Rev. 1.1 1 Nov. 2018

Samsung uNVMe 2.0 SDK Evaluation Guide

Revision History

Revision No.	History	Draft Date	Remark
Rev. 0.1	Initial draft	2014.03.30	
Rev. 0.2	Add Fio-plugin / NUMA-aware setting	2018.04.14	
Rev. 0.3	Add exist command / iterate with delete / keyspace option	2018.06.19	
Rev. 1.0	Add mkfs / fuse / unvme_rocksdb	2018. 07. 03	
Rev. 1.1	Updates for General KV SSD features	2018. 11. 01	

Contents

1 DEVICE SUPPORT INFORMATION

This document describes the means to evaluate NVMe SSD with uNVMe SDK

Samsung invites, and looks forward to, future customer discussions that explore potential uNVMe implementation.

1.1 Supported Devices

Guide Version	Supported Product(s)	Interface(s)
uNVMe2.0 SDK Evaluation Guide ver 1.1	NVMe SSD (LBA/KV)	NVMe 1.2

2 TERMINOLOGY

2.1 Acronyms and Definitions

Acronym/Term	Description
uNVMe	User Level NVMe
SDK	Software Development Kit
UDD	User-level Device Diver (cf. Kernel Device Driver)
KV SSD	Key Value SSD
LBA SSD	Legacy SSD using Logical Block Address space

2.2 Feature option

[DEFAULT]: a default value or selection if not specified explicitly

[OPTION]: a feature marked as OPTION is optional and vendor-specific

[NOTE]: precautions or matters that require attention

(TBD): to be developed (not supported yet)

3 INTRODUCTION

This document describes uNVMe SDK that support Host-side SW Stack based user level device driver for both of KV and LBA SSD. KV SSD is a brand-new SSD storage device that is able to handle IO with native *key-value* interfaces. Document provides information about the SDK and guidance regarding how to build and use SDK and Tools provided. To make your own application, please refer to *Programming Guide* document.

The library routines this document defines allow users to create and use SSD objects, or key-value pairs, while permitting code portability. The library:

- Extends the C language with host and device SDK

Library routines and environment variables provide functionality to control KV SSD's behavior.

[NOTE] This document is being updated. Until finalized, the SDK and APIs syntax and semantics may change without notice.

3.1 Scope

This document covers uNVMe SDK and their semantics. It does not discuss specific protocol details such as ATA, SCSI, and NVMe, and the API's internal device implementation. For more NVME information, please refer to NVMe and KV NVMe specification

3.2 Assumption

This guide has several assumptions.

1. Both host and device use *little endian* memory and transport format. If a host uses big endian byte ordering (e.g., POWER architecture), the host needs to convert it to a little endian format.

4 DIRECTORY

uNVMe SDK is composed of a series of source code, headers, test applications, and scripts to help you become familiar with uNVMe SDK easily and quickly. The uNVMe directory of SDK is composed as follows:

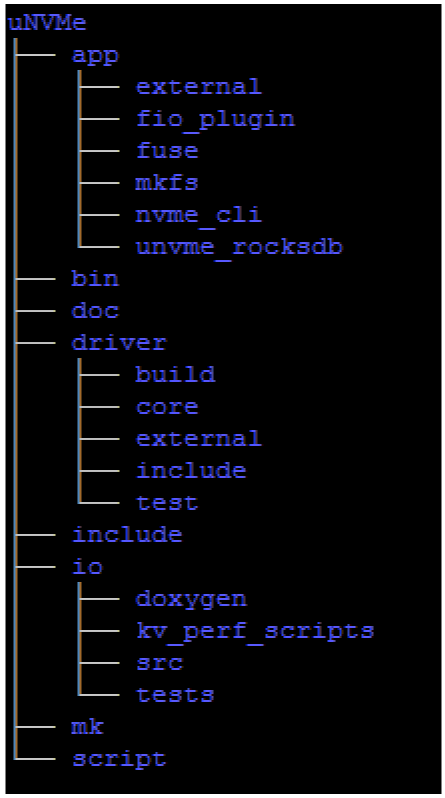


Figure 1. uNVMe Directory

4.1 bin

Figure 2. uNVMe – bin Directory

The *bin* directory contains the one KV libraries on debian, libuio.a. We have confirmed it operates normally on debian distro 4.9 with kernel version 4.9, 4.11.1, and 4.12.0 respectively.

libuio.a – provides API, Cache, Slab MM, Sync/Async IO Handler, and a way to make multithread application. And it provides a set of functions as KV NVMe User level device driver such NVMe Queue management, command SQ and CQ handling, and Command flow control.

[NOTE] A high level application must import this library to fully make use of uNVMe SDK without knowing device dependency.

4.2 include

```
include/
├── kv_apis.h
└── kv_types.h
```

Figure 3. include directory

The *include* directory contains kv_apis.h and kv_types.h that include APIs and structures

5 UNVME SDK INSTALLATION

This section explains how to build uNVMe SDK. Building SDK needs to be done in the following order.

5.1 Prerequisite

5.1.1 Supported Platforms

GNU/Linux is supported as a development and production platform.

5.1.2 Install required software

A number of software packages are needed to prior to build and use uNVMe SDK. Following steps are for installing the packages:

Install OS dependent packages in an autonomous manner (for Ubuntu/Debian/CentOS)

$ ./script/pkgdep.sh

Below is the "pkgdep.sh" file.

```
#!/bin/sh
# Please run this script as root.

SYSTEM=`uname -s`

if [ -s /etc/redhat-release ]; then
        # Includes Fedora, CentOS
        if [ -f /etc/centos-release ]; then
                # Add EPEL repository for CUnit-devel
                yum --enablerepo=extras install -y epel-release
        fi
        yum install -y gcc gcc-c++ make CUnit-devel libaio-devel openssl-devel \
                git astyle-devel python-pep8 lcov python clang-analyzer libuuid-devel \
                sg3_utils libiscsi-devel
        # Additional dependencies for NVMe over Fabrics
        yum install -y libibverbs-devel librdmacm-devel
        # Additional dependencies for DPDK
        yum install -y numactl-devel "kernel-devel-uname-r == $(uname -r)"
```

```
        # Additional dependencies for building docs
        yum install -y doxygen mscgen graphviz
        # Additional dependencies for building pmem based backends
        yum install -y libpmemblk-devel || true
        # Additional dependencies for SPDK CLI
        yum install -y python-configshell
        # Additional dependencies for uNVMe SDK
        yum install -y gflags-devel scons check-devel

elif [ -f /etc/debian_version ]; then
        # Includes Ubuntu, Debian
        apt-get install -y gcc g++ make libcunit1-dev libaio-dev libssl-dev \
                git astyle pep8 lcov clang uuid-dev sg3-utils libiscsi-dev libgflags-dev
        # Additional dependencies for NVMe over Fabrics
        apt-get install -y libibverbs-dev librdmacm-dev
        # Additional dependencies for DPDK
        apt-get install -y libnuma-dev
        # Additional dependencies for building docs
        apt-get install -y doxygen mscgen graphviz
        # Additional dependencies for SPDK CLI
        apt-get install -y "python-configshell*"
elif [ -f /etc/SuSE-release ]; then
        zypper install -y gcc gcc-c++ make cunit-devel libaio-devel libopenssl-devel \
                git-core lcov python-base python-pep8 libuuid-devel sg3_utils
        # Additional dependencies for NVMe over Fabrics
        zypper install -y rdma-core-devel
        # Additional dependencies for DPDK
        zypper install -y libnuma-devel
        # Additional dependencies for building nvml based backends
        zypper install -y libpmemblk-devel
        # Additional dependencies for building docs
        zypper install -y doxygen mscgen graphviz
elif [ $SYSTEM = "FreeBSD" ] ; then
        pkg install gmake cunit openssl git devel/astyle bash devel/pep8 \
                python misc/e2fsprogs-libuuid sysutils/sg3_utils
```

```
        # Additional dependencies for building docs
        pkg install doxygen mscgen graphviz
else
        echo "pkgdep: unknown system type."
        exit 1
fi
```

Or, manual installation for Ubuntu/Debian

$ apt-get install -y gcc g++ make libcunit1-dev libaio-dev libssl-dev libgflags-dev check

$ apt-get install xfsprogs

$ apt-get install linux-tools-common linux-cloud-tools-`uname -r` linux-tools-`uname -r` linux-tools-`uname -r`-generic linux-cloud-tools-`uname -r`-generic

Or, manual installation for CentOS

$ rpm -Uvh http://dl.fedoraproject.org/pub/epel/7/x86_64/Packages/e/epel-release-7-11.noarch.rpm

$ yum update

$ yum install google-perftools google-perftools-devel

$ yum install CUnit-devel openssl-devel gflags-devel libaio-devel check-devel scons

Other Apps

libcheck - unit test framework for C : https://github.com/libcheck/check

libfuse : https://github.com/libfuse/libfuse/tree/master#installation

```
[libfuse Installation]
You can download libfuse from https://github.com/libfuse/libfuse/releases (e.g. fuse-3.2.3.tar.xz)
To build and install, we recommend to use Meson (version 0.38 or newer) and Ninja. After extracting the libfuse tarball, create a (temporary) build directory and run Meson:
$ tar -xvf fuse-3.2.3.tar.xz
$ cd fuse-3.2.3
$ mkdir build; cd build
$ meson ..

Normally, the default build options will work fine. If you nevertheless want to adjust them, you can do so with the mesonconf command:
$ mesonconf # list options
$ mesonconf    -D disable-mtab=true # set an option
```

To build, test and install libfuse, you then use Ninja:

$ ninja

$ sudo python3 -m pytest test/

$ sudo ninja install

Optional libraries

- *srandom - fast random generator :* [*https://github.com/josenk/srandom*](https://github.com/josenk/srandom)

  If users want to use the ***verify*** option of kv_perf and sdk_perf, *srandom* should be installed.

  $ cd driver/external/srandom

  $ make load

  $ sudo make install

- *valgrind - a suite of tools for debugging and profiling : http://repo.or.cz/valgrind.git*

  $ sudo apt-get install valgrind

  $ sudo valgrind --tool=memcheck ./sdk_perf_async

5.2 build DPDK/SPDK

#make.sh intel

This will complie and create SPDK/DPDK libraries of which uNVMe SDK make use.

5.3 build uNVMe SDK

#make.sh all

This will compile and create uNVMe library, libuio.a, and sort of test applications such as kv_perf, udd_perf, sdk_perf and etc. Test applications are located under io and driver/tests.

```
io/
├── SConstruct
├── alloc_perf
├── alloc_udd_perf
├── cache_multi_perf
├── cache_perf
├── deps
├── doxygen
├── hash_perf
├── kv_perf
├── kv_perf_scripts
├── kv_sdk_async_config.json
├── kv_sdk_simple_config.json
├── kv_sdk_sync_config.json
├── libkvio.a
├── radix_perf
├── sdk_exist_append_iterate_getlog
├── sdk_iterate
├── sdk_iterate_async
├── sdk_perf
├── sdk_perf_async
├── src
├── tags
├── test_iterate
├── test_iterate_async
├── test_json
├── test_simple_config
├── test_simple_config_adv
└── tests
```

```
driver/test/
├── Makefile
├── command_ut
├── hash_perf
├── iterate
├── iterate_async
├── nvme-cli
├── udd_perf
└── udd_perf_async
```

5.4 build APP

#make.sh app

This will build and create unvme2_fio_plugin, kv_nvme, and kv_db_bench.

unvme2_fio_plugin is a fio_plugin that is compatible with fio cli tool as a ioengine.

kv_nvme is a cli tool that is compatible with legacy nvme cli tool.

kv_db_bench is geared to generate IO workload on KV_SSD based on db_bench that is a official BM tool of Rocksdb.

mkfs is a tool to install Blobstore file system on a LBA SSD.

fuse is a tool to mount a LBA SSD as fuse file system.

unvme_rocksdb is a RocksDB with uNVMe SDK for a LBA SSD.

[NOTE] kv_db_bench only operates on KV_SSD.

5.5 Setup Hugepage Memory

As uNVMe SDK requires hugepages to work on, hugepages preparation is needed before uNVME SDK run. When a system boots, the script/setup.sh has to be run once to prepare user device driver and reserve huge pages for NVMe IO as below.

(In the uNVMe SDK directory)

 # NRHUGE=2048 ./script/setup.sh

The NRHUGE indicates the number of huge pages that will be used during NVMe IO with user level device driver. Normally, one huge page occupies 2MB in size so your system will reserve 4GB size memory with huge pages on the above example script. The above script will unload kernel nvme device driver, which means NVMe SSDs (includes other LBA NVMe SSDs) in your system will be controlled by KV NVMe user level driver instead of NVMe Kernel driver.

In order to change system to use kernel device driver again, following command need to be run.

 # ./script/setup.sh reset

With the above commands, you can switch either to use user level driver or kernel level driver in turn.

However, when your system makes use of an NVMe SSD as Boot Device, it requires a further process for setting up KV user level device. Other H/W Buses such as SATA, USB, and etc. are fine. This is due to the fact that the *setup.sh* changes the device driver of all the connected NVMe SSD to use kernel or user device driver. You need to specify the list of PCI BDF (Block Device File) ID of changing nvme devices on setup.sh, except nvme boot device as below.

1. Check the PCI BDF ID of NVMe SSDs connected

*# ls -l /sys/block/nvme**

lrwxrwxrwx 1 root root 0 Aug 19 00:21 /sys/block/nvme0n1 -

> ../devices/pci0000:00/0000:00:01.1/**0000:02:00.0**/nvme/nvme0/nvme0n1/

lrwxrwxrwx 1 root root 0 Aug 19 00:21 /sys/block/nvme1n1 -

> ../devices/pci0000:00/0000:00:1c.7/**0000:0b:00.0**/nvme/nvme1/nvme1n1

lrwxrwxrwx 1 root root 0 Aug 19 00:21 /sys/block/nvme1n1 -

> ../devices/pci0000:00/0000:00:1c.7/**0000:0c:00.0**/nvme/nvme1/nvme1n1

2. Launch *setup.sh* with the list of PCI BDF, except nvme boot device. Assuming the PCI BDF of boot device above case is 0000:02:00.0, you need to change other two: 0000:0b:00.0 0000:0c:00.0

 # NRHUGE=2048 ./setup.sh config "0000:0b:00.0 0000:0c:00.0"

config on target_bdf: 0000:0b:00.0 0000:0c:00.0

0000:0b:00.0 (144d a808): nvme -> uio_pci_generic

0000:0c:00.0 (144d a808): nvme -> uio_pci_generic

3. check the status of kernel / user device driver in use

 # ./setup.sh status

NVMe devices

BDF	Numa Node	Driver name	Device name
0000:02:00.0	0	nvme	nvme0
0000:0b:00.0	0	uio_pci_generic	-
0000:0c:00.0	0	uio_pci_generic	-

Above message means that 0000:02:00.0 (=NVMe Boot DEVICE) is controlled by kernel device driver while 0000:0b:00.0, 0000:0c:00.0 is controlled by user device driver.

Please further note that, although your system has a sufficient amount of DRAM, requesting a large huge page, for instance, NRHUGE=100000, will fail depending on system status. On the error case, you should change a proc parameter for system to allow the large number of huge pages allocation as below.

 # echo 524240 > /proc/sys/vm/max_map_count

5.6 Validate installation of the SDK

To check whether the SDK installation is successful or not, we strongly recommend users to do test SDK. All about test is described in the Testing and Benchmarking tools section.

5.7 Update a Firmware of LBA/KV SSD via nvme-cli

5.7.1 Install nvme-cli

[NOTE] Prior to updating FW, users who is using uNVMe SDK and set up huge pages need to change back their system to use kernel device driver again.

A firmware updating needs *nvme-cli* which is NVMe user space tool for Linux. Therefore, nvme-cli should be installed on your system. Below is a set of linux commands for installation of nvme-cli:

$ sudo add-apt-repository ppa:sbates

$ sudo apt-get update

$ sudo apt-get install nvme-cli

5.7.2 Download and activate a firmware on a target SSD

A process of updating a firmware consists of downloading and activation. In this subsection, this document describes how to download and activate the firmware to the target SSD by using an example. Let /dev/nvme0 be a target SSD and EHA50K0B be a firmware to be downloaded. Then, below commands download the firmware into the target SSD and activate the firmware:

$ sudo nvme fw-download /dev/nvme0 --fw=EHA50K0B.bin

$ sudo nvme fw-activate /dev/nvme0 --slot=2 --action=1

$ sudo shutdown –h now

[NOTE] Updating or changing firmware of SSD needs hard reboot of the system or hot-plug.

5.7.3 Update Verification

After the hard reboot of the system, users are able to check whether the update is successful or not by using below command:

$ sudo nvme id-ctrl /dev/nvme0

```
root@kvssd:~# nvme id-ctrl /dev/nvme0
NVME Identify Controller:
vid     : 0x144d
ssvid   : 0x144d
sn      : S3VJNY0J600141
mn      : SAMSUNG MZQLB3T8HALS-000AZ
fr      : EHA50K0B
rab     : 2
ieee    : 002538
cmic    : 0
```

Figure 4. An example of update checking

If the user's update completed successfully, the *fr* field is updated to the new version of the firmware like Figure above.

[NOTE] If the target device is not found, try to validate the update once again after one more hard-reboot.

For more details about nvme-cli, please refer the nvme-cli website.

5.8 Update a Firmware of LBA/KV SSD via kv_nvme

[NOTE] Before update FW, users must enable UDD and set up huge pages.

5.8.1 Download and activate a firmware on a target SSD

A process of updating a firmware consists of downloading and a hard reset. In this subsection, this document describes how to download the firmware to the target SSD and hard reset by using an example. Let 0000:02:00.0 be a PCI address of target SSD and EHA50K0B be a firmware to be downloaded. Then, below commmads download the firmware into the target SSD and activate the firmware:

$ sudo ./kv_nvme fw-download 0000:02:00.0 --*fw=EHA50K0B.bin*

$ sudo shutdown –h now

[NOTE] Updating or changing firmware of SSD needs hard reboot of the system or hot-plug.

[NOTE] There is no need to do 'fw-activate' after 'fw-download' because kv_nvme downloads a given firmware and activates the firmware when users do 'fw-download'.

