NVIDIA Magnum IO GPUDirect Storage cuFile API

API Reference

DA-10158-001-01_v1.7.2 | October 2023

Table of Contents

Chapter 1. Introduction

NVIDIA® Magnum IO GPUDirect® Storage (GDS) is part of the GPUDirect family. GDS enables a direct data path for direct memory access (DMA) transfers between GPU memory and storage, which avoids a bounce buffer through the CPU. This direct path increases system bandwidth and decreases the latency and utilization load on the CPU.

This document provides information about the cuFile APIs that are used in applications and frameworks to leverage GDS technology and describes the intent, context, and operation of those APIs which are part of the GDS technology.

Note: The APIs and descriptions are subject to change without notice.

Chapter 2. Usage

This section describes the operation of the cuFile APIs.

Because the functionality is part of the CUDA Driver C API, the APIs use the `cuFile` prefix and camel case motif of the CUDA Driver.

- All APIs are thread-safe.
- The fork system call should not be used after the library is initialized. The behavior of the APIs after the fork system call is undefined in the child process.
- The APIs with GPU buffers should be called in a valid CUDA context and stream if applicable.
- All APIs are issued from the CPU, not the GPU.

Note: Starting from CUDA toolkit 12.2 (GDS version 1.7.x) release cuFile APIs support memory allocated on GPU device as well as host memory. peer to peer transfer using GPUDirect™ is supported to and from device memory on supported file system and hardware configurations. The APIs will refer to this memory address as buffer pointer unless the API specifically applies to a particular type of memory.

2.1. Dynamic Interactions

The following describes the dynamic interactions between the cuFile APIs.

Some of the cuFile APIs are optional. If they are not called proactively, their actions will occur reactively:

If `cuFile{DriverOpen, HandleRegister, BufRegister}` is called on a driver, file, or buffer, respectively that has been opened or registered by a previous `cuFile*` API call, this will result in an error. Calling `cuFile{BufDeregister, HandleDeregister, DriverClose}` on a buffer, file, or driver, respectively that has never been opened or registered by a previous `cuFile*` API call results in an error. For these errors, the output parameters of the APIs are left in an undefined state, and there are no other side effects.

- `cuFileDriverOpen` explicitly causes driver initialization.

  Its use is optional. If it is not used, driver initialization happens implicitly at the first use of the `cuFile{HandleRegister, Read, Write, BufRegister}` APIs.

- (Mandatory) `cuFileHandleRegister` turns an OS-specific file descriptor into a `CUfileHandle_t` and performs checking on the GDS supportability based on the mount point and the way that the file was opened.
- `cuFileBufRegister` explicitly registers a memory buffer.

  If this API is not called, an internal registered memory is used if required on the first time the buffer is used, for example, in `cuFile{Read, Write}`.
- `cuFile{BufDeregister, HandleDeregister}` explicitly frees a buffer and file resources, respectively.

  If this API is not called, the buffer and resources are implicitly freed when the driver is closed using `cuFileDriverClose`.
- `cuFileDriverClose` explicitly frees driver resources.

  If this API is not called, the driver resources are implicitly freed when `dlclose()` is performed on the library handle or when the process is terminated.

2.2. Driver, File, and Buffer Management

This section describes the overall workflow to manage the driver, the file, and buffer management:

1. Call `cuFileDriverOpen()` to initialize the state of the critical performance path.
2. Allocate GPU memory with `cudaMalloc`, `cudaMallocManaged`, `cuMem*` APIs or host memory using `cudaMallocHost`, `malloc` or `mmap`.
3. To register the buffer, call `cuFileBufRegister` to initialize the buffer state of the critical performance path.
4. Complete the following IO workflow:

   a). For Linux, open a file with POSIX open.

   b). Call `cuFileHandleRegister` to wrap an existing file descriptor in an OS-agnostic `CUfileHandle_t`. This step evaluates the suitability of the file state and the file mount for GDS and initializes the file state of the critical performance path.

   c). Call IO APIs such as `cuFileRead`/`cuFileWrite` on an existing cuFile handle and existing buffer.

   - If the `cuFileBufRegister` has not been previously called on the buffer pointer, `cuFileRead/cuFileWrite` will use internal registered buffers when required.
   - Not using `cuFileBufRegister` might not be performant for small IO sizes.
   - Refer to the GPUDirect Best Practices Guide for more information.

   d). Unless an error condition is returned, the IO is performed successfully.
5. Call `cuFileBufDeregister` to free the buffer-specific cuFile state.
6. Call `cuFileHandleDeregister` to free the file-specific cuFile state.

7. Call `cuFileDriverClose` to free up the cuFile state.

Note: Not using the `cuFileDeregister` and `cuFileDriverClose` APIs (steps 5, 6, and 7) might unnecessarily consume resources, as shown by tools such as valgrind. The best practice is to always call these APIs in the application cleanup paths.

2.3. cuFile Compatibility Mode

Use Cases

cuFile APIs can be used in different scenarios:

- Developers building GPUDirect Storage applications with cuFile APIs, but don't have the supported hardware configurations.
- Developers building applications running on GPU cards that have CUDA compute capability > 6, but don't have BAR space exposed.
- Deployments where `nvidia-fs.ko` is not loaded or cannot be loaded.
- Deployments where the Linux distribution does not support GPUDirect Storage.
- Deployments where the filesystem may be not supported with GPUDirect Storage.
- Deployments where the network links are not enabled with RDMA support.
- Deployment where the configuration is not optimal for GPUDirect Storage.

Behavior

The cuFile library provides a mechanism for cuFile reads and writes to use compatibility mode using POSIX `pread`, `pwrite`, and `aio_submit` APIS respectively to host memory and copying to GPU memory when applicable. The behavior of compatibility mode with cuFile APIs is determined by the following configuration parameters.

Configuration Option (default)	cuFile IO Behavior
`"allow_compat_mode": true`	If `true`, falls back to using compatibility mode when the library detects that the buffer file descriptor opened cannot use GPUDirect Storage.
`"force_compat_mode": false`	If `true`, this option can be used to force all IO to use compatibility mode. Alternatively the admin can unload the nvidia_fs.ko or not expose the character devices in the docker container environment.
`"gds_rdma_write_support": true`	If `false`, forces compatibility mode to be used for writes even when the underlying file system is capable of performing GPUDirect Storage writes. Note: If the option is "false", this option will override and disable any filesystem-specific option to enable RDMA writes.

Configuration Option (default)	cuFile IO Behavior
"posix_unaligned_writes" : false	If true, forces compatibility mode to be used for writes where the file offset and/or IO size is not aligned to Page Boundary (4KB).
"lustre:posix_gds_min_kb" : 0	For a lustre filesystem, if greater than 0, compatibility mode is used for IO sizes between [1 - posix_gds_min_kb] specified in kB. Note: This option will force posix mode even if "allow_compat_mode" is set to "false".
"weka:rdma_write_support" : false	If this option is false, all writes to WekaFS will use compatibility mode. Note: If the option is set to "false", cuFile library will use the posix path even if the allow_compat_mode option is true or false.
"gpfs:gds_write_support" : false	If this option is false, all writes to IBM Spectrum Scale will use compatibility mode. Note: If the option is set to "false", cuFile library will use the posix path even if the allow_compat_mode option is true or false.
"rdma_dynamic_routing": false, "rdma_dynamic_routing_order": ["SYS_MEM"]	If rdma_dynamic_routing is set to true and rdma_dynamic_routing_order is set to ["SYS_MEM"] , then all IO for DFS will use compatibility mode.

In addition to the above configuration options, compatibility mode will be used as a fallback option for following use cases.

Use Case	cuFile IO Behavior
No BAR1 memory in GPU.	Use compatibility mode.
For wekaFS or IBM Spectrum Scale mounts: If there are no rdma_dev_addr_list specified, or failure to register MR with ib device.	Use compatibility mode.
Bounce buffers cannot be allocated in GPU memory.	Use compatibility mode.
For WekaFS and IBM Spectrum Scale: If the kernel returns -ENOTSUP for GPUDirect Storage read/write.	Retry the IO operation internally using compatibility mode.
cuFile Stream and cuFile Batch APIs on IBM Spectrum Scale or WekaFS	All Async and batch operations will internally use compatibility mode IO.
The nvidia_fs.ko driver is not loaded.	All IO operations will use compatibility mode.

Limitations

- Compatible mode does not work in cases where the GPUs have CUDA compute capability less than 6.

- GDS Compat mode has been tested and works with GDS enabled file systems and environments. It has not been tested to work on all other filesystems.

Chapter 3. cuFile API Specification

This section provides information about the cuFile APIs that are used from the CPU to enable applications and frameworks.

