olcf / olcf-user-docs Public

Code Issues 17 Pull requests 17 Actions Projects Security Ins

master olcf-user-docs / systems / frontier_user_guide.rst

Go to file t

secondspass Merge pull request #686 from olcf/verolero86-darsha... 2 weeks ago

3382 lines (2510 loc) · 224 KB

Frontier User Guide

Note

Notable differences between Summit and Frontier:

Orion scratch filesystem

master olcf-user-docs / systems / frontier_user_guide.rst Top

Preview Code Blame Raw

See the :ref:`frontier-data-storage` section or this recording for more information.

Cray Programming Environment

Frontier utilizes the Cray Programming Environment. Many aspects including LMOD are similar to Summit's environment. But Cray's compiler wrappers and the Cray and AMD compilers are worth noting.

See the :ref:`frontier-compilers` section for more information.

AMD GPUs

Each frontier node has 4 AMD MI250X accelerators with two Graphic Compute Dies (GCDs) in each accelerator. The system identifies each GCD as an independent device (so for simplicity we use the term GPU when we talk about a GCD) for a total of 8 GPUs per node (compared to Summit's 6 Nvidia V100 GPUs per node). Each pair of GPUs is associated with a particular NUMA domain (see node diagram in :ref:`frontier-nodes` section) which might affect how your application should lay out data and computation.

See the :ref:`amd-gpus` section for more information.

Programming Models

Since Frontier uses AMD GPUs, code written in Nvidia's CUDA language will not work as is. They need to be converted to use HIP, which is AMD's GPU programming framework, or should be converted to some other GPU framework that supports AMD GPUs as a backend e.g. OpenMP Offload, Kokkos, RAJA, OCCA, SYCL/DPC++ etc .

See the :ref:`amd-hip` section for more information and links about HIP.

Slurm batch scheduler

Frontier uses SchedMD's Slurm Workload Manager for job scheduling instead of IBM's LSF. Slurm provides similar functionality to LSF, albeit with different commands. Notable are the separation in batch script submission (`sbatch`) and interactive batch submission (`salloc`).

See the :ref:`frontier-slurm` section for more infomation including a LSF to Slurm command comparison.

Srun job launcher

Frontier uses Slurm's job launcher, `srun` , instead of Summit's `jsrun` to launch parallel jobs within a batch script. Overall functionality is similar, but commands are notably different. Frontier's :ref:`compute node layout <frontier-simple>` should also be considered when selecting job layout.

See the :ref:`frontier-srun` section for more `srun` information, and see :ref:`frontier-mapping` for `srun` examples on Frontier.

OLCF Support

If you encounter any issues or have questions, please contact the OLCF via the following:

- Email us at help@olcf.ornl.gov
- Contact your OLCF liaison
- Sign-up to attend OLCF Office Hours

System Overview

Frontier is a HPE Cray EX supercomputer located at the Oak Ridge Leadership Computing Facility. With a theoretical peak double-precision performance of approximately 2 exaflops (2 quintillion calculations per second), it is the fastest system in the world for a wide range of traditional computational science applications. The system has 74 Olympus rack HPE cabinets, each with 128 AMD compute nodes, and a total of 9,408 AMD compute nodes.

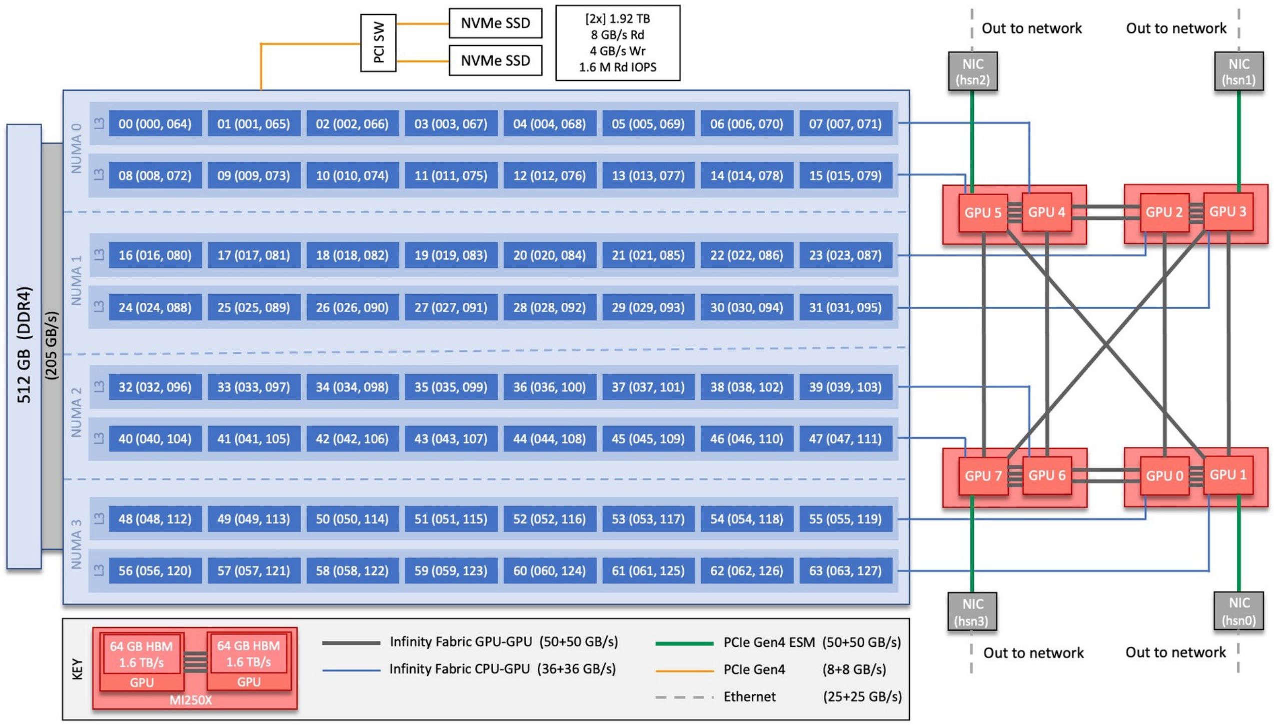
Frontier Compute Nodes

Each Frontier compute node consists of [1x] 64-core AMD "Optimized 3rd Gen EPYC" CPU (with 2 hardware threads per physical core) with access to 512 GB of DDR4 memory. Each node also contains [4x] AMD MI250X, each with 2 Graphics Compute Dies (GCDs) for a total of 8 GCDs per node. 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. The 2 GCDs on the same MI250X are connected with Infinity Fabric GPU-GPU with a peak bandwidth of 200 GB/s. The GCDs on different MI250X are connected with Infinity Fabric GPU-GPU in the arrangement shown in the Frontier Node Diagram below, where the peak bandwidth ranges from 50-100 GB/s based on the number of Infinity Fabric connections between individual GCDs.

Note

TERMINOLOGY:

The 8 GCDs contained in the 4 MI250X will show as 8 separate GPUs according to Slurm, `ROCR_VISIBLE_DEVICES` , and the ROCr runtime, so from this point forward in the quick-start guide, we will simply refer to the GCDs as GPUs.



Note

There are [4x] NUMA domains per node and [2x] L3 cache regions per NUMA for a total of [8x] L3 cache regions. The 8 GPUs are each associated with one of the L3 regions as follows:

NUMA 0:

- hardware threads 000-007, 064-071 | GPU 4
- hardware threads 008-015, 072-079 | GPU 5

NUMA 1:

- hardware threads 016-023, 080-087 | GPU 2
- hardware threads 024-031, 088-095 | GPU 3

NUMA 2:

- hardware threads 032-039, 096-103 | GPU 6
- hardware threads 040-047, 104-111 | GPU 7

NUMA 3:

- hardware threads 048-055, 112-119 | GPU 0
- hardware threads 056-063, 120-127 | GPU 1

By default, Frontier reserves the first core in each L3 cache region. Frontier uses low-noise mode, which constrains all system processes to core 0. Low-noise mode cannot be disabled by users. In addition, Frontier uses SLURM core specialization (`-S 8` flag at job allocation time, e.g., `sbatch`) to reserve one core from each L3 cache region, leaving 56 allocatable cores. Set `-S 0` at job allocation to override this setting.

Node Types

On Frontier, there are three major types of nodes you will encounter: Login, Launch, and Compute. While all of these are similar in terms of hardware (see: :ref:`frontier-nodes`), they differ considerably in their intended use.

Node Type	Description
Login	When you connect to Frontier, you're placed on a login node. This is the place to write/edit/compile your code, manage data, submit jobs, etc. You should never launch parallel jobs from a login node nor should you run threaded jobs on a login node. Login nodes are shared resources that are in use by many users simultaneously.
Compute	Most of the nodes on Frontier are compute nodes. These are where your parallel job executes. They're accessed via the `srun` command.

System Interconnect

The Frontier nodes are connected with [4x] HPE Slingshot 200 Gbps (25 GB/s) NICs providing a node-injection bandwidth of 800 Gbps (100 GB/s).

File Systems

Frontier is connected to Orion, a parallel filesystem based on Lustre and HPE ClusterStor, with a 679 PB usable namespace (`/lustre/orion/`). In addition to Frontier, Orion is available on the OLCF's data transfer nodes, Andes, and some other smaller resources. It is not available from Summit. Data will not be automatically transferred from Alpine to Orion. Frontier also has access to the center-wide NFS-based filesystem (which provides user and project home areas). Each compute node has two 1.92TB Non-Volatile Memory storage devices. See :ref:`frontier-data-storage` for more information.

Frontier connects to the center’s High Performance Storage System (HPSS) - for user and project archival storage - users can log in to the :ref:`dtn-user-guide` to move data to/from HPSS.

Operating System

Frontier is running Cray OS 2.4 based on SUSE Linux Enterprise Server (SLES) version 15.4.

GPUs

Each Frontier compute node contains 4 AMD MI250X. The AMD MI250X has a peak performance of 53 TFLOPS in double-precision for modeling and simulation. Each MI250X contains 2 GPUs, where each GPU has a peak performance of 26.5 TFLOPS (double-precision), 110 compute units, and 64 GB of high-bandwidth memory (HBM2) which can be accessed at a peak of 1.6 TB/s. The 2 GPUs on an MI250X are connected with Infinity Fabric with a bandwidth of 200 GB/s (in each direction simultaneously).

Connecting

To connect to Frontier, `ssh` to `frontier.olcf.ornl.gov`. For example:

```
$ ssh <username>@frontier.olcf.ornl.gov
```

For more information on connecting to OLCF resources, see :ref:`connecting-to-olcf`.

Data and Storage

Transition from Alpine to Orion

- Frontier mounts Orion, a parallel filesystem based on Lustre and HPE ClusterStor, with a 679 PB usable namespace (/lustre/orion/). In addition to Frontier, Orion is available on the OLCF's data transfer nodes, Andes, and some other smaller resources. It is not available from Summit. Frontier will not mount Alpine when Orion is in production.
- Data will not be automatically transferred from Alpine to Orion. Users should consider the data needed from Alpine and transfer it. Globus is the preferred method and there is access to both Orion and Alpine through the Globus OLCF DTN endpoint. See :ref:`data-transferring-data-globus`. Use of HPSS to specifically stage data for the Alpine to Orion transfer is discouraged.
- On Alpine, there was no user-exposed concept of file striping, the process of dividing a file between the storage elements of the filesystem. Orion uses a feature called Progressive File Layout (PFL) that changes the striping of files as they grow. Because

of this, we ask users not to manually adjust the file striping. If you feel the default striping behavior of Orion is not meeting your needs, please contact help@olcf.ornl.gov.

- As with Alpine, files older than 90 days are purged from Orion. Please plan your data management and lifecycle at OLCF before generating the data.

For more detailed information about center-wide file systems and data archiving available on Frontier, please refer to the pages on :ref:`data-storage-and-transfers`. The subsections below give a quick overview of NFS, Lustre,and HPSS storage spaces as well as the on node NVMe "Burst Buffers" (SSDs).

NFS Filesystem

Area	Path	Type	Permissions	Quota	Backups	Pur
User Home	`/ccs/home/[userid]`	NFS	User set	50 GB	Yes	No
Project Home	`/ccs/proj/[projid]`	NFS	770	50 GB	Yes	No

Note

Though the NFS filesystem's User Home and Project Home areas are read/write from Frontier's compute nodes, we strongly recommend that users launch and run jobs from the Lustre Orion parallel filesystem instead due to its larger storage capacity and superior performance. Please see below for Lustre Orion filesystem storage areas and paths.

Lustre Filesystem

Area	Path	Type	Permission
Member Work	`/lustre/orion/[projid]/scratch/[userid]`	Lustre HPE ClusterStor	700
Project Work	`/lustre/orion/[[projid]/proj-shared`	Lustre HPE ClusterStor	770

Area	Path	Type	Permission
World Work	`/lustre/orion/[[projid]/world-shared`	Lustre HPE ClusterStor	775

HPSS Archival Storage

Please note that the HPSS is not mounted directly onto Frontier nodes. There are two main methods for accessing and moving data to/from the HPSS. The first is to use the command line utilities `hsi` and `htar` . The second is to use the Globus data transfer service. See :ref:`data-hpss` for more information on both of these methods.

Area	Path	Type	Permissions	Quota
Member Archive	`/hpss/prod/[projid]/users/$USER`	HPSS	700	100 TB
Project Archive	`/hpss/prod/[projid]/proj-shared`	HPSS	770	100 TB
World Archive	`/hpss/prod/[projid]/world-shared`	HPSS	775	100 TB

Using Globus to Move Data to Orion

The following example is intended to help users who are making the transition from Summit to Frontier to move their data between Alpine GPFS and Orion Lustre. We strongly recommend using Globus for this transfer as it is the method that is most efficient for users and that causes the least contention on filesystems and data transfer nodes.

Note

Globus Warnings:

- Globus transfers do not preserve file permissions. Arriving files will have (rw-r--r--) permissions, meaning arriving file will have *user* read and write permissions and *group* and *world* read permissions. Note that the arriving files will not have any execute permissions, so you will need to use chmod to reset execute permissions before running a Globus-transferred executable.

- Globus will overwrite files at the destination with identically named source files. This is done without warning.
- Globus has restriction of 8 active transfers across all the users. Each user has a limit of 3 active transfers, so it is required to transfer a lot of data on each transfer than less data across many transfers.
- If a folder is constituted with mixed files including thousands of small files (less than 1MB each one), it would be better to tar the small files. Otherwise, if the files are larger, Globus will handle them.

Here is a recording of an example transfer from Alpine to Orion using Globus and the OLCF DTN: Using Globus to Move Data to Orion.

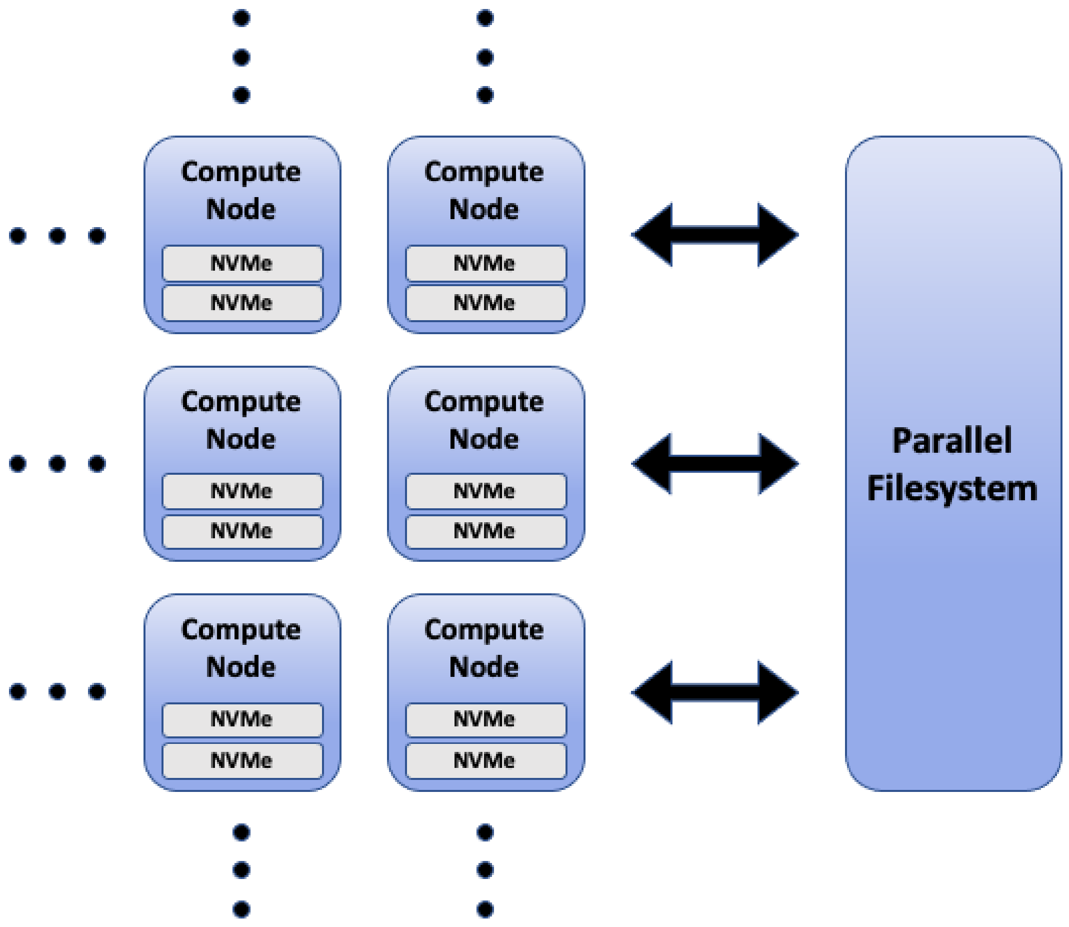
Below is a summary of the steps for data transfer given in the recording:

1. Login to globus.org using your globus ID and password. If you do not have a globusID, set one up here: Generate a globusID.

2. Once you are logged in, Globus will open the “File Manager” page. Click the left side “Collection” text field in the File Manager and type “OLCF DTN”.
3. When prompted, authenticate into the OLCF DTN endpoint using your OLCF username and PIN followed by your RSA passcode.
4. Click in the left side “Path” box in the File Manager and enter the path to your data on Alpine. For example, /gpfs/alpine/stf007/proj-shared/my_alpine_data. You should see a list of your files and folders under the left “Path” Box.
5. Click on all files or folders that you want to transfer in the list. This will highlight them.
6. Click on the right side “Collection” box in the File Manager and type “OLCF DTN”
7. Click in the right side “Path” box and enter the path where you want to put your data on Orion, for example, /lustre/orion/stf007/proj-shared/my_orion_data
8. Click the left "Start" button.
9. Click on “Activity“ in the left blue menu bar to monitor your transfer. Globus will send you an email when the transfer is complete.

NVMe

Each compute node on Frontier has [2x] 1.92TB **N**on-**V**olatile **Me**mory (NVMe) storage devices (SSDs), colloquially known as a "Burst Buffer" with a peak sequential performance of 5500 MB/s (read) and 2000 MB/s (write). The purpose of the Burst Buffer system is to bring improved I/O performance to appropriate workloads. Users are not required to use the NVMes. Data can also be written directly to the parallel filesystem.



The NVMes on Frontier are local to each node.

NVMe Usage

To use the NVMe, users must request access during job allocation using the `-C nvme` option to `sbatch` , `salloc` , or `srun` . Once the devices have been granted to a job, users can access them at `/mnt/bb/<userid>` . **Users are responsible for moving data to/from the NVMe before/after their jobs**. Here is a simple example script:

```
#!/bin/bash
#SBATCH -A <projid>
#SBATCH -J nvme_test
#SBATCH -o %x-%j.out
#SBATCH -t 00:05:00
```

```
#SBATCH -p batch
#SBATCH -N 1
#SBATCH -C nvme

date

# Change directory to user scratch space (GPFS)
cd /gpfs/alpine/<projid>/scratch/<userid>

echo " "
echo "*****ORIGINAL FILE*****"
cat test.txt
echo "***********************"

# Move file from GPFS to SSD
mv test.txt /mnt/bb/<userid>

# Edit file from compute node
srun -n1 hostname >> /mnt/bb/<userid>/test.txt

# Move file from SSD back to GPFS
mv /mnt/bb/<userid>/test.txt .

echo " "
echo "*****UPDATED FILE******"
cat test.txt
echo "***********************"
```

And here is the output from the script:

```
$ cat nvme_test-<jobid>.out

*****ORIGINAL FILE*****
This is my file. There are many like it but this one is mine.
***********************

*****UPDATED FILE******
This is my file. There are many like it but this one is mine.
frontier0123
***********************
```

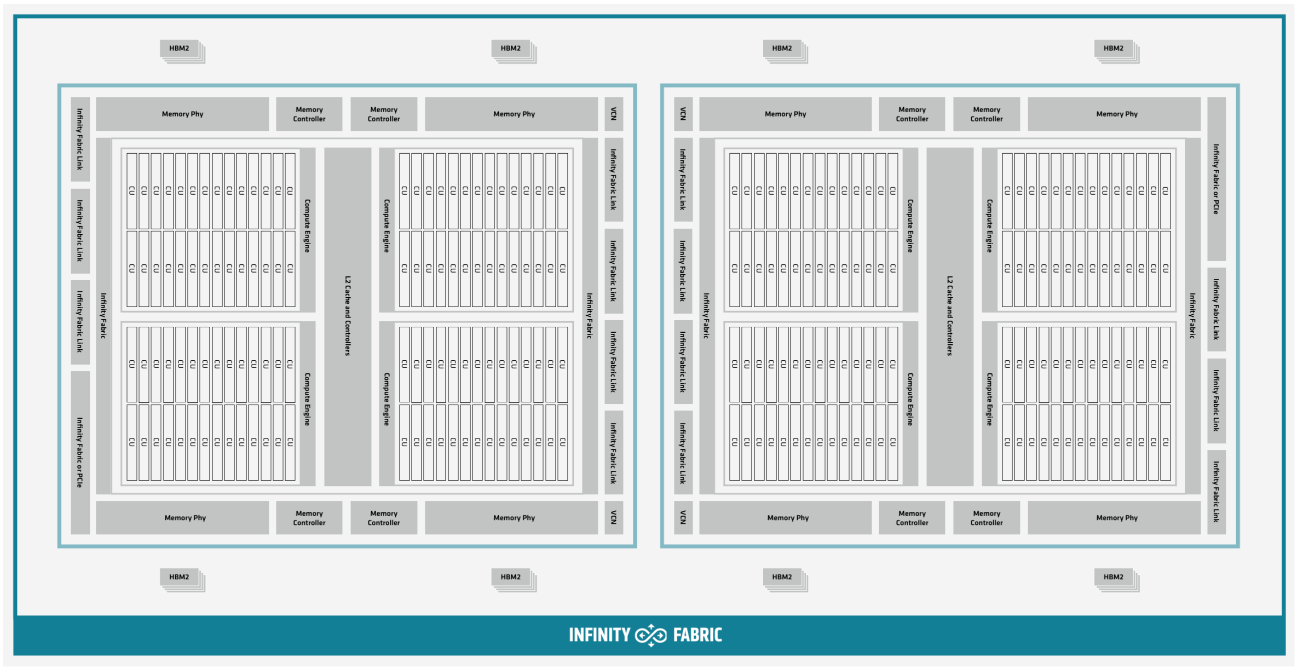
AMD GPUs

The AMD Instinct MI200 is built on advanced packaging technologies enabling two Graphic Compute Dies (GCDs) to be integrated into a single package in the Open Compute Project (OCP) Accelerator Module (OAM) in the MI250 and MI250X products. Each GCD is build on the AMD CDNA 2 architecture. A single Frontier node contains 4 MI250X OAMs for the total of 8 GCDs.

Note

The Slurm workload manager and the ROCr runtime treat each GCD as a separate GPU and visibility can be controlled using the `ROCR_VISIBLE_DEVICES` environment variable. Therefore, from this point on, the Frontier guide simply refers to a GCD as a GPU.

Each GPU contains 110 Compute Units (CUs) grouped in 4 Compute Engines (CEs). Physically, each GPU contains 112 CUs, but two are disabled. A command processor in each GPU receives API commands and transforms them into compute tasks. Compute tasks are managed by the 4 compute engines, which dispatch wavefronts to compute units. All wavefronts from a single workgroup are assigned to the same CU. In CUDA terminology, workgroups are "blocks", wavefronts are "warps", and work-items are "threads". The terms are often used interchangeably.



