AMD

High Performance Computing (HPC) Tuning Guide for AMD EPYC™ 7002 Series Processors

Publication	56827
Revision	2.0
Issue Date	November, 2020

Purpose

This document provides guidance for getting started tuning AMD 2nd Gen EPYC™ Processor based systems for HPC workloads. This is not an all-inclusive guide and some items may have similar, but different, names in specific OEM systems (e.g. OEM-specific BIOS settings). Every HPC workload varies in its performance characteristics. While this guide is a good starting point, you are encouraged to perform your own performance testing for additional tuning. This guide also provides suggestions on which items should be the focus of additional, application-specific tuning.

Contents

Date	Revision	Description
November, 2020	2.0	Merged with previously published version
January, 2020	1.0	Initial Public Release

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 enable optimization 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 and optimizations
- Workload-specific recommendations for both BIOS and operating system settings

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

1.1 Prerequisites

To use this document and perform tuning for HPC, the technical audience should have experience in configuring servers, along with the following:

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

Chapter 2 Microarchitecture and Settings

2.1 AMD EPYC™ 7002 Series Processor

AMD EPYC™ 7002 Series Processors are based on AMD 'Zen 2' cores and microarchitecture, built with leading-edge 7nm technology. The AMD EPYC™ SoC is x86 based and offers a consistent set of features across 8 to 64 cores, including 128 lanes of PCIe® Gen 4, 8 memory channels and access to up to 4 TB of high-speed memory. AMD EPYC™ 7002 Series processors are built with the following specifications:

AMD EPYC™ 7002 Series	
Processor technology	7nm
Max number of cores	64
Max memory speed	3200 MHz
Max memory capacity	4 TB
Peripheral Component Interconnect	128 lanes (max) PCIe®Gen4

The diagram below shows a high-level block diagram of a nine die AMD EPYC 7002 series processor. There is a central I/O die in the middle of the diagram and eight distinct CCDs (Compute Complex Die) surrounding the IO die. The specifics of the layout of memory channels, cores, and cache have implications on performance that will be explained in this section.

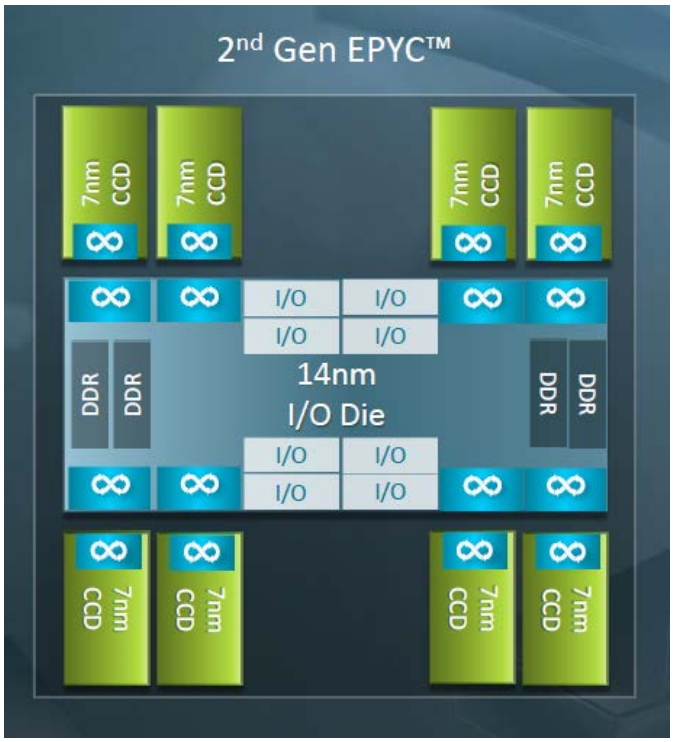


Figure 1 EPYC 7002 Configuration with 8 Core Complex Dies (CCDs) and central I/O Die (IOD)

2.2 Zen 2 core

The AMD EPYC 7002 Series processor is based on the new Zen2 processor core, that includes an L1 write-back cache and a private 512KB L2 cache. Each core can support Simultaneous Multi-Threading (SMT), allowing 2 execution threads to execute simultaneously per core.

2.3 Core Complex Die (CCD)

Each 2nd Gen EPYC processor consists of an I/O Die (IOD) and up to eight Core Complex Die (CCD). The CCDs contain the cores and cache of the CPU. The CCDs connect to the I/O Die using AMD's Infinity Fabric™. The CCDs connect to the IOD to access memory, I/O, and each other. There is support for up to 8 memory channels per socket, and 128 lanes of PCIe Gen 4.

Each CCD contains up to two Core Complexes (CCX), described in the next section.

2.4 Core-Complex (CCX)

A Core Complex (CCX) consists of up to 4 cores and a shared 16MB (last level) L3 cache. Each CCD contains up to two Core Complexes (CCX). While the two CCXs and their associated L3 Caches are on the same chiplet, they are separate.

Figure 2 Two Core Complexes CCXs) on a Core Complex Die (CCD)

It is possible to disable cores using one or both of the following approaches in BIOS:

- Reduce the cores per L3 from 4 down to 3,2 or 1, keeping the number of CCDs constant.
    - This approach increases the effective cache per core ratio. It also reduces the number of cores sharing the cache.
- Reduce the number of CCDs active, keeping the cores per CCD constant.
    - This approach maintains the advantages of cache sharing between the cores, while maintaining the same cache per core ratio.

2.5 Infinity Data Fabric (DF)

The infinity fabric provides the coherent memory connection between all major components of the processor and between the CPUs in a 2-socket system. It supports speeds up to 1467MHz (FCLK).

2.6 Unified Memory Controller (UMC)

Each memory controller can operate at up to 1600MHz (MEMCLOCK) and can therefore support DDR4 main memory up to 3200MHz.

If the memory used is DDR4-2933 then the memory controllers will operate at 1467MHz. This is the same speed as the data fabric, and this is referred to as Coupled Mode. This provides the lowest memory latency. However, maximum memory bandwidth is still achieved using DDR4-3200 R2 DIMMs.

Other modes of operation and 'Memory P-States' exist and are discussed in the Workload Tuning Guide but for HPC systems the highest performance pertains to the narrative just described.

2.7 Memory and I/O Layout

Each AMD EPYC 7002 Series processor supports 8 memory channels. Each memory channel supports up to 2 DIMMs. The system can have access to a maximum of 4TB of DDR4-3200 memory per processor. The PCI Gen 4 subsystem provides up to 128 lanes of high speed I/O.

While all memory and I/O connect to the single I/O Die, they can be abstracted into logical quadrants, each with 2 memory channels and 32 I/O lanes. The memory channels can be interleaved within a quadrant (2-way), all the way through 16-channel interleave, which would interleave across all memory channels of a 2-socket system (see the NUMA section for more information).

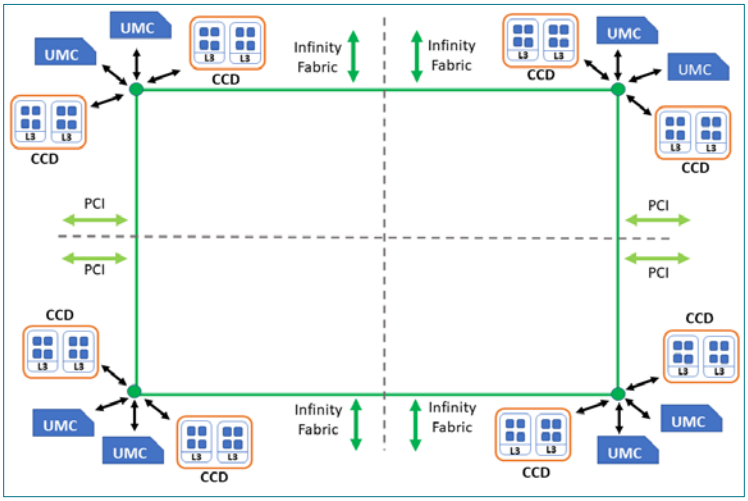


Figure 3 EPYC 7002 logical layout into quadrants around the IO Die.

Chapter 3 NUMA

The AMD EPYC 7002 Series processors use a Non-Uniform Memory Access (NUMA) memory architecture. The four logical quadrants in an AMD EPYC 7002 Series processor, as described in the previous section, allow the processor to be partitioned into different NUMA domains. These domains are designated as NUMA per socket (NPS).

Using BIOS settings, each server can be configured as NPS1, NPS2, NPS4, or NPS0 (not recommended), with an additional option to configure L3 cache as NUMA (L3CAN).

AMD EPYC 7002 Series processors are available in different core counts per processor and not all of them can support all NPS settings (for CPUs with 6 CCDs per socket NPS=4 is not available, only NPS=1 or 2):

https://developer.amd.com/wp-content/resources/56338_1.00_pub.pdf

3.1 NPS=1

NPS1 indicates a single NUMA node per socket. This setting configures all memory channels on the processor into a single NUMA domain, i.e. all the cores on the processor, all memory connected to it and all PCIe devices connected to the processor are in one NUMA domain. Memory is then interleaved across all eight memory channels on each processor.

3.2 NPS=2

In NPS2, the processor is partitioned into two NUMA domains. Half the cores and half the memory channels of each processor are grouped together into one NUMA domain, and the remaining cores and memory channels are grouped into a second domain. Memory is interleaved across the four memory channels in each NUMA domain.

3.3 NPS=4

NPS4 partitions the processor into four NUMA domains. Each logical quadrant of the processor is configured as its own NUMA domain. Memory is interleaved across the two memory channels in each quadrant. PCIe devices will be local to one of the four NUMA domains on the processor depending on the quadrant of the IO die that has the PCIe root for that device.

3.4 NPS=0

NPS = 0 interleaves memory access all memory channels on a 2-socket system. This configuration should not be used for HPC workloads, as it adds inter-socket latency to every memory access.

3.5 L3 Cache as NUMA Domain

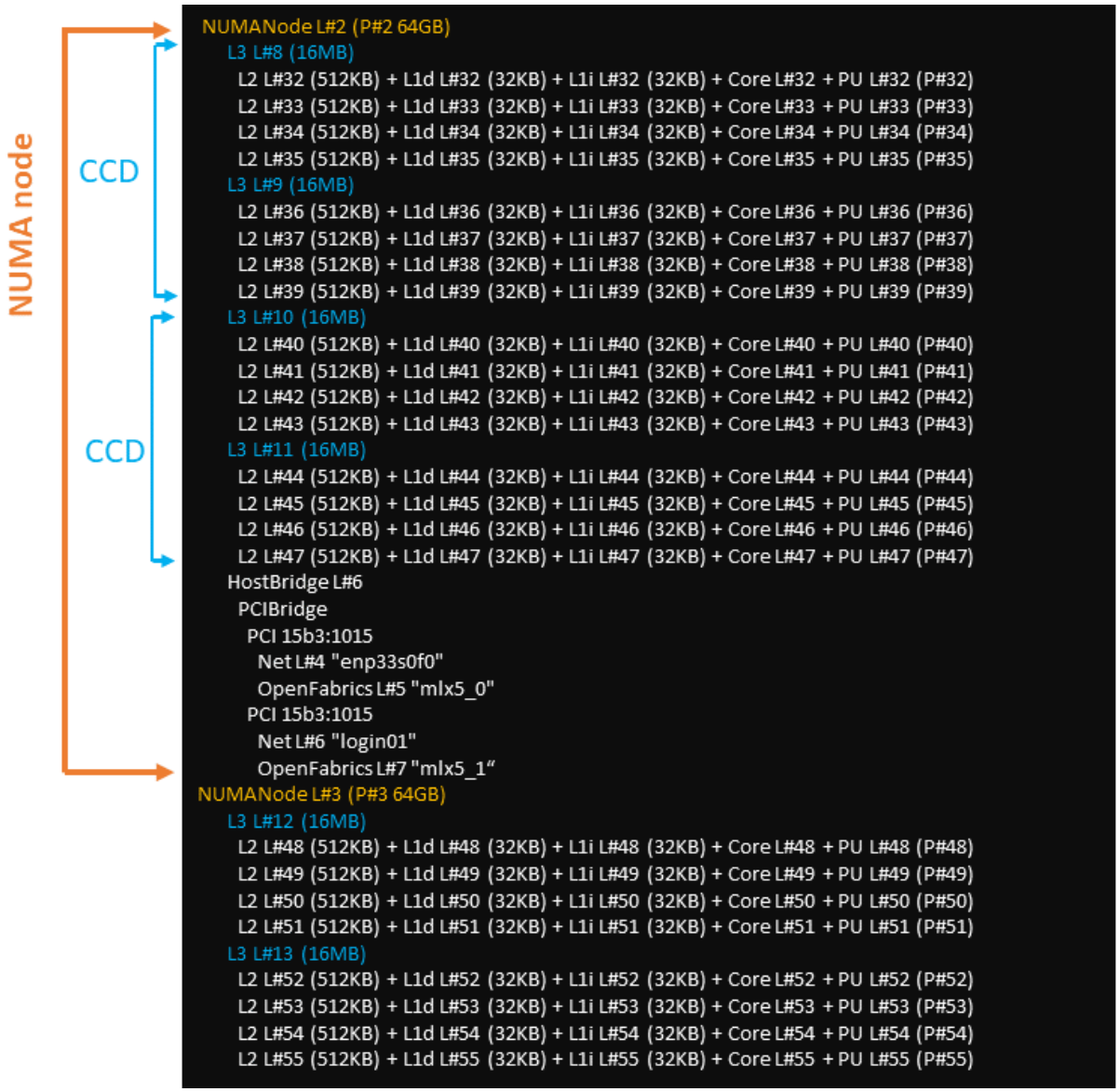
With L3 Cache As NUMA (L3CAN), each L3 Cache is exposed as its own NUMA node. On a dual processor system, with up to 16x L3 Caches per processor, this setting will expose up to 32 NUMA domains.

