cuFile API

API Reference
# Table of Contents

Chapter 1. Introduction........................................................................................................ 1

Chapter 2. Usage.................................................................................................................. 2
  2.1. Dynamic Interactions................................................................................................................ 2
  2.2. Driver, File, and Buffer Management...................................................................................... 3
  2.3. cuFile Compatibility Mode........................................................................................................ 4

Chapter 3. cuFile API Specification......................................................................................6
  3.1. Data Types.................................................................................................................................6
    3.1.1. Declarations and Definitions............................................................................................. 6
    3.1.2. Typedefs.............................................................................................................................. 8
    3.1.3. Enumerations......................................................................................................................8
  3.2. cuFile Driver APIs................................................................................................................... 10
  3.3. cuFile IO APIs..........................................................................................................................11
  3.4. cuFile File Handle APIs.......................................................................................................... 11
  3.5. cuFile Buffer APIs................................................................................................................... 12
  3.6. Future cuFile File Stream APIs............................................................................................. 12
  3.7. Future cuFile Batch APIs....................................................................................................... 12

Chapter 4. cuFile API Functional Specification ................................................................. 14
  4.1. cuFileDriver API Functional Specification............................................................................. 14
    4.1.1. cuFileDriverOpen.............................................................................................................. 14
    4.1.2. cuFileDriverClose............................................................................................................. 15
    4.1.3. cuFileDriverGetProperties................................................................................................16
    4.1.4. cuFileDriverSetPollMode(bool poll, size_t poll_threshold_size)....................................17
    4.1.5. cuFileDriverSetMaxDirectIOSize(size_t max_direct_io_size)......................................... 18
    4.1.6. cuFileDriverSetMaxCacheSize(size_t max_cache_size)................................................. 19
    4.1.7. cuFileDriverSetMaxPinnedMemSize(size_t max_pinned_memory_size).......................20
    4.1.8. JSON Configuration File .................................................................................................. 21
  4.2. cuFile IO API Functional Specification ...................................................................................23
    4.2.1. cuFileHandleRegister....................................................................................................... 24
    4.2.2. cuFileHandleDeregister................................................................................................... 25
    4.2.3. cuFileRead........................................................................................................................ 26
    4.2.4. cuFileWrite........................................................................................................................ 27
  4.3. cuFile Memory Management Functional Specification .........................................................28
    4.3.1. cuFileBufRegister............................................................................................................. 28
    4.3.2. cuFileBufDeregister......................................................................................................... 29
  4.4. cuFile Stream API Functional Specification.......................................................................... 30
Chapter 1. Introduction

NVIDIA® GPUDirect® Storage (GDS) is the newest addition to 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.

All APIs are issued from the CPU, not the GPU.

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{Open, 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 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, a memory buffer is registered the first time the buffer is used, for example, in cuFile{Read, Write}.

- cuFile{BufDeregister, HandleDeregister} explicitly frees a buffer and file resources.
  
  If this API is not called, the buffer and resources are implicitly freed when the driver is closed.
cuFileDriverClose explicitly frees driver resources. If this API is not called, the driver resources are implicitly freed when the process is terminated.

If cuFile\{Open, HandleRegister, BufRegister\} is called on a driver, file, or buffer, respectively that has been opened or registered by a previous cuFile* API call, results 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 also 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.

2.2. Driver, File, and Buffer Management

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

1. **Optional:** Call cuFileDriverOpen() to initialize the state of the critical performance path.
2. **Required:** Allocate GPU memory with cudaMalloc.
3. **Optional:** To register the buffer, call cuFileBufRegister to initialize the buffer state of the critical performance path.
4. Complete the following IO workflow:
   a. **Required:** For Linux, open a file with POSIX open.
   b. **Required:** Call cuFileHandleRegister to wrap an existing file descriptor in an OS-agnostic cuFileHandle. 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. **Required:** Call cuFileRead/cuFileWrite on an existing cuFile handle and existing buffer.
      ▶ If the cuFileBufRegister has not been previously called, the first time that cuFileRead/cuFileWrite is accessed, the GDS library performs a validation check on the GPU buffer and an IO is issued.
      ▶ 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. **Optional:** Call cuFileBufDeregister to free the buffer-specific cuFile state.
6. **Optional:** Call cuFileHandleDeregister to free the file-specific cuFile state.
7. **Optional:** Call cuFileDriverClose to free up the cuFile state.

