CUDA

CUFFT Library

PG-05327-040_V01
February, 2011

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

Table of Contents

CUFFT Library

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, and it is one of the most important and widely used numerical algorithms, with applications that include 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, GPU-based 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, while other implementations support arbitrary transform sizes. This version of the CUFFT library supports the following features:

- **1D, 2D, and 3D transforms of complex and real-valued data**
- **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 for real and complex data**
- **Double-precision transforms on compatible hardware (GT200 and later GPUs)**
- **Support for streamed execution, enabling simultaneous computation together with data movement**

CUFFT Types and Definitions

The next sections describe the CUFFT types and transform directions:

Type cufftHandle

```
typedef unsigned int cufftHandle;
```

A handle type used to store and access CUFFT plans (see “CUFFT API Functions” on page 11 for more information about plans). For example, the user receives a handle after creating a CUFFT plan and uses this handle to execute the plan.

Type cufftResult

```
typedef enum cufftResult_t cufftResult;
```

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

Return Values

CUFFT_SUCCESS	Any CUFFT operation is successful.
CUFFT_INVALID_PLAN	CUFFT is passed an invalid plan handle.
CUFFT_ALLOC_FAILED	CUFFT failed to allocate GPU memory.
CUFFT_INVALID_TYPE	The user requests an unsupported type.
CUFFT_INVALID_VALUE	The user specifies a bad memory pointer.
CUFFT_INTERNAL_ERROR	Used for all internal driver 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	The user specifies an unsupported FFT size.
CUFFT_UNALIGNED_DATA	Input or output does not satisfy texture alignment requirements.

Type cufftReal

```
typedef float cufftReal;
```

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

Type cufftDoubleReal

```
typedef double cufftDoubleReal;
```

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

Type cufftComplex

```
typedef cuComplex cufftComplex;
```

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

Type cufftDoubleComplex

typedef cuDoubleComplex cufftDoubleComplex;

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

Type cufftCompatibility

typedef enum cufftCompatibility_t cufftCompatibility;

An enumeration of values used to control FFTW data compatibility. See “FFTW Compatibility Mode” on page 9 for details.

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
    CUFFT_Z2D = 0x6c,  // Double-complex to double
    CUFFT_Z2Z = 0x69   // Double-complex to double-complex
} cufftType;
```

For complex FFTs, the input and output arrays must interleave the real and imaginary parts (the **cufftComplex** type). The transform size in each dimension is the number of **cufftComplex** elements. The **CUFFT_C2C** constant can be passed to any plan creation function to configure a single-precision complex-to-complex FFT. Pass the **CUFFT_Z2Z** constant to configure a double-precision complex-to-complex FFT.

For real-to-complex FFTs, the output array holds only the non-redundant complex coefficients. So 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 or $2 * (N/2 + 1)$ real elements. Note that the 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`**. However, for an in-place transform, the input size must be padded to $2 * (N/2 + 1)$ real elements. 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 1D 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. For in-place FFTs, the input stride is assumed to be $2 * (N/2 + 1)$ **`cufftReal`** elements or $N/2 + 1$ **`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 3.0, batched transforms are supported through the **`cufftPlanMany()`** function. Although this function takes input parameters that specify input- and output-data strides, as of version 3.0 it is assumed the data for each signal within the batch immediately follow the data of the previous one (a stride of 1).

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
```

For higher-dimensional transforms (2D and 3D), CUFFT performs FFTs in row-major or C order. For example, if the user requests a 3D transform plan for sizes X, Y, and Z, CUFFT transforms along Z, Y, and then X. The user can configure column-major FFTs by simply changing the order of the size parameters to the plan creation API functions.

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.

Streamed CUFFT Transforms

Execution of a transform of a particular size and type may take several stages of processing. A plan for the transform is generated, in which CUFFT specifies the internal steps that need to be taken. These steps may include multiple kernel launches, memory copies, and so on.

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 launches 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).

FFTW Compatibility Mode

For some transform sizes, FFTW requires additional padding bytes between rows and planes of Real2Complex (R2C) and Complex2Real (C2R) transforms of rank greater than 1. (For details, please refer to the FFTW online documentation at http://www.fftw.org.)

To speed up R2C and C2R transforms for power-of-2 sizes similar to their Complex2Complex (C2C) equivalent, one can disable FFTW-compatible layout using **`cufftSetCompatibilityMode()`**, introduced in release 3.1 and described on page 23. When native mode is selected for this function, power-of-2 transform sizes will be compact and CUFFT will not use padding. Non-power-of-2 sizes will continue to use the same padding layout as FFTW.

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 with power-of-2 size.
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.

CUFFT API Functions

The CUFFT API is modeled after FFTW, 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—that is, the minimum floating-point operation (flop)—plan of execution for a particular FFT size and data type. 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 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.

The CUFFT functions are as follows:

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 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`	Allocation of GPU resources for the plan failed.
`CUFFT_INVALID_TYPE`	The `type` parameter is not supported.
`CUFFT_INVALID_VALUE`	Invalid parameter(s) passed to the API.
`CUFFT_INTERNAL_ERROR`	Internal driver error is detected.
`CUFFT_SETUP_FAILED`	CUFFT library failed to initialize.
`CUFFT_INVALID_SIZE`	The `nx` parameter is not a supported size.

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. This function is the same as **`cufftPlan1d()`** except that it takes a second size parameter, `ny`, and does not support batching.

Input

`plan`	Pointer to a **`cufftHandle`** object
`nx`	The transform size in the X-dimension (number of rows)

Input (continued)

`ny`	The transform size in the Y-dimension (number of columns)
`type`	The transform data type (e.g., **`CUFFT_C2R`** for 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`	Allocation of GPU resources for the plan failed.
`CUFFT_INVALID_TYPE`	The `type` parameter is not supported.
`CUFFT_INVALID_VALUE`	Invalid parameter(s) passed to the API.
`CUFFT_INTERNAL_ERROR`	Internal driver error is detected.
`CUFFT_SETUP_FAILED`	CUFFT library failed to initialize.
`CUFFT_INVALID_SIZE`	The `nx` parameter is not a supported size.

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 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`	Allocation of GPU resources for the plan failed.

Return Values (continued)

CUFFT_INVALID_TYPE	The `type` parameter is not supported.
CUFFT_INVALID_VALUE	Invalid parameter(s) passed to the API.
CUFFT_INTERNAL_ERROR	Internal driver error is detected.
CUFFT_SETUP_FAILED	CUFFT library failed to initialize.
CUFFT_INVALID_SIZE	The `nx` parameter is not a supported size.

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 in parallel. With this function, batched plans of any dimension may be created.

The CUFFT Library now supports more complicated input and output data layouts as a Beta feature via the advanced data layout parameters `inembed`, `istride`, `idist`, `onembed`, `ostride`, and `odist` in **`cufftPlanMany()`**. In this release, these parameters are supported only for complex-to-complex (C2C) transforms. This feature allows transforming 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 functions as it did in the previous releases: it 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`**, **`cuComplex`**, **`cuDoubleComplex`**).

The `inembed` and `onembed` parameters are defined as pointers of size `rank` and indicate storage dimensions of the input and output data in memory respectively. The first value in the array, `inembed[0]` or `onembed[0]`, corresponding to the most significant (that is, the outermost) dimension, is effectively ignored since the `idist` or `odist` parameter provides this information instead. Note that each dimension of the transform should be less than or equal to the inembed and onembed values for the corresponding dimension, that is

$n[i] \le inembed[i]$, $n[i] \le onembed[i]$, where i is in $0..rank-1$.

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**. 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 highest ranking dimension output data, ostride should be set to the distance between the elements.

The idist and odist parameters indicate the distance between the first element of two consecutive batches in the input and output data.

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

- **1D**

$$input\_index = b * idist + x * istride$$

$$output\_index = b * odist + x * ostride$$

$$b = 0..count-1$$

$$x = 0..n[0]-1$$

- **2D**

$$input\_index = b * idist + (x * inembed[1] + y) * istride$$

$$output\_index = b * odist + (x * onembed[1] + y) * ostride$$

$$b = 0..count-1$$

$$x = 0..n[0]-1$$

$$y = 0..n[1]-1$$

- **3D**

$$input\_index = b * idist + ((x * inembed[1] + y) * inembed[2] + z) * istride$$

$$output\_index = b * odist + ((x * onembed[1] + y) * onembed[2] + z) * ostride$$

$$b = 0..count-1$$

```
x = 0..n[0] – 1
y = 0..n[1] – 1
z = 0..n[2] – 1
```

Input

plan	Pointer to a **cufftHandle** object
rank	Dimensionality of the transform (1, 2, or 3)
n	An array of size rank, describing the size of each dimension
inembed	This parameter is a pointer of size rank, and it indicates storage dimensions of the input data in memory.
istride	This parameters defines the distance between two successive input elements in the least significant (i.e., the innermost) dimension.
idist	This parameter indicates the distance between the first element of two consecutive batches in the input data.
onembed	This parameter is a pointer of size rank, and it indicates storage dimensions of the output data in memory.
ostride	This parameters defines the distance between two successive output elements in the output array in the least significant (i.e., the innermost) dimension.
odist	This parameter indicates the distance between the first element of two consecutive batches in the output data.
type	Transform data type (e.g., **CUFFT_C2C**, as per other CUFFT calls)
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	Allocation of GPU resources for the plan failed.
CUFFT_INVALID_TYPE	The type parameter is not supported.
CUFFT_INVALID_VALUE	Invalid parameter(s) passed to the API.
CUFFT_INTERNAL_ERROR	Internal driver error is detected.
CUFFT_SETUP_FAILED	CUFFT library failed to initialize.
CUFFT_INVALID_SIZE	The nx parameter is not a supported size.

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 created the FFT plan.
`CUFFT_INVALID_PLAN`	The `plan` parameter is not a valid handle.
`CUFFT_SETUP_FAILED`	CUFFT library failed to initialize.

Function cufftExecC2C()

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

Executes a CUFFT single-precision complex-to-complex transform plan as specified by `direction`. CUFFT uses as input data the GPU memory pointed to by the `idata` parameter. 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 update
`idata`	Pointer to the single-precision complex input data (in GPU memory) to transform
`odata`	Pointer to the single-precision 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	The `idata`, `odata`, and/or `direction` parameter is not valid.
CUFFT_INTERNAL_ERROR	Internal driver error is detected.
CUFFT_EXEC_FAILED	CUFFT failed to execute the transform on GPU.
CUFFT_SETUP_FAILED	CUFFT library failed to initialize.
CUFFT_UNALIGNED_DATA	Input or output does not satisfy texture alignment requirements.

Function cufftExecR2C()

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

Executes a CUFFT single-precision real-to-complex (implicitly forward) transform plan. CUFFT uses as input data the GPU memory pointed to by the `idata` parameter. This function stores the non-redundant Fourier coefficients in the `odata` array. If `idata` and `odata` are the same, this method does an in-place transform (See "CUFFT Transform Types" on page 7 for details on real data FFTs.)

Input

`plan`	The **`cufftHandle`** object for the plan to update
`idata`	Pointer to the single-precision real input data (in GPU memory) to transform
`odata`	Pointer to the single-precision 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	The `idata`, `odata`, and/or `direction` parameter is not valid.
CUFFT_INTERNAL_ERROR	Internal driver error is detected.
CUFFT_EXEC_FAILED	CUFFT failed to execute the transform on GPU.

Return Values (continued)

CUFFT_SETUP_FAILED	CUFFT library failed to initialize.
CUFFT_UNALIGNED_DATA	Input or output does not satisfy texture alignment requirements.

Function cufftExecC2R()

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

Executes a CUFFT single-precision complex-to-real (implicitly inverse) transform plan. CUFFT uses as input data the GPU memory pointed to by the `idata` parameter. The input array holds only the non-redundant complex Fourier coefficients. This function stores the real output values in the `odata` array. If `idata` and `odata` are the same, this method does an in-place transform. (See "CUFFT Transform Types" on page 7 for details on real data FFTs.)

Input

`plan`	The **`cufftHandle`** object for the plan to update
`idata`	Pointer to the single-precision complex input data (in GPU memory) to transform
`odata`	Pointer to the single-precision real output data (in GPU memory)

Output

`odata`	Contains the real-valued output data

Return Values

CUFFT_SUCCESS	CUFFT successfully created the FFT plan.
CUFFT_INVALID_PLAN	The `plan` parameter is not a valid handle.
CUFFT_INVALID_VALUE	The `idata`, `odata`, and/or `direction` parameter is not valid.
CUFFT_INTERNAL_ERROR	Internal driver error is detected.
CUFFT_EXEC_FAILED	CUFFT failed to execute the transform on GPU.
CUFFT_SETUP_FAILED	CUFFT library failed to initialize.
CUFFT_UNALIGNED_DATA	Input or output does not satisfy texture alignment requirements.

Function cufftExecZ2Z()

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

Executes a CUFFT double-precision complex-to-complex transform plan as specified by `direction`. CUFFT uses as input data the GPU memory pointed to by the `idata` parameter. 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 update
`idata`	Pointer to the double-precision complex input data (in GPU memory) to transform
`odata`	Pointer to the double-precision 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`	The `idata`, `odata`, and/or `direction` parameter is not valid.
`CUFFT_INTERNAL_ERROR`	Internal driver error is detected.
`CUFFT_EXEC_FAILED`	CUFFT failed to execute the transform on GPU.
`CUFFT_SETUP_FAILED`	CUFFT library failed to initialize.
`CUFFT_UNALIGNED_DATA`	Input or output does not satisfy texture alignment requirements.

Function cufftExecD2Z()

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

Executes a CUFFT double-precision real-to-complex (implicitly forward) transform plan. CUFFT uses as input data the GPU memory pointed to by the `idata` parameter. This function stores the non-redundant Fourier coefficients in the `odata` array. If `idata` and `odata` are the same, this method does an in-place transform (See "CUFFT Transform Types" on page 7 for details on real data FFTs.)

Input

`plan`	The **`cufftHandle`** object for the plan to update
`idata`	Pointer to the double-precision real input data (in GPU memory) to transform
`odata`	Pointer to the double-precision 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`	The `idata`, `odata`, and/or `direction` parameter is not valid.
`CUFFT_INTERNAL_ERROR`	Internal driver error is detected.
`CUFFT_EXEC_FAILED`	CUFFT failed to execute the transform on GPU.
`CUFFT_SETUP_FAILED`	CUFFT library failed to initialize.
`CUFFT_UNALIGNED_DATA`	Input or output does not satisfy texture alignment requirements.

Function cufftExecZ2D()

```
cufftResult
cufftExecZ2D(
    cufftHandle plan, cufftDoubleComplex *idata,
    cufftDoubleReal *odata );
```

Executes a CUFFT double-precision complex-to-real (implicitly inverse) transform plan. CUFFT uses as input data the GPU memory pointed to by the `idata` parameter. The input array holds only the non-redundant complex Fourier coefficients. This function stores the real output values in the `odata` array. If `idata` and `odata` are the same, this method does an in-place transform. (See "CUFFT Transform Types" on page 7 for details on real data FFTs.)

Input

`plan`	The **`cufftHandle`** object for the plan to update
`idata`	Pointer to the double-precision complex input data (in GPU memory) to transform
`odata`	Pointer to the double-precision real output data (in GPU memory)

Output

`odata`	Contains the real-valued output data

Return Values

CUFFT_SUCCESS	CUFFT successfully created the FFT plan.
CUFFT_INVALID_PLAN	The `plan` parameter is not a valid handle.
CUFFT_INVALID_VALUE	The `idata`, `odata`, and/or `direction` parameter is not valid.
CUFFT_INTERNAL_ERROR	Internal driver error is detected.
CUFFT_EXEC_FAILED	CUFFT failed to execute the transform on GPU.
CUFFT_SETUP_FAILED	CUFFT library failed to initialize.
CUFFT_UNALIGNED_DATA	Input or output does not satisfy texture alignment requirements.

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 (for example, 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()** (or 0 for the default stream)

Output

odata	Contains the real-valued output data

Return Values

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

Function cufftSetCompatibilityMode()

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

Configures the layout of CUFFT output in FFTW-compatible modes. When FFTW compatibility is desired, it can be configured for padding only, for asymmetric complex inputs only, or to be fully compatible.

Input

plan	The **cufftHandle** object to associate with the stream
mode	The **cufftCompatibility** option to be used (see "Type cufftCompatibility" on page 7): **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	CUFFT library failed to initialize.

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 to reduce the number of required operations and, thereby, 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 other sizes, single dimensional transforms are handled by the Bluestein 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 code path, specifically in single-precision mode, due to the accumulation of floating-point operation inaccuracies. On the other hand, 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, strictly multiples of 2, 3, 5, and 7), the most efficient implementation is obtained by applying the following constraints (listed in order of the most generic to the most specialized constraint, with each subsequent constraint providing the potential of an additional performance improvement).

- *Restrict the size along any dimension to be a multiple of 2, 3, 5, or 7 only*. **For example, a transform of size 3^n will likely 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 on Tesla-class and Fermi-class GPUs.**
- *Restrict the X-dimension of single-precision transforms to be strictly a power of two between either 2 and 2048 for Tesla-class GPUs or 2 and 8192 for Fermi-class GPUs*. These transforms are implemented as specialized hand-coded kernels that keep all intermediate results in shared memory.

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; specifically, 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) and 2D and 3D transforms benefit the most from the performance optimizations in the implementation of real-to-complex or complex-to-real transforms.

CUFFT Code Examples

This section provides six simple examples of 1D, 2D, and 3D complex and real data transforms that use the CUFFT to perform forward and inverse FFTs. The examples are as follows:

- "1D Complex-to-Complex Transforms" on page 26
- "1D Real-to-Complex Transforms" on page 27
- "2D Complex-to-Complex Transforms" on page 28
- "Batched 2D Complex-to-Complex Transforms" on page 29
- "2D Complex-to-Real Transforms" on page 30
- "3D Complex-to-Complex Transforms" on page 31

1D Complex-to-Complex Transforms

```
#define NX 256
#define BATCH 10

cufftHandle plan;
cufftComplex *data;
cudaMalloc((void**)&data, sizeof(cufftComplex)*NX*BATCH);

/* Create a 1D FFT plan. */
cufftPlan1d(&plan, NX, CUFFT_C2C, BATCH);

/* Use the CUFFT plan to transform the signal in place. */
cufftExecC2C(plan, data, data, CUFFT_FORWARD);

/* Inverse transform the signal in place. */
cufftExecC2C(plan, data, data, CUFFT_INVERSE);

/* 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
*/

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

1D Real-to-Complex Transforms

```
#define NX 256
#define BATCH 10

cufftHandle plan;
cufftComplex *data;
cudaMalloc((void**)&data, sizeof(cufftComplex)*(NX/2+1)*BATCH);

/* Create a 1D FFT plan. */
cufftPlan1d(&plan, NX, CUFFT_R2C, BATCH);

/* Use the CUFFT plan to transform the signal in place. */
cufftExecR2C(plan, (cufftReal*)data, data);

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

2D Complex-to-Complex Transforms

```
#define NX 256
#define NY 128

cufftHandle plan;
cufftComplex *idata, *odata;
cudaMalloc((void**)&idata, sizeof(cufftComplex)*NX*NY);
cudaMalloc((void**)&odata, sizeof(cufftComplex)*NX*NY);

/* Create a 2D FFT plan. */
cufftPlan2d(&plan, NX, NY, CUFFT_C2C);

/* Use the CUFFT plan to transform the signal out of place. */
cufftExecC2C(plan, idata, odata, CUFFT_FORWARD);

/* Note: idata != odata indicates an out-of-place transformation
         to CUFFT at execution time. */
/* Inverse transform the signal in place */
cufftExecC2C(plan, odata, odata, CUFFT_INVERSE);

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

Batched 2D Complex-to-Complex Transforms

```
#define NX 128
#define NY 256
#define BATCHSIZE 1000

int datalen;
cufftHandle plan;
cufftComplex *indata, *outdata;

datalen = NX * NY * BATCHSIZE;
cudaMalloc((void **)&indata, sizeof(cufftComplex)*datalen);
cudaMalloc((void **)&outdata, sizeof(cufftComplex)*datalen);

/* Create a batched 2D plan */
cufftPlanMany(&plan,2,{ NX, NY },NULL,1,0,NULL,1,0,CUFFT_C2C,BATCHSIZE);

/* Execute the transform out-of-place */
cufftExecC2C(plan, indata, outdata, CUFFT_FORWARD);

/* Destroy the CUFFT plan */
cufftDestroy(plan);
cudaFree(indata);
cudaFree(outdata);
```

2D Complex-to-Real Transforms

```
#define NX 256
#define NY 128

cufftHandle plan;
cufftComplex *idata;
cufftReal *odata;
cudaMalloc((void**)&idata, sizeof(cufftComplex)*NX*NY);
cudaMalloc((void**)&odata, sizeof(cufftReal)*NX*NY);

/* Create a 2D FFT plan. */
cufftPlan2d(&plan, NX, NY, CUFFT_C2R);

/* Use the CUFFT plan to transform the signal out of place. */
cufftExecC2R(plan, idata, odata);

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

3D Complex-to-Complex Transforms

```
#define NX 64
#define NY 64
#define NZ 128

cufftHandle plan;
cufftComplex *data1, *data2;
cudaMalloc((void**)&data1, sizeof(cufftComplex)*NX*NY*NZ);
cudaMalloc((void**)&data2, sizeof(cufftComplex)*NX*NY*NZ);

/* Create a 3D FFT plan. */
cufftPlan3d(&plan, NX, NY, NZ, CUFFT_C2C);

/* Transform the first signal in place. */
cufftExecC2C(plan, data1, data1, CUFFT_FORWARD);

/* Transform the second signal using the same plan. */
cufftExecC2C(plan, data2, data2, CUFFT_FORWARD);

/* Destroy the CUFFT plan. */
cufftDestroy(plan);
cudaFree(data1); cudaFree(data2);
```