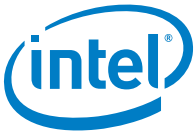

SPDK NVMe BDEV Performance Report Release 18.04

Test Date: July, 2018

Performed by:

John Kariuki (john.k.kariuki@intel.com)

Vishal Verma (vishal4.verma@intel.com)

Acknowledgements

James R Harris (james.r.harris@intel.com)

Benjamin Walker (benjamin.walker@intel.com)

Nate Marushak (nathan.marushak@intel.com)

Prital Shah (prital.b.shah@intel.com)

Revision History

Date	Revision	Comment
08/01/2018	V1.0	Complete write-up
08/23/2018	V1.1	Added review feedback.
09/20/2018	V1.2	Added review feedback.
09/26/2018	V1.3	Added review feedback.
10/12/2018	V1.4	Added review feedback.

Contents

Audience and Purpose

This report is intended for people who are interested in comparing the performance of the SPDK block device layer vs the Linux Kernel block device layer (`4.15.15-300.fc27.x86_64`). It provides performance and efficiency information between the two block layers under various test workloads.

The purpose of reporting these tests is not to imply a single "correct" approach, but rather to provide a baseline of well-tested configurations and procedures with repeatable and reproducible results. This report can also be viewed as information regarding best known method/practice when performance testing SPDK NVMe block device layer.

Test setup

Hardware Configuration

Item	Description
Server Platform	1-node Intel Server S2600WFT 2 Riser Cards (2U 3-slot PCIe riser card): Each riser(iPC – A2UL8RISER2) has 3 PCIe x8 electrical 2 PCIe switch on Socket 0 and 3 PCIe switch on socket 1: Each switch (iPC – AXXP3SWX08040) is a 4 port PCIe x8 Switch AIC
CPU	2x Intel® Xeon® Platinum 8180 Processor (38.5MB L3, 2.50 GHz) https://ark.intel.com/products/120496/Intel-Xeon-Platinum-8180-Processor-38_5M-Cache-2_50-GHz Number of cores 28, number of threads 56
Memory	Total 192 GBs / 12 channels / 16 GB/ 2667 MHz DDR4
BIOS	The following BIOS patch was installed to mitigate Spectre and Meltdown vulnerabilities BIOSX0115_SKX B1-13f_SKX H0-043
Operating System	Fedora 27
Linux kernel version	4.15.15-300.fc27.x86_64
SPDK version	18.04 SPDK_BDEV_IO_CACHE_SIZE changed from 256 to 2048
Storage	22x Intel® SSD DC P4600™ (1.6TB, 2.5in PCIe 3.1 x4, 3D1, TLC): Firmware Version QDV10130 (Boot) 1x Intel® SSD 520 Series (240GB, 2.5in SATA 6Gb/s, 25nm, MLC)

glibc	2.26

BIOS settings

Item	Description	
BIOS	Configuration 1: For Maximum Performance we used the following configuration to enable Turbo and other performance features on the CPU. Hyper threading Enabled CPU Power and Performance Policy <Performance> CPU C-state No Limit CPU P-state Enabled Enhanced Intel® Speedstep® Tech Enabled Turbo Boost Enabled	Configuration 2: We turned off Turbo and other performance features to obtain repeatable results in the scalability test using the following configuration. Hyper threading Disabled CPU Power and Performance Policy <Performance> CPU C-state No Limit CPU P-state Disabled Enhanced Intel® Speedstep® Tech Disabled Turbo Boost Disabled

Performance results are based on testing as of 07/06/2018 and may not reflect all publicly available security updates. See configuration disclosure for details. No product can be absolutely secure.

SSD Preconditioning

An empty NAND SSD will often show read performance far beyond what the drive claims to be capable of because the NVMe controller knows that the device is empty and completes the read request successfully without performing any data transfer. Therefore, prior to running each performance test case we preconditioned the SSDs by writing 128K blocks to the device sequentially to fill the SSD capacity (including the over-provisioned areas) twice and force the internal state of the device into some known state. Additionally, the 4K 100% random writes performance decreases from one test to the next until the NAND management overhead reaches steady state because the wear-levelling activity increases dramatically until the SSD reaches steady state. Therefore, to obtain accurate and repeatable results for the 4K 100% random write workload, we ran the workload for 90 minutes before starting the benchmark test and collecting performance data. For a highly detailed description of exactly how to force an SSD into a known state for benchmarking see the SNIA Solid State Storage Performance Test Specification.

Introduction to SPDK Block Device Layer

SPDK Polled Mode Driver: The NVMe PCIe driver is something that you usually would expect to be part of the kernel and your application would interact with the driver via the system call interface. SPDK takes a different approach. SPDK unbinds the NVMe devices from the kernel and bind the hardware queues to a userspace NVMe driver and from that point on your application will access the device queues directly from userspace. The SPDK NVMe driver is a C library that may be linked directly into an application that provides direct, zero-copy data transfer to and from NVMe SSDs. It is entirely passive, meaning that it spawns no threads and only performs actions in response to function calls from the application. The library controls NVMe devices by directly mapping the PCI BAR into the local process and performing MMIO. The SPDK NVMe driver is asynchronous, which means that the driver submits the I/O request as an NVMe submission queue entry on a queue pair and the function returns immediately, prior to the completion of the NVMe command. The application must poll for I/O completion on each queue pair with outstanding I/O to receive completion callbacks.

SPDK Block Device Layer: SPDK further provides a full block stack as a user space library that performs many of the same operations as a block stack in an operating system. The SPDK block device layer often simply called **bdev**, is a C library intended to be equivalent to the operating system block storage layer that often sits immediately above the device drivers in a traditional kernel storage stack. The bdev module provides an abstraction layer that provides common APIs for implementing block devices that interface with different types of block storage device. An application can use the APIs to enumerate and claim SPDK block devices, and then perform asynchronous I/O operations (such as read, write, unmap, etc.) in a generic way without knowing if the device is an NVMe device or SAS device or something else. The SPDK NVMe bdev module can create block devices for both local PCIe-attached NVMe device and remote devices exported over NVMe-oF. In this report, we benchmarked the performance and efficiency of the bdev in the local PCIe-attached NVMe devices usescase and demonstrate the benefits of the SPDK approaches like user-space polling, asynchronous I/O, no context switching etc. under different workloads.

SPDK provides an FIO plugin for integration with Flexible I/O benchmarking tool. The quickest way to generate a configuration file with all the bdevs for locally PCIe-attached NVMe devices is to use the gen_nvme.sh script as shown below.

```
# scripts/gen_nvme.sh > fio.spdk_bdev_conf
# cat fio.spdk_bdev_conf
[Nvme]
TransportId "trtype:PCIe traddr:0000:88:00.0" Nvme1
TransportId "trtype:PCIe traddr:0000:89:00.0" Nvme2
TransportId "trtype:PCIe traddr:0000:8a:00.0" Nvme3
TransportId "trtype:PCIe traddr:0000:8b:00.0" Nvme4
TransportId "trtype:PCIe traddr:0000:8f:00.0" Nvme5
TransportId "trtype:PCIe traddr:0000:90:00.0" Nvme6
TransportId "trtype:PCIe traddr:0000:91:00.0" Nvme7
TransportId "trtype:PCIe traddr:0000:b2:00.0" Nvme8
TransportId "trtype:PCIe traddr:0000:b3:00.0" Nvme9
TransportId "trtype:PCIe traddr:0000:b4:00.0" Nvme10
TransportId "trtype:PCIe traddr:0000:b5:00.0" Nvme11
TransportId "trtype:PCIe traddr:0000:d8:00.0" Nvme12
TransportId "trtype:PCIe traddr:0000:d9:00.0" Nvme13
TransportId "trtype:PCIe traddr:0000:1b:00.0" Nvme14
TransportId "trtype:PCIe traddr:0000:1c:00.0" Nvme15
```

```
TransportId "trtype:PCIe traddr:0000:1d:00.0" Nvme16
TransportId "trtype:PCIe traddr:0000:23:00.0" Nvme17
TransportId "trtype:PCIe traddr:0000:5e:00.0" Nvme18
TransportId "trtype:PCIe traddr:0000:21:00.0" Nvme19
TransportId "trtype:PCIe traddr:0000:20:00.0" Nvme20
TransportId "trtype:PCIe traddr:0000:22:00.0" Nvme21
TransportId "trtype:PCIe traddr:0000:5f:00.0" Nvme22
```

Figure 1 : NVMe bdev configuration

Configure an fio job file to use bdevs, by setting the *ioengine=spdk_bdev* and adding the *spdk_conf* parameter whose value points to the bdev configuration file as shown below.

```
[global]
ioengine=spdk_bdev
spdk_conf=/home/john/spdk/scripts/perf/nvme/fio.spdk_bdev_conf
direct=1
time_based=1
norandommap=1
ramp_time=60s
runtime=600s
thread=1
group_reporting=1
percentile_list=50:99:99.9:99.99:99.999

rw=${RW}
rwmixread=${MIX}
bs=${BLK_SIZE}
iodepth=${IODEPTH}
numjobs=1

[filename1]
filename=Nvme1n1
filename=Nvme2n1
filename=Nvme3n1
filename=Nvme4n1
filename=Nvme5n1
cpumask=0x10000000

[filename2]
filename=Nvme6n1
filename=Nvme7n1
filename=Nvme8n1
filename=Nvme9n1
filename=Nvme11n1
cpumask=0x20000000

[filename3]
filename=Nvme14n1
filename=Nvme15n1
filename=Nvme16n1
filename=Nvme20n1
filename=Nvme13n1
cpumask=0x1

[filename4]
filename=Nvme17n1
filename=Nvme18n1
filename=Nvme19n1
filename=Nvme21n1
filename=Nvme22n1
cpumask=0x2
```

Figure 2 : Adding NVMe bdevs to the fio configuration file

The fio configuration file in figure 2, shows how to define multiple fio jobs and assign NVMe bdevs to each job. Each job is also pinned to a CPU core on the same NUMA node as the NVMe SSDs that the job will access to minimize cross socket I/O latency.

Finally, to use the bdev fio plugin specify the LD_PRELOAD when running fio.

LD_PRELOAD=<path to spdk repo>/examples/bdev/fio_plugin/fio_plugin fio <fio job file>

Test Case 1: SPDK NVMe BDEV IOPS Test

Purpose: The purpose of this test case was to measure the maximum performance in IOPS/Core of the NVMe block layer on a single CPU core. This test was done using different benchmarking tools (bdevperf vs. fio) to understand the overhead of benchmarking tools.

Test Workloads: The following Random Read/Write mixes were used

- 4KB 100% Random Read
- 4KB 100% Random Write
- 4KB Random 70% Read 30% Write

Test Execution: For each workload we followed the following steps:

1) Precondition all the SSDs prior to each workload test to ensure accurate and repeatable results.
2) Run each test workload: Start with a configuration that has 22 Intel P4600x NVMe devices and decrease the number of SSDs by 2 on each subsequent run.
    - This shows us the minimum number of SSDs we need to achieve the maximum IOPS/Core.
    - Starting with 22 SSDs and reducing the number of SSDs on subsequent eliminates having to precondition between runs because all SSDs used in the subsequent run were used in the previous run so they should still be in a steady state.
3) Repeat each workload three times for each number of SSDs. The data reported is the average of the 3 runs.

Item	Description
Test Case	Test SPDK NVMe BDEV IOPS Test
Test configuration	Number of Intel P4600x NVMe SSDs: Start with all 22 SSDs and decrease the number of SSDs in each subsequent test as follows 22, 20, 18 …1 by removing 2 SSDs on each test run to avoid preconditioning between test runs. Number of CPU Cores: 1 Queue Depth: 256 (QD=32 for 4KB 100% random write workload) Block Size: 4096 We started the test with Turbo Boost disabled using BIOS configuration 2. NOTE: On systems with PCIe switches between the SSDs and root port, use SSDs on behind different PCIe switches to avoid reaching the switch saturation point before saturating the CPU core.

Test Execution	1. Use the following command to pre-condition the SSDs. ./perf -q 32 -s 131072 -w write -t 1200 Workload Specific Pre-conditioning: For 4K 100% Random Write workload, the SSDs were 90 minutes using the workload by setting the ramp_time=5400 in the fio configuration file. For the test cases where the benchmarking tool was perf/bdefperf we did the workload specific preconditioning using perf with the following parameters: ./perf -q 32 -s 4096 -w randwrite -t 5400 2. Run the benchmark tool (fio/perf/bdevperf) for each workload below. 3. The test results are the average performance (IOPS and average latency) observed during the 3 tests.

Table 1 : NVMe bdevs IOPS/Core (Block Size=4K, 1 CPU Core, Turbo=Disabled, fio)

Workload	IOPS/Core
4K Random Read (QD=256)	1,663,555.42
4K Random Write (QD=32)	1,473,846.80
4K 70% Read 30% Write (QD=256)	1,454,642.06

The data in table 1 shows the maximum IOPS on a single CPU Core for the three 4K Workloads using fio. All the bdevs used in this test were on the same NUMA node as the CPU core.

IOPS/Core with Turbo Boost

The Intel® Xeon® Platinum 8180 Processor has a base frequency of 2.5 GHz and maximum Turbo frequency of 3.8 GHz. In this test, Turbo was enabled (using BIOS configuration 1) and the maximum IOPS/Core for the 3 workloads measured.

Table 2 : NVMe bdevs IOPS/Core with Turbo Boost (Block Size=4K, 1 CPU Core, fio) Turbo=Disabled vs. Turbo=Enabled

Workload	IOPS/Core (Turbo=Disabled)	IOPS/Core (Turbo=Enabled)	Performance Improvement
4K Random Read (QD=256)	1,663,555.42	2,213,111.60	33%
4K Random Write (QD=32)	1,473,846.80	1,881,063.11	28%
4K 70% Read 30% Write (QD=256)	1,454,642.06	1,967,776.60	35%

Depending on the workload the Intel® Turbo Boost Technology improved performance by 28 % – 35 %.

