AFFINITY – Placement, Order and Binding

Gina Sitaraman, Bob Robey

AMD @HLRS
Sept 25-28th, 2023

AMD
together we advance_

Authors and Contributors

- Tom Papatheodore, (previously ORNL, now at AMD)
- Marcus Wagner, HPE
- Alfio Lazzaro, HPE
- Georgios Markomanolis, AMD
- Bill Brantley, AMD
- Noel Chalmers, AMD
- Kjetil Haugen, AMD

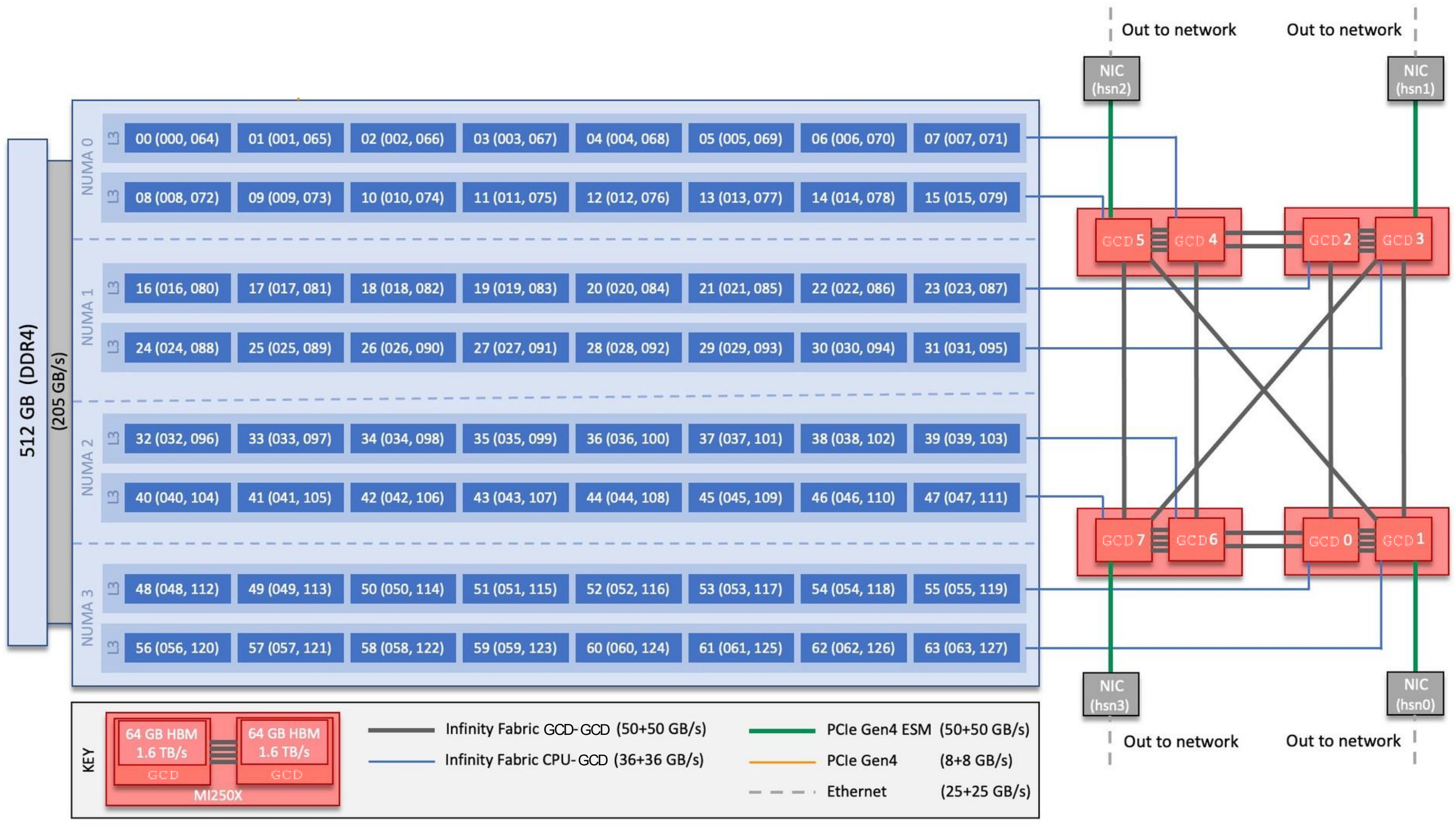
Agenda

- 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

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

LUMI Node Architecture

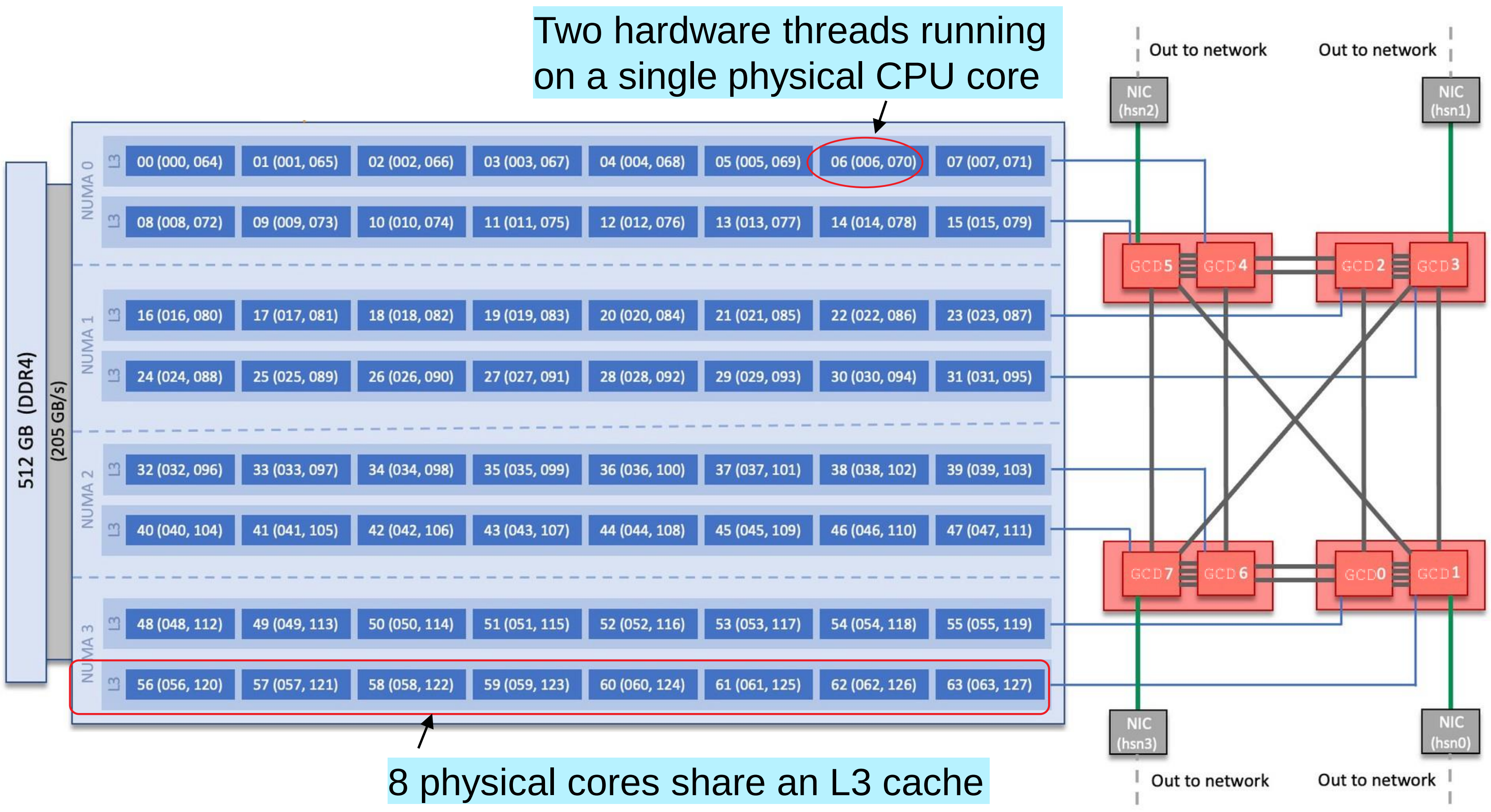


- 64 cores on a single socket CPU
- 4 MI250X GPUs, each with 2 GCDs
  - Each GCD is presented as a GPU device to `rocm-smi`
- 512 GB of DDR4 RAM
- Infinity Fabric™ links between GCDs and between GCDs and CPU cores
- 4 NICs attached to odd numbered GCDs

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

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_

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

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?
  - Improved cache reuse
  - Improved 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