5.8.2 Update Verification

After the hard reboot of the system, users are able to check whether the update is successful or failed by using below command:

$ sudo ./kv_nvme id-ctrl 0000:02:00.0

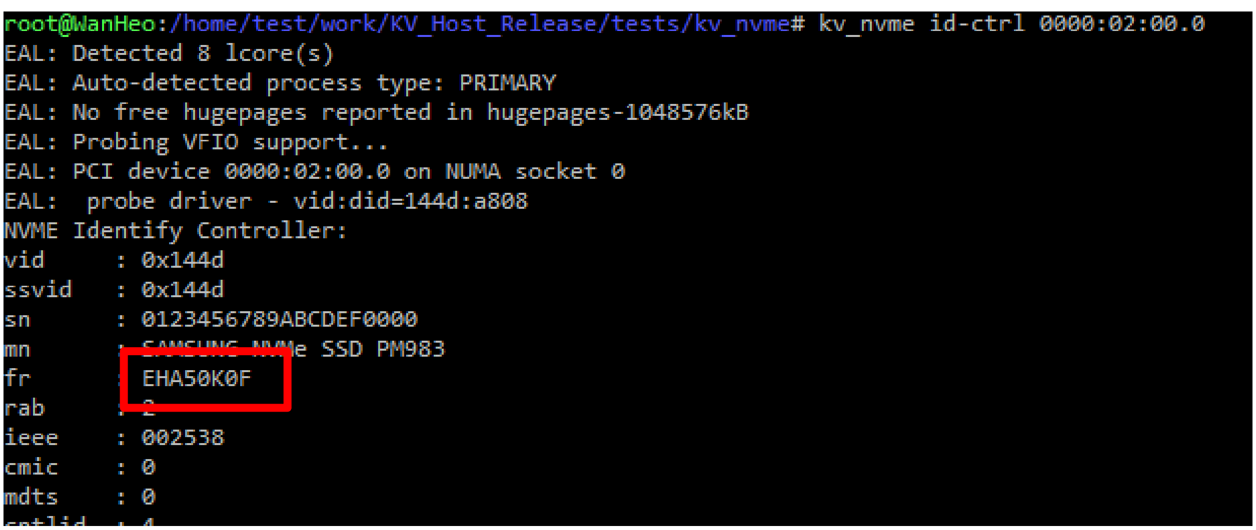


Figure 5. An example of update checking

If the user's update completed successfully, the *fr* field is updated to the new version of the firmware like Figure above.

[NOTE] If the target device is not found, try to validate the update once again after one more hard-reboot.

For more details about nvme-cli, please refer the README file of kv_nvme.

5.9 KV SSD Hot-Plug with uNVMe

uNVMe supports limited hot-plug functionality on KV SSDs. Technically, it is not the pure hot-plug operation, but the manual exchange of SSD without system reboot. The limitation is originated from the availability of H/W PCI interface, kernel PCI driver, and the fact that current PCI hot-plug event lead to loading/unloading LBA NVMe Kernel driver for the PCI NVMe class code, 0x010802, reserved for NVMe. Please refer to figure below how to exchange KV SSD without system reboot.

Firstly, by using lspci command, you can check the availability of a KV SSD on a PCI link.

```
lspci –v    (when KV SSD is plugged)
0b:00.0 Non-Volatile memory controller: Samsung Electronics Co Ltd Device a808 (prog-if 02 [NVM Express])
  Subsystem: Samsung Electronics Co Ltd Device a801

lspci –v    (when KV SSD is unplugged)
0b:00.0 Non-Volatile memory controller: Samsung Electronics Co Ltd Device a808 (rev ff) (prog-if ff)

Or it could show nothing.
```

Next, need to run command below according to SSD unplug/plug event.

```
After Unplug device
        # echo 1 > /sys/bus/pci/rescan
        # {path}/{to}/{uNVMe SDK}/script/setup.sh reset

After Plug device
        # echo 1 > /sys/bus/pci/rescan
        # {path}/{to}/{uNVMe SDK}/script/setup.sh reset
```

Having done the sequence above, it is able to recognize KV SSD without rebooting system.

[NOTE] In our test, all the 10 PCI slots of a server machine (Supermicro) support hot-plug, while only 0 or 1 out of 2 and 3 slot is available on a desktop machine. In this sense, the availability of the hot-plug varies on H/W PCI interface and kernel PCI driver.

[NOTE] For some system, the manual pci rescan command is not needed, but, we've experienced that complying above sequence is better to make plug/unplug work.

6 TESTING AND BENCHMARKING TOOLS

For those who generate IO, check performance, and validate SSD status over uNVMe SDK, tools below are provided.

- unvme2_fio_plugin: fio-plugin as ioengine of fio. Support KV/LBA SSD.
- kv_nvme: nvme-cli compatible tool to issue admin/io command. Support KV/LBA SSD.
- mkfs: a tool to install Blobstore file system on a LBA SSD.
- fuse: a tool to mount a LBA SSD as fuse file system.
- unvme_rocksdb: rocksdb with uNVMe SDK for a LBA SSD.
- kv_rocksdb: rocksdb for a KV SSD, cast store/retrieve/delete io request directly on a KV SSD, bypassing the majority of original rocksdb operation such as LSM and compaction.
- kv_perf: I/O workloads generator and performance analyzer. Support KV/LBA SSD.
- sdk_perf: simple application to verify the device and issue store/retrieve/delete/exist request. The primary purpose of the app is to show how to make use of uNVMe SDK API. Support KV/LBA SSD.
- sdk_iterate : Based on sdk_perf, further performs iterate operations. Support KV/LBA SSD.

6.1 unvme2_fio_plugin

unvme2_fio_plugin is provided to be used as an *ioengine* of fio. By setting '*ioengine'* to *unvme2_fio_plugin* and '*json_path*' to json-formatted configuration file (either at job file or at cmd line options), users can make fio working based on uNVMe.

unvme2_fio_plugin has two main advantages.

1) Users can generate IO workload to KV SSD as well as LBA SSD just by changing the option named '*ssd_type*'. There are benchmark tools developed by Samsung Electronics for KV-SSD verification (*kv_perf*, *sdk_perf*, etc.), but this plugin has a meaning to enable the verification of KV-SSD by the workload of fio which is one of the most popular IO benchmark tools.
2) The performance of the fio can be improved by supporting the CPU affinity customizing. Corresponding method and operation principles are described in the FIO options and NUMA_aware CPU affinity Setting.

6.1.1 How to use

[NOTE] Before building and running unvme2_fio_plugin, please check uNVMe SDK was installed properly on your system and hugepage memory was set enough to run uNVMe SDK applications. See uNVMe SDK Installation, especially from Prerequistie to Setup Hugepage Memory for more details.

6.1.1.1 Build fio

In this section, how to build and install fio from its source code is described. Below commands are written in assumption that a user wants to install fio-2.20.

```
$> cd {path}/{to}/{uNVMe SDK}/app/external/fio
or get from open source "$> wget https://github.com/axboe/fio/archive/fio-2.20.tar.gz"
$> tar xvf fio-2.20.tar.gz
$> cd fio-fio-2.20
$> ./configure
$> make
#> make install
```

The user can check installation with below command:

```
$> fio –version
```

If the installation is successful, result of the command is '`fio-2.20`'. These commands can be applied to other version than 2.20, simply modifying version.

6.1.1.2 Build unvme2 fio plugin

Users can build *unvme2_fio_plugin* through `*./make.sh app*`, as introduced in build APP. In this case, *unvme2_fio_plugin* is basically built to fit the fio version of users' system. However, if users want a binary that is compatible with a specific version of fio, it is possible to build *unvme2_fio_plugin* binary that is compatible with a specific version of fio by modifying *unvme2_fio_plugin*'s Makefile.

```
ifeq ($(KV_DIR),)
export KV_DIR = $(shell pwd)/../../
endif

# if you want specific version fio instead of system's fio, enable both  below options
# ex) fio-3.3 -> FIO_MAJOR_VERSION=3, FIO_MINOR_VERSION=3
#FIO_MAJOR_VERSION=3
#FIO_MINOR_VERSION=3

```

Figure 6. Makefile of unvme2_fio_plugin

As you can see in Figure 4, variables *FIO_MAJOR_VERSION* and *FIO_MINOR_VERSION* are disabled in Makefile. When the users enable it and set the variables to fio version, and then the build is performed again, the *unvme2_fio_plugin* binary compatible with the set fio version is generated.

[NOTE] *unvme2_fio_plugin* supports **fio version from 2.7 to 3.5**

```
$> vi {path}/{to}/{uNVMe SDK}/app/fio_plugin/Makefile
FIO_MAJOR_VERSION=3
FIO_MINOR_VERSION=3

After chaging Makefile, build app for new fio version
$> cd {path}/{to}/{uNVMe SDK}/
$> ./make.sh app

Execute fio using built fio version
$> cd {path}/{to}/{uNVMe SDK}/app/fio_plugin/
$> ./fio-3.3 Sample_Write.fio
```

6.1.1.3 Run fio

For users' convenience, a sample *unvme2_fio_plugin* job files are provided (located at the same directory of the plugin). Below command will generate 4KB sequential write workload to the device *0000:02:00.0* during 5 sec, with iodepth=128, 1 job thread.

```
#> fio Sample_Write.fio
```

6.1.1.4 Setting fio and device

As you can find in *Sample_Write.fio*, there are **five major differences** to use unvme2_fio_plugin compared to fio's original io engine like *libaio*.

```
[global]
ioengine=./unvme2_fio_plugin
json_path=./unvme2_config.json
clat_percentiles=1
percentile_list=1.0:5.0:10.0:20.0:30.0:40.0:50.0
thread=1
group_reporting=1
direct=1
verify=0
time_based=0
ramp_time=0
runtime=5
rw=write
iodepth=128
bs=4k
ks=16

[test]
filename=0000.02.00.0
numjobs=1
```

Figure 7. Sample_Write.fio

1) ***'ioengine'*** should be set to the */{path}/{to}/{unvme2_fio_plugin binary}.*
2) ***'json_path'*** should be set to the */{path}/{to}/{unvme2 device configuration file}.* This is added newly as unvme2_fio_plugin's own parameter.
3) ***'bs'*** is used to specify IO size. It implies block size of single IO request on LBA SSD, while value size of a KV pair on KV SSD.
4) ***'ks'*** is used to specify key size(default: 16(B)). Please note that this option applies on KV SSD only, not on LBA SSD.
5) ***'filename'*** should be set to the *devices' PCI bdf address, NOT to the* devices' file path like */dev/nvme0*.

In the json configuration file specified at *json_path*, json formmated information of the SSDs should be included. For example, if users want to run *unvme2_fio_plugin* at device '0000:02:00.0', users should describe the device's type (LBA or KV), PCI BDF address (0000:02:00.0), core mask (which cores will be used to deal IO submission), and cq thread mask (at which cores completion thread will be pinning) into the json file. Please refer to json_path and filename sections.

[NOTE] Json formatted config file is generally used to initialize uNVMe SDK. Users can find more details and exact meaning of parameters described in the config file at uNVMe SDK Configuration section.

[NOTE] Users can run fio at the devices ONLY described at json config file.

6.1.1.4.1 Case: Single Device

Sample configuration1: unvme2_config.json
// to run fio at single device (0000:02:00.0)

```
{
   "ssd_type" : "lba",                     // Type of SSD ("lba" or "kv")
   "device_description" : [
      {
         "dev_id" : "0000:02:00.0",        // PCI BDF address of SSD; NOTE that uNVMe SDK cannot be initialized with wrong dev_id.
         "core_mask" : 3,                  // core_mask of dev 0000:02:00.0
         "cq_thread_mask" : "c"            // cq thread mask of dev 0000:02:00.0
      }
   ]
}
```

After setting parameter ***--json_path=unvme2_config.json***, users can generate and test workloads to the device 0000:02:00.0 by setting fio parameter ***--filename=0000.02.00.0***.

[NOTE] If *cq_thread_mask* of config.json is set to 0, fio will run without completion thread(s).

6.1.1.4.2 Case: Multiple Devices

Sample configuration2: unvme2_config_multi.json

```
// to run fio at 2 devices (0000:01:00.0 and 0000:02:00.0)
{
   "ssd_type" : "lba",
   "device_description" : [
      {
         "dev_id" : "0000:01:00.0",
         "core_mask" : 1,
         "cq_thread_mask" : 4
      },
      {
         "dev_id" : "0000:02:00.0",
         "core_mask" : 1,
         "cq_thread_mask" : 8
      }
   ]
}
```

After setting parameter ***--json_path=unvme2_config_multi.json***, users can generate and test workloads to the devices 0000:01:00.0 and 0000:02:00.0 by setting fio parameters (job description) like below.

For single IO job to two devices,

```
    [test0]
    filename=0000.01.00.0:0000.02.00.0
    numjobs=1
```

For single IO job per each device,

```
    [test0]
    filename=0000.01.00.0
    numjobs=1

    [test1]
    filename=0000.02.00.0
    numjobs=1
```

For more details about multiple devices settings, please see Example of FIO options for multiple devices testing.

6.1.1.4.3 Case: Multiple Jobs

Because **the number of driver IO queues is determined by the number of cores set by *core_mask*** in config JSON, if users want to run multiple jobs on single device, it is recommended to match the number of cores set (by *core_mask*) with the value of *numjobs*.

Sample configuration3

(config.json 1)

```
{
  …
    {
      "dev_id" : "0000:01:00.0",
      "core_mask" : "F"        //means core 0, 1, 2 and 3 are set; 4 IO queues will be generated
      "cq_thread_mask" : 4
    }
}
```

(FIO job description 1)

```
[test0]
    filename=0000.01.00.0
    numjobs=4                    //each FIO job(i.e. IO thread) will be mapped with each IO queue
```

Sample configuration4

(config.json 2)

```
{
    …
        {
            "dev_id" : "0000:01:00.0",
            "core_mask" : "0x1"              //means just 1 IO queue will be generated
            "cq_thread_mask" : 4
        }
}
```

(FIO job description 2)

```
[test0]
        filename=0000.01.00.0
        numjobs=4                    //all of IO jobs will share 1 IO queue; may cause performance drop
```

6.1.2 Supporting FIO Options

Because of architectural difference, there are some changes in fio options in terms of usage or meaning. In this section, the fio options which have possibility to cause users confusion are described.

■ **Support FIO Options belows**

Options	Description	Support
rw	read/write/randread/randwrite	○
blocksize	Set blocksize (default=4K), means value_size at KV SSD	○
blocksize_range	Specify a range of I/O block sizes. Example: bsrange=1k-4k,2k-8k	○
blockalign	At what boundary to align random IO offsets.	○
rwmixread=*int*	Percentage of a mixed workload that should be reads. Default: 50	○
random_distribution=*random* | *zipf* | *pareto*	By default, fio will use a completely uniform random distribution when asked to perform random IO. Sometimes it is useful to skew the distribution in specific ways, ensuring that some parts of the data is more hot than others.	○
norandommap	Normally fio will cover every block of the file when doing random I/O. If this parameter is given, a new offset will be chosen without looking at past I/O history.	○
cpumask=*int*	Set CPU affinity for this job. *int* is a bitmask of allowed CPUs the job may run on.	△ (override)
cpus_allowed=*str*	Same as cpumask, but allows a comma-delimited list of CPU numbers.	△ (override)
numa_cpu_nodes=*str*	Set this job running on spcified NUMA nodes' CPUs. The arguments allow comma delimited list of cpu numbers, A-B ranges, or 'all'.	△ (override)
sync=*bool*	Use synchronous I/O for buffered writes. For the majority of I/O engines, this means using O_SYNC. Default: false.	
do_verify=*bool*	Run the verify phase after a write phase. Only valid if verify is set. Default: true.	○
verify=*str*	Method of verifying file contents after each iteration of the job. Allowed values are: md5 crc16 crc32 crc32c crc32c-intel crc64 crc7 sha256 sha512 sha1 Store appropriate checksum in the header of each block. crc32c-intel is hardware accelerated SSE4.2 driven, falls back to regular crc32c if not supported by the system.	○
group_reporting	If set, display per-group reports instead of per-job when numjobs is specified.	○
thread	Use threads created with pthread_create(3) instead of processes created with fork(2).	○
fsync(fdatasync)	How many I/Os to perform before issuing an fsync(2)/fdayasync(2) of dirty data. If 0, don't sync. (default=false)	○
size	Total size of I/O for this job. fio will run until this many bytes have been transferred, unless limited by other options (runtime, for instance, or increased/descreased by io_size). It is also possible to give size as a percentage between 1 and 100.	○
time_based / runtime	time based workload	○
ramptime		

6.1.2.1 cpumask / cpus_allowed / cpus_allowed_policy / numa_cpu_nodes

Originally, these options are used to distribute CPUs on jobs. But in this plugin, these options are override by JSON configuration file (parameter '*core_mask*').

6.1.2.2 json_path

As described at How to use, parameter '*json_path*' is a newly added as a ***MANDATORY*** option on unvme2_fio_plugin. This option is used to specify a json type configuration file which is for configuration for uNVMe SDK. (For more information about unvme2 configuration, see uNVMe SDK Configuration).

[NOTE] There is a restriction on the use of the option that json type configuration file must have configuration of devices which are in '*filename*' option. If there is single incorrect device(s) in '*json_path*', fio CANNOT start even though there are proper devices only in a '*filename*' option.

6.1.2.3 filename

[NOTE] On unvme2_fio_plugin, all devices of the '*filename*' option MUST be configured by a json type file which is specified by '*json_path*' option.

When users are using kernel inbox drivers, users may set block device(s) as argument of '*filename*' option like below example.

filename=/dev/nvme0

BUT, if users want to use fio with uNVMe SDK, ***users must set PCI BDF address (es) as 'filename' argument*** like below example.

filename=0000.02.00.0 (Assume that users' target device's PCI BDF address is 0000:02:00.0)

Like using of original fio, users can specify multi-device to file name option with ':'

filename=0000.02.00.0:0000.03.00.0 (Assume that users' target block devices are 0000.02.00.0 and 0000.03.00.0)

6.1.2.4 Example: FIO options for multiple devices testing

As described above sections, users can run fio with uNVMe SDK by setting BDF addresses of the devices at *filename*. Please refer to below json configuration and fio job description file for more details.

