Rev.1.2 7 Jan. 2019

Samsung® uNVMe 2.0 SDK Programming Guide

Revision History

Revision No.	History	Draft Date	Remark
Rev. 1.1	Updates for General KV SSD features	2018.11.01	
Rev. 1.2	Updates for BlobFS features	2019.01.07	

Contents

1 Device Support Information 7
1.1 Supported Devices 7
2 Terminology 8
2.1 Acronyms and Definitions 8
2.2 Feature option 8
3 Introduction 9
3.1 Scope 9
3.2 Assumption 9
4 Directory 10
4.1 bin 10
4.2 include 11
5 Constants & Data Structures 12
5.1 Constants 12
5.1.1 KV_MIN_KEY_LEN 12
5.1.2 KV_MAX_KEY_LEN 12
5.1.3 KV_MIN_VALUE_LEN 12
5.1.4 KV_MAX_IO_VALUE_LEN 12
5.1.5 KV_ITERATE_READ_BUFFER_OFFSET 12
5.1.6 KV_ITERATE_READ_BUFFER_SIZE 12
5.1.7 KV_MAX_ITERATE_HANDLE 13
5.2 Enum Constants 14
5.2.1 kv_sdk_init_types 14
5.2.2 kv_sdk_ssd_types 14
5.2.3 kv_cache_algorithm 14
5.2.4 kv_cache_reclaim 14
5.2.5 kv_slab_mm_alloc_policy 14
5.2.6 kv_store_option 15
5.2.7 kv_retrieve_option 15
5.2.8 kv_delete_option 15
5.2.9 kv_append_option(deprecated) 15
5.2.10 kv_exist_option 15
5.2.11 kv_format_option 16
5.2.12 kv_iterate_request_option 16
5.2.13 kv_iterate_handle_type 16
5.2.14 kv_keyspace_id 16
5.2.15 kv_iterate_read_option 16

5.2.16 kv_sdk_iterate_status 17
5.2.17 kv_result 17
5.3 Data Structures 19
5.3.1 kv_nvme_io_options 19
5.3.2 kv_sdk 19
5.3.3 kv_key 20
5.3.4 kv_value 20
5.3.5 kv_param 20
5.3.6 kv_pair 21
5.3.7 kv_iterate 21
5.3.8 kv_iterate_handle_info 21
6 Key Value SSD API 22
6.1 Device 22
6.1.1 kv_sdk_init 23
6.1.2 kv_is_sdk_initialized 25
6.1.3 kv_sdk_finalize 26
6.1.4 kv_get_device_handles 27
6.1.5 kv_get_core_id 28
6.1.6 kv_get_devices_on_cpu 29
6.1.7 kv_get_cpus_on_devices 30
6.1.8 kv_get_total_size 31
6.1.9 kv_get_used_size 32
6.1.10 kv_get_waf 33
6.1.11 kv_get_log_page 34
6.1.12 kv_format_device 35
6.1.13 kv_io_queue_type 36
6.1.14 kv_sdk_info 37
6.1.15 kv_process_completion 38
6.1.16 kv_process_completion_queue 39
6.2 Key-Value pair 40
6.2.1 kv_store 41
6.2.2 kv_store_async 43
6.2.3 kv_retrieve 44
6.2.4 kv_retrieve_async 46
6.2.5 kv_delete 47
6.2.6 kv_delete_async 48
6.2.7 kv_append (deprecated) 49
6.2.8 kv_iterate_open 50
6.2.9 kv_iterate_close 52
6.2.10 kv_iterate_read 53
6.2.11 kv_iterate_read_async 55
6.2.12 kv_iterate_info 56
6.2.13 kv_exist 57
6.2.14 kv_exist_async 58
7 New BlobFS API (Block SSD ONLY) 59
7.1 spdk_file_cache_free 59

7.2 spdk_file_set_retain_cache 60
7.3 spdk_file_get_retain_cache 61
7.4 spdk_file_set_prefetch_size 62
7.5 spdk_file_get_prefetch_size 63
7.6 spdk_file_set_prefetch_threshold 64
7.7 spdk_file_get_prefetch_threshold 65
7.8 spdk_file_set_direct_io 66
7.9 spdk_file_get_direct_io 67
8 unvme SDK Installation 68
8.1 Prerequisite 68
8.1.1 Supported Platforms 68
8.1.2 Install required software 68
8.2 Setup Hugepage Memory 72
8.3 Validate installation of the SDK 73
8.4 Update a Firmware of Block/KV SSD via nvme-cli 74
8.4.1 Install nvme-cli 74
8.4.2 Download and activate a firmware on a target SSD 74
8.4.3 Update Verification 75
8.5 Update a Firmware of Block/KV SSD with kv_nvme_cli 76
8.5.1 Download and activate a firmware on a target SSD 76
8.5.2 Update Verification 76
8.6 KV SSD Hot-Plug with uNVMe 77
9 Testing and Benchmarking tools 78
9.1 unvme2_fio_plugin 79
9.1.1 How to use 79
9.1.2 Supporting FIO Options 84
9.1.3 NUMA-aware CPU affinity Setting 88
9.1.4 Limitation 92
9.2 kv_nvme_cli 94
9.2.1 Main usage 94
9.3 unvme_rocksdb (Block SSD Only) 97
9.3.1 Prerequisite 97
9.3.2 Build 97
9.3.3 mkfs 97
9.3.4 RocksDB (db_bench) 98
9.3.5 Fuse 98
9.4 kv_perf 100

9.4.1 Options....100
9.4.2 Workloads....101
9.4.3 Scripts....102
9.5 sdk_perf / sdk_perf_async....103
9.5.1 Details of Performance Report Message....103
9.6 sdk_iterate / sdk_iterate_async (KV SSD Only)....105
9.7 blobfs_perf (Block SSD Only)....106
10 Appendix....107
10.1 Multi Process Support....107
10.1.1 Limitation....107
10.2 uNVMe SDK Configuration....108
10.3 uNVMe App Life Cycle....114
10.4 uNVMe Cache....115
10.5 uNVMe Slab M/M....116
10.6 uNVMe Blobfs Multi-queue Support (Block SSD Only)....117
10.6.1 SPDK Limitation....117
10.6.2 uNVMe MQ....117
10.7 uNVMe BlobFS Read Cache (Block SSD Only)....123
10.8 Performance Guideline....126
10.8.1 Configuration Guide....126
10.8.2 Performance Comparison among Different Drivers....130
10.9 Known Issues....133
10.9.1 Failure on uNVMe SDK Initialization due to tmpfs full....133
10.9.2 db_bench (rocksdb) malfunction on certain workload characteristics....134

1 DEVICE SUPPORT INFORMATION

This document describes Samsung® uNVMe *SSD* SDK software: Library, API, and Samples.

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 Programming Guide ver. 1.2	NVMe SSD (Block/KV)	NVMe 1.2

2 TERMINOLOGY

2.1 Acronyms and Definitions

Acronym/Term	Description
uNVMe	User Level NVMe Driver supporting both KV and Block SSD
KV	Key-value
NVMe	NVM Express (Non-Volatile Memory Express)
PCIe	PCI Express (Peripheral Component Interconnect Express)
SSD	Solid State Drive
Key-Value pair	Data entity in key-value SSD
SDK	Software Development Kit
UDD	User-level Device Diver (cf. Kernel Device Driver)
Iterate	Find out a set of matching keys in KV SSD. Only applied on KV SSD

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 supports Host-side SW Stack based user level device driver for both of KV and Block SSD. KV SSD is a brand-new SSD storage device that is able to handle IO with native *key-value* interfaces. Document information about the SDK and guidance regarding how to use the SDK to make your own KV/Block IO App.

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 protocols such as ATA, SCSI, and NVMe, and the API's internal device implementation. For more NVMe command protocol information, please refer to NVMe and KV NVMe specification.

3.2 Assumption

This guide has several assumptions.

1. Users of this SDK conduct device memory management. Any input and output buffers of SDK (and APIs) must be allocated before calling the routines. No memory the library allocates is accessible by user programming.
2. 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 library, 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:

```
uNVMe
├── app
│   ├── blobfs_perf
│   ├── external
│   ├── fio_plugin
│   ├── fuse
│   ├── mkfs
│   └── unvme_rocksdb
├── bin
├── doc
├── driver
│   ├── build
│   ├── core
│   ├── external
│   ├── include
│   └── test
├── include
├── io
│   ├── doxygen
│   ├── kv_perf_scripts
│   ├── src
│   └── tests
├── mk
└── script
```

Figure 1. uNVMe Directory

4.1 bin

```
bin
└── libuio.a
```

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 CONSTANTS & DATA STRUCTURES

This section defines uNVMe SDK core constants, data structures, and functions.

5.1 Constants

5.1.1 KV_MIN_KEY_LEN

The minimum key length in bytes that Samsung KV SSD can support. The default value is 4(B).

5.1.2 KV_MAX_KEY_LEN

The maximum key length in bytes that Samsung KV SSD can support. The default value is 255(B).

5.1.3 KV_MIN_VALUE_LEN

The minimum value length in bytes that Samsung KV SSD can support. The default value is 0(B).

5.1.4 KV_MAX_IO_VALUE_LEN

The maximum value length of a KV pair in byte that a single KV API call can support on KV SSD. The default value is 2097152(2MB).

5.1.5 KV_ITERATE_READ_BUFFER_OFFSET

This is the size of meta data that indicates number of keys and size of each key. The default value is 4(B).

5.1.6 KV_ITERATE_READ_BUFFER_SIZE

The iterate_read value buffer size in bytes that Samsung KV SSD can support. The default value is 32768(32KB).

5.1.7 KV_MAX_ITERATE_HANDLE

The maximum number of iterate handle that can be opened simultaneously. The default value is 16.

5.2 Enum Constants

The enum constants from 5.2.1 to 5.2.6 are used to initialize uNVMe SDK and uNVMe cache in *kv_sdk_init*.

5.2.1 kv_sdk_init_types

```
enum kv_sdk_init_types {
    KV_SDK_INIT_FROM_JSON       = 0x00,    // [DEFAULT] initialize sdk with json file
    KV_SDK_INIT_FROM_STR        = 0x01,    // initialize sdk with data structure 'kv_sdk'
};
```

5.2.2 kv_sdk_ssd_types

```
enum kv_sdk_ssd_types {
    KV_TYPE_SSD                 = 0x00,    // [DEFAULT] KV type of SSD
    LBA_TYPE_SSD                = 0x01,    // normal type of SSD
};
```

5.2.3 kv_cache_algorithm

```
enum kv_cache_algorithm {
    CACHE_ALGORITHM_RADIX       = 0x00,    // [DEFAULT] radix tree based cache
};
```

5.2.4 kv_cache_reclaim

```
enum kv_cache_reclaim {
    CACHE_RECLAIM_LRU           = 0x00,    // [DEFAULT] lru based reclaim policy
};
```

5.2.5 kv_slab_mm_alloc_policy

```
enum kv_slab_mm_alloc_policy {
    SLAB_MM_ALLOC_POSIX         = 0x10,    // slab allocator for using heap memory
    SLAB_MM_ALLOC_HUGE          =0x20,     // [DEFAULT] slab allocator for using hugepage memory
};
```

[NOTE] *SLAB_MM_ALLOC_HUGE* is available only for now.

The enum constants from 5.2.6 to 5.2.17 are used to set type(s) of Key-Value operation. Please see Key-Value pair APIs for more detail.

5.2.6 kv_store_option

```
enum kv_store_option {
    KV_STORE_DEFAULT              = 0x00,    // [DEFAULT] storing key value pair(or overwriting given value if key exists)
    KV_STORE_COMPRESSION          = 0x01,    // (TBD) compressing value before writing it to the storage
    KV_STORE_IDEMPOTENT           = 0x02,    // storing KV pair only if the key in the pair does not exist already in the device
};
```

[NOTE] *KV_STORE_DEFAULT* is available only for now.

5.2.7 kv_retrieve_option

```
enum kv_retrieve_option {
    KV_RETRIEVE_DEFAULT           = 0x00,    // [DEFAULT] retrieving value as it is written
                                             //          (even compressed value is also retrieved in its compressed form)
    KV_RETRIEVE_DECOMPRESSION     = 0x01,    // (TBD) returning value after decompressing it
};
```

[NOTE] *KV_RETRIEVE_DEFAULT* is available only for now. *KV_RETRIEVE_VALUE_SIZE is suspended (2018.05.31)*

5.2.8 kv_delete_option

```
enum kv_delete_option {
    KV_DELETE_DEFAULT                 = 0x00,    // [DEFAULT] default operation for command
    KV_DELETE_CHECK_IDEMPOTENT        = 0x01,    // check whether the key being deleted exists in SSD
};
```

5.2.9 kv_append_option(deprecated)

```
enum kv_append_option {
    KV_APPEND_DEFAULT             = 0x00,    // [DEFAULT] default operation for command
};
```

5.2.10 kv_exist_option

```
enum kv_exist_option {
```

```
    KV_EXIST_DEFAULT                = 0x00,     // [DEFAULT] default operation for command
};
```

5.2.11 kv_format_option

```
enum kv_format_option {
    KV_FORMAT_MAPDATA               = 0x00,
    KV_FORMAT_USERDATA              = 0x01,     //   [DEFAULT] default operation for format
};
```

5.2.12 kv_iterate_request_option

```
enum kv_iterate_request_option {
    KV_ITERATE_REQUEST_DEFAULT      = 0x00,     //   [DEFAULT] default operation for command
    KV_ITERATE_REQUEST_OPEN         = 0x01,     //   iterator open
    KV_ITERATE_REQUEST_CLOSE        = 0x02,     //   iterator close
};
```

5.2.13 kv_iterate_handle_type

```
enum kv_iterate_handle_type {
    KV_KEY_ITERATE                      = 0x01,     // return keys matched with specified prefix
    KV_KEY_ITERATE_WITH_DELETE          = 0x03      // delete key-value pairs which keys are matched with specified prefix
};
```

5.2.14 kv_keyspace_id

```
enum kv_keyspace_id {
            KV_KEYSPACE_IODATA          = 0x00,
            KV_KEYSPACE_METADATA        =0x01
};
```

5.2.15 kv_iterate_read_option

```
enum kv_iterate_read_option {
    KV_ITERATE_READ_DEFAULT             = 0x00,     // [DEFAULT] default operation for command
};
```

5.2.16 kv_sdk_iterate_status

```
enum kv_sdk_iterate_status {
    ITERATE_HANDLE_OPENED              = 0x01,
    ITERATE_HANDLE_CLOSED              = 0x00,
};
```

5.2.17 kv_result

Device APIs of the SDK return a return value after finishing its operation. **Please Note that** some return values are not implemented yet in the APIs. So please check pages carefully for each APIs to ensure their operations and returning values.

```
enum kv_result {
    KV_SUCCESS                                  = 0,        // successful

    KV_ERR_INVALID_VALUE_SIZE                   = 0x01,     // invalid value length(size)
    KV_ERR_INVALID_VALUE_OFFSET                 = 0x02,     // invalid value offset
    KV_ERR_INVALID_KEY_SIZE                     = 0x03,     // invalid key length(size)
    KV_ERR_INVALID_OPTION                       = 0x04,     // invalid I/O options
    KV_ERR_INVALID_KEYSPACE_ID                  = 0x05      // invalid keyspace ID (should be 0 or 1)

    KV_ERR_MISALIGNED_VALUE_SIZE                = 0x08,     // misaligned value length(size)
    KV_ERR_MISALIGNED_VALUE_OFFSET              = 0x09,     // misaligned value offset
    KV_ERR_MISALIGNED_KEY_SIZE                  = 0x0A,     // misaligned key length(size)

    KV_ERR_NOT_EXIST_KEY                        = 0x10,     // not existing key (unmapped key)
    KV_ERR_UNRECOVERED_ERROR                    = 0x11,     // internal I/O error
    KV_ERR_CAPACITY_EXCEEDED                    = 0x12,     // capacity limit

    KV_ERR_IDEMPOTENT_STORE_FAIL                = 0x80,     // overwrite fail (key is already stored with IDEMPOTENT option)
    KV_ERR_MAXIMUM_VALUE_SIZE_LIMIT_EXCEEDED    = 0x81,     // value length of given key is already full

    KV_ERR_ITERATE_FAIL_TO_PROCESS_REQUEST      = 0x90,     // fail to read/close handle with given handle id
    KV_ERR_ITERATE_NO_AVAILABLE_HANDLE          = 0x91,     // no more available handle
    KV_ERR_ITERATE_HANDLE_ALREADY_OPENED        = 0x92,     // fail to open iterator with given prefix/bitmask (already opened)
    KV_ERR_ITERATE_READ_EOF                     = 0x93,     // end-of-file for iterate_read with given iterator
    KV_ERR_ITERATE_REQUEST_FAIL                 = 0x94,     // fail to process the iterate request due to fw internal status
    KV_ERR_ITERATE_TCG_LOCKED                   = 0x95      // iterate TCG locked
    KV_ERR_ITERATE_ERROR                        = 0x96      // error while iterating

    KV_ERR_DD_NO_DEVICE                         = 0x100,    //   fail to find proper device
    KV_ERR_DD_INVALID_PARAM                     = 0x101,    //   invalid function parameter (NULL, option, and etc.)
    KV_ERR_DD_INVALID_QUEUE_TYPE                = 0x102,    //   invalid function call on async/sync queue
    KV_ERR_DD_NO_AVAILABLE_RESOURCE             = 0x103,    //   NVMe command pool empty, need to try later
    KV_ERR_DD_NO_AVAILABLE_QUEUE                = 0x104,    //   No available entry in submission queue, need to try later
    KV_ERR_DD_UNSUPPORTED_CMD                   = 0x105,    //   Unsupported KV command

    KV_ERR_SDK_OPEN                             = 0x200,    // device(sdk) open fail
    KV_ERR_SDK_CLOSE                            = 0x201,    // device(sdk) close fail
```

```
    KV_ERR_CACHE_NO_CACHED_KEY              = 0x202,    // (kv cache) cache miss
    KV_ERR_CACHE_INVALID_PARAM              = 0x203,    // (kv cache) invalid parameters
    KV_ERR_HEAP_ALLOC_FAILURE               = 0x204,    // heap allocation fail for sdk operations
    KV_ERR_SLAB_ALLOC_FAILURE               = 0x205,    // slab allocation fail for sdk operations
    KV_ERR_SDK_INVALID_PARAM                = 0x206,    // invalid parameters for sdk operations

    KV_WRN_MORE                             = 0x300,    // more results are available (for iterate API)
    KV_ERR_BUFFER                           = 0x301,    // not enough buffer (for retrieve, exist and iterate APIs)
    KV_ERR_DECOMPRESSION                    = 0x302,    // retrieving uncompressed value with DECOMPRESSION option
    KV_ERR_IO                               = 0x303,    // internal operation error (remained type for compatibility)
};
```

5.3 Data Structures

5.3.1 kv_nvme_io_options

```
struct kv_nvme_io_options{
        uint64_t core_mask;                // describing which CPU cores are used to run the I/O threads
        uint64_t sync_mask;                // specifying whether the thread performs sync(1) or async(0) I/O
        uint64_t num_cq_threads;           // number of CQ processing threads to be created.
        uint64_t cq_thread_mask;           // describing which CPU cores are used to run the CQ threads
        uint32_t queue_depth;              // queue depth of the NVMe device's I/O Queues
        uint32_t mem_size_mb;              // Shared memory size in MB
} kv_nvme_io_options;
```

This structure contains device I/O information.

