# CUFFT LIBRARY

v5.0 | October 2012

# TABLE OF CONTENTS

# 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 quickly leverage the floating-point power and parallelism of the GPU in a highly optimized and tested FFT library.

The CUFFT Library aims to support a wide range of FFT inputs and options efficiently on NVIDIA GPUs. This version of the CUFFT library supports the following features:

- Algorithms highly optimized for input sizes that can be written in the form $2^a \cdot 3^b \cdot 5^c \cdot 7^d$
- An $O(n \log n)$ algorithm for every input data size
- Complex and real-valued input and output:
  - C2C - Complex input to complex output
  - R2C - Real input to complex output
  - C2R - Symmetric complex input to real output
- 1D, 2D, and 3D transforms
- Batch execution for multiple transforms of any dimension
- Single-precision (32-bit floating point) and double-precision (64-bit floating point)
- In-place and out-of-place transforms
- FFTW compatible data layouts
- Arbitrary intra- and inter-dimension element strides (strided layout)
- Streamed execution, enabling asynchronous computation and data movement
- Transform sizes up to 128 million elements in single precision and up to 64 million elements in double precision in any dimension, limited by the available GPU memory
- Thread-safe API that can be called from multiple independent host threads

# Chapter 2. USING THE CUFFT API

This chapter provides a general overview of the CUFFT library API. For more complete information on specific functions, see CUFFT API Reference. We recommend reading this chapter before continuing with more detailed descriptions.

The Discrete Fourier transform (DFT) maps a complex-valued vector $x_k$ (*time domain*) into its *frequency domain representation* given by:

$$X_k = \sum_{n=0}^{N-1} x_n e^{2\pi i \frac{kn}{N}}$$

where $X_k$ is a complex-valued vector of the same size. Depending on $N$, different algorithms are deployed for the best performance.

The CUFFT API is modeled after FFTW, which is one of the most popular and efficient CPU-based FFT libraries. CUFFT provides a simple configuration mechanism called a *plan* that pre-configures internal building blocks such that the execution time of the transform is as low as possible for the given configuration and the particular GPU hardware selected. 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 retains whatever state is needed to execute the plan multiple times without recalculation of the configuration. This model works well for CUFFT because different kinds of FFTs require different thread configurations and GPU resources, and the plan interface provides a simple way of reusing configurations.

Computing a number `BATCH` of one-dimensional DFTs of size `NX` using CUFFT will typically look like this:

```
#define NX 256
#define BATCH 10
...
{
    cufftHandle plan;
    cufftComplex *data;
    ...
    cudaMalloc((void**)&data, sizeof(cufftComplex)*NX*BATCH);
    cufftPlan1d(&plan, NX, CUFFT_C2C, BATCH);
    ...
    cufftExecC2C(plan, data, data, CUFFT_FORWARD);
    cudaThreadSynchronize();
    ...
```

```
    cufftDestroy(plan);
    cudaFree(data);
}
```

The 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() - Create a simple plan for a 1D/2D/3D transform respectively.

Among the plan creation functions, cufftPlanMany() allows use of 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 buffer allocations (on CPU/GPU memory) 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, `rank` is the number of dimensions of the input data (see Multidimensional transforms) 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 is 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.

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.

## 2.1 Fourier Transform Types

Apart from the general complex-to-complex (C2C) transform, CUFFT implements efficiently two other types: real-to-complex (R2C) and complex-to-real (C2R). In many practical applications the input vector is real-valued. It can be easily shown that in this case the output satisfies Hermitian symmetry ($X_k = X^*_{N-k}$, where the star denotes complex conjugation). The converse is also true: for complex-Hermitian input the inverse transform will be purely real-valued. CUFFT takes advantage of this redundancy and works only on the first half of the Hermitian vector.

Transform execution functions for single and double-precision are defined separately as:

- cufftExecC2C() / cufftExecZ2Z() - complex-to-complex transforms for single/double precision.
- cufftExecR2C() / cufftExecD2Z() - real-to-complex forward transform for single/double precision.

- cufftExecC2R() / cufftExecZ2D() - complex-to-real inverse transform for single/double precision.

Each of those functions demands different input data layout (see Data Layout for details).

## 2.2 Data Layout

In the CUFFT Library, data layout depends strictly on the configuration and the transform type. In the case of general complex-to-complex transform both the input and output data shall be a `cufftComplex`/`cufftDoubleComplex` array in single- and double-precision modes respectively. In C2R mode an input array $(x_1, x_2, \ldots, x_{\lfloor \frac{N}{2} \rfloor + 1})$ of only non-redundant complex elements is required. The output array $(X_1, X_2, \ldots, X_N)$ consists of `cufftReal`/`cufftDouble` elements in this mode. Finally, R2C demands an input array $(X_1, X_2, \ldots, X_N)$ of real values and returns an array $(x_1, x_2, \ldots, x_{\lfloor \frac{N}{2} \rfloor + 1})$ of non-redundant complex elements.

In real-to-complex and complex-to-real transforms the size of input data and the size of output data differ. For out-of-place transforms a separate array of apropriate size is created. For in-place transforms the user can specify one of two supported data layouts: `native` or `padded`. The first is used for best performance and the latter for FFTW compatibility.

In the padded layout output signals begin at the same memory addresses as the input data. In other words - input data for real-to-complex and output data for complex-to-real must be padded. In the native layout no padding is required and both input and output data is formed as arrays of adequate types and sizes.

Sizes of input/output data for all types of transforms are summarized in the table below:

Input/output data sizes

| FFT type | input data size | output data size |
|---|---|---|
| C2C | $x$ `cufftComplex` | $x$ `cufftComplex` |
| C2R | $\left\lfloor \frac{x}{2} \right\rfloor + 1$ `cufftComplex` | $x$ `cufftReal` |
| R2C* | $x$ `cufftReal` | $\left\lfloor \frac{x}{2} \right\rfloor + 1$ `cufftComplex` |

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

For an in-place real-to-complex transform where FFTW compatible output is desired, the input size must be padded to $2\left(\lfloor \frac{N}{2} \rfloor + 1\right)$ real elements. For out-of-place transforms, input and output strides match the logical transform size N and the non-redundant size $\lfloor \frac{N}{2} \rfloor + 1$, respectively.

The complex-to-real transform is implicitly inverse. For in-place complex-to-real FFTs where FFTW compatible output is selected (default padding mode, see

Parameter [Transform Directions] for details), the input stride is assumed to be $\lfloor\frac{N}{2}\rfloor+1$ `cufftComplex` elements.For out-of-place transforms, input and output strides match the logical transform non-redundant size $\lfloor\frac{N}{2}\rfloor+1$ and size N, 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 signals in a batch (see Advanced Data Layout).

## 2.2.1 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.)

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` mode disables FFTW compatibility and 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` 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 (both `CUFFT_COMPATIBILITY_FFTW_PADDING` and `CUFFT_COMPATIBILITY_FFTW_ASYMMETRIC`). Refer to the FFTW online documentation for detailed FFTW data layout specifications.

# 2.3 Multidimensional transforms

Multidimensional DFT transforms a $d$-dimensional array $x_{\mathbf{n}}$, where $\mathbf{n} = (n_1, n_2, \ldots, n_d)$ into its frequency domain array given by:

$$X_{\mathbf{k}} = \sum_{\mathbf{n}=0}^{\mathbf{N}-1} e^{-2\pi i \mathbf{k} \cdot (\mathbf{n}/\mathbf{N})} x_{\mathbf{n}}$$

where $\frac{\mathbf{n}}{\mathbf{N}} = (\frac{n_1}{N_1}, \frac{n_2}{N_2}, \ldots, \frac{n_d}{N_d})$, and the sum denotes the set of nested summations $\sum_{n_1=0}^{N_1-1} \sum_{n_2=0}^{N_2-1} \ldots \sum_{n_d=0}^{N_d-1}$ CUFFT supports one-dimensional, two-dimensional and three-dimensional transforms, which can all be called by the same cufftExec* functions (see Fourier Transform Types).

Similarly to the one-dimensional case, the frequency domain representation of real-valued input data satisfies Hermitian symmetry, defined as: $x_{(n_1, n_2, \ldots, n_d)} = x^*_{(N_1 - n_1, N_2 - n_2, \ldots, N_d - n_d)}$. C2R and R2C algorithms take advantage of this fact by operating only on half of the elements of signal array, namely on: $x_{\mathbf{n}}$ for $\mathbf{n} \in \{1, \ldots, N_1\} \times \ldots \times \{1, \ldots, N_{d-1}\} \times \{1, \ldots, \lfloor \frac{N_d}{2} \rfloor + 1\}$.

The general rules of data alignment described in Data Layout apply to higher-dimensional transforms. The following table summarizes input and output data sizes for multidimensional DFTs:

Input/output data sizes

| Dims | FFT type | input data size | output data size |
|---|---|---|---|
| 1D | C2C | $N_1$ `cufftComplex` | $N_1$ `cufftComplex` |
| | C2R | $\lfloor \frac{N_1}{2} \rfloor + 1$ `cufftComplex` | $N_1$ `cufftReal` |
| | R2C | $N_1$ `cufftReal` | $\lfloor \frac{N_1}{2} \rfloor + 1$ `cufftComplex` |
| 2D | C2C | $N_1 N_2$`cufftComplex` | $N_1 N_2$`cufftComplex` |
| | C2R | $N_1(\lfloor \frac{N_2}{2} \rfloor + 1)$ `cufftComplex` | $N_1 N_2$`cufftReal` |
| | R2C | $N_1 N_2$`cufftReal` | $N_1(\lfloor \frac{N_2}{2} \rfloor + 1)$ `cufftComplex` |
| 3D | C2C | $N_1 N_2 N_3$`cufftComplex` | $N_1 N_2 N_3$`cufftComplex` |
| | C2R | $N_1 N_2(\lfloor \frac{N_3}{2} \rfloor + 1)$ `cufftComplex` | $N_1 N_2 N_3$`cufftReal` |
| | R2C | $N_1 N_2 N_3$`cufftReal` | $N_1 N_2(\lfloor \frac{N_3}{2} \rfloor + 1)$`cufftComplex` |

For example static declaration of three-dimensional array for an output of out-of-place real-to-complex transform will look like this:

```
float odata[N1][N2][N3/2+1];
```

## 2.4 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. It can be set by calling function:

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

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 advanced parameters are to be used, then all of the advanced interface parameters must be specified correctly. Advanced parameters are defined in units of the relevant data type (`cufftReal`, `cufftDoubleReal`, `cufftComplex`, or `cufftDoubleComplex`).

Advanced layout can be percieved as an additional layer of abstraction above the access to input/output data arrays. An element of coordinates `[z][y][x]` in signal number `b` in the batch will be associated with the following addresses in the memory:

- 1D

  `input[ b * idist + x * istride ]`

  `output[ b * odist + x * ostride ]`
- 2D

  `input[ b * idist + (x * inembed[1] + y) * istride ]`

  `output[ b * odist + (x * onembed[1] + y) * ostride ]`
- 3D

  `input[ b * idist + ((x * inembed[1] + y) * inembed[2] + z) * istride ]`

  `output[ b * odist + ((x * onembed[1] + y) * onembed[2] + z) * 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 `inemebed` and `onembed` values for the corresponding dimension, that is `n[i]` ≤ `inembed[i]`, `n[i]` ≤ `onembed[i]`, where $i \in \{0, \ldots, rank - 1\}$.

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

## 2.5 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.

## 2.6 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`). The only requirement is that the output data memory intervals are disjoint.

## 2.7 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 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 $2^a \cdot 3^b \cdot 5^c \cdot 7^d$ (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, 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, representable as $2^a \cdot 3^b \cdot 5^c \cdot 7^d$), 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 representable as* $2^a \cdot 3^b \cdot 5^c \cdot 7^d$.

  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 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.

# Chapter 3. CUFFT API REFERENCE

This chapter specifies the behavior of the CUFFT library functions by describing their input/output parameters, data types, and error codes. The CUFFT library is initialized upon the first invocation of an API function, and CUFFT shuts down automatically when all user-created FFT plans are destroyed.

## 3.1 Return value cufftResult

All CUFFT Library return values (other than CUFFT_SUCCESS) indicate that the current API call failed and the user should reconfigure to correct the problem. The possible return values are defined as follows:

```
typedef enum cufftResult_t {
    CUFFT_SUCCESS        = 0, //  The CUFFT operation was successful
    CUFFT_INVALID_PLAN   = 1, //  CUFFT was passed an invalid plan handle
    CUFFT_ALLOC_FAILED   = 2, //  CUFFT failed to allocate GPU or CPU memory
    CUFFT_INVALID_TYPE   = 3, //  No longer used
    CUFFT_INVALID_VALUE  = 4, //  User specified an invalid pointer or parameter
    CUFFT_INTERNAL_ERROR = 5, //  Driver or internal CUFFT library error
    CUFFT_EXEC_FAILED    = 6, //  Failed to execute an FFT on the GPU
    CUFFT_SETUP_FAILED   = 7, //  The CUFFT library failed to initialize
    CUFFT_INVALID_SIZE   = 8, //  User specified an invalid transform size
    CUFFT_UNALIGNED_DATA = 9  //  No longer used
} cufftResult;
```

## 3.2 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`.

All arrays are assumed to be in CPU memory.

