Best Practices Guide

Release r1.12

NVIDIA Corporation

Mar 11, 2025

Contents

Chapter 1. NVIDIA GPUDirect Storage Best Practices Guide

The Best Practices guide provides guidance from experts who are knowledgeable about NVIDIA® GPUDirect® Storage (GDS).

Chapter 2. Introduction

The purpose of the Best Practices guide is to provide guidance from experts who are knowledgeable about NVIDIA® GPUDirect® Storage (GDS). This guide also provides information about the lessons learned when building and scaling massive GPU accelerated I/O storage infrastructures. The intended audience includes data center planning staff, system builders, developers, and storage vendors.

Chapter 3. Software Settings

This section describes the settings required for GDS.

For the best performance, multiple software settings are required across the entire system, and some settings are specific to the filesystem that you are using.

For more information, refer to the GPUDirect Storage Installation and Troubleshooting Guide.

3.1. System Settings

For GDS p2p support on the Grace CPU based DGX™ (Grace Hopper) platform, IOMMU should be enabled and passthrough settings should be disabled.

The following are system settings that we recommend for the best performance on a bare metal x86_64 based platform.

- PCIe Access Control Services (ACS).

  ACS forces P2P PCIe transactions to go up through the PCIe Root Complex, which does not enable GDS to bypass the CPU on paths between a network adapter or NVMe and the GPU in systems that include a PCIe switch.

  For optimal GDS performance, disable ACS.

  Note

  To list all of the PCI switches that have ACS enabled, issue `/usr/local/cuda/gds/tools/gdscheck -p`.

- IOMMU

  When the IOMMU setting is enabled, PCIe traffic will be routed through the CPU root ports. This routing limits the maximum achievable throughput for configurations where the GPU and NIC are under the same PCIe switch. **Before** you install GDS, you **must** disable IOMMU. Refer to Installing GPUDirect Storage for more information.

  Note

  To determine whether the IOMMU setting is enabled, check the output from cat /proc/cmdline or use the `gdscheck` command..

As an example, the following output shows IOMMU is enabled on this system:

```
$ cat /proc/cmdline
BOOT_IMAGE=/boot/vmlinuz-5.19.0-38-generic root=UUID=fb2a25a8-9d2e-4e1c-9d8a-
↪efabdf165adc ro rootflags=data=ordered amd_iommu=on
```

Similarly, using `gdscheck` you should see the following output if the IOMMU is disabled on the system:

```
$ /usr/local/cuda/gds/tools/gdscheck -p
 IOMMU: disabled
 Platform verification succeeded
```

- NIC affinity

  For the P2P DMA to function efficiently, NICs, NVMes and GPUs should be under a PCIe switch when possible. For the P2P DMA to function efficiently on NVIDIA DGX™ based platforms, ensure at least one NIC is in the same CPU socket as the GPU.
- Avoid configurations where the NICs are assigned across the CPU sockets that require PCIe traffic to cross the CPU root ports or go across CPU sockets that use QPI.
- NIC versions
  - When using Mellanox ConnectX-5 or later, the HCAs must be configured in InfiniBand or RoCE v2 mode.
  - For GDS support, MLNX_OFED 5.4 or later, or DOCA 2.9.0 or later is required.

3.2. Use of CUDA Context in GPU Kernels and Storage IO

There are scenarios where the GDS workload data can be posted through intermediate buffers called bounce buffers. Hence a D2D copy is involved to/from these GPU bounce buffers to/from the application's GPU buffers. The cuFile library posts these IOs on a stream created on the primary CUDA context. If a heavy compute job or application kernel is running in the background in the form of GPU kernels on a separate context (not the primary context), it can interfere with the D2D copies and increase the D2D copy launch times. This problem does not happen if the compute kernels are running in the primary context, so it is recommended that the application launch GPU kernels on the primary context instead of using a separate context.

Note

If the application uses CUDA runtime API, the kernel launches will happen in the primary context by default.

3.3. cuFile Configuration Settings

The cuFile configuration settings in GDS are stored in the /etc/cufile.json file.