3.6 NUMA Per Socket (NPS) and Memory Bandwidth

Memory bandwidth reduces with fewer CCDs and is clearly demonstrated with the synthetic test, STREAM. However for real HPC workloads which have more random memory access patterns then NPS=1 or 2 can provide very similar peak performance to NPS=4 (see Chapter below on "Strategy for Characterizing 7002 'Rome' CPUs" for an example with the molecular dynamics application NAMD)

3.7 Understanding hwloc-ls and hwloc-info

This section explains how the NUMA nodes, caches and cores are seen by the operating system. The example below shows the output of `hwloc-ls` on a dual socket system with 64 cores per socket configured with NPS=4 running CentOS 7.6.



This example illustrates several important points: (Note that SMT is not enabled on this system)

- 4 cores per L3 cache are shown grouped together along with the CPU IDs associated with each 16 MB L3 Cache, i.e. 32,33,34,35. These groupings of 4 cores per L3 represents a single CCX.
- 2 CCXs per CCD (shown grouped in the diagram by the blue brackets to the left of the output) are not shown logically grouped by `hwloc-ls` because they are logically separate, even though they physically reside on the same CCD.
- The 4 CCXs are shown logically grouped under each NUMA node.
- Each NUMA node consists of 64GB of memory.
- There is a Mellanox HCA attached to the PCIe lanes associated with this NUMA node. CPU IDs 32-47 are logically 'closest' to this Mellanox network card.

`hwloc-ls` is a very useful tool for obtaining CPU IDs when you need to be aware of what cores to pin to your job to. You can also use `numactl -H` which will provide a list of cores per NUMA Node.

The above `hwloc-ls` output demonstrates the case for a single thread per core. The example below shows the output when Simultaneous Multi-Threading (SMT) is enabled within the BIOS, enabling a second thread on each core.

```
Machine (512GB total)
  NUMANode L#0 (P#0 64GB)
    Package L#0
      L3 L#0 (16MB)
        L2 L#0 (512KB) + L1d L#0 (32KB) + L1i L#0 (32KB) + Core L#0
          PU L#0 (P#0)
          PU L#1 (P#128)
        L2 L#1 (512KB) + L1d L#1 (32KB) + L1i L#1 (32KB) + Core L#1
          PU L#2 (P#1)
          PU L#3 (P#129)
        L2 L#2 (512KB) + L1d L#2 (32KB) + L1i L#2 (32KB) + Core L#2
          PU L#4 (P#2)
          PU L#5 (P#130)
        L2 L#3 (512KB) + L1d L#3 (32KB) + L1i L#3 (32KB) + Core L#3
          PU L#6 (P#3)
          PU L#7 (P#131)
      L3 L#1 (16MB)
        L2 L#4 (512KB) + L1d L#4 (32KB) + L1i L#4 (32KB) + Core L#4
          PU L#8 (P#4)
          PU L#9 (P#132)
        L2 L#5 (512KB) + L1d L#5 (32KB) + L1i L#5 (32KB) + Core L#5
          PU L#10 (P#5)
          PU L#11 (P#133)
        L2 L#6 (512KB) + L1d L#6 (32KB) + L1i L#6 (32KB) + Core L#6
          PU L#12 (P#6)
          PU L#13 (P#134)
        L2 L#7 (512KB) + L1d L#7 (32KB) + L1i L#7 (32KB) + Core L#7
          PU L#14 (P#7)
          PU L#15 (P#135)
```

Typically, on a dual socket system with 2x 64 core CPUs with SMT enabled, the first thread on each core is within the range 0-127, while the second thread will have CPU IDs within the range 128-255. In the above example you can see how core-0 has two threads associated with it: one with CPU IDs 0 and one with CPU ID 128 attached to it.

If installing older Linux kernels, you will also need to ensure it recognizes the correct cache hierarchy in the system, use `hwloc-info:`

```
$ hwloc-info
depth 0:         1 Machine (type #1)
 depth 1:        2 Package (type #3)
  depth 2:       8 NUMANode (type #2)
   depth 3:      32 L3Cache (type #4)
    depth 4:     128 L2Cache (type #4)
     depth 5:    128 L1dCache (type #4)
      depth 6:   128 L1iCache (type #4)
       depth 7: 128 Core (type #5)
        depth 8:         128 PU (type #6)
Special depth -3:        11 Bridge (type #9)
Special depth -4:        8 PCI Device (type #10)
Special depth -5:        12 OS Device (type #11)
```

Older kernels may misalign or completely ignore L3, and this will be evident using either `hwloc-info` or `hwloc-ls`.

3.8 C-States

There are several Core-States, or C-States, that the CPU can idle within. The root user has some control over which of these CPU states the processor uses.

- 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.

User root can enable and disable C-States. As an example, here is the output of `cpupower monitor` with all cores generally idling in C2.

```
# cpupower monitor
               |Mperf                 || Idle_Stats
PKG  |CORE|CPU | C0    | Cx    | Freq || POLL  | C1    | C2
   0|   0|   0|  0.03| 99.97|  1937||  0.00|  0.00| 99.97
   0|   0|   8|  0.22| 99.78|  1973||  0.00|  0.00| 99.80
   0|   0|  16|  0.01| 99.99|  1935||  0.00|  0.00| 100.0
   0|   0|  24|  0.00|100.00|  1703||  0.00|  0.00| 100.0
   0|   0|  64|  0.00|100.00|  1854||  0.00|  0.00| 100.0
   0|   0|  72|  0.00|100.00|  1649||  0.00|  0.00| 100.0
   0|   0|  80|  0.00|100.00|  1694||  0.00|  0.00| 100.0
   0|   0|  88|  0.00|100.00|  1712||  0.00|  0.00| 100.0
   0|   1|   1|  0.01| 99.99|  1824||  0.00|  0.00| 100.0
   0|   1|   9|  0.01| 99.99|  1720||  0.00|  0.00| 100.0
   0|   1|  17|  0.00|100.00|  1758||  0.00|  0.00| 100.0
...etc ....
```

Here is an example of disabling C2 (-d 2) for cores 0 to 3 (-c 0-3), which prevents them from idling down into C2, keeping them in state C1 or above.

```
cpupower -c 0-3 idle-set -d 2
```

```
# cpupower -c 0-11 monitor
                |Mperf                || Idle_Stats
PKG |CORE|CPU | C0   | Cx    | Freq || POLL | C1    | C2
   0|   0|   0|  0.01| 99.99|  1996||  0.00| 99.99|  0.00
   0|   0|   8|  0.20| 99.80|  1938||  0.00|  0.00| 99.75
   0|   1|   1|  0.00|100.00|  1896||  0.00| 99.96|  0.00
   0|   1|   9|  0.00|100.00|  1675||  0.00|  0.00| 99.95
   0|   2|   2|  0.00|100.00|  1856||  0.00| 99.96|  0.00
   0|   2|  10|  0.00|100.00|  1658||  0.00|  0.00| 99.97
   0|   3|   3|  0.00|100.00|  1872||  0.00| 99.96|  0.00
   0|   3|  11|  0.00|100.00|  1680||  0.00|  0.00| 99.97
   0|   4|   4|  0.00|100.00|  1690||  0.00|  0.00| 99.96
   0|   5|   5|  0.00|100.00|  1665||  0.00|  0.00| 99.96
   0|   6|   6|  0.00|100.00|  1661||  0.00|  0.00| 99.96
...etc ....
```

The power gated C2 idle state can be re-enabled (-e 2) on these 4 cores with:

```
cpupower -c 0-3 idle-set -e 2
```

These settings can be applied to all cores by omitting the `-c` argument in the `cpupower idle-set` command. With SMT enabled a core cannot enter C2 if either thread is in the C0 (active) state or C1 idle state. Therefore, if disabling C2 on any logical CPU, you should also disable C2 on the SMT sibling (e.g. in a 2x32 core system if you disable C2 on CPU 7 you should also disable C2 on CPU 71).

Disabling C2 is important for running a high performance, low latency network such as Infiniband. The latency required in moving from the power gated C2 state to the active C0 state can add latency to network IO. All cores must idle in C1 (with C2 disabled) to support a high-performance network.

3.9 P-States, Frequencies and Boosting

P-States are execution power states that manage power usage while the cores are active, like how C-States manage the power levels when the cores are idle. Higher P-states are only achievable when the processor is in the C0 "active" state.

- P-States are execution power states
- C-States are idle power saving states

Once active in the C0 state, a core can move between its various P-states based on its utilization to balance performance and power usage. When a core is fully utilized, it will target P0, which is its quoted base frequency.

User root can observe these P-States by executing

```
cpupower frequency-info
```

The example below shows the output for a 7742 CPU which shows the three P-states at frequencies of 1.5GHz (P2), 2.0GHz (P1) and 2.25GHz (P0).

```
[root@mysystem ~]# cpupower frequency-info
analyzing CPU 0:
  driver: acpi-cpufreq
  CPUs which run at the same hardware frequency: 0
  CPUs which need to have their frequency coordinated by software: 0
  maximum transition latency:  Cannot determine or is not supported.
  hardware limits: 1.50 GHz - 2.25 GHz
  available frequency steps:  2.25 GHz, 2.00 GHz, 1.50 GHz
  available cpufreq governors: conservative userspace powersave ondemand
performance
  current policy: frequency should be within 1.50 GHz and 2.25 GHz.
                  The governor "performance" may decide which speed to use
                  within this range.
  current CPU frequency: 2.25 GHz (asserted by call to hardware)
  boost state support:
    Supported: yes
    Active: no
    Boost States: 0
    Total States: 3
    Pstate-P0:  2250MHz
    Pstate-P1:  2000MHz
    Pstate-P2:  1500MHz
```

Enabling Boost allows a core to achieve yet another, higher frequency. This frequency is quoted as the Boost frequency for a CPU. Max Boost is the max frequency that any core can boost up to, provided there is enough thermal and computational headroom in the floating-point unit.

From the example above, the 7742 CPU has a quoted base frequency of 2.25 GHz. With Boost enabled, the cores of a 7742 can boost up to its quoted Boost frequency of 3.4 GHz. If all cores within a CPU attempt to boost this high, the overall CPU will likely reach a thermal power limit, thus reducing the effective frequency to range between the base and boost frequencies.

To benefit from Boost, it must be enabled. This can be set either in the BIOS or, for example, on a Red Hat Linux command line as root by issuing (Boost needs to be set to ENABLED in the BIOS in order to allow toggling on and off from the command line):

```
echo 1 > /sys/devices/system/cpu/cpufreq/boost
```

It can be returned to off with:

```
echo 0 > /sys/devices/system/cpu/cpufreq/boost
```

Changing the Boost setting in BIOS will require a system reboot.

For HPC workloads we recommend setting the CPU governor to performance. The user can observe this by running `cpupower monitor` or the AMD Micro Profiler (uProf) command line interface as root. (See the later section on CPU Governors for more information.)

3.10 CPU Governors

AMD EPYC supports several CPU governors. Different governors can be applied to different cores. For a High-Performance Computing environment, the 'performance' governor is often widely used:

- performance: this sets the core frequency to the highest available frequency within P0. With Boost set to OFF it will operate at the base frequency, e.g. 2.25GHz on a 7742 CPU. If Boost is ON, then it will attempt to boost the frequency up to the Max Boost frequency of 3.4Ghz. While operating at the boosted frequencies this still represents the P0 P-state.
- ondemand: sets the core frequency depending on *trailing* load. This favors a rapid ramp to highest operating frequency with a subsequent slow step down to P2 when idle. This could penalize short-lived threads.
- conservative: Similar to ondemand but favors a more graceful ramp to highest frequency and a rapid return to P2 at idle.
- powersave: sets the lowest supported core frequency, locking it to P2.

