HPC Tuning Guide for AMD EPYC™ Processors

Publication # **56420** Revision: **0.7**
Issue Date: **December 2018**

Advanced Micro Devices

Contents

Revision History

Date	Revision	Description
December 2018	0.70	Initial public release.

Chapter 1 Introduction

AMD launched the new ‘EPYC’ x86_64 CPU for the data center in June 2017. Based on the 14nm Zen core architecture it is the first in a new series of CPUs designed for the data center that will take AMD well into the next decade.

This guide is intended for vendors, system integrators, resellers, system managers and developers who are interested in EPYC system configuration details.

There is also a discussion on the AMD EPYC software development environment, and we include four appendices on how to install and run the HPL, HPCG, DGEMM, and STREAM benchmarks. The results produced are ‘good’ but are not necessarily exhaustively tested across a variety of compilers with their optimization flags.

Chapter 2 Cheat Sheets

The appendices include recipes for building several synthetic benchmarks, and also some checks to ensure a Mellanox InfiniBand network is operating at proper bandwidth and latency

2.1 Quick High-Performance Set-Up

In the BIOS (or equivalent, thereof) set:

- **SME** = OFF
- **SEV** = OFF
- **SMT** = OFF
- **Boost** = ON
- **Determinism Slider** = Performance

Certain vendor platforms may include Workload Profiles or System Tunings for High Performance Compute. 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.

On the system (please see corresponding sections in this document for explanation. DO NOT implement these changes without understanding what they do and how to reverse them):

- For a HPC cluster with a high performance low latency interconnect such as Mellanox disable the C2 idle state. Assuming a dual socket 2x32 core system

  ```
  cpupower -c 0-63 idle-set -d 2
  ```

- Set the CPU governor to ‘performance’:

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

The settings will enable the user to establish a baseline. Once set and a baseline is taken for the applications, developers are then advised to pursue further system tuning.

2.2 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, DDR4-2666 Dual Rank, 1 DIMM slot per channel, all DIMM slots populated:

```
circa 285GB/s with an Intel or PGI compiler
```

Please note, there is a heavily optimized version of STREAM from AMD that will provide 300GB/s memory bandwidth. Please contact your AMD representative if you are interested in accessing this code.

2.3 Other Sundry Tips

2.3.1 Core Pinning and Memory Locality:

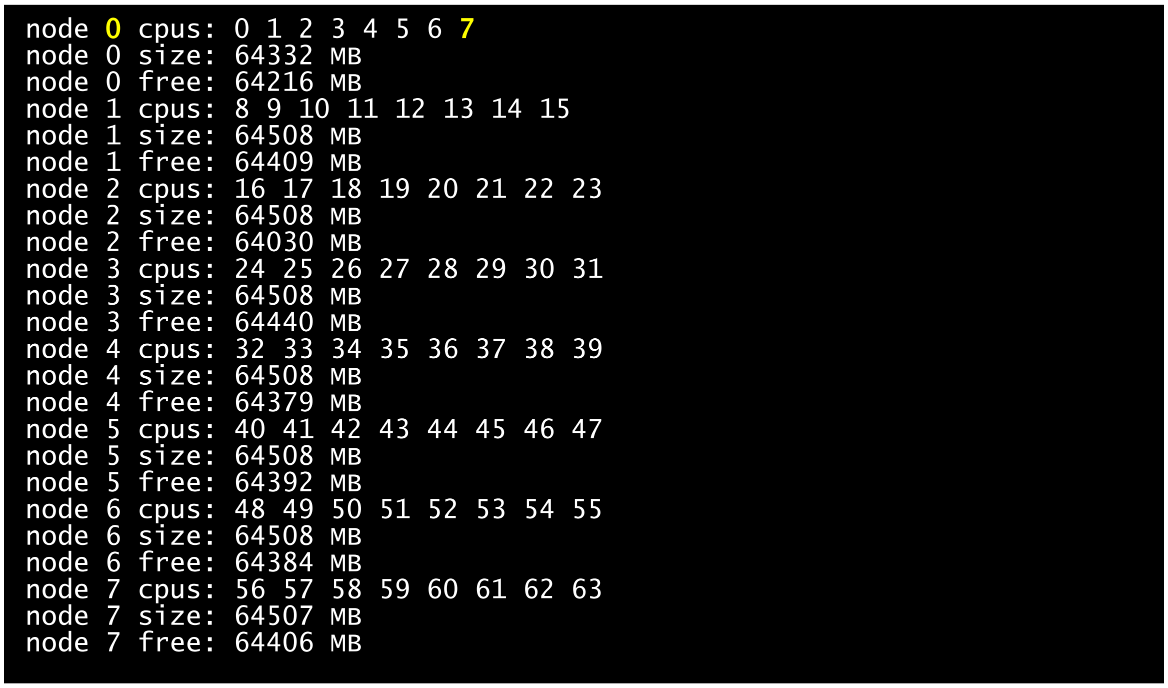
You can pin your binary to run on a specific core, e.g. core 7:

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

This will allow the kernel to freely choose which memory slots to use. If you want to ensure that you are only using the memory that's logically closest to the core, i.e. on the same NUMAnode then use the '--membind=' flag also:

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

You can establish which NUMAnode your core belongs to using the output from `numactl -H` :



```
node 0 cpus: 0 1 2 3 4 5 6 7
node 0 size: 64332 MB
node 0 free: 64216 MB
node 1 cpus: 8 9 10 11 12 13 14 15
node 1 size: 64508 MB
node 1 free: 64409 MB
node 2 cpus: 16 17 18 19 20 21 22 23
node 2 size: 64508 MB
node 2 free: 64030 MB
node 3 cpus: 24 25 26 27 28 29 30 31
node 3 size: 64508 MB
node 3 free: 64440 MB
node 4 cpus: 32 33 34 35 36 37 38 39
node 4 size: 64508 MB
node 4 free: 64379 MB
node 5 cpus: 40 41 42 43 44 45 46 47
node 5 size: 64508 MB
node 5 free: 64392 MB
node 6 cpus: 48 49 50 51 52 53 54 55
node 6 size: 64508 MB
node 6 free: 64384 MB
node 7 cpus: 56 57 58 59 60 61 62 63
node 7 size: 64507 MB
node 7 free: 64406 MB
```

2.3.2 Faster ‘make’ with ‘make -j’

A common way to build code from a source tarball is with

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

However, the ‘make’ step will implicitly use a single core. With AMD EPYC you can significantly speed this up by passing the -j flag with a number representing the maximum number of threads you would like to commit to the make process. For example, on a dual socket 2x32 core system with SMT=ON you now have 128 threads and could do

```
./configure
make -j 128
make install
```

For large code sources (e.g installing a new GCC compiler, installing OpenMPI) this will dramatically speed up build time, sometimes from hours down to about 30 minutes for GCC or about 20 minutes on OpenMPI (these are ballpark figures)

Chapter 3 BIOS Settings

This section describes some common settings within a High-Performance Computing environment

- **Determinism Slider =** Performance
  Reduces machine to machine variability within a cluster
- **SEV =** OFF
  Secure Encrypted Virtualization. Allows memory encryption on a per Virtual Machine basis
- **SME =** OFF
  Secure Memory Encryption (encrypts all the memory)
- **SMT =** OFF or ON
  Simultaneous Multi-Threading. The benefit of this setting is application-dependent
- **Boost** = ON or OFF
  Turns the boost function ON or OFF on all cores. This can also be switched on or off via the Linux command line as root in RHEL/CentOS for example
- **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. Reducing from 2600 MTS to 2400 MTS saves approximately 15 watts per socket.
- **Downcoring**
  Sets the number of active cores per CCX (see later). For a 32-core part AUTO will leave 4 cores active per CCX. 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.
- **C-States =** Enabled
  Leave these enabled. If required, users should disable C2 via the command line as root (see later)

Chapter 4 Linux Kernel Considerations

Since the launch of EPYC there have been several patches issued specifically in relation to EPYC. System Managers 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 kernel 3.10-862. This kernel ships with RHEL/CentOS v7.5. It is possible to use CentOS v7.4 or older v7.x, however this installs the default CentOS 3.10.0-693.17.1 kernel. Older kernels should not be used.

Kernel.org

At least kernel 4.13. Patches relating to Spectre and Meltdown entered the kernel at version 4.15.

SUSE

Most of the EPYC support has been back-ported to the SLES 12 SP3 4.4 kernel. Do not use an older version of the SLES kernel.

Chapter 5 System Settings: /proc and /sys

There are a number of ways memory can be consumed over the uptime of a system. The following provides a (very brief!) summary of some of the areas users 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*)

5.1 /proc/sys/vm/zone_reclaim_mode

From kernel.org, "*Zone_reclaim_mode allows someone to set more or less 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.

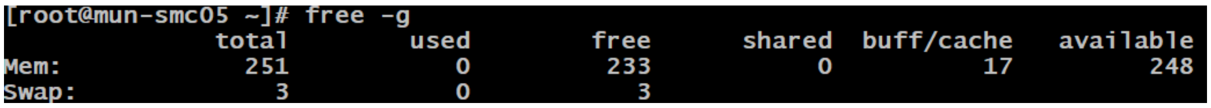
For workloads that benefit from having file system data cached, zone reclaim is usually turned off; but it is a balance and if job size exceeds the memory of a NUMAnode, and/or your job is multi-cored and extends outside the NUMAnode then it is probably sensible to turn this on. See more details:

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.

5.2 /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



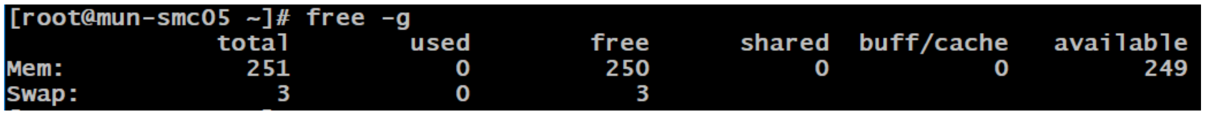
```
[root@mun-smc05 ~]# free -g
              total        used        free      shared  buff/cache   available
Mem:            251           0         233           0          17         248
Swap:             3           0           3
```

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 through the use of an epilog script for example.