(unvme2_config_multi.json)

```
{
  "ssd_type" : "lba",
  "device_description" : [
    {
      "dev_id" : "0000:01:00.0",
      "core_mask" : 1,
      "cq_thread_mask" : 4
    },
    {
      "dev_id" : "0000:02:00.0",
      "core_mask" : 1,
      "cq_thread_mask" : 8
    }
  ]
}
```

(Sample_Write_Multi.fio)

```
[global]
ioengine=./unvme2_fio_plugin
json_path=./unvme2_config_multi.json
thread=1
group_reporting=0
direct=1
verify=0
```

```
time_based=0
ramp_time=0
size=10G
offset_increment=10G
rw=write
iodepth=128
bs=4k
ks=16

#test0 will be run on the device 0000:01:00.0, and run on the CPU 0 because the 'core_mask (at json)' of the device is "0x1"
[test0]
filename=0000.01.00.0
numjobs=1

#test1 will be run on the device 0000:02:00.0, and run on the CPU 0 because the 'core_mask (at json)' of the device is "0x1"
[test1]
filename=0000.02.00.0
numjobs=1

#test2 will be run on the devices both 0000:01:00.0 and 0000:02:00.0 (num io jobs : num devices = 1 : 2),
#and run on the CPU 0 because CPU 0 is the only one which can submit IO to both devices (core_mask of the devices is same as '0x1')
[test2]
filename=0000.01.00.0:0000.02.00.0
numjobs=1
```

6.1.2.5 Advanced configuration

On unvme2_fio_plugin, each job is pinned to single core. In addition, IO submission is allowed on some cores which are configured by a json type configuration file. Due to these differences, some workloads and a combination of environmental settings may not work as expected. A representative example is as follows.

6.1.2.5.1 Multi-jobs on single device / single core

Multi jobs on single device may occurs performance drop if *core_mask* of the device is not set properly. Please refer to section Multiple Jobs for more details.

6.1.2.5.2 Single job on multi-devices

Single job test on multi-devices can be run 1) if *core_mask*s of the devices are same or 2) if there is a core(s) set by all of the devices. Below figure shows proper configurations for the single job on multi-devices workload.

```
{
  "ssd_type" : "lba",
  "device_description" : [
    {
      "dev_id" : "0000:01:00.0",
      "core_mask" : 1,
      "cq_thread_mask" : 4
    },
    {
      "dev_id" : "0000:02:00.0",
      "core_mask" : 1,
      "cq_thread_mask" : 8
    }
  ]
}
```

Figure 8. Proper configuration for single-job on multi-devices

If there is not a core(s) set by all of the devices, IO thread cannot generate IO to the devices. Below figure shows wrong configurations for the single job on multi-devices workload. Under the configuration, device 0000:01:00.0 is set on core 0, whereas device 0000:02:00.0 is set on core 1 (i.e. there is no overlapped core(s) between two devices).

```
{
  "ssd_type" : "lba",
  "device_description" : [
    {
      "dev_id" : "0000:01:00.0",
      "core_mask" : 1,
      "cq_thread_mask" : 4
    },
    {
      "dev_id" : "0000:02:00.0",
      "core_mask" : 2,
      "cq_thread_mask" : 8
    }
  ]
}
```

Figure 9. Wrong configuration for single-job on multi-devices

[NOTE] On multi numa-node system, a measured performance is greatly affected by CPU affinity. Thus, when setting the option core_mask / cq_thread_mask which are related to CPU affinity, factors affect performance (ex) NUMA node, CPU redundancy, etc) should be considered well. Detail of CPU affinity setting will be described subsequent section.

6.1.3 NUMA-aware CPU affinity Setting

For your understanding of impact of CPU affinity setup, we give two configuration examples and compare performance of them. One is NUMA-aware configuration; the other is Non NUMA-aware configuration.

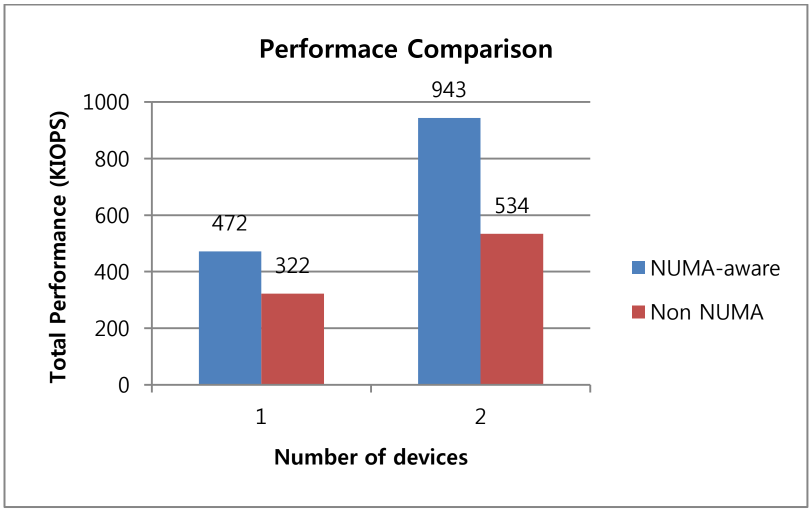


Figure 10. Performance comparison by CPU affinity setting

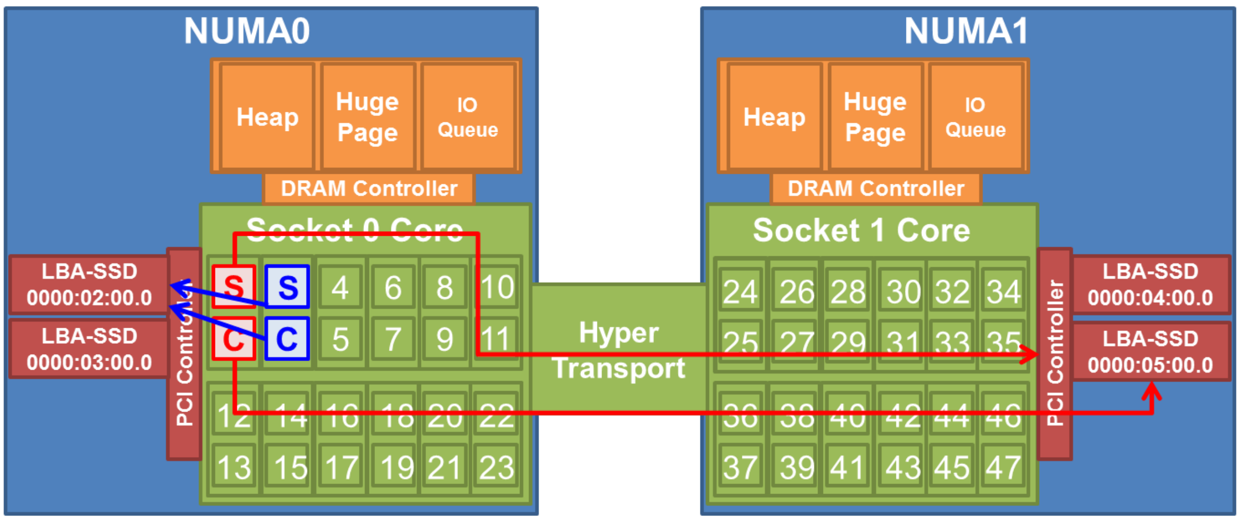
NUMA-aware CPU affinity setting (2 devices)	Non NUMA-aware CPU affinity setting (2 devices)

{
  "ssd_type" : "lba",
  "device_description" : [
	{
	  "ssd_type" : "lba",
	  "device_description" : [

```
    {
      "dev_id" : "0000:05:00.0",
      "core_mask" : 1,
      "cq_thread_mask" : 2
    },
    {
      "dev_id" : "0000:02:00.0",
      "core_mask" : 4,
      "cq_thread_mask" : 8
    }
  ]
}
```

```
    {
      "dev_id" : "0000:05:00.0",
      "core_mask" : 1000000,
      "cq_thread_mask" : 1
    },
    {
      "dev_id" : "0000:02:00.0",
      "core_mask" : 1,
      "cq_thread_mask" : 2000000
    }
  ]
}
```

Average performance of good case is about 472 KIOPS regardless number of devices. However, average performance of bad case is 322 KIOPS when number of device is 1. Worse, average performance of bad case is decreasing to 267 KIOPS when the number of device is 2.



* **S:** *set as submission core*, * ***C****: set as completion core*

Figure 11. Applied NUMA-aware configuration

Let's analyze why this performance drop happens. In this system, the device 0000:02:00.0 locates on NUMA node 0, and the device 0000:05:00.0 locates on NUMA node 1. Core 0 ~ 23 are in NUMA node 0, and core 24 ~ 47 are in NUMA node 1.

In Figure above, with NUMA-aware configuration, all devices' **submission/completion cores locates in the same NUMA Node**. Also there is **NO core redundancy**.

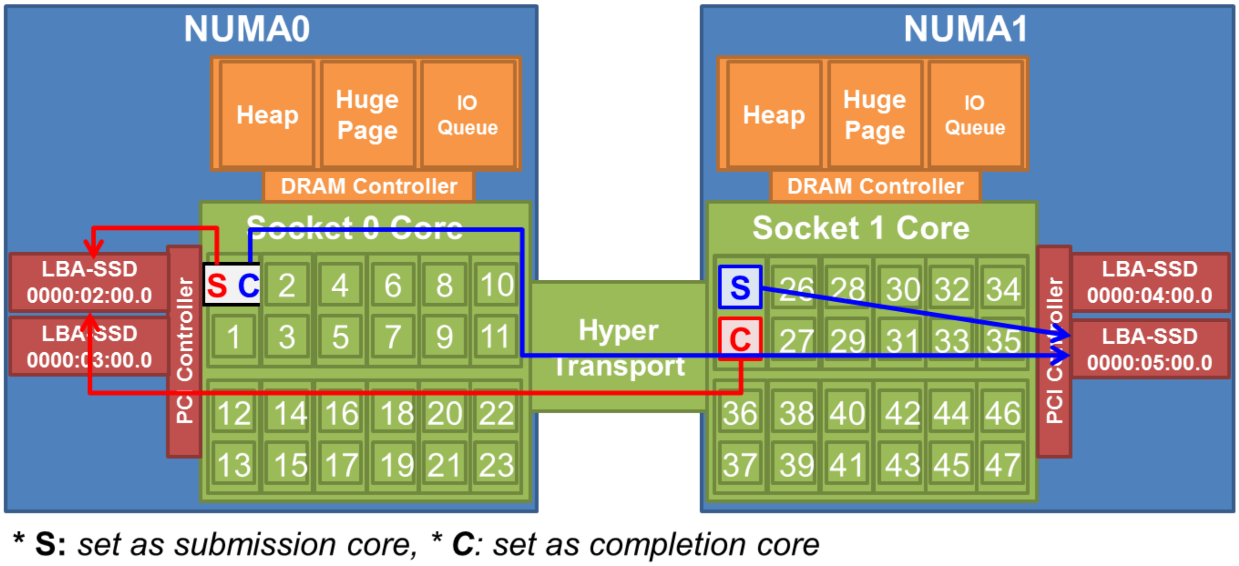


Figure 12. Applied Non NUMA-aware configuration

While, in Non-NUMA-aware case, all devices' core_mask and cq_thread_mask locate in different NUMA Node. (core 0,24, and 25). **Mismatch between the submission/completion cores' NUMA node** occurs (See Figure above). In addition, in the non NUMA-aware configuration**, core 0 is used by both devices** (for submission on 0000:02:00.0, for completion on 0000:05:00.0). In short, decreasing performance caused by Non NUMA-aware and core redundancy.

Test environment:

- CentOS 7, 3.10.0-693.el7.x86 64
- SMC US2023-TR4
- AMD EPYC 7451p (24 core / 48 threads) x 2
- Device: PM983 (4TB) x 2
- FW version : EDA53W0Q (171116)
- LBA SSD Mode
- 4K Sequential Write (QD 128, After format devices)
- fio-3.5

[NOTE] Even with non NUMA-aware configuration, there may be no IO performance drop in some system. If each allocated NUMA nodes locates on one CPU socket, even with mismatch of NUMA nodes between submission/completion, there may be no performance drop. In this case, no memory access via interconnect network occurs.

6.1.3.1 CPU Cache-aware CPU affinity Setting

Even if CPU affinity is set in consideration of NUMA Node, performance drop may occur on some architecture. The AMD architecture of the test bed in 5.1.3 is an example thereof. The CPU consists of a total of 24 cores and there are 4 NUMA nodes.

NUMA Node	core id	L3Cache sharing
NUMA 0	0, 1, 2	L3
	3, 4, 5	L3
NUMA 1	6, 7, 8	L3
	9, 10, 11	L3
NUMA 2	12, 13, 14	L3
	15, 16, 17	L3
NUMA 3	18, 19, 20	L3
	21, 22, 23	L3

As described above, since the core for submission and the core for completion are located in the same NUMA Node 0, the measured performance of both configurations should be good. However, performance drop occurred when using “NUMA-aware / NO L3 cache sharing” configuration (right).

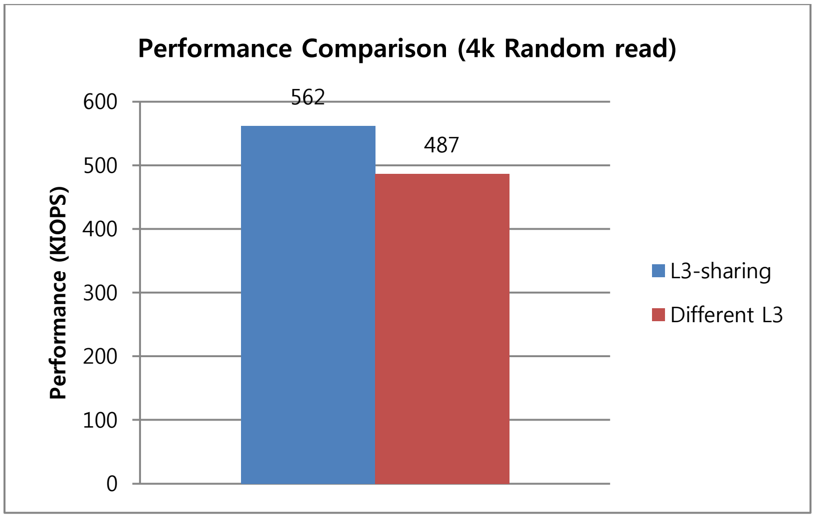


Figure 13. Performance comparison by L3 Cache

<table>
<tr><th>NUMA-aware / L3 cache sharing</th><th>NUMA-aware / NO L3 cache sharing</th></tr>
<tr><td><pre>{
  "ssd_type" : "lba",
  "device_description" : [
    {
      "dev_id" : "0000:05:00.0",
      "core_mask" : 1,
      "cq_thread_mask" : 2
    }
  ]
}</pre></td><td><pre>{
  "ssd_type" : "lba",
  "device_description" : [
    {
      "dev_id" : "0000:05:00.0",
      "core_mask" : 1,
      "cq_thread_mask" : 8
    }
  ]
}</pre></td></tr>
</table>

The performance drop was caused by non L3-cache sharing. L3-cache of the CPU is shared by every three cores. That is, core 0-2 share L3-cache, and core 3-5 share L3-cache. Therefore, in the case of the “NUMA-aware / L3 cache sharing” configuration (on the left), the core for submission (core 0) and the core for completion (core 1) share L3-cache. However, in the case of “NUMA-aware / NO L3 cache sharing” configuration (on the right), the core for submission (core 0) and the core for completion (core 3) use different L3-cache.

Therefore, we recommend that users use cores sharing L3-cache for both submission and completion.

6.1.4 Limitation

unvme2_fio_plugin needs json configuration file which contains NVM devices' information. So users should set parameter '*json_path*' to the json file.

Parameter '*json_path*' is one of the IO engine's own parameters, means that it is valid only for *unvme2_fio_plugin*. It is used identically to other normal parameters, with the caveat that when used on the command line, it must come after the parameter '*ioengine*'.

unvme2_fio_plugin is limited to the thread usage model for now, so fio jobs must also specify thread=1 when using the plugin.

unvme2_fio_plugin doesn't support 'delete' operations of KV SSDs. Write (correspond to store operation in KV) and read (correspond to retrieve operation in KV) is supported.

unvme2_fio_plugin does not support performance reports by group (device).

Without CPU affinity setting, multi jobs (or multi devices) test can cause drop of performance, because of CPU (core) overlapping.

To test on KV SSD, users must set "ssd_type" to "kv" on json configuration file (at *json_path*). Mode change(LBA↔KV) does NOT supported via fio parameter.

If there is a problem with the device sometimes, unvme2_fio_plugin can show performance as 0 instead of stopping. To confirm the abnormality of the device, it is recommended to terminate all processes using unvme2 sdk and to submit any admin command through kv_nvme. If there is an abnormality in the device, the message for the failure of 'nvme_ctrlr_process_init: ***ERROR***' will be displayed. It will be modified in the future.

6.2 kv_nvme

kv_nvme is developed as a tool for managing uNVMe SSD(s) based on *nvme-cli*. Like nvme-cli, kv_nvme provides functions for managing device(s) like format, id-ctrl, get-log. Also, kv_nvme provides unit IO functions for testing.

