NVIDIA Magnum IO GPUDirect Storage

Best Practices Guide

DA-10088-001_v1.0.0 | July 2021

Table of Contents

Chapter 1. Introduction 1

Chapter 2. Software Settings 2

2.1. System Settings 2

2.2. cuFile Configuration Settings 3

Chapter 3. API Usage 4

3.1. cuFileDriverOpen 4

3.2. cuFileHandleRegister 4

3.3. cuFileBufRegister, cuFileRead, and cuFileWrite 5

3.3.1. IO Pattern 1 6

3.3.2. IO Pattern 2 7

3.3.3. IO Pattern 3 8

3.3.4. IO Pattern 4 9

3.3.5. IO Pattern 5 10

3.3.6. IO Pattern 6 11

3.4. cuFileHandleDeregister 13

3.5. cuFileBufDeregister 14

3.5.1. cuFileDriverClose 14

Chapter 1. Introduction

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

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.

GDS is enabled on the following filesystems:

- DDN EXAScaler: https://www.ddn.com/
- WekaFS filesystem: https://www.weka.io
- VAST Data's NFSoRDMA implementation: https://www.vastdata.com

GDS documents and online resources provide additional context for the optimal use of, and understanding of GDS. Refer to the following guides for more information about GDS:

- GPUDirect Storage Design Guide
- GPUDirect Storage Overview Guide
- cuFile API Reference Guide
- GPUDirect Storage Release Notes
- GPUDirect Storage Troubleshooting Guide
- GPUDirect Storage O_DIRECT Requirements Guide

To learn more about GDS, refer to the following blogs:

- GPUDirect Storage: A Direct Path Between Storage and GPU Memory.
- The Magnum IO series.

Chapter 2. Software Settings

This section provides information about the settings required for GDS and the settings that are specific to the filesystem that you are using.

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

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

2.1. System Settings

The following are system settings we recommend for the best performance.

- PCIe Access Control Services (ACS).

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

  For the optimal GDS performance, disable ACS.

> **Note:** To list all of the PCI switches that have ACS enabled, issue `gdschecker -p`.

- IOMMU

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

> **Note:** To determine whether the IOMMU setting is enabled, check the `cat /proc/cmdline` output.

- NIC affinity

  In NVIDIA DGX™-based platforms, complete the following tasks:

  - For the peer-to-peer DMA to function efficiently, provision at least one NIC in the same PCIe switch as the GPU.
  - Avoid configurations where the NICs are assigned across the PCIe switches that require PCIe traffic to cross the CPU root ports or go across CPU sockets that use QPI.

- NIC versions
  - When using Mellanox ConnectX-5 or ConnectX-6 the HCAs must be configured in InfiniBand or RoCE v2 mode.
  - For GDS support, MLNX_OFED 4.6 or later is required.

2.2. cuFile Configuration Settings

This section provides information about the cuFile configuration changes in GDS. The cuFile configuration settings in GDS are stored in the `/etc/cufile.json` file.

To display the configuration setting, run the following command:

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

Here is a portion of the sample output:

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

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

The `max_direct_io_size_kb` parameter can be set to a value that is a multiple of 64K. This process defines the additional system memory that is used for each buffer during `cuFileBufRegister` up to a maximum value that is defined by the `properties:max_direct_io_size_kb` parameter. The maximum direct IO size that GDS can handle is 16MB, and this value can be reduced to 1MB to reduce the amount of system memory that is used per buffer.

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

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

1. Run the following command:

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

2. Review the output:

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

Chapter 3. API Usage

This section provides information about the best practices to remember when you use the GDS APIs.

Note: The cuFile APIs are designed to be 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.

Note: The APIs are not designed to work with the `fork` call.

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

3.1. cuFileDriverOpen

Here is some additional information about the `cuFileDriverOpen` API.