You can edit the file for best performance for your application as shown below. For information on the parameters in the file, refer to https://docs.nvidia.com/gpudirect-storage/configuration-guide/index.html#gds-parameters.

To display the configuration setting, run the following command:

```
$ cat /etc/cufile.json
```

A portion of the sample output:

```
"properties": {
            // max IO size issued by cuFile to nvidia-fs driver (in KB)
            "max_direct_io_size_kb" : 16384,
            ...
    }
```

For the requested IO size, GDS issues IO requests sequentially in chunks of reads/writes based on the `max_direct_io_size` parameter. Larger values of `max_direct_io_size` will result in a reduced number of calls to the IO stack and might result in higher throughput.

The `max_direct_io_size_kb` parameter can be set to a value that is a multiple of 64K. This process defines the additional system memory that is used for each buffer during `cuFileBufRegister` up to a maximum value for the `properties:max_direct_io_size_kb` parameter of 16MB. This value can be reduced to 1MB to lower the amount of system memory that is used per buffer.

The total system memory that is used can be obtained from `nvidia-fs` stats.

In this example, each of 256 threads register a 1MB buffer for GDS.

1. Run the following command:

   ```
   $ cat /proc/driver/nvidia-fs/stats
   ```

2. Review the output:

   ```
   NVFS statistics(ver:1.0)
   Active Shadow-Buffer (MB): 256...
   ```

   There are many tunables available in `cufile.json`. Refer to GPUDirect Storage Parameters.

Chapter 4. API Usage

This section describes best practices to remember when you use the GDS APIs.

The cuFile APIs are designed to be thread safe.

The fork system call should not be used after the cuFile library is initialized. The behavior of the APIs after the fork system call is undefined in the child process.

APIs with GPU buffers should be called in a valid CUDA context.

The following table outlines recommendations for various IO-specific use cases and their corresponding cuFile APIs which would be best suited.

Table 1: cuFile API Use Cases

Mode	IO Behavior	Use Case	Pros/Cons
`cuFileRead` `cuFileWrite`	Synchronous submission Synchronous completion	Single-threaded application using standard file system calls for a single large file and large buffers (>16MB)	**Pros** ▶ Simple to use **Cons** ▶ Does not help for multiple buffers

cuFile Threadpool enabled
`cuFileRead`
`cuFileWrite`
	Synchronous submission
	Synchronous completion
		Single-threaded application using standard file system calls for a single large file and large buffers
		Multi-threaded application using standard file system calls for multiple files and buffers.
		Application has thread pools for its IO pipeline.
			Pros
			▶ Simple to use
			▶ Lower submission latency
			▶ Better for medium-sized IO requests of 64K and above.
			Cons
			▶ Scalability limited by number of CPU threads used.
			▶ Higher CPU cost for smaller IO sizes (4k-64k).

continues on next page

Table 1 – continued from previous page

Mode	IO Behavior	Use Case	Pros/Cons
`cuFileBatchIOSetup` `cuFileBatchIOSubmit` `cuFileBatchIOGetStatus`	Synchronous submission Asynchronous completion	Single-threaded application using standard filesystem calls that performs IO for multiple non-contiguous file offsets, sizes, and GPU buffers. Each IO request is small (< 64KB) Can track completion of IOs asynchronously or wait in the same thread.	**Pros** ▶ Lower average completion latency ▶ Lower CPU cost because of batch submission **Cons** ▶ Higher submission latency, can be reduced by partial submission ▶ More complex to code: submit followed by polling for completion of the batch
`cuFileStreamRegister` `cuFileReadAsync` `cuFileWriteAsync` `cuFileStreamDeregister`	Asynchronous submission Asynchronous completion	Single threaded application using standard file system calls for multiple non-contiguous file offsets, sizes and GPU buffers. IO sizes - buffer data is dependent upon prior CUDA work.	**Pros** ▶ Simple to use for CUDA developers ▶ Works with CUDA semantics: fire and forget. ▶ Lower submission latency **Cons** ▶ Higher execution latency for IO size (<1 MB) ▶ Needs multiple streams to submit in parallel. ▶ Higher CPU utilization if synchronizing periodically.