Administrators can set the CPU governor via the `cpupower` command. Here is an example of setting the CPU Governor to Performance:

```
cpupower frequency-set -g performance
```

A more extensive discussion and explanation around CPU governors within Linux can be found on kernel.org, https://www.kernel.org/doc/Documentation/cpu-freq/governors.txt

3.11 Useful 'cpupower' Command Examples

The `cpupower` command is a very useful utility for querying and setting a range of conditions on the CPU. We list some examples here:

```
cpupower -c 0-15 monitor
```

Displays the frequencies on cores 0 to 15. Useful if a user needs to observe the changes while turning Boost ON and OFF.

```
cpupower frequency-info
```

Lists the boost state, CPU governor, and other useful information about the CPU configuration.

```
cpupower frequency-set -g performance
```

Changes the CPU governor to 'performance'.

```
cpupower -c 0-15 idle-set -d 2
```

Disables the C2 idle state on CPUs 0 to 15.

Please refer to https://www.kernel.org/doc/Documentation/cpu-freq/governors.txt for more detailed information on CPU governors in general.

Chapter 4 Quick Reference High-Performance Set-Up

This section contains a quick reference summary of tuning tips which can serve as a quick starting point for BIOS and OS setting for an HPC system.

A reader unfamiliar with the AMD EPYC processor's architecture may find it beneficial to read additional sections of this document before proceeding.

4.1 Quick Reference: BIOS and OS

BIOS Settings:

- SMT = OFF | ON (application-dependent whether it provides advantage)
- IOMMU = OFF
  NOTE: Special case for 2 x 64 cores with SMT=ON (i.e. 256 threads):
  - IOMMU=ON and recommend **`iommu=pt`** as kernel boot parameter
  - X2APIC=AUTO
- APBDIS = 1 ; and Fixed SOC P-state = P0
- Core Performance Boost = ON
- Determinism Slider = Power
- cTDP = 240 (set to the max cTDP setting of your processor, e.g. 7742 is 240W)
- PPL = 240 (set to the max cTDP setting of your processor, e.g. 7742 is 240W)
- NPS = 4
  - Or with 6x CCDs use NPS=2 (NPS=4 forbidden if the CPU has 6 CCDs)
- DF C-States = Disabled
- Preferred-IO Control = Manual
  - Preferred-IO Device = <Bus number from Infiniband card: use **`lspci`** >
  - Enhanced Preferred-IO Mode = Enabled
- Core C-states = Enabled
- TSME = OFF

Platform vendors may include Workload Profiles or System Tunings for HPC. These may disable C states (not recommended) and/or disable Core Performance Boost (also not recommended). If these two settings cannot be reverted in the workload profile, use a custom setting instead.

OS Settings:

- For a HPC cluster with a high-performance low latency interconnect such as Mellanox disable the C2 idle state. Assuming a dual socket 2x64 core system

```
cpupower -c 0-127 idle-set -d 2
```

- Set the CPU governor to 'performance':

```
cpupower frequency-set -g performance
```

Use these settings establish a performance baseline. Once established, developers are then encouraged to pursue further system tuning based on this baseline.

4.2 Quick Reference: Basic System Checks

- Check if SMT is enabled [Thread(s)=1 implies SMT=OFF; Threads (2)=2 implies SMT=ON]:

  ```
  lscpu
  ```

- Check which NUMAnode your Infiniband card or other peripherals are attached to:

  ```
  hwloc-ls
  ```

- Check if boost is ON (1) or OFF (0) :

  ```
  cat /sys/devices/system/cpu/cpufreq/boost
  ```

- Check CPU governor and other useful settings:

  ```
  cpupower frequency-info
  ```

- Visually check which cores/threads are busy

  ```
  htop
  ```

- Check frequencies and idle states on cores

  ```
  cpupower monitor
  ```

- Run STREAM: Dual CPU Socket, DDR4-3200 Dual Rank, 1 DIMM slot per channel (1 core per L3 on 64 core CPU) should yield approximately 350GB/s with Intel or AOCC compiler

4.3 Other Tips

Build CPU ID lists with 'seq'

It may be necessary to build comma-delimited lists of CPU IDs at various times. Use of the Linux command `seq` is recommended:

System with 2x 64 core CPU, 2 cores per L3:

```
seq -s , 0 2 127
```

System with 2x 64 core CPU, 3 cores per L3 (join 2 lists, every second core + every fourth core):

```
seq -s , 0 2 127 | tr -d '\n';  echo ,|  tr -d '\n';  echo
"$(seq -s , 1 4 127)"
```

Core Pinning and Memory Locality:

You can pin your binary to run on a specific core, e.g. core 7:

```
numactl -C 7 ./mybinary
```

This places the thread, but still allows the kernel to freely choose which memory slots to use. To ensure the memory is allocated logically closest to the core, then include `'--membind='`

```
numactl -C 7 --membind=0 ./mybinary
```

To establish which NUMA Node your core belongs to, use the output from `numactl -H`

```
node 0 cpus: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
node 0 size: 65422 MB
node 0 free: 296 MB
node 1 cpus: 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
node 1 size: 65535 MB
node 1 free: 40 MB
node 2 cpus: 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47
node 2 size: 65535 MB
node 2 free: 3602 MB
node 3 cpus: 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63
node 3 size: 65523 MB
node 3 free: 41 MB
node 4 cpus: 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79
node 4 size: 65535 MB
node 4 free: 321 MB
```

Faster 'make' with 'make -j'

A common way to build code from a source tar ball is with

```
./configure ; make ; make install
```

The 'make' step will implicitly use a single core for the compilation. With AMD EPYC, you can significantly speed up compilation by passing the `-j` flag with a number representing the maximum number of threads you would like the compiler to use. For example, on a dual socket 2x 64 core system with SMT=ON you now have 256 threads and could use the following:

```
./configure ; make -j 256 ; make install
```

For example: core with CPU ID 7 is in NUMA domain 0

This allows the compiler to leverage up to 256 threads for compilation. For large code sources this will dramatically reduce build time.

Chapter 5 BIOS Settings

This section describes common BIOS settings within a High-Performance Computing environment. For a more detailed information on of these settings see the Workload Tuning Guide.

5.1 Recommended BIOS Settings for Bare Metal Workloads

The guidelines below are intended to get you started with the tuning. Make sure to evaluate your needs and apply the appropriate settings that are best for your requirements. Check your server manufacturer's guide if you cannot find the exact options below. These options represent changes from the default BIOS settings on AMD reference platforms, but not necessarily on every server platform.

- x2APIC → Enabled
- SMT → Disabled
- NPS → 4 *(Set Memory Frequency Properly)*
- APBDIS → 1
- Fixed SOC Pstate → P0
- Preferred IO

- Determinism Slider = Performance | Power
  Power and Perform determinism mode (System management Unit (SMU) policy choices. The effective GHz each core will run at is under the control of the SMU. This tracks power, current draw and temperatures across all parts of the processor and has knowledge of the cTDP power budget.

  '*Performance*': reduces CPU to CPU performance variability within a cluster. Provides repeatable performance for identical SKUs across your cluster by using a lowest baseline reference setting common to all CPUs within that SKU range.

  '*Power*': takes advantage of yield variability and will provide improved runtime performance and guarantees the power draw of each part will be up to but not exceed the cTDP power limit. If you wish to set the cTDP to its maximum value, then you must set 'Determinism Slider=Power'

- cTDP (configurable Thermal Design Point)
  Ensure cTDP=PPL in BIOS. Each CPU has a minimum and maximum cTDP threshold.

- PPL (Package Power Limit)
  Every CPU has a maximum PPL limit. Ensure PPL=cTDP. PPL can be set lower than the cTDP minimum
- SMT (Simultaneous Multi-Threading)
  When enabled each core will exhibit two threads. Users can refer to hwloc-ls on the command line to obtain a list of CPUs IDs for each thread. In HPC workloads the SMT is typically turned off. If you are not in a compute bound scenario you may see some benefit from SMT. If your code is not licensed per core or if there is no monetary impact for enabling SMT you may want to experiment to see if it is beneficial for your workload.

- Enabled: This allows 1 core to execute 2 threads.
- Disabled: This allows 1 core to execute 1 thread.

The benefit of this setting is application dependent. Note that for a dual socket platform with 2x64 and SMT=ON users also need to set the following:

- IOMMU=ON and recommend **`iommu=pt`** as kernel boot parameter
- X2APIC=AUTO

- X2APIC
  This option helps the operating system deal with interrupts more efficiently in high core count configurations. This option must be enabled if using > 255 threads and only needed when we need SMT=ON, on a 64 core CPU SKU.

- APBDIS (Algorithmic Performance Boost Disable)
  1 = enabled (i.e. APB is disabled)
  0 = disabled (i.e. APB is enabled)

  This setting governs boost behavior of the Infinity Fabric on EPYC. For HPC workloads we recommend a fixed frequency i.e. not boosting and therefore set APBDIS state to '1'. In some OEM BIOS' once this is set to 1 user will see a new option appear fixed SOC P-state which needs to be set to P0 (memory 'p-states' are ne to Rome. P0 the highest performance memory p-state)

- Core Performance Boost = OFF | ON
  Turns the boost function ON or OFF on all cores. This can also be toggled on or off via the Linux command line as root in RHEL/CentOS for example (this requires Core Performance Boost to already be enabled/on in the BIOS at boot up. If it set to OFF in BIOS, it cannot be toggled on the Linux command line).

- Memory speed = AUTO
  AUTO will allow the system to automatically train to the correct speed setting for a given DIMM population and memory rank. Users can clock this down if they wish to, e.g. for applications that are not sensitive to memory speed, and therefore save on power and provide greater boost headroom

- Core C-States = Enabled
  Refers to CPU C-states. Leave these enabled. If required, users should disable C2 via the command line as root (see later)

- DF C-States (Data Fabric C-States)
  In case of long idle periods the Infinity fabric can enter lower C-States to save power. For HPC workloads this should be disabled. If core C2 is disabled from the OS this setting does not matter.

- NPS = 1 | 2 |4

Sets the NUMA domains Per Socket by interleaving pairs of memory channels. . In many HPC applications, ranks and memory can be pinned to cores and NUMA nodes, and the typical recommendation in this case is to use the NPS4 option. If your workload is not very well NUMA aware or suffers when NUMA complexity increases, you can experiment with NPS1.

- Preferred-I/O Control
  For systems with a single Mellanox PCI card this needs to be enabled when using a high performance low-latency interconnect such as Mellanox's Infiniband fabric. This setting will
  1) Provide enhanced priority to a single PCI device [in BIOS: Preferred-IO Device]
  2) Increase the PCI clock 'LCLK' [in BIOS: Enhanced Preferred-IO Mode]
  Depending on OEM BIOS it will ask for either the PCI device or PCI slot. Use lspci in advance to determine which PCI device the Mellanox card is hosted on, for example:

  This requires a BIOS based on AMDs AGESA BIOS v 1.0.0.5 and later. For a BIOS based on an earlier AGESA please consult your System Integrator.

```
[jason@mysystem ~]$ lspci | grep -i Mel
c1:00.0 Infiniband controller: Mellanox Technologies MT28908 Family [ConnectX-6]
c1:00.1 Infiniband controller: Mellanox Technologies MT28908 Family [ConnectX-6]
```

- CCD Control
  This option allows the user to modify the number of active CCDs in the processor. It can be used in combination with Core Control to change the effective layout of the part.
- Down-coring / Core control
  This option allows you to modify the number of active cores in a CCX. The options are listed as (x+x) where x is the number of active cores per ccx. For example: Setting this to (2+2) means there are 2 active cores per CCX and 4 active cores per CCD. If the part has 8 CCDs, then you have a total of 32 cores. 4 cores per CCD * 8 CCDs = 32 total cores.

  Note: CCD and Core control options are typically used to simulate the behavior of different part configurations with your workload. This experiment can help you to identify the best part configuration for your workload and needs.

  Sets the number of active cores per L3 cache. For a 64-core part AUTO will leave 4 cores active per L3. Other options would be:
  - 3-3: 3 cores active per CCX, i.e. turn off 1 core per CCX
  - 2-2: 2 cores active per CCX, i.e. turn off 2 cores per CCX
  - 1-1: 1 core active per CCX, i.e. turn off 3 cores per CCX

  Users may wish to disable cores to maximize the L3-cache per core ratio on certain codes.

- Memory Frequency, Infinity Fabric Frequency, and coupled vs uncoupled mode
  - The Memory clock and the Infinity Fabric clock can either run at synchronous frequencies, called coupled mode, or at asynchronous frequencies, called uncoupled mode.

- AMD EPYC supports DDR4 frequencies up to 3200 MT/s, however the fabric clock can be synchronous to a maximum speed of 2933 MT/s (or 2667 MT/s, for lower-power Group B infrastructure parts).
- If your memory is clocked at or lower than 2933 MT/s, the memory and fabric will always run in coupled mode which will provide the lowest memory latency.
- If you are running DDR4 memory at 3200 MT/s, the memory and fabric clocks will run in uncoupled mode. This provides slightly higher bandwidth at the cost of increased memory latency.
- If your system supports 3200 MT/s memory, you can experiment with coupled mode at 2933 MT/s and uncoupled mode at 3200 MT/s to determine which is best for your workload.
- In the BIOS, set your memory frequency to the desired speed and make sure APBDIS is set to 1 and fixed SOC Pstate is set to P0.