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 one hardware thread (HWT) on each physical core
  - 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

- 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 by the OS

- Affinity controls in the OS and MPI have greatly improved and changed

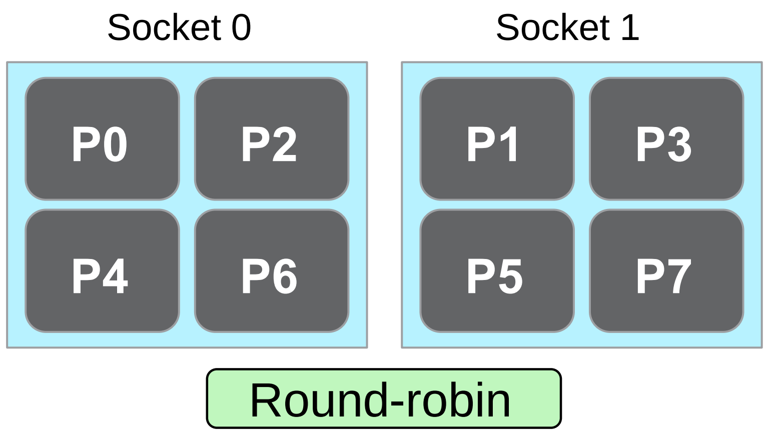
Order of Processes

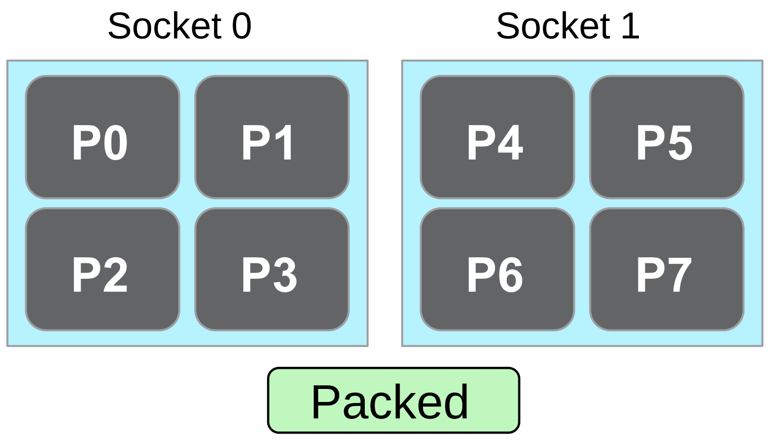
- Order defines how processes of a parallel job are distributed across the sockets of the node
  - Lower latency if processes communicating with each other are close together
  - Load balancing heavy workloads by scattering across compute resources

- **Round-robin** or **Cyclic**:
  - Processes are distributed in a round-robin fashion across sockets
  - 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
  - Improved performance due to data locality if ranks that communicate the most are accessing data in the same memory node and sharing cache

8 MPI ranks and 2 sockets



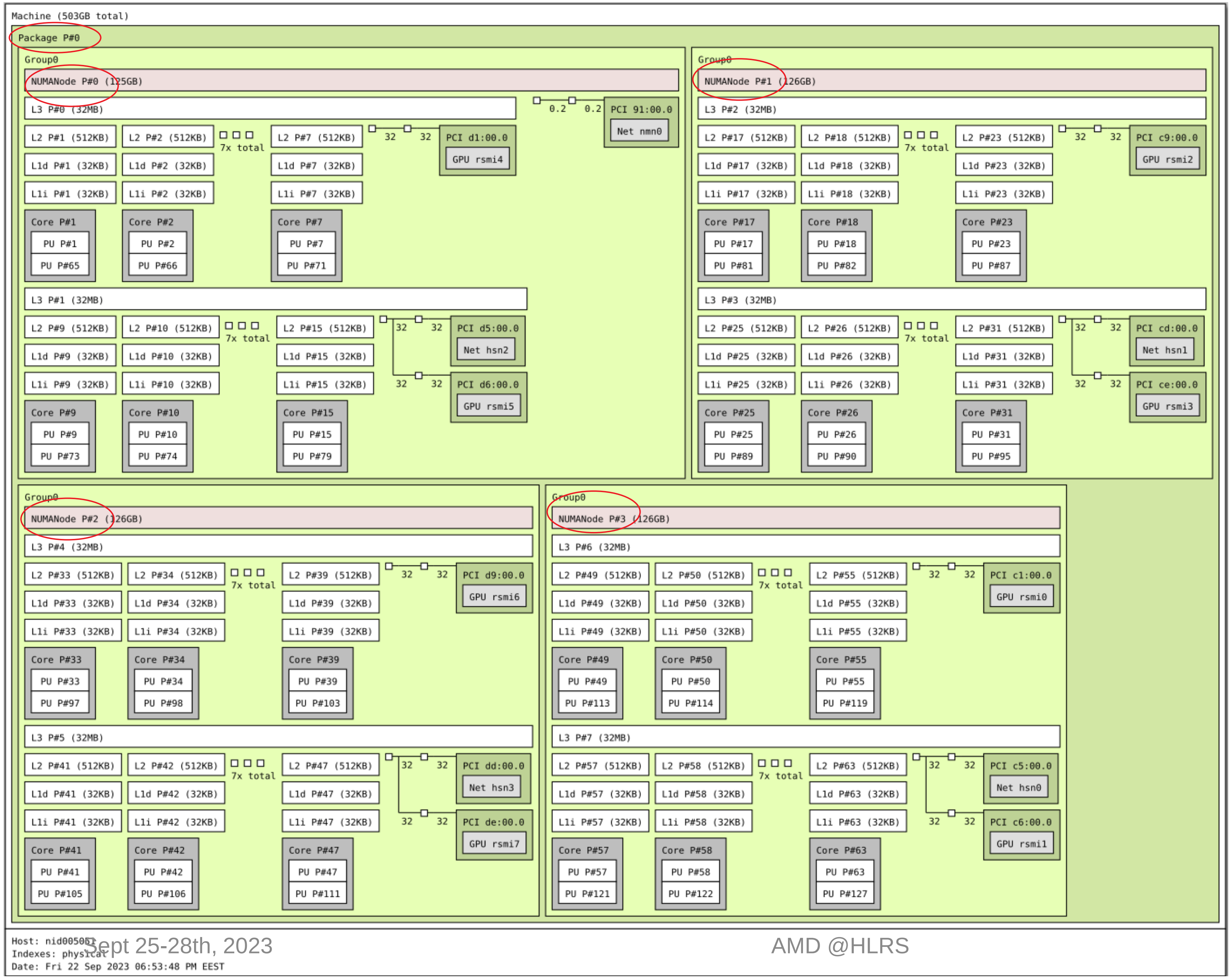


Understanding Node Topology

Understanding Node Topology

- On any given 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.

