High Performance Storage Solution for Virtual Environment on Xinnor RAID Engine and Kioxia PCIe 5.0 Drives

XINNOR KIOXIA

Contents

XINNOR KIOXIA

1. Introduction

SSDs continue to advance in speed, particularly with the market shift to NVMe™ PCIe® 5.0 SSDs. With this technology, a single SSD can now achieve remarkable sequential read speeds of up to 14GB/s and sequential write speeds of 7GB/s, along with handling millions of Input/Output per second (IOps). In modern servers, it is common to find 24 or more of these high-performance SSDs. Given that a single application may not fully utilize such high performance, multiple virtual machines often run on the same system. Therefore, ensuring protection against drive failures and maintaining stable performance with low latency becomes increasingly crucial, especially for handling mission-critical applications.

Cloud services (public and private) rely on a software-defined approach, enabling them to deploy resources on demand when it matters. In this research, we will consider two options for creating a software RAID array and using it as storage resources for virtual machines.

2. Objective of Testing

The aim of this testing is to assess the applicability of software RAID arrays with high-performance NVMe drives, using the created volumes for virtual machines requiring fast storage.

We plan to demonstrate the scalability of the solution and provide an assessment of the test results. Two methods of creating software RAID arrays will be considered: 1) mdraid configured for optimal performance with NVMe SSDs, operating in the Linux kernel space, and 2) a commercial product by Xinnor (xiRAID Opus), operating in the user space.

The created volumes are exported to virtual machines using vhost interface. The vhost target is a process running on the host machine and is capable of exposing virtualized block devices to QEMU instances or other arbitrary processes.

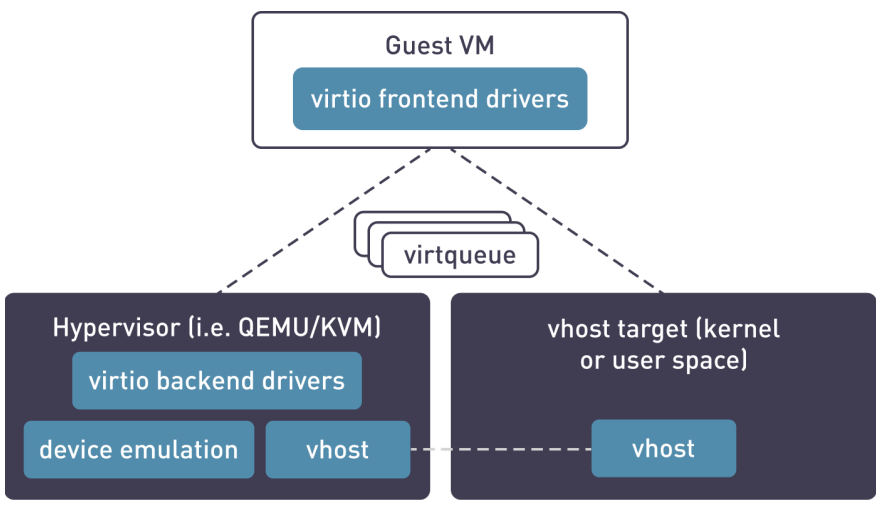


Figure 1. vhost operation scheme

For mdraid, we utilized the kernel vhost target and configured it using the targrtcli utility. xiRAID SPDK has a built-in target that operates in user space.

3. Testing Environment

Hardware Configuration:

- **Motherboard:** Supermicro H13DSH
- **CPU:** Dual AMD EPYC 9534 64-Core Processors
- **Memory:** 773,672 MB
- **Drives:** 10 x 3.20 TB KIOXIA CM7 Series Enterprise NVMe SSDs (KCMYXVUG3T20)

Software Configuration:

- **OS:** Ubuntu 22.04.3 LTS
- **Kernel:** Version 5.15.0-91-generic
- **xiRAID Opus:** Version xnr-859
- **QEMU Emulator:** Version 6.2.0

RAID Configuration:

To avoid intra-NUMA communication, we created two RAID groups (4+1 configuration), each utilizing drives from a single NUMA. The stripe size was set to 64K. A full RAID initialization was conducted prior to benchmarking.

Each RAID group was divided into 8 segments, with each segment being allocated to a virtual machine via a dedicated vhost controller.

Summary of Resources Allocated:

- **RAID Groups:** 2
- **Volumes:** 16
- **vhost Controllers:** 16
- **VMs:** 16, with each using segmented RAID volumes as storage devices.