- Preferred IO
  Preferred IO allows one PCIe device in the system to be configured in a preferred state. This device gets preferential treatment on the infinity fabric. This is typically enabled for fabric adapters that provide the interconnect between systems.
- TSME = OFF

  Secure Memory Encryption (encrypts all the memory)

Chapter 6 Operating Systems

6.1 Linux Kernel Considerations

Since the launch of EPYC there have been several patches issued specifically in relation to EPYC. You can either choose to apply these patches manually or deploy an OS with a suitably up--to-date kernel to avoid manual patching.

Red Hat / CentOS

Use at least RHEL/Centos v 7.6 with kernel 3.10-957.

Kernel.org

At least kernel 4.19. Patches relating to Spectre and Meltdown entered the kernel at version 4.15.

SUSE

SLES 12P4 and 115P1 are both support AMD EPYC ROME.

6.2 /proc and /sys

There are several ways memory can be consumed over the uptime of a system. The following provides a summary of some of the areas that you should be familiar with in relation to NUMA systems that affect both performance and which enable memory 'clean-up'. A more rigorous and thorough discussion on the Linux Virtual Memory system is available through the Kernel Documentation (https://www.kernel.org/doc/Documentation/sysctl/vm.txt)

/proc/sys/vm/zone_reclaim_mode

From kernel.org, "*Zone_reclaim_mode allows someone to set aggressive approaches to reclaim memory when a zone runs out of memory. If it is set to zero, then no zone reclaim occurs. Allocations will be satisfied from other zones / nodes in the system.*"

The kernel behavior is controlled by the following three bits:

```
1 = Zone reclaim on
2 = Zone reclaim writes dirty pages out
4 = Zone reclaim swaps pages
```

These can be logically OR'ed, i.e. a setting of 3 (1+2) is permitted.

For workloads that benefit from having file system data cached, zone reclaim is usually turned off; but it is a balance and if job size exceeds the memory of a NUMANode, and/or your job is multi-cored and extends outside the NUMANode then it is probably sensible to turn this on. Read this article for more details → :https://www.kernel.org/doc/Documentation/sysctl/vm.txt

Note all 2P EPYC systems implement Zone Reclaim with respect to the remote socket. The recommended setting for this setting is a value of 1 or 3.

/proc/sys/vm/drop_caches

After running applications in Linux, you may find that your available memory reduces while your buffer memory increases, despite not running any applications, e.g.

```
[root@mun-smc05 ~]# free -g
              total        used        free      shared  buff/cache   available
Mem:            251           0         233           0          17         248
Swap:             3           0           3
```

Issuing **`numactl -H`** will show which NUMANode(s) the memory is buffered with (possibly all).

In Linux users can clean the caches in 3 ways to return buffered or cached memory to 'free':

`echo 1 > /proc/sys/vm/drop_caches` [frees page-cache]
`echo 2 > /proc/sys/vm/drop_caches` [frees slab objects e.g. dentries, inodes]
`echo 3 > /proc/sys/vm/drop_caches` [cleans page-cache and slab objects]

Users will need to be root or have SUDO permissions to execute the above.

```
[root@mun-smc05 ~]# free -g
              total        used        free      shared  buff/cache   available
Mem:            251           0         250           0           0         249
Swap:             3           0           3
```

On HPC systems it is often useful to clean up the memory after a job has finished before the next user is assigned the same node. SLURM can accomplish this using an epilog script for example. This is especially important if running with BIOS option NPS=4 where old I/O memory caches buffers can easily consume all the memory in a single NUMA region and future allocations spill over to non-local memory.

```
cat /proc/sys/vm/swappiness
10
```

It is recommended to disable swap to prevent any unwanted swap usage. If you need to use swap you need to buy more memory capacity for your nodes.

Ensure your node have sufficient memory to work your workloads. Disabling swap without sufficient memory can have undesired effects.

```
swapoff -a
```

6.3 Transparent Huge Pages (THP)

Should hugepages be required, users can disable Transparent Huge Pages by

```
echo 'never' > /sys/kernel/mm/transparent_hugepage/enabled
echo 'never' > /sys/kernel/mm/transparent_hugepage/defrag
```

6.4 Hugepages

Transparent Huge pages could be disabled if the application provides support for explicit hugepages. Follow the application's guidance for allocating explicit hugepages at boot time. For example, for HPL there is a small gain to be achieved by disabling THP and enabling hugepages.

To reserve 240GB with 2m hugepages:

```
echo 3 > /proc/sys/vm/drop_caches
echo 1 > /proc/sys/vm/compact_memory
$number_huge_2m_pages = 240 * 1024 / 2
echo $number_huge_2m_pages > /proc/sys/vm/nr_hugepages
```

where $number_huge_2m_pages = 240 * 1024 / 2. Execute mybinary.bin using hugectl (with the reserved hugepages in place)

```
hugectl --force-preload –heap mybinary.bin
```

This has another advantage over THP: code will warn you if it runs out of huge pages or cannot allocate the hugepages the binary requires

6.5 Randomize_va_space

Address space randomization is a security feature and guards against security hacks. Disabled with

```
echo 0 > /proc/sys/kernel/randomize_va_space
```

6.6 NUMA balancing

NUMA balancing is a feature that allows the Operating System to scan memory and attempt to migrate to a DIMM that is logically closer to the cores accessing it. However, there is an overhead to scanning for 'poor NUMA location' pages; the OS is second guessing the users' NUMA allocations. This can be a be useful if the NUMA locality access is very poor.

For most HPC codes it can be advantageous to disable NUMA balancing as it will likely introduce variability across nodes. It also has a performance overhead. For example, STREAM memory bandwidth benchmark runs a little slower with NUMA balancing enabled. Disable with

```
echo 0 > /proc/sys/kernel/numa_balancing
```

More details can be found here:

https://documentation.suse.com/sles/15-SP1/html/SLES-all/cha-tuning-numactl.html

https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/virtualization_tuning_and_optimization_guide/sect-virtualization_tuning_optimization_guide-numa-auto_numa_balancing

6.7 Spectre and Meltdown

Google Project Zero (GPZ) announced in early 2018 several vulnerabilities concerning speculative execution that take three variants. AMD EPYC CPUs are not affected by 'Variant-3', also known as 'Meltdown'. AMD EPYC CPUs are affected by Variant-1 and Variant-2, 'Spectre'. A more complete discussion by the AMD Chief Technology Officer, on these vulnerabilities can be read here:

https://www.amd.com/en/corporate/speculative-execution-previous-updates#paragraph-337801

Newer kernels (see chapter on Linux Kernels in this guide) will have patches automatically applied to protect against these. However, it does come with a small effect on performance (a few percent). Some customers are electing to keep their 'edge nodes' or 'head' nodes [nodes that are connected from their organization to the outside] patched against these vulnerabilities but are electing to turn the patches off on those compute nodes that (logically) securely inside their organization.

Red Hat has described this process in detail: https://access.redhat.com/articles/3311301

In summary, root can turn these off by setting the single-entry value to '0' in two files:

```
echo 0 > /sys/kernel/debug/x86/retp_enabled
echo 0 > /sys/kernel/debug/x86/ibpb_enabled
```

You can then view the 2 files to check if the Spectre patches are now disabled:

```
cat /sys/kernel/debug/x86/retp_enabled
cat /sys/kernel/debug/x86/ibpb_enabled
```

Chapter 7 Libraries and Compilers

7.1 AOCC – AMD Compilers

AMD libraries, compilers, user guides are available on https://developer.amd.com. Downloading forms do not require signing in, when you accept the terms, and the download begins. Apart from OpenBLAS, everything in this chapter is available from the AMD developer portal.

AOCC refers to the AMD compiler suite; AOCL refers to the AMD Math Library suite. Both are under active development and are a focal point of AMDs strategy going forward to meet the needs of HPC users. The latest major update to AOCC and AOCL, version 2.0, was made available shortly prior to Rome launch. Version 2.1 was released January 2020 and includes several key updates to v2.0

AOCC consists of C/C++ compiler (clang) and a Fortran compiler (flang) and can be downloaded from

- AOCC (AMD Compilers) → https://developer.amd.com/amd-aocc/
- AOCL (AMD Libraries) → https://developer.amd.com/amd-aocl/

The AOCC compiler system is a high performance, production quality code generation tool. The AOCC environment provides various options to developers when building and optimizing C, C++, and Fortran applications targeting 32-bit and 64-bit Linux® platforms. The AOCC compiler system offers a high level of advanced optimizations, multi-threading and processor support that includes global optimization, vectorization, inter-procedural analyses, loop transformations, and code generation. AMD also provides highly optimized libraries, which extract the optimal performance from each x86 processor core when utilized.

- Tuned for AMD Family 17h processors
- Machine dependent optimizations for better performance in AMD EPYC 7xx2-series
- Enhanced high-level optimizations towards AMD EPYC 7xx2-series of architectures
- Improved Flang – as default Fortran front-end with added F2008 features
- Based on LLVM 9.0 release (llvm.org, 19th Sep 2019) with bug fixes
- Optimized libraries including AMDLibM (libM math library v3.3)
- LLVM linker (lld) as the default linker
- Tested on RHEL 7.4, SLES 12 sp3, Ubuntu 18.04 LTS, Ubuntu 19.04

Examples of useful compiler options to pass to clang (and therefore flag) are provided in the Makefiles for DGEMM and STREAM in the Appendices.

7.1.1 AOCC Clang

clang is a C, C++, and Objective-C compiler which encompasses preprocessing, parsing, optimization, code generation, assembly, and linking.

Clang supports the **`-march=znver2`** flag to enable best code generation and tuning for AMD's Zen2 7002-series 'Rome' based x86 architecture.

7.1.2 AOCC Flang

Flang is the Fortran front-end designed for integration with LLVM and suitable for interoperability with Clang/LLVM. It supports all the clang compiler options plus a number of flang-specific options.

AMD extends the GitHub version of flang (https://github.com/flang-compiler/flang) which in turn is based upon the Nvidia/PGI commercial Fortran compiler.

7.1.3 Useful AOCC Compiler Options

Examples of useful compiler options to pass to clang (and therefore flag) are provided in the Makefiles for DGEMM and STREAM in the Appendices.

General	
Generate instructions that run on 2nd generation EPYC	-march=znver2
Generate instructions for the local machine	-march=native
OpenMP threads and affinity (N number of cores)	export OMP_NUM_THREADS=N export GOMP_CPU_AFFINITY="0-(N-1):1"
Enable vector library	-vector-library=LIBMVEC
Link to vector library	-L/libm-install/lib -lmvec
Link to AMD library	-L/libm-install/lib -lamdlib
Other options	
Disable all optimizations	-O0
Minimal level speed and code optimization	-O1
Moderate level optimization (default)	-O2
Aggressive optimization	-O3
Maximize performance	-Ofast

Aggressive Optimization	
Enable aggressive optimizations	-lv-function-specialization -unroll-threshold=[50|100] -funroll-loops
Enable faster less precise math operations	-ffast-math -freciprocal-math
Compile free form FORTRAN	-ffree-form
Enable link time optimizations	-flto
Enable unrolling	-funroll-loops
Enable aggressive loop optimization	-enable-loop-versioning-licm -enable-loop-distribute -enable-partial-unswitch -unroll-aggressive
Enable aggressive inline optimizations	-function-specialize -finline-aggresive
Enable aggressive vectorization	-vectorize-memory-aggressively -enable-strided-vectorization
Enable memory layout optimizations	-fstruct-layout=[1|2|3|4|5] -fremap-arrays (use with -flto)
Profile Guided optimizations	-fprofile-instr-generate (1st invocation)
OpenMP	-fopenmp
Enable Streaming Stores	-fnt-store
Enable removal of un-used array computation	-reduce-array-computation=3

7.2 GCC Compiler

The default compiler that ships with RHEL/CentOS 7.6 is version 4.8.5. For HPC users this GCC compiler version often does not deliver the performance required in supercomputing environments. We have undertaken tests starting with the later GCC versions 7.3, 8.1, and 9.1 and used these to run and derive good performance on HPL, HPCG, and DGEMM.

7.3 Intel

AMD has performed limited testing on some codes with the Intel compiler v18, v19, and v20 initial release. We demonstrate in the appendices how to build several binaries for synthetic benchmarks on both compilers.

To enable Intel Math Kernel Libraries (MKL v19 and earlier) to issue AVX2 instructions to AMD CPUs, the following environment variables will need to be set in the runtime environment.

```
export MKL_DEBUG_CPU_TYPE=5
export MKL_ENABLE_INSTRUCTIONS=AVX2
```

Without these MKL will not use an AVX2 code path on EPYC 7002 series and DGEMM runs many times slower that it could do. (The Intel MKL library fails to correctly check the x86 ISA standard CPUID AVX/AVX2 ISA capability bits – processors x86 ISA capabilities are reported in /proc/cpuinfo.)

