Linux® Network Tuning Guide for AMD EPYC™ Processor Based Servers

Application Note

Publication #	**56224** Revision: **1.10**
Issue Date:	**May 2018**

Advanced Micro Devices 

Contents

Revision History

Date	Revision	Description
May 2018	1.10	Added Appendix C and D to demonstrate how to test 25GbE and 100GbE NIC
November 2017	1.00	Initial public release.

Chapter 1 Introduction

There is no one golden rule for tuning a network interface card (NIC) for all conditions. Different adapters have different parameters that can be changed. Operating systems also have settings that can be modified to help with overall network performance. Depending on the exact hardware topology, one may have to make different adjustments to network tuning to optimize for a specific workload. With Ethernet speeds going higher, up to 100 Gb, and the number of ports being installed in servers growing, these tuning guidelines become even more important to get the best performance possible.

This guide does not provide exact settings for modifying every scenario, but rather provides some steps to check and modify if it turns out to be beneficial for the scenario. In this case, the steps are focused around TCP/IP network performance. Appendix B shows an example for multiport 10Gb NIC, Appendix C shows an example for multiport 25Gb NIC, and Appendix D shows an example for single port 100Gb NIC of how we verified performance following these settings changes.

One general rule of thumb for all performance testing is to ensure your memory subsystem is properly configured. All IO utilizes data transfers into or out of memory, and so the IO bandwidth can never exceed the capabilities of the memory subsystem. For the maximum memory bandwidth on modern CPUs one must populate at least one DIMM in every DDR channel. For AMD EPYC™ processor-based servers there are eight DDR4 memory channels on each CPU socket, so for a single socket platform that means you must populate all eight memory channels. Likewise, on a dual socket platform one must populate 16 memory channels.

AMD EPYC processors are based on the Zen architecture from AMD. This architecture allows AMD to create large core count processors by using multiple die in a single package. Each die will typically be represented to the operating system (OS) and applications as a unique Non-Uniform Memory Access (NUMA) node. While this document is not intended to describe NUMA architecture, nor the complete EPYC architecture, one should understand NUMA concepts to be able to fully optimize network performance on an EPYC based platform. For more information on NUMA topologies, please reference "NUMA Topology for AMD EPYC™ Naples Family Processors" *here*. Each die, identified as a unique NUMA node, will have local processing cores, local memory, and local IO. You will find in this document that optimal performance comes when explicitly pinning all device driver and applications to the same NUMA node local to the adapter.

In addition to this document, AMD recommends consulting any tuning guides available from your NIC vendor. They will sometimes enable specific tuning options for their devices with parameters that can be modified to further improve performance. One example could be the ability to enable or disable interrupt coalescing. Another could allow the user to change the number of interrupts the device utilizes (this variable will be important in Section 3.2).

Chapter 2 Ensure Transmit and Receive Ring Sizes are Correct

One of the first places to start tuning the TCP/IP stack for adapters under Linux® is to ensure you are using the maximum number of both transmit (TX) and receive (RX) ring buffers. Some drivers will automatically set up the maximum size for the silicon on the NIC, but to be sure this is occurring there are some simple commands that can be executed.

2.1 Use ethtool to Check TX and RX Ring Sizes

The NIC ring values represent the number of buffers that a NIC uses to DMA data into system memory. With more buffers available, more work can be queued up to be processed during a single interrupt. Using the **ethtool** utility, one can find both the current ring size and the maximum allowed by the NIC.

Here's an example:

```
root@testsystem:~# ethtool -g enp33s0f0
Ring parameters for enp33s0f0:
Pre-set maximums:
RX:             4096
RX Mini:        0
RX Jumbo:       0
TX:             4096
Current hardware settings:
RX:             512
RX Mini:        0
RX Jumbo:       0
TX:             512
```

You will notice that for this particular adapter, although the silicon and driver allow for a maximum ring size of 4096, it is currently set for 512. This is 1/8th of the maximum allowed. You can use **ethtool** to modify the current setting to have it match the maximum. This is one key component for improving network performance.

```
root@testsystem:~# ethtool -G enp33s0f0 rx 4096 tx 4096
```

After issuing **ethtool** with a "**-G**" (to set the ring size), you should reissue the original "**-g**" command to verify the change was accepted. These setting changes will not be persistent upon reboot. It is left up to the reader to understand how their Linux distribution needs to be updated to make these changes persistent.