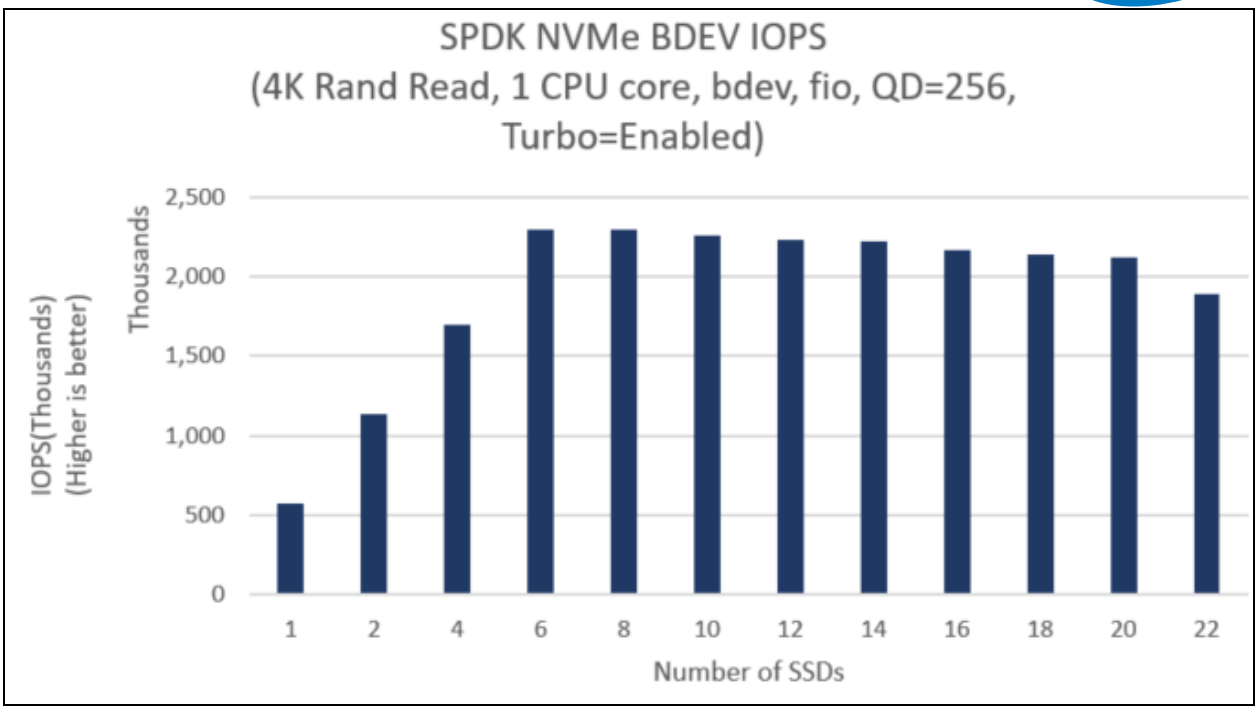

Figure 3: NVMe bdev IOPS scalability with addition of SSDs (4K 100% Random Read IOPS, 1 CPU Core, Turbo=Enabled, QD=256)

Fio vs. bdevperf IOPS/Core

The fio benchmarking tool provides a lot of great features to enable users to quickly define workloads, scale the workloads and collect many data points for detailed performance analysis. SPDK provides the bdevperf benchmarking tool that provides minimal capabilities needed to define basic workloads and collects a limited amount of data. This test compares the performance in IOPS/core of the fio and bdevperf benchmarking tools against the same bdevs.

Table 3 : NVMe bdevs IOPS/Core (Block Size=4K, 1 CPU Core, Turbo=Enabled) fio vs. bdevperf

Workload	IOPS/Core (fio+bdev)	IOPS/Core (bdevperf)	Performance Gain
4K Random Read (QD=256)	2,213,111.60	3,502,175.58	58%
4K Random Write (QD=32)	1, 881,063.11	No Data	
4K 70% Read 30% Write (QD=256)	1,967,776.60	3,292,617.28	67%

No Data: System lacks enough storage capacity to saturate a CPU core.

The overhead of the benchmarking tools is important when you are testing a system that is capable of millions of IOPS/Core. Using a benchmarking tool that has minimal overhead like the SPDK bdevperf yields up to 67% more IOPS/Core vs. fio.

NVMe BDEV vs. Polled-Mode Driver IOPS/Core

Table 4 : NVMe bdevs vs polled-mode driver IOPS/Core (Block Size=4K, 1 CPU Core, Turbo=Enabled)

Workload	IOPS/Core (bdevperf)	IOPS/Core (perf)	Performance Gain
4K Random Read (QD=256)	3,502,175.58	4,603,559.85	31%

In this test case, we compared the throughput of the NVMe bdev with that of the polled-mode driver. How to read this data? The SPDK block layer provides several key features at a cost of up to 31% more CPU utilization. If you are building a system with many SSDs that is capable of millions of IOPS, you can take advantage of the block layer features at the cost of approximately 1 additional CPU core for every 3 I/O cores.

Test Case 2: SPDK NVMe BDEV I/O Cores Scaling

Purpose: The purpose of this test case is to demonstrate the I/O throughput scalability of the NVMe bdev module with the addition of more CPU cores to perform I/O. The number of CPU cores used was scaled as 2, 4, 6 and 10.

Test Workloads: We use the following Random Read/Write mixes

- 4KB 100% Random Read
- 4KB 100% Random Write
- 4KB Random 70% Read 30% Write

Test Execution:

Repeat the following steps for each workload.

1. Precondition all the SSDs prior to each workload test to ensure accurate and repeatable results
2. Run fio for the workload under test for a given number of CPU cores.
    a. Start with all 22 SSDs and decrease the number of SSDs in each subsequent test run by removing 1 SSD from each fio job. For example, when testing with 2 CPU cores, the fio configuration file had 2 jobs; each job was configured with 11 SSDs on the first run, 10 SSDs in the next run, 9 SSDs third run and so on until we have 1 SSD in job in the last run. The test with 4 CPU cores starts out with 2 jobs on NUMA node 1 with 6 SSDs each and 2 jobs on NUMA node 0 with 5 SSDs each.
    b. Pin fio jobs to CPU cores and minimize cross socket I/O traffic by configuring the fio jobs to use SSDs in the same NUMA node. The platform has 9 SSDs on NUMA node 0 and 13 SSDs on NUMA node 1, therefore, in some test runs the jobs on NUMA node 0 had to use SSDs on NUMA node 1 (E.g. when testing with 2 CPU cores, the fio job on NUMA node 0 started out with 9 SSDs from NUMA node 0 and 2 SSDs from NUMA node 1). In subsequent runs we removed SSDs that were on a different socket than the fio job before removing the SSDs in the same socket.
3. Repeat each workload three times for each number of CPU cores (X). The data reported is the average of the 3 runs.
4. Increase the CPU core count by adding 2 CPU cores (one core on each NUMA node). This is accomplished by adding 2 fio jobs in the fio configuration file; one job is pinned to a CPU core on NUMA node 0 and the other to a CPU core on NUMA node 1. We redistribute the SSDs between the fio jobs and minimize the cross I/O traffic by assigning all jobs SSDs that are in the same NUMA locality before assigning the jobs SSDs on a different NUMA node. Go back to step 1.

Item	Description

Test Case	Test SPDK NVMe BDEV I/O Cores Scalability Test
Test Configuration	Number of Intel P4600x NVMe SSDs: 22 Number of CPU Cores: Scaled as follows 2, 4, 6, 8 and 10 Queue Depth: 256 (QD=32 for 4KB 100% random write workload) Block Size: 4096 BIOS configuration 2 setting were used for this test to disable Turbo Boost so that we could obtain repeatable results when using more than 1 CPU core.
Test Execution	We used the following command to Pre-condition the SSDs. ./perf -q 32 -s 131072 -w write -t 1200 Workload Specific Pre-conditioning: For 4K 100% Random Write workload, the workload was ran for 90 minutes to pre-condition the SSD by setting the ramp_time=5400 in the fio configuration file. The following workload were used for this test case. 4K 100% Random Read 4K 100% Random Write 4K 100% Random Read/Write 70/30 The test results are the average performance (IOPS and average latency) observed during the 3 tests.

Table 5 : NVMe bdev I/O core scalability (4K 100% Random Read IOPS, Turbo=Disabled, QD=256)

# of CPU Cores	Throughput (IOPS)	Avg. Latency (usec)
1	1,661,143.19	555.85
2	3,223,063.04	573.23
4	6,455,655.37	552.14
6	9,524,237.60	539.56
8	9,734,551.53	572.91
10	10,183,569.18	552.37

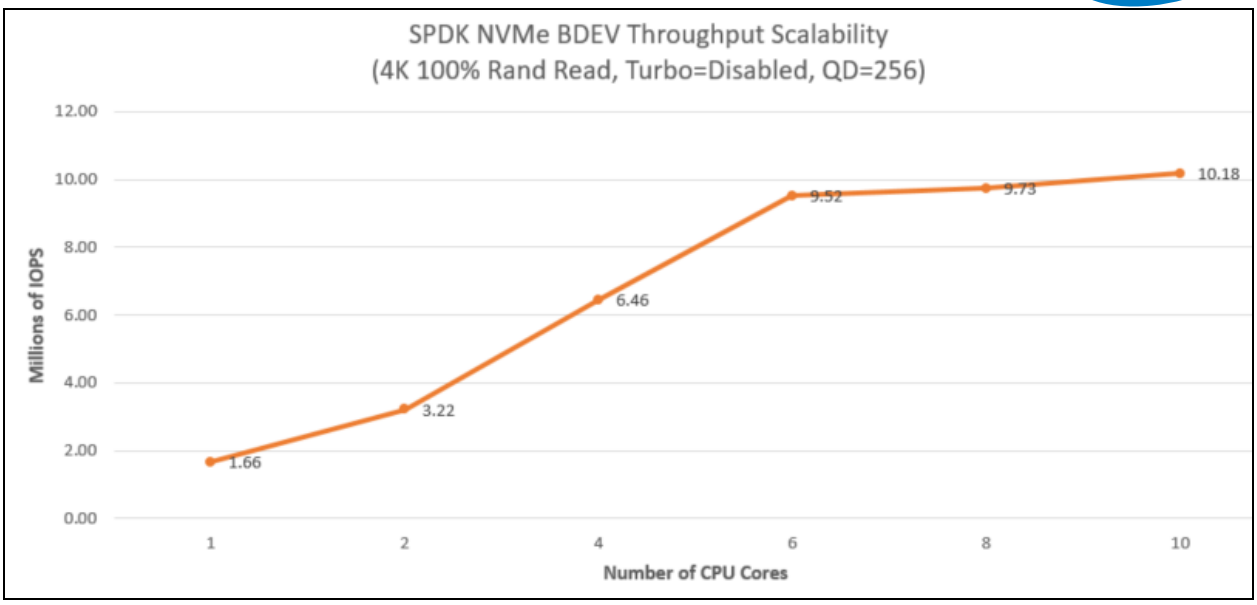

Figure 4: NVMe bdev IOPS scalability with addition of I/O cores (4K 100% Random Read IOPS Turbo=Disabled, QD=256)

The IOPS for the 4K Random Read Workload scaled linearly with addition of I/O cores until the PCIe switches on this system were saturated. Furthermore, as we added NVMe SSDs and I/O processing cores the average latency did not change. The data in Table 5 shows that the NVMe bdev average I/O latency does not deteriorate with the addition of I/O processing cores. Therefore, you once you have established a model for the number of SSDs/core you can scale your system throughput by adding SSDs and I/O processing cores without impacting the average I/O latency.

Table 6 : NVMe bdev I/O core scalability (4K 100% Random Write IOPS, Turbo=Disabled, QD=32)

# of CPU Cores	Throughput (IOPS)	Avg. Latency (usec)
1	1,473,846.80	175.63
2	2,561,888.13	204.25
4	2,680,892.80	282.46

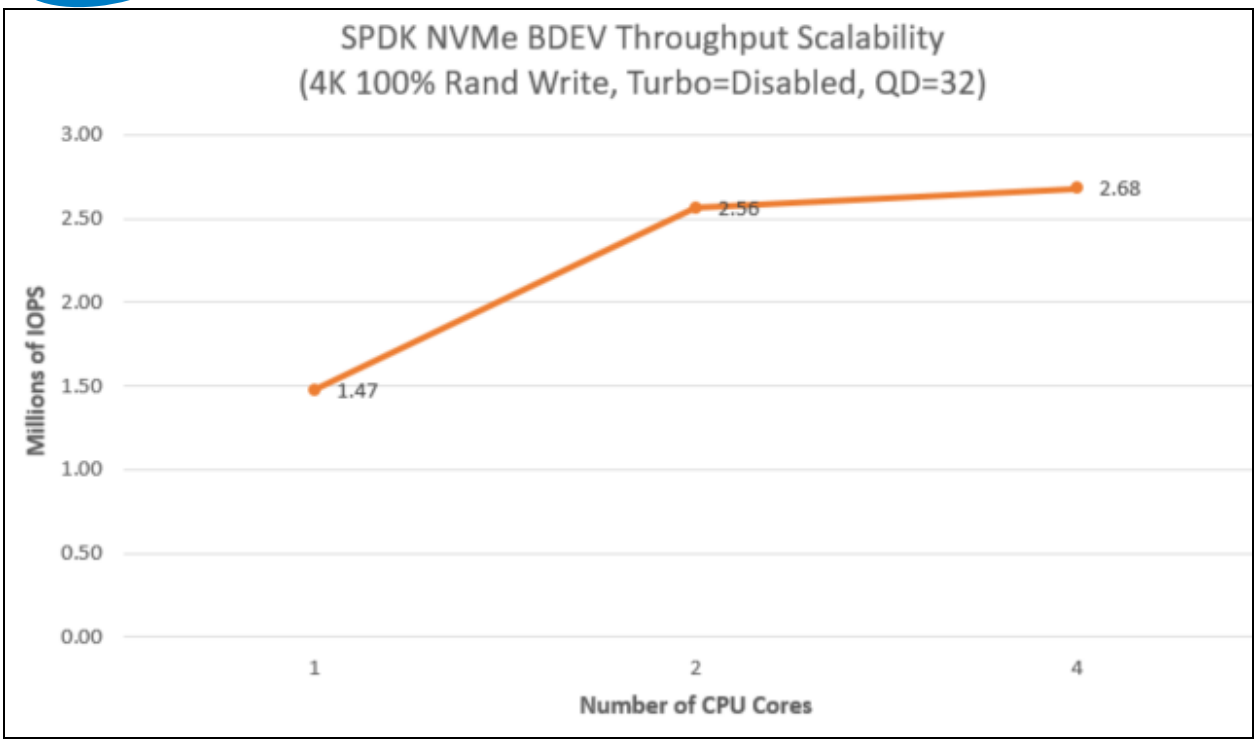

Figure 5 : NVMe bdev I/O core scalability with addition of I/O cores (4K 100% Random Write IOPS, Turbo=Disabled, QD=32)

The IOPS for the 4K Random Write Workload scaled linearly with addition of I/O cores until the 22 NVMe SSDs on this system were saturated.