6.2.1 Main usage

```
--------------------- kv_nvme Howto ----------------------------
1) kv_nvme usage
   ./kv_nvme <command> [<device>] [<args>]

   Admin Commands:
      list           List all NVMe devices and namespaces on machine
      id-ctrl        Send NVMe Identify Controller
      list-ctrl      Send NVMe Identify Controller List, display structure
      get-log        Generic NVMe get log, returns log in raw format
         ./kv_nvme get-log <device> [--log-id=<log-id> | -i <log-id>]
                                    [--log-len=<log-len> | -l <log-len>]
         ex) ./kv_nvme get-log 0000:02:00.0 --log-id=0xc0, --log-len=512
      smart-log      Retrieve SMART Log, show it
      format         Format namespace with new block format
         ./kv_nvme format <device> [--ses=<ses> | -s <ses>] // (optional) Secure Erase Settings. 1: default, 0: No secure
      fw-download    Download new firmware then activate the new firmware
         ./kv_nvme fw-download <device> [--fw=<firmware-file> | -f <firmware-file>]
                                        [--xfer=<transfer-size> | -x <transfer-size>]
                                        [--offset=<offset> | -o <offset>]
         ex) ./kv_nvme fw-download 0000:02:00.0 --fw=EHA50K0F_171208_ENC.bin
      reset          Resets the controller
      version        Shows the program version
      help           Display this help

   IO Commands:        (currently, only support KV-SSD)
      write          Submit a write command
         ./kv_nvme write <device>  [ -k <key>]            // (required), key to be written
                                   [ -l <key-len>]        // (required), length of the key to be written
                                   [ -v <value-file>]     // (required) file includes data to be written
                                   [ -s <value-size>]     // (required) size of data to be written
                                   [ -o <value-offset>]   // (optional) currently not supported
                                   [ -i <io-option>]      // (optional) 0: default, 1: idempotent
                                   [ -n <namespace>]      // identifier of desired namespace (0 or 1), defaults to 0
                                   [ -w ]                 // (optional) displaying arguments of the command
                                   [ -t ]                 // (optional) show latency of the executed command
                                   [ -a ]                 // (optional) for using asynchronous mode, sync mode is default
```

```
    ex) ./kv_nvme write 0000:02:00.0 -k keyvalue12345678 -l 16 -v value.txt -o 0 -s 4096 -a -w -t
        // write (16B key, 4KB value from value.txt) in async mode.
read            Submit a read command
    ./kv_nvme read <device>   [ -k <key>]             // (required), key to be read
                             [ -l <key-len>]         // (required), length of the key to be read
                             [ -s <value-size>]      // (required) size of data to be read
                             [ -v <value-file>]      // (optional) file where the read value to be written,
                                                        if not set, read value will be written to stdout
                             [ -o <value-offset>]    // (optional) currently not supported
                             [ -i <io-option>]        // (optional) 0: default, 2: only read value size of the give key
                                                                         (support large-value only)
                             [ -n <namespace>]       // identifier of desired namespace (0 or 1), defaults to 0
                             [ -w ]                  // (optional) displaying arguments of the command
                             [ -t ]                  // (optional) show latency of the executed command
                             [ -a ]                  // (optional) for using asynchronous mode, sync mode is default
    ex) ./kv_nvme read 0000:02:00.0 -k keyvalue12345678 -l 16 -s 4096 -t -w
         // read (16B key, 4KB value) sync mode
    ex) ./kv_nvme read 0000:02:00.0 -k keyvalue12345678 -l 16 -s 4096 -v output.txt -t -w
         // read (16B key, 4KB value) sync mode, then write read value to output.txt
delete          Submit a delete command
    ./kv_nvme delete <device> [ -k <key>]           // (required), key to be read
                             [ -l <key-len>]        // (required), length of the key to be read
                             [ -i <io-option>]      // (optional) 0: default
                             [ -n <namespace>]      // identifier of desired namespace (0 or 1), defaults to 0
                             [ -w ]                 // (optional) displaying arguments of the command
                             [ -t ]                 // (optional) show latency of the executed command
                             [ -a ]                 // (optional) for using asynchronous mode, sync mode is default
    ex) ./kv_nvme delete 0000:02:00.0 -k keyvalue12345678 -l 16 -t -w
exist           Submit a exist command
    ./kv_nvme exist <device>  [ -k <key>]           // (required), key to check
                             [ -l <key-len>]        // (required), length of the key to be read
                             [ -i <io-option>]      // (optional) 0: default
                             [ -n <namespace>]      // identifier of desired namespace (0 or 1), defaults to 0
                             [ -w ]                 // (optional) displaying arguments of the command
                             [ -t ]                 // (optional) show latency of the executed command
                             [ -a ]                 // (optional) for using asynchronous mode, sync mode is default
    ex) ./kv_nvme exist 0000:02:00.0 -k keyvalue12345678 -l 16 -t -w
          // exist command sync mode
open-it         Open a new iterator
    ./kv_nvme open-it <device> [ -p <prefix>]      // prefix of keys to iterate, defaults to 0x0
                              [ -b <bitmask>]       // bitmask of prefix to apply, defaults to 0xffffffff
                              [ -n <namespace>]     // identifier of desired namespace (0 or 1), defaults to 0
                              [ -i <iterate_type>]   // type of iterator
                                                        (1: KEY_ONLY, 3: KEY_WITH_DELETE)
    ex) ./kv_nvme open-it 0000:02:00.0 -p 0x30303030 -i 2
         // open a new KEY_ONLY iterator whose prefix is "0000" (0x30303030)
```

```
        read-it        Open a new iterator
            ./kv_nvme read-it <device> [ -i <iterator-id>]   // (required) identifier of iterator to read
                                       [ -s <value-size>]      // size of length to iterate read once
                                       [ -v <value-file>]      // (optional) file where the read value to be written, if not set,
read value will be written to stdout
                                       [ -w ]                 // (optional) displaying arguments of the command
                                       [ -t ]                 // (optional) show latency of the executed command
                                       [ -a ]                 // (optional) for using asynchronous mode, sync mode is default
            ex) ./kv_nvme read-it 0000:02:00.0 -i 1 -w
                // issue a iterate read command on iterate_handle (0x1)
        close-it       Close an opened iterator
            ./kv_nvme close-it <device> [ -i <iterator-id>]    // (required) identifier of iterator to close
            ex) ./kv_nvme close-it 0000:02:00.0 -i 1
                 // Close an iterator whose iterator_id is 0x1

2) unsupported commands:
    dsm / flush / get-feature / set-feature / security-send / security-recv / compare / write-zeroes / write-uncor /
subsystem-reset
```

[NOTE] Below commands are unsupported:

dsm / flush / get-feature / set-feature / security-send / security-recv / compare / write-zeroes / write-uncor

6.2.2 Difference with legacy nvme-cli

6.2.2.1 Option --binary | -b

On kernel nvme-cli, users can use 'binary' option without any arguments. However, 'binary' option of kv_nvme **needs a file path**.

cf) On kernel nvme case: nvme id-ctrl /dev/nvme0 –b > result.txt

On kv_nvme case: kv_nvme id-ctrl 0000:02:00.0 ***–b result.txt***

6.2.2.2 Option –dry-run | -d

Option 'dry-run' is used to check without sending commands. However option 'dry-run' is **not supported** on kv_nvme.

6.2.2.3 Option –show | -w

Option 'show' is used to check raw nvme commands. However, Option 'show' is **used to check options of commands in human readable format**.

6.2.2.4 Option –ses | -s

Option 'ses' is used to determine to erase securely when users format devices. On kv_nvme, default value of option 'ses' is **1**.

6.2.2.5 Function subsystem-reset

Function 'subsystem-reset' is used to reset the nvme subsystem of target device. However, 'Subsystem-reset' is **not supported** on kv_nvme.

6.2.2.6 Function fw-activate

Function 'fw-activate' is used to verify and commit a firmware image. On kv_nvme, 'fw-activate' **is issued automatically after 'fw-download'** ***(slot=2, action=1)***. Therefore, FW-ACTIVATE PROHIBITED message will show when users do 'fw-activate' after 'fw-download'.

6.2.2.7 Function write / read / delete / exist

Above functions **can be used on KV-SSD ONLY**. However, Fuction write and read will be supported in the future.

6.2.2.8 INVALID_OPCDE in Function 'list'

If you have at least one LBA-SSD on your system, 'list' may display error message ("INVALID OPCODE"). However, it **does NOT mean abnormal state**. Because kv_nvme submit a command which LBA does not have to determine whether a device is KV or LBA.

```
BDF Address      SN                   Model                                    Namespace Usage                      Format           SSD Type   FW Rev
---------------- -------------------- ---------------------------------------- --------- -------------------------- ---------------- ---------- --------
0000:07:00.0     S3VJNY0J600147       SAMSUNG MZQLB3T8HALS-000AZ               1         764.56  GB /   3.84  TB    512   B +  0 B   LBA        EDA53W0Q
0000:08:00.0     S3VJNY0J600158       SAMSUNG MZQLB3T8HALS-000AZ               1         767.23  GB /   3.84  TB    512   B +  0 B   LBA        EDA53W0Q
0000:83:00.0     S3VJNY0J600159       SAMSUNG MZQLB3T8HALS-000AZ               1         759.58  GB /   3.84  TB    512   B +  0 B   LBA        EDA53W0Q
0000:84:00.0     S3VJNY0J600217       SAMSUNG MZQLB3T8HALS-000AZ               1         759.58  GB /   3.84  TB    512   B +  0 B   LBA        EDA53W0Q
IO COMMAND (a1) sqid:1 cid:254 nsid:1
INVALID OPCODE (00/01) sqid:1 cid:254 cdw0:0 sqhd:0001 p:1 m:0 dnr:0
command error: SC 1 SCT 0
NVMe Status:INVALID_OPCODE(1) Command Result:00005557
0000:05:00.0     S3VJNY0J600218       SAMSUNG MZQLB3T8HALS-000AZ               1           3.84  TB /   3.84  TB    512   B +  0 B   LBA        EDA53W0Q
IO COMMAND (a1) sqid:1 cid:254 nsid:1
INVALID OPCODE (00/01) sqid:1 cid:254 cdw0:0 sqhd:0001 p:1 m:0 dnr:0
command error: SC 1 SCT 0
NVMe Status:INVALID_OPCODE(1) Command Result:00000000
0000:02:00.0     20KO0007309011860080 Samsung SSD NVMe Test                    1           3.84  TB /   3.84  TB    512   B +  0 B   LBA        EDA53W0Q
```

6.2.3 Limited multi-process supports

kv_nvme supports "limited multi-process". In this section, we describe the meaning of limited multi-process support, principles, and situations in which errors may occur.

Unlike kernel driver, uNVMe is a user-level driver. Therefore,

- Driver is initialized when an application is executed.
- Driver is finalized, when application is terminated.

Because of these characteristics, multi-process on the same device is NOT supported. This means that, when a device is already initialized and in use, if the device is intialized again, the process that is in use first CANNOT do the IO properly.

Thus, for the support of simultaneous execution of the command through kv_nvme, which is a kind of application, commands are issued as follows:

1. First, make sure the target device is in use or not by checking whether the driver of the target device.

2-1. If the target device is in use, kv_nvme issues command via already the initialized driver

2-2. If the target device is NOT in use, kv_nvme initializes a driver of the target device. Then, kv_nvme issues command.

[NOTE] Even though kv_nvme is executed later, if the process that was executed first has not been initialized driver, it may cause a disruption to the IOs of the ealier process.

[NOTE] When users issue IO commands (write, exist, etc.) as Secondary process, if the core_mask overlaps with a primary process or another secondary process, completion processing by another process is possible to cause an ABORTED - SQ DELETION error

6.3 mkfs

mkfs application is used to prepare a BlobFS filesystem on the NVMe device to be used.

```
After the uNVMe SDk source code is built, from the home directory, navigate to app/mkfs directory.

        cd app/mkfs

Modify the `lba_sdk_config.json` to contain the BDF of the NVMe device to be used.

Execute the following command,

        ./mkfs unvme_bdev0n1 ./lba_sdk_config.json

blobFS is prepared as devices order of configuration file.
  ex) ./mkfs unvme_bdev0n1 ./lba_sdk_config.json > prepare a blobFS on "first" device of lba_sdk_config.json
  ex) ./mkfs unvme_bdev2n1 ./lba_sdk_config.json > prepare a blobFS on "third" device of lba_sdk_config.json
```

6.4 fuse

Fuse application is used to mount the BlobFS filesystem present on the NVMe device to a mount point in the system.

```
To use the Fuse application, libfuse has to be installed in the system.

To install libfuse, please follow the instructions present in the link

       https://github.com/libfuse/libfuse/

After the libfuse is built and installed, the libfuse3.so will be present at the following location in the system

   /usr/local/lib/x86_64-linux-gnu/libfuse3.so
(Please verify if the path for the libfuse3.so is the same as mentioned above. If the path is different, it has to be used accordingly)

The uNVMe SDK source code needs to be rebuilt after the libfuse installation.

After the uNVMe SDK source code is built, from the home directory, navigate to app/fuse directory.

       cd app/fuse

Modify the `dev_id` of `lba_sdk_config.json` to contain the BDF of the NVMe device to be used.

Execute the following command,

       LD_LIBRARY_PATH=/usr/local/lib/x86_64-linux-gnu/ ./fuse unvme_bdev0n1 ./lba_sdk_config.json /mnt/fuse

Fuse application will be waiting for the Ctrl+C signal on the current terminal.

In the other terminal, the contents of the /mnt/fuse can be checked by executing the following command

       ls /mnt/fuse

Note : Like mkfs, fuse also follows devices order of configuration file. Therefore, mismatch of devices' order of configurations files between mkfs and fuse may cause mount error.
Note : The Rocksdb(db_bench), Fuse, and Mkfs work for LBA SSD at the moment. (Not for Samsung KV SSD)
```

6.5 unvme_rocksdb

unvme_rocksdb is provided to make traditional rocksdb work on a LBA SSD. As a result of unvme_rocksdb compilation, db_bench will be created, which can generate IO workload on LBA SSD. unvme_rocksdb can be tested like below using script file.

6.5.1 Prerequisite

1) To build unvme_rocksdb, libfuse have to be installed. Please refer below link to install *libfuse*.
   https://github.com/libfuse/libfuse/tree/master#installation
2) format nvme device using *nvme-cli*
   Make sure the NVMe device which is to be used is properly formatted. NVMe devices can be formatted using the "**nvme**" cli utility using the Linux inbox kernel driver. To do the formatting of the devices, please execute the following commands
   cd driver/external/spdk *// From the uNVMe SDK home directory, navigate to the SPDK directory*
   ./script/setup.sh reset *// Install the Linux inbox kernel driver*
   nvme format --ses=1 /dev/nvme0n1 // Format the NVMe device (select the correct NVMe device)
3) After the NVMe device is formatted, uninstall the inbox kernel driver and setup the hugepages in the system. Hugepages have to be setup as the SDK performs DMA IO effectively with less paging overhead, taking advantage of Intel DPDK/SPDK nature. The number of hugepages should be reserved based on the application's requirements. In the example, we are reserving 4096 hugepages, which implies 8GB DRAM reservation as one hugepages would occupy 2MB in size.
   NRHUGE=4096 ./script/setup.sh

[Note] Users need to check a possible DRAM size for hugepages reservation in users' system, and properly set application's hugepage memory size with '*app_hugemem_size*' option in json configuration.

6.5.2 Build

Build mkfs & unvme_rocksdb & fuse (it is possible to build by *make.sh* script file)

./make.sh app

6.5.3 mkfs

mkfs application is used to prepare a BlobFS filesystem on the NVMe device to be used. After the uNVMe SDK source code is built, from the home directory, navigate to *app/mkfs* directory. Modify the `*lba_sdk_config.json*` to contain the BDF of the NVMe device to be used, then execute the following command

cd app/mkfs

./mkfs unvme_bdev0n1 ./lba_sdk_config.json *//./mkfs <bdevname> <config.json path>*

6.5.4 RocksDB(db_bench)

Navigate *app/unvme_rocksdb*, then modify the `*dev_id*` of `*lba_sdk_config.json*` to contain the BDF of the NVMe device to be used in. Run the script *run_test.sh* that performs RocksDB sample workload tests (Assuming the first NVMe device). The script file can be modified based on the workload requirements.

cd app/unvme_rocksdb

./run_tests.sh ./db_bench

6.5.5 Fuse

Fuse application is used to mount the BlobFS filesystem present on the NVMe device to a mount point in the system. To use the Fuse application, *libfuse* has to be installed in the system. Please follow the instructions present in the link

https://github.com/libfuse/libfuse/

After the *libfuse* is built and installed, the *libfuse3.so* will be present at the following location in the system. (Please verify if the path for the libfuse3.so is the same as mentioned below. If the path is different, it has to be used accordingly).

/usr/local/lib/x86_64-linux-gnu/libfuse3.so

After installing fuse and building SDK, navigate to *app/fuse* directory, modify the `*dev_id*` of `*lba_sdk_config.json*` to contain the BDF of the NVMe device to be used. Then execute following commands.

cd app/fuse

LD_LIBRARY_PATH=/usr/local/lib/x86_64-linux-gnu/ ./fuse unvme_bdev0n1 ./lba_sdk_config.json /mnt/fuse

Fuse application will be waiting for the *Ctrl+C* signal on the current terminal. In the other terminal, the contents of the */mnt/fuse* can be checked by executing the following command.

ls /mnt/fuse

[Note] The Rocksdb(db_bench), Fuse, and mkfs work for LBA SSD at the moment, not for Samsung KV SSD.

6.6 kv_rocksdb

kv_rocksdb has developed to make traditional rocksdb work on a KV SSD. It passes primitive IO operations such as put/get/delete to KV SSD directly, skipping most of rocksdb original behavior, so-called LSM and compaction operations. It is designed to offload data management functions on application to KV SSD. As a result of kv_rocksdb compilation, *kv_db_bench* will be created, which can generate IO workload out of db_bench on KV SSD.

```
1. description
       1)db_bench workload generator over KV SSD. (based on official rocksdb db_bench)
       2) Huge page setting is required before test. (NRHUGE=2048 ./setup.sh)
       3) Parameters
              - benchmarks: Support fillseq | fillrandom | filluniquerandom | readseq | readrandom | deleteseq | deleterandom
                     * Readrandom should be followed by fillseq or fillrandom
              - num: the number of test
              - threads: Number of I/O threads
              - histogram: provide qos (e.g. P99.99) information when it ends
              - disable_wal: disable wal behavior(Write Ahead Log, recommended)
       4) kv_sdk_sync_config.json : kv_db_bench read the config.json when it run
              - "dev_id": PCI BDF address of KV SSD (could be checked by lspci)
              - "cache": on/off kv readcache. (set it off to generate actual read workload out of a KV SSD)

2. build kv_db_bench
       cd app/kv_rocksdb
       make kv_db_bench

3. run kv_db_bench
       ./kv_db_bench -benchmarks {fillseq | fillrandom | filluniquerandom | readseq | readrandom | deleteseq | deleterandom }
-num 1000000 -key_size 16 -value_size 4096 -threads 1 -histogram -disable_wal -kv_async

       or

       run_kv_db_bench.py (Note: iterates kv_db_bench)
```

6.7 kv_perf

kv_perf is a CLI-based tool used for generating a variety of I/O workload to KV SSD directly. Below is a list of options to generate the various workload the kv_perf can process.

