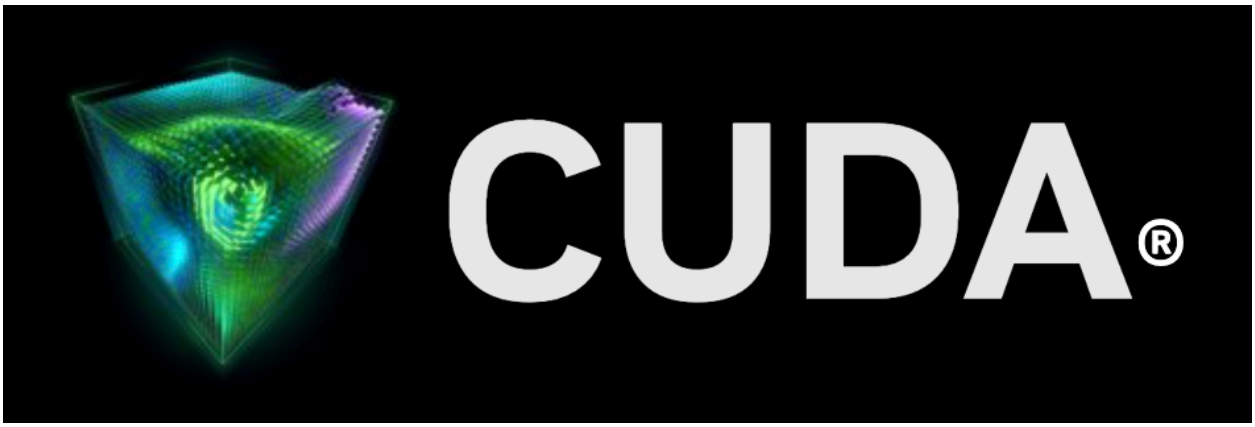

cuFFT

Release 12.8

NVIDIA Corporation

Jan 21, 2025

Contents

cuFFT API Reference

The API reference guide for cuFFT, the CUDA Fast Fourier Transform library.

cuFFT Release Notes: CUDA Toolkit Release Notes

cuFFT GitHub Samples: CUDA Library Samples

Nvidia Developer Forum: GPU-Accelerated Libraries

Provide Feedback: Math-Libs-Feedback@nvidia.com

Related FFT Libraries:

- cuFFTMP
- cuFFTDx
- cuFFT LTO EA (DEPRECATED)
- NVPL FFT

Relevant cuFFT Blog Posts and GTC presentations:

- Accelerating GPU Applications with NVIDIA Math Libraries
- Multinode Multi-GPU: Using NVIDIA cuFFTMp FFTs at Scale
- New Asynchronous Programming Model Library Now Available with NVIDIA HPC SDK v22.11
- Just-In-Time Link-Time Optimization Adoption in cuSPARSE/cuFFT: Use Case Overview

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
- Half-precision (16-bit floating point), single-precision (32-bit floating point) and double-precision (64-bit floating point). Transforms of lower precision have higher performance.
- 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 layout
- Execution of transforms across multiple GPUs
- Streamed execution, enabling asynchronous computation and data movement

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

Please note that starting from CUDA 11.0, the minimum supported GPU architecture is SM35. See Deprecated Functionality.