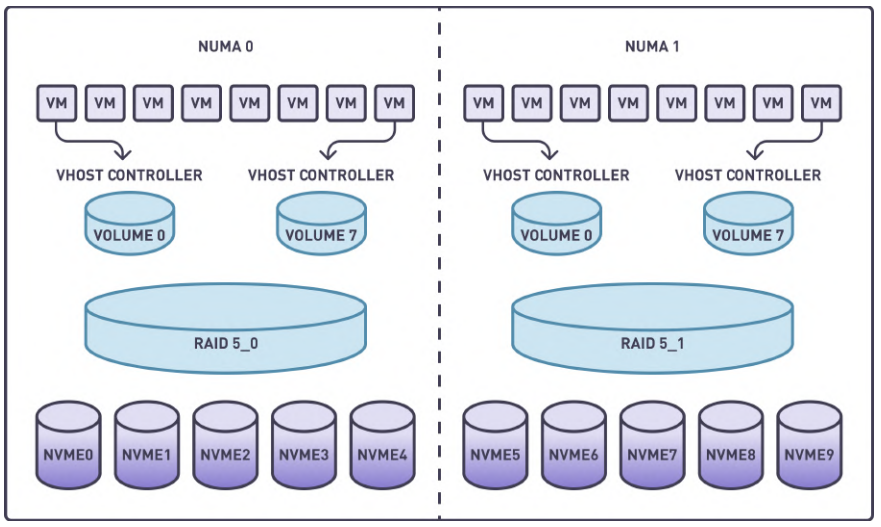


Figure 2. Distribution of virtual machines, vhost controllers, RAID groups and NVMe drives

The CPUs have 8 Core Complex Die (CCDs), with each CCD containing 8 ZEN cores with shared L3 cache.