The 110 CUs in each GPU deliver peak performance of 26.5 TFLOPS in double precision. Also, each GPU contains 64 GB of high-bandwidth memory (HBM2) accessible at a peak bandwidth of 1.6 TB/s. The 2 GPUs in an MI250X are connected with [4x] GPU-to-GPU Infinity Fabric links providing 200+200 GB/s of bandwidth. (Consult the diagram in the :ref:`frontier-nodes` section for information on how the accelerators are connected to each other, to the CPU, and to the network.

Note

The X+X GB/s notation describes bidirectional bandwidth, meaning X GB/s in each direction.

AMD vs NVIDIA Terminology

AMD	NVIDIA
Work-items or Threads	Threads
Workgroup	Block
Wavefront	Warp
Grid	Grid

We will be using these terms interchangeably as they refer to the same concepts in GPU programming, with the exception that we will only be using "wavefront" (which refers to a unit of 64 threads) instead of "warp" (which refers to a unit of 32 threads) as they mean different things.

Blocks (workgroups), Threads (work items), Grids, Wavefronts

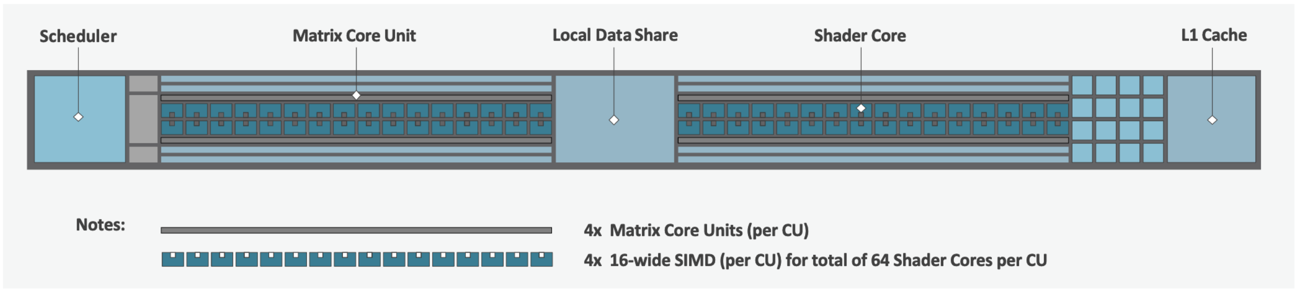
When kernels are launched on a GPU, a "grid" of thread blocks are created, where the number of thread blocks in the grid and the number of threads within each block are defined by the programmer. The number of blocks in the grid (grid size) and the number of threads within each block (block size) can be specified in one, two, or three dimensions during the kernel launch. Each thread can be identified with a unique id within the kernel, indexed along the X, Y, and Z dimensions.

- Number of blocks that can be specified along each dimension in a grid: (2147483647, 2147483647, 2147483647)
- Max number of threads that can be specified along each dimension in a block: (1024, 1024, 1024)
    - However, the total of number of threads in a block has an upper limit of 1024 [i.e. (size of x dimension * size of y dimension * size of z dimension) cannot exceed

1024].

Each block (or workgroup) of threads is assigned to a single Compute Unit i.e. a single block won’t be split across multiple CUs. The threads in a block are scheduled in units of 64 threads called wavefronts (similar to warps in CUDA, but warps only have 32 threads instead of 64). When launching a kernel, up to 64KB of block level shared memory called the Local Data Store (LDS) can be statically or dynamically allocated. This shared memory between the threads in a block allows the threads to access block local data with much lower latency compared to using the HBM since the data is in the compute unit itself.

The Compute Unit


Scheduler
Matrix Core Unit
Local Data Share
Shader Core
L1 Cache
Notes:
4x Matrix Core Units (per CU)
4x 16-wide SIMD (per CU) for total of 64 Shader Cores per CU

Each CU has 4 Matrix Core Units (the equivalent of NVIDIA's Tensor core units) and 4 16-wide SIMD units. For a vector instruction that uses the SIMD units, each wavefront (which has 64 threads) is assigned to a single 16-wide SIMD unit such that the wavefront as a whole executes the instruction over 4 cycles, 16 threads per cycle. Since other wavefronts occupy the other three SIMD units at the same time, the total throughput still remains 1 instruction per cycle. Each CU maintains an instructions buffer for 10 wavefronts and also maintains 256 registers where each register is 64 4-byte wide entries.

HIP

The Heterogeneous Interface for Portability (HIP) is AMD’s dedicated GPU programming environment for designing high performance kernels on GPU hardware. HIP is a C++ runtime API and programming language that allows developers to create portable applications on different platforms, including the AMD MI250X. This means that developers can write their GPU applications and with very minimal changes be able to run their code in any environment. The API is very similar to CUDA, so if you're already familiar with CUDA there is almost no additional work to learn HIP. See here for a series of tutorials on programming with HIP and also converting existing CUDA code to HIP with the hipify tools .

Things To Remember When Programming for AMD GPUs

- The MI250X has different denormal handling for FP16 and BF16 datatypes, which is relevant for ML training. Prefer using the BF16 over the FP16 datatype for ML models as you are more likely to encounter denormal values with FP16 (which get flushed to zero, causing failure in convergence for some ML models). See more in :ref:`using-reduced-precision`.
- Memory can be automatically migrated to GPU from CPU on a page fault if XNACK operating mode is set. No need to explicitly migrate data or provide managed memory. This is useful if you're migrating code from a programming model that relied on 'unified' or 'managed' memory. See more in :ref:`enabling-gpu-page-migration`. Information about how memory is accessed based on the allocator used and the XNACK mode can be found in :ref:`migration-of-memory-allocator-xnack`.
- HIP has two kinds of memory allocations, coarse grained and fine grained, with tradeoffs between performance and coherence. Particularly relevant if you want to ues the hardware FP atomic instructions. See more in :ref:`fp-atomic-ops-coarse-fine-allocations`.
- FP32 atomicAdd operations on Local Data Store (i.e. block shared memory) can be slower than the equivalent FP64 operations. See more in :ref:`performance-lds-atomicadd`.

See the :ref:`frontier-compilers` section for information on compiling for AMD GPUs, and see the :ref:`tips-and-tricks` section for some detailed information to keep in mind to run more efficiently on AMD GPUs.

Programming Environment

Frontier users are provided with many pre-installed software packages and scientific libraries. To facilitate this, environment management tools are used to handle necessary changes to the shell.

Environment Modules (Lmod)

Environment modules are provided through Lmod, a Lua-based module system for dynamically altering shell environments. By managing changes to the shell's environment variables (such as `PATH`, `LD_LIBRARY_PATH`, and `PKG_CONFIG_PATH`), Lmod allows you to alter the software available in your shell environment without the risk of creating package and version combinations that cannot coexist in a single environment.

General Usage

The interface to Lmod is provided by the `module` command:

Command	Description
`module -t list`	Shows a terse list of the currently loaded modules
`module avail`	Shows a table of the currently available modules
`module help <modulename>`	Shows help information about `<modulename>`
`module show <modulename>`	Shows the environment changes made by the `<modulename>` modulefile
`module spider <string>`	Searches all possible modules according to `<string>`
`module load <modulename> [...]`	Loads the given `<modulename>` (s) into the current environment
`module use <path>`	Adds `<path>` to the modulefile search cache and `MODULESPATH`
`module unuse <path>`	Removes `<path>` from the modulefile search cache and `MODULESPATH`
`module purge`	Unloads all modules
`module reset`	Resets loaded modules to system defaults
`module update`	Reloads all currently loaded modules

Searching for Modules

Modules with dependencies are only available when the underlying dependencies, such as compiler families, are loaded. Thus, module avail will only display modules that are compatible with the current state of the environment. To search the entire hierarchy across all possible dependencies, the `spider` sub-command can be used as summarized in the following table.

Command	Description
`module spider`	Shows the entire possible graph of modules
`module spider <modulename>`	Searches for modules named `<modulename>` in the graph of possible modules
`module spider <modulename>/<version>`	Searches for a specific version of `<modulename>` in the graph of possible modules
`module spider <string>`	Searches for modulefiles containing `<string>`

Command	Description

Compilers

Cray, AMD, and GCC compilers are provided through modules on Frontier. The Cray and AMD compilers are both based on LLVM/Clang. There is also a system/OS versions of GCC available in `/usr/bin` . The table below lists details about each of the module-provided compilers. Please see the following :ref:`frontier-compilers` section for more detailed inforation on how to compile using these modules.

Cray Programming Environment and Compiler Wrappers

Cray provides `PrgEnv-<compiler>` modules (e.g., `PrgEnv-cray`) that load compatible components of a specific compiler toolchain. The components include the specified compiler as well as MPI, LibSci, and other libraries. Loading the `PrgEnv-<compiler>` modules also defines a set of compiler wrappers for that compiler toolchain that automatically add include paths and link in libraries for Cray software. Compiler wrappers are provided for C (`cc`), C++ (`CC`), and Fortran (`ftn`).

Note

Use the `-craype-verbose` flag to display the full include and link information used by the Cray compiler wrappers. This must be called on a file to see the full output (e.g., `CC -craype-verbose test.cpp`).

MPI

The MPI implementation available on Frontier is Cray's MPICH, which is "GPU-aware" so GPU buffers can be passed directly to MPI calls.

Compiling

Compilers

Cray, AMD, and GCC compilers are provided through modules on Frontier. The Cray and AMD compilers are both based on LLVM/Clang. There is also a system/OS versions of GCC available in `/usr/bin` . The table below lists details about each of the module-provided compilers.

Note

It is highly recommended to use the Cray compiler wrappers (`cc` , `CC` , and `ftn`) whenever possible. See the next section for more details.

Vendor	Programming Environment	Compiler Module	Language	Compiler Wrapper	Compiler
Cray	`PrgEnv-cray`	`cce`	C	`cc`	`craycc`
			C++	`CC`	`craycxx` or `crayC`
			Fortran	`ftn`	`crayftn`
AMD	`PrgEnv-amd`	`amd`	C	`cc`	`amdclang`
			C++	`CC`	`amdclang++`
			Fortran	`ftn`	`amdflang`
GCC	`PrgEnv-gnu`	`gcc`	C	`cc`	`${GCC_PATH}/bin/`
			C++	`CC`	`${GCC_PATH}/bin/`
			Fortran	`ftn`	`${GCC_PATH}/bin/`

Cray Programming Environment and Compiler Wrappers

Cray provides `PrgEnv-<compiler>` modules (e.g., `PrgEnv-cray`) that load compatible components of a specific compiler toolchain. The components include the specified compiler as well as MPI, LibSci, and other libraries. Loading the `PrgEnv-<compiler>` modules also defines a set of compiler wrappers for that compiler toolchain that automatically add include paths and link in libraries for Cray software. Compiler wrappers are provided for C (`cc`), C++ (`CC`), and Fortran (`ftn`).

For example, to load the AMD programming environment, do:

```
module load PrgEnv-amd
```

This module will setup your programming environment with paths to software and libraries that are compatible with AMD compilers.

Note

Use the `-craype-verbose` flag to display the full include and link information used by the Cray compiler wrappers. This must be called on a file to see the full output (e.g., `CC -craype-verbose test.cpp`).

Exposing The ROCm Toolchain to your Programming Environment

If you need to add the tools and libraries related to ROCm, the framework for targeting AMD GPUs, to your path, you will need to use a version of ROCm that is compatible with your programming environment.

The following modules help you expose the ROCm Toolchain to your programming Environment:

Programming Environment Module	Module that gets you ROCm Toolchain	How you load it:
`PrgEnv-amd`	`amd`	amd is loaded automatically with `module load PrgEnv-amd`
`PrgEnv-cray` or `PrgEnv-gnu`	`amd-mixed`	`module load amd-mixed`

Note

Both the CCE and ROCm compilers are Clang-based, so please be sure to use consistent (major) Clang versions when using them together. You can check which version of Clang is being used with CCE and ROCm by giving the `--version` flag to `CC` and `hipcc`, respectively.

MPI

The MPI implementation available on Frontier is Cray's MPICH, which is "GPU-aware" so GPU buffers can be passed directly to MPI calls.

Implementation	Module	Compiler	Header Files & Linking
Cray MPICH	`cray-mpich`	`cc`, `CC`, `ftn` (Cray compiler wrappers)	MPI header files and linking is built into the Cray compiler wrappers
		`hipcc`	`-L${MPICH_DIR}/lib -lmpi ${CRAY_XPMEM_POST_LINK_OPTS} -lxpmem ${PE_MPICH_GTL_DIR_amd_gfx90a} ${PE_MPICH_GTL_LIBS_amd_gfx90a} -I${MPICH_DIR}/include`

Note

hipcc requires the ROCm Toolclain, See :ref:`exposing-the-rocm-toolchain-to-your-programming-environment`

GPU-Aware MPI

To use GPU-aware Cray MPICH, with Frontier's PrgEnv modules, users must set the following modules and environment variables:

If using `PrgEnv-amd` :

```
module load craype-accel-amd-gfx90a

export MPICH_GPU_SUPPORT_ENABLED=1
```

If using `PrgEnv-cray` :

```
module load craype-accel-amd-gfx90a
module load amd-mixed

export MPICH_GPU_SUPPORT_ENABLED=1
```

Note

There are extra steps needed to enable GPU-aware MPI on Frontier, which depend on the compiler that is used (see 1. and 2. below).

1. Compiling with the Cray compiler wrappers, `cc` or `CC`

When using GPU-aware Cray MPICH with the Cray compiler wrappers, most of the needed libraries are automatically linked through the environment variables.

Though, the following header files and libraries must be included explicitly:

```
-I${ROCM_PATH}/include
-L${ROCM_PATH}/lib -lamdhip64
```

where the include path implies that `#include <hip/hip_runtime.h>` is included in the source file.

2. Compiling with `hipcc`

To use `hipcc` with GPU-aware Cray MPICH, use the following environment variables to setup the needed header files and libraries.

```
-I${MPICH_DIR}/include
-L${MPICH_DIR}/lib -lmpi \
  ${CRAY_XPMEM_POST_LINK_OPTS} -lxpmem \
  ${PE_MPICH_GTL_DIR_amd_gfx90a} ${PE_MPICH_GTL_LIBS_amd_gfx90a}

HIPFLAGS = --amdgpu-target=gfx90a
```

Determining the Compatibility of Cray MPICH and ROCm

Releases of `cray-mpich` are each built with a specific version of ROCm, and compatibility across multiple versions is not guaranteed. OLCF will maintain compatible default modules when possible. If using non-default modules, you can determine compatibility by reviewing the *Product and OS Dependencies* section in the `cray-mpich` release notes. This can be displayed by running `module show cray-mpich/<version>`. If the notes indicate compatibility with *AMD ROCM X.Y or later*, only use `rocm/X.Y.Z` modules. If using a non-default version of `cray-mpich`, you must add `${CRAY_MPICH_ROOTDIR}/gtl/lib` to either your `LD_LIBRARY_PATH` at run time or your executable's rpath at build time.

The compatibility table below was determined by linker testing with all current combinations of `cray-mpich` and ROCm-related modules on Frontier.

cray-mpich	ROCm
8.1.17	5.4.0, 5.3.0, 5.2.0, 5.1.0
8.1.23	5.4.0, 5.3.0, 5.2.0, 5.1.0

OpenMP

This section shows how to compile with OpenMP using the different compilers covered above.

Vendor	Module	Language	Compiler	OpenMP flag (CPU thread)
Cray	`cce`	C, C++	`cc` (wraps `craycc`) `CC` (wraps `crayCC`)	`-fopenmp`
		Fortran	`ftn` (wraps `crayftn`)	`-homp`

Vendor	Module	Language	Compiler	OpenMP flag (CPU thread)
				`-fopenmp` (alias)
AMD	`amd`	C C++ Fortran	`cc` (wraps `amdclang`) `CC` (wraps `amdclang++`) `ftn` (wraps `amdflang`)	`-fopenmp`
GCC	`gcc`	C C++ Fortran	`cc` (wraps `$GCC_PATH/bin/gcc`) `CC` (wraps `$GCC_PATH/bin/g++`) `ftn` (wraps `$GCC_PATH/bin/gfortran`)	`-fopenmp`

OpenMP GPU Offload

This section shows how to compile with OpenMP Offload using the different compilers covered above.

Note

Make sure the `craype-accel-amd-gfx90a` module is loaded when using OpenMP offload.

Vendor	Module	Language	Compiler	OpenMP flag (GPU)
Cray	`cce`	C C++	`cc` (wraps `craycc`) `CC` (wraps `crayCC`)	`-fopenmp`
		Fortran	`ftn` (wraps `crayftn`)	`-homp` `-fopenmp` (alias)
AMD	`amd`	C C++ Fortran	`cc` (wraps `amdclang`) `CC` (wraps `amdclang++`) `ftn` (wraps `amdflang`) `hipcc` (requires flags below)	`-fopenmp`

Note

If invoking `amdclang`, `amdclang++`, or `amdflang` directly for `openmp offload`, or using `hipcc` you will need to add:

`-fopenmp -target x86_64-pc-linux-gnu -fopenmp-targets=amdgcn-amd-amdhsa -Xopenmp-target=amdgcn-amd-amdhsa -march=gfx90a` .

HIP

This section shows how to compile HIP codes using the Cray compiler wrappers and `hipcc` compiler driver.

Note

Make sure the `craype-accel-amd-gfx90a` module is loaded when compiling HIP with the Cray compiler wrappers.

Compiler	Compile/Link Flags, Header Files, and Libraries

`CC`
Only with `PrgEnv-cray` `PrgEnv-amd`
	`CFLAGS = -std=c++11 -D__HIP_ROCclr__ -D__HIP_ARCH_GFX90A__=1 --rocm-path=${ROCM_PATH} --offload-arch=gfx90a -x hip`
	`LFLAGS = --rocm-path=${ROCM_PATH}`
	`-L${ROCM_PATH}/lib -lamdhip64`

`hipcc`	Can be used directly to compile HIP source files. To see what is being invoked within this compiler driver, issue the command, `hipcc --verbose` To explicitly target AMD MI250X, use `--amdgpu-target=gfx90a`

Note

hipcc requires the ROCm Toolclain, See :ref:`exposing-the-rocm-toolchain-to-your-programming-environment`

Note

Information about compiling code for different XNACK modes (which control page migration between GPU and CPU memory) can be found in the :ref:`compiling-hip-kernels-for-xnack-modes` section.

HIP + OpenMP CPU Threading

This section shows how to compile HIP + OpenMP CPU threading hybrid codes.

Note

Make sure the `craype-accel-amd-gfx90a` module is loaded when compiling HIP with the Cray compiler wrappers.

Vendor	Compiler	Compile/Link Flags, Header Files, and Libraries

AMD/Cray	`CC`	`CFLAGS = -std=c++11 -D__HIP_ROCclr__ -D__HIP_ARCH_GFX90A__=1 --rocm-path=${ROCM_PATH} --offload-arch=gfx90a -x hip -fopenmp`
		`LFLAGS = --rocm-path=${ROCM_PATH} -fopenmp -L${ROCM_PATH}/lib -lamdhip64`

	`hipcc`	Can be used to directly compile HIP source files, add `-fopenmp` flag to enable OpenMP threading To explicitly target AMD MI250X, use `--amdgpu-target=gfx90a`
GNU	`CC`	The GNU compilers cannot be used to compile HIP code, so all HIP kernels must be separated from CPU code. During compilation, all non-HIP files must be compiled with `CC` while HIP kernels must be compiled with `hipcc`. Then linking must be performed with the `CC` wrapper. NOTE: When using `cmake`, HIP code must currently be compiled using `amdclang++` instead of `hipcc`.

Note

hipcc requires the ROCm Toolclain, See :ref:`exposing-the-rocm-toolchain-to-your-programming-environment`

Running Jobs

Computational work on Frontier is performed by *jobs*. Jobs typically consist of several componenets:

- A batch submission script
- A binary executable
- A set of input files for the executable
- A set of output files created by the executable

In general, the process for running a job is to:

1. Prepare executables and input files.

2. Write a batch script.
3. Submit the batch script to the batch scheduler.
4. Optionally monitor the job before and during execution.

The following sections describe in detail how to create, submit, and manage jobs for execution on Frontier. Frontier uses SchedMD's Slurm Workload Manager as the batch scheduling system.

Login vs Compute Nodes

Recall from the System Overview that Frontier contains two node types: Login and Compute. When you connect to the system, you are placed on a *login* node. Login nodes are used for tasks such as code editing, compiling, etc. They are shared among all users of the system, so it is not appropriate to run tasks that are long/computationally intensive on login nodes. Users should also limit the number of simultaneous tasks on login nodes (e.g. concurrent tar commands, parallel make

Compute nodes are the appropriate place for long-running, computationally-intensive tasks. When you start a batch job, your batch script (or interactive shell for batch-interactive jobs) runs on one of your allocated compute nodes.

Warning

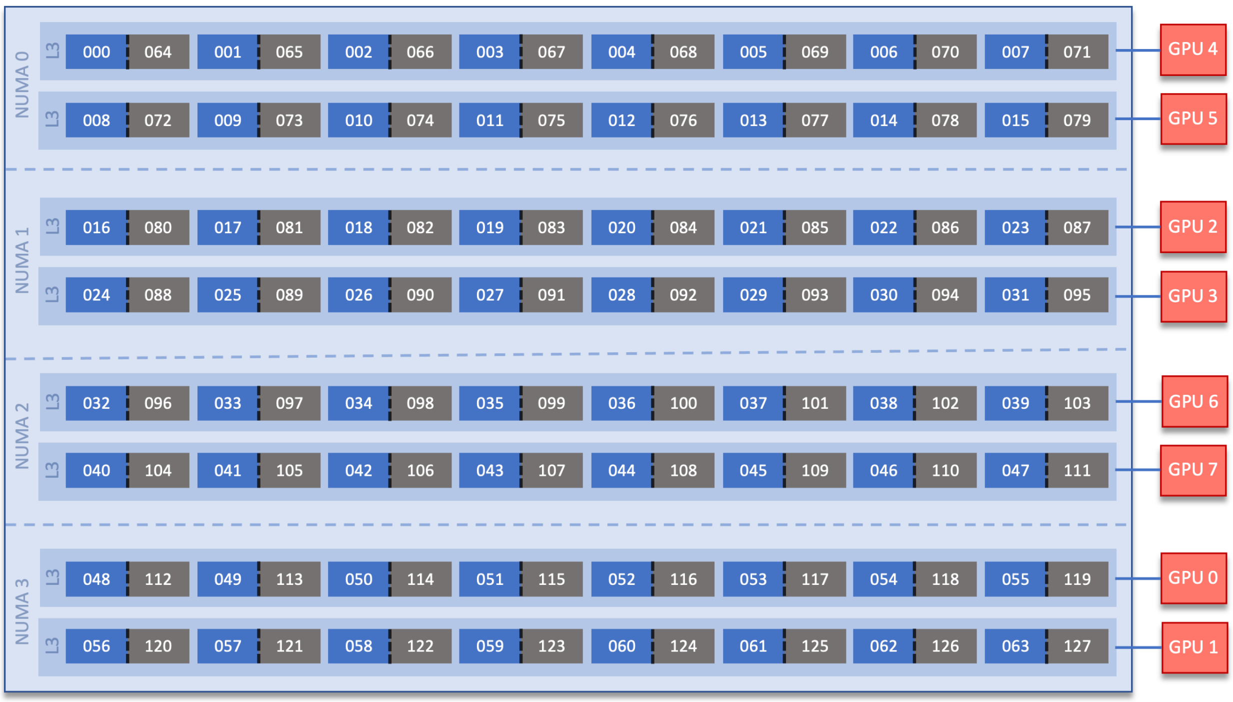
Compute-intensive, memory-intensive, or other disruptive processes running on login nodes may be killed without warning.

Note

Unlike Summit and Titan, there are no launch/batch nodes on Frontier. This means your batch script runs on a node allocated to you rather than a shared node. You still must use the job launcher (`srun`) to run parallel jobs across all of your nodes, but serial tasks need not be launched with `srun` .