3.1. Data Types

3.1.1. Declarations and Definitions

Here are the relevant cuFile enums and their descriptions.

```
typedef struct CUfileError {
        CUfileOpError err; // cufile error
        enum CUresult cu_err; // for CUDA-specific errors
} CUfileError_t;

/**
 * error macros to inspect error status of type CUfileOpError
 */

#define IS_CUFILE_ERR(err) \
        (abs((err)) > CUFILEOP_BASE_ERR)

#define CUFILE_ERRSTR(err) \
        cufileop_status_error(static_cast<CUfileOpError>(abs((err))))

#define IS_CUDA_ERR(status) \
        ((status).err == CU_FILE_CUDA_DRIVER_ERROR)

#define CU_FILE_CUDA_ERR(status) ((status).cu_
```

The following enum and two structures enable broader cross-OS support:

```
enum CUfileFileHandleType {
    CU_FILE_HANDLE_TYPE_OPAQUE_FD = 1, /* linux based fd    */
    CU_FILE_HANDLE_TYPE_OPAQUE_WIN32 = 2, /* windows based handle */
CU_FILE_HANDLE_TYPE_USERSPACE_FS  = 3, /* userspace based FS */
};

typedef struct CUfileDescr_t {
CUfileFileHandleType type; /* type of file being registered */
union {
int fd;             /* Linux   */
void *handle;         /* Windows */
} handle;
const CUfileFSOps_t *fs_ops;     /* file system operation table */
}CUfileDescr_t;

/* cuFile handle type */
```

```
typedef void*  CUfileHandle_t;

typedef struct cufileRDMAInfo
{
        int version;
        int desc_len;
        const char *desc_str;
}cufileRDMAInfo_t;

typedef struct CUfileFSOps {
      /* NULL means discover using fstat */
      const char* (*fs_type) (void *handle);

      /* list of host addresses to use,  NULL means no restriction */
      int (*getRDMADeviceList)(void *handle, sockaddr_t **hostaddrs);

      /* -1 no pref */
      int (*getRDMADevicePriority)(void *handle, char*, size_t,
                                   loff_t, sockaddr_t* hostaddr);

      /* NULL means try VFS */
      ssize_t (*read) (void *handle, char*, size_t, loff_t, cufileRDMAInfo_t*);
      ssize_t (*write) (void *handle, const char *, size_t, loff_t ,
 cufileRDMAInfo_t*);
}CUfileFSOps_t;

typedef enum CUfileDriverStatusFlags {
        CU_FILE_LUSTRE_SUPPORTED = 0,        /*!< Support for DDN LUSTRE */
        CU_FILE_WEKAFS_SUPPORTED = 1,        /*!< Support for WEKAFS */
        CU_FILE_NFS_SUPPORTED = 2,           /*!< Support for NFS */
        CU_FILE_GPFS_SUPPORTED = 3,          /*! < Support for GPFS */
        CU_FILE_NVME_SUPPORTED = 4,          /*!< Support for NVMe */
        CU_FILE_NVMEOF_SUPPORTED = 5,        /*!< Support for NVMeOF */
        CU_FILE_SCSI_SUPPORTED = 6,          /*!< Support for SCSI */
        CU_FILE_SCALEFLUX_CSD_SUPPORTED = 7, /*!< Support for Scaleflux CSD*/
        CU_FILE_NVMESH_SUPPORTED = 8,        /*!< Support for NVMesh Block Dev*/
        CU_FILE_BEEGFS_SUPPORTED = 9,        /*!< Support for BeeGFS */
}CUfileDriverStatusFlags_t;


enum CUfileDriverControlFlags {
      CU_FILE_USE_POLL_MODE = 0, /*!< use POLL mode. properties.use_poll_mode*/
      CU_FILE_ALLOW_COMPAT_MODE = 1 /*!< allow COMPATIBILITY mode.
 properties.allow_compat_mode*/
};

typedef enum CUfileFeatureFlags {
    CU_FILE_DYN_ROUTING_SUPPORTED =0,
    CU_FILE_BATCH_IO_SUPPORTED = 1,
    CU_FILE_STREAMS_SUPPORTED = 2
} CUfileFeatureFlags_t;;

/* cuFileDriverGetProperties describes this structure's members */
typedef struct CUfileDrvProps {
   struct {
     unsigned int major_version;
     unsigned int minor_version;
     size_t poll_thresh_size;
     size_t max_direct_io_size;
     unsigned int dstatusflags;
     unsigned int dcontrolflags;
   } nvfs;
   CUfileFeatureFlags_t fflags;
   unsigned int max_device_cache_size;
   unsigned int per_buffer_cache_size;
   unsigned int max_pinned_memory_size;
   unsigned int max_batch_io_timeout_msecs;
```

```
}CUfileDrvProps_t;

/* Parameter block for async cuFile IO */
/* Batch APIs use an array of these    */
/* Status must be CU_FILE_WAITING when submitted, and is
   updated when enqueued and when complete, so this user-allocated
   structure is live until the operation completes.    */
typedef enum CUFILEStatus_enum {
        CUFILE_WAITING = 0x000001,  /* required value prior to submission */
        CUFILE_PENDING = 0x000002,  /* once enqueued */
        CUFILE_INVALID = 0x000004,  /* request was ill-formed or could not be
 enqueued */
        CUFILE_CANCELED = 0x000008, /* request successfully canceled */
        CUFILE_COMPLETE = 0x0000010, /* request successfully completed */
        CUFILE_TIMEOUT = 0x0000020,  /* request timed out */
        CUFILE_FAILED  = 0x0000040  /* unable to complete */
}CUfileStatus_t;

typedef enum cufileBatchMode {
        CUFILE_BATCH = 1,
} CUfileBatchMode_t;

typedef struct CUfileIOParams {
        CUfileBatchMode_t mode; // Must be the very first field.
        union {
                struct  {
                        void *devPtr_base;
                        off_t file_offset;
                        off_t devPtr_offset;
                        size_t size;
                }batch;
        }u;
        CUfileHandle_t fh;
        CUfileOpcode_t opcode;
        void *cookie;
}CUfileIOParams_t;

typedef struct CUfileIOEvents {
        void *cookie;
        CUfileStatus_t   status;      /* status of the operation */
        size_t ret;       /* -ve error or amount of I/O done. */
}CUfileIOEvents_t;
```

3.1.2. Typedefs

cuFile typedefs:

```
typedef struct CUfileDescr CUfileDesr_t
typedef struct CUfileError CUfileError_t
typedef struct CUfileDrvProps CUfileDrvProps_t
typedef enum CUfileFeatureFlags CUfileFeatureFlags_t
typedef enum CUfileDriverStatusFlags_enum CUfileDriverStatusFlags_t
typedef enum CUfileDriverControlFlags_enum CUfileDriverControlFlags_t
typedef struct CUfileIOParams CUfileIOParams_t
typedef enum CUfileBatchOpcode CUfileBatchOpcode_t
```

3.1.3. Enumerations

cuFile enums:

- `enum CUfileOpcode_enum`

  This is the cuFile operation code for batch mode.

OpCode	Value	Description
CU_FILE_READ	0	Batch Read
CU_FILE_WRITE	1	Batch Write

```
/* cuFile Batch IO operation kind */
enum CUfileOpcode {
     CU_FILE_READ,
     CU_FILE_WRITE,
};
```

- `enum CUfileStatus`

  The cuFile Status codes for batch mode.

Status	Value	Description
CUFILE_WAITING	0x01	The initial value.
CUFILE_PENDING	0x02	Set once enqueued into the driver.
CUFILE_INVALID	0x04	Invalid parameters.
CUFILE_CANCELED	0x08	Request successfully canceled.
CUFILE_COMPLETE	0x10	Successfully completed.
CUFILE_TIMEOUT	0x20	The operation has timed out.
CUFILE_FAILED	0x40	IO has failed.

- `enum CUfileOpError`
  - The cuFile Operation error types.
  - All error code values, other than `CU_FILE_SUCCESS`, are considered failures that might leave the output and input parameter values of APIs in an undefined state.

    These values cannot have any side effects on the file system, the application process, and the larger system.

    Note: cuFile-specific errors will be greater than `CUFILEOP_BASE_ERR` to enable users to distinguish between POSIX errors and cuFile errors.

    ```
    #define CUFILEOP_BASE_ERR 5000
    ```

Error Code	Value	Description
CU_FILE_SUCCESS	0	The cufile is successful.
CU_FILE_DRIVER_NOT_INITIALIZED	5001	The nvidia-fs driver is not loaded.
CU_FILE_DRIVER_INVALID_PROPS	5002	An invalid property.
CU_FILE_DRIVER_UNSUPPORTED_LIMIT	5003	A property range error.
CU_FILE_DRIVER_VERSION_MISMATCH	5004	An nvidia-fs driver version mismatch.
CU_FILE_DRIVER_VERSION_READ_ERROR	5005	An nvidia-fs driver version read error.