Table 7 : NVMe bdev I/O core scalability (4K 70/30 Random Read/Write IOPS, Turbo=Disabled, QD=256)

# of CPU Cores	Throughput (IOPS)	Avg. Latency (usec)
1	1,453,333.70	1,680.03
2	2,889,441.20	1,677.09
4	5,598,686.50	1,414.89
6	6,926,862.42	1,330.21
8	7,283,553.50	1,279.13
10	7,029,488.21	1,328.57

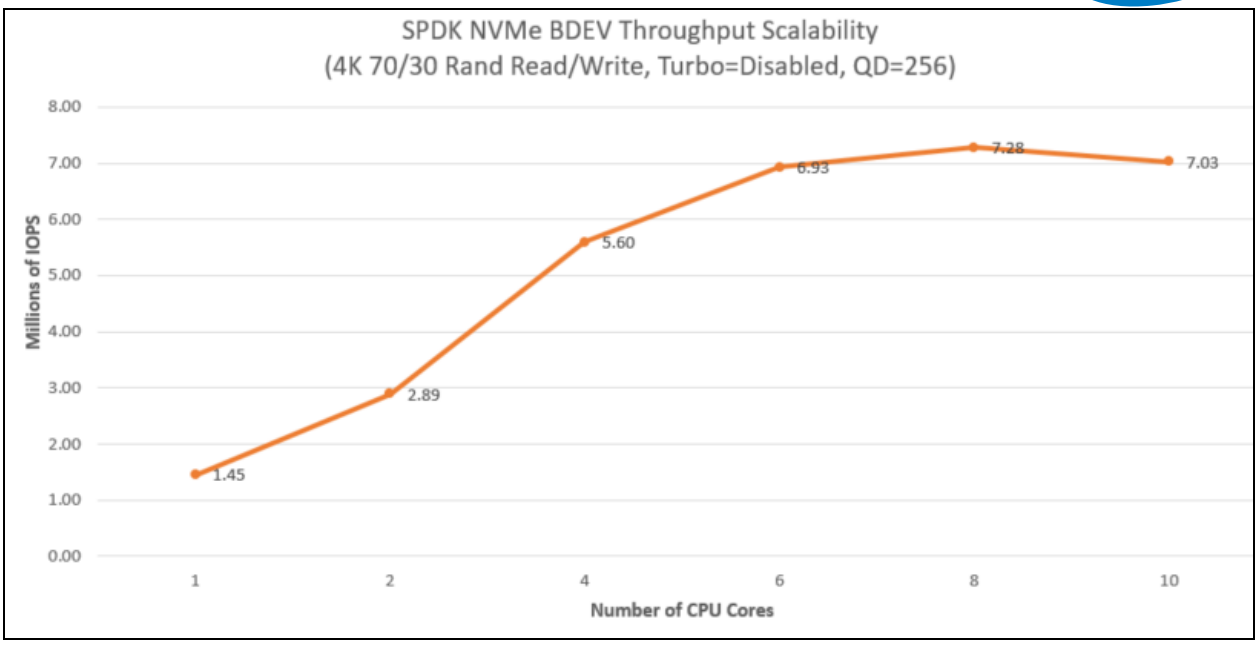

Figure 6 : NVMe bdev IOPS scalability with addition of I/O cores (4K 70/30 Random Read/Write IOPS, Turbo=Disabled, QD=256)

The IOPS for the 4K 70/30 random read/write workload scaled linearly with addition of I/O cores until all the SSDs in the systems were saturated.

Conclusion:

For all the 3 workloads, throughput scales up almost linearly with addition I/O cores until the SSDs or PCIe switches were saturated.

Test Case 3: NVMe BDEV Latency Tests

This test case was carried out to understand latency characteristics while running SPDK NVMe bdev and its comparison to Linux Kernel NVMe block device layer. FIO was ran for 1 hour targeting a single block device over a single NVMe drive. This test compares consistency between latency of the SPDK and Linux Kernel block layers over time in a histogram. The Linux Kernel block layer provides I/O polling capabilities to eliminate overhead such as context switch, IRQ delivery delay and IRQ handler scheduling. This test case includes a comparison of the I/O latency for the kernel I/O polling vs. SPDK.

Test Workloads: We use the following workloads.

- 4KB 100% Random Read
- 4KB 100% Random Write

Item	Description
Test Case	NVMe BDEV Latency
Test configuration	**FIO Configuration:** FIO, 1 hour test on a single NVMe SSD Queue Depth: 1 Block Size: 4096 FIO thread: 1 FIO thread running on CPU 0 **SPDK NVMe Driver Configuration:** *ioengine=spdk_bdev* **Linux Kernel NVMe Driver (Default) Configuration:** *ioengine: libaio* **Linux Kernel Hybrid Polling:** 1 NVMe configured with hybrid polling enabled. To enable hybrid polling: *echo 0 > /sys/block/nvme0n1/queue/io_poll_delay* Setting the io_poll_delay to a value of 0 enables adaptive hybrid polling, where the polling thread sleeps for half the mean device execution time. The latency improves if the device is woken up with enough head room for a context switch. Note: that there actually 2 context switches: 1 to put the polling thread to sleep and 1 to wake up the polling thread after the sleep delay. Hybrid polling reduces the CPU load. The following setting were added to the fio configuration file: *ioengine=pvsync2* *hipri=100* **Linux Kernel Classic Polling:** 1 NVMe configured with the kernel classic polling (100% load on the polling CPU core, no sleep). To enable classic polling: *echo -1 > /sys/block/nvme0n1/queue/io_poll_delay* The following setting were added to the fio configuration file: *ioengine=pvsync2*

	hipri=100
FIO config	SPDK fio configuration file: [global] ioengine=spdk_bdev spdk_conf=/home/john/spdk/scripts/perf/nvme/fio_bdev_configs/fio.spdk_bdev_conf direct=1 time_based=1 norandommap=1 ramp_time=300s runtime=3600s thread=1 group_reporting=1 rw=randrw rwmixread={100,0} bs=4096 iodepth=1 numjobs=1 log_avg_msec=250 write_lat_log=spdk_lat_test_logfile.out [filename1] filename=Nvme1n1 cpumask=0x10000000
	Linux Kernel fio configuration file [global] ioengine=libaio direct=1 time_based=1 norandommap=1 runtime=3600 ramp_time=300s thread=1 rw=${RW} rwmixread=${MIX} bs=4096 iodepth=1 numjobs=1 group_reporting=1 log_avg_msec=250 write_lat_log=kernel_lat_test_logfile.out [filename1] filename=/dev/nvme9n1 cpus_allowed=28

	Linux Kernel fio configuration file for hybrid-polling and classic polling [global] ioengine=pvsync2 hipri=100 direct=1 time_based=1 norandommap=1 runtime=3600s ramp_time=300s thread=1 rw=randrw rwmixread={100,0} bs=4096 iodepth=1 numjobs=1 group_reporting=1 log_avg_msec=250 write_lat_log=kernel_hybrid_polling_test_logfile.out [filename1] filename=/dev/nvme9n1 cpus_allowed=28
Test Execution	We used the following command to Pre-condition the SSDs. ./perf -q 32 -s 131072 -w write -t 1200 Workload Specific Pre-conditioning: For 4K 100% Random Write workload we run the workload for 90 minutes to pre-condition the SSD. ./perf -q 32 -s 4096 -w randwrite -t 5400 For each configuration (SPDK, Linux Kernel, Linux Kernel Hybrid Polling, Linux Kernel Classic Polling) 1) Pre-condition all the SSDs in the system 2) Run the Random Read workload for 1 hour using the fio configuration files above. 3) Pre-condition for 4K Random Writes 4) Run the Random Write workload for 1 hour using the fio configuration files above. Results in the chart represent Latency data at 250ms sample interval over 1 hour

The Linux block layer implements I/O polling on the completion queue. Polling can remove context switch(cs) overhead, IRQ delivery and IRQ handler scheduling overhead[1].

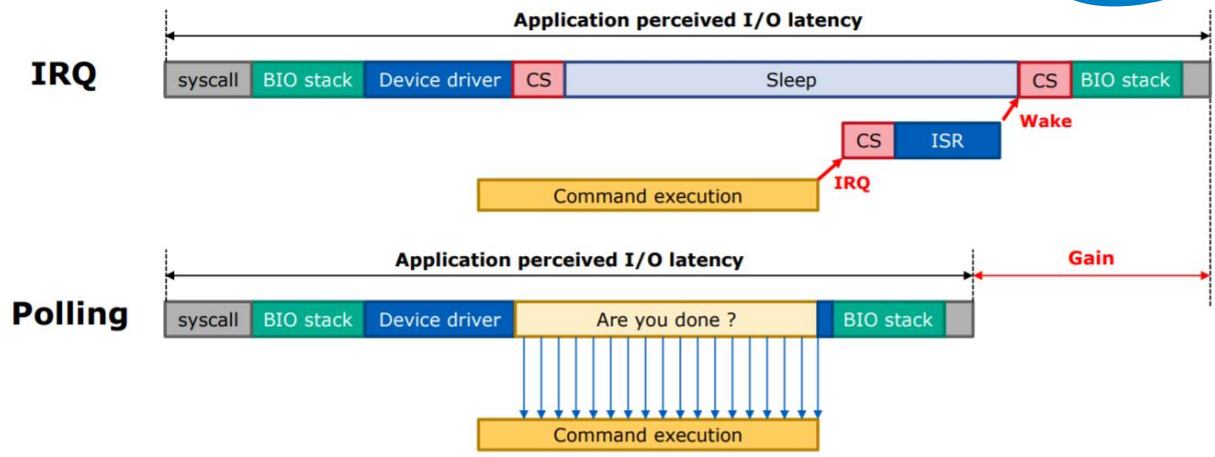

Figure 7 : Linux Block Layer I/O Optimization with Polling. Source [1]

Furthermore, the Linux block I/O polling provides a mechanism to reduce the CPU load. In the *Classic Polling* model the CPU spin-waits for the command completion and utilizes 100% of a CPU core[1]. Adaptive hybrid polling reduces the CPU load by putting the I/O polling thread to sleep for about half of the command execution time, but the polling thread must be woken up before the I/O completes with enough heads-up time for a context switch[1].

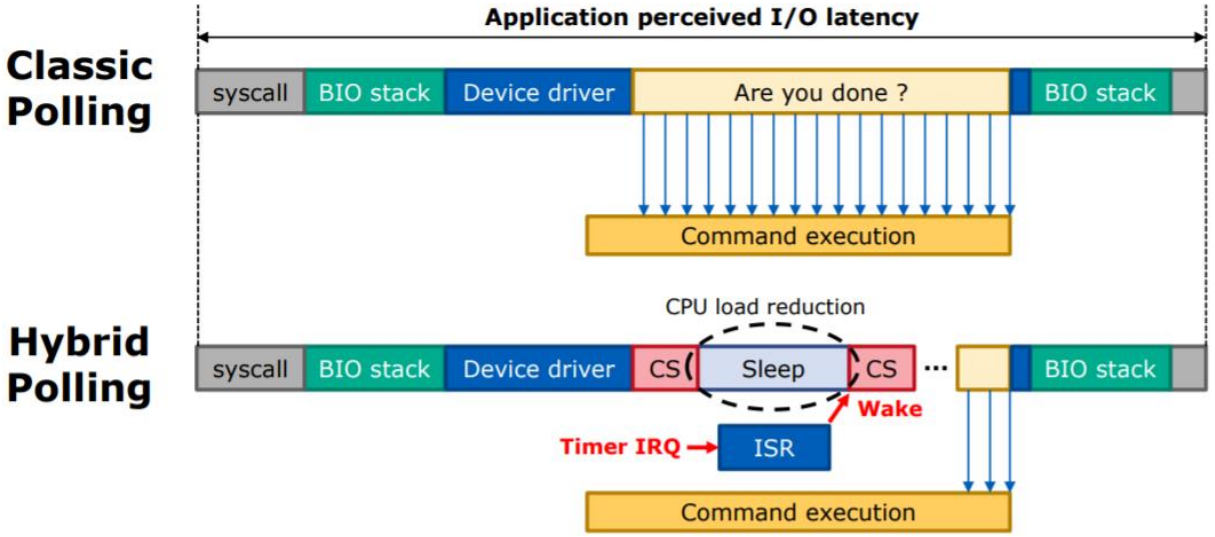

Figure 8 : Linux Block I/O Classic and Hybrid Polling latency breakdown. Source [1]

The data in Table 8 compares the I/O latency for a 4K Random Read performed using the SPDK vs. Linux block layer default I/O model vs. Hybrid and Classic polling I/O models.