5.3 /proc/sys/vm/swappiness

Determines the extent the kernel will swap memory pages "*A value of 0 instructs the kernel not to initiate swap until the amount of free and file-backed pages is less than the high water mark in a zone*"

5.3.1 Transparent Huge Pages

For HPC we have found it best to leave these on. Provides several percent in performance gain. This setting should be disabled if the application provides support for explicit huge pages. Follow the application's guidance for allocating explicit huge pages at boot time.

5.3.2 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 are (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
```

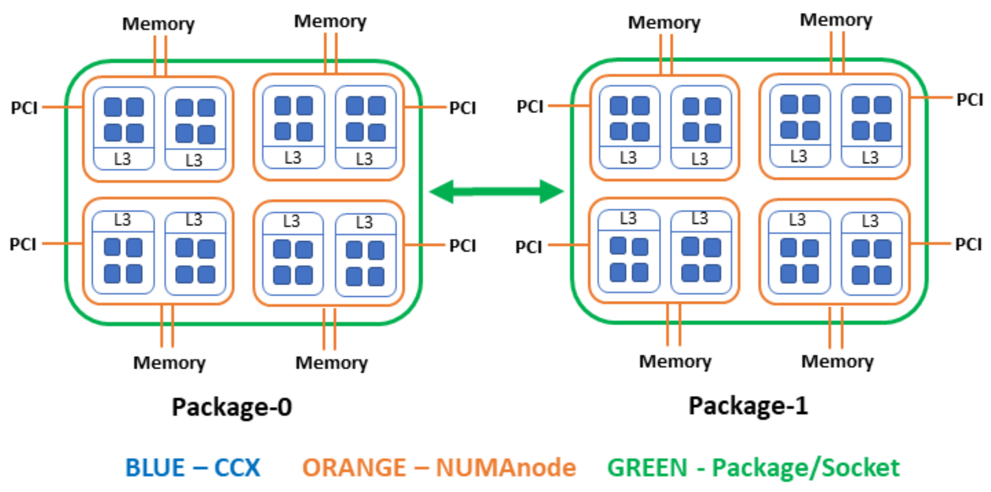
You can then view the 2 files to check if the Spectre patches are now disabled:

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

Chapter 6 CPU Settings

6.1 CPU Layout

The following diagram shows a simplified schematic of a dual-socket AMD server.



There is a rich hierarchy within the EPYC CPU which we explain here.

Each socket is comprised of 4 silicon dies (NUMA nodes, or 'Zeppelins') shown in orange. Each Zeppelin

- provides 2 memory channels, i.e. 8 memory channels per socket
- provides PCIe gen3 slots (each OEM individually decides how to present/consume these resources)
- Is connected to the remaining 3 Zeppelins within the socket via the internal Global Memory Interconnect, or "Infinity Fabric". This enables each NUMA node to access the memory and PCI capabilities associated with its counterparts both within the socket and between sockets

Within each Zeppelin there are 2 Compute Complexes (CCX). Each CCX has

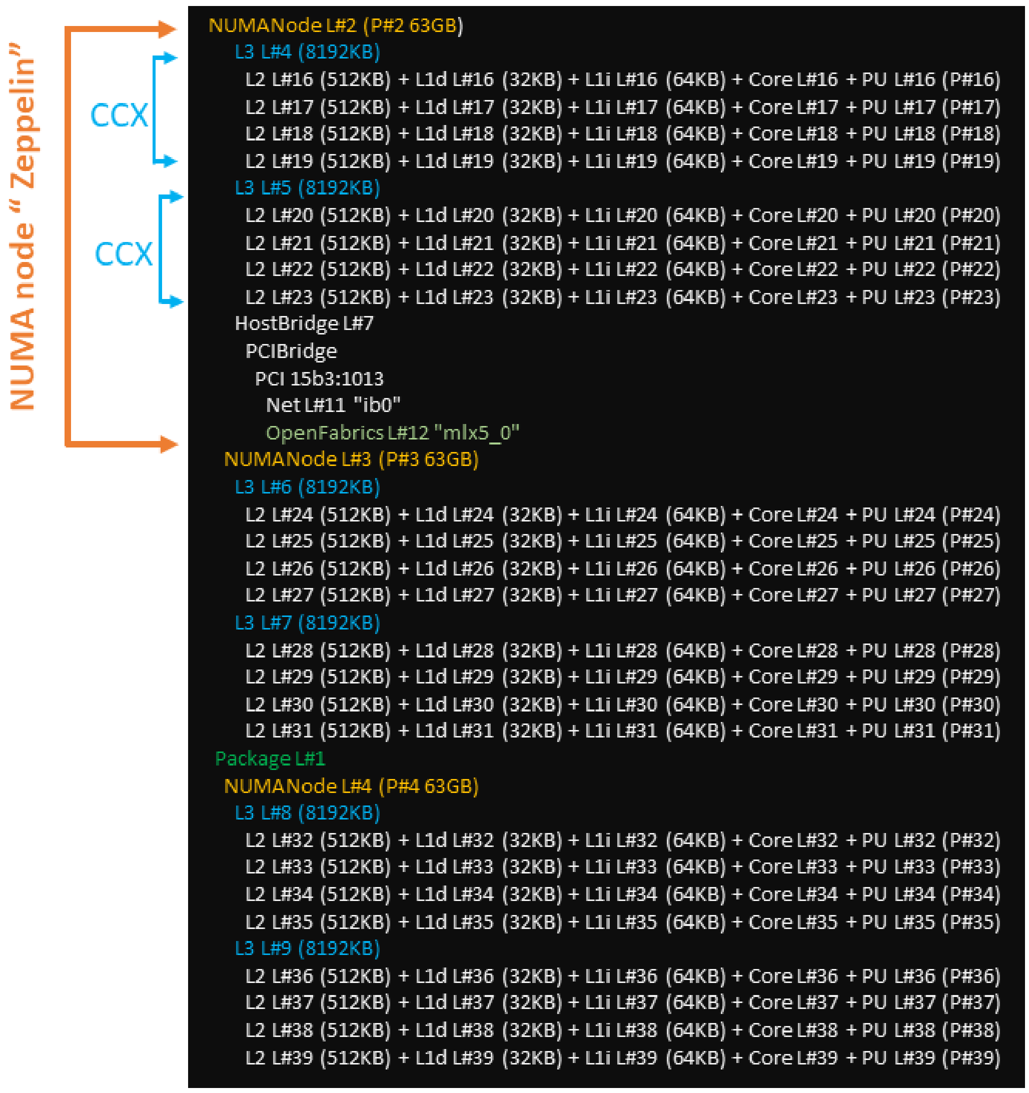
- its own L3 cache. Each core has its own L2 and L1i and L1d caches
- up to 4 cores per CCX, i.e. 32 cores per socket. Other EPYC core counts follow:
  - 24 core SKU has 3 cores per CCX
  - 16 core SKU has 2 cores per CCX
  - 8 core SKU has 1 core per CCX

It is important to understand this hierarchy. Doing so will aid will the developer in optimizing their application layout so as to choose the correct locality of memory and possibly L3 cache; for example,

- given the choice would you launch a 4-core application within a single CCX or would you thread it across all 4 NUMA nodes within the socket? This is application-dependent and would be necessary to investigate.
- For network connectivity

6.2 Understanding hwloc-ls and hwloc-info

NOTE: Dell adopts an alternative CPU ID numbering convention. The discussion below is still valid and relevant to the Dell platforms; one just needs to remember the CPU IDs will change under each CCX. Please refer to Appendix A: Dell CPU ID Numbering Convention for the Dell numbering convention



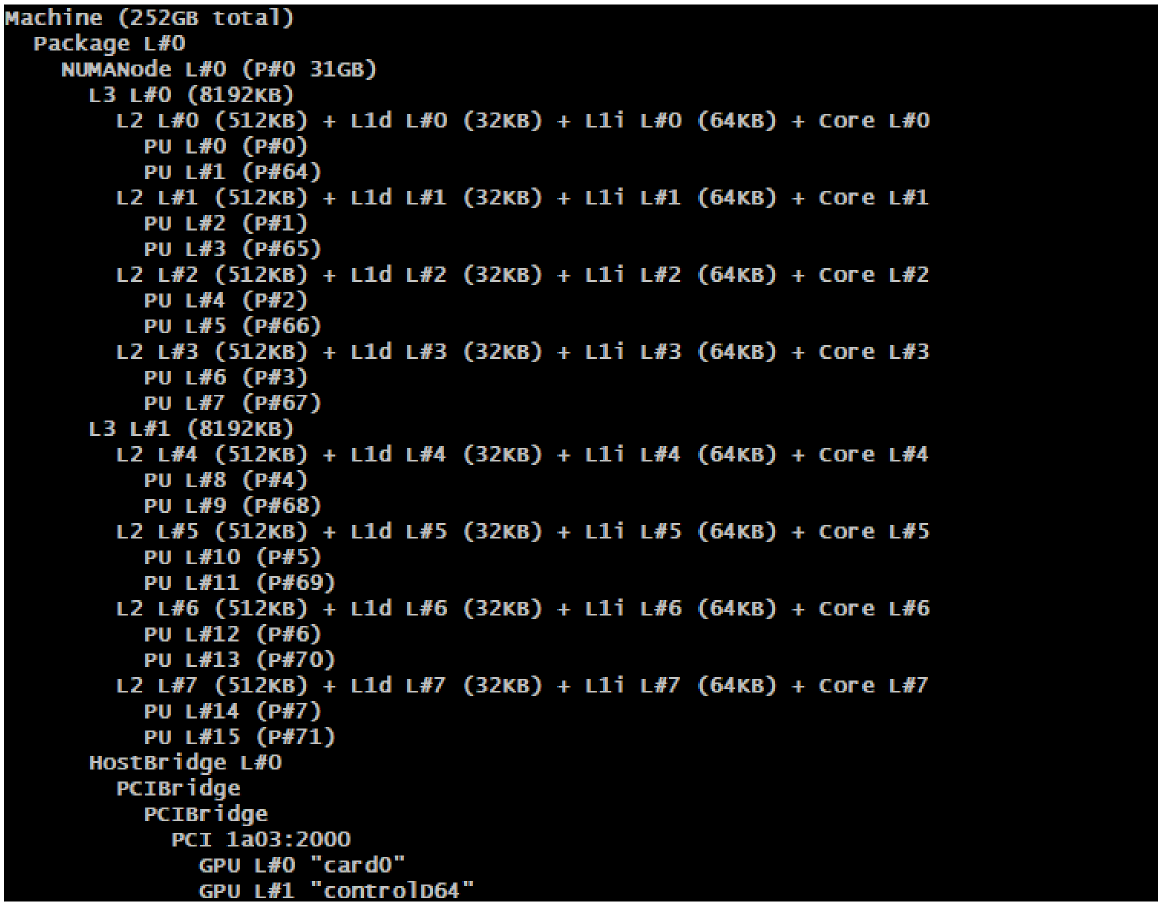
```
NUMANode L#2 (P#2 63GB)
  L3 L#4 (8192KB)
   L2 L#16 (512KB) + L1d L#16 (32KB) + L1i L#16 (64KB) + Core L#16 + PU L#16 (P#16)
   L2 L#17 (512KB) + L1d L#17 (32KB) + L1i L#17 (64KB) + Core L#17 + PU L#17 (P#17)
   L2 L#18 (512KB) + L1d L#18 (32KB) + L1i L#18 (64KB) + Core L#18 + PU L#18 (P#18)
   L2 L#19 (512KB) + L1d L#19 (32KB) + L1i L#19 (64KB) + Core L#19 + PU L#19 (P#19)
  L3 L#5 (8192KB)
   L2 L#20 (512KB) + L1d L#20 (32KB) + L1i L#20 (64KB) + Core L#20 + PU L#20 (P#20)
   L2 L#21 (512KB) + L1d L#21 (32KB) + L1i L#21 (64KB) + Core L#21 + PU L#21 (P#21)
   L2 L#22 (512KB) + L1d L#22 (32KB) + L1i L#22 (64KB) + Core L#22 + PU L#22 (P#22)
   L2 L#23 (512KB) + L1d L#23 (32KB) + L1i L#23 (64KB) + Core L#23 + PU L#23 (P#23)
  HostBridge L#7
   PCIBridge
    PCI 15b3:1013
     Net L#11 "ib0"
     OpenFabrics L#12 "mlx5_0"
 NUMANode L#3 (P#3 63GB)
  L3 L#6 (8192KB)
   L2 L#24 (512KB) + L1d L#24 (32KB) + L1i L#24 (64KB) + Core L#24 + PU L#24 (P#24)
   L2 L#25 (512KB) + L1d L#25 (32KB) + L1i L#25 (64KB) + Core L#25 + PU L#25 (P#25)
   L2 L#26 (512KB) + L1d L#26 (32KB) + L1i L#26 (64KB) + Core L#26 + PU L#26 (P#26)
   L2 L#27 (512KB) + L1d L#27 (32KB) + L1i L#27 (64KB) + Core L#27 + PU L#27 (P#27)
  L3 L#7 (8192KB)
   L2 L#28 (512KB) + L1d L#28 (32KB) + L1i L#28 (64KB) + Core L#28 + PU L#28 (P#28)
   L2 L#29 (512KB) + L1d L#29 (32KB) + L1i L#29 (64KB) + Core L#29 + PU L#29 (P#29)
   L2 L#30 (512KB) + L1d L#30 (32KB) + L1i L#30 (64KB) + Core L#30 + PU L#30 (P#30)
   L2 L#31 (512KB) + L1d L#31 (32KB) + L1i L#31 (64KB) + Core L#31 + PU L#31 (P#31)
Package L#1
 NUMANode L#4 (P#4 63GB)
  L3 L#8 (8192KB)
   L2 L#32 (512KB) + L1d L#32 (32KB) + L1i L#32 (64KB) + Core L#32 + PU L#32 (P#32)
   L2 L#33 (512KB) + L1d L#33 (32KB) + L1i L#33 (64KB) + Core L#33 + PU L#33 (P#33)
   L2 L#34 (512KB) + L1d L#34 (32KB) + L1i L#34 (64KB) + Core L#34 + PU L#34 (P#34)
   L2 L#35 (512KB) + L1d L#35 (32KB) + L1i L#35 (64KB) + Core L#35 + PU L#35 (P#35)
  L3 L#9 (8192KB)
   L2 L#36 (512KB) + L1d L#36 (32KB) + L1i L#36 (64KB) + Core L#36 + PU L#36 (P#36)
   L2 L#37 (512KB) + L1d L#37 (32KB) + L1i L#37 (64KB) + Core L#37 + PU L#37 (P#37)
   L2 L#38 (512KB) + L1d L#38 (32KB) + L1i L#38 (64KB) + Core L#38 + PU L#38 (P#38)
   L2 L#39 (512KB) + L1d L#39 (32KB) + L1i L#39 (64KB) + Core L#39 + PU L#39 (P#39)
```

Using a Linux based system (CentOS 7.5 in this case) on a Supermicro server, we show above a section of the output of `hwloc-ls` on a dual socket system with 32 cores per socket. This illustrates a number of points:

- 4 cores per L3 cache; this is a CCX, and you can see the CPU IDs associated with them, i.e. 16, 17, 18, 19 in the top CCX followed by 20,21,22,23 in the second CCX
- The memory associated with each NUMA node (63GB)
- An attachment into the top NUMA node (shown in green). This is a Mellanox HCA for a high performance and low latency InfiniBand network. CPU IDs 16-23 are logically 'closest' to this Mellanox network card.
- The snippet also shows the start of the second sockets' output "Package-1" with CPU IDs 32-29 in the first NUMA node on that socket.

`hwloc-ls` is a very useful tool for obtaining your 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 NUMAnode.

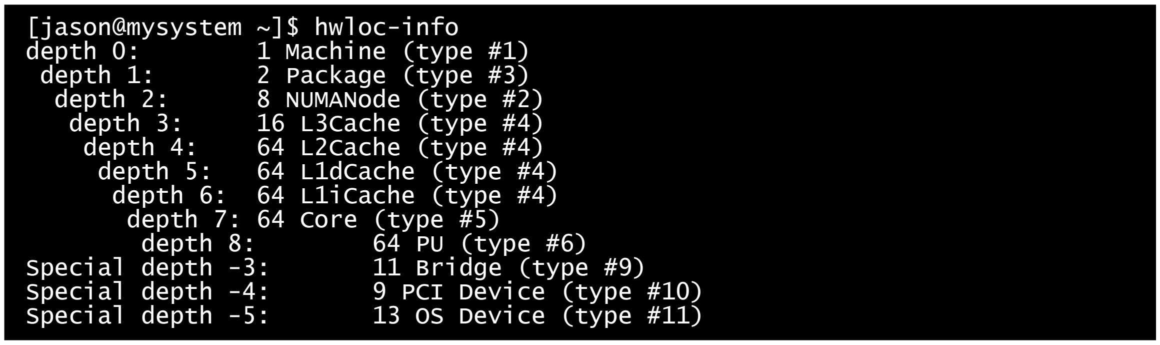
The above `hwloc-ls` output demonstrates the case for a single thread per core. When Simultaneous Multi-Threading (SMT) is enabled within the BIOS the second thread on each core is shown as:



```
Machine (252GB total)
  Package L#0
    NUMANode L#0 (P#0 31GB)
      L3 L#0 (8192KB)
        L2 L#0 (512KB) + L1d L#0 (32KB) + L1i L#0 (64KB) + Core L#0
          PU L#0 (P#0)
          PU L#1 (P#64)
        L2 L#1 (512KB) + L1d L#1 (32KB) + L1i L#1 (64KB) + Core L#1
          PU L#2 (P#1)
          PU L#3 (P#65)
        L2 L#2 (512KB) + L1d L#2 (32KB) + L1i L#2 (64KB) + Core L#2
          PU L#4 (P#2)
          PU L#5 (P#66)
        L2 L#3 (512KB) + L1d L#3 (32KB) + L1i L#3 (64KB) + Core L#3
          PU L#6 (P#3)
          PU L#7 (P#67)
      L3 L#1 (8192KB)
        L2 L#4 (512KB) + L1d L#4 (32KB) + L1i L#4 (64KB) + Core L#4
          PU L#8 (P#4)
          PU L#9 (P#68)
        L2 L#5 (512KB) + L1d L#5 (32KB) + L1i L#5 (64KB) + Core L#5
          PU L#10 (P#5)
          PU L#11 (P#69)
        L2 L#6 (512KB) + L1d L#6 (32KB) + L1i L#6 (64KB) + Core L#6
          PU L#12 (P#6)
          PU L#13 (P#70)
        L2 L#7 (512KB) + L1d L#7 (32KB) + L1i L#7 (64KB) + Core L#7
          PU L#14 (P#7)
          PU L#15 (P#71)
      HostBridge L#0
        PCIBridge
          PCIBridge
            PCI 1a03:2000
              GPU L#0 "card0"
              GPU L#1 "controlD64"
```

Thus, on a dual socket system with 2x 32 core CPUs the first thread on each core is within the range 0-63, while the second thread will have CPU IDs within the range 64-127. In the above picture you can see how core-0 has CPU IDs 0 and 64 attached to it (i.e. its first and second thread).

When installing Linux kernels (especially older kernels) the user will also need to ensure it recognizes the correct cache-hierarchy in the system, use `hwloc-info`:



```
[jason@mysystem ~]$ hwloc-info
depth 0:        1 Machine (type #1)
 depth 1:       2 Package (type #3)
  depth 2:      8 NUMANode (type #2)
   depth 3:     16 L3Cache (type #4)
    depth 4:    64 L2Cache (type #4)
     depth 5:   64 L1dCache (type #4)
      depth 6:  64 L1iCache (type #4)
       depth 7: 64 Core (type #5)
        depth 8:        64 PU (type #6)
Special depth -3:       11 Bridge (type #9)
Special depth -4:       9 PCI Device (type #10)
Special depth -5:       13 OS Device (type #11)
```

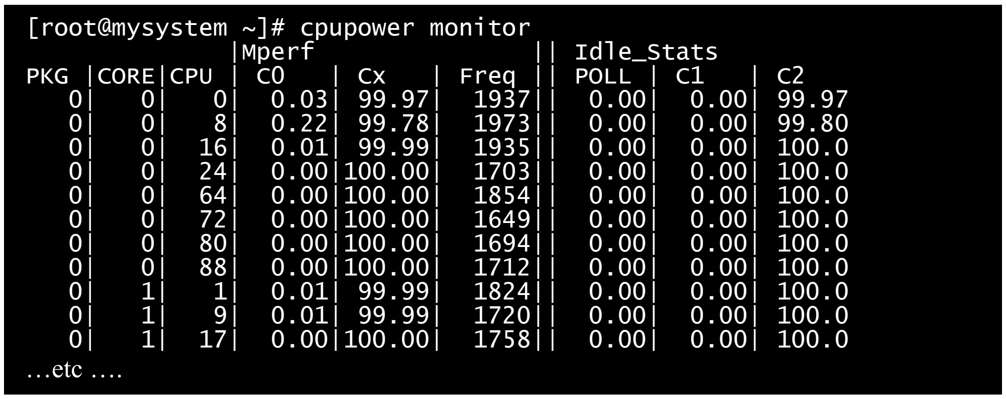
Older kernels will misalign or completely ignore L3, and this be evident using `hwloc-info` or `hwloc-ls`

6.3 C-States, Frequencies and Boosting

There are several Core-States, or C-States, that the CPU can idle within and which the user has control over.

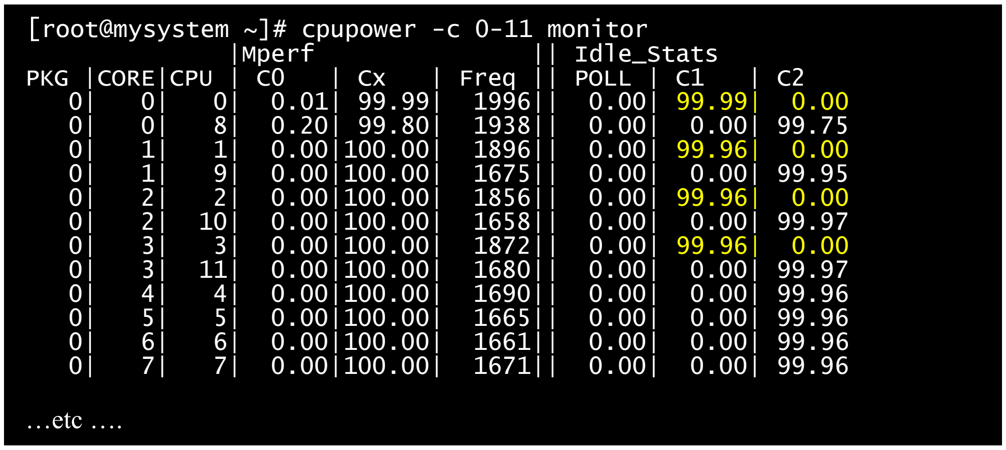
- C0: active, the active frequency while running an application/job.
- C1: idle
- C2: idle and power-gated. This is a deeper sleep state and will have a greater latency in getting to C0 compared to when it idles in C1.

User root can enable and disable C-States. As an example, here is a snip of '`cpupower monitor`' with the CPU idling in C2



```
[root@mysystem ~]# 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 ….
```

Now let us disable C2 on cores 0 to 3. Doing so sends those cores into state C1:



```
[root@mysystem ~]# 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
   0|   7|   7|  0.00|100.00|  1671||  0.00|  0.00| 99.96

…etc ….
```

Disabling C2 on all cores is also possible by omitting the `-c` argument in the `cpupower idle-set` command. This is a required setting for running a high-performance low latency network such as InfiniBand from Mellanox: the latency required in getting from C2 to the active C0 state is too great; instead all the cores must idle in C1.

In the above example we can now set those 4 cores back to idle in C2 with

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

Once active in C0 a core can enter 1 of 3 frequency levels:

1. **Base frequency:** On a 7601 CPU this is quoted as being at 2.2GHz. If the CPU governor is set to performance (more below on governors), its active frequency will be within a few 10s of MHz, often within a few MHz of 2.2GHz. The user can observe this by running `cpupower monitor` as root.
2. **All Cores Boost frequency:** Provided there is enough thermal and electrical headroom the CPU can boost up to its quoted All Cores Boost frequency, on a 7601 this is 2.7GHz

3. **Max Cores Boost frequency**: If any NUMAnode has no more than 3 cores in C0 or C1 then the core can boost further, up to a maximum of 3.2GHZ on a 7601 CPU for example. This provides boost for up to 12 cores per socket. If 13th core enters C0 then the frequency of all cores will drop back to the All Cores Boost frequency.

To benefit from either boost state, boost 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:

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

It can be returned to off with:

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

Alternatively, the Boost state can be set within the BIOS (requires a system reboot).

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).

6.4 P-States

We now introduce the concept of P-States on the CPU. At this point, to avoid confusion we highlight:

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

User root can observe these P-States by executing

```
cpupower frequency-info
```

We show here the output for a 7351 CPU which shows the 3 P-states at 1.2GHz (P2), 1.8GHz (P1) and 2.4GHz (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: 1.20 GHz - 2.40 GHz
  available frequency steps:  2.40 GHz, 1.80 GHz, 1.20 GHz
  available cpufreq governors: conservative userspace powersave ondemand
performance schedutil
  current policy: frequency should be within 1.20 GHz and 2.40 GHz.
                  The governor "performance" may decide which speed to use
                  within this range.
  current CPU frequency: 2.40 GHz (asserted by call to hardware)
  boost state support:
    Supported: yes
    Active: yes
    Boost States: 0
    Total States: 3
    Pstate-P0:  2400MHz
    Pstate-P1:  1800MHz
    Pstate-P2:  1200MHz
```

6.5 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. 2.2GHz on a 7601 CPU. If Boost is ON, then it will attempt to boost the frequency up to the Max Core Boost frequency of 2.7Ghz on the 7601 or to the 3.2GHz on the 7601 if sufficient number of cores are idling. 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.

- **Schedutil** (new): Like OnDemand but uses the load balance information from the Linux scheduler instead of trailing load.

- **Conservative:** Like 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. Locks it to P2.

- **Userspace**: allows the user to specify the core frequency in the range between P2 and P0. Note currently EPYC only supports specifying the specific maximum frequencies for P0, P1 and P2. Specifying any other frequency will cause the core to operate at the next highest fixed frequency.

Administrators can set the CPU governor via the cpupower command. E.g. for the performance governor:

```
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*

6.6 'cpupower' Command

The preceding sections introduced the concepts CPU governors, Boost, and C-States. It is instructive to collate a useful list of commands here, which also act as a single future reference for the future.

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
```

or

```
watch -n 1 cpupower 0-15 monitor
```

Display 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 pertaining to 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 7 Libraries and Compilers

AMD now publishes several libraries, compilers, user guides through *http://developer.amd.com* users do not need to sign up or fill in forms to download software: accept the terms and the download begins. Apart from OpenBLAS and the FFTW, everything in this chapter is available from the AMD developer portal

7.1 Libraries

7.1.1 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 EPYC processor family.

BLIS works well up to 16 OMP threads. If the number of threads needs to exceed this we recommend OpenBLAS.

Download options:

- Pre-compiled single or multi-threaded versions of the library from *http://developer.amd.com* (both static and dynamic versions are packaged within the tarball).
- download the source and build library: *https://github.com/amd/blis* For those wishing to build single-threaded version on EPYC, use the Zen framework:

```
./configure zen
make -j 64
make install
```

  [on a dual socket 2x 32core system, for example]

  An OpenMP multi-threaded version, using the Zen framework, can be built by issuing

```
./configure --enable-threading=openmp zen
```

  or with POSIX threads (pthreads) via

```
./configure --enable-threading=pthreads zen
```

We use BLIS in our HPL tests, as shown in Appendix C: HPL

7.1.2 OpenBLAS

OpenBLAS in an alternative optimized version of BLAS and has a make target to specifically build for EPYC We have tested this with DGEMM on a dual socket 2x 32core system and found we can derive good performance with 64 OMP threads.

Website: *https://www.openblas.net/*

Or view project on github: *https://github.com/xianyi/OpenBLAS*

Basic build on EPYC:

```
make TARGET=ZEN
make install PREFIX=/path/to/install/openblas
```

Other flags to pass to make include:

```
make TARGET=ZEN USE_OPENMP=1
make TARGET=ZEN NO_AFFINTY=1
```

By default, OpenBLAS will assume to load balance any threads across all Zen cores. Invoking the NO_AFFINITY=1 flag disables this ability and allows the user to pin the threads to whatever CPU IDs they choose.

More details on building and calling interfaces can be found on the GitHub wiki on this project:

https://github.com/xianyi/OpenBLAS/wiki/Document

7.1.3 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.

Download options:

- Pre-compiled static or dynamic versions of the library from *http://developer.amd.com*

7.1.4 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, 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 EPYC processor family, libFLAME enables running high performing LAPACK functionalities on AMD platforms.

Download options:

- Pre-compiled single or multi-threaded versions of the library from *http://devloper.amd.com* (both static and dynamic versions are packaged within the tarball).
- Download the source from *https://github.com/amd/libflame*

7.2 Compilers

The AMD Optimizing C/C++ Compiler (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.

