Hewlett Packard Enterprise

LUMI Architecture, programming and runtime environment

LUMI Advanced Workshop

March 5–7, 2025

Agenda

- LUMI Architecture

Recap on

- The Cray Programming Environment
- Controlling the Environment with modules
- Running a job
  - Assuming you already know about the basics of using LUMI

LUMI System

Image © CSC, Finland

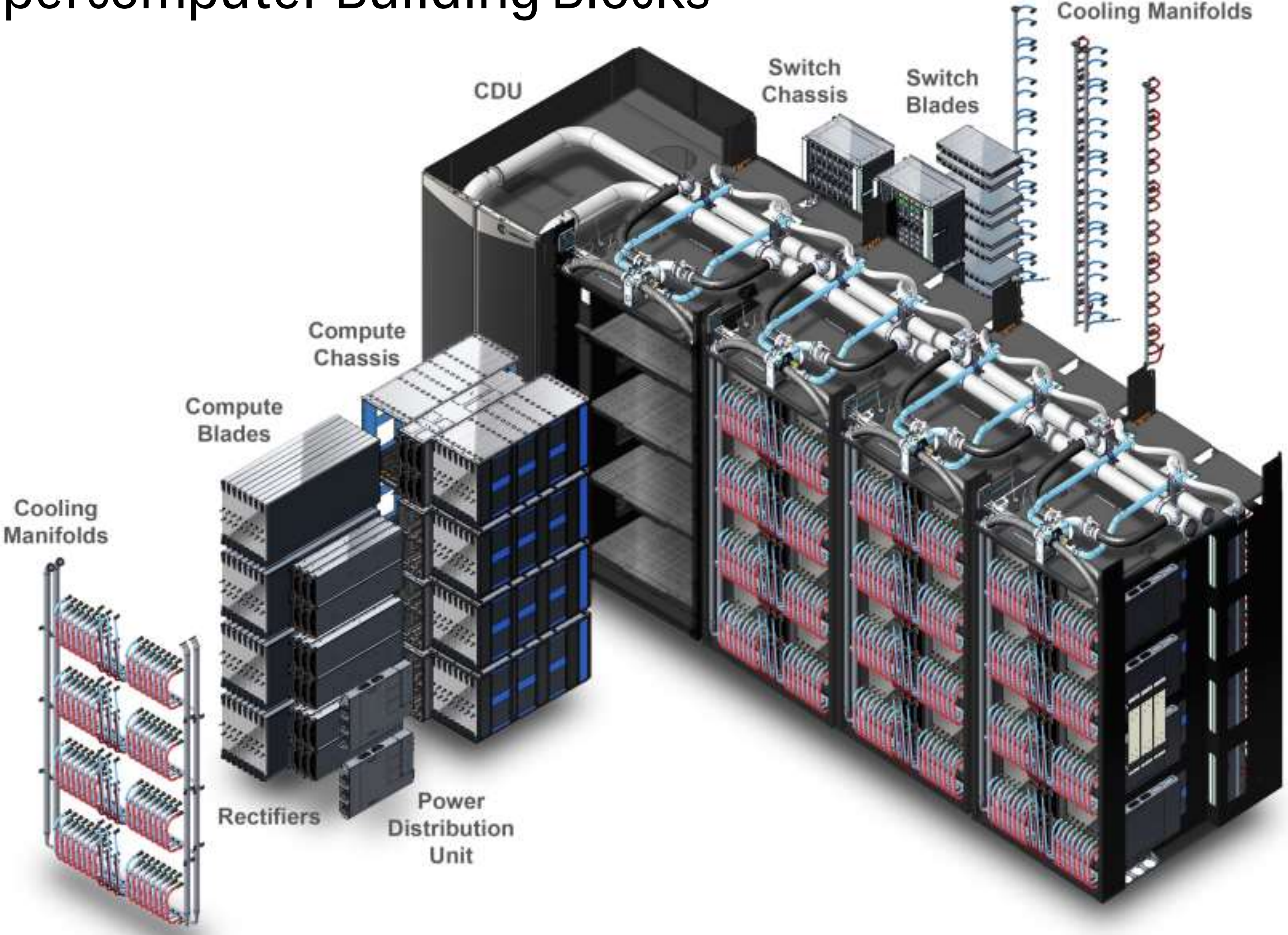
LUMI-C

- 2048 nodes
  - 1888 256GB nodes
  - 128 512GB nodes
  - 32 1TB nodes
- 2 × AMD EPYC 7763 2.45GHz base (3.5GHz boost), 64c processor
- 128 (2x64) cores per node
- 8 NUMA regions per node
- HPE Slingshot interconnect

LUMI-G

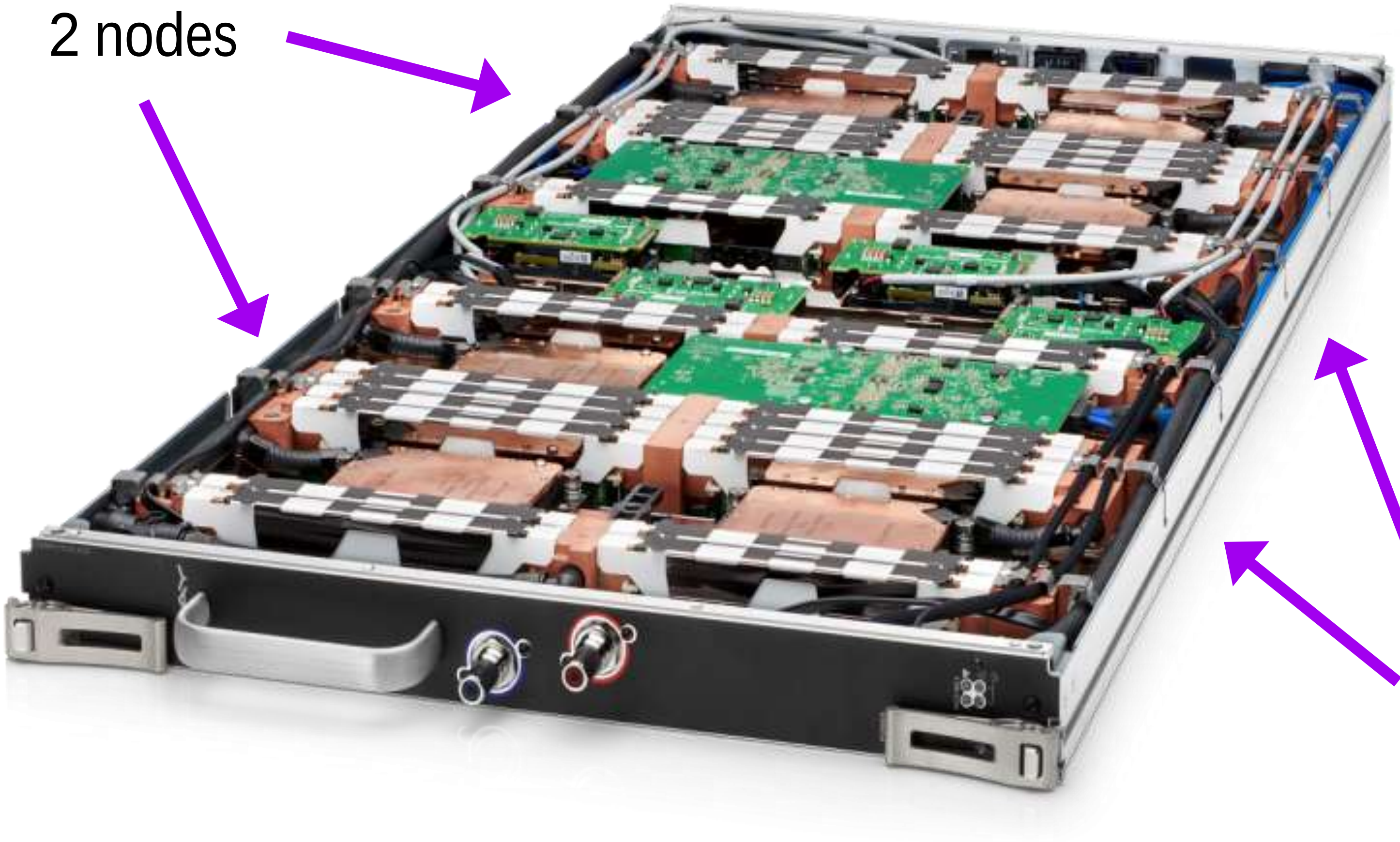
- 2978 nodes with 1 AMD CPU and 4 AMD MI-250X GPU