Error Code	Value	Description
CU_FILE_DRIVER_CLOSING	5006	Driver shutdown in progress.
CU_FILE_PLATFORM_NOT_SUPPORTED	5007	GDS is not supported on the current platform.
CU_FILE_IO_NOT_SUPPORTED	5008	GDS is not supported on the current file.
CU_FILE_DEVICE_NOT_SUPPORTED	5009	GDS is not supported on the current GPU.
CU_FILE_NVFS_DRIVER_ERROR	5010	An nvidia-fs driver ioctl error.
CU_FILE_CUDA_DRIVER_ERROR	5011	A CUDA Driver API error. This error indicates a CUDA driver-api error. If this is set, a CUDA-specific error code is set in the cu_err field for cuFileError.
CU_FILE_CUDA_POINTER_INVALID	5012	An invalid device pointer.
CU_FILE_CUDA_MEMORY_TYPE_INVALID	5013	An invalid pointer memory type.
CU_FILE_CUDA_POINTER_RANGE_ERROR	5014	The pointer range exceeds the allocated address range.
CU_FILE_CUDA_CONTEXT_MISMATCH	5015	A CUDA context mismatch.
CU_FILE_INVALID_MAPPING_SIZE	5016	Access beyond the maximum pinned memory size.
CU_FILE_INVALID_MAPPING_RANGE	5017	Access beyond the mapped size.
CU_FILE_INVALID_FILE_TYPE	5018	An unsupported file type.
CU_FILE_INVALID_FILE_OPEN_FLAG	5019	Unsupported file open flags.
CU_FILE_DIO_NOT_SET	5020	The fd direct IO is not set.
CU_FILE_INVALID_VALUE	5022	Invalid API arguments.
CU_FILE_MEMORY_ALREADY_REGISTERED	5023	Device pointer is already registered.
CU_FILE_MEMORY_NOT_REGISTERED	5024	A device pointer lookup failure has occurred.
CU_FILE_PERMISSION_DENIED	5025	A driver or file access error.
CU_FILE_DRIVER_ALREADY_OPEN	5026	The driver is already open.
CU_FILE_HANDLE_NOT_REGISTERED	5027	The file descriptor is not registered.
CU_FILE_HANDLE_ALREADY_REGISTERED	5028	The file descriptor is already registered.
CU_FILE_DEVICE_NOT_FOUND	5029	The GPU device cannot be not found.
CU_FILE_INTERNAL_ERROR	5030	An internal error has occurred. Refer to `cufile.log` for more details.

Error Code	Value	Description
CU_FILE_GETNEWFD_FAILED	5031	Failed to obtain a new file descriptor.
CU_FILE_NVFS_SETUP_ERROR	5033	An NVFS driver initialization error has occurred.
CU_FILE_IO_DISABLED	5034	GDS is disabled by config on the current file.
CU_FILE_BATCH_SUBMIT_FAILED	5035	Failed to submit a batch operation.
CU_FILE_GPU_MEMORY_PINNING_FAILED	5036	Failed to allocate pinned GPU memory.
CU_FILE_BATCH_FULL	5037	Queue full for batch operation.
CU_FILE_ASYNC_NOT_SUPPORTED	5038	cuFile stream operation is not supported.

Note: Data path errors are captured via standard error codes by using errno. The APIs will return -1 on error.

3.2. cuFile Driver APIs

The following cuFile APIs that are used to initialize, finalize, query, and tune settings for the cuFile system.

```
/* Initialize the cuFile infrastructure */
CUfileError_t cuFileDriverOpen();

/* Finalize the cuFile system */
CUfileError_t cuFileDriverClose();

/* Query capabilities based on current versions, installed functionality */
CUfileError_t cuFileGetDriverProperties(CUfileDrvProps_t *props);

/*API to set whether the Read/Write APIs use polling to do IO operations */
CUfileError_t cuFileDriverSetPollMode(bool poll, size_t poll_threshold_size);

/*API to set max IO size(KB) used by the library to talk to nvidia-fs driver */
CUfileError_t cuFileDriverSetMaxDirectIOSize(size_t max_direct_io_size);

/* API to set maximum GPU memory reserved per device by the library for internal
 buffering */
CUfileError_t cuFileDriverSetMaxCacheSize(size_t max_cache_size);

/* Sets maximum buffer space that is pinned in KB for use by  cuFileBufRegister
CUfileError_t cuFileDriverSetMaxPinnedMemSize(size_t
   max_pinned_memory_size);
```

Note: Refer to sample_007 for usage.

3.3. cuFile Synchronous IO APIs

The core of the cuFile IO APIs are the read and write functions.

```
ssize_t cuFileRead(CUFileHandle_t fh, void *bufPtr_base, size_t size, off_t
 file_offset, off_t devPtr_offset);
ssize_t cuFileWrite(CUFileHandle_t fh, const void *bufPtr_base, size_t size, off_t
 file_offset, off_t devPtr_offset);
```

The starting offset of the buffer on the device or host is determined by a base (`bufPtr_base`) and offset (`bufPtr_offset`). This offset is distinct from the offset in the file.

Note: To use the registered buffer, the bufPtr_base must be the buffer pointer used to register during `cuFileBufRegister`. Otherwise `cuFileRead` and `cuFileWrite` APIs may use internal memory buffers for GPUDirect Storage peer to peer operations.

Note: The default behavior for all paths where GDS is not supported is for the cuFile IO API to attempt IO using file system supported posix mode APIs when `properties.allow_compat_mode` is set to true. In order to disable cuFile APIs falling back to posix APIs for unsupported GDS paths, `properties.allow_compat_mode` in the `/etc/cufile.json` file should be set to false.

Note: Refer to sample sample_003 for usage.

3.4. cuFile File Handle APIs

Here is some information about the cuFile Handle APIs.

The `cuFileHandleRegister` API makes a file descriptor or handle that is known to the cuFile subsystem by using an OS-agnostic interface. The API returns an opaque handle that is owned by the cuFile subsystem.

To conserve memory, the `cuFileHandleDeregister` API is used to release cuFile-related memory objects. Using only the POSIX close will not clean up resources that were used by cuFile. Additionally, the clean up of cuFile objects associated with the files that were operated on in the cuFile context will occur at `cuFileDriverClose`.

```
CUfileError_t cuFileHandleRegister(CUFileHandle_t *fh, CUFileDescr_t *descr);
void cuFileHandleDeregister(CUFileHandle_t fh);
```

Note: Refer to sample_003 for usage.

3.5. cuFile Buffer APIs

The `cuFileBufRegister` API incurs a significant performance cost, so registration costs should be amortized where possible. Developers must ensure that buffers are registered up front and off the critical path.

The `cuFileBufRegister` API is optional. If this is not used, instead of pinning the user's memory, cuFile-managed and internally pinned buffers are used.

The `cuFileBufDeregister` API is used to optimally clean up cuFile-related memory objects, but CUDA currently has no analog to `cuFileBufDeregister`. The cleaning up of objects associated with the buffers operated on in the cuFile context occurs at `cuFileDriverClose`. If explicit APIs are used, the incurred errors are reported immediately, but if the operations of these explicit APIs are performed implicitly, error reporting and handling are less clear.

```
CUfileError_t cuFileBufRegister(const void *devPtr_base, size_t size, int flags);
CUfileError_t cuFileBufDeregister(const void *devPtr_base);
```

Note: Refer to sample_005 for usage.

3.6. cuFile Stream APIs

Operations that are enqueued with cuFile Stream APIs are FIFO ordered with respect to other work on the stream and must be completed before continuing with the next action in the stream.

```
CUfileError_t cuFileReadAsync(CUFileHandle_t fh, void *bufPtr_base,
                  size_t *size_p, off_t *file_offset_p, off_t *bufPtr_offset_p,
                  ssize_t *bytes_read_p, CUStream stream);
CUfileError_t cuFileWriteAsync(CUFileHandle_t fh, void *bufPtr_base,
                  size_t *size_p, off_t *file_offset_p, off_t *bufPtr_offse_pt,
                  ssize_t *bytes_written_p, CUstream stream);
```

Note: Refer to samples sample_031, sample_032, sample_033, and sample_034 for usage.

3.7. cuFile Batch APIs

Batch APIs are submitted synchronously, but executed asynchronously with respect to host thread.

These operations can be submitted on different files, different locations in the same file, or a mix. Completion of IO can be checked asynchronously using a status API in the same host thread or in a different thread. The `cuFileBatchIOGetStatus` API takes an array of `CUfileIOEvents_t` and minimum number of elements to poll for. which describes the IO

action, status, errors, and bytes transacted for each instance. The bytes transacted field is valid only when the status indicates a successful completion.

Note: Refer to samples sample_019, sample_020, sample_021, and sample_022 for usage.

Chapter 4. cuFile API Functional Specification

This section provides information about the cuFile API functional specification.

See the GPUDirect Storage Overview Guide for a high-level analysis of the set of functions and their relation to each other. We anticipate adding additional return codes for some of these functions.

All cuFile APIs are called from the host code.

4.1. cuFileDriver API Functional Specification

This section provides information about the cuFileDriver API functional specification.

4.1.1. cuFileDriverOpen

```
CUfileError_t cuFileDriverOpen();
```

Opens the Driver session to support GDS IO operations.

Parameters

- None

Returns

- `CU_FILE_SUCCESS` on a successful open, or if the driver is already open.
- `CU_FILE_DRIVER_NOT_INITIALIZED` on a failure to open the driver.
- `CU_FILE_PERMISSION_DENIED` on a failure to open.