- *sync / async I/O*
- *write / read / delete / mixed*
- *single-threaded / multi-threaded*
- *key distribution (seq, rand, etc.)*

6.7.1 Options

Below is the list of kv_perf options.

[USAGE] *./kv_perf [OPTIONS]…*	
--write, -w	Run write(store) test
--read, -r	Run read(retrieve) test
--verify, -x	Run verify test (read and check integrities of value after write the Key-value pairs)
--delete, -e	Run delete test
--blend, -b	Run mixed workload test
--format, -z	Format device
--num_keys, -n <number of keys>	Number of keys used for test, default: 1000
--num_tests, -t <number of tests>	Number of runs for each test, default: 1
--value_size, -v <value size in bytes>	The size of values used for test, default: 4096
--workload, -p <key distribution>	Key distribution, default: 0 (0: SEQ INC, 1: SEQ DEC, 2: RAND, 3: UNIQUE RAND, 4: RANGE RAND)
--json_conf, -j <json config file path>	Device configuration file path, default: "./kv_perf_scripts/kv_perf_default_config.json"
--send_get_log, -i	Send get_log_page cmd during IO operation, default: off
--slab_size, -l <slab size in MB>	Slab memory size, default: 512(MB)
--use_cache, -u	Enable read cache, default: NONE
--device, -d <PCI addr of the device>	PCI address of the device, default: "0000:02:00.0" (you can check by " ls -l /sys/block/nvme*")
--ssd_type, -m <SSD types>	Type of the device to be tested, default: 1 (0: LBA SSD, 1: KV SSD)
--core_mask, -c <CPU core mask>	cpu cores to execute the I/O threads(e.g. FF), default: 1
--sync_mask, -s <sync mask>	I/O threads to perform sync operations(e.g. FF), default: 1
--cq_mask, -q <CQ thread mask>	CQ processing threads CPU core mask(e.g. FF), default: 2
--qd, -a <IO queue depth>	I/O queue depth, default: 64
--offset, -o <offset>	Start number from which key ID# will be generated, default: 0
--seed, -g <seed_value>	Seed for generating random keys, default: 0
--def_value, -k <key_value>	8 bytes string filled into the value, default: NULL
--read_file, -f <file name>	Path and name of the file to read the key/values, default: NULL
--key_range, -y <key_start-key_end>	Key range for blend test, default: NULL
--use_sdk_cmd, -L	Use SDK level IO cmds. If this options is not set, use low level IO cmds
--help, -h	Print help menu

Options for device initialization (e.g. device ID, ssd type, etc.) can be set from both config (json) file and command options. Format of the json file used for *kv_perf* is exactly same to what already introduced previously in of *"Programming guide"* document.

[NOTE] Options specified by the command are prior to the options by the json file.

Below is a list of sample commands.

<table>
<tr>
<td>

(sample_config.json)
```
{
  "cache": "off",
  "cache_algorithm": "radix",
  "cache_reclaim_policy" : "lru",
  "slab_size" : 512,
  "slab_alloc_policy" : "huge",
  "ssd_type" : "kv",
  "driver" : "udd",
  "log_level" : 0,
  "log_file" : "/tmp/kvsdk.log",
   "device_description" : {
       "dev_id" : "0000:02:00.0",
       "core_mask" : 1,
       "sync_mask" : 1,
       "cq_thread_mask": 2,
       "queue_depth" : 64
     }
}
```

</td>
<td>

$ ./kv_perf -n 1000 -v 4096 -w -j sample_config.json

➔ store 1000 KV pairs with 4096B of value size to the device configured at sample_config.json

$ ./kv_perf -n 1000 -v 4096 -w -r -j sample_config.json -u

➔ store 1000 KV pairs and retrieve 1000 pairs with 4096B of value size to the device configured at sample_config.json, read cache enabled (NOTE that even "cache" is "off" in json)

$. ./kv_perf -n 1000 -v 4096 2048 -w -j sample_config.json

➔ store 1000 KV pairs with number of value size, which is randomly chosen from {4096, 2048}, to the device configured at sample_config.json

$. ./kv_perf -n 100 80 30 -v 4096 -b -j sample_config.json

➔ test mixed workload("store : retrieve : delete = 100 : 80 : 30") with 4096B of value size to the device configured at sample_config.json

$. ./kv_perf -n 1000 -v 4096 -w -j sample_config.json -s 0

➔ store 1000 KV pairs with 4096B of value size to the device configured at sample_config.json, async I/O (NOTE that even "sync_mask" is "1(=sync)" in json)

</td>
</tr>
</table>

If users want to run *kv_perf* with multiple devices, there are two ways possible.

- Describing multiple devices in json file. See sample configuration at *kv_sdk_init* page of *"Programming Guide"*.
- Specifying multiple parameters for cmd options (-d, -c, -s, -q, -a). Below cmd is exactly same configuration to the sample json at kv_sdk_init page of *"Programming Guide"*. (Note that parameters for same options are separated by space)

  $./kv_perf –d 0000:01:00.0 0000:02:00.0 –c F F0 –s 0 0 –q 2 4 –a 64 256 –n 1000 –v 4096

[NOTE] Key size in *kv_perf* is fixed by 16B for now.

[NOTE] In low_cmd_mode(default), user must set num_keys and value_size regarding system's huge page size.

6.7.2 Workloads

Types of workload are generated by *kv_perf*. One is called *single job*, the other is called *multiple jobs*.

- Single jobs: Writes, reads and deletes will be done sequentially.
- Multiple jobs: Writes, reads and deletes will be done sequentially independent for every job.

e.g) **2 jobs and 10000 keys:**

Job 1:

1) Write test for 10000 keys
2) Read test for 10000 keys
3) Delete test for 10000 keys

Job 2

1) Write test for 10000 keys
2) Read test for 10000 keys
3) Delete test for 10000 keys

6.7.3 Scripts

There are five python sample scripts provided to show how to exploit kv_perf.

- *kv_perf_async_singleiq.py*
- *kv_perf_async_multiq.py*
- *kv_perf_format.py*
- *kv_perf_sync_multiq.py*
- *kv_perf_sync_singleq.py*

These scripts above are executed without arguments.

$ sudo ./kv_perf_async_multiq.py

Instead of using CLI-based options, some parameters are included in the scripts. Below are descriptions of the parameters.

- *WRITE_READ_DELETE: Write, read and delete test*
- *WRITE_READ: Write and read test*
- *ONLY_WRITE: Only write test*
- *ONLY_READ: Only read test*
- *PCI_ADDRESS: Address of SSD on PCIe bus*
- *NUM_KEYS: Number of keys to transfer*
- *SEQ_INQ: Sequential increment key distribution*
- *RAND: Random key distribution without collisions*

6.8 sdk_perf / sdk_perf_async

sdk_perf and sdk_perf_async are geared to show how to implement a real application that works on uNVMe SDK.

sdk_perf and sdk_perf_async provide the ways

- to define configuration json file and its options
- to access single/multiple uNVMe SSDs from single/multiple threads
- to map CPU cores / IO threads / uNVMe SSDs
- to show device capacity and utilization percentage
- to enable uNVMe cache
- to implement async callback function and private data
- to check miscompare by setting *check_miscompare = 1*

Additionally, test results of the store, retrieve and delete operations are shown after the end of running of sdk_perf and sdk_perf_async. For more details, please refer to below subsection 7.2.1.

6.8.1 Details of Performance Reporting

The section further explains the details of performance report message out of uNVMe BM tool with the example below.

For the first time, below figure shows the meaning of Q2C time.

kv_sdk_multi_retrieve latency: 7.953 us per operation
← This implies average Q2C (from submission to completion) time of all IO requests

Regarding the result of sdk_perf_async store operation below:

kv_sdk_multi_store start: 1511847139s.951541us end: 1511847140s.746797us
kv_sdk_multi_store total elapsed: 795256us 795.256ms
kv_sdk_multi_store latency: 7.953 us per operation
kv_sdk_multi_store ops: 125745.672
kv_sdk_multi_store throughput: 52982 KB

kv_sdk_multi_store latency QoS:

lat(usec): __min=216__, __max=7266__, __mean=2001.601__

__lat percentiles__ (usec):

| 1.00%=[2] 5.00%=[333] 10.00%=[702] 20.00%=[1148]

|30.00%=[1470] 40.00%=[1745] 50.00%=[2002] 60.00%=[2258]

|70.00%=[2533] 80.00%=[2855] 90.00%=[3301] 95.00%=[3670]

|99.00%=[4361] 99.50%=[4614] 99.90%=[5135] 99.95%=[5339]

|99.99%=[5773]

- ***latency: 7.953 us per operation*** indicates average Q2C (from submission to completion) time of all IO requests <- The latency is Q2C(from submission to completion) time, yet average of all IOs handled concurrently.
- ***min = 216*** indicates the shortest IO time of all IO requests. <- The latency is also Q2C time, yet the fastest individual IO
- ***max = 7266*** indicates the longest IO time of all IO requests. <- The latency is also Q2C time, yet the slowest individual IO
- ***mean = 2001*** indicates the average IO time of all IO requests. <- The latency is also Q2C time, yet the average time of individual IO.
- **lat percentiles =** *indicates latency QoS (1~99.99%)*

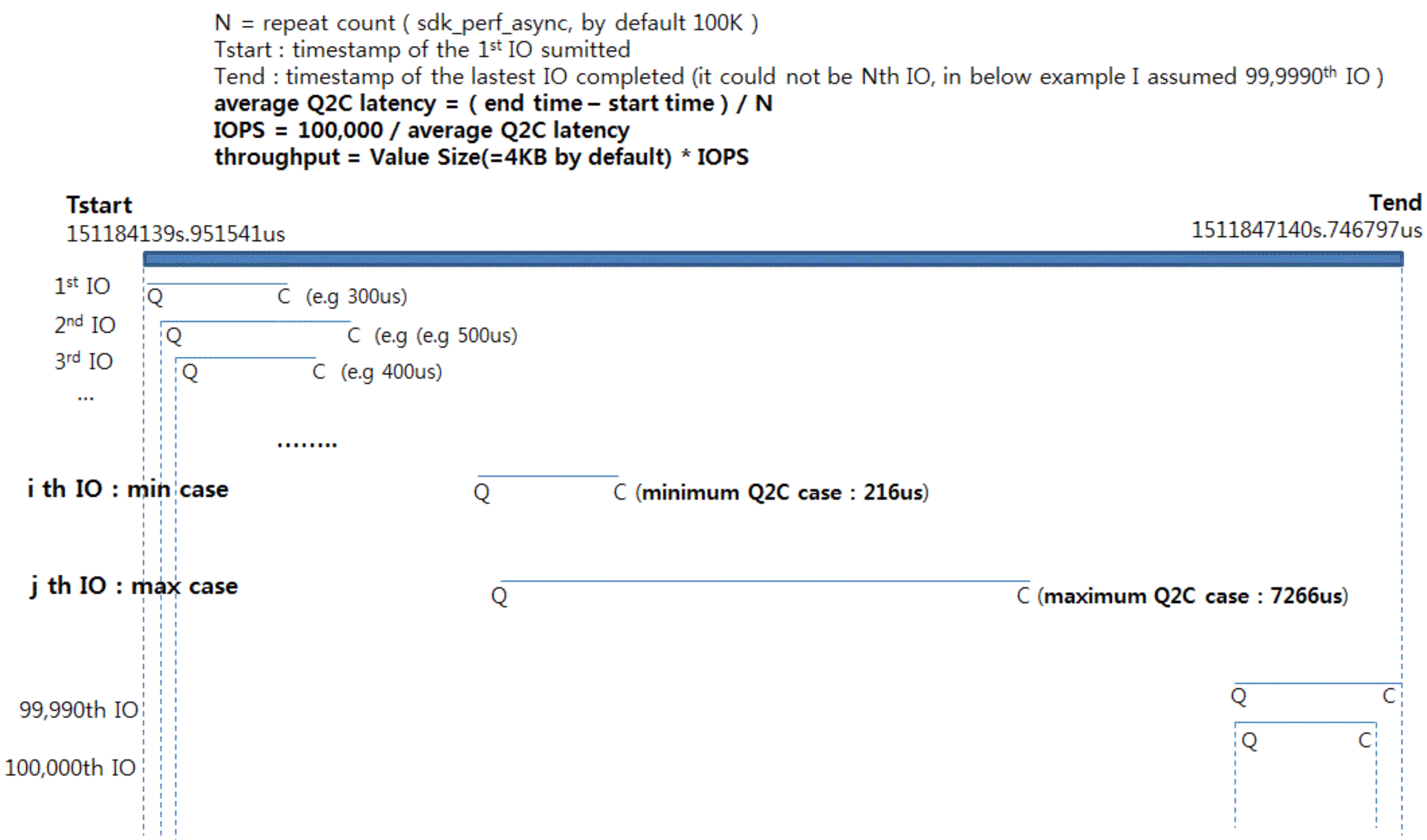


Figure 14. Result analysis of sdk_perf (sdk_perf_async)

6.9 sdk_iterate / sdk_iterate_async

sdk_iterate and sdk_iterate_async are basically same with sdk_perf and sdk_perf_async, except including iterate operations(i.e. *kv_iterate_open, kv_iterate_read(_async), and kv_iterate_close*).

sdk_iterate and sdk_iterate_async provide the ways

- to open / close iterator handle with *bitmask* and *prefix*. For more details, see *the list of kv_iterate APIs* of *"Programming Guide".*
- to get information of iterator opened. See *kv_iterate_info* of *"Programming Guide"*
- to read(get) keys by specific prefix. For more details, see *kv_iterate_read* and *kv_iterate_read_async*. (in *"Programming Guide")*

7 APPENDIX

7.1 Multi Process Support

Each process creates dedicated shared memory region and reserves per-process memory (32MB per a SSD by default). Therefore, it is allowed for processes using uNVMe SDK to access to its dedicated SSDs.

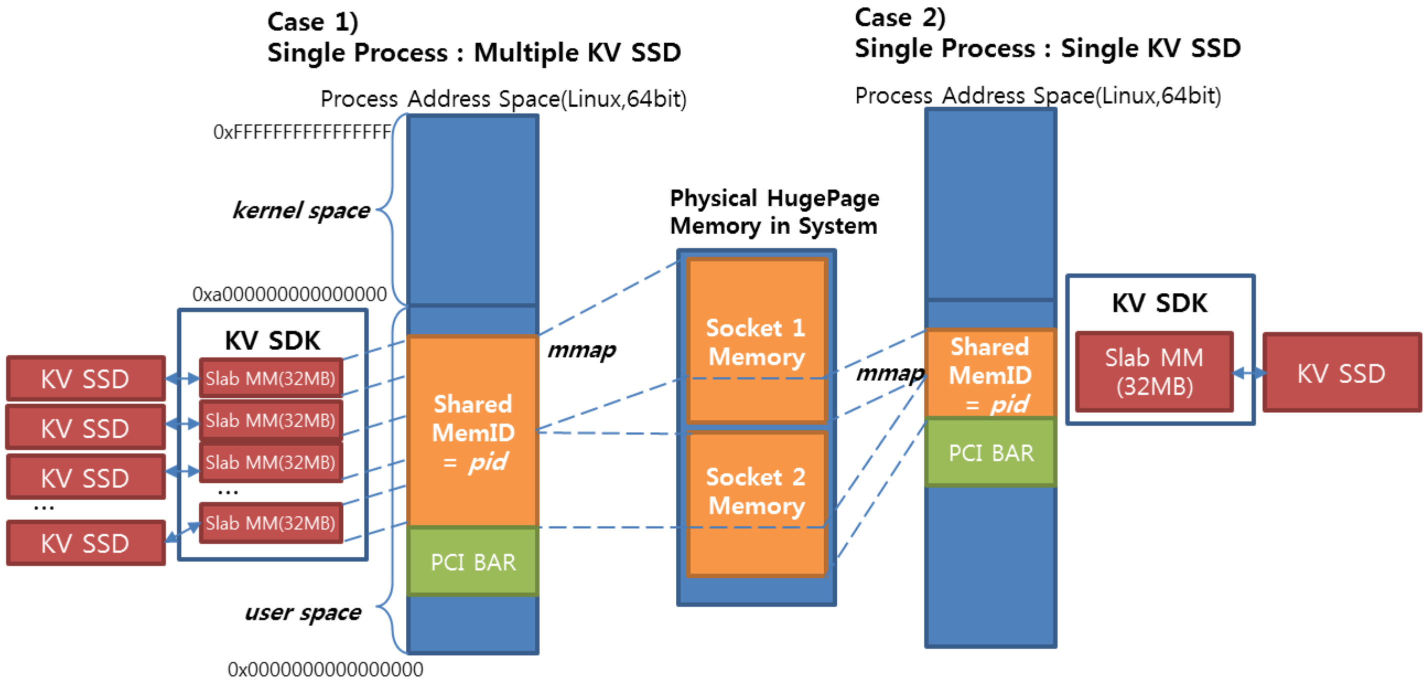


Figure 15. Use case of multi process

7.1.1 Limitation

Having access on a single uNVMe SSD from multiple processes is not allowed

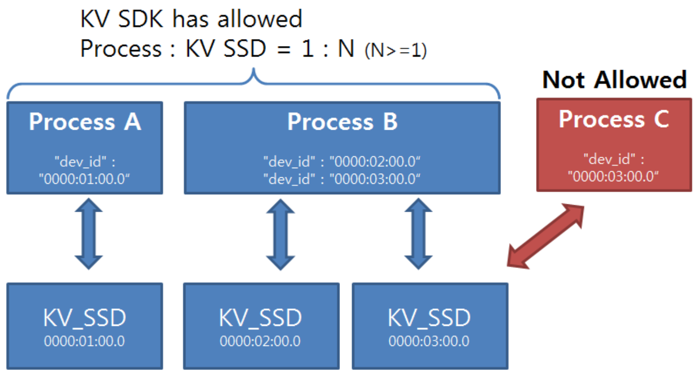


Figure 16. Limited multi process use

7.2 uNVMe SDK Configuration

Below is a sample configuration introduced at *kv_sdk_init*. This page provides more details about each configuration parameters, including device descriptions.

```
{
  "cache": "off",
  "cache_algorithm": "radix",
  "cache_reclaim_policy" : "lru",
  "slab_size" : 512,
  "app_hugemem_size" : 1024,
  "slab_alloc_policy" : "huge",
  “ssd_type” : “kv”
  "log_level" : 0,
  "log_file" : "/tmp/kvsdk.log",
  …….
```

- *cache* : uNVMe Read Cache ON / OFF
- *slab_size*: the total amount of Slab size in MB
- app_hugemem_size: the size of additional hugepage memory set by user application in MB

  Please note that before changing *slab_size* and *app_hugemem_size* in the json config file, be sure that available huge page memory in your system should be larger than *slab_size + app_hugemem_size*. The granularity is MB in size. You can check the huge page memory size with following commands.