Cray EX Supercomputer Building Blocks



COMPUTE BLADE ARCHITECTURE (LUMI-C)

2 nodes

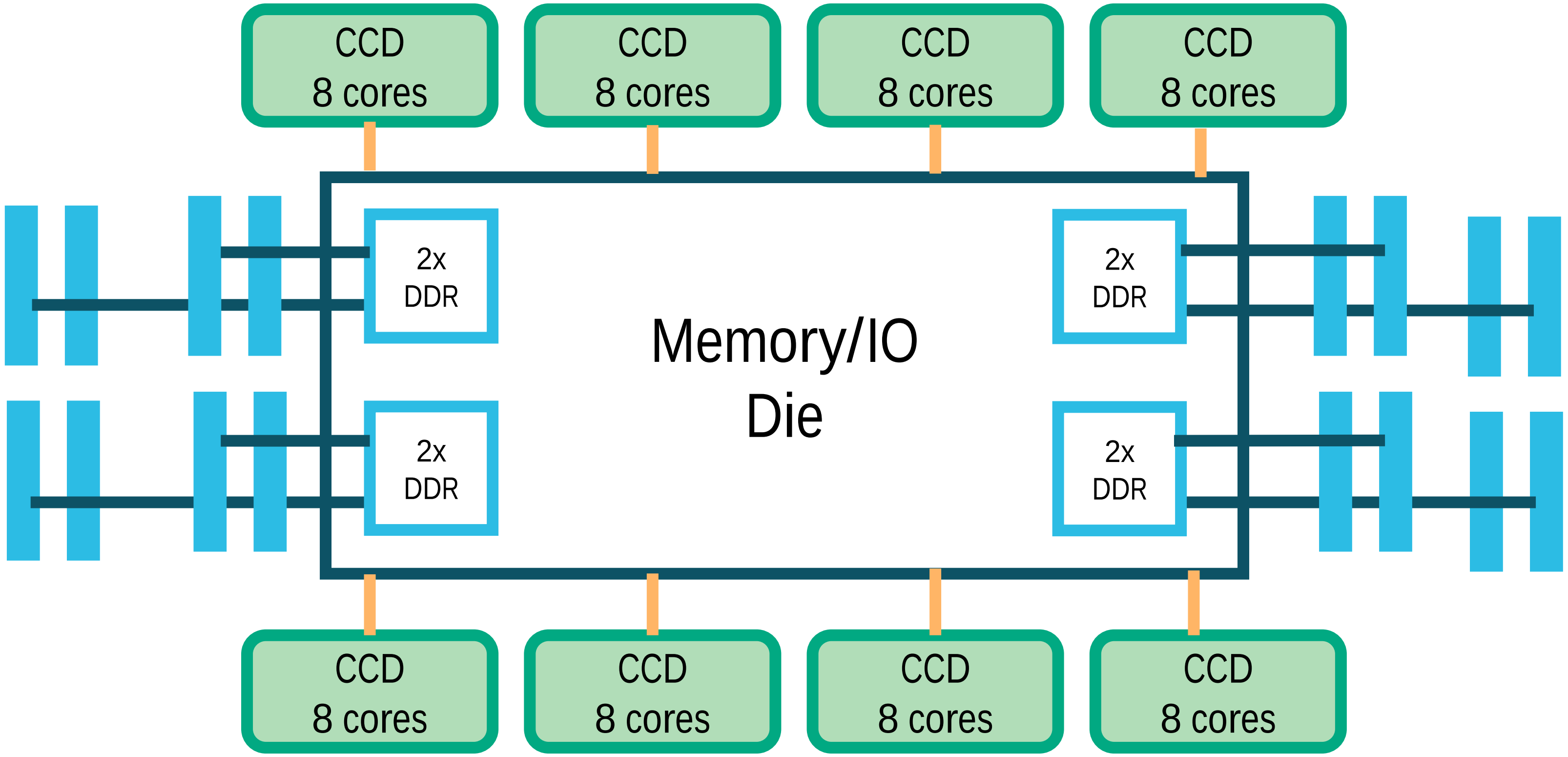


2 nodes

Each node:

- 2 x AMD EPYC 7763 64 core 2.45GHz
- 16 x 16/32/64 GB DDR4 3200
- 256/512/1024 GB per node 2-8GB per core
- 1 x 200Gb/s injection ports per node