Chapter 3 Optimize Interrupt Handling for Your NIC

By default, Linux® has a service called irqbalance running. The intent of irqbalance is help balance the CPU load across the entire system when processing interrupts. This can result in added latency if the interrupts are not being handled by a CPU core local to the NIC. To prevent such latency additions, we must pin the interrupts to CPU cores that are local to our NIC that we are optimizing. Since, as mentioned earlier, every platform configuration is a little different due to environmental and use case differences, it is recommended that the reader tests the impact irqbalance being enabled versus the manual pinning described in the rest of this chapter.

3.1 Determine Which CPUs Are Local to Your NIC

Use the following steps:

1. To pin the device's interrupts local to the adapter, first determine what NUMA node the device is attached to. To do so, for our example device named **enp33s0f0**, issue the following command:

   ```
   root@testsystem:/sys/class/net/enp33s0f0/device# cat numa_node
   2
   ```

   The highlighted output shows that our device is attached to NUMA node 2.

2. Next, we need to determine what CPU cores are associated with NUMA node 2.

   ```
   root@testsystem:~# lscpu
   Architecture:          x86_64
   CPU op-mode(s):        32-bit, 64-bit
   Byte Order:            Little Endian
   CPU(s):                128
   On-line CPU(s) list:   0-127
   Thread(s) per core:    2
   Core(s) per socket:    32
   Socket(s):             2
   NUMA node(s):          8
   Vendor ID:             AuthenticAMD
   CPU family:            23
   Model:                 1
   Model name:            AMD EPYC 7601 32-Core Processor
   Stepping:              2
   CPU MHz:               1200.000
   CPU max MHz:           2200.0000
   CPU min MHz:           1200.0000
   BogoMIPS:              4391.67
   Virtualization:        AMD-V
   L1d cache:             32K
   L1i cache:             64K
   ```

```
L2 cache:              512K
L3 cache:              8192K
NUMA node0 CPU(s):     0-7,64-71
NUMA node1 CPU(s):     8-15,72-79
NUMA node2 CPU(s):     16-23,80-87
NUMA node3 CPU(s):     24-31,88-95
NUMA node4 CPU(s):     32-39,96-103
NUMA node5 CPU(s):     40-47,104-111
NUMA node6 CPU(s):     48-55,112-119
NUMA node7 CPU(s):     56-63,120-127
Flags:                 ...
```

Upon issuing the **lscpu** command, you will find with the AMD EPYC™ 7601 processors being used in this system, our NUMA node 2 CPU cores are 16-23 and 80-87 as highlighted above.

3.2 Change the Number of Interrupt Queues Used by Your NIC

One key item to help network performance is to not have more interrupt queues than you have CPU cores per NUMA node. With multiple queues assigned to a single core you risk thrashing the interrupt handler swapping between queues. A more efficient manner would be to have a single queue per core and eliminate that thrashing.

There are three types of interrupt queues: receive (RX), transmit (TX), and combined. Combined utilizes a single queue to handle both receive and transmit interrupts. Some vendors still have separate RX and TX queues while others only implement combined queues.

Use the same lscpu output from Section 3.1to determine the number of cores per die. The output above show the physical core numbers followed by the logical core numbers. For instance, the highlighted row for NUMA node 2 has physical cores 16-23 and logical cores 80-87. So, in our case we have 8 physical cores per die, therefore we want 8 total interrupt queues. If you have a NIC that combines the RX and TX interrupts, then the following command would be used:

```
root@testsystem:~# ethtool -L enp33s0f0 combined 8
```

3.3 Pin the Interrupts to CPUs Local to Your NIC

Now that we know what CPU cores are local to our adapter, we need to modify what CPUs handle the interrupts for the NIC.

1. First, we must stop the **irqbalance** service.

   ```
   root@testsystem:~# service irqbalance stop
   ```

   This must be done first to ensure the service does not impact any further changes.