**Note:** Not using the Deregister and Close APIs (steps 5, 6, and 7) might unnecessarily consume resources, as shown by tools such as valgrind. The best practice is to always use these APIs.
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 and pwrite APIs respectively to system memory and copying to GPU memory. The behavior of compatibility mode with cuFile APIs is determined by the following configuration parameters.

<table>
<thead>
<tr>
<th>Configuration Option (default)</th>
<th>cuFile I/O Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;allow_compat_mode&quot;: true</td>
<td>If true, falls back to using compatibility mode when the library detects that the buffer file descriptor opened cannot use GPUDirect Storage.</td>
</tr>
<tr>
<td>&quot;GPUDirect Storage_rdma_write_support&quot;: true</td>
<td>If false, forces compatibility mode to be used for writes even when the underlying file system is capable of performing GPUDirect Storage writes.</td>
</tr>
<tr>
<td>&quot;posix_unaligned_writes&quot;: false</td>
<td>If true, forces compatibility mode to be used for writes where the file offset and/or I/O size is not aligned to Page Boundary (4KB).</td>
</tr>
<tr>
<td>&quot;lustre:posix_GPUDirect Storage_min_kb&quot;: 0</td>
<td>If greater than 0, compatibility mode is used for I/O sizes between [1 - posix_GPUDirect Storage_min_kb] specified in kB. Note: This option will force posix mode even if &quot;allow_compat_mode&quot; is set to &quot;false&quot;.</td>
</tr>
<tr>
<td>&quot;weka:rdma_write_support&quot;: false</td>
<td>If this option is false, WekaFs will use compatibility mode for all writes to the filesystem. Note: if the option is set to &quot;false&quot;, cuFile library will use the posix path even if the allow_compat_mode option is true or false.</td>
</tr>
</tbody>
</table>
**Usage**

<table>
<thead>
<tr>
<th>Configuration Option (default)</th>
<th>cuFile IO Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;rdma_dynamic_routing&quot;: false, &quot;rdma_dynamic_routing_order&quot;: [ &quot;SYS_MEM&quot; ]</td>
<td>If <code>rdma_dynamic_routing</code> is set to true and <code>rdma_dynamic_routing_order</code> is set to [&quot;SYS_MEM&quot;], then all IO for DFS will use compatibility mode.</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Use Case</th>
<th>cuFile IO Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM Spectrum Scale File System writes.</td>
<td><strong>Note:</strong> All IBM Spectrum Scale in GPUDirect Storage 1.0.0.x release use posix path for writes even if the <code>allow_compat_mode</code> option is set to false.</td>
</tr>
<tr>
<td>No BAR1 memory in GPU.</td>
<td>Use compatibility mode.</td>
</tr>
<tr>
<td>For wekaFS or IBM Spectrum Scale mounts: If there are no <code>rdma_dev_addr_list</code> specified, or failure to register MR with ib device.</td>
<td>Use compatibility mode.</td>
</tr>
<tr>
<td>Bounce buffers cannot be allocated in GPU memory.</td>
<td>Use compatibility mode.</td>
</tr>
<tr>
<td>For WekaFS and IBM Spectrum Scale: If the kernel returns -ENOTSUP for GPUDirect Storage read/write.</td>
<td>Retry the IO operation internally using compatibility mode.</td>
</tr>
<tr>
<td>The <code>nvidia_fs.ko</code> driver is not loaded.</td>
<td>All IO operations will use compatibility mode.</td>
</tr>
</tbody>
</table>

**Limitations**

- Compatible mode does not work in cases where the GPUs have CUDA compute capability less than 6.
- Compatible mode is supported only on GPUDirect Storage enabled filesystems.
- There is no option to force all IO to use compatibility mode. The user can unload the `nvidia_fs.ko` or not expose the character devices in the docker container environment.
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

Data types are used by the cuFile APIs first, the typedefs second, and finally, the enumerations.