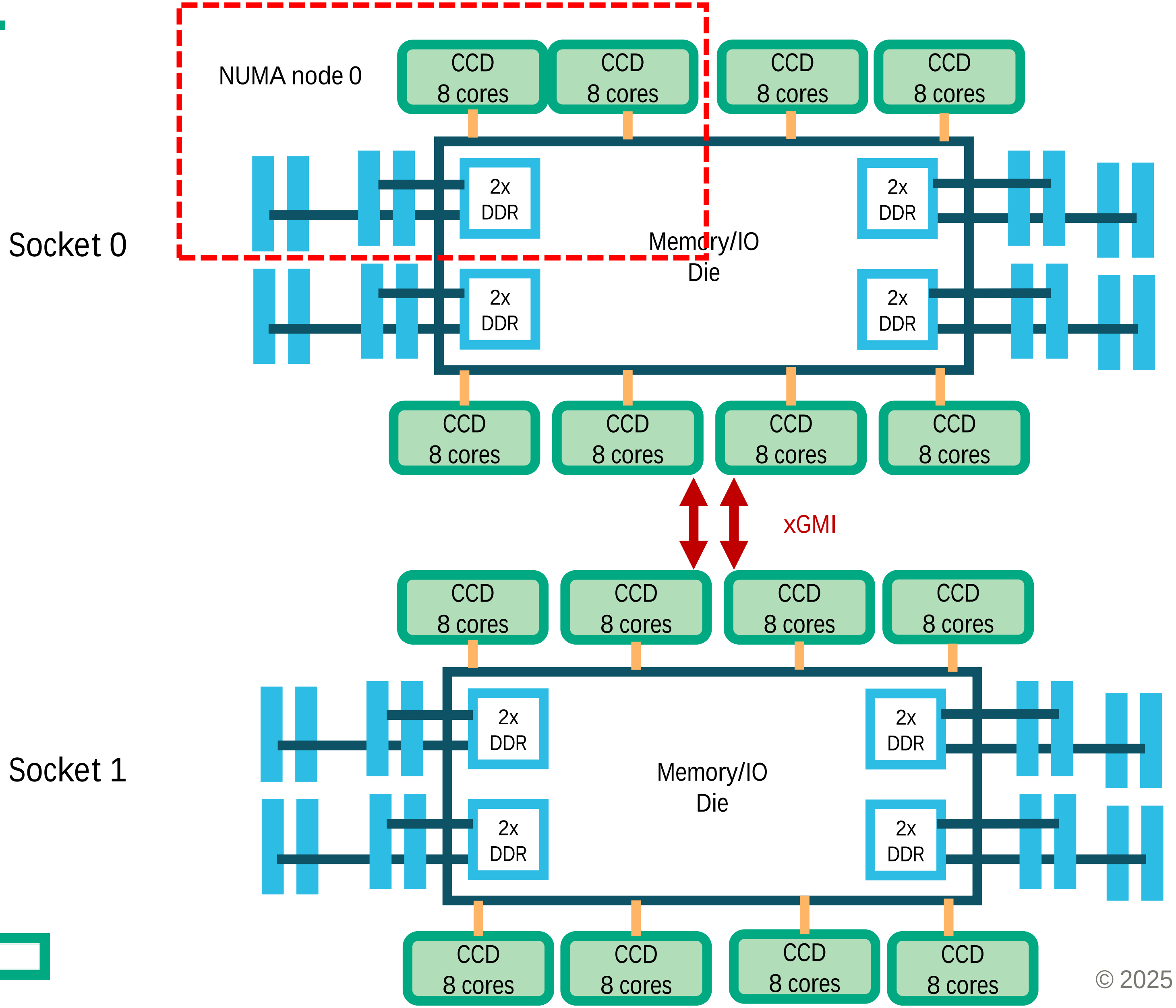
AMD EPYC Processor (Milan / LUMI-C)



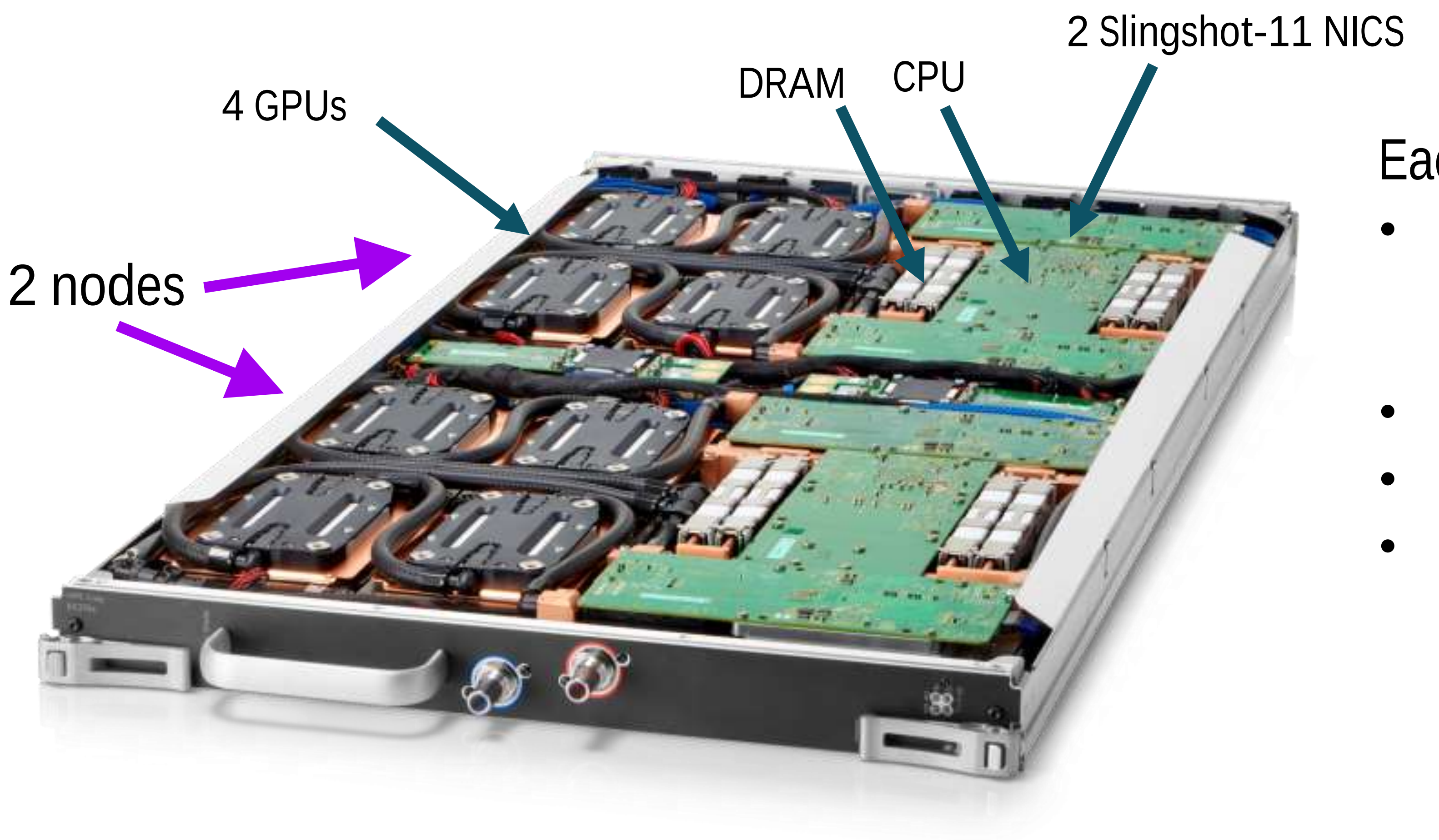
AMD EPYC 7763

- Base clock 2.45GHz
- Boost clock 3.5 GHz
- 280W TDP
- 64 cores,
  128 SMT threads
- L1 cache 32kB / core
- L2 cache 512kB / core
- L3 cache 32MB / 8-cores
  256MB L3 cache in total
- 128 PCIe 4.0 lanes
- 8 channel DDR4 3200MHz,
  204.8 GB/s peak b/w
- Configured as 4
  NUMA nodes
- Vector support: AVX2