Simplified Node Layout

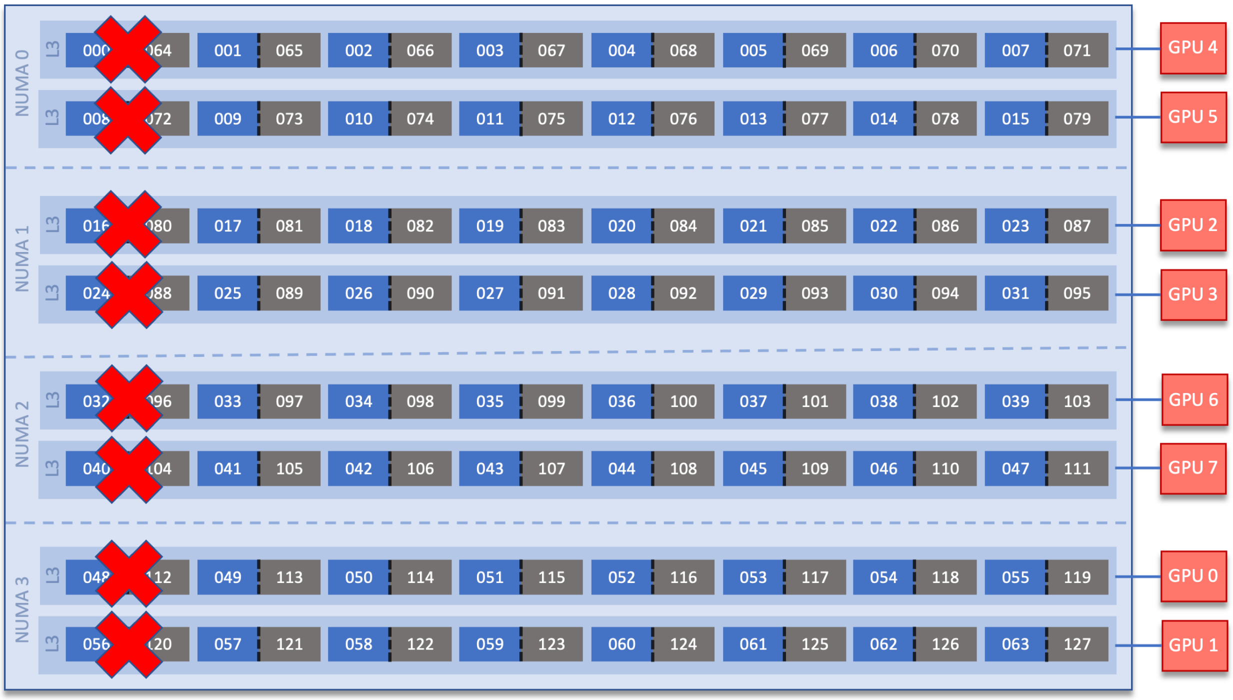
To easily visualize job examples (see :ref:`frontier-mapping` further below), the compute node diagram has been simplified to the picture shown below.


NUMA 0
L3 000 064 001 065 002 066 003 067 004 068 005 069 006 070 007 071
GPU 4
L3 008 072 009 073 010 074 011 075 012 076 013 077 014 078 015 079
GPU 5
NUMA 1
L3 016 080 017 081 018 082 019 083 020 084 021 085 022 086 023 087
GPU 2
L3 024 088 025 089 026 090 027 091 028 092 029 093 030 094 031 095
GPU 3
NUMA 2
L3 032 096 033 097 034 098 035 099 036 100 037 101 038 102 039 103
GPU 6
L3 040 104 041 105 042 106 043 107 044 108 045 109 046 110 047 111
GPU 7
NUMA 3
L3 048 112 049 113 050 114 051 115 052 116 053 117 054 118 055 119
GPU 0
L3 056 120 057 121 058 122 059 123 060 124 061 125 062 126 063 127
GPU 1

In the diagram, each **physical** core on a Frontier compute node is composed of two **logical** cores that are represented by a pair of blue and grey boxes. For a given physical core, the blue box represents the logical core of the first hardware thread, where the grey box represents the logical core of the second hardware thread.

Low-noise Mode Layout

Frontier uses low-noise mode and core specialization (`-S` flag at job allocation, e.g., `sbatch`). Low-noise mode constrains all system processes to core 0. Core specialization (by default, `-S 8`) reserves the first core in each L3 region. This prevents the user running on the core that system processes are constrained to. This also means that there are only 56 allocatable cores by default instead of 64. Therefore, this modifies the simplified node layout to:



To override this default layout (not recommended), set `-S 0` at job allocation.

Slurm

Frontier uses SchedMD's Slurm Workload Manager for scheduling and managing jobs. Slurm maintains similar functionality to other schedulers such as IBM's LSF, but provides unique control of Frontier's resources through custom commands and options specific to Slurm. A few important commands can be found in the conversion table below, but please visit SchedMD's Rosetta Stone of Workload Managers for a more complete conversion reference.

Slurm documentation for each command is available via the `man` utility, and on the web at https://slurm.schedmd.com/man_index.html. Additional documentation is available at https://slurm.schedmd.com/documentation.html.

Some common Slurm commands are summarized in the table below. More complete examples are given in the Monitoring and Modifying Batch Jobs section of this guide.

Command	Action/Task	LSF Equivalent
`squeue`	Show the current queue	`bjobs`
`sbatch`	Submit a batch script	`bsub`
`salloc`	Submit an interactive job	`bsub -Is $SHELL`

Command	Action/Task	LSF Equivalent
`srun`	Launch a parallel job	`jsrun`
`sinfo`	Show node/partition info	`bqueues` or `bhosts`
`sacct`	View accounting information for jobs/job steps	`bacct`
`scancel`	Cancel a job or job step	`bkill`
`scontrol`	View or modify job configuration.	`bstop` , `bresume` , `bmod`

Batch Scripts

The most common way to interact with the batch system is via batch scripts. A batch script is simply a shell script with added directives to request various resoruces from or provide certain information to the scheduling system. Aside from these directives, the batch script is simply the series of commands needed to set up and run your job.

To submit a batch script, use the command `sbatch myjob.sl`

Consider the following batch script:

In the script, Slurm directives are preceded by `#SBATCH` , making them appear as comments to the shell. Slurm looks for these directives through the first non-comment, non-whitespace line. Options after that will be ignored by Slurm (and the shell).

Line	Description
1	Shell interpreter line
2	OLCF project to charge
3	Job name
4	Job standard output file (`%x` will be replaced with the job name and `%j` with the Job ID)
5	Walltime requested (in `HH:MM:SS` format). See the table below for other formats.
6	Partition (queue) to use
7	Number of compute nodes requested
8	Blank line

Line	Description
9	Change into the run directory
10	Copy the input file into place
11	Run the job (add layout details)
12	Copy the output file to an appropriate location.

Interactive Jobs

Most users will find batch jobs an easy way to use the system, as they allow you to "hand off" a job to the scheduler, allowing them to focus on other tasks while their job waits in the queue and eventually runs. Occasionally, it is necessary to run interactively, especially when developing, testing, modifying or debugging a code.

Since all compute resources are managed and scheduled by Slurm, it is not possible to simply log into the system and immediately begin running parallel codes interactively. Rather, you must request the appropriate resources from Slurm and, if necessary, wait for them to become available. This is done through an "interactive batch" job. Interactive batch jobs are submitted with the `salloc` command. Resources are requested via the same options that are passed via `#SBATCH` in a regular batch script (but without the `#SBATCH` prefix). For example, to request an interactive batch job with the same resources that the batch script above requests, you would use `salloc -A ABC123 -J RunSim123 -t 1:00:00 -p batch -N 1024` . Note there is no option for an output file...you are running interactively, so standard output and standard error will be displayed to the terminal.

Common Slurm Options

The table below summarizes options for submitted jobs. Unless otherwise noted, they can be used for either batch scripts or interactive batch jobs. For scripts, they can be added on the `sbatch` command line or as a `#BSUB` directive in the batch script. (If they're specified in both places, the command line takes precedence.) This is only a subset of all available options. Check the Slurm Man Pages for a more complete list.

Option	Example Usage	Description
`-A`	`#SBATCH -A ABC123`	Specifies the project to which the job should be charged
`-N`	`#SBATCH -N 1024`	Request 1024 nodes for the job

Option	Example Usage	Description
`-t`	`#SBATCH -t 4:00:00`	Request a walltime of 4 hours. Walltime requests can be specified as minutes, hours:minutes, hours:minuts:seconds days-hours, days-hours:minutes, or days-hours:minutes:seconds
`--threads-per-core`	`#SBATCH --threads-per-core=2`	Number of active hardware threads per core. Can be 1 or 2 (1 is default) **Must** be used if using `--threads-per-core=2` in your `srun` command.
`-d`	`#SBATCH -d afterok:12345`	Specify job dependency (in this example, this job cannot start until job 12345 exits with an exit code of 0. See the Job Dependency section for more information
`-C`	`#SBATCH -C nvme`	Request the burst buffer/NVMe on each node be made available for your job. See the Burst Buffers section for more information on using them.
`-J`	`#SBATCH -J MyJob123`	Specify the job name (this will show up in queue listings)
`-o`	`#SBATCH -o jobout.%j`	File where job STDOUT will be directed (%j will be replaced with the job ID). If no -e option is specified, job STDERR will be placed in this file, too.
`-e`	`#SBATCH -e joberr.%j`	File where job STDERR will be directed (%j will be replaced with the job ID). If no -o option is

Option	Example Usage	Description
		specified, job STDOUT will be placed in this file, too.
`--mail-type`	`#SBATCH --mail-type=END`	Send email for certain job actions. Can be a comma-separated list. Actions include BEGIN, END, FAIL, REQUEUE, INVALID_DEPEND, STAGE_OUT, ALL, and more.
`--mail-user`	`#SBATCH --mail-user=user@somewhere.com`	Email address to be used for notifications.
`--reservation`	`#SBATCH --reservation=MyReservation.1`	Instructs Slurm to run a job on nodes that are part of the specified reservation.
`-S`	`#SBATCH -S 8`	Instructs Slurm to reserve a specific number of cores per node (default is 8). Reserved cores cannot be used by the application.
`--signal`	`#SBATCH --signal=USR1@300`	Send the given signal to a job the specified time (in seconds) seconds before the job reaches its walltime. The signal can be by name or by number (i.e. both 10 and USR1 would send SIGUSR1). Signaling a job can be used, for example, to force a job to write a checkpoint just before Slurm kills the job (note that this option only sends the signal; the user must still make sure their job script traps the signal and handles it in the desired manner). When used with `sbatch`, the signal can be prefixed by "B:" (e.g. `--signal=B:USR1@300`) to

Option	Example Usage	Description
		tell Slurm to signal only the batch shell; otherwise all processes will be signaled.

Slurm Environment Variables

Slurm reads a number of environment variables, many of which can provide the same information as the job options noted above. We recommend using the job options rather than environment variables to specify job options, as it allows you to have everything self-contained within the job submission script (rather than having to remember what options you set for a given job).

Slurm also provides a number of environment variables within your running job. The following table summarizes those that may be particularly useful within your job (e.g. for naming output log files):

Variable	Description
`$SLURM_SUBMIT_DIR`	The directory from which the batch job was submitted. By default, a new job starts in your home directory. You can get back to the directory of job submission with `cd $SLURM_SUBMIT_DIR`. Note that this is not necessarily the same directory in which the batch script resides.
`$SLURM_JOBID`	The job’s full identifier. A common use for `$SLURM_JOBID` is to append the job’s ID to the standard output and error files.
`$SLURM_JOB_NUM_NODES`	The number of nodes requested.
`$SLURM_JOB_NAME`	The job name supplied by the user.
`$SLURM_NODELIST`	The list of nodes assigned to the job.

Job States

A job will transition through several states during its lifetime. Common ones include:

State Code	State	Description
CA	Canceled	The job was canceled (could've been by the user or an administrator)
CD	Completed	The job completed successfully (exit code 0)
CG	Completing	The job is in the process of completing (some processes may still be running)
PD	Pending	The job is waiting for resources to be allocated
R	Running	The job is currently running

Job Reason Codes

In addition to state codes, jobs that are pending will have a "reason code" to explain why the job is pending. Completed jobs will have a reason describing how the job ended. Some codes you might see include:

Reason	Meaning
Dependency	Job has dependencies that have not been met
JobHeldUser	Job is held at user's request
JobHeldAdmin	Job is held at system administrator's request
Priority	Other jobs with higher priority exist for the partition/reservation
Reservation	The job is waiting for its reservation to become available
AssocMaxJobsLimit	The job is being held because the user/project has hit the limit on running jobs
ReqNodeNotAvail	The requested a particular node, but it's currently unavailable (it's in use, reserved, down, draining, etc.)
JobLaunchFailure	Job failed to launch (could due to system problems, invalid program name, etc.)
NonZeroExitCode	The job exited with some code other than 0

Many other states and job reason codes exist. For a more complete description, see the `squeue` man page (either on the system or online).

Scheduling Policy

In a simple batch queue system, jobs run in a first-in, first-out (FIFO) order. This can lead to inefficient use of the system. If a large job is the next to run, a strict FIFO queue can cause nodes to sit idle while waiting for the large job to start. *Backfilling* would allow smaller, shorter jobs to use those resources that would otherwise remain idle until the large job starts. With the proper algorithm, they would do so without impacting the start time of the large job. While this does make more efficient use of the system, it encourages the submission of smaller jobs.

The DOE Leadership-Class Job Mandate

As a DOE Leadership Computing Facility, OLCF has a mandate that a large portion of Frontier's usage come from large, *leadership-class* (a.k.a. *capability*) jobs. To ensure that OLCF complies with this directive, we strongly encourage users to run jobs on Frontier that are as large as their code will allow. To that end, OLCF implements queue policies that enable large jobs to run in a timely fashion.

Note

The OLCF implements queue policies that encourage the submission and timely execution of large, leadership-class jobs on Frontier.

The basic priority mechanism for jobs waiting in the queue is the time the job has been waiting in the queue. If your jobs require resources outside these policies such as higher priority or longer walltimes, please contact help@olcf.ornl.gov

Job Priority by Node Count

Jobs are *aged* according to the job's requested node count (older age equals higher queue priority). Each job's requested node count places it into a specific *bin*. Each bin has a different aging parameter, which all jobs in the bin receive.

Bin	Min Nodes	Max Nodes	Max Walltime (Hours)	Aging Boost (Days)
1	5,645	9,408	12.0	15
2	1,882	5,644	12.0	10
3	184	1,881	12.0	0
4	92	183	6.0	0
5	1	91	2.0	0

`batch` Queue Policy

The `batch` queue is the default queue for production work on Frontier. Most work on Frontier is handled through this queue. The following policies are enforced for the `batch` queue:

- Limit of four *eligible-to-run* jobs per user. (Jobs in excess of this number will be held, but will move to the eligible-to-run state at the appropriate time.)
- Users may have only 100 jobs queued in the `batch` queue at any time (this includes jobs in all states). Additional jobs will be rejected at submit time.

`debug` Quality of Service Class

The `debug` quality of service (QOS) class can be used to access Frontier's compute resources for short non-production debug tasks. The QOS provides a higher priority compare to jobs of the same job size bin in production queues. Production work and job chaining using the `debug` QOS is prohibited. Each user is limited to one job in any state at any one point. Attempts to submit multiple jobs to this QOS will be rejected upon job submission.

To submit a job to the `debug` QOS, add the -q debug option to your `sbatch` or `salloc` command or `#SBATCH –q debug` to your job script.

Allocation Overuse Policy

Projects that overrun their allocation are still allowed to run on OLCF systems, although at a reduced priority. Like the adjustment for the number of processors requested above, this is an adjustment to the apparent submit time of the job. However, this adjustment has the effect of making jobs appear much younger than jobs submitted under projects that have not exceeded their allocation. In addition to the priority change, these jobs are also limited in the amount of wall time that can be used. For example, consider that job1 is submitted at the same time as job2. The project associated with job1 is over its allocation, while the project for job2 is not. The batch system will consider job2 to have been waiting for a longer time than job1. Additionally, projects that are at 125% of their allocated time will be limited to only 3 running jobs at a time. The adjustment to the apparent submit time depends upon the percentage that the project is over its allocation, as shown in the table below:

% of Allocation Used	Priority Reduction
< 100%	none
>=100% but <=125%	30 days

% of Allocation Used	Priority Reduction
> 125%	365 days

System Reservation Policy

Projects may request to reserve a set of nodes for a period of time by contacting help@olcf.ornl.gov. If the reservation is granted, the reserved nodes will be blocked from general use for a given period of time. Only users that have been authorized to use the reservation can utilize those resources. Since no other users can access the reserved resources, it is crucial that groups given reservations take care to ensure the utilization on those resources remains high. To prevent reserved resources from remaining idle for an extended period of time, reservations are monitored for inactivity. If activity falls below 50% of the reserved resources for more than (30) minutes, the reservation will be canceled and the system will be returned to normal scheduling. A new reservation must be requested if this occurs.

The requesting project’s allocation is charged according to the time window granted, regardless of actual utilization. For example, an 8-hour, 2,000 node reservation on Frontier would be equivalent to using 16,000 Frontier node-hours of a project’s allocation.

Note

Reservations should not be confused with priority requests. If quick turnaround is needed for a few jobs or for a period of time, a priority boost should be requested. A reservation should only be requested if users need to guarantee availability of a set of nodes at a given time, such as for a live demonstration at a conference.

Job Dependencies

Oftentimes, a job will need data from some other job in the queue, but it's nonetheless convenient to submit the second job before the first finishes. Slurm allows you to submit a job with constraints that will keep it from running until these dependencies are met. These are specified with the `-d` option to Slurm. Common dependency flags are summarized below. In each of these examples, only a single jobid is shown but you can specify multiple job IDs as a colon-delimited list (i.e. `#SBATCH -d afterok:12345:12346:12346`). For the `after` dependency, you can optionally specify a `+time` value for each jobid.

Flag	Meaning (for the dependent job)
`#SBATCH -d after:jobid[+time]`	The job can start after the specified jobs start or are canceled. The optional `+time` argument is a number of minutes. If specified, the job cannot start until that many

Flag	Meaning (for the dependent job)
	minutes have passed since the listed jobs start/are canceled. If not specified, there is no delay.
`#SBATCH -d afterany:jobid`	The job can start after the specified jobs have ended (regardless of exit state)
`#SBATCH -d afternotok:jobid`	The job can start after the specified jobs terminate in a failed (non-zero) state
`#SBATCH -d afterok:jobid`	The job can start after the specified jobs complete successfully (i.e. zero exit code)
`#SBATCH -d singleton`	Job can begin after any previously-launched job with the same name and from the same user have completed. In other words, serialize the running jobs based on username+jobname pairs.

Monitoring and Modifying Batch Jobs

`scontrol hold` and `scontrol release`: Holding and Releasing Jobs

Sometimes you may need to place a hold on a job to keep it from starting. For example, you may have submitted it assuming some needed data was in place but later realized that data is not yet available. This can be done with the `scontrol hold` command. Later, when the data is ready, you can release the job (i.e. tell the system that it's now OK to run the job) with the `scontrol release` command. For example:

`scontrol hold 12345`	Place job 12345 on hold
`scontrol release 12345`	Release job 12345 (i.e. tell the system it's OK to run it)

`scontrol update`: Changing Job Parameters

There may also be occasions where you want to modify a job that's waiting in the queue. For example, perhaps you requested 2,000 nodes but later realized this is a different data set and only needs 1,000 nodes. You can use the `scontrol update` command for this. For example:

`scontrol update NumNodes=1000 JobID=12345`	Change job 12345's node request to 1000 nodes

`scontrol update TimeLimit=4:00:00 JobID=12345`	Change job 12345's max walltime to 4 hours

`scancel`: Cancel or Signal a Job

In addition to the `--signal` option for the `sbatch` / `salloc` commands described :ref:`above <common-slurm-options>`, the `scancel` command can be used to manually signal a job. Typically, this is used to remove a job from the queue. In this use case, you do not need to specify a signal and can simply provide the jobid (i.e. `scancel 12345`). If you want to send some other signal to the job, use `scancel` the with the `-s` option. The `-s` option allows signals to be specified either by number or by name. Thus, if you want to send `SIGUSR1` to a job, you would use `scancel -s 10 12345` or `scancel -s USR1 12345`.

`squeue`: View the Queue

The `squeue` command is used to show the batch queue. You can filter the level of detail through several command-line options. For example:

`squeue -l`	Show all jobs currently in the queue
`squeue -l -u $USER`	Show all of *your* jobs currently in the queue

`sacct`: Get Job Accounting Information

The `sacct` command gives detailed information about jobs currently in the queue and recently-completed jobs. You can also use it to see the various steps within a batch jobs.

`sacct -a -X`	Show all jobs (`-a`) in the queue, but summarize the whole allocation instead of showing individual steps (`-X`)
`sacct -u $USER`	Show all of your jobs, and show the individual steps (since there was no `-X` option)
`sacct -j 12345`	Show all job steps that are part of job 12345
`sacct -u $USER -S 2022-07-01T13:00:00 -o "jobid%5,jobname%25,nodelist%20" -X`	Show all of your jobs since 1 PM on July 1, 2022 using a particular output format

`scontrol show job`: Get Detailed Job Information

In addition to holding, releasing, and updating the job, the `scontrol` command can show detailed job information via the `show job` subcommand. For example, `scontrol show job 12345`.

Srun

The default job launcher for Frontier is srun . The `srun` command is used to execute an MPI binary on one or more compute nodes in parallel.

Srun Format

```
srun  [OPTIONS... [executable [args...]]]
```

Single Command (non-interactive)

```
$ srun -A <project_id> -t 00:05:00 -p <partition> -N 2 -n 4 --ntasks-per-
<output printed to terminal>
```

The job name and output options have been removed since stdout/stderr are typically desired in the terminal window in this usage mode.

`srun` accepts the following common options:

`-N`	Number of nodes
`-n`	Total number of MPI tasks (default is 1)
`-c, --cpus-per-task=<ncpus>`	Logical cores per MPI task (default is 1) When used with `--threads-per-core=1` : `-c` is equivalent to *physical* cores per task By default, when `-c > 1` , additional cores per task are distributed within one L3 region first before filling a different L3 region.
`--cpu-bind=threads`	Bind tasks to CPUs. `threads` - (default, recommended) Automatically generate masks binding tasks to threads.
`--threads-per-core=<threads>`	In task layout, use the specified maximum number of hardware threads per core

	(default is 1; there are 2 hardware threads per physical CPU core). Must also be set in `salloc` or `sbatch` if using `--threads-per-core=2` in your `srun` command.
`-m, --distribution=<value>:<value>:<value>`	Specifies the distribution of MPI ranks across compute nodes, sockets (L3 regions), and cores, respectively. The default values are `block:cyclic:cyclic`, see `man srun` for more information. Currently, the distribution setting for cores (the third "<value>" entry) has no effect on Frontier
`--ntasks-per-node=<ntasks>`	If used without `-n`: requests that a specific number of tasks be invoked on each node. If used with `-n`: treated as a *maximum* count of tasks per node.
`--gpus`	Specify the number of GPUs required for the job (total GPUs across all nodes).
`--gpus-per-node`	Specify the number of GPUs per node required for the job.
`--gpu-bind=closest`	Binds each task to the GPU which is on the same NUMA domain as the CPU core the MPI rank is running on.
`--gpu-bind=map_gpu:<list>`	Bind tasks to specific GPUs by setting GPU masks on tasks (or ranks) as specified where `<list>` is `<gpu_id_for_task_0>,<gpu_id_for_task_1>,...`. If the number of tasks (or ranks) exceeds the number of elements in this list, elements in the list will be reused as needed starting from the beginning of the list. To simplify support for large task counts, the lists may follow a map with an asterisk and repetition count. (For example `map_gpu:0*4,1*4`)
`--ntasks-per-gpu=<ntasks>`	Request that there are ntasks tasks invoked for every GPU.

Below is a comparison table between `srun` and `jsrun`.