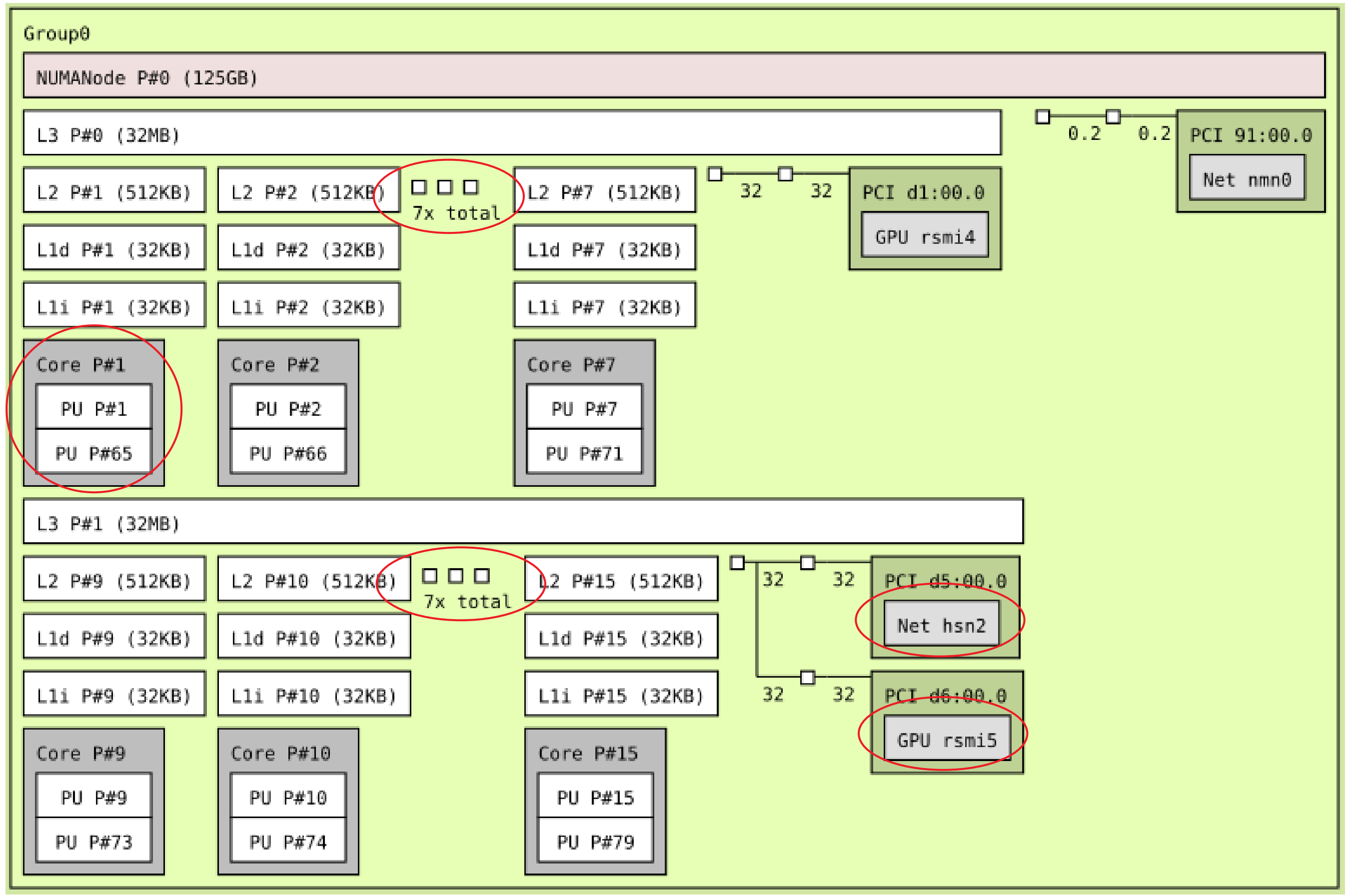


LUMI Node Topology

- `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 :)

Understanding Node Topology – lstopo NUMA domain #1



- 8 physical cores + 8 virtual cores share an L3 cache
- First core in each set is reserved for system operations
- Two sets of 8 physical cores in a NUMA domain
- Two GCDs in a NUMA domain
- One high-speed NIC per NUMA domain

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
Model name:                      AMD EPYC 7A53 64-Core Processor
L1d cache:                       2 MiB
L1i cache:                       2 MiB
L2 cache:                        32 MiB
L3 cache:                        256 MiB
NUMA node(s):                    4
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_

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
```

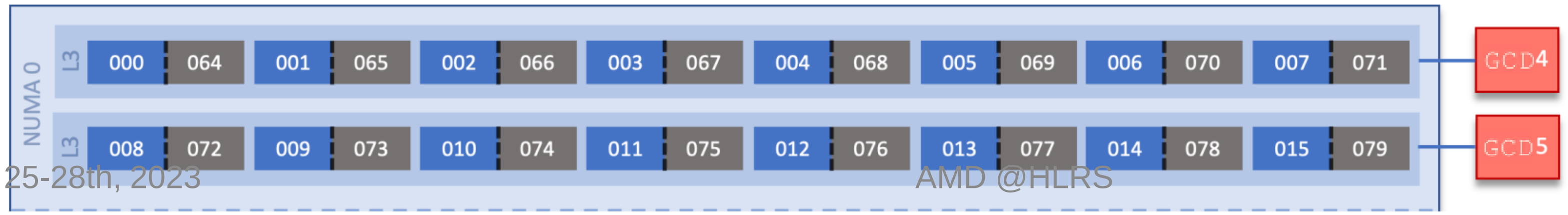
More obvious on multiple socket nodes

Understanding NUMA Configuration for GPUs

rocm-smi --showtoponuma

```
================================== 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



