VM-VM Single Flow Performance on IPSec Crypto Offload on Nvidia ConnectX devices

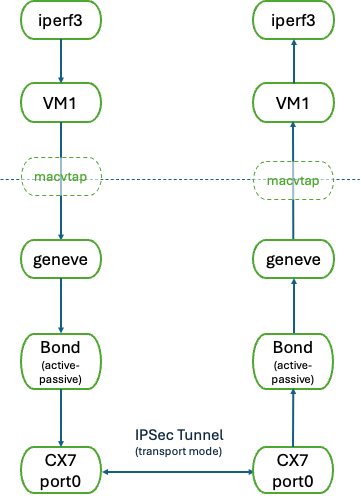
# Overview

Leveraging Geneve for traffic virtualization, IPSec for data protection, and bonding interfaces for redundancy delivers a comprehensive solution for modern network environments. This approach ensures efficient traffic management, robust security, and high availability, making it suitable for dynamic and critical applications in data centers and cloud infrastructures. Using an **active-passive bond** (e.g., in Linux, via mode=1 bonding) provides redundancy and failover capabilities that are not available when using a single NIC device directly.

NVIDIA BlueField-2 and ConnectX-7 offer two IPSec offload options for securing traffic between VMs on different hosts. The more preferred and performant option is of course  [Packet Offload or otherwise known as Full Offload.](https://docs.nvidia.com/networking/display/mlnxofedv24010331/ipsec+full+offload)

In this document though, I discuss [crypto offload](https://docs.nvidia.com/networking/display/mlnxofedv24010331/ipsec+crypto+offload), which enables the user to offload IPsec crypto encryption and decryption operations to the NIC hardware. This is a much less performant option than packet offload. However, this may be the only choice in some scenarios. For example, the host configuration does not allow or is restricted from using SRIOV and passthrough VFs to the VM. Crypto offload is supported on earlier Linux distributions and kernels unlike packet offload which requires the latest Linux kernels (version 6.6 or later). .Finally, compared to kernel or software based IPSec, Crypto offload can provide up to 2X performance boost for a single network flow and scales to be up to 4X better for 4 flows.

This document addresses how I was able to maximize performance using crypto offload using iperf3 TCP as the benchmark using the configuration below.



### Host/Hypervisor

* Ubuntu 24.04 LTS
  + Only using inbox drivers and commands (no MLNX\_OFED)
* Custom-built Linux kernel: 6.10.0-rc5+
  + Includes Linux kernel commit#27eb6de954a2 bonding: add ESP offload features when slaves support
  + Plus other patches.
* 200 Gbps ConnectX-7 connected directly (no switch) across a 200 Gbps link
  + Firmware Version: 28.41.1000
  + All traffic is between port 0s of the two CX7s
  + Flow Control off.
* Default sysctl settings

### VM

* Ubuntu 24.04 LTS
  + Default kernel: 6.8.0-49-generic
* Default sysctl settings
* iperf3 version 3.16

### Test

* On server VM::
  + iperf3 -s -B <server IP address> -i 60
* On client VM::
  + iperf3 -i 10 -c 192.169.1.1 -B 192.169.1.2 -t 60

# Host/Hypervisor Setup

The following script (gsetup) can be used for setting up the above test environment. It needs to be modified to match the ConnectX7 interface names on the test system.

$ ./gsetup

Usage: gsetup [-p] [-k|N] [-H] [-m <mtu>] -s|-c

-p: Use PF instead of bond

-k: Kernel encryption (no crypto offload)

-n: No IPSec

-m <mtu>: Set MTU for devices

-s : Server side (for iperf3)

-c : Client side (for iperf3)



Note that the script also allows for running this setup without a bond device and choosing kernel encryption in lieu of crypto offload, which can be used to make performance comparisons.

# VM Installation

The create\_vm script can be used to initialize and create the VMs for running these tests. Pass the “-i” flag if you also need to pre-install the dependent packages.

$ create\_vm

Starting install...

Retrieving 'vmlinuz' | 0 B 00:00:00 ...

Retrieving 'initrd' | 0 B 00:00:00 ...

Allocating 'vlinux1-023.qcow2' | 0 B 00:00:00 ...

Creating domain... | 0 B 00:00:00

Domain is still running. Installation may be in progress.

You can reconnect to the console to complete the installation process.

$ virsh

Welcome to virsh, the virtualization interactive terminal.