2. Next, we need to pin the interrupts to the CPU cores that are local to the NIC. We will utilize the CPU cores we found in the previous section, cores 16-23 and 80-87. Some NIC vendors will provide scripts to automate this pinning. One such vendor is Mellanox, who provides a script called **set_irq_affinity_cpulist.sh** to bind the interrupts properly. The script is contained in their driver package which you can download from their website. The syntax of the Mellanox script is as follows:

```
root@testsystem:~# set_irq_affinity_cpulist.sh
16,17,18,19,20,21,22,23,80,81,82,83,84,85,86,87 enp33s0f0
```

This will result in the interrupts being pinned to the cores that we previously determined are local to the adapter. The pinning will occur in a round robin fashion, evenly distributing them amongst the cores that you passed to the script.

Just like in section2.1, the changes will not be persistent. Consult your Linux distribution documentation for the necessary steps to run this interrupt pinning script upon each boot.

3.3.1 Manually Pinning NIC Interrupts

If your NIC vendor does not provide such a script, then one must take more care and time in pinning the interrupts. If you can take advantage of a NIC vendor provided script, then you can skip this section.

1. As the case above, you must stop the **irqbalance** service.

```
root@testsystem:~# service irqbalance stop
```

2. The next thing you need to do to manually pin your NIC interrupts is you must determine what interrupts are associated with your NIC. To do so, you need to view the **/proc/interrupts** file.

```
root@testsystem:~#cat /proc/interrupts
```

It is easiest to pipe the output into a file and open with a text editor. Search for all entries that contain your device name, in our case **enp33s0f0**. An example in the output is like this:

```
            CPU0       CPU1
... [output truncated] ...
    341:          0          0     PCI-MSI 17301504-edge    enp33s0f0-TxRx-0
    342:          0          0     PCI-MSI 17301504-edge    enp33s0f0-TxRx-1
... [output truncated] ...
```

The above is a simplified version of the output. The real version will have a column for every CPU core in the system, and a row for every interrupt assigned to the device **enp33s0f0**. It has been truncated to ease reading. You will find the interrupt numbers assigned to the NIC in the highlighted value, in our case interrupts 341 and 342. In a high performance NIC, there will be multiple interrupts created for the adapter – typically one for each transmit and receive queue.

3. Once you know the interrupt or interrupts that you need to manually pin, you now need to modify the **smp_affinity** file for each interrupt. For our example **enp33s0f0**, the first interrupt being modified is 341. The file that must be modified is **/proc/irq/341/smp_affinity**. This file contains a hexadecimal bitmask of all the CPU cores in the system. Since our cores we want to assign start at CPU core 16, and we have 128 total cores, we need to perform the following operation:

   ```
   root@testsystem:~#echo 00000000,00000000,00000000,00010000
   >/proc/irq/341/smp_affinity
   ```

   For the next interrupt, 342, we will pass a bitmask of 00000000,00000000,00000000,00020000. This will be the bitmask to indicate core 17.

These changes will take effect upon the next interrupt. If you generate some network traffic then look at the **/proc/interrupts** file again, you will see that the interrupt count for the CPU cores you pinned to have increased while the previous values remain unchanged.

As was the case for modifying the ring buffer sizes in Section 2.1, these changes will not be persistent upon reboot. Please consult your Linux distribution documentation on how to make the changes permanent.

Chapter 4 Other Tunings to Optimize for Throughput

Many NIC vendors have tuning guides to help end-users optimize around specific use cases. Those use cases usually involve optimizing around the highest throughput possible or the lowest latency possible but can rarely achieve both at the same time. One common way to improve network throughput performance is to enable Large Receive Offload (LRO). To enable LRO on an adapter consult the proper documentation from your NIC provider. Some providers will use standard commands available via **ethtool**, while others might have extra parameters that are needed to fully enable LRO. Please keep in mind that while enabling LRO will help improve your throughput, it could have a negative effect on network latency, so be sure to tune as appropriate for your workload.