This API should be invoked only once per process and **before** you invoke any other GDS API. The application should call this API to avoid the latency of the driver that will be otherwise incurred in the first IO call.

3.2. cuFileHandleRegister

The following is information about the `cuFileHandleRegister` API.

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

Note: There should be one handle for each file descriptor.

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

Note: In the compatibility mode, an additional `fd` can be opened without requiring the `O_DIRECT` mode. This mode can alos handle unaligned reads/writes, even when POSIX cannot.

3.3. cuFileBufRegister, cuFileRead, and cuFileWrite

The following is information about the `cuFileBufRegister`, `cuFileRead`, and `cuFileWrite` APIs.

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

Note: The process to accomplish this mapping is called registration.

Explicit GPU buffer registration with the cuFileBufRegister API is optional. If the user buffer is not registered, an intermediate pre-registered GPU buffer that is owned by the cuFile implementation is used, and there is an extra copy from there to the user buffer. The following table provides guidance on whether registration is profitable.

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

Use Case	Description	Recommendation
A 4KB-aligned GPU buffer is reused as an intermediate buffer to read or write data by using optimal IO sizes for storage systems in multiples of 4KB.	The GPU buffer is used as an intermediate buffer to stream the contents or to populate a different data structure in GPU memory. You can implement this use case for IO libraries with DSG.	Register this reusable intermediate buffer to avoid the additional internal staging of data by using GPU bounce buffers in the cuFile library. See IO Pattern 2 for the recommended usage.
Filling a large GPU buffer for one use.	The GPU buffer is the final location of the data. Since the buffer will not be reused, the registration cost will not be amortized. A usage example is reading large preformatted checkpoint binary data. Registering a large buffer can have a latency impact when the buffer is registered.	This can also cause BAR memory exhaustion because running multiple threads or applications will compete for BAR memory. Read or write the data without buffer registration. See IO Pattern 3 for the recommended usage.
Partitioning a GPU buffer to be accessed across multiple threads.	The main thread allocates a large chunk of memory and creates multiple threads. Each thread registers a portion of the memory chunk independently and uses that as in IO Pattern 2.	Allocate, register, and deregister the buffers in each thread independently for simple IO workflows. For cases where the GPU memory is preallocated, each

Use Case	Description	Recommendation
	You can also register the entire memory in the parent thread and use this registered buffer with the size and `devPtr_offset` parameters set appropriately with the buffer offsets for each thread. A cudaContext must be established in each thread before registering the GPU buffers.	thread can set the appropriate context and register the buffers independently. See IO Pattern 6 for the recommended usage. After you install the GDS package, see `cufile_sample_016.cc` and `cufile_sample_017.cc` under `/usr/local/CUDA-X.y/samples/` for more details.
GPU offsets, file offsets, and IO request sizes are unaligned.	The IO reads or writes are mostly unaligned. An intermediate aligned buffer might be needed to handle alignment issues with GPU offsets, file offsets, and IO sizes.	**Do not** register the buffer. See IO Pattern 4 and IO Pattern 5.
Working on a GPU with a small BAR space as compared to the available GPU memory.	In some GPU SKUs, the BAR memory is smaller than the total device memory.	To avoid failures because of BAR memory exhaustion, do not register the buffer. See IO Pattern 3.

3.3.1. IO Pattern 1

Here is the code sample for IO Pattern 1.

```
#define MB(x) ((x)*1024*1024L)
  2 #define GB(x) ((x)*1024*1024L*1024L)
  3
  4
  5void thread_func(CUfileHandle_t cuHandle)
  6 {
  7         void *devPtr_base;
  8         int readSize = MB(100);
  9         int devPtr_offset = 0;
 10         int file_offset = 0;
 11         int ret = 0;
 12
 13
 14         cudaSetDevice(0);
 15         cudaMalloc(&devPtr_base, GB(1));
 16
 17         for (int i = 0; i < 10; i++) {
 18
 19              cuFileBufRegister((char *)devPtr_base + devPtr_offset, readSize,
 0);
 20
 21              ret = cuFileRead(cuHandle, (char *)devPtr_base + devPtr_offset,
                                 readSize,  file_offset, 0);
 22
 23
         <... launch cuda kernel using contents at devPtr_base + devPtr_offset … >

 24              file_offset += readSize;
 25              devPtr_offset += readSize;
 26
```

```
27              cuFileBufDeregister((char *)devPtr_base + devPtr_offset);
28          }
29  }
```