Table 8 : SPDK bdev vs. Linux Kernel Polling latency comparison (4K Random Read, QD=1, runtime=1hour, fio)

	SPDK bdev	Linux Kernel (Default)	Linux Kernel (Hybrid Polling)	Linux Kernel (Classic Polling)
Average Latency (usec)	77.04	93.76	85.83	79.37
P90 Latency (usec)	77.88	94.45	86.49	79.93
P99 Latency (usec)	78.89	95.08	87.09	80.44

Average submission latency (usec)	0.16	5.33	No Data	No Data
Average completion latency (usec)	76.88	88.25	85.68	79.34

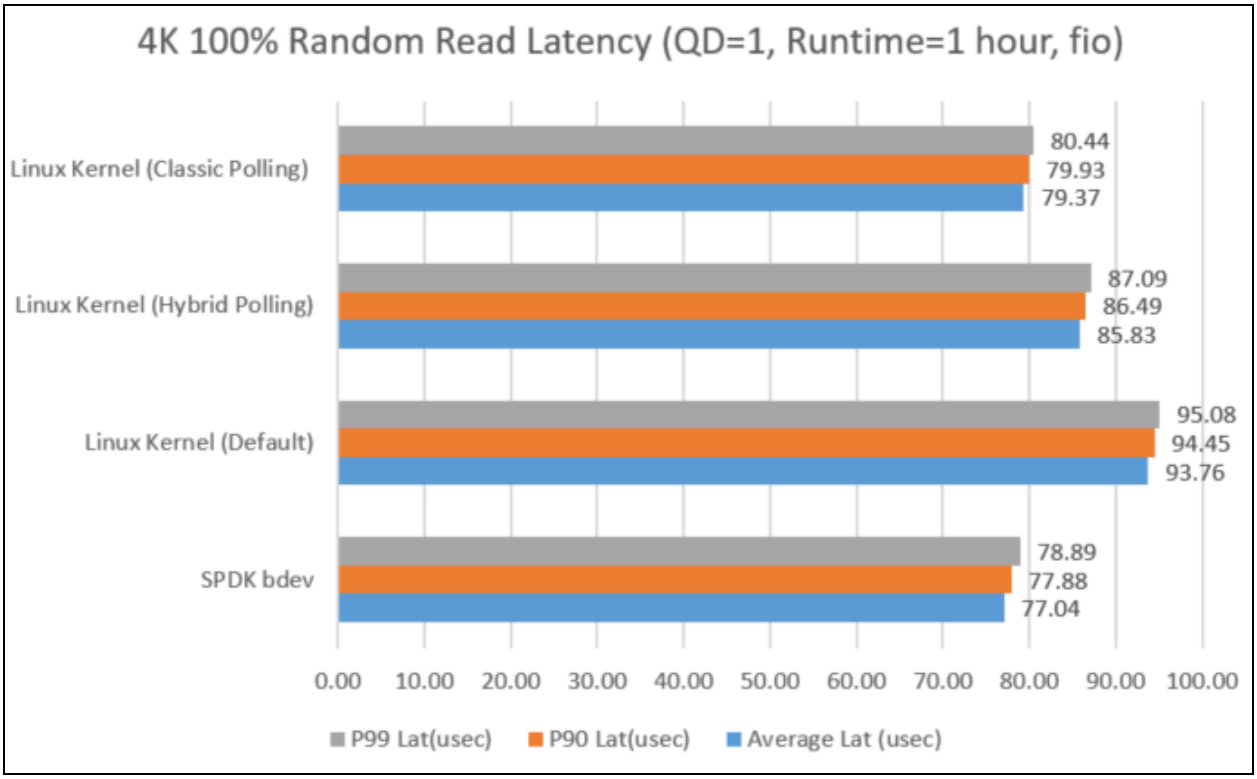

Figure 9 : SPDK bdev vs. Linux Kernel block layer (default, hybrid-polling and classic polling) 4K Random Read latency comparisons

The SPDK NVMe bdev read and write latency was approximately 18% and 35% better than the default Linux Kernel driver latency, respectively. Linux classic polling eliminates most of the Linux Block I/O overhead but the application still has to make a system call to the Linux Kernel block I/O stack as shown in **Figure 8**. The SPDK NVMe bdev latency was approximately 2 microseconds lower than the Linux classic polling latency for reads and approximately 1 microsecond lower for writes because the SPDK bdev eliminates the overhead associated with the system call and block I/O stack.

Table 9 : SPDK bdev vs. Linux Kernel Polling latency comparison (4K Random Write, QD=1, runtime=1hour, fio)

	SPDK bdev	Linux Kernel (Default)	Linux Kernel (Hybrid-Polling)	Linux Kernel (Classic Polling)
Average Latency (usec)	9.88	15.30	11.46	10.84
P90 Latency (usec)	10.47	16.55	12.28	11.42
P99 Latency (usec)	11.15	17.99	13.60	12.00
Average submission latency (usec)	0.19	2.39		
Average completion latency (usec)	9.69	12.82	11.39	10.82

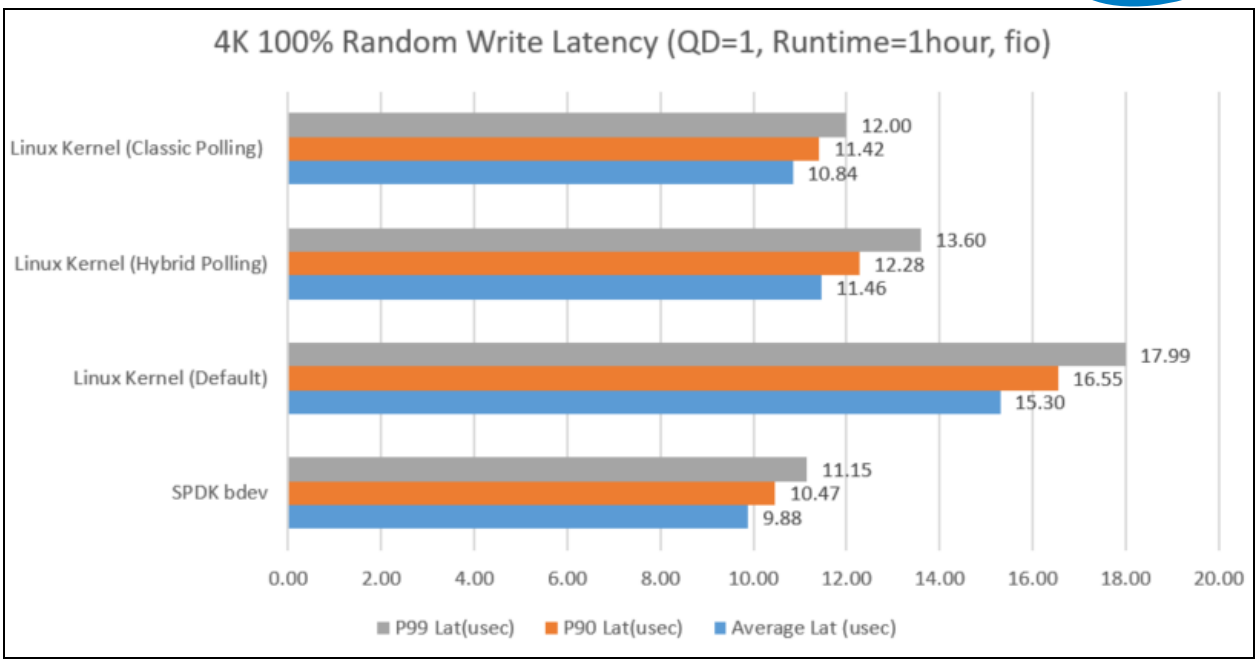

Figure 10 : SPDK bdev vs. Linux Kernel block layer (default, hybrid-polling and classic polling) 4K Random Write latency comparisons

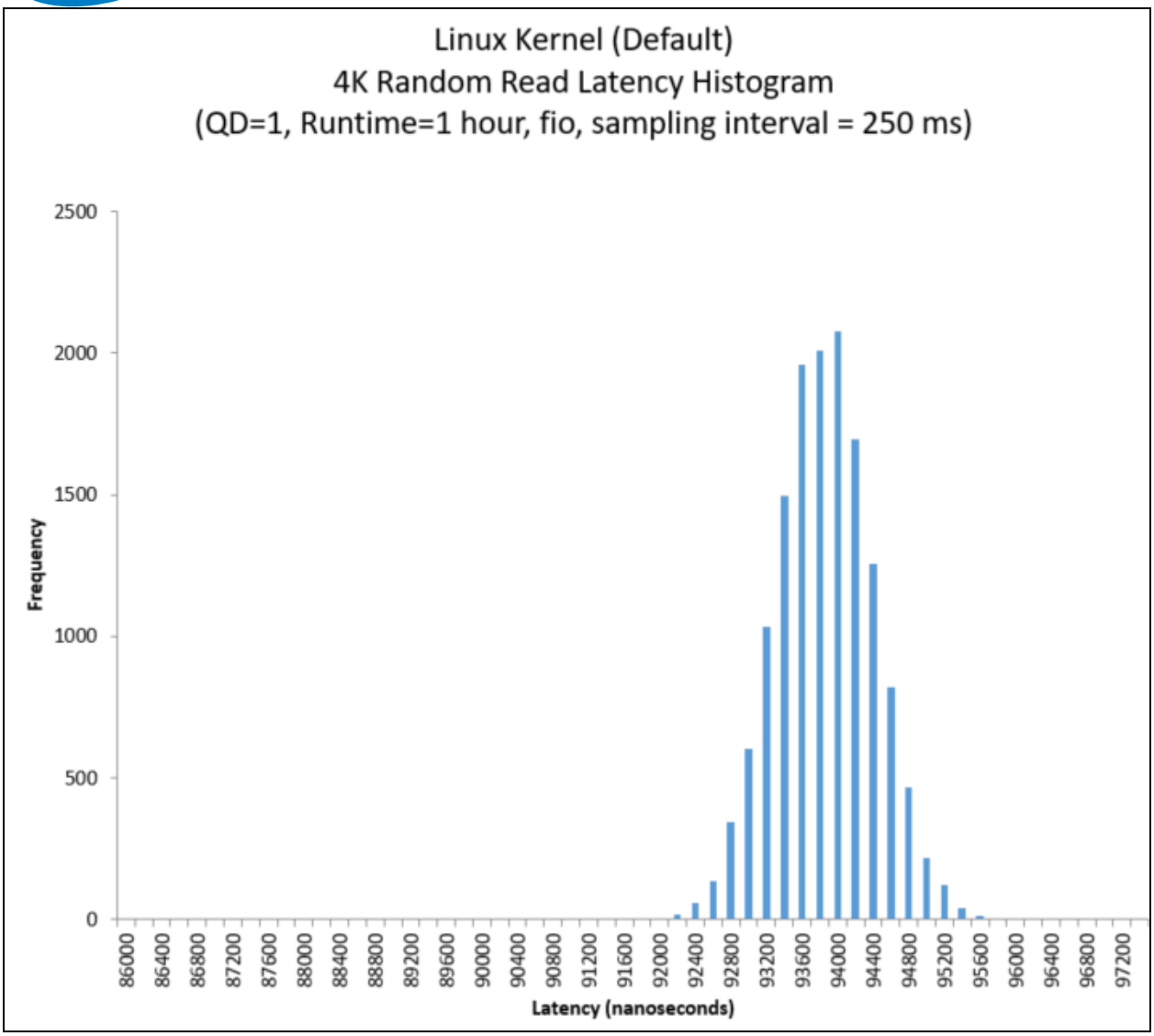

Figure 11 : Linux Kernel (Default) 4K Random Read Latency Histogram (QD=1, Runtime=1 hour, fio, sampling interval = 250 ms)

The 4K random read latency for the default Linux Kernel driver has a normal distribution around the mean of 93,756.38 nanoseconds. The standard deviation is 560.89 nanoseconds and almost all the samples are within 2 microseconds of the mean.