4.1. cuFileDriverOpen

The `cuFileDriverOpen` API should be invoked only once per process and must occur **before** any other cuFile API is invoked. The application should call this routine to avoid the latency of the driver initialization that will be otherwise incurred in the first IO call.

4.2. cuFileHandleRegister

The `cuFileHandleRegister` API converts a file descriptor to a `cuFileHandle` and checks the ability of the named file, at its mount point, to be supported via GDS on this platform. This routine is required for calling all cuFile API calls that take a cuFileHandle parameter.

Note

There should be only one handle created for each file descriptor.

The same handle can be shared by multiple threads. Refer to the sample programs for more information about using the same handle by multiple threads.

Note

In compatibility mode, an additional file descriptor can be opened on the file without requiring `O_DIRECT` mode. This mode can also handle unaligned reads/writes, even when POSIX cannot.

4.3. cuFileBufRegister, cuFileRead, cuFileWrite, cuFileBatchIOSubmit, cuFileBatchIOGetStatus, cuFileReadAsync, cuFileWriteAsync, and cuFileStreamRegister

GPU buffers need to be exposed to third-party devices to enable DMA by those devices. The set of pages that span those buffers in the GPU virtual address space need to be mapped to the Base Address Register (BAR) space, and this mapping is an overhead.

The mechanism to accomplish this mapping is called registration. Explicit GPU buffer registration with the `cuFileBufRegister` API is optional. If a user buffer is not registered, an intermediate pre-registered GPU buffer that is owned by the cuFile implementation is used, and there is an extra copy from there to the user buffer. The following table and IO pattern descriptions provide guidance on whether registration is profitable.

Note

IO Pattern 1 is a suboptimal baseline case and is not referenced in this table.

Use Case	Description	Recommendation
A 4KB-aligned GPU buffer is reused as an intermediate buffer to read or write data using optimal IO sizes for storage systems in multiples of 4KB.	The GPU buffer is used as an intermediate buffer to stream the contents or to populate a different data structure in GPU memory. You can implement this use case for IO libraries with DSG.	Register this reusable intermediate buffer to avoid the additional internal staging of data by using GPU bounce buffers in the cuFile library. Refer to *IO Pattern 2* for the recommended usage.
Filling a large GPU buffer for one use.	The GPU buffer is the final location of the data. Since the buffer will not be reused, the registration cost will not be amortized. A usage example is reading large preformatted checkpoint binary data. Registering a large buffer can have a latency impact when the buffer is registered.	This can also cause BAR memory exhaustion because running multiple threads or applications will compete for BAR memory. Read or write the data without buffer registration. Refer to *IO Pattern 3* for the recommended usage.
Partitioning a GPU buffer to be accessed across multiple threads.	The main thread allocates a large memory buffer and creates multiple threads. Each thread registers a portion of the memory buffer independently and uses that as in *IO Pattern 2*. You can also register the entire buffer in the parent thread and use this registered buffer with the size and `devPtr_offset` parameters set appropriately with the buffer offsets for each thread. A `cudaContext` must be established in each thread before registering the GPU buffers.	Allocate, register, and deregister the buffers in each thread independently for simple IO workflows. For cases where the GPU memory is preallocated, each thread can set the appropriate context and register the buffers independently. Refer to IO Pattern 6 for the recommended usage. After you install the GDS package, see `cufile_sample_016.cc` and `cufile_sample_017.cc` under `/usr/local/CUDA-X.y/samples/` for more details.
GPU offsets, file offsets, and IO request sizes are unaligned.	IO reads or writes are mostly unaligned. An intermediate aligned buffer might be needed to handle alignment issues with GPU offsets, file offsets, and IO sizes.	**Do not** register the buffer. Refer to *IO Pattern 4* and *IO Pattern 5*.
Working on a GPU with a small BAR space as compared to the available GPU memory.	In some GPU SKUs, the BAR memory is smaller than the total device memory.	To avoid failures because of BAR memory exhaustion, do not register the buffer. Refer to *IO Pattern 3*.