Option	jsrun (Summit)	srun (Frontier)
Number of nodes	`-nnodes`	`-N, --nnodes`
Number of tasks	defined with resource set	`-n, --ntasks`

Option	jsrun (Summit)	srun (Frontier)
Number of tasks per node	defined with resource set	`--ntasks-per-node`
Number of CPUs per task	defined with resource set	`-c, --cpus-per-task`
Number of resource sets	`-n, --nrs`	N/A
Number of resource sets per host	`-r, --rs_per_host`	N/A
Number of tasks per resource set	`-a, --tasks_per_rs`	N/A
Number of CPUs per resource set	`-c, --cpus_per_rs`	N/A
Number of GPUs per resource set	`-g, --gpus_per_rs`	N/A
Bind tasks to allocated CPUs	`-b, --bind`	`--cpu-bind`
Performance binding preference	`-l,--latency_priority`	`--hint`
Specify the task to resource mapping pattern	`--launch_distribution`	`-m, --distribution`

Process and Thread Mapping Examples

This section describes how to map processes (e.g., MPI ranks) and process threads (e.g., OpenMP threads) to the CPUs and GPUs on Frontier.

Users are highly encouraged to use the CPU- and GPU-mapping programs used in the following sections to check their understanding of the job steps (i.e., `srun` commands) they intend to use in their actual jobs.

- For the :ref:`frontier-cpu-map` and :ref:`frontier-multi-map` sections:

  A simple MPI+OpenMP "Hello, World" program (hello_mpi_omp) will be used to clarify the mappings.

- For the :ref:`frontier-gpu-map` section:

  An MPI+OpenMP+HIP "Hello, World" program (hello_jobstep) will be used to clarify the GPU mappings.

Additionally, it may be helpful to cross reference the :ref:`simplified Frontier node diagram <frontier-simple>` -- specifically the :ref:`low-noise mode diagram <frontier-lownoise>`.

Warning

Unless specified otherwise, the examples below assume the default low-noise core specialization setting (`-S 8`). This means that there are only 56 allocatable cores by default instead of 64. See the :ref:`frontier-lownoise` section for more details. Set `-S 0` at job allocation to override this setting.

CPU Mapping

This subsection covers how to map tasks to the CPU without the presence of additional threads (i.e., solely MPI tasks -- no additional OpenMP threads).

The intent with both of the following examples is to launch 8 MPI ranks across the node where each rank is assigned its own logical (and, in this case, physical) core. Using the `-m` distribution flag, we will cover two common approaches to assign the MPI ranks -- in a "round-robin" (`cyclic`) configuration and in a "packed" (`block`) configuration. Slurm's :ref:`frontier-interactive` method was used to request an allocation of 1 compute node for these examples: `salloc -A <project_id> -t 30 -p <parition> -N 1`

Note

There are many different ways users might choose to perform these mappings, so users are encouraged to clone the `hello_mpi_omp` program and test whether or not processes and threads are running where intended.

8 MPI Ranks (round-robin)

Assigning MPI ranks in a "round-robin" (`cyclic`) manner across L3 cache regions (sockets) is the default behavior on Frontier. This mode will assign consecutive MPI tasks to different sockets before it tries to "fill up" a socket.

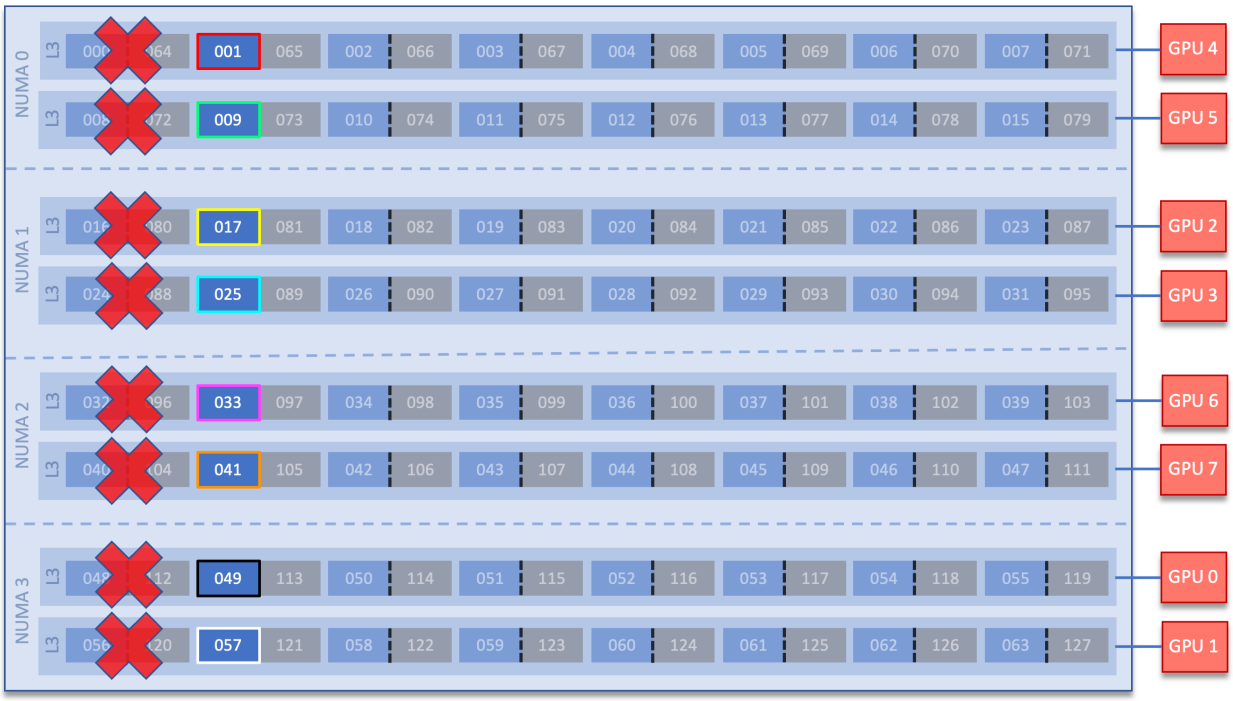
Recall that the `-m` flag behaves like: `-m <node distribution>:<socket distribution>` . Hence, the key setting to achieving the round-robin nature is the `-m block:cyclic` flag, specifically the `cyclic` setting provided for the "socket distribution". This ensures that the MPI tasks will be distributed across sockets in a cyclic (round-robin) manner.

The below `srun` command will achieve the intended 8 MPI "round-robin" layout:

```
$ export OMP_NUM_THREADS=1
$ srun -N1 -n8 -c1 --cpu-bind=threads --threads-per-core=1 -m block:cycl

MPI 000 - OMP 000 - HWT 001 - Node frontier00144
MPI 001 - OMP 000 - HWT 009 - Node frontier00144
MPI 002 - OMP 000 - HWT 017 - Node frontier00144
MPI 003 - OMP 000 - HWT 025 - Node frontier00144
MPI 004 - OMP 000 - HWT 033 - Node frontier00144
MPI 005 - OMP 000 - HWT 041 - Node frontier00144
MPI 006 - OMP 000 - HWT 049 - Node frontier00144
MPI 007 - OMP 000 - HWT 057 - Node frontier00144
```

srun -N1 -n8 -c1 --cpu-bind=threads --threads-per-core=1 -m block:cyclic



Breaking down the `srun` command, we have:

- `-N1` : indicates we are using 1 node
- `-n8` : indicates we are launching 8 MPI tasks
- `-c1` : indicates we are assigning 1 logical core per MPI task. In this case, because of `--threads-per-core=1` , this also means 1 **physical** core per MPI task.
- `--cpu-bind=threads` : binds tasks to threads
- `--threads-per-core=1` : use a maximum of 1 hardware thread per physical core (i.e., only use 1 logical core per physical core)
- `-m block:cyclic` : distribute the tasks in a block layout across nodes (default), and in a **cyclic** (round-robin) layout across L3 sockets
- `./hello_mpi_omp` : launches the "hello_mpi_omp" executable
- `| sort` : sorts the output

Note

Although the above command used the default settings `-c1` , `--cpu-bind=threads` , `--threads-per-core=1` and `-m block:cyclic` , it is always better to be explicit with your `srun` command to have more control over your node layout. The above command is equivalent to `srun -N1 -n8` .
As you can see in the node diagram above, this results in the 8 MPI tasks (outlined in different colors) being distributed "vertically" across L3 sockets.

7 MPI Ranks (packed)

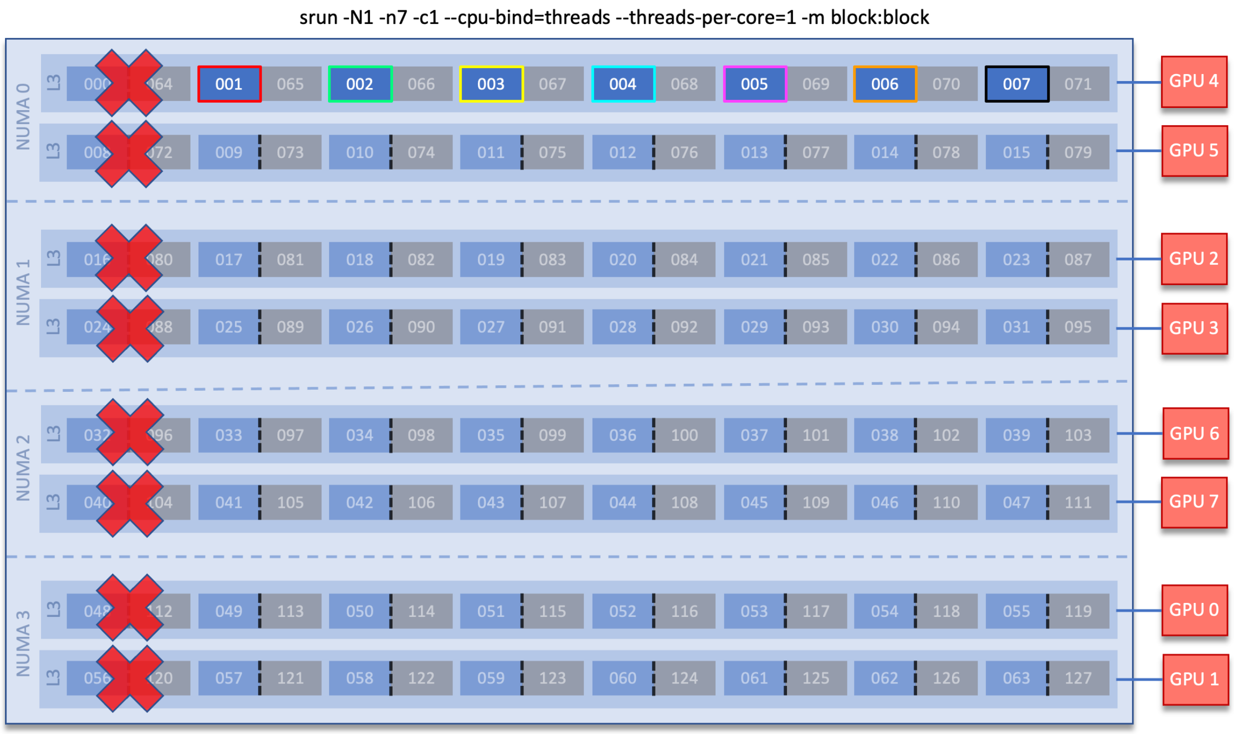
Instead, you can assign MPI ranks so that the L3 regions are filled in a "packed" (`block`) manner. This mode will assign consecutive MPI tasks to the same L3 region (socket) until it is "filled up" or "packed" before assigning a task to a different socket.

Recall that the `-m` flag behaves like: `-m <node distribution>:<socket distribution>` . Hence, the key setting to achieving the round-robin nature is the `-m block:block` flag, specifically the `block` setting provided for the "socket distribution". This ensures that the MPI tasks will be distributed in a packed manner.

The below `srun` command will achieve the intended 7 MPI "packed" layout:

```
$ export OMP_NUM_THREADS=1
$ srun -N1 -n7 -c1 --cpu-bind=threads --threads-per-core=1 -m block:bloc

MPI 000 - OMP 000 - HWT 001 - Node frontier00144
MPI 001 - OMP 000 - HWT 002 - Node frontier00144
MPI 002 - OMP 000 - HWT 003 - Node frontier00144
MPI 003 - OMP 000 - HWT 004 - Node frontier00144
MPI 004 - OMP 000 - HWT 005 - Node frontier00144
MPI 005 - OMP 000 - HWT 006 - Node frontier00144
MPI 006 - OMP 000 - HWT 007 - Node frontier00144
```


srun -N1 -n7 -c1 --cpu-bind=threads --threads-per-core=1 -m block:block
NUMA 0
NUMA 1
NUMA 2
NUMA 3
L3
GPU 4
GPU 5
GPU 2
GPU 3
GPU 6
GPU 7
GPU 0
GPU 1

Breaking down the `srun` command, the only difference than the previous example is:

- `-m block:block` : distribute the tasks in a block layout across nodes (default), and in a **block** (packed) socket layout

As you can see in the node diagram above, this results in the 7 MPI tasks (outlined in different colors) being distributed "horizontally" *within* a socket, rather than being spread across different L3 sockets like with the previous example. However, if an 8th task was requested it would be assigned to the next L3 region on core 009.

Multithreading

Because a Frontier compute node has two hardware threads available (2 logical cores per physical core), this enables the possibility of multithreading your application (e.g., with OpenMP threads). Although the additional hardware threads can be assigned to additional MPI tasks, this is not recommended. It is highly recommended to only use 1 MPI task per physical core and to use OpenMP threads instead on any additional logical cores gained when using both hardware threads.

The following examples cover multithreading with hybrid MPI+OpenMP applications. In these examples, Slurm's :ref:`frontier-interactive` method was used to request an allocation of 1 compute node: `salloc -A <project_id> -t 30 -p <parition> -N 1`

Note

There are many different ways users might choose to perform these mappings, so users are encouraged to clone the `hello_mpi_omp` program and test whether or not processes and threads are running where intended.

2 MPI ranks - each with 2 OpenMP threads

In this example, the intent is to launch 2 MPI ranks, each of which spawn 2 OpenMP threads, and have all of the 4 OpenMP threads run on different physical CPU cores.

First (INCORRECT) attempt

To set the number of OpenMP threads spawned per MPI rank, the `OMP_NUM_THREADS` environment variable can be used. To set the number of MPI ranks launched, the `srun` flag `-n` can be used.

```
$ export OMP_NUM_THREADS=2
$ srun -N1 -n2 ./hello_mpi_omp | sort

WARNING: Requested total thread count and/or thread affinity may result i
oversubscription of available CPU resources!  Performance may be degrade
Explicitly set OMP_WAIT_POLICY=PASSIVE or ACTIVE to suppress this messag
Set CRAY_OMP_CHECK_AFFINITY=TRUE to print detailed thread-affinity messa
WARNING: Requested total thread count and/or thread affinity may result i
oversubscription of available CPU resources!  Performance may be degrade
Explicitly set OMP_WAIT_POLICY=PASSIVE or ACTIVE to suppress this messag
Set CRAY_OMP_CHECK_AFFINITY=TRUE to print detailed thread-affinity messa

MPI 000 - OMP 000 - HWT 001 - Node frontier001
MPI 000 - OMP 001 - HWT 001 - Node frontier001
MPI 001 - OMP 000 - HWT 009 - Node frontier001
MPI 001 - OMP 001 - HWT 009 - Node frontier001
```

The first thing to notice here is the `WARNING` about oversubscribing the available CPU cores. Also, the output shows each MPI rank did spawn 2 OpenMP threads, but both OpenMP threads ran on the same logical core (for a given MPI rank). This was not the intended behavior; each OpenMP thread was meant to run on its own physical CPU core.

The problem here arises from two default settings; 1) each MPI rank is only allocated 1 logical core (`-c 1`) and, 2) only 1 hardware thread per physical CPU core is enabled (`--threads-per-core=1`). When using `--threads-per-core=1` and `--cpu-bind=threads` (the default setting), 1 logical core in `-c` is equivalent to 1 physical core. So in this case, each MPI rank only has 1 physical core (with 1 hardware thread) to run on - including any threads the process spawns - hence the WARNING and undesired behavior.

Second (CORRECT) attempt

Recall that in this scenario, because of the `--threads-per-core=1` setting, 1 logical core is equivalent to 1 physical core when using `-c`. Therefore, in order for each OpenMP thread to run on its own physical CPU core, each MPI rank should be given 2 physical CPU cores (`-c 2`). Now the OpenMP threads will be mapped to unique hardware threads on separate physical CPU cores.

```
$ export OMP_NUM_THREADS=2
$ srun -N1 -n2 -c2 ./hello_mpi_omp | sort

MPI 000 - OMP 000 - HWT 001 - Node frontier001
MPI 000 - OMP 001 - HWT 002 - Node frontier001
MPI 001 - OMP 000 - HWT 009 - Node frontier001
MPI 001 - OMP 001 - HWT 010 - Node frontier001
```

Now the output shows that each OpenMP thread ran on its own physical CPU core. More specifically (see the Frontier Compute Node diagram), OpenMP thread 000 of MPI rank 000 ran on logical core 001 (i.e., physical CPU core 01), OpenMP thread 001 of MPI rank 000 ran on logical core 002 (i.e., physical CPU core 02), OpenMP thread 000 of MPI rank 001 ran on logical core 009 (i.e., physical CPU core 09), and OpenMP thread 001 of MPI rank 001 ran on logical core 010 (i.e., physical CPU core 10) - as intended.

Third attempt - Using multiple threads per core

To use both available hardware threads per core, the *job* must be allocated with `--threads-per-core=2` (as opposed to only the job step - i.e., `srun` command). That value will then be inherited by `srun` unless explcitly overridden with `--threads-per-core=1`. Because we are using `--threads-per-core=2`, the usage of `-c` goes back to purely meaning the amount of **logical** cores (i.e., it is no longer equivalent to 1 physical core).

```
$ salloc -N1 -A <project_id> -t <time> -p <partition> --threads-per-core

$ export OMP_NUM_THREADS=2
$ srun -N1 -n2 -c2 ./hello_mpi_omp | sort

MPI 000 - OMP 000 - HWT 001 - Node frontier001
MPI 000 - OMP 001 - HWT 065 - Node frontier001
MPI 001 - OMP 000 - HWT 009 - Node frontier001
MPI 001 - OMP 001 - HWT 073 - Node frontier001
```

Comparing this output to the Frontier Compute Node diagram, we see that each pair of OpenMP threads is contained within a single physical core. MPI rank 000 ran on logical cores 001 and 065 (i.e. physical CPU core 01) and MPI rank 001 ran on logical cores 009 and 073 (i.e. physical CPU core 09).

GPU Mapping

In this sub-section, an MPI+OpenMP+HIP "Hello, World" program (hello_jobstep) will be used to clarify the GPU mappings. Again, Slurm's :ref:`frontier-interactive` method was used to request an allocation of 2 compute nodes for these examples: `salloc -A <project_id> -t 30 -p <parition> -N 2`. The CPU mapping part of this example is very similar to the example used above in the Multithreading sub-section, so the focus here will be on the GPU mapping part.

In general, GPU mapping can be accomplished in different ways. For example, an application might map MPI ranks to GPUs programmatically within the code using, say, `hipSetDevice`. In this case, since all GPUs on a node are available to all MPI ranks on that node by default, there might not be a need to map to GPUs using Slurm (just do it in the code). However, in another application, there might be a reason to make only a subset of GPUs available to the MPI ranks on a node. It is this latter case that the following examples refer to.

Note

There are many different ways users might choose to perform these mappings, so users are encouraged to clone the `hello_jobstep` program and test whether or not processes and threads are running where intended.

Warning

Due to the unique architecture of Frontier compute nodes and the way that Slurm currently allocates GPUs and CPU cores to job steps, it is suggested that all 8 GPUs on a node are allocated to the job step to ensure that optimal bindings are possible.

Mapping 1 task per GPU

In the following examples, each MPI rank (and its OpenMP threads) will be mapped to a single GPU.

Example 0: 1 MPI rank with 1 OpenMP thread and 1 GPU (single-node)

Somewhat counterintuitively, this common test case is currently among the most difficult. Slurm ignores GPU bindings for nodes with only a single task, so we do not use `--gpu-bind` here. We must allocate only a single GPU to ensure that only one GPU is available to the task, and since we get the first GPU available we should bind the task to the CPU closest to the allocated GPU.

```
$ export OMP_NUM_THREADS=1
$ srun -N1 -n1 -c1 --cpu-bind=map_cpu:49 --gpus=1 ./hello_jobstep

MPI 000 - OMP 000 - HWT 049 - Node frontier001 - RT_GPU_ID 0 - GPU_ID 0 -
```

Alternatively, you can combine the `--gpus-per-task` flag with `--gpu-bind` :

```
$ export OMP_NUM_THREADS=1
$ srun -N1 -n1 -c1 --gpus-per-task=1 --gpu-bind=closest ./hello_jobstep

MPI 000 - OMP 000 - HWT 049 - Node frontier001 - RT_GPU_ID 0 - GPU_ID 0 -
```

Example 1: 8 MPI ranks - each with 1 OpenMP thread, 1 GPU, and 7 cores (single-node)

This example launches 8 MPI ranks (`-n8`), each with 7 physical CPU cores (`-c7`) and 1 OpenMP thread (`OMP_NUM_THREADS=1`). Because of Frontier's default core specialization (`-S 8`), this effectively maps all remaining 7 cores in a given L3 cache region to a given GPU. If desired, the additional cores can also be used with more OpenMP threads until all cores are filled.

```
$ export OMP_NUM_THREADS=1
$ srun -N1 -n8 -c7 --cpu-bind=threads --threads-per-core=1 --gpus-per-ta

MPI 000 - OMP 000 - HWT 007 - Node frontier05717 - RT_GPU_ID 0 - GPU_ID
MPI 001 - OMP 000 - HWT 015 - Node frontier05717 - RT_GPU_ID 0 - GPU_ID
MPI 002 - OMP 000 - HWT 023 - Node frontier05717 - RT_GPU_ID 0 - GPU_ID
MPI 003 - OMP 000 - HWT 031 - Node frontier05717 - RT_GPU_ID 0 - GPU_ID
MPI 004 - OMP 000 - HWT 039 - Node frontier05717 - RT_GPU_ID 0 - GPU_ID
MPI 005 - OMP 000 - HWT 047 - Node frontier05717 - RT_GPU_ID 0 - GPU_ID
MPI 006 - OMP 000 - HWT 055 - Node frontier05717 - RT_GPU_ID 0 - GPU_ID
MPI 007 - OMP 000 - HWT 063 - Node frontier05717 - RT_GPU_ID 0 - GPU_ID
```

Example 2: 8 MPI ranks - each with 2 OpenMP threads and 1 GPU (single-node)

This example launches 8 MPI ranks (`-n8`), each with 2 physical CPU cores (`-c2`) to launch 2 OpenMP threads (`OMP_NUM_THREADS=2`) on. In addition, each MPI rank (and its 2 OpenMP threads) should have access to only 1 GPU. To accomplish the GPU mapping, two new `srun` options will be used:

- `--gpus-per-node` specifies the number of GPUs required for the job
- `--gpu-bind=closest` binds each task to the GPU which is closest.

Note

Without these additional flags, all MPI ranks would have access to all GPUs (which is the default behavior).

```
$ export OMP_NUM_THREADS=2
$ srun -N1 -n8 -c2 --gpus-per-node=8 --gpu-bind=closest ./hello_jobstep

MPI 000 - OMP 000 - HWT 001 - Node frontier08413 - RT_GPU_ID 0 - GPU_ID
MPI 000 - OMP 001 - HWT 002 - Node frontier08413 - RT_GPU_ID 0 - GPU_ID
MPI 001 - OMP 000 - HWT 009 - Node frontier08413 - RT_GPU_ID 0 - GPU_ID
MPI 001 - OMP 001 - HWT 010 - Node frontier08413 - RT_GPU_ID 0 - GPU_ID
MPI 002 - OMP 000 - HWT 017 - Node frontier08413 - RT_GPU_ID 0 - GPU_ID
MPI 002 - OMP 001 - HWT 018 - Node frontier08413 - RT_GPU_ID 0 - GPU_ID
MPI 003 - OMP 000 - HWT 025 - Node frontier08413 - RT_GPU_ID 0 - GPU_ID
MPI 003 - OMP 001 - HWT 026 - Node frontier08413 - RT_GPU_ID 0 - GPU_ID
MPI 004 - OMP 000 - HWT 033 - Node frontier08413 - RT_GPU_ID 0 - GPU_ID
MPI 004 - OMP 001 - HWT 034 - Node frontier08413 - RT_GPU_ID 0 - GPU_ID
MPI 005 - OMP 000 - HWT 041 - Node frontier08413 - RT_GPU_ID 0 - GPU_ID
MPI 005 - OMP 001 - HWT 042 - Node frontier08413 - RT_GPU_ID 0 - GPU_ID
MPI 006 - OMP 000 - HWT 049 - Node frontier08413 - RT_GPU_ID 0 - GPU_ID
MPI 006 - OMP 001 - HWT 050 - Node frontier08413 - RT_GPU_ID 0 - GPU_ID
MPI 007 - OMP 000 - HWT 057 - Node frontier08413 - RT_GPU_ID 0 - GPU_ID
MPI 007 - OMP 001 - HWT 058 - Node frontier08413 - RT_GPU_ID 0 - GPU_ID
```

The output from the program contains a lot of information, so let's unpack it. First, there are different IDs associated with the GPUs so it is important to describe them before moving on. `GPU_ID` is the node-level (or global) GPU ID, which is labeled as one might expect from looking at the Frontier Node Diagram: 0, 1, 2, 3, 4, 5, 6, 7. `RT_GPU_ID` is the HIP runtime GPU ID, which can be though of as each MPI rank's local GPU ID number (with zero-based indexing). So in the output above, each MPI rank has access to only 1 unique GPU - where MPI 000 has access to "global" GPU 4, MPI 001 has access to "global" GPU 5, etc., but all MPI ranks show a HIP runtime GPU ID of 0. The reason is that each MPI rank only "sees" one GPU and so the HIP runtime labels it as "0", even though it might be global GPU ID 0, 1, 2, 3, 4, 5, 6, or 7. The GPU's bus ID is included to definitively show that different GPUs are being used.