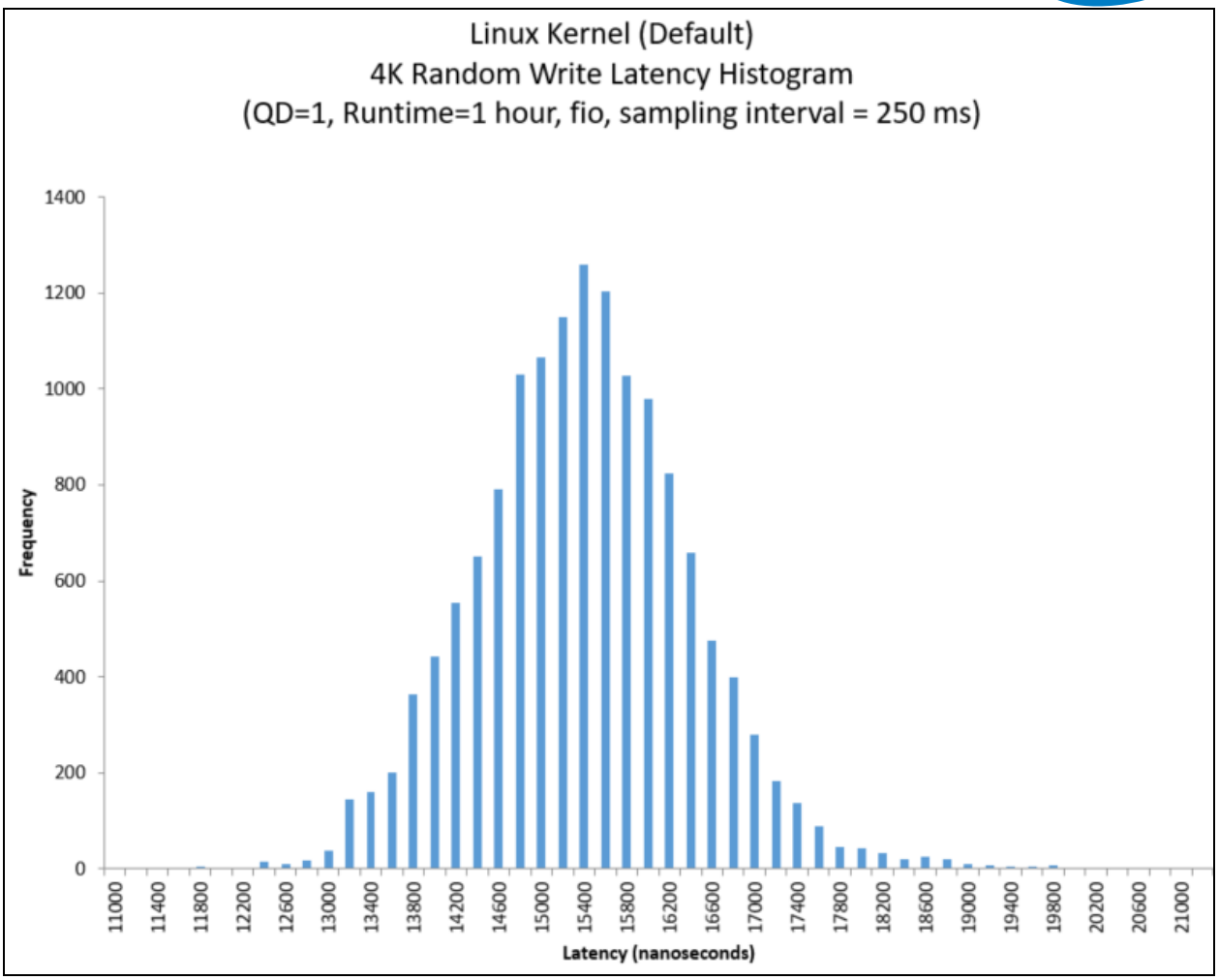

Figure 12 : Linux Kernel (Default) 4K Random Write Latency Histogram (QD=1, Runtime=1 hour, fio, sampling interval = 250 ms)

Figure 12 shows the 4K random write latency for the default Linux Kernel driver has a normal distribution around the mean of 15,299.08 nanoseconds. The standard deviation is 1011.13 nanoseconds and the latency varies from just under 12 microseconds to over 20 microseconds.

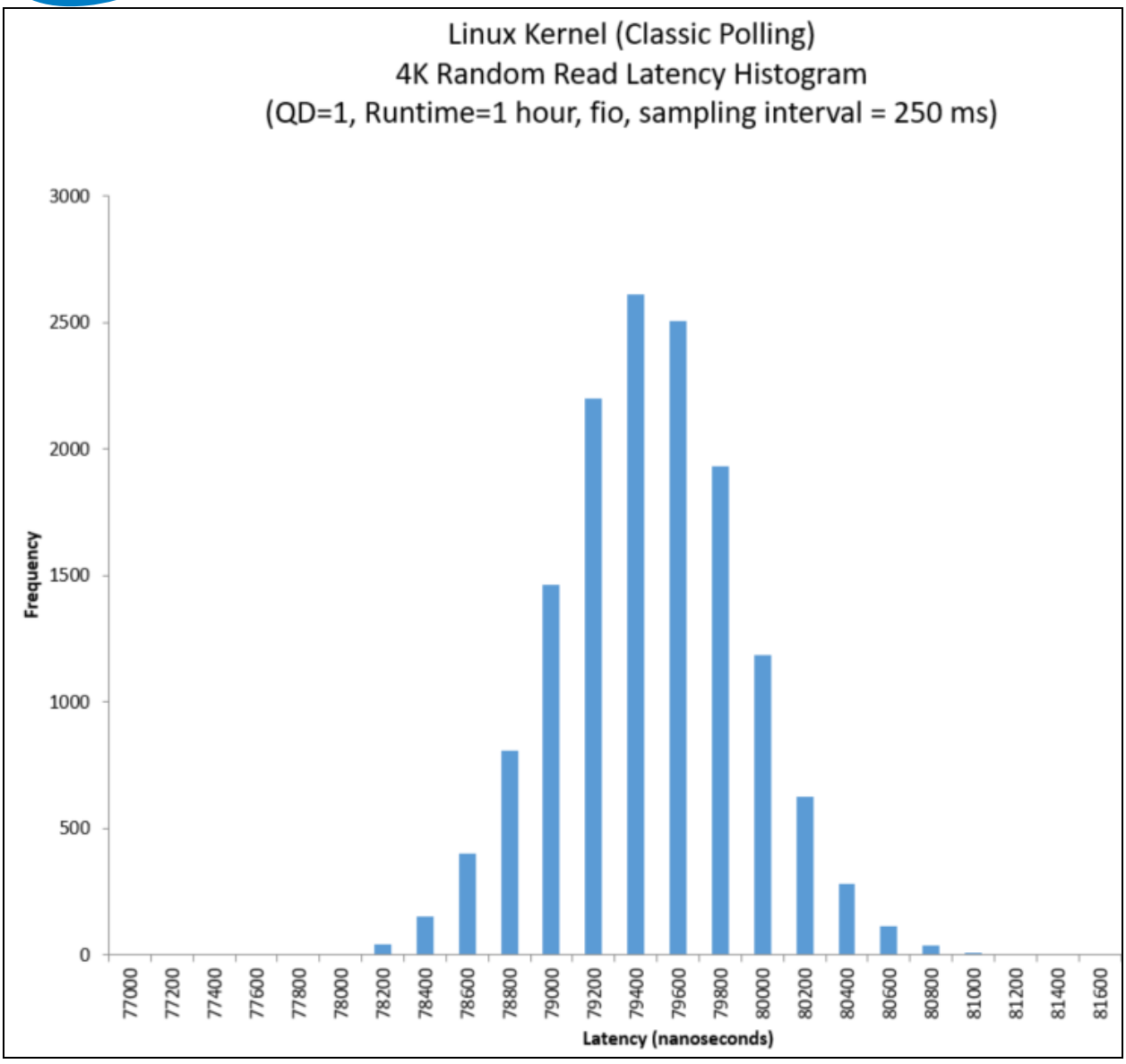

Figure 13 : Linux Kernel (Classic Polling) 4K Random Read Latency Histogram (QD=1, Runtime=1 hour, fio, sampling interval = 250 ms)

Figure 13 shows the 4K random read latency for the Linux Kernel driver in classic polling mode has a normal distribution around the mean of 79,366.27 nanoseconds. The standard deviation is 443.22 nanoseconds and almost all the values are within 2 microseconds of the mean. Approximately 99.9 % of the samples are less than 81 microseconds.

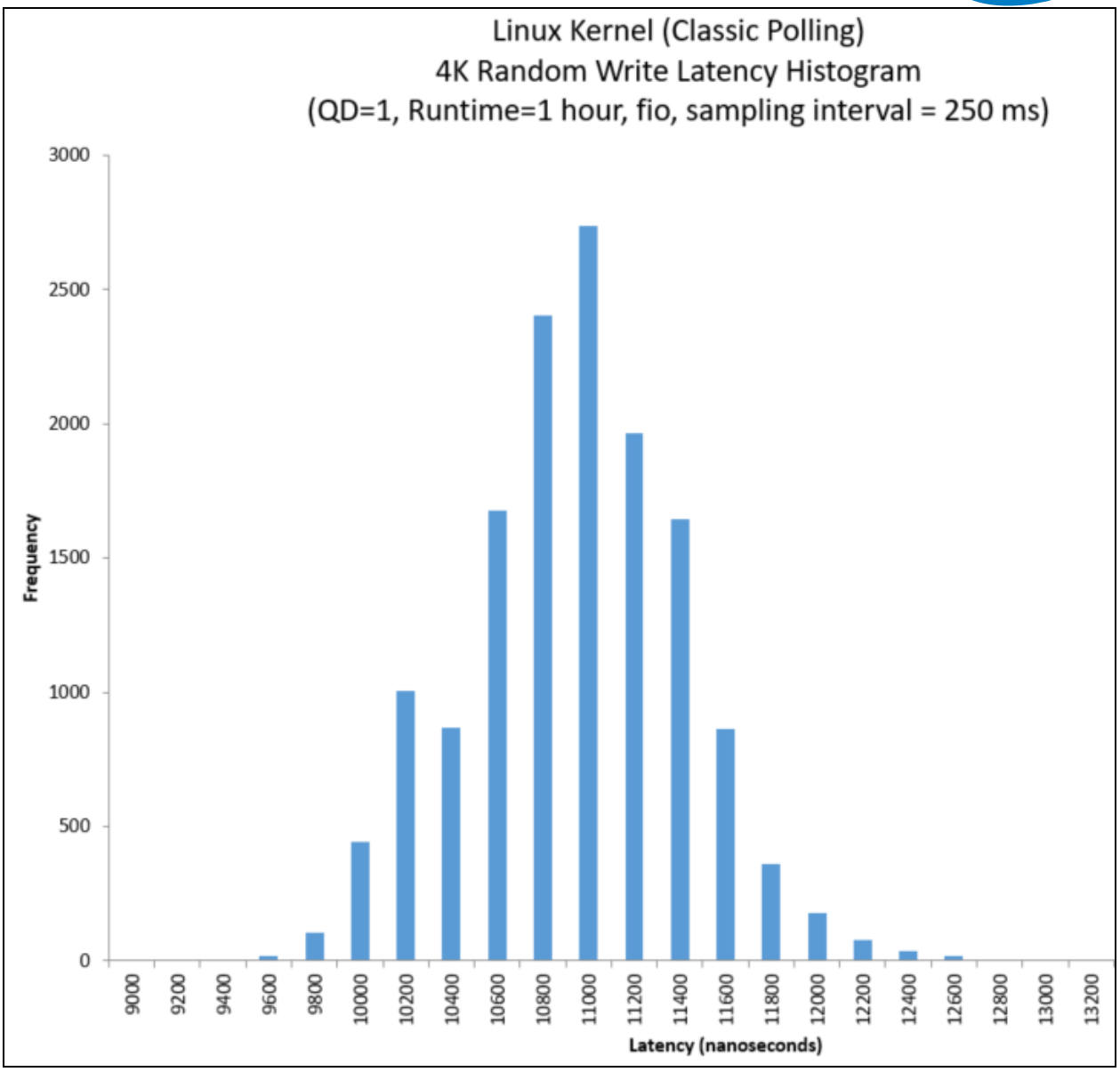

Figure 14 : Linux Kernel (Classic Polling) 4K Random Write Latency Histogram (QD=1, Runtime=1 hour, fio, sampling interval = 250 ms)

Figure 14 shows the 4K random read latency for the Linux Kernel driver in classic polling mode has a bell-shaped distribution around the mean of 10,842.81 nanoseconds. The standard deviation is 470.91 nanoseconds and 99.9% of the samples collected were within 2 microseconds of the mean.

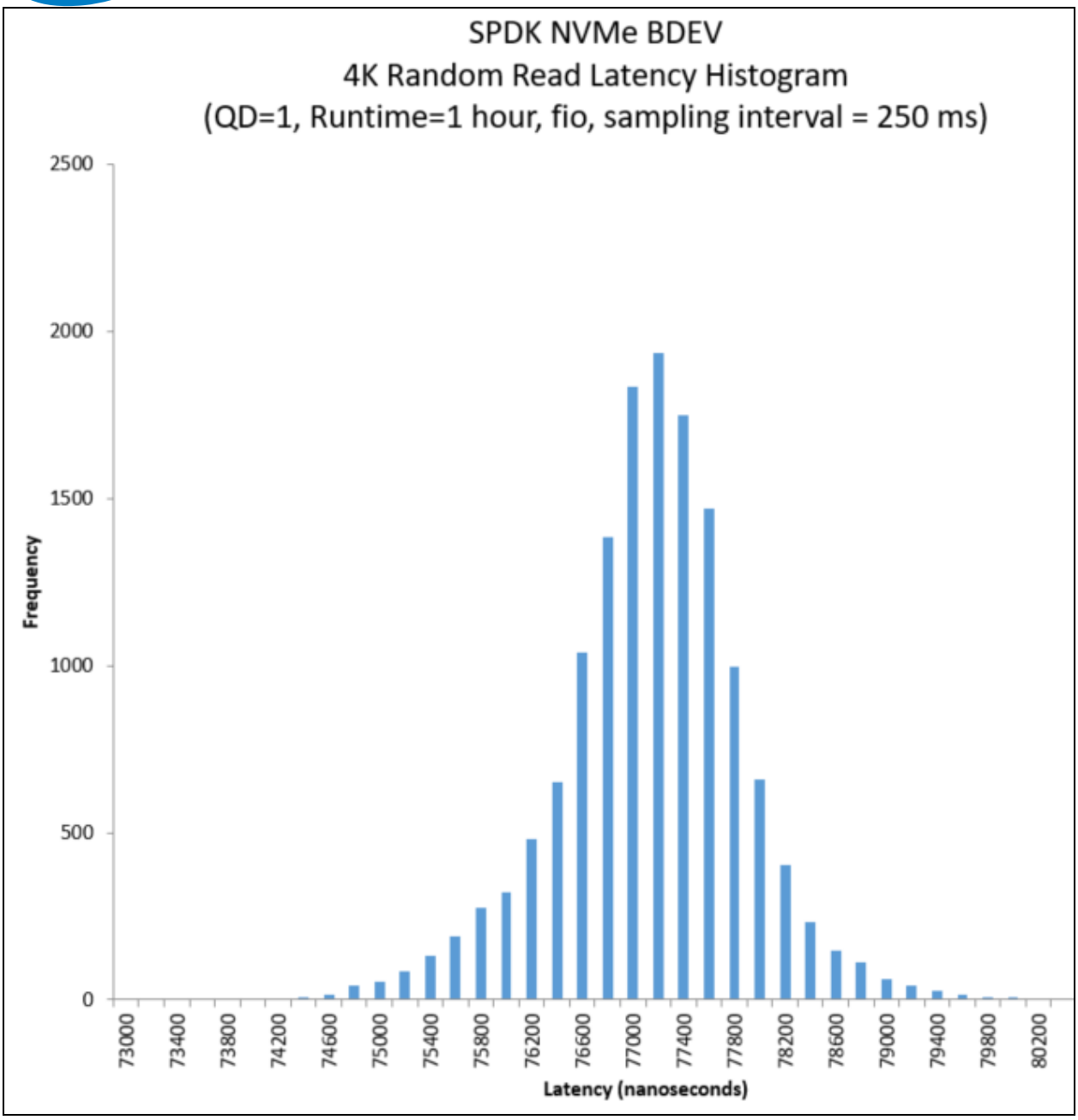

Figure 15 : SPDK 4K Random Read Latency Histogram (QD=1, Runtime=1 hour, fio, sampling interval = 250 ms)

Figure 15 shows the 4K random read latency for the SPDK bdev has a bell-shaped distribution around the mean of 77,041.72 nanoseconds. Over 99.9 of the samples collected are less than 80 microseconds and the standard deviation is 730.77 nanoseconds.