7.4 AOCL – AMD Math Libraries

AOCL are a set of numerical libraries tuned specifically for AMD EPYC™ processor family. They have a simple interface to take advantage of the latest hardware innovations.

More detail on the AOCL, tar balls of prebuilt libraries, and how to build AMD libraries from source is available at https://developer.amd.com/amd-aocl/

For issues concerning AOCL please contact your local AMD Field Application Engineer or send an email to toolchainsupport@amd.com

7.4.1 BLIS

BLIS is a portable open-source software framework for instantiating high-performance Basic Linear Algebra Subprograms (BLAS) – like dense linear algebra libraries. The framework was designed to isolate essential kernels of computation that, when optimized, immediately enable optimized implementations of most of its commonly used and computationally intensive operations. Select kernels have been optimized for the AMD EPYC processor family.

Source code is available on GitHub https://github.com/amd/blis

7.4.2 libFLAME

libFLAME is a portable library for dense matrix computations, providing much of the functionality present in Linear Algebra Package (LAPACK). It includes a compatibility layer, FLAPACK, the FORTRAN interface, which includes complete LAPACK implementation. The library provides scientific and numerical computing communities with a modern, high-performance dense linear algebra library that is extensible, easy to use, and available under an open source license. In combination with the BLIS library which includes optimizations for the AMD EPYCTM processor family, libFLAME enables running high performing LAPACK functionalities on AMD platforms.

Source code is available on GitHub https://github.com/amd/libflame

7.4.3 FFTW

FFTW is a comprehensive collection of fast C routines for computing the Discrete Fourier Transform (DFT) and various special cases thereof. It is an open-source implementation of the Fast Fourier transform algorithm. It can compute transforms of real and complex-values arrays of arbitrary size and dimension. An AMD optimized FFTW that includes selective kernels and routines optimized for the AMD EPYC™ processor family is available.

Source code is available on GitHub https://github.com/amd/amd-fftw

7.4.4 LibM

AMD LibM is a software library containing a collection of basic math functions optimized for x86-64 processor-based machines. It provides many routines from the list of standard C99 math functions.

Applications can link into AMD LibM library and invoke math functions instead of compiler's math functions for better accuracy and performance.

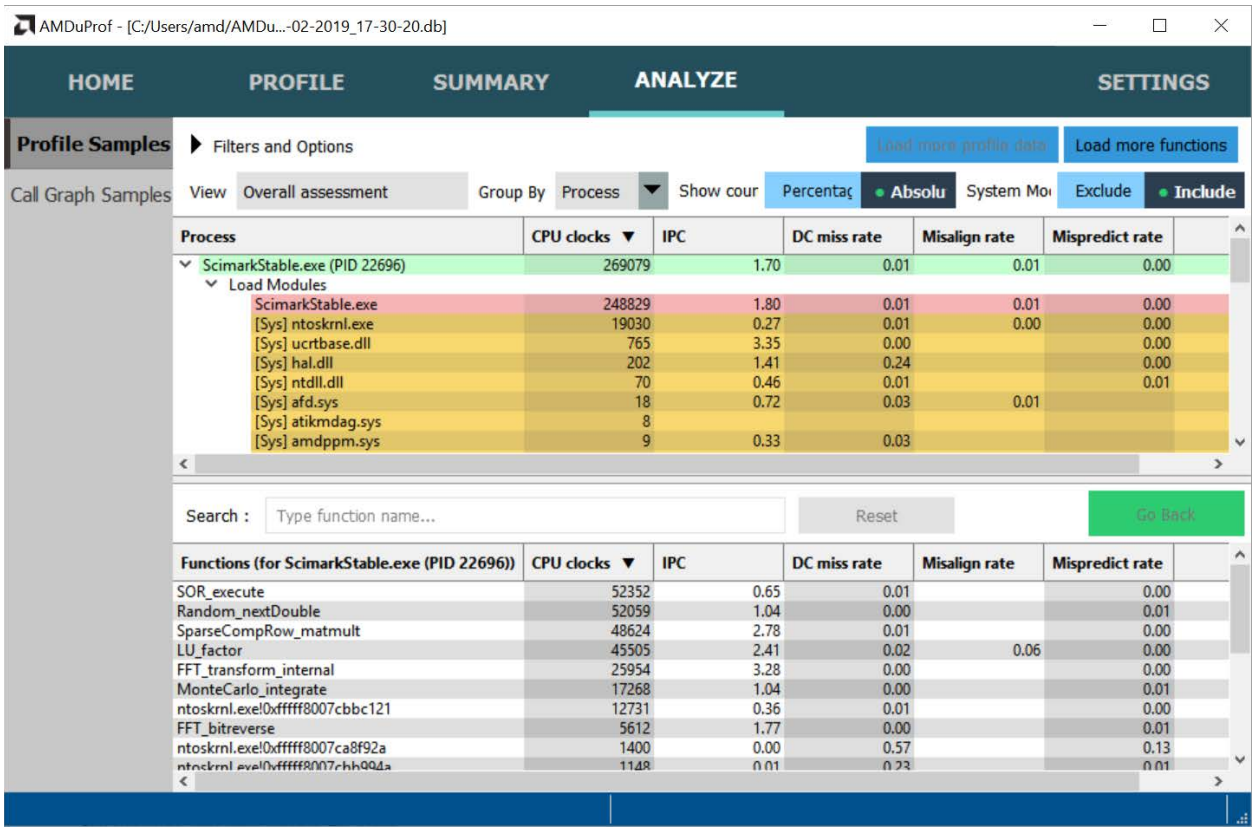
7.4.5 ScaLAPACK

ScaLAPACK is a library of high-performance linear algebra routines for parallel distributed memory machines. It depends on external libraries including BLAS and LAPACK for Linear Algebra computations. AMD's optimized version of ScaLAPACK enables using BLIS and libFLAME library that have optimized dense matrix functions and solvers for AMD EPYC processor family CPUs.

ScaLAPACK can be installed either from source or pre-built binaries.

ScaLAPACK for AMD source can be cloned from https://github.com/amd/scalapack. A pre-built AMD optimized ScaLAPACK can be installed from the AOCL master installer tar file.

7.5 uProf



AMD uProf is a performance analysis tool for applications running on Windows and Linux operating systems. It allows developers to better understand the runtime performance of their application and to identify ways to improve its performance. Users can download it for free from https://developer.amd.com/amd-uprof/

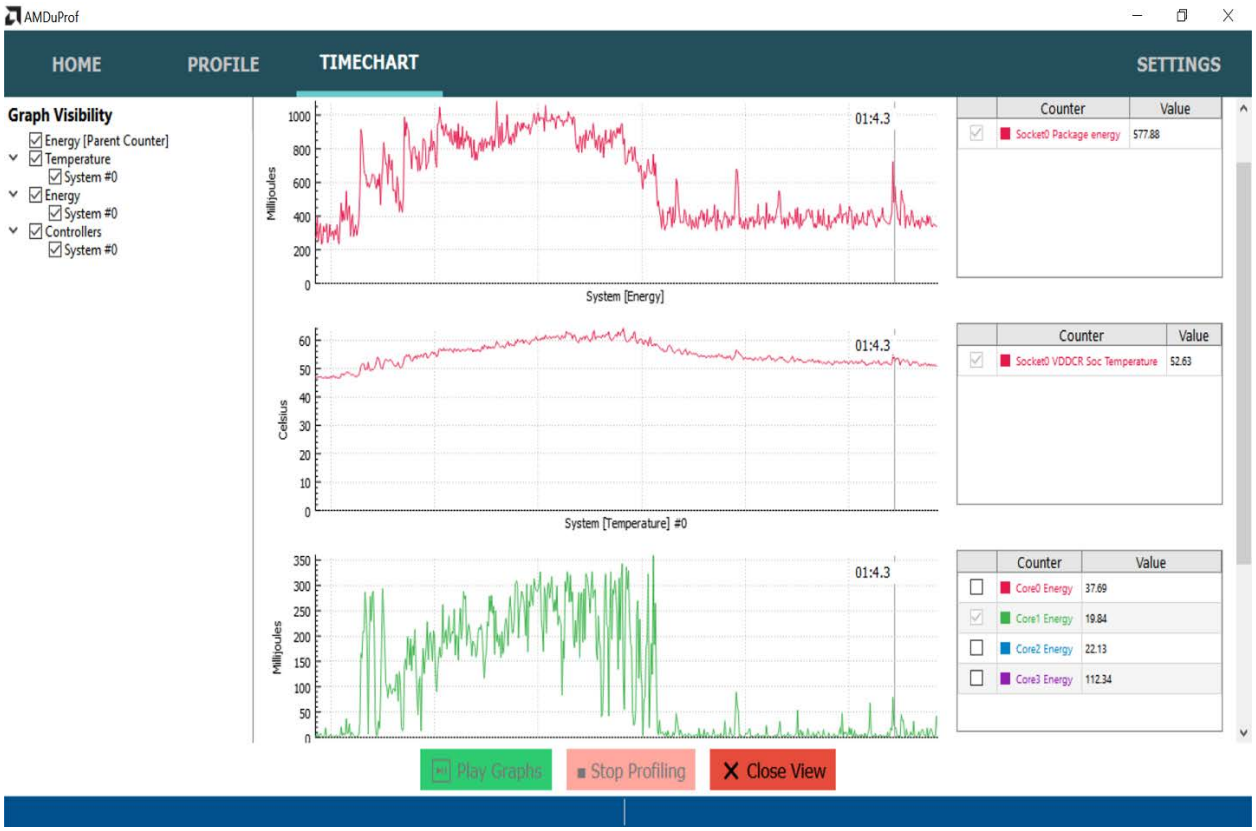
AMD uProf offers:

- Performance Analysis
  - CPU Profiling – to identify runtime performance bottlenecks of the application.
- Energy Analysis
  - Power Application Analysis – to identify energy hotspots in the application (Windows only).
- Power Profiling
  - System-wide Power Profiling – to monitor thermal and power characteristics of the system.
- System Analysis
  - Performance Counter Monitor utility – to monitor system performance metrics (Linux & FreeBSD only)

AMD uProf can effectively be used to:

- Analyze the performance of one or more processes or the entire system
- Track down the performance bottlenecks (hotspots & micro-architecture) in the source code
- Identify ways to optimize the source code for better performance and power efficiency
- Examine the behavior of kernel, drivers, and system modules
- Analyze Thread concurrency
- Observe frequency, thermal and power characteristics (Power profiling)
- Observe system metrics like IPC, Core effective frequency, memory bandwidth, etc.

AMD uProf profiler can be used to monitor the frequency, thermal and energy metrics of various components in the system. The GUI offers a live timeline graphs of various metrics in TIMECHART page.



Chapter 8 Executing Applications on AMD EPYC 7002 Series Processors

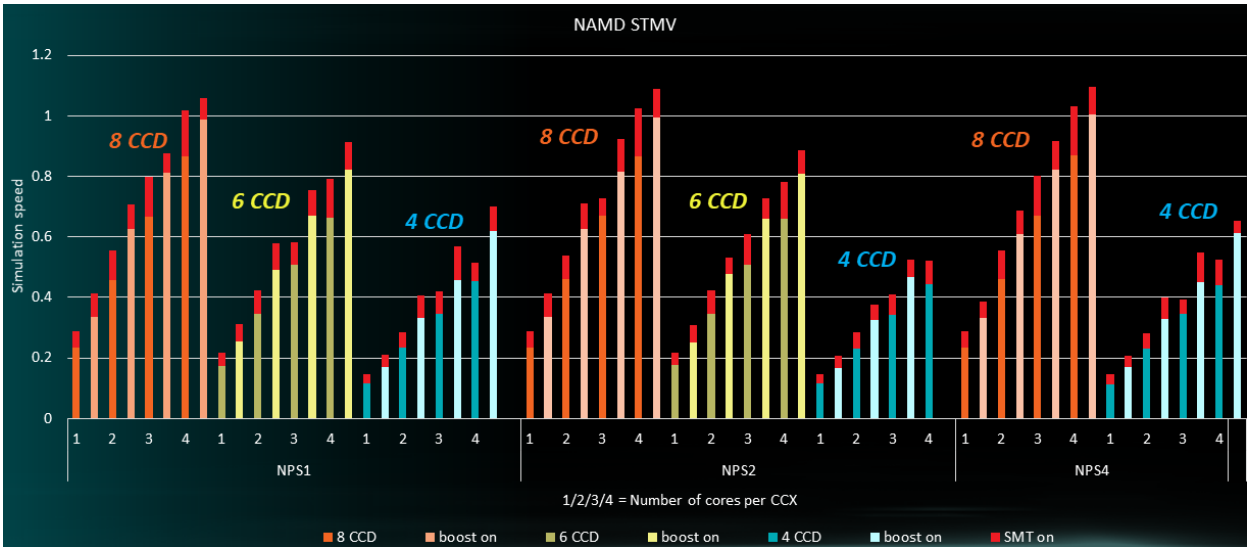
8.1 Strategy for Characterizing

When deciding which system settings work best for your application, several settings may need to be considered for tuning to get an optimal HPC server. The below example table below shows a matrix of SMT ON/OFF, Boost ON/OFF, across 4 different CPU core counts/configuration. This type of table can help guide your investigation and test plan.