4.3.1. IO Pattern 1

The following is a code sample for IO Pattern 1.

```
1 #define MB(x) ((x)*1024*1024L)
2 #define GB(x) ((x)*1024*1024L*1024L)
3
4
5 void thread_func(CUfileHandle_t cuHandle)
6 {
7         void *devPtr_base;
8         int readSize = MB(100);
9         int devPtr_offset = 0;
10        int file_offset = 0;
11        int ret = 0;
12
13        cudaSetDevice(0);
14        cudaMalloc(&devPtr_base, GB(1));
15
16        for (int i = 0; i < 10; i++) {
17
18            cuFileBufRegister((char *)devPtr_base + devPtr_offset, readSize, 0);
19
20            ret = cuFileRead(cuHandle, (char *)devPtr_base + devPtr_offset,
                                readSize,  file_offset, 0);
21
22
          <... launch cuda kernel using contents at devPtr_base + devPtr_offset … >

23              file_offset += readSize;
24              devPtr_offset += readSize;
25
26              cuFileBufDeregister((char *)devPtr_base + devPtr_offset);
27         }
28 }
```

1. Allocate 1 GB of GPU memory with `cudaMalloc`.
2. Fill the 1 GB by reading 100 MB at a time from the file as seen in the following loop:
   a. At line 18, the GPU buffer of 100 MB is registered.
   b. Submit the read for 100MB (readsize is 100 MB).
   c. At line 26, the GPU buffer of 100 MB is deregistered.

Although semantically correct, this loop might not provide the best performance because `cuFileBufRegister` and `cuFileBufDeregister` are continuously issued in the loop. For example, this problem can be addressed as shown in *IO Pattern 2*.

4.3.2. IO Pattern 2

The following is a code sample for IO Pattern 2.

```
#define MB(x) ((x)*1024*1024L)
#define GB(x) ((x)*1024*1024L*1024L)


void thread_func(CUfileHandle_t cuHandle)
{
         void *devPtr_base;
         int readSize = MB(100);
         int devPtr_offset = 0;
         int file_offset = 0;
         int ret = 0;

         cudaSetDevice(0);
         cudaMalloc(&devPtr_base, GB(1));
         cuFileBufRegister(devPtr_base, GB(1), 0);

         for (int i = 0; i < 10; i++) {

                 ret = cuFileRead(cuHandle, devPtr_base,
                                   readSize, file_offset, devPtr_offset);

             <... launch cuda kernel using contents at devPtr_base + devPtr_offset
↪… >

                 file_offset += readSize;
                 devPtr_offset += readSize;

         }
        cuFileBufDeregister(devPtr_base);
}
```

4.3.3. IO Pattern 3

The following is a code sample for IO Pattern 3.

```
#define MB(x) ((x)*1024*1024L)
#define GB(x) ((x)*1024*1024L*1024L)


void thread_func(CUfileHandle_t cuHandle)
{
         void *devPtr_base;
         int readSize = MB(100);
         int devPtr_offset = 0;
         int file_offset = 0;
         int ret = 0;

         cudaSetDevice(0);
         cudaMalloc(&devPtr_base, GB(1));
```

(continues on next page)

(continued from previous page)

```

        for (int i = 0; i < 10; i++) {

             ret = cuFileRead(cuHandle, (char *)devPtr_base,
                                          readSize, file_offset, devPtr_offset);

         <... launch cuda kernel using contents at devPtr_base + devPtr_offset … >

             file_offset += readSize;
             devPtr_offset += readSize;
        }
}
```

This example demonstrates the usage of `cuFileRead/cuFileWrite` APIs without using the `cuFileBufRegister` and `cuFileBufDeRegister` APIs. The IO-Pattern - 3 code snippet is the same as the *IO Pattern 1* and *IO Pattern 2* code snippets but the `cuFileBufRegister` API is not used.

