AMD Instinct™ High Performance Computing (HPC) and Tuning Guide

Revision:	**1027**
Issue Date:	**October 2021**

Table of Contents

[This page is left blank intentionally]

Chapter 1 Overview

HPC workloads have unique requirements. The default hardware and BIOS configurations for OEM platforms may not provide optimal performance for HPC workloads. To help enable optimal HPC settings on a per-platform and workload level, this guide calls out:

- BIOS settings that can impact performance
- hardware configuration best practices
- supported versions of operating systems
- workload-specific recommendations for optimal BIOS and operating system settings

There is also a discussion on the AMD Instinct™ software development environment, including information on how to install and run the DGEMM and STREAM benchmarks as well as GROMACS. This guidance provides a good starting point but is not exhaustively tested across all compilers.

Prerequisites to understanding this document and to perform tuning of HPC applications include:

- Experience configuring servers
- Administrative access to the Server's Management Interface (BMC)
- Administrative access to the operating system
- Familiarity with OEMs Server's Management Interface (BMC) is strongly recommended
- Familiarity with the OS specific tools for configuration, monitoring and troubleshooting is strongly recommended

This document provides guidance on tuning systems with AMD Instinct™ accelerators for High Performance Computing (HPC) workloads. This document is not an all-inclusive guide, and some items referred to may have similar, but different, names in various OEM systems (for example, OEM-specific BIOS settings). This document also provides suggestions on items that should be the initial focus of additional, application-specific tuning.

This document is based on the AMD EPYC™ 7002 series processor family (former codename "Rome"). One can expect very similar results for the AMD EYPC™ 7003 series processor family (former codename "Milan"). Specific differences in the configuration options or performance obtained will be explicitly called out through the document where needed.

While this guide is a good starting point, developers are encouraged to perform their own performance testing for additional tuning.

Chapter 2 AMD Instinct™ Hardware

This chapter, will briefly review hardware aspects of the AMD Instinct™ accelerators and the CNDA architecture that is the foundation of these GPUs.

2.1 System Architecture

Figure 1 shows the node-level architecture of a system that comprises two AMD EPYC™ processors and (up to) eight AMD Instinct™ accelerators. The two EPYC processors are connected to each other with the AMD Infinity™ fabric which provides a high-bandwidth (up to 18 GT/sec) and coherent links such that each processor can access the available node memory as a single shared-memory domain in a non-uniform memory architecture (NUMA) fashion. In a 2P, or dual-socket, configuration, three AMD Infinity™ fabric links are available to connect the processors plus one PCIe Gen 4 x16 link per processor can attach additional I/O devices such as the host adapters for the network fabric.

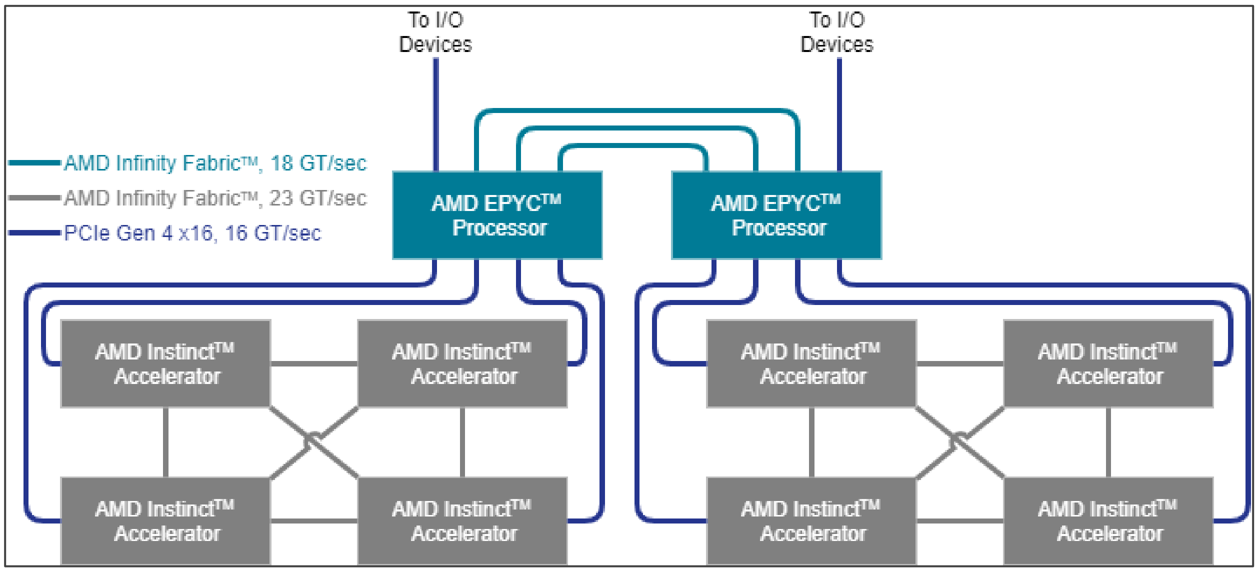

To I/O Devices
To I/O Devices
AMD Infinity Fabric™, 18 GT/sec
AMD Infinity Fabric™, 23 GT/sec
PCIe Gen 4 x16, 16 GT/sec
AMD EPYC™ Processor
AMD EPYC™ Processor
AMD Instinct™ Accelerator
AMD Instinct™ Accelerator
AMD Instinct™ Accelerator
AMD Instinct™ Accelerator
AMD Instinct™ Accelerator
AMD Instinct™ Accelerator
AMD Instinct™ Accelerator
AMD Instinct™ Accelerator

Figure 1: Node-level system architecture with two AMD EPYC™ processors and eight AMD Instinct™ accelerators.

In a typical node configuration, each processor can host up to four AMD Instinct™ accelerators that are attached using PCIe Gen 4 links at 16 GT/sec, which corresponds to a peak bidirectional link bandwidth of 32 GB/sec. Each hive of four accelerators can participate in a fully connected, coherent AMD Instinct™ fabric that connects the four accelerators using 23 GT/sec AMD Infinity fabric links that run at a higher frequency than the inter-processor links. This inter-GPU link can be established in certified server systems if the GPUs are mounted in neighboring PCIe slots by installing the AMD Infinity Fabric™ bridge for the AMD Instinct™ accelerators.

2.2 Micro-architecture

The micro-architecture of the AMD Instinct accelerators is based on the AMD CNDA architecture, which targets compute applications such as high-performance computing (HPC) and AI & machine learning (ML) that run on everything from individual servers to the world's largest exascale supercomputers. The overall system architecture is designed for extreme scalability and compute performance.

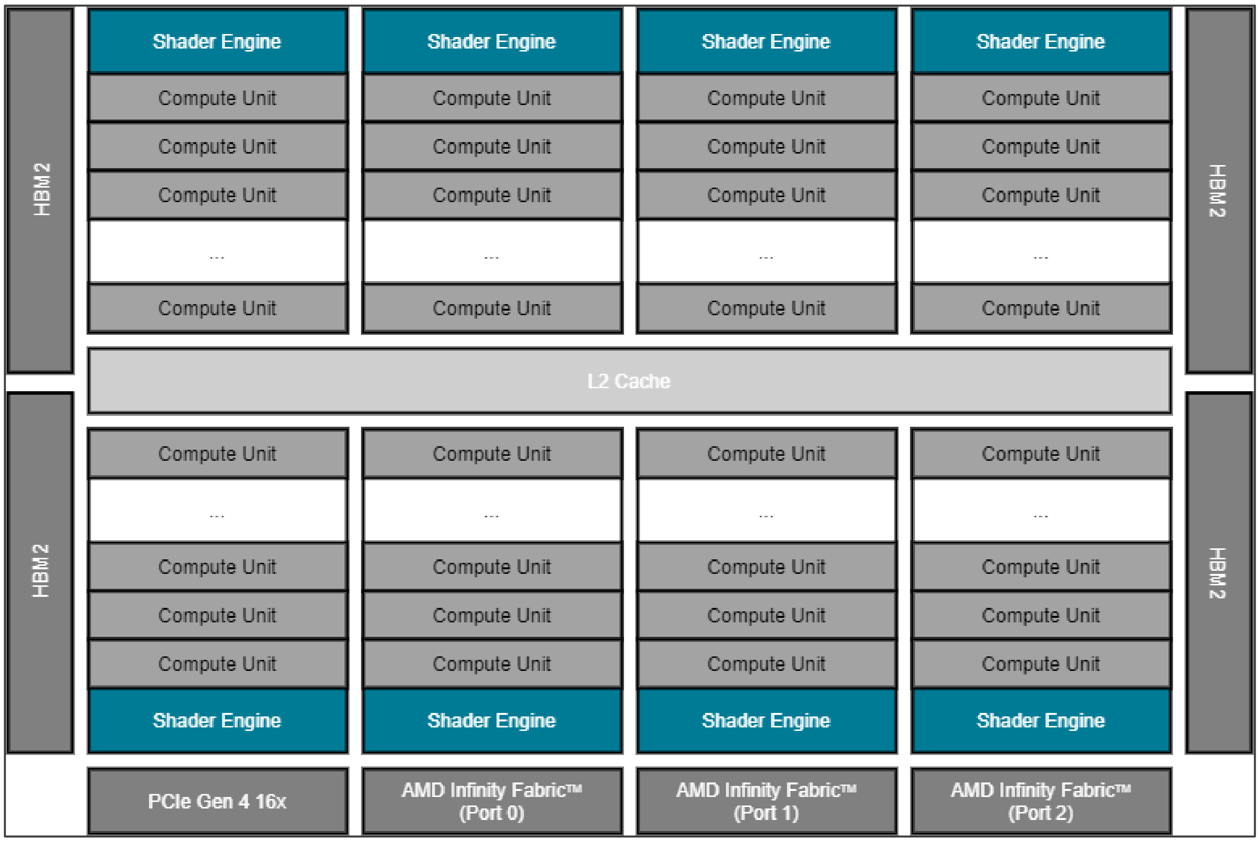

HBM2
Shader Engine
Compute Unit
...
L2 Cache
PCIe Gen 4 16x
AMD Infinity Fabric™ (Port 0)
AMD Infinity Fabric™ (Port 1)
AMD Infinity Fabric™ (Port 2)

Figure 2: Structure of the AMD Instinct accelerator (MI100 generation).

Figure 2 shows the AMD Instinct accelerator with its PCIe Gen 4 x16 link (16 GT/sec, at the bottom) that connects the GPU to (one of) the host processor(s). It also shows the three AMD Infinity Fabric ports that provide high-speed links (23 GT/sec, also at the bottom) to the other GPUs of the local hive as shown in Figure 1.

On the left and right of the floor plan, the High Bandwidth Memory (HBM) attaches via the GPU's memory controller. The MI100 generation of the AMD Instinct accelerator offers four stacks of HBM generation 2 (HBM2) for a total of 32GB with a 4,096 bit-wide memory interface. The peak memory bandwidth of the attached HBM2 is 1.228 TB/sec at a memory clock frequency of 1.2 GHz.

The execution units of the GPU are depicted in Figure 2 as Compute Units (CU). There are a total of 120 compute units that are physically organized into eight Shader Engines (SE) with fifteen compute units per shader engine. Each compute unit is further sub-divided into four SIMD units that process SIMD instructions of 16 data elements per instruction. This can enable the CU to process 64 floating point (FP64) operations (in a so-called 'wavefront') per clock cycle at a peak clock frequency of 1.5 GHz. Therefore, the theoretical maximum FP64 peak performance can be 11.5 TFLOPS:

4 [*SIMD units*] x 16 [*ops per cycle*] x 120 [*CU*] x 1.5 [*GHz*]

For 32-bit floating point data (FP32) the theoretical peak can be as high as 23 TFLOPS.

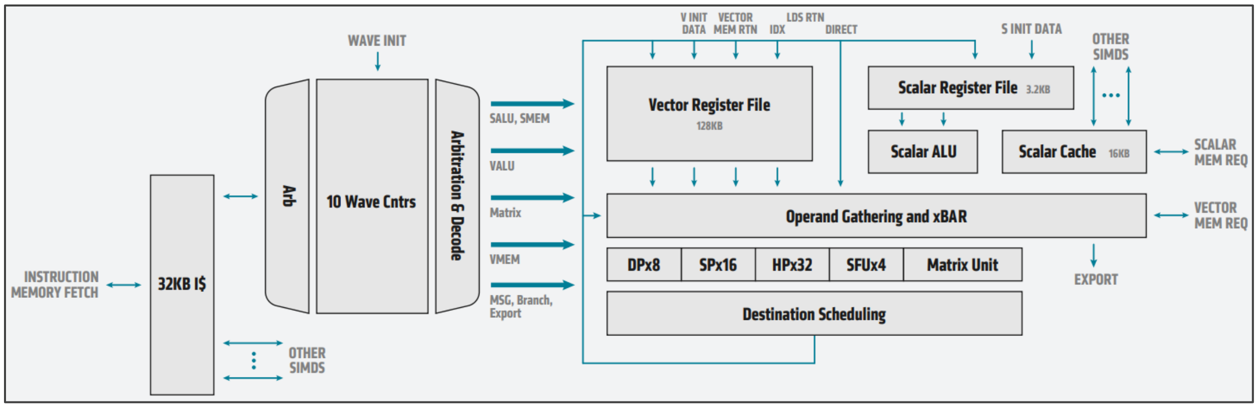


Figure 3: Block diagram of an MI100 compute unit with detailed SIMD view of the AMD CNDA architecture.

Figure 3 shows the block diagram of a single CU of an AMD Instinct™ MI100 accelerator and summarizes how instructions flow through the execution engines. The CU fetches the instructions via a 32KB instruction cache and moves them forward to execution via a dispatcher. The CU can handle up to ten wavefronts at a time and feed their instructions into the execution unit. The execution unit contains 256 vector general-purpose registers (VGPR) and 800 scalar general-purpose registers (SGPR). The VGPR and SGPR are dynamically allocated to the executing wavefronts. A wavefront can access a maximum of 102 scalar registers. Excess scalar-register usage will cause register spilling and thus may affect execution performance.

A wavefront can occupy any number of VGPRs from 0 to 256, directly affecting occupancy; that is, the number of concurrently active wavefronts in the CU. For instance, with 119 VPGRs used, only two wavefronts can be active in the CU at the same time. With the instruction latency of four cycles per SIMD instruction, the occupancy should be as high as possible such that the compute unit can improve execution efficiency by scheduling instructions from multiple wavefronts.

Table 1: Peak-performance capabilities of MI100 for different data types.

Computation	MI100 CU (FLOPS/CLOCK/CU)	MI100 GPU (Peak TFLOPS)
Matrix FP16	1,024	184.6
Matrix bfloat16	512	92.3
Matrix FP32	256	46.1
Vector FP32	128	23.1
Vector FP64	64	11.5

Table 1 summarizes the expected peak performance of the AMD Instinct accelerator for different data types and execution units. The middle column lists the expected peak performance of a single compute unit if a

SIMD (or matrix) instruction is being retired in each clock cycle. The third column lists the theoretical peak performance of the full GPU. The theoretical peak memory bandwidth of the GPU is 1.228 TB/sec. For measurements for the FP64 and memory performance, please see Chapter 8.

Chapter 3 System Settings

The following chapter reviews systems settings that are required to configure the system for AMD Instinct™ accelerators and to improve the system to obtain optimal performance of the GPUs. It is advised to configure the system for best possible host configuration according to the “High Performance Computing (HPC) Tuning Guide for AMD EPYC™ 7002 Series Processors” or “High Performance Computing (HPC) Tuning Guide for AMD EPYC™ 7003 Series Processors” depending on the processor generation of the system.