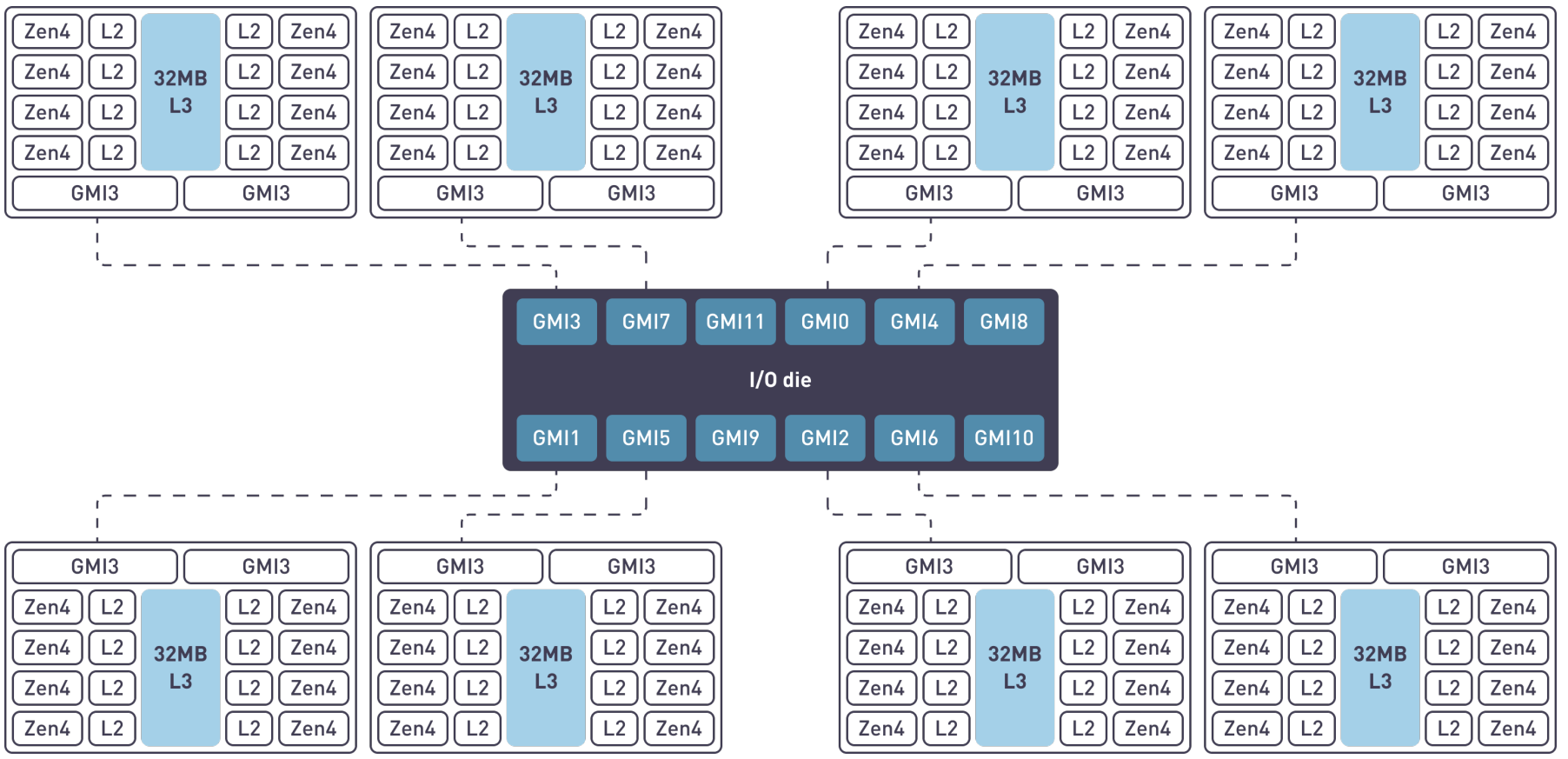


Figure 3. 8 CCD Configuration

The proper allocation of resources across cores is a crucial component in achieving high performance. One of the feature of xiRAID Opus is to allow the user to specify which cores will be dedicated to run the applications.

Placing VMs and their corresponding vhost controllers on the same CCD is a way to enhance performance by reducing latency and improving data throughput. This is because it takes advantage of the direct and faster communication paths within the same CCD, as opposed to across different CCDs, which can be slower due to inter-CCD communication overhead. This approach is especially beneficial in systems where VMs require high-speed data processing and minimal latency.

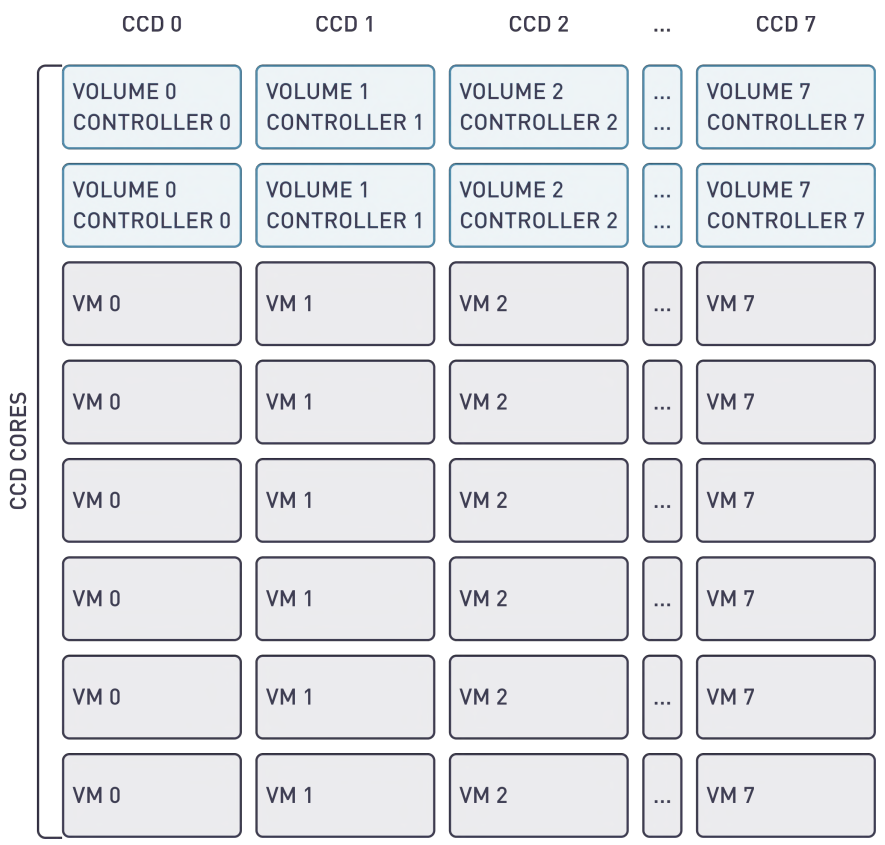


Figure 4. Distribution of resources across CCD cores

Virtual Machine Configuration

The configuration differs when utilizing xiRAID Opus and mdraid. In case of xiRAID Opus, specific cores can be allocated, while the remaining cores are utilized by the virtual machines. When operating in the kernel space with mdraid, it is not possible to allocate specific cores; instead, all cores are used concurrently for both storage infrastructure and virtual machines.