Type: 'help' for help with commands

'quit' to quit

virsh # list

Id Name State

-----------------------------

1 vlinux1-023 running

virsh # console 1

Connected to domain 'vlinux1-023'

Escape character is ^] (Ctrl + ])



Connect to the VM’s console as shown above to complete the installation.

# VM-VM Tuning

The following describes some techniques for how I tuned the VMs performing all the necessary tuning outside the VM (that is inside the host/hypervisor).

## NUMA Node

On NUMA systems, it will be useful to find the NUMA affinity of the CX7 device. This can be done as follows (in the example, the NUMA node is 0 for the CX7).

# Find the NUMA node of my CX7

$ lspci | grep Mellanox

17:00.0 Ethernet controller: Mellanox Technologies MT2910 Family [ConnectX-7]

17:00.1 Ethernet controller: Mellanox Technologies MT2910 Family [ConnectX-7]

$ cat /sys/bus/pci/devices/0000:17:00.0/numa\_node

0



## Pinning CPUs

This can be done during virt-install or later by updating the XML file.

### During VM Install

* Pass the cpuset option to --vcpus.
  + For example: --vcpus 6,cpuset=20,22,24,26,28,30
  + It is best to choose all CPUs in the same node as the NIC device.
    - The NUMA nodes of the CPUs in the system can be found using either *numactl -H* or *lscpu*

### Post-install XML update

You can instead directly update the XML any time after installation using virsh edit <domain> and adding the following:

 <vcpu placement='static'>6</vcpu>

<cputune>

<vcpupin vcpu='0' cpuset='20'/>

<vcpupin vcpu='1' cpuset='22'/>

<vcpupin vcpu='2' cpuset='24'/>

<vcpupin vcpu='3' cpuset='26'/>

<vcpupin vcpu='4' cpuset='28'/>

<vcpupin vcpu='5' cpuset='30'/>

</cputune>



### Isolating the CPUs

* It will be useful to further “isolate these CPUs” in the host by adding the following to the kernel command line:
  + Note that isolating the CPUs prevents the host from running any other threads on them other than those that are explicitly scheduled to run on them.
* For example:
  + Edit /etc/default/grub and append (or add) the following (or similar line) to the GRUB\_CMDLINE\_LINUX variable:  
    isolcpus=no\_hz,domain,20,22,24,26,28,30
    - For example, I have:  
      GRUB\_CMDLINE\_LINUX="isolcpus=no\_hz,domain,16,18,20,22,24,26,28,30
      * Note: Here I have added two additional isolated CPUs (16 and 18) for potential future uses.
  + Then run:
    - sudo update-grub
  + Reboot.
  + After reboot, you can verify this by checking the value of */proc/cmdline*, and you should see something like in the example below:

$ cat /proc/cmdline

BOOT\_IMAGE=/vmlinuz-6.10.0+ root=/dev/mapper/ubuntu--vg-ubuntu--lv ro isolcpus=no\_hz,domain,16,18,20,22,24,26,28,30



## Pinning Memory

This can again be done during virt-install or later by updating the XML file.

### During VM Install

* Pass the --numatune to virt-install.
  + For example: ---numatune 0
  + Make sure to choose the node matching your pinned CPUs.

### Post-install XML update

You can directly update the XML any time after installation using virsh edit <domain> as follows:

 <numatune>

<memory mode='preferred' nodeset='0'/>

</numatune>



## virtio optimizations

### Queue Sizes

* Increase rx\_queue\_size to 1024
  + Default is 256
  + Allows the virtual NIC to buffer more incoming packets before they’re processed by the guest OS.
  + Larger RX queue size means the CPU can handle bursts of network traffic better, reducing the chances of packet drops
    - Very important for TCP
* Update the XML file as follows by adding the rx\_queue\_size line as shown below.

 <interface type='direct'>

<mac address='52:54:00:7a:0c:d6'/>

<source dev='geneve0' mode='vepa'/>

<target dev='macvtap0'/>

<model type='virtio'/>

**+ <driver name='vhost' queues='4' rx\_queue\_size='1024'/>**

<alias name='net1'/>

<address type='pci' domain='0x0000' bus='0x08' slot='0x00' function='0x0'/>

</interface>



### Poll Cycle Adjustment

* poll-us: maximum number of microseconds that could be spent on busy polling by the vhost tap driver.
* The default is zero. That is, the vhost thread relies on interrupts for packet processing rather than polling the virtio rings.
* Might be useful to change this to say 200 or 500 microseconds.
* Unfortunately, libvirt does not support this directly, although qemu does through a command line option.
  + One possible solution then for those who need this extra optimization is to add a qemu: command line stanza to the XML

## Further CPU optimizations

### Optimizing the vhost CPU thread

Often, the vhost thread in the host kernel may conflict with the vCPU threads, especially on the TX side. Here is an example when running iperf3.

