6	7
CPU set	48-55	56-63	16-23	24-31	0-7	8-15	32-39	40-47

Run with the script:

```
$ N=1; salloc -A $MYPROJ -p small-g -N $N --gpus $((N*8)) --threads-per-core 1 --mem 0 --exclusive -t 05:00
$ N=1;   srun -A $MYPROJ -p small-g -N $N --gpus $((N*8)) --threads-per-core 1 -n $((N*8)) ./set_gpu_device.sh
./hello_jobstep
```

```
MPI 000 - OMP 000 - HWT 001 - Node nid007556 - RT_GPU_ID 0 - GPU_ID 0 - Bus_ID c1
MPI 001 - OMP 000 - HWT 002 - Node nid007556 - RT_GPU_ID 0 - GPU_ID 1 - Bus_ID c6
MPI 002 - OMP 000 - HWT 003 - Node nid007556 - RT_GPU_ID 0 - GPU_ID 2 - Bus_ID c9
MPI 003 - OMP 000 - HWT 004 - Node nid007556 - RT_GPU_ID 0 - GPU_ID 3 - Bus_ID ce
MPI 004 - OMP 000 - HWT 005 - Node nid007556 - RT_GPU_ID 0 - GPU_ID 4 - Bus_ID d1
MPI 005 - OMP 000 - HWT 006 - Node nid007556 - RT_GPU_ID 0 - GPU_ID 5 - Bus_ID d6
MPI 006 - OMP 000 - HWT 007 - Node nid007556 - RT_GPU_ID 0 - GPU_ID 6 - Bus_ID d9
MPI 007 - OMP 000 - HWT 008 - Node nid007556 - RT_GPU_ID 0 - GPU_ID 7 - Bus_ID de
```

Rank 0 got HWT 1 and GCD 0

Mapping is not optimal

Only hardware threads 001-008 were selected

We got different GPU devices per task

AMD
together we advance_

[Public]

Mapping Processes to GCDs on LUMI

We need proper GPU + CPU affinity for each task. Use Slurm's `mask_cpu` binding option.

```
$ N=1; salloc -A $MYPROJ -p small-g -N $N --gpus $((N*8)) --threads-per-core 1 --mem 0 --exclusive -t 05:00
$ export OMP_NUM_THREADS=1
$ N=1; c=fe;
MYMASK="0x${c}000000000000,0x${c}00000000000000,0x${c}0000,0x${c}000000,0x${c},0x${c}00,0x${c}00000000,0x${c}00
00000000"; srun -A ${MYPROJ} -p small-g -N $N --gpus $((N*8)) --threads-per-core 1 -n $((N*8)) --cpu-
bind=mask_cpu:$MYMASK ./set_gpu_device.sh ./hello_jobstep
```

But how do I generate this mask??

Expected mapping:

GCD ID	0	1	2	3	4	5	6	7
CPU set	48-55	56-63	16-23	24-31	0-7	8-15	32-39	40-47

```
MPI 001 - OMP 000 - HWT 061 - Node nid007255 - RT_GPU_ID 0 - GPU_ID 1 - Bus_ID c6
MPI 003 - OMP 000 - HWT 028 - Node nid007255 - RT_GPU_ID 0 - GPU_ID 3 - Bus_ID ce
MPI 005 - OMP 000 - HWT 015 - Node nid007255 - RT_GPU_ID 0 - GPU_ID 5 - Bus_ID d6
MPI 006 - OMP 000 - HWT 034 - Node nid007255 - RT_GPU_ID 0 - GPU_ID 6 - Bus_ID d9
MPI 007 - OMP 000 - HWT 044 - Node nid007255 - RT_GPU_ID 0 - GPU_ID 7 - Bus_ID de
MPI 000 - OMP 000 - HWT 053 - Node nid007255 - RT_GPU_ID 0 - GPU_ID 0 - Bus_ID c1
MPI 002 - OMP 000 - HWT 022 - Node nid007255 - RT_GPU_ID 0 - GPU_ID 2 - Bus_ID c9
MPI 004 - OMP 000 - HWT 004 - Node nid007255 - RT_GPU_ID 0 - GPU_ID 4 - Bus_ID d1
```

One HWT from each core set is selected for each task

1 GPU device is selected per task

AMD
together we advance_

[Public]

Generating CPU Mask for Low Noise Mode

```
$ cat generate_mask.py
#!/usr/bin/env python3
cpu_of_rank_thread = [ # sparing first core of each 8-core CCD
     [49,50,51,52,53,54,55] , # local rank 0
     [57,58,59,60,61,62,63] , # local rank 1
     [17,18,19,20,21,22,23] , # local rank 2
     [25,26,27,28,29,30,31] , # local rank 3
     [ 1, 2, 3, 4, 5, 6, 7] , # local rank 4
     [ 9,10,11,12,13,14,15] , # local rank 5
     [33,34,35,36,37,38,39] , # local rank 6
     [41,42,43,44,45,46,47] ] # local rank 7
num_ranks = len(cpu_of_rank_thread)
mask = ""
for rank in range(num_ranks):
    sum = 0
    num_threads_this_rank = len(cpu_of_rank_thread[rank])
    for thread in range( num_threads_this_rank ):
        cpu = cpu_of_rank_thread[rank][thread]
        two_pow = 2 ** cpu
        sum += two_pow
        if  thread == num_threads_this_rank - 1:
            if  rank > 0:
                mask += ","
            mask += hex(sum)
    if  rank == num_ranks - 1:
        print("mask=", mask)
        print(mask.replace("0x",""))
```

In this example, we are skipping the first core of each CPU set

Sample output:

```
$ python3 generate_mask_lumi_order.py
mask=
0xfe000000000000,0xfe00000000000000,0xfe0000,0xfe000000,0xfe,0xfe00,
0xfe00000000,0xfe0000000000
```

Courtesy: Marcus Wagner, HPE

AMD
together we advance_

[Public]

Case Studies: 1 MPI rank per GCD, 2 OpenMP® threads per rank

```
$ export OMP_NUM_THREADS=2
$ export OMP_PROC_BIND=close
$ N=1; c=fe;
MYMASK="0x${c}000000000000,0x${c}00000000000000,0x${c}0000,0x${c}000000,0x${c},0x${c}00,0x${c}00000000,0x${c}0000000000"; srun -A
${MYPROJ} -p small-g -N $N --gpus $((N*8)) --threads-per-core 1 -n $((N*8)) --cpu-bind=mask_cpu:$MYMASK ./set_gpu_device.sh
./hello_jobstep

MPI 001 - OMP 000 - HWT 057 - Node nid007255 - RT_GPU_ID 0 - GPU_ID 1 - Bus_ID c6
MPI 001 - OMP 001 - HWT 058 - Node nid007255 - RT_GPU_ID 0 - GPU_ID 1 - Bus_ID c6
MPI 000 - OMP 000 - HWT 049 - Node nid007255 - RT_GPU_ID 0 - GPU_ID 0 - Bus_ID c1
MPI 000 - OMP 001 - HWT 050 - Node nid007255 - RT_GPU_ID 0 - GPU_ID 0 - Bus_ID c1
MPI 007 - OMP 000 - HWT 041 - Node nid007255 - RT_GPU_ID 0 - GPU_ID 7 - Bus_ID de
MPI 007 - OMP 001 - HWT 042 - Node nid007255 - RT_GPU_ID 0 - GPU_ID 7 - Bus_ID de
MPI 006 - OMP 000 - HWT 033 - Node nid007255 - RT_GPU_ID 0 - GPU_ID 6 - Bus_ID d9
MPI 006 - OMP 001 - HWT 034 - Node nid007255 - RT_GPU_ID 0 - GPU_ID 6 - Bus_ID d9
MPI 004 - OMP 000 - HWT 001 - Node nid007255 - RT_GPU_ID 0 - GPU_ID 4 - Bus_ID d1
MPI 004 - OMP 001 - HWT 002 - Node nid007255 - RT_GPU_ID 0 - GPU_ID 4 - Bus_ID d1
MPI 005 - OMP 000 - HWT 009 - Node nid007255 - RT_GPU_ID 0 - GPU_ID 5 - Bus_ID d6
MPI 005 - OMP 001 - HWT 010 - Node nid007255 - RT_GPU_ID 0 - GPU_ID 5 - Bus_ID d6
MPI 003 - OMP 000 - HWT 025 - Node nid007255 - RT_GPU_ID 0 - GPU_ID 3 - Bus_ID ce
MPI 003 - OMP 001 - HWT 026 - Node nid007255 - RT_GPU_ID 0 - GPU_ID 3 - Bus_ID ce
MPI 002 - OMP 000 - HWT 017 - Node nid007255 - RT_GPU_ID 0 - GPU_ID 2 - Bus_ID c9
MPI 002 - OMP 001 - HWT 018 - Node nid007255 - RT_GPU_ID 0 - GPU_ID 2 - Bus_ID c9
```

Combining OpenMP® settings with srun options, we can pin a separate core for each thread of each rank

Expected mapping:

GCD ID	0	1	2	3	4	5	6	7
CPU set	48-55	56-63	16-23	24-31	0-7	8-15	32-39	40-47

AMD
together we advance_

[Public]

