# Compiling and running code on CPU nodes

Johan Hellsvik

# Cray programming environment (CPE)

Reference pages: Compilers and libraries, Build systems course

The Cray Programming Environment (CPE) provides consistent interface to multiple compilers and libraries.

- For building on CPU nodes, we recommend use of one the toolchains
  - `ml PrgEnv-cray` The Cray Compiler Environment
  - `ml PrgEnv-gnu` The GNU Compiler Collection
  - `ml PrgEnv-aocc` AMD Optimizing C/C++ and Fortran compilers
- The `PrgEnv-cray` is the default toolchain
- Load a PDC module to make available a large set of programs and libraries. Most recent version is currently 24.11. `ml PDC/24.11`

# Compiler wrappers

- Compiler wrappers for different programming languages
  - `cc` : C compiler wrapper
  - `CC` : C++ compiler wrapper
  - `ftn` : Fortran compiler wrapper
- The wrappers choose the required compiler version and target architecture optinons.
- Automatically link to MPI library and math libraries
  - MPI library: `cray-mpich`
  - Math libraries: `cray-libsci` and `cray-fftw`

# Compile a simple MPI code

- `hello_world_mpi.f90`

```
program hello_world_mpi
include "mpif.h"
integer myrank,mysize,ierr
call MPI_Init(ierr)
call MPI_Comm_rank(MPI_COMM_WORLD,myrank,ierr)
call MPI_Comm_size(MPI_COMM_WORLD,mysize,ierr)
write(*,*) "Processor ",myrank," of ",mysize,": Hello World!"
call MPI_Finalize(ierr)
end program
```

```
ftn hello_world_mpi.f90 -o hello_world_mpi.x
```

# What flags does the `ftn` wrapper activate?

- Use the flag `-craype-verbose`

```
ftn -craype-verbose hello_world_mpi.f90 -o hello_world_mpi.x
```

# Test run the hello world code

Request 8 cores in the shared partition for interactive use

```
salloc -n 8 -t 10 -p shared -A <name-of-allocation> --reservation=<name of reservation>
```

Run on 8 cores in the `shared` partition

```
user@uan01:> srun -n 8 ./hello_world_mpi.x
 Processor             4  of             8 : Hello World!
 Processor             6  of             8 : Hello World!
 Processor             7  of             8 : Hello World!
 Processor             0  of             8 : Hello World!
 Processor             1  of             8 : Hello World!
 Processor             2  of             8 : Hello World!
 Processor             3  of             8 : Hello World!
 Processor             5  of             8 : Hello World!
```

# Compile a simple linear algebra code

Link to the code

Use cray-libsci. This module is available by default. Load Gnu toolchain

```
ml PrgEnv-gnu
```

Build the code

```
cc dgemm_test.c -o dgemm_test_craylibsci.x
```

# Compile a simple linear algebra code

Use openblas. Load the PDC module and the openblas module.

```
ml PDC/24.11 openblas/0.3.29-cpeGNU-24.11
```

```
cc dgemm_test.c -o dgemm_test_openblas.x -I$EBROOTOPENBLAS/include -L$EBROOTOPENBLAS/lib -lopenblas
```

where the environment variable `EBROOTOPENBLAS` had been set when loading the OpenBLAS module. Its module file includes a statement

```
local root = "/pdc/software/24.11/eb/software/openblas/0.3.29-cpeGNU-24.11"
setenv("EBROOTOPENBLAS", root)
```

which corresponds to an export statement

```
export EBROOTOPENBLAS=/pdc/software/24.11/eb/software/openblas/0.3.29-cpeGNU-24.11
```

# Check the linked libraries

```
ldd dgemm_test_craylibsci.x
```

```
ldd dgemm_test_openblas.x
```

# Check the linked libraries

```
ldd dgemm_test_craylibsci.x

...
libsci_cray.so.6 => /opt/cray/pe/lib64/libsci_cray.so.6
...
```

```
ldd dgemm_test_openblas.x

...
libopenblas.so.0 => /pdc/software/24.11/eb/software/openblas/0.3.29-cpeGNU-24.11/lib/libopenblas.so.0
...
```

# Test run the dgemm_test code

- Run on a single core in the `shared` partition

```
salloc -n 1 -t 10 -p shared -A <name-of-allocation>
srun -n 1 ./dgemm_test_craylibsci.x
srun -n 1 ./dgemm_test_openblas.x
exit
```