 PID USER PR NI VIRT RES SHR S P %CPU %MEM TIME+ COMMAND

9089 libvirt+ 20 0 17.3g 1.1g 21324 R 20 40.4 0.9 1:35.74 vhost-9076

9096 libvirt+ 20 0 17.3g 1.1g 21324 R 20 40.4 0.9 1:34.87 CPU 5/KVM



Both the vhost thread in the host kernel (this is the thread that manages the other side of the virtio queue in the host kernel) and one of the vcpu thread in the VM are running on the same CPU which effectively reduces the CPU time the vcpu gets to run in the VM.

There is a tag called emulatorpin to move vhost that we can add to the XML after installation.

<cputune>

<vcpupin vcpu='0' cpuset='20'/>

<vcpupin vcpu='1' cpuset='22'/>

<vcpupin vcpu='2' cpuset='24'/>

<vcpupin vcpu='3' cpuset='26'/>

<vcpupin vcpu='4' cpuset='28'/>

<vcpupin vcpu='5' cpuset='30'/>

<emulatorpin cpuset='16,18'/>

</cputune>



### Optimizing for softirq (on RX side)

In some cases, we also find softirq and a vcpu from the VM competing for time on the same CPU. Here is an example when running iperf3 (on the server or RX side):

 PID USER PR NI VIRT RES SHR S P %CPU %MEM TIME+ COMMAND

10071 libvirt+ 20 0 17.4g 1.1g 21104 R 28 68.1 0.9 1:25.04 CPU 4/KVM

10064 libvirt+ 20 0 17.4g 1.1g 21104 S 16 34.6 0.9 0:42.51 vhost-10053

189 root 20 0 0 0 0 S 28 25.6 0.0 0:41.38 ksoftirqd/28



The main iperf3 thread is running on vcpu1 = CPU 1/KVM, which is CPU 22. The RX thread is running on vCPU 4 = CPU 28 on the host where we are getting the softirqs also.

In this case, the bandwidth can drop significantly..

Simply changing the vCPU on which we are running iperf3 by using the taskset command inside the VM does not always work. For example, here is another possible scenario:

 PID USER PR NI VIRT RES SHR S P %CPU %MEM TIME+ COMMAND

2978 libvirt+ 20 0 17.3g 1.1g 20936 R 20 98.7 0.9 3:51.77 CPU 0/KVM

26 root 20 0 0 0 0 R 1 98.3 0.0 3:48.23 ksoftirqd/1

2979 libvirt+ 20 0 17.3g 1.1g 20936 R 22 98.0 0.9 3:48.28 CPU 1/KVM

2973 libvirt+ 20 0 17.3g 1.1g 20936 R 16 57.9 0.9 2:14.35 vhost-2962



In this case, for some reason the interrupts are coming into CPU 1, which is not in node 0, where the NIC device is located, and again the impact can be quite bad.

**Caveat:** Solving this issue in general depends upon a lot of factors including the workload behavior, including I/O characteristics, how IRQs are being distributed for all the active devices on the system and their impact on performance, etc.

Solving this for a single flow (or even multiflows) on a generally idle system is a lot easier.

Here is one simple strategy for how one could resolve this.

1. Find the Linux IRQ associated with the ConnectX7 that has affinity to the CPU above

(CPU 28 in the first example, CPU 1 in the second)

* + This can be found either by checking the CPU vs interrupt counts for CX7 in /proc/interrupts and if necessary making some educated guesses.

1. Next set /proc/irq/<irq\_num>/smp/affinity to another CPU in the same node that the CX7 device has affinity and which is not being used by any of the VM’s vCPUs.

There is an immediate improvement in performance as in the example below

# Run iperf3

# On server (RX) side

$ iperf3 -s -B 192.169.1.1 -i 60

-----------------------------------------------------------

Server listening on 5201 (test #2)

-----------------------------------------------------------

Accepted connection from 192.169.1.2, port 39673

[ 5] local 192.169.1.1 port 5201 connected to 192.169.1.2 port 54467

[ ID] Interval Transfer Bitrate

[ 5] 0.00-60.05 sec 78.4 GBytes 11.2 Gbits/sec

[ 5] 60.05-120.02 sec 78.5 GBytes 11.2 Gbits/sec

[ 5] 120.02-180.03 sec 78.3 GBytes 11.2 Gbits/sec

...