For high bandwidth adapters (25 Gb and above), both latency and throughput could also be impacted by the CPU core transitioning between different c-states. One way to prevent this would be to disable c-states through platform BIOS options or via a kernel parameter. Please check with your Linux distribution on the best way to implement this should you choose a kernel parameter method.

Through experimentation we have found that, on occasion, adaptive receive needs to be disabled for some high bandwidth network adapters under select OS's. There is no general guideline for this setting, but one should experiment to determine if it improves performance or not. To modify adaptive receive, issue the following command

```
root@testsystem:~# ethtool -G enp33s0f0 adaptive-rx <on/off>
```

Appendix A OS Specific Tunings

The Linux kernel is constantly being updated. The official mainline releases from The Linux® Foundation are found on *http://kernel.org* (the mainline release is maintained by Linus Torvalds and typically contains the latest and greatest features). The common enterprise level Linux distributions, however, rarely use a mainline kernel.

AMD has been contributing code into the Linux kernel for several years. Most recently, the focus has been around enabling the Zen architecture contained in AMD EPYC™ processors. One area of code contribution has been focused around optimizing the input-output memory management unit (IOMMU) code for AMD EPYC processors. These IOMMU patches can have a direct impact to TCP/IP performance, even in a bare metal (non-virtualized) environment. While the mainline kernel at the time of this writing contains patches to improve IOMMU performance for AMD EPYC processors, not all OSVs have back-ported the patches and included them in their distributions. If a particular distribution does not contain the patches, then the IOMMU driver in the kernel must be set to pass-through mode. Even with the code changes in the recently released drivers, for high bandwidth adapters it may still be required to set the IOMMU to pass-through or disable it. To set the IOMMU to pass-through mode, the following kernel parameter must be passed in during boot time:

iommu=pt

For the following Linux distributions, the following is required to ensure optimal TCP/IP performance in a non-virtualized environment for 10 Gb adapters. In certain cases, it will also work for 25 Gb adapters, but for 100Gb, IOMMU should always be set to pass-through or off.

Distribution + version	Minimum required
Mainline kernel	4.13 or later
SLES 12 SP3	Update patches from October 25, 2017, or later
RHEL 7.5	Kernel parameter – iommu=pt
RHEL 7.4	Kernel parameter – iommu=pt
RHEL 7.3	Kernel parameter – iommu=pt
Ubuntu 16.04	Kernel parameter – iommu=pt

Ubuntu 17.04	Kernel parameter – iommu=pt
Ubuntu 17.10	Default kernel or above
Ubuntu 18.04	Default kernel or above

Appendix B How to Test the Performance of your Network Subsystem – Multi-port 10 Gb example

Now that you have optimized your NIC settings, the next step is typically to perform a micro benchmark to test the performance. In our example, we will utilize **iperf2, version 2.09** (*https://sourceforge.net/projects/iperf2/*) to perform this test. Once installed, the command to execute iperf2 is simply **iperf**. Our test setup is a pair of dual port 10 Gb network adapters in our system under test (SUT) and two driver systems to generate the load. The configuration is shown in the following tables.

Table 1. SUT Network Interface Names and IP Addresses

Port Number	NIC Interface	IP Address	NUMA Node	TCP Port ID
1	enp33s0f0	192.168.100.1	2	5201
2	enp33s0f1	192.168.101.1	2	5203
3	enp34s0f0	192.168.102.1	2	5205
4	enp34s0f1	192.168.103.1	2	5207

Table 2. Driver1 Network Interface Names

Port Number	NIC Interface	IP Address	NUMA Node	TCP Port ID
1	enp33s0f0	192.168.100.2	N/A	5201
2	enp33s0f1	192.168.101.2	N/A	5203

Table 3. Driver2 Network Interface Names

Port Number	NIC Interface	IP Address	NUMA Node	TCP Port ID
1	enp33s0f0	192.168.102.2	N/A	5205
2	enp33s0f1	192.168.103.2	N/A	5207

B.1 Prepare Your Systems for Network Testing

1. Follow the guidelines described in the main portion of this document.

2. Install **iperf** on all three systems (SUT plus two drivers). The exact syntax will vary based on your Linux distribution.