  This can happen when the character device (`/dev/nvidia_fs[0-15]`) is restricted to certain users by an administrator, for example, admin, where `/dev` is not exposed with read permissions in the container.
- `CU_FILE_DRIVER_VERSION_MISMATCH`, when there is a mismatch between the cuFile library and its kernel driver.
- `CU_FILE_CUDA_DRIVER_ERROR` if the CUDA driver failed to initialize.
  `CU_FILE_PLATFORM_NOT_SUPPORTED` if the current platform is not supported by GDS.

- `CU_FILE_NVFS_SETUP_ERROR` for a cuFile-specific internal error.

Refer to the `cufile.log` file for more information.

Description

- This API opens the session with the NVFS kernel driver to communicate from userspace to kernel space and calls the GDS driver to set up the resources required to support GDS IO operations.
- The API checks whether the current platform supports GDS and initializes the cuFile library.
- This API loads the cuFile settings from a JSON configuration file in `/etc/cufile.JSON`.

  If the JSON configuration file does not exist, the API loads the default library settings. To modify this default config file, administrative privileges are needed. The administrator can modify it to grant cuFile access to the specified devices and mount paths and also tune IO parameters (in KB, 4K aligned) that are based on the type of workload. Refer to the default config file (`/etc/cufile.json`) for more information.

4.1.2. cuFileDriverClose

```
CUfileError_t cuFileDriverClose();
```

- Closes the driver session and frees any associated resources for GDS.
- This happens implicitly upon process exit.
- The driver can be reopened once it is closed.

Parameters

- None

Returns

- `CU_FILE_SUCCESS` on a successful close.
- `CU_FILE_DRIVER_NOT_INITIALIZED` on failure.

Description

- Close the GDS session and any associated memory resources. If there are buffers registered by using `cuFileBufRegister`, which are not unregistered, a `cuFileDriverClose` implicitly unregisters those buffers. Any in-flight IO when `cuFileDriverClose` is in-progress will receive an error.

4.1.3. cuFileDriverGetProperties

The `cuFileDrvProps_t` structure can be queried with `cuFileDriverGetProperties` and selectively modified with `cuFileDriverSetProperties`. The structure is self-describing, and its fields are consistent with the major and minor API version parameters.

```
CUfileError_t cuFileDriverGetProperties(cuFileDrvProps_t *props);
```

- Gets the Driver session properties for GDS functionality.

Parameters

`props`

- Pointer to the cuFile Driver properties.

Returns

- `CU_FILE_SUCCESS` on a successful completion.
- `CU_FILE_DRIVER_NOT_INITIALIZED` on failure.
- `CU_FILE_DRIVER_VERSION_MISMATCH` on a driver version mismatch.
- `CU_FILE_INVALID_VALUE` if input is invalid.

Description

This API is used to get current GDS properties and nvidia-fs driver properties and functionality, such as support for SCSI, NVMe, and NVMe-OF.

This API is used to get the current `nvidia-fs` drivers-specific properties such as the following:

- `major_version`: the cuFile major version
- `minor_version`: the cuFile minor version
- `props.nvfs.dstatusflags`, which are bit flags that indicate support for the following driver features:
  - `CU_FILE_EXASCALER_SUPPORTED`, a bit to check whether the DDN EXAScaler parallel filesystem solutions (based on the Lustre filesystem) client supports GDS.
  - `CU_FILE_WEKAFS_SUPPORTED`, a bit to check whether WekaFS supports GDS.
- `Props.nvfs.dcontrolflags`, which are bit flags that indicate the current activation for driver features:
  - `CU_FILE_USE_POLL_MODE`, when bit is set, IO uses polling mode.
  - `CU_FILE_ALLOW_COMPAT_MODE`, if the value is 1 compatible mode is set.

    Otherwise, the compatible mode is disabled.
- `Props.fflags`, which are bit flags that indicate whether the following library features are supported:
  - `CU_FILE_STREAMS_SUPPORTED`, an attribute that checks whether CUDA-streams are supported.
  - `CU_FILE_DYN_ROUTING_SUPPORTED`, an attribute that checks whether dynamic routing feature is supported.
- `Props.nvfs.poll_thresh_size`, a maximum IO size, in KB and must be 4K-aligned, that is used for the POLLING mode.
- `Props.nvfs.max_direct_io_size`, a maximum GDS IO size, in KB and must be 4K-aligned, that is requested by the nvidia-fs driver to the underlying filesystem.
- `Props.max_device_cache_size`, a maximum GPU buffer space per device, in KB and must be 4K-aligned. Used internally, for example, to handle unaligned IO and optimal IO path routing. This value might be rounded down to the nearest GPU page size.

- `Props.max_device_pinned_mem_size`, a maximum buffer space, in KB and must be 4K-aligned, that is pinned and mapped to the GPU BAR space. This might be rounded down to the nearest GPU page size.
- `Props.per_buffer_cache_size`, a GPU bounce buffer size, in KB, used for internal pools.

Additional Information

See the following for more information:

- cuFileDriverSetPollMode(bool poll, size_t poll_threshold_size)
- cuFileDriverSetMaxDirectIOSize(size_t max_direct_io_size)
- cuFileDriverSetMaxCacheSize(size_t max_cache_size)
- cuFileDriverSetMaxPinnedMemSize(size_t max_pinned_memory_size)

4.1.4. cuFileDriverSetPollMode(bool poll, size_t poll_threshold_size)

`cuFileDriverSetPollMode(bool poll, size_t poll_threshold_size)` API

```
CUfileError_t cuFileDriverSetPollMode(bool poll,
                                      size_t poll_threshold_size);
```

- Sets whether the Read/Write APIs use polling to complete IO operations. If poll mode is enabled, an IO size less than or equal to the threshold value is used for polling.
- The `poll_threshold_size` must be 4K aligned.

Parameters

`poll`

- Boolean to indicate whether to use the poll mode.

`poll_threshold_size`

- IO size to use for POLLING mode in KB.
- The default value is 4KB.

Returns

- `CU_FILE_SUCCESS` on a successful completion.
- `CU_FILE_DRIVER_NOT_INITIALIZED` on failure to load the driver.
- `CU_FILE_DRIVER_UNSUPPORTED_LIMIT` on failure to set with valid threshold size

Description

This API is used in conjunction with `cuFileGetDriverProperties`. This API is used to set whether the library should use polling and the maximum IO threshold size less than or equal to which it will poll.

This API overrides the default value that may be set through the JSON configuration file using the config keys `properties.poll_mode` and `properties.poll_max_size_kb` for the current process.

See the following for more information:

- cuFileDriverGetProperties

4.1.5. cuFileDriverSetMaxDirectIOSize(size_t max_direct_io_size)

```
CUfileError_t cuFileDriverSetMaxDirectIOSize(size_t max_direct_io_size);
```

- Sets the max IO size, in KB.

  This parameter is used by the nvidia-fs driver as the maximum IO chunk size in which IO is issued to the underlying filesystem. In compatible mode, this is the maximum IO chunk size that the library uses to issue POSIX read/writes.
- The max direct IO size must be 4K aligned.

Parameters

`max_direct_io_size`

- The maximum allowed direct IO size in KB.
- The default value is 16384KB. This is because typically parallel-file systems perform better with bulk read/writes.

Returns

- `CU_FILE_SUCCESS` on successful completion.
- `CU_FILE_DRIVER_NOT_INITIALIZED` on failure to load the driver.
- `CU_FILE_DRIVER_UNSUPPORTED_LIMIT` on failure to set with valid size.

Description

This API is used with `cuFileGetDriverProperties` and is used to set the maximum direct IO size used by the library to specify the nvidia-fs kernel driver the maximum chunk size in which the latter can issue IO to the underlying filesystem. In compatible mode, this is the maximum IO chunk size which the library uses for issuing POSIX read/writes. This parameter is dependent on the underlying GPU hardware and system memory.

This API overrides the default value that might be set through the JSON configuration file by using the `properties.max_direct_io_size_kb` config key for the current process.

Refer to the following for more information:

- cuFileDriverGetProperties

4.1.6. cuFileDriverSetMaxCacheSize(size_t max_cache_size)

```
CUfileError_t cuFileDriverSetMaxCacheSize(size_t max_cache_size);
```

- Sets the maximum GPU buffer space, in KB, per device and is used for internal use, for example, to handle unaligned IO and optimal IO path routing. This value might be rounded down to the nearest GPU page size.
- The max cache size must be 4K aligned.
- This API overrides the default value that might be set through the JSON configuration file using the `properties.max_device_cache_size_kb` config key for the current process.

Parameters

`max_cache_size`

- The maximum GPU buffer space, in KB, per device used for internal use, for example, to handle unaligned IO and optimal IO path routing. This value might be rounded down to the nearest GPU page size.
- The default value is 131072KB.