2-Socket Node



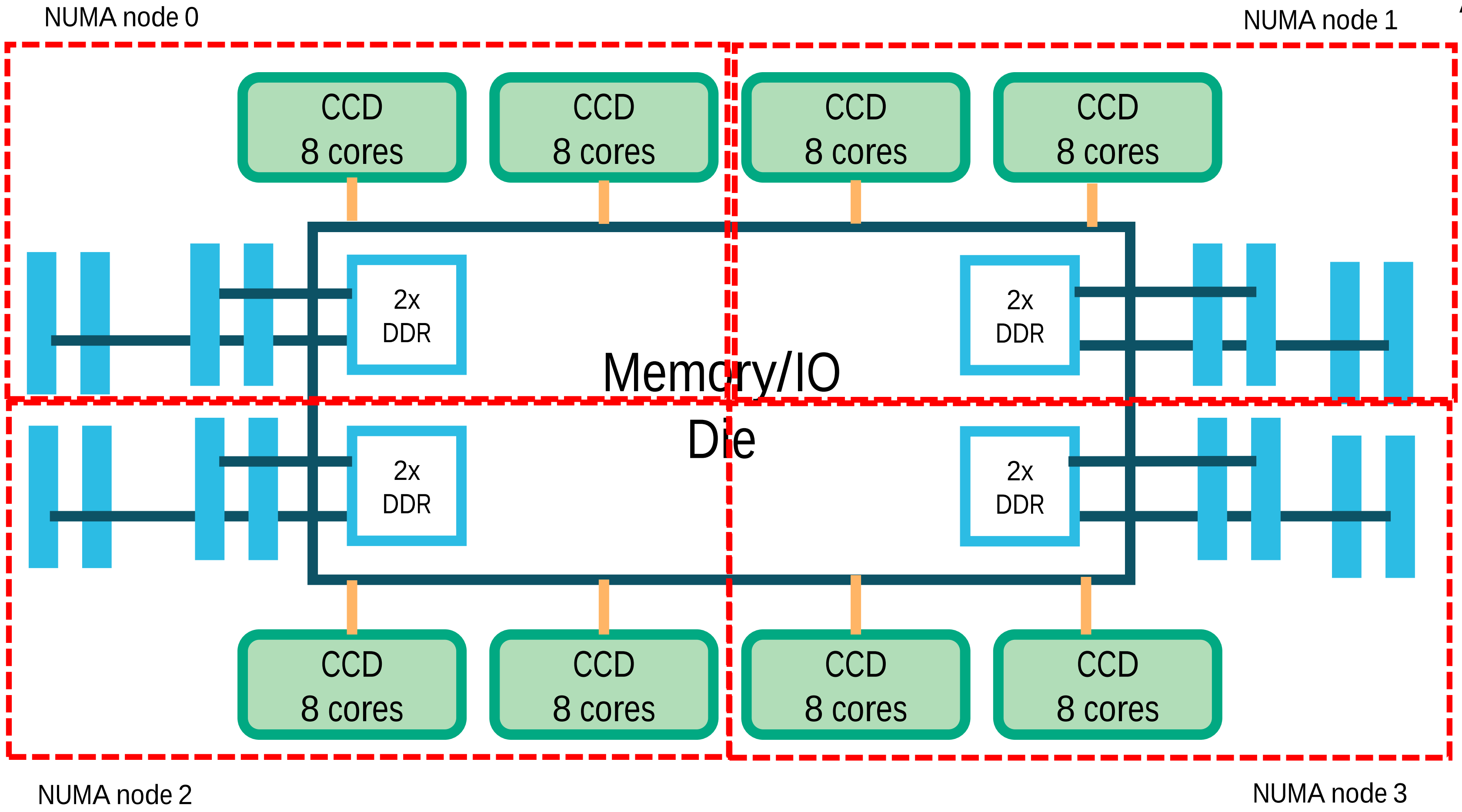
Compute Blade Architecture (LUMI-G)



Each node:

- AMD EPYC 7A53 “Optimized 3rd Gen EPYC” 64-Core Processor, 2.00 GHz
- 512 GB DDR4 memory
- 4x AMD MI-250X GPU
- Each GPU connected to a Slingshot 200Gb/s NIC

AMD EPYC processor (LUMI-G)



AMD EPYC 7A53

- Base clock 2.00 GHz
- 64 cores,
  128 hardware threads
- L1 cache 32kB / core
- L2 cache 512kB / core
- L3 cache 32MB / 8-cores
  256MB L3 cache in total
- 128 PCIe 4.0 lanes
- 8 channel DDR 3200MHz,
  204.8 GB/s peak b/w
- Configured as 4
  NUMA nodes
- Vector support: AVX2

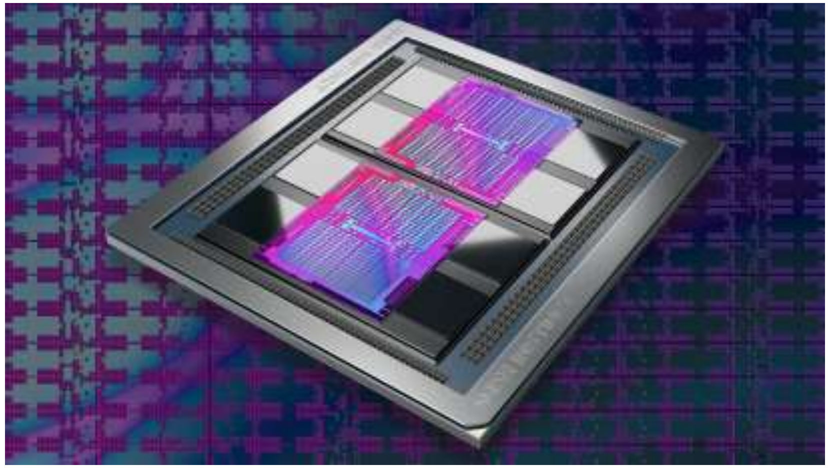
Mi250X GPU Architecture

- Two compute dies (Graphics Compute Dies (GCDs))
  - Interconnected with 200 GB/s per direction