1. Allocate 1 GB of GPU memory.
2. Fill the entire GPU memory of 1 GB by reading 100 MB at a time from the file as seen in the loop.

Note

Although semantically correct, this loop might not be optimal.

Internally, GDS uses GPU bounce buffers to perform IOs. Bounce buffers are GPU memory allocations that are internal to GDS, and these buffers are registered and managed by the GDS library. The number of bounce buffers is capped based on the `max_device_cache_size` (representing the total size of the bounce buffer cache) and `per_buffer_cache_size` (representing the size of each buffer) setting in the `/etc/cufile.json` file. The default values for `max_device_cache_size` and `per_buffer_cache_size` are 128MB and 1MB respectively, which amounts to 128 bounce buffers in total by default.

4.3.4. IO Pattern 4

The following is a code sample for IO Pattern 4. This is an unaligned IO due to file offset being unaligned.

```
#define MB(x) ((x)*1024*1024L)
#define GB(x) ((x)*1024*1024L*1024L)


void thread_func(CUfileHandle_t cuHandle)
{
        void *devPtr_base;
        int readSize = MB(100);
        int devPtr_offset = 0;
        int file_offset = 3; // Start from odd offset
        int ret = 0;

        cudaSetDevice(0);
        cudaMalloc(&devPtr_base, GB(1));
        cuFileBufRegister(devPtr_base, GB(1), 0);
```

(continues on next page)

(continued from previous page)

```
16
17          for (int i = 0; i < 10; i++) {
18                  // IO issued at offsets which are not 4K aligned
19                  ret = cuFileRead(cuHandle, devPtr_base,
                                              readSize, file_offset, devPtr_offset);
20                  assert(ret >= 0);
              <... launch cuda kernel using contents at devPtr_base + devPtr_offset … >
21
22                  file_offset += readSize;
23                  devPtr_offset += readSize;
24
25          }
26       cuFileBufDeRegister(devPtr_base);
27 }
```

This example demonstrates the usage of `cuFileRead` or `cuFileWrite` when IO is unaligned.

An IO is unaligned if one of the following conditions is true:

- The `file_offset` that was issued in `cuFileRead` or `cuFileWrite` is not 4K aligned.
- The size that was issued in `cuFileRead` or `cuFileWrite` is not 4K aligned.
- The `devPtr_base` that was issued in `cuFileRead` or `cuFileWrite` is not 4K aligned.
- The `devPtr_offset` that was issued in `cuFileRead` or `cuFileWrite` is not 4K aligned.

Note

In the above example, the initialization of `file_offset` is on line 10.

1. After allocating 1 GB of GPU memory, `cuFileBufRegister` is immediately invoked for the entire range of 1 GB as seen on line 15.
2. Fill the entire 1 GB GPU memory by reading 100 MB at a time from file as seen in the following loop:
    a. The initial file_offset is at 3, and reads are submitted with a `readSize` value of 100MB at an offset of 3 for each iteration.

       Therefore, `file_offset` during each read is not 4K aligned.
    b. Since `file_offset` is not 4K aligned, the GDS library will internally use GPU bounce buffers to complete the IO.

       The GPU bounce buffer mechanism is identical to *IO Pattern 3*.
3. Unaligned IOs might not be optimal and should be avoided by reading the size value that is specified in multiples of 4KB and the `file_offsets` value that is specified in multiples of 4KB.

   In the above example, an entire 1GB of GPU memory was registered using `cuFileBufRegister`. However, because the IO was unaligned, the GDS library cannot perform IO directly to these registered buffers. To handle unaligned IOs, the library will use GPU bounce buffers to perform the IO and copy the data from the bounce buffers to the application buffers. As a best practice, if the application typically performs unaligned IO, the application buffers do not need to be registered using the GDS library.

   The example in IO Pattern 4 demonstrates what happens when `file_offset` is unaligned; the previously mentioned points are accurate if any of the unaligned conditions is true.

