CUDA Toolkit 5.0
CUFFT Library

PG-05327-050_v01 | April 2012

Programming Guide

Contents

Published by
NVIDIA Corporation
2701 San Tomas Expressway
Santa Clara, CA 95050

Notice

Trademarks

Copyright

Chapter 1
Introduction

This document describes CUFFT, the NVIDIA® CUDA™ Fast Fourier Transform (FFT) library. The FFT is a divide-and-conquer algorithm for efficiently computing discrete Fourier transforms of complex or real-valued data sets. It is one of the most important and widely used numerical algorithms in computational physics and general signal processing. The CUFFT library provides a simple interface for computing parallel FFTs on an NVIDIA GPU, which allows users to leverage the floating-point power and parallelism of the GPU without having to develop a custom, CUDA FFT implementation.

FFT libraries typically vary in terms of supported transform sizes and data types. For example, some libraries only implement radix-2 FFTs, restricting the transform size to a power of two. The CUFFT Library aims to support a wide range of FFT options efficiently on NVIDIA GPUs. This version of the CUFFT library supports the following features:

- Complex and real-valued input and output
- 1D, 2D, and 3D transforms
- Batch execution for doing multiple transforms of any dimension in parallel
- Transform sizes up to 64 million elements in single precision and up to 128 million elements in double precision in any dimension, limited by the available GPU memory
- In-place and out-of-place transforms
- Double-precision (64-bit floating point) on compatible hardware (sm1.3 and later)
- Support for streamed execution, enabling asynchronous computation and data movement
- FFTW compatible data layouts
- Arbitrary intra- and inter-dimension element strides
- Thread-safe API that can be called from multiple independent host threads

Chapter 2
Using the CUFFT API

This section describes how to use the CUFFT library API. The CUFFT API is modeled after FFTW (`http://www.fftw.org`), which is one of the most popular and efficient CPU-based FFT libraries. FFTW provides a simple configuration mechanism called a *plan* that completely specifies the optimal plan of execution, in terms of minimum floating-point operations (FLOPs) for a particular FFT size and data type. Then, when the execution function is called, the actual transform takes place following the plan of execution. The advantage of this approach is that once the user creates a plan, the library stores whatever state is needed to execute the plan multiple times without recalculation of the configuration. The FFTW model works well for CUFFT because different kinds of FFTs require different thread configurations and GPU resources, and plans are a simple way to store and reuse configurations.

The first basic step in using the CUFFT Library is to create a plan using one of the following:

- cufftPlanMany() - Creates a plan supporting batched input and strided data layouts.
- cufftPlan1D()/cufftPlan2D()/cufftPlan3D() - Creates a simple plan for a 1D/2D/3D transform respectively.

Among the plan creation functions, cufftPlanMany() allows using more complicated data layouts and batched executions. Execution of a transform of a particular size and type may take several stages of processing. When a plan for the transform is generated, CUFFT derives the internal steps that need to be taken. These steps may include multiple kernel launches, memory copies, and so on. In addition, all the intermediate buffers (on CPU/GPU memory) allocations take place during planning. These buffers are released when the plan is destroyed. In the worst case, the CUFFT Library allocates space for `8*batch*n[0]*..*n[rank-1] cufftComplex` or `cufftDoubleComplex` elements (where `batch` denotes the number of transforms that will be executed in parallel and `n[]` is the array of transform dimensions) for single and double-precision transforms respectively. Depending on the configuration of the plan, less memory may be used. In some specific cases, the temporary space allocations can be as low as `1*batch*n[0]*..*n[rank-1] cufftComplex` or `cufftDoubleComplex` elements. This temporary space will be allocated separately for each individual plan when it is created (i.e., temporary space is not shared between the plans).

The next step in using the library is to call an execution function which will perform the transform with the specifications defined at planning. Transform execution functions for single and double-precision are separately defined as follows:

- cufftExecC2C()/cufftExecZ2Z()() - Performs complex-to-complex transforms.
- cufftExecR2C()/cufftExecD2Z() - Performs real-to-complex transforms.
- cufftExecC2R()/cufftExecZ2D() - Performs complex-to-real transforms.

One can create a CUFFT plan and perform multiple transforms on different data sets by providing different input and output pointers. Once the plan is no longer needed, the cufftDestroy() function should be called to release the resources allocated for the plan.

The rest of this chapter is organized as follows: The layout specifications for input and output data are covered in Section 2.1.Section 2.2 provides basic information regarding performance and accuracy of various transform configurations. Section 2.3 presents how to use streamed CUFFT executions.

2.1 Data Layout

In the CUFFT Library, data layout depends strictly on the configuration and the transform type. If the transform type is single precision real-to-complex, the input data shall be `cufftReal`/`float` type data. For complex FFTs, the input and output arrays must interleave the real and imaginary parts (the `cufftComplex`/`cufftDoubleComplex` types in single- and double-precision modes respectively). The transform size in each dimension is the number of `cufftComplex`/`cufftDoubleComplex` elements.

For 1D single-precision complex-to-complex transforms, the stride between signals in a batch is assumed to be the number of `cufftComplex` elements in the logical transform size. However, for real-data FFTs, the distance between signals in a batch depends on whether the transform is in-place or out-of-place, and on layout specifications set in the `SetCompatibilityMode()` API. The number of elements for input and output data for different transform configurations is summarized in Table 2.1 for default padding modes, where the output matches with FFTW input/output formats. When speed is favored over FFTW compatible output, "native" mode can be used, where there are no additional padding bytes in C2R/R2C modes, as described in 2.2. Note that for double-precision real-to-complex (R2C) transforms, the input element type is `cufftDoubleReal` and output element type is `cufftDoubleComplex`.