5.3.2 kv_sdk

```
#define DEV_ID_LEN        32
#define NR_MAX_SSD        64
#define MAX_CPU_CORES     64

struct kv_sdk{
        bool use_cache;                                  // read cache enable/disable
        int cache_algorithm;                             // cache indexing algorithms (radix only)
        int cache_reclaim_policy;                        // cache eviction and reclaim policies (lru only)
        uint64_t slab_size;                              // size of slab memory used for cache and I/O buffer(B)
        int slab_alloc_policy;                           // slab memory allocation source (hugepage only)
        int ssd_type;                                    // type of ssds. (KV SSD only)
        int submit_retry_interval;                       // submit retry interval (us unit)

        int nr_ssd;                                      // # of SSDs
        char dev_id[NR_MAX_SSD][DEV_ID_LEN];             // PCI device address
        kv_nvme_io_options dd_options[NR_MAX_SSD];       // structure about description for devices
        uint64_t dev_handle[NR_MAX_SSD];                 // device handle

        int log_level;                                   // logging level for uNVMe SDK operations(from 0 to 3)
        char log_file[1024];                             // path of log file
        pthread_mutex_t cb_cnt_mutex;                    // mutex for callback counter
        uint64_t app_hugemem_size;                       // size of additional hugepage memory set by user app
};
```

This structure contains overall configuration and options for uNVMe SDK. Please refer to Enum Constants and kv_sdk_init.

5.3.3 kv_key

```
struct kv_key {
    void *key;                  // a pointer to a key
    uint16_t length;            // key length in byte unit
};
```

A key consists of a pointer and its length. For a string type key, the key buffer holds a byte string without a null terminator. The key field shall not be null. If the key buffer is NULL, an interface shall return an error, KV_ERR_KEY_INVALID.

5.3.4 kv_value

```
struct kv_value {
    void *value;                    // buffer address for value
    uint32_t length;                // value buffer size in byte unit
    uint32_t actual_value_size;     // total value size that is stored in the device (applied only on KV SSD)
    uint32_t offset;                // offset for value
};
```

A *kv_value* consists of a buffer pointer and its length. The *value* field must not be null. The *length* field specifies the size (length) of the value whose size of data will be stored or retrieved. The *offset* field is used only for *kv_retrieve* and *kv_retrieve_async* APIs to specify the offset from which the value will be retrieved.

[NOTE] *length* and *offset* fields should be aligned to KV_ALIGNMENT_UNIT.

5.3.5 kv_param

```
struct kv_param {
    void (*async_cb)();             // async notification callback (valid only for async I/O)
    void* private_data;             // private data address which can be used in callback (valid only for async I/O)
    union {
        int store_option;
        int retrieve_option;
        int delete_option;
        int iterate_request_option;
        int iterate_read_option;
        int exist_option;
    }io_option;
};                                  // union for IO operations
```

This structure contains I/O option such as an async I/O callback function, private data for the callback, or IO options.

The *async_cb* is used to specify which function will be called after the end of async I/O command. For more details about the configuration of async I/O options, please refer to sdk_perf.

5.3.6 kv_pair

```
struct kv_pair {
    uint8_t keyspace_id;
    kv_key key;
    kv_value value;
    kv_param param;
};
```

A pair of structures of key, value, and kv_param described right before. Please note that *kv_pair* is a basic unit for KV I/O operations, i.e. kv_store, kv_retrieve and kv_delete.

5.3.7 kv_iterate

```
typedef struct {
        uint32_t iterator;          // iterator handle opening
        kv_pair kv;                 // key/value pair to be retrieved
} kv_iterate;
```

This structure defines information for *kv_iterate_read() and kv_iterate_read_async() that* will return a set of existing keys matched with given *bit_pattern* within a range of bits masked by *bitmask*.

5.3.8 kv_iterate_handle_info

```
typedef struct kv_iterate_handle_info {
        uint8_t handle_id;
        uint8_t status;
        uint8_t type;
        uint8_t keyspace_id;
        uint32_t prefix;
        uint32_t bitmask;
        uint8_t is_eof;
        uint8_t reserved[3];
} kv_iterate_handle_info;
```

This structure defines information for *kv_iterate_handle_info* that will return a list of the given device's opened iterators. For more details, see *kv_iterate_info*.

6 KEY VALUE SSD API

6.1 Device

```
int kv_sdk_init (int init_from, void *option);

int kv_is_sdk_initialized(void);

int kv_sdk_finalize (void);

int kv_get_device_handles(int* nr_device, uint64_t* arr_handle);

int kv_get_core_id(void);

int kv_get_devices_on_cpu(int core_id, int* nr_device, uint64_t* arr_handle);

int kv_get_cpus_on_device(uint64_t handle, int* nr_core, int* arr_core);

uint64_t kv_get_total_size(uint64_t handle);

uint64_t kv_get_used_size(uint64_t handle);

uint64_t kv_get_waf(uint64_t handle);

int kv_get_log_page(uint64_t handle, uint8_t log_id, void* buffer, uint32_t buffer_size);

int kv_format_device(uint64_t handle, int erase_user_data);

int kv_io_queue_type(uint64_t handle, int core_id);

void kv_sdk_info(void);

void kv_process_completion(uint64_t handle);

void kv_process_completion_queue(uint64_t handle, uint32_t queue_id);
```

[NOTE] *kv_get_total_size(), kv_get_used_size(), kv_get_waf()* returns ***KV_ERR_INVALID_VALUE*** when the APIs fail. ***KV_ERR_INVALID_VALUE*** is defined as ***UINT64_MAX.***

6.1.1 kv_sdk_init

int kv_sdk_init (int init_from, void *option);

This API initializes uNVMe SSD(s) and uNVMe cache from the given option. There are two ways for the initialization as follows:

- Initializing from json file and
- Initializing from data structure (by using kv_sdk structure).

Please note that the *kv_sdk_init()* has to be called at once at the beginning of uNVMe-based applications prior to issue an IO such as store, retrieve, delete, and so on. The *kv_is_sdk_initialized()* can be called to check whether SDK is ready to be used. For a proper use of uNVMe SDK, please refer to uNVMe App Life Cycle.

If parameter *init_from* is specified as *KV_SDK_INIT_FROM_JSON*, this API will load options from json file whose path is specified in *option*.

```
char *json_path = "./kv_sdk_configuration.json";
int ret = kv_sdk_init(KV_SDK_INIT_FROM_JSON, json_path);
```

Below example is **a sample configuration** in a json file used for initializing and preparing the whole SDK and device(s).

```
Example: Set 2 devices for async I/O, without cache
{
  "cache": "off",                        // read cache enable/disable. "on" or "off"
  "cache_algorithm": "radix",            // cache indexing algorithms ("radix" only possible for now)
  "cache_reclaim_policy" : "lru",        // cache eviction and reclaim policies ("lru" only possible for now)
  "slab_size" : 512,                     // size of slab memory used for cache and I/O buffer. (MB)
  "slab_alloc_policy" : "huge",          // slab memory allocation source ("huge" page only for now)
  "ssd_type" : "kv"                      // type of ssds. "kv" or "lba"
  "log_level" : 0,                       // logging (verbose) level for uNVMe SDK operations. (including cache, from 0 to 3)
  "log_file" : "/tmp/kvsdk.log",         // path and name of a log file
   "device_description" : [
      {                                  // description for device #0
        "dev_id" : "0000:01:00.0",       // PCI device Address
        "core_mask" : F,                 // the number of NVMe submission queue and mapped cores on the SSD
        "sync_mask" : 0,                 // the IO mode of submission queue as sync(=1) or async mode(=0)
        "cq_thread_mask": 2,             // the number of CQ thread and mapped cores on the SSD
        "queue_depth" : 64               // the maximum length of submission queue device driver can see (valid only for async)
      }
      {                                  // description for device #1
        "dev_id" : "0000:02:00.0",
        "core_mask" : F0,
        "sync_mask" : 0,
        "cq_thread_mask" : 4,
        "queue_depth" : 256
      }
   ]
}
```

When parameter *init_from* is specified as *KV_SDK_INIT_FROM_STR,* this API regards *option* as a pointer of *kv_sdk* structure. Below code shows an initialization example when *KV_SDK_INIT_FROM_STR* is used for the parameter, *init_from*.

```
kv_sdk sdk_opt;

sdk_opt.use_cache = false;
sdk_opt.cache_algorithm = CACHE_ALGORITHM_RADIX;
sdk_opt.cache_reclaim_policy = CACHE_RECLAIM_LRU;
sdk_opt.slab_size = 512*1024*1024;
sdk_opt.slab_alloc_policy = SLAB_MM_ALLOC_HUGE;
sdk_opt.ssd_type = KV_TYPE_SSD;
sdk_opt.nr_ssd = 2;
sdk_opt.log_level = 0;
strcpy(sdk_opt.log_file, "/tmp/kvsdk.log");

/*description for device 0*/
strcpy(sdk_opt.dev_id[0], "0000:01:00.0");
sdk_opt.dd_options[0].core_mask = 0xF;
sdk_opt.dd_options[0].sync_mask = 0x0;
sdk_opt.dd_options[0].num_cq_threads = 1;
sdk_opt.dd_options[0].cq_thread_mask = 0x2;
sdk_opt.dd_options[0].queue_depth = 256;

/*description for device 1*/
strcpy(sdk_opt.dev_id[1], "0000:02:00.0");
sdk_opt.dd_options[1].core_mask = 0xF0;
sdk_opt.dd_options[1].sync_mask = 0x0;
sdk_opt.dd_options[1].num_cq_threads = 1;
sdk_opt.dd_options[1].cq_thread_mask = 0x2;
sdk_opt.dd_options[1].queue_depth = 256;

int ret = kv_sdk_init(KV_SDK_INIT_FROM_STR, &sdk_opt);
```

You can find more details about configuration features at here: uNVMe SDK Configuration

PARAMETERS

IN init_from	types of initializing SDK
IN option	configuration file's path information or pointer of config structure

RETURNS

KV_SUCCESS

ERROR CODE

KV_ERR_SDK_OPEN	device initialization fail

6.1.2 kv_is_sdk_initialized

int kv_is_sdk_initialized ();

This API returns whether uNVMe SDK is initialized (=1) or not (=0). Note that when this API returns 0, most of uNVMe APIs will not work. Calling *kv_sdk_init()* and normal completion of the call will make this function return 1.

PARAMETERS

RETURNS

1	initialized
0	not initialized

6.1.3 kv_sdk_finalize

int kv_sdk_finalize ();

This API de-initializes uNVMe SDK and uNVMe cache.

PARAMETERS

RETURNS

KV_SUCCESS

ERROR CODE

KV_ERR_SDK_CLOSE	device de-initialization fail

6.1.4 kv_get_device_handles

int kv_get_device_handles (int* nr_device, uint64_t* arr_handle);

This API is used to get the number of the uNVMe SSD handles and the array of handles that users (caller) process can access.

PARAMETERS

OUT nr_device	the number of the uNVMe SSDs able to access
OUT arr_handle	the array containing the uNVMe SSDs' handles

RETURNS

KV_SUCCESS

ERROR CODE

KV_ERR_IO	get handle fail

6.1.5 kv_get_core_id

int kv_get_core_id ();

This API returns an ID of the CPU on which the calling thread is being executed.

PARAMETERS

RETURNS

>=0	# of CPU ID

ERROR CODE

-1	CPU ID read fail

6.1.6 kv_get_devices_on_cpu

int kv_get_devices_on_cpu (int core_id, int* nr_device, uint64_t* arr_handle);

This API is used to get the number of the uNVMe SSD handles and the array of handles on which application context from the given *core_id* can access. Please note that *core_id* can be found from *kv_get_core_id*.

PARAMETERS

IN core_id	CPU ID
OUT nr_device	the number of the uNVMe SSDs able to access
OUT arr_handle	the array containing the uNVMe SSDs' handles

RETURNS

KV_SUCCESS

ERROR CODE

KV_ERR_IO	get handle fail

6.1.7 kv_get_cpus_on_devices

int kv_get_cpus_on_device (uint64_t handle, int* nr_core, int* arr_core);

This API returns the number of the cores and the array of core IDs from which the device of the given handle can be accessed.

PARAMETERS

IN handle	device handle
OUT nr_core	the number of the cores that the uNVMe SSD can be accessed
OUT arr_core	the array containing the core IDs

RETURNS

KV_SUCCESS

ERROR CODE

KV_ERR_IO	get CPU ID fail

6.1.8 kv_get_total_size

uint64_t kv_get_total_size (uint64_t handle);

This API returns a total size of the uNVMe SSD specified by *handle* in bytes. Please note that *handle* can be found from *kv_get_devices_on_cpu*.

PARAMETERS

IN handle	device handle

RETURNS

> 0	total size of the device in bytes

ERROR CODE

KV_ERR_INVALID_VALUE	size read fail
KV_ERR_SDK_INVALID_PARAM	invalid parameter (handle)

6.1.9 kv_get_used_size

uint64_t kv_get_used_size (uint64_t handle);

This API returns a used ratio of the uNVMe SSD specified by *handle*, from 0(0.00%) to 10,000(100.00%). Please note that *handle* can be found from *kv_get_devices_on_cpu*.

PARAMETERS

IN handle	device handle

RETURNS

0 ~ 10000 total used ratio of the device, In order to transfer the value into decimal two point, caller has to divide the value by 100.

ERROR CODE

KV_ERR_INVALID_VALUE	ratio read fail
KV_ERR_SDK_INVALID_PARAM	invalid parameter (handle)

6.1.10 kv_get_waf

uint64_t kv_get_waf (uint64_t handle);

This API returns the write amplification factor of the SSD specified by *handle*. Please note that *handle* can be found from *kv_get_devices_on_cpu*.

PARAMETERS

IN handle	device handle

RETURNS

> 0 W.A.F value from vendor log page. , In order to transfer the value into decimal one point, caller has to divide the value by 10.

ERROR CODE

KV_ERR_INVALID_VALUE	W.A.F value read fail
KV_ERR_SDK_INVALID_PARAM	invalid parameter (handle)

6.1.11 kv_get_log_page

int kv_get_log_page(uint64_t handle, uint8_t log_id, void* buffer, uint32_t buffer_size);

This API returns the result of get log page admin command.

PARAMETERS

IN handle	device handle
IN log_id	log id
OUT buffer	buffer pointer to store log data
IN buffer_size	size of buffer

RETURNS

KV_SUCCESS

ERROR CODE

KV_ERR_SDK_INVALID_PARAM	invalid parameter (handle)
KV_ERR_IO	device format fail

6.1.12 kv_format_device

int kv_format_device (uint64_t handle, int erase_user_data);

This API formats the SSD specified by the *handle*

PARAMETERS

IN handle	device handle
IN erase_user_data	0 = map only, 1 = erase user data

RETURNS

KV_SUCCESS

ERROR CODE

KV_ERR_SDK_INVALID_PARAM	invalid parameter (handle)
KV_ERR_IO	device format fail

6.1.13 kv_io_queue_type

int kv_io_queue_type (uint64_t handle, int core_id);

This API returns I/O Queue type for current (I/O) thread.

PARAMETERS

IN handle	device handle
IN core_id	CPU (OR IO queue) ID

RETURNS

SYNC_IO_QUEUE

ASYNC_IO_QUEUE

ERROR CODE

KV_ERR_IO	device format fail

6.1.14 kv_sdk_info

void kv_sdk_info (void);

This API shows the API information (build time / build system information)

6.1.15 kv_process_completion

void kv_process_completion(uint64_t handle);

This API processes IO completion on given ssd *handle.* With this API, user can complete IO on IO submission context without initializing IO completion thread(s).

PARAMETERS

IN handle	device handle

6.1.16 kv_process_completion_queue

void kv_process_completion_queue(uint64_t handle, uint32_t queue_id);

This API is almost same with kv_process_completion. However, this API ONLY processes completion on given queue id of ssd *handle*.

PARAMETERS

IN handle	device handle
IN queue_id	queue id

6.2 Key-Value pair

```
int kv_store(uint64_t handle, kv_pair *kv);

int kv_store_async (uint64_t handle, kv_pair *kv);

int kv_retrieve (uint64_t handle, kv_pair *kv);

int kv_retrieve_async (uint64_t handle, kv_pair*kv);

int kv_delete (uint64_t handle, kv_pair *kv);

int kv_delete_async (uint64_t handle, kv_pair *kv);

int kv_append (uint64_t handle, kv_pair *kv);

uint64_t kv_get_value_size (uint64_t handle, kv_pair *kv)

int kv_exist (uint64_t handle, kv_pair *kv);

int kv_exist_async (uint64_t handle, kv_pair *kv);

int kv_iterate (uint64_t handle, kv_iterate_options *iterate_option, kv_value *result);

uint32_t kv_iterate_open (uint64_t handle, const uint8_t keyspace_id, const uint32_t bitmask, const uint32_t prefix, const uint8_t
iterate_type);

int kv_iterate_close (uint64_t handle, const uint8_t iterator);

int kv_iterate_read (uint64_t handle, kv_iterate* it);

int kv_iterate_read_async (uint64_t handle, kv_iterate* it);

int kv_iterate_info (uint64_t handle, kv_iterate_handle_info* info, int nr_handle);
```

6.2.1 kv_store

int kv_store (uint64_t handle, kv_pair *kv)

This API stores a key-value pair in the KV device with sync I/O.

It supports several modes depending on the given option. (*kv.param.io_option.store_option*).

- ***KV_STORE_DEFAULT***: If a key exists, this operation will overwrite existing key-value pair with given value. If a key does not exist, this will store a new key-value pair with the given value. The *offset* in *kv.value* parameter will be ignored.
- ***KV_STORE_COMPRESSION***: Value will be compressed before written to KV SSD.
- ***KV_STORE_IDEMPOTENT***: With this option, users can store a value only once. If a key does not exist, this operation succeeds. If a key already exists, this operation returns *KV_ERR_IDEMPOTENT_STORE_FAIL*, without affecting existing one.

Note that *KV_STORE_COMPRESSION* and *KV_STORE_IDEMPOTENT* options can be set together.

handle can be found from *kv_get_devices_on_cpu()*. Please see *kv_get_devices_on_cpu* for more information.