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

   a). At line 19, the GPU buffer of 100 MB is registered.

   b). Submit the read for 100MB (readsize is 100 MB).

   c). At line 27, the GPU buffer of 100 MB is deregistered.

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

3.3.2. IO Pattern 2

Here is the code sample for IO Pattern 2.

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

 22             <... launch cuda kernel using contents at devPtr_base +
 devPtr_offset … >
 23
 24                 file_offset += readSize;
 25                 devPtr_offset += readSize;
 27
 28         }
 29        cuFileBufDeregister(devPtr_base);
 30 }
```

3.3.3. IO Pattern 3

Here is the code sample for IO Pattern 3.

```
1 #define MB(x) ((x)*1024*1024L)
  2 #define GB(x) ((x)*1024*1024L*1024L)
  3
  4
  5 void thread_func(CUfileHandle_t cuHandle)
  6{
  7         void *devPtr_base;
  8         int readSize = MB(100);
  9         int devPtr_offset = 0;
 10         int file_offset = 0;
 11         int ret = 0;
 12
 13         cudaSetDevice(0);
 14         cudaMalloc(&devPtr_base, GB(1));
 15
 16         for (int i = 0; i < 10; i++) {
 17
 18              ret = cuFileRead(cuHandle, (char *)devPtr_base,
                                        readSize, file_offset, devPtr_offset);
 19
 20          <... launch cuda kernel using contents at devPtr_base + devPtr_offset …
 >
 21
 22              file_offset += readSize;
 23              devPtr_offset += readSize;
 24         }
 25 }
```

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

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

Note: Although semantically correct, this loop might not be optimal.

Internally, GDS uses GPU bounce buffers to perform IOs. Bounce buffers are GPU memory allocations that are internal to GDS, and these buffers are registered and managed by the GDS library. The number of bounce buffers and size of each bounce buffer is capped based on the `max_device_cache_size` setting in the `/etc/cufile.json` file.

The number of GPU bounce buffers can be tuned using configurable property in the `/etc/cufile.json` file. The `max_device_cache_size` setting states the maximum cache size in KB set per GPU. By default, it is set to 128 MB.

3.3.4. IO Pattern 4

Here is the code sample for IO Pattern 4. This is an unaligned IO on an EXAScaler® Filesystem.