AOCC compiler binaries are suitable for Linux systems having glibc version 2.17 and above.

The compiler suite consists of a C/C++ compiler (clang), a Fortran compiler (flang) and a Fortran front end to clang (Dragon Egg).

We provide a summary introduction here to the compiler environment for AMD EPYC; more extensive documentation relating to AOCC can be found on *http://developer.amd.com* or AMD-external websites

7.2.1 Clang

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

Clang support the `-march=znver1` flag to enable best code generation and tuning for AMD's Zen based x86 architecture.

7.2.2 Flang

Flang will be AMDs supported Fortran compiler going forward.

The flang compiler is a recent addition to the AOCC suite (added April 2018) and is currently in 'alpha' for developers to download and test.

Based on Fortran 2008, AMD extends the GitHub version of flang (*https://github.com/flang-compiler/flang*) . The Flang compiler supports all clang compiler options and an additional number of flang-specific compiler options (full list available via flang document on *http://developer.amd.com*)

7.2.3 Dragon Egg

DragonEgg is a GCC plugin that replaces GCC's optimizers and code generators with those from the LLVM project. DragonEgg that comes with AOCC works with gcc-4.8.x, has been tested for x86-32/x86-64 targets and has been successfully used on various Linux platforms.

GFortran is the actual frontend for Fortran programs responsible for preprocessing, parsing and semantic analysis generating the GCC GIMPLE intermediate representation (IR). DragonEgg is a GNU plugin, plugging into GFortran compilation flow. It implements the GNU plugin API. With the plugin architecture, DragonEgg becomes the compiler driver, driving the different phases of compilation.

After following the download and installation instructions Dragon Egg can be invoked by:

```
$ gfortran [gFortran flags]                              \\
 -fplugin=/path/AOCC-1.2-Compiler/AOCC-1.2-               \\
FortranPlugin/dragonegg.so [plugin optimization flags]    \\
  -c xyz.f90
$ clang -O3 -lgfortran -o xyz xyz.o
$./xyz
```

7.2.4 Open64

The Open64 compiler is AMDs legacy compiler and includes a full Software Development Kit. The Compiler and SDK are available to download from *http://developer.amd.com* The SDK consists of:

- x86 Open64 Compiler Suite 4.2.4
- AMD CodeAnalyst Performance Analyzer 2.9.18
- AMD Core Math Library (ACML) 4.4.0
- AMD LibM 2.1
- AMD Core Math Library for Graphic Processors (ACML-GPU) 1.1.1

While development ceased in 2011, we have tested this on the STREAM benchmark (details in Appendix D: Stream) and have derived performance that is about the same as PGI or Intel on the same test.

7.2.5 GCC Compiler

The default compiler that ships with RHEL/CentOS 7.4 is version 4.8.5. For HPC this GCC compiler version often does not deliver the performance required in supercomputing. We have undertaken tests with the much later GCC 7.3 and 8.1 and used this to run and derive good performance on our HPL, HPCG, and DGEMM tests in the appendices.

7.2.6 PGI

AMD has performed limited testing on some codes with PGI Community Edition v17. We demonstrate in the appendices how to build several binaries for synthetic benchmarks on both of these compilers.

7.2.7 Intel

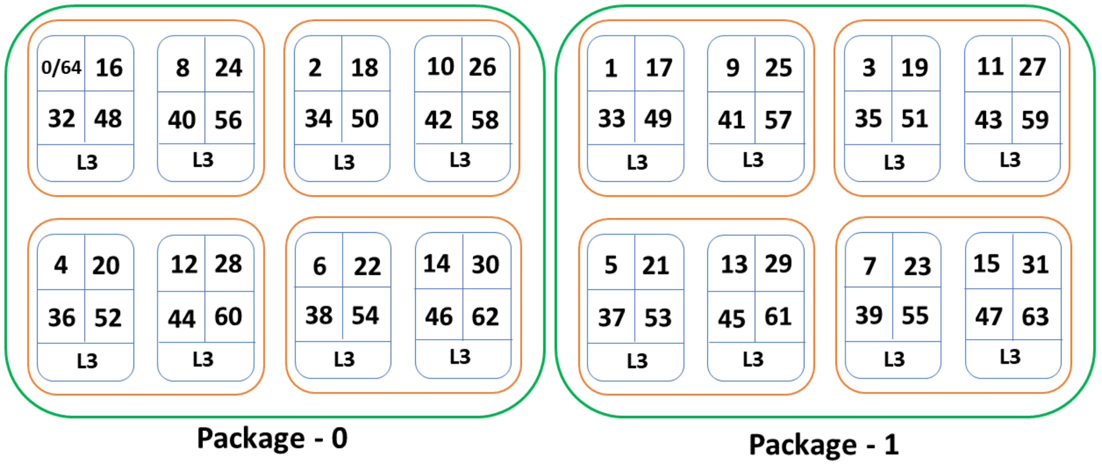
AMD has performed limited testing on some codes with the Intel v18 compiler. We demonstrate in the appendices how to build a number of binaries for synthetic benchmarks on both of these compilers.

Appendix A: Dell CPU ID Numbering Convention

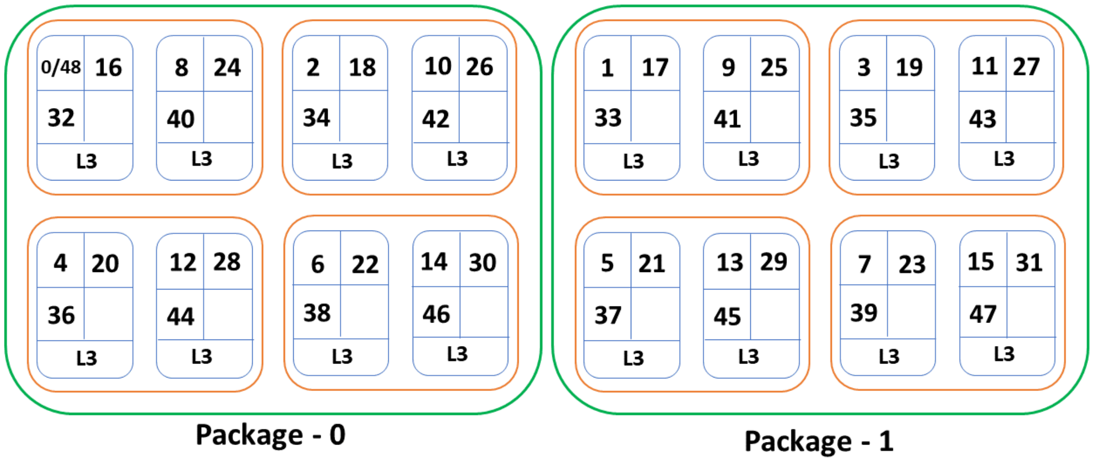
Dell has adopted an alternative CPU ID scheme compared to that shown in the previous `hwloc-ls` subsection.

Each core has 2 threads; we show here the CPU ID for 'thread #0' on each core.

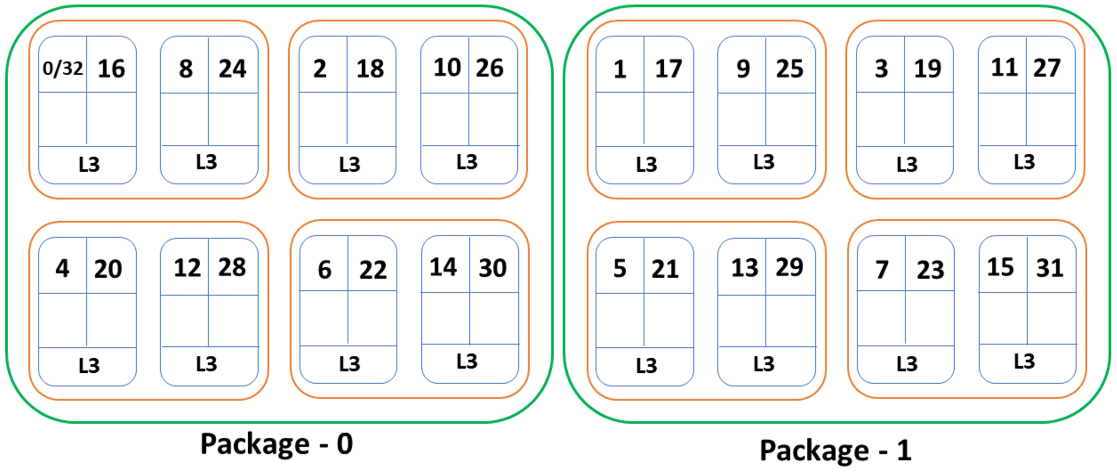
0/64 'thread #1' shows the CPU ID for the first and second thread on that core, i.e. CPU IDs 0 and 64 (add 64 to the value of the ID on the first thread to derive the CPU ID of the second thread)



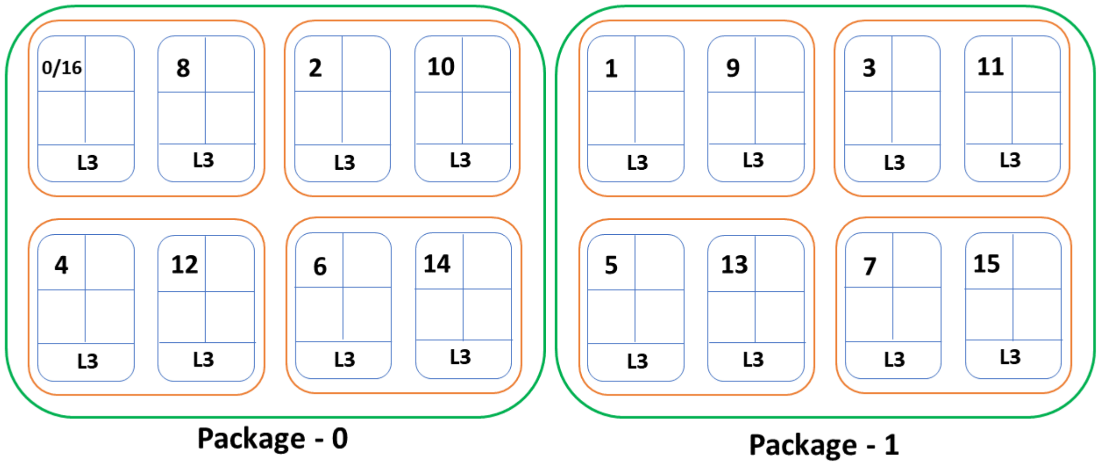
Dual socket 2 x 32 core server: ID of thread #1 = (ID of thread #0) + 64



Dual socket 2 x 24 core server: ID of thread #1 = (ID of thread #0) + 48



Dual socket 2 x 16 core server: ID of thread #1 = (ID of thread #0) + 32



Dual socket 2 x 8 core server: ID of thread #1 = (ID of thread #0) + 16

Appendix B: DGEMM

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

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

You will also need to download either

- the BLIS Multi-Threaded (MT) libraries from *http://developer.amd.com*
- OpenBLAS from *https://www.openblas.net/* which will require building (see previous section)

Makefile:

```
CFLAGS=-O3 -fopenmp -lm -D USE_NVBLAS
LIBS=/opt/amd/amd-blis-MT-0.95-beta/lib/libblis.a
INC=-I/opt/amd/amd-blis-MT-0.95-beta/include/blis

gcc:
        gcc $(CFLAGS) -o mt-dgemm.gcc mt-dgemm.c $(INC) $(LIBS)

pgi:
        pgcc -O3 -D USE_NVBLAS -lm -o mt-dgemm.pgi mt-dgemm.c $(INC) $(LIBS)
```

Alter PATHs accordingly.

Passing the USE_NVBLAS flag in the makefile, in combination with explicitly specifying the INCLUDE path to point to the BLIS INCLUDE directory, enables us to use our own choice of library to call DGEMM (i.e. BLIS or OpenBLAS)

Modification

The only modification we had to do to the code was to add an underscore when calling dgemm in the NVBLAS section, thus '**dgemm**' now becomes '**dgemm**_':

```
#elif defined( USE_NVBLAS )
        char transA = 'N';
        char transB = 'N';

        dgemm(&transA, &transB, &N, &N, &N, &alpha, matrixA,
                &N,matrixB, &N, &beta, matrixC, &N);
#else
```

Execution

For example, when compiled with gcc to run on a single core for a 4096 x 4096 problem size

```
export OMP_NUM_THREADS
numactl -c 0 --membind=0 ./mt-dgemm.gcc 4096
```

To run on all the cores in a dual socket 2x 32 core system

```
export OMP_NUM_TRHEADS=64
./stream.gcc 4096
```

Other combinations can be derived (per CCX, per NUMAnode) by carefully choosing which cores to pin to. For example, on a 32-core CPU on a single CCX (i.e. 4 cores that share the same L3) we refer to `hwloc-ls`, and on a supermicro system we can take the first CCX which has CPU IDs 0,1,2,3. Therefore to run on this CCX, which is a member of NUMAnode '0':

```
export OMP_NUM_THREADS=4
numactl -C 0-3 --membind=0 ./mt-dgemm.gcc 4096
```

Results

We obtain about 95% efficiency on a single core.

BLIS worked well up to 16 threads but if you wish to extend this across more threads in a system, e.g. a dual socket 2 x 32 core system, then we would recommend using the OpenBLAS library to call DGEMM. We observe about 80% efficiency across 64 cores, 1 thread per core using OpenBLAS.

Appendix C: HPL

What it does: Stresses the cores 100% and calculates the FLOPS

Available from: http:// *http://www.netlib.org/benchmark/hpl/*

Setup

You need a C compiler and MPI library installed: we used GCC 7.2 and OpenMPI 3.0.0 included with Mellanox OFED. Also download the BLIS library from *http://developer.amd.com*

```
cd /path/hpl-2.2/setup
cp Make.Linux_Intel64 ../Make.EPYC
cd ..
```

You should now have /path/hpl-2.2/Make.EPYC which you need to edit with your favorite editor, see 'Makefile; Make.EPYC' subsection immediately below. Once edited, then

```
make arch=EPYC
cd /path/hpl-2.2/bin/EPYC
```

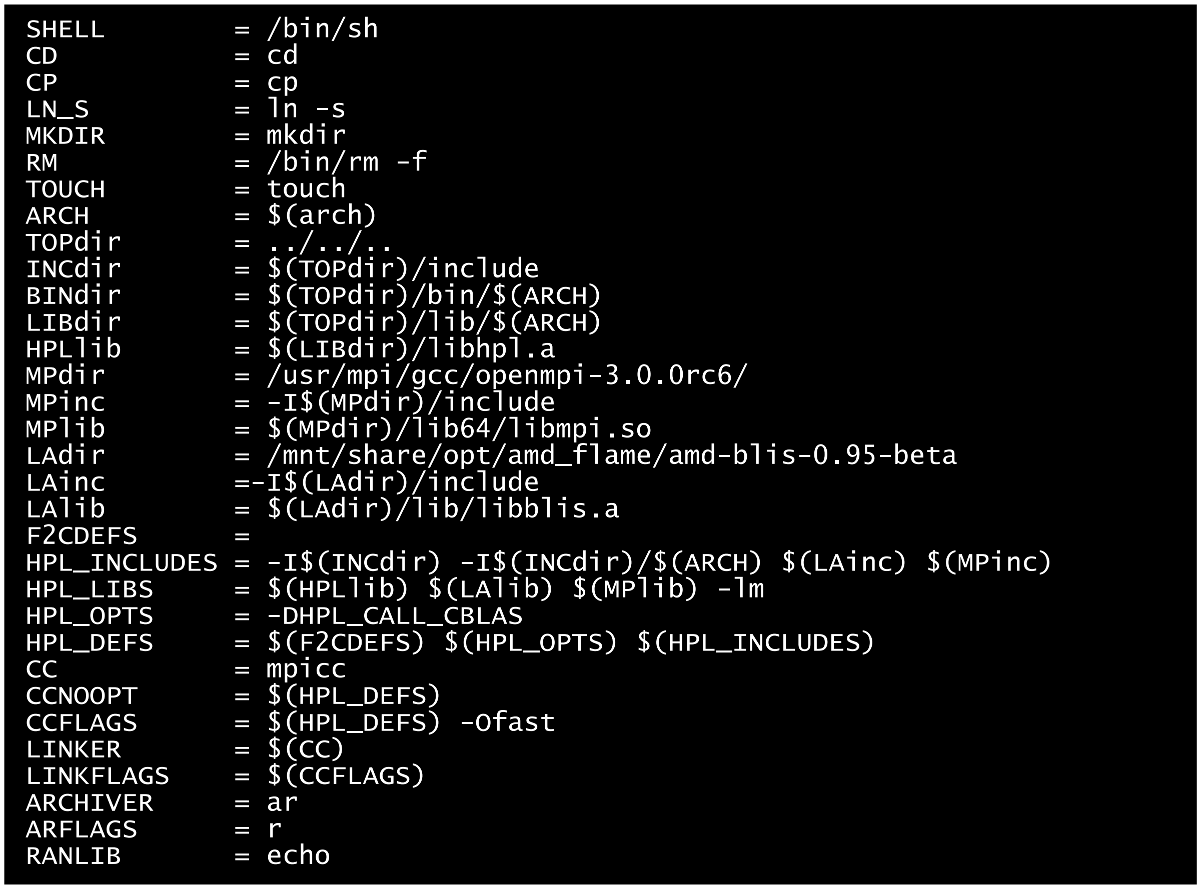
Now edit HPL.dat:

```
HPLinpack benchmark input file
Innovative Computing Laboratory, University of Tennessee
HPL.out      output file name (if any)
file         device out (6=stdout,7=stderr,file)
1            # of problems sizes (N)
240000       Ns
1            # of NBs
191          NBs
0            PMAP process mapping (0=Row-,1=Column-major)
1            # of process grids (P x Q)
8            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.
2            RFACTs (0=left, 1=Crout, 2=Right)
1            # of broadcast
0            BCASTs (0=1rg,1=1rM,2=2rg,3=2rM,4=Lng,5=LnM)
1            # of lookahead depth
1            DEPTHs (>=0)
2            SWAP (0=bin-exch,1=long,2=mix)
8            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)
```

After saving your HPL.dat (on a single dual socket 2x32c)

```
mpirun -np 64 -bind-to core ./xhpl
```

Makefile: Make.EPYC



```
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       = $(LIBdir)/libhpl.a
MPdir        = /usr/mpi/gcc/openmpi-3.0.0rc6/
MPinc        = -I$(MPdir)/include
MPlib        = $(MPdir)/lib64/libmpi.so
LAdir        = /mnt/share/opt/amd_flame/amd-blis-0.95-beta
LAinc        =-I$(LAdir)/include
LAlib        = $(LAdir)/lib/libblis.a
F2CDEFS      =
HPL_INCLUDES = -I$(INCdir) -I$(INCdir)/$(ARCH) $(LAinc) $(MPinc)
HPL_LIBS     = $(HPLlib) $(LAlib) $(MPlib) -lm
HPL_OPTS     = -DHPL_CALL_CBLAS
HPL_DEFS     = $(F2CDEFS) $(HPL_OPTS) $(HPL_INCLUDES)
CC           = mpicc
CCNOOPT      = $(HPL_DEFS)
CCFLAGS      = $(HPL_DEFS) -Ofast
LINKER       = $(CC)
LINKFLAGS    = $(CCFLAGS)
ARCHIVER     = ar
ARFLAGS      = r
RANLIB       = echo
```

Results

With the above settings, CPU governor=performance, SMT=OFF, BIOS is per Supermicro settings, on a dual socket 2x32core 7601 with 512GB DDR4 dual rank memory configured to 2400MHz we obtain the following results:

	N	**Memory**	**block (NB)**	**BOOST**	**GFLOPS**	**percent peak***
Boost ON	240000	480 GB	191	on	1115	99.0%
Vary N	30000	50 GB	191	off	732.7	65.0%
	40000	55 GB	191	off	811.4	72.1%
	60000	71 GB	191	off	880.6	78.2%
	120000	154 GB	191	off	960.0	85.2%
	240000	480 GB	191	off	995.4	88.4%

* *Measured relative to the base frequency, 2.2GHz on a 7601 CPU*

With Boost=ON we visually observe the frequency (`watch -n 1 cpupower monitor`) at circa 2.45GHz +/- 40MHz, occasionally up to 2.6GHz

Appendix D: Stream

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

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

Setup: We provide some examples here using 4 different compilers: gcc, intel, pgi, open64

Maximum memory bandwidth is achieved at 2 cores per CCX, i.e. 16 cores per CPU. Therefore, on a dual-socket system with 2 x 32cores we need to set the following environment variables

GCC

```
export OMP_NUM_THREADS=32
export GOMP_CPU_AFFINITY=0-63:2
```

PGI

```
export OMP_NUM_THREADS=32
export MP_BIND=yes
export MP_BLIST="$(seq -s, 1 2 63)"
```

Intel

```
export OMP_NUM_THREADS=32
export OMP_PROC_BIND=true
export OMP_DISPLAY_ENV=true
export OMP_PLACES="{0},{2},{4},{6},\\
{8},{10},{12},{14},{16},{18},{20},\\
{22},{24},{26},{28},{30},{32},{34},\\
{36},{38},{40},{42},{44},{46},{48},\\
{50},{52},{54},{56},{58},{60},{62}"
```

Open64

```
export OMP_NUM_THREADS=32
export O64_OMP_SET_AFFINITY=TRUE
export O64_OMP_AFFINITY_MAP=0,2,4,6,8,10,\\
12,14,16,18,20,22,24,26,28,30,32,34,36,38,\\
40,42,44,46,48,50,52,54,56,58,60,62
```

OMP settings for dual socket 2 x 32 cores

Relate this to the output in `hwloc-ls` to observe which CPU IDs are being used.

For a dual socket 24 core CPU we use the following environment variables (only CPU ID lists change):

GCC export OMP_NUM_THREADS=32 export GOMP_CPU_AFFINITY=0,1,3,4,6,\\ 7,9,10,12,13,15,16,18,19,21,22,24,\\ 25,27,28,30,31,33,34,36,37,39,40,42,\\ 43,45,46	*PGI* export OMP_NUM_THREADS=32 export MP_BIND=yes export MP_BLIST= 0,1,3,4,6,7,9,10,\\ 12,13,15,16,18,19,21,22,24,25,27,\\ 28,30,31,33,34,36,37,39,40,42,43,45,46
Intel export OMP_NUM_THREADS=32 export OMP_PROC_BIND=true export OMP_DISPLAY_ENV=true export OMP_PLACES="{0},{1},{3},{4},\\ {6},{7},{9},{10},{12},{13},{15},{16},\\ {18},{19},{21},{22},{24},{25},{27},\\ {28},{30},{31},{33},{34},{36},{37},\\ {39},{40},{42},{43},{45},{46}"	*Open64* export OMP_NUM_THREADS=32 export O64_OMP_SET_AFFINITY=TRUE export O64_OMP_AFFINITY_MAP=0,1,3,4,6,\\ 7,9,10,12,13,15,16,18,19,21,22,24,25,\\ 27,28,30,31,33,34,36,37,39,40,42,43,45,46

OMP settings for dual socket 2 x 24 cores

Again, we keep the total number of threads to 16 threads per CPU (32 in total) and again we use 2 cores per CCX

For a 16 core EPYC CPU no such environment variables need to be set

Makefile:

```
STACK= -mcmodel=medium
OMP= -fopenmp
OPTIMIZATIONS= -O3 -mavx -mtune=znver1 -lm
STREAM_PARAMETERS= -DSTREAM_ARRAY_SIZE=1000000000 -DNTIMES=20 -DVERBOSE

OPTIMIZATIONS_PGI= -mp -fast -tp piledriver -Mmovnt -Mvect=prefetch -Munroll -
Mipa=fast,inline

OPTIMIZATION_O64= -Ofast
OMP_O64= -mp

gcc:
        gcc $(OPTIMIZATIONS) $(OMP) $(STACK) $(STREAM_PARAMETERS) stream.c   \\
          -o stream.gcc

pgi:
        pgcc $(OPTIMIZATIONS_PGI) $(STACK)  -DSTREAM_ARRAY_SIZE=800000000    \\
          stream.c -o stream.pgi

open64:
        opencc $(OPTIMIZATION_O64) $(STACK) $(OMP_O64) $(STREAM_PARAMETERS)  \\
         stream.c -o stream.o64

intel:
        icc -o stream.intel stream.c -DSTATIC -DSTREAM_ARRAY_SIZE=800000000  \\
         -mcmodel=large -shared-intel -Ofast -qopenmp
```

Results:

Intel and PGI should get you close to 286GB/s on a system with 2666MHz dual rank memory 1x DIMM per channel with just 1 DIMM slot available per channel

GCC will not provide maximum bandwidth results because it does not enable non-temporal stores (i.e. cache 'by-pass')

Note also, SMT on or off makes no difference to this result; just pin the threads to the first thread on each core.

Your memory speed may or may not clock down depending on the number of DIMM slots per channel in your system, whether those slots are fully or partially populated:

- Single DIMM populated per channel + 2666 single rank → stays at 2666MHz speed
- Single DIMM populated per channel + 2666 dual rank → clocks down to 2400MHz

Optimum performance for R-DIMM memory is achieved where the system has just a single DIMM slot per channel and DDR4-266 dual rank memory is installed

Appendix E: HPCG

What it does: stresses the main memory to such an extent that it has a punishing effect on the FLOPS that each core can produce. The more FLOPS each core produces then the more efficient a system is under this test.

Available from: *http://www.hpcg-benchmark.org/software/index.html*

Note: most of the releases on the website landing page point to CUDA-based benchmarks. To run the benchmark on the CPU you will need 'HPCG 3.0 Reference Code'. Keep clicking through the archives.

Setup:

untar and cd into your hpcg source directory

```
cd setup
cp Make.Linux_MPI Make.EPYC
```

edit Make.EPYC with your favourite editor. Settings described below

```
mkdir /path/hpcg-3.0/mybuild-mpi
cd ../mybuild-mpi
/full/path/to/hpcg-3.0/configure 7601
make -j 64
```

(change 64 to however many threads you have)

Makefile:

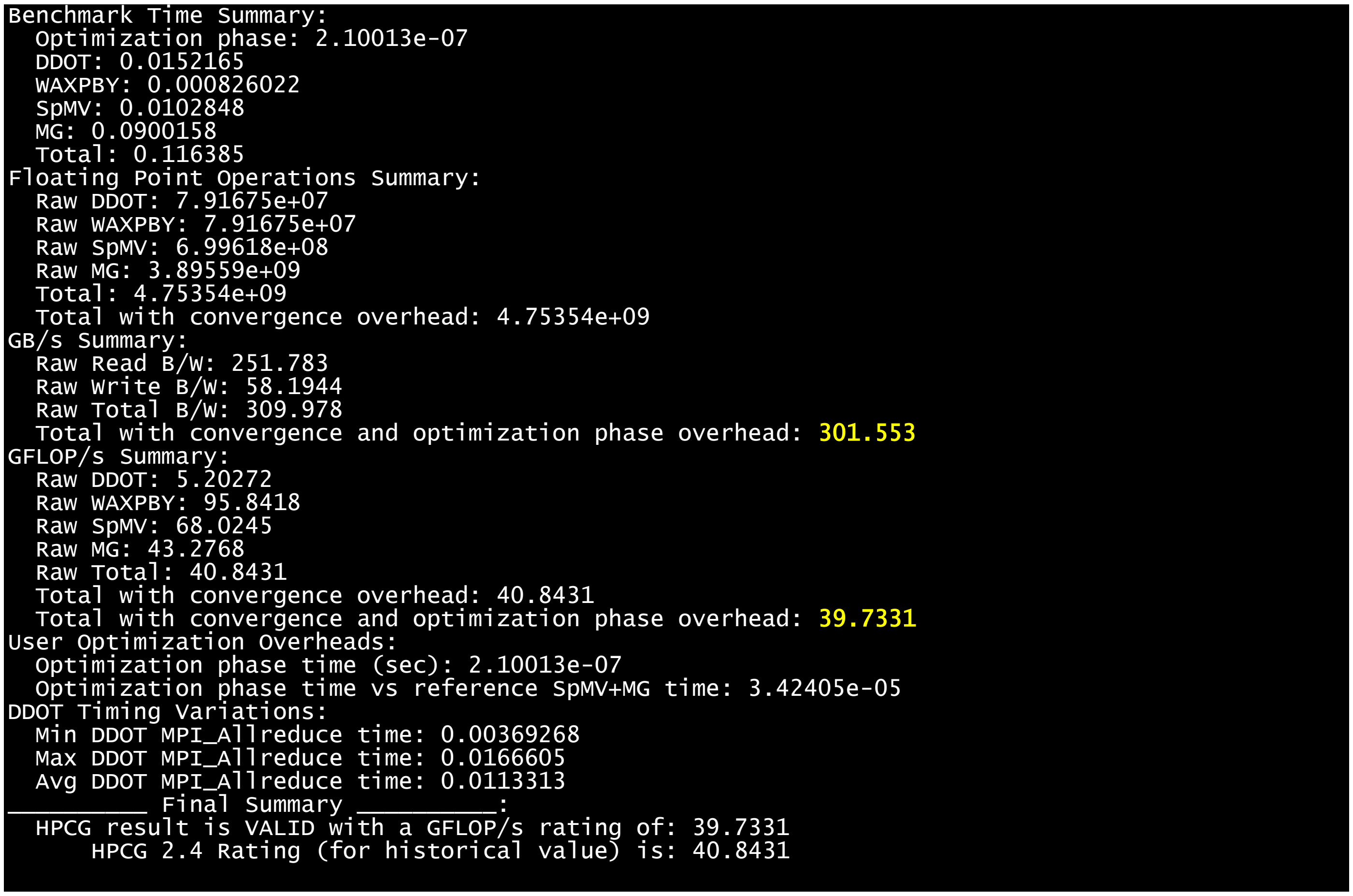
For gcc 7.2, edit */path/hpcg-3.0/setup/Make.EPYC* with your favorite editor and set the following:

```
SHELL        = /bin/sh
CD           = cd
CP           = cp
LN_S         = ln -s -f
MKDIR        = mkdir -p
RM           = /bin/rm -f
TOUCH        = touch
TOPdir       = .
SRCdir       = $(TOPdir)/src
INCdir       = $(TOPdir)/src
BINdir       = $(TOPdir)/bin
MPdir        = /usr/mpi/gcc/openmpi-3.0.0rc6
MPinc        =
MPlib        =
HPCG_INCLUDES = -I$(INCdir) -I$(INCdir)/$(arch) $(MPinc)
HPCG_LIBS     =
HPCG_OPTS      = -DHPCG_NO_OPENMP
HPCG_DEFS      = $(HPCG_OPTS) $(HPCG_INCLUDES)
CXX            = mpicxx
CXXFLAGS        = $(HPCG_DEFS) -Ofast -ffast-math -ftree-
               vectorize -ftree-vectorizer-verbose=0  -fomit-
               frame-pointer    -funroll-loops
LINKER        = $(CXX)
LINKFLAGS     = $(CXXFLAGS)
ARCHIVER      = ar
ARFLAGS       = r
RANLIB        = echo
```

Our `MPdir` shows the path to the OpenMPI bundled with Mellanox OFED. Change accordingly.

Results:

On a dual socket 2 x 32 core system (takes less than a minute on a single node) we derive:


```
Benchmark Time Summary:
  Optimization phase: 2.10013e-07
  DDOT: 0.0152165
  WAXPBY: 0.000826022
  SpMV: 0.0102848
  MG: 0.0900158
  Total: 0.116385
Floating Point Operations Summary:
  Raw DDOT: 7.91675e+07
  Raw WAXPBY: 7.91675e+07
  Raw SpMV: 6.99618e+08
  Raw MG: 3.89559e+09
  Total: 4.75354e+09
  Total with convergence overhead: 4.75354e+09
GB/s Summary:
  Raw Read B/W: 251.783
  Raw Write B/W: 58.1944
  Raw Total B/W: 309.978
  Total with convergence and optimization phase overhead: 301.553
GFLOP/s Summary:
  Raw DDOT: 5.20272
  Raw WAXPBY: 95.8418
  Raw SpMV: 68.0245
  Raw MG: 43.2768
  Raw Total: 40.8431
  Total with convergence overhead: 40.8431
  Total with convergence and optimization phase overhead: 39.7331
User Optimization Overheads:
  Optimization phase time (sec): 2.10013e-07
  Optimization phase time vs reference SpMV+MG time: 3.42405e-05
DDOT Timing Variations:
  Min DDOT MPI_Allreduce time: 0.00369268
  Max DDOT MPI_Allreduce time: 0.0166605
  Avg DDOT MPI_Allreduce time: 0.0113313
__________ Final Summary __________:
  HPCG result is VALID with a GFLOP/s rating of: 39.7331
      HPCG 2.4 Rating (for historical value) is: 40.8431
```

These were generated on a dual socket 2x 32c (7601) with boost=off, SMT=OFF, and using the OpenMPI v 3.0.0 bundled in Mellanox's OFED. The HPCG binary, xhpcg, was compiled with GCC 7.2; identical results were obtained with Intel 2017) For the results to be 'legal' we need to add the '--rt=1800' flag:

```
mpirun -np 64 ./xhpcg 16 16 16 --rt=1830
```

(letting it run for 1830 seconds allows for some initialization).

Appendix F: 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 would recommend using the OFED version that Mellanox bundles and provides for free from their website.

You must use OFED v4.2.1 or greater on AMD EPYC HPC clusters.

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. In particular, we draw your attention to the following flags that should be passed to `./configure` if your MPI library supports them:

```
./configure –with-knem=/path/to/knem –with-mxm=/path/to/mxm
```

or issue

```
./configure -h
```

to check if your MPI library supports these flags. Mellanox include knem and mxm as part of their OFED release.

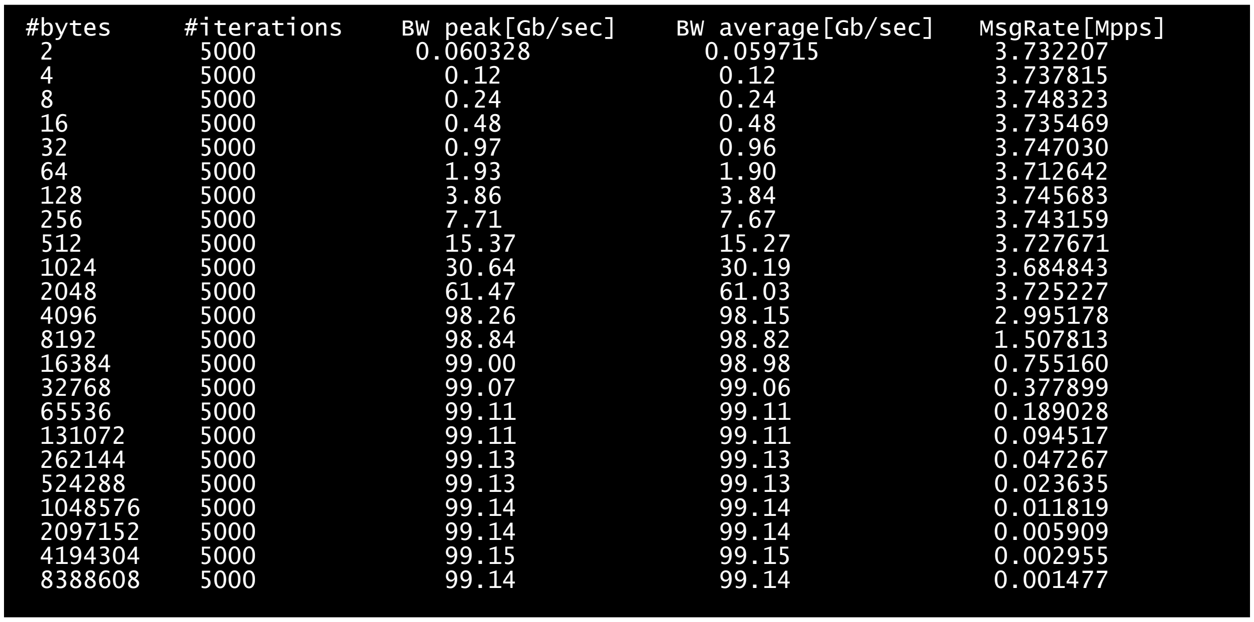
Once OFED is installed there are a number of tests the system manager can issue. For example, to ensure the correct bandwidth profile exists between any 2 compute nodes within the cluster open 2 terminal windows and connect to node-3 and node-5 for example:

- 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. Recall, we can establish which cores to choose for this purpose by inspecting `hwloc-ls` beforehand and noting the cores
- ensured the cores will idle in the C1 State (see discussion above)
- we have a dual port Mellanox card in node-5 so we have explicitly stated which port on the card via the -d flag

In the case of an EDR Mellanox IB network the following profile should be displayed:



```
 #bytes     #iterations    BW peak[Gb/sec]    BW average[Gb/sec]   MsgRate[Mpps]
  2          5000           0.060328           0.059715             3.732207
  4          5000             0.12               0.12               3.737815
  8          5000             0.24               0.24               3.748323
  16         5000             0.48               0.48               3.735469
  32         5000             0.97               0.96               3.747030
  64         5000             1.93               1.90               3.712642
  128        5000             3.86               3.84               3.745683
  256        5000             7.71               7.67               3.743159
  512        5000             15.37              15.27              3.727671
  1024       5000             30.64              30.19              3.684843
  2048       5000             61.47              61.03              3.725227
  4096       5000             98.26              98.15              2.995178
  8192       5000             98.84              98.82              1.507813
  16384      5000             99.00              98.98              0.755160
  32768      5000             99.07              99.06              0.377899
  65536      5000             99.11              99.11              0.189028
  131072     5000             99.11              99.11              0.094517
  262144     5000             99.13              99.13              0.047267
  524288     5000             99.13              99.13              0.023635
  1048576    5000             99.14              99.14              0.011819
  2097152    5000             99.14              99.14              0.005909
  4194304    5000             99.15              99.15              0.002955
  8388608    5000             99.14              99.14              0.001477
```

That is, to witness a uni-directional peak bandwidth of 100Gb/s and one which increase steadily in bandwidth with increasing packet size and then remains at peak bandwidth with increasing packet size. If for example your cores idle in C2 then you would observe a possible brief EDR profile in the middle of the band, with decreasing bandwidth as packet size continues to increase.

The above is a very useful test as it demonstrates no broken system (cables, cards, PCI slots etc) and also that the system is configured correctly.

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 EDR network with EPYC

Systems managers should also monitor /var/log/messages for any Mellanox-related warnings/errors

Appendix G: Additional Resources And References

AMD Developer Central

http://developer.amd.com

AMD Server Gurus Community

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

Wikichip Listing of AMD Zen Architecture

https://en.wikichip.org/wiki/amd/microarchitectures/zen

HPC Toolkit

http://hpctoolkit.org/

SUSE Tuning Guide

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