CPU Allocation: 6/8 vCPUs are designated per VM, directly corresponding to the host server's physical CPU cores. Process-to-core affinity is managed with the taskset utility to optimize performance.

QEMU CPU Configuration:

```
-cpu host -smp 6 / 8
```

QEMU Memory Configuration:

Memory Allocation: Each VM is provisioned with 16 GB of RAM via Hugepages. Memory is pre-allocated and bound to the same NUMA node as the allocated vCPUs to ensure efficient CPU-memory interaction.

```
-m 16G -object memory-backend-
file,id=mem,size=16G,mem-path=/dev/
hugepages,share=on,prealloc=yes,host-
nodes=0,policy=bind
```

- **Operating System:** The VMs run Debian GNU/Linux 12 (Bookworm)
- **Benchmarking Tool:** The 'fio' tool, version 3.33

4. Setting up xiRAID Opus for VM Infrastructure

In this section, we will briefly outline the steps required to create an array, volumes, and export them to a virtual machine. For detailed instructions, please refer to the manual.

Launch Command:

Running an application with a 300GB HugePage on a NumaNode and utilizing 2 cores on each CCD.

```
xnr_xiraid --xnr-hugemem=300000,300000 -m
0x818181818181818181818181
```

RAID Creation Steps:

1. **Drive Attachment:** Attach drives using the xnr_cli drive-manager.
2. **RAID Creation:** Create a RAID array with the following command:

```
raid create -n vhost -l 5 -s 64 -d
0000:01:00.0n1,0000:03:00.0n1,0000:05:00.0n1,0000:
07:00.0n1,0000:41:00.0n1
```

3. **Initialization:** Wait for the RAID initialization process to complete.
4. **Volume Creation:** Create 8 volumes per RAID using the script:

```
/root/spdk/scripts/rpc.py bdev_split_create vhost
8 -s 512000
```

'-s' means each volume size

5. **Volume Exporting:** Export volumes via a vhost controller with the command (to be implemented for each controller):

```
xnr_cli vhost-blk create -c vhost.0 -d vhostp0 -C
[0,7]
…
```

'-c' is a mask used to launch specific target

5. Test Configuration Overview

The objective of this test scenario was to evaluate the cumulative performance of virtual machines (VMs) utilizing Vhost target and to assess the scalability of I/O operations across multiple CPU cores. The assessment involved conducting tests with both a single VM and a group of 16 VMs. Each VM executed a series of FIO benchmarks to measure I/O performance under various workload conditions, including:

- Small, random read operations (4KiB)
- Small, random write operations (4KiB)
- Mixed random I/O with a read-write ratio of 70:30 (4KiB)
- Large, sequential read operations (1MiB)
- Large, sequential write operations (1MiB)

These workloads simulate a range of scenarios to provide a comprehensive understanding of the system's performance characteristics.

6. Test Results

Small Random Operations Test Results

Random Operations Performance, K IOps

Operation	xiRAID 1 VM (6 Cores), 2 Cores for RAID	mdraid 1 VM (8 Cores)	xiRAID 16 VMs (6 Cores each), 32 Cores for RAID	mdraid 16 VMs (8 Cores each)
Random Read (4KiB)	1,985	308	23,112	4,176
Random Write (4KiB)	710	154	2,988	768
Mixed I/O (70:30, 4KiB)	810 / 347	160 / 69	7,088 / 2,960	1,336 / 572
Degraded Random Read (4KiB)	1,400	173	17376	766

Random Operations Efficiency (16 VMs)

The efficiency is calculated by comparing the RAID performance with the theoretical performance of 10 drives. Each drive is capable of 2.7 M IOps random read, 0.7 M IOps random write.

Operation	xiRAID	mdraid
Random Read (4KiB)	86%	15%
Random Write (4KiB)	65%	22%
Degraded Random Read (4KiB)	80%	4%

Mean Latency, µs

Operation	xiRAID 1 VM (6 Cores), 2 Cores for RAID	mdraid 1 VM (8 Cores)	xiRAID 16 VMs (6 Cores each), 32 Cores for RAID	mdraid 16 VMs (8 Cores each)
Random Read (4KiB)	98	623	140	736
Random Write (4KiB)	270	1,242	1,082	3,955
Mixed I/O (70:30, 4KiB)	143 / 216	798 / 931	194 / 579	1,217 / 2,522
Degraded Random Read (4KiB)	137	1,107	176	3,950