- Expected output:

```
    2.700     4.500     6.300     8.100     9.900    11.700    13.500
    4.500     8.100    11.700    15.300    18.900    22.500    26.100
    6.300    11.700    17.100    22.500    27.900    33.300    38.700
```

# Exercise: Compile and run `fftw_test` code

```
ml cray-fftw/3.3.10.9

wget https://people.math.sc.edu/Burkardt/c_src/fftw/fftw_test.c

cc --version
cc fftw_test.c -o fftw_test.x

ldd fftw_test.x

salloc -n 1 -t 10 -p shared -A <name-of-allocation>
srun -n 1 ./fftw_test.x
```

# Environment variables for manual installation of software

- Environment variables for compilers

```
export CC=cc
export CXX=CC
export FC=ftn
export F77=ftn
```

- Environment variables for compiler flags
  - add `-I`, `-L`, `-l`, etc. to Makefile
- Environment variables at runtime
  - prepend to `PATH`, `LD_LIBRARY_PATH`, etc.

# What happens when loading a module

```
ml show elpa/2025.01.001-cpeGNU-24.11
```

```
whatis("Description: ELPA - Eigenvalue SoLvers for Petaflop-Applications")
prepend_path("CMAKE_PREFIX_PATH","/pdc/software/24.11/eb/software/elpa/2025.01.001-cpeGNU-24.11")
prepend_path("LD_LIBRARY_PATH","/pdc/software/24.11/eb/software/elpa/2025.01.001-cpeGNU-24.11/lib")
prepend_path("LIBRARY_PATH","/pdc/software/24.11/eb/software/elpa/2025.01.001-cpeGNU-24.11/lib")
prepend_path("PATH","/pdc/software/24.11/eb/software/elpa/2025.01.001-cpeGNU-24.11/bin")
prepend_path("PKG_CONFIG_PATH","/pdc/software/24.11/eb/software/elpa/2025.01.001-cpeGNU-24.11/lib/pkgconfig")
...
```

# When running your own code

- Load correct programming environment (e.g. `cpeGNU`)
- Load correct dependencies (e.g. `openblas` if your code depends on it)
- Properly prepend to environment variables (e.g. `PATH`, `LD_LIBRARY_PATH`)
- Choose correct SLURM settings

# Compiling and running code on GPU nodes

**Johan Hellsvik**

Reference pages:

Building for AMD GPUs

Introduction to GPUs course

# Generalized programming for GPUs

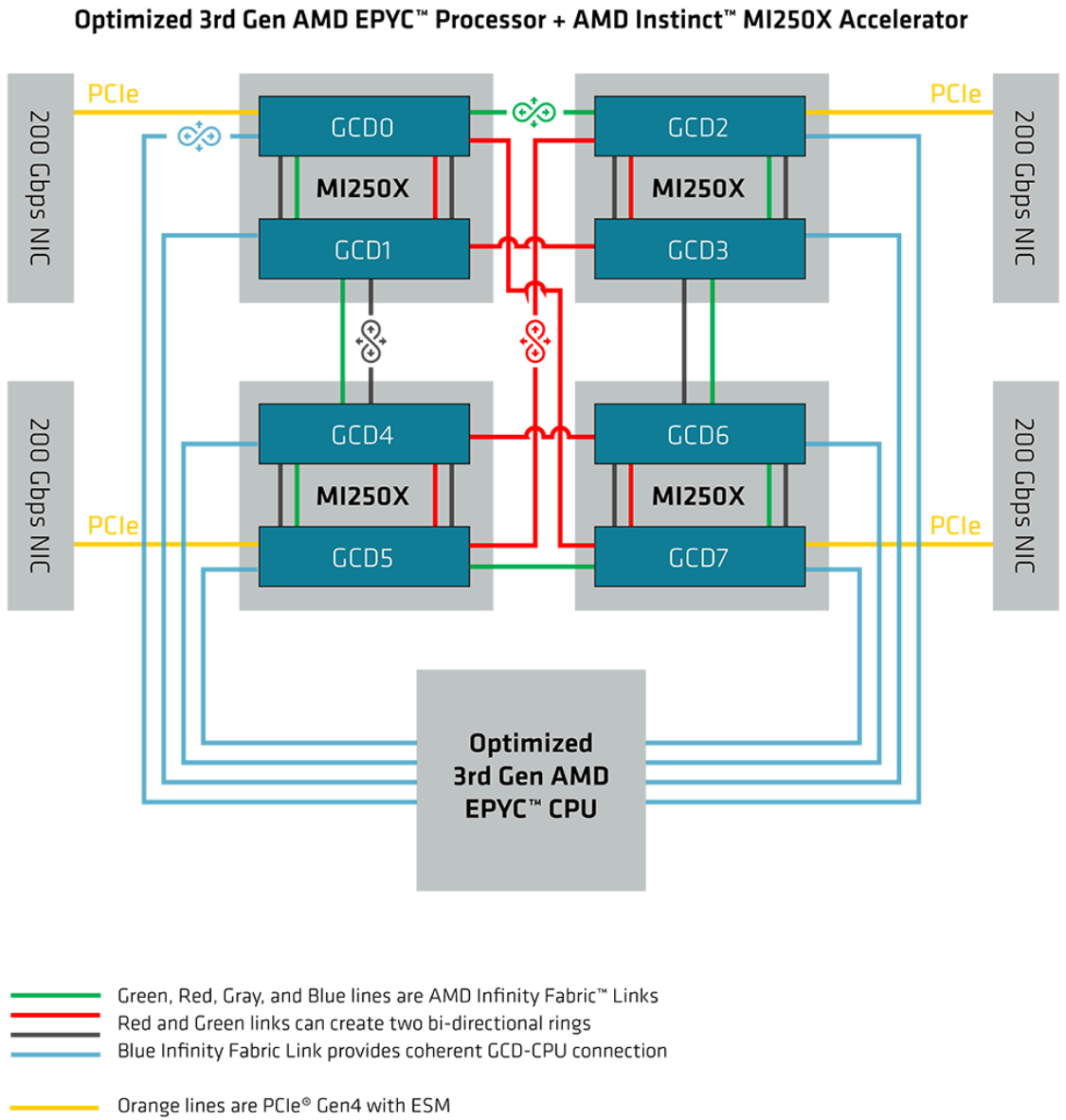
Central processing units (CPU) and graphics processing units (GPU) do different work

- CPUs have large instruction sets and execute general code.
- GPUs have smaller instructions sets. Runs compute intensive work in parallel on large number of compute units (CU).
- Code execution is started and controlled from the CPU. Compute intensive work is offloaded to the GPU.

# Dardel GPU nodes

Dardel has 62 GPU nodes, each of which is equipped with

- One AMD EPYC™ processor with 64 cores
- 512 GB of shared fast HBM2E memory
- Four AMD Instinct™ MI250X GPUs (with an impressive performance of up to 95.7 TFLOPS in double precision when using special matrix operations)



# AMD Radeon Open Compute (ROCm)

The AMD Radeon Open Compute (ROCm) platform is a software stack for programming and running of programs on GPUs.

- The ROCm platform supports different programming models
  - Heterogeneous interface for portability (HIP)
  - Offloading to GPU with OpenMP directives
  - The SYCL programming model
- AMD ROCm Information Portal

# Setting up a GPU build environment

- Load the PDC/24.11 module and version 6.3.3 of ROCm with
  - `ml PDC/24.11`
  - `ml rocm/6.3.3`
- Set the accelerator target to **amd-gfx90a** (AMD MI250X GPU)
  - `ml craype-accel-amd-gfx90a`
- Choose one of the available toolchains (Cray, Gnu, AMD)
  - `ml PrgEnv-cray` The Cray Compiler Environment
  - `ml PrgEnv-gnu` The GNU Compiler Collection
  - `ml PrgEnv-amd` AMD Compiler Environment for GPU

## The ROCM info command

Information on the available GPU hardware can be displayed with the `rocminfo` command. Example output (truncated)

```
ROCk module is loaded
=====================
HSA System Attributes
=====================
Runtime Version:         1.1
System Timestamp Freq.:  1000.000000MHz

==========
HSA Agents
==========
*******
Agent 1
*******
  Name:                    AMD EPYC 7A53 64-Core Processor
  Uuid:                    CPU-XX
```

## The CRAY_ACC_DEBUG runtime environment variable

For executables that are built with the compilers of the Cray Compiler Environment (CCE), verbose runtime information can be enabled with the environment variable `CRAY_ACC_DEBUG` which takes values 1, 2 or 3.