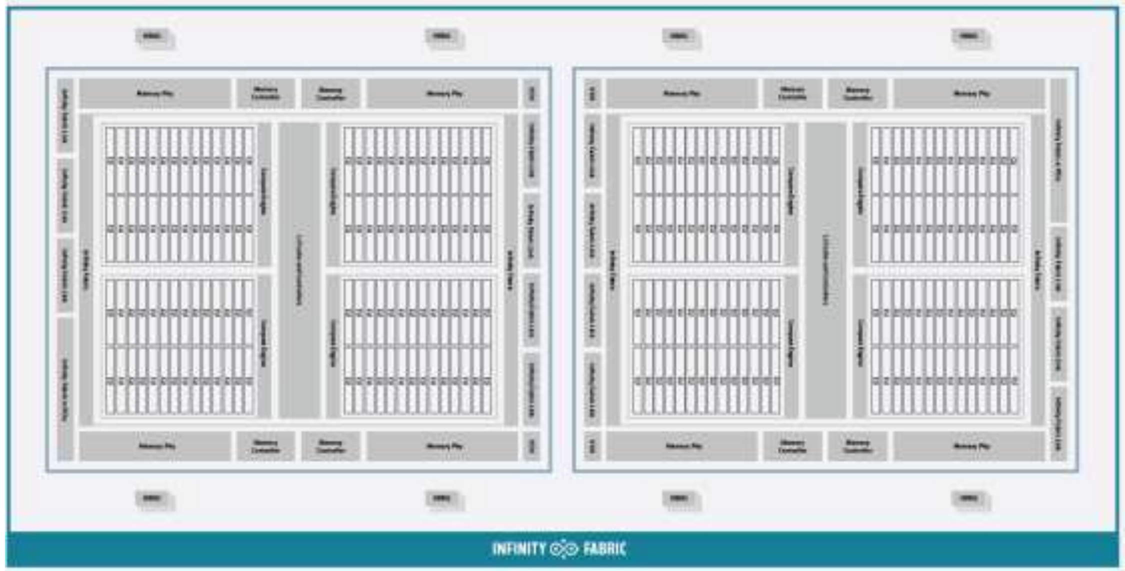
- Dedicated Memory (HBM2e) Size: 128 GB
  - High bandwidth device memory (up to 3.2 TB/s)
  - Memory Clock: 1.6 GHz

- AMD CDNA2 Architecture
  - 110 Compute Units (CU) per each die = 220 CUs
  - 64 SIMD threads per each CU = 14080 Stream Processors
  - Peak FP64/FP32 Vector: 47.9 TFLOPS
  - Peak FP64/FP32 Matrix: 95.7 TFLOPS
  - Total L2 cache: 8 MB per each die (64kB per CU)
  - Frequency: up to 1700 MHz
  - Max power: up to 560 Watts

- Memory coherency with CPU



Source: https://www.amd.com/en/products/server-accelerators/instinct-mi250

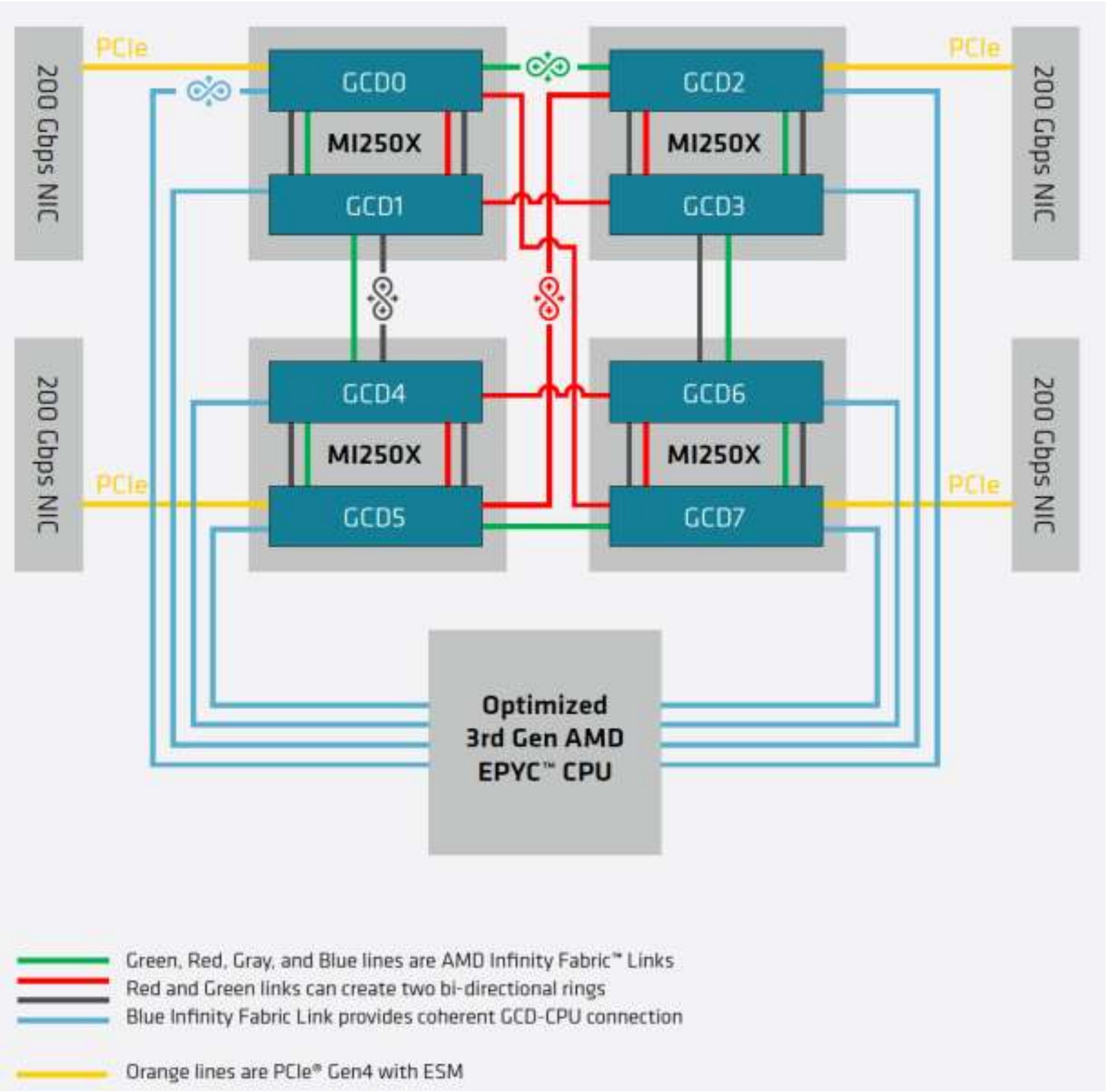


Source: https://www.amd.com/system/files/documents/amd-cdna2-white-paper.pdf

Node Architecture (LUMI-G)

- The programmer can think of the 8 GCDs as 8 separate GPUs, each having 64 GB of high-bandwidth memory (HBM2E)
- The CPU is connected to each GCD via Infinity Fabric CPU-GPU, allowing a peak host-to-device (H2D) and device-to-host (D2H) bandwidth of 36+36 GB/s
  - Coherent memory CPU-GPU
- The 2 GCDs on the same MI250X are connected with Infinity Fabric GPU-GPU
- The GCDs on different MI250X are connected with Infinity Fabric GPU-GPU in the arrangement shown in the diagram on the right, where the peak bandwidth ranges from 50-100 GB/s based on the number of Infinity Fabric connections between individual GCDs