99,95 Latency, µs

Operation	xiRAID 1 VM (6 Cores), 2 Cores for RAID	mdraid 1 VM (8 Cores)	xiRAID 16 VMs (6 Cores each), 32 Cores for RAID	mdraid 16 VMs (8 Cores each)
Random Read (4KiB)	188	1,385	388	1,467
Random Write (4KiB)	1,549	2,507	6,132	20,579
Mixed I/O (70:30, 4KiB)	1,090 / 1,221	1,745 / 2,024	1,876 / 5,412	5,538 / 9,372
Degraded Random Read (4KiB)	255	2,180	766	19,820

Outcomes

Random Reads

In the observed results, each RAID configuration is capable of delivering close to one million IOps per core, while maintaining a notably low latency. The results are exceptional, especially for latency-sensitive applications: the single-VM RAID setup achieved latency levels below 100 microseconds, with the 99.95th percentile latency remaining under 200 microseconds. These figures are indicative of superior performance, demonstrating the system's capability to handle intensive workloads with minimal delay.

Upon scaling, the performances decline by approximately 30% due to fio tool consuming all available CPU resources. This indicates a trade-off between scaling and CPU availability, with the latter becoming a limiting factor under extensive load conditions.

Random Writes

While write performance in RAID implementation are impacted by the necessary additional read-modify-write operations, the results show that xiRAID is 4 times more efficient then MDRAID.

As expected, in writing the scaling losses are more pronounced as the additional read-modify-write cycles increase resource contention with the fio processes. This effect is especially noticeable in scaled-up configurations where numerous VMs are competing for the same CPU resources.

Degraded Mode

xiRAID performance in degraded mode for read operations are 20X better versus MDRAID, displaying minimal performance loss and only a slight increase in latency. This robustness in degraded mode is particularly advantageous for applications where consistent read performance is critical, even in the event of partial system failure or maintenance scenarios.

Mdraid and the kernel space target demonstrate significantly lower efficiency levels, making them less economically viable.

Sequential Operations Test Results

Sequential Operations Performance, K IOps

Operation	xiRAID 1 VM (6 Cores), 2 Cores for RAID	mdraid 1 VM (8 Cores)	xiRAID 16 VMs (6 Cores each), 32 Cores for RAID	mdraid 16 VMs (8 Cores each)
1M Sequential Read	55.9	29.3	116	108
1M Sequential Write	17	5.8	50	10.2
1M Sequential Read Degraded	41.8	4	99.2	8

Sequential Operations Efficiency (16 VMs)

The efficiency is calculated by comparing with theoretical performance of 10 drives with the following measured single drive performance: 14 GB/s at sequential read, 6.75 GB/s at sequential write.

Operation	xiRAID	mdraid
1M Sequential Read	83%	77%
1M Sequential Write (4KiB)	93%	19%
1M Sequential Read Degraded	89%	7%

Outcomes

The performance of the sequential operations closely approaches the theoretical maximum and remains at high even in degraded modes. This robustness allows for the deployment of data-intensive applications within a virtual infrastructure, ensuring they can operate efficiently without significant performance penalties, even in suboptimal conditions.

The performance of sequential degraded read and sequential write on mdraid significantly lags behind xiRAID, even in small-scale installations, making the solution unsuitable for performance-sensitive applications.

7. xiRAID Opus Overview

xiRAID Opus (Optimized Performance in User Space) is the user space version of xiRIAD software RAID engine. It leverages the SPDK (Storage Performance Development Kit) libraries to move outside of linux Kernel to allow customers not to be concerned regarding Linux Kernel versions, bypassing the update process. This architecture grants full CPU control, facilitating straightforward execution with affinity to specific CPU cores. xiRAID Opus goes beyond the confines of Linux hosts, demonstrating adaptability by facilitating straightforward portability to other operating systems and seamless integration with specialized hardware such as Data Processing Units (DPUs). xiRAID Opus has built-in interfaces for accessing data from virtual infrastructures: vhost SPDK, NVMe-oF™.

8. Summary

The testing aimed to assess the performance and scalability of virtual machines (VMs) using xiRAID Opus Vhost, focusing on I/O operations across multiple CPU cores. Through various FIO benchmarks, including small random reads and writes, mixed I/O, and large sequential operations, the evaluation provided insights into system performance under diverse workload conditions. Notably, results revealed that each RAID configuration could deliver close to one million IOps per core with low latency. The single-VM RAID setup exhibited exceptional latency levels below 100 microseconds, demonstrating superior performance even under intense workloads or in degraded mode. Additionally, sequential operations approached theoretical maximums, ensuring high performance even in case of a drive failure, thereby supporting the efficient deployment of data-intensive applications within virtual infrastructures.