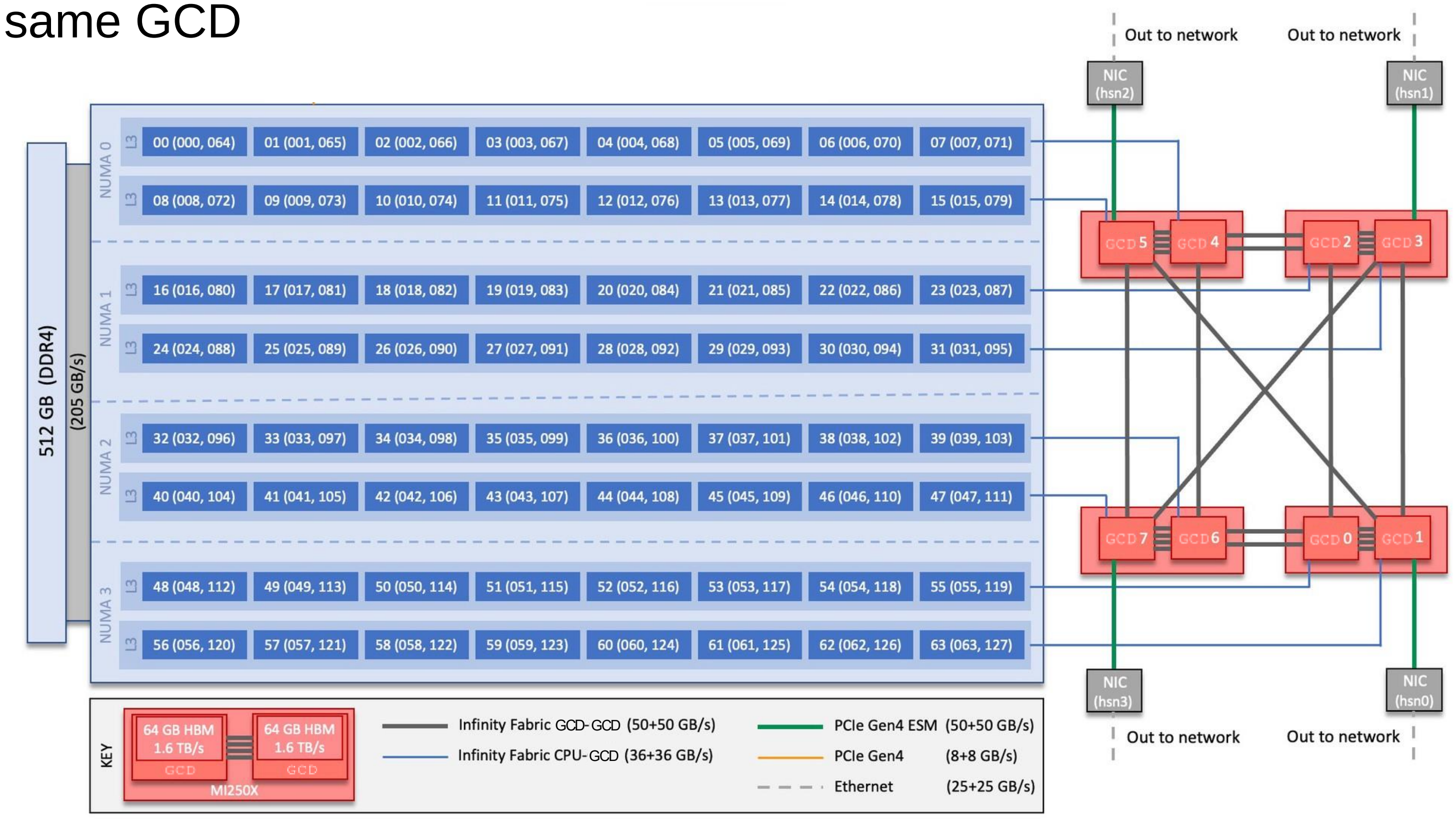
AMD
together we advance_

Placement Considerations on LUMI nodes

Placement Considerations on LUMI nodes

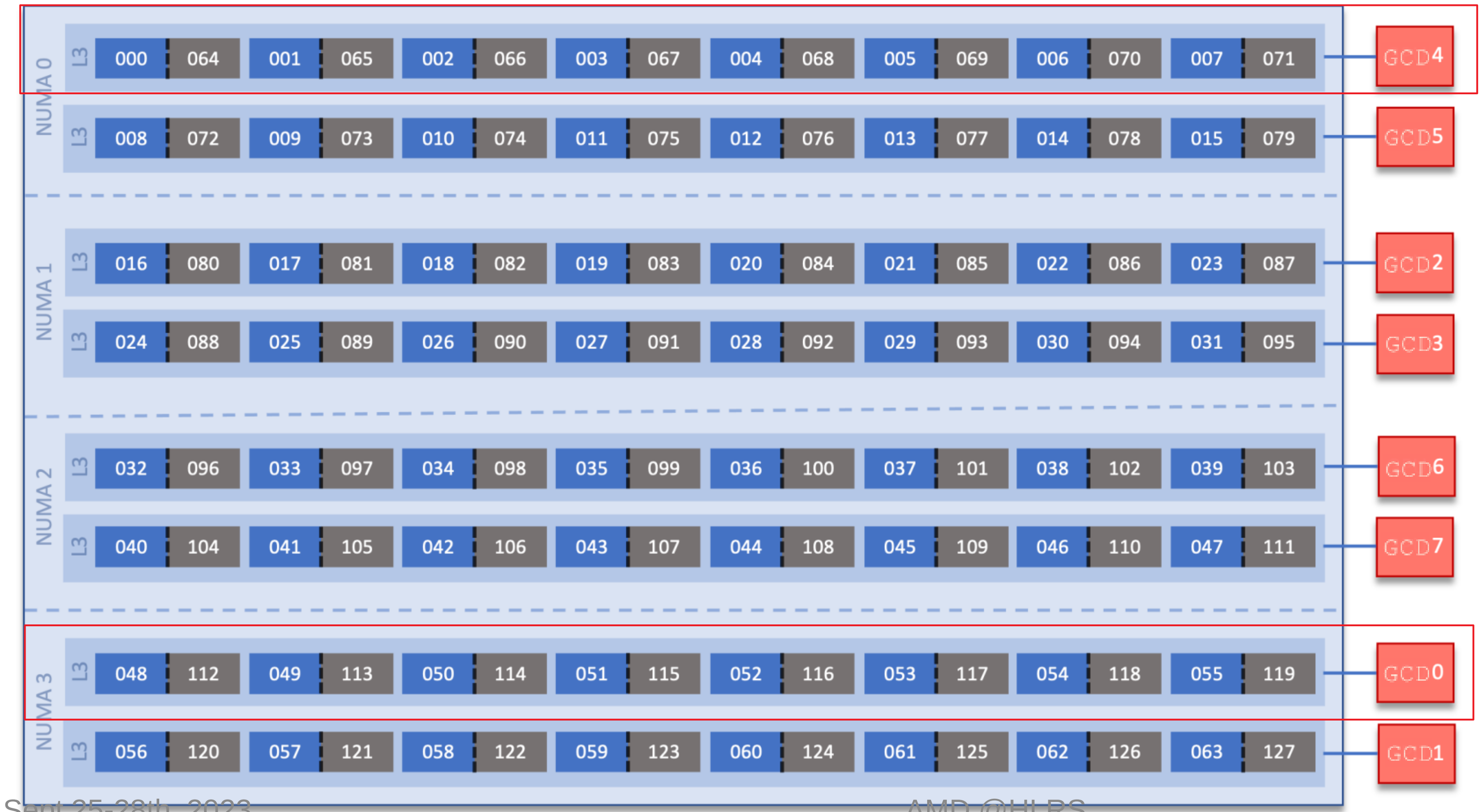
- Each GCD is connected to one of the NUMA domains via a high-speed Infinity Fabric™ link
- Memory bandwidth is highest between GCDs of the same MI250X GPU
- NICs are directly connected to odd numbered GCDs
- Multiple processes can run on the same GCD