3.1.1. Declarations and Definitions

Here are the relevant 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 */
}
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;

eenum CUfileDriverStatusFlags {
    CU_FILE_LUSTRE_SUPPORTED = 0,
    CU_FILE_WEKAFS_SUPPORTED = 1
};

eenum 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;

/* cuFileDescr_t;*/
	/* cuFile handle type */
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;

eenum CUfileDriverStatusFlags {
    CU_FILE_LUSTRE_SUPPORTED = 0,
    CU_FILE_WEKAFS_SUPPORTED = 1
};

eenum 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;

/* cuFileDescr_t;*/
	/* cuFile handle type */
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;

eenum CUfileDriverStatusFlags {
    CU_FILE_LUSTRE_SUPPORTED = 0,
    CU_FILE_WEKAFS_SUPPORTED = 1
};

eenum 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;

/* cuFileDescr_t;*/
	/* cuFile handle type */
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;
/* Status of Batch IO operation */
enum CUfileBatchIOStatus {
    CU_FILE_WAITING,  /* required value prior to submission */
    CU_FILE_PENDING,  /* once enqueued */
    CU_FILE_INVALID,  /* request was ill-formed or could not be enqueued */
    CU_FILE_CANCELED, /* request successfully canceled */
    CU_FILE_COMPLETE, /* request successfully completed */
    CU_FILE_TIMEOUT,  /* request timed out */
    CU_FILE_FAILED    /* unable to complete */
};

struct CUfileBatchIOParams {
    CUFileHandle_t        cfh;         /* File descriptor */
    off_t                 file_offset; /* File offset in bytes */
    volatile void*        target_buf;  /* Location of buffer in memory */
    size_t                nbytes;      /* length of the buffer */
    size_t                nbytes_transacted;  /* what was actually moved */
    CufileBatchOpcode_t   opcode;      /* op code to perform */
    CUfileStatus_t        status;      /* status of the operation */
    CUfileError_t         error;       /* operation level error */
};

3.1.2. Typedefs
cuFile typedefs:

typedef struct CUfileDescr CUfileDescr_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 CUfileBatchOpcode_enum
  This is the cuFile operation code for batch mode.

<table>
<thead>
<tr>
<th>OpCode</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CU_FILE_READ</td>
<td>0</td>
<td>Batch Read</td>
</tr>
<tr>
<td>CU_FILE_WRITE</td>
<td>1</td>
<td>Batch Write</td>
</tr>
</tbody>
</table>

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

- enum CUfileStatus
  The cuFile Status codes for batch mode.

<table>
<thead>
<tr>
<th>Status</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CU_FILE_WAITING</td>
<td>0</td>
<td>The initial value.</td>
</tr>
</tbody>
</table>
### cuFile API Specification

<table>
<thead>
<tr>
<th>Status</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CU_FILE_PENDING</td>
<td>1</td>
<td>Set once enqueued into the driver.</td>
</tr>
<tr>
<td>CU_FILE_INVALID</td>
<td>2</td>
<td>Invalid parameters.</td>
</tr>
<tr>
<td>CU_FILE_COMPLETE</td>
<td>3</td>
<td>Successfully completed.</td>
</tr>
<tr>
<td>CU_FILE_TIMEOUT</td>
<td>4</td>
<td>The operation has timed out.</td>
</tr>
<tr>
<td>CU_FILE_FAILED</td>
<td>5</td>
<td>IO has failed.</td>
</tr>
</tbody>
</table>

- **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.

  We selected a base number for error codes that enables users to distinguish between POSIX errors and cuFile errors.

```
#define CUFILEOP_BASE_ERR 5000
```

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CU_FILE_SUCCESS</td>
<td>0</td>
<td>The cufile is successful.</td>
</tr>
<tr>
<td>CU_FILE_DRIVER_NOT_INITIALIZED</td>
<td>5001</td>
<td>The nvidia-fs driver is not loaded.</td>
</tr>
<tr>
<td>CU_FILE_DRIVER_INVALID_PROPS</td>
<td>5002</td>
<td>An invalid property.</td>
</tr>
<tr>
<td>CU_FILE_DRIVER_UNSUPPORTED_LIMIT</td>
<td>5003</td>
<td>A property range error.</td>
</tr>
<tr>
<td>CU_FILE_DRIVER_VERSION_MISMATCH</td>
<td>5004</td>
<td>An nvidia-fs driver version mismatch.</td>
</tr>
<tr>
<td>CU_FILE_DRIVER_VERSION_READ_ERROR</td>
<td>5005</td>
<td>An nvidia-fs driver version read error.</td>
</tr>
<tr>
<td>CU_FILE_DRIVER_CLOSING</td>
<td>5006</td>
<td>Driver shutdown in progress.</td>
</tr>
<tr>
<td>CU_FILE_PLATFORM_NOT_SUPPORTED</td>
<td>5007</td>
<td>GDS is not supported on the current platform.</td>
</tr>
<tr>
<td>CU_FILE_IO_NOT_SUPPORTED</td>
<td>5008</td>
<td>GDS is not supported on the current file.</td>
</tr>
<tr>
<td>CU_FILE_DEVICE_NOT_SUPPORTED</td>
<td>5009</td>
<td>GDS is not supported on the current GPU.</td>
</tr>
<tr>
<td>CU_FILE_NVFS_DRIVER_ERROR</td>
<td>5010</td>
<td>An nvidia-fs driver ioctl error.</td>
</tr>
<tr>
<td>CU_FILE_CUDA_DRIVER_ERROR</td>
<td>5011</td>
<td>A CUDA Driver API error.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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.</td>
</tr>
<tr>
<td>CU_FILE_CUDA_POINTER_INVALID</td>
<td>5012</td>
<td>An invalid device pointer.</td>
</tr>
</tbody>
</table>
### Error Code