Returns

- `CU_FILE_SUCCESS` on successful completion.
- `CU_FILE_DRIVER_NOT_INITIALIZED` on failure to load the driver.
- `CU_FILE_DRIVER_UNSUPPORTED_LIMIT` on failure to set with valid IO size

Description

This API is used with `cuFileGetDriverProperties` and is used to set the upper limit on the cache size per device for internal use by the library.

See cuFileDriverGetProperties for more information.

4.1.7. cuFileDriverSetMaxPinnedMemSize(size_t max_pinned_memory_size)

```
CUfileError_t cuFileDriverSetMaxPinnedMemSize(size_t max_pinned_mem_size);
```

- Sets the maximum GPU buffer space, in KB, that is pinned and mapped. This value might be rounded down to the nearest GPU page size.
- The max pinned size must be 4K aligned.
- The default value corresponds to the maximum `PinnedMemory` or the physical memory size of the device.
- This API overrides the default value that may be set by the `properties.max_device_pinned_mem_size_kb` JSON config key for the current process.

Parameters

`max_pinned_memory_size`

- The maximum buffer space, in KB, that is pinned and mapped to the GPU BAR space.
- This value might be rounded down to the nearest GPU page size.
- The maximum limit may be set to UINT64_MAX, which is equivalent to no enforced limit. It may be set to something smaller than the size of the GPU's physical memory.

Returns

- `CU_FILE_SUCCESS` on successful completion.
- `CU_FILE_DRIVER_NOT_INITIALIZED` on failure to load driver.
- `CU_FILE_DRIVER_UNSUPPORTED_LIMIT` on failure to set with valid size.

Description

This API is used with `cuFileGetDriverProperties` and is used to set an upper limit on the maximum size of GPU memory that can be pinned and mapped and is dependent on the underlying GPU hardware and system memory. This API is related to `cuFileBufRegister`, which is used to register GPU device memory. SeecuFileDriverGetProperties for more information.

4.2. cuFile IO API Functional Specification

This section provides information about the cuFile IO API function specification.

The device pointer addresses referred to in these APIs pertain to the current context for the caller.

Unlike the non-async version of `cuMemcpy`, the `cuFileHandleRegister`, `cuFileHandleDeregister`, `cuFileRead`, and `cuFileWrite` APIs do not have the semantic of being ordered with respect to other work in the null stream.

4.2.1. cuFileHandleRegister

```
CUfileError_t cuFileHandleRegister(CUFileHandle_t *fh, CUfileDescr_t *descr);
```

- Register an open file.
- `cuFileHandleRegister` is required and performs extra checking that is memoized to provide increased performance on later cuFile operations.
- This API is OS agnostic.

> Note: CUDA toolkit 12.2 (GDS version 1.7.x) supports non O_DIRECT open flags as well as O_DIRECT. Application is allowed to open a file in non O_DIRECT mode in compat mode and also with nvidia-fs.ko installed. In the latter case, an O_DIRECT path between GPU and Storage will be used if such a path exists.

Parameters

- `fh`

  Valid pointer to the OS-neutral cuFile handle structure supplied by the user but populated and maintained by the cuFile runtime.

- `desc`

  Valid pointer to the OS-neutral file descriptor supplied by the user carrying details regarding the file to be opened such as fd for Linux-based files.

Returns

- `CU_FILE_SUCCESS` on successful completion.
- `CU_FILE_DRIVER_NOT_INITIALIZED` on failure to load the driver.
- `CU_FILE_IO_NOT_SUPPORTED`, if the filesystem is not supported.
- `CU_FILE_INVALID_VALUE` if there are null or bad API arguments.
- `CU_FILE_INVALID_FILE_OPEN_FLAG`, if the file is opened with unsupported modes such as no `O_APPEND`, `O_NOCTTY`, `O_NONBLOCK`, `O_DIRECTORY`, `O_NOFOLLOW`, `O_NOATIME`, and `O_TMPFILE`.
- `CU_FILE_INVALID_FILE_TYPE`, if the file path is not valid, not a regular file, not a symbolic link, or not a device file.
- `CU_FILE_HANDLE_ALREADY_REGISTERED` if the file is already registered using the same file-descriptor.

Description

- Given a file-descriptor will populate and return the `CUfileHandle_t` needed for issuing IO with cuFile APIs.
- A return value of anything other than CU_FILE_SUCCESS leaves fh in an undefined state but has no other side effects.
- By default this API accepts whether the file descriptor is opened with O_DIRECT mode or non O_DIRECT mode.

Refer to the following for more information:

- cuFileRead
- cuFileWrite
- cuFileReadAsync
- cuFileWriteAsync
- cuFileHandleDeregister

4.2.2. cuFileHandleDeregister

```
CUfileError_t cuFileHandleDeregister(CUFileHandle_t *fh);
```

Parameters

- `fh`

The file handle obtained from cuFileHandleRegister.

Returns

None

Note: This API only logs an ERROR level message in the cufile.log file for valid inputs.

Description

- The API is used to release resources that are claimed by `cuFileHandleRegister`.

  This API should be invoked only after the application ensures there are no outstanding IO operations with the handle. If `cuFileHandleDeregister` is called while IO on the file is in progress might result in undefined behavior.
- The user is still expected to close the file descriptor outside the cuFile subsystem after calling this API using `close` system call.

  Closing a file handle without calling `cuFileHandleDeregister` does not release the resources that are held in the cuFile library. If this API is not called, the cuFile subsystem releases the resources lazily or when the application exits.

See the following for more information:

- cuFileRead
- cuFileWrite
- cuFileHandleDeregister

4.2.3. cuFileRead

```
ssize_t cuFileRead(CUfileHandle_tfh, void *bufPtr_base, size_t size, off_t
 file_offset, off_t bufPtr_offset);
```

- Reads specified bytes from the file descriptor into the device memory or the host memory.

Parameters

- `fh`

  File descriptor for the file.
- `bufPtr_base`

  Base address of buffer in device memory or host memory. For registered buffers, `bufPtr_base` must remain set to the base address used in the `cuFileBufRegister` call.
- `size`

  Size in bytes to read.
- `file_offset`

  Offset in the file to read from.
- `bufPtr_offset`

Offset relative to the `bufPtr_base` pointer to read into. This parameter should be used only with registered buffers.

Returns

- Size of bytes that were successfully read.
- -1 on an error, so errno is set to indicate filesystem errors.
- All other errors return a negative integer value of the `CUfileOpError` enum value.

Description

This API reads the data from a specified file handle at a specified offset and size bytes into the GPU memory by using GDS functionality or into the host memory based on the type of memory pointer. The API works correctly for unaligned offsets and any data size, although the performance might not match the performance of aligned reads.This is a synchronous call and blocks until the IO is complete.

Note: For the `bufPtr_offset`, if data will be read starting exactly from the `bufPtr_base` that is registered with `cuFileBufRegister`, `bufPtr_offset` should be set to 0. To read starting from an offset in the registered buffer range, the relative offset should be specified in the `bufPtr_offset`, and the `bufPtr_base` must remain set to the base address that was used in the `cuFileBufRegister` call.

See the following for more information:

- cuFileWrite
- cuFileReadAsync
- cuFileWriteAsync

4.2.4. cuFileWrite

```
ssize_t cuFileWrite(CUfileHandle_t fh, const void *bufPtr_base, size_t size, off_t
 file_offset, off_t bufPtr_offset);
```

- Writes specified bytes from the device memory into the file descriptor using GDS.

Parameters

- `fh`

  File descriptor for the file

- `bufPtr_base`

  Base address of buffer in device memory or host memory. For registered buffers, `bufPtr_base` must remain set to the base address used in the `cuFileBufRegister` call.

- `size`

  Size in bytes to which to write.

- `file_offset`

  Offset in the file to which to write.

- `bufPtr_offset`

  Offset relative to the `bufPtr_base` pointer from which to write. This parameter should be used only with registered buffers.

Returns

- Size of bytes that were successfully written.
- -1 on an error, so errno is set to indicate filesystem errors.
- All other errors return a negative integer value of the `CUfileOpError` enum value.

Description

This API writes the data from the GPU memory or the host memory to a file specified by the file handle at a specified offset and size bytes by using GDS functionality. The API works correctly for unaligned offset and data sizes, although the performance is not on-par with aligned writes.This is a synchronous call and will block until the IO is complete.

Note: GDS functionality modified the standard file system metadata in SysMem. However, GDS functionality does not take any special responsibility for writing that metadata back to permanent storage. The data is not guaranteed to be present after a system crash unless the application uses an explicit `fsync(2)` call. If the file is opened with an `O_SYNC` flag, the metadata will be written to the disk before the call is complete.

Refer to the note in cuFileRead for more information about `bufPtr_offset:`.

Refer to the following for more information:

- cuFileWrite
- cuFileReadAsync
- cuFileWriteAsync

4.3. cuFile Memory Management Functional Specification

The device pointer addresses that are mentioned in the APIs in this section pertain to the current context for the caller. cuFile relies on users to complete their own allocation before using the `cuFileBufRegister` API and free after using the `cuFileBufDeregister` API.