Choose rank order and placement carefully to optimize communication



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

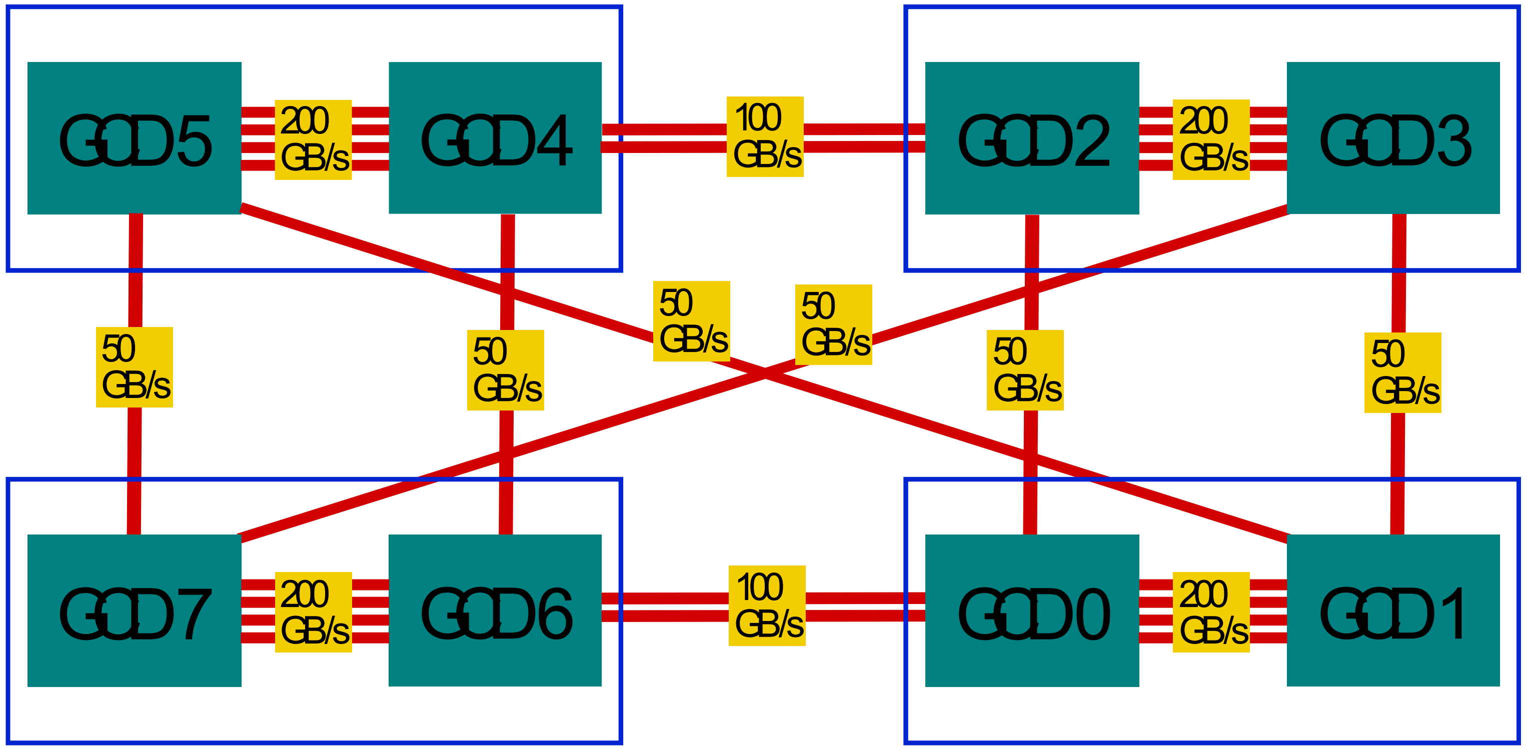

NUMA 0
L3 000 064 001 065 002 066 003 067 004 068 005 069 006 070 007 071 GCD4
L3 008 072 009 073 010 074 011 075 012 076 013 077 014 078 015 079 GCD5
NUMA 1
L3 016 080 017 081 018 082 019 083 020 084 021 085 022 086 023 087 GCD2
L3 024 088 025 089 026 090 027 091 028 092 029 093 030 094 031 095 GCD3
NUMA 2
L3 032 096 033 097 034 098 035 099 036 100 037 101 038 102 039 103 GCD6
L3 040 104 041 105 042 106 043 107 044 108 045 109 046 110 047 111 GCD7
NUMA 3
L3 048 112 049 113 050 114 051 115 052 116 053 117 054 118 055 119 GCD0
L3 056 120 057 121 058 122 059 123 060 124 061 125 062 126 063 127 GCD1