Here is a summary of the different GPU IDs reported by the example program:

- `GPU_ID` is the node-level (or global) GPU ID read from `ROCR_VISIBLE_DEVICES`. If this environment variable is not set (either by the user or by Slurm), the value of `GPU_ID` will be set to `N/A` by this program.
- `RT_GPU_ID` is the HIP runtime GPU ID (as reported from, say `hipGetDevice`).
- `Bus_ID` is the physical bus ID associated with the GPUs. Comparing the bus IDs is meant to definitively show that different GPUs are being used.

So the job step (i.e., `srun` command) used above gave the desired output. Each MPI rank spawned 2 OpenMP threads and had access to a unique GPU. The `--gpus-per-node=8` allocated 8 GPUs for node and the `--gpu-bind=closest` ensured that the closest GPU to each rank was the one used.

Note

This example shows an important peculiarity of the Frontier nodes; the "closest" GPUs to each MPI rank are not in sequential order. For example, MPI rank 000 and its two OpenMP threads ran on hardware threads 000 and 001. As can be seen in the Frontier node diagram, these two hardware threads reside in the same L3 cache region, and that L3 region is connected via Infinity Fabric (blue line in the diagram) to GPU 4. This is an important distinction that can affect performance if not considered carefully.

Example 3: 16 MPI ranks - each with 2 OpenMP threads and 1 GPU (multi-node)

This example will extend Example 2 to run on 2 nodes. As the output shows, it is a very straightforward exercise of changing the number of nodes to 2 (`-N2`) and the number of MPI ranks to 16 (`-n16`).

```
$ export OMP_NUM_THREADS=2
$ srun -N2 -n16 -c2 --gpus-per-node=8 --gpu-bind=closest ./hello_jobstep
```

```
MPI 000 - OMP 000 - HWT 001 - Node frontier04975 - RT_GPU_ID 0 - GPU_ID
MPI 000 - OMP 001 - HWT 002 - Node frontier04975 - RT_GPU_ID 0 - GPU_ID
MPI 001 - OMP 000 - HWT 009 - Node frontier04975 - RT_GPU_ID 0 - GPU_ID
MPI 001 - OMP 001 - HWT 010 - Node frontier04975 - RT_GPU_ID 0 - GPU_ID
MPI 002 - OMP 000 - HWT 017 - Node frontier04975 - RT_GPU_ID 0 - GPU_ID
MPI 002 - OMP 001 - HWT 018 - Node frontier04975 - RT_GPU_ID 0 - GPU_ID
MPI 003 - OMP 000 - HWT 025 - Node frontier04975 - RT_GPU_ID 0 - GPU_ID
MPI 003 - OMP 001 - HWT 026 - Node frontier04975 - RT_GPU_ID 0 - GPU_ID
MPI 004 - OMP 000 - HWT 033 - Node frontier04975 - RT_GPU_ID 0 - GPU_ID
MPI 004 - OMP 001 - HWT 034 - Node frontier04975 - RT_GPU_ID 0 - GPU_ID
MPI 005 - OMP 000 - HWT 041 - Node frontier04975 - RT_GPU_ID 0 - GPU_ID
MPI 005 - OMP 001 - HWT 042 - Node frontier04975 - RT_GPU_ID 0 - GPU_ID
MPI 006 - OMP 000 - HWT 049 - Node frontier04975 - RT_GPU_ID 0 - GPU_ID
MPI 006 - OMP 001 - HWT 050 - Node frontier04975 - RT_GPU_ID 0 - GPU_ID
MPI 007 - OMP 000 - HWT 057 - Node frontier04975 - RT_GPU_ID 0 - GPU_ID
MPI 007 - OMP 001 - HWT 058 - Node frontier04975 - RT_GPU_ID 0 - GPU_ID
MPI 008 - OMP 000 - HWT 001 - Node frontier04976 - RT_GPU_ID 0 - GPU_ID
MPI 008 - OMP 001 - HWT 002 - Node frontier04976 - RT_GPU_ID 0 - GPU_ID
MPI 009 - OMP 000 - HWT 009 - Node frontier04976 - RT_GPU_ID 0 - GPU_ID
MPI 009 - OMP 001 - HWT 010 - Node frontier04976 - RT_GPU_ID 0 - GPU_ID
MPI 010 - OMP 000 - HWT 017 - Node frontier04976 - RT_GPU_ID 0 - GPU_ID
MPI 010 - OMP 001 - HWT 018 - Node frontier04976 - RT_GPU_ID 0 - GPU_ID
MPI 011 - OMP 000 - HWT 025 - Node frontier04976 - RT_GPU_ID 0 - GPU_ID
MPI 011 - OMP 001 - HWT 026 - Node frontier04976 - RT_GPU_ID 0 - GPU_ID
MPI 012 - OMP 000 - HWT 033 - Node frontier04976 - RT_GPU_ID 0 - GPU_ID
MPI 012 - OMP 001 - HWT 034 - Node frontier04976 - RT_GPU_ID 0 - GPU_ID
MPI 013 - OMP 000 - HWT 041 - Node frontier04976 - RT_GPU_ID 0 - GPU_ID
MPI 013 - OMP 001 - HWT 042 - Node frontier04976 - RT_GPU_ID 0 - GPU_ID
MPI 014 - OMP 000 - HWT 049 - Node frontier04976 - RT_GPU_ID 0 - GPU_ID
MPI 014 - OMP 001 - HWT 050 - Node frontier04976 - RT_GPU_ID 0 - GPU_ID
MPI 015 - OMP 000 - HWT 057 - Node frontier04976 - RT_GPU_ID 0 - GPU_ID
MPI 015 - OMP 001 - HWT 058 - Node frontier04976 - RT_GPU_ID 0 - GPU_ID
```

Example 4: 8 MPI ranks - each with 2 OpenMP threads and 1 *specific* GPU (single-node)

This example will be very similar to Example 2, but instead of using `--gpu-bind=closest` to map each MPI rank to the closest GPU, `--gpu-bind=map_gpu` will be used to map each MPI rank to a *specific* GPU. The `map_gpu` option takes a comma-separated list of GPU IDs to specify how the MPI ranks are mapped to GPUs, where the form of the comma-separated list is `<gpu_id_for_task_0>, <gpu_id_for_task_1>,...` .

```
$ export OMP_NUM_THREADS=2
$ srun -N1 -n8 -c2 --gpus-per-node=8 --gpu-bind=map_gpu:4,5,2,3,6,7,0,1
```

```
MPI 000 - OMP 000 - HWT 001 - Node frontier08413 - RT_GPU_ID 0 - GPU_ID
MPI 000 - OMP 001 - HWT 002 - Node frontier08413 - RT_GPU_ID 0 - GPU_ID
MPI 001 - OMP 000 - HWT 009 - Node frontier08413 - RT_GPU_ID 0 - GPU_ID
MPI 001 - OMP 001 - HWT 010 - Node frontier08413 - RT_GPU_ID 0 - GPU_ID
MPI 002 - OMP 000 - HWT 017 - Node frontier08413 - RT_GPU_ID 0 - GPU_ID
MPI 002 - OMP 001 - HWT 018 - Node frontier08413 - RT_GPU_ID 0 - GPU_ID
MPI 003 - OMP 000 - HWT 025 - Node frontier08413 - RT_GPU_ID 0 - GPU_ID
MPI 003 - OMP 001 - HWT 026 - Node frontier08413 - RT_GPU_ID 0 - GPU_ID
MPI 004 - OMP 000 - HWT 033 - Node frontier08413 - RT_GPU_ID 0 - GPU_ID
MPI 004 - OMP 001 - HWT 034 - Node frontier08413 - RT_GPU_ID 0 - GPU_ID
MPI 005 - OMP 000 - HWT 041 - Node frontier08413 - RT_GPU_ID 0 - GPU_ID
MPI 005 - OMP 001 - HWT 042 - Node frontier08413 - RT_GPU_ID 0 - GPU_ID
MPI 006 - OMP 000 - HWT 049 - Node frontier08413 - RT_GPU_ID 0 - GPU_ID
MPI 006 - OMP 001 - HWT 050 - Node frontier08413 - RT_GPU_ID 0 - GPU_ID
MPI 007 - OMP 000 - HWT 057 - Node frontier08413 - RT_GPU_ID 0 - GPU_ID
MPI 007 - OMP 001 - HWT 058 - Node frontier08413 - RT_GPU_ID 0 - GPU_ID
```

Here, the output is the same as the results from Example 2. This is because the 8 GPU IDs in the comma-separated list happen to specify the GPUs within the same L3 cache region that the MPI ranks are in. So MPI 000 is mapped to GPU 4, MPI 001 is mapped to GPU 5, etc.

While this level of control over mapping MPI ranks to GPUs might be useful for some applications, it is always important to consider the implication of the mapping. For example, if the order of the GPU IDs in the `map_gpu` option is reversed, the MPI ranks and the GPUs they are mapped to would be in different L3 cache regions, which could potentially lead to poorer performance.

Example 5: 16 MPI ranks - each with 2 OpenMP threads and 1 *specific* GPU (multi-node)

Extending Examples 3 and 4 to run on 2 nodes is also a straightforward exercise by changing the number of nodes to 2 (`-N2`) and the number of MPI ranks to 16 (`-n16`).

```
$ export OMP_NUM_THREADS=2
$ srun -N2 -n16 -c2 --gpus-per-node=8 --gpu-bind=map_gpu:4,5,2,3,6,7,0,1

MPI 000 - OMP 000 - HWT 001 - Node frontier04975 - RT_GPU_ID 0 - GPU_ID
MPI 000 - OMP 001 - HWT 002 - Node frontier04975 - RT_GPU_ID 0 - GPU_ID
MPI 001 - OMP 000 - HWT 009 - Node frontier04975 - RT_GPU_ID 0 - GPU_ID
MPI 001 - OMP 001 - HWT 010 - Node frontier04975 - RT_GPU_ID 0 - GPU_ID
MPI 002 - OMP 000 - HWT 017 - Node frontier04975 - RT_GPU_ID 0 - GPU_ID
MPI 002 - OMP 001 - HWT 018 - Node frontier04975 - RT_GPU_ID 0 - GPU_ID
MPI 003 - OMP 000 - HWT 025 - Node frontier04975 - RT_GPU_ID 0 - GPU_ID
MPI 003 - OMP 001 - HWT 026 - Node frontier04975 - RT_GPU_ID 0 - GPU_ID
MPI 004 - OMP 000 - HWT 033 - Node frontier04975 - RT_GPU_ID 0 - GPU_ID
MPI 004 - OMP 001 - HWT 034 - Node frontier04975 - RT_GPU_ID 0 - GPU_ID
MPI 005 - OMP 000 - HWT 041 - Node frontier04975 - RT_GPU_ID 0 - GPU_ID
MPI 005 - OMP 001 - HWT 042 - Node frontier04975 - RT_GPU_ID 0 - GPU_ID
MPI 006 - OMP 000 - HWT 049 - Node frontier04975 - RT_GPU_ID 0 - GPU_ID
MPI 006 - OMP 001 - HWT 050 - Node frontier04975 - RT_GPU_ID 0 - GPU_ID
MPI 007 - OMP 000 - HWT 057 - Node frontier04975 - RT_GPU_ID 0 - GPU_ID
MPI 007 - OMP 001 - HWT 058 - Node frontier04975 - RT_GPU_ID 0 - GPU_ID
MPI 008 - OMP 000 - HWT 001 - Node frontier04976 - RT_GPU_ID 0 - GPU_ID
MPI 008 - OMP 001 - HWT 002 - Node frontier04976 - RT_GPU_ID 0 - GPU_ID
MPI 009 - OMP 000 - HWT 009 - Node frontier04976 - RT_GPU_ID 0 - GPU_ID
MPI 009 - OMP 001 - HWT 010 - Node frontier04976 - RT_GPU_ID 0 - GPU_ID
MPI 010 - OMP 000 - HWT 017 - Node frontier04976 - RT_GPU_ID 0 - GPU_ID
MPI 010 - OMP 001 - HWT 018 - Node frontier04976 - RT_GPU_ID 0 - GPU_ID
MPI 011 - OMP 000 - HWT 025 - Node frontier04976 - RT_GPU_ID 0 - GPU_ID
MPI 011 - OMP 001 - HWT 026 - Node frontier04976 - RT_GPU_ID 0 - GPU_ID
MPI 012 - OMP 000 - HWT 033 - Node frontier04976 - RT_GPU_ID 0 - GPU_ID
MPI 012 - OMP 001 - HWT 034 - Node frontier04976 - RT_GPU_ID 0 - GPU_ID
MPI 013 - OMP 000 - HWT 041 - Node frontier04976 - RT_GPU_ID 0 - GPU_ID
MPI 013 - OMP 001 - HWT 042 - Node frontier04976 - RT_GPU_ID 0 - GPU_ID
MPI 014 - OMP 000 - HWT 049 - Node frontier04976 - RT_GPU_ID 0 - GPU_ID
MPI 014 - OMP 001 - HWT 050 - Node frontier04976 - RT_GPU_ID 0 - GPU_ID
MPI 015 - OMP 000 - HWT 057 - Node frontier04976 - RT_GPU_ID 0 - GPU_ID
MPI 015 - OMP 001 - HWT 058 - Node frontier04976 - RT_GPU_ID 0 - GPU_ID
```

Mapping multiple MPI ranks to a single GPU

In the following examples, 2 MPI ranks will be mapped to 1 GPU. For the sake of brevity, `OMP_NUM_THREADS` will be set to `1`, so `-c1` will be used unless otherwise specified.

Note

On AMD's MI250X, multi-process service (MPS) is not needed since multiple MPI ranks per GPU is supported natively.

Example 6: 16 MPI ranks - where 2 ranks share a GPU (round-robin, single-node)

This example launches 16 MPI ranks (`-n16`), each with 1 physical CPU core (`-c1`) to launch 1 OpenMP thread (`OMP_NUM_THREADS=1`) on. The MPI ranks will be assigned to GPUs in a round-robin fashion so that each of the 8 GPUs on the node are shared by 2 MPI ranks. To accomplish this GPU mapping, a new `srun` options will be used:

- `--ntasks-per-gpu` specifies the number of MPI ranks that will share access to a GPU.

```
$ export OMP_NUM_THREADS=1
$ srun -N1 -n16 -c1 --ntasks-per-gpu=2 --gpu-bind=closest ./hello_jobste

MPI 000 - OMP 000 - HWT 001 - Node frontier08413 - RT_GPU_ID 0 - GPU_ID
MPI 001 - OMP 000 - HWT 009 - Node frontier08413 - RT_GPU_ID 0 - GPU_ID
MPI 002 - OMP 000 - HWT 017 - Node frontier08413 - RT_GPU_ID 0 - GPU_ID
MPI 003 - OMP 000 - HWT 025 - Node frontier08413 - RT_GPU_ID 0 - GPU_ID
MPI 004 - OMP 000 - HWT 033 - Node frontier08413 - RT_GPU_ID 0 - GPU_ID
MPI 005 - OMP 000 - HWT 041 - Node frontier08413 - RT_GPU_ID 0 - GPU_ID
MPI 006 - OMP 000 - HWT 049 - Node frontier08413 - RT_GPU_ID 0 - GPU_ID
MPI 007 - OMP 000 - HWT 057 - Node frontier08413 - RT_GPU_ID 0 - GPU_ID
MPI 008 - OMP 000 - HWT 002 - Node frontier08413 - RT_GPU_ID 0 - GPU_ID
MPI 009 - OMP 000 - HWT 010 - Node frontier08413 - RT_GPU_ID 0 - GPU_ID
MPI 010 - OMP 000 - HWT 018 - Node frontier08413 - RT_GPU_ID 0 - GPU_ID
MPI 011 - OMP 000 - HWT 026 - Node frontier08413 - RT_GPU_ID 0 - GPU_ID
MPI 012 - OMP 000 - HWT 034 - Node frontier08413 - RT_GPU_ID 0 - GPU_ID
MPI 013 - OMP 000 - HWT 042 - Node frontier08413 - RT_GPU_ID 0 - GPU_ID
MPI 014 - OMP 000 - HWT 050 - Node frontier08413 - RT_GPU_ID 0 - GPU_ID
MPI 015 - OMP 000 - HWT 058 - Node frontier08413 - RT_GPU_ID 0 - GPU_ID
```

The output shows the round-robin (`cyclic`) distribution of MPI ranks to GPUs. In fact, it is a round-robin distribution of MPI ranks *to L3 cache regions* (the default distribution). The GPU mapping is a consequence of where the MPI ranks are distributed; `--gpu-bind=closest` simply maps the GPU in an L3 cache region to the MPI ranks in the same L3 region.

Example 7: 32 MPI ranks - where 2 ranks share a GPU (round-robin, multi-node)

This example is an extension of Example 6 to run on 2 nodes.

```
$ export OMP_NUM_THREADS=1
$ srun -N2 -n32 -c1 --ntasks-per-gpu=2 --gpu-bind=closest ./hello_jobste

MPI 000 - OMP 000 - HWT 001 - Node frontier04975 - RT_GPU_ID 0 - GPU_ID
MPI 001 - OMP 000 - HWT 009 - Node frontier04975 - RT_GPU_ID 0 - GPU_ID
MPI 002 - OMP 000 - HWT 017 - Node frontier04975 - RT_GPU_ID 0 - GPU_ID
MPI 003 - OMP 000 - HWT 025 - Node frontier04975 - RT_GPU_ID 0 - GPU_ID
MPI 004 - OMP 000 - HWT 033 - Node frontier04975 - RT_GPU_ID 0 - GPU_ID
MPI 005 - OMP 000 - HWT 041 - Node frontier04975 - RT_GPU_ID 0 - GPU_ID
MPI 006 - OMP 000 - HWT 049 - Node frontier04975 - RT_GPU_ID 0 - GPU_ID
MPI 007 - OMP 000 - HWT 057 - Node frontier04975 - RT_GPU_ID 0 - GPU_ID
MPI 008 - OMP 000 - HWT 002 - Node frontier04975 - RT_GPU_ID 0 - GPU_ID
MPI 009 - OMP 000 - HWT 010 - Node frontier04975 - RT_GPU_ID 0 - GPU_ID
MPI 010 - OMP 000 - HWT 018 - Node frontier04975 - RT_GPU_ID 0 - GPU_ID
MPI 011 - OMP 000 - HWT 026 - Node frontier04975 - RT_GPU_ID 0 - GPU_ID
MPI 012 - OMP 000 - HWT 034 - Node frontier04975 - RT_GPU_ID 0 - GPU_ID
MPI 013 - OMP 000 - HWT 042 - Node frontier04975 - RT_GPU_ID 0 - GPU_ID
MPI 014 - OMP 000 - HWT 050 - Node frontier04975 - RT_GPU_ID 0 - GPU_ID
MPI 015 - OMP 000 - HWT 058 - Node frontier04975 - RT_GPU_ID 0 - GPU_ID
MPI 016 - OMP 000 - HWT 001 - Node frontier04976 - RT_GPU_ID 0 - GPU_ID
MPI 017 - OMP 000 - HWT 009 - Node frontier04976 - RT_GPU_ID 0 - GPU_ID
MPI 018 - OMP 000 - HWT 017 - Node frontier04976 - RT_GPU_ID 0 - GPU_ID
MPI 019 - OMP 000 - HWT 025 - Node frontier04976 - RT_GPU_ID 0 - GPU_ID
MPI 020 - OMP 000 - HWT 033 - Node frontier04976 - RT_GPU_ID 0 - GPU_ID
MPI 021 - OMP 000 - HWT 041 - Node frontier04976 - RT_GPU_ID 0 - GPU_ID
MPI 022 - OMP 000 - HWT 049 - Node frontier04976 - RT_GPU_ID 0 - GPU_ID
MPI 023 - OMP 000 - HWT 057 - Node frontier04976 - RT_GPU_ID 0 - GPU_ID
MPI 024 - OMP 000 - HWT 002 - Node frontier04976 - RT_GPU_ID 0 - GPU_ID
MPI 025 - OMP 000 - HWT 010 - Node frontier04976 - RT_GPU_ID 0 - GPU_ID
MPI 026 - OMP 000 - HWT 018 - Node frontier04976 - RT_GPU_ID 0 - GPU_ID
MPI 027 - OMP 000 - HWT 026 - Node frontier04976 - RT_GPU_ID 0 - GPU_ID
MPI 028 - OMP 000 - HWT 034 - Node frontier04976 - RT_GPU_ID 0 - GPU_ID
MPI 029 - OMP 000 - HWT 042 - Node frontier04976 - RT_GPU_ID 0 - GPU_ID
MPI 030 - OMP 000 - HWT 050 - Node frontier04976 - RT_GPU_ID 0 - GPU_ID
MPI 031 - OMP 000 - HWT 058 - Node frontier04976 - RT_GPU_ID 0 - GPU_ID
```

Example 8: 16 MPI ranks - where 2 ranks share a GPU (packed, single-node)

Warning

This example assumes the use of a core specialization of `-S 0`. Because Frontier's default core specialization (`-S 8`) reserves the first core in each L3 region, the "packed" mode can be problematic because the 7 cores available in each L3 region won't necessarily divide evenly. This can lead to tasks potentially spanning multiple L3 regions with its assigned cores, which creates problems when Slurm tries to assign GPUs to a given task.

This example launches 16 MPI ranks (`-n16`), each with 4 physical CPU cores (`-c4`) to launch 1 OpenMP thread (`OMP_NUM_THREADS=1`) on. The MPI ranks will be assigned to GPUs in a packed fashion so that each of the 8 GPUs on the node are shared by 2 MPI ranks. Similar to Example 6, `-ntasks-per-gpu=2` will be used, but a new `srun` flag will be used to change the default round-robin (`cyclic`) distribution of MPI ranks across NUMA domains:

- `--distribution=<value>[:<value>][:<value>]` specifies the distribution of MPI ranks across compute nodes, sockets (L3 cache regions on Frontier), and cores, respectively. The default values are `block:cyclic:cyclic`, which is where the `cyclic` assignment comes from in the previous examples.

Note

In the job step for this example, `--distribution=*:block` is used, where `*` represents the default value of `block` for the distribution of MPI ranks across compute nodes and the distribution of MPI ranks across L3 cache regions has been changed to `block` from its default value of `cyclic` .

Note

Because the distribution across L3 cache regions has been changed to a "packed" (`block`) configuration, caution must be taken to ensure MPI ranks end up in the L3 cache regions where the GPUs they intend to be mapped to are located. To accomplish this, the number of physical CPU cores assigned to an MPI rank was increased - in this case to 4. Doing so ensures that only 2 MPI ranks can fit into a single L3 cache region. If the value of `-c` was left at `1` , all 8 MPI ranks would be "packed" into the first L3 region, where the "closest" GPU would be GPU 4 - the only GPU in that L3 region.