4.3.1. cuFileBufRegister

```
CUfileError_t cuFileBufRegister(const void *bufPtr_base,
                                size_t size, int flags);
```

- Based on the memory type, this API registers existing cuMemAlloc'd (pinned) memory for GDS IO operations or host memory for IO operations.

Parameters

- `bufPtr_base`

  Address of device pointer. `cuFileRead` and `cuFileWrite` must use this `bufPtr_base` as the base address.

- `size`

  Size in bytes from the start of memory to map.

- `flags`

  Reserved for future use, must be 0.

Returns

- `CU_FILE_SUCCESS` on a successful registration.
- `CU_FILE_NVFS_DRIVER_ERROR` if the nvidia-fs driver cannot handle the request.
- `CU_FILE_INVALID_VALUE` on a failure.
- `CU_FILE_CUDA_DRIVER_ERROR` on CUDA-specific errors. CUresult code can be obtained using `CU_FILE_CUDA_ERR`(err).
- `CU_FILE_MEMORY_ALREADY_REGISTERED`, if memory is already registered.
- `CU_FILE_INTERNAL_ERROR`, an internal library-specific error.
- `CU_FILE_CUDA_MEMORY_TYPE_INVALID`, for device memory that is not allocated via `cudaMalloc` or `cuMemAlloc`.
- `CU_FILE_CUDA_POINTER_RANGE_ERROR`, if the size exceeds the bounds of the allocated memory.
- `CU_FILE_INVALID_MAPPING_SIZE`, if the size exceeds the GPU resource limits.
- `CU_FILE_GPU_MEMORY_PINNING_FAILED`, if not enough pinned memory is available.

Description

Based on the memory type, this API either registers the specified GPU address or host memory address and size for use with the `cuFileRead` and `cuFileWrite` operations. The user must call `cuFileBufDeregister` to release the pinned memory mappings for GPU memory if needed.

See the following for more information:

- cuFileBufDeregister

4.3.2. cuFileBufDeregister

```
CUfileError_t cuFileBufDeregister(const void *bufPtr_base);
```

- Based on the memory type, this API either deregisters CUDA memory or the host memory registered using the `cuFileBufRegister` API.

Parameters

- `bufPtr_base`

  Address of device pointer to release the mappings that were provided to `cuFileBufRegister`

Returns

- `CU_FILE_SUCCESS` on a successful deregistration.
- `CU_FILE_MEMORY_NOT_REGISTERED`, if `bufPtr_base` was not registered.
- `CU_FILE_ERROR_INVALID_VALUE` on failure to find the registration for the specified memory.
- `CU_FILE_INTERNAL_ERROR`, an internal library-specific error.

Description

This API deregisters memory mappings that were registered by `cuFileBufRegister`. Refer to cuFileBufRegister for more information.

4.4. cuFile Stream API Functional Specification

This section provides information about the cuFile stream API functional specification.

The stream APIs are similar to Read and Write, but they take a stream parameter to support asynchronous operations and execute in the CUDA stream order.

4.4.1. cuFileStreamRegister

```
CUfileError_t cuFileStreamRegister(CUStream_t stream, unsigned flags);
```

- Defines the input behavior for stream I/O APIs.

Parameters

- `stream`

  CUDA stream in which to enqueue the operation. If NULL, make this operation in the default CUDA stream.

- `flags`

  The following are valid values:

Value	Description
0x0	All the I/O parameters are valid only at the time of execution.
0x1	Buffer offset value is valid at submission time.
0x2	File offset value is valid at submission time.
0x4	Size is valid at submission time.
0x8	All inputs i.e. buffer offset, file offset and size are 4K aligned.

Value	Description
0xf	All inputs are aligned and known at submission time.

Note: Using the flag '0XF' will perform best as the workflow can be optimized during submission time.

Description

This optional API registers the stream with the cuFile subsystem.

This API will allocate resources to handle stream operations for cuFile.

The API will synchronize on the stream before allocating resources.

The stream pointer is expected to be a valid pointer.

Returns

- `CU_FILE_SUCCESS` on a successful submission.
- `CU_FILE_ERROR_INVALID_VALUE`on a invalid stream specification.
- `CU_FILE_DRIVER_ERROR` if the NVIDIA-fs driver cannot handle the request.
- `CU_FILE_PLATFORM_NOT_SUPPORTED` on unsupported platforms.

4.4.2. cuFileStreamDeregister

```
CUfileError_t cuFileStreamDeregister(CUStream_t stream);
```

Parameters

- `stream`

  CUDA stream in which to enqueue the operation. If NULL, make this operation in the default CUDA stream.

- `flags`

  Reserved for future use.

Description

This optional API deregisters the stream with the cuFile subsystem.

This API will free allocated cuFile resources associated with the stream.

The API will synchronize on the stream before releasing resources.

The stream pointer is expected to be a valid pointer.

The stream will be automatically deregistered as part of `cuFileDriverClose`.

Returns

- `CU_FILE_SUCCESS` on a successful submission.
- `CU_FILE_ERROR_INVALID_VALUE`on a invalid stream specification.

- `CU_FILE_PLATFORM_NOT_SUPPORTED` on unsupported platforms.

4.4.3. cuFileReadAsync

```
CUfileError_t cuFileReadAsync(CUFileHandle_t fh,
                        void *bufPtr_base,
                        size_t *size_p,
                        off_t file_offset_p,
                        off_t bufPtr_offset_p,
                        int *bytes_read_p,
                        CUstream stream);
```

- Enqueues a read operation for the specified bytes from the cuFile handle into the device memory by using GDS functionality or to the host memory based on the type of memory pointer.
- If non-NULL, the action is ordered in the stream.
- The current context of the caller is assumed.

Parameters

- `fh`

  The cuFile handle for the file.
- `bufPtr_base`
  - The base address of the buffer in the memory into which to read.
  - The buffer can be allocated using either `cudaMemory`/`cudaMallocHost`/`malloc`/`mmap`.
  - For registered buffers, `bufPtr_base` must remain set to the base address used in `cuFileBufRegister` call.
- `size_p`

  Pointer to size in bytes to read. If the exact size is not known at the time of I/O submission, then you must set it to the maximum possible I/O size for that stream I/O.
- `file_offset_p`

  Pointer to offset in the file from which to read. Unless otherwise set using `cuFileStreamRegister` API, this value will not be evaluated until execution time.
- `bufPtr_offset_p`

  Pointer to the offset relative to the `bufPtr_base` pointer from which to write. Unless otherwise set using cuFileStreamRegister API, this value will not be evaluated until execution time.
- `bytes_read_p`

  Pointer to the bytes read from the specified filehandle. This pointer should be a non NULL value and `*bytes_read_p` set to 0. After successful execution of the operation in the stream, the value `*bytes_read_p` will contain either:
  - The number of bytes successfully read.
  - -1 on IO errors.

- All other errors return a negative integer value of the `CUfileOpError` enum value.

- `stream`
  - CUDA stream in which to enqueue the operation.
  - If NULL, make this operation synchronous.

Returns

- `CU_FILE_SUCCESS` on a successful submission.
- `CU_FILE_DRIVER_ERROR`, if the nvidia-fs driver cannot handle the request.
- `CU_FILE_ERROR_INVALID_VALUE` on an input failure.
- `CU_FILE_CUDA_ERROR` on CUDA-specific errors.

  CUresult code can be obtained by using `CU_FILE_CUDA_ERR(err)`.

Description

- This API reads the data from the specified file handle at the specified offset and size bytes into the GPU memory using GDS functionality.

  This is an asynchronous call and enqueues the operation into the specified CUDA stream and will not block the host thread for IO completion. The operation can be waited upon using `cuStreamSynchronize(stream)`.
- The `bytes_read_p` memory should be allocated with `cuMemHostAlloc/malloc/mmap` or registered with `cuMemHostRegister`.

  The pointer to access that memory from the device can be obtained by using `cuMemHostGetDevicePointer`.
- Operations that are enqueued with cuFile Stream APIs are FIFO ordered with respect to other work on the stream and must be completed before continuing to the next action in the stream.
- Unless otherwise specified through `cuFileStreamRegister` API, file offset, buffer offset or size parameter will not be evaluated until execution time. In these scenarios, size parameters should be set to the maximum possible I/O size at the time of submission and can be set to the actual size prior to the stream I/O execution.

Refer to the following for more information:

- cuFileRead
- cuFileWrite
- cuFileWriteAsync

4.4.4. cuFileWriteAsync

```
CUfileError_t cuFileWriteAsync(CUFileHandle_t fh,
                        void *bufPtr_base,
                        size_t *size_p,
                        off_t file_offset_p,
                        off_t bufPtr_offset_p,
                        int *bytes_written_p,
                        CUstream_t stream);
```

- Queues Write operation for the specified bytes from the device memory into the cuFile handle by using GDS.

