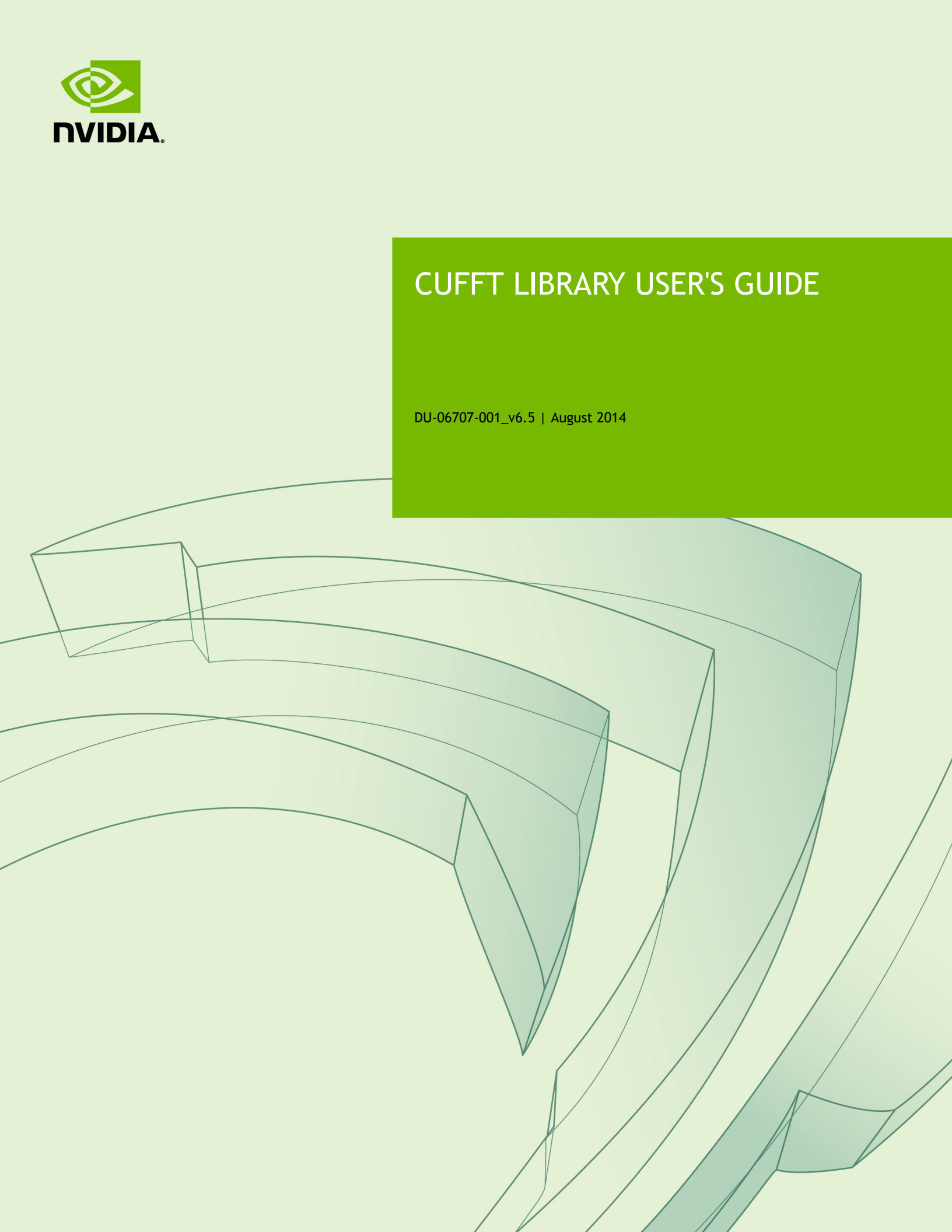

CUFFT LIBRARY USER'S GUIDE

DU-06707-001_v6.5 | August 2014

TABLE OF CONTENTS

Chapter 1. INTRODUCTION

This document describes cuFFT, the NVIDIA® CUDA™ Fast Fourier Transform (FFT) product. It consists of two separate libraries: cuFFT and cuFFTW. The cuFFT library is designed to provide high performance on NVIDIA GPUs. The cuFFTW library is provided as a porting tool to enable users of FFTW to start using NVIDIA GPUs with a minimum amount of effort.

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 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 product supports 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 $2^a \times 3^b \times 5^c \times 7^d$. In general the smaller the prime factor, the better the performance, i.e., powers of two are fastest.
- An $O(n \log n)$ algorithm for every input data size
- Single-precision (32-bit floating point) and double-precision (64-bit floating point). Single-precision transforms have higher performance than double-precision transforms.
- Complex and real-valued input and output. Real valued input or output require less computations and data than complex values and often have faster time to solution. Types supported are:
  - C2C - Complex input to complex output
  - R2C - Real input to complex output
  - C2R - Symmetric complex input to real output
- 1D, 2D and 3D transforms
- Execution of multiple 1D, 2D and 3D transforms simultaneously. These batched transforms have higher performance than single transforms.
- In-place and out-of-place transforms
- Arbitrary intra- and inter-dimension element strides (strided layout)

- FFTW compatible data layouts
- Execution of transforms across two GPUs
- Streamed execution, enabling asynchronous computation and data movement
- Transform sizes up to 512 million elements in single precision and up to half that in double precision in any dimension, limited by the type of transform chosen and the available GPU memory

The cuFFTW library provides the FFTW3 API to facilitate porting of existing FFTW applications.

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. Users are encouraged to read 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. This is known as a *forward* DFT. If the sign on the exponent of e is changed to be positive, the transform is an *inverse* transform. 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 uses internal building blocks to optimize the transform 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
#define RANK 1
...
{
    cufftHandle plan;
    cufftComplex *data;
    ...
    cudaMalloc((void**)&data, sizeof(cufftComplex)*NX*BATCH);
    cufftPlanMany(&plan, RANK, NX, &iembed, istride, idist,
        &oembed, ostride, odist, CUFFT_C2C, BATCH);
    ...
    cufftExecC2C(plan, data, data, CUFFT_FORWARD);
    cudaDeviceSynchronize();
    ...
    cufftDestroy(plan);
    cudaFree(data);
}
```

2.1. Accessing cuFFT

The cuFFT and cuFFTW libraries are available as shared libraries. They consist of compiled programs ready for users to incorporate into applications with the compiler and linker. cuFFT can be downloaded from http://developer.nvidia.com/cufft. By selecting **Download CUDA Production Release** users are all able to install the package containing the CUDA Toolkit, SDK code samples and development drivers. The CUDA Toolkit contains cuFFT and the samples include **simplecuFFT**.

The Linux release for **simplecuFFT** assumes that the root install directory is **/usr/local/cuda** and that the locations of the products are contained there as follows. Modify the Makefile as appropriate for your system.

Product	Location and name	Include file
nvcc compiler	**/bin/nvcc**	
cuFFT library	**{lib, lib64}/libcufft.so**	**inc/cufft.h**
cuFFT library with Xt functionality	**{lib, lib64}/libcufft.so**	**inc/cufftXt.h**
cuFFTW library	**{lib, lib64}/libcufftw.so**	**inc/cufftw.h**

The most common case is for developers to modify an existing CUDA routine (for example, **filename.cu**) to call cuFFT routines. In this case the include file **cufft.h** or **cufftXt.h** should be inserted into **filename.cu** file and the library included in the link line. A single compile and link line might appear as

- **/usr/local/cuda/bin/nvcc [options] filename.cu … -I/usr/local/cuda/inc -L/usr/local/cuda/lib -lcufft**

Of course there will typically be many compile lines and the compiler **g++** may be used for linking so long as the library path is set correctly.

Users of the FFTW interface (see FFTW Interface to cuFFT) should include **cufftw.h** and link with both cuFFT and cuFFTW libraries.

For the best performance input data should reside in device memory. Therefore programs in the cuFFT library assume that the data is in GPU memory. For example, if one of the execution functions is called with data in host memory, the program will return **CUFFT_EXEC_FAILED**. Programs in the cuFFTW library assume that the input data is in host memory since this library is a porting tool for users of FFTW. If the data resides in GPU memory, the program will abort.

2.2. Fourier Transform Setup

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

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

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 such as **cufftExecC2C()** (see Parameter cufftType) 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.

2.3. 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).

2.4. 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 appropriate size is created. For in-place transforms the user can specify one of two supported data layouts: **`padded`** or **`native`** (deprecated). The default is **`padded`** for FFTW compatibility.

In the **`padded`** layout output signals begin at the same memory addresses as the input data. Therefore input data for real-to-complex and output data for complex-to-real must be padded. In the **`native`** (deprecated) layout no padding is required and both input and output data are formed as arrays of adequate types and sizes.

Expected sizes of input/output data for 1-d transforms are summarized in the table below:

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

The real-to-complex transform is implicitly a forward transform. For an in-place real-to-complex transform where FFTW compatible output is desired, the input size must be padded to $\left(\lfloor\frac{N}{2}\rfloor+1\right)$ complex elements. For out-of-place transforms, input and output sizes 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), the input size is assumed to be $\lfloor\frac{N}{2}\rfloor+1$ **cufftComplex** elements. Note that in-place complex-to-real FFTs may overwrite arbitrary imaginary input point values when non-unit input and output strides are chosen. For out-of-place transforms, input and output sizes match the logical transform non-redundant size $\lfloor\frac{N}{2}\rfloor+1$ and size N, respectively.

2.4.1. 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()** (deprecated). 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.

The FFTW compatibility modes are as follows:

CUFFT_COMPATIBILITY_NATIVE mode disables FFTW compatibility and packs data most compactly.(this mode has been deprecated)

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.

The default mode is **CUFFT_COMPATIBILITY_FFTW_PADDING**

2.5. Multidimensional Transforms

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

$$X_{\mathbf{k}} = \sum_{n=0}^{N-1} x_{\mathbf{n}} e^{-2\pi i \frac{\mathbf{kn}}{\mathbf{N}}}$$

where $\frac{\mathbf{n}}{\mathbf{N}} = (\frac{n_1}{N_1}, \frac{n_2}{N_2}, \ldots, \frac{n_d}{N_d})$, and the summation denotes the set of nested summations

$$\sum_{n_1=0}^{N_1-1} \sum_{n_2=0}^{N_2-1} \cdots \sum_{n_d=0}^{N_d-1}$$

cuFFT supports one-dimensional, two-dimensional and three-dimensional transforms, which can all be called by the same **`cufftExec*`** functions (see Fourier Transform Types).

Similar to the one-dimensional case, the frequency domain representation of real-valued input data satisfies Hermitian symmetry, defined as: $x_{(n_1,n_2,..,n_d)} = x^*_{(N_1-n_1,N_2-n_2,..,N_d-n_d)}$.

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:

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 a three-dimensional array for the output of an out-of-place real-to-complex transform will look like this:

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

2.6. 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);
```

Passing **inembed** or **onembed** set to **NULL** is a special case and is equivalent to passing **n** for each. This is same as the basic data layout and other advanced parameters such as **istride** are ignored.

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 perceived 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 single 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 single 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*]** ≤ **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.

2.7. 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. Note that many plan executions require multiple kernel launches.

cufftSetStream() returns an error in the multiple GPU case as multiple GPU plans perform operations in their own streams.

2.8. Multiple GPU cuFFT Transforms

cuFFT supports using two GPUs connected to a CPU to perform Fourier Transforms whose calculations are distributed across the GPUs. An API has been defined to allow users to write new code or modify existing code to use this functionality.

Some existing functions such as the creation of a plan using **cufftCreate()** also apply in the two GPU case. Multiple GPU unique routines contain **Xt** in their name.

The memory on the GPUs is managed by helper functions **cufftXtMalloc()/ cufftXtFree()** and **cufftXtMemcpy()** using the **cudaLibXtDesc** descriptor.

Performance is a function of the bandwidth between the GPUs, the computational ability of the individual GPUs, and the type and number of FFT to be performed. The fastest performance is obtained using PCI Express 3.0 between the GPUs and ensuring that both GPUs are on the same switch. Note that two GPU execution is not guaranteed to solve a given size problem in a shorter time than single GPU execution.