3. For optimal performance, we will want to pin the **iperf** instances to ensure they are running on the same NUMA node as the adapter. If we do not perform this pinning, the Linux® scheduler

has the freedom to assign the various threads to any CPU core in the system. This can result in a wide variation in measured results. To pin **iperf** appropriately, we will install and utilize the application **numactl**. n**umactl** will allow us to pin **iperf** not to a specific core but rather tell the scheduler to utilize cores within the NUMA node specified. The exact syntax to install **numactl** will vary based on your Linux distribution.

B.2 Launch iperf Server Instances

With the above IP addresses, we now need to set up **iperf** as a server on the SUT and driver systems to test bi-directional throughput. Using x-terms in Linux, the easiest method is to open four separate terminal sessions on the SUT and two on each of the drivers. This could all be scripted, but that is beyond the scope of this document

1. In the terminal sessions opened above on the SUT, we will start separate **iperf** server sessions listening on different ports. Launch each of the following commands, one per terminal session.

   ```
   numactl -N <NUMA node> iperf -s -p <unique port ID>
   ```

   In our test setup, the NUMA node to be entered is 2 for the SUT, from the table in the beginning of the appendix. The unique port ID is the TCP Port ID listed in the table. For the first terminal window, utilize 5201, and the last 5207.

2. In the terminal sessions opened above on each of the drivers, we will start separate **iperf** server sessions also listening on different ports. Launch each of the following commands, one per terminal session.

   ```
   iperf -s -p <unique port ID>
   ```

   It is assumed that the driver systems do not need the explicit NUMA pinning, as each driver will have less workload and not require the tuning. The unique port ID is the TCP Port ID listed in the table. We created unique ports IDs on both drivers for simplicity sake, but they could be identical if desired to ease scripting.

B.3 Launch iperf Client Instances

Now that we have server **iperf** instances listening for a connection request on all ports of the SUT and drivers, we need to launch **iperf** client instances to generate the traffic. The easiest method is to open four separate terminal sessions on the SUT and two on each of the drivers.

1. In the terminal sessions opened for the **iperf** clients on the SUT, we will prepare separate **iperf** client sessions. The syntax for launching an **iperf** client is as follows:

   ```
   iperf -c <server IP address> -p <server TCP port to attach to> -P <parallel threads> -w <TCP window size> -i <update frequency in seconds> -t <time to run>
   ```

AMD has found that the TCP window size, **-w** parameter, only needs to be modified from the default size on the SUT, increasing the window to 256KB. The driver client instances do not need this parameter. We have also done a thread sensitivity study and found that two threads per **iperf** instance is the sweet spot of performance on AMD EPYC™ processor platforms. With the syntax above, we will prepare the four terminal sessions on the SUT to launch clients with the following commands (each command goes into a different terminal).

```
numactl -N 2 iperf -c 192.168.100.2 -p 5201 -P 2 -w 256KB -i 5 -t 120s
numactl -N 2 iperf -c 192.168.101.2 -p 5203 -P 2 -w 256KB -i 5 -t 120s
numactl -N 2 iperf -c 192.168.102.2 -p 5205 -P 2 -w 256KB -i 5 -t 120s
numactl -N 2 iperf -c 192.168.103.2 -p 5207 -P 2 -w 256KB -i 5 -t 120s
```

Do not launch the client instances until step 3 below.

2. Much like on the SUT, we now move over to the drivers to prepare the **iperf** clients. On driver1, prepare the following commands – one per terminal session:

```
iperf -c 192.168.100.1 -p 5201 -P 2 -i 5 -t 120s
iperf -c 192.168.101.1 -p 5203 -P 2 -i 5 -t 120s
```

On driver2, prepare the following commands – one per terminal:

```
iperf -c 192.168.102.1 -p 5205 -P 2 -i 5 -t 120s
iperf -c 192.168.103.1 -p 5207 -P 2 -i 5 -t 120s
```

Just like in step 1 above, do not launch the client instances on the drivers until step 3

3. Now that all servers are running and all clients are prepared, launch all eight clients (four on the SUT and two per driver) as close to simultaneously as possible. Once again, using a script to launch the **iperf** client instances could be utilized to achieve this, but is beyond the scope of the document.