```
1 #define MB(x) ((x)*1024*1024L)
  2 #define GB(x) ((x)*1024*1024L*1024L)
  3
  4
  5 void thread_func(CUfileHandle_t cuHandle)
  6 {
  7         void *devPtr_base;
  8         int readSize = MB(100);
  9         int devPtr_offset = 0;
 10         int file_offset = 3; // Start from odd offset
 11         int ret = 0;
 12
 13
 14        cudaSetDevice(0);
 15         cudaMalloc(&devPtr_base, GB(1));
 16         cuFileBufRegister(devPtr_base, GB(1), 0);
 17
 18         for (int i = 0; i < 10; i++) {
 19                 // IO issued at offsets which are not 4K aligned
 20                 ret = cuFileRead(cuHandle, devPtr_base,
                                    readSize, file_offset, devPtr_offset);
 21                 assert(ret >= 0);
             <... launch cuda kernel using contents at devPtr_base + devPtr_offset …
>
 23
 24                 file_offset += readSize;
 25                 devPtr_offset += readSize;
 27
 28         }
       cuFileBufDeRegister(devPtr_base);
 29 }
```

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

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

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

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

1. After allocating 1 GB of GPU memory, `cuFileBufRegister` is immediately invoked for the entire range of 1 GB as seen on line 16.
2. Fill the entire 1 GB GPU memory by reading 100 MB at a time from file as seen in the following loop:

   a). The initial file_offset is at 3, and reads are submitted with a `readSize` value of 100MB at an offset of 3 for each iteration.

For example, `file_offset` during each read is not 4K aligned.

b). Since `file_offset` is not 4K aligned, the GDS library can internally use GPU bounce buffers to complete the IO.

The GPU bounce buffer mechanism is identical to IO-Pattern-3.

3. Unaligned IOs might not be optimal and should be avoided by reading the size value that is specified in multiples of 4KB and the `file_offsets` value that is specified in multiples of 4KB.

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

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

If the applications cannot issue 4K aligned IO, instead of using the `cuFileBufRegister` API, use the `cuFileRead`/`cuFileWrite` APIs as described in IO-Pattern-2.

Remember the following information:

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

  Note: Read-Modify-Write is not atomic. For more information, see the `cufile_sample_018.cc` sample program in the `/usr/local/CUDA-X.y/samples` directory.

- Applications must ensure that no other thread is reading/writing in this given range.

  If required, range locks (using flock) must be used before submitting IO.

3.3.5. IO Pattern 5

Here is the code sample for IO Pattern 5. This IO is an unaligned IO on an WekaFS filesystem.

```
1 #define MB(x) ((x)*1024*1024L)
  2 #define GB(x) ((x)*1024*1024L*1024L)
  3
  4
  5 void thread_func(CUfileHandle_t cuHandle)
  6 {
  7         void *devPtr_base;
  8         int readSize = MB(100);
  9         int devPtr_offset = 3; // Start from odd offset
 10         int file_offset = 0;
 11         int ret = 0;
 12
 13
 14         cudaSetDevice(0);
 15         cudaMalloc(&devPtr_base, GB(1));
```

```
16          cuFileBufRegister(devPtr_base, GB(1), 0);
17
18          for (int i = 0; i < 10; i++) {
19                  // IO issued at gpu buffer offsets which are not 4K aligned
20                  ret = cuFileRead(cuHandle, devPtr_base,
                                     readSize, file_offset, devPtr_offset);
21                  assert (ret >= 0);
                    <... launch cuda kernel using contents at devPtr_base +
devPtr_offset … >
23
24                  file_offset += readSize;
25                  devPtr_offset += readSize;
27
28          }
        cuFileBufDeRegister(devPtr_base);
29  }
```

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

If the IO is unaligned, the cuFile library will issue IO through the internal GPU bounce buffer cache. Also, if the allocation of internal cache fails, the IO fails. To avoid IO failure in this case, you can set `allow_compat_mode` to `true` in the `/etc/cufile.json` file. With this setting, IO will fallback to the POSIX APIs.

3.3.6. IO Pattern 6

Here is the code sample for IO Pattern 6.

```
typedef struct thread_data
{
   void *devPtr;
    loff_t offset;
    loff_t devPtr_offset;
    CUfileHandle_t cfr_handle;
}thread_data_t;

static void *thread_fn(void *data)
{
    int ret;
    thread_data_t *t = (thread_data_t *)data;

    /*
     *  Threads do not inherit cuda context from the parent. Before submitting reads
 to
     *  cuFileRead/cuFileWrite, threads should have cuda context associated with it.
     *  The context should be set based on the context where devPtr was allocated.
     */

    cudaSetDevice(0);
    cudaCheckError();

    /*
     * Note the usage of devPtr_offset. Every thread has same devPtr handle
     * which was registered using cuFileBufRegister; however all threads are
     * working at different devPtr offsets. This is optimal as GPU memory is
     * registered once in the main thread.
     */
     ret = cuFileRead(t->cfr_handle, t->devPtr, MB(100), t->offset, t-
>devPtr_offset);
     if (ret < 0) {
```