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CU_FILE_CUDA_MEMORY_TYPE_INVALID</td>
<td>5013</td>
<td>An invalid pointer memory type.</td>
</tr>
<tr>
<td>CU_FILE_CUDA_POINTER_RANGE_ERROR</td>
<td>5014</td>
<td>The pointer range exceeds the allocated address range.</td>
</tr>
<tr>
<td>CU_FILE_CUDA_CONTEXT_MISMATCH</td>
<td>5015</td>
<td>A CUDA context mismatch.</td>
</tr>
<tr>
<td>CU_FILE_INVALID_MAPPING_SIZE</td>
<td>5016</td>
<td>Access beyond the maximum pinned memory size.</td>
</tr>
<tr>
<td>CU_FILE_INVALID_MAPPING_RANGE</td>
<td>5017</td>
<td>Access beyond the mapped size.</td>
</tr>
<tr>
<td>CU_FILE_INVALID_FILE_TYPE</td>
<td>5018</td>
<td>An unsupported file type.</td>
</tr>
<tr>
<td>CU_FILE_INVALID_FILE_OPEN_FLAG</td>
<td>5019</td>
<td>Unsupported file open flags.</td>
</tr>
<tr>
<td>CU_FILE_DIO_NOT_SET</td>
<td>5020</td>
<td>The fd direct IO is not set.</td>
</tr>
<tr>
<td>CU_FILE_INVALID_VALUE</td>
<td>5022</td>
<td>Invalid API arguments.</td>
</tr>
<tr>
<td>CU_FILE_MEMORY_ALREADY_REGISTERED</td>
<td>5023</td>
<td>Device pointer is already registered.</td>
</tr>
<tr>
<td>CU_FILE_MEMORY_NOT_REGISTERED</td>
<td>5024</td>
<td>A device pointer lookup failure has occurred.</td>
</tr>
<tr>
<td>CU_FILE_PERMISSION_DENIED</td>
<td>5025</td>
<td>A driver or file access error.</td>
</tr>
<tr>
<td>CU_FILE_DRIVER_ALREADY_OPEN</td>
<td>5026</td>
<td>The driver is already open.</td>
</tr>
<tr>
<td>CU_FILE_HANDLE_NOT_REGISTERED</td>
<td>5027</td>
<td>The file descriptor is not registered.</td>
</tr>
<tr>
<td>CU_FILE_HANDLE_ALREADY_REGISTERED</td>
<td>5028</td>
<td>The file descriptor is already registered.</td>
</tr>
<tr>
<td>CU_FILE_DEVICE_NOT_FOUND</td>
<td>5029</td>
<td>The GPU device cannot be not found.</td>
</tr>
<tr>
<td>CU_FILE_INTERNAL_ERROR</td>
<td>5030</td>
<td>An internal error has occurred.</td>
</tr>
<tr>
<td>CU_FILE_NEWFD_FAILED</td>
<td>5031</td>
<td>Failed to obtain new file descriptor.</td>
</tr>
<tr>
<td>CU_FILE_NVFS_SETUP_ERROR</td>
<td>5033</td>
<td>An NVFS driver initialization error has occurred.</td>
</tr>
<tr>
<td>CU_FILE_IO_DISABLED</td>
<td>5034</td>
<td>GDS is disabled by config on the current file.</td>
</tr>
</tbody>
</table>

**Note:** Data path errors are captured via standard error codes by using errno. The long-term expectation is that these error codes will be folded into CUresult, and CUfileOpError will go away.