Cores/L3	Cores / socket	SMT = OFF		SMT = ON	
		Boost=OFF	Boost=ON	Boost=OFF	Boost=ON
1	16				
2	32				
3	48				
4	64				

This specific example provides a method to understand the benefits of SMT, boost and also to understand the benefits of increasing core count per L3, for example there may be a tailing-off in performance beyond 3 cores and this can provide insight as to the most appropriate setting for executing that application, or even an entire procurement (however, more cores can also provide the benefit of greater capacity for bursting throughput at critical times). The above table could also be repeated for different NPS settings and number of CCDs.

The following is an example of characterizing the molecular dynamics application NAMD with the STMV test case (higher is better):



A server with 2x 64 core 7742 CPUs was used for the above characterization study using 1/2/3/4 cores per L3 (i.e. 16/32/48/64 cores per socket) in a system with 2x7742 CPUs. Using this CPU allows us the flexibility to test NPS=1/2/4 and reducing CCD count from 8 to 4 CCDs. This provides insight as to whether we would recommend a 64, or 48, or 32 core CPU as providing the optimum performance.

For a given core count per L3 (e.g. 1 core per L3) there are 2 bars: left bar refers to boost=off, right bar refers to boost=on. The red portion at the top of the bar shows the incremental benefit of SMT=ON

Best performance is achieved with NPS=4, 8x CCDs, 4 cores per L3 (64 core CPU) Boost=ON and SMT=ON.

8.2 Pinning Strategies and Hybrid Codes

Using OpenMPI as an example we generally find the most reliable way to pin cores is via an appfile.

We have found slightly reduced performance if we rely on OpenMPI's interface for CPU ID ('-cpu-list')

We have also found that when using all the cores/threads within the socket explicit pinning is still strongly advised to ensure minimum execution time.

For hybrid MPI + OpenMP codes there is normally a choice of three strategies that may be adopted when executing code:

1. All MPI ranks per available thread
2. Map by L3: 1 MPI rank per L3
    a. If SMT=OFF, then OMP_NUM_THREADS is normally set to the number of cores residing on that L3 cache. If it is a 64 core CPU then OMP_NUM_THREADS=4

    b. If SMT=ON, then OMP_NUM_THREADS is normally set to the number of threads residing on that L3 cache. If it is a 4-core part, then OMP_NUM_THREADS=8

3. Map by Core: if SMT=ON then users can choose 1 MPI rank per core + OMP_NUM_THREADS=2.

By way of example, for executing **`mybinary`** using OpenMPI with 2x7742 i.e. 2x 64 core CPUs with:

- SMT=OFF
- 1 MPI rank per L3, no OpenMP threads

```
export CPULIST=$(seq -s , 0 4 127)
mpirun --bind-to none -cpu-list $CPULIST --mca pml ucx --mca osc ucx \\
--mca spml ucx --mca btl ^self,vader,openib --mca coll_hcoll_enable 0\\
-x UCX_NET_DEVICES=mlx5_2:1 -x UCX_TLS=self,sm -x PMIX_MCA_gds=^ds21 \\
--app myappfile-1rankperL3.txt
```

The Infiniband card in this example is a dual port device represented as mlx5_2 and we are using port 1, hence UCX_NET_DEVICES=mlx5_2:1

The accompanying appfile '*myappfile-1rankperL3.txt* shows as follows,

```
-np 1 numactl --physcpubind=0 mybinary -parallel
-np 1 numactl --physcpubind=4 mybinary -parallel
-np 1 numactl --physcpubind=8 mybinary -parallel
-np 1 numactl --physcpubind=12 mybinary -parallel
-np 1 numactl --physcpubind=16 mybinary -parallel
-np 1 numactl --physcpubind=20 mybinary -parallel
-np 1 numactl --physcpubind=24 mybinary -parallel
-np 1 numactl --physcpubind=28 mybinary -parallel
-np 1 numactl --physcpubind=32 mybinary -parallel
-np 1 numactl --physcpubind=36 mybinary -parallel
-np 1 numactl --physcpubind=40 mybinary -parallel
-np 1 numactl --physcpubind=44 mybinary -parallel
-np 1 numactl --physcpubind=48 mybinary -parallel
-np 1 numactl --physcpubind=52 mybinary -parallel
-np 1 numactl --physcpubind=56 mybinary -parallel
-np 1 numactl --physcpubind=60 mybinary -parallel
-np 1 numactl --physcpubind=64 mybinary -parallel
-np 1 numactl --physcpubind=68 mybinary -parallel
-np 1 numactl --physcpubind=72 mybinary -parallel
-np 1 numactl --physcpubind=76 mybinary -parallel
-np 1 numactl --physcpubind=80 mybinary -parallel
-np 1 numactl --physcpubind=84 mybinary -parallel
-np 1 numactl --physcpubind=88 mybinary -parallel
-np 1 numactl --physcpubind=92 mybinary -parallel
-np 1 numactl --physcpubind=96 mybinary -parallel
-np 1 numactl --physcpubind=100 mybinary -parallel
-np 1 numactl --physcpubind=104 mybinary -parallel
-np 1 numactl --physcpubind=108 mybinary -parallel
-np 1 numactl --physcpubind=112 mybinary -parallel
-np 1 numactl --physcpubind=116 mybinary -parallel
-np 1 numactl --physcpubind=120 mybinary -parallel
-np 1 numactl --physcpubind=124 mybinary -parallel
```

Chapter 9 APPENDIX

9.1 DGEMM

What it does: Stresses the compute cycles. Calculates the FLOPS of a core / CCX / NUMA node / socket

Available from: http://portal.nersc.gov/project/m888/apex/

You will also need to download the BLIS Multi-Threaded (MT) libraries from

https://developer.amd.com/amd-aocl/blas-library/

Makefile:

```
CFLAGS= -Ofast -fopenmp -lm -D USE_CBLAS -mavx2 -funroll-loops -lomp \\
        -ffp-contract=fast -mtune=znver2 -march=znver2
LIBS=/home/software/aocl/aocl-2.1-1910/amd/aocl/2.1-1910/amd-blis-mt/lib/libblis-mt.a
INC=-I/home/software/aocl/aocl-2.1-1910/amd/aocl/2.1-1910/amd-blis-mt/include/blis
```

Execution

When compiled with AOCC if a user wishes to run across all 64 cores on socket-0 in a dual socket 2x7742 system:

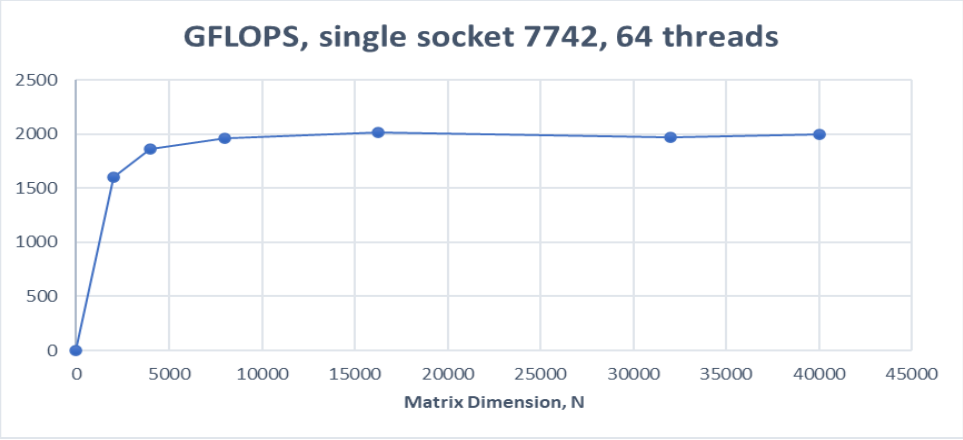
```
OMP_NUM_THREADS=64 GOMP_CPU_AFFINITY="0-63:1"  \\
numactl --membind=0-3 ./mt-dgemm.aocc 8000
```

Other combinations can be derived (per L3, per CCD) by carefully choosing which cores to pin to.

Results

On a system with 2x 7742 and 16x 64GB DDR4-3200 R2, Boost=Off, SMT=OFF

Theoretical peak is 2.25GHz * 16 FLOPS per cycle * 64 cores = 2304 GFLOPS, i.e. about 2000/2304 = 87% efficiency.



9.2 HPL

Introduction:

HPL is a software package that solves a (random) dense linear system in double precision (64 bits) arithmetic on distributed-memory computers.
HPL can be download from its official website https://www.netlib.org/benchmark/hpl/

Prerequisites:

For Building and running HPL on AMD platforms using AOCC and AOCL we require.

1. AMD-BLIS
2. Jemalloc
3. KNEM
4. OpenMPI

These are required for building an optimized OpenMPI library. Prebuilt libraries for AMD BLIS can be downloaded from https://developer.amd.com/amd-aocl

For a 'quick start' a user could just build OpenMPI without knem or jemalloc and then build HPL using the *Make.zen* file below: run with **`mpirun -bind-to core ./xhpl`**

Set the following environment variables:

```
export CC=clang
export FC=flang
export CXX=clang++
export F90=flang
export F77=flang
export AR=llvm-ar
export RANLIB=llvm-ranlib
export NM=llvm-nm
export COMPILERROOT=<Path to AOCC Compiler root directory>
export OMPI=<Path to the AOCC Include Directory>
export OMPL=<Path to the AOCC Library Directory>
```

Now set some compilation flags:

```
export CFLAGS="-O3 -ffast-math -march=znver2 -fopenmp -I${OMPI}"
export CXXFLAGS="-O3 -ffast-math -march=znver2 -fopenmp -I${OMPI}"
export FCFLAGS="-O3 -ffast-math -march=znver2 -fopenmp -I${OMPI}"
export LDFLAGS="-L${OMPL}"
export INCLUDE=$OMPI:$INCLUDE
export PATH=$COMPILERROOT/bin:$PATH
export LD_LIBRARY_PATH=$OMPL:$LD_LIBRARY_PATH
```

Build Jemalloc

Jemalloc is a general purpose malloc (3) implementation that emphasizes fragmentation avoidance and scalable concurrency support.

```
export JEMALLOCROOT=<JEMALLOC_ROOT_PATH>
git clone https://github.com/jemalloc/jemalloc.git jemalloc
cd jemalloc/
./autogen.sh
CFLAGS=$CFLAGS  ./configure --prefix=$JEMALLOCROOT
make -j
make install
```

Build KNEM

KNEM is a Linux kernel module enabling high-performance intra-node MPI communication for large messages. KNEM transfers data from one process to another through a single copy within the Linux kernel.

```
export KNEMROOT=<KNEM_ROOT_PATH>
git clone https://gforge.inria.fr/git/knem/knem.git knem
cd knem
./autogen.sh
./configure --prefix=$KNEMROOT CFLAGS="$CFLAGS          \\
  -I${JEMALLOCROOT}/include/jemalloc" LDFLAGS="$LDFLAGS \\
  -L${JEMALLOCROOT}/lib -ljemalloc" --host=x86_64
make clean
make
make install
```

Build OpenMPI

A High-Performance Message Passing Library. The Open MPI Project is an open source Message Passing Interface implementation. Official website: https://www.open-mpi.org/

```
wget https://download.open-mpi.org/release/open-mpi/v4.0/openmpi-
4.0.0.tar.bz2
tar -xvf openmpi-4.0.0.tar.bz2
cd openmpi-4.0.0
 ./configure --prefix=$OPENMPIROOT --with-knem=$KNEMROOT          \\
   CC=${CC} CXX=${CXX} FC=${FC} CFLAGS="${CFLAGS}"                \\
   CXXFLAGS="${CXXFLAGS}" FCFLAGS="${FCFLAGS}"                    \\
   --enable-mpi-fortran --enable-shared=yes --enable-static=yes   \\
   --enable-mpi1-compatibility
make -j
make install
ln -s $OPENMPIROOT/lib/libmpi.so.40.20.0 $OPENMPIROOT/lib/libmpi.so.20
```

```
export PATH=$OPENMPIROOT/bin:$PATH
export LD_LIBRARY_PATH=$OPENMPIROOT/lib:$LD_LIBRARY_PATH
```