Chapter 1. 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);
}
```

1.1. 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 https://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.

Functions in the cuFFT and cuFFTW library assume that the data is in GPU visible memory. This means any memory allocated by `cudaMalloc`, `cudaMallocHost` and `cudaMallocManaged` or registered with `cudaHostRegister` can be used as input, output or plan work area with cuFFT and cuFFTW functions. For the best performance input data, output data and plan work area should reside in device memory.

cuFFTW library also supports input data and output data that is not GPU visible.

1.2. 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.
- `cufftXtMakePlanMany()` - Creates a plan supporting batched input and strided data layouts for any supported precision.

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.

1.2.1. Free Memory Requirement

The first program call to any cuFFT function causes the initialization of the cuFFT kernels. This can fail if there is not enough free memory on the GPU. It is advisable to initialize cufft first (e.g. by creating a plan) and then allocating memory.

1.2.2. Plan Initialization Time

During plan initialization, cuFFT conducts a series of steps, including heuristics to determine which kernels to be used as well as kernel module loads. Starting from CUDA 12.0, cuFFT delivers a larger portion of kernels using the CUDA Parallel Thread eXecution assembly form (PTX code), instead of the binary form (cubin object). The PTX code of cuFFT kernels are loaded and compiled further to the binary code by the CUDA device driver at runtime when a cuFFT plan is initialized. This is called just-in-time (JIT) compilation.

JIT compilation slightly increases cuFFT plan initialization time, depending on the transform size and the speed of the host CPU (see Module load driver API) . But the JIT overhead occurs only when a binary code is generated for the first time during plan initialization using one of the plan creation functions. The device driver automatically caches a copy of the generated binary code to avoid repeating the compilation in subsequent invocations. If necessary, `CUDA_CACHE_PATH` or `CUDA_CACHE_MAXSIZE` can be customized to set the cache folder and max size (see detail in CUDA Environmental Variables), but the default settings are fine in general.

1.3. 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).

Note: Complex-to-real (C2R) transforms accept complex-Hermitian input. For one-dimensional signals, this requires the 0th element (and the $\frac{N}{2}$th input if N is even) to be real-valued, i.e. its imaginary part should be zero. For d-dimension signals, this means $x_{(n_1,n_2,\ldots,n_d)} = x^*_{(N_1-n_1,N_2-n_2,\ldots,N_d-n_d)}$. Otherwise, the behavior of the transform is undefined. Also see Multidimensional Transforms.

Functions `cufftXtExec()` and `cufftXtExecDescriptor()` can perform transforms on any of the supported types.

1.3.1. Half-precision cuFFT Transforms

Half-precision transforms have the following limitations:

- Minimum GPU architecture is SM_53
- Sizes are restricted to powers of two only
- Strides on the real part of real-to-complex and complex-to-real transforms are not supported
- More than one GPU is not supported
- Transforms spanning more than 4 billion elements are not supported

Please refer to `cufftXtMakePlanMany` function for plan creation details.

The CUDA Toolkit provides the `cuda_fp16.h` header with types and intrinsic functions for handling half-precision arithmetic.

1.3.2. Bfloat16-precision cuFFT Transforms

cuFFT supports bfloat16 precision using the `nv_bfloat16` data type. Please note that cuFFT utilizes a combination of single- and bfloat16-precision arithmetic operations when computing the FFT in bfloat16 precision. Bfloat16-precision transforms have similar limitations to half-precision transforms:

- Minimum GPU architecture is SM_80
- Sizes are restricted to powers of two only
- Strides on the real part of real-to-complex and complex-to-real transforms are not supported
- More than one GPU is not supported
- Transforms spanning more than 4 billion elements are not supported

Please refer to `cufftXtMakePlanMany` function for plan creation details.

The CUDA Toolkit provides the `cuda_bf16.h` header with types and intrinsic functions for handling bfloat16-precision arithmetic.

1.4. 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 should use `padded` data layout. This layout is FFTW compatibile.

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.

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 $(\lfloor \frac{N}{2} \rfloor + 1)$ 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. Out-of-place complex-to-real FFT will always **overwrite** input buffer. 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.

1.5. 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:

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

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

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

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: $x_{(n_1,n_2,\ldots,n_d)} = x^*_{(N_1-n_1,N_2-n_2,\ldots,N_d-n_d)}$.

C2R and R2C algorithms take advantage of this fact by operating only on half of the elements of signal array, namely on: $x_{\mathbf{n}}$ for $\mathbf{n} \in \{1, \ldots, N_1\} \times \ldots \times \{1, \ldots, N_{d-1}\} \times \{1, \ldots, \lfloor \frac{N_d}{2} \rfloor + 1\}$.

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

Dims	FFT type	Input data size	Output data size
1D	C2C	$\mathbf{N}_1$`cufftComplex`	$\mathbf{N}_1$`cufftComplex`
1D	C2R	$\lfloor \frac{\mathbf{N}_1}{2} \rfloor + 1$`cufftComplex`	$\mathbf{N}_1$`cufftReal`
1D	R2C	$\mathbf{N}_1$`cufftReal`	$\lfloor \frac{\mathbf{N}_1}{2} \rfloor + 1$`cufftComplex`
2D	C2C	$\mathbf{N}_1\mathbf{N}_2$`cufftComplex`	$\mathbf{N}_1\mathbf{N}_2$`cufftComplex`
2D	C2R	$\mathbf{N}_1(\lfloor \frac{\mathbf{N}_2}{2} \rfloor + 1)$`cufftComplex`	$\mathbf{N}_1\mathbf{N}_2$`cufftReal`
2D	R2C	$\mathbf{N}_1\mathbf{N}_2$`cufftReal`	$\mathbf{N}_1(\lfloor \frac{\mathbf{N}_2}{2} \rfloor + 1)$`cufftComplex`
3D	C2C	$\mathbf{N}_1\mathbf{N}_2\mathbf{N}_3$`cufftComplex`	$\mathbf{N}_1\mathbf{N}_2\mathbf{N}_3$`cufftComplex`
3D	C2R	$\mathbf{N}_1\mathbf{N}_2(\lfloor \frac{\mathbf{N}_3}{2} \rfloor + 1)$`cufftComplex`	$\mathbf{N}_1\mathbf{N}_2\mathbf{N}_3$`cufftReal`
3D	R2C	$\mathbf{N}_1\mathbf{N}_2\mathbf{N}_3$`cufftReal`	$\mathbf{N}_1\mathbf{N}_2(\lfloor \frac{\mathbf{N}_3}{2} \rfloor + 1)$`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];
```