### 3.2. cuFile Driver APIs

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

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

3.3. cuFile IO APIs

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

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

The buffer on the device has both a base (`devPtr_base`) and offset (`devPtr_offset`). This offset is distinct from the offset in the file.

Note that by default for all paths where GDS is not supported, the cuFile IO API will be attempting 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`.

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`.

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

3.6. Future cuFile File 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. cuFile Stream APIs require special enabling with the NVreg_EnableStreamMemOPs=1 modprobe.

Note: Support for these APIs might be staged over time.

There are two flavors of these two APIs, for runtime APIs and types (cudaStream_t) and those for the driver (CUstream). We anticipate that the runtime APIs will be integrated into cuda.h, and the driver APIs will be integrated into cuda_runtime.h.

CUfileError_t cuFileReadAsync(CUFileHandle_t fh, void *devPtr_base, size_t *size, off_t *file_offset, off_t *devPtr_offset, ssize_t *bytes_read, CUStream stream);
CUfileError_t cuFileWriteAsync(CUFileHandle_t fh, void *devPtr_base, size_t *size, off_t *file_offset, off_t *devPtr_offset, ssize_t *bytes_written, CUStream stream);

3.7. Future cuFile Batch APIs

cuFile Batch APIs enable an arbitrary mix of read and write transactions, multiple files, and multiple locations in a file in one dispatch. Operations 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.

Individual operations in each batch might be reordered with respect to each other.

**Note:** The batch APIs are considered experimental and might take a different form later or be absorbed into other approaches such as CUDA Graphs.

Support for these APIs might be staged over time. There are two flavors of these two APIs: runtime APIs and types (`cudaStream_t`) and APIs for the driver (`CUstream`). We anticipate that the runtime APIs will be integrated into `cuda.h`, while the driver APIs will be integrated into `cuda_runtime.h`.

```c
CUfileError_t cudaFileBatchIOSubmit(int *batch_idp, int nr, CUfileIOParams_t *iocbp,
                                    unsigned int flags, cudaStream_t stream);
CUfileError_t cuFileBatchIOSubmit(int *batch_idp, int nr, CUfileIOParams_t *iocbp,
                                 unsigned int flags, CUstream stream);
CUfileError_t cuFileBatchIOGetStatus(int batch_idp, int *nr, CUfileIOParams_t *iocbp);
CUfileError_t cuFileBatchIOCancel(int batch_idp);
void cuFileBatchIODestroy(int batch_idp);
```
Chapter 4. cuFile API Functional Specification

This section provides information about the cuFile API functional specification. See the GPU Direct 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 CPU.