For real-to-complex FFTs, the output array holds only the non-redundant complex coefficients. For an N-element transform, the output array holds $N/2 + 1$ `cufftComplex` terms. For higher-dimensional real transforms of the form $N0 \times N1 \times \ldots \times Nn$, the last dimension is cut in half such that the output data is $N0 \times N1 \times \ldots \times (Nn/2 + 1)$ complex elements. Therefore, in order to perform an in-place FFT, the user has to pad the input array in the last dimension to $Nn/2 + 1$ complex elements interleaved. Note that the

Table 2.1: Padded Data Layouts

Dims	Type	In-place		Out-of-place	
		input	**output**	**input**	**output**
1D	C2C	x	x	x	x
	C2R	(x/2 + 1)	2(x/2 + 1)	(x/2 + 1)	x
	R2C	2(x/2 + 1)	(x/2 + 1)	x	(x/2+1)
2D	C2C	xy	xy	xy	xy
	C2R	x(y/2 + 1)	2x(y/2 + 1)	x(y/2 + 1)	xy
	R2C	2x(y/2+1)	x(y/2 + 1)	xy	x(y/2+1)
3D	C2C	xyz	xyz	xyz	xyz
	C2R	xy(z/2 + 1)	2xy(z/2 + 1)	xy(z/2 + 1)	xyz
	R2C	2xy(z/2+1)	xy(z/2 + 1)	xyz	xy(z/2+1)

Table 2.2: Native Data Layouts

Dims	Type	In-place		Out-of-place	
		input	**output**	**input**	**output**
1D	C2C	x	x	x	x
	C2R	(x/2 + 1)	x	(x/2 + 1)	x
	R2C*	x	(x/2 + 1)	x	(x/2+1)
2D	C2C	xy	xy	xy	xy
	C2R	x(y/2 + 1)	xy	x(y/2 + 1)	xy
	R2C*	xy	x(y/2 + 1)	xy	x(y/2+1)
3D	C2C	xyz	xyz	xyz	xyz
	C2R	xy(z/2 + 1)	xyz	xy(z/2 + 1)	xyz
	R2C*	xyz	xy(z/2 + 1)	xyz	xy(z/2+1)

(*total transform size is limited to 2^{27} elements in in-place R2C "native" transforms)

real-to-complex transform is implicitly forward. Passing the `CUFFT_R2C` constant to any plan creation function configures a single-precision real-to-complex FFT. Passing the `CUFFT_D2Z` constant configures a double-precision real-to-complex FFT.

The requirements for complex-to-real FFTs are similar to those for real-to-complex. In this case, the input array holds only the non-redundant, $N/2 + 1$ complex coefficients from a real-to-complex transform. The output is simply N elements of type `cufftReal`. For an in-place transform where FFTW compatible output is desired, the input size must be padded to $2 * (N/2 + 1)$ real elements. For details on padding options, please refer to Section 3.9. The complex-to-real transform is implicitly inverse. Passing the `CUFFT_C2R` constant to any plan creation function configures a single-precision complex-to-real FFT. Passing `CUFFT_Z2D` constant configures a double-precision complex-to-real FFT.

For in-place complex-to-real FFTs where FFTW compatible output is selected (default padding mode, see 3.9 for details), the input stride is assumed to be $N/2 + 1$

`cufftComplex` elements. For out-of-place transforms, input and output strides match the logical transform size N and the non-redundant size $N/2 + 1$, respectively.

Starting with CUFFT version 4.1, transforms with advanced data layout are supported through the `cufftPlanMany()` function. In this mode, the developer can define strides between each element as well as between the batches (see 2.1.2).

2.1.1 FFTW Compatibility Mode

For some transform sizes, FFTW requires additional padding bytes between rows and planes of real-to-complex (R2C) and complex-to-real (C2R) transforms of rank greater than 1. (For details, please refer to the FFTW online documentation at `http://www.fftw.org`.)

One can disable FFTW-compatible layout using `cufftSetCompatibilityMode()`. Setting the input parameter to CUFFT_COMPATIBILITY_NATIVE disables padding and ensures compact data layout for the input/output data for Real-to-Complex/Complex-To-Real transforms. Disabling padding using CUFFT native mode might provide significant speed-up especially in power-of-two sized transforms.

The FFTW compatibility modes are as follows:

```
CUFFT_COMPATIBILITY_NATIVE
CUFFT_COMPATIBILITY_FFTW_PADDING
CUFFT_COMPATIBILITY_FFTW_ASYMMETRIC
CUFFT_COMPATIBILITY_FFTW_ALL
```

`CUFFT_COMPATIBILITY_NATIVE` mode disables FFTW compatibility, but achieves the highest performance.

`CUFFT_COMPATIBILITY_FFTW_PADDING` supports FFTW data padding by inserting extra padding between packed in-place transforms for batched transforms (default).

`CUFFT_COMPATIBILITY_FFTW_ASYMMETRIC` waives the C2R symmetry requirement. Once set, it guarantees FFTW-compatible output for non-symmetric complex inputs for transforms with power-of-2 size. This is only useful for artificial (that is, random) data sets as actual data will always be symmetric if it has come from the real plane. Enabling this mode can significantly impact performance.

`CUFFT_COMPATIBILITY_FFTW_ALL` enables full FFTW compatibility. Refer to the FFTW documentation (`http://www.fftw.org`) for FFTW data layout specifications.