In terms of performance, mdraid and kernel vhost target significantly lag behind the xiRAID Opus. Additionally, the inconsistency of certain settings greatly complicates administration tasks. Mdraid demonstrates minimal effectiveness in degraded mode, which is precisely the scenario for which RAID exists.

Appendix 1: Fio Configurations

Small Block operations

```
[global]
# Set block size to 4 kilobytes
bs=4k
# Enable direct I/O for bypassing the buffer cache
direct=1
# Set the queue depth per thread/job to 32
iodepth=32
# Run 6 parallel jobs
numjobs=6
# Disable random map
norandommap=1
# Enable group reporting for a summarized output
group_reporting
rw=randread/randwite/randrw
# Use the Linear Feedback Shift Register generator
for random numbers
random_generator=lfsr
# Use io_uring, a high-performance I/O engine
ioengine=io_uring
hipri=1
# Fixed buffers option optimizes memory handling
during direct I/O
fixedbufs=1
# Register the files with the kernel for more
efficient I/O operations
registerfiles=1
```

Sequential Operations

```
# Job section for the device
[vda]
# Specify the device file for the test
filename=/dev/vda
```

```
# Fio Global Configuration
[global]
# Set the block size for read/write operations.
bs=1M
# Use direct I/O for bypassing the buffer cache
direct=1
# Set the number of I/O operations that can be
queued per job
iodepth=32
numjobs=2
# Aggregate and report I/O statistics for all jobs
together
group_reporting
# Use the Linux-native asynchronous I/O facility
ioengine=libaio
# Set the starting offset increment for each
subsequent job
offset_increment=20%
```

```
# Device-specific configuration
[vda]
# Define the device to test
filename=/dev/vda
```

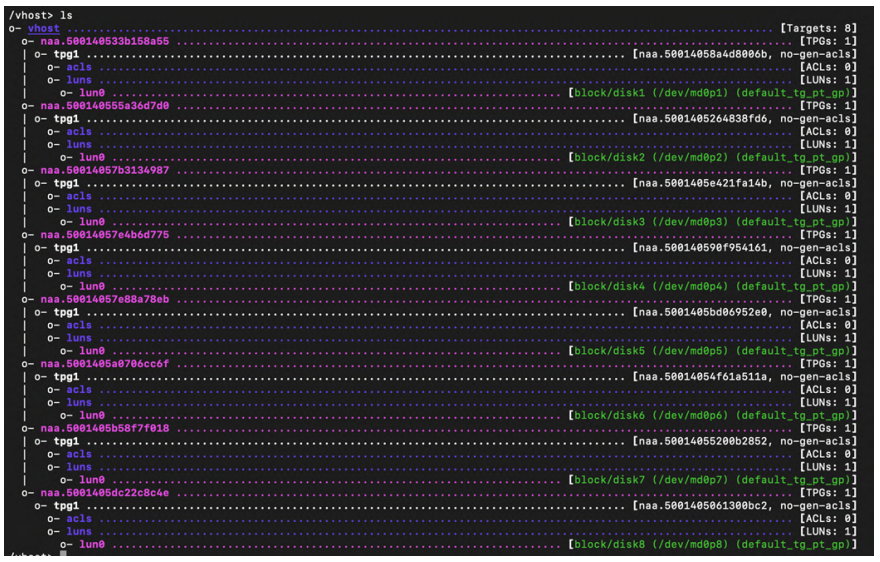
Appendix 2: mdraid Configuration

On Each NUMA Node:

```
md0 : active raid5 nvme40n2[5] nvme45n2[3]
nvme36n2[2] nvme46n2[1] nvme35n2[0]
      12501939456 blocks super 1.2 level 5, 64k
chunk, algorithm 2 [5/5] [UUUUU]
```

```
Bitmaps disabled
cat /sys/block/md0/md/group_thread_cnt
16

Vhost target
```



Example Code for Launching VMs

```
taskset -a -c $CPU qemu-system-x86_64 -enable-kvm
-smp 8 -cpu host -m 32G -drive
file=$DISK_FILE,format=qcow2 --nographic \
-device vhost-scsi-
pci,wwpn=naa.5001405dc22c8c4e,bus=pci.0,addr=0x5
```

Learn more at **xinnor.io**

request@xinnor.io

Learn more at **kioxia.com**