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

**cuFileDriverClose API**

```c
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 driver close implicitly unregisters those buffers. Any in-flight IO when `cuFileDriverClose` is in-progress will receive an error.
4.1.3. cuFileDriverGetProperties

cuFileDriverGetProperties API.

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

Support for NVMe, NVMe-OF, and SCSI are experimental.

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)

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)

The following is information about the cuFileDriverSetMaxDirectIOSize(size_t max_direct_io_size) API.

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)

The following is information about the cuFileDriverSetMaxCacheSize(size_t max_cache_size) API.

CUFilenError_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)

Here is some information about the cuFileDriverSetMaxPinnedMemSize(size_t max_pinned_memory_size) API.

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. See cuFileDriverGetProperties for more information.
4.1.8. JSON Configuration File

This section provides the schema for the cufile.json configuration file. The values for each parameter are default values, if the parameters are not listed in the file. APIs are available that correspond to a subset of these parameters. When they are invoked, they override the cufile.json parameter values for the process from which they are called.

The following table lists the usage of different configuration parameters:

<table>
<thead>
<tr>
<th>Configuration Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>logging:dir</td>
<td>The log directory for the cufile.log file. If the log directory has not been enabled, the log file is created under the current working directory. The default value is currently the working directory.</td>
</tr>
</tbody>
</table>
| logging:level           | The level indicates the type of messages that will be logged.  
  - ERROR indicates log critical errors only.  
  - DEBUG indicates the log information that includes error, informational, and debugging the library.  
  The default value is set to ERROR. |
<p>| properties:max_direct_io_size_kb | This parameter indicates the maximum unit of IO size that is exchanged between the cufile library and the storage system. The default value is set to 16MB. |
| properties:max_device_cache_size_kb | This parameter indicates the maximum per GPU memory-size in KB that can be reserved for internal bounce buffers. The default value is set to 128MB. |
| properties:max_device_pinned_mem_size_kb | This parameter indicates the maximum per GPU memory-size in KB that can be pinned including the memory for internal bounce buffers. The default value is set to 32GB. |
| properties:use_poll_mode | Boolean that indicates whether the cufile library uses polling or synchronous wait for the storage to complete IO. Polling might be useful for small IO transactions. The default value is false. |
| properties:poll_mode_max_size_kb | The maximum IO size in KB used as the threshold if polling mode is set to true. |
| properties:allow_compat_mode | If this option is set to true, cuFile APIs work functionally with the nvidia-fs driver. The purpose is to test newer file systems for environments |</p>
<table>
<thead>
<tr>
<th>Configuration Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>properties:rdma_dev_addr_list</td>
<td>Where GDS applications do not have the kernel driver installed, or for comparison tests.</td>
</tr>
<tr>
<td>fs:generic:posix_unaligned_writes</td>
<td>This parameter list provides the list of IPv4 addresses for all the interfaces that can be used for RDMA.</td>
</tr>
<tr>
<td>fs:lustre:posix_gds_min_kb</td>
<td>This option is applicable only for the EXAScaler filesystem. It provides an option to fallback to the POSIX compatible mode for IO sizes that are smaller than or equal to the set value. This is applicable for reads and writes.</td>
</tr>
<tr>
<td>denylist:drivers</td>
<td>An administrative setting to disable specific supported storage drivers on the node.</td>
</tr>
<tr>
<td>denylist:devices</td>
<td>An administrative setting to disable specific supported block devices on the node. Not applicable for DFS.</td>
</tr>
<tr>
<td>denylist:mounts</td>
<td>An administrative setting to disable specific mounts in the supported GDS enabled file systems on the node.</td>
</tr>
<tr>
<td>denylist:filesystems</td>
<td>An administrative setting to disable specific supported GDS-ready file systems on the node.</td>
</tr>
<tr>
<td>CUFILE_ENV_PATH_JSON</td>
<td>An environment variable to change the default path of /etc/cufile.json for a specific application instance to use different settings for the application and further restrict using the blacklist option if the application is not ready for that filesystem or the mount paths.</td>
</tr>
</tbody>
</table>

The following is the JSON schema:

```json
# /etc/cufile.json
{
  "logging": {
"dir": "/home/<xxxx>",
"level": "ERROR"
  },

  "profile": {
"nvtx": false,
"cufile_stats": 0
  },

  "properties": {
"max_direct_io_size_kb": 16384,
"max_direct_io_size_kb": 16384,
"device_memory_size (4K aligned) for reserving bounce buffers"
```
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

The following is information about the cuFileHandleRegister API.

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.

**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 like no O_DIRECT, without compat mode enabled, O_APPEND, O_NOCPTY, 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 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 expects the file descriptor to be opened with O_DIRECT mode. But if compatibility mode is enabled, then this requirement is relaxed.
It is recognized that in order to be fully compatible, `cuFileHandleRegister` should not limit the set of flags that are supported, relative to a POSIX `pread` or `pwrite`. But those conditions are not fully tested. Currently checks for `O_DIRECT` and GDS supported file system-specific checks are relaxed. We anticipate additional relaxation on restrictions over time.

Refer to the following for more information:
- `cuFileRead`
- `cuFileWrite`
- `cuFileReadAsync`
- `cuFileWriteAsync`
- `cuFileHandleDeregister`

### 4.2.2. `cuFileHandleDeregister`

Here is some information about the `cuFileHandleDeregister` API.

```c
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`
4.2.3. cuFileRead

The following is information about the cuFileRead API.

```c
ssize_t cuFileRead(CUFileHandle fh, void *devPtr_base, size_t size, off_t file_offset, off_t devPtr_offset);
```

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

**Parameters**

- **fh**
  File descriptor for the file.

- **devPtr_base**
  Base address of buffer in device memory. For registered buffers, `devPtr_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.

- **devPtr_offset**
  Offset relative to the `devPtr_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. 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 `devPtr_offset`, if data will be read starting exactly from the `devPtr_base` that is registered with `cuFileBufRegister`, `devPtr_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 `devPtr_offset`, and the `devPtr_base` must remain set to the base address that was used in the `cuFileBufRegister` call.