1.6. 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.

1.7. 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.

cuFFT uses private streams internally to sort operations, including event syncrhonization. cuFFT does not guarantee ordering of internal operations, and the order is only preserved with respect to the streams set by the user.

As of CUDA 11.2 (cuFFT 10.4.0), `cufftSetStream()` is supported in multiple GPU cases. However, calls to `cufftXtMemcpy()` are still synchronous across multiple GPUs when using streams. In previous versions of cuFFT, `cufftSetStream()` returns an error in the multiple GPU case. Likewise, calling certain multi-GPU functions such as `cufftXtSetCallback()` after setting a stream with `cufftSetStream()` will result in an error (see API functions for more details).

Please note that in order to overlap plans using single plan handle user needs to manage work area buffers. Each concurrent plan execution needs it's exclusive work area. Work area can be set by `cufftSetWorkArea` function.

1.8. Multiple GPU cuFFT Transforms

cuFFT supports using up to sixteen 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 multiple GPU case. Multiple GPU 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 highest performance is obtained using NVLink interconnect (https://www.nvidia.com/object/nvlink.html). The second best option is using PCI Express 3.0 between the GPUs and ensuring that both GPUs are on the same switch. Note that multiple 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

1.8.1. 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 multiple 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 multiple GPU plan, the returned work area has multiple 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 multiple 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 multiple GPU case.

1.8.2. Helper 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. cuFFT provides functions to assist users in manipulating data on multiple 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 multiple 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 multiple GPU case, cuFFT includes `cufftXtMemcpy()` which allows users to copy between host and multiple 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.

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

For single 2D or 3D transforms on multiple GPUs, when `cufftXtMemcpy()` distributes the data to the GPUs, the array is divided on the X axis. E.G. for two GPUs 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 multiple 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 pointwise 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

1.8.4. Supported Functionality

Starting with cuFFT version 7.0, a subset of single GPU functionality is supported for multiple GPU execution.

Requirements and limitations:

- All GPUs must have the same CUDA architecture level and support Unified Virtual Address Space.
- On Windows, the GPU boards must be operating in Tesla Compute Cluster (TCC) mode.
- For an application that uses the CUDA Driver API, running cuFFT on multiple GPUs is only compatible with applications using the primary context on each GPU.
- Strided input and output are not supported.
- Running cuFFT on more than 8 GPUs (16 GPUs is max) is supported on machines with NVLink only.

While transforms with batch count greater than one do not impose additional constraints, those with a single batch have some restrictions. Single-batch FFTs support only in-place mode, and have additional constraints depending on the FFT type. This behavior is summarized in the following table:

batch=1	1D	2D	3D
C2C/Z2Z	▶ 2,4,8,16 GPUs ▶ power of 2 sizes only ▶ Minimum size for 2-4 GPUs is 64 ▶ Minimum size for 8 GPUs is 128 ▶ Minimum size for 16 GPUs is 1024	▶ 2-16 GPUs ▶ One of the following conditions is met for each dimension: ▶ Dimension must factor into primes less than or equal to 127 ▶ Maximum dimension size is 4096 for single precision ▶ Maximum dimension size is 2048 for double precision ▶ Minimum size is 32 ▶ No LTO callback support	

continues on next page

Table 1 – continued from previous page

batch=1	1D	2D	3D

R2C/D2Z	not supported	▶ 2-16 GPUs
		▶ One of the following conditions is met for each dimension:
		  ▶ Dimension must factor into primes less than or equal to 127
		  ▶ Maximum dimension size is 4096 for single precision
		  ▶ Maximum dimension size is 2048 for double precision
		▶ Minimum size is 32
		▶ Fastest changing dimension size needs to be even
		▶ Supports only `CUFFT_XT_FORMAT_INPLACE` input descriptor format
		▶ No legacy callback / LTO callback support	