# Performance is less than the expected 14.5 Gbps.

# Running top -H

PID USER PR NI VIRT RES SHR S P %CPU %MEM TIME+ COMMAND

26 root 20 0 0 0 0 R 1 98.7 0.0 3:55.67 ksoftirqd+

3144 libvirt+ 20 0 17.6g 1.1g 21064 R 20 98.7 0.9 3:59.20 CPU 0/KVM

3145 libvirt+ 20 0 17.6g 1.1g 21064 R 22 98.7 0.9 3:56.08 CPU 1/KVM

3139 libvirt+ 20 0 17.6g 1.1g 21064 S 16 58.3 0.9 2:17.77 vhost-3128

# Some IRQs are going to CPU 1.

# Parsing the output of /proc/interrupts for the CX7 at PCI address 0000:17:00.0

$ ./m\_irqs.py 0000:17:00.0 --start

Saved IRQ counts in .m\_irqs.start

# Wait a few seconds

$ ./m\_irqs.py 0000:17:00.0 --end --compress

HW(VIRQ) Diff

0(201) 1062

1(202) 4

9(221) 406290

17(229) 50

CPU Diff

0 4

1 50

8 1062

16 406290

# We guess (another tool might give the answer without the need to guess)

# that one of the CX7's IRQs,namely IRQ 229 has affinity to CPU 1, which is on

# node 1, while the device has affinity to node 0.

# This is clearly not ideal

# Bind IRQ 229 to CPU 4 (on the same node)

$ ./irq\_affinity -s 229 4

IRQ 229 was bound to 1, now bound 4

# top -H confirms the change

PID USER PR NI VIRT RES SHR S P %CPU %MEM TIME+ COMMAND

3144 libvirt+ 20 0 17.6g 1.1g 21064 R 20 99.3 0.9 23:32.28 CPU 0/KVM

3145 libvirt+ 20 0 17.6g 1.1g 21064 R 22 99.0 0.9 23:23.06 CPU 1/KVM

44 root 20 0 0 0 0 R 4 95.7 0.0 3:47.67 ksoftirqd+

3139 libvirt+ 20 0 17.6g 1.1g 21064 R 16 56.5 0.9 13:40.99 vhost-3128

# And we see an immediate difference in performance

[ 5] 0.00-60.05 sec 78.4 GBytes 11.2 Gbits/sec

[ 5] 60.05-120.02 sec 78.5 GBytes 11.2 Gbits/sec

[ 5] 120.02-180.03 sec 78.3 GBytes 11.2 Gbits/sec

[ 5] 180.03-240.06 sec 97.3 GBytes 13.9 Gbits/sec

[ 5] 240.06-300.04 sec 101 GBytes 14.5 Gbits/sec

[ 5] 300.04-360.04 sec 101 GBytes 14.5 Gbits/sec

[ 5] 360.04-420.06 sec 101 GBytes 14.5 Gbits/sec

## 

## MTU considerations

It is very important in this kind of configuration, where the host (hypervisor) adds additional layers (Geneve, IPSec) over a packet coming from the VM, that the GSO size is computed correctly by the VM’s kernel. Large packets must be segmented into MTU-size chunks for transmission and the GSO size is essentially the necessary segmentation length of the individual segments after accounting for or setting aside headroom for all the different headers that will be added to the packet as it traverses the networking stack.

This segmentation can be done in software by the kernel or offloaded to the NIC - there can be up to a 5X performance hit when doing this in software.

The VM’s kernel computes the GSO size taking into account only the size of IP and TCP headers. However, as the packet moves through the hypervisor, the Geneve and IPSec encapsulation headers will also be added to the packet. The packet processing in the hypervisor takes a slow path in the kernel because the extra bytes of geneve+IPSec headers results in the total size (gso\_size + headers) > MTU. In the slow path, the hypervisor kernel goes ahead and does the segmentation directly instead of offloading to the NIC.

This is not a bug and so the simple fix is to ensure that the MTU of the VM’s NIC is at least 100 bytes < MTU of the host’s bond and/ or PF interface.

Unfortunately, there is no simple way to set the vNIC’s MTU at install or later through the XML for the kind of vNIC configuration we have above. Instead, you will need to configure the netplan for the VMs to hard-code their MTU requirements or have some way of communicating this through the DHCP mechanism.