In addition, to the BIOS settings listed below the following settings (Section 3.1) will also have to be enacted via the command line (see Section 3.1.1):

- Core C states
- AMD-PCI-UTIL (on AMD EPYC™ 7002 series processors)
- IOMMU (if needed)

3.1 System BIOS Settings

For maximum expected MI100 GPU performance on systems with AMD EPYC™ 7002 series processors (codename ‘Rome’) and AMI System BIOS, the following configuration of System BIOS settings has been validated. These settings must be used for the qualification process and should be set as default values for the system BIOS. Analogous settings for other non-AMI System BIOS providers could be set similarly. For systems with Intel processors, some settings may not apply or be available as listed in Table 2.

Table 2: Recommended settings for the system BIOS in a GIGABYTE platform

BIOS Setting Location	Parameter	Value	Comments
Advanced **|- PCI Subsystem Settings**	Above 4G Decoding	Enabled	GPU Large BAR Support
|- AMD CBS **|- CPU Common Options** **|- Performance**	Global C-state Control CCD/Core/Thread Enablement |- SMT Control	Auto Accept Disable	Global Core C-States
|- DF Common Options **|- Memory Addressing**	NUMA nodes per socket Memory interleaving	NPS1,2,4 Auto	NUMA Nodes (NPS)
|- Link	4-link xGMI max speed 3-link xGMI max speed	18Gbps 18Gbps	Set AMD CPU xGMI speed to highest rate supported
|- NBIO Common Options	IOMMU	Disabled	

BIOS Setting Location	Parameter	Value	Comments 3
	PCIe Ten Bit Tag Support Preferred IO Preferred IO Bus Enhanced Preferred IO Mode	Enable Manual "Use lspci to find pci device id" Enable	
|- SMU Common Options	Determinism Control Determinism Slider cTDP Control cTDP Package Power Limit Control Package Power Limit xGMI Link Width Control xGMI Force Link Width xGMI Force Link Width Control APBDIS DF Cstates Fixed SOC Pstate	Manual Power Manual 240 Manual 240 Manual 2 Force 1 Auto P0	Set cTDP to 240W Set Package Power Limit to 240W Set AMD CPU xGMI width to 16 bits
|- UMC Common Options **|- DDR4 Common Options** **|-Enforce POR** **|- DRAM Controller Configuration** **|- DRAM Power Options**	|- Overclock |- Memory Clock Speed Power Down Enable	Accept Enabled 1600MHz Disabled	Set to max Memory Speed, if using 3200MHz DIMM's: 1600MHz. RAM Power Down

BIOS Setting Location	Parameter	Value	Comments
|- Security	TSME	Disabled	Memory Encryption

3.1.1 NBIO Link Clock Frequency

The NBIOs (4x per AMD EPYC™ processor) are the serializers/deserializers (also known as "SerDes") that convert and prepare the I/O signals for the processor's 128 external I/O interface lanes (32 per NBIO).

LCLK (short for link clock frequency) controls the link speed of the internal bus that connects the NBIO silicon with the data fabric. All data between the processor and its PCIe lanes flow to the data fabric based on these LCLK frequency settings. The link clock frequency of the NBIO components need to be forced to the maximum frequency for optimal PCIe performance.

For AMD EPYC™ 7002 series processors, this setting cannot be modified via configuration options in the server BIOS alone. Instead, the *AMD-IOPM-UTIL* (see Section 3.2.3) must be run at every server boot to disable Dynamic Power Management for all PCIe Root Complexes and NBIOs within the system and to lock the logic into the highest performance operational mode.

For AMD EPYC™ 7003 series processors, configuring all NBIOs to be in "Enhanced Preferred I/O" mode is sufficient to enable highest link clock frequency for the NBIO components.

3.1.2 Memory Configuration

For the memory addressing modes (see Table 2), especially the number of NUMA nodes per socket/processor (NPS), the recommended setting is to follow the guidance of the "High Performance Computing (HPC) Tuning Guide for AMD EPYC™ 7002 Series Processors" and "High Performance Computing (HPC) Tuning Guide for AMD EPYC™ 7003 Series Processors" to provide the optimal configuration for host side computation.

If the system is set to one NUMA domain per socket/processor (NPS1), bidirectional copy bandwidth between host memory and GPU memory may be slightly higher (up to about 16% more) than with four NUMA domains per socket/processor (NPS4). For memory bandwidth sensitive applications using MPI, NPS4 is recommended. For applications that are not optimized for NUMA locality, NPS1 is the recommended setting.

3.2 Operating System Settings

3.2.1 CPU Core State - ‘C States’

There are several Core-States, or C-states that an AMD EPYC CPU can idle within:

- **C0**: active. This is the active state while running an application.
- **C1**: idle
- **C2**: idle and power gated. This is a deeper sleep state and will have a greater latency when moving back to the C0 state, compared to when the CPU is coming out of C1.

Disabling C2 is important for running with a high performance, low-latency network. To disable power-gating on all cores run the following on Linux systems:

```
$ cpupower idle-set -d 2
```

Note that the `cpupower` tool must be installed, as it is not part of the base packages of most Linux® distributions. The package needed varies with the respective Linux distribution.

For Ubuntu

```
$ sudo apt install linux-tools-common
```

For CentOS/RHEL

```
$ sudo yum install cpupowerutils
```

For SLES

```
$ sudo zypper install cpupower
```

3.2.2 AMD-IOPM-UTIL

This section applies to AMD EPYC™ 7002 processors to optimize advanced Dynamic Power Management (DPM) in the I/O logic (see NBIO description above) for performance. Certain I/O workloads may benefit from disabling this power management. This utility disables DPM for all PCI-e root complexes in the system and locks the logic into the highest expected performance operational mode.

Disabling I/O DPM will reduce the latency and/or improve the throughput of low-bandwidth messages for PCI-e InfiniBand NICs and GPUs. Other workloads with low-bandwidth bursty PCI-e I/O characteristics may benefit as well if multiple such PCI-e devices are installed in the system.

The actions of the utility do not persist across reboots. There is no need to change any existing firmware settings when using this utility. The "Preferred I/O" and "Enhanced Preferred I/O" settings should remain unchanged at enabled.

The recommended method to use the utility is either to create a system start-up script, for example, a one-shot *systemd* service unit, or run the utility when starting up a job scheduler on the system. The installer packages (see Power Management Utility at *https://developer.amd.com/iopm-utility/*) will create and enable a *systemd* service unit for you. This service unit is configured to run in one-shot mode. This means that even when the service unit runs as expected, the status of the service unit will show inactive. This is the expected behavior when the utility runs normally. If the service unit shows failed, the utility did not run as expected. The output in either case can be shown with the `systemctl status` command.

NOTE: Stopping the service unit has no effect since the utility does not leave anything running. To undo the effects of the utility, disable the service unit with the `systemctl` disable command and reboot the system.

NOTE: The utility does not have any command-line options, and it must be run with super-user permissions.

3.2.3 Systems with 256 CPU Threads - IOMMU Configuration

For systems that have 256 logical CPU cores or more (e.g., 64-core AMD EPYC™ 7763 in a dual-socket configuration and SMT enabled), setting the IOMMU configuration to "disabled" can limit the number of available logical cores to 255. The reason is that the Linux® kernel disables X2APIC in this case and falls back to APIC, which can only enumerate a maximum of 255 (logical) cores.

If SMT is enabled by setting "CCD/Core/Thread Enablement > SMT Control" to "enable", the following steps can be applied to the system to enable all (logical) cores of the system:

- In the server BIOS, set IOMMU to "`Enabled`".
- When configuring the Grub boot loader, add the following arguments for the Linux kernel:
  `amd_iommu=on iommu=pt`
- Update Grub to use the modified configuration:

```
sudo grub2-mkconfig -o /boot/grub2/grub.cfg
```

- Reboot the system.
- Verify IOMMU passthrough mode by inspecting the kernel log via `dmesg`:

```
[...]
[    0.000000] Kernel command line: [...] amd_iommu=on iommu=pt
   [...]
```

Note, once the system is properly configured, the AMD ROCm platform can be installed.

Chapter 4 ROCm – System Management

4.1 AMD ROCm™ Overview

The AMD ROCm™ platform is a collection of software and middleware components (drivers, runtimes, libraries, and developer tools) for the AMD Instinct MI100 GPUs. The software projects are provided as separate packages that can be installed using the package managers of supported Linux distributions. This allows users to install only the packages they need. These packages will install most of the ROCm software into `/opt/rocm-x.y.z` (with `x.y.z` being a version string) by default. A symbolic link named `/opt/rocm` is also created and points to the installed `/opt/rocm-x.y.z` directory.

The AMD ROCm platform is designed to support the following operating systems:

- Ubuntu 20.04.2 HWE (5.4 and 5.6-oem) and 18.04.5 (Kernel 5.4)
- CentOS 7.9 (3.10.0-1127) and RHEL 7.9 (3.10.0-1160.6.1.el7) (using devtoolset-7 runtime support)
- CentOS 8.3 (4.18.0-193.el8) and RHEL 8.3 (4.18.0-193.1.1.el8) (devtoolset is not required)
- SLES 15 SP2

Some useful commands that come with the AMD ROCm™ platform are:

- `rocminfo`: Gives information about the installed and online AMD EYPC processors as well as the AMD Instinct GPUs. This tool will fail if the ROCm™ installation is not successful.

- `rocm-smi`: Queries details information about the AMD Instinct accelerators in the system, including firmware versions, fan speed, temperature, usage of compute units and memory. The tools are very useful to determine the correct installation and functioning of the GPU hardware and the ROCm software packages.

4.2 Installing ROCm

NOTE: If your systems are using a Mellanox ConnectX NIC, you must install Mellanox OFED before installing the AMD ROCm™ platform packages.

To make it easier to install ROCm, the AMD binary repositories provide several meta-packages that will automatically install multiple other packages. For example, `rocm-dkms` is the primary meta-package that is used to install most of the base components needed for ROCm to operate. It will build and install the *amdgpu* kernel driver, and another meta-package (`rocm-dev`) which installs most of the user-land ROCm core components, support software, and development tools.

The `rocm-utils` meta-package will install useful utilities that, while not required for the ROCm platform to operate, may still be beneficial to have. Finally, the `rocm-libs` meta-package will install some (but not all) of the libraries that are part of the software stack.

The URL *https://rocmdocs.amd.com/en/latest/Installation_Guide/Installation-Guide.html* describes how the ROCm software stack is installed.

Here is a summary of the steps to install the AMD ROCm™ platform for CentOS or RHEL 8.x. Please refer to the manuals for your Linux distribution for the command-line options of its respective package manager. The steps for the yum package manager are as follows:

1. Create the file `/etc/yum.repos.d/rocm.repo` file with the following contents:

```
[ROCm]
name=ROCm
baseurl=https://repo.radeon.com/rocm/yum/rpm
enabled=1
gpgcheck=1
gpgkey=https://repo.radeon.com/rocm/rocm.gpg.key
```

2. The issue the following commands to install the proper packages and reboot the system to load the kernel module from the boot image:
3. The issue the following commands to

```
$ sudo rpm -ivh epel-release-latest-8
$ sudo yum install -y epel-release
$ sudo yum install -y dkms kernel-headers-$(uname -r) \
                      kernel-devel-$(uname -r)
$ sudo yum install rocm-dkms && sudo reboot
```

After restarting the system, run the following command to verify that the ROCm installation is successful. The output will list the installed GPUs.

```
$ /opt/rocm/bin/rocminfo
```

The best option for an installation is to completely remove the previous version of the ROCm software stack. While most components of the ROCm software stack can co-exist, the device driver for AMD Instinct™ GPUs is bound to the latest software version that was installed on the system.

Note, that all users who want to access the GPU devices, will have to be in the group `video` and/or `render` depending on the Linux® distribution (see also Section 4.7.1). To add a user `doe` to the group, issue the following command:

```
$ sudo usermod -a -G video doe
```

The final step is to make the ROCm binaries to the search path for the shell. For this, add the bin directories of the ROCm platform to the PATH environment variable:

```
$ echo 'export PATH=$PATH:/opt/rocm/bin' \
    | sudo tee -a /etc/profile.d/rocm.sh
$ echo 'export PATH=$PATH:/opt/rocm/profiler/bin' \
    | sudo tee -a /etc/profile.d/rocm.sh
$ echo 'export PATH=$PATH:/opt/rocm/opencl/bin' \
    | sudo tee -a /etc/profile.d/rocm.sh
```

Finally, to uninstall the ROCm packages, use the following command:

```
$ sudo yum autoremove rocm-opencl rocm-dkms rock-dkms
```

To validate the ROCm installation, the ROCm validation suite (RVS) can be used.

4.3 Troubleshooting the ROCm Installation

The following error example may be encountered during installation:

```
$ /opt/rocm/bin/rocminfo
ROCk module is loaded
Unable to open /dev/kfd read-write: Cannot allocate memory
```

Below are potential reasons that you may have encountered this error when running `rocminfo` and calling other ROCm applications in the stack. The issues expose themselves usually through an error message that is coming from the `rocminfo` tool, such as the one above. Some more potential reasons are the following.

4.3.1 Unsupported GPU(s)

To determine if the installed GPU device is supported by the ROCm platform, please check the following link to find the list of supported devices: *https://github.com/RadeonOpenCompute/ROCm#supported-gpus*.

4.3.2 Unsupported CPU(s)

AMD Instinct™ accelerators require certain features from the host system to function correctly. Please check *https://github.com/RadeonOpenCompute/ROCm#supported-cpus* to check if the host processor is listed as a supported processor.

4.3.2.1 Group and GPU ownership not properly set

The current user has not been assigned to the `video` and/or the `render` group, depending on the Linux distribution. Alternatively, a custom group can be assigned as outlined in Section 4.7.1 of this document.

Confirm which groups the active user is assigned:

```
$ id $LOGNAME
uid=2110(doe) gid=2110(doe) groups=2110(doe) 3000(video)
```

If `video`, `render`, or the custom group for accessing the GPU devices is not listed then the active user cannot access the GPU devices as these are assigned ownership to one of these groups depending on configuration or OS.

4.3.2.2 Mismatching package versions for kernel packages

The installation of ROCm builds the Linux® kernel module for the AMD Instinct™ accelerator devices. Thus, the kernel headers must be installed and have to match the version of the installed and running kernel images. In the example below, the minor kernel version is active, while a different version of the kernel development package has been installed:

```
$ rpm -qa | grep ^kernel
kernel-3.10.0-1160.21.1.el7.x86_64
kernel-tools-3.10.0-1160.21.1.el7.x86_64
kernel-headers-3.10.0-1160.21.1.el7.x86_64
kernel-devel-3.10.0-1160.el7.x86_64
kernel-tools-libs-3.10.0-1160.21.1.el7.x86_64
```

Check that the running kernel, the respective development packages and the header packages are all installed in the same version.

If the versioning is different the resolution is to install the development (minor) version to match the running kernel and header versions listed for the active kernel. Proceed to uninstall the ROCm packages as guided above and reinstall it using the installation guide above.

NOTE: If you have updated using a package manager, make sure to reboot before running the installation of ROCm platform.

4.4 RVS - ROCm Validation Suite

The ROCm Validation Suite (RVS) is a system administrator’s and cluster manager's tool for detecting and troubleshooting common problems affecting AMD GPU(s) running in a high-performance computing environment.

It is a collection of tests, benchmarks, and qualification tools each targeting a specific sub-system of the ROCm platform. The RVS can be obtained by building it from the source code base or by installing from pre-built package.

To install via the package manager, use the following RHEL command (similar for other Linux® distributions):

```
$ sudo yum install rocm-validation-suite
```

The source code can be obtained at *https://github.com/ROCm-Developer-Tools/ROCmValidationSuite.*

To install from a package, you can download rocm-validation-suite-3.10.0.deb or .rpm file from AMD site. Then install the package using your favorite package manager. RVS components are installed in `/opt/rocm/rvs`.