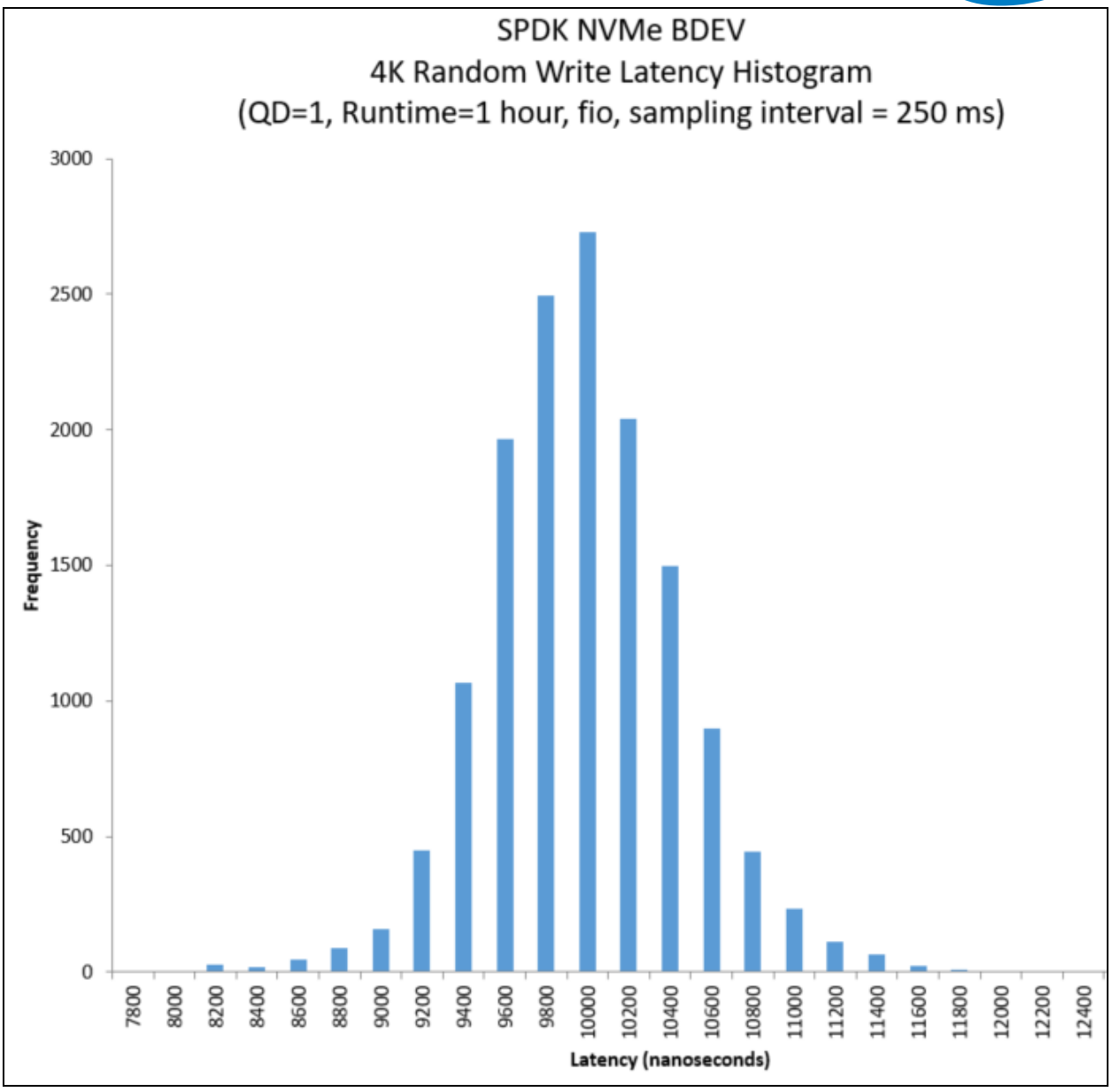

Figure 16 : SPDK 4K Random Write Latency Histogram (QD=1, Runtime=1 hour, fio, sampling interval = 250 ms)

Figure 16 shows the 4K random read latency for the SPDK bdev has a bell-shaped distribution around the mean of 9,884.97 nanoseconds. All 14,400 sample are within 2 microseconds of the mean latency and the standard deviation is 474.53 nanoseconds.

Conclusion:
Polling hardware for completions instead of relying on interrupts, lowers both total latency and latency variance. The SPDK NVMe bdev latency was 1 – 2 microseconds lower than the Linux polling latency because the bdev provides the application direct access to the NVMe SSD in user space eliminating the overhead of making a kernel system calls.

Hybrid Polling and Classic Polling Performance as QD Increases

This test was performed to understand the performance in IOPS and average latency of the Linux NVMe hybrid-polling and classic polling block layer as the queue depth increases by powers of 2 from 1 to 256.

Table 10: IOPS Scalability of SPDK vs. Linux Kernel I/O Polling Block Layers (4 K Random Read, 1 SSD, NumJob=1)

	SPDK		**Linux Kernel (Default)**		**Linux Kernel (Classic Polling)**		**Linux Kernel (Hybrid Polling)**	
QD	IOPS	Ave Lat (usec)	IOPS	Ave Lat (usec)	IOPS	Ave Lat (usec)	IOPS	Ave Lat (usec)
1	12,902.59	77.21	10,671.83	92.69	12,599.62	79.13	11,455.05	86.21
2	25,504.79	78.12	21,183.29	93.42	12,589.28	79.18	11,387.16	86.73
4	49,699.03	80.20	41,737.13	94.94	12,587.62	79.19	11,372.28	86.84
8	94,294.31	84.56	86,696.54	91.81	12,579.79	79.25	11,365.97	86.89
16	170,478.11	93.58	163,378.53	97.65	12,572.02	79.30	11,352.57	86.99
32	283,607.17	112.57	272,521.05	117.14	12,561.16	79.36	11,348.81	87.02
64	414,900.97	153.99	369,559.00	172.85	12,550.24	79.44	11,390.81	86.71
128	524,763.02	243.66	371,683.96	344.05	12,544.35	79.48	11,373.96	86.82
256	568,910.12	449.62	373,306.48	685.43	12,526.55	79.57	11,308.13	87.34

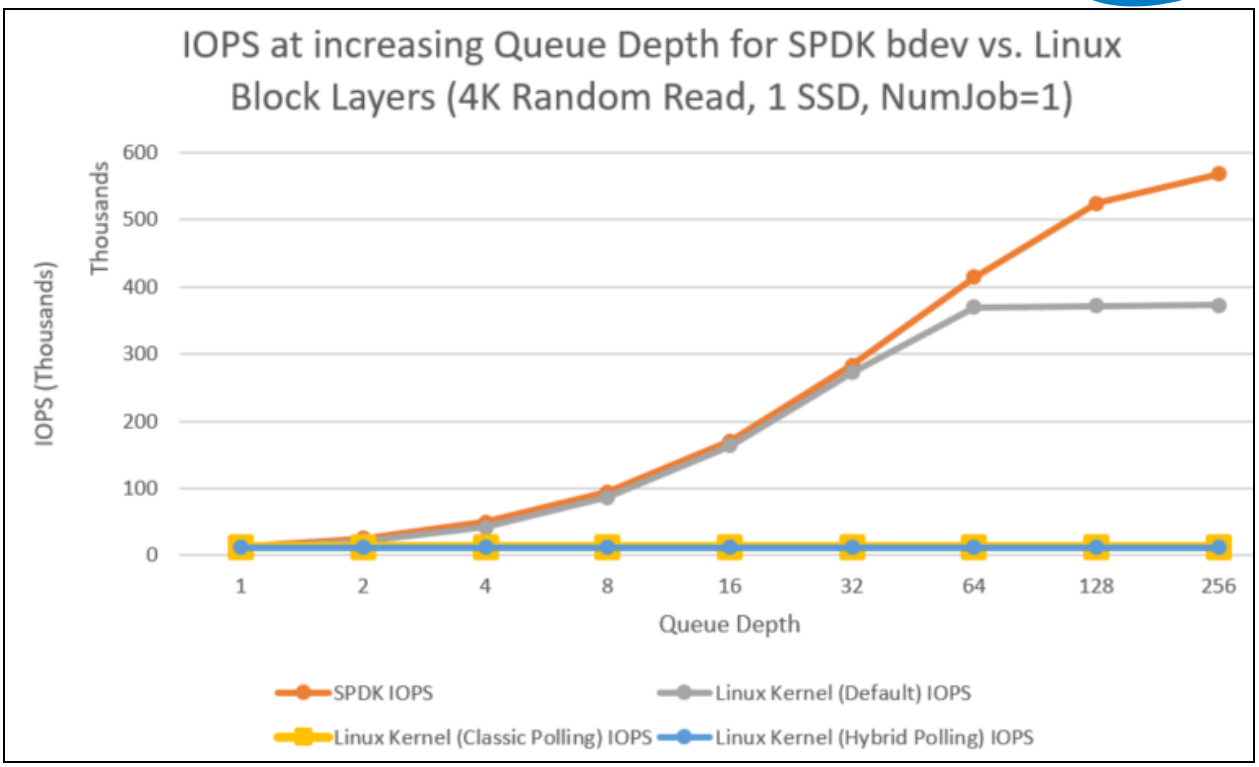

Figure 17 : IOPS at increasing Queue Depth for SPDK bdev vs Linux Block Layer (4K Rand Read, 1 SSD, Numjob=1)

SPDK NVMe bdev IOPS scaled linearly as the queue depth increased until the NVMe SSD was saturated. The default Linux Kernel NVMe driver IOPS increased linearly as the queue depth increased until the CPU core was saturated. Both, hybrid and classic polling IOPS remained constant as the queue depth increased because the Linux Kernel polling implementation performs blocking I/O.

Hybrid Polling and Classic Polling Performance as FIO Threads Increase

The test case was performed to understand the performance in IOPS and average latency of the Linux NVMe driver with hybrid polling enabled as the number of FIO threads increases.

Table 11 : Linux Kernel Hybrid-Polling and Classic Polling performance with increasing FIO threads (4K Rand Read, 1 SSD, Runtime=600 sec, QD=1)

	Linux Kernel (Hybrid Polling)			**Linux Kernel (Classic Polling)**		
NumJobs	IOPS	Ave Lat (usec)	CPU Cores Utilized	IOPS	Ave Lat (usec)	CPU Cores Utilized
1	11,415.69	86.47	0.6	12,527.87	79.58	1.04
2	22,616.94	87.29	1.1	24,790.10	80.44	2.04
4	44,435.14	88.85	2.2	48,331.99	82.51	4.08
8	85,019.05	92.97	4.3	92,074.46	86.63	8.08
16	155,517.51	101.75	8.4	167,145.27	95.47	16.07
32	262,385.12	120.84	17.0	278,709.84	114.55	32.07
64	392,155.41	162.06	35.1	409,723.09	155.89	65.17

128	517,499.88	246.61	68.5	501,251.02	254.88	112.00
256	574,105.55	444.92	99.0	502,234.13	508.64	112.00
512	588,676.51	868.92	105.7	503,924.72	1013.35	112.00

Conclusion:
The data in Table 10 shows that the IOPS for the linux classic and hybrid polling I/O models remained constant when the queue depth was scaled from 2 to 256. Table 11 demonstrates that the Linux Kernel polling implementation requires scaling the number of fio threads to scale the IOPS. Increasing the number of fio threads increases the CPU utilization. The IOPS for the Linux Kernel classic polling I/O model scaled linearly with the number of fio threads until all CPU cores were utilized. The hybrid polling I/O model lowers the CPU load by putting the I/O thread to sleep instead of polling all the time as showing in Figure 8. Hybrid polling lowers the CPU load vs. classic polling so the IOPS for the hybrid polling I/O model scaled linearly with the number of fio threads until the NVMe SSD was saturated.