B.4 Read and Interpret the Results

The **iperf** server will give an output that looks like this once the client that connected to it has completed its test:

```
------------------------------------------------------------
Server listening on TCP port 5201
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
[  4] local 192.168.100.1 port 5201 connected with 192.168.100.2 port 45070
[  5] local 192.168.100.1 port 5201 connected with 192.168.100.2 port 45074
[ ID] Interval       Transfer     Bandwidth
```

```
[  4]  0.0-120.0 sec  65.9 GBytes  4.72 Gbits/sec
[  5]  0.0-120.0 sec  62.9 GBytes  4.50 Gbits/sec
[SUM]  0.0-120.0 sec   129 GBytes  9.22 Gbits/sec
```

Once all the clients are complete, look at the output from each of the eight clients (four on the SUT and two on each driver). You will find, with the tuning from the guide and launching **iperf** as described in this appendix, that all ports, bi-directionally, will be able to achieve 9.0 Gbps or greater on each of our 10 Gbps capable ports. The highlighted line gives the total measured throughput on an individual port. In our case, it was 9.22 Gbps during this run.

Appendix C How to Test the Performance of your Network Subsystem – Multi-port 25 Gb example

Now that you have optimized your NIC settings, the next step is typically to perform a micro benchmark to test the performance. In our example, we will utilize **iperf2, version 2.09** (*https://sourceforge.net/projects/iperf2/*) to perform this test. Once installed, the command to execute iperf2 is simply **iperf**. Our test setup is a Mellanox ConnectX-4 based dual port 25 Gb network adapters in our system under test (SUT) and two driver systems to generate the load. The configuration is shown in the following tables.

Table 4. SUT Network Interface Names and IP Addresses

Port Number	NIC Interface	IP Address	NUMA Node	TCP Port ID
1	enp33s0f0	192.168.100.1	2	5201
2	enp33s0f1	192.168.101.1	2	5203

Table 5. Driver Network Interface Names

Port Number	NIC Interface	IP Address	NUMA Node	TCP Port ID
1	enp33s0f0	192.168.100.2	N/A	5201
2	enp33s0f1	192.168.101.2	N/A	5203

C.1 Prepare Your Systems for Network Testing

1. Follow the guidelines described in the main portion of this document. Since we used a Mellanox adapter in this example, we took advantage of some of their scripts. Other NIC vendors may have similar scripts. Ensure that the following steps are taken:

2. BIOS settings:

   - Disable c-states
   - Disable IOMMU or set to pass-through mode via the OS kernel parameter explained in Appendix A

3. OS commands:

```
root@testsystem:~#systemctl stop firewalld.service
root@testsystem:~#mlnx_tune -p HIGH_THROUGHPUT
root@testsystem:~#systemctl stop irqbalance.service
```

```
root@testsystem:~#ethtool -L <NIC interface> combined <core count
per die>
root@testsystem:~#set_irq_affinity_cpulist.sh <list of CPU's in the
NUMA node> <NIC_interface>
root@testsystem:~#ethtool -G <NIC_interface> tx 8192 rx 8192
```

4. Install **iperf** on both systems (SUT plus driver). The exact syntax will vary based on your Linux distribution.

5. For optimal performance, we will want to pin the **iperf** instances to ensure they are running on the same NUMA node as the adapter. If we do not perform this pinning, the Linux® scheduler has the freedom to assign the various threads to any CPU core in the system. This can result in a wide variation in measured results. To pin **iperf** appropriately, we will install and utilize the application **numactl**. n**umactl** will allow us to pin **iperf** not to a specific core but rather tell the scheduler to utilize cores within the NUMA node specified. The exact syntax to install **numactl** will vary based on your Linux distribution.

C.2 Launch iperf Server Instances

With the above IP addresses, we now need to set up **iperf** as a server on the SUT and driver systems to test bi-directional throughput. Using x-terms in Linux, the easiest method is to open two separate terminal sessions on the SUT and two on the driver. This could all be scripted, but that is beyond the scope of this document

1. In the terminal sessions opened above on the SUT, we will start separate **iperf** server sessions listening on different ports. Launch each of the following commands, one per terminal session.