Cores 0-7 are closest to GCD 4

Cores 48-55 are closest to GCD 0

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

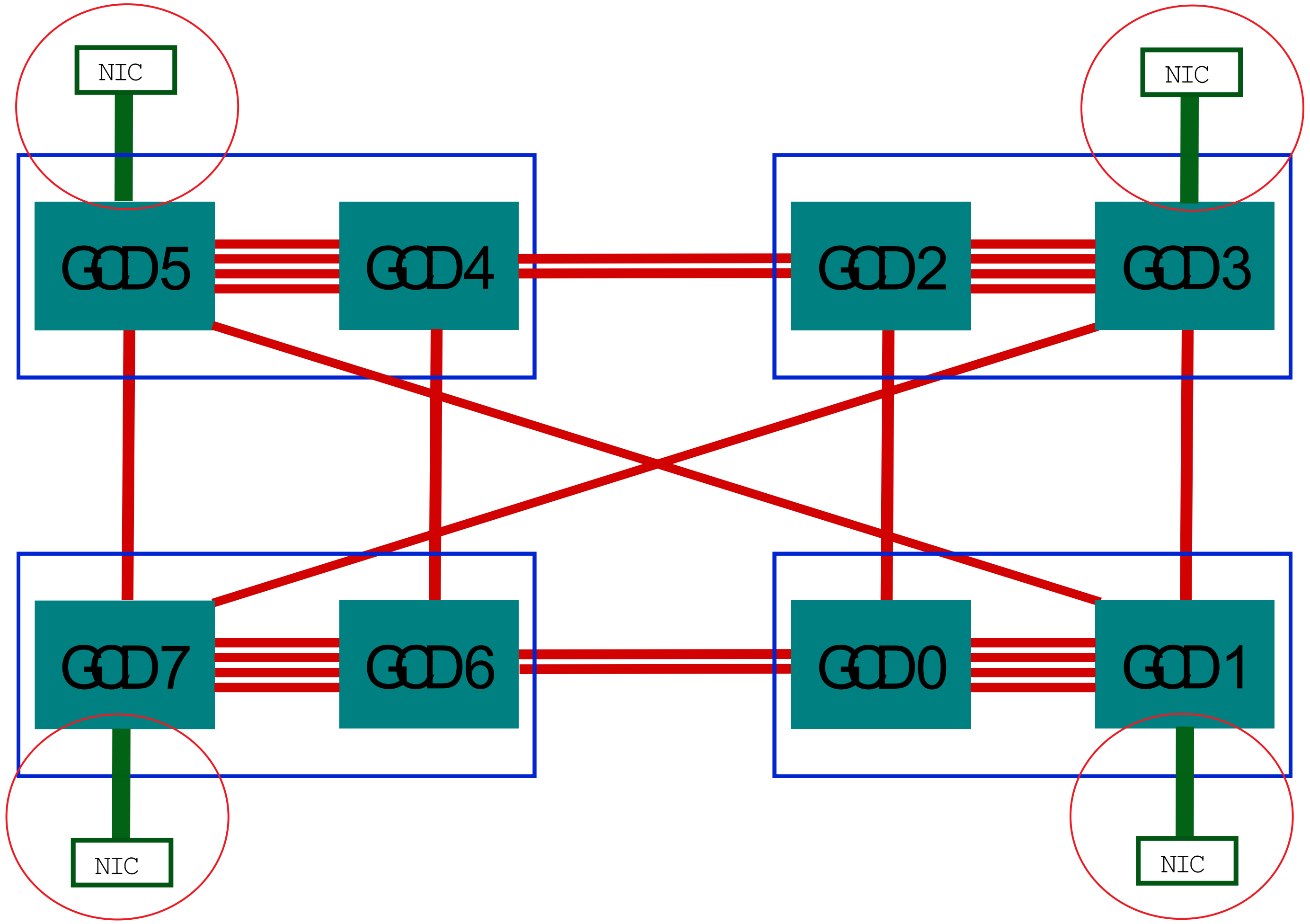


- Peak Bandwidth in each direction of Infinity Fabric™ link shown
- Even though bandwidths are different between GCDs, communication using device buffers will be at least as fast as communication using host buffers

AMD together we advance_

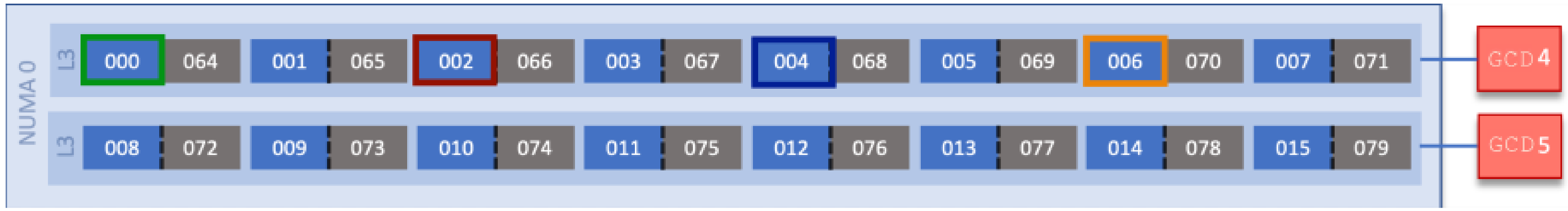
Placement Considerations on LUMI nodes

- 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



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

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

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, usually run prior to the application in the same Slurm batch job: https://github.com/olcf/XC30-Training/blob/master/affinity/Xthi.c
- Example code from Bob Robey's book, Essentials of Parallel Computing, Chapter 14 can be used to verify mappings for OpenMP®, MPI and MPI+OpenMP cases: https://github.com/essentialsofparallelcomputing/Chapter14

Tip: To check if you got affinity right in your MPI job, run `hello_jobstep` with the same configuration before running your job

AMD together we advance_

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®**
  - Using mpirun and OpenMP® binding options
    - 4 MPI ranks, 2 OpenMP threads per rank
- **MPI Applications + OpenMP® + HIP**
  - Using a helper script and mpirun binding options
    - 1 MPI rank per GPU, 2 OpenMP threads per rank
    - 2 MPI ranks per GPU, 8 OpenMP threads per rank
- Pivoting to resources we are going to use today
  - Check the configuration of the node you get

Topology of AAC node:

```
$ lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                128
On-line CPU(s) list:   0-127
Thread(s) per core:    1
Core(s) per socket:    64
Socket(s):             2
NUMA node(s):          2
Vendor ID:             AuthenticAMD
CPU family:            25
Model:                 1
Model name:            AMD EPYC 7763 64-Core Processor
Stepping:              1
CPU MHz:               2450.000
CPU max MHz:           3529.0520
CPU min MHz:           1500.0000
BogoMIPS:              4891.15
Virtualization:        AMD-V
L1d cache:             32K
L1i cache:             32K
L2 cache:              512K
L3 cache:              32768K
NUMA node0 CPU(s):     0-63
NUMA node1 CPU(s):     64-127
```

```
$ numactl -H
available: 2 nodes (0-1)
node 0 cpus: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38
39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58
59 60 61 62 63
node 0 size: 257523 MB
node 0 free: 250999 MB
node 1 cpus: 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79
80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99
100 101 102 103 104 105 106 107 108 109 110 111 112 113 114
115 116 117 118 119 120 121 122 123 124 125 126 127
node 1 size: 258029 MB
node 1 free: 251589 MB
node distances:
node   0   1
  0:  10  32
  1:  32  10
```

```
$ rocm-smi --showtoponuma
======================= ROCm System Management Interface =======================
================================== Numa Nodes ==================================
GPU[0]          : (Topology) Numa Node: 0
GPU[0]          : (Topology) Numa Affinity: 0
GPU[1]          : (Topology) Numa Node: 0
GPU[1]          : (Topology) Numa Affinity: 0
GPU[2]          : (Topology) Numa Node: 0
GPU[2]          : (Topology) Numa Affinity: 0
GPU[3]          : (Topology) Numa Node: 0
GPU[3]          : (Topology) Numa Affinity: 0
GPU[4]          : (Topology) Numa Node: 1
GPU[4]          : (Topology) Numa Affinity: 1
GPU[5]          : (Topology) Numa Node: 1
GPU[5]          : (Topology) Numa Affinity: 1
GPU[6]          : (Topology) Numa Node: 1
GPU[6]          : (Topology) Numa Affinity: 1
GPU[7]          : (Topology) Numa Node: 1
GPU[7]          : (Topology) Numa Affinity: 1
============================= End of ROCm SMI Log ==============================
```

- 2 sockets, 64 cores each, 1 HWT per core
- NPS1: 2 NUMA domains, 64 cores each
- 8 MI210 GPUs
- NUMA affinity = 4 GPUs per socket

AMD together we advance_

Case Studies: Serial Application + OpenMP®

Setting CPU Affinity

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`

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
```

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

AMD
together we advance_

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

Case Studies: MPI + OpenMP® + HIP

Setting CPU + GPU affinity

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

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 003 - OMP 000 - HWT 024 - Node smc-r06-06
MPI 003 - OMP 001 - HWT 025 - Node smc-r06-06
```