  # cd {path}/{to}/{uNVMe SDK}/script

  # NRHUGE=2048 ./setup.sh

  The 2048 is the number of huge page request to allocate. As a huge page is 2MB in size, the command is attempting to allocate 4GB huge page memory:

  # cat /proc/meminfo | grep HugePages_Total

  This shows the maximum size of available huge page in your system. It varies depending on the total amount of DRAM in your system. In many case it would larger than 1GB at least.
- *ssd_type*: “kv” means KV-type SSD(s). Set *ssd_type* to *“lba” if users want to enable LBA SSD(s).*
- *log_level* : remain SDK footprint at *log_file*. There are 4 levels supported.

  level 0: no log

  level 1: + logging error

  level 2: + logging SDK initialize / finalize information

  level 3: + logging I/O (store/retrieve/delete) operations

  Please be sure that any kinds of errors are printed by *fprintf(stderr, “error…”)*.
- *log_file* : the path of log file (by default /tmp/kvsdk.log)

Below figure provides more details about device description in json configuration file.

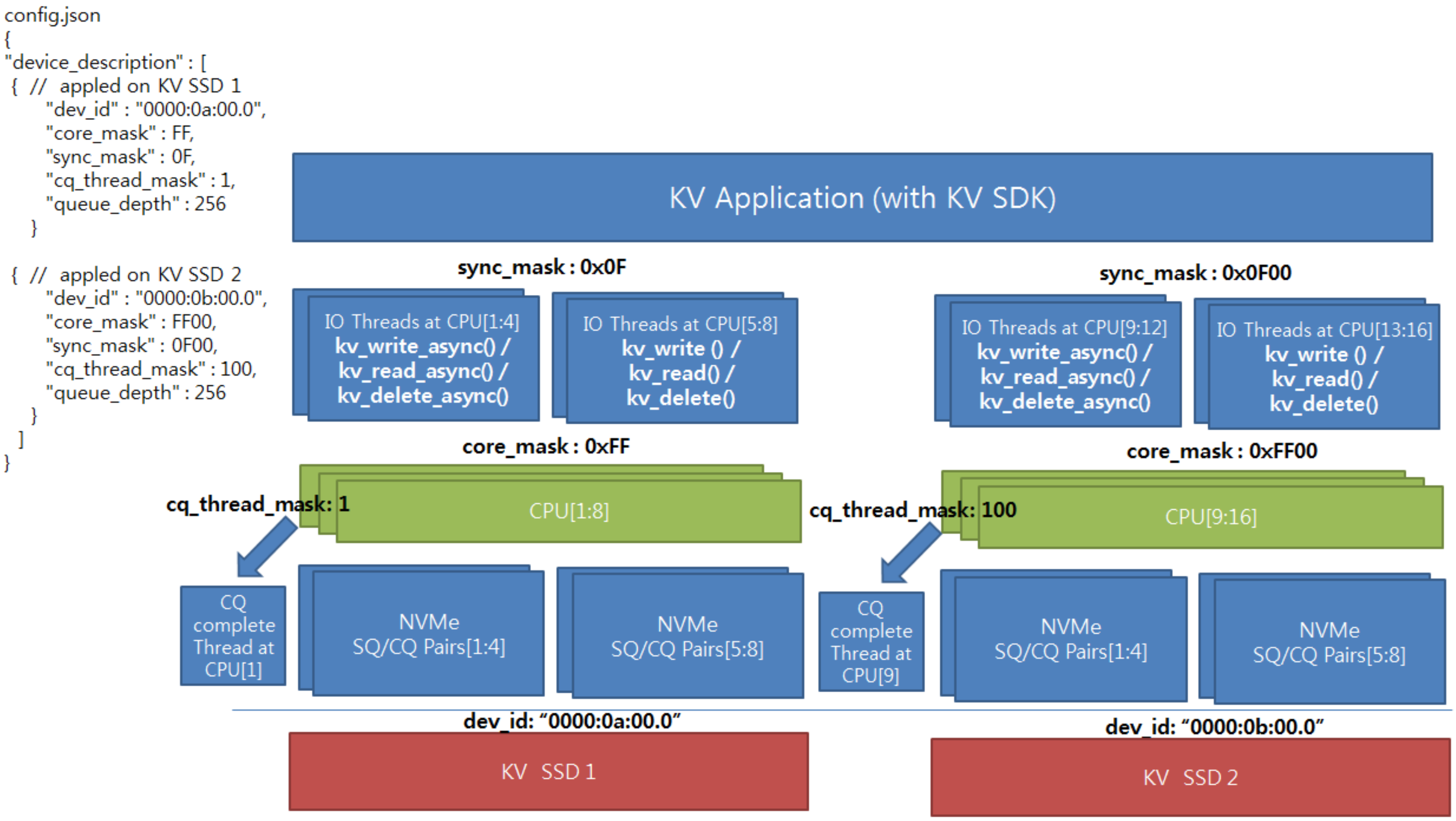


Figure 17. two KV SSDs' Description and internal mapping

The Figure above depicts the correlation of dev_id, core_mask, sync_mask, and cq_thread_mask when "device_description".

- *dev_id*: the unique PCI Device Address of a SSD. Can be checked from 'kv_nvme list' command
- *core_mask*: implies *the number and location of Host cores allowed to issue IO on a SSD.*
- *sync_mask*: configure the IO mode of submission Q as sync (=1) or async mode (=0). On sync mode, an NVMe IO which is being summited will not be returned until it gets CQ entry from SSD and finalizes the summited IO (so-called polling mode.) In contrast, on async mode, the summited IO will return instantly just after queueing the NVMe IO into submission queue. A CQ handling thread will carry out the completion process and it will call an async IO handler of the application registered. It is directly mapped with core_mask bity as 1:1
- *cq_thread_mask*: configure the number and location of Host cores that handle async IO completion of the SSD. This can resolve CQ completion overhead from tons of NVMe IOs stream depending on the CPU capability of system. It is recommended to locate CQ_thread_mask differing from core_mask not to lead to contention of a certain core.
- *queue_depth*: the maximum length of Submission queue that device driver sees. This is applied only when the IO Queue is configured as async mode (sync_mask=0) and will lead to a higher IO performance due to concurrency in SSD.

In addition, it is allowed for a thread to access on multiple SSDs. The two figures below complement an example of config.json to have access on multiple SSDs from a single IO thread and how an application have to be implemented.

config.json

```
{
"device_description" : [
 {  //  appled on KV SSD 1
       "dev_id" : "0000:0a:00.0",
       "core_mask" : 00FF,
       "sync_mask" : 000F,
       "cq_thread_mask" : 0001,
       "queue_depth" : 256
   }

   {  //  appled on KV SSD 2
       "dev_id" : "0000:0b:00.0",
       "core_mask" : 0FF0,
       "sync_mask" : 00F0,
       "cq_thread_mask" : 0010,
       "queue_depth" : 256
   }

   {  //  appled on KV SSD 3
       "dev_id" : "0000:0c:00.0",
       "core_mask" : 7F80,
       "sync_mask" : 0780,
       "cq_thread_mask" : 100,
       "queue_depth" : 256
   }
  ]
}
```

- **core_mask** implies ***the number and location of Host cores allowed to issue IO on a KV SSD***
  - On the left config.json,
  - Threads on Core[0:7], core[4:11], and core[7:15] are having access on "0000:0a:00.0", "0000:0b:00.0", and "0000:0c:00.0" KV SSD, respectively
  - Threads on Core[4:6] are having access on both of KV SSD "0000:0a:00.0" and "0000:0b:00.0"
  - Only one thread on core[7] is having access on the three multiple KV SSD, "0000:0a:00.0", "0000:0b:00.0", and "0000:0c:00.0"
- **sync_mask** implies Sync or Async property of an NVMe IO Queue on a KV SSD, **directly mapping with core_mask bit as 1:1**
  - Threads on core[4:6] have to do async IO on KV SSD "0000:0a:00.0", while do sync IO on KV SSD "0000:0b:00.0"
- **cq_thread_mask** implies ***the number and location of Host cores that handle async IO completions***
  - There is one thread on Core[1], Core[4], and Core[8] to complete Async IO of "0000:0a:00.0", "0000:0b:00.0", and "0000:0c:00.0" KV SSD, respectively

< mapping status with the left config.json >

KV SSD	Config.json	Host Core ID															
		15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
"dev_id" : "0000:0a:00.0"	"core_mask" : "0x00FF"	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
	"sync_mask" : "0x000F"	-	-	-	-	-	-	-	-	0	0	0	0	1	1	1	1
	"cq_thread_mask":"0x0001"																1
"dev_id" : "0000:0b:00.0"	"core_mask" : "0x0FF0"	0	0	0	0	1	1	1	1	1	1	1	1	0	0	0	0
	"sync_mask" : "0x00F0"	-	-	-	-	0	0	0	0	1	1	1	1	-	-	-	-
	"cq_thread_mask":"0x0010"												1				
"dev_id" : "0000:0c:00.0"	"core_mask" : "0xFF80"	0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0
	"sync_mask" : "0x0F80"	-	0	0	0	0	1	1	1	1	-	-	-	-	-	-	-
	"cq_thread_mask":"0x0100"								1								

Figure 18. Configuration example – accessing multiple devices from single I/O thread (1)

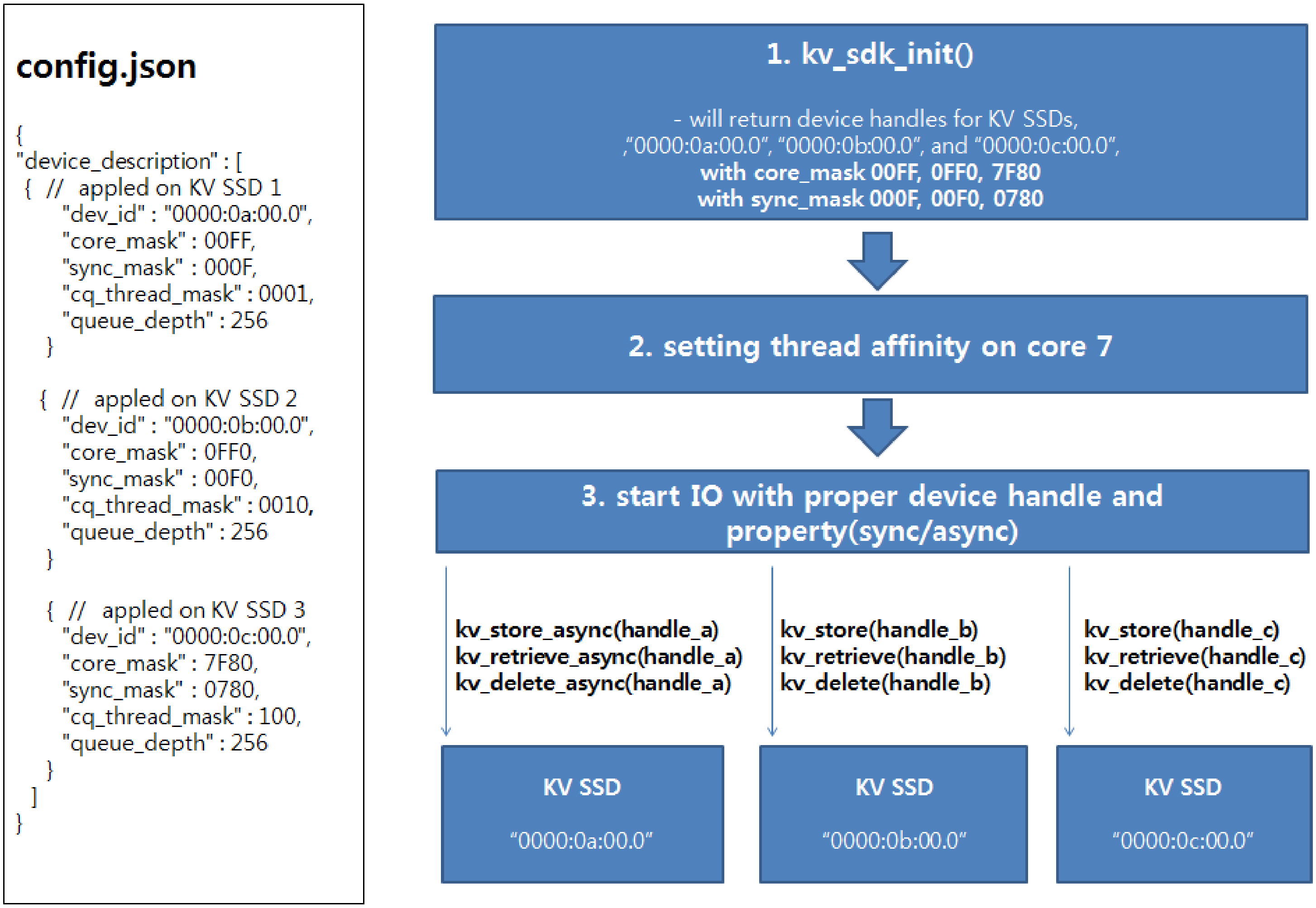
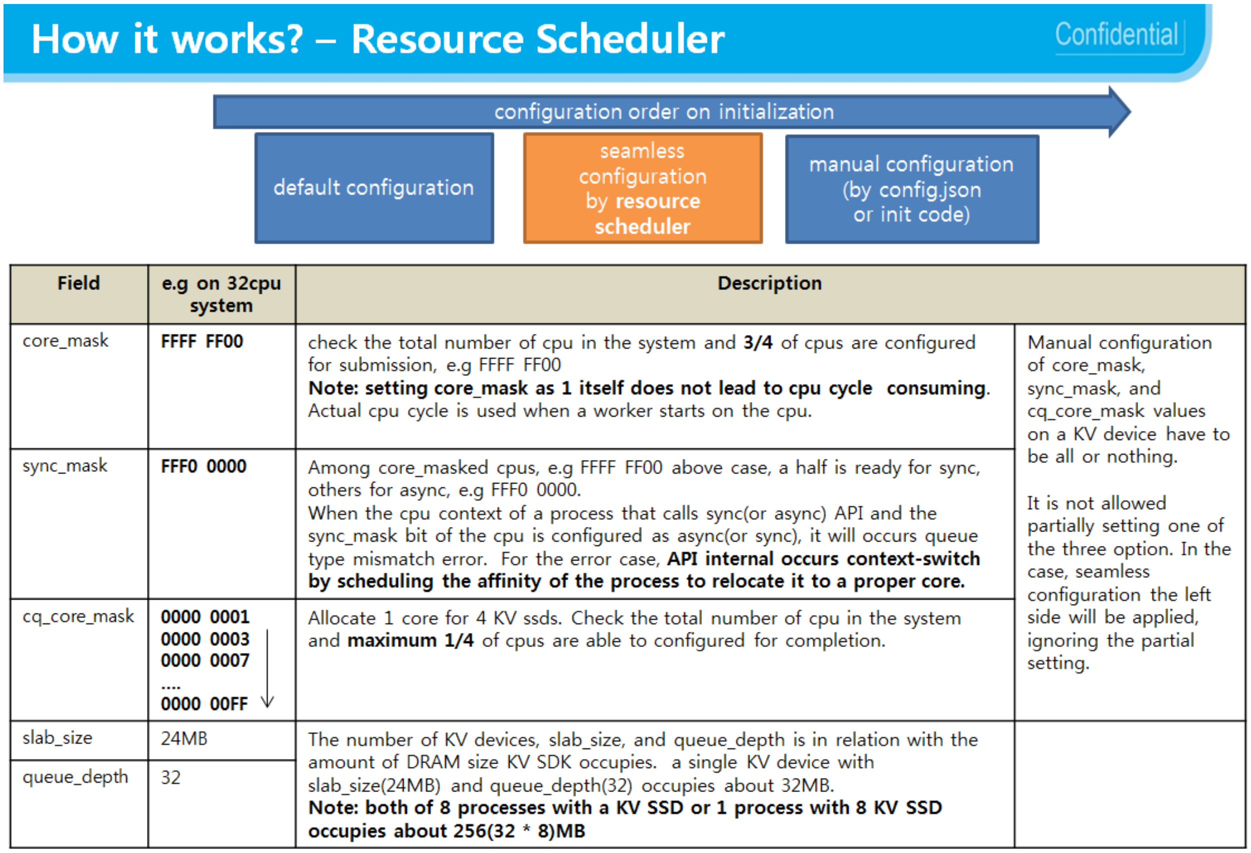


Figure 19. Diagram for description in Figure 18

Howver, without the understanding of above configuration, users can process IO through SSD easily. Because uNVMe SDK provides seamless configuration and seamless core affinity scheduling by *Resource scheduler*. Figures above describe how resource scheduler works.