Source: https://www.amd.com/system/files/documents/amd-cdna2-white-paper.pdf



Cray Developer Environment for EX

Development

Programming Models
- HPE Cray MPI
- SHMEM
- OpenMP | OpenACC
- AMD ROCm HIP | NVIDIA CUDA
- UPC | Fortran co-arrays
- Chapel

Programming Environments
- Compiling Environment
- GNU
- Intel Programming Environment
- AMD Programming Environment
- NVIDIA HPC SDK

Programming Languages
- C
- C++
- Fortran
- Python
- Chapel
- R

Optimized Libraries
- LibSci (BLAS)
- LAPACK & ScaLAPACK
- LibSci_ACC
- IRT
- FFTW

I/O Libraries
- NetCDF
- HDF5

DL / AI Tools
- Deep Learning Plug-in

Debugging

- Comparative Debugger
  Compare two versions of an application
- GDB for HPC
  Parallelized gdb for HPC
- Valgrind for HPC
  Memory debugging at scale
- Stack Trace Analysis Tool
  Stack tracing at scale
- Tool for Abnormal Termination Processing
  Manage core files at scale
- TotalView
- DDT

Performance Analysis & Optimization

- Performance Analysis Tool (PAT)
  Whole program performance analysis, exposing wide set of indicators, identifying bottlenecks and automatically generating suggestions to improve performance.
- Visualization Tool
  Complements text reports with summary of performance data in graphs and charts, allowing users to drill down and resolve issues
- Code Parallelization Assistant
  Reveal hidden potential of an application via code restructuring

Setup & Runtime

Environment Setup
- Tool Enablement
  (for Spack, CMake, EasyBuild, etc)
- Modules / Lmod

Supported systems:

- HPE Cray EX supercomputers
- HPE Apollo 2000
- HPE Apollo 6500
- HPE ProLiant DL systems
- Previous Cray XC systems

HPE –authored | HPE Added-value to 3rd party | 3rd party

Components of the Programming Environment

- Cray Compiling Environment (CCE)
  Optimizing compilers for Fortran, C, C++ and UPC
- Cray Scientific and Math Libraries (CSML)
  High Performance libraries for scientific applications (BLAS, LAPACK, ScaLAPCK, FFTW, NetCDF)
- Cray Message Passing Toolkit (CMPT)
  Provides the Message Passing Interface (MPI) and OpenSHMEM
- Cray Environment Setup and Compiling Support (CENV)
  Infrastructure to support the development environment.
  Includes compiler drivers, hugepages support and the PE packaging API
- Cray Performance Measurement and Analysis Tools (CPMAT)
  Provides tools to analyse performance and behaviour of programs and the PAPI performance API
- Cray Debugging Support Tools (CDST)
  Provides debugging tools including gdb4hpc and valgrind4hpc

Documentation at https://cpe.ext.hpe.com/docs/24.03/index.html

Other Compilers and Programming Language Support

- Compilers
  - GNU
  - AOCC (AMD CPU Compiler)
  - AMD LLVM (AMD GPU support)

- Python: 3.11.7
- R: Includes R 4.3.2

Cray Scientific and Maths Libraries (CSML)

The programming environment provides integrated tuned versions of popular scientific libraries

- LibSci
  - BLAS (Basic Linear Algebra Subroutines)
  - BLACS (Basic Linear Algebra Communication Subprograms)
  - CBLAS (wrappers providing a C interface to the FORTRAN BLAS library)
  - IRT (Iterative Refinement Toolkit)
  - LAPACK (Linear Algebra Routines)
  - LAPACKE (C interfaces to LAPACK Routines)
  - ScaLAPACK (Scalable LAPACK)
- LibSci_ACC
  - Subset of GPU-optimized GPU routines from LibSci
- FFTW3
  - Fastest Fourier Transforms in the West, release 3
- Data libraries
  - NetCDF and HDF5

Debugging Support Tools (CDST)

- gdb4hpc
  Command-line parallel debugger
- valgrid4hpc
  Parallel-debugging tool for detection of memory leaks parallel application errors.
  (partially implemented on Shasta)
- Stack Trace Analysis Tool (STAT)
  Merged-stack backtrace analysis tool
- Abnormal Termination Processing (ATP)
  Scalable core file generator and analysis tool
- Cray Comparative Debugging (CCDB)
  Side by side debugging tool for two ‘versions’ of an application.

Cray Performance Measurement and Analysis Tools

- Perftools lite modules for one-shot build/run/analysis
  - perftools-lite, perftools-lite-events, perftools-lite-loops, perftools-lite-hbm
- Full featured perftools module for multi-step collection and analysis
  - pat_build (program instrumentation)
  - pat_report (report generation)
  - Craypat runtime library
- pat_run Launches a dynamically-linked program for analysis
- Also
  - PAPI, Apprentice2, stat_view and reveal

Cray MPI and SHMEM

Cray MPI
- Implementation based on MPICH3 source from ANL
- Includes many improved algorithms and tweaks for Cray hardware
  - Improved algorithms for many collectives
  - Asynchronous progress engine allows overlap of computation and comms
  - Customizable collective buffering when using MPI-IO
  - Optimized Remote Memory Access (one-sided) fully supported including passive RMA
- Full MPI-3.1 support with the exception of
  - MPI_LONG_DOUBLE and MPI_C_LONG_DOUBLE_COMPLEX for CCE
- Includes support for Fortran 2008 bindings (since CCE 8.3.3)

Cray SHMEM
- Fully optimized Cray SHMEM library supported
- Fully compliant with OpenSHMEM v1.5
- Cray XC implementation close to the T3E model

AMD ROCm

Components of the AMD ROCm stack include

- Clang-based compilers for example
- GPU libraries
- Profiling and debugging tools

Controlling the Environment with Modules

Modules

- The Cray Programming Environment uses a “modules” framework to support multiple software versions and to create integrated software packages
- Either Environment Modules or Lmod will be set as the default system wide
- As new versions of the supported software and associated man pages become available, they are installed and added to the Programming Environment as a new module version, while earlier versions are retained to support legacy applications
- System administrators will set the default software versions, or you can choose another version by using modules system commands
- Users can create their own modules, or administrators can install site specific modules available to many users
- Modules are used both to set high-level context (for example choose a compiler toolchain) and to select individual tool and library components and versions

Compiler choice when using the PE

- Use modules to select compiling environment
  - Automatically uses our math, scientific, and communication libraries with chosen compiler
  - Can use debug and profiling tools with chosen compiler

HPE Cray Programming Environment	AMD Programming Environment	Intel Programming Environment	NVIDIA Programming Environment	GNU Programming Environment
Compiling Environment (CCE)	AMD AOCC and ROCm compilers	Intel® C, C++, and Fortran compilers	NVIDIA compilers	GNU Compiler Collection