Parameters

- `fh`

  The cuFile handle for the file.
- `bufPtr_base`

  The base address of the buffer in the memory from which to write. The buffer can be allocated using either `cudaMemory/cudaMallocHost/malloc/mmap`. For registered buffers, `bufPtr_base` must remain set to the base address used in the `cuFileBufRegister` call.
- `size_p`

  Pointer to the size in bytes to write. If the exact size is not known at the time of I/O submission, then you must set it to the maximum possible I/O size for that stream I/O.
- `file_offset_p`

  Pointer to the offset in the file from which to write. Unless otherwise set using `cuFileStreamRegister` API, this value will not be evaluated until execution time.
- `bufPtr_offset_p`

  Pointer to the offset relative to the `bufPtr_base` pointer from which to write. Unless otherwise set using cuFileStreamRegister API, this value will not be evaluated until execution time.
- `bytes_written_p`

  Pointer to the bytes written to the specified filehandle.This pointer should be a non NULL value and `*bytes_written_p` set to 0. After successful execution of the operation in the stream, the value `*bytes_written_p` will contain either:
  - The number of bytes successfully written.
  - -1 on IO errors.
  - All other errors will return a negative integer value of the `CUfileOpError` enum value.
- `stream`

  The CUDA stream to enqueue the operation.

Returns

- `CU_FILE_SUCCESS` on a successful submission.
- `CU_FILE_DRIVER_ERROR`, if the nvidia-fs driver cannot handle the request.
- `CU_FILE_ERROR_INVALID_VALUE` on an input failure.
- `CU_FILE_CUDA_ERROR` on CUDA-specific errors.

  The CUresult code can be obtained by using `CU_FILE_CUDA_ERR(err)`.

Description

- This API writes the data from the GPU memory to a file specified by the file handle at a specified offset and size bytes by using GDS functionality. This is an asynchronous call and enqueues the operation into the specified CUDA stream and will not block the host thread for IO completion. The operation can be waited upon by using `cuStreamSynchronize(stream)`.
- The `bytes_written` pointer should be allocated with `cuMemHostAlloc` or registered with `cuMemHostRegister`, and the pointer to access that memory from the device can be obtained by using `cuMemHostGetDevicePointer`.
- Operations that are enqueued with cuFile Stream APIs are FIFO ordered with respect to other work on the stream and must be completed before continuing to the next action in the stream.
- Unless otherwise specified through `cuFileStreamRegister` API, file offset, buffer offset or size parameter will not be evaluated until execution time. In these scenarios, size parameters should be set to the maximum possible I/O size at the time of submission and can be set to the actual size prior to the stream I/O execution.

See the following for more information:

- cuFileRead
- cuFileWrite
- cuFileReadAsync

4.5. cuFile Batch API Functional Specification

This section provides information about the cuFile Batch API functional specification.

4.5.1. cuFileBatchIOSetUp

```
CUfileError_t
cuFileBatchIOSetUp(CUfileBatchHandle_t *batch_idp, int max_nr);
```

Parameters

- `max_nr`

  (Input) The maximum number of events this batch will hold.

  Note: The number should be between 1 - “`properties.io_batch_size`”

- `batch_idp`

  (Output) Will be used in subsequent batch IO calls.

Returns

- `CU_FILE_SUCCESS` on success.
- `CU_FILE_INTERNAL_ERROR` on on any failures.

Description

This interface should be the first call in the sequence of batch I/O operation. This takes the maximum number of batch entries the caller intends to use and returns a `CUFileBatchHandle_t` which should be used by the caller for subsequent batch I/O calls.

See the following for more information:

- cuFileRead
- cuFileWrite
- cuFileReadAsync
- cuFileWriteAsync
- cuFileBatchIOGetStatus
- cuFileBatchIOCancel
- cuFileBatchIODestroy

4.5.2. cuFileBatchIOSubmit

```
CUfileError_t cuFileBatchIOSubmit(CUfileBatchHandle_t batch_idp,
                                  unsigned nr,
                                  CUfileIOParams_t *iocbp,
                                  unsigned int flags)
```

Parameters

- `batch_idp`

  The address of the output parameter for the newly created batch ID, which was obtained from a `cuFileBatchSetup` call.

- `nr`
  - The number of requests for the batch request.
  - The value must be greater than 0 and less than or equal to `max_nr` specified in `cuFileBatchIOSetup`.
- `iocbp`

  The pointer contains the `CUfileIOParams_t` array structures of the length `nr` array.

- `flags`

  Reserved for future use. Should be set to 0.

Returns

- `CU_FILE_SUCCESS` on success.
- `CU_FILE_INTERNAL_ERROR` on any failures.

Description

- This API will need to be used to submit a read/write operation on an array of GPU/CPU data pointers from their respective file handle, offset, and size bytes.

Based on the type of memory pointer, the data is transferred to/from the GPU memory by using GDS or the data is transferred to/from the CPU memory.

- This is an asynchronous call and will enqueue the operation on a `batch_id` provided by the `cuFileIOSetup` API. The operation can be monitored when using this `batch_id` through `cuFileBatchIOGetStatus`.
- The operation can be canceled by calling `cuFileBatchIOCancel` or destroyed by `cuFileBatchIODestroy`.

- The entries in the `CUfileIOParams_t` array describe individual IOs.

  The bytes transacted field is valid only when the status indicates a completion.
- Operations that are enqueued with cuFile Batch APIs are FIFO ordered with respect to other work on the stream and must be completed before continuing to the next action in the stream. Operations in each batch might be reordered with respect to each other.
- The status field of individual IO operations via `CUfileIOParams_t` entries will have undefined values before the entire batch is complete. This definition is subject to change.

See the following for more information:

- cuFileRead
- cuFileWrite
- cuFileReadAsync
- cuFileWriteAsync
- cuFileBatchIOGetStatus
- cuFileBatchIOCancel
- cuFileBatchIODestroy

4.5.3. cuFileBatchIOGetStatus

```
CUfileError_t cuFileBatchIOGetStatus(CUfileBatchHandle_t batch_idp,
                                     unsigned min_nr,
                                     unsigned *nr,
                                     CUfileIOEvents_t *iocbp,
                                     struct timespec* timeout));
```

Parameters

- `batch_idp`

  Obtained during setup.
- `min_nr`

  The minimum number of IO entries for which status is requested. The `min_nr` should be greater than or equal to zero and less than or equal to `*nr`.
- `nr`

This is a pointer to max requested IO entries to poll for completion and is used as an Input/Output parameter. As an input `*nr` must be set to pass the maximum number of IO requests to poll for. As an output, `*nr` returns the number of completed I/Os.

- `iocbp`

  `CUFileIOEvents_t` array containing the status of completed I/Os in that batch.

- `timeout`

  This parameter is used to specify the amount of time to wait for in this API, even if the minimum number of requests have not completed. If the timeout hits, it is possible that the number of returned IOs can be less than `min_nr`.

Returns

- `CU_FILE_SUCCESS` on success.

  The success here refers to the completion of the API. Individual IO status and error can be obtained by examining the returned status and error in the array iocbp.

- `CU_FILE_ERROR_INVALID_VALUE` for an invalid batch ID.

Description

- This is a batch API to monitor the status of batch IO operations by using the `batch_id` that was returned by `cuFileBatchIOSubmit`. The operation will be canceled automatically if `cuFileBatchIOCancel` is called and the status will reflect `CU_FILE_CANCELED` for all canceled IO operations.
- The status of each member of the batch is queried, which would not be possible with one `CUEvent`. The status field of individual IO operations via `CUfileIOParams_t` entries will have undefined values before the entire batch is completed. This definition is subject to change.

See the following for more information:

- cuFileBatchIOSubmit
- cuFileBatchIODestroy

4.5.4. cuFileBatchIOCancel

```
CUfileError_t cuFileBatchIOCancel(CUfileBatchHandle_t batch_idp)
```

Parameters

- `batch_idp`

  The batch ID to cancel.

Returns

- `CU_FILE_SUCCESS` on success.
- `CU_FILE_ERROR_INVALID_VALUE`on any failures.

Description

- This is a batch API to cancel an ongoing IO batch operation by using the `batch_id` that was returned by `cuFileBatchIOSubmit`. This API tries to cancel an individual IO operation in the batch if possible and provides no guarantee about canceling an ongoing operation.

Refer to the following for more information:

- cuFileBatchIOGetStatus
- cuFileBatchIOSubmit
- cuFileBatchIODestroy

4.5.5. cuFileBatchIODestroy

```
void cuFileBatchIODestroy(CUfileBatchHandle_t batch_idp)
```

Parameters

- `batch_idp`

  The batch handle to be destroyed.

Returns

void

Description

This is a batch API that destroys a batch context and the resources that are allocated with `cuFileBatchIOSetup`.