   ```
   numactl -N <NUMA node> iperf -s -p <unique port ID>
   ```

   In our test setup, the NUMA node to be entered is 2 for the SUT, from the table in the beginning of the appendix. The unique port ID is the TCP Port ID listed in the table. For the first terminal window, utilize 5201, and the last 5203.

2. In the terminal sessions opened above on the driver, we will start separate **iperf** server sessions also listening on different ports. Launch each of the following commands, one per terminal session.

   ```
   iperf -s -p <unique port ID>
   ```

   Depending on your driver system you may or may not need to pin the iperf instances. The unique port ID is the TCP Port ID listed in the table. We created unique ports IDs on both drivers for simplicity sake, but they could be identical if desired to ease scripting.

C.3 Launch iperf Client Instances

Now that we have server **iperf** instances listening for a connection request on all ports of the SUT and driver, we need to launch **iperf** client instances to generate the traffic. The easiest method is to open two separate terminal sessions on the SUT and two on the driver.

1. In the terminal sessions opened for the **iperf** clients on the SUT, we will prepare separate **iperf** client sessions. The syntax for launching an **iperf** client is as follows:

   ```
   iperf -c <server IP address> -p <server TCP port to attach to> -P <parallel threads> -i <update frequency in seconds> -t <time to run>
   ```

   AMD has done a thread sensitivity study and found that two threads per **iperf** instance is the sweet spot of performance on AMD EPYC™ processor platforms. With the syntax above, we will prepare the four terminal sessions on the SUT to launch clients with the following commands (each command goes into a different terminal).

   ```
   numactl -N 2 iperf -c 192.168.100.2 -p 5201 -P 2 -i 5 -t 120s
   numactl -N 2 iperf -c 192.168.101.2 -p 5203 -P 2 -i 5 -t 120s
   ```

   Do not launch the client instances until step 3 below.

2. Much like on the SUT, we now move over to the driver to prepare the **iperf** clients. On the driver, prepare the following commands – one per terminal session:

   ```
   iperf -c 192.168.100.1 -p 5201 -P 2 -i 5 -t 120s
   iperf -c 192.168.101.1 -p 5203 -P 2 -i 5 -t 120s
   ```

   Just like in step 1 above, do not launch the client instances on the driver until step 3

3. Now that all servers are running and all clients are prepared, launch all four clients (two on the SUT and two on the driver) as close to simultaneously as possible. Once again, using a script to launch the **iperf** client instances could be utilized to achieve this, but is beyond the scope of the document.

C.4 Read and Interpret the Results

The **iperf** server will give an output that looks similar to those found in Appendix B. With our settings, we were able to measure uni-directional performance, per port, at 23.0 Gbps. Bi-directional performance, adding all ports and all directions together, we were able to measure 91.4 Gbps. Increasing the default MTU from 1500 to 8192, one can achieve even higher results.

Appendix D How to Test the Performance of your Network Subsystem – Single-port 100 Gb example

In this next example, we will utilize **iperf2, version 2.09** (*https://sourceforge.net/projects/iperf2/*) to perform a test on a single port of 100Gb ethernet, using a Mellanox adapter. Once installed, the command to execute iperf2 is simply **iperf**. We ran these tests under Ubuntu 18.04. The network configuration is shown in the following tables.

Table 6. SUT Network Interface Names and IP Addresses

Port Number	NIC Interface	IP Address	NUMA Node	TCP Port ID
1	enp33s0f0	192.168.100.1	2	5201

Table 7. Driver1 Network Interface Names

Port Number	NIC Interface	IP Address	NUMA Node	TCP Port ID
1	enp33s0f0	192.168.100.2	N/A	5201

D.1 Prepare Your Systems for Network Testing

1. Follow the guidelines described in the main portion of this document. Since we used a Mellanox adapter in this example, we took advantage of some of their scripts. Other NIC vendors may have similar scripts. Ensure that the following steps are taken:

2. BIOS settings:

   - Disable c-states
   - Disable IOMMU or set to pass-through mode via the OS kernel parameter explained in Appendix A

3. OS commands:

```
root@testsystem:~#systemctl stop firewalld.service
root@testsystem:~#mlnx_tune -p HIGH_THROUGHPUT
root@testsystem:~#systemctl stop irqbalance.service
root@testsystem:~#ethtool -L <NIC interface> combined <core count
per die>
root@testsystem:~#set_irq_affinity_cpulist.sh <list of CPU's in the
NUMA node> <NIC_interface>
root@testsystem:~#ethtool -G <NIC_interface> tx 8192 rx 8192
```

4. Install **iperf** on both systems (SUT plus driver). The exact syntax will vary based on your Linux distribution.

5. For optimal performance, we will want to pin the **iperf** instances to ensure they are running on the same NUMA node as the adapter. If we do not perform this pinning, the Linux® scheduler has the freedom to assign the various threads to any CPU core in the system. This can result in a wide variation in measured results. To pin **iperf** appropriately, we will install and utilize the application **numactl**. n**umactl** will allow us to pin **iperf** not to a specific core but rather tell the scheduler to utilize cores within the NUMA node specified. The exact syntax to install **numactl** will vary based on your Linux distribution.

D.2 Launch iperf Server Instances

With the above IP addresses, we now need to set up **iperf** as a server on the SUT and client on the driver system to test throughput. If in a X-windows environment, open an x-term on both the SUT and driver. If non-graphical, utilize the console for the next steps.

1. In the terminal sessions opened above on the SUT, we will start an **iperf** server session listening on a unique port. Launch the following command:.

   ```
   numactl -N <NUMA node> iperf -s -p <unique port ID>
   ```

   In our test setup, the NUMA node to be entered is 2 for the SUT, from the table in the beginning of the appendix. The unique port ID is the TCP Port ID listed in the table above. Our example used the following command line for the server instance, launched on the SUT:

   ```
   numactl -N 2 iperf -s -p 5201
   ```

D.3 Launch iperf Client Instance

Now that we have server **iperf** instance listening for a connection request on the SUT, we need to launch an **iperf** client instance on the driver to generate the traffic. Just as in Section C.2, either open an x-term on the driver or go to the console

1. In the terminal session opened for the **iperf** client on the driver, we will prepare an **iperf** client session. The syntax for launching an **iperf** client is as follows:

   ```
   numactl -N <NUMA node> iperf -c <server IP address> -p <server TCP port to attach to> -P <parallel threads> -w <TCP window size> -i <update frequency in seconds> -t <time to run>
   ```

   AMD has done a thread sensitivity study and found that four threads per **iperf** instance is the sweet spot of performance on AMD EPYC™ processor platforms with a 100Gb link. Our example used the following command line for the client instance, launched on the driver system

   ```
   numactl -N 2 iperf -c 192.168.100.2 -p 5201 -P 4 -i 5 -t 60s
   ```

D.4 Read and Interpret the Results

The **iperf** server will give an output that looks like this once the client that connected to it has completed its test:

```
------------------------------------------------------------
Server listening on TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
[  4] local 10.10.10.1 port 5001 connected with 10.10.10.2 port 41732
[  5] local 10.10.10.1 port 5001 connected with 10.10.10.2 port 41734
[  6] local 10.10.10.1 port 5001 connected with 10.10.10.2 port 41738
[  7] local 10.10.10.1 port 5001 connected with 10.10.10.2 port 41736
[  4]  0.0-60.0 sec   164 GBytes  23.5 Gbits/sec
[  5]  0.0-60.0 sec   164 GBytes  23.5 Gbits/sec
[  6]  0.0-60.0 sec   164 GBytes  23.5 Gbits/sec
[  7]  0.0-60.0 sec   165 GBytes  23.6 Gbits/sec
[SUM]  0.0-60.0 sec   657 GBytes  94.1 Gbits/sec
```

Once the client has finished the test, look at the output. You will find, with the tuning from the guide and launching **iperf** as described in this appendix, that all ports will be able to achieve 90 Gbps or greater 100 Gbps capable ports. The highlighted line gives the total measured throughput on an individual port. In our case, it was 94.1 Gbps during this run.