Cray MPI and SHMEM

Performance Analysis Tools

Debugger Support Tools

Scientific and Math Libraries

Environment Setup and Compiling Support

Third Party Products

Viewing the Current Module State

- Each login session has its own module state which can be modified by loading, swapping or unloading the available *modules*
- This state affects the functioning of the compiler wrappers and in some cases runtime of applications
- A standard, default set of modules is always loaded at login for all users
- Current state can be viewed by running:

```
$> module list
```

Default Modules Example: PE modules based on version 23.09 (*year.month*)

```
% module list
Currently Loaded Modules:
  1) craype-x86-rome                          8) cray-dsmml/0.3.0
  2) libfabric/1.15.2.0                       9) cray-mpich/8.1.29
  3) craype-network-ofi                      10) cray-libsci/24.03.0
  4) perftools-base/24.03.0                  11) PrgEnv-cray/8.5.0
  5) xpmem/2.8.2-1.0_5.1__g84a27a5.shasta    12) ModuleLabel/label     (S)
  6) cce/17.0.1                              13) lumi-tools/24.05      (S)
  7) craype/2.7.31.11                        14) init-lumi/0.2         (S)

  Where:
   S:  Module is Sticky, requires --force to unload or purge
```

Summary of Useful Module Commands

Which modules are available?

- `module avail`, `module avail cce/`
- `module spider cce; module spider cce/17.0.1`

Which modules are currently loaded?

- `module list`

Load software

- `module load perftools`

Change software version

- `module swap cce/17.0.1  cce/16.0.1`

Unload module

- `module unload cce`

Display module release notes

- `module help cce`

Show summary of module environment changes

- `module show cce      (or module display cce)`

Compiler Driver Wrappers

- All applications *that will run in parallel* on the Cray EX should be compiled with the standard language wrappers
- The compiler drivers for each language are:
  - `cc` – wrapper for the C compiler
  - `CC` – wrapper for the C++ compiler
  - `ftn` – wrapper for the Fortran compiler
- These wrappers will choose the required compiler version, target architecture options, scientific libraries and required include files automatically from the module environment.
  - They enable MPI compilation by default
- Use them exactly like you would the original compiler, e.g. to compile `prog1.f90` run

```
ftn -c prog1.f90
```

- It can be necessary to set the env variables CC, CXX, FC for building tools, e.g.

```
CC=cc CXX=CC FC=ftn ./configure <flags>
cmake -DCMAKE_C_COMPILER=cc -DCMAKE_CXX_COMPILER=CC -DCMAKE_FC_COMPILER=ftn <other flags>
```

Backend Compiler Version

- Check the compiler version to know which backend you are using in the compiler wrappers
  - `--version` flag
- E.g.
  - PrgEnv-cray
```
> cc --version
Cray clang version 17.0.1
> ftn --version
Cray Fortran : Version 17.0.1
```
  - PrgEnv-amd
```
> cc --version
AMD clang version 17.0.0
> ftn --version
AMD flang-classic version 17.0.0
```

Wrappers features

- Use the `-craype-verbose` flag to check the flags which are injected by the compiler wrappers, e.g.

```
> cc -craype-verbose
clang -march=znver2 -dynamic
```

- For libraries and include files covered by module files, you should not add anything to your Makefile
  - No additional MPI flags are needed (included by wrappers)
  - You do not need to add any `-I`, `-l` or `–L` flags for the Cray provided libraries
- If your Makefile needs an input for `–L` to work correctly, try using '.'
- If you really need a specific path, try checking 'module show X' for some environment variables
  - You will want to avoid a reference to a specific version as this fixes you build to that version or may create a conflict if the module environment changes
- cmake *should* detect the right libraries and paths when using the wrappers
  - Use newest cmake versions when possible
  - Use wrapper flags to customize compiler flags, see: https://cpe.ext.hpe.com/docs/24.03/craype/cc.html, https://cpe.ext.hpe.com/docs/24.03/craype/ccpp.html, https://cpe.ext.hpe.com/docs/24.03/craype/ftn.html, e.g.

    `---craype-append-opt[=flag]`: Add flag after all PE-generated flags, i.e. used to override flags set by PE
  - In some *extreme* cases, you might want to use the backend compilers and pass the right flags

Choosing a Programming Environment

- The wrappers choose which compiler to use from the `PrgEnv` collection loaded

`PrgEnv`	Description	`Real Compilers`
`PrgEnv-cray`	Cray Compilation Environment	`crayftn, craycc, crayCC`
`PrgEnv-gnu`	GNU Compiler Collection	`gfortran, gcc, g++`
`PrgEnv-aocc`	AMD Optimizing Compilers (AOCC, CPU only support)	`flang, clang, clang++`
`PrgEnv-amd`	AMD LLVM Compilers (GPU support)	`amdflang, amdclang, amdclang++`
`PrgEnv-cray-amd`	AMD Clang C/C++ compiler and the Cray Compiling Environment (CCE) Fortran compiler	
`PrgEnv-gnu-amd`	AMD Clang C/C++ compiler and the GNU compiler suite Fortran compiler	

Choosing a Programming Environment

- `PrgEnv-cray` is loaded by default at login
  - use `module list` to check what is currently loaded
- List the PrgEnv- meta modules

```
> module avail PrgEnv
```

- Switch to a new programming environment

```
> module swap PrgEnv-cray PrgEnv-gnu

Lmod is automatically replacing "cce/17.0.1" with "gcc-native/13.2".


Due to MODULEPATH changes, the following have been reloaded:
  1) cray-libsci/24.03.0     2) cray-mpich/8.1.29
```

  - The Cray MPI module is loaded by default (`cray-mpich`) and reloaded accordingly upon PrgEnv change
  - Compiler wrappers will link to the proper compiler version of the modules
  - Check the compiler version to know which backend you are using (e.g. `cc -v`)

Recap

- The HPE Cray Programming Environment provides a consistent interface into a host of user and developer software
- Various toolchains are supported: Cray compilers, AMD, GNU, etc.
- Different architectures of CPU and GPU are supported

- The environment is used via a combination of
  - Modules
  - Compiler wrappers

What modules to load

For LUMI-C

- The default gives you PrgEnv-cray, providing access to the CCE compilers
- You can switch to the PrgEnv you want
  - be specific to compute node with   module load craype-x86-milan

For LUMI-G

- Choose the PrgEnv you want
- compute node architecture
  module load craype-x86-trento
  GPU modules
  module load craype-accel-amd-gfx90a
  module load rocm

- Load other library/tool modules as required

Python and R Modules

- R, cray-R (version 4.3.2)