Test Case 4: IOPS vs. Latency at different queue depths

This test case was performed in order to understand throughput and latency trade-offs with varying queue depth while running SPDK vs. Kernel NVMe block layers.

Results in the table represent performance in IOPS and average latency for the SPDK NVMe driver and Linux Kernel NVMe driver. We limited both the SPDK and Linux NVMe driver to use the same number of CPU Cores.

Test Workloads: We use the following Random Read/Write mixes

- 4KB 100% Random Read
- 4KB 100% Random Write
- 4KB Random 70% Read 30% Write

Item	Description
Test Case	IOPS vs. Latency at different queue depths
Test configuration	Number of Intel P4600x NVMe SSDs: 20 Number of CPU Cores: 4 Queue Depth: 2^n (where n = 0,1,2,3,4...7). Block Size: 4096 SPDK fio configuration file: [global] ioengine=spdk_bdev spdk_conf=/home/john/spdk/scripts/perf/nvme/fio.spdk_bdev_conf direct=1 time_based=1 norandommap=1 ramp_time=60s runtime=600s thread=1 group_reporting=1 percentile_list=50:99:99.9:99.99:99.999 rw=${RW} rwmixread=${MIX} bs=${BLK_SIZE} iodepth=${IODEPTH} numjobs=1 [filename1] filename=Nvme1n1

	filename=Nvme2n1 filename=Nvme3n1 filename=Nvme4n1 filename=Nvme5n1 cpumask=0x10000000 [filename2] filename=Nvme6n1 filename=Nvme7n1 filename=Nvme8n1 filename=Nvme9n1 filename=Nvme11n1 cpumask=0x20000000 [filename3] filename=Nvme14n1 filename=Nvme15n1 filename=Nvme16n1 filename=Nvme20n1 filename=Nvme13n1 cpumask=0x1 [filename4] filename=Nvme17n1 filename=Nvme18n1 filename=Nvme19n1 filename=Nvme21n1 filename=Nvme22n1 cpumask=0x2 Linux Kernel fio configuration file: [global] ioengine=libaio direct=1 time_based=1 norandommap=1 runtime=${RUNTIME} ramp_time=60s rw=${RW} rwmixread=${MIX} bs=${BLK_SIZE} iodepth=${IODEPTH} numjobs=1 group_reporting=1 percentile_list=50:99:99.9:99.99:99.999 [filename1] filename=/dev/nvme0n1 cpus_allowed=0-1 [filename2] filename=/dev/nvme1n1 cpus_allowed=0-1

	[filename3] filename=/dev/nvme2n1 cpus_allowed=0-1 [filename4] filename=/dev/nvme3n1 cpus_allowed=0-1 [filename5] filename=/dev/nvme4n1 cpus_allowed=0-1 [filename6] filename=/dev/nvme5n1 cpus_allowed=0-1 [filename7] filename=/dev/nvme6n1 cpus_allowed=0-1 [filename8] filename=/dev/nvme7n1 cpus_allowed=0-1 [filename9] filename=/dev/nvme8n1 cpus_allowed=0-1 [filename10] filename=/dev/nvme9n1 cpus_allowed=28-29 [filename11] filename=/dev/nvme10n1 cpus_allowed=28-29 [filename12] filename=/dev/nvme11n1 cpus_allowed=28-29 [filename13] filename=/dev/nvme12n1 cpus_allowed=28-29 [filename14] filename=/dev/nvme13n1 cpus_allowed=28-29 [filename15] filename=/dev/nvme14n1 cpus_allowed=28-29 [filename16] filename=/dev/nvme15n1

	cpus_allowed=28-29 [filename17] filename=/dev/nvme16n1 cpus_allowed=28-29 [filename18] filename=/dev/nvme17n1 cpus_allowed=28-29 [filename19] filename=/dev/nvme18n1 cpus_allowed=28-29 [filename20] filename=/dev/nvme19n1 cpus_allowed=28-29

Table 12 : Linux vs. SPDK Block Layer IOPS vs Latency with increasing Queue Depth (4K 100% Random Read on 4 CPU cores – 2 on each NUMA Node)

QD	SPDK (fio+bdev) IOPS	Kernel IOPS	SPDK (fio+bdev) Latency (usec)	Kernel Latency (usec)	CPU Cores utilized by SPDK	CPU Cores utilized by Kernel
1	257,674.35	215,555.14	77.36	91.96	4	4
2	509,574.07	421,046.41	78.25	94.33	4	4
4	993,729.57	754,316.00	80.25	105.52	4	4
8	1,887,372.95	1,118,949.33	84.50	142.58	4	4
16	3,408,437.57	1,232,035.11	93.52	259.35	4	4
32	5,642,317.70	1,304,691.71	112.64	490.17	4	4
64	7,962,416.43	1,423,228.61	154.36	898.39	4	4
128	8,464,001.81	1,418,177.65	260.03	1803.31	4	4
256	8,161,853.84	1,417,872.13	584.19	3609.18	4	4

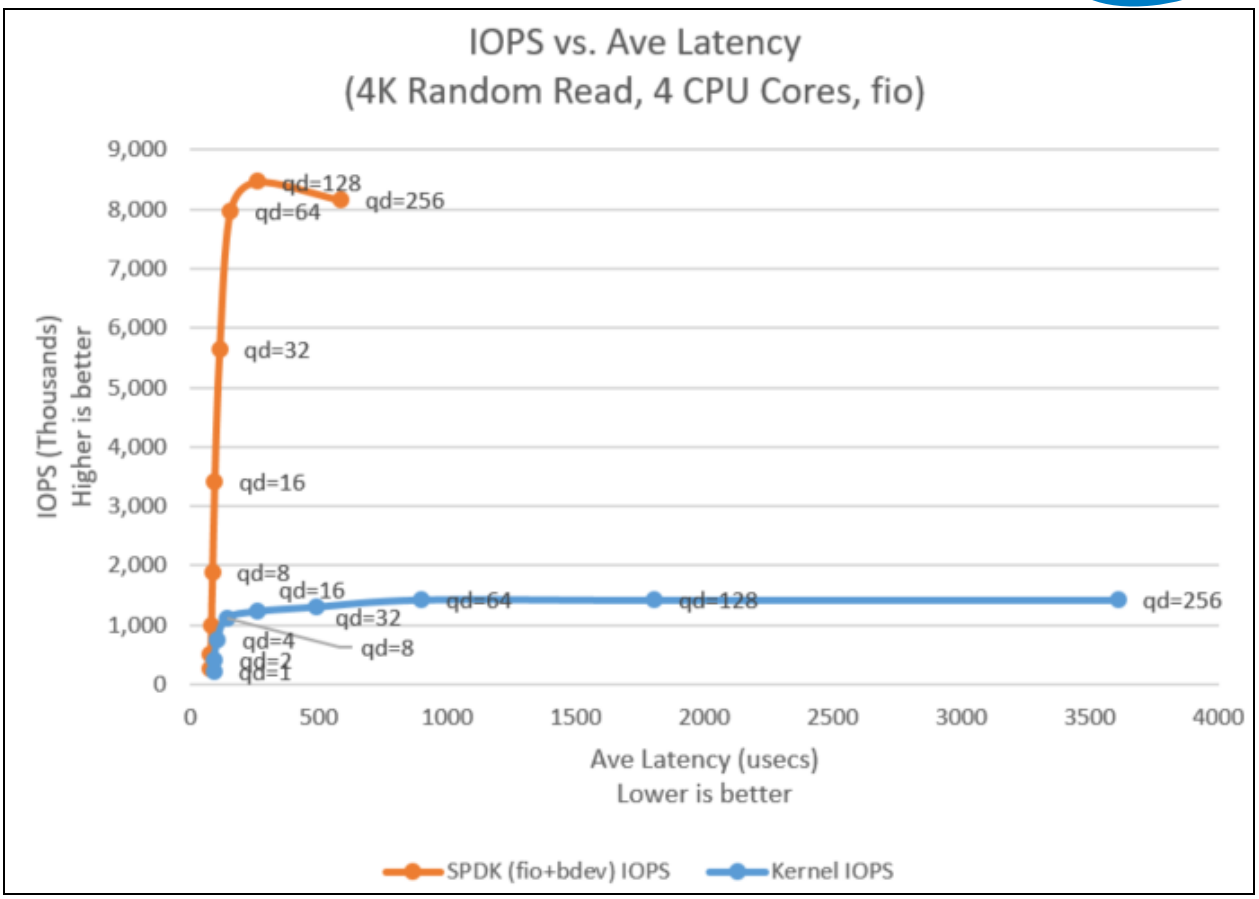

Figure 18 : Linux vs. SPDK Block Layer IOPS vs Latency with increasing Queue Depth (fio, 4K 100% Random Read on 4 CPU cores – 2 on each NUMA Node)

For the 4K 100% Random Read workload, both the SPDK and Linux Kernel NVMe block drivers IOPS scale linearly as the queue depth increases until all 4 CPU cores are saturated. SPDK thoughput in IOPS/Core was almost 6x higher than the Kernel throughput with fio. SPDK latency is also consistently lower than Linux Kernel latency; SPDK latency was 20 - 30 % lower before the Kernel block driver saturated all the 4 CPU cores (QD=4 and lower, at QD>8 the Kernel has saturated all 4 CPU cores so latency just goes up).

Table 13 : Linux vs. SPDK Block Layer IOPS vs Latency with increasing Queue Depth (4K 100% Random Write on 4 CPU cores – 2 on each NUMA Node)

QD	SPDK (fio+bdev) IOPS	Kernel IOPS	SPDK (fio+bdev) Latency (usec)	Kernel Latency (usec)	CPU Cores utilized by SPDK	CPU Cores utilized by Kernel
1	1,875,143.22	603,077.19	10.39	32.63	4	4
2	2,502,239.71	808,002.88	15.61	49.05	4	4
4	2,669,468.75	1,160,640.09	29.36	68.55	4	4
8	2,605,399.72	1,312,885.12	60.28	121.49	4	4
16	2,779,492.05	1,393,921.23	113.22	228.97	4	4
32	2,727,893.68	1,386,689.53	232.61	460.28	4	4
64	2,861,174.73	1,381,787.73	443.21	924.93	4	4
128	2,853,758.52	1,353,952.01	883.83	1,889.18	4	4
256	2,812,828.44	1,322,481.04	1,802.83	3,869.69	4	4

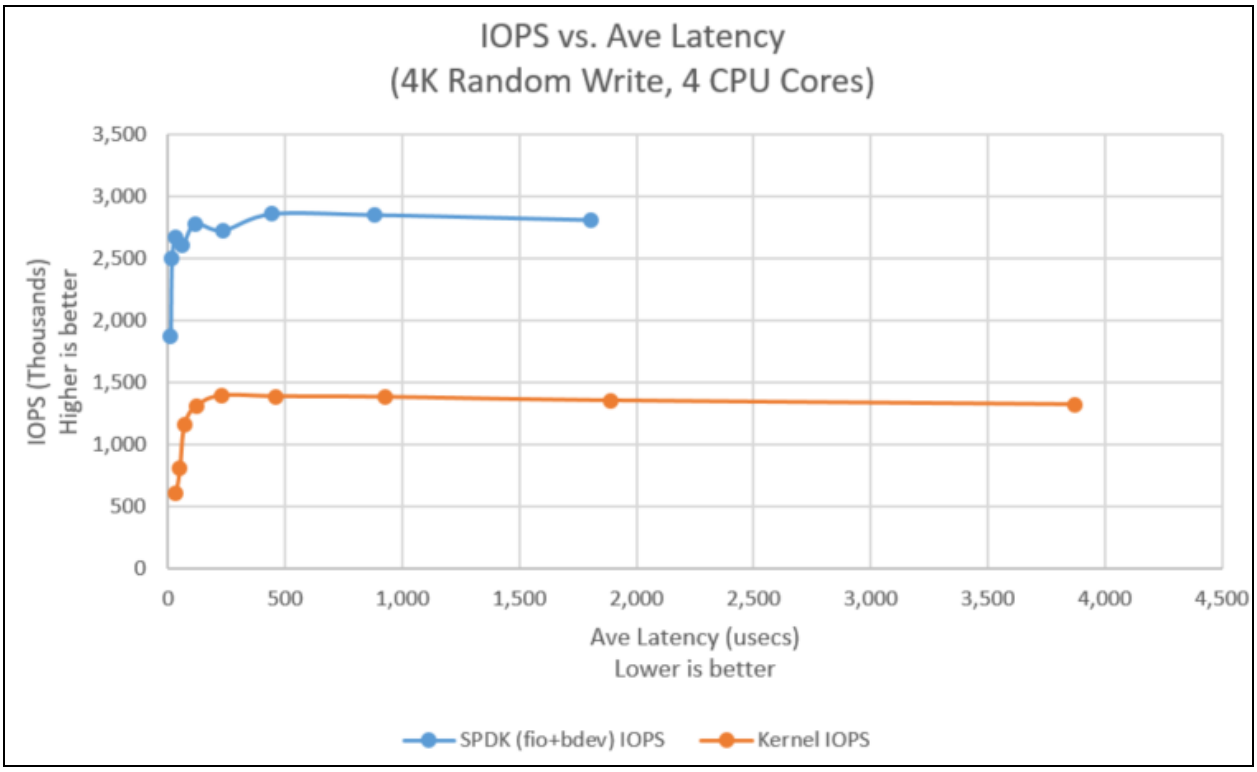

Figure 19 : Linux vs. SPDK Block Layer IOPS vs Latency with increasing Queue Depth (fio, 4K 100% Random Write on 4 CPU cores – 2 on each NUMA Node)