Notice that this is not a workaround like in Example 7, but a requirement due to the `block` distribution of MPI ranks across NUMA domains.

```
$ export OMP_NUM_THREADS=1
$ srun -N1 -n16 -c4 --ntasks-per-gpu=2 --gpu-bind=closest --distribution=

MPI 000 - OMP 000 - HWT 000 - Node frontier002 - RT_GPU_ID 0 - GPU_ID 4 -
MPI 001 - OMP 000 - HWT 004 - Node frontier002 - RT_GPU_ID 0 - GPU_ID 4 -
MPI 002 - OMP 000 - HWT 008 - Node frontier002 - RT_GPU_ID 0 - GPU_ID 5 -
MPI 003 - OMP 000 - HWT 012 - Node frontier002 - RT_GPU_ID 0 - GPU_ID 5 -
MPI 004 - OMP 000 - HWT 016 - Node frontier002 - RT_GPU_ID 0 - GPU_ID 2 -
MPI 005 - OMP 000 - HWT 020 - Node frontier002 - RT_GPU_ID 0 - GPU_ID 2 -
MPI 006 - OMP 000 - HWT 024 - Node frontier002 - RT_GPU_ID 0 - GPU_ID 3 -
MPI 007 - OMP 000 - HWT 028 - Node frontier002 - RT_GPU_ID 0 - GPU_ID 3 -
MPI 008 - OMP 000 - HWT 032 - Node frontier002 - RT_GPU_ID 0 - GPU_ID 6 -
MPI 009 - OMP 000 - HWT 036 - Node frontier002 - RT_GPU_ID 0 - GPU_ID 6 -
MPI 010 - OMP 000 - HWT 040 - Node frontier002 - RT_GPU_ID 0 - GPU_ID 7 -
MPI 011 - OMP 000 - HWT 044 - Node frontier002 - RT_GPU_ID 0 - GPU_ID 7 -
MPI 012 - OMP 000 - HWT 048 - Node frontier002 - RT_GPU_ID 0 - GPU_ID 0 -
MPI 013 - OMP 000 - HWT 052 - Node frontier002 - RT_GPU_ID 0 - GPU_ID 0 -
MPI 014 - OMP 000 - HWT 056 - Node frontier002 - RT_GPU_ID 0 - GPU_ID 1 -
MPI 015 - OMP 000 - HWT 060 - Node frontier002 - RT_GPU_ID 0 - GPU_ID 1 -
```

The overall effect of using `--distribution=*:block` and increasing the number of physical CPU cores available to each MPI rank is to place the first two MPI ranks in the first L3 cache region with GPU 4, the next two MPI ranks in the second L3 cache region with GPU 5, and so on.

Example 9: 32 MPI ranks - where 2 ranks share a GPU (packed, multi-node)

Warning

This example assumes the use of a core specialization of `-S 0`. Because Frontier's default core specialization (`-S 8`) reserves the first core in each L3 region, the "packed" mode can be problematic because the 7 cores available in each L3 region won't necessarily divide evenly. This can lead to tasks potentially spanning multiple L3 regions with its assigned cores, which creates problems when Slurm tries to assign GPUs to a given task.

This example is an extension of Example 8 to use 2 compute nodes. With the appropriate changes put in place in Example 8, it is a straightforward exercise to change to using 2 nodes (`-N2`) and 32 MPI ranks (`-n32`).

```
$ export OMP_NUM_THREADS=1
$ srun -N2 -n32 -c4 --ntasks-per-gpu=2 --gpu-bind=closest --distribution=

MPI 000 - OMP 000 - HWT 000 - Node frontier002 - RT_GPU_ID 0 - GPU_ID 4 -
MPI 001 - OMP 000 - HWT 004 - Node frontier002 - RT_GPU_ID 0 - GPU_ID 4 -
MPI 002 - OMP 000 - HWT 010 - Node frontier002 - RT_GPU_ID 0 - GPU_ID 5 -
MPI 003 - OMP 000 - HWT 012 - Node frontier002 - RT_GPU_ID 0 - GPU_ID 5 -
MPI 004 - OMP 000 - HWT 016 - Node frontier002 - RT_GPU_ID 0 - GPU_ID 2 -
MPI 005 - OMP 000 - HWT 021 - Node frontier002 - RT_GPU_ID 0 - GPU_ID 2 -
MPI 006 - OMP 000 - HWT 024 - Node frontier002 - RT_GPU_ID 0 - GPU_ID 3 -
MPI 007 - OMP 000 - HWT 028 - Node frontier002 - RT_GPU_ID 0 - GPU_ID 3 -
MPI 008 - OMP 000 - HWT 032 - Node frontier002 - RT_GPU_ID 0 - GPU_ID 6 -
MPI 009 - OMP 000 - HWT 037 - Node frontier002 - RT_GPU_ID 0 - GPU_ID 6 -
MPI 010 - OMP 000 - HWT 041 - Node frontier002 - RT_GPU_ID 0 - GPU_ID 7 -
MPI 011 - OMP 000 - HWT 044 - Node frontier002 - RT_GPU_ID 0 - GPU_ID 7 -
MPI 012 - OMP 000 - HWT 049 - Node frontier002 - RT_GPU_ID 0 - GPU_ID 0 -
MPI 013 - OMP 000 - HWT 052 - Node frontier002 - RT_GPU_ID 0 - GPU_ID 0 -
MPI 014 - OMP 000 - HWT 056 - Node frontier002 - RT_GPU_ID 0 - GPU_ID 1 -
MPI 015 - OMP 000 - HWT 060 - Node frontier002 - RT_GPU_ID 0 - GPU_ID 1 -
MPI 016 - OMP 000 - HWT 000 - Node frontier004 - RT_GPU_ID 0 - GPU_ID 4 -
MPI 017 - OMP 000 - HWT 004 - Node frontier004 - RT_GPU_ID 0 - GPU_ID 4 -
MPI 018 - OMP 000 - HWT 008 - Node frontier004 - RT_GPU_ID 0 - GPU_ID 5 -
MPI 019 - OMP 000 - HWT 013 - Node frontier004 - RT_GPU_ID 0 - GPU_ID 5 -
MPI 020 - OMP 000 - HWT 016 - Node frontier004 - RT_GPU_ID 0 - GPU_ID 2 -
MPI 021 - OMP 000 - HWT 020 - Node frontier004 - RT_GPU_ID 0 - GPU_ID 2 -
MPI 022 - OMP 000 - HWT 024 - Node frontier004 - RT_GPU_ID 0 - GPU_ID 3 -
MPI 023 - OMP 000 - HWT 028 - Node frontier004 - RT_GPU_ID 0 - GPU_ID 3 -
MPI 024 - OMP 000 - HWT 034 - Node frontier004 - RT_GPU_ID 0 - GPU_ID 6 -
MPI 025 - OMP 000 - HWT 036 - Node frontier004 - RT_GPU_ID 0 - GPU_ID 6 -
MPI 026 - OMP 000 - HWT 040 - Node frontier004 - RT_GPU_ID 0 - GPU_ID 7 -
MPI 027 - OMP 000 - HWT 044 - Node frontier004 - RT_GPU_ID 0 - GPU_ID 7 -
MPI 028 - OMP 000 - HWT 048 - Node frontier004 - RT_GPU_ID 0 - GPU_ID 0 -
MPI 029 - OMP 000 - HWT 052 - Node frontier004 - RT_GPU_ID 0 - GPU_ID 0
MPI 030 - OMP 000 - HWT 056 - Node frontier004 - RT_GPU_ID 0 - GPU_ID 1
MPI 031 - OMP 000 - HWT 060 - Node frontier004 - RT_GPU_ID 0 - GPU_ID 1
```

Example 10: 56 MPI ranks - where 7 ranks share a GPU (packed, single-node)

An alternative solution to Example 8 and 9's `-S 8` issue is to use `-c 1` instead. There is no problem when running with 1 core per MPI rank (i.e., 7 ranks per GPU) because the task can't span multiple L3s.

```
$ export OMP_NUM_THREADS=1
$ srun -N1 -n56 -c1 --ntasks-per-gpu=7 --gpu-bind=closest --distribution

MPI 000 - OMP 000 - HWT 001 - Node frontier08413 - RT_GPU_ID 0 - GPU_ID
MPI 001 - OMP 000 - HWT 002 - Node frontier08413 - RT_GPU_ID 0 - GPU_ID
MPI 002 - OMP 000 - HWT 003 - Node frontier08413 - RT_GPU_ID 0 - GPU_ID
MPI 003 - OMP 000 - HWT 004 - Node frontier08413 - RT_GPU_ID 0 - GPU_ID
MPI 004 - OMP 000 - HWT 005 - Node frontier08413 - RT_GPU_ID 0 - GPU_ID
MPI 005 - OMP 000 - HWT 006 - Node frontier08413 - RT_GPU_ID 0 - GPU_ID
MPI 006 - OMP 000 - HWT 007 - Node frontier08413 - RT_GPU_ID 0 - GPU_ID
MPI 007 - OMP 000 - HWT 009 - Node frontier08413 - RT_GPU_ID 0 - GPU_ID
MPI 008 - OMP 000 - HWT 010 - Node frontier08413 - RT_GPU_ID 0 - GPU_ID
MPI 009 - OMP 000 - HWT 011 - Node frontier08413 - RT_GPU_ID 0 - GPU_ID
MPI 010 - OMP 000 - HWT 012 - Node frontier08413 - RT_GPU_ID 0 - GPU_ID
MPI 011 - OMP 000 - HWT 013 - Node frontier08413 - RT_GPU_ID 0 - GPU_ID
MPI 012 - OMP 000 - HWT 014 - Node frontier08413 - RT_GPU_ID 0 - GPU_ID
MPI 013 - OMP 000 - HWT 015 - Node frontier08413 - RT_GPU_ID 0 - GPU_ID
MPI 014 - OMP 000 - HWT 017 - Node frontier08413 - RT_GPU_ID 0 - GPU_ID
MPI 015 - OMP 000 - HWT 018 - Node frontier08413 - RT_GPU_ID 0 - GPU_ID
MPI 016 - OMP 000 - HWT 019 - Node frontier08413 - RT_GPU_ID 0 - GPU_ID
MPI 017 - OMP 000 - HWT 020 - Node frontier08413 - RT_GPU_ID 0 - GPU_ID
MPI 018 - OMP 000 - HWT 021 - Node frontier08413 - RT_GPU_ID 0 - GPU_ID
MPI 019 - OMP 000 - HWT 022 - Node frontier08413 - RT_GPU_ID 0 - GPU_ID
MPI 020 - OMP 000 - HWT 023 - Node frontier08413 - RT_GPU_ID 0 - GPU_ID
MPI 021 - OMP 000 - HWT 025 - Node frontier08413 - RT_GPU_ID 0 - GPU_ID
MPI 022 - OMP 000 - HWT 026 - Node frontier08413 - RT_GPU_ID 0 - GPU_ID
MPI 023 - OMP 000 - HWT 027 - Node frontier08413 - RT_GPU_ID 0 - GPU_ID
MPI 024 - OMP 000 - HWT 028 - Node frontier08413 - RT_GPU_ID 0 - GPU_ID
MPI 025 - OMP 000 - HWT 029 - Node frontier08413 - RT_GPU_ID 0 - GPU_ID
MPI 026 - OMP 000 - HWT 030 - Node frontier08413 - RT_GPU_ID 0 - GPU_ID
MPI 027 - OMP 000 - HWT 031 - Node frontier08413 - RT_GPU_ID 0 - GPU_ID
MPI 028 - OMP 000 - HWT 033 - Node frontier08413 - RT_GPU_ID 0 - GPU_ID
MPI 029 - OMP 000 - HWT 034 - Node frontier08413 - RT_GPU_ID 0 - GPU_ID
MPI 030 - OMP 000 - HWT 035 - Node frontier08413 - RT_GPU_ID 0 - GPU_ID
MPI 031 - OMP 000 - HWT 036 - Node frontier08413 - RT_GPU_ID 0 - GPU_ID
MPI 032 - OMP 000 - HWT 037 - Node frontier08413 - RT_GPU_ID 0 - GPU_ID
MPI 033 - OMP 000 - HWT 038 - Node frontier08413 - RT_GPU_ID 0 - GPU_ID
MPI 034 - OMP 000 - HWT 039 - Node frontier08413 - RT_GPU_ID 0 - GPU_ID
MPI 035 - OMP 000 - HWT 041 - Node frontier08413 - RT_GPU_ID 0 - GPU_ID
MPI 036 - OMP 000 - HWT 042 - Node frontier08413 - RT_GPU_ID 0 - GPU_ID
MPI 037 - OMP 000 - HWT 043 - Node frontier08413 - RT_GPU_ID 0 - GPU_ID
MPI 038 - OMP 000 - HWT 044 - Node frontier08413 - RT_GPU_ID 0 - GPU_ID
MPI 039 - OMP 000 - HWT 045 - Node frontier08413 - RT_GPU_ID 0 - GPU_ID
MPI 040 - OMP 000 - HWT 046 - Node frontier08413 - RT_GPU_ID 0 - GPU_ID
MPI 041 - OMP 000 - HWT 047 - Node frontier08413 - RT_GPU_ID 0 - GPU_ID
MPI 042 - OMP 000 - HWT 049 - Node frontier08413 - RT_GPU_ID 0 - GPU_ID
MPI 043 - OMP 000 - HWT 050 - Node frontier08413 - RT_GPU_ID 0 - GPU_ID
MPI 044 - OMP 000 - HWT 051 - Node frontier08413 - RT_GPU_ID 0 - GPU_ID
MPI 045 - OMP 000 - HWT 052 - Node frontier08413 - RT_GPU_ID 0 - GPU_ID
MPI 046 - OMP 000 - HWT 053 - Node frontier08413 - RT_GPU_ID 0 - GPU_ID
MPI 047 - OMP 000 - HWT 054 - Node frontier08413 - RT_GPU_ID 0 - GPU_ID
MPI 048 - OMP 000 - HWT 055 - Node frontier08413 - RT_GPU_ID 0 - GPU_ID
MPI 049 - OMP 000 - HWT 057 - Node frontier08413 - RT_GPU_ID 0 - GPU_ID
MPI 050 - OMP 000 - HWT 058 - Node frontier08413 - RT_GPU_ID 0 - GPU_ID
MPI 051 - OMP 000 - HWT 059 - Node frontier08413 - RT_GPU_ID 0 - GPU_ID
MPI 052 - OMP 000 - HWT 060 - Node frontier08413 - RT_GPU_ID 0 - GPU_ID
MPI 053 - OMP 000 - HWT 061 - Node frontier08413 - RT_GPU_ID 0 - GPU_ID
MPI 054 - OMP 000 - HWT 062 - Node frontier08413 - RT_GPU_ID 0 - GPU_ID
MPI 055 - OMP 000 - HWT 063 - Node frontier08413 - RT_GPU_ID 0 - GPU_ID
```

Example 11: 4 independent and simultaneous job steps in a single allocation

This example shows how to run multiple job steps simultaneously in a single allocation. The example below demonstrates running 4 independent, single rank MPI executions on a single node, however the example could be extrapolated to more complex invocations using the above examples.

Submission script:

```
#!/bin/bash
#SBATCH -A <projid>
#SBATCH -N 1
#SBATCH -t 10

srun -N1 -c1 --gpus-per-task=1 --exact ./hello_jobstep &
srun -N1 -c1 --gpus-per-task=1 --exact ./hello_jobstep &
srun -N1 -c1 --gpus-per-task=1 --exact ./hello_jobstep &
srun -N1 -c1 --gpus-per-task=1 --exact ./hello_jobstep &
wait
```

Output:

```
MPI 000 - OMP 000 - HWT 004 - Node frontier25 - RT_GPU_ID 0 - GPU_ID 3 -
MPI 000 - OMP 000 - HWT 002 - Node frontier25 - RT_GPU_ID 0 - GPU_ID 1 -
MPI 000 - OMP 000 - HWT 003 - Node frontier25 - RT_GPU_ID 0 - GPU_ID 2 -
MPI 000 - OMP 000 - HWT 001 - Node frontier25 - RT_GPU_ID 0 - GPU_ID 0 -
```

Note

The `--exact` parameter is important to avoid the error message `srun: Job <job id> step creation temporarily disabled, retrying (Requested nodes are busy)`. The `wait` command is also critical, or your job script and allocation will immediately end after launching your jobs in the background.

Note

This may result in a sub-optimal alignment of CPU and GPU on the node, as shown in the example output. Unfortunately, at the moment there is not a workaround for this, however improvements are possible in future SLURM updates.

Multiple GPUs per MPI rank

As mentioned previously, all GPUs are accessible by all MPI ranks by default, so it is possible to *programatically* map any combination of GPUs to MPI ranks. It should be noted however that Cray MPICH does not support GPU-aware MPI for multiple GPUs per rank, so this binding is not suggested.

Tips for Launching at Scale

SBCAST your executable and libraries

Slurm contains a utility called `sbcast`. This program takes a file and broadcasts it to each node's node-local storage (ie, `/tmp`, NVMe). This is useful for sharing large input files, binaries and shared libraries, while reducing the overhead on shared file systems and overhead at startup. This is highly recommended at scale if you have multiple shared libraries on Lustre/NFS file systems.

SBCASTing a single file

Here is a simple example of a file `sbcast` from a user's scratch space on Lustre to each node's NVMe drive:

```
#!/bin/bash
#SBATCH -A <projid>
```

```
#SBATCH -J sbcast_to_nvme
#SBATCH -o %x-%j.out
#SBATCH -t 00:05:00
#SBATCH -p batch
#SBATCH -N 2
#SBATCH -C nvme

date

# Change directory to user scratch space (Orion)
cd /lustre/orion/<projid>/scratch/<userid>

echo "This is an example file" > test.txt
echo
echo "*****ORIGINAL FILE*****"
cat test.txt
echo "***********************"

# SBCAST file from Orion to NVMe -- NOTE: ``-C nvme`` is required to use
sbcast -pf test.txt /mnt/bb/$USER/test.txt
if [ ! "$?" == "0" ]; then
    # CHECK EXIT CODE. When SBCAST fails, it may leave partial files on
    # your application may pick up partially complete shared library fil
    echo "SBCAST failed!"
    exit 1
fi

echo
echo "*****DISPLAYING FILES ON EACH NODE IN THE ALLOCATION*****"
# Check to see if file exists
srun -N ${SLURM_NNODES} -n ${SLURM_NNODES} --ntasks-per-node=1 bash -c "
echo "*********************************************************"

echo
# Showing the file on the current node -- this will be the same on all o
echo "*****SBCAST FILE ON CURRENT NODE******"
cat /mnt/bb/$USER/test.txt
echo "**************************************"
```

and here is the output from that script:

```
Fri 03 Mar 2023 03:43:30 PM EST

*****ORIGINAL FILE*****
This is an example file
***********************
```

```
*****DISPLAYING FILES ON EACH NODE IN THE ALLOCATION*****
frontier00001: -rw-r--r-- 1 hagertnl hagertnl 24 Mar  3 15:43 /mnt/bb/hag
frontier00002: -rw-r--r-- 1 hagertnl hagertnl 24 Mar  3 15:43 /mnt/bb/hag
*********************************************************

*****SBCAST FILE ON CURRENT NODE******
This is an example file
**************************************
```

SBCASTing a binary with libraries stored on shared file systems

`sbcast` also handles binaries and their libraries:

```
#!/bin/bash
#SBATCH -A <projid>
#SBATCH -J sbcast_binary_to_nvme
#SBATCH -o %x-%j.out
#SBATCH -t 00:05:00
#SBATCH -p batch
#SBATCH -N 2
#SBATCH -C nvme

date

# Change directory to user scratch space (Orion)
cd /lustre/orion/<projid>/scratch/<userid>

# For this example, I use a HIP-enabled LAMMPS binary, with dependencies
exe="lmp"

echo "*****ldd ./${exe}*****"
ldd ./${exe}
echo "*************************"

# SBCAST executable from Orion to NVMe -- NOTE: ``-C nvme`` is needed in
# NOTE: dlopen'd files will NOT be picked up by sbcast
# SBCAST automatically excludes several directories: /lib,/usr/lib,/lib6
#   - These directories are node-local and are very fast to read from, s
#   - see ``$ scontrol show config | grep BcastExclude`` for current lis
#   - OLCF-provided libraries in ``/sw`` are not on the exclusion list.
#   - To override, add ``--exclude=NONE`` to arguments
sbcast --send-libs -pf ${exe} /mnt/bb/$USER/${exe}
if [ ! "$?" == "0" ]; then
    # CHECK EXIT CODE. When SBCAST fails, it may leave partial files on
    # your application may pick up partially complete shared library fil
    echo "SBCAST failed!"
    exit 1
```

```
fi

# Check to see if file exists
echo "*****ls -lh /mnt/bb/$USER*****"
ls -lh /mnt/bb/$USER/
echo "*****ls -lh /mnt/bb/$USER/${exe}_libs*****"
ls -lh /mnt/bb/$USER/${exe}_libs

# SBCAST sends all libraries detected by `ld` (minus any excluded), and s
# Any libraries opened by `dlopen` are NOT sent, since they are not known

# At minimum: prepend the node-local path to LD_LIBRARY_PATH to pick up t
# It is also recommended that you **remove** any paths that you don't nee
# Failure to remove may result in unnecessary calls to stat shared file s
export LD_LIBRARY_PATH="/mnt/bb/$USER/${exe}_libs:${LD_LIBRARY_PATH}"

# If you SBCAST **all** your libraries (ie, `--exclude=NONE`), you may us
#export LD_LIBRARY_PATH="/mnt/bb/$USER/${exe}_libs:$(pkg-config --variab
# Use with caution -- certain libraries may use ``dlopen`` at runtime, an
# If you use this option, we recommend you contact OLCF Help Desk for the

# You may notice that some libraries are still linked from /sw/frontier,
# This is because the Spack-build modules use RPATH to find their depende
echo "*****ldd /mnt/bb/$USER/${exe}*****"
ldd /mnt/bb/$USER/${exe}
echo "*************************************"
```