If KV cache is enabled (see kv_sdk_init), this operation will update cache after the end of the device I/O. (Write Through)

PARAMETERS

IN handle	device handle
IN kv	kv_pair structure which contains key, value, and I/O options

RETURNS

KV_SUCCESS

ERROR CODE

KV_ERR_INVALID_VALUE_SIZE	invalid value length (size)
KV_ERR_INVALID_KEY_SIZE	invalid key length (size)
KV_ERR_INVALID_OPTION	invalid I/O option
KV_ERR_MISALIGNED_VALUE_SIZE	misaligned value length (size)
KV_ERR_MISALIGNED_KEY_SIZE	misaligned key length (size)
KV_ERR_UNRECOVERED_ERROR	internal I/O fail
KV_ERR_CAPACITY_EXCEEDED (TBD)	device capacity limit exceed
KV_ERR_IDEMPOTENT_STORE_FAIL (TBD)	overwrite fail when given key is already written with IDEMPOTENT option
KV_ERR_HEAP_ALLOC_FAILURE	heap memory allocation fail for sdk operations
KV_ERR_SLAB_ALLOC_FAILURE	slab memory allocation fail for sdk operations

KV_ERR_SDK_INVALID_PARAM	invalid parameters
KV_ERR_INVALID_KEYSPACE_ID	invalid keyspace id

6.2.2 kv_store_async

int kv_store_async (uint64_t handle, kv_pair *kv)

This API stores a key-value pair into the device with async I/O.

Basic features (including options, parameters, return types, etc.) are almost same with *kv_store()* (see kv_store) except that async callback function is required. Before using this API async callback function (and private data if needed) should be set in kv_param (*kv.param*). Please refer to kv_param.

PARAMETERS

IN handle	device handle
IN kv	kv_pair structure which contains key, value, I/O options

RETURNS

KV_SUCCESS

ERROR CODE

KV_ERR_INVALID_VALUE_SIZE	invalid value length (size)
KV_ERR_INVALID_KEY_SIZE	invalid key length (size)
KV_ERR_INVALID_OPTION	invalid I/O option
KV_ERR_MISALIGNED_VALUE_SIZE	misaligned value length (size)
KV_ERR_MISALIGNED_KEY_SIZE	misaligned key length (size)
KV_ERR_UNRECOVERED_ERROR	internal I/O fail
KV_ERR_CAPACITY_EXCEEDED (TBD)	device capacity limit exceed
KV_ERR_IDEMPOTENT_STORE_FAIL (TBD)	overwrite fail when given key is already written with IDEMPOTENT option
KV_ERR_HEAP_ALLOC_FAILURE	heap memory allocation fail for sdk operations
KV_ERR_SLAB_ALLOC_FAILURE	slab memory allocation fail for sdk operations
KV_ERR_SDK_INVALID_PARAM	invalid parameters
KV_ERR_INVALID_KEYSPACE_ID	invalid keyspace id

6.2.3 kv_retrieve

int kv_retrieve (uint64_t handle, kv_pair *kv)

This API retrieves a value with a given key as much as specified in *kv.value.length*, from given offset (*kv.value.offset*) of stored value in sync I/O.

The *value.value* in *kv* (kv_pair) should be set by enough size of output buffer's address for retrieving value. On return, *kv.value.length* will be set to the actual retrieved size of the value, whereas actual_value_size will be set to the total value size stored in the KV SSD. Note that 0 of *kv.value.length* after retrieving means the key (*kv.key.key*) doesn't exist.

Case 1) Actual stored value 10KB, retrieving with *kv->value.length* = 4KB → *kv->value.length* after operations will be 4KB.
Case 2) Actual stored value 10KB, retrieving with *kv->value.length* = 20KB → *kv->value.length* after operations will be 10KB.
Case 3) Actual stored value 10KB, retrieving with *kv->value.length* = 10KB and *kv->value.offset* = 5KB → *kv->value.length* after operations will be 5KB.
Case 3) Actual stored value 10KB, retrieving with *kv->value.length* = 10KB and *kv->value.offset* = 10KB → *kv->value.length* after operations will be 0.

~~If given value buffer is not big enough to hold the entire value, this API will set *kv.value.length* to the actual size that should be transferred and return *KV_ERR_BUFFER*. Given buffer will be filled with partial value to fit given buffer's size. (TBD)~~

Below is retrieve I/O options supported. (*kv.param.io_option.retrieve_option)*

- ***KV_RETRIEVE_DEFAULT***: Default retrieve operation (retrieving value as it is written)
- ***KV_RETRIEVE_DECOMPRESSION***: Returns value after decompressing it

Note that retrieving uncompressed value with *KV_RETRIEVE_DECOMPRESSION* option returns an error *KV_ERR_DECOMPRESSION. (TBD)*

PARAMETERS

IN handle	device handle
IN kv	kv_pair structure which contains key, I/O options (value and its size will be set from API)

RETURNS

KV_SUCCESS

ERROR CODE

KV_ERR_INVALID_VALUE_SIZE	invalid value length (size)

KV_ERR_INVALID_VALUE_OFFSET	invalid value offset
KV_ERR_INVALID_KEY_SIZE	invalid key length (size)
KV_ERR_INVALID_OPTION	invalid I/O option
KV_ERR_MISALIGNED_VALUE_SIZE	misaligned value length (size)
KV_ERR_MISALIGNED_VALUE_OFFSET	misaligned value offset (size)
KV_ERR_MISALIGNED_KEY_SIZE	misaligned key length (size)
KV_ERR_UNRECOVERED_ERROR	internal I/O fail
KV_ERR_HEAP_ALLOC_FAILURE	heap memory allocation fail for sdk operations
KV_ERR_SLAB_ALLOC_FAILURE	slab memory allocation fail for sdk operations
KV_ERR_SDK_INVALID_PARAM	invalid parameters
KV_ERR_NOT_EXIST_KEY	not existing key
KV_ERR_DECOMPRESSION (TBD)	decompression fail
KV_ERR_INVALID_KEYSPACE_ID	invalid keyspace id

6.2.4 kv_retrieve_async

int kv_retrieve_async (uint64_t handle, kv_pair *kv)

This API retrieves a value with a given key in async I/O.

Basic features (including options, parameters, return types, etc.) are almost same with *kv_retrieve()* (see kv_retrieve) except that async callback function is required. Before using this API async callback function (and private data if needed) should be set in kv_param (*kv.param*). Please refer to kv_param.

PARAMETERS

IN handle	device handle
IN kv	kv_pair structure which contains key, I/O options (value and its size will be set from API)

RETURNS

KV_SUCCESS

ERROR CODE

KV_ERR_INVALID_VALUE_SIZE	invalid value length (size)
KV_ERR_INVALID_VALUE_OFFSET	invalid value offset
KV_ERR_INVALID_KEY_SIZE	invalid key length (size)
KV_ERR_INVALID_OPTION	invalid I/O option
KV_ERR_MISALIGNED_VALUE_SIZE	misaligned value length (size)
KV_ERR_MISALIGNED_VALUE_OFFSET	misaligned value offset (size)
KV_ERR_MISALIGNED_KEY_SIZE	misaligned key length (size)
KV_ERR_UNRECOVERED_ERROR	internal I/O fail
KV_ERR_HEAP_ALLOC_FAILURE	heap memory allocation fail for sdk operations
KV_ERR_SLAB_ALLOC_FAILURE	slab memory allocation fail for sdk operations
KV_ERR_SDK_INVALID_PARAM	invalid parameters
KV_ERR_NOT_EXIST_KEY	not existing key
KV_ERR_DECOMPRESSION (TBD)	decompression fail
KV_ERR_INVALID_KEYSPACE_ID	invalid keyspace id

6.2.5 kv_delete

int kv_delete (uint64_t handle, kv_pair *kv)

This API deletes value with a given key in sync I/O.

Below is delete I/O options supported. (*kv.param.io_option.delete_option)*

- ***KV_DELETE_DEFAULT***: try to delete the key
- ***KV_DELETE_CHECK_IDEMPOTENT***: try to delete the key, then, check whether the key being deleted exists in KV SSD

PARAMETERS

IN handle	device handle
IN kv	kv_pair structure which contains key, I/O options

RETURNS

KV_SUCCESS

ERROR CODE

KV_ERR_INVALID_OPTION invalid I/O option

KV_ERR_NOT_EXIST_KEY not existing key (only when *KV_DELETE_CHECK_IDEMPOTENT option is used, KV_DELETE_DEFAULT doesn't return the error for the deletetion trial)*

KV_ERR_UNRECOVERED_ERROR	internal I/O fail
KV_ERR_HEAP_ALLOC_FAILURE	heap memory allocation fail for sdk operations
KV_ERR_SLAB_ALLOC_FAILURE	slab memory allocation fail for sdk operations
KV_ERR_SDK_INVALID_PARAM	invalid parameters
KV_ERR_INVALID_KEYSPACE_ID	invalid keyspace id

6.2.6 kv_delete_async

int kv_delete_async (uint64_t handle, kv_pair *kv)

This API deletes value with a given key in async I/O.

Basic features (including options, parameters, return types, etc.) are almost same with *kv_delete()* (see kv_delete) except that async callback function is required. Before using this API async callback function (and private data if needed) should be set in kv_param (*kv.param*). Please refer to kv_param.

PARAMETERS

IN handle	device handle
IN kv	kv_pair structure which contains key, I/O options

RETURNS

KV_SUCCESS

ERROR CODE

KV_ERR_INVALID_OPTION invalid I/O option

KV_ERR_NOT_EXIST_KEY not existing key (only when *KV_DELETE_CHECK_IDEMPOTENT option is used, KV_DELETE_DEFAULT doesn't return the error for the deletion trial)*

KV_ERR_UNRECOVERED_ERROR	internal I/O fail
KV_ERR_HEAP_ALLOC_FAILURE	heap memory allocation fail for sdk operations
KV_ERR_SLAB_ALLOC_FAILURE	slab memory allocation fail for sdk operations
KV_ERR_SDK_INVALID_PARAM	invalid parameters
KV_ERR_INVALID_KEYSPACE_ID	invalid keyspace id

6.2.7 kv_append (deprecated)

int kv_append (uint64_t handle, kv_pair *kv)

This API appends a new value (*kv.value.value*) to the value already stored by the given key. When the key does not exist, it will return an error.

There is only one option for this API. (*kv.param.io_option.append_option*)

- ***KV_APPEND_DEFAULT***

[NOTE] **Please check below before using this API**

1) Async operation of append is not supported.

2) The size of the appended value should be multiple of *KV_ALIGNMENT_UNIT*.

3) Total value size for each key can grow up to *KV_MAX_TOTAL_VALUE_LEN*.

PARAMETERS

IN handle	device handle
IN kv	kv_pair structure which contains key, value, I/O options

RETURNS

KV_SUCCESS

ERROR CODE

KV_ERR_INVALID_VALUE_SIZE	invalid value length (size)
KV_ERR_INVALID_KEY_SIZE	invalid key length (size)
KV_ERR_INVALID_OPTION	invalid I/O option
KV_ERR_MISALIGNED_VALUE_SIZE	misaligned value length (size)
KV_ERR_MISALIGNED_KEY_SIZE	misaligned key length (size)
KV_ERR_NOT_EXIST_KEY	not existing key
KV_ERR_UNRECOVERED_ERROR	internal I/O fail
KV_ERR_CAPACITY_EXCEEDED (TBD)	device capacity limit exceed
KV_ERR_MAXIMUM_VALUE_SIZE_LIMIT_EXCEEDED	value of given key is already full
KV_ERR_HEAP_ALLOC_FAILURE	heap memory allocation fail for sdk operations
KV_ERR_SLAB_ALLOC_FAILURE	slab memory allocation fail for sdk operations
KV_ERR_SDK_INVALID_PARAM	invalid parameters

6.2.8 kv_iterate_open

uint32_t kv_iterate_open(uint64_t handle, const uint8_t keyspace_id, const uint32_t bitmask, const uint32_t prefix, const uint8_t iterate_type);

This API and the other iterate APIs such as *kv_iterate_close*, *kv_iterate_read*, *kv_iterate_read_async* and *kv_iterate_info* enable users to iterate existing keys within the device by getting subsets of valid keys.

kv_iterate_open() will open iterator handle with given *keyspace_id*, *bitmas*k, and *prefix* while *kv_iterate_close()* closes the iterator opened. Note that *iterate_type* determines attributes of the iterate handles and returning values of *kv_iterate_read() with the handles*. See kv_iterate_handle_type for more details.

When an iterate handle is opened as KV_KEY_ITERATE, *kv_iterate_read() and kv_iterate_read_async()* will return a set of keys (or keys with its values) which exist in the device by matching iterator handle to all keys in the device considering bitmask and prefix specified in *kv_iterate_open()*

When an iterate handle is open as KV_KEY_ITERATE_WITH_DELETE type, kv_iterate_read() and kv_iterate_read_async() should not be called. KV SSD will return an error for the call. Instead, KV SSD automatically starts removing the matching key and Application is able to retrieve its progress by calling kv_iterate_info(). kv_iterate_handle_info.is_eof = 0 or 1

Bits which are enabled by *bitmask* are related to *kv_iterate_open*() API. The *prefix* is used to specify the pattern in bits in keys. Given that *bitmask* is 0xF0000000, if *prefix* is 0x30000000, *kv_iterate_open()* will return an iterator handle to retrieve keys that includes 0x3XXXXXXX. In short, the device will return all the keys (or key-value pairs) matching that “(*bitmask* & key) == *bit_pattern*” for *kv_iterate_read()*

bitmask should be set from the first bit of a key and it is not allowed setting bitmask from a middle position of a key. Hence, Setting *bitmask* / *prefix* as 0xFF / 0x0F is allowed while 0x0F / 0x0F is not allowed.

Below are some examples.

1) If users want to get all the existing keys within the device with the first bit of a key is 1, this API should be called with *bitmask* = 0x80000000 (1000 0000 0000 0000 0000 0000 0000 0000) and *prefix* = 0x80000000 (1000 0000 0000 0000 0000 0000 0000 0000).
2) If users want to get all the existing keys within the device with the first bit of key is 0, *bitmask* should be 0x80000000 (1000 0000 0000 0000 0000 0000 0000 0000) and *prefix* should be 0x0 (0000 0000 0000 0000 0000 0000 0000 0000).
3) If users want to get all the existing keys with the second bytes (bit 8 ~ bit15) is equal to 0x04, *bitmask* should be 0xFFFF0000 (0000 0000 1111 1111 0000 0000 0000 0000) and *prefix* should be 0x00040000 (0000 0000 0000 0100 0000 0000 0000 0000).
4) If user wants to get all the existing keys with bit 1 ~ bit 4 is equal to (0101), *bitmask* should be 0xF8000000 (1111 1000 0000 0000 0000 0000 0000 0000) and *prefix* should be 0x28000000 (0010 1000 0000 0000 0000 0000 0000 0000).

PARAMETERS

IN handle	device handle
IN keyspace_id	keyspace_id
IN bitmask	bitmask
IN prefix	prefix
IN iterate_type	iterate type

RETURNS

> 0	an iterator's handles

ERROR CODE

KV_ERR_DD_INVALID_PARAM	invalid parameter(s)
KV_ERR_ITERATE_NO_AVAILABLE_HANDLE	no more available handle
KV_ERR_ITERATE_HANDLE_ALREADY_OPENED	given prefix/bitmask is already opened
KV_ERR_ITERATE_ERROR	open error

[NOTE] The response of KV_KEY_ITERATE may include already deleted (invalidated) keys. Also, it may return a same valid key more than twice. The strict requirement for key iterate and key iterate with retrieve is returning all valid keys, that is, keys written in the KV-SSD before the iterate open is called, at least once to the host

6.2.9 kv_iterate_close

int kv_iterate_close(uint64_t handle, const uint8_t iterator);

This API closes the target iterator

PARAMETERS

IN handle	device handle
IN iterator	iterator handle

RETURNS

KV_SUCCESS

ERROR CODE

KV_ERR_DD_INVALID_PARAM	invalid parameter(s)
KV_ERR_ITERATE_FAIL_TO_PROCESS_REQUEST	fail to close handle with given handle id

6.2.10 kv_iterate_read

int kv_iterate_read(uint64_t handle, kv_iterate* it);

This API reads keys or a key/value pair with the given iterator.

kv_iterate_read() caller is responsible for passing *valid iterator handle* and allocating buffer in *value.value.* The API will return a buffer that is filled with matching key set, and mark the size in *value.length*. Below is the iterate response format.

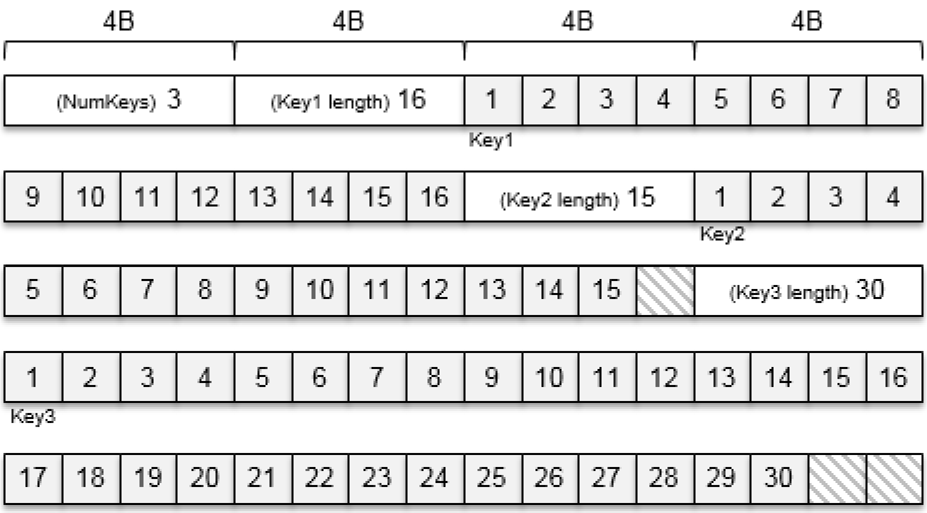


Figure 4. Iterate response format

By repeating calling the *kv_iterate_read(),* when it reach up to the end of matching key set, it will return KV_ERR_ITERATOR_EOF error code, with the last chunk of read buffer and its size.

There is only one option for this API.

KV_ITERATE_READ_DEFAULT

PARAMETERS

IN handle	device handle
IN it	*kv_iterator* structure which contains an opened iterator handle, read size and I/O options (Its value (list of keys or a key/value pair matched) will be set from the device specified by handle)

RETURNS

KV_SUCCESS

ERROR CODE

KV_ERR_SDK_INVALID_PARAM	invalid parameters
KV_ERR_INVALID_VALUE_SIZE	invalid iterate_read buffer length (*it->value.length*)
KV_ERR_MISALIGNED_VALUE_SIZE	misaligned iterate_read buffer length (*it->value.length*)
KV_ERR_HEAP_ALLOC_FAILURE	heap memory allocation fail for sdk operations
KV_ERR_SLAB_ALLOC_FAILURE	slab memory allocation fail for sdk operations
KV_ERR_ITERATE_FAIL_TO_PROCESS_REQUEST	fail to read handle with given handle id
KV_ERR_ITERATOR_EOF	end-of-file for iterate_read with given iterator
KV_ERR_ITERATE_REQUEST_FAIL	fail to iterate_read
KV_ERR_INVALID_KEYSPACE_ID	invalid keyspace id

6.2.11 kv_iterate_read_async

int kv_iterate_read_async(uint64_t handle, kv_iterate* it);

This API reads keys or a key/value pair with the given iterator in async I/O.

Basic features (including options, parameters, return types, etc.) are almost same with *6.2.12 kv_iterate_read()* (see kv_iterate_read) except that async callback function is required. Before using this API async callback function (and private data if needed) should be set in kv_param (*kv.param*). Please refer to kv_param.

PARAMETERS

IN handle	device handle
IN it	*kv_iterator* structure which contains an opened iterator handle, read size and I/O options (Its value (list of keys) will be set from the device specified by handle)

RETURNS

KV_SUCCESS	keys to be read more remain

ERROR CODE

KV_ERR_SDK_INVALID_PARAM	invalid parameters
KV_ERR_INVALID_VALUE_SIZE	invalid iterate_read buffer length (*it->value.length*)
KV_ERR_MISALIGNED_VALUE_SIZE	misaligned iterate_read buffer length (*it->value.length*)
KV_ERR_HEAP_ALLOC_FAILURE	heap memory allocation fail for sdk operations
KV_ERR_SLAB_ALLOC_FAILURE	slab memory allocation fail for sdk operations
KV_ERR_ITERATE_FAIL_TO_PROCESS_REQUEST	fail to read handle with given handle id
KV_ERR_ITERATOR_EOF	end-of-file for iterate_read with given iterator
KV_ERR_ITERATE_REQUEST_FAIL	fail to iterate_read
KV_ERR_INVALID_KEYSPACE_ID	invalid keyspace id

6.2.12 kv_iterate_info

int kv_iterate_info (uint64_t handle, kv_iterate_handle_info* info, int nr_handle);

This API returns information of iterate handle(s) including handle id, status, bitmask, prefix, and so on. Users can specify how many handles' information this API returns by setting *nr_handle*. Note that field *status* in *info* will be set by *ITERATE_HANDLE_OPENED* or *ITERATE_HANDLE_CLOSED* (see kv_iterate_handle_info and kv_sdk_iterate_status).

PARAMETERS

IN handle	device handle
OUT info	kv_iterator_handle_info structure which contains opened iterators' handle, status, prefix, and bitmask
IN nr_handle	number of iterators to get information

RETURNS

KV_SUCCESS

ERROR CODE

KV_ERR_SDK_INVALID_PARAM	invalid parameter (handle)
KV_ERR_IO	get iterate info fail

6.2.13 kv_exist

int kv_exist (uint64_t handle, kv_pair* kv)

This API checks if a given key already exists or not and returns status.

Note that just like any other commands and operations, the device cannot guarantee the ordering between *kv_exist()* and other operations. Therefore, *kv_exist()* call will return the status at the time of process of this command within the device.

There is only one option for this API.

- ***KV_EXIST_DEFAULT***

PARAMETERS

IN handle	device handle
IN kv	kv_pair structure which contains key, I/O options

RETURNS

KV_SUCCESS

KV_ERR_NOT_EXIST_KEY

ERROR CODE

KV_ERR_SDK_INVALID_PARAM	invalid parameters
KV_ERR_IO	exist I/O fail
KV_ERR_INVALID_KEYSPACE_ID	invalid keyspace id

6.2.14 kv_exist_async

int kv_exist_async (uint64_t handle, kv_pair* kv)

This API checks if a given key already exists or not and returns status in an asynchronous manner.

Note that just like any other commands and operations, the device cannot guarantee the ordering between *kv_exist_async()* and other operations. Therefore, *kv_exist_async()* call will return the status at the time of process of this command within the device.

PARAMETERS

IN handle	device handle
IN kv	kv_pair structure which contains key, I/O options

RETURNS

KV_SUCCESS

KV_ERR_NOT_EXIST_KEY

ERROR CODE

KV_ERR_SDK_INVALID_PARAM	invalid parameters
KV_ERR_IO	exist I/O fail
KV_ERR_INVALID_KEYSPACE_ID	invalid keyspace id

7 NEW BLOBFS API (BLOCK SSD ONLY)

7.1 spdk_file_cache_free

void spdk_file_cache_free(struct spdk_file *file);

This API frees cache of the given file.

PARAMETERS

IN file	A target BlobFS file

7.2 spdk_file_set_retain_cache

void spdk_file_set_retain_cache(struct spdk_file *file, bool retain);

This API makes the given target file retaining its cache after *cache read*

PARAMETERS

IN file	A target BlobFS file
IN retain	Flags

7.3 spdk_file_get_retain_cache

bool spdk_file_get_retain_cache(struct spdk_file *file);

This API gets the status of given target file whether cache_retain of the file is enabled or not

PARAMETERS

IN file	A target BlobFS file

RETURNS

Flags whether the cache_retain is enabled or not of the given target file

7.4 spdk_file_set_prefetch_size

void spdk_file_set_prefetch_size(struct spdk_file *file, int size);

This API sets the *cache read ahead size* of the given file

PARAMETERS

IN file	A target BlobFS file
IN size	cache read ahead size

7.5 spdk_file_get_prefetch_size

int spdk_file_get_prefetch_size(struct spdk_file *file);

This API gets the *cache read ahead size* of the given file

PARAMETERS

IN file	A target BlobFS file

RETURNS

cache read ahead size (B) of the given target BlobFS file

7.6 spdk_file_set_prefetch_threshold

void spdk_file_set_prefetch_threshold(struct spdk_file *file, int size);

This API sets the prefetch threshold the given file

PARAMETERS

IN file	A target BlobFS file
IN size	prefetching threshold

7.7 spdk_file_get_prefetch_threshold

int spdk_file_get_prefetch_threshold(struct spdk_file *file);

This API gets the *prefetch threshold* of the given file

PARAMETERS

IN file	A target BlobFS file

RETURNS

prefetch threshold (B) of the given target BlobFS file

7.8 spdk_file_set_direct_io

*void spdk_file_set_direct_io(struct spdk_file *file, uint32_t direct_io);*

This API sets IO type of the given target file

PARAMETERS

IN file	A target BlobFS file
IN direct_io	bitmask for IO types

BLOBFS_BUFFERED_READ 0x0

BLOBFS_BUFFERED_WRITE 0x0

BLOBFS_DIRECT_READ 0x1

BLOBFS_DIRECT_WRITE 0x2

7.9 spdk_file_get_direct_io

uint32_t spdk_file_get_direct_io(struct spdk_file *file);

This API gets IO types of the given target file

PARAMETERS

IN file	A target BlobFS file

RETURNS

bitmask for IO types

BLOBFS_BUFFERED_READ 0x0

BLOBFS_BUFFERED_WRITE 0x0

BLOBFS_DIRECT_READ 0x1

BLOBFS_DIRECT_WRITE 0x2

8 UNVME SDK INSTALLATION

This section explains how to build uNVMe SDK for uNVMe SDK new comers.

8.1 Prerequisite

8.1.1 Supported Platforms

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

8.1.2 Install required software

A number of software packages are needed to 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*

  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

8.2 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 Block 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

8.3 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.

8.4 Update a Firmware of Block/KV SSD via nvme-cli

8.4.1 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

8.4.2 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.

8.4.3 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 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 nvme-cli website.

8.5 Update a Firmware of Block/KV SSD with kv_nvme_cli

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

8.5.1 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 commands 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

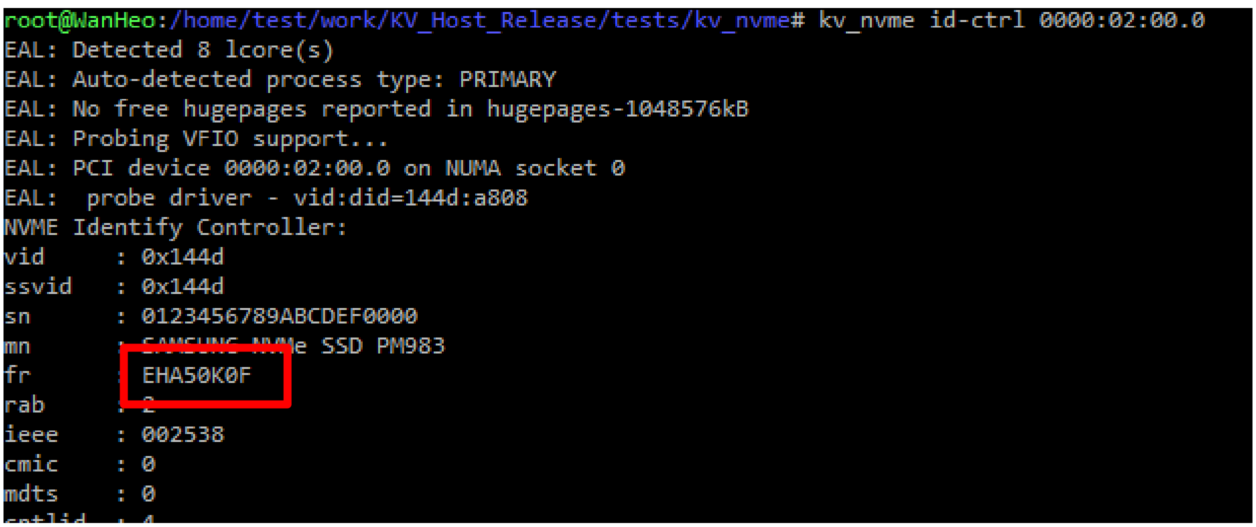
[NOTE] Updating or changing firmware of SSD needs hard reboot of the system or hot-plug.

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

8.5.2 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



```
root@WanHeo:/home/test/work/KV_Host_Release/tests/kv_nvme# kv_nvme id-ctrl 0000:02:00.0
EAL: Detected 8 lcore(s)
EAL: Auto-detected process type: PRIMARY
EAL: No free hugepages reported in hugepages-1048576kB
EAL: Probing VFIO support...
EAL: PCI device 0000:02:00.0 on NUMA socket 0
EAL:  probe driver - vid:did=144d:a808
NVME Identify Controller:
vid     : 0x144d
ssvid   : 0x144d
sn      : 0123456789ABCDEF0000
mn      : SAMSUNG NVMe SSD PM983
fr      : EHA50K0F
rab     : 2
ieee    : 002538
cmic    : 0
mdts    : 0
```

Figure 6. 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_cli.

8.6 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 Block 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.

9 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/Block SSD.
- kv_nvme_cli: nvme-cli compatible tool to issue admin/io command. Support KV/Block SSD.
- mkfs: a tool to install Blobstore file system on a Block SSD.
- fuse: a tool to mount a Block SSD as fuse file system.
- unvme_rocksdb: rocksdb with uNVMe SDK for a Block 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/Block 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/Block SSD.
- sdk_iterate : Based on sdk_perf, further performs iterate operations. Support KV/Block SSD.

9.1 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 SDK.

unvme2_fio_plugin has two main advantages.

1) Users can generate IO workload to KV SSD as well as Block 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.

9.1.1 How to use

[NOTE] Before building and running unvme2_fio_plugin, please check hugepage memory was set enough to run uNVMe SDK applications. See Setup Hugepage Memory for more details.

9.1.1.1 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.

9.1.1.2 Build unvme2 fio plugin

Users can build *unvme2_fio_plugin* through below cmd.

```
$> cd {path}/{to}/{uNVMe SDK}
$> ./make.sh 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 7. Makefile of unvme2_fio_plugin

As you can see in Figure above, 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
```

9.1.1.3 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
```

9.1.1.4 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 8. 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 Block 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 Block 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.

9.1.1.4.1 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).

9.1.1.4.2 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.

9.1.1.4.3 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" : "0xF"          //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
```

9.1.2 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		

9.1.2.1 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*').

9.1.2.2 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.

9.1.2.3 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)

9.1.2.4 Example of 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
```

9.1.2.5 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.

9.1.2.5.1 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.

9.1.2.5.2 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 9. 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 10. 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 (e.g. NUMA node, CPU redundancy, etc.) should be considered well. Detail of CPU affinity setting will be described subsequent section.

9.1.3 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.

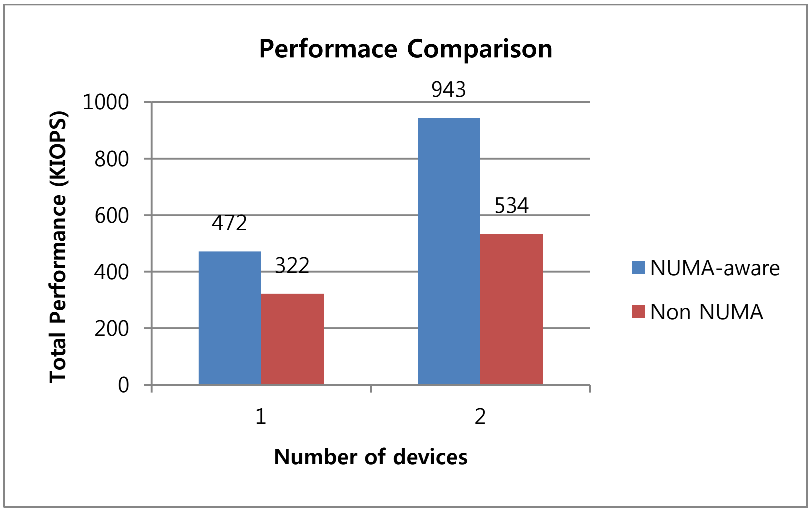
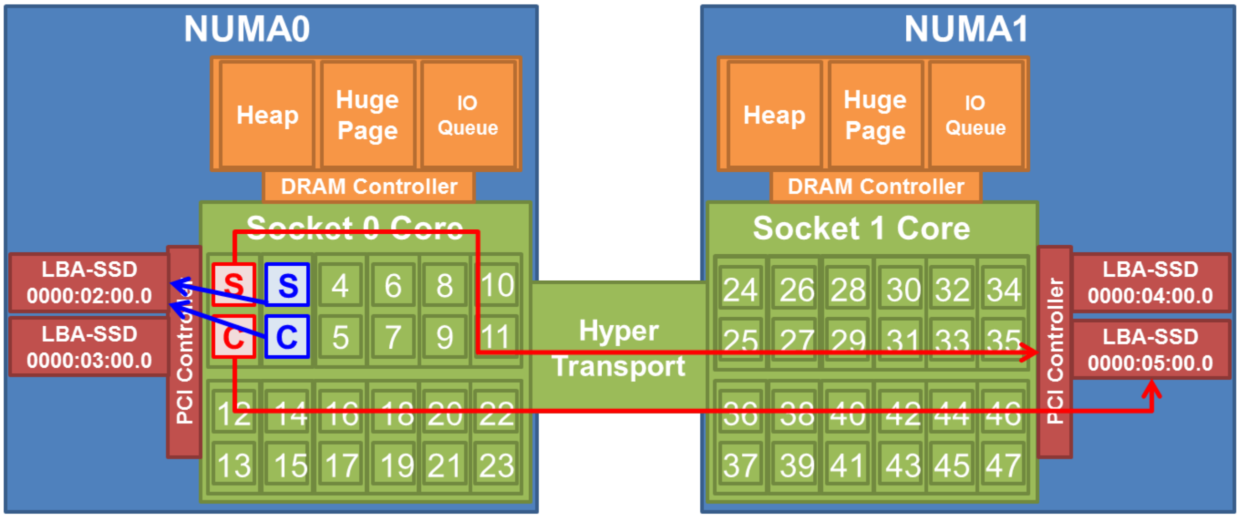


Figure 11. Performance comparison by CPU affinity setting

NUMA-aware CPU affinity setting (2 devices)	Non NUMA-aware CPU affinity setting (2 devices)
{ "ssd_type" : "lba", "device_description" : [{ "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 }] }	{ "ssd_type" : "lba", "device_description" : [{ "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.

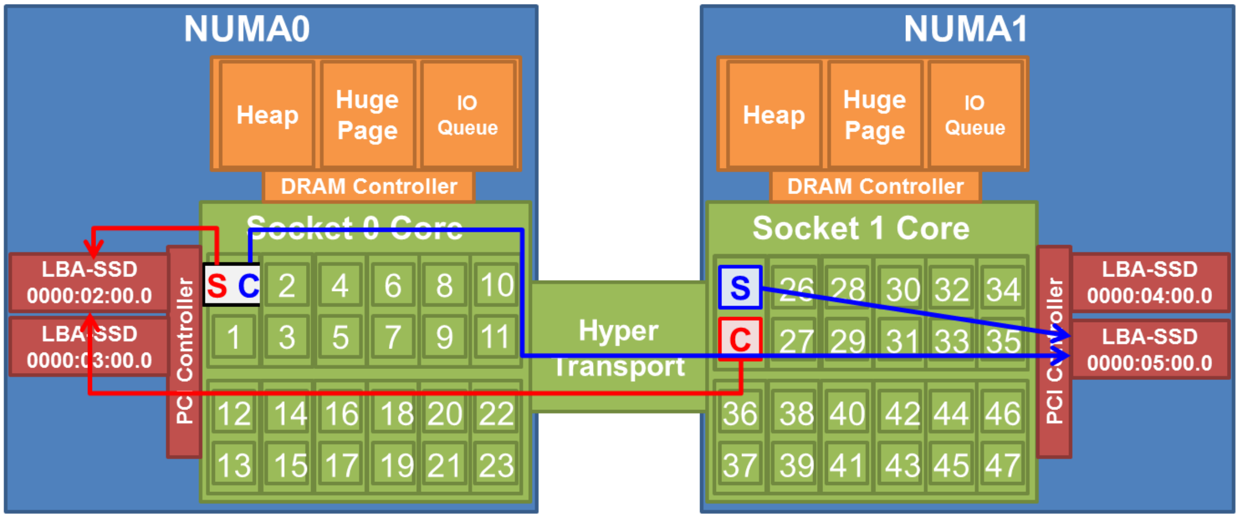


* **S:** *set as submission core,* * **C**: *set as completion core*

Figure 12. 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**.



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

Figure 13. 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)
- Block 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.

9.1.3.1 L3Cache-aware CPU affinity Setting

Even if CPU affinity is set in consideration of NUMA Node, performance drop may occur on some CPUs. The CPU of the test bed in 7.5.3 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).

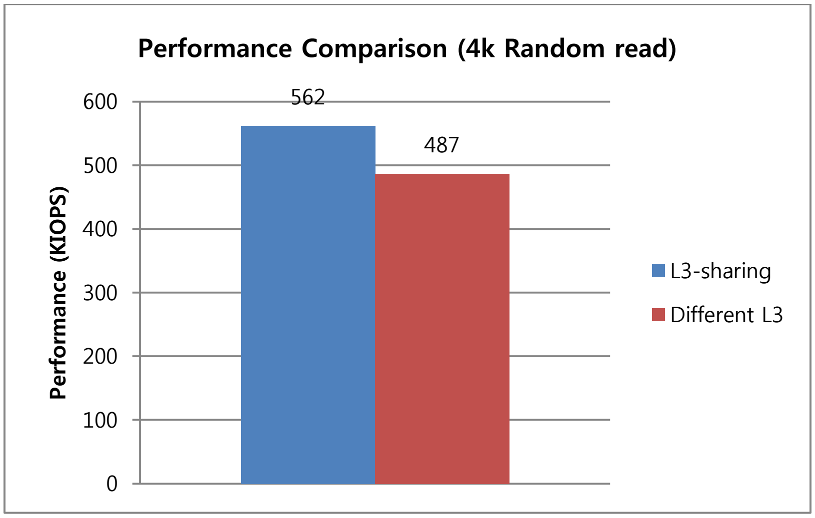


Figure 14. Performance comparison by L3 Cache

NUMA-aware / L3 cache sharing	NUMA-aware / NO L3 cache sharing
`{ "ssd_type" : "lba", "device_description" : [ { "dev_id" : "0000:05:00.0", "core_mask" : 1, "cq_thread_mask" : 2 } ] }`	`{ "ssd_type" : "lba", "device_description" : [ { "dev_id" : "0000:05:00.0", "core_mask" : 1, "cq_thread_mask" : 8 } ] }`

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.

9.1.4 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_cli. 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.

9.2 kv_nvme_cli

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

9.2.1 Main usage

```
--------------------- kv_nvme_cli Howto ----------------------------
1) kv_nvme_cli 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

9.3 unvme_rocksdb (Block SSD Only)

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

9.3.1 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.

9.3.2 Build

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

./make.sh app

9.3.3 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(*dev_id*) 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>*

[Note] mkfs could not be used at the same time for multiple devices.

9.3.4 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

Several db_bench options have been added to uNVMe RocksDB (db_bench) for supporting new features (direct I/O, retain read cache, and prefetch size control) of uNVMe Blobfs. Below is the list of the options.

-***mpdk*** *(path of json configuration file) type: string default: “”*

-***use_retain_cache*** *(flag to retain blobfs readcache) type: bool default: false*

-***use_prefetch_ctl*** *(flag to control prefetch size) type: bool default: false*

-***prefetch_threshold*** *(blobfs prefetch(readahead) threshold) type: int32 default: 131072*

-***use_blobfs_direct_read*** *(flag to use blobfs direct read) type: bool default: false*

-***use_blobfs_direct_write*** *(flag to use blobfs direct write) type: bool default: false*

-***use_manual_schedule_io*** *(If this flag sets, threads of generating KV workloads are scheduled in order from core 0. Otherwise, scheduled by kernel automatically) type: bool default: false*

-***use_manual_schedule_bg*** *(If this flag sets, threads of background flush/compactions are scheduled in order from core 0. Otherwise, scheduled by kernel automatically) type: bool default: false*

For more details about uNVMe Blobfs cache operations, please see uNVMe BlobFS Read Cache.

[Note] db_bench could not be used at the same time for multiple devices.

9.3.5 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 Block SSD at the moment, not for Samsung KV SSD.

[Note] Fuse could not be used at the same time for multiple devices.

9.4 kv_perf

kv_perf is a CLI-based tool used for generating a variety of I/O workload to SSDs 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.)*

9.4.1 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: Block 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 *kv_sdk_init*.

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

Below is a list of sample commands.

(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* *}* *}*	$ ./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)

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.
- 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. (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.

9.4.2 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

9.4.3 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*

9.5 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

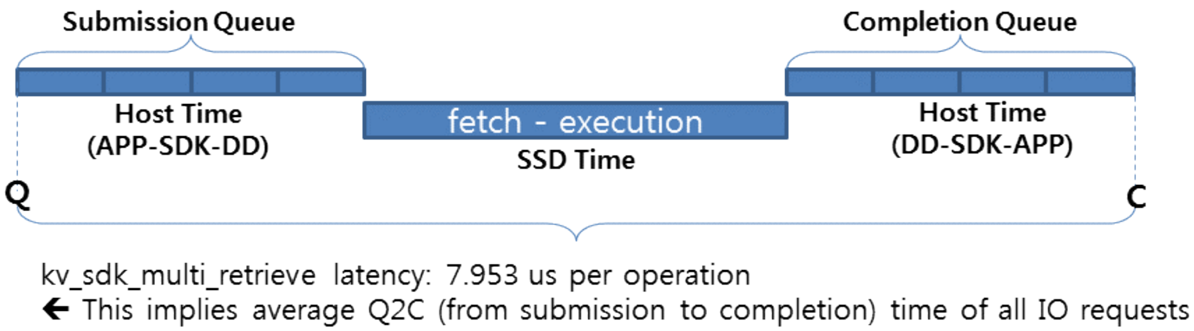
- to define configuration json file and its options
- to access single/multiple uNVMe SSDs from single/multiple threads
- to map CPU cores / IO threads / uNVMe SSDs
- to show device capacity and utilization percentage
- to enable uNVMe cache
- to implement async callback function and private data
- to 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.

9.5.1 Details of Performance Report Message

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.



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
<u>kv_sdk_multi_store latency: 7.953 us per operation</u>
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%)*

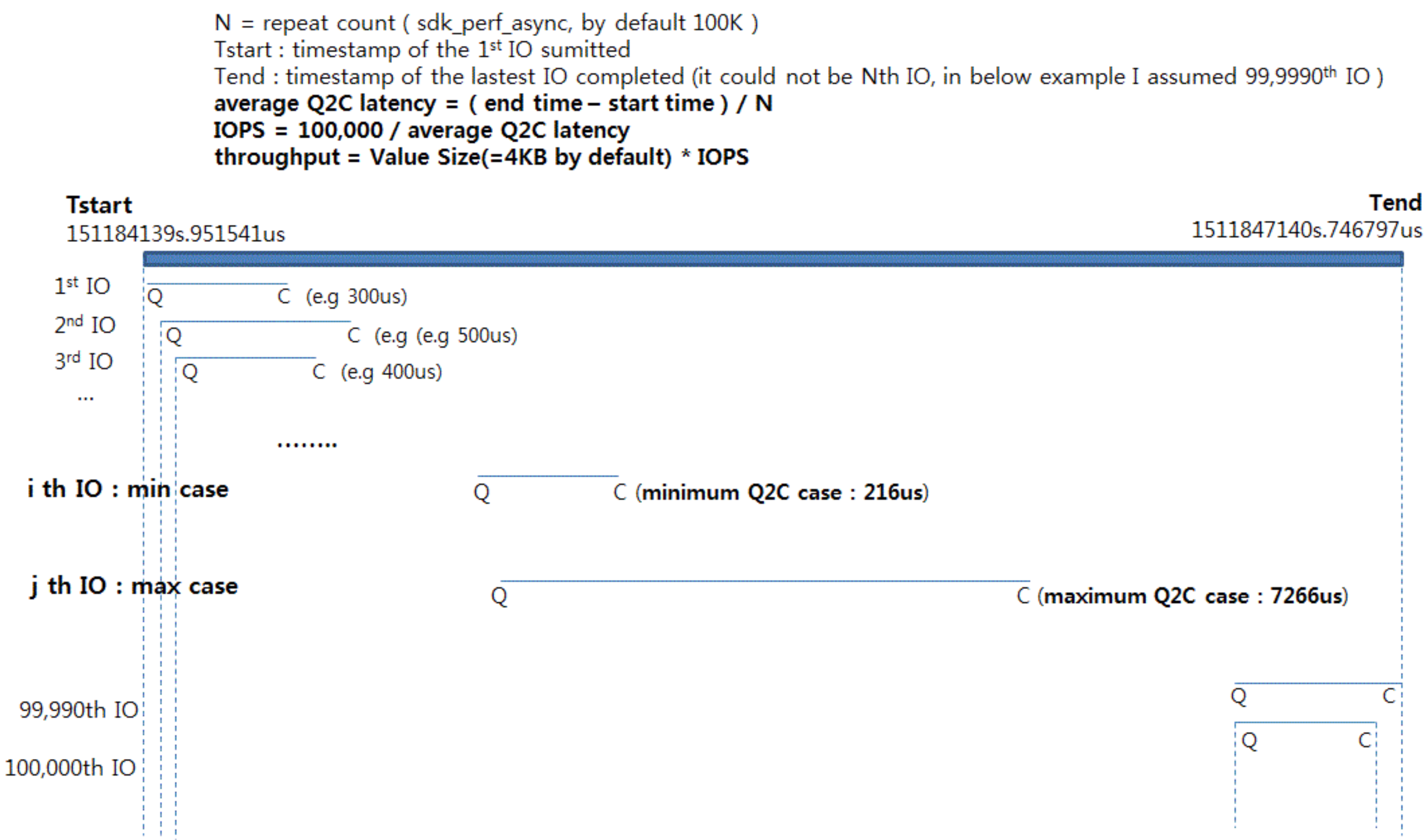


Figure 15. Result analysis of sdk_perf (sdk_perf_async)

9.6 sdk_iterate / sdk_iterate_async (KV SSD Only)

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.*
- to get information of iterator opened. See *kv_iterate_info*.
- to read(get) keys by specific prefix. For more details, see *kv_iterate_read* and *kv_iterate_read_async*.

9.7 blobfs_perf (Block SSD Only)

blobfs_perf is a CLI-based tool used for generating simple File I/O workload on BlobFS. Below is a list of options to generate some workload that blobfs_perf can process.

[NOTE] To use the blobfs_perf, users have to create bdev by *mkfs* first. Please refer to *Prerequisite* and *mkfs* on '*unvme_rocksdb'* section.

[NOTE] To use the blobfs_perf, core 0 is always compulsory. User have to set core 0 on json configuration file(at core_mask)

blobfs_perf provides the ways

- to define configuration json file and its options
- to access single/multiple BlobFS Files from single/multiple threads
- to map CPU cores / Working threads / Multi queues
- to show file size and its performance
- to enable BlobFS cache configuration (prefetch control / retain cache)
- to check miscompare by setting *g_ctx.option*

Additionally, test results of the write and read (sequential /random) operations are shown after the end of running of blobfs_perf.

```
[USAGE] ./blobfs_perf [options] ...
        --workload | -w <workload>, Specify test workload (write | read | randread), default: write and read
        --disable_rdcache | -a, Delete read cache before read, default: false
        --prefetch_ctl | -p, Apply prefetch control, default: false
        --prefetch_threshold | -f, Set threshold size of prefetch, default: 131072
        --cache_retain | -t, Retain cache after read, default: false
        --bdev_name | -u, <bdev name>, default: "unvme_bdev0n1"
        --json_path | -j, <json config file path>, default: "lba_sdk_config.json"
        --verify | -v, Verify test(valid only for read/randread workloads), default: false
        --direct | -d, Direct IO, default: false(Buffered IO)
        --use_existing_file | -e, Use written files, i.e. test_static_${core_id}, default: false
        --block_size | -b <block size in Bytes>, specify block size (B), default: 262144
        --size | -s <IO size in Bytes>, Specify io size per job, default: 1073741824
        --numjobs | -m <Number of jobs for operation>, Specify number of jobs for test, default: 2
        --nr_read_repeat | -r <Number of repeat for read operation>, default: 1
        --help | -h, Showing command menu
```

10 APPENDIX

10.1 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.

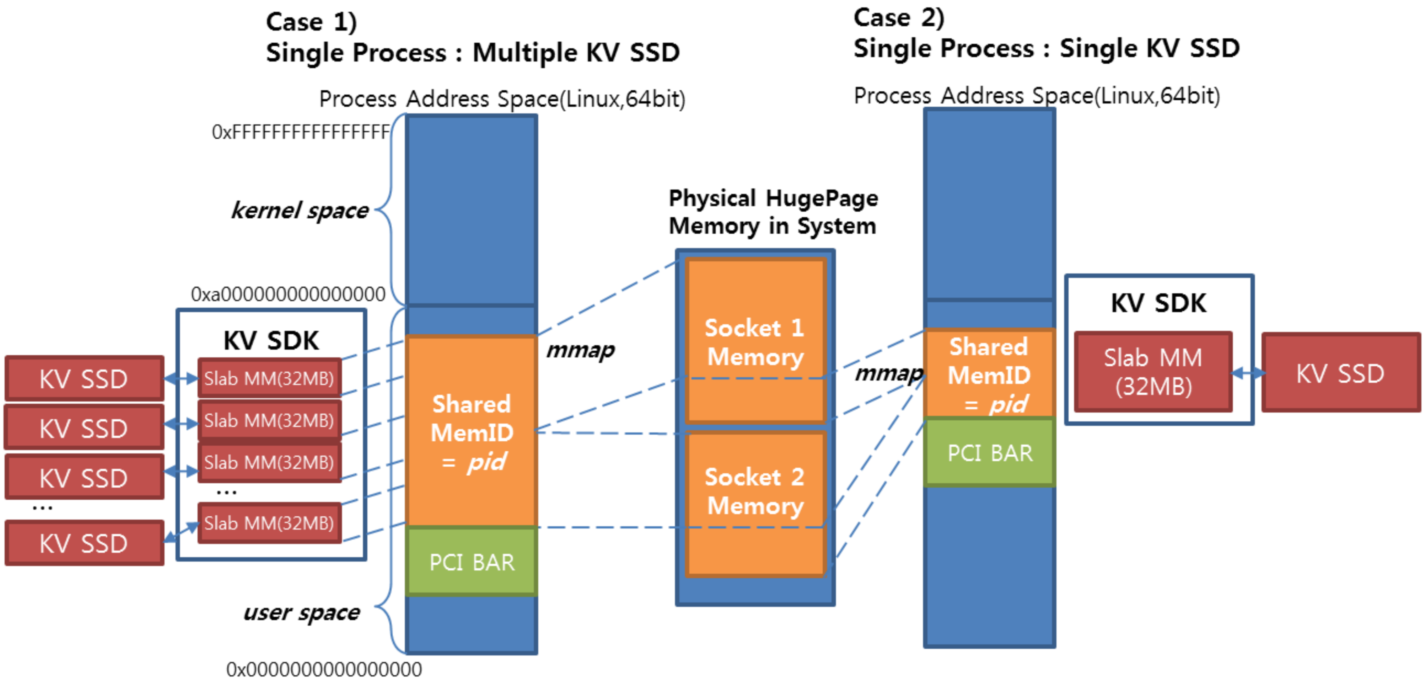


Figure 16. Use case of multi process

10.1.1 Limitation

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

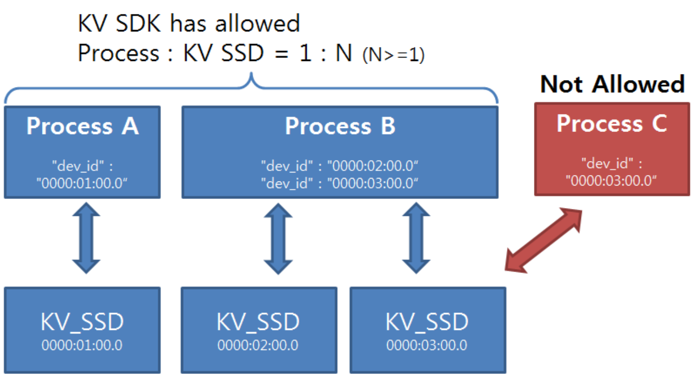


Figure 17. Limited multi process use

10.2 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 Block 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.

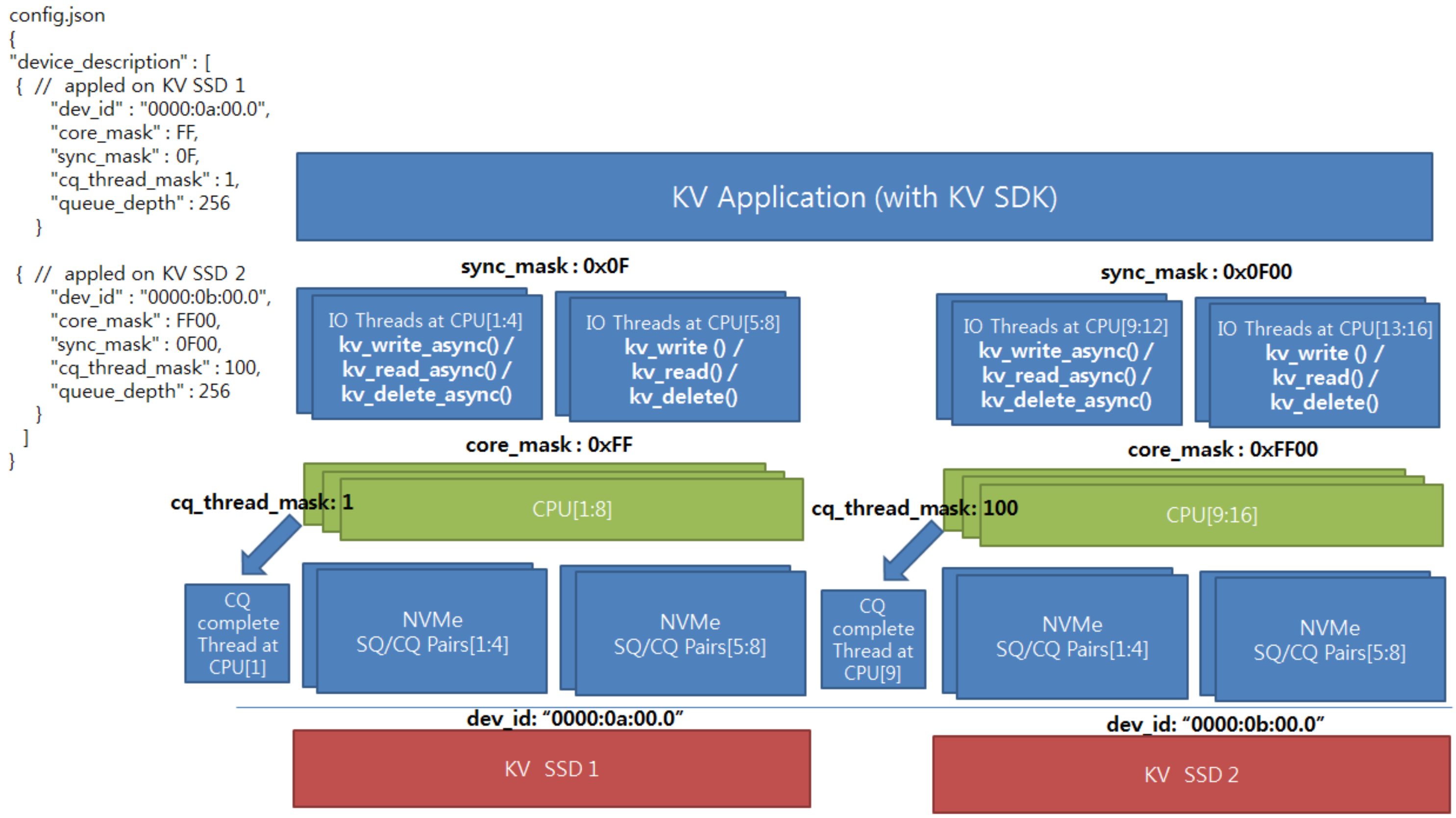


Figure 18. two KV SSDs' Description and internal mapping

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

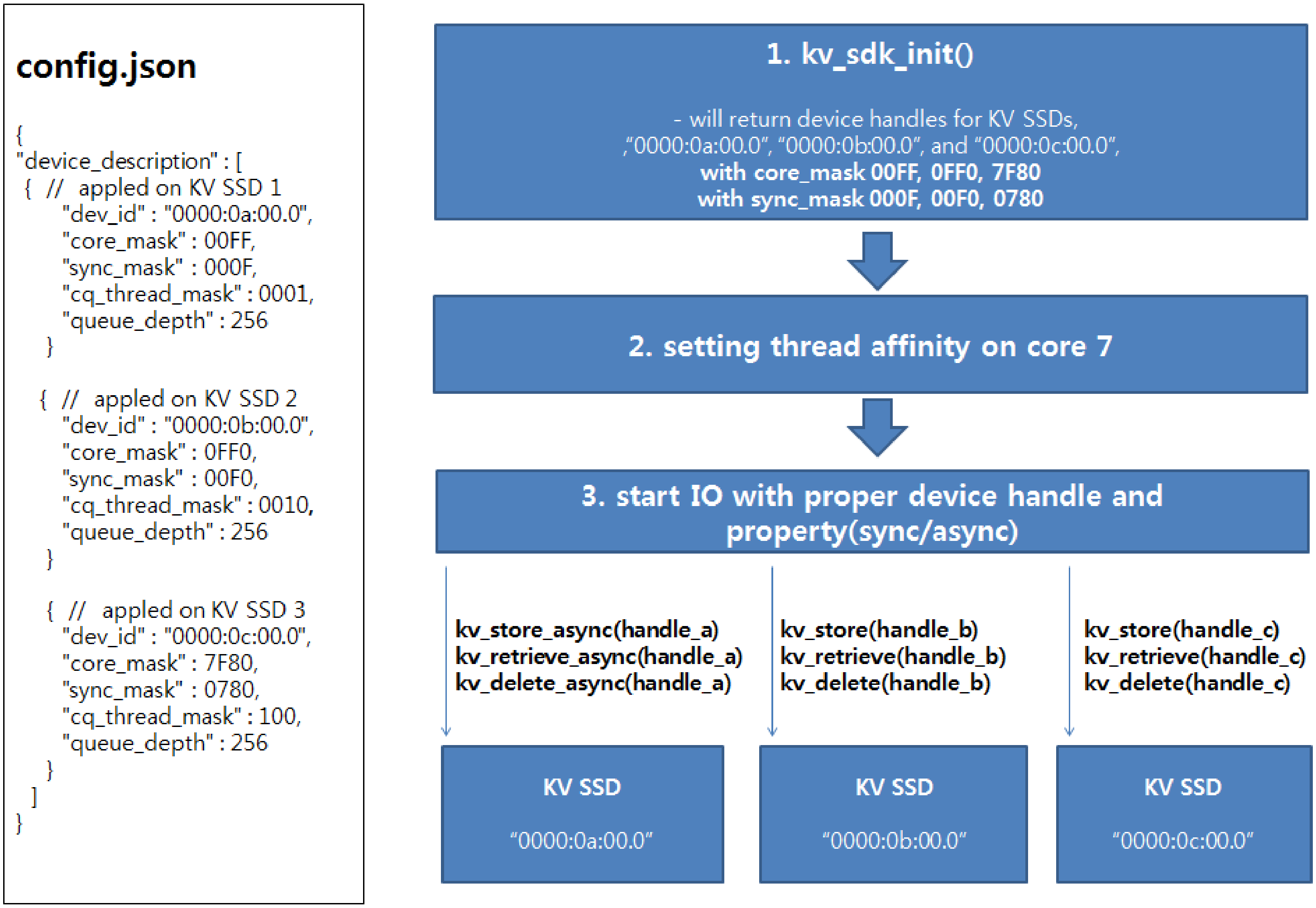
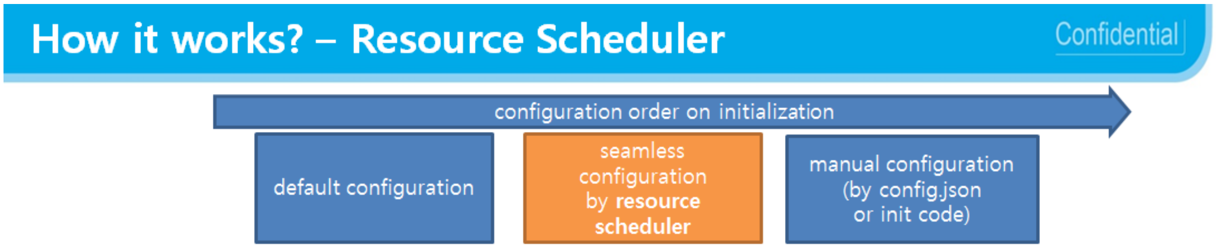


Figure 20. Diagram for description in Figure 18

However, 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 below 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 21. Resource scheduler description

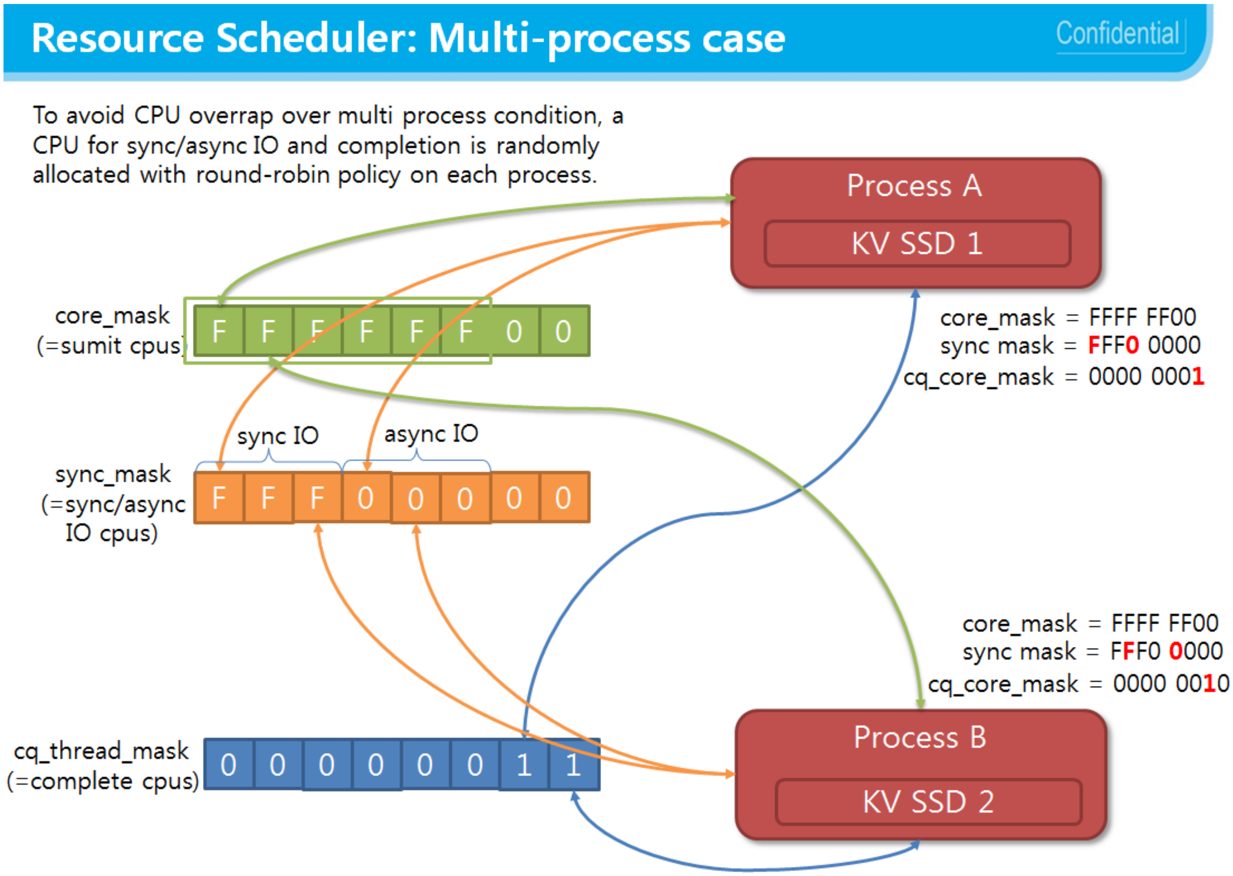


Figure 22. Resource scheduling on multi process

10.3 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.

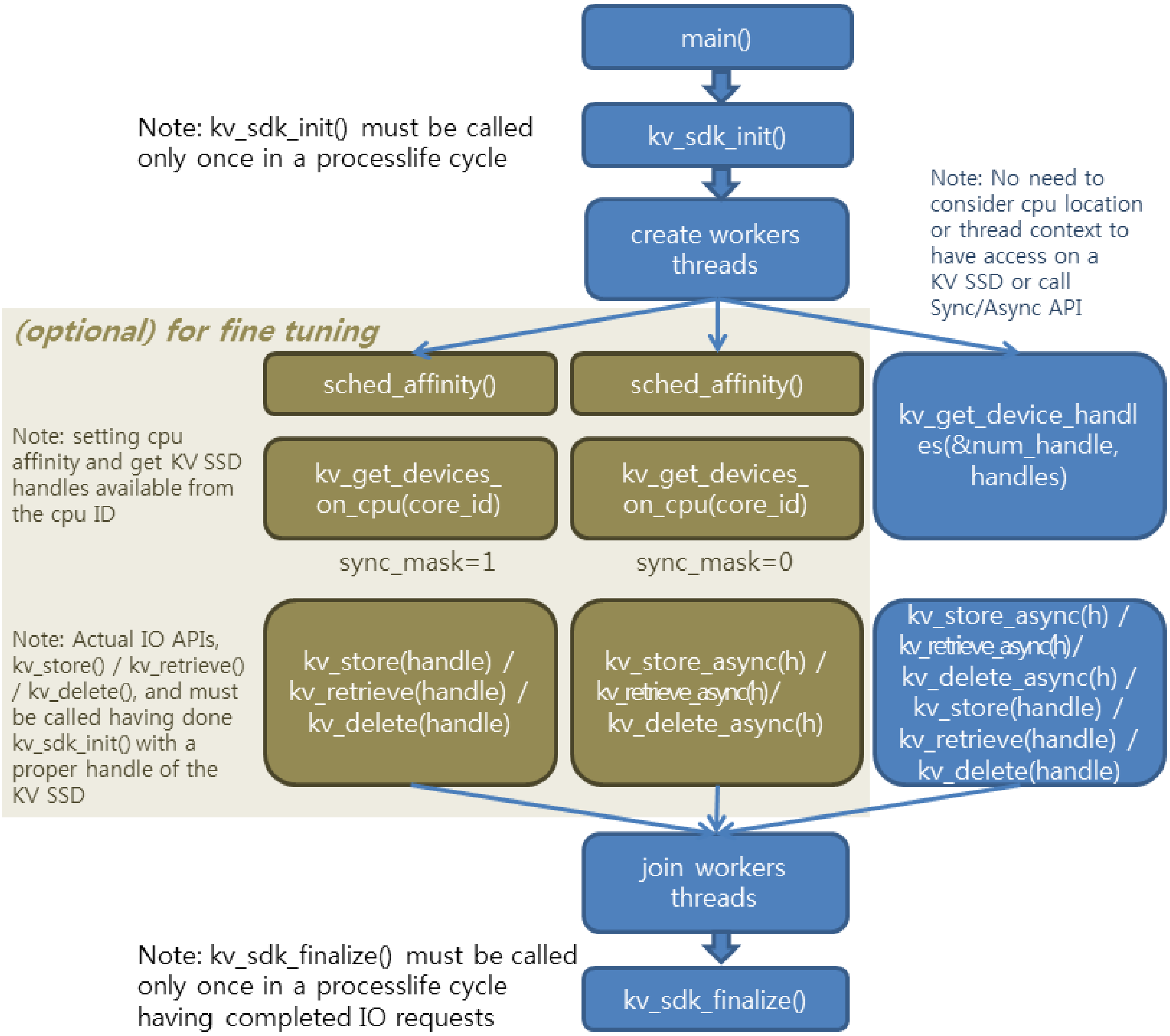


Figure 23. uNVMe App life cycle

10.4 uNVMe Cache

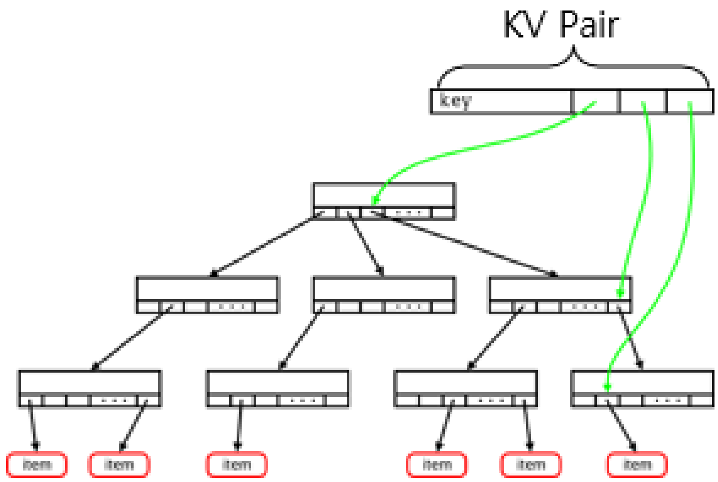


Figure 24. uNVMe cache (radix tree based)

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 kv_sdk_init and sdk_perf for more details about configuring and using the cache.

10.5 uNVMe Slab M/M

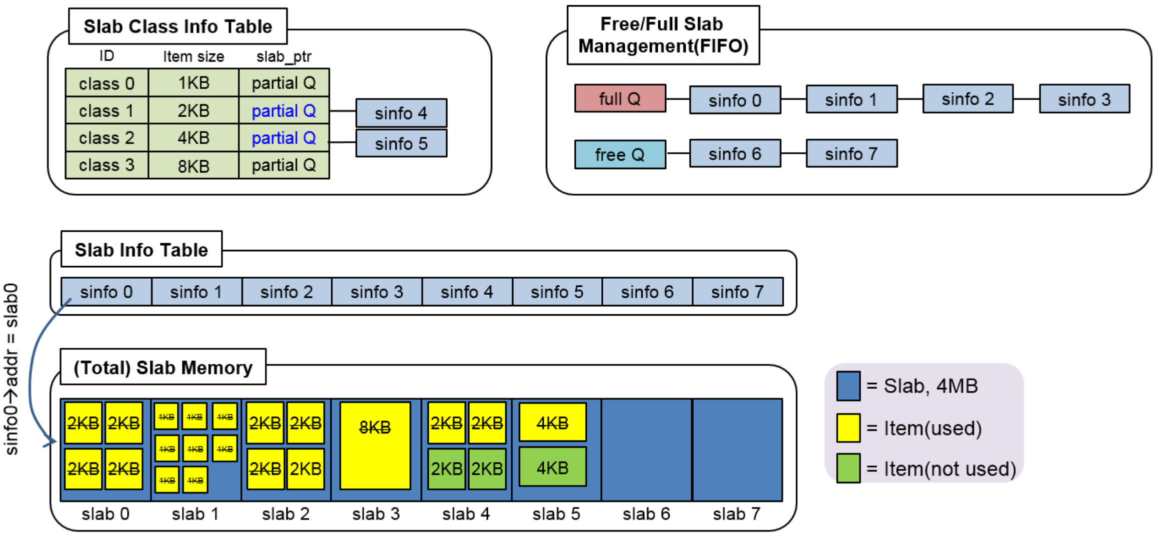


Figure 25. Slab memory management Diagram

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) 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.

10.6 uNVMe Blobfs Multi-queue Support (Block SSD Only)

10.6.1 SPDK Limitation

BlobFS and bdev layer has 3 limitations in that it is still designed to utilize single block layer queue on a dedicated CPU. Therefore,

1) Request queue contention: on every single IO request for queue manipulation.

2) Scalability: doesn't make use of multi core and multi-queue SSD capability, unlikely current kernel IO stack.

3) Remote memory access: single CPU on bdev layer forces remote memory access across CPU sockets. Hence, BlobFS is not NUMA-aware.

10.6.2 uNVMe MQ

To increase file read performance, uNVMe BlobFS MQ supports multi queue for overcoming SPDK Limitation. Therefore, uNVMe-MQ Developing multi queue block layer over SPDK BlobFS supports per CPU queue to avoid the architectural inefficiencies. As a result, around 65% gain on Seq/Rand Read workload at Blob filesystem layer (by blobfs_perf). Also, around 25% gain on Seq/Rand read workload at RocksDB layer by applying uNVMe-MQ on many test beds path.