2.1.2 Advanced Data Layout

The advanced data layout feature allows transforming only a subset of an input array, or outputting to only a portion of a larger data structure. If `inembed` or `onembed` are set to

NULL, then the CUFFT Library assumes a basic data layout and ignores the other advanced parameters. If the the advanced parameters are to be used, then all of the advanced interface parameters should be specified correctly. Advanced parameters are defined in units of the relevant data type (`cufftReal`, `cufftDoubleReal`, `cufftComplex`, or `cufftDoubleComplex`).

The following equations illustrate how these parameters are used to calculate the index for each element in the input or output array:

$\texttt{b} = 0 \text{ .. } \texttt{batch} - 1$

$\texttt{x} = 0 \text{ .. } \texttt{n[0]} - 1$

$\texttt{y} = 0 \text{ .. } \texttt{n[1]} - 1$

$\texttt{z} = 0 \text{ .. } \texttt{n[2]} - 1$

- 1D

$$\texttt{input\_index} = \texttt{b} * \texttt{idist} + \texttt{x} * \texttt{istride}$$
$$\texttt{output\_index} = \texttt{b} * \texttt{odist} + \texttt{x} * \texttt{ostride}$$

- 2D

$$\texttt{input\_index} = \texttt{b} * \texttt{idist} + (\texttt{x} * \texttt{inembed[1]} + \texttt{y}) * \texttt{istride}$$
$$\texttt{output\_index} = \texttt{b} * \texttt{odist} + (\texttt{x} * \texttt{onembed[1]} + \texttt{y}) * \texttt{ostride}$$

- 3D

$$\texttt{input\_index} = \texttt{b} * \texttt{idist} + ((\texttt{x} * \texttt{inembed[1]} + \texttt{y}) * \texttt{inembed[2]} + \texttt{z}) * \texttt{istride}$$
$$\texttt{output\_index} = \texttt{b} * \texttt{odist} + ((\texttt{x} * \texttt{onembed[1]} + \texttt{y}) * \texttt{onembed[2]} + \texttt{z}) * \texttt{ostride}$$

The `istride` and `ostride` parameters denote the distance between two successive input and output elements in the least significant (that is, the innermost) dimension respectively. In a 1D transform, if every input element is to be used in the transform, `istride` should be set to 1; if every other input element is to be used in the transform, then `istride` should be set to 2. Similarly, in a 1D transform, if it is desired to output final elements one after another compactly, `ostride` should be set to 1; if spacing is desired between the least significant dimension output data, `ostride` should be set to the distance between the elements.

The `inembed` and `onembed` parameters define the number of elements in each dimension in the input array and the output array respectively. The `inembed[rank-1]` contains the number of elements in the least significant (innermost) dimension of the input data excluding the `istride` elements; the number of total elements in the least significant dimension of the input array is then `istride*inembed[rank-1]`. The `inembed[0]` or `onembed[0]` corresponds to the most significant (that is, the outermost) dimension and is

effectively ignored since the `idist` or `odist` parameter provides this information instead. Note that the size of each dimension of the transform should be less than or equal to the `inembed` and `onembed` values for the corresponding dimension, that is `n[`i`]` $\leq$ `inembed[`i`]`, `n[`i`]` $\leq$ `onembed[`i`]`, where i is in 0 .. `rank` $-$ 1.

The `idist` and `odist` parameters indicate the distance between the first element of two consecutive batches in the input and output data. Once can derive the total input data size as `isize * batch` in units of transform elements (e.g. cufftComplex in a C2C single-precision transform).

2.2 Accuracy and Performance

A general DFT can be implemented as a matrix vector multiplication that requires O(N^2) operations. However, the CUFFT Library employs the Cooley-Tukey algorithm (`http://en.wikipedia.org/wiki/Cooley-Tukey_FFT_algorithm`) to reduce the number of required operations to optimize the performance of particular transform sizes. This algorithm expresses a DFT recursively in terms of smaller DFT building blocks. The CUFFT Library implements the following DFT building blocks: radix-2, radix-3, radix-5, and radix-7. Hence the performance of any transform size that can be factored as $2^a * 3^b * 5^c * 7^d$ (where a, b, c, and d are non-negative integers) is optimized in the CUFFT library. For transform sizes with large prime factors (>49), single dimensional transforms might be handled by the Bluestein algorithm (`http://en.wikipedia.org/wiki/Bluestein's_FFT_algorithm`), which is built on top of the Cooley-Tukey algorithm. The accuracy of the Bluestein implementation degrades with larger sizes compared to the pure Cooley-Tukey implementation, specifically in single-precision mode, due to the accumulation of floating-point operation inaccuracies. The pure Cooley-Tukey implementation has excellent accuracy, with the relative error growing proportionally to $\log_2(N)$, where N is the transform size in points.

For sizes handled by the Cooley-Tukey code path (that is, strict multiples of 2, 3, 5, and 7), the most efficient implementation is obtained by applying the following constraints (listed in order from the most generic to the most specialized constraint, with each subsequent constraint providing the potential of an additional performance improvement).

- *Restrict the size along all dimensions to be a multiple of* 2*,* 3*,* 5*, or* 7 *only.*
  The CUFFT library has highly optimized kernels for tranforms whose dimensions have these prime factors.
- *Restrict the size along each dimension to use fewer distinct prime factors.*
  For example, a transform of size 3^n will usually be faster than one of size $2^i * 3^j$ even if the latter is slightly smaller.
- *Restrict the power-of-two factorization term of the x dimension to be at least a multiple of either* 16 *for single-precision transforms or* 8 *for double-precision transforms.*
  This aids with memory coalescing on Tesla-class and Fermi-class GPUs.

- *Restrict the power-of-two factorization term of the x dimension to be a multiple of either* 256 *for single-precision transforms or* 64 *for double-precision transforms.*
  This further aids with memory coalescing.
- *Restrict the x dimension of single-precision transforms to be strictly a power of two either between* 2 *and* 8192 *for Fermi-class GPUs or between* 2 *and* 2048 *for earlier architectures.*
  These transforms are implemented as specialized hand-coded kernels that keep all intermediate results in shared memory.
- *Use Native compatibility mode for in-place complex-to-real or real-to-complex transforms.*
  This scheme reduces the write/read of padding bytes hence helping with coalescing of the data.

Starting with version 3.1 of the CUFFT Library, the conjugate symmetry property of real-to-complex output data arrays and complex-to-real input data arrays is exploited when the power-of-two factorization term of the x dimension is at least a multiple of 4. Large 1D sizes (powers-of-two larger than $65,536$), 2D, and 3D transforms benefit the most from the performance optimizations in the implementation of real-to-complex or complex-to-real transforms.

2.3 Streamed CUFFT Transforms

Every CUFFT plan may be associated with a CUDA stream. Once so associated, all launches of the internal stages of that plan take place through the specified stream. Streaming of CUFFT execution allows for potential overlap between transforms and memory copies. (See the *NVIDIA CUDA Programming Guide* for more information on streams.) If no stream is associated with a plan, launches take place in stream 0, the default CUDA stream, and no overlap will be possible. Note that many plan executions require multiple kernel launches.

2.4 Thread Safety

Starting with CUFFT version 4.1, the CUFFT Library is thread safe and its functions can be called from multiple host threads, even with the same plan (`cufftHandle`).

Chapter 3
CUFFT Types and Definitions

This section describes the CUFFT API data-types and transform directions.

3.1 cufftHandle

A handle type used to store and access CUFFT plans. The user receives a handle after creating a CUFFT plan and uses this handle to execute the plan.

```
typedef unsigned int cufftHandle;
```

3.2 cufftResult

An enumeration of values used exclusively as API function return values. The possible return values are defined as follows:

```
typedef enum cufftResult_t {
    CUFFT_SUCCESS,        //  The CUFFT operation was successful
    CUFFT_INVALID_PLAN,   //  CUFFT was passed an invalid plan handle
    CUFFT_ALLOC_FAILED,   //  CUFFT failed to allocate GPU or CPU memory
    CUFFT_INVALID_TYPE,   //  No longer used
    CUFFT_INVALID_VALUE,  //  User specified an invalid pointer or parameter
    CUFFT_INTERNAL_ERROR, //  Used for all driver and internal CUFFT  library errors
    CUFFT_EXEC_FAILED,    //  CUFFT failed to execute an FFT on the GPU
    CUFFT_SETUP_FAILED,   //  The CUFFT library failed to initialize
    CUFFT_INVALID_SIZE,   //  User specified an invalid transform size
    CUFFT_UNALIGNED_DATA  //  No longer used
} cufftResult;
```

All CUFFT Library return values (except CUFFT_SUCCESS) indicate that the current API call failed and the user should reconfigure to correct the problem.

3.3 cufftReal

A single-precision, floating-point real data type.

```
typedef float cufftReal;
```

3.4 cufftDoubleReal

A double-precision, floating-point real data type.

```
typedef double cufftDoubleReal;
```

3.5 cufftComplex

A single-precision, floating-point complex data type that consists of interleaved real and imaginary components.

```
typedef cuComplex cufftComplex;
```

3.6 cufftDoubleComplex

A double-precision, floating-point complex data type that consists of interleaved real and imaginary components.

```
typedef cuDoubleComplex cufftDoubleComplex;
```

3.7 cufftCompatibility

CUFFT Library defines FFTW compatible data layouts using the following enumeration of values. See **??** for more details.

```
typedef enum cufftCompatibility_t {
    CUFFT_COMPATIBILITY_NATIVE          = 0x00,
    CUFFT_COMPATIBILITY_FFTW_PADDING    = 0x01, // The default value
    CUFFT_COMPATIBILITY_FFTW_ASYMMETRIC = 0x02, // asymmetric input (C2R or Z2D only)
    CUFFT_COMPATIBILITY_FFTW_ALL        = 0x03, // asymmetric and padding mode
} cufftCompatibility;
```

3.8 CUFFT Transform Types

The CUFFT library supports complex- and real-data transforms. The `cufftType` data type is an enumeration of the types of transform data supported by CUFFT.

```
typedef enum cufftType_t {
    CUFFT_R2C = 0x2a,  // Real to complex (interleaved)
    CUFFT_C2R = 0x2c,  // Complex (interleaved) to real
    CUFFT_C2C = 0x29,  // Complex to complex (interleaved)
    CUFFT_D2Z = 0x6a,  // Double to double-complex (interleaved)
    CUFFT_Z2D = 0x6c,  // Double-complex (interleaved) to double
    CUFFT_Z2Z = 0x69   // Double-complex to double-complex (interleaved)
} cufftType;
```

3.9 CUFFT Transform Directions

The CUFFT library defines forward and inverse Fast Fourier Transforms according to the sign of the complex exponential term.

```
#define CUFFT_FORWARD -1
#define CUFFT_INVERSE  1
```

CUFFT performs un-normalized FFTs; that is, performing a forward FFT on an input data set followed by an inverse FFT on the resulting set yields data that is equal to the input, scaled by the number of elements. Scaling either transform by the reciprocal of the size of the data set is left for the user to perform as seen fit.

Chapter 4
CUFFT API Reference

The CUFFT library initializes internal data upon the first invocation of an API function. Therefore, all API functions could return the `CUFFT_SETUP_FAILED` error code if the library fails to initialize. CUFFT shuts down automatically when all user-created FFT plans are destroyed.

4.1 Function cufftPlanMany()

```
cufftResult
    cufftPlanMany(cufftHandle *plan, int rank, int *n, int *inembed,
        int istride, int idist, int *onembed, int ostride,
        int odist, cufftType type, int batch);
```

Creates a FFT plan configuration of dimension `rank`, with sizes specified in the array `n`. The `batch` input parameter tells CUFFT how many transforms to configure. With this function, batched plans of 1, 2, or 3 dimensions may be created.

The `cufftPlanMany()` API supports more complicated input and output data layouts via the advanced data layout parameters `inembed`, `istride`, `idist`, `onembed`, `ostride`, and `odist`.

Input	
`plan`	Pointer to a `cufftHandle` object
`rank`	Dimensionality of the transform (1, 2, or 3)
`n`	Array of size `rank`, describing the size of each dimension
`inembed`	Pointer of size `rank` that indicates the storage dimensions of the input data in memory
`istride`	Indicates the distance between two successive input elements in the least significant (i.e., innermost) dimension
`idist`	Indicates the distance between the first element of two consecutive batches in the input data
`onembed`	Pointer of size `rank` that indicates the storage dimensions of the output data in memory
`ostride`	Indicates the distance between two successive output elements in the output array in the least significant (i.e., innermost) dimension
`odist`	Indicates the distance between the first element of two consecutive batches in the output data
`type`	The transform data type (e.g. `CUFFT_R2C` for single precision real to complex)
`batch`	Batch size for this transform

Output	
`plan`	Contains a CUFFT plan handle

Return Values	
`CUFFT_SUCCESS`	CUFFT successfully created the FFT plan.
`CUFFT_ALLOC_FAILED`	The allocation of GPU resources for the plan failed.
`CUFFT_INVALID_TYPE`	The `type` parameter is not supported.
`CUFFT_INVALID_VALUE`	One or more invalid parameters were passed to the API.
`CUFFT_INTERNAL_ERROR`	An internal driver error was detected.
`CUFFT_SETUP_FAILED`	The CUFFT library failed to initialize.
`CUFFT_INVALID_SIZE`	One or more of the `n[0]..n[rank-1]` parameters is not a supported size.

4.2 Function cufftPlan1d()

```
cufftResult
    cufftPlan1d(cufftHandle *plan, int nx, cufftType type, int batch)
```

Creates a 1D FFT plan configuration for a specified signal size and data type. The `batch` input parameter tells CUFFT how many 1D transforms to configure.

Input	
`plan`	Pointer to a `cufftHandle` object
`nx`	The transform size (e.g. 256 for a 256-point FFT)
`type`	The transform data type (e.g., `CUFFT_C2C` for single precision complex to complex)
`batch`	Number of transforms of size `nx`

Output	
`plan`	Contains a CUFFT 1D plan handle value

Return Values	
`CUFFT_SUCCESS`	CUFFT successfully created the FFT plan.
`CUFFT_ALLOC_FAILED`	The allocation of GPU resources for the plan failed.
`CUFFT_INVALID_TYPE`	The `type` parameter is not supported.
`CUFFT_INVALID_VALUE`	One or more invalid parameters were passed to the API.
`CUFFT_INTERNAL_ERROR`	An internal driver error was detected.
`CUFFT_SETUP_FAILED`	The CUFFT library failed to initialize.
`CUFFT_INVALID_SIZE`	The `nx` parameter is not a supported size.

4.3 Function cufftPlan2d()

```
cufftResult
    cufftPlan2d(cufftHandle *plan, int nx, int ny, cufftType type)
```

Creates a 2D FFT plan configuration according to specified signal sizes and data type.

Input	
`plan`	Pointer to a `cufftHandle` object
`nx`	The transform size in the x dimension (number of rows)
`ny`	The transform size in the y dimension (number of columns)
`type`	The transform data type (e.g. `CUFFT_C2R` for single precison complex to real)

Output	
`plan`	Contains a CUFFT 2D plan handle value

Return Values	
`CUFFT_SUCCESS`	CUFFT successfully created the FFT plan.
`CUFFT_ALLOC_FAILED`	The allocation of GPU resources for the plan failed.
`CUFFT_INVALID_TYPE`	The `type` parameter is not supported.
`CUFFT_INVALID_VALUE`	One or more invalid parameters were passed to the API.
`CUFFT_INTERNAL_ERROR`	An internal driver error was detected.
`CUFFT_SETUP_FAILED`	The CUFFT library failed to initialize.
`CUFFT_INVALID_SIZE`	Either or both of the `nx` or `ny` parameters is not a supported size.

4.4 Function cufftPlan3d()

```
cufftResult
    cufftPlan3d(cufftHandle *plan, int nx, int ny, int nz, cufftType type)
```

Creates a 3D FFT plan configuration according to specified signal sizes and data type. This function is the same as `cufftPlan2d()` except that it takes a third size parameter `nz`.

Input	
`plan`	Pointer to a `cufftHandle` object
`nx`	The transform size in the x dimension
`ny`	The transform size in the y dimension
`nz`	The transform size in the z dimension
`type`	The transform data type (e.g. `CUFFT_R2C` for single precision real to complex)

Output	
`plan`	Contains a CUFFT 3D plan handle value

Return Values	
`CUFFT_SUCCESS`	CUFFT successfully created the FFT plan.
`CUFFT_ALLOC_FAILED`	The allocation of GPU resources for the plan failed.
`CUFFT_INVALID_TYPE`	The `type` parameter is not supported.
`CUFFT_INVALID_VALUE`	One or more invalid parameters were passed to the API.
`CUFFT_INTERNAL_ERROR`	An internal driver error was detected.
`CUFFT_SETUP_FAILED`	The CUFFT library failed to initialize.
`CUFFT_INVALID_SIZE`	One or more of the `nx`, `ny`, or `nz` parameters is not a supported size.

4.5 Function cufftDestroy()

```
cufftResult
    cufftDestroy((cufftHandle plan)
```

Frees all GPU resources associated with a CUFFT plan and destroys the internal plan data structure. This function should be called once a plan is no longer needed, to avoid wasting GPU memory.

Input	
`plan`	The `cufftHandle` object of the plan to be destroyed.

Return Values	
`CUFFT_SUCCESS`	CUFFT successfully destroyed the FFT plan.
`CUFFT_INVALID_PLAN`	The `plan` parameter is not a valid handle.
`CUFFT_SETUP_FAILED`	The CUFFT library failed to initialize.

4.6 Function cufftExecC2C()/cufftExecZ2Z()

```
cufftResult
    cufftExecC2C(cufftHandle *plan, cufftComplex *idata,
        cufftComplex *odata, int direction);
cufftResult
    cufftExecZ2Z(cufftHandle *plan, cufftDoubleComplex *idata,
        cufftDoubleComplex *odata, int direction);
```

cufftExecC2C(/cufftExecZ2Z) executes a single-precision(/double-precision) complex-to-complex transform plan in the transform direction as specified by `direction` parameter. CUFFT uses the GPU memory pointed to by the `idata` parameter as input data. This function stores the Fourier coefficients in the `odata` array. If `idata` and `odata` are the same, this method does an in-place transform.

Input	
`plan`	The `cufftHandle` object for the plan to be executed
`idata`	Pointer to the complex input data (in GPU memory) to transform
`odata`	Pointer to the complex output data (in GPU memory)
`direction`	The transform direction: `CUFFT_FORWARD` or `CUFFT_INVERSE`

Output	
`odata`	Contains the complex Fourier coefficients

Return Values	
`CUFFT_SUCCESS`	CUFFT successfully created the FFT plan.
`CUFFT_INVALID_PLAN`	The `plan` parameter is not a valid handle.
`CUFFT_INVALID_VALUE`	At least one of the parameters `idata`, `odata`, and `direction` is not valid.
`CUFFT_INTERNAL_ERROR`	An internal driver error was detected.
`CUFFT_EXEC_FAILED`	CUFFT failed to execute the transform on the GPU.
`CUFFT_SETUP_FAILED`	The CUFFT library failed to initialize.
`CUFFT_UNALIGNED_DATA`	No longer used.

4.7 Function cufftExecR2C()/cufftExecD2Z()

```
cufftResult
    cufftExecR2C(cufftHandle *plan, cufftReal *idata, cufftComplex *odata);
cufftResult
    cufftExecD2Z(cufftHandle *plan, cufftDoubleReal *idata, cufftDoubleComplex *odata);
```

cufftExecR2C(/cufftExecD2Z) executes a single-precision(/double-precision) real-to-complex (implicitly forward) CUFFT transform plan. CUFFT uses as input data the GPU memory pointed to by the `idata` parameter. This function stores the nonredundant Fourier coefficients in the `odata` array. `idata` and `odata` pointers are both required to be aligned to `cufftComplex` data type in single-precision transforms and `cufftDoubleComplex` data type in double-precision transforms. If `idata` and `odata` are the same, this method does an in-place transform. Note the data layout differences between in-place and out-of-place transforms as described in Section 3.8.

Input	
`plan`	The `cufftHandle` object for the plan to be executed
`idata`	Pointer to the real input data (in GPU memory) to transform
`odata`	Pointer to the complex output data (in GPU memory)
Output	
`odata`	Contains the complex Fourier coefficients
Return Values	
`CUFFT_SUCCESS`	CUFFT successfully created the FFT plan.
`CUFFT_INVALID_PLAN`	The `plan` parameter is not a valid handle.
`CUFFT_INVALID_VALUE`	At least one of the parameters `idata` and `odata` is not valid.
`CUFFT_INTERNAL_ERROR`	An internal driver error was detected.
`CUFFT_EXEC_FAILED`	CUFFT failed to execute the transform on the GPU.
`CUFFT_SETUP_FAILED`	The CUFFT library failed to initialize.
`CUFFT_UNALIGNED_DATA`	No longer used.

4.8 Function cufftExecC2R()/cufftExecZ2D

```
cufftResult
    cufftExecC2R(cufftHandle plan, cufftComplex *idata, cufftReal *odata);
cufftResult
    cufftExecZ2D(cufftHandle plan, cufftComplex *idata, cufftReal *odata);
```

cufftExecC2R(/cufftExecZ2D) executes a single-precision(/double-precision) complex-to-real (implicitly inverse) CUFFT transform plan. CUFFT uses as input data the GPU memory pointed to by the `idata` parameter. The input array holds only the nonredundant complex Fourier coefficients. This function stores the real output values in the `odata` array. `idata` and `odata` pointers are both required to be aligned to `cufftComplex` data type in single-precision transforms and `cufftDoubleComplex` type in double-precision transforms. If `idata` and `odata` are the same, this method does an in-place transform.

Input	
`plan`	The `cufftHandle` object for the plan to be executed
`idata`	Pointer to the complex input data (in GPU memory) to transform
`odata`	Pointer to the real output data (in GPU memory)
Output	
`odata`	Contains the real Fourier coefficients
Return Values	
`CUFFT_SUCCESS`	CUFFT successfully created the FFT plan.
`CUFFT_INVALID_PLAN`	The `plan` parameter is not a valid handle.
`CUFFT_INVALID_VALUE`	At least one of the parameters `idata` and `odata` is not valid.
`CUFFT_INTERNAL_ERROR`	An internal driver error was detected.
`CUFFT_EXEC_FAILED`	CUFFT failed to execute the transform on the GPU.
`CUFFT_SETUP_FAILED`	The CUFFT library failed to initialize.
`CUFFT_UNALIGNED_DATA`	No longer used.

4.9 Function cufftSetStream()

```
cufftResult
    cufftSetStream(cufftHandle plan, cudaStream_t stream);
```

Associates a CUDA stream with a CUFFT plan. All kernel launches made during plan execution are now done through the associated stream, enabling overlap with activity in other streams (e.g. data copying). The association remains until the plan is destroyed or the stream is changed with another call to `cufftSetStream()`.

Input	
`plan`	The `cufftHandle` object to associate with the stream
`stream`	A valid CUDA stream created with `cudaStreamCreate()`; `0` for the default stream

Return Values	
`CUFFT_SUCCESS`	The stream was associated with the plan.
`CUFFT_INVALID_PLAN`	The `plan` parameter is not a valid handle.

4.10 Function cufftSetCompatibilityMode()

```
cufftResult
    cufftSetCompatibilityMode(cufftHandle plan, cufftCompatibility mode);
```

Configures the layout of CUFFT output in FFTW-compatible modes. When desired, FFTW compatibility can be configured for padding only, for asymmetric complex inputs only, or for full compatibility. If the SetCompatibilityMode() API fails, later `cufftExecute*()` calls are not guaranteed to work.

Input	
`plan`	The `cufftHandle` object to associate with the stream

`mode`	The `cufftCompatibility` option to be used:
	`CUFFT_COMPATIBILITY_NATIVE`
	`CUFFT_COMPATIBILITY_FFTW_PADDING` (default)
	`CUFFT_COMPATIBILITY_FFTW_ASYMMETRIC`
	`CUFFT_COMPATIBILITY_FFTW_ALL`

Return Values	
`CUFFT_SUCCESS`	CUFFT successfully executed the FFT plan.
`CUFFT_INVALID_PLAN`	The `plan` parameter is not a valid handle.
`CUFFT_SETUP_FAILED`	The CUFFT library failed to initialize.

Chapter 5
CUFFT Code Examples

This chapter provides six simple examples of complex and real 1D, 2D, and 3D transforms that use CUFFT to perform forward and inverse FFTs.

5.1 1D Complex-to-Complex Transforms

```
#define NX 256
#define BATCH 10

cufftHandle plan;
cufftComplex *data;
cudaMalloc((void**)&data, sizeof(cufftComplex)*NX*BATCH);
if (cudaGetLastError() != cudaSuccess){
    fprintf(stderr, "Cuda error: Failed to allocate\n");
    return;
}

/* Create a 1D FFT plan. */
if (cufftPlan1d(&plan, NX, CUFFT_C2C, BATCH) != CUFFT_SUCCESS){
    fprintf(stderr, "CUFFT error: Plan creation failed");
    return;
}

/* Use the CUFFT plan to transform the signal in place. */
if (cufftExecC2C(plan, data, data, CUFFT_FORWARD) != CUFFT_SUCCESS){
    fprintf(stderr, "CUFFT error: ExecC2C Forward failed");
    return;
}

/* Inverse transform the signal in place. */
if (cufftExecC2C(plan, data, data, CUFFT_INVERSE) != CUFFT_SUCCESS){
    fprintf(stderr, "CUFFT error: ExecC2C Inverse failed");
    return;
}

/* Note:
(1) Divide by number of elements in data set to get back original data
(2) Identical pointers to input and output arrays implies in-place
transformation
*/

if (cudaThreadSynchronize() != cudaSuccess){
```

```
    fprintf(stderr, "Cuda error: Failed to synchronize\n");
    return;
}


/* Destroy the CUFFT plan. */
cufftDestroy(plan);
cudaFree(data);
```

5.2 1D Real-to-Complex Transforms

```
#define NX 256
#define BATCH 10

cufftHandle plan;
cufftComplex *data;
cudaMalloc((void**)&data, sizeof(cufftComplex)*(NX/2+1)*BATCH);
if (cudaGetLastError() != cudaSuccess){
    fprintf(stderr, "Cuda error: Failed to allocate\n");
    return;
}

/* Create a 1D FFT plan. */
if (cufftPlan1d(&plan, NX, CUFFT_R2C, BATCH) != CUFFT_SUCCESS){
    fprintf(stderr, "CUFFT error: Plan creation failed");
    return;
}

/* Use the CUFFT plan to transform the signal in place. */
if (cufftExecR2C(plan, (cufftReal*)data, data) != CUFFT_SUCCESS){
    fprintf(stderr, "CUFFT error: ExecC2C Forward failed");
    return;
}

/* Note:
(1) Divide by number of elements in data set to get back original data
(2) Identical pointers to input and output arrays implies in-place
transformation
*/

if (cudaThreadSynchronize() != cudaSuccess){
    fprintf(stderr, "Cuda error: Failed to synchronize\n");
    return;
}

/* Destroy the CUFFT plan. */
cufftDestroy(plan);
cudaFree(data);
```

5.3 2D Complex-to-Real Transforms

```
#define NX 256
#define NY 128
#define NRANK 2

cufftHandle plan;
cufftComplex *data;
int n[NRANK] = {NX, NY};

cudaMalloc((void**)&data, sizeof(cufftComplex)*NX*(NY/2+1));
if (cudaGetLastError() != cudaSuccess){
    fprintf(stderr, "Cuda error: Failed to allocate\n");
    return;
}

/* Create a 2D FFT plan. */
if (cufftPlanMany(&plan, NRANK, n,
                  NULL, 1, 0,
                  NULL, 1, 0,
                  CUFFT_C2R,BATCH) != CUFFT_SUCCESS){
    fprintf(stderr, "CUFFT Error: Unable to create plan\n");
    return;
}

if (cufftSetCompatibilityMode(plan, CUFFT_COMPATIBILITY_NATIVE)!= CUFFT_SUCCESS){
    fprintf(stderr, "CUFFT Error: Unable to set compatibility mode to native\n");
    return;
}


/* Use the CUFFT plan to transform the signal out of place. */
if (cufftExecC2R(plan, data, data) != CUFFT_SUCCESS){
    fprintf(stderr, "CUFFT Error: Unable to execute plan\n");
    return;
}

if (cudaThreadSynchronize() != cudaSuccess){
    fprintf(stderr, "Cuda error: Failed to synchronize\n");
    return;
}

/* Destroy the CUFFT plan. */
cufftDestroy(plan);
cudaFree(data);
```

5.4 3D Complex-to-Complex Transforms

```
#define NX 64
#define NY 128
#define NX 128
#define BATCH 10
#define NRANK 3

cufftHandle plan;
cufftComplex *data;
int n[NRANK] = {NX, NY, NZ};

cudaMalloc((void**)&data, sizeof(cufftComplex)*NX*NY*NZ*BATCH);
if (cudaGetLastError() != cudaSuccess){
    fprintf(stderr, "Cuda error: Failed to allocate\n");
    return;
}

/* Create a 3D FFT plan. */
if (cufftPlanMany(&plan, NRANK, n,
                  NULL, 1, NX*NY*NZ, // *inembed, istride, idist
                  NULL, 1, NX*NY*NZ, // *onembed, ostride, odist
                  CUFFT_C2C, BATCH) != CUFFT_SUCCESS){
    fprintf(stderr, "CUFFT error: Plan creation failed");
    return;
}

/* Use the CUFFT plan to transform the signal in place. */
if (cufftExecC2C(plan, data, data, CUFFT_FORWARD) != CUFFT_SUCCESS){
    fprintf(stderr, "CUFFT error: ExecC2C Forward failed");
    return;
}

/* Inverse transform the signal in place. */
if (cufftExecC2C(plan, data, data, CUFFT_INVERSE) != CUFFT_SUCCESS){
    fprintf(stderr, "CUFFT error: ExecC2C Inverse failed");
    return;
}

/* Note:
(1) Divide by number of elements in data set to get back original data
(2) Identical pointers to input and output arrays implies in-place
transformation
*/

if (cudaThreadSynchronize() != cudaSuccess){
    fprintf(stderr, "Cuda error: Failed to synchronize\n");
    return;
}

/* Destroy the CUFFT plan. */
cufftDestroy(plan);
cudaFree(data);
```

5.5 2D Advanced Data Layout Use

```
#define NX 128
#define NY 256
#define BATCH 10
#define NRANK 2

/* Advanced interface parameters, arbitrary strides */
#define ISTRIDE 2
#define OSTRIDE 1
#define IX (NX+2)
#define IY (NY+1)
#define OX (NX+3)
#define OY (NY+4)
#define IDIST (IX*IY*ISTRIDE+3)
#define ODIST (OX*OY*OSTRIDE+5)

cufftHandle plan;
cufftComplex *idata, *odata;
int isize = IDIST * BATCH;
int osize = ODIST * BATCH;
int n[NRANK] = {NX, NY};
int inembed[NRANK] = {IX, IY};
int onembed[NRANK] = {OX, OY};

cudaMalloc((void **)&idata, sizeof(cufftComplex)*isize);
cudaMalloc((void **)&odata, sizeof(cufftComplex)*osize);
if (cudaGetLastError() != cudaSuccess){
    fprintf(stderr, "Cuda error: Failed to allocate\n");
    return;
}

/* Create a batched 2D plan */
if (cufftPlanMany(&plan, NRANK, n,
                  inembed,ISTRIDE,IDIST,
                  onembed,OSTRIDE,ODIST,
                  CUFFT_C2C,BATCH) != CUFFT_SUCCESS){
    fprintf(stderr, "CUFFT Error: Unable to create plan\n");
    return;
}

/* Execute the transform out-of-place */
if (cufftExecC2C(plan, idata, odata, CUFFT_FORWARD) != CUFFT_SUCCESS){
    fprintf(stderr, "CUFFT Error: Failed to execute plan\n");
    return;
}

if (cudaThreadSynchronize() != cudaSuccess){
    fprintf(stderr, "Cuda error: Failed to synchronize\n");
    return;
}

/* Destroy the CUFFT plan */
cufftDestroy(plan);
cudaFree(idata);
cudaFree(odata);
```

Notice

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.

Information furnished is believed to be accurate and reliable. However, NVIDIA Corporation assumes no responsibility for the consequences of use of such information or for any infringement of patents or other rights of third parties that may result from its use. No license is granted by implication or otherwise under any patent rights of NVIDIA Corporation. Specifications mentioned in this publication are subject to change without notice. This publication supersedes and replaces all other information previously supplied. NVIDIA Corporation products are not authorized as critical components in life support devices or systems without express written approval of NVIDIA Corporation.

Trademarks

NVIDIA and the NVIDIA logo are trademarks and/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