Case Study: 4 MPI Ranks, 2 OpenMP® Threads Per Rank

```
OMP_NUM_THREADS=2 OMP_PROC_BIND=close mpirun -np 4 --map-by L3cache ./hello_mpi_omp
MPI 001 - OMP 000 - HWT 008 - Node smc-r06-06
MPI 001 - OMP 001 - HWT 009 - Node smc-r06-06
MPI 000 - OMP 000 - HWT 000 - Node smc-r06-06
MPI 000 - OMP 001 - HWT 001 - Node smc-r06-06
MPI 002 - OMP 000 - HWT 016 - Node smc-r06-06
MPI 002 - OMP 001 - HWT 017 - Node smc-r06-06
MPI 003 - OMP 000 - HWT 024 - Node smc-r06-06
MPI 003 - OMP 001 - HWT 025 - Node smc-r06-06
```

- OMP settings + mpirun options help achieve binding we need
- Each rank gets 2 cores from a different core set

```
$ lscpu
Architecture:         x86_64
CPU op-mode(s):       32-bit, 64-bit
Byte Order:           Little Endian
CPU(s):               128
On-line CPU(s) list: 0-127
Thread(s) per core:   1
Core(s) per socket:   64
Socket(s):            2
NUMA node(s):         2

NUMA node0 CPU(s):    0-63
NUMA node1 CPU(s):    64-127
```

CPU architecture for reference

AMD together we advance_

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	Local GPU device IDs i.e, 0, 1, .. 7

Sample output:

```
MPI 002 - OMP 000 - HWT 032 - Node smc-r06-07 - RT_GPU_ID 0 - GPU_ID 2 - Bus_ID 03
MPI 002 - OMP 001 - HWT 033 - Node smc-r06-07 - RT_GPU_ID 0 - GPU_ID 2 - Bus_ID 03
```

Mapping Processes to GPUs – Expected Mapping

- We will use the following GPU to core mapping for optimal performance on this node, and we want to see a core picked from each set for each rank

GPU ID	0	1	2	3	4	5	6	7
CPU set	0-15	16-31	32-47	48-63	64-79	80-95	96-111	112-127

```
$ rocm-smi --showtoponuma
======================= ROCm System Management Interface =======================
================================== Numa Nodes ==================================
GPU[0]          : (Topology) Numa Node: 0
GPU[0]          : (Topology) Numa Affinity: 0
GPU[1]          : (Topology) Numa Node: 0
GPU[1]          : (Topology) Numa Affinity: 0
GPU[2]          : (Topology) Numa Node: 0
GPU[2]          : (Topology) Numa Affinity: 0
GPU[3]          : (Topology) Numa Node: 0
GPU[3]          : (Topology) Numa Affinity: 0
GPU[4]          : (Topology) Numa Node: 1
GPU[4]          : (Topology) Numa Affinity: 1
GPU[5]          : (Topology) Numa Node: 1
GPU[5]          : (Topology) Numa Affinity: 1
GPU[6]          : (Topology) Numa Node: 1
GPU[6]          : (Topology) Numa Affinity: 1
GPU[7]          : (Topology) Numa Node: 1
GPU[7]          : (Topology) Numa Affinity: 1
============================= End of ROCm SMI Log ==============================
```

On this node, we have 8 MI210 GPUs and 2 CPUs with 64 cores each

GPUs 0-3 are closest to NUMA node 0 and GPUs 4-7 are closest to NUMA node 1

AMD together we advance_

Setting GPU Device Visibility

- 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

```
======================= ROCm System Management Interface ========================
================================= Concise Info ===================================
GPU  Temp (DieEdge)  AvgPwr  SCLK    MCLK     Fan  Perf  PwrCap  VRAM%  GPU%
0    45.0c           40.0W   800Mhz  1600Mhz  0%   auto  300.0W    0%   0%
1    49.0c           41.0W   800Mhz  1600Mhz  0%   auto  300.0W    0%   0%
2    48.0c           40.0W   800Mhz  1600Mhz  0%   auto  300.0W    0%   0%
3    43.0c           41.0W   800Mhz  1600Mhz  0%   auto  300.0W    0%   0%
4    42.0c           41.0W   800Mhz  1600Mhz  0%   auto  300.0W    0%   0%
5    46.0c           40.0W   800Mhz  1600Mhz  0%   auto  300.0W    0%   0%
6    46.0c           40.0W   800Mhz  1600Mhz  0%   auto  300.0W    0%   0%
7    44.0c           40.0W   800Mhz  1600Mhz  0%   auto  300.0W    0%   0%
==================================================================================
============================= End of ROCm SMI Log ================================
```

← 8 MI210 GPUs

Helper Script to set CPU and GPU Affinity

```
#!/bin/bash
export global_rank=${OMPI_COMM_WORLD_RANK}
export local_rank=${OMPI_COMM_WORLD_LOCAL_RANK}
export ranks_per_node=${OMPI_COMM_WORLD_LOCAL_SIZE}

if [ -z "${NUM_CPUS}" ]; then
    let NUM_CPUS=128
fi

if [ -z "${RANK_STRIDE}" ]; then
    let RANK_STRIDE=$(( ${NUM_CPUS}/${ranks_per_node} ))
fi

if [ -z "${OMP_STRIDE}" ]; then
    let OMP_STRIDE=1
fi

if [ -z "${NUM_GPUS}" ]; then
    let NUM_GPUS=8
fi

if [ -z "${GPU_START}" ]; then
    let GPU_START=0
fi

if [ -z "${GPU_STRIDE}" ]; then
    let GPU_STRIDE=1
fi
```

```
cpu_list=($(seq 0 127))
let cpus_per_gpu=${NUM_CPUS}/${NUM_GPUS}
let cpu_start_index=$((
($RANK_STRIDE*${local_rank})+${GPU_START}*$cpus_per_gpu ))
let cpu_start=${cpu_list[$cpu_start_index]}
let cpu_stop=$(($cpu_start+$OMP_NUM_THREADS*$OMP_STRIDE-1))

gpu_list=(0 1 2 3 4 5 6 7)
let ranks_per_gpu=$(((${ranks_per_node}+${NUM_GPUS}-1)/${NUM_GPUS}))
let my_gpu_index=$(($local_rank*$GPU_STRIDE/$ranks_per_gpu))+${GPU_START}
let my_gpu=${gpu_list[${my_gpu_index}]}

export GOMP_CPU_AFFINITY=$cpu_start-$cpu_stop:$OMP_STRIDE

export ROCR_VISIBLE_DEVICES=$my_gpu

"$@"
```

CPU affinity

GPU affinity

Powerful script, can be tuned for a variety of pinning needs:

- GPU_START=4 to use last 4 GPUs and second socket only
- RANK_STRIDE to specify starting core for each rank
- OMP_STRIDE>1 to place threads of a process farther apart
- NUM_CPUS, NUM_GPUS can be set to reflect your system