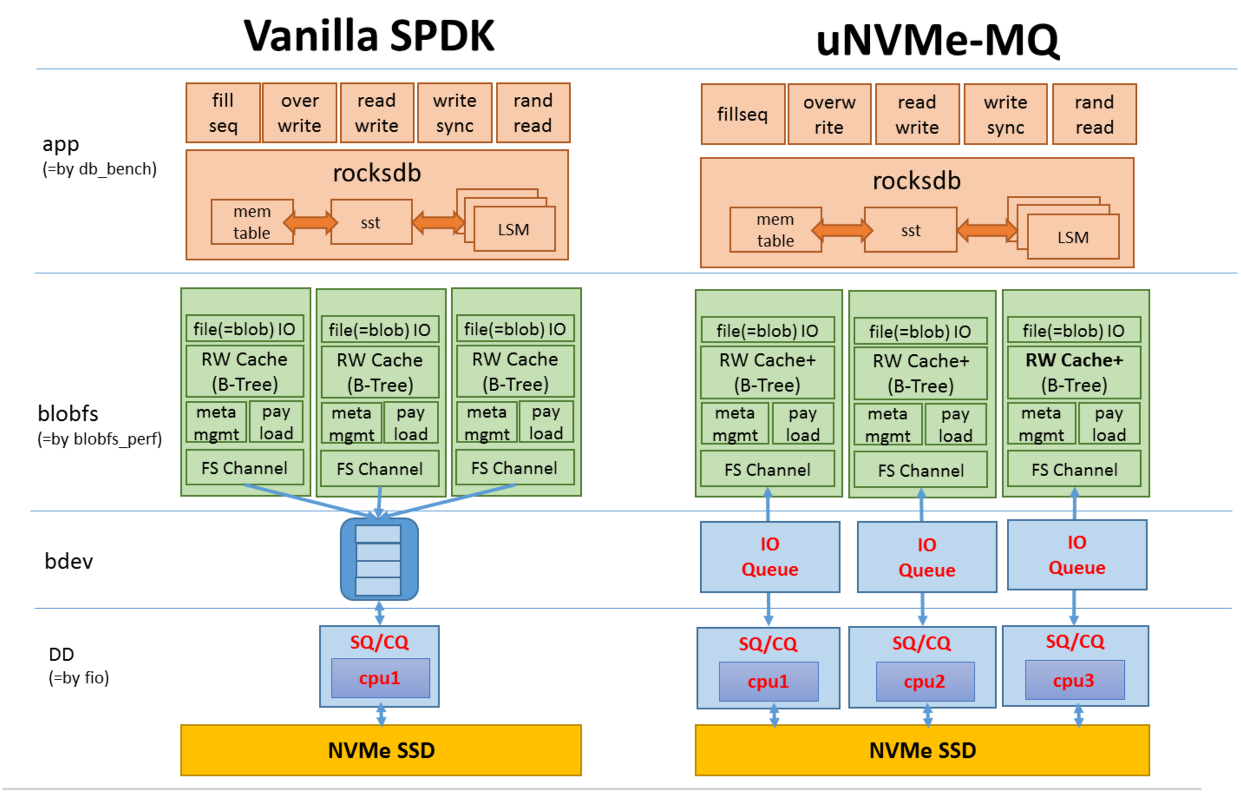
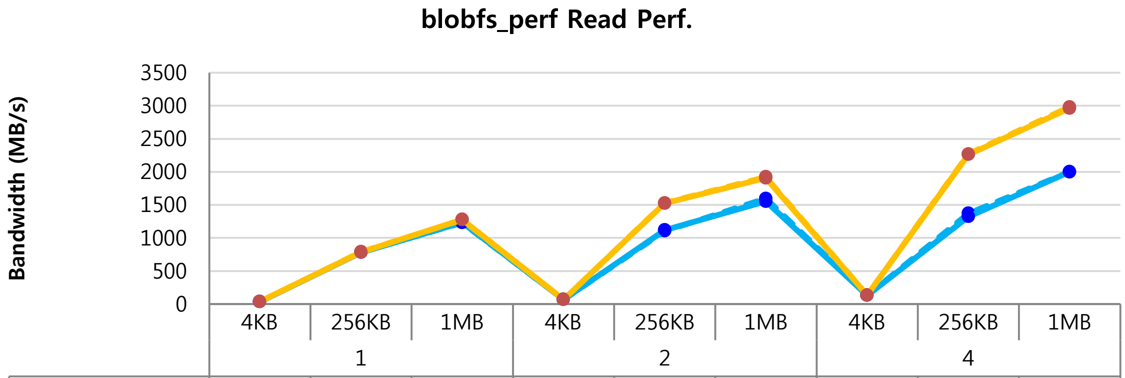


Figure 26. Difference of BlobFS between SPDK and uNVMe-MQ

Experiments



	1			2			4		
	4KB	256KB	1MB	4KB	256KB	1MB	4KB	256KB	1MB
vanilla_SR	35.8	784.4	1230.5	69.6	1105.3	1596.4	130.7	1371.3	2000.6
vanilla_RR	35.8	781.2	1233.9	69.3	1119	1552.5	132.3	1323.7	1997.1
uNVMe_SR	36.1	789.3	1273.2	69.2	1528	1923.6	134.7	2269.2	2980.3
uNVMe_RR	36.2	783.7	1278.7	69.2	1523.7	1907.2	134.4	2262.4	2960.9

Figure 27. blobfs_perf performance comparison

Test bed / configuration

Factor	Value
CPU / Max frequency	I7-7700k (4c8t) / 4.5GHz(used 0.8GHz frequency)
Device	PM1725a (1.6TB)
Memory / System Hugepage memory	16GB / 8GB
Evaluation Date	181020
OS / Kernel	Ubuntu 16.04.3 LTS / 4.9.82 (x86_64)
Buffered / Direct IO	Direct
Queue depth	64
Number of sessions	1 / 2 / 4
Total IO size	10 GB
IO size per file	Total IO size / nr_session
app_hugemem_size	12288 (12GB)
core_mask	as nr_session
chunk size	4KB / 256KB / 1024KB
use_existing_file	true
g_ctx.use_retain_cache	false
use_prefetch_ctl	false
nr_read_repeat	1
option	MISCOMPARE / DISABLE_RDCACHE

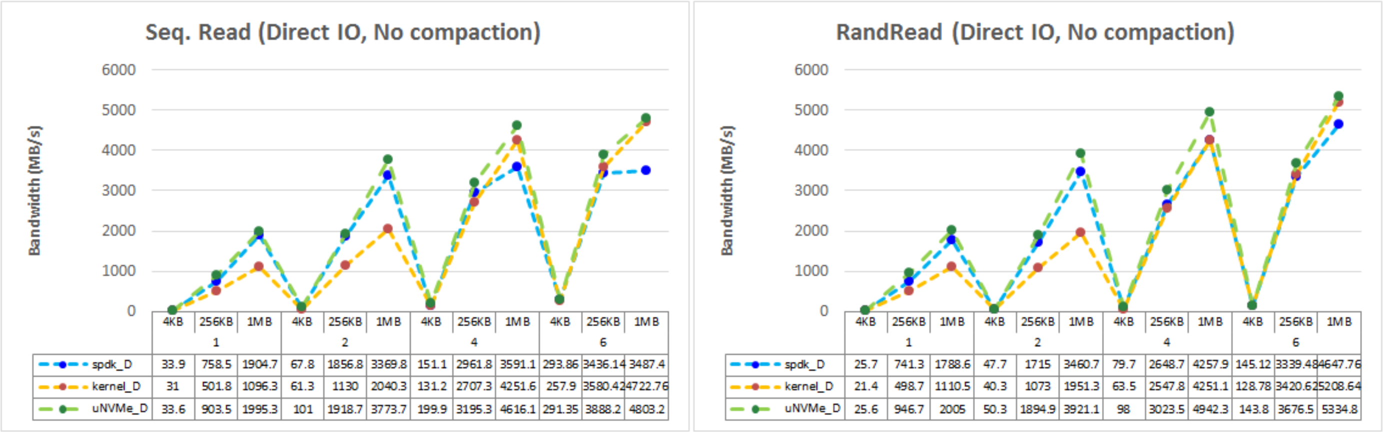


Figure 28. RocksDB Read performance comparison (No compaction)

Test bed / configuration

Factor	Value
CPU / Max frequency	I7-7700k (4c8t) / 4.5GHz (used 2.4GHz frequency)
Device	PM1725a (1.6TB)
Memory / System Hugepage memory	16GB / 8GB
Evaluation Date	181114
OS / Kernel	Ubuntu 16.04.3 LTS / 4.9.82 (x86_64)
Buffered / Direct IO	Direct
Queue depth	64
Number of sessions	1 / 2 / 4 / 6
threads	As nr_session (If workload=fillseq, --threads=1)
run_time	60
num_key	2100000 (4KB block)/ 100000 (256KB block) / 25000(1MB block)
core_mask	as nr_session
app_hugemem_size	5120
value_size	3072 (4KB block)/ 261120 (256KB block)/ 1047552(1MB block) *Block size - 1KB
block size	4KB / 256KB / 1024KB
Thread scheduling	By kernel
workload	fillseq -> seqread -> fillseq -> randread -> overwrite -> readwrite - > writesync
Memory / System Hugepage memory	16GB / 8GB
common db_bench configuration	--disable_seek_compaction=1 --mmap_read=0 --statistics=1 --histogram=1 --key_size=16 --cache_size=0 --bloom_bits=10

	--cache_numshardbits=4 --open_files=500000 --db=/mnt/rocksdb --sync=0 --compression_type=none --stats_interval=1000000 --compression_ratio=1 --disable_data_sync=0 --target_file_size_base=67108864 --max_write_buffer_number=3 --max_bytes_for_level_multiplier=10 --max_background_compactions=10 --num_levels=10 --delete_obsolete_files_period_micros=3000000 --max_grandparent_overlap_factor=10 --stats_per_interval=1 --max_bytes_for_level_base=10485760 --use_direct_io_for_flush_and_compaction=1 --use_direct_reads=1 --verify_checksum=0 --disable_wal=1 --use_existing_db=1 --mpdk=lba_sdk_config.json --spdk_bdev=unvme_bdev0n1 --spdk_cache_size=4096 --use_retain_cache=0 --use_prefetch_ctl=0 --use_blobfs_direct_read=1 --use_blobfs_direct_write=1

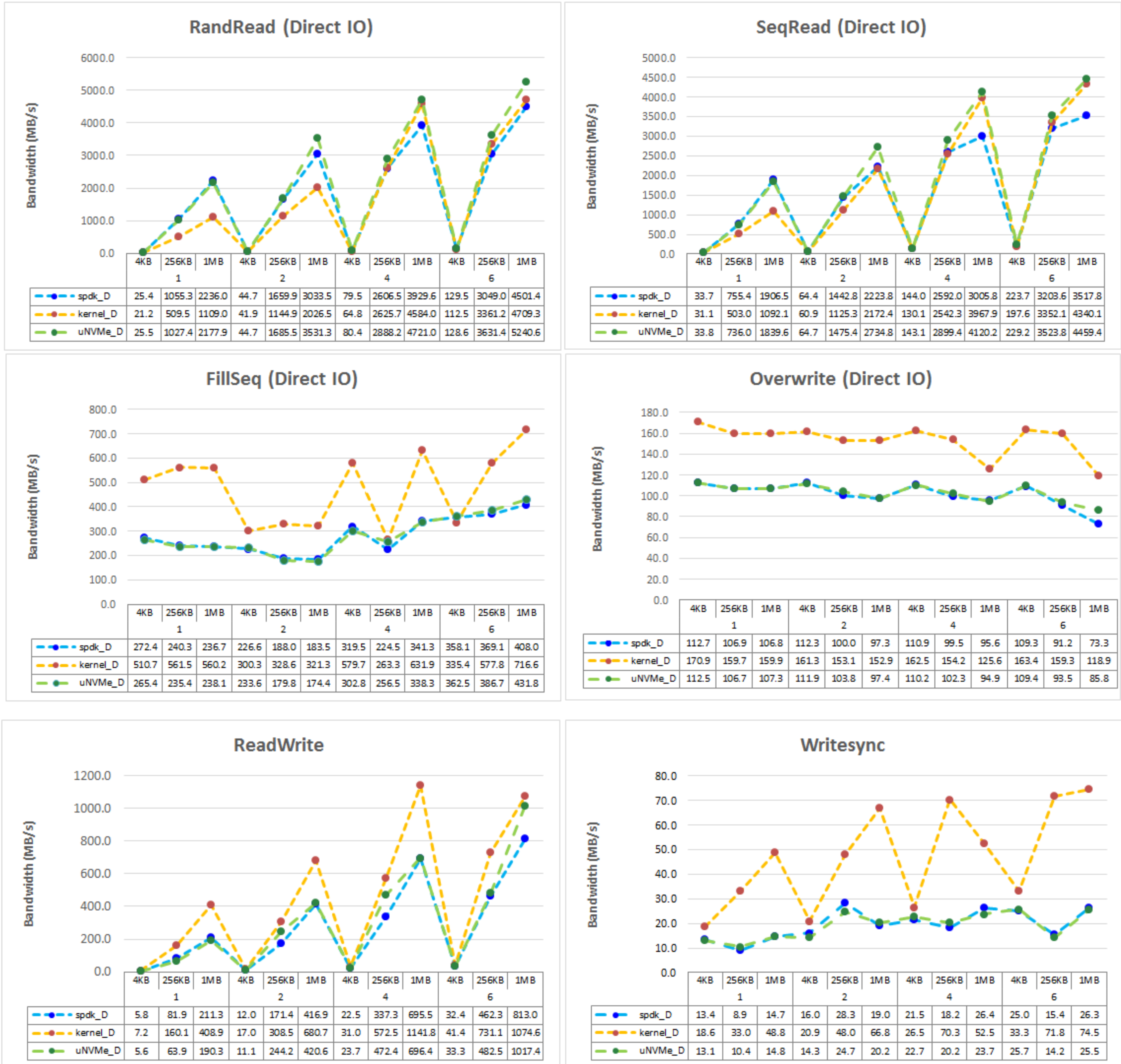


Figure 29. RocksDB performance comparison (Direct IO)

Test bed / configuration

Factor	Value
CPU / Max frequency	I7-7700k (4c8t) / 4.5GHz (used 2.4GHz frequency)
Device	PM1725a (1.6TB)
Memory / System Hugepage memory	16GB / 8GB

Evaluation Date	181114
OS / Kernel	Ubuntu 16.04.3 LTS / 4.9.82 (x86_64)
Buffered / Direct IO	Direct
Queue depth	64
Number of sessions	1 / 2 / 4 / 6
run_time	60
num_key	2100000 (4KB)/ 100000 (256KB) / 25000(1MB)
core_mask	as nr_session
app_hugemem_size	5120
value_size	3072 (4KB)/ 261120 (256KB)/ 1047552(1MB) *Block size - 1KB
block size	4KB / 256KB / 1024KB
Thread scheduling	By kernel
workload	fillseq -> seqread -> fillseq -> randread -> overwrite -> readwrite -> writesync
Memory / System Hugepage memory	16GB / 8GB
common db_bench configuration	--disable_seek_compaction=1 --mmap_read=0 --statistics=1 --histogram=1 --key_size=16 --cache_size=0 --bloom_bits=10 --cache_numshardbits=4 --open_files=500000 --db=/mnt/rocksdb --sync=0 --compression_type=none --stats_interval=1000000 --compression_ratio=1 --disable_data_sync=0 --target_file_size_base=67108864 --max_write_buffer_number=3 --max_bytes_for_level_multiplier=10 --max_background_compactions=10 --num_levels=10 --delete_obsolete_files_period_micros=3000000 --max_grandparent_overlap_factor=10 --stats_per_interval=1 --max_bytes_for_level_base=10485760 --use_direct_io_for_flush_and_compaction=1 --use_direct_reads=1 --verify_checksum=0 --disable_wal=1 --use_existing_db=1 /* 0 for fillseq only*/ --mpdk=lba_sdk_config.json --spdk_bdev=unvme_bdev0n1 --spdk_cache_size=4096 --use_retain_cache=0 --use_prefetch_ctl=0 --use_blobfs_direct_read=1 --use_blobfs_direct_write=1

[NOTE] uNVMe-MQ is applied ONLY on read I/O

10.7 uNVMe BlobFS Read Cache (Block SSD Only)

uNVMe BlobFS supports read cache to increase file read performance. Below is the list of improvements compared to the original SPDK BlobFS read cache.

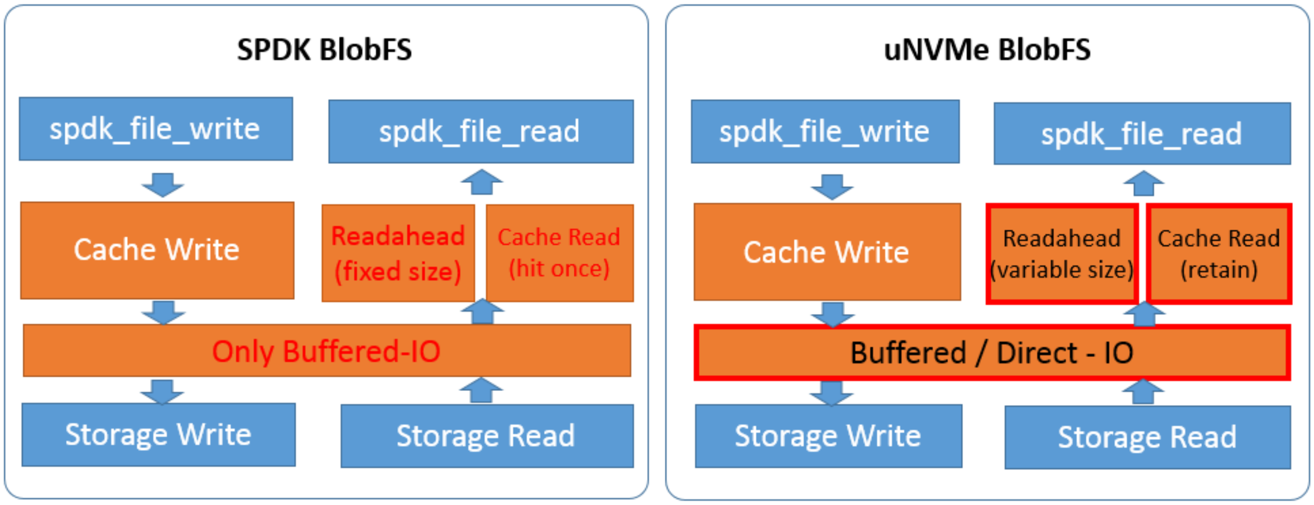


Figure 30. Difference of Read Cache between SPDK and uNVMe BlobFS

1) Readahead

SPDK BlobFS: When the amount of read requests exceeds consecutive 128KB, it is judged as 'sequential read', 512KB from next offset of the file is pre-read and stored in file cache constantly. Otherwise prefetch (readahead) operations are never triggered.

uNVMe BlobFS: provides an API to configure best-fit read prefetch size and threshold depending on IO size out of an application.

2) Cache reclaim policy

SPDK BlobFS: When a cached data hits on a read request, it is reclaimed (evicted) instantly. Thus when the read data is revisited, it always leads to disk read I/O.

uNVMe BlobFS: provides an API to determine reclaim policy whether retaining read cache or not.

3) Buffered / Direct I/O

SPDK BlobFS: supports buffered I/O only so it always occupies GB size of cache memory.

uNVMe BlobFS: provides API to support direct I/O as well as buffered I/O, as a selective way for cache supporting application.

Please refer to the 7. Blobfs API for related APIs

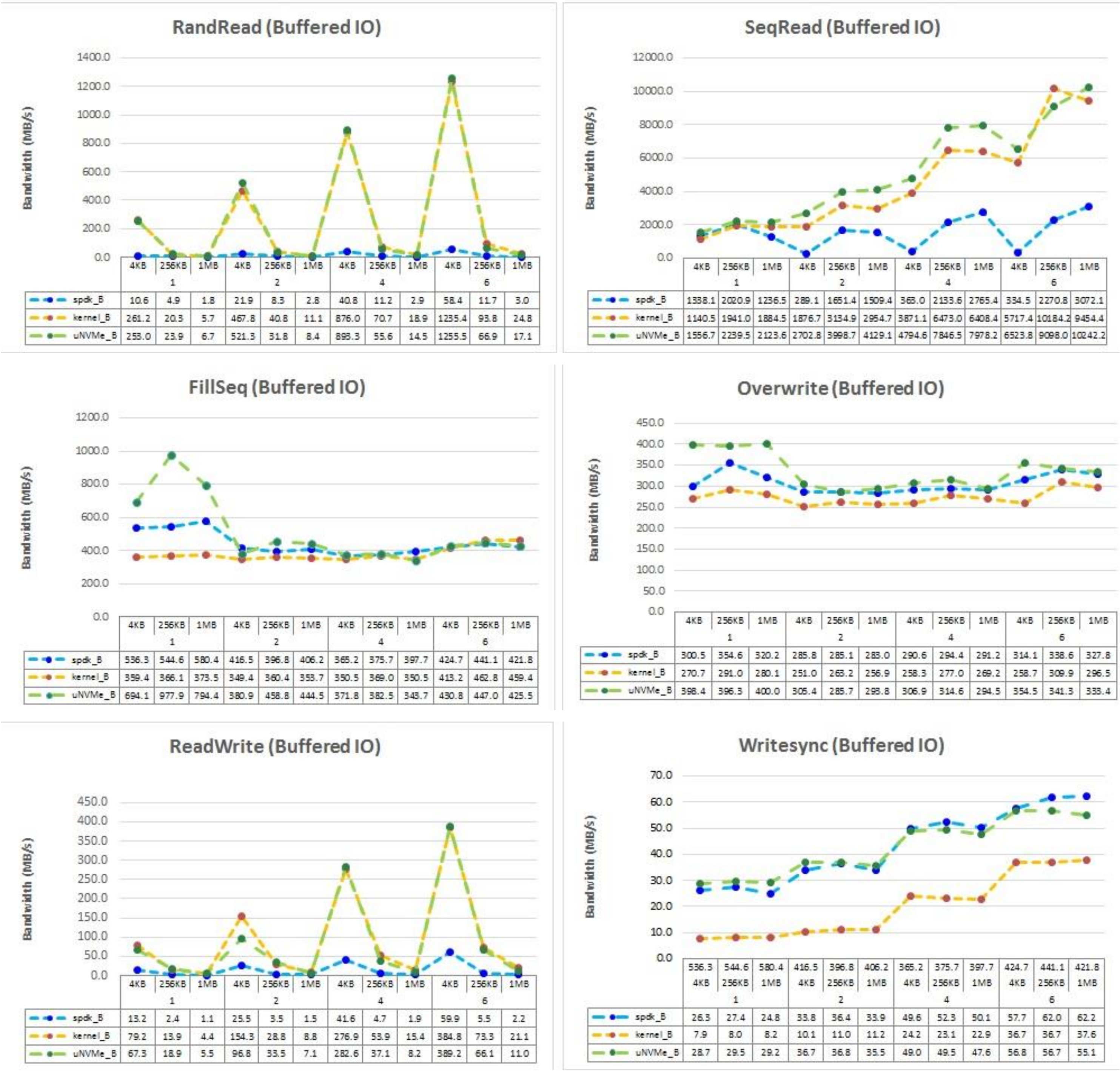


Figure 31. RocksDB performance comparison (Buffered IO)

Test bed / configuration

Factor	Value
CPU / Max frequency	Xeon E5-2667 v3 (32cores) / 3.6GHz (used max-frequency)
Device	PM983 (3.84TB)

Memory / System Hugepage memory	256GB / 16GB
Evaluation Date	181127
OS / Kernel	Debian 9.1 / 4.9.30
Buffered / Direct IO	Buffered
Queue depth	64
Number of sessions	1 / 2 / 4 / 6
run_time	60
num_key	1000000
core_mask	as nr_session
app_hugemem_size	5120
value_size	1000B
block size	4KB / 256KB / 1024KB
Thread scheduling	By kernel
workload	fillseq -> seqread -> fillseq -> randread -> overwrite -> readwrite -> writesync
Memory / System Hugepage memory	16GB / 8GB
Prefetch threshold	4KB prefetch_threshold for 4KB block_size 128KB prefetch_threshold for 256KB/1MB block_size
common db_bench configuration	--statistics=1 --histogram=1 --key_size=16 --cache_size=0 --bloom_bits=10 --cache_numshardbits=4 --open_files=500000 --db=/mnt/rocksdb --sync=0 --compression_type=none --stats_interval=1000000 --compression_ratio=1 --disable_data_sync=0 --target_file_size_base=67108864 --max_write_buffer_number=3 --max_bytes_for_level_multiplier=10 --max_background_compactions=10 --num_levels=10 --delete_obsolete_files_period_micros=3000000 --max_grandparent_overlap_factor=10 --stats_per_interval=1 --max_bytes_for_level_base=10485760 --use_direct_io_for_flush_and_compaction=0 --use_direct_reads=0 --verify_checksum=0 --disable_wal=1 --use_existing_db=1 /* 0 for fillseq only*/ --mpdk=lba_sdk_config.json --spdk_bdev=unvme_bdev0n1 --spdk_cache_size=4096 --use_retain_cache=1 --use_prefetch_ctl=1 --use_blobfs_direct_read=0 --use_blobfs_direct_write=0

10.8 Performance Guideline

10.8.1 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:

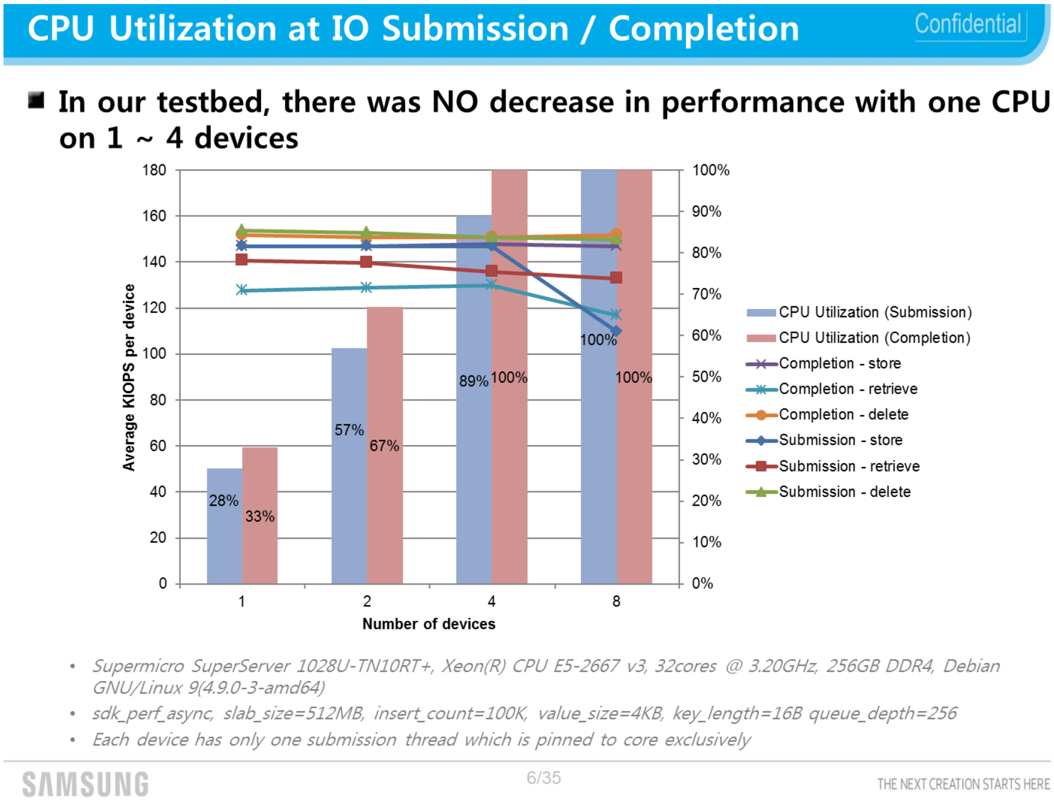


Figure 32. Number of devices (on single core)' impact on performance

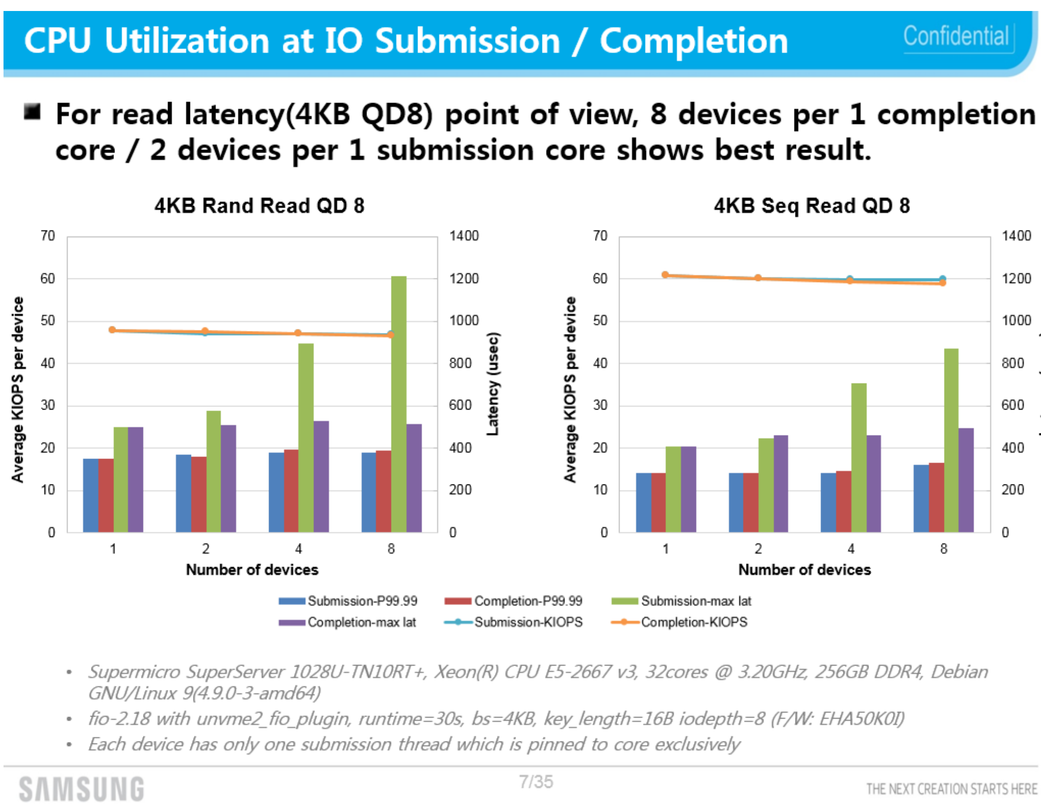


Figure 33. Number of devices (on single core)' impact on QD8 read latency QoS (P99.99, max)

(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 steeply 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.

10.8.2 Performance Comparison among Different Drivers

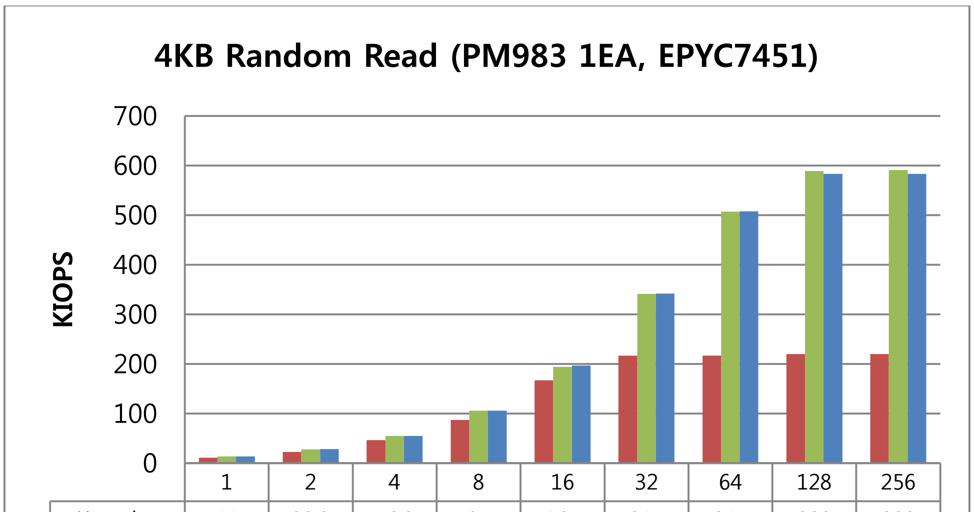
In this section, uNVMe SDK's performance advantage is showed by comparing the performance of three user level nvme drivers (SPDK, Kernel NVMe, and uNVMe SDK) measured by fio. Performance metric used for performance comparison are 4k random read.

10.8.2.1 Environment

- SSD - PM983 3.8TB (EDA53W0Q_20180315)
- Server - SMC US2023-TR4
- CPU - EPYC 7451(24c) 2P
- OS - CentOS 7.4 (Kernel 3.10)
- fio version – 3.3
- performance metric – 4KB random read (KIOPS)

10.8.2.2 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.



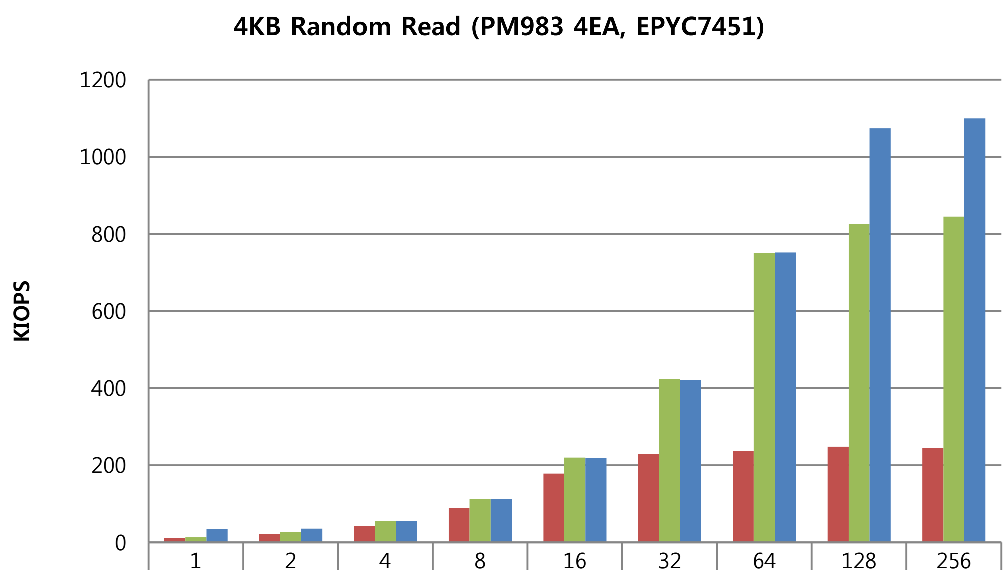
	1	2	4	8	16	32	64	128	256
Kernel	11	22.3	46.2	87	167	217	217	220	220
SPDK	13.8	27.9	54.9	106	194	341	507	589	591
uNVMe2.0	13.8	28	55	106	197	342	508	583	583

Figure 34. 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.

10.8.2.3 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 below. Each axis on the graph represents IO depth and KIOPS.



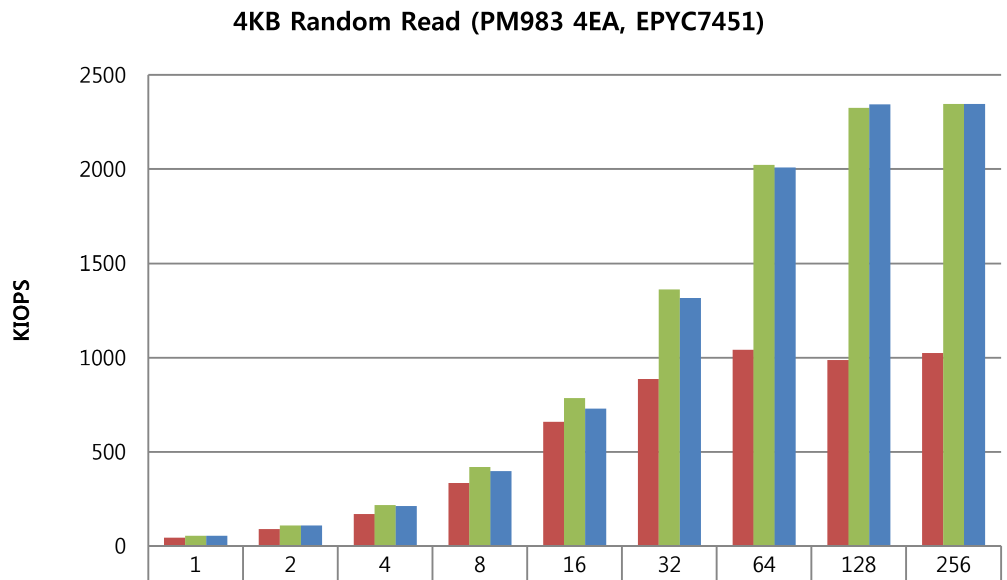
	1	2	4	8	16	32	64	128	256
Kernel	11.4	22.6	43.6	90.3	179	230	237	248	245
SPDK	13.8	27.6	55.5	112	220	424	751	826	845
uNVMe2.0	35	35.7	55.6	112	219	421	752	1074	1100

Figure 35. 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.

10.8.2.4 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 configured by configuration file. 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 36. Performance comparison – Core : Device = 4:4

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

10.9 Known Issues

10.9.1 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 37. 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 38. Initialization 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
```

10.9.2 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 18.04.1 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.
'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.
Segmentation fault when dbbench 'use_plain_table'	If the "db_bench" is run using the "PlainTable SST format", observed segmentation fault.
Errors while "run_flash_bench.sh"	While running db_bench workload using "./tools/run_flash_bench.sh" - observed few errors : [with vanilla spdk] a) ERROR: unknown command line flag 'compaction_measure_io_stats' b) Segmentation fault : after removing flag 'compaction_measure_io_stats' [with uNVMe2.0] a) blobstore.c: 841:_spdk_blob_load_cpl: ERROR: Metadata page 2 crc mismatch b) io_channel.c: 124:spdk_allocate_thread: ERROR: Double allocated SPDK thread
Segmentation fault	db_bench run_tests.sh fails during overwrite test with seg fault with below config.

during “overwrite” with some config	[overwrite_flags.txt] --disable_seek_compaction=1 --mmap_read=0 --statistics=1 --histogram=1 --key_size=16 --value_size=1000 --block_size=4096 --cache_size=0 --bloom_bits=10 --cache_numshardbits=4 --open_files=500000 --verify_checksum=0 --db=/mnt/rocksdb --sync=0 --compression_type=none --stats_interval=1000000 --compression_ratio=1 --disable_data_sync=0 --target_file_size_base=67108864 --max_write_buffer_number=3 --max_bytes_for_level_multiplier=10 --max_background_compactions=10 --num_levels=10 --delete_obsolete_files_period_micros=3000000 --max_grandparent_overlap_factor=10 --stats_per_interval=1 --max_bytes_for_level_base=10485760 --use_direct_io_for_flush_and_compaction=0 --use_direct_reads=0 --verify_checksum=0 --disable_wal=1 --use_existing_db=1 --spdk_bdev=unvme_bdev0n1 --spdk_cache_size=4096 --use_retain_cache=1 --use_prefetch_ctl=1 --use_blobfs_direct_read=0 --use_blobfs_direct_write=0 --benchmarks=overwrit --threads=1 --duration=36000 --disable_wal=1 --use_existing_db=1 --num=1000000000 --mpdk=lba_sdk_config.json --spdk_bdev=unvme_bdev0n1 --spdk_cache_size=4096 --use_retain_cache=0 --use_prefetch_ctl=0 --use_blobfs_direct_read=0 --use_blobfs_direct_write=0 --prefetch_threshold=131702
Longevity run failed in 'writesync'	Longevity run test(db_bench) is failed after 72+ hours of IOs during workload running 'writesync'(segmentation fault).