We can now start to build HPL

```
export HPLROOT=<HPL_ROOT_PATH>

wget https://www.netlib.org/benchmark/hpl/hpl-2.3.tar.gz
tar -xvf hpl-2.3.tar.gz
mv hpl-2.3 hpl
make arch=zen
```

using the following sample Make.zen file for the build

```
SHELL        = /bin/sh
CD           = cd
CP           = cp
LN_S         = ln -s
MKDIR        = mkdir
RM           = /bin/rm -f
TOUCH        = touch
ARCH         = $(arch)
TOPdir       = ../../..
INCdir       = $(TOPdir)/include
BINdir       = $(TOPdir)/bin/$(ARCH)
LIBdir       = $(TOPdir)/lib/$(ARCH)
HPLlib       = $(LIBdir)/libhpl.a
MPdir        = ${OPENMPIROOT}
MPinc        = -I$(MPdir)/include
MPlib        = $(MPdir)/lib/libmpi.so
LAdir        = ${BLISROOT}
LAinc        = -I$(LAdir)/include/blis
LAlib        = $(LAdir)/lib/libblis-mt.a
DF77_INTEGER = short : Fortran 77 INTEGER is a C short.
F2CDEFS      = -Dadd__ -DF77_INTEGER=int -DStringSunStyle
HPL_INCLUDES = -I$(INCdir) -I$(INCdir)/$(ARCH) $(LAinc) $(MPinc)
HPL_LIBS     = $(HPLlib) $(LAlib) $(MPlib) -lm
HPL_OPTS     = -DHPL_PROGRESS_REPORT
HPL_DEFS     = $(F2CDEFS) $(HPL_OPTS) $(HPL_INCLUDES)
CC           = clang
CCNOOPT      = $(HPL_DEFS)
CCFLAGS      = $(HPL_DEFS) -O3 -ffast-math -funroll-loops -march=znver2 -fopenmp \\
   -I${JEMALLOCROOT}/include/jemalloc -I${COMPILERROOT}/include
LINKER       = clang
LINKFLAGS    = -fopenmp -O3 -ffast-math -funroll-loops -march=znver2 \\
   -L${JEMALLOCROOT}/lib $(CCFLAGS) -L${COMPILERROOT}/lib -ljemalloc -lamdlibm -lm
ARCHIVER     = llvm-ar
ARFLAGS      = r
RANLIB       = llvm-ranlib
```

Now make the following Operating System changes if not already set

```
echo 1          > /sys/devices/system/cpu/cpufreq/boost
echo performance> /sys/devices/system/cpu/cpu0/cpufreq/scaling_governor
echo 0          > /proc/sys/kernel/randomize_va_space
echo 'never'    > /sys/kernel/mm/transparent_hugepage/enabled
echo ‘never'    > /sys/kernel/mm/transparent_hugepage/defrag
echo 0          > /proc/sys/kernel/yama/ptrace_scope
echo 0          > /proc/sys/kernel/numa_balancing
```

```
echo 3          > /proc/sys/vm/drop_caches
echo 1          > /proc/sys/vm/compact_memory
```

For the case of HPL we find an uplift is provided by disabling Transparent Huge Pages and using Hugepages instead. On a system with 256GB of main memory you can set 240GB of 2MB hugepages with

```
echo 122880     > /proc/sys/vm/nr_hugepages
```

For the case of HPL we find an uplift is provided by disabling Transparent Huge Pages and using Hugepages instead.

We are now in a position to start running HPL. Again, users could perform a simple (OpenMPI assumed):

```
mpirun -bind-to core ./xhpl
```

but to get maximum performance further we need to

1. Set the inner loops of BLIS
2. Use hybrid MPI + OpenMP
3. Explicitly pin each thread to a specific CPU ID

Ensure the knem module is loaded and execute (requires root/sudo to perform hugectl):

```
export mpi_options="--mca mpi_leave_pinned 1 --bind-to none \\
  --report-bindings --mca btl self,vader --map-by ppr:1:l3cache \\
  -x OMP_NUM_THREADS=4 -x OMP_PROC_BIND=TRUE -x OMP_PLACES=cores"


mpirun $mpi_options -app ./appfile_ccx
```

```
-np 1 ./xhpl_ccx.sh 0 0-3 4
-np 1 ./xhpl_ccx.sh 0 4-7 4
-np 1 ./xhpl_ccx.sh 0 8-11 4
-np 1 ./xhpl_ccx.sh 0 12-15 4
-np 1 ./xhpl_ccx.sh 1 16-19 4
-np 1 ./xhpl_ccx.sh 1 20-23 4
-np 1 ./xhpl_ccx.sh 1 24-27 4
-np 1 ./xhpl_ccx.sh 1 28-31 4
-np 1 ./xhpl_ccx.sh 2 32-35 4
-np 1 ./xhpl_ccx.sh 2 36-39 4
-np 1 ./xhpl_ccx.sh 2 40-43 4
-np 1 ./xhpl_ccx.sh 2 44-47 4
-np 1 ./xhpl_ccx.sh 3 48-51 4
-np 1 ./xhpl_ccx.sh 3 52-55 4
-np 1 ./xhpl_ccx.sh 3 56-59 4
-np 1 ./xhpl_ccx.sh 3 60-63 4
-np 1 ./xhpl_ccx.sh 4 64-67 4
-np 1 ./xhpl_ccx.sh 4 68-71 4
-np 1 ./xhpl_ccx.sh 4 72-75 4
-np 1 ./xhpl_ccx.sh 4 76-79 4
-np 1 ./xhpl_ccx.sh 5 80-83 4
-np 1 ./xhpl_ccx.sh 5 84-87 4
-np 1 ./xhpl_ccx.sh 5 88-91 4
-np 1 ./xhpl_ccx.sh 5 92-95 4
-np 1 ./xhpl_ccx.sh 6 96-99 4
-np 1 ./xhpl_ccx.sh 6 100-103 4
-np 1 ./xhpl_ccx.sh 6 104-107 4
-np 1 ./xhpl_ccx.sh 6 108-111 4
-np 1 ./xhpl_ccx.sh 7 112-115 4
-np 1 ./xhpl_ccx.sh 7 116-119 4
-np 1 ./xhpl_ccx.sh 7 120-123 4
-np 1 ./xhpl_ccx.sh 7 124-127 4
```

To do this you require the following files: appfile_cxx (performs pinning) and xhpl_ccx.sh (sets cores per L3 on inner BLIS loops)

The appfile above calls xhpl_ccx.sh:

```
#! /bin/bash
#
# Bind memory to node $1 and four child threads to CPUs specified in $2
# Kernel parallelization is performed at the 2nd innermost loop (IC)

export LD_LIBRARY_PATH=$BLISROOT/lib:$LD_LIBRARY_PATH
export LD_LIBRARY_PATH=$OPENMPIROOT/lib:$LD_LIBRARY_PATH
export OMP_NUM_THREADS=$3
export GOMP_CPU_AFFINITY="$2"
export OMP_PROC_BIND=TRUE

# BLIS_JC_NT=1 (No outer loop parallelization):
export BLIS_JC_NT=1
# BLIS_IC_NT= number of cores per L3 (# of 2nd level threads – one per core in the
# shared L3 cache domain):
export BLIS_IC_NT=$OMP_NUM_THREADS
# BLIS_JR_NT=1 (No 4th level threads):
export BLIS_JR_NT=1
# BLIS_IR_NT=1 (No 5th level threads):
export BLIS_IR_NT=1
hugectl --force-preload –heap numactl --membind=$1 ./xhpl
RANLIB       = llvm-ranlib
```

Here is a sample HPL.dat For Rome architecture with 256 GB of Memory and 2x64 AMD EPYC 7742 64-core processor. Please change this file as per your system configuration: change NB, N, P and Q.

```
HPLinpack benchmark input file
Innovative Computing Laboratory, University of Tennessee
HPL.out     output file name (if any)
6           device out (6=stdout,7=stderr,file)
1           # of problems sizes (N)
171776      Ns
1           # of NBs
244         # of problems sizes (N)
0           MAP process mapping (0=Row-,1=Column-major)
1           # of process grids (P x Q)
4           Ps
8           Qs
16.0        threshold
1           # of panel fact<
2           PFACTs (0=left, 1=Crout, 2=Right)
1           # of recursive stopping criterium
4           NBMINs (>= 1)
1           # of panels in recursion
2           NDIVs
1           # of recursive panel fact.
1           RFACTs (0=left, 1=Crout, 2=Right)
1           # of broadcast
1           BCASTs (0=1rg,1=1rM,2=2rg,3=2rM,4=Lng,5=LnM)
1           # of lookahead depth
0           DEPTHs (>=0)
2           SWAP (0=bin-exch,1=long,2=mix)
64          swapping threshold
0           L1 in (0=transposed,1=no-transposed) form
0           U in (0=transposed,1=no-transposed) form
1           Equilibration (0=no,1=yes)
8           memory alignment in double (> 0)
```

Using the above process with AOCL v2.1 on 2-socket test servers we derive (boost=on, cTDP and PPL set to maximum, Determinism Slider=Power)

- 2x 7742 (2x 64 cores): 3.81 TFLOPS
- 2x 7532 (2x 32 cores): 2.41 TFLOPS

9.3 STREAM

What it does: tests the maximum memory bandwidth of a core, or entire CPU for example

Available from: http://www.cs.virginia.edu/stream/

Setup: We provide some examples here using 3 different compilers: GCC, Intel, AOCC

Maximum memory bandwidth is achieved at 1 core per L3 cache on a 64 core CPU, i.e. 16 cores per CPU. Therefore, on a dual-socket system with 2 x 64 cores we need to set the following environment variables. If the system is already booted with all cores active users can achieve maximum memory bandwidth by suitable core pinning. The following Operating Systems settings are required (root/sudo permissions):

```
#!/bin/bash
echo 0 > /proc/sys/kernel/randomize_va_space
echo 0 > /proc/sys/vm/nr_hugepages
echo 0 > /proc/sys/kernel/numa_balancing
echo 'never' > /sys/kernel/mm/transparent_hugepage/enabled
echo 'never' > /sys/kernel/mm/transparent_hugepage/defrag
```

Run with the following environment variables for the appropriate compiler

GCC	export OMP_NUM_THREADS=32 export GOMP_CPU_AFFINITY=0-127:4
Intel	export OMP_PROC_BIND=true export OMP_NUM_THREADS=32 export OMP_PLACES= "$(echo "{";seq -s },{ 0 4 127; echo "}")"
AOCC	export OMP_SCHEDULE=static export OMP_DYNAMIC=false export OMP_THREAD_LIMIT=256 export OMP_NESTED=FALSE export OMP_STACKSIZE=192M export OMP_NUM_THREADS=32 export GOMP_CPU_AFFINITY=0-127:4

Makefile:

```
intel:
        icc -o stream.intel-2500000000   stream.c -DSTATIC -DNTIMES=10   \\
          -DSTREAM_ARRAY_SIZE=2500000000  -mcmodel=large -shared-intel   \\
          -Ofast -qopenmp -ffreestanding  -qopt-streaming-stores always

clang:
        clang stream.c -O3  -mcmodel=medium -DSTREAM_TYPE=double -mavx2  \\
          -DSTREAM_ARRAY_SIZE=2500000000 -DNTIMES=10 -ffp-contract=fast  \\
          -march=znver2  -fno-unroll-loops   -lomp   -fopenmp -fnt-store \\
          -o stream.clang
```

NOTE: you will not derive maximum performance from GCC as the compiler does not enable streaming stores.

Results:
We now compare STREAM Triad results (MB/s) on a system with 2x 7742 and check the effect of changing NUMA domains per socket from 4 to 2 to 1 (NPS) (system used 64GB out of a total 512GB DDR4-3200 R2; we compiled STREAM with Intel but the AOCC compiler delivers equivalent performance)

These results demonstrate:

CCD=8 NPS=4		SMT = OFF			SMT = ON		
		THREADS	Boost=OFF	Boost=ON	THREADS	Boost=OFF	Boost=ON
Cores/L3	1	32	354,045	354,326	64	339,322	340,233
	2	64	339,420	340,511	128	324,179	325,024
	3	96	331,466	332,242	192	313,955	314,683
	4	128	324,828	325,397	256	307,286	308,198

CCD=8 NPS=2		SMT = OFF			SMT = ON		
		THREADS	Boost=OFF	Boost=ON	THREADS	Boost=OFF	Boost=ON
Cores/L3	1	32	322,523	323,551	64	305,656	305,698
	2	64	305,658	306,082	128	293,386	293,879
	3	96	295,923	296,221	192	282,972	283,383
	4	128	285,429	286,518	256	281,536	281,792

CCD=8 NPS=1		SMT = OFF			SMT = ON		
		THREADS	Boost=OFF	Boost=ON	THREADS	Boost=OFF	Boost=ON
Cores/L3	1	32	300,768	302,505	64	287,146	287,659
	2	64	287,123	287,694	128	282,037	283,127
	3	96	282,868	283,034	192	277,052	277,656
	4	128	279,416	280,622	256	276,206	276,686

- Bandwidth reduces as we reduce the number of NUMA domains per socket
- Setting Boost=on has a marginal positive impact
- Setting SMT=ON has a negative impact for this synthetic test

We also show how performance changes when the CCDs are reduced from CCD=8 to CCD=4 per socket:

CCD=4 NPS=4		SMT = OFF			SMT = ON		
		THREADS	Boost=OFF	Boost=ON	THREADS	Boost=OFF	Boost=ON
Cores/L3	1	32	312,953	317,666	64	314,067	315,701
	2	64	311,103	312,668	128	294,007	295,190
	3	96	297,717	299,107	192	283,904	285,086
	4	128	291,366	292,545	256	278,066	279,472

9.4 Mellanox Configuration

This chapter describes some basic considerations when configuring a Mellanox Infiniband fabric on your AMD EPYC HPC cluster. Mellanox uses the OFED middleware stack to operate their fabric. While there is a community OFED version available from openfabrics.org we recommend using the OFED version that Mellanox bundles and provides for free from their website.