If the application can't issue 4K aligned IO, instead of using the `cuFileBufRegister` API, use the `cuFileRead` or `cuFileWrite` APIs as described in IO-Pattern-2.

Note

When the write workload is unaligned, GDS uses Read-Modify-Write internally using POSIX mode.

4.3.5. IO Pattern 5

The following is a code sample for IO Pattern 5. This IO is an unaligned IO due to buffer pointer and offset not being 4K aligned.

```
#define MB(x) ((x)*1024*1024L)
#define GB(x) ((x)*1024*1024L*1024L)


void thread_func(CUfileHandle_t cuHandle)
{
         void *devPtr_base;
         int readSize = MB(100);
         int devPtr_offset = 3; // Start from odd offset
         int file_offset = 0;
         int ret = 0;

         cudaSetDevice(0);
         cudaMalloc(&devPtr_base, GB(1));
         cuFileBufRegister(devPtr_base, GB(1), 0);

         for (int i = 0; i < 10; i++) {
                 // IO issued at gpu buffer offsets which are not 4K aligned
                 ret = cuFileRead(cuHandle, devPtr_base,
                                   readSize, file_offset, devPtr_offset);
                 assert (ret >= 0);
                  <... launch cuda kernel using contents at devPtr_base + devPtr_
↪offset … >

                 file_offset += readSize;
                 devPtr_offset += readSize;

         }
      cuFileBufDeRegister(devPtr_base);
 }
```

This example demonstrates using `cuFileRead/cuFileWrite` when IO is unaligned. The `devPtr_base + devPtr_offset` that are issued to `cuFileRead` or `cuFileWrite` are not 4K aligned.

If the IO is unaligned, the cuFile library will issue IO through its internal GPU bounce buffer cache. However, if the allocation of the internal cache fails, the IO will fail. To avoid IO failure in this case, you can set `allow_compat_mode` to `true` in the `/etc/cufile.json` file. With this setting, IO will fall back to using POSIX API calls withing GDS.

4.3.6. IO Pattern 6

The following program snippet demonstratess the use cuFile batch APIs.

```
int main(int argc, char *argv[]) {
        int fd[MAX_BATCH_IOS];
        void *devPtr[MAX_BATCH_IOS];
        CUfileDescr_t cf_descr[MAX_BATCH_IOS];
        CUfileHandle_t cf_handle[MAX_BATCH_IOS];
        CUfileIOParams_t io_batch_params[MAX_BATCH_IOS];
        CUfileIOEvents_t io_batch_events[MAX_BATCH_IOS];

        <Get program inputs>

        status = cuFileDriverOpen();
        if (status.err != CU_FILE_SUCCESS) {
                std::cerr << "cufile driver open error: "
                        << cuFileGetErrorString(status) << std::endl;
                return -1;
        }

        <Open files and call cuFileHandleRegister for each of the batch entry file
↪handles>

        <Allocate cuda memory and register buffers using cuFileBufRegister for each
↪of the
           batch entries>

        for(i = 0; i < batch_size; i++) {
                io_batch_params[i].mode = CUFILE_BATCH;
                io_batch_params[i].fh = cf_handle[i];
                io_batch_params[i].u.batch.devPtr_base = devPtr[i];
                io_batch_params[i].u.batch.file_offset = i * size;
                io_batch_params[i].u.batch.devPtr_offset = 0;
                io_batch_params[i].u.batch.size = size;
                io_batch_params[i].opcode = CUFILE_READ;
        }
        std::cout << "Setting Up Batch" << std::endl;
        errorBatch = cuFileBatchIOSetUp(&batch_id, batch_size);
        if(errorBatch.err != 0) {
                std::cerr << "Error in setting Up Batch" << std::endl;
                goto error;
        }

        errorBatch = cuFileBatchIOSubmit(batch_id, batch_size, io_batch_params,
↪flags);
        if(errorBatch.err != 0) {
                std::cerr << "Error in IO Batch Submit" << std::endl;
                goto error;
        }

        // Setting min_nr to batch_size for this example.
        min_nr = batch_size;
        while(num_completed != min_nr) {
                memset(io_batch_events, 0, sizeof(*io_batch_events));
                nr = batch_size;
                errorBatch = cuFileBatchIOGetStatus(batch_id, batch_size, &nr, io_
```