The multiple GPU extensions to cuFFT are built on the extensible cuFFT API. The general steps in defining and executing a transform with this API are:

- **cufftCreate()** - create an empty plan, as in the single GPU case
- **cufftXtSetGPUs()** - define which GPUs are to be used
- Optional: **cufftEstimate{1d,2d,3d,Many}()** - estimate the sizes of the work areas required. These are the same functions used in the single GPU case although the definition of the argument **workSize** reflects the number of GPUs used.
- **cufftMakePlan{1d,2d,3d,Many}()** - create the plan. These are the same functions used in the single GPU case although the definition of the argument **workSize** reflects the number of GPUs used.
- Optional: **cufftGetSize{1d,2d,3d,Many}()** - refined estimate of the sizes of the work areas required. These are the same functions used in the single GPU case

although the definition of the argument **workSize** reflects the number of GPUs used.

- Optional: **cufftGetSize()** - check workspace size. This is the same function used in the single GPU case although the definition of the argument **workSize** reflects the number of GPUs used.
- Optional: **cufftXtSetWorkArea()** - do your own workspace allocation.
- **cufftXtMalloc()** - allocate descriptor and data on the GPUs
- **cufftXtMemcpy()** - copy data to the GPUs
- **cufftXtExecDescriptorC2C()/cufftXtExecDescriptorZ2Z()** - execute the plan
- **cufftXtMemcpy()** - copy data from the GPUs
- **cufftXtFree()** - free any memory allocated with **cufftXtMalloc()**
- **cufftDestroy()** - free cuFFT plan resources

2.8.1. Plan Specification and Work Areas

In the single GPU case a plan is created by a call to **cufftCreate()** followed by a call to **cufftMakePlan*()**. For two GPUs, the GPUs to use for execution are identified by a call to **cufftXtSetGPUs()** and this must occur after the call to **cufftCreate()** and prior to the call to **cufftMakePlan*()**.

Note that when **cufftMakePlan*()** is called for a single GPU, the work area is on that GPU. In a two GPU plan, the returned work area has two entries; one value per GPU. That is **workSize** points to a **size_t** array, one entry per GPU. Also the strides and batches apply to the entire plan across all GPUs associated with the plan.

Once a plan is locked by a call to **cufftMakePlan*()**, different descriptors may be specified in calls to **cufftXtExecDescriptor*()** to execute the plan on different data sets, but the new descriptors must use the same GPUs in the same order.

As in the single GPU case, **cufftEstimateSize{Many,1d,2d,3d}()** and **cufftGetSize{Many,1d,2d,3d}()** give estimates of the work area sizes required for a two GPU plan and in this case **workSize** points to a **size_t** array, one entry per GPU.

Similarly the actual work size returned by **cufftGetSize()** is a **size_t** array, one entry per GPU in the two GPU case.

2.8.2. Helper Functions

Two GPU cuFFT execution functions assume a certain data layout in terms of what input data has been copied to which GPUs prior to execution, and what output data resides in which GPUs post execution. cuFFT provides functions to assist users in manipulating data on two GPUs. These must be called after the call to **cufftMakePlan*()**.

On a single GPU users may call **cudaMalloc()** and **cudaFree()** to allocate and free GPU memory. To provide similar functionality in the two GPU case, cuFFT includes **cufftXtMalloc()** and **cufftXtFree()** functions. The function **cufftXtMalloc()** returns a descriptor which specifies the location of these memories.

On a single GPU users may call **cudaMemcpy()** to transfer data between host and GPU memory. To provide similar functionality in the two GPU case, cuFFT includes

cufftXtMemcpy() which allows users to copy between host and two GPU memories or even between the GPU memories.

All single GPU cuFFT FFTs return output the data in natural order, that is the ordering of the result is the same as if a DFT had been performed on the data. Some Fast Fourier Transforms produce intermediate results where the data is left in a permutation of the natural output. When batch is one, data is left in the GPU memory in a permutation of the natural output.

When **cufftXtMemcpy()** is used to copy data from GPU memory back to host memory, the results are in natural order regardless of whether the data on the GPUs is in natural order or permuted. Using **CUFFT_COPY_DEVICE_TO_DEVICE** allows users to copy data from the permuted data format produced after a single transform to the natural order on GPUs.

2.8.3. Multiple GPU 2D and 3D Transforms on Permuted Input

For single 2D or 3D transforms on two GPUs, when **cufftXtMemcpy()** distributes the data to the GPUs, the array is divided on the X axis. I.E. half of the X dimenson points, for all Y (and Z) values, are copied to each of the GPUs. When the transform is computed, the data are permuted such that they are divided on the Y axis. I.E. half of the Y dimension points, for all X (and Z) values are on each of the GPUs.

When cuFFT creates a 2D or 3D plan for a single transform on two GPUs, it actually creates two plans. One plan expects input to be divided on the X axis. The other plan expects data to be divided on the Y axis. This is done because many algorithms compute a forward FFT, then perform some point-wise operation on the result, and then compute the inverse FFT. A memory copy to restore the data to the original order would be expensive. To avoid this, **cufftXtMemcpy** and **cufftXtExecDescriptor()** keep track of the data ordering so that the correct operation is used.

The ability of cuFFT to process data in either order makes the following sequence possible.

- **cufftCreate()** - create an empty plan, as in the single GPU case
- **cufftXtSetGPUs()** - define which GPUs are to be used
- **cufftMakePlan{1d,2d,3d,Many}()** - create the plan.
- **cufftXtMalloc()** - allocate descriptor and data on the GPUs
- **cufftXtMemcpy()** - copy data to the GPUs
- **cufftXtExecDescriptorC2C()/cufftXtExecDescriptorZ2Z()** - compute the forward FFT
- **userFunction()** - modify the data in the frequency domain
- **cufftXtExecDescriptorC2C()/cufftXtExecDescriptorZ2Z()** - compute the inverse FFT
- Note that it was not necessary to copy/permute the data between execute calls
- **cufftXtMemcpy()** - copy data to the host
- **cufftXtFree()** - free any memory allocated with **cufftXtMalloc()**
- **cufftDestroy()** - free cuFFT plan resources

2.8.4. Supported Functionality

In Version 6.5 only a subset of single GPU functionality is supported for two GPU execution.

Supported functionality:

- Two GPUs are supported.
- The GPUs must both be on the same GPU board, such as a Tesla K10 or GeForce GTX690.
- The GPUs must support the Unified Virtual Address Space.
- On Windows, the GPU board must be operating in Tesla Compute Cluster (TCC) mode.
- Running cuFFT on multiple GPUs is not compatible with an application that uses the CUDA Driver API.
- Strided input and output are not supported.
- When the number of batches is 1:
  - Only `C2C` and `Z2Z` transform types are supported.
  - Only in-place transforms are supported.
  - The transform dimensions must be powers of 2.
  - The size of the transform must be greater than or equal to 32.

General guidelines are:

- The data for the entire transform must fit within the memory of the GPUs assigned to it.
- When the number of batches is 1 or evenly divisible by the number of GPUs:
  - Each GPU processes the same amount of data.
  - Each GPU executes the same number of operations.
- Otherwise:
  - GPU 0 executes the first $\left\lceil \frac{\text{batches} + 1}{2} \right\rceil$ transforms.
  - GPU 1 executes the remaining $\left\lceil \frac{\text{batches}}{2} \right\rceil$ transforms.

Batch size output differences:

Single GPU cuFFT results are always returned in natural order. When two GPUs are used to perform more than one transform, the results are also returned in natural order. When two GPUs are used to perform a single transform the results are returned in a permutation of the normal results to reduce communication time.

Number of GPUs	Number of transforms	Output Order on GPUs
One	One or multiple transforms	Natural order
Two	One	Permuted results
Two	Multiple	Natural order

To produce natural order results in GPU memory in the 1D single transform case, requires calling **cufftXtMemcpy()** with **CUFFT_COPY_DEVICE_TO_DEVICE**.

2D and 3D transforms support execution of a transform given permuted order results as input. After execution in this case, the output will be in natural order. It is also possible to use **cufftXtMemcpy()** with **CUFFT_COPY_DEVICE_TO_DEVICE** to return 2D or 3D data to natural order.

See the cuFFT Code Examples section for multiple GPU examples.

2.9. cuFFT Callback Routines

Callback routines are user-supplied kernel routines that cuFFT will call when loading or storing data. They allow the user to do data pre- or post- processing without additional kernel calls.

2.9.1. Overview of the cufFFT Callback Routine Feature

cuFFT provides a set of APIs that allow the cuFFT user to provide CUDA functions that re-direct or manipulate the data as it is loaded prior to processing the FFT, or stored once the FFT has been done. For the load callback, cuFFT passes the callback routine the address of the input data and the offset to the value to be loaded from device memory, and the callback routine returns the value it wishes cuFFT to use instead. For the store callback, cufFFT passes the callback routine the value it has computed, along with the address of the output data and the offset to the value to be written to device memory, and the callback routine modifies the value and stores the modified result.

In order to provide a callback to cuFFT, a plan is created and configured normally using the extensible plan APIs. After the call to **cufftCreate** and **cufftMakePlan**, the user may associate a load callback routine, or a store callback routine, or both, with the plan, by calling**cufftXtSetCallback**. The caller also has the option to specify a device pointer to an opaque structure they wish to associate with the plan. This pointer will be passed to the callback routine by the cuFFT library. The caller may use this structure to remember plan dimensions and strides, or have a pointer to auxiliary data, etc.

With some restrictions, the callback routine is allowed to request shared memory for its own use. If the requested amount of shared memory is available, cufft will pass a pointer to it when it calls the callback routine.

CUFFT allows for 8 types of callback routine, one for each possible combination of: load or store, real or complex, single precision or double. **It is the caller's responsibility to provide a routine that matches the function prototype for the type of routine specified.** If there is already a callback of the specified type associated with the plan, the set callback function will replace it with the new one.

The callback routine extensions to cuFFT are built on the extensible cuFFT API. The general steps in defining and executing a transform with callbacks are:

- **cufftCreate()** - create an empty plan, as in the single GPU case
- **cufftMakePlan{1d,2d,3d,Many}()** - create the plan. These are the same functions used in the single GPU case.

- **`cufftSetCallback()`** - called for load and/or store callback for this plan
- **`cufftExecC2C() etc.`** - execute the plan
- **`cufftDestroy()`** - free cuFFT plan resources

Callback functions are not supported on transforms with a dimension size that does not factor into primes smaller than 127. Callback functions on plans whose dimensions' prime factors are limited to 2, 3, 5, and 7 can safely call **`__syncthreads()`**. On other plans, results are not defined.

NOTE: The callback API is available in the statically linked cuFFT library only, and only on 64 bit LINUX operating systems. Use of this API requires a current license. Free evaluation licenses are available for registered developers until 6/30/2015. To learn more please visit the cuFFT developer page.

A license key can be set up as follows:

- Download the license key file (with extension .lic) to your machine.
- Set the environment variable LM_LICENSE_FILE to point to this license key.
- If the LM_LICENSE_FILE variable is already in use, you can simply append the path to your license key file to it. Alternatively, you can use the variable NVIDIA_LICENSE_FILE in place of LM_LICENSE_FILE.
- You can rename the license key file if you choose, but please make sure the .lic extension remains.

The end date for a license key file is mentioned in clear text in the license key.

2.9.2. Specifying Load and Store Callback Routines

In order to associate a callback routine with a plan, it is necessary to obtain a device pointer to the callback routine.

As an example, if the user wants to specify a load callback for an R2C transform, they would write the device code for the callback function, and define a global device variable that contains a pointer to the function:

```
__device__  cufftReal myOwnCallback(void *dataIn,
                                    size_t offset,
                                    void *callerInfo,
                                    void *sharedPtr) {
    cufftReal ret;
    // use offset, dataIn, and optionally callerInfo to
    // compute the return value
    return ret;
}
__device__ cufftCallbackLoadR myOwnCallbackPtr = myOwnCallback;
```

From the host side, the user then has to get the address of the callback routine, which is stored in **`myOwnCallbackPtr`**. This is done with **`cudaMemcpyFromSymbol`**, as follows:

```
cufftCallbackLoadR hostCopyOfCallbackPtr;

cudaMemcpyFromSymbol(&hostCopyOfCallbackPtr,
                     myOwnCallbackPtr,
                     sizeof(hostCopyOfCallbackPtr));
```

hostCopyOfCallbackPtr then contains the device address of the callback routine, that should be passed to **cufftXtSetCallback**. Note that, for multi-GPU transforms, **hostCopyOfCallbackPtr** will need to be an array of pointers, and the **cudaMemcpyFromSymbol** will have to be invoked for each GPU.

2.9.3. Callback Routine Function Details

Below are the function prototypes, and typedefs for pointers to the user supplied callback routines that cuFFT calls to load data prior to the transform.

```
typedef  cufftComplex (*cufftCallbackLoadC)(void *dataIn,
                                            size_t offset,
                                            void *callerInfo,
                                            void *sharedPointer);

typedef  cufftDoubleComplex (*cufftCallbackLoadZ)(void *dataIn,
                                                  size_t offset,
                                                  void *callerInfo,
                                                  void *sharedPointer);

typedef  cufftReal (*cufftCallbackLoadR)(void *dataIn,
                                         size_t offset,
                                         void *callerInfo,
                                         void *sharedPointer);

typedef  cufftDoubleReal (*cufftCallbackLoadD)(void *dataIn,
                                               size_t offset,
                                               void *callerInfo,
                                               void *sharedPointer);
```

Parameters for all of the load callbacks are defined as below:

- **offset**: offset of the input element from the start of output data. This is not a byte offset, rather it is the number of elements from start of data.
- **dataIn**: device pointer to the start of the input array that was passed in the **cufftExecute** call.
- **callerInfo**: device pointer to the optional caller specified data passed in the **cufftSetCallback** call.
- **sharedPointer**: pointer to shared memory, valid only if the user has called **cufftXtSetCallbackSharedSize()**.

Below are the function prototypes, and typedefs for pointers to the user supplied callback routines that cuFFT calls to store data after completion of the transform. Note that the store callback functions do not return a value. This is because a store callback function is responsible not only for transforming the data as desired, but also for writing

the data to the desired location. This allows the store callback to rearrange the data, for example to shift the zero frequency result to the center of the ouput.

```
typedef  void (*cufftCallbackStoreC)(void *dataOut,
                                     size_t offset,
                                     cufftComplex element,
                                     void *callerInfo,
                                     void *sharedPointer);

typedef  void (*cufftCallbackStoreZ)(void *dataOut,
                                     size_t offset,
                                     cufftDoubleComplex element,
                                     void *callerInfo,
                                     void *sharedPointer);

typedef  void (*cufftCallbackStoreR)(void *dataOut,
                                     size_t offset,
                                     cufftReal element,
                                     void *callerInfo,
                                     void *sharedPointer);

typedef  void (*cufftCallbackStoreD)(void *dataOut,
                                     size_t offset,
                                     cufftDoubleReal element,
                                     void *callerInfo,
                                     void *sharedPointer);
```

Parameters for all of the store callbacks are defined as below:

- **`offset`**: offset of the output element from the start of output data. This is not a byte offset, rather it is the number of elements from start of data.
- **`dataOut`**: device pointer to the start of the output array that was passed in the **`cufftExecute`** call.
- **`element`**: the real or complex result computed by CUFFT for the element specified by the offset argument.
- **`callerInfo`**: device pointer to the optional caller specified data passed in the **`cufftSetCallback`** call.
- **`sharedPointer`**: pointer to shared memory, valid only if the user has called **`cufftXtSetCallbackSharedSize()`**.

2.9.4. Coding Considerations for the cuFFT Callback Routine Feature

cuFFT supports callbacks on all types of transforms, regardless of precision, dimension, batch, stride between elements, or number of GPUs.

cuFFT supports a wide range of parameters, and based on those for a given plan, it attempts to optimize performance. The number of kernels launched, and for each of those, the number of blocks launched and the number of threads per block, will vary depending on how cuFFT decomposes the transform. For some configurations, cuFFT will load or store (and process) multiple inputs or outputs per thread. For some configurations, threads may load or store inputs or outputs in any order, and cuFFT does not guarantee that the inputs or outputs handled by a given thread will be contiguous. These characteristics may vary with transform size, transform type (e.g.

C2C vs C2R), number of dimensions, and GPU architecture. These variations may also change from one library version to the next.

cuFFT will call the load callback routine, for each point in the input, once and only once. Similarly it will call the store callback routine, for each point in the output, once and only once. If cuFFT is implementing a given FFT in multiple phases, it will only call the load callback routine from the first phase kernel(s), and it will only call the store callback routine from the last phase kernel(s).

When cufft is using only a single kernel, both the load and store callback routines will be called from the same kernel. In this case, if the transform is being done in-place (i.e. input data and output data are in the same memory location) the store callback can not safely write outside the confines of the specified element, unless it is writing the data to a completley separate output buffer.

When more than one kernel are used to implement a transform, the thread and block structure of the first kernel (the one that does the load) is often different from the thread and block structure of the last kernel (the one that does the store)

For multi-GPU transforms, the index passed to the callback routine is the element index from the start of data *on that GPU*, not from the start of the entire input or output data array.

For transforms whose dimensions can be factored into powers of 2, 3, 5, or 7, cuFFT guarantees that it will call the load and store callback routines from points in the kernel at which the code has converged (all threads in any given block will invoke the callback routine). This allows the callback routine to invoke **`__syncthreads`** internally as needed. Of course the caller is responsible for guaranteeing that the callback routine is at a point where the callback code has converged, to avoid deadlock. For plans whose dimensions are factored into higher primes, results of a callback routine calling **`__syncthreads`** are not defined.

2.10. Thread Safety

cuFFT APIs are thread safe as long as different host threads execute FFTs using different plans and the output data are disjoint.

2.11. Static Library and Callback Support

Starting with release 6.5, the cuFFT Libraries are also delivered in a static form as libcufft_static.a and libcufftw_static.a on Linux and Mac and as cufft_static.lib. Static libraries are not supported on Windows. The static cufft and cufftw libraries depend on thread abstraction layer library libculibos.a.

For example, on Linux, to compile a small application using cuFFT, against the dynamic library, the following command can be used:

```
gcc myCufftApp.c    -lcufft  -o myCufftApp
```

For cufftw on Linux, to compile a small application against the dynamic library, the following command can be used:

```
    gcc myfftwApp.c  -lcufftw  -lcufft  -o myfftwApp
```

Whereas to compile against the static cufft library, the following command has to be used:

```
    gcc myCufftApp.c    libcufft_static.a   libculibos.a  -o myCufftApp
```

Similarly to compile against the static cufftw library, the following command has to be used:

```
    gcc myfftwApp.c    libcufftw_static.a  libcufft_static.a   libculibos.a  -o
myfftwApp
```

An application can be built with the cuFFT static library that will run on any CUDA hardware with SM20 or above. However cuFFT is not built with code for all possible SMs. If you compile your application with NVCC and specify an SM, you must specify one of the SM's cuFFT includes. On X86 platforms, these are SM20, SM30, SM35, and SM50. On Arm, cuFFT is built for SM32 only.

On 64 bit LINUX operating systems, the cuFFT static library supports user supplied callback routines. The callback routines are CUDA device code, and must be separately compiled with NVCC and linked with the cuFFT library. Refer to the NVCC documentation regarding separate compilation for details.

2.12. Accuracy and Performance

A 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 the DFT matrix as a product of sparse building block matrices. The cuFFT Library implements the following building blocks: radix-2, radix-3, radix-5, and radix-7. Hence the performance of any transform size that can be factored as $2^a \times 3^b \times 5^c \times 7^d$ (where a, b, c, and d are non-negative integers) is optimized in the cuFFT library. There are also radix-m building blocks for other primes, m, whose value is < 128. When the length cannot be decomposed as multiples of powers of primes from 2 to 127, Bluestein's algorithm is used. Since the Bluestein implementation requires more computations per output point than the Cooley-Tukey implementation, the accuracy of the Cooley-Tukey algorithm is better. 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, 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).

Applies to	Recommendation	Comment
All	Use single precision transforms.	Single precision transforms require less bandwidth per computation than double precision transforms.
All	Restrict the size along all dimensions to be representable as $2^a \times 3^b \times 5^c \times 7^d$.	The cuFFT library has highly optimized kernels for transforms whose dimensions have these prime factors. In general the best performance occurs when using powers of 2, followed by powers of 3, then 5, 7.
All	Restrict the size along each dimension to use fewer distinct prime factors.	A transform of size 2^n or 3^n will usually be faster than one of size $2^i \times 3^j$ even if the latter is slightly smaller, due to the composition of specialized paths.
All	Restrict the data to be contiguous in memory when performing a single transform. When performing multiple transforms make the individual datasets contiguous	The cuFFT library has been optimized for this data layout.
All	Perform multiple (i.e., batched) transforms.	Additional optimizations are performed in batched mode.
real-to-complex transforms or complex-to-real transforms	Ensure problem size of x dimension is a multiple of 4.	This scheme uses more efficient kernels to implement conjugate symmetry property.
real-to-complex transforms or complex-to-real transforms	Use `out-of-place` mode.	This scheme uses more efficient kernels than `in-place` mode.
Multiple GPU transforms	Use PCI Express 3.0 between GPUs and ensure the GPUs are on the same switch.	The faster the interconnect between the GPUs, the faster the performance.

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 except for **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
    CUFFT_INCOMPLETE_PARAMETER_LIST = 10, //  Missing parameters in call
    CUFFT_INVALID_DEVICE = 11, //  Execution of a plan was on different GPU than
 plan creation
    CUFFT_PARSE_ERROR    = 12, //  Internal plan database error
    CUFFT_NO_WORKSPACE   = 13  //  No workspace has been provided prior to plan
 execution
} cufftResult;
```

Users are encouraged to check return values from cuFFT functions for errors as shown in cuFFT Code Examples.

3.2. cuFFT Basic Plans

3.2.1. 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**. Deprecated - use **cufftPlanMany** for multiple transforms.

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_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** or **batch** parameter is not a supported size.

3.2.2. 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_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.

3.2.3. 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_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.

3.2.4. 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_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. cuFFT Extensible Plans

This API separates handle creation from plan generation. This makes it possible to change plan settings, which may alter the outcome of the plan generation phase, before the plan is actually generated.

3.3.1. Function cufftCreate()

```
cufftResult
    cufftCreate(cufftHandle *plan);
```

Creates only an opaque handle, and allocates small data structures on the host. The **cufftMakePlan*()** calls actually do the plan generation. It is recommended that **cufftSet*()** calls, such as **cufftSetCompatibilityMode()**, that may require a plan to be broken down and re-generated, should be made after **cufftCreate()** and before one of the **cufftMakePlan*()** calls.

Input

plan	Pointer to a **cufftHandle** object

Output

plan	Contains a cuFFT plan handle value

Return Values

CUFFT_SUCCESS	cuFFT successfully created the FFT plan.
CUFFT_ALLOC_FAILED	The allocation of resources for the plan failed.
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.

3.3.2. Function cufftMakePlan1d()

```
cufftResult
    cufftMakePlan1d(cufftHandle plan, int nx, cufftType type, int batch,
        size_t *workSize);
```

Following a call to **cufftCreate()** makes 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.

If **cufftXtSetGPUs()** was called prior to this call with two GPUs, then **workSize** will contain two sizes. See sections on multiple GPUs for more details.

Input

plan	**cufftHandle** returned by **cufftCreate**

nx	The transform size (e.g. 256 for a 256-point FFT). For 2 GPUs, this must be a power of 2.
type	The transform data type (e.g., **CUFFT_C2C** for single precision complex to complex). For 2 GPUs this must be a complex to complex transform.
batch	Number of transforms of size **nx**. Deprecated - use **cufftMakePlanMany** for multiple transforms.
***workSize**	Pointer to the size(s) of the work areas. For 2 GPUs worksize must be declared to have two elements.

Output

***workSize**	Pointer to the size(s) of the work areas.

Return Values

CUFFT_SUCCESS	cuFFT successfully created the FFT plan.
CUFFT_INVALID_PLAN	The **plan** parameter is not a valid handle.
CUFFT_ALLOC_FAILED	The allocation of GPU resources for the plan failed.
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** or **batch** parameter is not a supported size.

3.3.3. Function cufftMakePlan2d()

```
cufftResult
    cufftMakePlan2d(cufftHandle plan, int nx, int ny, cufftType type,
        size_t *workSize);
```

Following a call to **cufftCreate()** makes a 2D FFT plan configuration according to specified signal sizes and data type.

If **cufftXtSetGPUs()** was called prior to this call with two GPUs, then **workSize** will contain two sizes. See sections on multiple GPUs for more details.

Input

plan	**cufftHandle** returned by **cufftCreate**
nx	The transform size in the *x* dimension (number of rows). For 2 GPUs, this must be a power of 2.
ny	The transform size in the *y* dimension (number of columns). For 2 GPUs, this must be a power of 2.
type	The transform data type (e.g., **CUFFT_C2R** for single precision complex to real). For 2 GPUs this must be a complex to complex transform.
***workSize**	Pointer to the size(s) of the work areas. For 2 GPUs worksize must be declared to have two elements.

Output

***workSize**	Pointer to the size(s) of the work areas.

Return Values

CUFFT_SUCCESS	cuFFT successfully created the FFT plan.
CUFFT_INVALID_PLAN	The **plan** parameter is not a valid handle.
CUFFT_ALLOC_FAILED	The allocation of GPU resources for the plan failed.
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.

3.3.4. Function cufftMakePlan3d()

```
cufftResult
    cufftMakePlan3d(cufftHandle plan, int nx, int ny, int nz, cufftType type,
        size_t *workSize);
```

Following a call to **cufftCreate()** makes 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**.

If **cufftXtSetGPUs()** was called prior to this call with two GPUs, then **workSize** will contain two sizes. See sections on multiple GPUs for more details.

Input

plan	**cufftHandle** returned by **cufftCreate**
nx	The transform size in the *x* dimension. For 2 GPUs, this must be a power of 2.
ny	The transform size in the *y* dimension. For 2 GPUs, this must be a power of 2.
nz	The transform size in the *z* dimension. For 2 GPUs, this must be a power of 2.
type	The transform data type (e.g., **CUFFT_R2C** for single precision real to complex). For 2 GPUs this must be a complex to complex transform.
***workSize**	Pointer to the size(s) of the work areas. For 2 GPUs worksize must be declared to have two elements.

Output

***workSize**	Pointer to the size(s) of the work area(s).

Return Values

CUFFT_SUCCESS	cuFFT successfully created the FFT plan.

CUFFT_INVALID_PLAN	The **plan** parameter is not a valid handle.
CUFFT_ALLOC_FAILED	The allocation of GPU resources for the plan failed.
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.

3.3.5. Function cufftMakePlanMany()

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

Following a call to **cufftCreate()** makes 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**.

If **cufftXtSetGPUs()** was called prior to this call with two GPUs, then **workSize** will contain two sizes. See sections on multiple GPUs for more details.

All arrays are assumed to be in CPU memory.

Input

plan	**cufftHandle** returned by **cufftCreate**
rank	Dimensionality of the transform (1, 2, or 3)
n	Array of size **rank**, describing the size of each dimension. For 2 GPUs, the sizes must be a power of 2.
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). For 2 GPUs this must be a complex to complex transform.
batch	Batch size for this transform
***workSize**	Pointer to the size(s) of the work areas. For 2 GPUs worksize must be declared to have two elements.

Output

***workSize**	Pointer to the size(s) of the work areas.

Return Values

CUFFT_SUCCESS	cuFFT successfully created the FFT plan.
CUFFT_INVALID_PLAN	The **plan** parameter is not a valid handle.
CUFFT_ALLOC_FAILED	The allocation of GPU resources for the plan failed.
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.

3.4. cuFFT Estimated Size of Work Area

During plan execution, cuFFT requires a work area for temporary storage of intermediate results. The **cufftEstimate*()** calls return an estimate for the size of the work area required, given the specified parameters, and assuming default plan settings. Some problem sizes require much more storage than others. In particular powers of 2 are very efficient in terms of temporary storage. Large prime numbers, however, use different algorithms and may need up to the eight times that of a similarly sized power of 2. These routines return estimated **workSize** values which may still be smaller than the actual values needed especially for values of **n** that are not multiples of powers of 2, 3, 5 and 7. More refined values are given by the **cufftGetSize*()** routines, but these values may still be conservative.

3.4.1. Function cufftEstimate1d()

```
cufftResult
    cufftEstimate1d(int nx, cufftType type, int batch, size_t *workSize);
```

During plan execution, cuFFT requires a work area for temporary storage of intermediate results. This call returns an estimate for the size of the work area required, given the specified parameters, and assuming default plan settings. Note that changing some plan settings, such as compatibility mode, may alter the size required for the work area.

Input

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**. Deprecated - use **cufftEstimateMany** for multiple transforms.
***workSize**	Pointer to the size of the work space.

Output

***workSize**	Pointer to the size of the work space

Return Values

CUFFT_SUCCESS	cuFFT successfully returned the size of the work space.
CUFFT_ALLOC_FAILED	The allocation of GPU resources for the plan failed.
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.

3.4.2. Function cufftEstimate2d()

```
cufftResult
    cufftEstimate2d(int nx, int ny, cufftType type, size_t *workSize);
```

During plan execution, cuFFT requires a work area for temporary storage of intermediate results. This call returns an estimate for the size of the work area required, given the specified parameters, and assuming default plan settings. Note that changing some plan settings, such as compatibility mode, may alter the size required for the work area.

Input

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)
***workSize**	Pointer to the size of the work space.

Output

***workSize**	Pointer to the size of the work space

Return Values

CUFFT_SUCCESS	cuFFT successfully returned the size of the work space.
CUFFT_ALLOC_FAILED	The allocation of GPU resources for the plan failed.

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.

3.4.3. Function cufftEstimate3d()

```
cufftResult
    cufftEstimate3d(int nx, int ny, int nz, cufftType type, size_t *workSize);
```

During plan execution, cuFFT requires a work area for temporary storage of intermediate results. This call returns an estimate for the size of the work area required, given the specified parameters, and assuming default plan settings. Note that changing some plan settings, such as compatibility mode, may alter the size required for the work area.

Input

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)
***workSize**	Pointer to the size of the work space.

Output

***workSize**	Pointer to the size of the work space

Return Values

CUFFT_SUCCESS	cuFFT successfully returned the size of the work space.
CUFFT_ALLOC_FAILED	The allocation of GPU resources for the plan failed.
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.

3.4.4. Function cufftEstimateMany()

```
cufftResult
    cufftEstimateMany(int rank, int *n, int *inembed,
        int istride, int idist, int *onembed, int ostride,
        int odist, cufftType type, int batch, size_t *workSize);
```

During plan execution, cuFFT requires a work area for temporary storage of intermediate results. This call returns an estimate for the size of the work area required, given the specified parameters, and assuming default plan settings. Note that changing some plan settings, such as compatibility mode, may alter the size required for the work area.

The **cufftEstimateMany()** 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

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
***workSize**	Pointer to the size of the work space.

Output

***workSize**	Pointer to the size of the work space

Return Values

CUFFT_SUCCESS	cuFFT successfully returned the size of the work space.
CUFFT_ALLOC_FAILED	The allocation of GPU resources for the plan failed.
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.

3.5. cuFFT Refined Estimated Size of Work Area

The **cufftGetSize*()** routines give a more accurate estimate of the work area size required for a plan than the **cufftEstimate*()** routines as they take into account any plan settings that may have been made. As discussed in the section cuFFT Estimated Size of Work Area, the **workSize** value(s) returned may be conservative especially for values of **n** that are not multiples of powers of 2, 3, 5 and 7.

3.5.1. Function cufftGetSize1d()

```
cufftResult
    cufftGetSize1d(cufftHandle plan, int nx, cufftType type, int batch,
        size_t *workSize);
```

This call gives a more accurate estimate of the work area size required for a plan than **cufftEstimate1d()**, given the specified parameters, and taking into account any plan settings that may have been made.

Input

plan	**cufftHandle** returned by **cufftCreate**
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**. Deprecated - use **cufftGetSizeMany** for multiple transforms.
***workSize**	Pointer to the size of the work space. For 2 GPUs worksize must be declared to have two elements.

Output

***workSize**	Pointer to the size of the work space

Return Values

CUFFT_SUCCESS	cuFFT successfully returned the size of the work space.
CUFFT_INVALID_PLAN	The **plan** parameter is not a valid handle.
CUFFT_ALLOC_FAILED	The allocation of GPU resources for the plan failed.
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.

3.5.2. Function cufftGetSize2d()

```
cufftResult
    cufftGetSize2d(cufftHandle plan, int nx, int ny, cufftType type,
        size_t *workSize);
```

This call gives a more accurate estimate of the work area size required for a plan than **cufftEstimate2d()**, given the specified parameters, and taking into account any plan settings that may have been made.

Input

plan	**cufftHandle** returned by **cufftCreate**
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)
***workSize**	Pointer to the size of the work space. For 2 GPUs worksize must be declared to have two elements.

Output

***workSize**	Pointer to the size of the work space

Return Values

CUFFT_SUCCESS	cuFFT successfully returned the size of the work space.
CUFFT_INVALID_PLAN	The **plan** parameter is not a valid handle.
CUFFT_ALLOC_FAILED	The allocation of GPU resources for the plan failed.
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.

3.5.3. Function cufftGetSize3d()

```
cufftResult
    cufftGetSize3d(cufftHandle plan, int nx, int ny, int nz, cufftType type,
        size_t *workSize);
```

This call gives a more accurate estimate of the work area size required for a plan than **cufftEstimate3d()**, given the specified parameters, and taking into account any plan settings that may have been made.

Input

plan	**cufftHandle** returned by **cufftCreate**
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)
***workSize**	Pointer to the size of the work space. For 2 GPUs worksize must be declared to have two elements.

Output

***workSize**	Pointer to the size of the work space.

Return Values

CUFFT_SUCCESS	cuFFT successfully returned the size of the work space.
CUFFT_INVALID_PLAN	The **plan** parameter is not a valid handle.
CUFFT_ALLOC_FAILED	The allocation of GPU resources for the plan failed.
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.

3.5.4. Function cufftGetSizeMany()

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

This call gives a more accurate estimate of the work area size required for a plan than **cufftEstimateSizeMany()**, given the specified parameters, and taking into account any plan settings that may have been made.

Input

plan	**cufftHandle** returned by **cufftCreate**
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
***workSize**	Pointer to the size of the work space. For 2 GPUs worksize must be declared to have two elements.

Output

***workSize**	Pointer to the size of the work area

Return Values

CUFFT_SUCCESS	cuFFT successfully returned the size of the work space.
CUFFT_INVALID_PLAN	The **plan** parameter is not a valid handle.
CUFFT_ALLOC_FAILED	The allocation of GPU resources for the plan failed.
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.

3.6. Function cufftGetSize()

```
cufftResult
    cufftGetSize(cufftHandle plan, size_t *workSize);
```

Once plan generation has been done, either with the original API or the extensible API, this call returns the actual size of the work area required to support the plan. Callers who choose to manage work area allocation within their application must use this call after plan generation, and after any **cufftSet*()** calls subsequent to plan generation, if those calls might alter the required work space size.

Input

plan	**cufftHandle** returned by **cufftCreate**
***workSize**	Pointer to the size of the work space. For 2 GPUs worksize must be declared to have two elements.

Output

***workSize**	Pointer to the size of the work space

Return Values

CUFFT_SUCCESS	cuFFT successfully returned the size of the work space.
CUFFT_INVALID_PLAN	The **plan** parameter is not a valid handle.
CUFFT_INTERNAL_ERROR	An internal driver error was detected.

3.7. cuFFT Caller Allocated Work Area Support

3.7.1. Function cufftSetAutoAllocation()

```
cufftResult
    cufftSetAutoAllocation(cufftHandle plan, int autoAllocate);
```

cufftSetAutoAllocation() indicates that the caller intends to allocate and manage work areas for plans that have been generated. cuFFT default behavior is to allocate the work area at plan generation time. If **cufftSetAutoAllocation()** has been called with autoAllocate set to 0 ("false") prior to one of the **cufftMakePlan*()** calls, cuFFT does not allocate the work area. This is the preferred sequence for callers wishing to manage work area allocation.

Input

plan	**cufftHandle** returned by **cufftCreate**.
autoAllocate	Indicates whether to allocate work area.

Return Values

CUFFT_SUCCESS	cuFFT successfully allows user to manage work area.
CUFFT_INVALID_PLAN	The **plan** parameter is not a valid handle.
CUFFT_INTERNAL_ERROR	An internal driver error was detected.

3.7.2. Function cufftSetWorkArea()

```
cufftResult
    cufftSetWorkArea(cufftHandle plan, void *workArea);
```

cufftSetWorkArea() overrides the work area pointer associated with a plan. If the work area was auto-allocated, cuFFT frees the auto-allocated space. The **cufftExecute*()** calls assume that the work area pointer is valid and that it points to a contiguous region in device memory that does not overlap with any other work area. If this is not the case, results are indeterminate.

Input

plan	**cufftHandle** returned by **cufftCreate**
workArea	Pointer to workArea. For two GPUs, two work area pointers must be given.

Return Values

CUFFT_SUCCESS	cuFFT successfully allows user to override workArea pointer.

CUFFT_INVALID_PLAN	The **plan** parameter is not a valid handle.
CUFFT_INTERNAL_ERROR	An internal driver error was detected.
CUFFT_SETUP_FAILED	The cuFFT library failed to initialize.

3.8. 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.

3.9. cuFFT Execution

3.9.1. Functions cufftExecC2C() and 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	**cufftHandle** returned by **cufftCreate**
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 executed 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.

3.9.2. Functions cufftExecR2C() and 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	**cufftHandle** returned by **cufftCreate**
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 executed 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.

3.9.3. Functions cufftExecC2R() and 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	**cufftHandle** returned by **cufftCreate**
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 output data

Return Values

CUFFT_SUCCESS	cuFFT successfully executed 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.

3.10. cuFFT and Multiple GPUs

3.10.1. Function cufftXtSetGPUs()

```
cufftResult
    cufftXtSetGPUs(cufftHandle plan, int nGPUs, int *whichGPUs);
```

cufftXtSetGPUs() indentifies which GPUs are to be used with the plan. As in the single GPU case **cufftCreate()** creates a plan and **cufftMakePlan*()** does the plan generation. This call will return an error if a non-default stream has been associated with the plan.

Note that the call to **cufftXtSetGPUs()** must occur after the call to **cufftCreate()** and prior to the call to **cufftMakePlan*()**.

Input

plan	**cufftHandle** returned by **cufftCreate**
nGPUs	Number of GPUs to use
whichGPUs	The GPUs to use

Return Values

CUFFT_SUCCESS	cuFFT successfully sets the GPUs to use.
CUFFT_INVALID_PLAN	The **plan** parameter is not a valid handle, or a non-default stream has been associated with the plan.
CUFFT_ALLOC_FAILED	The allocation of GPU resources for the plan failed.
CUFFT_INTERNAL_ERROR	An internal driver error was detected.
CUFFT_SETUP_FAILED	The cuFFT library failed to initialize.
CUFFT_INVALID_DEVICE	An invalid GPU index was specified.

3.10.2. Function cufftXtSetWorkArea()

```
cufftResult
    cufftXtSetWorkArea(cufftHandle plan, void **workArea);
```

cufftXtSetWorkArea() overrides the work areas associated with a plan. If the work area was auto-allocated, cuFFT frees the auto-allocated space. The **cufftXtExec*()** calls assume that the work area is valid and that it points to a contiguous region in each device memory that does not overlap with any other work area. If this is not the case, results are indeterminate.

Input

plan	**cufftHandle** returned by **cufftCreate**
workArea	Pointer to the pointers to workArea

Return Values

CUFFT_SUCCESS	cuFFT successfully allows user to override workArea pointer.
CUFFT_INVALID_PLAN	The **plan** parameter is not a valid handle.
CUFFT_INTERNAL_ERROR	An internal driver error was detected.
CUFFT_SETUP_FAILED	The cuFFT library failed to initialize.
CUFFT_INVALID_DEVICE	A GPU associated with the plan could not be selected.

3.10.3. cuFFT Multiple GPU Execution

3.10.3.1. Functions cufftXtExecDescriptorC2C() and cufftXtExecDescriptorZ2Z()

```
cufftResult
    cufftXtExecDescriptorC2C(cufftHandle plan, cudaLibXtDesc *idata,
        cudaLibXtDesc *idata, int direction);
cufftResult
    cufftXtExecDescriptorZ2Z(cufftHandle plan, cudaLibXtDesc *idata,
        cudaLibXtDesc *idata, int direction);
```

cufftXtExecDescriptorC2C() (**cufftXtExecDescriptorZ2Z()**) 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 **cudaLibXtDesc *idata** as input data. Since only in-place two GPU functionality is support, this function also stores the result in the **cudaLibXtDesc *idata** arrays.

Input

plan	**cufftHandle** returned by **cufftCreate**
idata	Pointer to the complex input data (in GPU memory) to transform
idata	Pointer to the complex output data (in GPU memory)
direction	The transform direction: **CUFFT_FORWARD** or **CUFFT_INVERSE**

Output

idata	Contains the complex Fourier coefficients

Return Values

CUFFT_SUCCESS	cuFFT successfully executed 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 **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_INVALID_DEVICE	An invalid GPU index was specified in a descriptor.

3.10.4. Memory Allocation and Data Movement Functions

Multiple GPU cuFFT execution functions assume a certain data layout in terms of what input data has been copied to which GPUs prior to execution, and what output data resides in which GPUs post execution. The following functions assist in allocation, setup and retrieval of the data. They must be called after the call to **cufftMakePlan*()**.

3.10.4.1. Function cufftXtMalloc()

```
cufftResult
    cufftXtMalloc(cufftHandle plan, cudaLibXtDesc **descriptor,
        cufftXtSubFormat format);
```

cufftXtMalloc() allocates a descriptor, and all memory for data in GPUs associated with the plan, and returns a pointer to the descriptor. Note the descriptor contains an array of device pointers so that the application may preprocess or postprocess the data on the GPUs. The enumerated parameter **cufftXtSubFormat_t** indicates if the buffer will be used for input or output.

Input

plan	**cufftHandle** returned by **cufftCreate**
descriptor	Pointer to a pointer to a **cudaLibXtDesc** object
format	**cufftXtSubFormat** value

Output

descriptor	Pointer to a pointer to a **cudaLibXtDesc** object

Return Values

CUFFT_SUCCESS	cuFFT successfully allows user to allocate descriptor and GPU memory.
CUFFT_INVALID_PLAN	The **plan** parameter is not a valid handle or it is not a multiple GPU **plan**.
CUFFT_ALLOC_FAILED	The allocation of GPU resources for the plan failed.
CUFFT_INTERNAL_ERROR	An internal driver error was detected.
CUFFT_SETUP_FAILED	The cuFFT library failed to initialize.
CUFFT_INVALID_DEVICE	An invalid GPU index was specified in the descriptor.

3.10.4.1.1. Parameter cufftXtSubFormat

cufftXtSubFormat_t is an enumerated type that indicates if the buffer will be used for input or output and the ordering of the data.

```
typedef enum cufftXtSubFormat_t {
    CUFFT_XT_FORMAT_INPUT,              //by default input is in linear order
 across GPUs
    CUFFT_XT_FORMAT_OUTPUT,             //by default output is in scrambled
 order depending on transform
    CUFFT_XT_FORMAT_INPLACE,            //by default inplace is input order,
 which is linear across GPUs
    CUFFT_XT_FORMAT_INPLACE_SHUFFLED,   //shuffled output order after execution
 of the transform
    CUFFT_FORMAT_UNDEFINED
} cufftXtSubFormat;
```

3.10.4.2. Function cufftXtFree()

```
cufftResult
    cufftXtFree(cudaLibXtDesc *descriptor);
```

cufftXtFree() frees the descriptor and all memory associated with it. The descriptor and memory must have been returned by a previous call to **cufftXtMalloc()**.

Input

descriptor	Pointer to a **cudaLibXtDesc** object

Return Values

CUFFT_SUCCESS	cuFFT successfully allows user to free descriptor and associated GPU memory.
CUFFT_INTERNAL_ERROR	An internal driver error was detected.

3.10.4.3. Function cufftXtMemcpy()

```
cufftResult
    cufftXtMemcpy(cufftHandle plan, void *dstPointer, void *srcPointer,
        cufftXtCopyType type);
```

cufftXtMemcpy() copies data between buffers on the host and GPUs or between GPUs. The enumerated parameter **cufftXtCopyType_t** indicates the type and direction of transfer.

Input

plan	**cufftHandle** returned by **cufftCreate**
dstPointer	Pointer to the destination address(es)
srcPointer	Pointer to the source address(es)
type	**cufftXtCopyType**value

Return Values

CUFFT_SUCCESS	cuFFT successfully allows user to copy memory between host and GPUs or between GPUs.
CUFFT_INVALID_PLAN	The **plan** parameter is not a valid handle.
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_DEVICE	An invalid GPU index was specified in a descriptor.

3.10.4.3.1. Parameter cufftXtCopyType

cufftXtCopyType_t is an enumerated type for multiple GPU functions that specifies the type of copy for **cufftXtMemcpy()**.

CUFFT_COPY_HOST_TO_DEVICE copies data from a contiguous host buffer to multiple device buffers, in the layout cuFFT requires for input data. **dstPointer** must point to a **cudaLibXtDesc** structure, and **srcPointer** must point to a host memory buffer.

CUFFT_COPY_DEVICE_TO_HOST copies data from multiple device buffers, in the layout cuFFT produces for output data, to a contiguous host buffer. **dstPointer** must point to a host memory buffer, and **srcPointer** must point to a **cudaLibXtDesc** structure.

CUFFT_COPY_DEVICE_TO_DEVICE copies data from multiple device buffers, in the layout cuFFT produces for output data, to multiple device buffers, in the layout cuFFT requires for input data. **dstPointer** and **srcPointer** must point to different **cudaLibXtDesc** structures (and therefore memory locations). That is, the copy cannot be in-place.

```
typedef enum cufftXtCopyType_t {
    CUFFT_COPY_HOST_TO_DEVICE,
    CUFFT_COPY_DEVICE_TO_HOST,
    CUFFT_COPY_DEVICE_TO_DEVICE
} cufftXtCopyType;
```

3.10.5. General Multiple GPU Descriptor Types

3.10.5.1. cudaXtDesc

A descriptor type used in multiple GPU routines that contains information about the GPUs and their memory locations.

```
    struct cudaXtDesc_t{
    int version;                              //descriptor version
    int nGPUs;                                //number of GPUs
    int GPUs[MAX_CUDA_DESCRIPTOR_GPUS];       //array of device IDs
    void *data[MAX_CUDA_DESCRIPTOR_GPUS];     //array of pointers to data, one
 per GPU
    size_t size[MAX_CUDA_DESCRIPTOR_GPUS];    //array of data sizes, one per GPU
    void *cudaXtState;                        //opaque CUDA utility structure
};
typedef struct cudaXtDesc_t cudaXtDesc;
```

3.10.5.2. cudaLibXtDesc

A descriptor type used in multiple GPU routines that contains information about the library used.

```
struct cudaLibXtDesc_t{
    int version;                //descriptor version
    cudaXtDesc *descriptor;     //multi-GPU memory descriptor
    libFormat library;          //which library recognizes the format
    int subFormat;              //library specific enumerator of sub formats
    void *libDescriptor;        //library specific descriptor e.g. FFT transform
 plan object
};
typedef struct cudaLibXtDesc_t cudaLibXtDesc;
```

3.11. cuFFT Callbacks

3.11.1. Function cufftXtSetCallback()

```
cufftResult
    cufftXtSetCallback(cufftHandle plan, void **callbackRoutine,
 cufftXtCallbackType type, void **callerInfo)
```

cufftXtSetCallback() specifies a load or store callback to be used with the plan. This call is valid only after a call to **cufftMakePlan*()**, which does the plan generation. If there was already a callback of this type associated with the plan, this new callback routine replaces it. If the new callback requires shared memory, you must call **cufftXtSetCallbackSharedSize** with the amount of shared memory it needs. cuFFT will not retain the amount of shared memory associated with the previous callback.

Input

plan	**cufftHandle** returned by **cufftCreate**
callbackRoutine	Array of callback routine pointers, one per GPU
type	type of callback routine
callerInfo	optional array of device pointers to caller specific information, one per GPU

Return Values

CUFFT_SUCCESS	cuFFT successfully associated the callback function with the plan.
CUFFT_INVALID_PLAN	The **plan** parameter is not a valid handle, or a non-default stream has been associated with the plan.
CUFFT_NO_LICENSE	This feature requires a valid license.
CUFFT_INTERNAL_ERROR	An internal driver error was detected.
CUFFT_SETUP_FAILED	The cuFFT library failed to initialize.

3.11.2. Function cufftXtClearCallback()

```
cufftResult
    cufftXtClearCallback(cufftHandle plan, cufftXtCallbackType type)
```

cufftXtClearCallback() instructs cuFFT to stop invoking the specified callback type when executing the plan. Only the specified callback is cleared. If no callback of this type had been specified, the return code is **CUFFT_SUCCESS**.

Input

plan	**cufftHandle** returned by **cufftCreate**
type	type of callback routine

Return Values

CUFFT_SUCCESS	cuFFT successfully disassociated the callback function with the plan.

CUFFT_INVALID_PLAN	The **plan** parameter is not a valid handle, or a non-default stream has been associated with the plan.
CUFFT_INTERNAL_ERROR	An internal driver error was detected.

3.11.3. Function cufftXtSetCallbackSharedSize()

```
cufftResult
    cufftXtSetCallbackSharedSize(cufftHandle plan, cufftXtCallbackType type,
 size_t sharedSize)
```

cufftXtSetCallbackSharedSize() instructs cuFFT to dynamically allocate shared memory at launch time, for use by the callback. The maximum allowable amount of shared memory is 16K bytes. cuFFT passes a pointer to this shared memory to the callback routine at execution time. This shared memory is only valid for the life of the load or store callback operation. During execution, cuFFT may overwrite shared memory for its own purposes.

Input

plan	**cufftHandle** returned by **cufftCreate**
type	type of callback routine
sharedSize	amount of shared memory requested

Return Values

CUFFT_SUCCESS	cuFFT will invoke the callback routine with a pointer to the requested amount of shared memory.
CUFFT_INVALID_PLAN	The **plan** parameter is not a valid handle, or a non-default stream has been associated with the plan.
CUFFT_INTERNAL_ERROR	An internal driver error was detected.
CUFFT_ALLOC_FAILED	cuFFT will not be able to allocate the requested amount of shared memory.

3.12. 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()**. This call will return an error for multiple GPU plans.

Input

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

Status Returned

CUFFT_SUCCESS	The stream was associated with the plan.
CUFFT_INVALID_PLAN	The **plan** parameter is not a valid handle, or it is a multiple GPU **plan**.

3.13. Function cufftGetVersion()

```
cufftResult
    cufftGetVersion(int *version);
```

Returns the version number of cuFFT.

Input

version	Pointer to the version number

Output

version	Contains the version number

Return Values

CUFFT_SUCCESS	cuFFT successfully returned the version number.

3.14. 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 **cufftSetCompatibilityMode()** 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** (deprecated) **CUFFT_COMPATIBILITY_FFTW_PADDING** (default) **CUFFT_COMPATIBILITY_FFTW_ASYMMETRIC** (deprecated) **CUFFT_COMPATIBILITY_FFTW_ALL**

Return Values

CUFFT_SUCCESS	cuFFT successfully set compatibiltiy mode.
CUFFT_INVALID_PLAN	The **plan** parameter is not a valid handle.
CUFFT_SETUP_FAILED	The cuFFT library failed to initialize.

3.15. 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 {
    CUFFT_COMPATIBILITY_NATIVE       = 0, // Compact data in native format
                                          // (deprecated. Use
                                          // CUFFT_COMPATIBILITY_FFTW_PADDING)
    CUFFT_COMPATIBILITY_FFTW_PADDING = 1, // FFTW-compatible alignment
                                          // (default value)
    CUFFT_COMPATIBILITY_FFTW_ASYMMETRIC = 2, // Deprecated. Asymmetric input is
                                             // always treated as in FFTW.
    CUFFT_COMPATIBILITY_FFTW_ALL        = 3
} cufftCompatibility;
```

3.16. cuFFT Types

3.16.1. 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;
```

3.16.2. Parameters for Transform Direction

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.

3.16.3. Type definitions for callbacks

The cuFFT library supports callback funtions for all combinations of single or double precision, real or complex data, load or store. These are enumerated in the parameter **`cufftXtCallbackType`**.

```
typedef enum cufftXtCallbackType_t {
    CUFFT_CB_LD_COMPLEX = 0x0,
    CUFFT_CB_LD_COMPLEX_DOUBLE = 0x1,
    CUFFT_CB_LD_REAL = 0x2,
    CUFFT_CB_LD_REAL_DOUBLE = 0x3,
    CUFFT_CB_ST_COMPLEX = 0x4,
    CUFFT_CB_ST_COMPLEX_DOUBLE = 0x5,
    CUFFT_CB_ST_REAL = 0x6,
    CUFFT_CB_ST_REAL_DOUBLE = 0x7,
    CUFFT_CB_UNDEFINED = 0x8
} cufftXtCallbackType;
```

The corresponding function prototypes and pointer type definitions are as follows:

```
typedef cufftComplex (*cufftCallbackLoadC)(void *dataIn, size_t offset, void
 *callerInfo, void *sharedPointer);

typedef cufftDoubleComplex (*cufftCallbackLoadZ)(void *dataIn, size_t offset,
 void *callerInfo, void *sharedPointer);

typedef cufftReal (*cufftCallbackLoadR)(void *dataIn, size_t offset, void
 *callerInfo, void *sharedPointer);

typedef cufftDoubleReal(*cufftCallbackLoadD)(void *dataIn, size_t offset, void
 *callerInfo, void *sharedPointer);


typedef void (*cufftCallbackStoreC)(void *dataOut, size_t offset, cufftComplex
 element, void *callerInfo, void *sharedPointer);

typedef void (*cufftCallbackStoreZ)(void *dataOut, size_t offset,
 cufftDoubleComplex element, void *callerInfo, void *sharedPointer);

typedef void (*cufftCallbackStoreR)(void *dataOut, size_t offset, cufftReal
 element, void *callerInfo, void *sharedPointer);

typedef void (*cufftCallbackStoreD)(void *dataOut, size_t offset,
 cufftDoubleReal element, void *callerInfo, void *sharedPointer);
```

3.16.4. Other cuFFT Types

3.16.4.1. 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;
```

3.16.4.2. cufftReal

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

```
typedef float cufftReal;
```

3.16.4.3. cufftDoubleReal

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

```
typedef double cufftDoubleReal;
```

3.16.4.4. cufftComplex

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

```
typedef cuComplex cufftComplex;
```

3.16.4.5. 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 multiple 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 1

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;
}

/*
 *  Results may not be immediately available so block device until all
 *  tasks have completed
 */

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

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

...

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 1

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 (cudaDeviceSynchronize() != 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 default FFTW padding mode.

```
#define NX 256
#define NY 128
#define NRANK 2
#define BATCH 1

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 (cufftExecC2R(plan, data, data) != CUFFT_SUCCESS){
 fprintf(stderr, "CUFFT Error: Unable to execute plan\n");
 return;
}

if (cudaDeviceSynchronize() != 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 NZ 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 (cudaDeviceSynchronize() != 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   // distance between successive input elements in innermost
 dimension
#define OSTRIDE 1   // distance between successive output elements in innermost
 dimension
#define IX (NX+2)
#define IY (NY+1)
#define OX (NX+3)
#define OY (NY+4)
#define IDIST (IX*IY*ISTRIDE+3) // distance between first element of two
 consecutive signals in a batch of input data
#define ODIST (OX*OY*OSTRIDE+5) // distance between first element of two
 consecutive signals in a batch of output data

cufftHandle plan;
cufftComplex *idata, *odata;
int isize = IDIST * BATCH;
int osize = ODIST * BATCH;
int n[NRANK] = {NX, NY};
int inembed[NRANK] = {IX, IY}; // pointer that indicates storage dimensions of
 input data
int onembed[NRANK] = {OX, OY}; // pointer that indicates storage dimensions of
 output data

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

...

cufftDestroy(plan);
cudaFree(idata);
cudaFree(odata);
```

4.6. 3D Complex-to-Complex Transforms using Two GPUs

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

```
// Demonstrate how to use CUFFT to perform 3-d FFTs using 2 GPUs
//
// cufftCreate() - Create an empty plan
    cufftHandle plan_input; cufftResult result;
    result = cufftCreate(&plan_input);
    if (result != CUFFT_SUCCESS) { printf ("*Create failed\n"); return; }
//
// cufftXtSetGPUs() - Define which GPUs to use
    int nGPUs = 2, whichGPUs[2];
    whichGPUs[0] = 0; whichGPUs[1] = 1;
    result = cufftXtSetGPUs (plan_input, nGPUs, whichGPUs);
    if (result != CUFFT_SUCCESS) { printf ("*XtSetGPUs failed\n"); return; }
//
// Initialize FFT input data
    size_t worksize[2];
    cufftComplex *host_data_input, *host_data_output;
    int nx = 64, ny = 128, nz = 32;
    int size_of_data = sizeof(cufftComplex) * nx * ny * nz;
    host_data_input = malloc(size_of_data);
    if (host_data_input == NULL) { printf ("malloc failed\n"); return; }
    host_data_output = malloc(size_of_data);
    if (host_data_output == NULL) { printf ("malloc failed\n"); return; }
    initialize_3d_data (nx, ny, nz, host_data_input, host_data_output);
//
// cufftMakePlan3d() - Create the plan
    result = cufftMakePlan3d (plan_input, nz, ny, nx, CUFFT_C2C, worksize);
    if (result != CUFFT_SUCCESS) { printf ("*MakePlan* failed\n"); return; }
//
// cufftXtMalloc() - Malloc data on multiple GPUs
    cudaLibXtDesc *device_data_input;
    result = cufftXtMalloc (plan_input, (void*)&device_data_input,
        CUFFT_XT_FORMAT_INPLACE);
    if (result != CUFFT_SUCCESS) { printf ("*XtMalloc failed\n"); return; }
//
// cufftXtMemcpy() - Copy data from host to multiple GPUs
    result = cufftXtMemcpy (plan_input, device_data_input,
        host_data_input, CUFFT_COPY_HOST_TO_DEVICE);
    if (result != CUFFT_SUCCESS) { printf ("*XtMemcpy failed\n"); return; }
//
// cufftXtExecDescriptorC2C() - Execute FFT on multiple GPUs
    result = cufftXtExecDescriptorC2C (plan_input, device_data_input,
        device_data_input, CUFFT_FORWARD);
    if (result != CUFFT_SUCCESS) { printf ("*XtExec* failed\n"); return; }
//
// cufftXtMemcpy() - Copy data from multiple GPUs to host
    result = cufftXtMemcpy (plan_input, host_data_output,
        device_data_input, CUFFT_COPY_DEVICE_TO_HOST);
    if (result != CUFFT_SUCCESS) { printf ("*XtMemcpy failed\n"); return; }
//
// Print output and check results
    int output_return = output_3d_results (nx, ny, nz,
        host_data_input, host_data_output);
    if (output_return != 0) { return; }
//
// cufftXtFree() - Free GPU memory
    result = cufftXtFree(device_data_input);
    if (result != CUFFT_SUCCESS) { printf ("*XtFree failed\n"); return; }
//
// cufftDestroy() - Destroy FFT plan
    result = cufftDestroy(plan_input);
    if (result != CUFFT_SUCCESS) { printf ("*Destroy failed: code\n"); return; }
    free(host_data_input); free(host_data_output);
```

4.7. 1D Complex-to-Complex Transforms using Two GPUs with Natural Order

In this example a one-dimensional complex-to-complex transform is applied to the input data using two GPUs. The output data is in natural order in GPU memory.

```
// Demonstrate how to use CUFFT to perform 1-d FFTs using 2 GPUs
// Output on the GPUs is in natural output
// Function return codes should be checked for errors in actual code
//
// cufftCreate() - Create an empty plan
    cufftHandle plan_input; cufftResult result;
    result = cufftCreate(&plan_input);
//
// cufftXtSetGPUs() - Define which GPUs to use
    int nGPUs = 2, whichGPUs[2];
    whichGPUs[0] = 0; whichGPUs[1] = 1;
    result = cufftXtSetGPUs (plan_input, nGPUs, whichGPUs);
//
// Initialize FFT input data
    size_t worksize[2];
    cufftComplex *host_data_input, *host_data_output;
    int nx = 1024, batch = 1, rank = 1, n[1];
    int inembed[1], istride, idist, onembed[1], ostride, odist;
    n[0] = nx;
    int size_of_data = sizeof(cufftComplex) * nx * batch;
    host_data_input = malloc(size_of_data);
    host_data_output = malloc(size_of_data);
    initialize_1d_data (nx, batch, rank, n, inembed, &istride, &idist,
        onembed, &ostride, &odist, host_data_input, host_data_output);
//
// cufftMakePlanMany() - Create the plan
    result = cufftMakePlanMany (plan_input, rank, n, inembed, istride, idist,
        onembed, ostride, odist, CUFFT_C2C, batch, worksize);
//
// cufftXtMalloc() - Malloc data on multiple GPUs
    cudaLibXtDesc *device_data_input, *device_data_output;
    result = cufftXtMalloc (plan_input, (void*)&device_data_input,
        CUFFT_XT_FORMAT_INPLACE);
    result = cufftXtMalloc (plan_input, (void*)&device_data_output,
        CUFFT_XT_FORMAT_INPLACE);
//
// cufftXtMemcpy() - Copy data from host to multiple GPUs
    result = cufftXtMemcpy (plan_input, device_data_input,
        host_data_input, CUFFT_COPY_HOST_TO_DEVICE);
//
// cufftXtExecDescriptorC2C() - Execute FFT on multiple GPUs
    result = cufftXtExecDescriptorC2C (plan_input, device_data_input,
        device_data_input, CUFFT_FORWARD);
//
// cufftXtMemcpy() - Copy the data to natural order on GPUs
    result = cufftXtMemcpy (plan_input, device_data_output,
        device_data_input, CUFFT_COPY_DEVICE_TO_DEVICE);
//
// cufftXtMemcpy() - Copy natural order data from multiple GPUs to host
    result = cufftXtMemcpy (plan_input, host_data_output,
        device_data_output, CUFFT_COPY_DEVICE_TO_HOST);
//
// Print output and check results
    int output_return = output_1d_results (nx, batch,
        host_data_input, host_data_output);
//
// cufftXtFree() - Free GPU memory
    result = cufftXtFree(device_data_input);
    result = cufftXtFree(device_data_output);
//
// cufftDestroy() - Destroy FFT plan
    result = cufftDestroy(plan_input);
    free(host_data_input); free(host_data_output);
```

4.8. 1D Complex-to-Complex Convolution using Two GPUs

In this example a one-dimensional convolution is calculated using complex-to-complex transforms.

```
//
// Demonstrate how to use CUFFT to perform a convolution using 1-d FFTs and
// 2 GPUs. The forward FFTs use both GPUs, while the inverse FFT uses one.
// Function return codes should be checked for errors in actual code.
//
// cufftCreate() - Create an empty plan
    cufftResult result; cudaError_t cuda_status;
    cufftHandle plan_forward_2_gpus, plan_inverse_1_gpu;
    result = cufftCreate(&plan_forward_2_gpus);
    result = cufftCreate(&plan_inverse_1_gpu);
//
// cufftXtSetGPUs() - Define which GPUs to use
    int nGPUs = 2, whichGPUs[2];
    whichGPUs[0] = 0; whichGPUs[1] = 1;
    result = cufftXtSetGPUs (plan_forward_2_gpus, nGPUs, whichGPUs);
//
// Initialize FFT input data
    size_t worksize[2];
    cufftComplex *host_data_input, *host_data_output;
    int nx = 1048576, batch = 2, rank = 1, n[1];
    int inembed[1], istride, idist, onembed[1], ostride, odist;
    n[0] = nx;
    int size_of_one_set = sizeof(cufftComplex) * nx;
    int size_of_data = size_of_one_set * batch;
    host_data_input = (cufftComplex*)malloc(size_of_data);
    host_data_output = (cufftComplex*)malloc(size_of_one_set);
    initialize_1d_data (nx, batch, rank, n, inembed, &istride, &idist,
        onembed, &ostride, &odist, host_data_input, host_data_output);
//
// cufftMakePlanMany(), cufftPlan1d - Create the plans
    result = cufftMakePlanMany (plan_forward_2_gpus, rank, n, inembed,
        istride, idist, onembed, ostride, odist, CUFFT_C2C, batch, worksize);
    result = cufftPlan1d (&plan_inverse_1_gpu, nx, CUFFT_C2C, 1);
//
// cufftXtMalloc(), cudaMallocHost - Allocate data for GPUs
    cudaLibXtDesc *device_data_input; cufftComplex *GPU0_data_from_GPU1;
    result = cufftXtMalloc (plan_forward_2_gpus, &device_data_input,
        CUFFT_XT_FORMAT_INPLACE);
    int device0 = device_data_input->descriptor->GPUs[0];
    cudaSetDevice(device0) ;
    cuda_status = cudaMallocHost ((void**)&GPU0_data_from_GPU1,size_of_one_set);
//
// cufftXtMemcpy() - Copy data from host to multiple GPUs
    result = cufftXtMemcpy (plan_forward_2_gpus, device_data_input,
        host_data_input, CUFFT_COPY_HOST_TO_DEVICE);
//
// cufftXtExecDescriptorC2C() - Execute forward FFTs on multiple GPUs
    result = cufftXtExecDescriptorC2C (plan_forward_2_gpus, device_data_input,
        device_data_input, CUFFT_FORWARD);
//
// cudaMemcpy result from GPU1 to GPU0
    cufftComplex *device_data_on_GPU1;
    device_data_on_GPU1 = (cufftComplex*)
        (device_data_input->descriptor->data[1]);
    cuda_status = cudaMemcpy (GPU0_data_from_GPU1, device_data_on_GPU1,
        size_of_one_set, cudaMemcpyDeviceToDevice);
//
// Continued on next page
//
```

```
//
// Demonstrate how to use CUFFT to perform a convolution using 1-d FFTs and
// 2 GPUs. The forward FFTs use both GPUs, while the inverse FFT uses one.
// Function return codes should be checked for errors in actual code.
//
// Part 2
//
// Multiply results and scale output
    cufftComplex *device_data_on_GPU0;
    device_data_on_GPU0 = (cufftComplex*)
        (device_data_input->descriptor->data[0]);
    cudaSetDevice(device0) ;
    ComplexPointwiseMulAndScale<<<32, 256>>>((cufftComplex*)device_data_on_GPU0,
        (cufftComplex*) GPU0_data_from_GPU1, nx);
//
// cufftExecC2C() - Execute inverse FFT on one GPU
    result = cufftExecC2C (plan_inverse_1_gpu, GPU0_data_from_GPU1,
        GPU0_data_from_GPU1, CUFFT_INVERSE);
//
// cudaMemcpy() - Copy results from GPU0 to host
    cuda_status = cudaMemcpy(host_data_output, GPU0_data_from_GPU1,
        size_of_one_set, cudaMemcpyDeviceToHost);
//
// Print output and check results
    int output_return = output_1d_results (nx, batch,
        host_data_input, host_data_output);
//
// cufftDestroy() - Destroy FFT plans
    result = cufftDestroy(plan_forward_2_gpus);
    result = cufftDestroy(plan_inverse_1_gpu);
//
// cufftXtFree(), cudaFreeHost(), free() - Free GPU and host memory
    result = cufftXtFree(device_data_input);
    cuda_status = cudaFreeHost (GPU0_data_from_GPU1);
    free(host_data_input); free(host_data_output);
```

```
//
// Utility routine to perform complex pointwise multiplication with scaling
__global__ void ComplexPointwiseMulAndScale
    (cufftComplex *a, cufftComplex *b, int size)
{
    const int numThreads = blockDim.x * gridDim.x;
    const int threadID = blockIdx.x * blockDim.x + threadIdx.x;
    float scale = 1.0f / (float)size;
    cufftComplex c;
    for (int i = threadID; i < size; i += numThreads)
    {
        c = cuCmulf(a[i], b[i]);
        b[i] = make_cuFloatComplex(scale*cuCrealf(c), scale*cuCimagf(c));
    }
    return;
}
```

Chapter 5. MULTIPLE GPU DATA ORGANIZATION

This chapter explains how data are distributed between the GPUs, before and after a multiple GPU transform. For simplicity, it is assumed in this chapter that the caller has specified GPU 0 and GPU 1 to perform the transform.

5.1. Multiple GPU Data Organization for Batched Transforms

For batches of transforms, each individual transform is executed on a single GPU. The first $\left\lceil \frac{\text{batch size} + 1}{2} \right\rceil$ transforms are executed on GPU 0, and the remainder are executed on GPU 1. This approach removes the need for data exchange between the GPUs, and results in nearly perfect scaling for even batch sizes.

5.2. Multiple GPU Data Organization for Single 2D and 3D Transforms

Single transforms performed on multiple GPUs require the data to be divided between the GPUs. Then execution takes place in phases. First, for 2D and 3D transforms, each GPU does half of the transform in (rank - 1) dimensions. Then data are exchanged between the GPUs so that the final dimension can be processed.

For 2D transforms using an array declared in C as **`data[x][y]`**, the surface is distributed prior to the transform such that each GPU receives half the surface, with dimensions [x/2][y]. GPU 0 receives all y elements for x indices ranging from 0...(x/2-1). GPU 1 receives all y elements for x indices ranging from (x/2)...(x-1). After the transform, each GPU again has half the surface, but with dimensions [x][y/2]. GPU 0 has all x elements for y indices ranging from 0...(y/2-1). GPU 1 has all x elements for y indices ranging from (y/2)...(y-1).

For 3D transforms using an array declared in C as **`data[x][y][z]`**, the volume is distributed prior to the transform such that each GPU receives half the volume, with dimensions [x/2][y][z]. GPU 0 receives the [y][z] planes with x indices ranging from 0...

(x/2-1). GPU 1 receives the [y][z] planes with x indices ranging from (x/2)...(x-1). After the transform, each GPU again has half the volume, but with dimensions [x][y/2][z]. GPU 0 has the [x][z] planes with y indices ranging from 0...(y/2-1). GPU 1 has the [x][z] planes with y indices ranging from (y/2)...(y-1).

5.3. Multiple-GPU Data Organization for Single 1D Transforms

For 1D transforms, the initial distribution of data to the GPUs is similar to the 2D and 3D cases. For a transform of dimension x, GPU 0 receives data ranging from 0...(x/2-1). GPU 1 receives data ranging from (x/2)...(x-1). The partitioning of the work between the GPUs results in a more complex output arrangement than in the 2D and 3D cases.

cuFFT performs multiple GPU 1D transforms by decomposing the transform size into factors **Factor1** and **Factor2**, and treating the data as a grid of size **Factor1** x **Factor2**. The four steps done to calculate the 1D FFT are: **Factor1** transforms of size **Factor2**, data exchange between the GPUs, a pointwise twiddle multiplication, and **Factor2** transforms of size **Factor1**. To gain efficiency by overlapping computation with data exchange, cuFFT breaks the whole transform into independent segments or strings, which can be processed while others are in flight. A side effect of this algorithm is that the output of the transform is not in linear order. The output in GPU memory is in strings, each of which is composed of **Factor2** substrings of equal size. Each substring contains contiguous results starting **Factor1** elements subsequent to start of the previous substring. Each string starts substring size elements after the start of the previous string. The strings appear in order, the first half on GPU 0, and the second half on GPU 1. See the example below:

```
transform size = 1024
number of strings = 8
Factor1 = 64
Factor2 = 16
substrings per string for output layout is Factor2 (16)
string size = 1024/8 = 128
substring size = 128/16 = 8
stride between substrings = 1024/16 = Factor1 (64)

On GPU 0:
string 0 has substrings with indices 0...7   64...71   128...135 ... 960...967
string 1 has substrings with indices 8...15  72...79   136...143 ... 968...975
...
On GPU 1:
string 4 has substrings with indices 32...39  96...103 160...167 ... 992...999
...
string 7 has substrings with indices 56...63 120...127 184...191 ... 1016...1023
```

The cufftXtQueryPlan API allows the caller to retrieve a structure containing the number of strings, the decomposition factors, and (in the case of power of 2 size) some useful mask and shift elements. The example below shows how cufftXtQueryPlan is invoked. It

also shows how to translate from an index in the host input array to the corresponding index on the device, and vice versa.

```
/*
 * These routines demonstrate the use of cufftXtQueryPlan to get the 1D
 * factorization and convert between permuted and linear indexes.
 */
/*
 * Set up a 1D plan that will execute on GPU 0 and GPU1, and query
 * the decomposition factors
 */
int main(int argc, char **argv){
    cufftHandle plan;
    cufftResult stat;
    int whichGPUs[2] = { 0, 1 };
    cufftXt1dFactors factors;
    stat = cufftCreate( &plan );
    if (stat != CUFFT_SUCCESS) {
        printf("Create error %d\n",stat);
        return 1;
    }
    stat = cufftXtSetGPUs( plan, 2, whichGPUs );
    if (stat != CUFFT_SUCCESS) {
        printf("SetGPU error %d\n",stat);
        return 1;
    }
    stat = cufftMakePlan1d( plan, size, CUFFT_C2C, 1, workSizes );
    if (stat != CUFFT_SUCCESS) {
        printf("MakePlan error %d\n",stat);
        return 1;
    }
    stat = cufftXtQueryPlan( plan, (void *) &factors, CUFFT_QUERY_1D_FACTORS );
    if (stat != CUFFT_SUCCESS) {
        printf("QueryPlan error %d\n",stat);
        return 1;
    }
    printf("Factor 1 %zd, Factor2 %zd\n",factors.factor1,factors.factor2);
    cufftDestroy(plan);
    return 0;
}
```

```
/*
 * Given an index into a permuted array, and the GPU index return the
 * corresponding linear index from the beginning of the input buffer.
 *
 * Parameters:
 *      factors     input:  pointer to cufftXt1dFactors as returned by
 *                          cufftXtQueryPlan
 *      permutedIx  input:  index of the desired element in the device output
 *                          array
 *      linearIx    output: index of the corresponding input element in the
 *                          host array
 *      GPUix       input:  index of the GPU containing the desired element
 */
cufftResult permuted2Linear( cufftXt1dFactors * factors,
                             size_t permutedIx,
                             size_t *linearIx,
                             int GPUIx ) {
    size_t indexInSubstring;
    size_t whichString;
    size_t whichSubstring;
    // the low order bits of the permuted index match those of the linear index
    indexInSubstring = permutedIx & factors->substringMask;
    // the next higher bits are the substring index
    whichSubstring = (permutedIx >> factors->substringShift) &
                     factors->factor2Mask;
    // the next higher bits are the string index on this GPU
    whichString = (permutedIx >> factors->stringShift) & factors->stringMask;
    // now adjust the index for the second GPU
    if (GPUIx) {
        whichString += factors->stringCount/2;
    }
    // linear index low order bits are the same
    // next higher linear index bits are the string index
    *linearIx = indexInSubstring + ( whichString << factors->substringShift );
    // next higher bits of linear address are the substring index
    *linearIx += whichSubstring << factors->factor1Shift;
    return CUFFT_SUCCESS;
}
```

```
/*
 * Given a linear index into a 1D array, return the GPU containing the permuted
 * result, and index from the start of the data buffer for that element.
 *
 * Parameters:
 *      factors     input:  pointer to cufftXt1dFactors as returned by
 *                          cufftXtQueryPlan
 *      linearIx    input:  index of the desired element in the host input
 *                          array
 *      permutedIx  output: index of the corresponding result in the device
 *                          output array
 *      GPUix       output: index of the GPU containing the result
 */
cufftResult linear2Permuted( cufftXt1dFactors * factors,
                             size_t linearIx,
                             size_t *permutedIx,
                             int *GPUIx ) {
    size_t indexInSubstring;
    size_t whichString;
    size_t whichSubstring;
    size_t whichStringMask;
    int whichStringShift;
    if (linearIx >= factors->size) {
        return CUFFT_INVALID_VALUE;
    }
    // get a useful additional mask and shift count
    whichStringMask = factors->stringCount -1;
    whichStringShift = (factors->factor1Shift + factors->factor2Shift) -
                       factors->stringShift ;
    // the low order bits identify the index within the substring
    indexInSubstring = linearIx & factors->substringMask;
    // first determine which string has our linear index.
    // the low order bits indentify the index within the substring.
    // the next higher order bits identify which string.
    whichString = (linearIx >> factors->substringShift) & whichStringMask;
    // the first stringCount/2 strings are in the first GPU,
    // the rest are in the second.
    *GPUIx = whichString/(factors->stringCount/2);
    // next determine which substring within the string has our index
    // the substring index is in the next higher order bits of the index
    whichSubstring = (linearIx >>(factors->substringShift + whichStringShift)) &
                     factors->factor2Mask;
    // now we can re-assemble the index
    *permutedIx = indexInSubstring;
    *permutedIx += whichSubstring << factors->substringShift;
    if ( !*GPUIx ) {
        *permutedIx += whichString << factors->stringShift;
    } else {
        *permutedIx += (whichString - (factors->stringCount/2) ) <<
                       factors->stringShift;
    }
    return CUFFT_SUCCESS;
}
```

Chapter 6. FFTW CONVERSION GUIDE

cuFFT differs from FFTW in that FFTW has many plans and a single execute function while cuFFT has fewer plans, but multiple execute functions. The cuFFT execute functions determine the precision (single or double) and whether the input is complex or real valued. The following table shows the relationship between the two interfaces.

FFTW function	cuFFT function
`fftw_plan_dft_1d(), fftw_plan_dft_r2c_1d(), fftw_plan_dft_c2r_1d()`	`cufftPlan1d()`
`fftw_plan_dft_2d(), fftw_plan_dft_r2c_2d(), fftw_plan_dft_c2r_2d()`	`cufftPlan2d()`
`fftw_plan_dft_3d(), fftw_plan_dft_r2c_3d(), fftw_plan_dft_c2r_3d()`	`cufftPlan3d()`
`fftw_plan_dft(), fftw_plan_dft_r2c(), fftw_plan_dft_c2r()`	`cufftPlanMany()`
`fftw_plan_many_dft(), fftw_plan_many_dft_r2c(), fftw_plan_many_dft_c2r()`	`cufftPlanMany()`
`fftw_execute()`	`cufftExecC2C(), cufftExecZ2Z(), cufftExecR2C(), cufftExecD2Z(), cufftExecC2R(), cufftExecZ2D()`
`fftw_destroy_plan()`	`cufftDestroy()`

Chapter 7. FFTW INTERFACE TO CUFFT

NVIDIA provides FFTW3 interfaces to the cuFFT library. This allows applications using FFTW to use NVIDIA GPUs with minimal modifications to program source code. To use the interface first do the following two steps

- It is recommended that you replace the include file **`fftw3.h`** with **`cufftw.h`**
- Instead of linking with the double/single precision libraries such as **`fftw3/fftw3f`** libraries, link with both the cuFFT and cuFFTW libraries
- Ensure the search path includes the directory containing **`cuda_runtime_api.h`**

After an application is working using the FFTW3 interface, users may want to modify their code to move data to and from the GPU and use the routines documented in the FFTW Conversion Guide for the best performance.

The following tables show which components and functions of FFTW3 are supported in cuFFT.

Section in FFTW manual	Supported	Unsupported
Complex numbers	**`fftw_complex`**, **`fftwf_complex`** types	
Precision	double **`fftw3`**, single **`fftwf3`**	long double **`fftw3l`**, quad precision **`fftw3q`** are not supported since CUDA functions operate on double and single precision floating-point quantities
Memory Allocation		**`fftw_malloc()`**, **`fftw_free()`**, **`fftw_alloc_real()`**, **`fftw_alloc_complex()`**, **`fftwf_alloc_real()`**, **`fftwf_alloc_complex()`**
Multi-threaded FFTW		**`fftw3_threads`**, **`fftw3_omp`** are not supported
Distributed-memory FFTW with MPI		**`fftw3_mpi`**,**`fftw3f_mpi`** are not supported

Note that for each of the double precision functions below there is a corresponding single precision version with the letters **fftw** replaced by **fftwf**.

Section in FFTW manual	Supported	Unsupported
Using Plans	`fftw_execute()`, `fftw_destroy_plan()`, `fftw_cleanup()`, `fftw_print_plan()`	`fftw_cost()`, `fftw_flops()` exist but are not functional
Basic Interface		
Complex DFTs	`fftw_plan_dft_1d()`, `fftw_plan_dft_2d()`, `fftw_plan_dft_3d()`, `fftw_plan_dft()`	
Planner Flags		Planner flags are ignored and the same plan is returned regardless
Real-data DFTs	`fftw_plan_dft_r2c_1d()`, `fftw_plan_dft_r2c_2d()`, `fftw_plan_dft_r2c_3d()`, `fftw_plan_dft_r2c()`, `fftw_plan_dft_c2r_1d()`, `fftw_plan_dft_c2r_2d()`, `fftw_plan_dft_c2r_3d()`, `fftw_plan_dft_c2r()`	
Read-data DFT Array Format		Not supported
Read-to-Real Transform		Not supported
Read-to-Real Transform Kinds		Not supported
Advanced Interface		
Advanced Complex DFTs	`fftw_plan_many_dft()` with multiple 1D, 2D, 3D transforms	`fftw_plan_many_dft()` with 4D or higher transforms or a 2D or higher batch of embedded transforms
Advanced Real-data DFTs	`fftw_plan_many_dft_r2c()`, `fftw_plan_many_dft_c2r()` with multiple 1D, 2D, 3D transforms	`fftw_plan_many_dft_r2c()`, `fftw_plan_many_dft_c2r()` with 4D or higher transforms or a 2D or higher batch of embedded transforms
Advanced Real-to-Real Transforms		Not supported
Guru Interface		
Interleaved and split arrays	Interleaved format	Split format
Guru vector and transform sizes	`fftw_iodim` struct	
Guru Complex DFTs	`fftw_plan_guru_dft()`, `fftw_plan_guru_dft_r2c()`, `fftw_plan_guru_dft_c2r()` with multiple 1D, 2D, 3D transforms	`fftw_plan_guru_dft()`, `fftw_plan_guru_dft_r2c()`, `fftw_plan_guru_dft_c2r()` with

Section in FFTW manual	Supported	Unsupported
		4D or higher transforms or a 2D or higher batch of transforms
Guru Real-data DFTs		Not supported
Guru Real-to-real Transforms		Not supported
64-bit Guru Interface		Not supported
New-array Execute Functions	`fftw_execute_dft()`, `fftw_execute_dft_r2c()`, `fftw_execute_dft_c2r()` with interleaved format	Split format and real-to-real functions
Wisdom		`fftw_export_wisdom_to_file()`, `fftw_import_wisdom_from_file()` exist but are not functional. Other wisdom functions do not have entry points in the library.

Chapter 8. DEPRECATED FUNCTIONALITY

The cuFFT native data layout, specified by **CUFFT_COMPATIBILITY_NATIVE** has been deprecated. Use **CUFFT_COMPATIBILITY_FFTW_PADDING.**

The cuFFT asymmetric data input layout, specified by **CUFFT_COMPATIBILITY_FFTW_ASYMMETRIC** has been deprecated. Use **CUFFT_COMPATIBILITY_FFTW_PADDING.** cuFFT always treats asymmetric input in the same way as FFTW.

Batch sizes other than 1 for **cufftPlan1d()** have been deprecated. Use **cufftPlanMany()** for multiple batch execution.