- Python 3

- cray-python/3.11.7 (from CPE 24.03) contains:

  python 3.11.7
  numpy 1.24.4
  pandas 1.5.3
  mpi4py 3.1.4
  dask   2023.6.1

Hugepages

- COS supports multiple pagesizes, hugepages are larger than the default (4k)
- There can be a performance advantage for application that use large datasets and/or MPI buffers
- The hugetlbfs library is used to map application memory segments into hugepages locations
- Modules are provided to load hugepages support as appropriate

```
richards@uan03:~> module avail craype-hugepages

-------------------------- HPE-Cray PE target modules ---------------------------
   craype-hugepages128M    craype-hugepages2G     craype-hugepages512M
   craype-hugepages16M     craype-hugepages2M     craype-hugepages64M
   craype-hugepages1G      craype-hugepages32M    craype-hugepages8M
   craype-hugepages256M    craype-hugepages4M

richards@uan03:~> module load craype-hugepages8M
```

- See `man intro_hugepages`
- Load module at link time, choose page size via module at runtime.
- Set `HUGETLB_VERBOSE` to get runtime information (0-99)

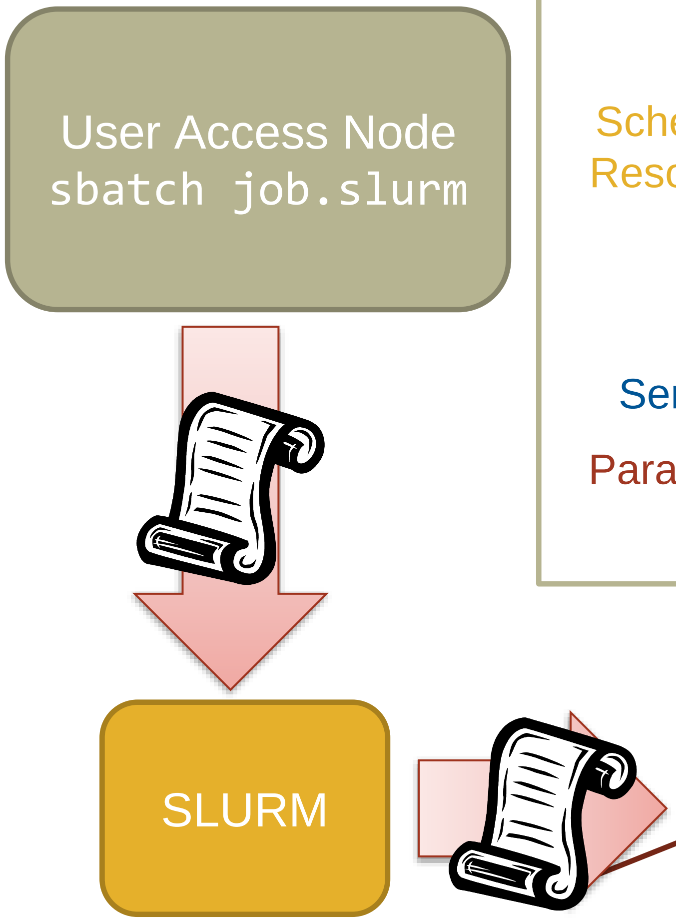
Requesting Resources in Slurm

- Users interact with Slurm by executing commands on the User Access Nodes (UANs)
- Slurm provides multiple mechanisms for users to access compute node resources

- Interactive use
  srun command
- Interactive use within a predefined allocation
  salloc to allocate resources
  srun command(s) to start an application
- Batch usage
  sbatch submits a batch script
  srun command(s) within the batch script

- Resource requests are made by
  - Structured comments in a batch job
  - Environment variables
  - Command-line arguments to sbatch, salloc or srun

Lifecycle of a Batch Script



Example Batch Job Script – `job.slurm`

Scheduler Resources

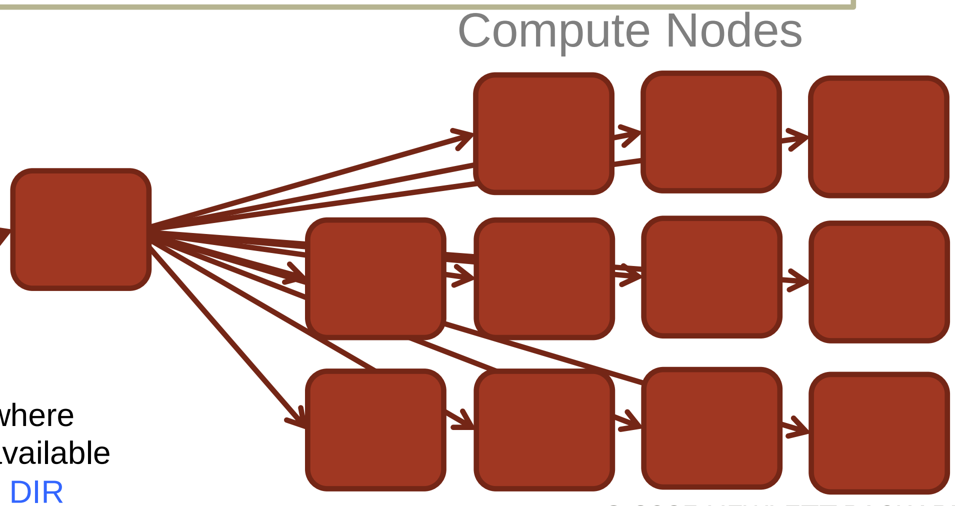
```
#!/bin/bash
#SBATCH -p <your_partition>
#SBATCH -A <your_project>
#SBATCH --time=00:02:00
#SBATCH --nodes=2
#SBATCH --exclusive

cd <some_working_directory>
srun -n 256 ./simulation.exe
```

Serial

Parallel

Check your projects via the command `groups`, e.g.

```
> groups
lazzaroa
project_462000003
project_465001154
```



The script will start by default in the directory where `sbatch` has been executed. This directory is available in the environment variable SLURM_SUBMIT_DIR

Useful Resource-Related Options (srun/sbatch/salloc)

Description	Option
Total Number of tasks	-n,--ntasks
Number of tasks per compute node	--ntasks-per-node
Number of threads per task	-c,--cpus-per-task
Number of nodes	-N,--nodes
Walltime	-t,--time
Request *N* GPUs	--gres=gpu:*N*

Example: Access and check available GPUs via ROCM-SMI

Example Batch Job Script – `job.slurm`

```
#!/bin/bash
#SBATCH -p <partition>
#SBATCH -A <your_project>
#SBATCH --time=00:02:00
#SBATCH --nodes=1
#SBATCH --gres=gpu:8
#SBATCH --exclusive

srun -n 1 rocm-smi
```

Check your projects via the command `groups`, e.g.

```
> groups
lazzaroa project_462000003 project_462000031
```

Request 8 GPUs

- Submit via `sbatch job.slurm`

Recap

Recap

- LUMI Architecture
- The Cray Programming Environment
- Controlling the Environment with modules
- Running a job

What we did not cover:

- Architecture:
  - Slingshot network architecture
  - Cooling infrastructure
- Storage (Lustre)
- Binding jobs to cpu resources

Questions?