See the following for more information:
4.2.4. cuFileWrite

The following is information about the cuFileWrite API.

```
ssize_t cuFileWrite(CUFileHandle fh, const void *devPtr_base, size_t size, off_t file_offset, off_t devPtr_offset);
```

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

**Parameters**

- `fh`
  File descriptor for the file

- `devPtr_base`
  Base address of buffer in device memory. For registered buffers, `devPtr_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.

- `devPtr_offset`
  Offset relative to the `devPtr_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 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 `devPtr_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

The following is information about the `cuFileBufRegister` API.

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

- Registers existing `cuMemAlloc’d [pinned] memory for GDS IO operations.

**Parameters**

- `devPtr_base`
  
  Address of device pointer. `cuFileRead` and `cuFileWrite` **must** use this `devPtr_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.

Description
This API registers the specified GPU address and size for use with the cuFileRead and cuFileWrite operations. The user must call cuFileBufDeregister to release the pinned memory mappings.

See the following for more information:

4.3.2. cuFileBufDeregister
The following is information about the cuFileBufDeregister API.

CUfileError_t cuFileBufDeregister(const void *devPtr_base);

Deregisters CUDA memory registered using the cuFileBufRegister API.

Parameters

devPtr_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 devPtr_base was not registered.

CU_FILE_ERROR_INVALID_VALUE on failure to find the registration for the specified device 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. cuFileReadAsync

The following is information about the cuFileReadAsync API.

```c
CUfileError_t cudaFileReadAsync(CUFileHandle_t fh, void *devPtr_base, size_t *size, off_t file_offset, off_t devPtr_offset, int *bytes_read, cudaStream_t stream);
CUfileError_t cuFileReadAsync(CUFileHandle_t fh, void *devPtr_base, size_t *size, off_t file_offset, off_t devPtr_offset, int *bytes_read, CUstream stream);
```

- Enqueues a read operation for the specified bytes from the cuFile handle into the device memory by using GDS functionality.
- 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.
- `devPtr_base`
  - The base address of the buffer in the device memory into which to read.
  - For registered buffers, `devPtr_base` must remain set to the base address used in `cuFileBufRegister` call.
- `size`
  Size in bytes to read.
- `file_offset`
  Offset in the file from which to read.
- `devPtr_offset`
  The offset relative to the `devPtr_base` pointer from which to write.
- `bytes_read`
  - 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 a 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 memory should be allocated with cuMemHostAlloc 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.

Refer to the following for more information:
- cuFileRead
- cuFileWrite
- cuFileWriteAsync

4.4.2. cuFileWriteAsync

The following is information about the cuFileWriteAsync API.

```c
CUfileError_t cudaFileWriteAsync(CUFileHandle_t fh, void *devPtr_base,
                                 size_t *size, off_t file_offset,
                                 int *bytes_written, cudaStream_t stream);
```

```c
CUfileError_t cuFileWriteAsync(CUFileHandle_t fh, void *devPtr_base,
```

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.
- **devPtr_base**
  The base address of the buffer in the device memory from which to write. For registered buffers, **devPtr_base** must remain set to the base address used in the cuFileBufRegister call.
- **size**
  Size in bytes to write.
- **file_offset**
  Offset in the file from which to write.
- **devPtr_offset**
  Offset relative to the **devPtr_base** pointer from which to write.
- **bytes_written**
  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 a 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.

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.

Batch APIs can perform a set of IO operations, and these operations can be completed on 
different files, different locations in the same file, or a mix. The parameter with the array of 
CUfileIOParams_t describes the IO action, status, errors, and bytes transacted for each 
instance. The bytes transacted field is valid only when the status indicates a completion. The 
completion description includes a stream and a NULL value that indicates that the batch 
should occur without stream ordering.

4.5.1. cuFileBatchIOSubmit

Here is some information about the cuFileBatchIOSubmit API.

```
CUfileError_t cudaFileBatchIOSubmit(int *batch_idp, int nr,  
    CUfileIOParams_t *iocbp,  
    unsigned int flags,  
    cudaStream_t stream)
```

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

**Parameters**

- batch_idp
  The address of the output parameter for the newly created batch ID, which is a -1 on an 
  error.

- nr
  The number of requests in the params array.
The value must be greater than 0.

