Xilinx Storage Services

Jamon Bowen
Product Marketing Director

SmartSSD® CSD

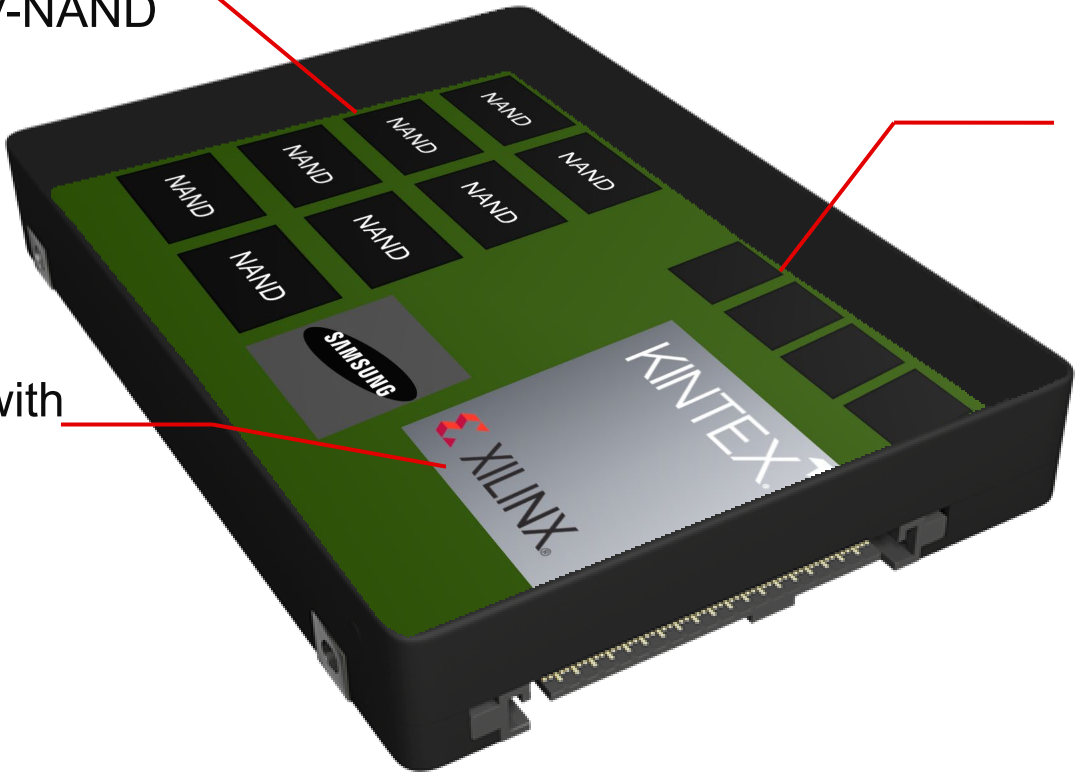
From the outside it looks just like a standard NVMe SSD

XILINX

SmartSSD® CSD

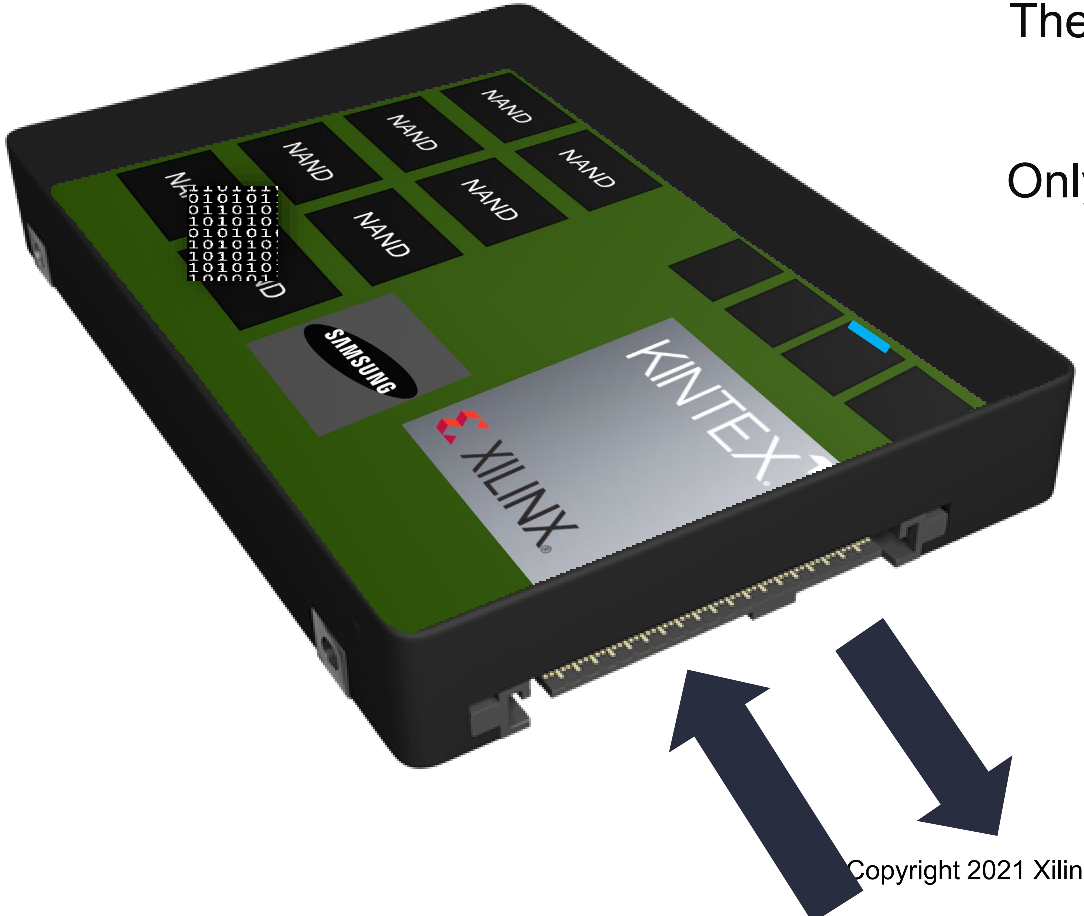
4 TB
5th Generation
Samsung V-NAND

4 GBs accelerator
memory



Xilinx FPGA with
customizable
accelerator

SmartSSD® CSD



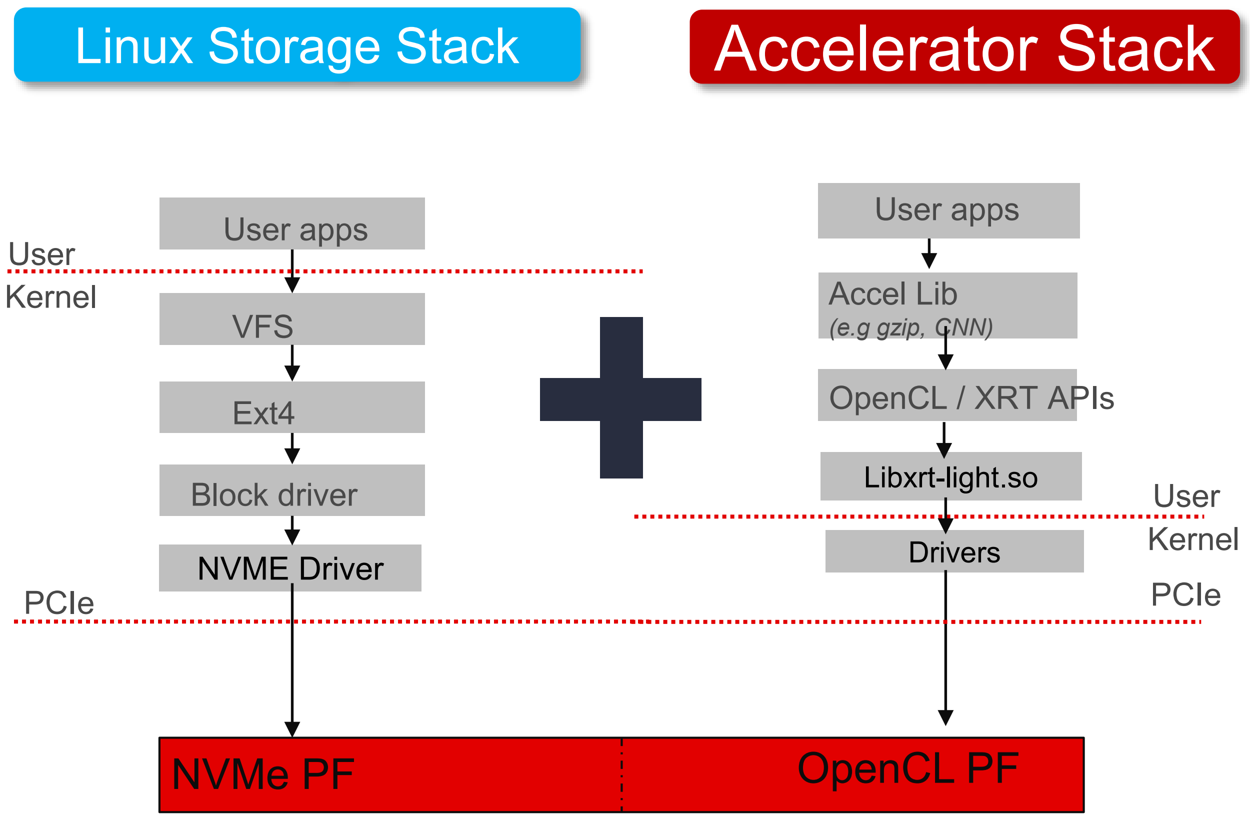
A data processing command is sent

The data is locally processed

Only the processed results are returned

Computational Storage API

Runtime Stack



- Storage Accessed via NVMe Stack
- Computational Storage / Accelerator Discovered, Managed, Orchestrated by XRT Stack
- Shared Memory Space in the Compute Function Glues the Datapaths together

XILINX

P2P example (Open CL)

https://xilinx.github.io/XRT/master/html/p2p.html#p2p-data-transfer-between-fpga-card-and-nvme-device

OpenCL coding style

Typical coding style

1. Create P2P buffer
2. Map P2P buffer to the host space
3. Access the SSD location through Linux File System, the file needs to be opened with *O_DIRECT*.
4. Read/Write through Linux *pread*/*pwrite* function

```
// Creating P2P buffer
cl_mem_ext_ptr_t p2pBOExt = {0};
p2pBOExt.flags = XCL_MEM_EXT_P2P_BUFFER;
p2pBO = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_EXT_PTR_XILINX, chunk_size, &p2pBOExt, NULL);
clSetKernelArg(kernel, 0, sizeof(cl_mem), p2pBO);

// Map P2P Buffer into the host space
p2pPtr = (char *) clEnqueueMapBuffer(command_queue, p2pBO, CL_TRUE, CL_MAP_WRITE | CL_MAP_READ, 0, chunk_size, 0, NULL, NULL,NULL);
filename = <full path to SSD>
fd = open(filename, O_RDWR | O_DIRECT);

// Read chunk_size bytes starting at offset 0 from fd into p2pPtr
pread(fd, p2pPtr, chunk_size, 0);

// Write chunk_size bytes starting at offset 0 from p2pPtr into fd
pwrite(fd, p2pPtr, chunk_size, 0)
```


XILINX

Xilinx Storage Services Reference Design

Xilinx Storage Services – Introduction

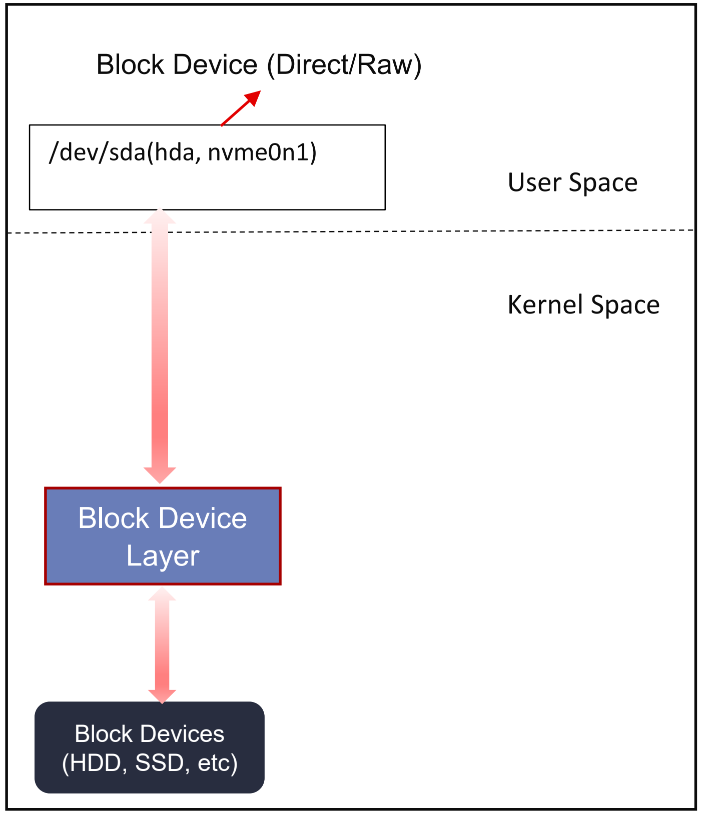
- The standard API is unaware of accelerator and SSD colocation
  - Requires application to map NVMe and Accelerators to one another.
- Many Linux Software tools run in kernel space and need to call kernel libraries
- Memory allocation overhead and FPGA program times create challenges for storage block level applications

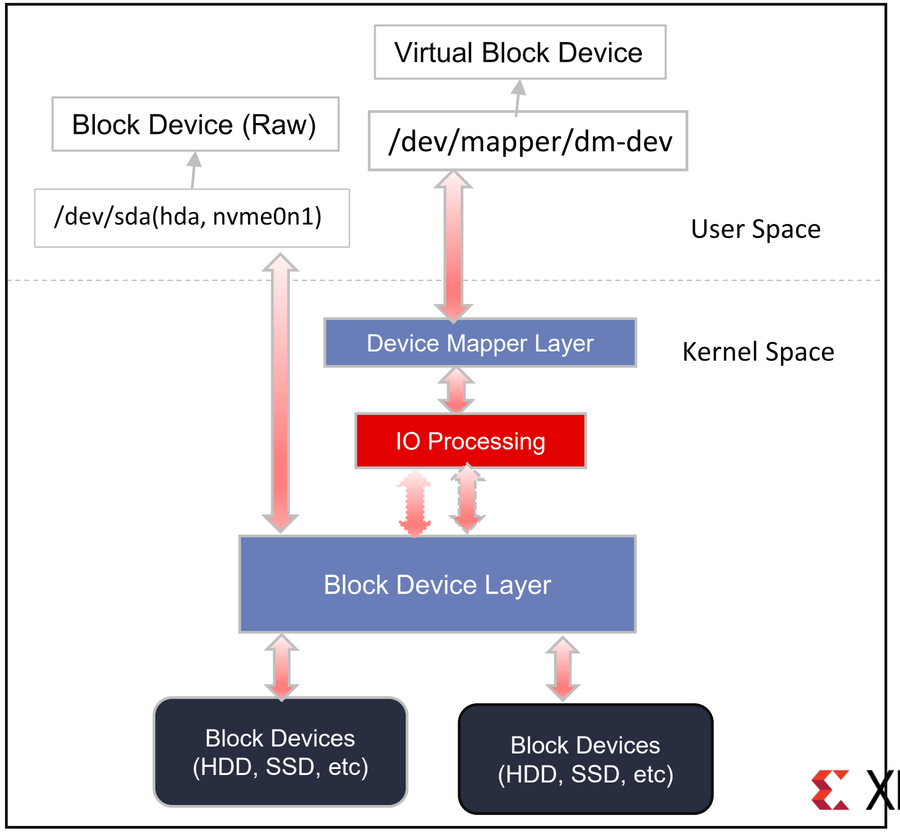
Xilinx Storage Services provide an easy-to-use API solution to accelerate storage kernel applications

XILINX.

XSS Example: Accelerating device mapper

- The device mapper is a framework provided by the Linux kernel for mapping physical block devices onto higher-level virtual block devices.
- Device mapper works by passing data from a virtual block device, which is provided by the device mapper itself, to another block device. Data can be also modified/processed in transition, which is performed, for example, in the case of device mapper providing disk encryption.





XILINX

XSS API

xss.h

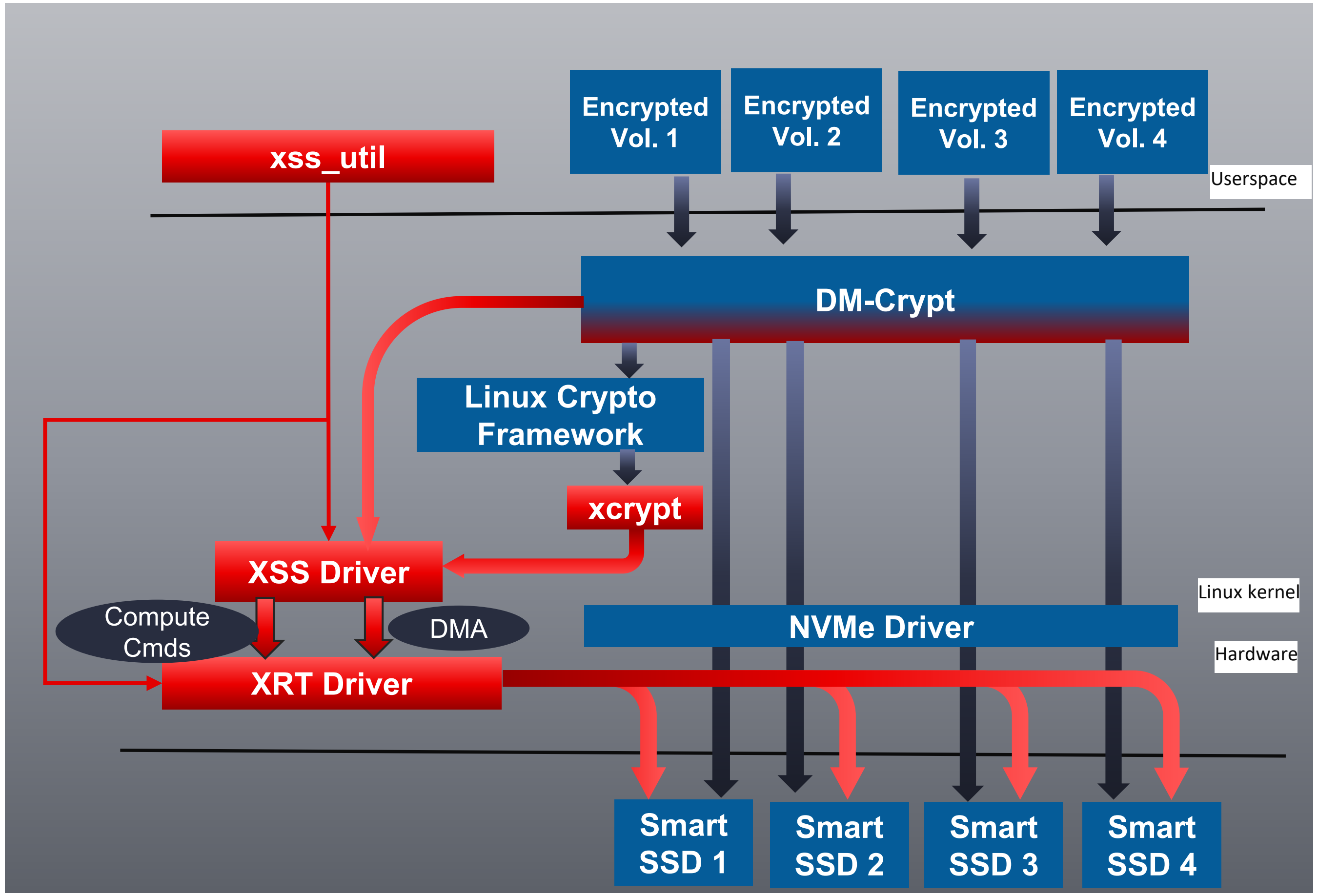
- Provides an API to kernel applications to leverage XSS
- Can be Extended to create new kernel storage applications

```
//The functions below enable allocation of p2p memory
struct page* xss_alloc_p2p_page(int ctx_id, gfp_t gfp_mask);
void xss_free_p2p_page(int ctx_id, struct page *page);

//stand alone api to just use HW aes as library to enc/dec and return results to host/ p2p buffer.
// for p2p - leverage the p2p allocated handles
int xss_crypt(int ctx_id, struct xss_cipher_req *req);

//Lists the devices that support a particular service (assumes that the xclbin is
//  programmed on the device with the service)
int xss_get_devs_with_service(int service_id, int dev_id[]);
bool xss_is_service_avail_on_dev(int dev_id, int service_id);
int xss_get_pci_addr(int dev_id, unsigned *pci_addr);
int xss_get_dev_id(unsigned pci_addr);
//For the smartSSD give the accelerator device id associated with the NVMe device.
int xss_get_peer_dev(const char * blkdev_path);
//Creates a context on the accelerator device for an application using XSS.
//  Client is a user name for the context that can be referenced in utilities
//  dev_id - accelerator device ID
//  Services - list from the supported services
//  num_services - length of the services array.
//  ret - handle to the context.
int xss_create_ctx(const char * client, int dev_id, int services[], int num_services);
void xss_delete_ctx(int ctx_id);
bool xss_is_user_ptr_p2p(struct page *page, int ctx_id);
```


XILINX.

Acceleration Example: XSS/dm-crypt

- DM-Crypt is an existing Linux solution that
  - Provides inline full disk encryption (FDE)
  - leverages Linux device-mapper layer and Linux Crypto Framework to accomplish FDE.
- XSS Driver
  - Provides easy to use APIs for kernel applications matching kernel storage library interfaces.
    - Creates accelerator contexts for applications and manages P2P buffers.
    - Manages complexity from typical in mult-accelerator SmartSSD deployments.
  - Contains services Xilinx developed.
- xss_util
  - Programs the xclbin and passes configuration information to the XSS Driver.
- xcrypt
  - Linux crypto framework module that leverages a HW accelerator on the SmartSSD using XSS.



- DM-Crypt
  - Updated to use P2P Buffers for IO using XSS.

XILINX

Xilinx Storage Services : XSS/dm-crypt Example

XSS Configuration:

1. Update /etc/xss.conf

```
#<bdf_address>  <xclbin_path>
0000:05:00.1     /home/xss/fa_aes_xts2_rtl_enc_dec.xclbin
```

2. Load xss configuration on device

```
# xss_util load-config
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Reading configuration from /etc/xss.conf:
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Configuring device 0000:05:00.1... Loaded xclbin:/home/xss/fa_aes_xts2_rtl_enc_dec.xclbin, Added device info to XSS.
```

Dm-crypt VBD Creation

1. Using dmsetup

```
# dmsetup create xss-dev --table "0 $(blockdev --getsz /dev/nvme0n1) crypt capi:xss_aes_xts_async-plain64 <128-byte-key>  0 /dev/nvme0n1 0 1 sector_size:4096"
```

2. Using Cryptsetup

```
# cryptsetup --type luks2 --cipher capi:xss_aes_xts_async-plain64 --key-size 512 --sector-size 4096 luksFormat /dev/nvme0n1

# cryptsetup luksOpen /dev/nvme0n1 xss-dev
```

Use VBD with a file-system

```
# mkfs.ext4 /dev/mapper/xss-dev

# mount /dev/mapper/xss-dev /mnt
```

Kernel application modification

- Dm-crypt – Updated to allocate buffers from P2P region of SmartSSD CSD.

```
static void *crypt_page_alloc(gfp_t gfp_mask, void *pool_data)
{
    struct crypt_config *cc = pool_data;
    struct page *page;
    struct crypto_tfm *base;
    struct xss_aes_ctx *xctx;

    if (unlikely(percpu_counter_compare(&cc->n_allocated_pages, dm_crypt_pages_per_client) >= 0) &&
        likely(gfp_mask & __GFP_NORETRY))
        return NULL;

    if (!test_bit(DM_CRYPT_USE_P2P_PAGES, &cc->flags))
        page = alloc_page(gfp_mask);
    else {
        base = crypto_skcipher_tfm(any_tfm(cc));
        xctx = crypto_tfm_ctx(base);
        page = xss_alloc_p2p_page(xctx->ctx_id, gfp_mask);
    }

    if (likely(page != NULL))
        percpu_counter_add(&cc->n_allocated_pages, 1);

    return page;
}

static void crypt_page_free(void *page, void *pool_data)
{
    struct crypt_config *cc = pool_data;
    struct crypto_tfm *base = crypto_skcipher_tfm(any_tfm(cc));
    struct xss_aes_ctx *xctx = crypto_tfm_ctx(base);

    if (!test_bit(DM_CRYPT_USE_P2P_PAGES, &cc->flags))
        __free_page(page);
    else {
        xss_free_p2p_page(xctx->ctx_id, page);
    }
    percpu_counter_sub(&cc->n_allocated_pages, 1);
}
```

Modified to have the bio buffers allocated in the P2P address space. Once that is done the reads/ writes to NVMe device happen via P2P – automatically!

Life of a Block: Write

- Application Writes to storage:

```
filename = <full path dm-crypt block device>
fd = open(filename, O_RDWR);
// Write chunk_size bytes starting at offset 0 from p2pPtr into fd
pwrite(fd, data_pointer, size, offset )
```

- BIO Layer in Linux forwards the IO to DM-crypt.

- DM-Crypt creates a crypto work item to encrypt the write data

- In dm-crypt-5.4.c:

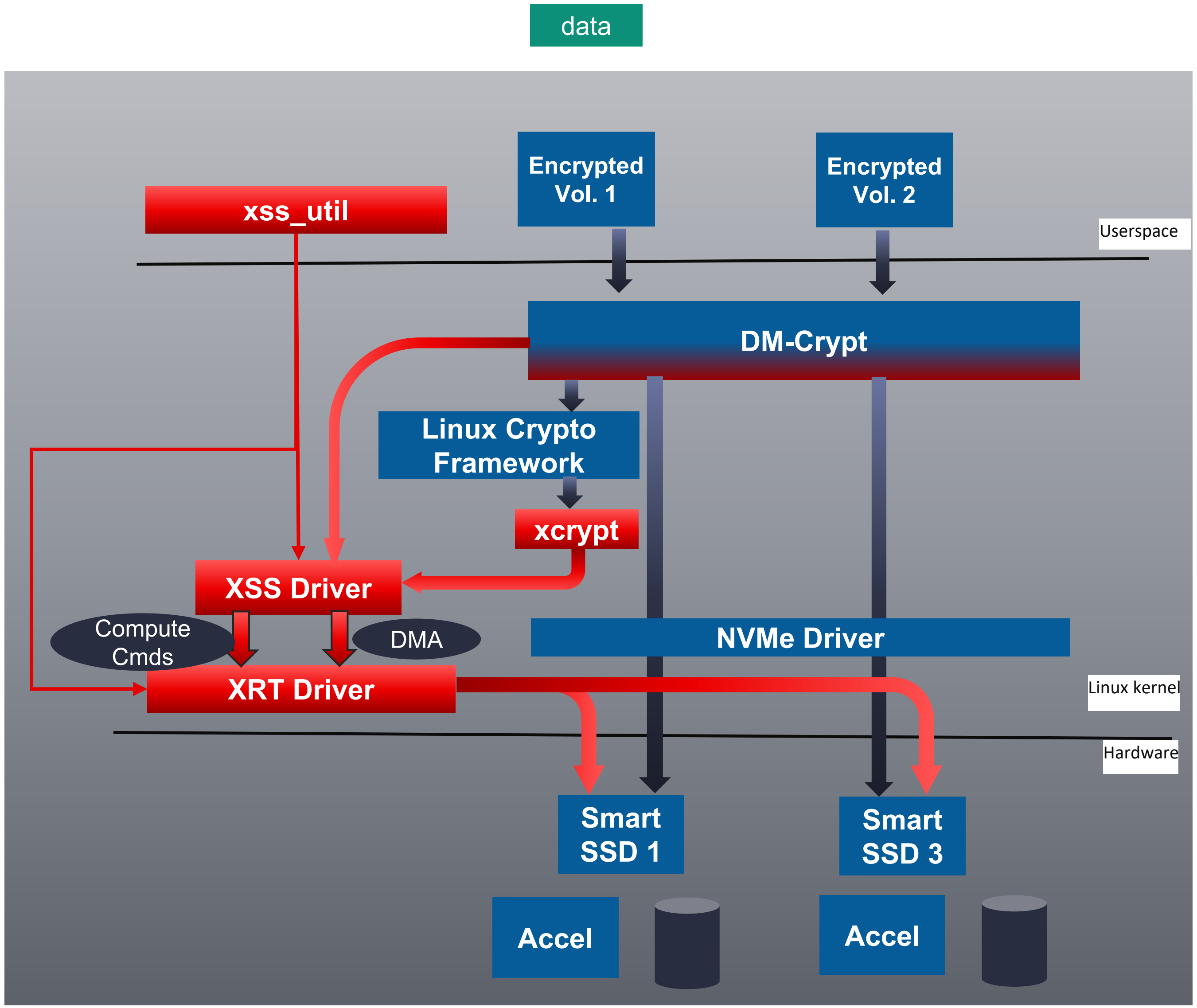
```
static int crypt_convert_block_skcipher(struct crypt_config *cc,
                                        struct convert_context *ctx,
                                        struct skcipher_request *req,
                                        unsigned int tag_offset)
{
    struct bio_vec bv_in = bio_iter_iovec(ctx->bio_in, ctx->iter_in);
    struct bio_vec bv_out = bio_iter_iovec(ctx->bio_out, ctx->iter_out);
    struct scatterlist *sg_in, *sg_out;
```

```
    if (bio_data_dir(ctx->bio_in) == WRITE)
        r = crypto_skcipher_encrypt(req);
    else
        r = crypto_skcipher_decrypt(req);
```

No modifications – calls the HW accelerated crypto based on the dm-crypt VBD specifying that module under the linux crypto framework

XILINX

Write to Linux Crypt



XILINX

Life of a Block: Write

- Xcrypt.c – accelerated linux crypto framework module. Provides all of the linux crypto framework interfaces. Actual hw accelerated work done by xss_crypt() in XSS.

```
xcrypt_requeue_request
    list : struct list_head
    req : struct skcipher_request*
    encrypt : bool
    xss_work : struct work_struct
xcrypt_handle_requeue_reqs(struct work_struct*) : void
xcrypt_enqueue_request(int, struct skcipher_request*) : void
xss_alg_skcipher_init(struct crypto_skcipher*) : int
xss_alg_skcipher_exit(struct crypto_skcipher*) : void
xss_alg_sync_skcipher_encrypt(struct skcipher_request*) : int
xss_alg_sync_skcipher_decrypt(struct skcipher_request*) : int
xss_alg_sync_skcipher_xts_setkey(struct crypto_skcipher*, const u8*, unsigned int) : int
callback(void*, int) : void
xcrypt_enqueue_request(int, struct skcipher_request*) : void
xss_alg_async_skcipher_encrypt(struct skcipher_request*) : int
xss_alg_async_skcipher_decrypt(struct skcipher_request*) : int
xcrypt_handle_requeue_reqs(struct work_struct*) : void
xss_alg_async_skcipher_xts_setkey(struct crypto_skcipher*, const u8*, unsigned int) : int
xss_crypto_algs : struct skcipher_alg[]
xss_crypt_init(void) : int
xss_crypt_exit(void) : void
module_init()
module_exit()
```

```c
static int xss_alg_async_skcipher_encrypt(struct skcipher_request *req)
{
    int ret;
    struct crypto_skcipher *cipher = crypto_skcipher_reqtfm(req);
    struct crypto_tfm *tfm = crypto_skcipher_tfm(cipher);
    struct xss_aes_ctx *xctx = crypto_tfm_ctx(tfm);

    struct xss_cipher_req xss_req;
    xss_req.src = req->src;
    xss_req.dst = req->dst;
    xss_req.encrypt = true;
    xss_req.iv = (uint8_t *)req->iv;
    xss_req.datalen = req->cryptlen;
    xss_req.datatype = SGL_TYPE;

    xss_req.xss_cb.func = (void *)callback;
    xss_req.xss_cb.data = (void *)&req->base;

    xss_req.key_len = xctx->key_len;
    xss_req.key = xctx->key;
    ret = xss_crypt(xctx->ctx_id, &xss_req);
    if(ret == -EBUSY) {
        xcrypt_enqueue_request(true, req);
    }
    return ret;
}
```

Life of a Block: Write

- xss_crypt:

```
int xss_crypt(int ctx_id, struct xss_cipher_req *cipher_req)
{
    struct xss_context *ctx = xss_get_ctx_struct(ctx_id);
    int ret=0;
    int cu_index;
    int svc_offset;
    struct xss_request *xss_req;
    struct xss_bo **bo_list;
```

```
    svc_offset = xss_service_offset_in_ctx(ctx, crypt_services[cipher_req->encrypt]);
    if (svc_offset<0)
        return svc_offset;

    cu_index = xss_get_service_cu(ctx, svc_offset, crypt_kernels[cipher_req->encrypt]);
```

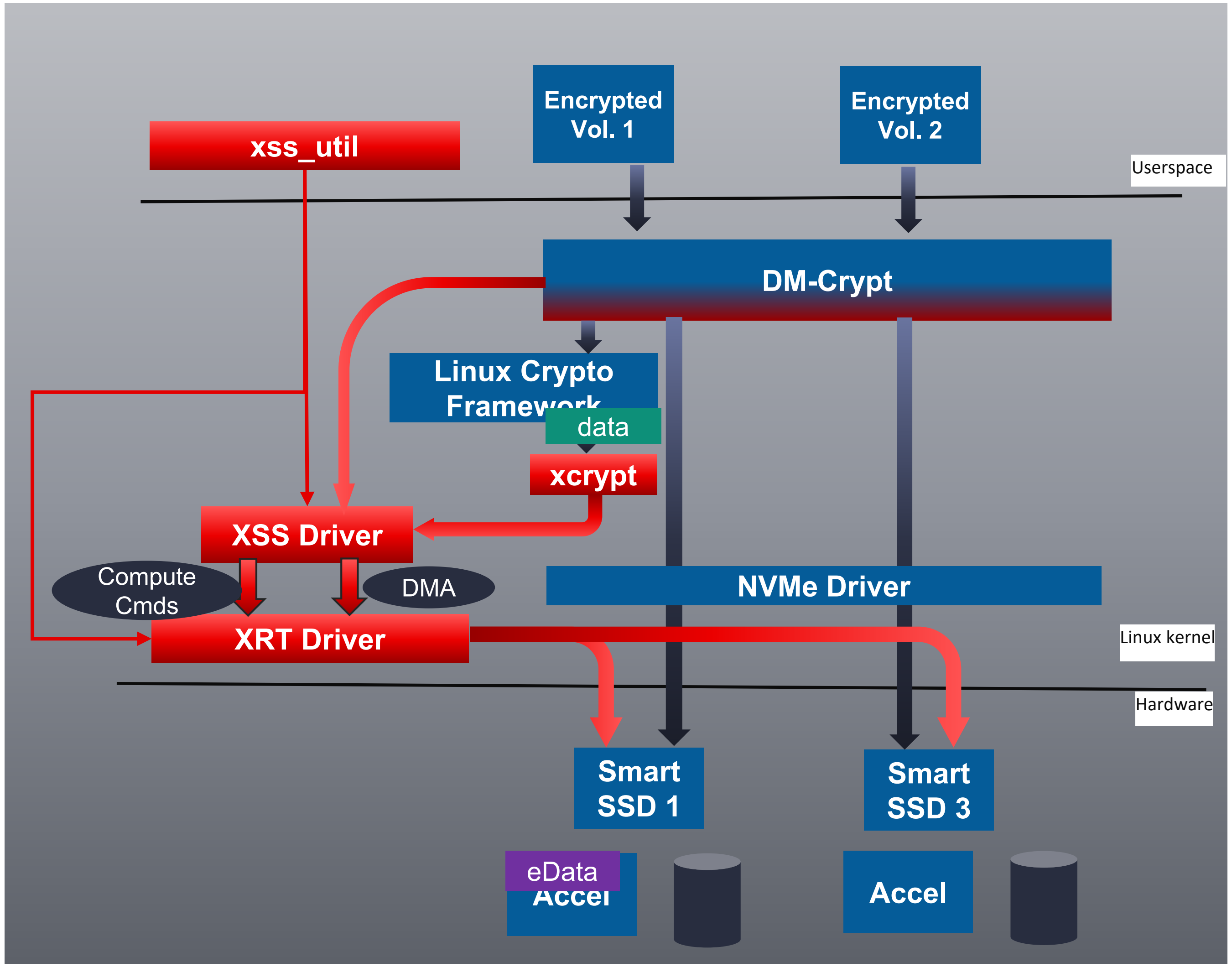
Get the Compute unit from XSS

```
    ret = crypt_setup_ert_packet(ctx, cipher_req, bo_list, cu_index);
    if (ret)
        goto out;

    xss_req->ctx        = ctx;
    xss_req->svc_id     = crypt_services[cipher_req->encrypt];
    xss_req->cu_index = cu_index;
    xss_req->bo_list    = bo_list;
    xss_req->num_bos    = CRYPT_NUM_BOS;
    xss_req->op_dir     = cipher_req->encrypt ? XSS_OP_DIR_WRITE:XSS_OP_DIR_READ;
    xss_req->complete = xss_crypt_req_cmpltn;
    xss_req->user_cb    = cipher_req->xss_cb.func;
    xss_req->user_cb_data = cipher_req->xss_cb.data;

    if ((ret = xss_submit_request(xss_req)) < 0)
        goto out;
```

Submit the encryption request to XSS with the P2P buffers.

XILINX

Write to Linux Crypt



Dm-crypt

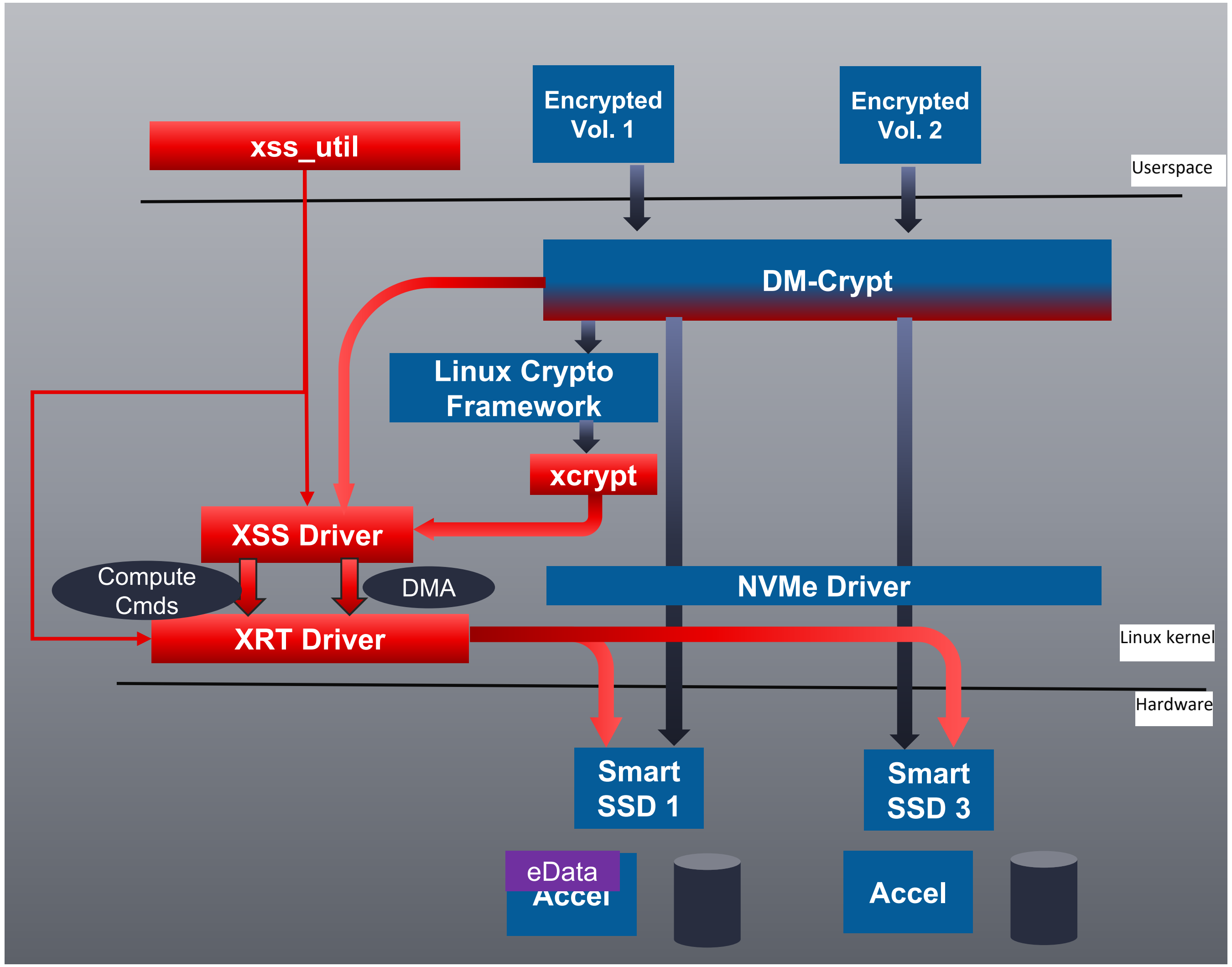
- Encrypted data written by Dm-crypt to backing storage via the BIO layer
- BIO Layer forwards the IO to NVMe device.

```
static int dmcrypt_write(void *data)
{
    struct crypt_config *cc = data;
    struct dm_crypt_io *io;

    while (1) {
        struct rb_root write_tree;
        struct blk_plug plug;

        spin_lock_irq(&cc->write_thread_lock);
continue_locked:

        if (!RB_EMPTY_ROOT(&cc->write_tree))
            goto pop_from_list;

        set_current_state(TASK_INTERRUPTIBLE);

        spin_unlock_irq(&cc->write_thread_lock);

        if (unlikely(kthread_should_stop())) {
            set_current_state(TASK_RUNNING);
            break;
        }
```

No modifications – The encrypted data input is already in a P2P buffer, so that when the write goes to the NVMe DMA it is pulled from the buffer and written to disk automatically.

XILINX.

Write to Linux Crypt



XILINX.

Next steps

- Build accelerated storage applications!
- Register for access to the Xilinx Storage Services reference design:

https://www.xilinx.com/products/intellectual-property/xss.html

XILINX

Thank You