Field	e.g on 32cpu system	Description	
core_mask	**FFFF FF00**	check the total number of cpu in the system and **3/4** of cpus are configured for submission, e.g FFFF FF00 **Note: setting core_mask as 1 itself does not lead to cpu cycle consuming.** Actual cpu cycle is used when a worker starts on the cpu.	Manual configuration of core_mask, sync_mask, and cq_core_mask values on a KV device have to be all or nothing. It is not allowed partially setting one of the three option. In the case, seamless configuration the left side will be applied, ignoring the partial setting.
sync_mask	**FFF0 0000**	Among core_masked cpus, e.g FFFF FF00 above case, a half is ready for sync, others for async, e.g FFF0 0000. When the cpu context of a process that calls sync(or async) API and the sync_mask bit of the cpu is configured as async(or sync), it will occurs queue type mismatch error. For the error case, **API internal occurs context-switch by scheduling the affinity of the process to relocate it to a proper core.**	
cq_core_mask	**0000 0001** **0000 0003** **0000 0007** **....** **0000 00FF**	Allocate 1 core for 4 KV ssds. Check the total number of cpu in the system and **maximum 1/4** of cpus are able to configured for completion.	
slab_size	24MB	The number of KV devices, slab_size, and queue_depth is in relation with the amount of DRAM size KV SDK occupies. a single KV device with slab_size(24MB) and queue_depth(32) occupies about 32MB. **Note: both of 8 processes with a KV SSD or 1 process with 8 KV SSD occupies about 256(32 * 8)MB**	
queue_depth	32		

Figure 20. Resource scheduler description

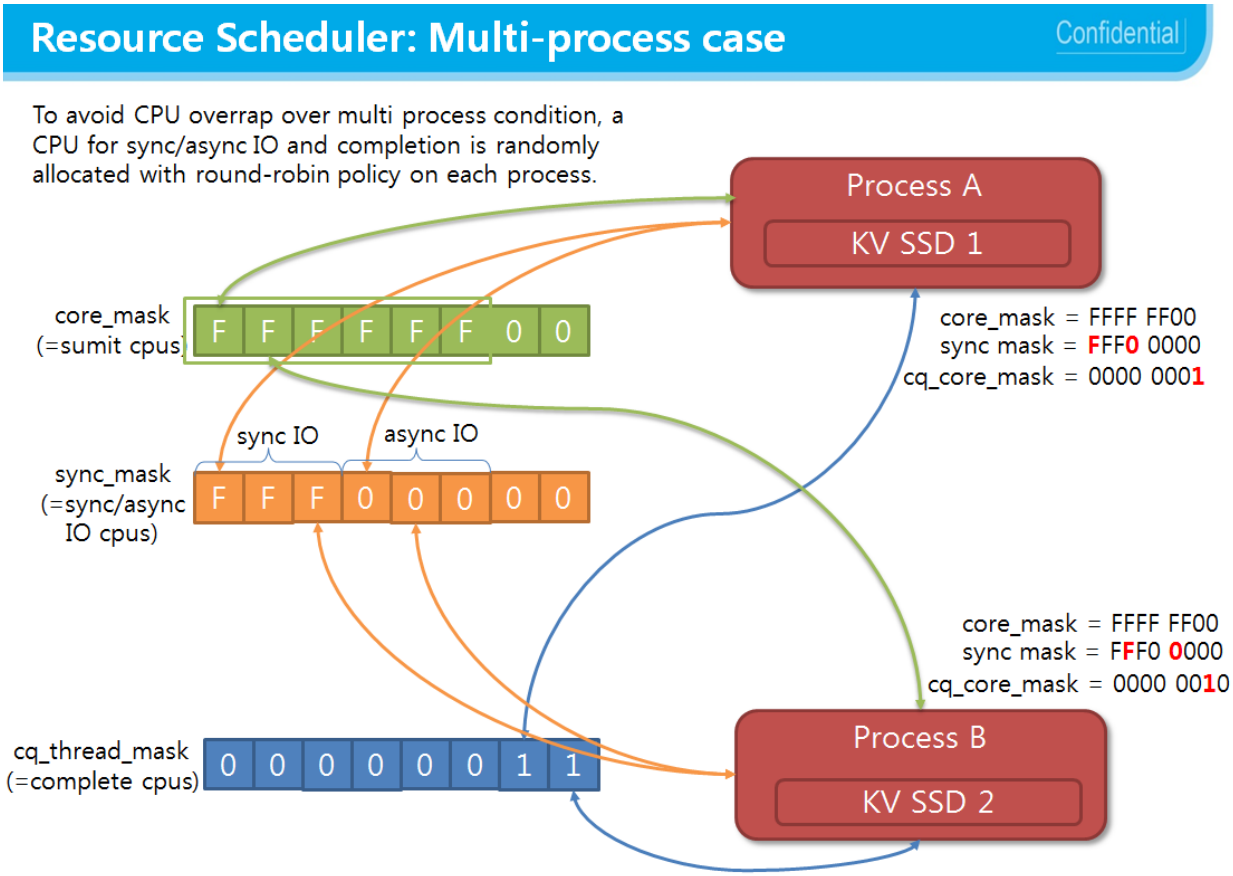


Figure 21. Resource scheduling on multi process

7.3 uNVMe App Life Cycle

An application making use of uNVMe SDK has to call APIs in the following order. Please also refer to *simple_ut* or *sdk_perf* (fine tuning example) source code implementation.

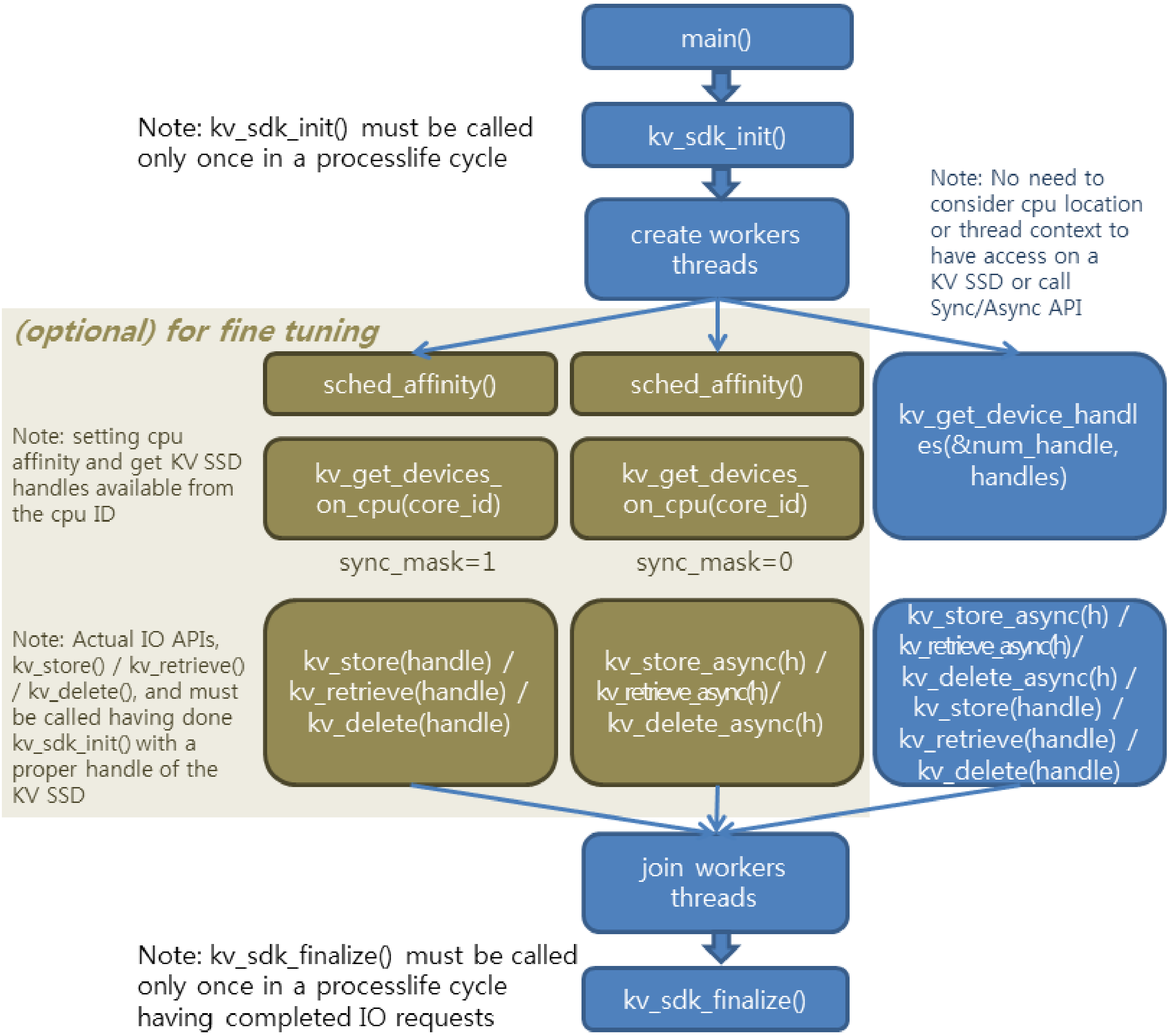


Figure 22. uNVMe App life cycle

7.4 uNVMe Cache

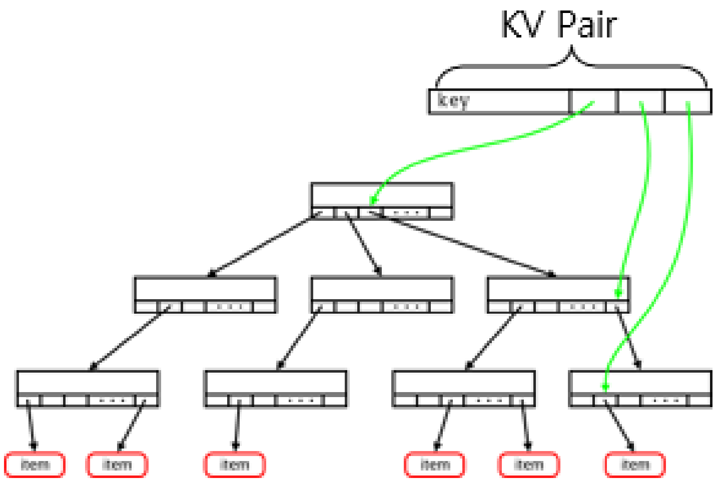


Figure 23. uNVMe cache (radix tree based)

uNVMe SDK provides its own cache implementation for better read performance. Key-value (pair) entries are managed by radix tree, a kind of AVL tree.

Cache update: there are two cases invoking cache update.

- When users call *kv_store()* for specific key and value, this pair will be loaded on the cache after storing the data to SSD. (write-through operation)
- When users call *kv_retrieve()* for specific key, and if the key entry is not in the cache(cache miss), the KV pair will be loaded on the cache after retrieving the data from SSD.

Cache eviction (reclaim): entries cached are evicted when

- Users call *kv_delete()* for specific key. The API will remove KV entry both from SSD and cache.
- The memory allocated for cache is full. Slab M/M evicts cache entries based on LRU policy for now.

Users can enable or disable the cache, also can specify how much memory will be allocated for the cache easily. Please refer to [sdk_perf] for more details about configuring and using the cache.

7.5 uNVMe Slab M/M

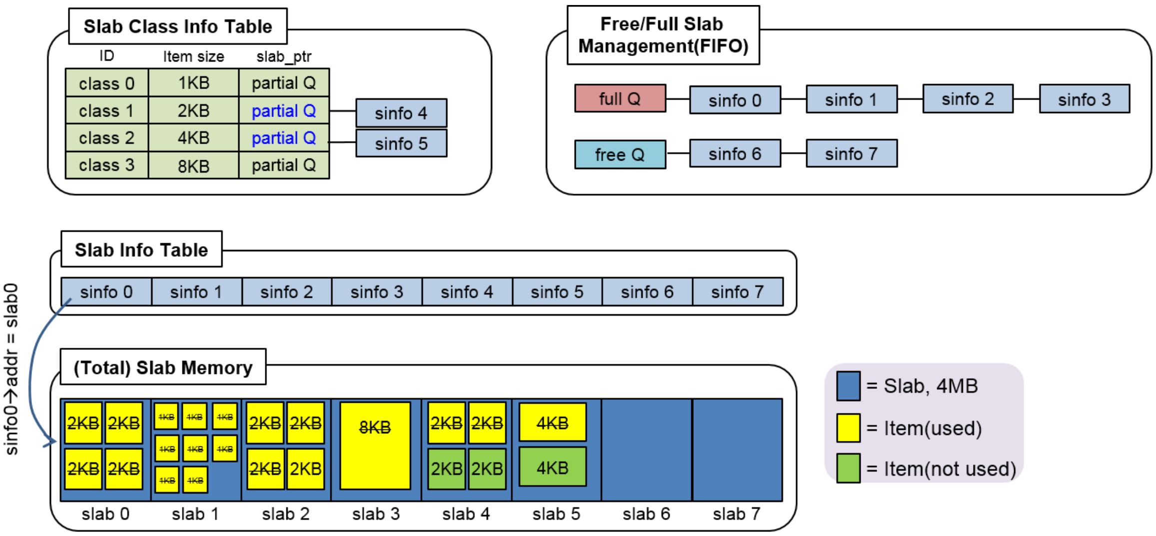


Figure 24. Slab memory management Diagram

To store and retrieve KV pair to SSD through uNVMe SDK, hugepage memory managed by slab allocator is used. Slab allocator is implemented like Figure above.

Slab memory allocation:

- When users call *kv_store()* or *kv_retrieve()*, the APIs request specific amount of hugepage memory to the allocator. In this case, the memory is used like I/O buffer.
- (If cache is enabled) To reduce memory resources and operation time, buffer above is also (re)used for cache entries. I/O buffer that contains KV data is updated directly to cache tree after the end of each I/O operations.

Slab memory collection (reclaim):

- Allocator collects and reclaims the hugepage memory when all the memory (whose size is same with *slab_size* in *kv_sdk*, see kv_sdk_init of *"Programming Guide"*) is exhausted based on LRU policy. One reclaim operation can collect 4MB of hugepage approximately for now.
- If the 'victim' memory space is used for cache entries already, allocator will evict the entries from cache during reclaim operations.

[NOTE] Slab size (set by *kv_sdk_init()*) cannot excess total size of hugepages set in the system.

7.6 Performance Guideline

7.6.1 Configuration Guide

IOs of uNVMe SDK are processed by both submission queues and completion queues pinned to specific cores. In this section, we guide how to pin queues to cores for avoiding decline of performance (IOPS avg.) with optimal number of CPUs. (That is, we guide how to configure 'core_mask' and 'cq_thread_mask' to perform 100% with least number of cores)

To use NVMe SSDs with uNVMe SDK on optimal number of cores without performance drop,

First, users have to know that how many devices can operate on single core without performance drop in users' system.

Second, apply the optimal number of devices in first step to users' uNVMe SDK configuration

For your understanding, we give an example of above process.

[First step] In our test bed, the optimal number of devices on single CPU is as follow:

1. **For the throughput(IOPS) point of view**

   Number of devices for submission queues: 4 devices.

   Number of devices for completion queues: 4 devices.

2. **For the read latency QoS (QD8) point of view**

   Number of devices for submission queues: 2 devices.

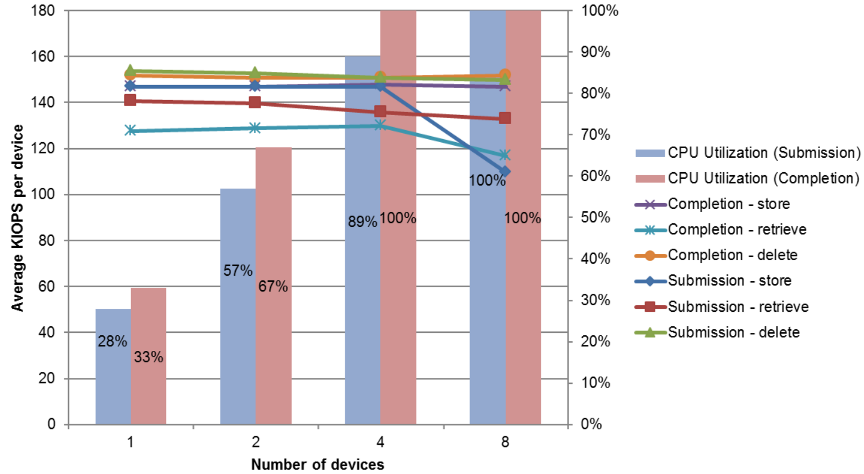
   Number of devices for completion queues: 8 devices.

To determine the optimal numbers, we measure average KIOPS per devices and latency QoS (P99.99 and max latency @ QD8) varying number of devices on single CPU. Number of devices on single CPU for submission / completion has impact on performance as follows:

CPU Utilization at IO Submission / Completion

Confidential

- **In our testbed, there was NO decrease in performance with one CPU on 1 ~ 4 devices**



- *Supermicro SuperServer 1028U-TN10RT+, Xeon(R) CPU E5-2667 v3, 32cores @ 3.20GHz, 256GB DDR4, Debian GNU/Linux 9(4.9.0-3-amd64)*
- *sdk_perf_async, slab_size=512MB, insert_count=100K, value_size=4KB, key_length=16B queue_depth=256*
- *Each device has only one submission thread which is pinned to core exclusively*

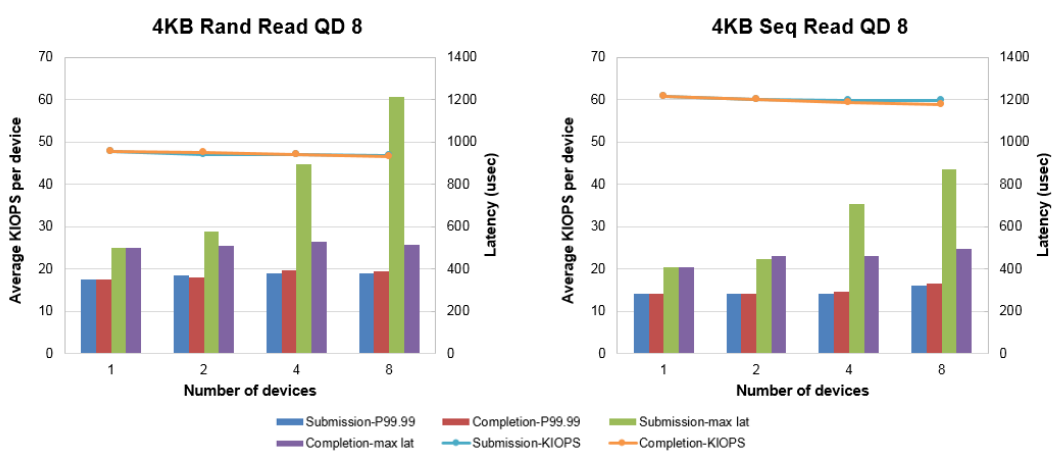
SAMSUNG 6/35 THE NEXT CREATION STARTS HERE

Figure 25. Number of devices (on single core)' impact on performance

CPU Utilization at IO Submission / Completion

Confidential

- **For read latency(4KB QD8) point of view, 8 devices per 1 completion core / 2 devices per 1 submission core shows best result.**



- *Supermicro SuperServer 1028U-TN10RT+, Xeon(R) CPU E5-2667 v3, 32cores @ 3.20GHz, 256GB DDR4, Debian GNU/Linux 9(4.9.0-3-amd64)*
- *fio-2.18 with unvme2_fio_plugin, runtime=30s, bs=4KB, key_length=16B iodepth=8 (F/W: EHA50K0I)*
- *Each device has only one submission thread which is pinned to core exclusively*

SAMSUNG 7/35 THE NEXT CREATION STARTS HERE

Figure 26. Number of devices (on single core)' impact on QD8 read latency QoS (P99.99, max)

(Figure 25) Aspect of IO submission, there is no decrease in KIOPS with one CPU on 1 ~ 4 devices. Also, aspect of IO completion, there is no decrease in KIOPS with one CPU on 1 ~ 4 devices.