and here is the output from that script:

```
Tue 28 Mar 2023 05:01:41 PM EDT
*****ldd ./lmp*****
    linux-vdso.so.1 (0x00007fffeda02000)
    libgcc_s.so.1 => /lib64/libgcc_s.so.1 (0x00007fffed5bb000)
    libpthread.so.0 => /lib64/libpthread.so.0 (0x00007fffed398000)
    libm.so.6 => /lib64/libm.so.6 (0x00007fffed04d000)
    librt.so.1 => /lib64/librt.so.1 (0x00007fffece44000)
    libamdhip64.so.4 => /opt/rocm-4.5.2/lib/libamdhip64.so.4 (0x00007fff
    libmpi_cray.so.12 => /opt/cray/pe/lib64/libmpi_cray.so.12 (0x00007ff
    libmpi_gtl_hsa.so.0 => /opt/cray/pe/lib64/libmpi_gtl_hsa.so.0 (0x000
    libhsa-runtime64.so.1 => /opt/rocm-5.3.0/lib/libhsa-runtime64.so.1 (
    libhwloc.so.15 => /sw/frontier/spack-envs/base/opt/linux-sles15-x86_
    libdl.so.2 => /lib64/libdl.so.2 (0x00007fffe8b7e000)
    libhipfft.so => /opt/rocm-5.3.0/lib/libhipfft.so (0x00007fffed9d2000
    libstdc++.so.6 => /usr/lib64/libstdc++.so.6 (0x00007fffe875b000)
    libc.so.6 => /lib64/libc.so.6 (0x00007fffe8366000)
    /lib64/ld-linux-x86-64.so.2 (0x00007fffed7da000)
    libamd_comgr.so.2 => /opt/rocm-5.3.0/lib/libamd_comgr.so.2 (0x00007ff
    libnuma.so.1 => /usr/lib64/libnuma.so.1 (0x00007fffe04d4000)
    libfabric.so.1 => /opt/cray/libfabric/1.15.2.0/lib64/libfabric.so.1
    libatomic.so.1 => /usr/lib64/libatomic.so.1 (0x00007fffdffd9000)
    libpmi.so.0 => /opt/cray/pe/lib64/libpmi.so.0 (0x00007fffdfdd7000)
    libpmi2.so.0 => /opt/cray/pe/lib64/libpmi2.so.0 (0x00007fffdfb9e000)
    libquadmath.so.0 => /usr/lib64/libquadmath.so.0 (0x00007fffdf959000)
    libmodules.so.1 => /opt/cray/pe/lib64/cce/libmodules.so.1 (0x00007ff
    libfi.so.1 => /opt/cray/pe/lib64/cce/libfi.so.1 (0x00007fffdf3b4000)
    libcraymath.so.1 => /opt/cray/pe/lib64/cce/libcraymath.so.1 (0x00007
    libf.so.1 => /opt/cray/pe/lib64/cce/libf.so.1 (0x00007fffed83b000)
    libu.so.1 => /opt/cray/pe/lib64/cce/libu.so.1 (0x00007fffdf2ab000)
    libcsup.so.1 => /opt/cray/pe/lib64/cce/libcsup.so.1 (0x00007fffed832
    libamdhip64.so.5 => /opt/rocm-5.3.0/lib/libamdhip64.so.5 (0x00007fff
    libelf.so.1 => /usr/lib64/libelf.so.1 (0x00007fffdd871000)
    libdrm.so.2 => /usr/lib64/libdrm.so.2 (0x00007fffdd65d000)
    libdrm_amdgpu.so.1 => /usr/lib64/libdrm_amdgpu.so.1 (0x00007fffdd453
    libpciaccess.so.0 => /sw/frontier/spack-envs/base/opt/linux-sles15-x
    libxml2.so.2 => /sw/frontier/spack-envs/base/opt/linux-sles15-x86_64
    librocfft.so.0 => /opt/rocm-5.3.0/lib/librocfft.so.0 (0x00007fffdca1
    libz.so.1 => /lib64/libz.so.1 (0x00007fffdc803000)
    libtinfo.so.6 => /lib64/libtinfo.so.6 (0x00007fffdc5d5000)
    libcxi.so.1 => /usr/lib64/libcxi.so.1 (0x00007fffdc3b0000)
    libcurl.so.4 => /usr/lib64/libcurl.so.4 (0x00007fffdc311000)
    libjson-c.so.3 => /usr/lib64/libjson-c.so.3 (0x00007fffdc101000)
    libpals.so.0 => /opt/cray/pe/lib64/libpals.so.0 (0x00007fffdbefc000)
    libgfortran.so.5 => /opt/cray/pe/gcc-libs/libgfortran.so.5 (0x00007f
    liblzma.so.5 => /sw/frontier/spack-envs/base/opt/linux-sles15-x86_64
    libiconv.so.2 => /sw/frontier/spack-envs/base/opt/linux-sles15-x86_6
    librocfft-device-0.so.0 => /opt/rocm-5.3.0/lib/librocfft-device-0.so
    librocfft-device-1.so.0 => /opt/rocm-5.3.0/lib/librocfft-device-1.so
    librocfft-device-2.so.0 => /opt/rocm-5.3.0/lib/librocfft-device-2.so
    librocfft-device-3.so.0 => /opt/rocm-5.3.0/lib/librocfft-device-3.so
    libnghttp2.so.14 => /usr/lib64/libnghttp2.so.14 (0x00007ffef7cc6000)
    libidn2.so.0 => /usr/lib64/libidn2.so.0 (0x00007ffef7aa9000)
    libssh.so.4 => /usr/lib64/libssh.so.4 (0x00007ffef783b000)
    libpsl.so.5 => /usr/lib64/libpsl.so.5 (0x00007ffef7629000)
    libssl.so.1.1 => /usr/lib64/libssl.so.1.1 (0x00007ffef758a000)
    libcrypto.so.1.1 => /usr/lib64/libcrypto.so.1.1 (0x00007ffef724a000)
    libgssapi_krb5.so.2 => /usr/lib64/libgssapi_krb5.so.2 (0x00007ffef6f
    libldap_r-2.4.so.2 => /usr/lib64/libldap_r-2.4.so.2 (0x00007ffef6da4
    liblber-2.4.so.2 => /usr/lib64/liblber-2.4.so.2 (0x00007ffef6b95000)
    libzstd.so.1 => /usr/lib64/libzstd.so.1 (0x00007ffef6865000)
    libbrotlidec.so.1 => /usr/lib64/libbrotlidec.so.1 (0x00007ffef665900
    libunistring.so.2 => /usr/lib64/libunistring.so.2 (0x00007ffef62d600
    libjitterentropy.so.3 => /usr/lib64/libjitterentropy.so.3 (0x00007ff
    libkrb5.so.3 => /usr/lib64/libkrb5.so.3 (0x00007ffef5df6000)
    libk5crypto.so.3 => /usr/lib64/libk5crypto.so.3 (0x00007ffef5bde000)
    libcom_err.so.2 => /lib64/libcom_err.so.2 (0x00007ffef59da000)
    libkrb5support.so.0 => /usr/lib64/libkrb5support.so.0 (0x00007ffef57
    libresolv.so.2 => /lib64/libresolv.so.2 (0x00007ffef55b3000)
```

```
        libsasl2.so.3 => /usr/lib64/libsasl2.so.3 (0x00007ffef5396000)
        libbrotlicommon.so.1 => /usr/lib64/libbrotlicommon.so.1 (0x00007ffef!
        libkeyutils.so.1 => /usr/lib64/libkeyutils.so.1 (0x00007ffef4f70000)
        libselinux.so.1 => /lib64/libselinux.so.1 (0x00007ffef4d47000)
        libpcre.so.1 => /usr/lib64/libpcre.so.1 (0x00007ffef4abe000)
*************************
*****ls -lh /mnt/bb/hagertnl*****
total 236M
-rwxr-xr-x 1 hagertnl hagertnl 236M Mar 28 17:01 lmp
drwx------ 2 hagertnl hagertnl  114 Mar 28 17:01 lmp_libs
*****ls -lh /mnt/bb/hagertnl/lmp_libs*****
total 9.2M
-rwxr-xr-x 1 hagertnl hagertnl 1.6M Oct  6  2021 libhwloc.so.15
-rwxr-xr-x 1 hagertnl hagertnl 1.6M Oct  6  2021 libiconv.so.2
-rwxr-xr-x 1 hagertnl hagertnl 783K Oct  6  2021 liblzma.so.5
-rwxr-xr-x 1 hagertnl hagertnl 149K Oct  6  2021 libpciaccess.so.0
-rwxr-xr-x 1 hagertnl hagertnl 5.2M Oct  6  2021 libxml2.so.2
*****ldd /mnt/bb/hagertnl/lmp*****
        linux-vdso.so.1 (0x00007fffeda02000)
        libgcc_s.so.1 => /lib64/libgcc_s.so.1 (0x00007fffed5bb000)
        libpthread.so.0 => /lib64/libpthread.so.0 (0x00007fffed398000)
        libm.so.6 => /lib64/libm.so.6 (0x00007fffed04d000)
        librt.so.1 => /lib64/librt.so.1 (0x00007fffece44000)
        libamdhip64.so.4 => /opt/rocm-4.5.2/lib/libamdhip64.so.4 (0x00007fff
        libmpi_cray.so.12 => /opt/cray/pe/lib64/libmpi_cray.so.12 (0x00007ff
        libmpi_gtl_hsa.so.0 => /opt/cray/pe/lib64/libmpi_gtl_hsa.so.0 (0x000
        libhsa-runtime64.so.1 => /opt/rocm-5.3.0/lib/libhsa-runtime64.so.1 (
        libhwloc.so.15 => /mnt/bb/hagertnl/lmp_libs/libhwloc.so.15 (0x00007f
        libdl.so.2 => /lib64/libdl.so.2 (0x00007fffe8b7e000)
        libhipfft.so => /opt/rocm-5.3.0/lib/libhipfft.so (0x00007fffed9d2000
        libstdc++.so.6 => /usr/lib64/libstdc++.so.6 (0x00007fffe875b000)
        libc.so.6 => /lib64/libc.so.6 (0x00007fffe8366000)
        /lib64/ld-linux-x86-64.so.2 (0x00007fffed7da000)
        libamd_comgr.so.2 => /opt/rocm-5.3.0/lib/libamd_comgr.so.2 (0x00007f
        libnuma.so.1 => /usr/lib64/libnuma.so.1 (0x00007fffe04d4000)
        libfabric.so.1 => /opt/cray/libfabric/1.15.2.0/lib64/libfabric.so.1
        libatomic.so.1 => /usr/lib64/libatomic.so.1 (0x00007fffdffd9000)
        libpmi.so.0 => /opt/cray/pe/lib64/libpmi.so.0 (0x00007fffdfdd7000)
        libpmi2.so.0 => /opt/cray/pe/lib64/libpmi2.so.0 (0x00007fffdfb9e000)
        libquadmath.so.0 => /usr/lib64/libquadmath.so.0 (0x00007fffdf959000)
        libmodules.so.1 => /opt/cray/pe/lib64/cce/libmodules.so.1 (0x00007ff
        libfi.so.1 => /opt/cray/pe/lib64/cce/libfi.so.1 (0x00007fffdf3b4000)
        libcraymath.so.1 => /opt/cray/pe/lib64/cce/libcraymath.so.1 (0x00007
        libf.so.1 => /opt/cray/pe/lib64/cce/libf.so.1 (0x00007fffed83b000)
        libu.so.1 => /opt/cray/pe/lib64/cce/libu.so.1 (0x00007fffdf2ab000)
        libcsup.so.1 => /opt/cray/pe/lib64/cce/libcsup.so.1 (0x00007fffed832
        libamdhip64.so.5 => /opt/rocm-5.3.0/lib/libamdhip64.so.5 (0x00007fff
        libelf.so.1 => /usr/lib64/libelf.so.1 (0x00007fffdd871000)
        libdrm.so.2 => /usr/lib64/libdrm.so.2 (0x00007fffdd65d000)
        libdrm_amdgpu.so.1 => /usr/lib64/libdrm_amdgpu.so.1 (0x00007fffdd453
        libpciaccess.so.0 => /sw/frontier/spack-envs/base/opt/linux-sles15-x8
        libxml2.so.2 => /sw/frontier/spack-envs/base/opt/linux-sles15-x86_64/
        librocfft.so.0 => /opt/rocm-5.3.0/lib/librocfft.so.0 (0x00007fffdca1a
        libz.so.1 => /lib64/libz.so.1 (0x00007fffdc803000)
        libtinfo.so.6 => /lib64/libtinfo.so.6 (0x00007fffdc5d5000)
        libcxi.so.1 => /usr/lib64/libcxi.so.1 (0x00007fffdc3b0000)
        libcurl.so.4 => /usr/lib64/libcurl.so.4 (0x00007fffdc311000)
        libjson-c.so.3 => /usr/lib64/libjson-c.so.3 (0x00007fffdc101000)
        libpals.so.0 => /opt/cray/pe/lib64/libpals.so.0 (0x00007fffdbefc000)
        libgfortran.so.5 => /opt/cray/pe/gcc-libs/libgfortran.so.5 (0x00007ff
        liblzma.so.5 => /sw/frontier/spack-envs/base/opt/linux-sles15-x86_64/
        libiconv.so.2 => /sw/frontier/spack-envs/base/opt/linux-sles15-x86_64
        librocfft-device-0.so.0 => /opt/rocm-5.3.0/lib/librocfft-device-0.so
        librocfft-device-1.so.0 => /opt/rocm-5.3.0/lib/librocfft-device-1.so
        librocfft-device-2.so.0 => /opt/rocm-5.3.0/lib/librocfft-device-2.so
        librocfft-device-3.so.0 => /opt/rocm-5.3.0/lib/librocfft-device-3.so
        libnghttp2.so.14 => /usr/lib64/libnghttp2.so.14 (0x00007ffef7cc6000)
        libidn2.so.0 => /usr/lib64/libidn2.so.0 (0x00007ffef7aa9000)
        libssh.so.4 => /usr/lib64/libssh.so.4 (0x00007ffef783b000)
        libpsl.so.5 => /usr/lib64/libpsl.so.5 (0x00007ffef7629000)
        libssl.so.1.1 => /usr/lib64/libssl.so.1.1 (0x00007ffef758a000)
        libcrypto.so.1.1 => /usr/lib64/libcrypto.so.1.1 (0x00007ffef724a000)
        libgssapi_krb5.so.2 => /usr/lib64/libgssapi_krb5.so.2 (0x00007ffef6f
        libldap_r-2.4.so.2 => /usr/lib64/libldap_r-2.4.so.2 (0x00007ffef6da4
        liblber-2.4.so.2 => /usr/lib64/liblber-2.4.so.2 (0x00007ffef6b95000)
        libzstd.so.1 => /usr/lib64/libzstd.so.1 (0x00007ffef6865000)
        libbrotlidec.so.1 => /usr/lib64/libbrotlidec.so.1 (0x00007ffef665900
        libunistring.so.2 => /usr/lib64/libunistring.so.2 (0x00007ffef62d600
        libjitterentropy.so.3 => /usr/lib64/libjitterentropy.so.3 (0x00007ff
        libkrb5.so.3 => /usr/lib64/libkrb5.so.3 (0x00007ffef5df6000)
        libk5crypto.so.3 => /usr/lib64/libk5crypto.so.3 (0x00007ffef5bde000)
        libcom_err.so.2 => /lib64/libcom_err.so.2 (0x00007ffef59da000)
        libkrb5support.so.0 => /usr/lib64/libkrb5support.so.0 (0x00007ffef57
        libresolv.so.2 => /lib64/libresolv.so.2 (0x00007ffef55b3000)
        libsasl2.so.3 => /usr/lib64/libsasl2.so.3 (0x00007ffef5396000)
        libbrotlicommon.so.1 => /usr/lib64/libbrotlicommon.so.1 (0x00007ffef
        libkeyutils.so.1 => /usr/lib64/libkeyutils.so.1 (0x00007ffef4f70000)
        libselinux.so.1 => /lib64/libselinux.so.1 (0x00007ffef4d47000)
        libpcre.so.1 => /usr/lib64/libpcre.so.1 (0x00007ffef4abe000)
    *************************************
```

Notice that the libraries are sent to the `${exe}_libs` directory in the same prefix as the executable. Once libraries are here, you cannot tell where they came from, so consider doing an `ldd` of your executable prior to `sbcast`.

Alternative: SBCASTing a binary with all libraries

As mentioned above, you can alternatively use `--exclude=NONE` on `sbcast` to send all libraries along with the binary. Using `--exclude=NONE` requires more effort but substantially simplifies the linker configuration at run-time. A job script for the previous example, modified for sending all libraries is shown below.

```
#!/bin/bash
#SBATCH -A <projid>
#SBATCH -J sbcast_binary_to_nvme
#SBATCH -o %x-%j.out
#SBATCH -t 00:05:00
#SBATCH -p batch
#SBATCH -N 2
#SBATCH -C nvme

date

# Change directory to user scratch space (Orion)
cd /lustre/orion/<projid>/scratch/<userid>

# For this example, I use a HIP-enabled LAMMPS binary, with dependencies
exe="lmp"

echo "*****ldd ./${exe}*****"
ldd ./${exe}
echo "*************************"

# SBCAST executable from Orion to NVMe -- NOTE: ``-C nvme`` is needed in
# NOTE: dlopen'd files will NOT be picked up by sbcast
sbcast --send-libs --exclude=NONE -pf ${exe} /mnt/bb/$USER/${exe}
if [ ! "$?" == "0" ]; then
    # CHECK EXIT CODE. When SBCAST fails, it may leave partial files on
    # your application may pick up partially complete shared library fil
    echo "SBCAST failed!"
    exit 1
fi

# Check to see if file exists
echo "*****ls -lh /mnt/bb/$USER*****"
ls -lh /mnt/bb/$USER/
echo "*****ls -lh /mnt/bb/$USER/${exe}_libs*****"
ls -lh /mnt/bb/$USER/${exe}_libs

# SBCAST sends all libraries detected by `ld` (minus any excluded), and
# Any libraries opened by `dlopen` are NOT sent, since they are not know

# All required libraries now reside in /mnt/bb/$USER/${exe}_libs
export LD_LIBRARY_PATH="/mnt/bb/$USER/${exe}_libs"
```

```
# libfabric dlopen's several libraries:
export LD_LIBRARY_PATH="${LD_LIBRARY_PATH}:$(pkg-config --variable=libdi

# cray-mpich dlopen's libhsa-runtime64.so and libamdhip64.so (non-versio
srun -N ${SLURM_NNODES} -n ${SLURM_NNODES} --ntasks-per-node=1 --label -l
    bash -c "if [ -f libhsa-runtime64.so.1 ]; then ln -s libhsa-runtime6
    if [ -f libamdhip64.so.5 ]; then ln -s libamdhip64.so.5 libamdhip64.

# RocBLAS has over 1,000 device libraries that may be `dlopen`'d by RocBl
# It's impractical to SBCAST all of these, so you can set this path inst
#export ROCBLAS_TENSILE_LIBPATH=${ROCM_PATH}/lib/rocblas/library

# You may notice that some libraries are still linked from /sw/crusher,
# This is because the Spack-build modules use RPATH to find their depend
echo "*****ldd /mnt/bb/$USER/${exe}*****"
ldd /mnt/bb/$USER/${exe}
echo "*************************************"
```

Some libraries still resolved to paths outside of `/mnt/bb` , and the reason for that is that the executable may have several paths in `RPATH` .

Software

Visualization and analysis tasks should be done on the Andes cluster. There are a few tools provided for various visualization tasks, as described in the :ref:`andes-viz-tools` section of the :ref:`andes-user-guide`.

For a full list of software availability and latest news at the OLCF, please reference the :doc:`Software </software/software-news>` section in OLCF's User Documentation.

Debugging

Linaro DDT

Linaro DDT is an advanced debugging tool used for scalar, multi-threaded, and large-scale parallel applications. In addition to traditional debugging features (setting breakpoints, stepping through code, examining variables), DDT also supports attaching to already-running processes and memory debugging. In-depth details of DDT can be found in the Official DDT User Guide, and instructions for how to use it on OLCF systems can be found on the :doc:`Debugging Software </software/debugging/index>` page. DDT is the OLCF's recommended debugging software for large parallel applications.

One of the most useful features of DDT is its remote debugging feature. This allows you to connect to a debugging session on Frontier from a client running on your workstation. The local client provides much faster interaction than you would have if using the graphical client on Frontier. For guidance in setting up the remote client see the :doc:`Debugging Software </software/debugging/index>` page.

GDB

GDB, the GNU Project Debugger, is a command-line debugger useful for traditional debugging and investigating code crashes. GDB lets you debug programs written in Ada, C, C++, Objective-C, Pascal (and many other languages).

GDB is availableon Summit under all compiler families:

```
module load gdb
```

To use GDB to debug your application run:

```
gdb ./path_to_executable
```

Additional information about GDB usage can befound on the GDB Documentation Page.

Valgrind4hpc

Valgrind4hpc is a Valgrind-based debugging tool to aid in the detection of memory leaks and errors in parallel applications. Valgrind4hpc aggregates any duplicate messages across ranks to help provide an understandable picture of program behavior. Valgrind4hpc manages starting and redirecting output from many copies of Valgrind, as well as deduplicating and filtering Valgrind messages. If your program can be debugged with Valgrind, it can be debugged with Valgrind4hpc.

Valgrind4hpc is available on Frontier under all compiler families:

```
module load valgrind4hpc
```

Additional information about Valgrind4hpc usage can be found on the HPE Cray Programming Environment User Guide Page.

Profiling Applications

Getting Started with the HPE Performance Analysis Tools (PAT)

The Performance Analysis Tools (PAT), formerly CrayPAT, are a suite of utilities that enable users to capture and analyze performance data generated during program execution. These tools provide an integrated infrastructure for measurement, analysis, and visualization of computation, communication, I/O, and memory utilization to help users optimize programs for faster execution and more efficient computing resource usage.

There are three programming interfaces available: (1) `Perftools-lite` , (2) `Perftools` , and (3) `Perftools-preload` .

Below are two examples that generate an instrumented executable using `Perftools` , which is an advanced interface that provides full-featured data collection and analysis capability, including full traces with timeline displays.

The first example generates an instrumented executable using a `PrgEnv-amd` build:

```
module load PrgEnv-amd
module load craype-accel-amd-gfx90a
module load rocm
module load perftools

export PATH="${PATH}:${ROCM_PATH}/llvm/bin"
export CXX='CC -x hip'
export CXXFLAGS='-ggdb -O3 -std=c++17 -Wall'
export LD='CC'
export LDFLAGS="${CXXFLAGS} -L${ROCM_PATH}/lib"
export LIBS='-lamdhip64'

make clean
make

pat_build -g hip,io,mpi -w -f <executable>
```

The second example generates an instrumened executable using a `hipcc` build:

```
module load perftools
module load craype-accel-amd-gfx90a
module load rocm

export CXX='hipcc'
export CXXFLAGS="$(pat_opts include hipcc) \
  $(pat_opts pre_compile hipcc) -g -O3 -std=c++17 -Wall \
  --offload-arch=gfx90a -I${CRAY_MPICH_DIR}/include \
  $(pat_opts post_compile hipcc)"
```

```
export LD='hipcc'
export LDFLAGS="$(pat_opts pre_link hipcc) ${CXXFLAGS} \
  -L${CRAY_MPICH_DIR}/lib ${PE_MPICH_GTL_DIR_amd_gfx908}"
export LIBS="-lmpi ${PE_MPICH_GTL_LIBS_amd_gfx908} \
  $(pat_opts post_link hipcc)"

make clean
make

pat_build -g hip,io,mpi -w -f <executable>
```

The `pat_build` command in the above examples generates an instrumented executable with `+pat` appended to the executable name (e.g., `hello_jobstep+pat`).

When run, the instrumented executable will trace HIP, I/O, MPI, and all user functions and generate a folder of results (e.g., `hello_jobstep+pat+39545-2t`).

To analyze these results, use the `pat_report` command, e.g.:

```
pat_report hello_jobstep+pat+39545-2t
```

The resulting report includes profiles of functions, profiles of maximum function times, details on load imbalance, details on program energy and power usages, details on memory high water mark, and more.

More detailed information on the HPE Performance Analysis Tools can be found in the HPE Performance Analysis Tools User Guide.

Note

When using `perftools-lite-gpu` , there is a known issue causing `ld.lld` not to be found. A workaround this issue can be found here.

Getting Started with HPCToolkit

HPCToolkit is an integrated suite of tools for measurement and analysis of program performance on computers ranging from multicore desktop systems to the nation's largest supercomputers. HPCToolkit provides accurate measurements of a program's work, resource consumption, and inefficiency, correlates these metrics with the program's source code, works with multilingual, fully optimized binaries, has very low measurement overhead, and scales to large parallel systems. HPCToolkit's measurements provide support for analyzing a program execution cost, inefficiency, and scaling characteristics both within and across nodes of a parallel system.

Programming models supported by HPCToolkit include MPI, OpenMP, OpenACC, CUDA, OpenCL, DPC++, HIP, RAJA, Kokkos, and others.

Below is an example that generates a profile and loads the results in their GUI-based viewer.

Note

A full list of available HPCToolkit versions can be seen with the `module spider hpctoolkit` command.

```
module load hpctoolkit/2022.05.15-rocm

# 1. Profile and trace an application using CPU time and GPU performance
srun <srun_options> hpcrun -o <measurement_dir> -t -e CPUTIME -e gpu=amd

# 2. Analyze the binary of executables and its dependent libraries
hpcstruct <measurement_dir>

# 3. Combine measurements with program structure information and generat
hpcprof -o <database_dir> <measurement_dir>

# 4. Understand performance issues by analyzing profiles and traces with
hpcviewer <database_dir>
```

More detailed information on HPCToolkit can be found in the HPCToolkit User's Manual.

Note

HPCToolkit does not require a recompile to profile the code. It is recommended to use the -g optimization flag for attribution to source lines.

Getting Started with the ROCm Profiler

`rocprof` gathers metrics on kernels run on AMD GPU architectures. The profiler works for HIP kernels, as well as offloaded kernels from OpenMP target offloading, OpenCL, and abstraction layers such as Kokkos. For a simple view of kernels being run, `rocprof --stats --timestamp on` is a great place to start. With the `--stats` option enabled, `rocprof` will generate a file that is named `results.stats.csv` by default, but named `<output>.stats.csv` if the `-o` flag is supplied. This file will list all kernels being run, the number of times they are run, the total duration and the average duration (in nanoseconds) of the kernel, and the GPU usage percentage. More detailed infromation on `rocprof` profiling modes can be found at ROCm Profiler documentation.