To add the RVS binaries to the PATH environment variable, issue the following command:

```
$ echo 'export PATH=$PATH:/opt/rocm/rvs/' \
    | sudo tee -a /etc/profile.d/rocm.sh
```

To run the complete RVS test suite, use the command:

```
$ sudo /opt/rocm/rvs/testscripts/rvsqa.new.sh
```

A description of the tests therein now follows.

4.4.1 RVS Architecture

RVS is implemented as a set of modules each implementing a subset of the test functionality. Modules are invoked from one central place (aka the “Launcher”) that is responsible for reading the input (command line and test configuration file), loading, and running appropriate modules, and providing test output and report.

The available modules are as follows.

4.4.1.1 GPU Properties – GPUP

The GPU Properties module queries the configuration of a target device and returns the device’s static characteristics. These static values can be used to debug issues such as device support, performance and firmware problems.

4.4.1.2 GPU Monitor – GM module

The GPU monitor tool can run on one, some, or all the GPU(s) installed and will report various information at regular intervals. The module can be configured to halt another RVS module execution if one of the quantities exceeds a specified boundary value.

4.4.1.3 PCI Express State Monitor – PESM module

The PCIe State Monitor tool is used to actively monitor the PCIe interconnect between the host platform and the GPU. The module will register a “listener” on a target GPU’s PCIe interconnect and log a message whenever it detects a state change. The PESM can detect the following state changes:

- PCIe link speed
- GPU power states.

4.4.1.4 ROCm Configuration Qualification Tool – RCQT module

The ROCm Configuration Qualification Tool ensures the platform can run ROCm applications and is configured correctly. It checks the installed versions of the ROCm components and the platform configuration of the system. This includes checking the dependencies, corresponding to the associated operating system and run time environment, are installed correctly. Other qualification steps include checking:

- The existence of the `/dev/kfd` device
- The `/dev/kfd` device's permissions
- The existence of all required users and groups that support ROCm
- That the user mode components are compatible with the drivers, both the KFD and the amdgpu driver
- The configuration of the runtime linker/loader qualifying that all ROCm libraries are in the correct search path

4.4.1.5 PCI Express Qualification Tool – PEQT module

The PCIe Qualification Tool is used to qualify the PCIe bus on which the GPU is connected. The qualification test can determine the following characteristics of the PCIe bus interconnect to a GPU:

- Support for Gen 3 atomic completers
- DMA transfer statistics
- PCIe link speed
- PCIe link width

4.4.1.6 SBIOS Mapping Qualification Tool – SMQT module

The GPU SBIOS mapping qualification tool is designed to verify that a platform's SBIOS has satisfied the BAR mapping requirements for VDI and Radeon Instinct products for ROCm support. Please, refer to the "*ROCm Use of Advanced PCIe Features and Overview of How BAR Memory is Used In ROCm Enabled System*". For more information about how BAR memory is initialized by VDI and Radeon products, refer to *https://github.com/RadeonOpenCompute/ROCm_Documentation/blob/master/GCN_ISA_Manuals/PCIe-features.rst*

4.4.1.7 P2P Benchmark and Qualification Tool – PBQT module

The P2P Benchmark and Qualification Tool is designed to provide the list of all GPUs that support P2P and characterize the P2P links between peers. In addition to testing for P2P compatibility, this test will perform a peer-to-peer throughput test between all P2P pairs for performance evaluation. The P2P Benchmark and Qualification Tool will allow users to pick a collection of two or more GPUs on which to run. The user will also be able to select whether they want to run the throughput test on each of the pairs.

4.4.1.8 PCI Express Bandwidth Benchmark – PEBB module

The PCIe Bandwidth Benchmark attempts to saturate the PCIe bus with DMA transfers between system memory and a target GPU card's memory. The maximum bandwidth obtained is reported to help debug low bandwidth issues. The benchmark should be capable of targeting one, some or all the GPUs installed in a platform, reporting individual benchmark statistics for each.

4.4.1.9 GPU Stress Test - GST module

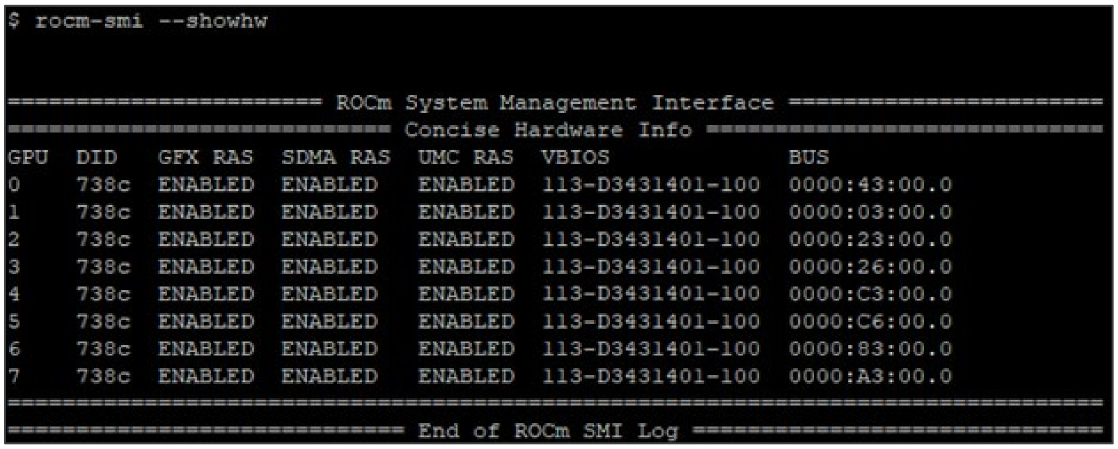
The GPU Stress Test runs a Graphics Stress test or SGEMM/DGEMM (Single/Double-precision General Matrix Multiplication) workload on one, some, or all GPUs. The GPUs can be of the same or different types. The duration of the benchmark is configurable, both in terms of time (how long to run) and iterations (how many times to run). The test is capable of driving the power level equivalent to the rated TDP of the card, or levels below that. The tool is capable of driving cards at TDP-50% to TDP-100%, in 10% incremental jumps. These settings are controllable by the user.

4.4.1.10 Input EDPp Test - IET module

The Input EDPp Test generates EDP peak power on all input rails. This test is used to verify if the system PSU can handle the worst-case power spikes of the board. Peak Current at defined period = 1 minute moving average power.

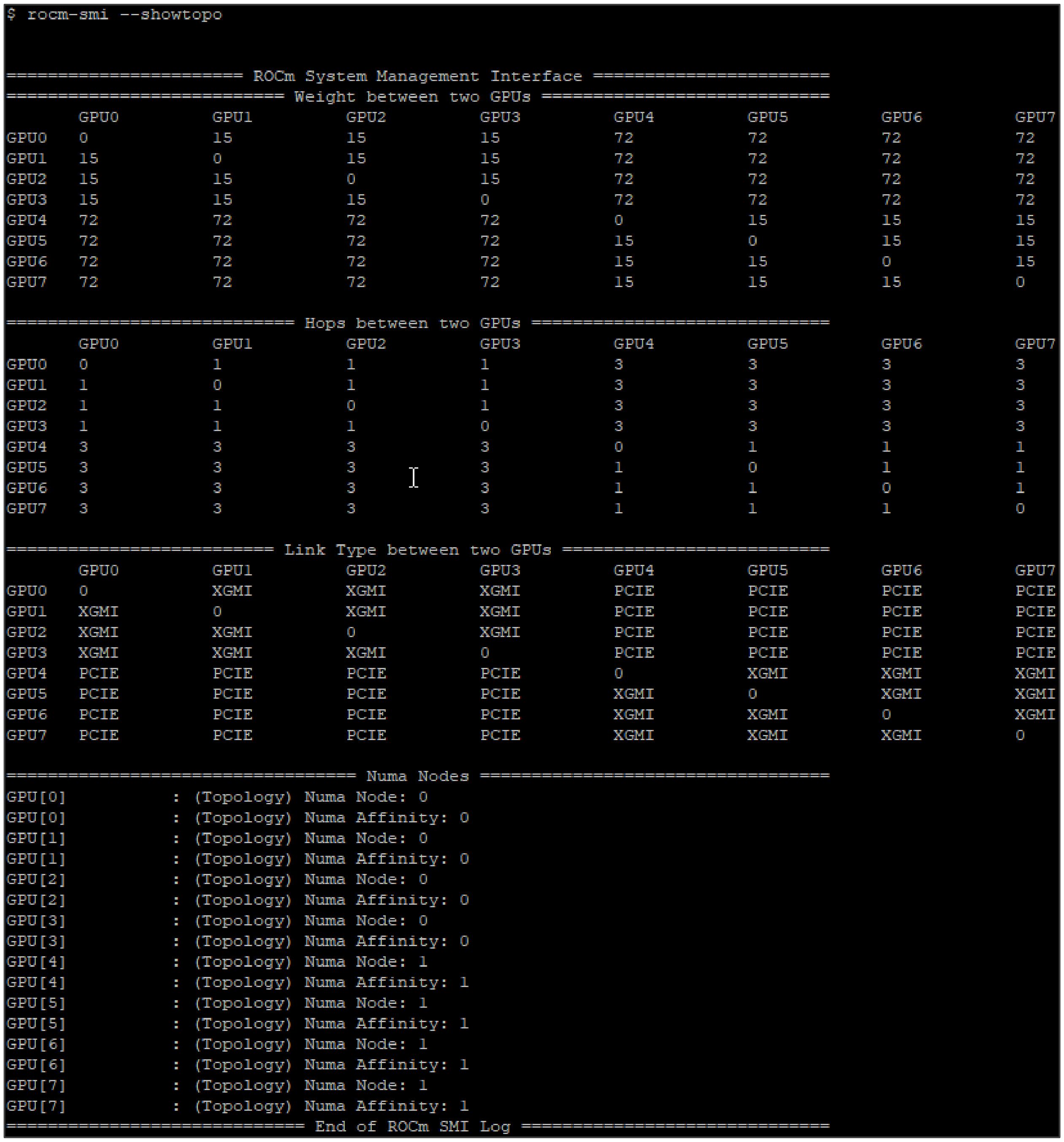
4.5 Hardware Verification with ROCm

The AMD ROCm™ platform ships with tools to query the system structure. To query the GPU hardware, the `rocm-smi` command is available. It can show available GPUs in the system with their device ID and their respective firmware (or VBIOS) versions:



```
$ rocm-smi --showhw


======================= ROCm System Management Interface ========================
============================ Concise Hardware Info =============================
GPU  DID   GFX RAS  SDMA RAS  UMC RAS  VBIOS            BUS
0    738c  ENABLED  ENABLED   ENABLED  113-D3431401-100  0000:43:00.0
1    738c  ENABLED  ENABLED   ENABLED  113-D3431401-100  0000:03:00.0
2    738c  ENABLED  ENABLED   ENABLED  113-D3431401-100  0000:23:00.0
3    738c  ENABLED  ENABLED   ENABLED  113-D3431401-100  0000:26:00.0
4    738c  ENABLED  ENABLED   ENABLED  113-D3431401-100  0000:C3:00.0
5    738c  ENABLED  ENABLED   ENABLED  113-D3431401-100  0000:C6:00.0
6    738c  ENABLED  ENABLED   ENABLED  113-D3431401-100  0000:83:00.0
7    738c  ENABLED  ENABLED   ENABLED  113-D3431401-100  0000:A3:00.0
=================================================================================
============================= End of ROCm SMI Log ==============================
```

Another important query is to show the system structure, the localization of the GPUs in the system, and the fabric connections between the system components:



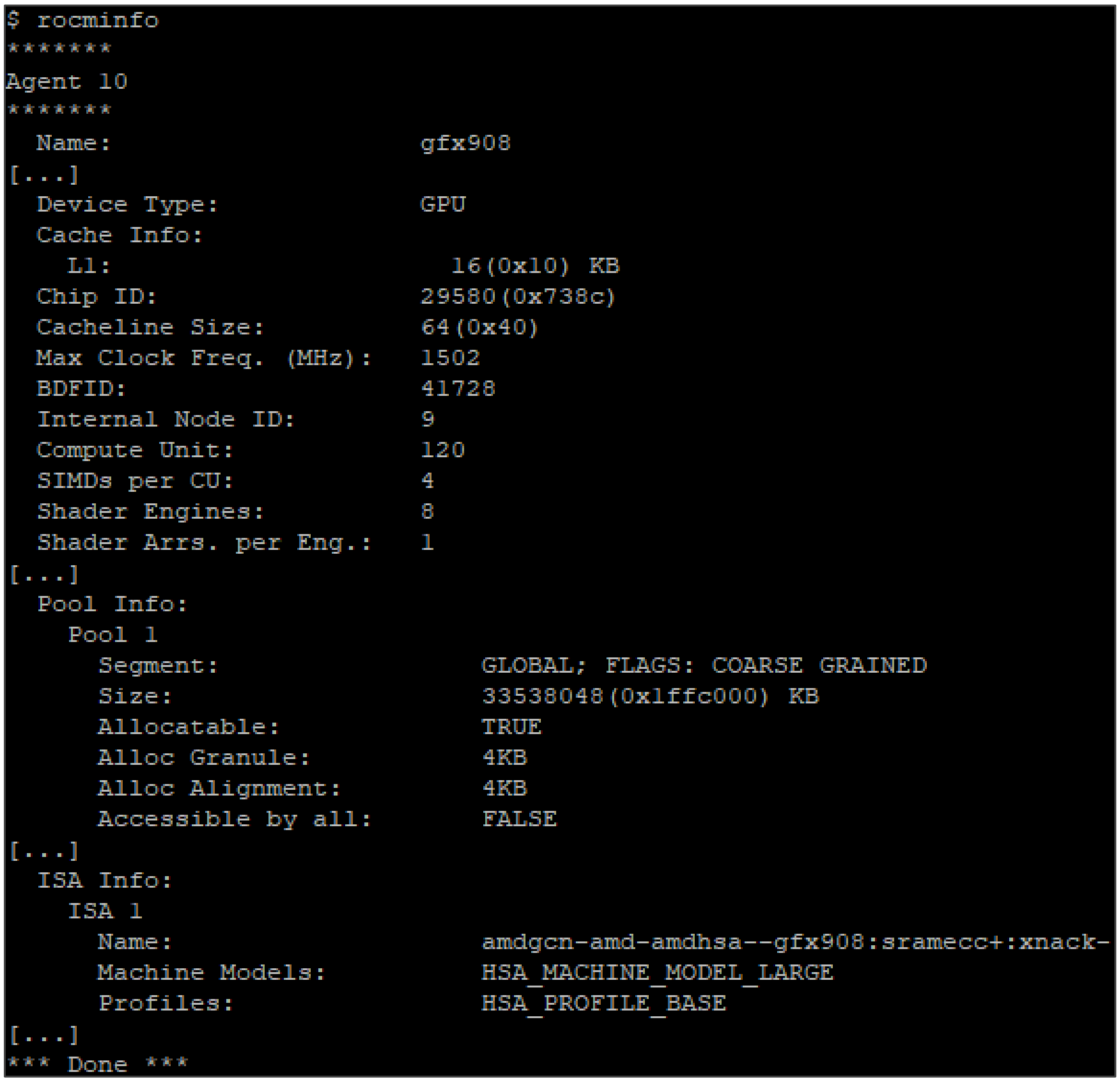
```
$ rocm-smi --showtopo


======================= ROCm System Management Interface =======================
=========================== Weight between two GPUs ============================
       GPU0         GPU1         GPU2         GPU3         GPU4         GPU5         GPU6         GPU7
GPU0   0            15           15           15           72           72           72           72
GPU1   15           0            15           15           72           72           72           72
GPU2   15           15           0            15           72           72           72           72
GPU3   15           15           15           0            72           72           72           72
GPU4   72           72           72           72           0            15           15           15
GPU5   72           72           72           72           15           0            15           15
GPU6   72           72           72           72           15           15           0            15
GPU7   72           72           72           72           15           15           15           0

============================ Hops between two GPUs =============================
       GPU0         GPU1         GPU2         GPU3         GPU4         GPU5         GPU6         GPU7
GPU0   0            1            1            1            3            3            3            3
GPU1   1            0            1            1            3            3            3            3
GPU2   1            1            0            1            3            3            3            3
GPU3   1            1            1            0            3            3            3            3
GPU4   3            3            3            3            0            1            1            1
GPU5   3            3            3            3            1            0            1            1
GPU6   3            3            3            3            1            1            0            1
GPU7   3            3            3            3            1            1            1            0

========================== Link Type between two GPUs ==========================
       GPU0         GPU1         GPU2         GPU3         GPU4         GPU5         GPU6         GPU7
GPU0   0            XGMI         XGMI         XGMI         PCIE         PCIE         PCIE         PCIE
GPU1   XGMI         0            XGMI         XGMI         PCIE         PCIE         PCIE         PCIE
GPU2   XGMI         XGMI         0            XGMI         PCIE         PCIE         PCIE         PCIE
GPU3   XGMI         XGMI         XGMI         0            PCIE         PCIE         PCIE         PCIE
GPU4   PCIE         PCIE         PCIE         PCIE         0            XGMI         XGMI         XGMI
GPU5   PCIE         PCIE         PCIE         PCIE         XGMI         0            XGMI         XGMI
GPU6   PCIE         PCIE         PCIE         PCIE         XGMI         XGMI         0            XGMI
GPU7   PCIE         PCIE         PCIE         PCIE         XGMI         XGMI         XGMI         0

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

The previous command shows the system structure in four blocks:

- The first block of the output shows the distance between the GPUs similar to what the `numactl` command outputs for the NUMA domains of a system. The weight is a qualitative measure for the "distance" data must travel to reach one GPU from another one. While the values do not carry a special (physical) meaning, the higher the value the more hops are needed to reach the destination from the source GPU.
- The second block has a matrix for the number of hops required to send data from one GPU to another. For the GPUs in the local hive, this number is one, while for the others it is three (one hop to leave the hive, one hop across the processors, and one hop within the destination hive).

- The third block outputs the link types between the GPUs. This can either be “XGMI” for AMD Infinity Fabric™ links or “PCIE” for PCIe Gen4 links.
- The fourth block reveals the localization of a GPU with respect to the NUMA organization of the shared memory of the AMD EPYC™ processors.

To query the compute capabilities of the GPU devices, the `rocminfo` command is available with the AMD ROCm™ platform. It lists specific details about the GPU devices, including but not limited to the number of compute units, width of the SIMD pipelines, memory information, and instruction set architecture:



```
$ rocminfo
*******
Agent 10
*******
  Name:                      gfx908
[...]
  Device Type:               GPU
  Cache Info:
    L1:                        16(0x10) KB
  Chip ID:                   29580(0x738c)
  Cacheline Size:            64(0x40)
  Max Clock Freq. (MHz):     1502
  BDFID:                     41728
  Internal Node ID:          9
  Compute Unit:              120
  SIMDs per CU:              4
  Shader Engines:            8
  Shader Arrs. per Eng.:     1
[...]
  Pool Info:
    Pool 1
      Segment:                   GLOBAL; FLAGS: COARSE GRAINED
      Size:                      33538048(0x1ffc000) KB
      Allocatable:               TRUE
      Alloc Granule:             4KB
      Alloc Alignment:           4KB
      Accessible by all:         FALSE
[...]
  ISA Info:
    ISA 1
      Name:                      amdgcn-amd-amdhsa--gfx908:sramecc+:xnack-
      Machine Models:            HSA_MACHINE_MODEL_LARGE
      Profiles:                  HSA_PROFILE_BASE
[...]
*** Done ***
```

Table 3 lists the architecture names for the AMD Instinct™ accelerators.

Table 3: Architecture names of the AMD Instinct accelerators.

Architecture Name	AMD Instinct Accelerator
gfx906	AMD Radeon Instinct™ MI50 Accelerator
gfx908	AMD Instinct™ MI100 Accelerator

4.6 Testing Inter-device Bandwidth

Section 4.5 showed the `rocm-smi --showtopo` command to show how the system structure and how the GPUs are located and connected in this structure. For more details, the `rocm-bandwidth-test` can run benchmarks to show the effective link bandwidth between the components of the system.

The ROCm Bandwidth Test program can be installed with the following package-manager command (for systems with other package managers, please use the appropriate command):

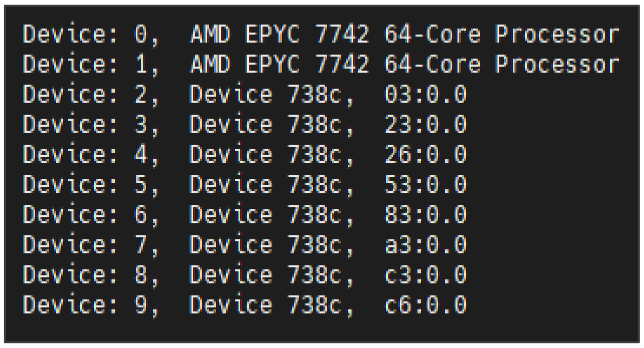
```
$ sudo yum install rocm-bandwidth-test
```

Alternatively, the source code can be downloaded at
https://github.com/RadeonOpenCompute/rocm_bandwidth_test

To compile and run the code, follow these steps:

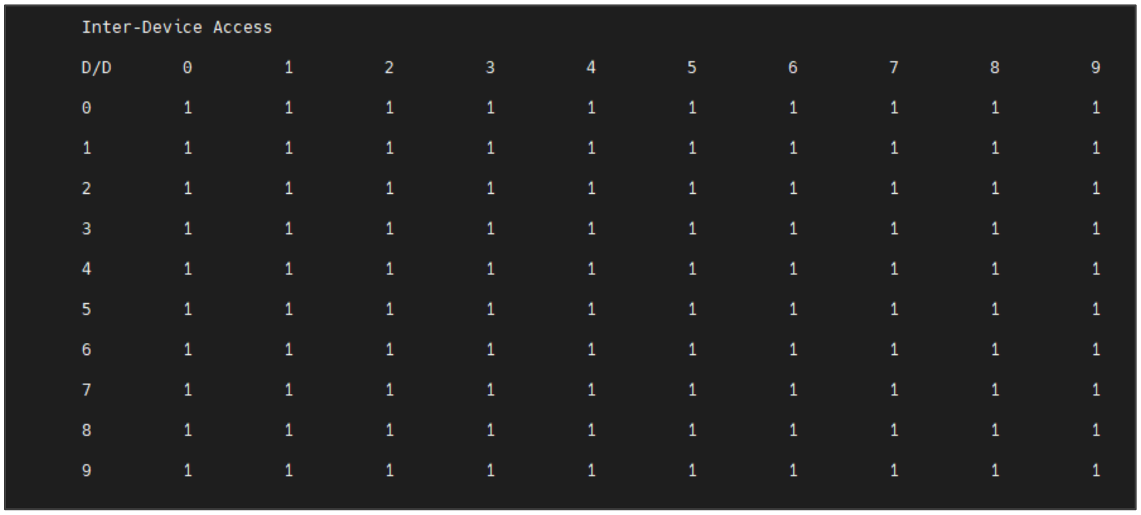
```
$ git clone https://github.com/RadeonOpenCompute/rocm_bandwidth_test
$ cd rocm_bandwidth_test
$ cmake .
$ make
$ ./rocm-bandwidth-test
```

The output will list the available compute devices (CPUs and GPUs):

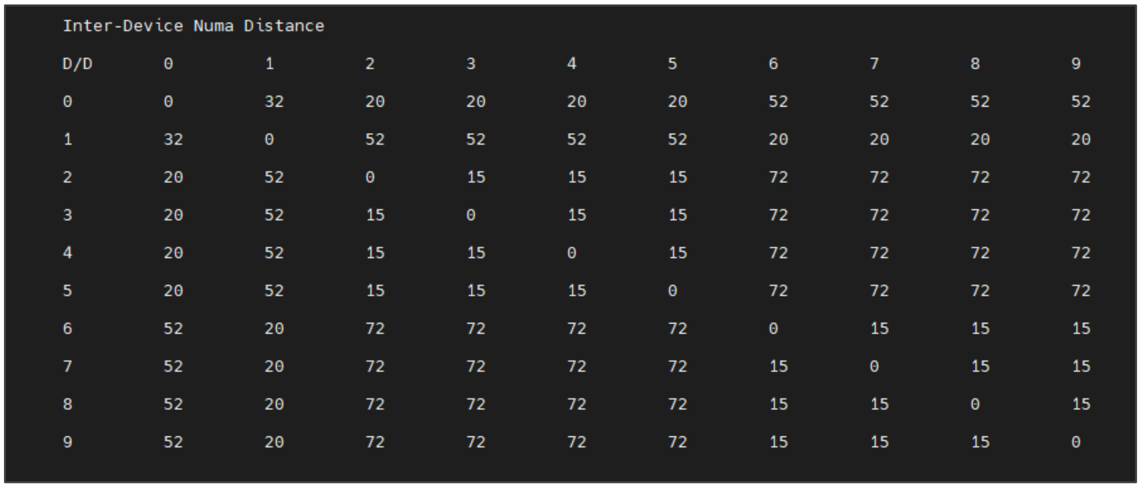


```
Device: 0,  AMD EPYC 7742 64-Core Processor
Device: 1,  AMD EPYC 7742 64-Core Processor
Device: 2,  Device 738c,  03:0.0
Device: 3,  Device 738c,  23:0.0
Device: 4,  Device 738c,  26:0.0
Device: 5,  Device 738c,  53:0.0
Device: 6,  Device 738c,  83:0.0
Device: 7,  Device 738c,  a3:0.0
Device: 8,  Device 738c,  c3:0.0
Device: 9,  Device 738c,  c6:0.0
```

The output will also show a matrix that contains a "1" if a device can communicate to another device (CPU and GPU) of the system and it will show the NUMA distance (similar to `rocm-smi`):

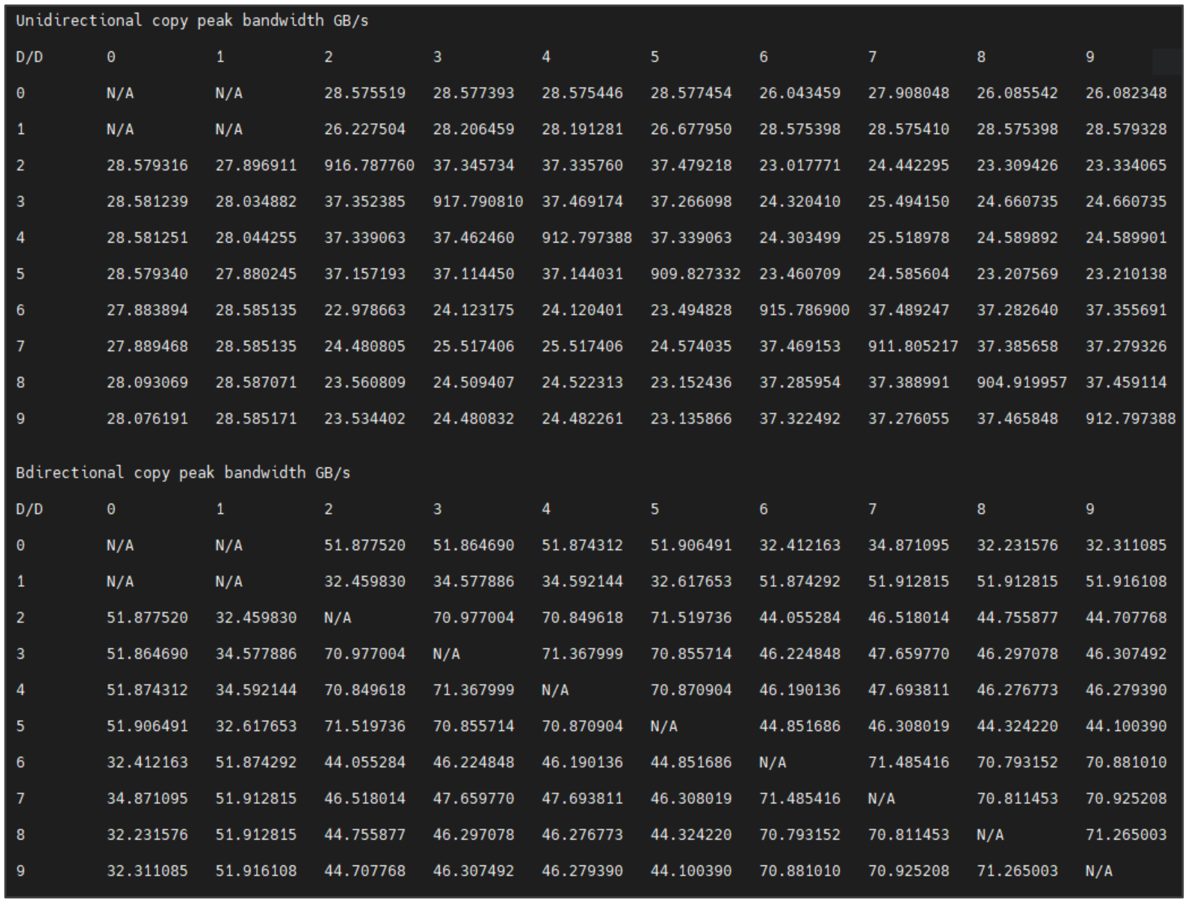


```
    Inter-Device Access
    D/D     0       1       2       3       4       5       6       7       8       9
    0       1       1       1       1       1       1       1       1       1       1
    1       1       1       1       1       1       1       1       1       1       1
    2       1       1       1       1       1       1       1       1       1       1
    3       1       1       1       1       1       1       1       1       1       1
    4       1       1       1       1       1       1       1       1       1       1
    5       1       1       1       1       1       1       1       1       1       1
    6       1       1       1       1       1       1       1       1       1       1
    7       1       1       1       1       1       1       1       1       1       1
    8       1       1       1       1       1       1       1       1       1       1
    9       1       1       1       1       1       1       1       1       1       1
```



```
    Inter-Device Numa Distance
    D/D     0       1       2       3       4       5       6       7       8       9
    0       0       32      20      20      20      20      52      52      52      52
    1       32      0       52      52      52      52      20      20      20      20
    2       20      52      0       15      15      15      72      72      72      72
    3       20      52      15      0       15      15      72      72      72      72
    4       20      52      15      15      0       15      72      72      72      72
    5       20      52      15      15      15      0       72      72      72      72
    6       52      20      72      72      72      72      0       15      15      15
    7       52      20      72      72      72      72      15      0       15      15
    8       52      20      72      72      72      72      15      15      0       15
    9       52      20      72      72      72      72      15      15      15      0
```

The output also contains the measured bandwidth for unidirectional and bidirectional transfers between the devices (CPU and GPU):



```
Unidirectional copy peak bandwidth GB/s
D/D     0           1           2           3           4           5           6           7           8           9
0       N/A         N/A         28.575519   28.577393   28.575446   28.577454   26.043459   27.908048   26.085542   26.082348
1       N/A         N/A         26.227504   28.206459   28.191281   26.677950   28.575398   28.575410   28.575398   28.579328
2       28.579316   27.896911   916.787760  37.345734   37.335760   37.479218   23.017771   24.442295   23.309426   23.334065
3       28.581239   28.034882   37.352385   917.790810  37.469174   37.266098   24.320410   25.494150   24.660735   24.660735
4       28.581251   28.044255   37.339063   37.462460   912.797388  37.339063   24.303499   25.518978   24.589892   24.589901
5       28.579340   27.880245   37.157193   37.114450   37.144031   909.827332  23.460709   24.585604   23.207569   23.210138
6       27.883894   28.585135   22.978663   24.123175   24.120401   23.494828   915.786900  37.489247   37.282640   37.355691
7       27.889468   28.585135   24.480805   25.517406   25.517406   24.574035   37.469153   911.805217  37.385658   37.279326
8       28.093069   28.587071   23.560809   24.509407   24.522313   23.152436   37.285954   37.388991   904.919957  37.459114
9       28.076191   28.585171   23.534402   24.480832   24.482261   23.135866   37.322492   37.276055   37.465848   912.797388