**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. If set to NULL all other advanced data layout parameters are ignored. |
| `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 signals in a batch of the input data |
| `onembed` | Pointer of size `rank` that indicates the storage dimensions of the output data in memory. If set to NULL all other advanced data layout parameters are ignored. |
| `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 signals in a batch of 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 parameters is not a supported size. |

# 3.3 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. |

# 3.4 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 precision 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. |

## 3.5 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. |

# 3.6 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` | The CUFFT library failed to initialize. |

# 3.7 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. |

## 3.8 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. Pointers to `idata` and `odata` 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 Parameter cufftType.

**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 real 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. |

## 3.9 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. and 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 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. |

## 3.10 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 |

**Output**

| | |
|---|---|
| `odata` | Contains the real-valued Fourier coefficients |

**Status Returned**

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

## 3.11 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. |

## 3.12 Parameter cufftCompatibility

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

```
typedef enum cufftCompatibility_t {
    // Compact data in native format (highest performance)
```

```
    CUFFT_COMPATIBILITY_NATIVE          = 0,

    // FFTW-compatible alignment (the default value)
    CUFFT_COMPATIBILITY_FFTW_PADDING    = 1,

    // Waives the C2R symmetry requirement input
    CUFFT_COMPATIBILITY_FFTW_ASYMMETRIC = 2,

    CUFFT_COMPATIBILITY_FFTW_ALL = CUFFT_COMPATIBILITY_FFTW_PADDING |
 CUFFT_COMPATIBILITY_FFTW_ASYMMETRIC
} cufftCompatibility;
```

# 3.13 CUFFT Types

## 3.13.1 Parameter cufftType

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;
```

## 3.13.2 Parameter [Transform Directions]

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

```
    #define CUFFTFORWARD -1
    #define CUFFTINVERSE 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.

## 3.13.3 Other CUFFT Types

### 3.13.3.1 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;
```

### 3.13.3.2 cufftReal

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

```
typedef float cufftReal;
```

### 3.13.3.3 cufftDoubleReal

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

```
typedef double cufftDoubleReal;
```

### 3.13.3.4 cufftComplex

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

```
typedef cuComplex cufftComplex;
```

### 3.13.3.5 cufftDoubleComplex

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

```
typedef cuDoubleComplex cufftDoubleComplex;
```

# Chapter 4.
# 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.

## 4.1 1D Complex-to-Complex Transforms

In this example a one-dimensional complex-to-complex transform is applied to the input data. Afterwards an inverse transform is performed on the computed frequency domain representation.

```
#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;
}

if (cufftPlan1d(&plan, NX, CUFFT_C2C, BATCH) != CUFFT_SUCCESS){
 fprintf(stderr, "CUFFT error: Plan creation failed");
 return;
}

...

/* Note:
 *  Identical pointers to input and output arrays implies in-place
 transformation
 */

if (cufftExecC2C(plan, data, data, CUFFT_FORWARD) != CUFFT_SUCCESS){
 fprintf(stderr, "CUFFT error: ExecC2C Forward failed");
 return;
}

if (cufftExecC2C(plan, data, data, CUFFT_INVERSE) != CUFFT_SUCCESS){
 fprintf(stderr, "CUFFT error: ExecC2C Inverse failed");
 return;
}
```

```
/*
 *  Divide by number of elements in data set to get back original data
 */

...

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

...

cufftDestroy(plan);
cudaFree(data);
```

## 4.2 1D Real-to-Complex Transforms

In this example a one-dimensional real-to-complex transform is applied to the input data.

```
#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;
}

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;
}

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

...

cufftDestroy(plan);
cudaFree(data);
```

## 4.3 2D Complex-to-Real Transforms

In this example a two-dimensional complex-to-real transform is applied to the input data arranged according to the requirements of the native compatibility mode.

```
#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;
}

...

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;
}

...

cufftDestroy(plan);
cudaFree(data);
```

## 4.4 3D Complex-to-Complex Transforms

In this example a three-dimensional complex-to-complex transform is applied to the input data.

```
#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;
}

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

...

cufftDestroy(plan);
cudaFree(data);
```

## 4.5 2D Advanced Data Layout Use

In this example a two-dimensional complex-to-complex transform is applied to the input data arranged according to the requirements the advanced layout.

```
#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;
}

...

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 of 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 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