(Figure 26) Aspect of IO submission, max read latency increased sharply with one CPU on 4 ~ 8 devices. Aspect of IO completion, however, there is no prominent increase in max latency with one CPU on 1 ~ 8 devices. Note that throughput performances (KIOPS) are almost same among the all cases.

[NOTE] The optimal number of devices on single core depends on performance of the single core of user's processor(s). Therefore, we recommend that users of uNVMe SDK use own number.

[Second step] If users determine the optimal number, then the number should be applied to users' configuration. Below is an example of configuration for using optimal number of cores in our test bed for the throughput point of view. In our test bed, there are 8 devices (0000:01:00.0 ~ 0000:08:00.0).

Core 0 is used for IO submission by 4 devices (0000:01:00.0 ~ 0000:04:00.0).

Core 1 is used for IO completion by 4 devices (0000:01:00.0 ~ 0000:04:00.0).

Core 2 is used for IO submission by 4 devices (0000:05:00.0 ~ 0000:08:00.0).

Core 3 is used for IO completion by 4 devices (0000:05:00.0 ~ 0000:08:00.0).

Total 4 cores, the least number of cores for 8 devices in our test bed, are used.

```
{
  "cache": "off",
  "cache_algorithm": "radix",
  "cache_reclaim_policy" : "lru",
  "slab_size" : 512,
  "slab_alloc_policy" : "huge",
  "ssd_type" : "kv",
  "log_level" : 0,
  "log_file" : "/tmp/kvsdk.log",
    "device_description" : [
      {
        "dev_id" : "0000:01:00.0",
        "core_mask" : 1,
        "sync_mask" : 0,
        "cq_thread_mask" : 2,
        "queue_depth" : 256
      },
      {
        "dev_id" : "0000:02:00.0",
        "core_mask" : 1,
      {
        "dev_id" : "0000:05:00.0",
        "core_mask" : 4,
        "sync_mask" : 0,
        "cq_thread_mask" : 8,
        "queue_depth" : 256
      },
      {
        "dev_id" : "0000:06:00.0",
        "core_mask" : 4,
        "sync_mask" : 0,
        "cq_thread_mask" : 8,
        "queue_depth" : 256
      },
      {
        "dev_id" : "0000:07:00.0",
        "core_mask" : 4,
        "sync_mask" : 0,
        "cq_thread_mask" : 8,
        "queue_depth" : 256
```

"sync_mask" : 0, ***"cq_thread_mask" : 2,*** *"queue_depth" : 256* *},* *{* *"dev_id" : "0000:03:00.0",* ***"core_mask" : 1,*** *"sync_mask" : 0,* ***"cq_thread_mask" : 2,*** *"queue_depth" : 256* *},* *{* *"dev_id" : "0000:04:00.0",* ***"core_mask" : 1,*** *"sync_mask" : 0,* ***"cq_thread_mask" : 2,*** *"queue_depth" : 256* *},*	*},* *{* *"dev_id" : "0000:08:00.0",* ***"core_mask" : 4,*** *"sync_mask" : 0,* ***"cq_thread_mask" : 8,*** *"queue_depth" : 256* *}* *]* *}*

[NOTE] If number of devices is over 4 (either submission or completion, in our test bed), QoS latency may be larger than expected on multi-processor because of NUMA architecture.

[NOTE] Using a same core for submission and completion may causes decline in uNVMe SDK and SSD performance.

7.6.2 Performance Comparison among Different Drivers

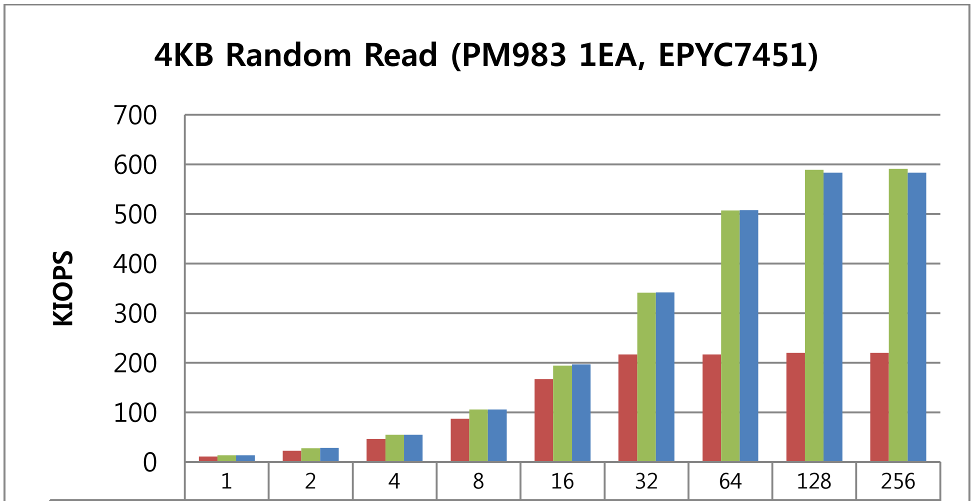
In this section, uNVMe SDK's performance advantage is showed by comparing the performance of three user level nvme drivers (SPDK, Kernel NVMe, and uNVMe SDK) measured by fio. Performance metric used for performance comparison are 4k random read.

7.6.2.1 Environment

- SSD - PM983 3.8TB (EDA53W0Q_20180315)
- Server - SMC UunvS2023-TR4
- CPU - EPYC 7451(24c) 2P
- OS - CentOS 7.4 (Kernel 3.10)
- fio version – 3.3
- performance metric – 4KB random read (KIOPS)

7.6.2.2 Core : Device = 1 : 1

The performance of random read is measured by using 4 KB unit reads for 30 seconds in a full device. At this time, there is no overlapped block addresses. The performance index uses number of IO operations per second and the unit is KIOPS. The performance measurement results are described as Figure below. Each axis on the graph represents iodpeth and KIOPS.



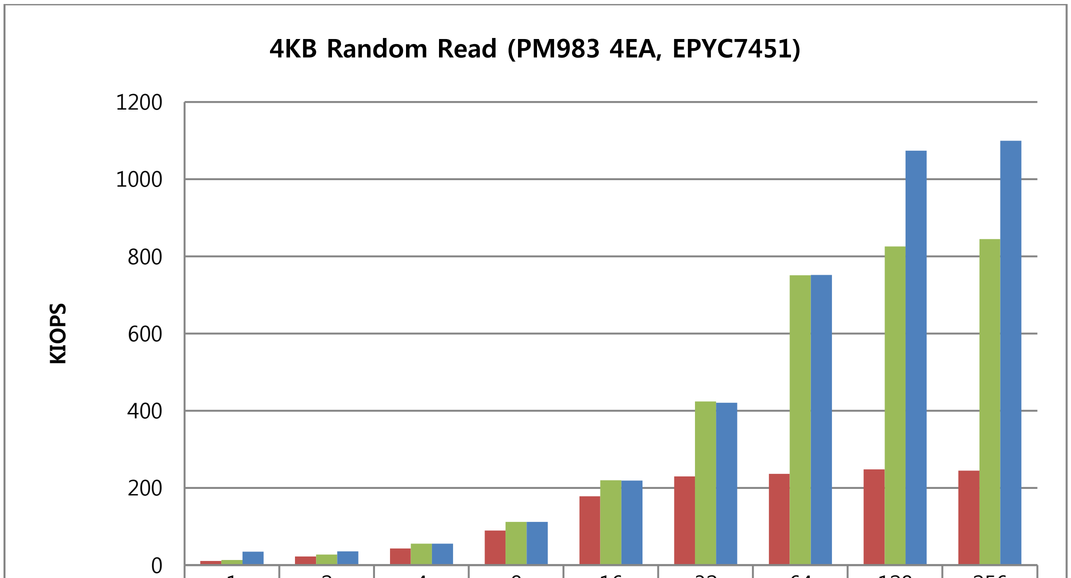
	1	2	4	8	16	32	64	128	256
Kernel	11	22.3	46.2	87	167	217	217	220	220
SPDK	13.8	27.9	54.9	106	194	341	507	589	591
uNVMe2.0	13.8	28	55	106	197	342	508	583	583

Figure 27. Performance comparison – Core : Device = 1:1

As a result of the performance measurement, two user-level drivers showed almost equal performance on whole iodepth (1 ~ 256). The performance of the user-level drivers is better than of kernel NVMe driver.

7.6.2.3 Core : Device = 1 : N

To compare performance aspects of scalability, we measure performance of IOs which are processed on multi-device using only one core. The performance is measured by using 4 KB unit reads for 30 seconds in 4 devices which is fully filled. At this time, there is no overlapped block addresses. The performance index uses number of IO operations per second and the unit is KIOPS. The performance measurement results are described as Figure above. Each axis on the graph represents IO depth and KIOPS.



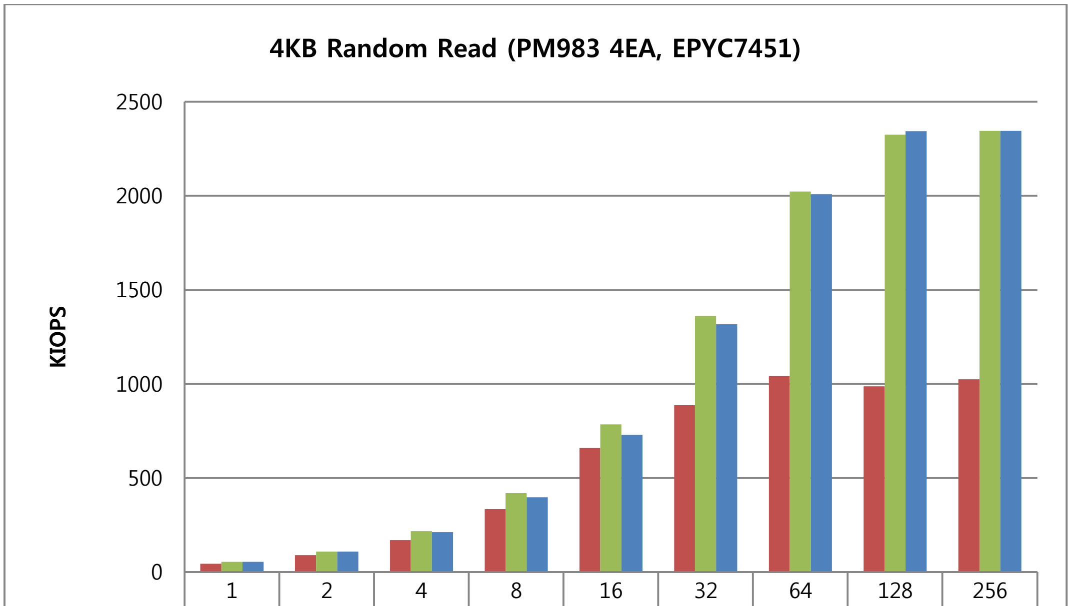
	1	2	4	8	16	32	64	128	256
Kernel	11.4	22.6	43.6	90.3	179	230	237	248	245
SPDK	13.8	27.6	55.5	112	220	424	751	826	845
uNVMe2.0	35	35.7	55.6	112	219	421	752	1074	1100

Figure 28. Performance comparison – Core : Device = 1:4

Like previous result, the performance of the user-level drivers is better than of kernel NVMe driver on whole IO depth. Also, two user-level drivers showed almost equal performance on Low to Middle IO depth (1 ~ 64). However, ***uNVMe SDK showed much better performance*** than of SPDK on high IO depth.

7.6.2.4 Core : Device = N : N

To compare performance on multi-core environment, performance is measured on core : device = 4 : 4 environment. Each core is dedicated to one device by configuration. The performance is measured by using 4 KB unit reads for 30 seconds in 4 devices which is fully filled. At this time, there is no overlapped block addresses. The performance index uses number of IO operations per second and the unit is KIOPS. The performance measurement results are described as Figure below. Each axis on the graph represents IO depth and KIOPS.



	1	2	4	8	16	32	64	128	256
Kernel	44	90.4	171	336	660	888	1043	987	1025
SPDK	55.2	110	218	420	785	1361	2023	2325	2345
uNVMe2.0	55.4	110	213	399	730	1317	2009	2344	2345

Figure 29. Performance comparison – Core : Device = 4 : 4

Like 1:1 case, two user-level drivers showed almost equal performance on whole iodepth (1 ~ 256). The performance of the user-level drivers is better than of kernel NVMe driver especially in high iodepth.

7.7 Known Issues

7.7.1 Failure on uNVMe SDK Initialization due to tmpfs full

uNVMe creates two files (.spdk*_config) on /var/run or /run to maintain mapping information of hugepage memory when initialized via kv_sdk_init (). These files are removed when the kv_sdk_finalize() is called. However, when a process is terminated abnormally without calling kv_sdk_finalize(), the files are NOT removed and remains at /var/run (or /run), occupying a few MB in size , which it could cause the full of tmpfs in the end. When reach up to the status, uNVMe SDK fail to initialize reporting "Bus Error". (Figures below)

```
root@wan:~# df -m
Filesystem     1M-blocks  Used Available Use% Mounted on
udev               16045     0     16045   0% /dev
tmpfs               3214  3214         0 100% /run
/dev/sda2         447878 28616    396489   7% /
tmpfs              16067     1     16066   1% /dev/shm
tmpfs                  5     1         5   1% /run/lock
tmpfs              16067     0     16067   0% /sys/fs/cgroup
/dev/sda1            511     4       508   1% /boot/efi
tmpfs               3214     1      3214   1% /run/user/1000
tmpfs               3214     0      3214   0% /run/user/0
root@wan:~#
```

Figure 30. Check whether tmpfs is full (e.g. mounted at /run)

```
root@wan:~/work/KV_Host_Release/tests/udd_perf# ./udd_perf
udd_perf start
hash_func: none
EAL: Detected 8 lcore(s)
EAL: Auto-detected process type: PRIMARY
EAL: No free hugepages reported in hugepages-1048576kB
Bus error (core dumped)
root@wan:~/work/KV_Host_Release/tests/udd_perf#
```

Figure 31. Initializaion failure with "Bus Error"

To maintain available space on /var/run (or /run), users may remove the files manually like below commands.

```
$> cd /var/run
$> sudo rm .spdk*_config
$> sudo rm .spdk*_hugepage_info
```

7.7.2 db_bench (rocksdb) malfunction on certain workload characteristics

Preparing Rocksdb plugin, we have experienced that some db_bench workloads and configurations cause abnormal behaviors on the running such as stuck operation or segmentation fault. The phenomenon happens on both MPDK and the SPDK 18.04 release and we yet thoroughly analysis the reason why. Please be aware this when you make use of the db_bench and rocksdb library.

Issue	Description
System crash after run 'run_flash_bench.sh'	Modify script "benchmark.sh" to run the test for ~1M keys. Run the script "run_flash_bench.sh" which internally use "benchmark.sh" script. This runs a sequence of tests in the following sequence: # step 1) load - bulkload, compact, fillseq, overwrite # step 2) read-only for each number of threads # step 3) read-write for each number of threads # step 4) merge for each number of threads Observe that system is crashing in between 30-60 minutes of time duration after starting the test while running 'step2' test "readrandom". This issue is reproducible on Ubuntu 16.04.4 LTS. Noticed high CPU utilization during this test. Please refer attached screenshot (last numbers captured using 'htop') Check the syslog/kern.log etc.. but did not notice any logs related to system crash.
Regression test results in segmentation fault	Regression test run results in seg fault. Sequence of tests in regression script (tools/regression_test.sh) fillseqdeterministic readrandom readwhilewriting deleterandom seekrandom seekrandomwhilewriting Once fillseqdeterministic is finished, seg fault occurs while doing device probe to run readrandom. Another observation is that even if we modify script to run fillseqdeterministic twice, then also seg fault is seen for second run. To run the test again device has to be formatted with nvme command.
'readwhilewriting' tests run for 36 hour got stuck	1. First run the test to fill the database (50M+ keys) using "filluniquerandom" **-** this test was successful. 2. Once the above test was completed, start the db_bench test for "readwhilewriting"- prefix size of 12 bytes is indexed. Database is initially loaded with 50+M unique keys by using db_bench's filluniquerandom mode, then read performance is measured for 36 hours run in readwhilewriting mode with 32 reader threads. Each reader thread issues random key request, which is guaranteed to be found. Another dedicated writer thread issues write requests in the meantime.

Issue	Description
WriteSync test getting stuck using 'db_bench'	We have observed that 'writeSync' test is getting stuck on two systems (both on Ubuntu 16.04.4 LTS and CentOS 7.5.1804) Steps: Run all the tests for 120 secs using 10 million keys, except set 'writesync' test for 3600 secs (one hour) Observe that all the tests before 'writesync' completed without issues, later 'writesync' test is getting stuck. #./run_tests.sh ./db_bench Start insert test phase...done. Generating perf report for insert test phase...done. Start overwrite test phase...done. Generating perf report for overwrite test phase...done. Start readwrite test phase...done. Generating perf report for readwrite test phase...done. Start writesync test phase... <<<< STUCK >>>>
Segmentation fault when dbbench 'use_plain_table'	If the "db_bench" is run using the "PlainTable SST format", observed segmentation fault.
Segmentation fault during "insert" after 2B+ keys	With the latest tagged build version MPDK-uNVMe2.0-18.07.04, during the insert - observe segmentation fault after 2B+ keys. Script "run_tests.sh" was run with default parameters except just changed the number of keys to 5B. Here is the stack trace: