Agenda

- Overview
- Programming Usage
- Walkthrough APIs by Example
- Specification Update
- Summary
Why Computational Storage APIs?

• Many Compute Interfaces available

• Memory based only

• SNIA includes all CSx types (CSD, CSP, CSA)

• Many Compute-Storage options

• Takes near storage compute into account

This Photo by Unknown Author is licensed under CC BY-NC-ND
SNIA CS APIs

1. One set for all CSx types
   - CSP, CSD, CSA
2. Hides Device Details
   - Hardware, Connectivity (local/remote)
   - Vendor specific Implementations
3. Abstracts Device Interface
   - Discovery
   - Access
   - Device Memory (mapped/unmapped)
   - Near Storage Access
   - Copy Device Memory
   - Download CSFs
   - Execute CSFs
   - Device Management
Programming Usage
Programming Modes

- 2 Programming Modes

  - Privileged Mode Operations (Administrator)
    - Configure CS Resources
    - Download CSFs
    - Manage Device

  - Non-privileged Mode Operations (Normal User)
    - Discover CSx, CSFs
    - Allocate Device Memory
    - Execute CSFs
    - Transfer data between Storage & Device Memory (P2P)
    - Copy data between Device Memory & Host Memory
Example

- Execute Data Filter
  1. Discovery Device
  2. Discover CSF
  3. Allocate Device Memory (FDM)
  4. Load Data from Storage
  5. Run Data Filter CSF on loaded Data
  6. Copy Results to Host Memory
Prepare for Computational Storage - Setup

1. Discover CSx
   - Discover by name
   - Access device

2. Discover CSF
   - Discover the function(s) you want to execute

3. Allocate FDM
   - Allocate Device Memory

⇒ Ready

CSx – Computational Storage Device
CSF – Computational Storage Function
Setup: Discover Device

1. Discover CSx
   - Discover CS device
     - Identify the device
       - By device path, path to file/directory
       - By list of all available CS devices
     - Skip this step if device is known
   - Access CS device
     - Request access to selected device
Setup: Discover Function

step 2

Discover CSF

- Discover your CSF
  - By Name or Global Identifier
- Returns a list of one or more
  - Each CSF instance contains
    - Relative Performance
    - Relative Power
    - Count for this instance
    - FDMs accessible
- Pick the best one if > 1 available
Setup: Allocate Device Memory

**step 3 Allocate FDM**

- Select FDM from Function chosen
  - Select FDM if > 1 available
    - Memory should be accessible by compute flow
- Allocate FDM as necessary
  - Request memory with additional details
    - Initialized (clear/fill)
    - Map to Host address space
      - If device permits
Setup: Code

1. Find CSx near storage
   ```c
   status = csGetCSxFromPath("my_file_path", &length, &csxBuff);
   if (status != CS_SUCCESS)
     ERROR_OUT("No CSx device found!
   ");
   ```

2. Get access to the CSF to run
   ```c
   status = csGetCSFId(devHandle, "filter", 0, &infoLength, &count, &csfInfo);
   if (status != CS_SUCCESS)
     ERROR_OUT("CSX does not contain any decrypt CSFs
   ");
   ```

3. Pick most performant CSF
   ```c
   CSFIdInfo *p = csfInfo;
   CSFIdInfo *myCSF = NULL;
   for (i=0; i<count; i++, p++) {
     if ((myCSF == NULL) ||
         ((myCSF != NULL) && (p->RelativePerformance > myCSF->RelativePerformance))) {
       myCSF = p;
     }
   }
   printf("CSFId: %d, RelativePerformance: %d\n", myCSF->CSFId,
   myCSF->RelativePerformance);
   ```

Allocate required size
Perform Computational Storage - Run

Load Storage Data

- Load Data near Compute

Execute CSF

- Run Compute Operation

Copy Results

- Copy from FDM into Host Memory

Done
Run: Load Storage Data

Load Storage Data

- Load data from storage into FDM
  - Data may be described as
    - LBA ranges
    - File handle & offset
  - Save on fabric bandwidth
- Data does not leave the device (P2P)
- More than one Completion options
  - Synchronous
  - Asynchronous
  - Callback or Event
Run: Execute Function

**step 5** Execute CSF

- Run compute on data loaded in FDM
  - Provide the following
    - CSF to run
    - Parameters to CSF
  - More than one Completion options
    - Synchronous
    - Asynchronous
    - Callback or Event
Run: Copy Results

step 6 Copy FDM contents to Host

- Copy Results from FDM to Host
- More than one Completion options
  - Synchronous
  - Asynchronous
    - Callback or Event
// Populate storage request with data from file

```c
CsStorageRequest storReq = malloc(sizeof(CsStorageRequest));
if (!storReq) { ERROR_OUT("not enough memory\n"); }

storReq->Mode = CS_STORAGE_FILE_IO;
storReq->DevHandle = devHandle;
storReq->u.CsFileIo.Type = CS_STORAGE_LOAD_TYPE;
storReq->u.CsFileIo.FileHandle = fd;  // file fd to access for data
storReq->u.CsFileIo.Offset = 0;  // data offset within file
storReq->u.CsFileIo.Bytes = CHUNK_SIZE;
status = csQueueStorageRequest(storReq, storReq, NULL, NULL, NULL, &compVal);
if (status != CS_SUCCESS) ERROR_OUT("Could not load storage data\n");
```

Callback for completion

// Populate compute request on data loaded from file

```c
CsComputeRequest compReq = malloc(sizeof(CsComputeRequest) + (sizeof(CsComputeArg) * 3));
if (!compReq) { ERROR_OUT("memory alloc error\n"); }

compReq->CSFId = myCSF->CSFId;  // filter function identifier
compReq->NumArgs = 3;   // function accepts 3 parameters
CsComputeArg argPtr = &compReq->Args[0];
```

// Populate copy request for results data

```c
CsCopyMemRequest copyReq = malloc(sizeof(CsCopyMemRequest));
if (!copyReq) { ERROR_OUT("memory alloc error\n"); }

copyReq->Type = CS_COPY_FROM_DEVICE;
copyReq->u.HostVAddress = results_buf;
copyReq->DevMem.MemHandle = afdmHandle2;
copyReq->Bytes = CHUNK_SIZE;
status = csQueueCopyMemRequest(copyReq, copyReq, NULL, NULL, NULL, &compVal);
if (status != CS_SUCCESS) ERROR_OUT("Could not copy data from FDM\n");
```

Event for completion

---

<table>
<thead>
<tr>
<th>Completion Type</th>
<th>Callback Parameter</th>
<th>Event Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synchronous</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td>Asynchronous - Callback</td>
<td>✓</td>
<td>NULL</td>
</tr>
<tr>
<td>Asynchronous - Event</td>
<td>NULL</td>
<td>✓</td>
</tr>
</tbody>
</table>
## API Summary

### 6 Steps

1. Discover Device (*CSx*)
2. Discover Function (*CSF*)
3. Allocate Device Memory (*FDM*)
4. Load Storage Data (*P2P*)
5. Execute Function (*CSF*)
6. Copy Results (*FDM contents to Host*)
Specification Update

- Multiple Updates
  - Include LBA Ranges
  - Advanced Device Memory usage
    - Device Memory Pools and Compute Proximity
    - Initialization Options
  - Cancelling I/O, Abort & Reset
  - Compute Function updates
    - Global Identifiers
  - NVMe Support
  - Configuration & Download updates

- SNIA CS API v0.9r1 Specification
  - Public review available
  - In SNIA membership vote towards v1.0
Summary
Summary

- Growing Compute requirements are redefining how hardware is used
- Computational Storage and APIs help build these solutions
- SNIA CS APIs v0.9r1 Specification Completed
  - Available for Public Review
- Simplified Programming Interface for CS
- Addresses NVMe CS Architecture
- Minimal steps to adopt
Backup
Complete Code

```c
// Find my CSx near storage
status = csGetCSxFromPath("my_file_path", &length, &csxBuffer);
if (status != CS_SUCCESS)
    ERROR_OUT("No CSx device found!\n");
// open device, init function and prealloc buffers
status = csOpenCSx(csxBuffer, devContext, &devHandle);
if (status != CS_SUCCESS)
    ERROR_OUT("Could not access CSx\n");

// Get access to the CSF to run
status = csGetCSFId(devHandle, "filter", 0, &infoLength, &count, &csfInfo);
if (status != CS_SUCCESS)
    ERROR_OUT("CSX does not contain any decrypt CSFs \n");

// pick highest performant CSF
CSFIdInfo *p = csfInfo;
CSFIdInfo *myCSF = NULL;
for (i=0; i< count; i++, p++) {
    if ((myCSF == NULL) ||
        ((myCSF != NULL) && (p->RelativePerformance > myCSF->RelativePerformance))) {
        myCSF = p;
        myCSF = p;
    }
}

// Pick most performant FDM from CSFIdInfo
FDMAccess *p = myCSF->FDMList;
FDMAccess *myFDM = NULL;
for (i = 0; i < myCSF->NumFDMs; i++, p++) {
    if ((myFDM == NULL) &&
        (p->RelativePerformance > myFDM->RelativePerformance))) {
        myFDM = p;
        myFDM = p;
    }
}

// allocate FDM for CSF usage
CsMemFlags f;
.f->FDMId = myFDM->FDMId;
.f->Flags = 0;  // may also be CS_FDM_CLEAR
status = csAllocMem(devHandle, CHUNK_SIZE, &f, &afdHandle1, NULL);
if (status != CS_SUCCESS)
    ERROR_OUT("AFDM alloc error\n");

// Populate storage request with data from file
CsStorageRequest *storReq = malloc(sizeof(CsStorageRequest));
if (!storReq) { ERROR_OUT("not enough memory!\n"); }
storReq->Mode = CS_STORAGE_FILE_IO;
storReq->DevHandle = devHandle;
storReq->u.CsFileIo.Type = CS_STORAGE_LOAD_TYPE;
storReq->u.CsFileIo.FileHandle = fd;   // file fd to access for data
storReq->u.CsFileIo.Offset = 0;   // data offset within file
storReq->u.CsFileIo.Bytes = CHUNK_SIZE;
storReq->u.CsFileIo.DevMem.MemHandle = afdHandle1;
storReq->u.CsFileIo.DevMem.ByteOffset = 0;
status = csQueueStorageRequest(storReq, storReq, NULL, NULL, &compVal);
if (status != CS_SUCCESS)
    ERROR_OUT("Could not load storage data\n");

// Populate compute request on data loaded from file
CsComputeRequest *compReq = malloc(sizeof(CsComputeRequest) + (sizeof(CsComputeArg) * 3));
if (!compReq) { ERROR_OUT("memory alloc error\n"); }
compReq->CSFId = myCSF->CSFId;  // filter function id
compReq->NumArgs = 3;   // takes 3 arguments
compReq->u.Args[0] = afdHandle1;  // input buffer
compReq->u.Args[1] = CHUNK_SIZE;  // length of input
compReq->u.Args[2] = afdHandle2;  // output buffer
status = csQueueComputeRequest(compReq, compReq, NULL, NULL, &compVal);
if (status != CS_SUCCESS)
    ERROR_OUT("Error in CSF execution\n");

// Populate copy request for results data
CsCopyMemRequest *copyReq = malloc(sizeof(CsCopyMemRequest));
if (!copyReq) { ERROR_OUT("memory alloc error\n"); }
 COPYReq->Type = Cs_COPY_FROM_DEVICE;
copyReq->u.COPY_FROM_DEVICE = results.buf;
copyReq->u.COPY_FROM_DEVICE = afdHandle2;
copyReq->u.COPY_FROM_DEVICE = afdHandle1;
copyReq->Bytes = CHUNK_SIZE;
status = csQueueCopyMemRequest(copyReq, copyReq, NULL, NULL, &compVal);
if (status != CS_SUCCESS)
    ERROR_OUT("Could not copy from FDM\n");
```

```c
1
2
3
4
5
6
```
Please take a moment to rate this session.

Your feedback is important to us.