Case Studies: 2 MPI ranks per GCD, 3 OpenMP® threads per rank Selecting GPU device

We need a new script to select GPU devices such that ranks are closely packed on GCDs (i.e., ranks 0 and 1 use GCD 0, ranks 2 and 3 use GCD 1, etc.)

```
$ cat set_gpu_device_multirank.sh
#!/bin/bash
export ranks_per_node=$(($SLURM_NTASKS/$SLURM_NNODES))
let NUM_GPUS=8
let ranks_per_gpu=$((${ranks_per_node}+${NUM_GPUS}-1)/${NUM_GPUS}))
let my_gpu=$(($SLURM_LOCALID/$ranks_per_gpu))
export ROCR_VISIBLE_DEVICES=$my_gpu
exec $*
```


AMD
together we advance_

[Public]

Case Studies: 2 MPI ranks per GCD, 3 OpenMP® threads per rank Generating the CPU Mask

```
$ cat generate_mask_lumi_order_16ranks.py
#!/usr/bin/env python3
cpu_of_rank_thread = [
   [49,50,51] , # local rank 0
   [52,53,54] , # local rank 1
   [57,58,59] , # local rank 2
   [60,61,62] , # local rank 3
   [17,18,19] , # local rank 4
   [20,21,22] , # local rank 5
   [25,26,27] , # local rank 6
   [28,29,30] , # local rank 7
   [ 1, 2, 3] , # local rank 8
   [ 4, 5, 6] , # local rank 9
   [ 9,10,11] , # local rank 10
   [12,13,14] , # local rank 11
   [33,34,35] , # local rank 12
   [36,37,38] , # local rank 13
   [41,42,43] , # local rank 14
   [44,45,46] ] # local rank 15
```

Skip first and last core of each 8-core set

```
num_ranks = len(cpu_of_rank_thread)
mask = ""
for rank in range(num_ranks):
    sum = 0
    num_threads_this_rank =
len(cpu_of_rank_thread[rank])
    for thread in range( num_threads_this_rank ):
        cpu = cpu_of_rank_thread[rank][thread]
        two_pow = 2 ** cpu
        sum += two_pow
        if  thread == num_threads_this_rank - 1:
            if  rank > 0:
                mask += ","
            mask += hex(sum)
    if  rank == num_ranks - 1:
        print("mask=", mask)
        print(mask.replace("0x",""))
```

```
$ python3 generate_mask_lumi_order_16ranks.py
mask=
0xe000000000000,0x70000000000000,0xe00000000000000,0x700000000000000,0xe0000,0x700000,0xe000000,0x70000000,0xe,0x70,0xe00,0x700
0,0xe00000000,0x7000000000,0xe0000000000,0x700000000000
```

Courtesy: Marcus Wagner, HPE

AMD
together we advance_

[Public]

Case Studies: 2 MPI ranks per GCD, 3 OpenMP® threads per rank

```
$ export OMP_NUM_THREADS=3
$ N=1;
MYMASK="0xe000000000000,0x7000000000000,0xe0000000000000,0x70000000000000,0xe0000,0x700000,0xe000000,0x70000000,0xe,0x70,0xe00,0
x7000,0xe00000000,0x7000000000,0xe0000000000,0x70000000000"; srun -A ${MYPROJ} -p small-g -N $N --gpus $((N*8)) --threads-per-core
1 -n $((N*8*2)) --cpu-bind=mask_cpu:$MYMASK ./set_gpu_device_multirank.sh ./hello_jobstep

<snip>

MPI 001 - OMP 000 - HWT 052 - Node nid007415 - RT_GPU_ID 0 - GPU_ID 0 - Bus_ID c1
MPI 001 - OMP 001 - HWT 053 - Node nid007415 - RT_GPU_ID 0 - GPU_ID 0 - Bus_ID c1
MPI 001 - OMP 002 - HWT 054 - Node nid007415 - RT_GPU_ID 0 - GPU_ID 0 - Bus_ID c1
MPI 003 - OMP 000 - HWT 060 - Node nid007415 - RT_GPU_ID 0 - GPU_ID 1 - Bus_ID c6
MPI 003 - OMP 002 - HWT 062 - Node nid007415 - RT_GPU_ID 0 - GPU_ID 1 - Bus_ID c6
MPI 003 - OMP 001 - HWT 061 - Node nid007415 - RT_GPU_ID 0 - GPU_ID 1 - Bus_ID c6
MPI 002 - OMP 000 - HWT 057 - Node nid007415 - RT_GPU_ID 0 - GPU_ID 1 - Bus_ID c6
MPI 002 - OMP 001 - HWT 058 - Node nid007415 - RT_GPU_ID 0 - GPU_ID 1 - Bus_ID c6
MPI 002 - OMP 002 - HWT 059 - Node nid007415 - RT_GPU_ID 0 - GPU_ID 1 - Bus_ID c6
MPI 000 - OMP 000 - HWT 049 - Node nid007415 - RT_GPU_ID 0 - GPU_ID 0 - Bus_ID c1
MPI 000 - OMP 001 - HWT 050 - Node nid007415 - RT_GPU_ID 0 - GPU_ID 0 - Bus_ID c1
MPI 000 - OMP 002 - HWT 051 - Node nid007415 - RT_GPU_ID 0 - GPU_ID 0 - Bus_ID c1
```