C2R/Z2D	not supported	▶ 2-16 GPUs
		▶ One of the following conditions is met for each dimension:
		  ▶ Dimension must factor into primes less than or equal to 127
		  ▶ Maximum dimension size is 4096 for single precision
		  ▶ Maximum dimension size is 2048 for double precision
		▶ Minimum size is 32
		▶ Fastest changing dimension size needs to be even
		▶ Supports only `CUFFT_XT_FORMAT_INPLACE_SHUFFLED` input descriptor format
		▶ No legacy callback / LTO callback support	

General guidelines are:

- Parameter `whichGPUs` of `cufftXtSetGPUs()` function determines ordering of the GPUs with respect to data decomposition (first data chunk is placed on GPU denoted by first element of `whichGPUs`)
- The data for the entire transform must fit within the memory of the GPUs assigned to it.
- For batch size `m` on `n` GPUs :
  - The first `m % n` GPUs execute $\lfloor \frac{m}{n} \rfloor + 1$ transforms.
  - The remaining GPUs execute $\lfloor \frac{m}{n} \rfloor$ transforms.

Batch size output differences:

Single GPU cuFFT results are always returned in natural order. When multiple GPUs are used to perform more than one transform, the results are also returned in natural order. When multiple GPUs are used to perform a single transform the results are returned in a permutation of the normal results to reduce communication time. This behavior is summarized in the following table:

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

To produce natural order results in GPU memory for multi-GPU runs in the 1D single transform case, requires calling `cufftXtMemcpy()` with `CUFFT_COPY_DEVICE_TO_DEVICE`.

2D and 3D multi-GPU 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 single GPU and multiple GPU examples.

1.9. 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.

Note: In CUDA 12.6 Update 2, we introduced support for Link-Time Optimized (LTO) callbacks as a replacement for the deprecated (legacy) callbacks. See more in LTO Load and Store Callback Routines.

Starting from CUDA 11.4, support for callback functionality using separately compiled device code (i.e. legacy callbacks) is deprecated on all GPU architectures. Callback functionality will continue to be supported for all GPU architectures.

1.9.1. Overview of the cuFFT 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 calls 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, cuFFT calls 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 using the extensible plan APIs. After the call to `cufftCreate`, the user may associate a load callback routine, or a store callback routine, or both, with the plan, by:

- Calling `cufftXtSetJITCallback` before `cufftMakePlan`, for LTO callbacks
- Calling `cufftXtSetCallback` after `cufftMakePlan`, for legacy callbacks

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 routines, one for each possible combination of: load or store, real or complex, single precision or double:

- For LTO callbacks, the user must provide an LTO routine that matches the function prototype for the type of routine specified. Otherwise, the planning function `cufftMakePlan` **will fail**.
- For legacy callbacks, 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 handle, the set callback functions 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.
- (For **LTO** callbacks) `cufftXtSetJITCallback()` - set a load and/or store LTO callback for this plan.
- `cufftMakePlan{1d,2d,3d,Many}()` - create the plan. These are the same functions used in the single GPU case.
- (For **legacy** callbacks) `cufftXtSetCallback()` - set a load and/or store legacy 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 LTO callback API is available in the dynamic and static cuFFT libraries on 64 bit Windows and LINUX operating systems. The LTO callback API requires compatible nvJitLink and NVRTC libraries present in the dynamic library path. See more details in LTO Load and Store Callback Routines.

The legacy callback API is available only in the static cuFFT library on 64 bit LINUX operating systems.

1.9.2. LTO Load and Store Callback Routines

LTO callbacks in cuFFT for a given toolkit version require using the nvJitLink library from the same toolkit or greater, but within the same toolkit major.