Bdirectional copy peak bandwidth GB/s
D/D     0           1           2           3           4           5           6           7           8           9
0       N/A         N/A         51.877520   51.864690   51.874312   51.906491   32.412163   34.871095   32.231576   32.311085
1       N/A         N/A         32.459830   34.577886   34.592144   32.617653   51.874292   51.912815   51.912815   51.916108
2       51.877520   32.459830   N/A         70.977004   70.849618   71.519736   44.055284   46.518014   44.755877   44.707768
3       51.864690   34.577886   70.977004   N/A         71.367999   70.855714   46.224848   47.659770   46.297078   46.307492
4       51.874312   34.592144   70.849618   71.367999   N/A         70.870904   46.190136   47.693811   46.276773   46.279390
5       51.906491   32.617653   71.519736   70.855714   70.870904   N/A         44.851686   46.308019   44.324220   44.100390
6       32.412163   51.874292   44.055284   46.224848   46.190136   44.851686   N/A         71.485416   70.793152   70.881010
7       34.871095   51.912815   46.518014   47.659770   47.693811   46.308019   71.485416   N/A         70.811453   70.925208
8       32.231576   51.912815   44.755877   46.297078   46.276773   44.324220   70.793152   70.811453   N/A         71.265003
9       32.311085   51.916108   44.707768   46.307492   46.279390   44.100390   70.881010   70.925208   71.265003   N/A
```

4.7 Advanced Configuration Options

The following section will describe some more advanced installation options that are not documented as part of the general software installation notes for ROCm.

4.7.1 Changing Ownership of GPU Devices

In a typical installation, the pseudo devices to access the installed GPUs are owned by the `video` and/or the `render` group as shown here:

```
$ ls -l /dev/kfd /dev/dri
0 crw-rw---- 1 root video 239, 0 Mar 24 14:28 /dev/kfd

/dev/dri:
total 0
0 drwxr-xr-x 2 root root        100 Mar 24 14:28 by-path/
0 crw-rw---- 1 root video  226,   0 Mar 24 14:28 card0
0 crw-rw---- 1 root video  226,   1 Mar 24 14:28 card1
0 crw-rw---- 1 root render 226, 128 Mar 24 14:28 renderD128
```

The default user and group ownership of these files is defined by the *udev* (short for "userspace /dev") framework that is used by most contemporary Linux distributions. Since the files are only accessible by members of the `video` group, all users who need access to the compute resources of an AMD Radeon™ Instinct™ device will have to be part of that group to gain access to the GPU devices.

If this default configuration is not suitable, the default user and group ownership of the pseudo devices can be modified to accommodate the needs of the system. This is done by writing a rule definition that modifies the ownership accordingly.

The first step is to create a new file `99-amdpgu.rules` and place it in `/etc/udev/rules.d`:

```
$ cat /etc/udev/rules.d/99-amdgpu.rules
SUBSYSTEM=="drm", GROUP="new_group", MODE="0660"
SUBSYSTEM=="kfd", GROUP="new_group", MODE="0660"
```

When the driver for the AMD Radeon™ Instinct™ GPU is loaded into the Linux kernel (e.g., when booting the system), the *udev* system will update the ownership automatically. An update can also be triggered manually with the following command:

```
$ sudo udevadm trigger -s drm; sudo udevadm trigger -s kfd
```

In either case, the ownership of the pseudo-devices should now be changed:

```
$ ls -l /dev/kfd /dev/dri
0 crw-rw---- 1 root new_group 239, 0 Apr  7 09:01 /dev/kfd

/dev/dri:
total 0
0 drwxr-xr-x 2 root root      100 Mar 24 14:28 by-path/
0 crw-rw---- 1 root new_group 226,   0 Apr  7 09:01 card0
0 crw-rw---- 1 root new_group 226,   1 Apr  7 09:01 card1
0 crw-rw---- 1 root new_group 226, 128 Apr  7 09:01 renderD128
```

Note that the resolution of the user and group names is done when the *udev* system is started at boot time. This is typically before services like LDAP are starting up and thus before the Linux system can resolve names to numerical IDs. If the desired user or group to own the pseudo devices is coming from an LDAP domain, then the numerical user or group ID must be used to not cause timeouts during boot.

4.7.2 Freeze the Version of ROCm Software Stack

The instructions for the ROCm installation at *https://rocmdocs.amd.com/en/latest/Installation_Guide/Installation-Guide.html* assume that the ROCm version of the system is always the latest that has been released via the repositories of the package managers. In case the system must stay on a particular ROCm version, the following steps need to be modified to restrict the package manager to the desired version of ROCm.

Following the installation instructions for ROCm software packages, the instructions configure a new package repository that points to *https://repo.radeon.com/rocm/apt/debian/* (for Ubuntu), *https://repo.radeon.com/rocm/yum/rpm* (for CentOS), or *https://repo.radeon.com/rocm/zyp/zypper/* (for SLES). These URLs correspond to the latest release of the ROCm platform and therefore will always pull the latest software packages that are available to the package manager.

The following tables Table 4 through Table 7 list the alternative URLs that provide a specific release version of the ROCm software stack. These URLs can be substituted for the above URLs in the repository configuration.

Table 4: URLs for the ROCm-platform packages for CentOS 8.

Version	URL
4.0	*https://repo.radeon.com/rocm/centos8/4.0*
4.0.1	*https://repo.radeon.com/rocm/centos8/4.0.1*
4.1	*https://repo.radeon.com/rocm/centos8/4.1*
4.1.1	*https://repo.radeon.com/rocm/centos8/4.1.1*
4.2	*https://repo.radeon.com/rocm/centos8/4.2*
latest	*https://repo.radeon.com/rocm/centos8/rpm*

Table 5: URLs for the ROCm-platform packages for CentOS 7.

Version	URL
4.0	*https://repo.radeon.com/rocm/yum/4.0*
4.0.1	*https://repo.radeon.com/rocm/yum/4.0.1*
4.1	*https://repo.radeon.com/rocm/yum/4.1*
4.1.1	*https://repo.radeon.com/rocm/yum/4.1.1*
4.2	*https://repo.radeon.com/rocm/yum/4.2*
latest	*https://repo.radeon.com/rocm/yum/rpm*

Table 6: URLs for the ROCm-platform packages for SLES 12.

Version	URL
4.0	*https://repo.radeon.com/rocm/zyp/4.0*
4.0.1	*https://repo.radeon.com/rocm/zyp/4.0.1*
4.1	*https://repo.radeon.com/rocm/zyp/4.1*
4.1.1	*https://repo.radeon.com/rocm/zyp/4.1.1*
4.2	*https://repo.radeon.com/rocm/zyp/4.2*
latest	*https://repo.radeon.com/rocm/zyp/zypper*

Table 7: URLs for the ROCm-platform packages for Ubuntu.

Version	URL
4.0	*https://repo.radeon.com/rocm/apt/4.0*
4.0.1	*https://repo.radeon.com/rocm/apt/4.0.1*
4.1	*https://repo.radeon.com/rocm/apt/4.1*
4.1.1	*https://repo.radeon.com/rocm/apt/4.1.1*
4.2	*https://repo.radeon.com/rocm/apt/4.2*
latest	*https://repo.radeon.com/rocm/apt/debian*

The following example shows the Yum configuration for a ROCm installation that has been pinned to version 4.0:

```
$ cat /etc/yum.repos.d/rocm.repo
[ROCm]
name=ROCm
baseurl=https://repo.radeon.com/rocm/centos8/4.0
enabled=1
gpgcheck=1
gpgkey=https://repo.radeon.com/rocm/rocm.gpg.key
```

Chapter 5 ROCm - Libraries and Compilers

In addition to system tools the ROCm stack also contains the software development environment for AMD Instinct GPUs. This chapter will provide a brief review of the compilers and most important libraries that ship with the AMD ROCm™ platform. Please refer to the respective URLs for further information and usage examples and to Chapter 4 for installation notes and how to install individual packages.

Table 8: Important components of the AMD ROCM software stack.

Tools	rocgdb	rocprof roctracer	hipify-perl hipify-clang	rocminfo rocm-smi RVS
Compilers	hipcc hipfort	clang clang-cl	flang	clang++
Libraries	hipBLAS/rocBLAS hipSPARSE/rocSPARSE rocSOLVER rocALUTION	MIOpen	rocTHRUST RCCL	hipFFT/rocFFT rocRAND

Table 8 summarizes the key tools, compilers, and libraries that ship with the AMD ROCm platform. Some tools have been illustrated earlier in this document.

The ROCm ecosystem allows you to write portable code that can be run on different GPU platforms. The following paragraphs will describe ways to offload computation to AMD GPUs.

5.1 ROCmCC and AOMP

The AMD ROCm™ platform ships with a compiler ("ROCmCC") that supports offloading computation to AMD Instinct™ accelerators via the HIP language and OpenMP Application Programming Interface (API). The compiler is installed via the `llvm-amdgpu` package and resides in `/opt/rocm/llvm`. The following tables list the most important compiler flags that can be used to compile HIP and OpenMP offload code for AMD Instinct accelerators.

The AOMP compiler is a separate package that can be downloaded from *https://github.com/ROCm-Developer-Tools/aomp*. It is an OpenMP research compiler that AMD uses to prototype new OpenMP features for ROCmCC. AOMP and ROCmCC accept the same command line options (except where explicitly noted) and thus AOMP can serve as a replacement for the clang(-cl), clang++, and flang compilers listed in Table 8.

Table 9: General compiler options for the ROCmCC and AOMP compilers.

General Options	Compiler Option
Enable OpenMP Support	-fopenmp
Select GPU Target (AOMP 13.0-3 or later, see **Table 3**)	--offload-arch=*arch* with *arch* either gfx906 or gfx908
Select GPU target	-fopenmp-targets=amdgcn-amd-amdhsa
Select GPU target	-Xopenmp-target=amdgcn-amd-amdhsa
Select the target GPU architecture (see **Table 3**)	-march=*arch* with *arch* either gfx906 or gfx908
Emit debugging symbols	-g

With AOMP 13.0-3 and ROCm 4.3, the compiler accepts the `--offload-arch` compiler switch that combines `-fopenmp-targets`, `-Xopenmp-target`, and `-march` into a single compiler switch.

Table 10 lists additional advanced optimization flags. Please consult the "*AMD ROCm™ Compiler Reference Guide*" for further information.

Table 10: Compiler options for GPU-code optimizations.

Optimization	Compiler Option
Disable compiler optimizations	-O0
Enable compiler optimizations	-O1, -O2, -O3, -Ofast
AMD-specific optimizations (ROCm 4.3, llvm-amdgpu-alt package)	-famd-opt
Allow aggressive, lossy floating-point optimizations	-ffast-math
Enable unsafe floating point atomic instructions (AMD GPUs only)	-munsafe-fp-atomics
Generate relocatable device code	-fgpu-rdc (-fno-gpu-rdc)

5.2 Understanding HIP

HIP is a source-portable language that can be compiled to run on AMD platform.

Useful information about installation and programming guides can be found in the HIP Programming Guide at *https://rocmdocs.amd.com/en/latest/Programming_Guides/Programming-Guides.html*.

5.2.1 HIPCC

The HIP language uses a portable compiler driver, `hipcc`, that will invoke the HIP backend compiler and give it all the necessary compiler options to find required include files and libraries required to compile and link a HIP program. A simple example to compile HIP code for AMD GPUs is:

```
$ hipcc -O3 source.hip.cpp -o program.exe
```

While the compiler determines the target platform to compile for automatically, it can be useful to select a compilation target (e.g., when compiling on a front-end node without GPUs). In that case, `HIP_PLATFORM=amd` can optionally be specified in the shell environment to override target detection. For ROCm platform stacks older than version 4.1, `HIP_PLATFORM=hcc` should be used instead.

The HIP compiler shares the same backend with the ROCmCC and AOMP compilers. As a result, the compiler switches in Section 5.1 to guide optimizations can also be used to compile HIP code.

5.2.2 HIPifying Code

The AMD ROCm™ platform optionally installs tools to translate CUDA® source code into portable HIP code. The tools offer two different approaches to the translation.

The `hipify-perl` program can be used to apply a simple text-based translation, that is, the tool searches through the CUDA code and performs a textual replacements of API routines and symbols of the CUDA API with their counterpart from the HIP API. While this translation is simple and fast, a drawback is that it does not perform a semantic analysis of the translated code and therefore can only detect a limited set of issues that may arise from the translation.

The `hipify-clang` tool in contrast is based on the clang compiler and uses the clang C++ parser to read and analyze the CUDA source code before the translation happens. Thus, the tool can detect more translation issues and produce better warnings about potential translation issues.

```
By default, the hipify-perl tool will output the translated source to the
console:
$ hipify-perl code.cu
#include <iostream>
#include "hip/hip_runtime.h"

int main(int argc, const char * argv[]) {
    /* code elided for space reasons */
}
```

The output can either be redirected to a new source file, or `hipify-perl` can be instructed to perform an in-place translation by using the `-i` command-line switch. The CUDA code is copied into a new file with the filename extension `.prehip`.

A usage example of the `hipify-clang` translation tool is:

```
$ hipify-clang --cuda-path=/usr/local/cuda-11.2
               code.cu
$ cat code.cu.hip
   #include <iostream>
#include "hip/hip_runtime.h"

int main(int argc, const char * argv[]) {
    /* code elided for space reasons */
}
```

By default, the hipify-clang tool creates a new source file for the HIP code and uses the filename extension `.hip` for it. For more information, please refer to the *HIP Porting Guide* at *https://github.com/ROCm-Developer-Tools/HIP/blob/main/docs/markdown/hip_porting_guide.md.*

5.3 Compiler Compatibility

As of the time of writing, the AMD software landscape offers three compilers:

- AOCC: The AMD Optimizing Compiler Collections for AMD EPYC™ processors.
- ROCmCC: The clang/LLVM compiler to target AMD Instinct™ accelerators.
- AOMP: The AMD OpenMP research compiler for AMD Instinct™ accelerators.

While mixing compilers is possible, for C/C++ source codes, special attention needs to be paid for applications written in Fortran. Fortran applications that do not use modern features such as INTERFACE definitions or MODULE files, linking source files compiled with any of the three compilers should produce a working executable program. For applications that rely on modern Fortran language features, it is advised to compile dependent packages (e.g., Open MPI) using the same compiler that is also used for the application code to avoid issues with the binary format of Fortran module files, etc.

Table 11 describes the capabilities of the compilers for the AMD Instinct™ accelerators in terms of APIs and programming language.

Table 11: Compiler support matrix for MPI and the OpenMP API.

	MPI		OpenMP API		
	C/C++	Fortran	C/C++	Fortran	C CUDA
ROCmCC	supported	kernel compiled with hipcc then linked with mpif90	clang, clang++	flang	via hipify and hipcc

5.4 Libraries

The ROCM software ecosystem provides several math and communication libraries optimized for AMD Instinct™ accelerators.

5.4.1 rocBLAS / hipBLAS

rocBLAS is a BLAS implementation on top of AMD's Radeon Open Compute runtime and toolchains. The hipBLAS library is a slim layer on top of the rocBLAS library to accommodate for easier access to the functionality from programs that are written in HIP or from "HIPified" source code (see also Section 5.2.2).

For more information, see the rocBLAS documentation at *https://rocblas.readthedocs.io/en/master/*.

5.4.2 rocSPARSE / hipSPARSE

rocSPARSE exposes a common interface that provides Basic Linear Algebra Subroutines for sparse computation implemented on top of AMD's Radeon Open Compute ROCm runtime and toolchains. The hipSPARSE library is an interface layer on top of the rocSOLVER library for easier access to the functionality from programs that are written in HIP or from "HIPified" source code (see also Section 5.2.2).

For more information, see the rocSPARSE documentation at *https://rocsparse.readthedocs.io/en/master/*.

5.4.3 rocSOLVER

rocSOLVER is an implementation of LAPACK routines on top of the AMD ROCm platform.

For more information, see the rocSOLVER documentation at *https://rocsolver.readthedocs.io/en/latest/*.

5.4.4 RCCL

RCCL (pronounced "Rickle") is a stand-alone library of standard collective communication routines for GPUs, implementing all-reduce, all-gather, reduce, broadcast, reduce-scatter, gather, scatter, and all-to-all. There is also initial support for direct GPU-to-GPU send and receive operations. It has been optimized to achieve high bandwidth on platforms using PCIe, xGMI as well as networking using InfiniBand Verbs or

TCP/IP sockets. RCCL supports an arbitrary number of GPUs installed in a single node or multiple nodes and can be used in either single- or multi-process (e.g., MPI) applications.

The collective operations are implemented using ring and tree algorithms and have been optimized for throughput and latency. For best performance, small operations can be either batched into larger operations or aggregated through the API.

Sources and examples can be found at *https://github.com/ROCmSoftwarePlatform/rccl*.

5.4.5 rocRAND

The rocRAND library provides functions that generate pseudo-random and quasi-random numbers.

It is implemented in the HIP programming language and optimized for AMD's latest discrete GPUs. It is designed to run on top of AMD's Radeon Open Compute ROCm runtime, but it also works on CUDA enabled GPUs.

Additionally, the rocRAND project includes a wrapper library called hipRAND which allows user to easily port CUDA applications that use cuRAND library to the HIP layer. In ROCm environment hipRAND uses rocRAND, however in CUDA environment cuRAND is used instead.

Sources and examples can be found at *https://github.com/ROCmSoftwarePlatform/rocRAND*.

For more information, see the rocRAND documentation at *https://rocrand.readthedocs.io/en/latest/*.

5.5 rocGDB

The AMD ROCm Debugger (ROCgdb) is the AMD ROCm source-level debugger for Linux, and is based on the GNU Debugger (GDB). It enables heterogeneous debugging on the AMD ROCm platform of an x86-based host architecture along with AMD GPU architectures and supported by the AMD Debugger API.

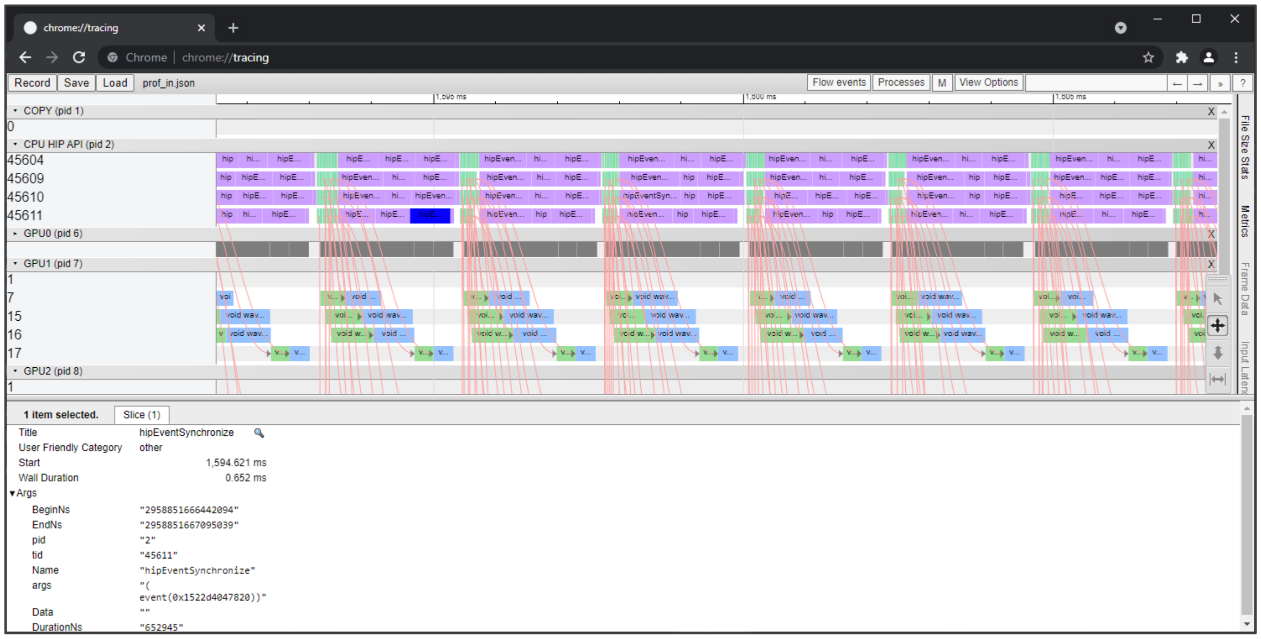
You can use the standard GDB commands for both CPU and GPU code debugging. For more information about ROCgdb, refer to the ROCgdb User Guide, which is installed at:

- /opt/rocm/share/info/gdb.info as a texinfo file
- /opt/rocm/share/doc/gdb/gdb.pdf as a PDF file

For more info see the documentation of the AMD ROCm Debugger at *https://rocmdocs.amd.com/en/latest/ROCm_Tools/ROCgdb.html*.

5.6 AMD ROCm profiler

The AMD ROCm profiler (rocprof) is a command line tool implemented on top of the ROC-Profiler and ROC-tracer APIs. The tool uses two profiling plugins loaded by ROCm runtime and based on ROC-Profiler and ROC-tracer for collecting metrics/counters, HW traces and runtime API/activity traces. The tool consumes an input XML or text file with counters list or trace parameters and provides output profiling data and statistics in various formats as text, CSV and JSON traces. Google Chrome™ tracing can be used to visualize the JSON traces with runtime API/activity timelines and per kernel counters data.



```
1 item selected.    Slice (1)
Title                    hipEventSynchronize
User Friendly Category   other
Start                         1,594.621 ms
Wall Duration                     0.652 ms
▾Args
    BeginNs              "2958851666442094"
    EndNs                "2958851667095039"
    pid                  "2"
    tid                  "45611"
    Name                 "hipEventSynchronize"
    args                 "(
                         event(0x1522d4047820))"
    Data                 ""
    DurationNs           "652945"
```

Figure 4: Google Chrome™ Tracing display of a JSON file created by rocprof.

For more information, see *https://rocmdocs.amd.com/en/latest/ROCm_Tools/ROCm-Tools.html*.

5.7 ROC-tracer API

The goal of the implementation is to provide a generic profiler, independent from specific runtime, to trace API and asynchronous activity. To use the ROC-tracer API, one needs to include the API header and link the application with the libroctracer64.so library:

- the API header: `/opt/rocm/roctracer/include/roctracer.h`
- the .so library: `/opt/rocm/lib/libroctracer64.so`

Examples and more information about the API can be found at *https://github.com/ROCm-Developer-Tools/roctracer*.

Chapter 6 Running Applications on AMD GPUs

In the following chapter, we will briefly show how to run programs that contain offload code for AMD Instinct™ accelerators. One can use these examples to have small tests that help further validate and test the AMD ROCm™ platform installation and make sure that there is a well-defined and functional environment for application porting or execution.

6.1 OpenMP Offload to GPU

The following example is a very short test program to illustrate how to compile and run a program that uses OpenMP target constructs to offload computation to GPUs.

The following program accesses each GPU in the system and determines if the offloaded code ran on a GPU device. It prints the result for each GPU on the console:

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char const * argv[]) {
    int ndev;
    int nteams;
    int thread_lmt;
    int on_host;

    ndev = omp_get_num_devices();
    printf("number of devices: %d\n", ndev);
    for (int i = 0; i < omp_get_num_devices(); ++i) {
        #pragma omp target teams \
                           device(i) \
                           map(from:on_host) map(from:nteams) \
                           map(from:thread_lmt)
        #pragma omp parallel
        #pragma omp master
        if (0 == omp_get_team_num()) {
            on_host = omp_is_initial_device();
            nteams = omp_get_num_teams();
            thread_lmt = omp_get_thread_limit();
        }
        printf("ran on GPU %d: %s, %d teams, "
               "limit of %d threads\n",
               i, on_host ? "no" : "yes", nteams, thread_lmt);
    }
    return 0;
}
```

The compiler flags to compile this program are:

```
$ /opt/rocm/llvm/bin/clang -g -O3 -std=c99 -fopenmp \
  -target x86_64-pc-linux-gnu -fopenmp-targets=amdgcn-amd-amdhsa \
  -Xopenmp-target=amdgcn-amd-amdhsa -march=gfx908 -o gpu_check gpu_check.c
```

At runtime, the code will then print the following output (when running on a server with eight GPUs):

```
$ OMP_TARGET_OFFLOAD=mandatory ./gpu_check
number of devices: 8
ran on GPU 0: yes, 480 teams, limit of 256 threads
ran on GPU 1: yes, 480 teams, limit of 256 threads
ran on GPU 2: yes, 480 teams, limit of 256 threads
ran on GPU 3: yes, 480 teams, limit of 256 threads
ran on GPU 4: yes, 480 teams, limit of 256 threads
ran on GPU 5: yes, 480 teams, limit of 256 threads
ran on GPU 6: yes, 480 teams, limit of 256 threads
ran on GPU 7: yes, 480 teams, limit of 256 threads
```

The following Table 12 shows useful environment variables and their values that can be set to influence how the OpenMP constructs interact with the GPU. Some of the environment settings are especially useful when debugging applications and to ensure that application code is not accidentally executed on the host part of the system. Please also consult Chapter 6 of the *OpenMP API specification, version 5.1*, or *https://www.openmp.org/spec-html/5.1/openmpch6.html* for further information.

Table 12: List of OpenMP API and clang/LLVM-specific environment variables.

Environment Variable	Settings
OMP_TARGET_OFFLOAD=mandatory	Always offload to GPU, fail if offloading was not possible.
OMP_TARGET_OFFLOAD=disabled	Disable offloading to GPU and fallback to host execution.
OMP_DEFAULT_DEVICE=*n*	Set the default GPU device to use to device with ID *n*

OMP_DISPLAY_ENV=true
OMP_DISPLAY_ENV=verbose
	Show the internal settings that will be effective for the OpenMP code.

OMP_NUM_TEAMS=*n*	Set the default number of teams to use on the GPU to be *n* (*n* should be a multiple of the number of CUs).
LIBOMPTARGET_KERNEL_TRACE=*n*	Enable tracing of offload kernels on the GPU, shows kernel invocations, incl. *n*=1: name and number of teams, thread limits, register usage. *n*=2: Same as *n=1* data plus data transfers and mapped pointers, incl. per-kernel timing information.

An example output with some the environment variables in effect might look like this (some information omitted from the output and replaced with "…"):

```
$ LIBOMPTARGET_KERNEL_TRACE=1 ./gpu_check
number of devices: 8
DEVID: 0 SGN:0 ConstWGSize:256  args: 3 teamsXthrds:( 480X 256) …
ran on GPU 0: yes, 480 teams, limit of 256 threads
DEVID: 1 SGN:0 ConstWGSize:256  args: 3 teamsXthrds:( 480X 256) …
ran on GPU 1: yes, 480 teams, limit of 256 threads
DEVID: 2 SGN:0 ConstWGSize:256  args: 3 teamsXthrds:( 480X 256) …
ran on GPU 2: yes, 480 teams, limit of 256 threads
DEVID: 3 SGN:0 ConstWGSize:256  args: 3 teamsXthrds:( 480X 256) …
ran on GPU 3: yes, 480 teams, limit of 256 threads
DEVID: 4 SGN:0 ConstWGSize:256  args: 3 teamsXthrds:( 480X 256) …
ran on GPU 4: yes, 480 teams, limit of 256 threads
DEVID: 5 SGN:0 ConstWGSize:256  args: 3 teamsXthrds:( 480X 256) …
ran on GPU 5: yes, 480 teams, limit of 256 threads
DEVID: 6 SGN:0 ConstWGSize:256  args: 3 teamsXthrds:( 480X 256) …
ran on GPU 6: yes, 480 teams, limit of 256 threads
DEVID: 7 SGN:0 ConstWGSize:256  args: 3 teamsXthrds:( 480X 256) …
ran on GPU 7: yes, 480 teams, limit of 256 threads
```

6.2 HIP Offload to GPU

The following hello world example code (i.e., `hello.cc`) will offload some kernel code to a GPU using HIP:

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

__global__ void helloGpu(int gpuId) {
    printf("device %d Gpu thread %d\n", gpuId, (int)threadIdx.x);
}

int main(int argc, const char * argv[]){
    int device = 0, gpuId = device;

    // Set GPU device
    hipSetDevice(device);

    // Launch kernel on the GPU with one block of 64 threads
    hipLaunchKernelGGL((helloGpu), dim3(1), dim3(64), 0, 0, gpuId);

    // Wait for kernel completion
    hipDeviceSynchronize();

    return 0;
}
```

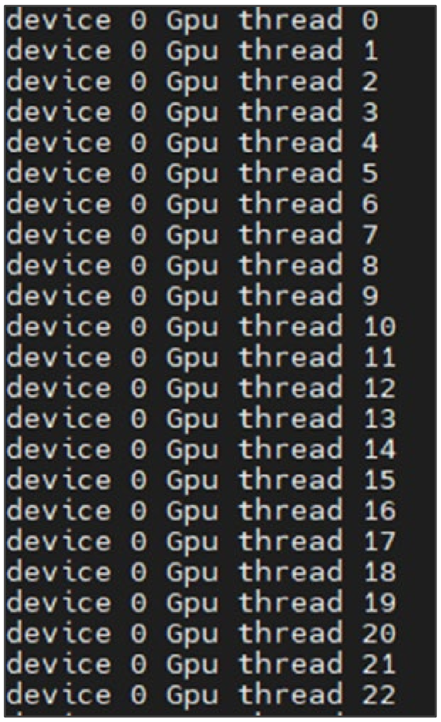
To compile:

```
$ hipcc -o hello hello.cc
```

To run:

```
$ ./hello
```

The sample output will then be similar to this:



```
device 0 Gpu thread 0
device 0 Gpu thread 1
device 0 Gpu thread 2
device 0 Gpu thread 3
device 0 Gpu thread 4
device 0 Gpu thread 5
device 0 Gpu thread 6
device 0 Gpu thread 7
device 0 Gpu thread 8
device 0 Gpu thread 9
device 0 Gpu thread 10
device 0 Gpu thread 11
device 0 Gpu thread 12
device 0 Gpu thread 13
device 0 Gpu thread 14
device 0 Gpu thread 15
device 0 Gpu thread 16
device 0 Gpu thread 17
device 0 Gpu thread 18
device 0 Gpu thread 19
device 0 Gpu thread 20
device 0 Gpu thread 21
device 0 Gpu thread 22
```

6.3 MPI/OpenMP Offload to GPU

The following hello world example code (i.e., `mpi_hello_gpu_omp.cpp`) will offload a loop to a GPU using OpenMP and can be run with multiple MPI ranks (comments on building OpenMPI are below the example):

```
#include <stdio.h>
#include <stdlib.h>

#include <mpi.h>
#include <omp.h>

int main(int argc, const char * argv[]) {
    int nproc;
    int rank;
    int offloaded = 0;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nproc);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    int ndev = omp_get_num_devices();
    if (!ndev) {
        fprintf(stderr, "No GPU devices found (rank %d)!\n",
                   rank);
        return EXIT_FAILURE;
    }

    if (!(rank % 2)) {
        int dev = (rank/2) % ndev;