Ranks 0 and 1 got GCD 0,
Ranks 2 and 3 got GCD 1

Threads are closely packed according to specified mask

With NPS4, we want to get the full CPU socket bandwidth. We need to have processes/threads on each core in each NUMA domain.

In addition, we oversubscribe the GCD with 2 ranks to better utilize its resources.

AMD
together we advance_

[Public]

Summary

- In parallel applications, Affinity involves Placement, Order and Binding

- Affinity is important for hybrid applications on the complex architectures of today
  - Higher memory bandwidth
  - Lower latency
  - Optimize communication
  - Avoid excessive thread/process migration

- Affinity can be achieved in many ways
  - Need to know the architecture
  - Need to know the performance limiters of the application and design the best strategy to use resources
  - Need to know the communication pattern between processes
  - Need to know how to control placement using a combination of MPI, OpenMP®, Pthread, Slurm options

AMD
together we advance_

[Public]

References

- Frontier User Guide, Oak Ridge Leadership Compute Facility, Oak Ridge National Laboratory, https://docs.olcf.ornl.gov/systems/frontier_user_guide.html#
- Parallel and High Performance Computing, Robert Robey and Yuliana Zamora, Manning Publications, May 2021
- Essentials of Parallel Computing, Chapter 14 Code Examples: https://github.com/essentialsofparallelcomputing/Chapter14
- Code Examples from Tom Papatheodore, ORNL:
  - https://code.ornl.gov/olcf/hello_mpi_omp
  - https://code.ornl.gov/olcf/hello_jobstep
- OpenMP® Specification: https://www.openmp.org/
- MPICH, https://www.mpich.org/
- OpenMPI, https://www.open-mpi.org/
- Slurm, https://slurm.schedmd.com/
- Performance Analysis of CP2K Code for Ab Initio Molecular Dynamics on CPUs and GPUs, Dewi Yokelson, Nikolay V. Tkachenko, Robert Robey, Ying Wai Li, and Pavel A. Dub, *Journal of Chemical Information and Modeling* **2022** *62* (10), 2378-2386, DOI: 10.1021/acs.jcim.1c01538

AMD
together we advance_

[Public]

Disclaimer

The information presented in this document is for informational purposes only and may contain technical inaccuracies, omissions, and typographical errors. The information contained herein is subject to change and may be rendered inaccurate for many reasons, including but not limited to product and roadmap changes, component and motherboard version changes, new model and/or product releases, product differences between differing manufacturers, software changes, BIOS flashes, firmware upgrades, or the like. Any computer system has risks of security vulnerabilities that cannot be completely prevented or mitigated. AMD assumes no obligation to update or otherwise correct or revise this information. However, AMD reserves the right to revise this information and to make changes from time to time to the content hereof without obligation of AMD to notify any person of such revisions or changes.

THIS INFORMATION IS PROVIDED ‘AS IS.” AMD MAKES NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE CONTENTS HEREOF AND ASSUMES NO RESPONSIBILITY FOR ANY INACCURACIES, ERRORS, OR OMISSIONS THAT MAY APPEAR IN THIS INFORMATION. AMD SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, OR FITNESS FOR ANY PARTICULAR PURPOSE. IN NO EVENT WILL AMD BE LIABLE TO ANY PERSON FOR ANY RELIANCE, DIRECT, INDIRECT, SPECIAL, OR OTHER CONSEQUENTIAL DAMAGES ARISING FROM THE USE OF ANY INFORMATION CONTAINED HEREIN, EVEN IF AMD IS EXPRESSLY ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

AMD, the AMD Arrow logo, ROCm and combinations thereof are trademarks of Advanced Micro Devices, Inc. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

The OpenMP name and the OpenMP logo are registered trademarks of the OpenMP Architecture Review Board.

HPE is a registered trademark of Hewlett Packard Enterprise Company and/or its affiliates.

Linux is the registered trademark of Linus Torvalds in the U.S. and other countries.

© 2023 Advanced Micro Devices, Inc. All rights reserved.


AMD