(continues on next page)

(continued from previous page)

```
↪batch_events, NULL);
                if(errorBatch.err != 0) {
                        std::cerr << "Error in IO Batch Get Status" << std::endl;
                        goto error;
                }
                std::cout << "Got events " << nr << std::endl;
                num_completed += nr;
                <Copy to the user buffer>
        }

        cuFileBatchIODestroy(batch_id);
        < Deregister the device memory using cuFileBufDeregister>

        status = cuFileDriverClose();
        std::cout << "cuFileDriverClose Done" << std::endl;
        if (status.err != CU_FILE_SUCCESS) {
              ...
        }
        ret = 0;
        return ret;
        ...
}
```

This program demonstrates a simple use case where cuFile batch APIs can be used to perform a read with a specified batch size. It provides an example of a sequence of calls where each entry uses registered buffers on each individual file descriptor.

It may be worthwhile to mention that `min_nr` passed to `cuFileBatchIOGetStatus()` in the above example was set to `batch_size`. It is possible that `min_nr` can be set to something less than `batch_size` and as the `min_nr` number of I/Os are completed, that many numbers of I/Os can be submitted subsequently to the I/O pipeline resulting in an enhanced I/O throughput.

4.3.7. IO Pattern 7

The following program snippet uses cuFile stream-based async I/O APIs to perform a data integrity test.

```
typedef struct io_args_s
{
   void *devPtr;
   size_t max_size;
   off_t offset;
   off_t buf_off;
   ssize_t read_bytes_done;
   ssize_t write_bytes_done;
} io_args_t;

int main(int argc, char *argv[]) {

        unsigned char iDigest[SHA256_DIGEST_LENGTH],
                                oDigest[SHA256_DIGEST_LENGTH];

        <Get inputs>

```

(continues on next page)

(continued from previous page)

```
        <Create a data file using some random data>

        // Allocate device Memory and register with cuFile
        check_cudaruntimecall(cudaMalloc(&args.devPtr, args.max_size));
        // Register buffers. For unregistered buffers, this call is not required.
        status = cuFileBufRegister(args.devPtr, args.max_size, 0);
        if (status.err != CU_FILE_SUCCESS) {
                        goto error;
        }

        < Open the data file just created for read and create a new data file to
↪write the content
           read from the datafile>

        <Register the filehandles>

        // Create stream for I/O.
        check_cudaruntimecall(cudaStreamCreateWithFlags(&io_stream,
                cudaStreamNonBlocking));

        // Register Streams for best performance
        // If all the inputs i.e. size, offset and buf_off are known and they are page
↪aligned, then
        // use CU_FILE_STREAM_FIXED_AND_ALIGNED flag. If they are not known but will
        // always be page aligned then use CU_FILE_STREAM_PAGE_ALIGNED_INPUTS flag
        // flag.
        check_cudaruntimecall(cuFileStreamRegister(io_stream,
                                                   CU_FILE_STREAM_FIXED_AND_ALIGNED));

        // special case for holes
        check_cudaruntimecall(cudaMemsetAsync(args.devPtr, 0, args.max_size, io_
↪stream));

        status = cuFileReadAsync(cf_rhandle, (unsigned char *)args.devPtr,
                                 &args.max_size, &args.offset, &args.buf_off,
                                               &args.read_bytes_done, io_stream);
        if (status.err != CU_FILE_SUCCESS) {
                        std::cerr << "read failed : "
                                << cuFileGetErrorString(status) << std::endl;
                ret = -1;
                goto error;
        }

        // Write loaded data from GPU memory to a new file
        status = cuFileWriteAsync(cf_whandle, (unsigned char *)args.devPtr,
                                  (size_t *)&args.max_size, &args.offset, &args.buf_
↪off,
                                  &args.write_bytes_done, io_stream);
        if (status.err != CU_FILE_SUCCESS) {
                 goto error;
        }

        std::cout << "writing submit done to file :" << TEST_WRITEFILE << std::endl;
        check_cudaruntimecall(cudaStreamSynchronize(io_stream));
        if((args.read_bytes_done < (ssize_t)args.max_size) ||
           (args.write_bytes_done < args.read_bytes_done))
```