**iocbp**
The pointer to contain the CUfileIOParams_t array structures of the len nr array.

**flags**
Reserved as 0.

**stream**
  - The CUDA stream to enqueue the operation.
  - Both CUDA runtime and CUDA driver variations are offered.

**Returns**
- **CU_FILE_SUCCESS** on success.
  - *batch_idp* will be a value greater than 0.
- **CU_FILE_MEM_ALLOC_FAILED** on memory allocation failures.
- **CU_FILE_DRIVER_ERROR** if the nvidia-fs driver cannot handle the request.
- **CU_FILE_ERROR_INVALID_VALUE** on a failure to correctly set the arguments. Individual status can be checked to determine whether arguments are invalid for one or more IO operation elements.

**Description**
- This is a batch API to submit a read/write operation on an array of GPU data pointers from their respective file handle, offset, and size bytes. The data is transferred into the GPU memory by using GDS.
  - This is an asynchronous call and will enqueue the operation and provide a batch_id. The operation can be monitored when using this batch_id through cuFileBatchIOGetStatus.
  - The operation might be canceled by calling cuFileBatchIOCcancel or cuFileBatchIODestroy.
  - The entries in the CUfileIOParams_t array describe the IO action and report the status, errors, and bytes that were transacted for that operation. 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 another.
  - 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**
4.5.2. cuFileBatchIOGetStatus

The following is information about the cuFileBatchIOGetStatus API.

```c
CUfileError_t cuFileBatchIOGetStatus(int batch_idp,
                                       int *nrp,
                                       const CUfileIOParams_t *iocbp)
```

**Parameters**

- **batch_idp**
  The address of the output parameter for the newly created batch ID, which is a -1 on an error.

- **nrp**
  The pointer to the number of events that were returned in the `batch_id`. This will match the `nr` field in `cuFileIOSubmit`.

- **iocbp**
  The `const` pointer that points to an array of `CUfileIOParams_t` with `nrp`-sized array of events. This pointer should not be freed and the `status` field in `CUfileIOParams_t` will be updated to reflect the current status of each IO in the batch operation. To get the current status, refer to the description of the `status` field in `cuFileStatus`. The error field will contain the errors encountered during the batch operations for that IO operation in the batch.

**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 $\text{CUfileIOParams_t}$ entries will have undefined values before the entire batch is completed. This definition is subject to change.

See the following for more information:

- cuFileBatchIOSubmit
- cuFileBatchIOCancel

### 4.5.3. cuFileBatchIOCancel

The following is information about the `cuFileBatchIOCancel` API.

```c
CUfileError_t cuFileBatchIOCancel(int batch_idp)
```

**Parameters**

- `batch_idp`  
  The batch ID to cancel.

**Returns**

- `CU_FILE_SUCCESS` on success.
- `CU_FILE_ERROR_INVALID_VALUE` for an invalid batch ID.

**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. The operation will wait for a specific time based on the `props.max_batch_io_timeout_msecs` value to cancel any ongoing operation or will set the status to `CU_FILE_TIMEOUT`. The operation will be canceled automatically if `cuFileBatchIODestroy` is called.

Refer to the following for more information:

- cuFileBatchIOGetStatus
- cuFileBatchIOSubmit
- cuFileBatchIODestroy

### 4.5.4. cuFileBatchIODestroy

The following is information about the `cuFileBatchIODestroy` API.

```c
void cuFileBatchIODestroy(int batch_idp)
```

**Parameters**

- `batch_idp`
The address of the output parameter for the newly created batch id, -1 on error.

**Returns**

void

**Description**

This is a batch API that cancels and destroys a batch context and the resources that are allocated with `cuFileBatchIOSubmit`. Refer to the following for more information:

- [cuFileBatchIOGetStatus](#)
- [cuFileBatchIOSubmit](#)
- [cuFileBatchIOCcancel](#)
Chapter 5. Sample Program with cuFile APIs

The following sample program uses the cuFile APIs:

```c
#include <fcntl.h>
#include <errno.h>
#include <unistd.h>
#include <cstdlib>
#include <cstring>
#include <iostream>
#include <cuda_runtime.h>
#include "cufile.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;
    size_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;
}

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);

// 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;
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.

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

Trademarks
NVIDIA and the NVIDIA logo are trademarks or registered trademarks of NVIDIA Corporation in the U.S. and other countries. Other company and product names may be trademarks of the respective companies with which they are associated.

Copyright
© 2020-2021 NVIDIA Corporation & affiliates. All rights reserved.