Case Study: 1 MPI rank per GPU, 2 OpenMP® threads per rank

```
$ OMP_NUM_THREADS=2 mpirun -np 8 --bind-to none ./helper.sh ./hello_jobstep
MPI 001 - OMP 000 - HWT 016 - Node smc-r06-07 - RT_GPU_ID 0 - GPU_ID 1 - Bus_ID 43
MPI 001 - OMP 001 - HWT 017 - Node smc-r06-07 - RT_GPU_ID 0 - GPU_ID 1 - Bus_ID 43
MPI 007 - OMP 000 - HWT 112 - Node smc-r06-07 - RT_GPU_ID 0 - GPU_ID 7 - Bus_ID a3
MPI 007 - OMP 001 - HWT 113 - Node smc-r06-07 - RT_GPU_ID 0 - GPU_ID 7 - Bus_ID a3
MPI 006 - OMP 000 - HWT 096 - Node smc-r06-07 - RT_GPU_ID 0 - GPU_ID 6 - Bus_ID 83
MPI 006 - OMP 001 - HWT 097 - Node smc-r06-07 - RT_GPU_ID 0 - GPU_ID 6 - Bus_ID 83
MPI 000 - OMP 000 - HWT 000 - Node smc-r06-07 - RT_GPU_ID 0 - GPU_ID 0 - Bus_ID 63
MPI 000 - OMP 001 - HWT 001 - Node smc-r06-07 - RT_GPU_ID 0 - GPU_ID 0 - Bus_ID 63
MPI 005 - OMP 000 - HWT 080 - Node smc-r06-07 - RT_GPU_ID 0 - GPU_ID 5 - Bus_ID c3
MPI 005 - OMP 001 - HWT 081 - Node smc-r06-07 - RT_GPU_ID 0 - GPU_ID 5 - Bus_ID c3
MPI 004 - OMP 000 - HWT 064 - Node smc-r06-07 - RT_GPU_ID 0 - GPU_ID 4 - Bus_ID e3
MPI 004 - OMP 001 - HWT 065 - Node smc-r06-07 - RT_GPU_ID 0 - GPU_ID 4 - Bus_ID e3
MPI 002 - OMP 000 - HWT 032 - Node smc-r06-07 - RT_GPU_ID 0 - GPU_ID 2 - Bus_ID 03
MPI 002 - OMP 001 - HWT 033 - Node smc-r06-07 - RT_GPU_ID 0 - GPU_ID 2 - Bus_ID 03
MPI 003 - OMP 000 - HWT 048 - Node smc-r06-07 - RT_GPU_ID 0 - GPU_ID 3 - Bus_ID 26
MPI 003 - OMP 001 - HWT 049 - Node smc-r06-07 - RT_GPU_ID 0 - GPU_ID 3 - Bus_ID 26
```

- The helper script sets CPU+GPU affinity
- mpirun option `--bind-to none` is necessary
- Each rank uses a different GPU
- Threads are closely packed, 2 from each core set for each rank

Expected mapping:

GPU ID	0	1	2	3	4	5	6	7
CPU set	0-15	16-31	32-47	48-63	64-79	80-95	96-111	112-127

Case Study: 2 MPI ranks per GPU, 8 OpenMP® threads per rank

```
$ OMP_NUM_THREADS=8 mpirun -np 16 -bind-to none ./helper.sh ./hello_jobstep
MPI 000 - OMP 000 - HWT 000 - Node smc-r06-07 - RT_GPU_ID 0 - GPU_ID 0 - Bus_ID 63
MPI 000 - OMP 005 - HWT 005 - Node smc-r06-07 - RT_GPU_ID 0 - GPU_ID 0 - Bus_ID 63
MPI 000 - OMP 007 - HWT 007 - Node smc-r06-07 - RT_GPU_ID 0 - GPU_ID 0 - Bus_ID 63
MPI 000 - OMP 001 - HWT 001 - Node smc-r06-07 - RT_GPU_ID 0 - GPU_ID 0 - Bus_ID 63
MPI 000 - OMP 006 - HWT 006 - Node smc-r06-07 - RT_GPU_ID 0 - GPU_ID 0 - Bus_ID 63
MPI 000 - OMP 003 - HWT 003 - Node smc-r06-07 - RT_GPU_ID 0 - GPU_ID 0 - Bus_ID 63
MPI 000 - OMP 002 - HWT 002 - Node smc-r06-07 - RT_GPU_ID 0 - GPU_ID 0 - Bus_ID 63
MPI 000 - OMP 004 - HWT 004 - Node smc-r06-07 - RT_GPU_ID 0 - GPU_ID 0 - Bus_ID 63
MPI 001 - OMP 000 - HWT 008 - Node smc-r06-07 - RT_GPU_ID 0 - GPU_ID 0 - Bus_ID 63
MPI 001 - OMP 003 - HWT 011 - Node smc-r06-07 - RT_GPU_ID 0 - GPU_ID 0 - Bus_ID 63
MPI 001 - OMP 005 - HWT 013 - Node smc-r06-07 - RT_GPU_ID 0 - GPU_ID 0 - Bus_ID 63
MPI 001 - OMP 001 - HWT 009 - Node smc-r06-07 - RT_GPU_ID 0 - GPU_ID 0 - Bus_ID 63
MPI 001 - OMP 002 - HWT 010 - Node smc-r06-07 - RT_GPU_ID 0 - GPU_ID 0 - Bus_ID 63
MPI 001 - OMP 007 - HWT 015 - Node smc-r06-07 - RT_GPU_ID 0 - GPU_ID 0 - Bus_ID 63
MPI 001 - OMP 006 - HWT 014 - Node smc-r06-07 - RT_GPU_ID 0 - GPU_ID 0 - Bus_ID 63
MPI 001 - OMP 004 - HWT 012 - Node smc-r06-07 - RT_GPU_ID 0 - GPU_ID 0 - Bus_ID 63
MPI 002 - OMP 000 - HWT 016 - Node smc-r06-07 - RT_GPU_ID 0 - GPU_ID 1 - Bus_ID 43
MPI 002 - OMP 003 - HWT 019 - Node smc-r06-07 - RT_GPU_ID 0 - GPU_ID 1 - Bus_ID 43
MPI 002 - OMP 007 - HWT 023 - Node smc-r06-07 - RT_GPU_ID 0 - GPU_ID 1 - Bus_ID 43
MPI 002 - OMP 001 - HWT 017 - Node smc-r06-07 - RT_GPU_ID 0 - GPU_ID 1 - Bus_ID 43
MPI 002 - OMP 002 - HWT 018 - Node smc-r06-07 - RT_GPU_ID 0 - GPU_ID 1 - Bus_ID 43
MPI 002 - OMP 004 - HWT 020 - Node smc-r06-07 - RT_GPU_ID 0 - GPU_ID 1 - Bus_ID 43
MPI 002 - OMP 005 - HWT 021 - Node smc-r06-07 - RT_GPU_ID 0 - GPU_ID 1 - Bus_ID 43
MPI 002 - OMP 006 - HWT 022 - Node smc-r06-07 - RT_GPU_ID 0 - GPU_ID 1 - Bus_ID 43
<snip>
```

Threads are closely packed

Ranks 0 and 1 got GPU 0

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 GPU with 2 ranks to better utilize its resources.

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

References

- Frontier User Guide, Oak Ridge Leadership Compute Facility, Oak Ridge National Laboratory (ORNL), https://docs.olcf.ornl.gov/systems/frontier_user_guide.html#id2#id2
- 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 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_

Disclaimer

AMD
together we advance_

AMD