Refer to the following for more information:

- cuFileBatchIOGetStatus
- cuFileBatchIOSubmit
- cuFileBatchIOCancel

Chapter 5. Sample Program with cuFile APIs

The following sample program uses the cuFile APIs:

```
// To compile this sample code:
//
// nvcc gds_helloworld.cxx -o gds_helloworld -lcufile
//
// Set the environment variable TESTFILE
// to specify the name of the file on a GDS enabled filesystem
//
// Ex:   TESTFILE=/mnt/gds/gds_test ./gds_helloworld
//
//
#include <fcntl.h>
#include <errno.h>
#include <unistd.h>

#include <cstdlib>
#include <cstring>
#include <iostream>
#include <cuda_runtime.h>
#include "cufile.h"

//#include "cufile_sample_utils.h"
using namespace std;

int main(void) {
        int fd;
        ssize_t ret;
        void *devPtr_base;
        off_t file_offset = 0x2000;
        off_t devPtr_offset = 0x1000;
        ssize_t IO_size = 1UL << 24;
        size_t buff_size = IO_size + 0x1000;
        CUfileError_t status;
        // CUResult cuda_result;
        int cuda_result;
        CUfileDescr_t cf_descr;
        CUfileHandle_t cf_handle;
        char *testfn;

        testfn=getenv("TESTFILE");
        if (testfn==NULL) {
            std::cerr << "No testfile defined via TESTFILE.  Exiting." << std::endl;
            return -1;
        }

        cout << std::endl;
        cout << "Opening File " << testfn << std::endl;
```

```
        fd = open(testfn, O_CREAT|O_WRONLY|O_DIRECT, 0644);
        if(fd < 0) {
                std::cerr << "file open " << testfn << "errno " << errno <<
std::endl;
                return -1;
        }

        // the above fd could also have been opened without O_DIRECT starting CUDA
toolkit 12.2
        // (gds 1.7.x version) as follows
        // fd = open(testfn, O_CREAT|O_WRONLY, 0644);

        cout << "Opening cuFileDriver." << std::endl;
        status = cuFileDriverOpen();
        if (status.err != CU_FILE_SUCCESS) {
                std::cerr << " cuFile driver failed to open " << std::endl;
                close(fd);
                return -1;
        }

        cout << "Registering cuFile handle to " << testfn << "." << std::endl;

        memset((void *)&cf_descr, 0, sizeof(CUfileDescr_t));
        cf_descr.handle.fd = fd;
        cf_descr.type = CU_FILE_HANDLE_TYPE_OPAQUE_FD;
        status = cuFileHandleRegister(&cf_handle, &cf_descr);
        if (status.err != CU_FILE_SUCCESS) {
                std::cerr << "cuFileHandleRegister fd " << fd << " status " <<
status.err << std::endl;
                close(fd);
                return -1;
        }

        cout << "Allocating CUDA buffer of " << buff_size << " bytes." << std::endl;

        cuda_result = cudaMalloc(&devPtr_base, buff_size);
        if (cuda_result != CUDA_SUCCESS) {
                std::cerr << "buffer allocation failed " << cuda_result <<
std::endl;
                cuFileHandleDeregister(cf_handle);
                close(fd);
                return -1;
        }

        cout << "Registering Buffer of " << buff_size << " bytes." << std::endl;
        status = cuFileBufRegister(devPtr_base, buff_size, 0);
        if (status.err != CU_FILE_SUCCESS) {
                std::cerr << "buffer registration failed " << status.err <<
std::endl;
                cuFileHandleDeregister(cf_handle);
                close(fd);
                cudaFree(devPtr_base);
                return -1;
        }

        // fill a pattern
        cout << "Filling memory." << std::endl;

        cudaMemset((void *) devPtr_base, 0xab, buff_size);
        cuStreamSynchronize(0);

        // perform write operation directly from GPU mem to file
        cout << "Writing buffer to file." << std::endl;
        ret = cuFileWrite(cf_handle, devPtr_base, IO_size, file_offset,
devPtr_offset);

        if (ret < 0 || ret != IO_size) {
```

```
                std::cerr << "cuFileWrite failed " << ret << std::endl;
        }

        // release the GPU memory pinning
        cout << "Releasing cuFile buffer." << std::endl;
        status = cuFileBufDeregister(devPtr_base);
        if (status.err != CU_FILE_SUCCESS) {
                std::cerr << "buffer deregister failed" << std::endl;
                cudaFree(devPtr_base);
                cuFileHandleDeregister(cf_handle);
                close(fd);
                return -1;
        }

        cout << "Freeing CUDA buffer." << std::endl;
        cudaFree(devPtr_base);
        // deregister the handle from cuFile
        cout << "Releasing file handle. " << std::endl;
        (void) cuFileHandleDeregister(cf_handle);
        close(fd);

        // release all cuFile resources
        cout << "Closing File Driver." << std::endl;
        (void) cuFileDriverClose();

        cout << std::endl;

        return 0;
}
```

Chapter 6. Known Limitations of cuFile Batch APIs

This section provides information about the known limitations of cuFile Batch APIs in this release of GDS.

- Batch I/Os will be supported mainly by either the local file systems which are hosted on NVMe or NVMeOF devices or by the native file system that supports Linux AIO. Following table provides an overview of the cuFile batch API support with respect to different file systems.

  The following table provides an overview of cuFile batch API support with respect to distributed file systems:

File System	GDS Batch Mode	Comments
Ext4/XFS	Read/Write support	
DDN EXAScaler	Read/Write support	
NFS	Read/Write support	
IBM Spectrum Scale	Not available	Will work in compat mode
Weka	Not available	Will work in compat mode
BeeGFS	Not available	Will work in compat mode

Notice

This document is provided for information purposes only and shall not be regarded as a warranty of a certain functionality, condition, or quality of a product. NVIDIA Corporation (“NVIDIA”) makes no representations or warranties, expressed or implied, as to the accuracy or completeness of the information contained in this document and assumes no responsibility for any errors contained herein. NVIDIA shall have no liability for the consequences or use of such information or for any infringement of patents or other rights of third parties that may result from its use. This document is not a commitment to develop, release, or deliver any Material (defined below), code, or functionality.

NVIDIA reserves the right to make corrections, modifications, enhancements, improvements, and any other changes to this document, at any time without notice.

Customer should obtain the latest relevant information before placing orders and should verify that such information is current and complete.

NVIDIA products are sold subject to the NVIDIA standard terms and conditions of sale supplied at the time of order acknowledgement, unless otherwise agreed in an individual sales agreement signed by authorized representatives of NVIDIA and customer (“Terms of Sale”). NVIDIA hereby expressly objects to applying any customer general terms and conditions with regards to the purchase of the NVIDIA product referenced in this document. No contractual obligations are formed either directly or indirectly by this document.

NVIDIA products are not designed, authorized, or warranted to be suitable for use in medical, military, aircraft, space, or life support equipment, nor in applications where failure or malfunction of the NVIDIA product can reasonably be expected to result in personal injury, death, or property or environmental damage. NVIDIA accepts no liability for inclusion and/or use of NVIDIA products in such equipment or applications and therefore such inclusion and/or use is at customer’s own risk.

NVIDIA makes no representation or warranty that products based on this document will be suitable for any specified use. Testing of all parameters of each product is not necessarily performed by NVIDIA. It is customer’s sole responsibility to evaluate and determine the applicability of any information contained in this document, ensure the product is suitable and fit for the application planned by customer, and perform the necessary testing for the application in order to avoid a default of the application or the product. Weaknesses in customer’s product designs may affect the quality and reliability of the NVIDIA product and may result in additional or different conditions and/or requirements beyond those contained in this document. NVIDIA accepts no liability related to any default, damage, costs, or problem which may be based on or attributable to: (i) the use of the NVIDIA product in any manner that is contrary to this document or (ii) customer product designs.

No license, either expressed or implied, is granted under any NVIDIA patent right, copyright, or other NVIDIA intellectual property right under this document. Information published by NVIDIA regarding third-party products or services does not constitute a license from NVIDIA to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property rights of the third party, or a license from NVIDIA under the patents or other intellectual property rights of NVIDIA.

Reproduction of information in this document is permissible only if approved in advance by NVIDIA in writing, reproduced without alteration and in full compliance with all applicable export laws and regulations, and accompanied by all associated conditions, limitations, and notices.

THIS DOCUMENT AND ALL NVIDIA DESIGN SPECIFICATIONS, REFERENCE BOARDS, FILES, DRAWINGS, DIAGNOSTICS, LISTS, AND OTHER DOCUMENTS (TOGETHER AND SEPARATELY, “MATERIALS”) ARE BEING PROVIDED “AS IS.” NVIDIA MAKES NO WARRANTIES, EXPRESSED, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE MATERIALS, AND EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES OF NONINFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE. TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL NVIDIA BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF NVIDIA HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. Notwithstanding any damages that customer might incur for any reason whatsoever, NVIDIA’s aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the Terms of Sale for the product.

OpenCL

OpenCL is a trademark of Apple Inc. used under license to the Khronos Group Inc.

Trademarks

NVIDIA, the NVIDIA logo, DGX, DGX-1, DGX-2, DGX-A100, Tesla, and Quadro are trademarks and/or registered trademarks of NVIDIA Corporation in the United States and other countries. Other company and product names may be trademarks of the respective companies with which they are associated.

Copyright

NVIDIA Corporation | 2788 San Tomas Expressway, Santa Clara, CA 95051
www.nvidia.com