(continues on next page)

(continued from previous page)

```
        {
                std::cerr << "io error issued size:" << args.max_size <<
                          " read:" << args.read_bytes_done <<
                          " write:" <<  args.write_bytes_done << std::endl;
                goto error;
        }
        // Compare file signatures
        ret = SHASUM256(TEST_READWRITEFILE, iDigest, args.max_size);
        if(ret < 0) {
              ...
        }
        DumpSHASUM(iDigest);
        ret = SHASUM256(TEST_WRITEFILE, oDigest, args.max_size);
        if(ret < 0) {
            ...
        }
        DumpSHASUM(oDigest);
        if (memcmp(iDigest, oDigest, SHA256_DIGEST_LENGTH) != 0) {
                std::cerr << "SHA SUM Mismatch" << std::endl;
                ret = -1;
        } else {
                std::cout << "SHA SUM Match" << std::endl;
                ret = 0;
        }
        if(io_stream) {
                check_cudaruntimecall(cuFileStreamDeregister(io_stream));
                check_cudaruntimecall(cudaStreamDestroy(io_stream));
        }
       <Free up all the resources>

        return ret;

error:
        ...
}
```

This program demonstrates a simple use case where the cuFile stream APIs can be used to perform a data integrity test using a single stream. It first creates a data file using random content. Then it reads the content through an I/O stream and writes that content into a new file. Finally it compares the content of the newly created data file against the original content using SHA (simple hash algorithm). It is possible that the exact size may not be known in the beginning and will be known later. In that scenario, one can set the actual size using the CUDA host call back function (`cuLaunchHostFunc`) on the same stream before calling `cuFileReadAsync` or `cuFileWriteAsync` APIs.

4.4. cuFileHandleDeregister

Prerequisite: Before calling this API, the application must ensure that the IO on that handle has completed and is no longer being used. The file descriptor should still be open.

To reclaim resources before ending the process, always invoke the `cuFileHandleDeregister` API.

4.5. cuFileBufDeregister

Prerequisite: Before calling this API, the application must ensure that all the cuFile IO operations using the buffer have completed.

For every buffer registered by using `cuFileBufRegister`, use this API to deregister it by using the same device pointer that was used for registration. This process ensures that all resources are reclaimed before ending the process.

4.6. cuFileStreamRegister

The `cuFileStreamRegister` API converts a file descriptor to a `cuFileHandle` and checks the ability of the named file, at its mount point, to be supported via GDS on this platform.

Explicit stream registration with the `cuFileStreamRegister` API is optional. If the stream is registered, then some internal buffers and associated metadata resources will be pre-allocated for subsequent stream I/O and may improve I/O latencies. Additionally these resources will be reused until deregistered using `cuFileStreamUnregister`. Without this API, all these resources will be allocated during actual I/O.

4.7. cuFileStreamDeregister

Prerequisite: Before calling this API, the application must ensure that the I/O on that stream has completed and the stream is no longer being used

For every stream registered by using `cuFileStreamRegister`, use this API to deregister it by using the same stream that was used for registration. To reclaim resources before ending the process, always invoke this API.

4.8. cuFileDriverClose

Prerequisites: Before calling this API, the application must ensure that all cuFile IO operations have completed and that all buffers and handles are deregistered.

In order to reduce the tear-down time of a GDS enabled application (i.e. expedited release of pinned GPU buffers and other cuFile resources), it is highly recommended to call the `cuFileDriverClose()` API at the end of the application.

Chapter 5. Notice

Chapter 6. OpenCL

Chapter 7. Trademarks

Copyright