You must use OFED v4.7-3.2.9 or greater on AMD EPYC Rome HPC clusters. Earlier OFED versions will not yield correct bandwidth profiles. Firmware on our ConnectX-6 cards is 20.26.1814. Users will also need to enable Preferred-IO Mode in the BIOS (refer to 'BIOS Settings' earlier).

Once OFED is installed there are several tests the system manager can issue. For example, to ensure the correct bandwidth profile exists between any 2 compute nodes within the cluster open 2 terminal windows and connect to node-3 and node-5 for example:

- On node-3: **`numactl -C 20 ib_write_bw -a --report_gbits`**
- On node-5: **`numactl -C 15 ib_write_bw -a --report_gbits -d mlx5_0 node-3`**

In this example we have

- tested the connection from node 5 to node 3
- used numactl to ensure we have selected a core that is 'logically' closest to the Mellanox Host Channel Adapter (HCA) PCI card, or cores are 'local' to the PCI slot hosting the ConnectX-6 card (see Appendix OSU for local-to-local versus remote-to-remote). Recall, we can establish which cores to choose for this purpose by issuing **`hwloc-ls`** beforehand and noting the cores
- ensured the cores idle in the C1 State (see discussion above)

we have dual port ConnectX-6 Mellanox cards in both nodes and have explicitly stated which port to use on the card via the -d flag for a HDR200 Mellanox IB network the following profile should be displayed:

```
#bytes     #iterations    BW peak[Gb/sec]    BW average[Gb/sec]
MsgRate[Mpps]
 2           5000           0.067164           0.066476           4.154756
 4           5000             0.13               0.13             4.187184
 8           5000             0.27               0.27             4.187324
 16          5000             0.54               0.54             4.197941
 32          5000             1.08               1.07             4.178680
 64          5000             2.15               2.14             4.176621
 128         5000             4.31               4.27             4.169307
 256         5000             8.61               8.57             4.182211
 512         5000            17.07              17.02             4.156448
 1024        5000            34.07              33.98             4.148378
 2048        5000            67.39              67.28             4.106271
 4096        5000           132.84             132.15             4.032974
 8192        5000           186.65             186.43             2.844643
 16384       5000           191.88             191.75             1.462959
 32768       5000           196.80             196.78             0.750658
 65536       5000           196.44             196.44             0.374673
 131072      5000           197.07             197.06             0.187927
 262144      5000           197.13             197.13             0.093999
 524288      5000           197.14             197.14             0.047001
 1048576     5000           197.13             197.13             0.023500
 2097152     5000           197.15             197.14             0.011751
 4194304     5000           197.11             197.10             0.005874
```

-

-

That is, we witness a unidirectional peak bandwidth of 200Gb/s and one which increases steadily in bandwidth with increasing packet size, and then remains at peak bandwidth with increasing packet size. This is the correct profile, and the cores need to be in the C1 idle state to achieve this. If your cores idle in C2 then you would observe a possible brief maximum in your bandwidth profile in the middle of the packet band, after which bandwidth would decrease with increasing packet size.

Network latency can also be tested with **`ib_write_lat`** in the same way **`ib_write_bw`** is used above. Users should expect to see latency of about 1microsecond on a Mellanox HDR200 network with EPYC Rome. We measure 0.99 microseconds with a 2byte size packet with Boost=ON.

These are very useful tests as they demonstrate no broken system components (cables, cards, PCI slots etc) and that the system is configured correctly. Systems managers should also monitor /var/log/messages for any Mellanox-related warnings/errors.

Mellanox also usefully compile a version of OpenMPI and bundle that as part of their OFED release. If you wish to use an alternative MPI library, we recommend reading how Mellanox builds OpenMPI in their latest release documentation. We draw your attention to the following flags that should be passed to `./configure` if your MPI library supports them:

```
./configure --enable-mpi1-compatibility --with-hcoll=/opt/mellanox/hcoll \
    --with-knem=/opt/knem-1.1.3.90mlnx1 --with-ucx=/opt/ucx/1.5.1       \
    --with-xpmem=/opt/xpmem/2.6.5
```

or issue

```
./configure -h
```

to check if your MPI library supports these flags.

9.5 OSU Network Tests

What it Does: Ohio State University (OSU) Micro Benchmarks are a set of MPI, SHMEM and UPC tests that measure the performance for a wide range of parallel message exchange operations.

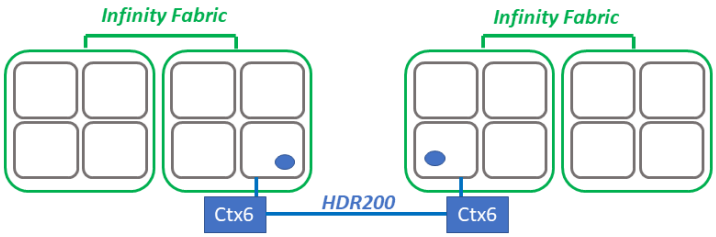
Available from: http://mvapich.cse.ohio-state.edu/benchmarks/

Setup: Please refer to the Appendix 'Mellanox Configuration' for minimum correct versions of OFED and firmware when using ConnectX-6 cards. We demonstrate 3 examples here of how to run and their subsequent results:

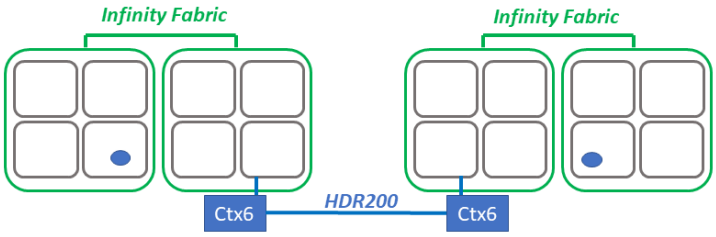
1. Latency: This test runs on 2 nodes with one core per node. One core is the sender and the core on the other node is the receiver. The sender sends a message with a certain data size to the receiver and waits for a reply from the receiver. The receiver receives the message from the sender and sends back a reply with the same data size. Many iterations of this ping-pong test are carried out and average one-way latency numbers are obtained. Blocking version of MPI functions (MPI_Send and MPI_Recv) are used in the tests.

2. 'Local to Local': all the cores on the CPU that are local to the Infiniband HCA are used. The cores on the other CPU are idle. As there are no messages to travel across the Infinity fabric, this should give the highest bandwidth.

3. 'Remote to Remote': all the cores on the CPU socket that do not have the Infiniband HCA attached to it are used. This is a worst-case scenario as all messages need to go over the Infinity fabric with the server first.



Local Socket to Local Socket
Socket is logically closest to IB card within Server



Remote Socket to Remote Socket
Socket is remote to IB card within Server

System configuration used for testing:

BIOS	American Megatrends Inc. Version: 5.14 Release Date: 08/15/2019
CPU	EPYC 7742 64-Core
Speed	2.25 GHz
NPS	4
System profile	Performance determinism
SMT	disabled
Boost	ON
Preferred I/O	C1
C2	disabled
CPU governor	performance
OS	CentOS 7.6
Kernel	3.10.0-957.10.1.el7.x86_64
HCA	Mellanox ConnectX-6
Speed	HDR200
OFED	MLNX_OFED_LINUX-4.7-3.2.9.0
HCA Firmware	20.26.1814

Makefile:

```
!#/bin/bash

#  Latency
mpirun --bind-to none --mca pml ucx --mca osc ucx --mca spml ucx \
    --mca btl ^self,vader,openib --mca coll_hcoll_enable 0 -x \
    UCX_NET_DEVICES=mlx5_2:1 -x UCX_TLS=self,sm,rc_x -host node02:1,node03:1 \
    -map-by ppr:1:node -np 2 taskset -c 92 osu_latency

# Local to Local

mpirun --bind-to none --mca pml ucx --mca osc ucx --mca spml ucx \
    --mca btl ^self,vader,openib --mca coll_hcoll_enable 0 -x \
    UCX_NET_DEVICES=mlx5_2:1 -x UCX_TLS=self,sm,rc_x -host node02:64,node03:64\
    -map-by ppr:64:node -np 128 numactl --localalloc --physcpubind=64-127 \
    osu_mbw_mr

# Remote to Remote

mpirun --bind-to none --mca pml ucx --mca osc ucx --mca spml ucx \
    --mca btl ^self,vader,openib --mca coll_hcoll_enable 0 -x \
    UCX_NET_DEVICES=mlx5_2:1 -x UCX_TLS=self,sm,rc_x -host node02:64,node03:64\
    -map-by ppr:64:node -np 128 numactl --localalloc --physcpubind=0-63\
    osu_mbw_mr
```

The CPU IDs should be chosen that they are in the same NUMA domain as the ConnectX-6 cards

Results:

#Size	Latency (us)	Local to Local		Remote to Remote	
		MB/s	Messages	MB/s	Messages
0	1.05	-	-	-	-
1	1.04	125.1	125069945.2	70.4	70412887.4
2	1.04	246.3	123166738.7	136.6	68285326.0
4	1.05	486.3	121561838.7	273.6	68395917.5
8	1.12	996.1	124511971.0	549.7	68707453.4
16	1.05	1943.7	121477972.2	1095.1	68446331.2
32	1.21	3924.2	122631907.4	2216.4	69261295.3
64	1.26	6531.6	102055478.6	5366.3	83847775.7
128	1.33	9216.7	72005515.4	7387.1	57711369.5
256	1.69	14971.9	58484084.7	13564.0	52984404.0
512	1.90	19486.1	38058783.0	17002.5	33208058.0
1024	2.30	22529.4	22001373.7	19901.4	19434928.7
2048	2.34	24768.5	12093987.4	21817.2	10652915.4
4096	2.92	24030.2	5866750.7	22443.3	5479313.4
8192	3.92	24300.7	2966392.3	22714.4	2772754.5
16384	4.86	24393.9	1488883.3	22832.7	1393598.1
32768	6.84	24824.9	757595.2	22840.5	697036.9
65536	10.20	24768.4	377936.4	23427.0	357467.5
131072	16.66	24725.8	188643.2	23415.2	178643.7
262144	15.64	24712.3	94269.8	23411.3	89307.1
524288	26.83	24706.3	47123.5	23407.0	44645.2
1048576	49.36	24700.3	23556.1	23404.9	22320.7
2097152	94.64	24696.9	11776.4	23403.1	11159.5
4194304	185.20	24695.7	5887.9	23405.2	5580.2

Chapter 10 Resources

Reference for all developers on AMD platforms

http://developer.amd.com

AMD Processors documentation and guides

https://developer.amd.com/resources/developer-guides-manuals/

Workload Tuning Guide for AMD EPYC 7002 Series Processor Based Servers

https://developer.amd.com/wp-content/resources/56745_0.80.pdf

AMD Optimizing CPU Libraries User Guide

https://developer.amd.com/wp-content/resources/AOCL_User_Guide_2.1.pdf

Software Optimization Guide for AMD Family 17h Models 30h and Greater Processors

https://developer.amd.com/wp-content/resources/56305_SOG_3.00_PUB.pdf

Community Forum on AMD to discuss technical issues and contact a Server Guru

https://community.amd.com/community/server-gurus

Other Resources

Best Practice Guide from PRACE. Naples Tuning Guide with useful reference for compiler options and porting codes:

http://www.prace-ri.eu/best-practice-guide-amd-epyc

SLES Tuning Guide on Rome

https://www.suse.com/documentation/suse-best-practices/optimizing-linux-for-amd-epyc-with-sle-12-sp3/data/optimizing-linux-for-amd-epyc-with-sle-12-sp3.html

Linux Virtual Memory:

https://www.kernel.org/doc/Documentation/sysctl/vm.txt

Tuning Guide from Red Hat:

https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/performance_tuning_guide/

Spectre and Meltdown:

https://access.redhat.com/articles/3311301

AMD Statement on Spectre and Meltdown:

https://www.amd.com/en/corporate/speculative-execution-previous-updates#paragraph-337801

Linux kernel, CPU Governors documentation:

https://www.kernel.org/doc/Documentation/cpu-freq/governors.txt