Table 14 : Linux vs. SPDK Block Layer IOPS vs Latency with increasing Queue Depth (4K 70/30 Random Read/Write on 4 CPU cores – 2 on each NUMA Node)

QD	SPDK (fio+bdev) IOPS	Kernel IOPS	SPDK (fio+bdev) Latency (usec)	Kernel Latency (usec)	CPU Cores utilized by SPDK	CPU Cores utilized by Kernel
1	326,841.93	276,850.87	90.61	111.86	4	4
2	579,842.34	486,407.70	101.87	128.99	4	4
4	956,895.62	776,900.35	122.81	167.35	4	4
8	1,480,433.40	1,074,049.83	157.92	241.72	4	4
16	2,205,392.97	1,221,378.39	211.01	411.83	4	4
32	3,147,165.82	1,294,615.28	294.56	759.96	4	4
64	4,450,058.16	1,354,966.42	415.51	1,429.12	4	4
128	5,744,499.44	1,383,657.15	659.56	2,828.88	4	4
256	6,254,851.38	1,374,669.73	1,347.51	6,223.11	4	4

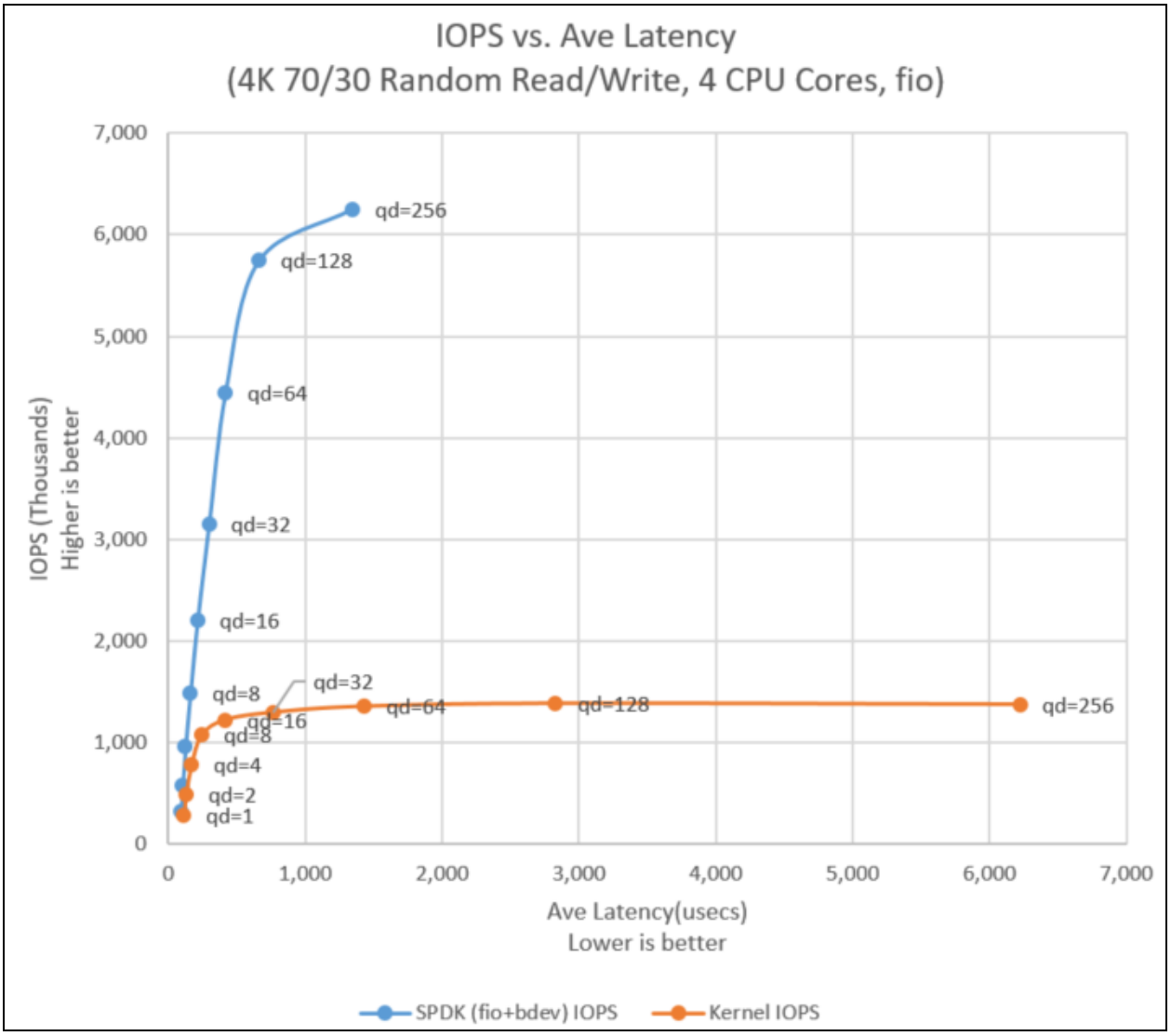

Figure 20 : Linux vs. SPDK Block Layer IOPS vs Latency with increasing Queue Depth (fio, 4K 70/30 Random Read/Write on 4 CPU cores – 2 on each NUMA Node)

For the 4K 70/30 Random Read/Write workload, both the SPDK and Linux Kernel NVMe block drivers IOPS scale linearly as the queue depth increases. The Linux Kernel block driver saturated all 4 CPU cores at about 1.38 million IOPS but SPDK did not saturate the 4 CPU cores. The data in Table 2 shows that SPDK can deliver up to 1.97 million IOPS/core for this workload. Therefore, SPDK throughput in IOPS/Core was almost 5.7x higher than the Kernel throughput with fio for this workload. SPDK latency is also consistently lower than Linux Kernel latency: SPDK latency was 23 - 36 % lower before the Kernel block driver saturated all the 4 CPU cores (QD=4 and lower, at QD>8 the Kernel has saturated all 4 CPU cores so latency just goes up).

Conclusion:

The test results from this test case demonstrate that the SPDK NVMe block driver can achieve over 2.1 million IOPS/core using a popular benchmarking tool like fio. The SPDK NVMe block driver throughput in IOPS/Core was almost 6x higher and latency at least 20% lower that the default Linux Kernel NVMe block driver.

Test Case 4: Platform Performance

Purpose: This test case was performed in order to understand maximum bandwidth of underlying platform while running I/O using SPDK NVMe bdev.

Test Workloads: The test was performed using the following Sequential Read/Write mixes

- 128KB 100% Sequential Read
- 128KB 100% Sequential Write
- 128KB Sequential 70% Read 30% Write

Item	Description
Test Case	Maximum Bandwidth Performance
Test configuration	Number of Intel P4600x NVMe SSDs: 22 Number of CPU Cores: 1,2, 4, 6 and 8 Queue Depth: 16 Block Size:128K

Table 15 : SPDK NVMe bandwidth with increasing number of CPU Cores (128K sequential access, fio, bdev, QD=16)

128K Seq, SPDK bdev, fio, QD=16			
Number of CPU Cores	100% Seq Read (KiB/s)	100% Seq Write (KiB/s)	70/30 Seq Read/Write (KiB/s)
1	17,445,235.00	13,288,008.00	23,034,698
2	37,187,878.00	26,585,813.00	47,045,700
4	38,328,746.00	30,029,329.00	52,915,455
6	40,087,552.00	31,064,733.00	53,971,425
8	40,086,780.00	31,503,810.00	53,954,310
10	42,417,195.00	30,317,015.00	54,197,324

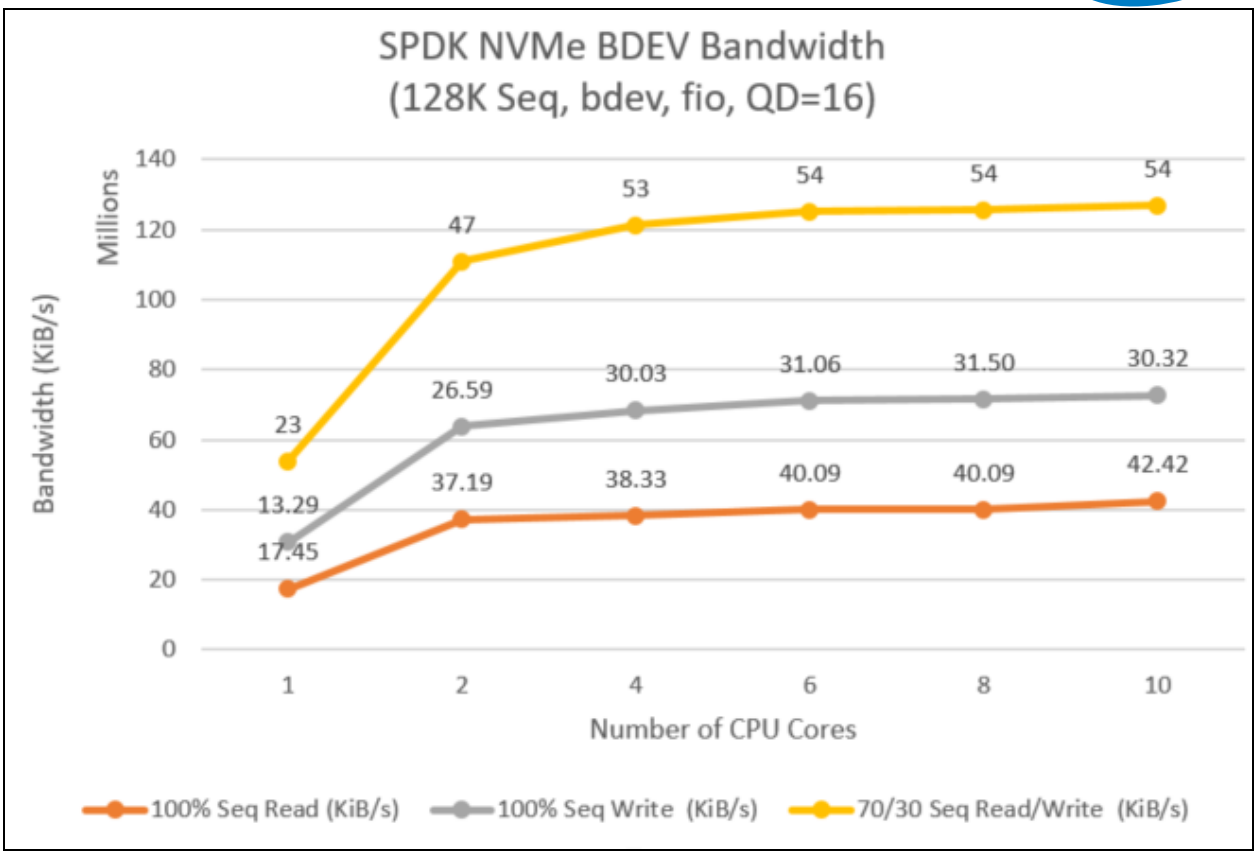

Figure 21 : SPDK NVMe bandwidth with increasing number of CPU Cores (128K sequential access, fio, bdev, QD=16)

Conclusion:

SPDK NVMe bdev throughput in KiB/s scales linearly with addition of CPU cores. SPDK NVMe bdev use just four Intel® Xeon® Platinum 8180 Processor CPU Cores to saturate the platform at 53 GiB/s.

Summary

1. Using fio, the SPDK NVMe bdev delivers up to 2.2 million IOPS on a single Intel® Xeon® Platinum 8180 Processor CPU Core (with turbo boost enabled). That is almost 6x more IOPS/Core vs the Linux Kernel NVMe driver.
    - The SPDK IOPS/Core are significantly impacted by the overhead of the fio benchmarking tool. Using the highly optimized SPDK bdevperf benchmarking tool instead of fio we demonstrated over 3.6 million IOPS on a single CPU core.
2. The SPDK NVMe bdev IOPS scale linearly with addition of CPU cores. We demonstrated 8.5 million IOPS on just 4 CPU Cores (Intel® Xeon® Platinum 8180 Processor with turbo boost enabled).
3. The SPDK NVMe bdev has lower QD=1 latency than the Linux Kernel NVMe block driver for small blocks (4K)
    - SPDK NVMe bdev latency was 21% lower than the default Linux Kernel driver latency.
    - SPDK NVMe bdev latency was 11% lower than the Linux Kernel hybrid polling latency.
    - SPDK NVMe bdev latency was about 1% lower than the Kernel classic polling latency.
    - The Linux Kernel polling implementation does blocking I/O so the IOPS remain constant as the queue depth increases. Scaling IOPS required scaling the number of fio threads which resulted in a much higher CPU load vs. SPDK; hybrid polling used 99 CPU cores to saturate a single Intel P4600 SSD while classic polling saturated all 112 CPU cores before the SSD.
4. The SPDK NVMe bdev throughput in KiB/s scale linearly with addition of CPU cores. It took just 4 CPU cores (Intel® Xeon® Platinum 8180 Processor with turbo boost enabled) to saturate the platform at 53 GiB/s

Performance results are based on testing as of 07/06/2018 and may not reflect all publicly available security updates. See configuration disclosure for details. No product can be absolutely secure.

5.

Table of Figures

References

[1] Damien Le Moal, "I/O Latency Optimization with Polling", Vault – Linux Storage and Filesystem Conference – 2017, May 22nd, 2017.

DISCLAIMERS

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

You may not use or facilitate the use of this document in connection with any infringement or other legal analysis concerning Intel products described herein.

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.

For more information go to http://www.intel.com/performance

Intel® AES-NI requires a computer system with an AES-NI enabled processor, as well as non-Intel software to execute the instructions in the correct sequence. AES-NI is available on select Intel® processors. For availability, consult your reseller or system manufacturer. **For more information, see http://software.intel.com/en-us/articles/intel-advanced-encryption-standard-instructions-aes-ni/**

Intel and the Intel logo are trademarks of Intel Corporation in the US and other countries