```
        fprintf(stderr, "cuFileRead failed with ret=%d\n", ret);
     }
     <... launch cuda kernel using contents at devPtr + devPtr_offset … >
    return NULL;
}
int main(int argc, char **argv) {
    void *devPtr;
    size_t offset = 0;
    int fd;
    CUfileError_t status;
    CUfileDescr_t cfr_descr;
    CUfileHandle_t cfr_handle;
    thread_data t[10];
    pthread_t thread[10];

    if (argc < 2) {
        fprintf(stderr, "Invalid input.\n");
        help();
        exit(1);
    }
    fd  = open(argv[1], O_RDWR | O_DIRECT);
    assert(fd > 0);

    memset((void *)&cfr_descr, 0, sizeof(CUfileDescr_t));
    cfr_descr.handle.fd = fd;
    cfr_descr.type = CU_FILE_HANDLE_TYPE_OPAQUE_FD;
    status = cuFileHandleRegister(&cfr_handle, &cfr_descr);
    if (status.err != CU_FILE_SUCCESS) {
        printf("file register error: %s\n", CUFILE_ERRSTR(status.err));
        close(fd);
        exit(1);
    }

    cudaSetDevice(0);
    cudaCheckError();

    cudaMalloc(&devPtr, GB(1));
    cudaCheckError();

    /*
     * Entire Memory is registered
     */
    status = cuFileBufRegister(devPtr, GB(1), 0);
    if (status.err != CU_FILE_SUCCESS) {
        printf("Buffer register failed :%s\n", CUFILE_ERRSTR(status.err));
        cuFileHandleDeregister(cfr_handle);
        close(fd);
        exit(1);
    }

    for (int i = 0; i < 10; i++) {
        /*
         * Every thread will get same devPtr address; additionally, every thread
         * will share the same cuFileHandle.
         */
        t[i].devPtr = devPtr;
        t[i].cfr_handle = cfr_handle;

        /*
         * Every thread will work on different devPtr offset
         */
        t[i].offset = offset;
        t[i].devPtr_offset = offset;
        offset += MB(100);
    }

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

```
        pthread_create(&thread[i], NULL, &thread_fn, &t[i]);
    }

    for (int i = 0; i < 10; i++) {
        pthread_join(thread[i], NULL);
    }

    // Deregister once all threads terminate
    status = cuFileBufDeregister(devPtr);
    if (status.err != CU_FILE_SUCCESS) {
        fprintf(stderr, "cuFileBufDeregister failed :%s\n",
 CUFILE_ERRSTR(status.err));
    }

    cuFileHandleDeregister(cfr_handle);
    close(fd);
    cudaFree(devPtr);
    return 0;
}
```

This example demonstrates using `cuFileBufRegister` once in the main thread and how child threads can access the registered buffer at a different GPU buffer offset.

The main thread completes the following tasks:

- Allocates GPU Memory of size 1GB.
- Registers the entire memory by using `cuFileBufRegister`.
- Creates a cuFileHandle for the opened file descriptor.
- Spawns, where each thread completes the following tasks:
  - Works on the same `cuFileHandle`.
  - Sets the CUDA context that is relevant to the `devPtr` context.
  - Submits reads to cuFileRead at different `devPtr_offset`.

This process ensures that the buffer is registered once in the main thread, and individual threads can focus on IO.

3.4. cuFileHandleDeregister

The following is additional information about the `cuFileHandleDeregister` API.

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

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

3.5. cuFileBufDeregister

The following is information about the `cuFileBufDeregister` API.

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

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

3.5.1. cuFileDriverClose

Here is some information about the `cuFileDriverClose` API.

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

This API should always be invoked at the end of the application, or when the application no longer needs to complete IO using GDS.

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.

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

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

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