Note

If you are using `sbcast`, you need to explicitly `sbcast` the AQL profiling library found in `${ROCM_PATH}/hsa-amd-aqlprofile/lib/libhsa-amd-aqlprofile64.so`. A symbolic link to this library can also be found in `${ROCM_PATH}/lib`. Alternatively, you may leave `${ROCM_PATH}/lib` in your `LD_LIBRARY_PATH`.

Roofline Profiling with the ROCm Profiler

The Roofline performance model is an increasingly popular way to demonstrate and understand application performance. This section documents how to construct a simple roofline model for a single kernel using `rocprof`. This roofline model is designed to be comparable to rooflines constructed by NVIDIA's NSight Compute. A roofline model plots the achieved performance (in floating-point operations per second, FLOPS/s) as a function of arithmetic (or operational) intensity (in FLOPS per Byte). The model detailed here calculates the bytes moved as they move to and from the GPU's HBM.

Note

Integer instructions and cache levels are currently not documented here.

To get started, you will need to make an input file for `rocprof`, to be passed in through `rocprof -i <input_file> --timestamp on -o my_output.csv <my_exe>`. Below is an example, and contains the information needed to roofline profile GPU 0, as seen by each rank:

```
pmc : TCC_EA_RDREQ_32B_sum TCC_EA_RDREQ_sum TCC_EA_WRREQ_sum TCC_EA_WRREQ_64B_
pmc : SQ_INSTS_VALU_ADD_F64 SQ_INSTS_VALU_MUL_F64 SQ_INSTS_VALU_FMA_F64 SQ_INS
gpu: 0
```

Note

In an application with more than one kernel, you should strongly consider filtering by kernel name by adding a line like: `kernel: <kernel_name>` to the `rocprof` input file.

This provides the minimum set of metrics used to construct a roofline model, in the minimum number of passes. Each line that begins with `pmc` indicates that the application will be re-run, and the metrics in that line will be collected. `rocprof` can collect up to 8 counters from each block (`SQ`, `TCC`) in each application re-run. To gather metrics across multiple MPI ranks, you will need to use a command that redirects the output of rocprof to a unique file for each task. For example:

```
srun -N 2 -n 16 --ntasks-per-node=8 --gpus-per-node=8 --gpu-bind=closest
```

Note

The `gpu:` filter in the `rocprof` input file identifies GPUs by the number the MPI rank would see them as. In the `srun` example above, each MPI rank only has 1 GPU, so each rank sees its GPU as GPU 0.

Theoretical Roofline

The theoretical (not attainable) peak roofline constructs a theoretical maximum performance for each operational intensity.

Note

`theoretical` peak is determined by the hardware specifications and is not attainable in practice. `attaiable` peak is the performance as measured by in-situ microbenchmarks designed to best utilize the hardware. `achieved` performance is what the profiled application actually achieves.

The theoretical roofline can be constructed as:

```
FLOPS_{peak} = minimum(ArithmeticIntensity * BW_{HBM}, TheoreticalFLOPS)
```

On Frontier, the memory bandwidth for HBM is 1.6 TB/s, and the theoretical peak floating-point FLOPS/s when using vector registers is calculated by:

```
TheoreticalFLOPS = 128 FLOP/cycle/CU * 110 CU * 1700000000 cycles/second = 23.
```

However, when using MFMA instructions, the theoretical peak floating-point FLOPS/s is calculated by:

```
TheoreticalFLOPS = flop\_per\_cycle(precision) FLOP/cycle/CU * 110 CU * 170000
```

where `flop_per_cycle(precision)` is the published floating-point operations per clock cycle, per compute unit. Those values are:

Data Type	Flops/Clock/CU
FP64	256
FP32	256
FP16	1024
BF16	1024
INT8	1024

Note

Attainable peak rooflines are constructed using microbenchmarks, and are not currently discussed here. Attainable rooflines consider the limitations of cooling and power consumption and are more representative of what an application can achieve.

Achieved FLOPS/s

We calculate the achieved performance at the desired level (here, double-precision floating point, FP64), by summing each metric count and weighting the FMA metric by 2, since a fused multiply-add is considered 2 floating point operations. Also note that these `SQ_INSTS_VALU_<ADD,MUL,TRANS>` metrics are reported as per-simd, so we mutliply by the wavefront size as well. The `SQ_INSTS_VALU_MFMA_MOPS_*` instructions should be multiplied by the `Flops/Cycle/CU` value listed above. We use this equation to calculate the number of double-precision FLOPS:

```
FP64\_FLOPS =   64  *&(SQ\_INSTS\_VALU\_ADD\_F64         \\\\
                     &+ SQ\_INSTS\_VALU\_MUL\_F64       \\\\
                     &+ SQ\_INSTS\_VALU\_TRANS\_F64     \\\\
                     &+ 2 * SQ\_INSTS\_VALU\_FMA\_F64)  \\\\
              + 256 *&(SQ\_INSTS\_VALU\_MFMA\_MOPS\_F64)
```

When `SQ_INSTS_VALU_MFMA_MOPS_*_F64` instructions are used, then 47.8 TF/s is considered the theoretical maximum FLOPS/s. If only `SQ_INSTS_VALU_<ADD,MUL,TRANS>` are found, then 23.9 TF/s is the theoretical maximum FLOPS/s. Then, we divide the number of FLOPS by the elapsed time of the kernel to find FLOPS per second. This is found from subtracting the `rocprof` metrics `EndNs` by `BeginNs`, provided by `--timestamp on`, then converting from nanoseconds to seconds by dividing by 1,000,000,000 (power(10,9)).

Note

For ROCm/5.2.0 and earlier, there is a known issue with the timings provided by `--timestamp on`. See :ref:`crusher-known-issues`.

Calculating for all precisions

The above formula can be adapted to compute the total FLOPS across all floating-point precisions (`INT` excluded).

```
TOTAL\_FLOPS =   64  *&(SQ\_INSTS\_VALU\_ADD\_F16        \\\\
                     &+ SQ\_INSTS\_VALU\_MUL\_F16       \\\\
                     &+ SQ\_INSTS\_VALU\_TRANS\_F16     \\\\
                     &+ 2 * SQ\_INSTS\_VALU\_FMA\_F16)  \\\\
              + 64  *&(SQ\_INSTS\_VALU\_ADD\_F32         \\\\
                     &+ SQ\_INSTS\_VALU\_MUL\_F32       \\\\
                     &+ SQ\_INSTS\_VALU\_TRANS\_F32     \\\\
                     &+ 2 * SQ\_INSTS\_VALU\_FMA\_F32)  \\\\
              + 64  *&(SQ\_INSTS\_VALU\_ADD\_F64         \\\\
                     &+ SQ\_INSTS\_VALU\_MUL\_F64       \\\\
                     &+ SQ\_INSTS\_VALU\_TRANS\_F64     \\\\
                     &+ 2 * SQ\_INSTS\_VALU\_FMA\_F64)  \\\\
              + 1024 &*(SQ\_INSTS\_VALU\_MFMA\_MOPS\_F16) \\\\
              + 1024 &*(SQ\_INSTS\_VALU\_MFMA\_MOPS\_BF16) \\\\
              + 256 *&(SQ\_INSTS\_VALU\_MFMA\_MOPS\_F32) \\\\
              + 256 *&(SQ\_INSTS\_VALU\_MFMA\_MOPS\_F64) \\\\
```

Arithmetic Intensity

Arithmetic intensity calculates the ratio of FLOPS to bytes moved between HBM and L2 cache. We calculated FLOPS above (FP64_FLOPS). We can calculate the number of bytes moved using the `rocprof` metrics `TCC_EA_WRREQ_64B`, `TCC_EA_WRREQ_sum`, `TCC_EA_RDREQ_32B`, and `TCC_EA_RDREQ_sum`. `TCC` refers to the L2 cache, and `EA` is the interface between L2 and HBM. `WRREQ` and `RDREQ` are write-requests and read-requests, respectively. Each of these requests is either 32 bytes or 64 bytes. So we calculate the number of bytes traveling over the EA interface as:

```
BytesMoved = BytesWritten + BytesRead
```

where

```
BytesWritten = 64 * TCC\_EA\_WRREQ\_64B\_sum + 32 * (TCC\_EA\_WRREQ\_sum - TCC
BytesRead = 32 * TCC\_EA\_RDREQ\_32B\_sum + 64 * (TCC\_EA\_RDREQ\_sum - TCC\_E
```

Tips and Tricks

This section details 'tips and tricks' and information of interest to users when porting from Summit to Frontier.

Using reduced precision (FP16 and BF16 datatypes)

Users leveraging BF16 and FP16 datatypes for applications such as ML/AI training and low-precision matrix multiplication should be aware that the AMD MI250X GPU has different denormal handling than the V100 GPUs on Summit. On the MI250X, the V_DOT2 and the matrix instructions for FP16 and BF16 flush input and output denormal values to zero. FP32 and FP64 MFMA instructions do not flush input and output denormal values to zero.

When training deep learning models using FP16 precision, some models may fail to converge with FP16 denorms flushed to zero. This occurs in operations encountering denormal values, and so is more likely to occur in FP16 because of a small dynamic range. BF16 numbers have a larger dynamic range than FP16 numbers and are less likely to encounter denormal values.

AMD has provided a solution in ROCm 5.0 which modifies the behavior of Tensorflow, PyTorch, and rocBLAS. This modification starts with FP16 input values, casting the intermediate FP16 values to BF16, and then casting back to FP16 output after the accumulate FP32 operations. In this way, the input and output types are unchanged. The behavior is enabled by default in machine learning frameworks. This behavior requires user action in rocBLAS, via a special enum type. For more information, see the rocBLAS link below.

If you encounter significant differences when running using reduced precision, explore replacing non-converging models in FP16 with BF16, because of the greater dynamic range in BF16. We recommend using BF16 for ML models in general. If you have further questions or encounter issues, contact help@olcf.ornl.gov.

Additional information on MI250X reduced precision can be found at:

- The MI250X ISA specification details the flush to zero denorm behavior at: https://developer.amd.com/wpcontent/resources/CDNA2_Shader_ISA_18November2021.pdf (See page 41 and 46)

- AMD rocBLAS library reference guide details this behavior at: https://rocblas.readthedocs.io/en/master/API_Reference_Guide.html#mi200-gfx90a-considerations

Enabling GPU Page Migration

The AMD MI250X and operating system on Frontier supports unified virtual addressing across the entire host and device memory, and automatic page migration between CPU and GPU memory. Migratable, universally addressable memory is sometimes called 'managed' or 'unified' memory, but neither of these terms fully describes how memory may behave on Frontier. In the following section we'll discuss how the heterogenous memory space on a Frontier node is surfaced within your application.

The accessibility of memory from GPU kernels and whether pages may migrate depends three factors: how the memory was allocated; the XNACK operating mode of the GPU; whether the kernel was compiled to support page migration. The latter two factors are intrinsically linked, as the MI250X GPU operating mode restricts the types of kernels which may run.

XNACK (pronounced X-knack) refers to the AMD GPU's ability to retry memory accesses that fail due to a page fault. The XNACK mode of an MI250X can be changed by setting the environment variable `HSA_XNACK` before starting a process that uses the GPU. Valid values are 0 (disabled) and 1 (enabled), and all processes connected to a GPU must use the same XNACK setting. The default MI250X on Frontier is `HSA_XNACK=0` .

If `HSA_XNACK=0` , page faults in GPU kernels are not handled and will terminate the kernel. Therefore all memory locations accessed by the GPU must either be resident in the GPU HBM or mapped by the HIP runtime. Memory regions may be migrated between the host DDR4 and GPU HBM using explicit HIP library functions such as `hipMemAdvise` and `hipPrefetchAsync` , but memory will not be automatically migrated based on access patterns alone.

If `HSA_XNACK=1` , page faults in GPU kernels will trigger a page table lookup. If the memory location can be made accessible to the GPU, either by being migrated to GPU HBM or being mapped for remote access, the appropriate action will occur and the access will be replayed. Page migration will happen between CPU DDR4 and GPU HBM according to page touch. The exceptions are if the programmer uses a HIP library call such as `hipPrefetchAsync` to request migration, or if a preferred location is set via `hipMemAdvise` , or if GPU HBM becomes full and the page must forcibly be evicted back to CPU DDR4 to make room for other data.

Migration of Memory by Allocator and XNACK Mode

Most applications that use "managed" or "unified" memory on other platforms will want to enable XNACK to take advantage of automatic page migration on Frontier. The following table shows how common allocators currently behave with XNACK enabled. The behavior of a specific memory region may vary from the default if the programmer uses certain API calls.

Note

The page migration behavior summarized by the following tables represents the current, observable behavior. Said behavior will likely change in the near future.

`HSA_XNACK=1` **Automatic Page Migration Enabled**

Allocator	Initial Physical Location	CPU Access after GPU First Touch	Default Behavior for GPU Access
System Allocator (malloc,new,allocate, etc)	CPU DDR4	Migrate to CPU DDR4 on touch	Migrate to GPU HBM on touch
hipMallocManaged	CPU DDR4	Migrate to CPU DDR4 on touch	Migrate to GPU HBM on touch
hipHostMalloc	CPU DDR4	Local read/write	Zero copy read/write over Infinity Fabric
hipMalloc	GPU HBM	Zero copy read/write over Inifinity Fabric	Local read/write

Disabling XNACK will not necessarily result in an application failure, as most types of memory can still be accessed by the AMD "Optimized 3rd Gen EPYC" CPU and AMD MI250X GPU. In most cases, however, the access will occur in a zero-copy fashion over the Infinity Fabric. The exception is memory allocated through standard system allocators such as `malloc`, which cannot be accessed directly from GPU kernels without previously being registered via a HIP runtime call such as `hipHostRegister`. Access to malloc'ed and unregistered memory from GPU kernels will result in fatal unhandled page faults. The table below shows how common allocators behave with XNACK disabled.

`HSA_XNACK=0` **Automatic Page Migration Disabled**

Allocator	Initial Physical Location	Default Behavior for CPU Access	Default Behavior for GPU Access
System Allocator (malloc,new,allocate, etc)	CPU DDR4	Local read/write	Fatal Unhandled Page Fault
hipMallocManaged	CPU DDR4	Zero copy read/write over Infinity Fabric	Local read/write
hipHostMalloc	CPU DDR4	Local read/write	Zero copy read/write over Infinity Fabric
hipMalloc	GPU HBM	Zero copy read/write over Inifinity Fabric	Local read/write

Compiling HIP kernels for specific XNACK modes

Although XNACK is a capability of the MI250X GPU, it does require that kernels be able to recover from page faults. Both the ROCm and CCE HIP compilers will default to generating code that runs correctly with both XNACK enabled and disabled. Some applications may benefit from using the following compilation options to target specific XNACK modes.

`hipcc --amdgpu-target=gfx90a` or `CC --offload-arch=gfx90a -x hip`
Kernels are compiled to a single "xnack any" binary, which will run correctly with both XNACK enabled and XNACK disabled.
`hipcc --amdgpu-target=gfx90a:xnack+` or `CC --offload-arch=gfx90a:xnack+ -x hip`
Kernels are compiled in "xnack plus" mode and will *only* be able to run on GPUs with `HSA_XNACK=1` to enable XNACK. Performance may be better than "xnack any", but attempts to run with XNACK disabled will fail.
`hipcc --amdgpu-target=gfx90a:xnack-` or `CC --offload-arch=gfx90a:xnack- -x hip`
Kernels are compiled in "xnack minus" mode and will *only* be able to run on GPUs with `HSA_XNACK=0` and XNACK disabled. Performance may be better than "xnack any", but attempts to run with XNACK enabled will fail.
`hipcc --amdgpu-target=gfx90a:xnack- --amdgpu-target=gfx90a:xnack+ -x hip` or `CC --offload-arch=gfx90a:xnack- --offload-arch=gfx90a:xnack+ -x hip`
Two versions of each kernel will be generated, one that runs with XNACK disabled and one that runs if XNACK is enabled. This is different from "xnack any" in that two versions of each kernel are compiled and HIP picks the appropriate one at runtime, rather than

there being a single version compatible with both. A "fat binary" compiled in this way will have the same performance of "xnack+" with `HSA_XNACK=1` and as "xnack-" with `HSA_XNACK=0`, but the final executable will be larger since it contains two copies of every kernel.
If the HIP runtime cannot find a kernel image that matches the XNACK mode of the device, it will fail with `hipErrorNoBinaryForGpu`.

```
$ HSA_XNACK=0 srun -n 1 -N 1 -t 1 ./xnack_plus.exe
"hipErrorNoBinaryForGpu: Unable to find code object for all current devices!"
srun: error: frontier002: task 0: Aborted
srun: launch/slurm: _step_signal: Terminating StepId=74100.0
```

One way to diagnose `hipErrorNoBinaryForGpu` messages is to set the environment variable `AMD_LOG_LEVEL` to 1 or greater:

```
$ AMD_LOG_LEVEL=1 HSA_XNACK=0 srun -n 1 -N 1 -t 1 ./xnack_plus.exe
:1:rocdevice.cpp            :1573: 43966598070 us: HSA_AMD_AGENT_INFO_SVM_DIRE
:1:rocdevice.cpp            :1573: 43966598762 us: HSA_AMD_AGENT_INFO_SVM_DIRE
:1:rocdevice.cpp            :1573: 43966599392 us: HSA_AMD_AGENT_INFO_SVM_DIRE
:1:rocdevice.cpp            :1573: 43966599970 us: HSA_AMD_AGENT_INFO_SVM_DIRE
:1:rocdevice.cpp            :1573: 43966600550 us: HSA_AMD_AGENT_INFO_SVM_DIRE
:1:rocdevice.cpp            :1573: 43966601109 us: HSA_AMD_AGENT_INFO_SVM_DIRE
:1:rocdevice.cpp            :1573: 43966601673 us: HSA_AMD_AGENT_INFO_SVM_DIRE
:1:rocdevice.cpp            :1573: 43966602248 us: HSA_AMD_AGENT_INFO_SVM_DIRE
:1:hip_code_object.cpp      :460 : 43966602806 us: hipErrorNoBinaryForGpu: Una
:1:hip_code_object.cpp      :461 : 43966602810 us:   Devices:
:1:hip_code_object.cpp      :464 : 43966602811 us:     amdgcn-amd-amdhsa--gfx9
:1:hip_code_object.cpp      :464 : 43966602811 us:     amdgcn-amd-amdhsa--gfx9
:1:hip_code_object.cpp      :464 : 43966602812 us:     amdgcn-amd-amdhsa--gfx9
:1:hip_code_object.cpp      :464 : 43966602813 us:     amdgcn-amd-amdhsa--gfx9
:1:hip_code_object.cpp      :464 : 43966602813 us:     amdgcn-amd-amdhsa--gfx9
:1:hip_code_object.cpp      :464 : 43966602814 us:     amdgcn-amd-amdhsa--gfx9
:1:hip_code_object.cpp      :464 : 43966602814 us:     amdgcn-amd-amdhsa--gfx9
:1:hip_code_object.cpp      :464 : 43966602815 us:     amdgcn-amd-amdhsa--gfx9
:1:hip_code_object.cpp      :468 : 43966602816 us:   Bundled Code Objects:
:1:hip_code_object.cpp      :485 : 43966602817 us:     host-x86_64-unknown-lir
:1:hip_code_object.cpp      :483 : 43966602818 us:     hipv4-amdgcn-amd-amdhsa
"hipErrorNoBinaryForGpu: Unable to find code object for all current devices!"
srun: error: frontier129: task 0: Aborted
srun: launch/slurm: _step_signal: Terminating StepId=74102.0
```

The above log messages indicate the type of image required by each device, given its current mode (`amdgcn-amd-amdhsa--gfx90a:sramecc+:xnack-`) and the images found in the binary (`hipv4-amdgcn-amd-amdhsa--gfx90a:xnack+`).

Floating-Point (FP) Atomic Operations and Coarse/Fine Grained Memory Allocations

The Frontier system, equipped with CDNA2-based architecture MI250X cards, offers a coherent host interface that enables advanced memory and unique cache coherency capabilities. The AMD driver leverages the Heterogeneous Memory Management (HMM) support in the Linux kernel to perform seamless page migrations to/from CPU/GPUs. This new capability comes with a memory model that needs to be understood completely to avoid unexpected behavior in real applications. For more details, please visit the previous section.

AMD GPUs can allocate two different types of memory locations: 1) Coarse grained and 2) Fine grained.

Coarse grained memory is only guaranteed to be coherent outside of GPU kernels that modify it, enabling higher performance memory operations. Changes applied to coarse-grained memory by a GPU kernel are only visible to the rest of the system (CPU or other GPUs) when the kernel has completed. A GPU kernel is only guaranteed to see changes applied to coarse grained memory by the rest of the system (CPU or other GPUs) if those changes were made before the kernel launched.

Fine grained memory allows CPUs and GPUs to synchronize (via atomics) and coherently communicate with each other while the GPU kernel is running, allowing more advanced programming patterns. The additional visibility impacts the performance of fine grained allocated memory.

The fast hardware-based Floating point (FP) atomic operations available on MI250X are assumed to be working on coarse grained memory regions; when these instructions are applied to a fine-grained memory region, they will silently produce a no-op. To avoid returning incorrect results, the compiler never emits hardware-based FP atomics instructions by default, even when applied to coarse grained memory regions. Currently, users can use the -munsafe-fp-atomics flag to force the compiler to emit hardware-based FP atomics. Using hardware-based FP atomics translates in a substantial performance improvement over the default choice.

Users applying floating point atomic operations (e.g., atomicAdd) on memory regions allocated via regular hipMalloc() can safely apply the -munsafe-fp-atomics flags to their codes to get the best possible performance and leverage hardware supported floating point atomics. Atomic operations supported in hardware on non-FP datatypes (e.g., INT32) will work correctly regardless of the nature of the memory region used.

In ROCm-5.1 and earlier versions, the flag -munsafe-fp-atomics is interpreted as a suggestion by the compiler, whereas from ROCm-5.2 the flag will always enforce the use of fast hardware-based FP atomics.

The following tables summarize the result granularity of various combinations of allocators, flags and arguments.

For `hipHostMalloc()` , the following table shows the nature of the memory returned based on the flag passed as argument.

API	Flag	Results
hipHostMalloc()	hipHostMallocDefault	Fine grained
hipHostMalloc()	hipHostMallocNonCoherent	Coarse grained

The following table shows the nature of the memory returned based on the flag passed as argument to `hipExtMallocWithFlags()` .

API	Flag	Result
hipExtMallocWithFlags()	hipDeviceMallocDefault	Coarse grained
hipExtMallocWithFlags()	hipDeviceMallocFinegrained	Fine grained

Finally, the following table summarizes the nature of the memory returned based on the flag passed as argument to `hipMallocManaged()` and the use of CPU regular `malloc()` routine with the possible use of `hipMemAdvise()` .

API	MemAdvice	Result
hipMallocManaged()		Fine grained
hipMallocManaged()	hipMemAdvise (hipMemAdviseSetCoarseGrain)	Coarse grained
malloc()		Fine grained
malloc()	hipMemAdvise	Coarse

API	MemAdvice	Result
	(hipMemAdviseSetCoarseGrain)	grained

Performance considerations for LDS FP atomicAdd()

Hardware FP atomic operations performed in LDS memory are usually always faster than an equivalent CAS loop, in particular when contention on LDS memory locations is high. Because of a hardware design choice, FP32 LDS atomicAdd() operations can be slower than equivalent FP64 LDS atomicAdd(), in particular when contention on memory locations is low (e.g. random access pattern). The aforementioned behavior is only true for FP atomicAdd() operations. Hardware atomic operations for CAS/Min/Max on FP32 are usually faster than the FP64 counterparts. In cases when contention is very low, a FP32 CAS loop implementing an atomicAdd() operation could be faster than an hardware FP32 LDS atomicAdd(). Applications using single precision FP atomicAdd() are encouraged to experiment with the use of double precision to evaluate the trade-off between high atomicAdd() performance vs. potential lower occupancy due to higher LDS usage.

System Updates

2023-05-09

On Tuesday, May 9, 2023, the darshan-runtime modulefile was added to DefApps and is now loaded by default on Frontier. This module will allow users to profile the I/O of their applications with minimal impact. The logs are available to users on the Orion file system in /lustre/orion/darshan/<system>/<yyyy>/<mm>/<dd>. Unloading darshan-runtime is recommended for users profiling their applications with other profilers to prevent conflicts.

Known Issues