        #pragma omp target map(tofrom:offloaded)
        {
            offloaded = 1;
        }
        if (offloaded) {
            printf("Hello World from rank %d of %d, "
                      "ran on GPU %d\n", rank, nproc, dev);
        }
    }
    else {
        printf("Hello World from rank %d of %d\n",
                  rank, nproc);
    }

    MPI_Finalize();

    return EXIT_SUCCESS;
}
```

The following command compiles the small "Hello World" program (assuming that it was saved in a file called `mpi_hello_gpu_omp.c`):

```
$ mpicc -g -std=c99 -fopenmp -target x86_64-pc-linux-gnu \
           -fopenmp-targets=amdgcn-amd-amdhsa \
           -Xopenmp-target=amdgcn-amd-amdhsa -march=gfx908 \
           -o mpi_hello_gpu_omp mpi_hello_gpu_omp.c
```

With the ROCm™ platform version 4.3, the command will simplify to:

```
$ mpicc -g -std=c99 -fopenmp --offload-arch=gfx908 \
           -o mpi_hello_gpu_omp mpi_hello_gpu_omp.c
```

The above command assumes Open MPI has been built with the following configuration:

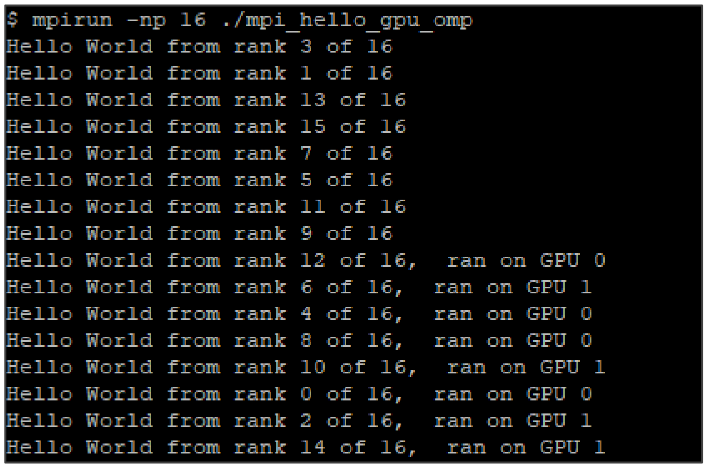
```
CC=/opt/rocm/llvm/bin/clang
   CXX=/opt/rocm/llvm/bin/clang++
   FC=/opt/rocm/llvm/bin/flang
```

If the Open MPI configuration was not configured to map the MPI compiler wrappers to the ROCm OpenMP compiler, one can use the following command line to compile the above example program:

```
$ clang -g -O3 -std=c99 -fopenmp -target x86_64-pc-linux-gnu \
           -fopenmp-targets=amdgcn-amd-amdhsa \
           -Xopenmp-target=amdgcn-amd-amdhsa -march=gfx908 \
           $(mpicc -showme:compile) $(mpicc -showme:link) \
           -o mpi-offload mpi-offload.c
```

Here, the compiler command line uses the `-showme` switch of the `mpicc` compiler wrapper to output the compiler options and linker options that are required for the Open MPI interfaces.

The example output for 16 MPI processes on a machine with two GPUs will then look like this:



```
$ mpirun -np 16 ./mpi_hello_gpu_omp
Hello World from rank 3 of 16
Hello World from rank 1 of 16
Hello World from rank 13 of 16
Hello World from rank 15 of 16
Hello World from rank 7 of 16
Hello World from rank 5 of 16
Hello World from rank 11 of 16
Hello World from rank 9 of 16
Hello World from rank 12 of 16,  ran on GPU 0
Hello World from rank 6 of 16,  ran on GPU 1
Hello World from rank 4 of 16,  ran on GPU 0
Hello World from rank 8 of 16,  ran on GPU 0
Hello World from rank 10 of 16,  ran on GPU 1
Hello World from rank 0 of 16,  ran on GPU 0
Hello World from rank 2 of 16,  ran on GPU 1
Hello World from rank 14 of 16,  ran on GPU 1
```

Another way of running programs on GPUs is by using containers, described in the following chapter.

Chapter 7 Using Containers on Instinct MI100

Containers provide a lightweight, fast, and isolated infrastructure to run your applications. The application, dependencies, libraries, binaries, and configuration files are usually bundled into the container image, and thus provide an easy approach to migrating your application anywhere. Containers are therefore flexible, consistent, and portable, and can run on almost any architecture that provides a sufficient container infrastructure.

Containers are instances of images that contain the data needed to create a container. The major difference between a container and an image is that containers have a writable layer and thus can modify data that can then be made persistent or discarded, depending on how the container is used.

AMD will be making containers for various GPU-ready applications available through its 'Infinity Hub' at *https://www.amd.com/en/technologies/infinity-hub*.

7.1 Obtaining Docker Images

As an example, the Docker® image for GROMACS can be obtained using the following command:

```
$ docker pull amdih/gromacs:2020.3
```

Once the command is completed, you can create a container from the image.

7.2 Running Containers

To instantiate a container from the previously downloaded image, execute the following command:

```
$ docker run --rm -it --device=/dev/kfd \
             --device=/dev/dri \
             --security-opt seccomp=unconfined \
             amdih/gromacs:2020.3 /bin/bash
```

The user account that invokes the above command must be in the `docker` group (or another user group that permits users to run Docker containers).

The above command instantiates the container from the downloaded image, runs it, and gives the user a command prompt inside the container. Once inside, the following sequence of commands will run GROMACS for the *adh_dodec* benchmark using 1 GPU:

```
$ cd /benchmarks/data/tmpi/adh_dodec
$ gmx grompp -f pme_verlet.mdp -c conf.gro -p topol.top -maxwarn 20
$ gmx mdrun -pin on -nsteps 100000 -resetstep 90000 \
              -ntmpi 2 -ntomp 28 -noconfout -nb gpu \
               -bonded cpu -pme gpu -npme 1 -v -gpu_id 0 \
               -s topol.tpr
```

Chapter 8 Appendix A - Standard HPC Benchmarks

The NUMA Per Socket (NPS) setting might impact performance depending on the workload type. It involves a trade-off between minimizing local memory latency versus maximizing per core memory bandwidth. Highly parallel workloads that are programmed to take NUMA into account might benefit from NPS4 (i.e., four NUMA nodes per CPU socket) while non-NUMA friendly applications might benefit from NPS1 (i.e., one NUMA node per CPU socket). The following benchmark tests can be used to illustrate the impact of the NPS setting.

8.1 STREAM and DGEMM

The following examples show how RVS (see Section 4.4) can be configured to run tests on a GPU system and to validate its functionality and performance.

8.1.1 Using RVS to Measure GPU Memory Bandwidth

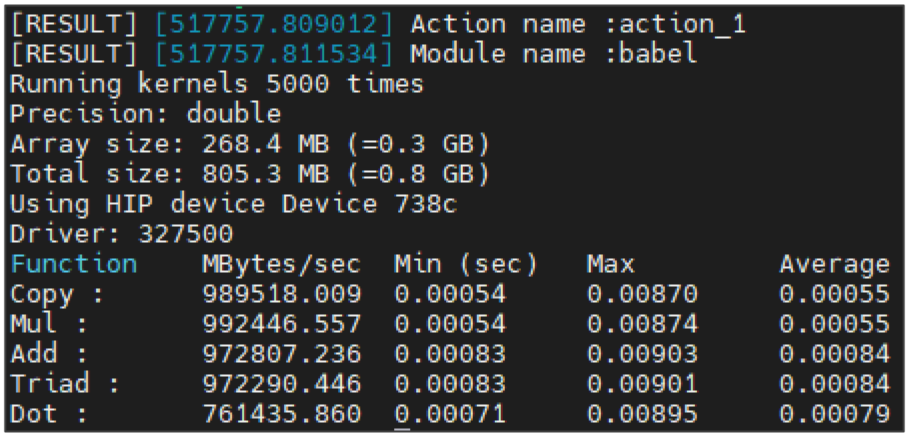
The STREAM bandwidth of the system can be measured using the following config file that configures a test for RVS that uses the BabelStream benchmark to determine memory bandwidth.

```
# BABEL test
#
# Preconditions:
#   Set device to all. If you need to run the rvs only on a
   #   subset of GPUs, please run rvs with -g option, collect the
   #   GPU IDs (e.g.: GPU[ 5 - 50599] -> 50599 is the GPU ID) and
   #   then specify all the GPUs IDs separated by white space
   #   (e.g., device: 50599 3245).
#   Set parallel execution to false.
#   Set buffer size to reflect the buffer you want to test.
#   Set run count to 2 (test will run twice).

actions:
- name: action_1
  device: all
  module: babel          # Name of the module
  parallel: true         # Parallel true or false
  count: 1               # Number of times you want to repeat the
                            # test from the begin ( A clean start
                            # every time)
  num_iter: 5000         # Number of iterations, this many
                            # kernels are launched simultaneously
                            # and stresses the system
  array_size: 33554432   # Buffer size the test operates,
                            # this is 32MB
  test_type: 2           # type of test, 1: Float, 2: Double,
                            # 3: Triad float, 4: Triad double
  mibibytes: false       # mibibytes , if you want to specify in
                            # bytes (array size in bytes),
                            # make this true
  o/p_csv: false         # o/p as csv file
  subtest: 1             # 1: copy 2: copy+mul 3: copy+mul+add
                            # 4:copy+mul+add+traid
                            # 5: copy+mul+add+traid+dot
```

After saving the configuration file as `babel.conf`, use the following command to run `rvs`:

```
$ /opt/rocm/rvs/rvs --config babel.conf
```

The output for one GPU should then look very similar to this:



```
[RESULT] [517757.809012] Action name :action_1
[RESULT] [517757.811534] Module name :babel
Running kernels 5000 times
Precision: double
Array size: 268.4 MB (=0.3 GB)
Total size: 805.3 MB (=0.8 GB)
Using HIP device Device 738c
Driver: 327500
Function    MBytes/sec  Min (sec)   Max         Average
Copy :      989518.009  0.00054     0.00870     0.00055
Mul  :      992446.557  0.00054     0.00874     0.00055
Add  :      972807.236  0.00083     0.00903     0.00084
Triad :     972290.446  0.00083     0.00901     0.00084
Dot  :      761435.860  0.00071     0.00895     0.00079
```

This benchmark determined an effective memory bandwidth of 972.3 GB/sec for Triad out of about 1,228 GB/sec peak memory bandwidth. This is about 79% of the theoretical peak memory bandwidth.

Table 13 shows the results using one GPU and different NPS settings.

Table 13: GPU memory bandwidth measured by BabelStream.

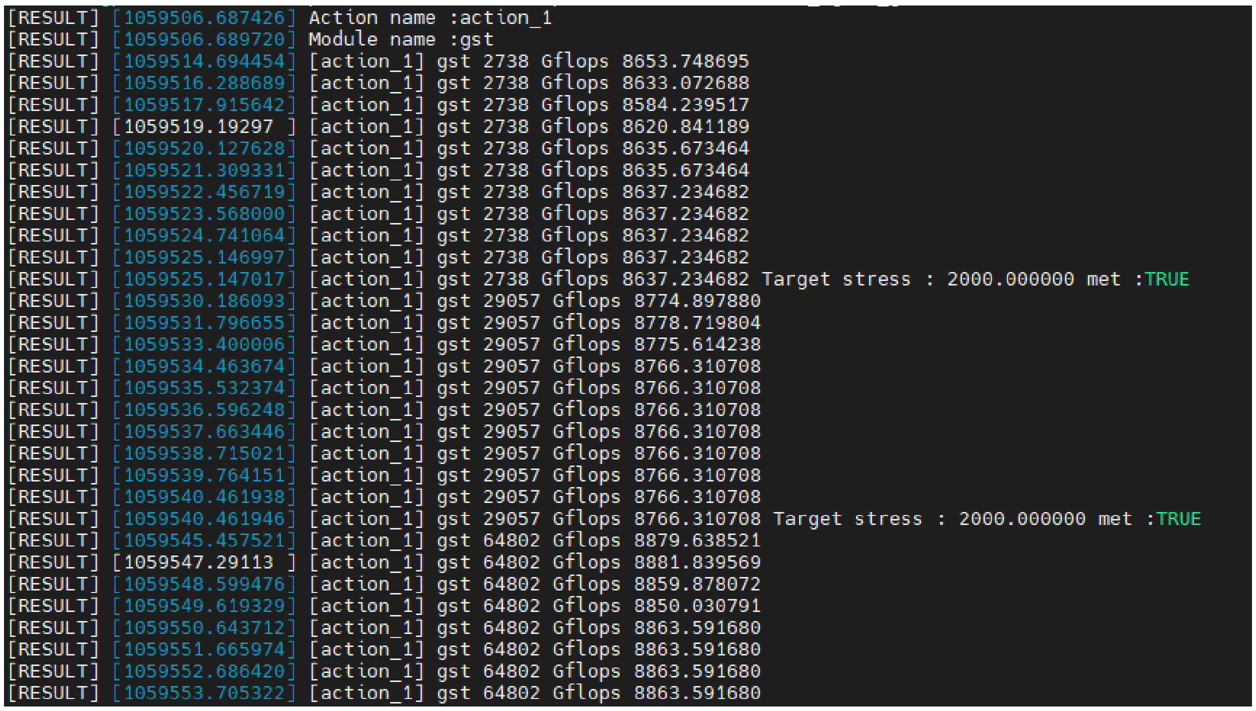
NPS1	NPS4
957.861 GB/s	972.290 GB/s

8.1.2 Using RVS to Measure DGEMM Performance

RVS comes with a pre-configured benchmark to test the compute performance of installed GPUs using the BLAS3 function DGEMM. You can run it via the following command:

```
$ /opt/rocm/rvs/rvs -c /opt/rocm/rvs/conf/Artus_dgemm_gst.conf
```

The output should look very similar to the following screenshot:



```
[RESULT] [1059506.687426] Action name :action_1
[RESULT] [1059506.689720] Module name :gst
[RESULT] [1059514.694454] [action_1] gst 2738 Gflops 8653.748695
[RESULT] [1059516.288689] [action_1] gst 2738 Gflops 8633.072688
[RESULT] [1059517.915642] [action_1] gst 2738 Gflops 8584.239517
[RESULT] [1059519.19297 ] [action_1] gst 2738 Gflops 8620.841189
[RESULT] [1059520.127628] [action_1] gst 2738 Gflops 8635.673464
[RESULT] [1059521.309331] [action_1] gst 2738 Gflops 8635.673464
[RESULT] [1059522.456719] [action_1] gst 2738 Gflops 8637.234682
[RESULT] [1059523.568000] [action_1] gst 2738 Gflops 8637.234682
[RESULT] [1059524.741064] [action_1] gst 2738 Gflops 8637.234682
[RESULT] [1059525.146997] [action_1] gst 2738 Gflops 8637.234682
[RESULT] [1059525.147017] [action_1] gst 2738 Gflops 8637.234682 Target stress : 2000.000000 met :TRUE
[RESULT] [1059530.186093] [action_1] gst 29057 Gflops 8774.897880
[RESULT] [1059531.796655] [action_1] gst 29057 Gflops 8778.719804
[RESULT] [1059533.400006] [action_1] gst 29057 Gflops 8775.614238
[RESULT] [1059534.463674] [action_1] gst 29057 Gflops 8766.310708
[RESULT] [1059535.532374] [action_1] gst 29057 Gflops 8766.310708
[RESULT] [1059536.596248] [action_1] gst 29057 Gflops 8766.310708
[RESULT] [1059537.663446] [action_1] gst 29057 Gflops 8766.310708
[RESULT] [1059538.715021] [action_1] gst 29057 Gflops 8766.310708
[RESULT] [1059539.764151] [action_1] gst 29057 Gflops 8766.310708
[RESULT] [1059540.461938] [action_1] gst 29057 Gflops 8766.310708
[RESULT] [1059540.461946] [action_1] gst 29057 Gflops 8766.310708 Target stress : 2000.000000 met :TRUE
[RESULT] [1059545.457521] [action_1] gst 64802 Gflops 8879.638521
[RESULT] [1059547.29113 ] [action_1] gst 64802 Gflops 8881.839569
[RESULT] [1059548.599476] [action_1] gst 64802 Gflops 8859.878072
[RESULT] [1059549.619329] [action_1] gst 64802 Gflops 8850.030791
[RESULT] [1059550.643712] [action_1] gst 64802 Gflops 8863.591680
[RESULT] [1059551.665974] [action_1] gst 64802 Gflops 8863.591680
[RESULT] [1059552.686420] [action_1] gst 64802 Gflops 8863.591680
[RESULT] [1059553.705322] [action_1] gst 64802 Gflops 8863.591680
```

The achieved 8,863.6 GFLOPS for the `dgemm` operation corresponds to about 77% of the theoretical compute performance of about 11,500 GFLOPS for FP64 operations.

Table 14 shows the results for one GPU and different NPS settings.

Table 14: GPU Performance of the dgemm benchmark.

NPS1	NPS4
8,753.82	8,781.35

8.1.3 HPCG

HPCG is intended to model the data access patterns of real-world applications such as sparse matrix calculations, thus testing the effect of limitations of the memory subsystem and internal interconnect of the supercomputer on its computing performance. HPCG testing generally achieves only a tiny fraction of the peak FLOPS of the computer because it is memory bound.

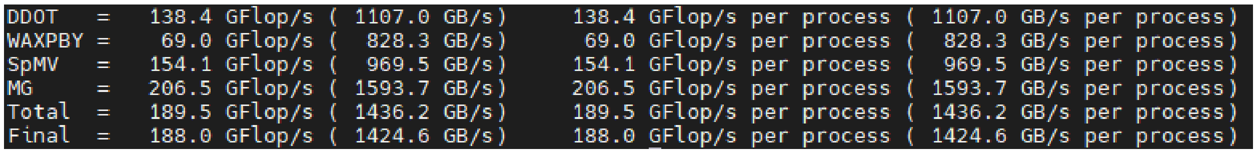
To set HPCG up for AMD Instinct™ GPUs, please download the HIP source code of HPCG and compile the code. Please make sure that the desired version of an MPI library is accessible and the environment variables of the shell are set up to point to that MPI library.

```
$ git clone https://github.com/ROCmSoftwarePlatform/rocHPCG.git
$ cd rocHPCG
$ ./install.sh -di
```

To run the compiled HPCG executable, use the following commands:

```
$ cd build/release/bin
$ ./rochpcg 280 280 280 1860 --dev=0
```

This command will run HPCG using a single GPU. The final output will look similar to this:



```
DDOT   =   138.4 GFlop/s ( 1107.0 GB/s)     138.4 GFlop/s per process ( 1107.0 GB/s per process)
WAXPBY =    69.0 GFlop/s (  828.3 GB/s)      69.0 GFlop/s per process (  828.3 GB/s per process)
SpMV   =   154.1 GFlop/s (  969.5 GB/s)     154.1 GFlop/s per process (  969.5 GB/s per process)
MG     =   206.5 GFlop/s ( 1593.7 GB/s)     206.5 GFlop/s per process ( 1593.7 GB/s per process)
Total  =   189.5 GFlop/s ( 1436.2 GB/s)     189.5 GFlop/s per process ( 1436.2 GB/s per process)
Final  =   188.0 GFlop/s ( 1424.6 GB/s)     188.0 GFlop/s per process ( 1424.6 GB/s per process)
```

This version of HPCG relies on a 1:1 mapping of MPI processes to GPUs. For example, to execute on eight GPUs the following command line must be used:

```
mpirun -np 8 rochpcg 280 280 280 1860
```

Table 15 shows the final GFLOPS for runs with different number of GPUs (one to eight) and with different NPS settings. In the NPS1 setting, there are 2 NUMA nodes with 4 GPUs each while for the NPS4 setting each GPU is on a different NUMA node.

Table 15: GPU performance of the HPCG benchmark.

Number of GPUs	**NPS1**	**NPS4**
	Final GFlop/s	**Final GFlop/s**
1	188.0	153.4
2	300.9	303.7
3	446.9	444.7
4	636.4	593.0
5	803.2	745.5
6	968.4	882.0
7	1,026.6	1,029.3
8	1,117.6	1,089.5

8.1.4 GROMACS

GROMACS is a versatile package for performing molecular dynamics for systems with hundreds to millions of particles. It is primarily designed for biochemical molecules like proteins, lipids, and nucleic acids that have a multitude of complicated bonded interactions. Since GROMACS is extremely fast at calculating the non-bonded interactions typically dominating simulations, many researchers use it for research on non-biological systems, such as polymers. GROMACS supports all the usual methods expected from a modern molecular dynamics implementation.

For more information, refer to the GROMACS official website at *https://www.gromacs.org*.

The following table shows the *ns/day* values obtained using different number of GPUs and using the GROMACS docker container described in Sections 7.1 and 7.2 to run the *adh_dodec* benchmark with the threaded MPI version.

The following command was used to run the test for one GPU:

```
$ gmx mdrun -pin on -nsteps 100000 -resetstep 90000 -ntmpi 2 -ntomp 28 \
            -noconfout -nb gpu -bonded cpu -pme gpu -npme 1 -v -gpu_id 0 \
            -s topol.tpr
```

Table 16 shows the maximum performance numbers obtained with different number of GPUs, CPU threads and NPS settings.

Table 16: GPU performance of GROMACS.

Number of GPUs	NPS1 ns/day	NPS4 ns/day	Command-line Arguments
1	159.916	157.018	-ntmpi 2 -ntomp 28 –gpu_id 0
2	217.687	223.911	-ntmpi 2 -ntomp 18 –gpu_id 01
1+1	202.339	190.503	-ntmpi 2 -ntomp 18 –gpu_id 04
3	275.339	263.353	-ntmpi 3 -ntomp 18 –gpu_id 012
4	309.525	292.321	-ntmpi 4 -ntomp 18 –gpu_id 0123
2+2	297.080	281.413	-ntmpi 4 -ntomp 18 –gpu_id 0145

In the table above, “1+1” denotes that the program used two GPUs which are in two different hives inside the same node, so one GPU per processor. This configuration shows the impact of communication between GPUs through PCIe. The “2+2” configuration involves four GPUs with two GPUs on the first hive (and processor) and the other two on the second hive (and processor).

Chapter 9 Appendix B - GPU-Enabled MPI

The Message Passing Interface (MPI, see *https://www.mpi-forum.org*) is a standard API for distributed and parallel application development that can scale to multi-node clusters. To facilitate the porting of applications to clusters with GPUs, ROCm enables various technologies. These technologies allow users to directly use GPU pointers in MPI calls and enable ROCm-aware MPI libraries to deliver optimal performance for both intra-node and inter-node GPU-to-GPU communication.

ROC Kernel driver exposes Remote Direct Memory Access (RDMA) through the *PeerDirect* interfaces to allow the HCA (NIC) to directly read and write to the GPU device memory with RDMA capabilities. These interfaces are currently registered as a `peer_memory_client` with Mellanox's OFED `ib_core` kernel module to allow high-speed DMA transfers between GPU and HCA. These interfaces are used to optimize inter-node MPI message communication.

This chapter exemplifies how to setup Open MPI with the ROCm platform. The Open MPI project is an open-source implementation of the Message Passing Interface (MPI) that is developed and maintained by a consortium of academic, research, and industry partners.

Several MPI implementations can be made ROCm-aware by compiling them with UCX support. One notable exception is MVAPICH2: it directly supports AMD GPUs without using UCX and can be downloaded at *http://mvapich.cse.ohio-state.edu/downloads/*. Please use the latest version of the MVAPICH2-GDR package.

The Unified Communication Framework, is an open-source cross-platform framework whose goal is to provide a common set of communication interfaces that targets a broad set of network programming models and interfaces. UCX is ROCm-aware and ROCm technologies are used directly to implement various network operation primitives. For more details on the UCX design please refer to the documentation at *http://www.openucx.org/documentation.*

9.1 Building UCX

The following section will describe how to setup UCX, so that it can be used to compile Open MPI. The following environment variable is set, such that all software components will be installed in the same base directory:

```
$ export INSTALL_DIR=$HOME/mpi_install
```

9.2 Install UCX

The next step then is to set up UCX by compiling its source code and install it:

```
$ export UCX_DIR=$INSTALL_DIR/ucx
$ cd ~
$ git clone https://github.com/openucx/ucx.git -b v1.11x
$ cd ucx
$ ./autogen.sh
$ mkdir build
$ cd build
$ ../contrib/configure-release --prefix=$UCX_DIR \
    --with-rocm=/opt/rocm --with-knem=/opt/knem-1.1.4.90mlnx1 \
    --with-xpmem=$XPMEM_DIR  --without-cuda \
    --enable-optimizations --disable-logging \
    --disable-debug --disable-assertions \
    --disable-params-check  --without-java
$ make -j$(nproc)
$ make -j$(nproc) install
```

9.3 Install Open MPI

These are the steps to build Open MPI with AOCC 3.0.0:

```
$ export OMPI_DIR=$INSTALL_DIR/ompi
$ export LD_LIBRARY_PATH=/opt/mellanox/hcoll/lib:$LD_LIBRARY_PATH
$ cd ~
$ git clone --recursive https://github.com/open-mpi/ompi.git \
    -b v4.1.x
$ cd ompi
$ ./autogen.pl
$ mkdir build
$ cd build
$ ../configure --prefix=$OMPI_DIR --with-ucx=$UCX_DIR \
    --enable-mca-no-build=btl-uct --enable-mpi1-compatibility \
    --enable-mpi-cxx --with-hcoll=/opt/mellanox/hcoll \
    --with-slurm --with-pmix CC=clang CXX=clang++ FC=flang
$ make -j $(nproc)
$ make -j $(nproc) install
```

9.4 ROCm-enabled OSU

The OSU Micro Benchmarks v5.7 (i.e., OMB) can be used to evaluate the performance of various primitives with AMD GPU device and ROCm support. This functionality is exposed when configured with `--enable-rocm` option. One can use the following steps to compile OMB:

```
$ wget http://mvapich.cse.ohio-state.edu/download/mvapich/osu-micro-
benchmarks-5.7.tar.gz
$ tar xfz osu-micro-benchmarks-5.7.tar.gz
$ mv osu-micro-benchmarks-5.7 osu
$ cd osu
$ ./configure --enable-rocm --with-rocm=/opt/rocm \
    CC=$OMPI_DIR/bin/mpicc CXX=$OMPI_DIR/bin/mpicxx \
    LDFLAGS="-L$OMPI_DIR/lib/ -lmpi -L/opt/rocm/lib/ \
    $(hipconfig -C) -lamdhip64" CXXFLAGS="-std=c++11"
$ make -j$(nproc)
```

9.5 Intra-node Run

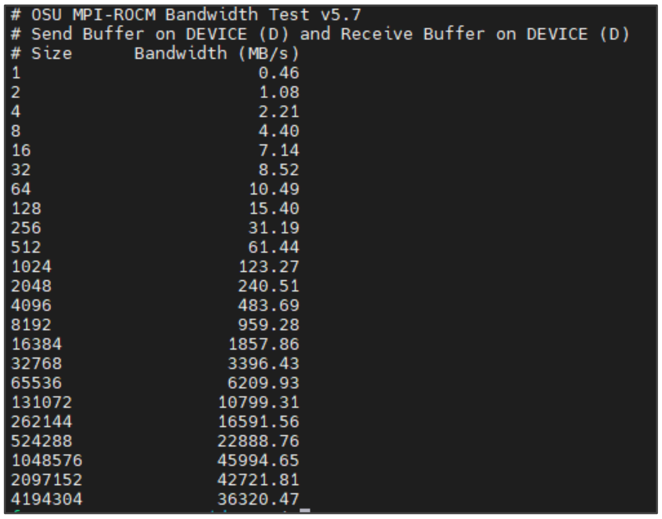
The following command will run the OSU bandwidth benchmark between the first two GPUs (i.e., GPU 0 and GPU 1) by default inside the same node. It measures the unidirectional bandwidth from the first device to the other.

```
$ $OMPI_DIR/bin/mpirun -np 2 --mca btl '^openib' \
   -x UCX_RNDV_THRESH=128 -x UCX_TLS=sm,self,rocm_copy,rocm_ipc \
   --mca pml ucx --mca osc ucx mpi/pt2pt/osu_bw -d rocm D D
```

To select different devices, for example 2 and 3, the following command can be used:

```
$ export HIP_VISIBLE_DEVICES=2,3
```

Output example:



```
# OSU MPI-ROCM Bandwidth Test v5.7
# Send Buffer on DEVICE (D) and Receive Buffer on DEVICE (D)
# Size      Bandwidth (MB/s)
1                       0.46
2                       1.08
4                       2.21
8                       4.40
16                      7.14
32                      8.52
64                     10.49
128                    15.40
256                    31.19
512                    61.44
1024                  123.27
2048                  240.51
4096                  483.69
8192                  959.28
16384                1857.86
32768                3396.43
65536                6209.93
131072              10799.31
262144              16591.56
524288              22888.76
1048576             45994.65
2097152             42721.81
4194304             36320.47
```

Chapter 10 Appendix C – Additional Resources

10.1 Resources

- AMD Developer Central
  http://developer.amd.com

- AMD Infinity Hub
  https://www.amd.com/en/technologies/infinity-hub

- AMD ROCm Debugger
  https://rocmdocs.amd.com/en/latest/ROCm_Tools/ROCgdb.html

- AMD ROCm Profiler
  https://rocmdocs.amd.com/en/latest/ROCm_Tools/ROCm-Tools.html

- AOMP Compiler Download
  https://github.com/ROCm-Developer-Tools/aomp

- Docker Documentation
  https://docs.docker.com/

- High Performance Computing (HPC) Tuning Guide for AMD EPYC™ 7002 Series Processors
  https://developer.amd.com/wp-content/resources/56827-1-0.pdf

- High Performance Computing (HPC) Tuning Guide for AMD EPYC™ 7003 Series Processors
  https://www.amd.com/system/files/documents/high-performance-computing-tuning-guide-amd-epyc7003-series-processors.pdf

- HIP Porting Guide
  https://github.com/ROCm-Developer-Tools/HIP/blob/main/docs/markdown/hip_porting_guide.md

- HIP Programming Guide
  https://rocmdocs.amd.com/en/latest/Programming_Guides/Programming-Guides.html

- Message Passing Interface
  https://www.mpi-forum.org

- OpenMP API Specification, Version 5.1
  https://www.openmp.org/spec-html/5.1/openmp.html

- Power Management Utility
  https://developer.amd.com/iopm-utility/

- RCCL Examples
  https://github.com/ROCmSoftwarePlatform/rccl
- ROC-tracer
  https://github.com/ROCm-Developer-Tools/roctracer
- rocBLAS Documentation
  https://rocblas.readthedocs.io/en/master/
- rocHPCG
  https://github.com/ROCmSoftwarePlatform/rocHPCG.git
- ROCm Bandwidth Test
  https://github.com/RadeonOpenCompute/rocm_bandwidth_test
- ROCm Validation Suite
  https://github.com/ROCm-Developer-Tools/ROCmValidationSuite
- rocRAND Documentation
  https://rocrand.readthedocs.io/en/latest/
- rocSOLVER Documentation
  https://rocsolver.readthedocs.io/en/latest/
- rocSPARSE Documentation
  https://rocsparse.readthedocs.io/en/master/