For the highest level of information

```
export CRAY_ACC_DEBUG=3
```

# Offloading to GPU with HIP

The heterogeneous interface for portability (HIP) is a hardware close (low level) programming model for GPUs. Example lines of code:

- Include statement for the HIP runtime

```
#include <hip/hip_runtime.h>
```

- HIP functions have names starting with `hip`

```
// Get number of GPUs available
if (hipGetDeviceCount(&ndevices) != hipSuccess) {
    printf("No such devices\n");
    return 1;
    }
printf("You can access GPU devices: 0-%d\n", (ndevices - 1));
```

- Explicit handling of memory on the GPU

```
// Allocate memory on device
hipMalloc(&devs1, size);
hipMalloc(&devs2, size);
// Copy data host -> device
hipMemcpy(devs1, hosts1, size, hipMemcpyHostToDevice);
```

- Call to run the compute kernel on the GPU

```
// Run kernel
hipLaunchKernelGGL(MyKernel, ngrid, nblock, 0, 0, devs1, devs2);
```

# Offloading to GPU with OpenMP

The OpenMP programming model can be used for directive based offloading to GPUs.

Example: A serial code that operates on arrays `vecA` , `vecB` , and `vecC`

```
! Dot product of two vectors
do i = 1, nx
   vecC(i) =  vecA(i) * vecB(i)
end do
```

Implement OpenMP offloading by inserting OpenMP directives. In Fortran `!$omp`

```
! Dot product of two vectors
!$omp target teams distribute map(from:vecC) map(to:vecA,vecB)
do i = 1, nx
   vecC(i) =  vecA(i) * vecB(i)
end do
!$omp end target teams distribute
```

# Exercise 1: Hello world with HIP

Build and test run a Hello World C++ code which offloads to GPU via HIP.

- Download the source code
  - `wget https://raw.githubusercontent.com/PDC-support/introduction-to-pdc/master/example/hello_world_gpu.cpp`
- Load the ROCm module and set the accelerator target to amd-gfx90a (AMD MI250X GPU)
  - `ml rocm/6.3.3`
  - `ml craype-accel-amd-gfx90a`
- Compile the code with the AMD hipcc compiler on the login node
  - `hipcc --offload-arch=gfx90a hello_world_gpu.cpp -o hello_world_gpu.x`

## Run the code as a batch job

- Edit job_gpu_helloworld.sh to specify the compute project and reservation
- Submit the script with `sbatch job_gpu_helloworld.sh`

with program output written to `output.txt`

```
You can access GPU devices: 0-7
GPU 0: hello world```
...
```

# Exercise 2: Dot product with OpenMP

Build and test run a Fortran program that calculates the dot product of vectors.

- Activate the PrgEnv-cray environment `ml PrgEnv-cray`
- Download the source code
  - `wget https://github.com/ENCCS/openmp-gpu/raw/main/content/exercise/ex04/solution/ex04.F90`
- Load the ROCm module and set the accelerator target to amd-gfx90a
  - `ml rocm/6.3.3 craype-accel-amd-gfx90a`
- Compile the code on the login node
  - `ftn -fopenmp ex04.F90 -o ex04.x`

## Run the code as a batch job

- Edit job_gpu_ex04.sh to specify the compute project and reservation
- Submit the script with `sbatch job_gpu_ex04.sh`
- with program output `The sum is:  1.25` written to `output.txt`

## Optionally, test the code in interactive session.

- First queue to get one GPU node reserved for 10 minutes
  - `salloc -N 1 -t 0:10:00 -A <project name> -p gpu`
- wait for a node, then run the program `srun -n 1 ./ex04.x`
- with program output to standard out `The sum is:  1.25`

- Alternatively, login to the reserved GPU node (here nid002792) `ssh nid002792`.
- Load ROCm, activate verbose runtime information, and run the program
  - `ml rocm/6.3.3`
  - `export CRAY_ACC_DEBUG=3`
  - `./ex04.x`
- with program output to standard out

```
ACC: Version 5.0 of HIP already initialized, runtime version 50013601
ACC: Get Device 0
...
...
ACC: End transfer (to acc 0 bytes, to host 4 bytes)
ACC:
The sum is:  1.25
ACC: __tgt_unregister_lib
```