Additionally, in order to specify custom names for the LTO callback routines, cuFFT requires using the NVRTC library. cuFFT uses NVRTC to compile a minimal wrapper around the user callback with custom symbol name. The custom symbol name provided to the cuFFT API must be a valid, null-terminated C-string containing the unmangled name; currently, keywords that alter the scope of the symbol name (such as `namespace`) or the mangling (such as `extern "C"`) are not supported.

The NVRTC library used must be from a toolkit that is either the same version or older than the nvJitLink library, and both must be from the same toolkit major.

For example, in toolkit version 12.6 cuFFT requires nvJitLink to be from toolkit version 12.X, where X >= 6, and NVRTC to be from toolkit version 12.Y, where 0 <= Y <= X.

Both the nvJitLink and the NVRTC libraries are loaded dynamically, and should be present in the system's dynamic linking path (e.g. LD_LIBRARY_PATH on Unix systems, or PATH on Windows systems).

Code samples for LTO callbacks are available in the public CUDA Library Samples github repository.

1.9.2.1 Specifying LTO Load and Store Callback Routines

Usage of LTO callbacks in cuFFT is divided in two parts:

- Generating the LTO callback (i.e. compiling the callback routine to LTO-IR).
- Associating the LTO callback with the cuFFT plan.

To generate the LTO callback, users can compile the callback device function to LTO-IR using nvcc with any of the supported flags (such as `-dlto` or `-gencode=arch=compute_XX,code=lto_XX`, with XX indicating the target GPU architecture); alternatively, users can generate the LTO callback using NVRTC to do runtime compilation via the `-dlto` flag.

Notice that PTX JIT is part of the JIT LTO kernel finalization trajectory, so architectures older than the current system architecture are supported; users can compile their callback function to LTO-IR for target arch XX and and execute plans which use the callback functions on GPUs with arch YY, where XX <= YY. Please see Compiler Support for Runtime LTO Using nvJitLink Library and Just-in-Time (JIT) Compilation for more details.

As an example, if a user wants to specify a load callback for a R2C transform, they could write the following code

```
__device__  cufftReal myOwnLTOCallback(void *dataIn,
                                       unsigned long long offset,
                                       void *callerInfo,
                                       void *sharedPtr) {
    cufftReal ret;
    // use offset, dataIn, and optionally callerInfo to
    // compute the return value
    return ret;
}
```

To compile the callback to LTO-IR, the user could do

```
# Compile the code to SM60 LTO-IR into a fatbin file
nvcc -gencode=arch=compute_60,code=lto_60 -dc -fatbin callback.cu -o callback.fatbin
#Turn the fatbin data into a C array inside a header, for easy inclusion in host code
bin2c --name my_lto_callback_fatbin --type longlong callback.fatbin > callback_fatbin.
↪h
```

To associate the LTO callback with the cuFFT plan, users can leverage the new API call `cufftXtSetJITCallback()`, which works similarly to `cufftXtSetCallback()`, with a few caveats.

First, `cufftXtSetJITCallback()` must be called after plan creation with `cufftCreate()`, and before calling the plan initialization function with `cufftMakePlan*()` and similar routines.

Second, removing the LTO callback from the plan (using `cufftXtClearCallback()`) is currently not supported. A new plan must be created.

```
#include <cufftXt.h>
#include "callback_fatbin.h"

int main() {
   cufftResult status;
   cufftHandle fft_plan;
   ...

   status = cufftCreate(&fft_plan);

   // NOTE: LTO callbacks must be set before plan creation and cannot be unset (yet)
   size_t lto_callback_fatbin_size = sizeof(my_lto_callback_fatbin);
   status = cufftXtSetJITCallback(fft_plan, "myOwnLTOCallback", (void*)my_lto_
↪callback_fatbin, lto_callback_fatbin_size, CUFFT_CB_LD_REAL, (void **)&device_
↪params));
   status = cufftMakePlan1d(fft_plan, signal_size, CUFFT_C2R, batches, &work_size);
   ...
}
```

1.9.2.2 LTO Callback Routine Function Details

Below are the function prototypes for the user-supplied LTO callback routines that cuFFT calls to load data prior to the transform.

```
typedef  cufftComplex (*cufftJITCallbackLoadC)(void *dataIn,
                                               unsigned long long offset,
                                               void *callerInfo,
                                               void *sharedPointer);

typedef  cufftDoubleComplex (*cufftJITCallbackLoadZ)(void *dataIn,
                                                     unsigned long long offset,
                                                     void *callerInfo,
                                                     void *sharedPointer);

typedef  cufftReal (*cufftJITCallbackLoadR)(void *dataIn,
                                            unsigned long long offset,
                                            void *callerInfo,
                                            void *sharedPointer);

typedef  cufftDoubleReal (*cufftJITCallbackLoadD)(void *dataIn,
                                                  unsigned long long offset,
                                                  void *callerInfo,
                                                  void *sharedPointer);
```

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

- `offset`: offset of the input element from the start of input 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 `cufftXtSetCallback` 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 LTO 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 (*cufftJITCallbackStoreC)(void *dataOut,
                                        unsigned long long offset,
                                        cufftComplex element,
                                        void *callerInfo,
                                        void *sharedPointer);

typedef  void (*cufftJITCallbackStoreZ)(void *dataOut,
                                        unsigned long long offset,
                                        cufftDoubleComplex element,
                                        void *callerInfo,
                                        void *sharedPointer);

typedef  void (*cufftJITCallbackStoreR)(void *dataOut,
                                        unsigned long long offset,
                                        cufftReal element,
                                        void *callerInfo,
                                        void *sharedPointer);

typedef  void (*cufftJITCallbackStoreD)(void *dataOut,
                                        unsigned long long offset,
                                        cufftDoubleReal element,
                                        void *callerInfo,
                                        void *sharedPointer);
```

Parameters for all of the LTO 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 `cufftXtSetCallback` call.
- `sharedPointer`: pointer to shared memory, valid only if the user has called `cufftXtSetCallbackSharedSize()`.

1.9.3. Legacy Load and Store Callback Routines

1.9.3.1 Specifying Legacy Load and Store Callback Routines

In order to associate a legacy 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 legacy 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. Please note that `__managed__` variables are not suitable to pass to `cufftSetCallback` due to restrictions on variable usage (See the *NVIDIA CUDA Programming Guide* for more information about `__managed__` variables).

1.9.3.2 Legacy Callback Routine Function Details

Below are the function prototypes, and typedefs for pointers to the user supplied legacy 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 legacy load callbacks are defined as below:

- `offset`: offset of the input element from the start of input 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 `cufftXtSetCallback` 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 legacy 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 legacy 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 `cufftXtSetCallback` call.
- `sharedPointer`: pointer to shared memory, valid only if the user has called `cufftXtSetCallbackSharedSize()`.

1.9.4. Coding Considerations for the cuFFT Callback Routine Feature

cuFFT supports callbacks on all types of transforms, dimension, batch, or stride between elements. Callbacks are supported for transforms of single and double precision.

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.

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).

One common use of callbacks is to reduce the amount of data read or written to memory, either by selective filtering or via type conversions. When more than one kernel are used to implement a transform, cuFFT alternates using the workspace and the output buffer to write intermediate results. This means that the output buffer must always be large enough to accommodate the entire transform.

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 where it is safe to call the `__syncthreads` function from within the callback routine. 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.

Note that there are no guarantees on the relative order of execution of blocks within a grid. As such, callbacks should not rely on any particular ordering within a kernel. For instance, reordering data (such as an FFT-shift) could rely on the order of execution of the blocks. Results in this case would be undefined.

1.9.4.1 Coding Considerations for LTO Callback Routines

cuFFT will call the LTO load callback routine, for each point in the input, once and only once for real-to-complex (R2C, D2Z) and complex-to-complex (C2C, Z2Z) transforms. Unlike with legacy callbacks, **LTO load callbacks may be called more than once per element for complex-to-real** (C2R, Z2D) transforms. The input value will not be updated twice (i.e. the transformed value will be stored in register and not memory, even for in-place transforms), but users should not rely on the amount of calls per element in their callback device functions.

Similarly to legacy callbacks, LTO store callbacks will be called once and only once for each point in the output. If the transform is being done in-place (i.e. the input and output data are in the same memory location) the store callback for a given element cannot overwrite other elements. It can either overwrite the given element, or write in a completely distinct output buffer.

cuFFT does not support LTO callbacks for multi-GPU transforms (yet).

1.9.4.2 Coding Considerations for Legacy Callback Routines

cuFFT supports legacy callbacks on any number of GPUs.

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 the transform is being done in-place (i.e. the input and output data are in the same memory location) the store callback for a given element cannot overwrite other elements. It can either overwrite the given element, or write in a completely distinct output buffer.

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.

1.10. Thread Safety

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

1.11. CUDA Graphs Support

Using CUDA Graphs with cuFFT is supported on single GPU plans. It is also supported on multiple GPU plans starting with cuFFT version 10.4.0. The stream associated with a cuFFT plan must meet the requirements stated in Creating a Graph Using Stream Capture.

Note: Starting from CUDA 11.8 (including CUDA 12.0 onward), CUDA Graphs are no longer supported for legacy callback routines that load data in out-of-place mode transforms. Starting from CUDA 12.6 Update 2, LTO callbacks can be used as a replacement for legacy callbacks without this limitation. cuFFT deprecated callback functionality based on separate compiled device code (legacy callbacks) in cuFFT 11.4.

1.12. 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. 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:

```
nvcc mCufftApp.c  -lcufft  -o myCufftApp
```

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

```
nvcc mCufftwApp.c  -lcufftw  -lcufft  -o myCufftwApp
```

Whereas to compile against the static cuFFT library, extra steps need to be taken. The library needs to be device linked. It may happen during building and linking of a simple program, or as a separate step. The entire process is described in Using Separarate Compilation in CUDA.

For cuFFT and cufftw in version 9.0 or later any supported architecture can be used to do the device linking:

Static cuFFT compilation command:

```
nvcc mCufftApp.c  -lcufft_static   -lculibos -o myCufftApp
```

Static cufftw compilation command:

```
nvcc mCufftwApp.c   -lcufftw_static  -lcufft_static   -lculibos  -o myCufftwApp
```

Prior to version 9.0 proper linking required specifying a subset of supported architectures, as shown in the following commands:

Static cuFFT compilation command:

```
nvcc mCufftApp.c  -lcufft_static   -lculibos -o myCufftApp\
    -gencode arch=compute_20,\"code=sm_20\"\
    -gencode arch=compute_30,\"code=sm_30\"\
    -gencode arch=compute_35,\"code=sm_35\"\
    -gencode arch=compute_50,\"code=sm_50\"\
    -gencode arch=compute_60,\"code=sm_60\"\
    -gencode arch=compute_60,\"code=compute_60\"
```

Static cufftw compilation command:

```
nvcc mCufftwApp.c    -lcufftw_static  -lcufft_static   -lculibos  -o myCufftwApp\
    -gencode arch=compute_20,\"code=sm_20\"\
    -gencode arch=compute_30,\"code=sm_30\"\
    -gencode arch=compute_35,\"code=sm_35\"\
    -gencode arch=compute_50,\"code=sm_50\"\
    -gencode arch=compute_60,\"code=sm_60\"\
    -gencode arch=compute_60,\"code=compute_60\"
```

Please note that the cuFFT library might not contain code for certain architectures as long as there is code for a lower architecture that is binary compatibile (e.g. SM52, SM61). This is reflected in link commands above and significant when using versions prior r9.0. To determine if a specific SM is included in the cuFFT library, one may use `cuobjdump` utility. For example, if you wish to know if SM_50 is included, the command to run is `cuobjdump -arch sm_50 libcufft_static.a`. Some kernels are built only on select architectures (e.g. kernels with half precision arithmetics are present only for SM53 and above). This can cause warnings at link time that architectures are missing from these kernels. These warnings can be safely ignored.

It is also possible to use the native Host C++ compiler and perform device link as a separate step. Please consult NVCC documentation for more details. Depending on the Host Operating system, some additional libraries like `pthread` or `dl` might be needed on the linking line.

Note that in this case, the library `cuda` is not needed. The CUDA Runtime will try to open explicitly the `cuda` library if needed. In the case of a system which does not have the CUDA driver installed, this allows the application to gracefully manage this issue and potentially run if a CPU-only path is available.

The cuFFT static library supports user supplied legacy callback routines. The legacy callback routines are CUDA device code, and must be separately compiled with NVCC and linked with the cuFFT library. Please refer to the NVCC documentation regarding separate compilation for details. If you specify an SM when compiling your callback functions, you must specify one of the SM's cuFFT includes.

1.12.1. Static library without legacy callback support

Starting with cuFFT version 9.2, a new variant of the cuFTT static library, `libcufft_static_nocallback.a`, was added. This new version does not contain legacy callback functionality and can be linked using the host compiler only.

1.13. 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).

Half precision transforms might not be suitable for all kinds of problems due to limited range represented by half precision floating point arithmetics. Please note that the first element of FFT result is the sum of all input elements and it is likely to overflow for certain inputs.

Results produced by the cuFFT library are deterministic (ie, bitwise reproducible) as long as the following are kept constant between runs: plan input parameters, cuFFT version, and GPU model.

cuFFT batched plans require that input data includes valid signal for all batches. Performance optimizations in batched mode can combine signal from different batches for processing. Optimizations used in cuFFT can vary from version to version.

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.

1.14. Caller Allocated Work Area Support

cuFFT plans may use additional memory to store intermediate results. The cuFFT library offers several functions to manage this temporary memory utilization behavior:

- `cufftSetAutoAllocation`
- `cufftEstimate1d`, `cufftEstimate2d`, `cufftEstimate3d` and `cufftEstimateMany`
- `cufftGetSize`
- `cufftXtSetWorkAreaPolicy`

The first two functions manage allocation and ownership of temporary memory. By default cuFFT always allocates its own work area in GPU memory. Each cuFFT handle allocates data separately. If multiple cuFFT plans are to be launched sequentially it is possible to assign the same memory chunk as work area to all those plans and reduce memory overhead.

The memory assigned as work area needs to be GPU visible. In addition to the regular memory acquired with `cudaMalloc`, usage of CUDA Unified Virtual Addressing enables cuFFT to use the following types of memory as work area memory: pinned host memory, managed memory, memory on GPU other than the one performing the calculations. While this provides flexibility, it comes with a performance penalty whose magnitude depends on the available memory bandwidth.

The `cufftEstimateNd`, `cufftEstimateMany`, and `cufftGetSize` functions provide information about the required memory size for cases where the user is allocating the work space buffer.

In version 9.2 cuFFT also introduced the `cufftXtSetWorkAreaPolicy` function. This function allows fine tuning of work area memory usage.
cuFFT 9.2 version supports only the `CUFFT_WORKAREA_MINIMAL` policy, which instructs cuFFT to re-plan the existing plan without the need to use work area memory.

Also as of cuFFT 9.2, supported FFT transforms that allow for `CUFFT_WORKAREA_MINIMAL` policy are as follows:

- Transforms of type C2C are supported with sizes up to 4096 in any dimension.
- Transforms of type Z2Z are supported with sizes up to 2048 in any dimension.
- Only single GPU transforms are supported.

Depending on the FFT transform size, a different FFT algorithm may be used when the `CUFFT_WORKAREA_MINIMAL` policy is set.

1.15. cuFFT Link-Time Optimized Kernels

Starting from CUDA 12.4, cuFFT ships Link-Time Optimized (LTO) kernels. These kernels are linked and finalized at runtime as part of the cuFFT planning routines. This enables the cuFFT library to generate kernels optimized for the underlying architecture and the specific problem to solve.

The current LTO kernel coverage includes:

- Kernels for 64-bit addressing (with FFTs spanning addresses greater than 2^(32)-1 elements).
- Some single- and double-precision R2C and C2R sizes.

The number and coverage of LTO kernels will grow with future releases of cuFFT. We encourage our users to test whether LTO kernels improve the performance for their use case.

Users can opt-in into LTO kernels by setting the `NVFFT_PLAN_PROPERTY_INT64_PATIENT_JIT` plan property using the `cufftSetPlanProperty` routine.

In order to finalize LTO kernels, cuFFT relies on the nvJitLink library that ships as part of the CUDA Toolkit. Finalizing the kernels at runtime can cause an **increase in planning time** (which could be in the order of hundreds of milliseconds, depending on the cuFFT plan and hardware characteristics of the host system), in exchange for faster execution time of the optimized kernels. Note that nvJitLink caches kernels linked at runtime to speed-up subsequent kernel finalizations in repeated planning routines.

If for any reason the runtime linking of the kernel fails, cuFFT will fall back to offline-compiled kernels to compute the FFT.

Note: cuFFT LTO kernels for a given toolkit version require using the nvJitLink library from the same toolkit or greater, but within the same toolkit major. For example, cuFFT in 12.4 requires nvJitLink to be from a CUDA Toolkit 12.X, with `X >= 4`.

The nvJitLink library is loaded dynamically, and should be present in the system's dynamic linking path (e.g. `LD_LIBRARY_PATH` on Unix systems, or `PATH` on Windows systems).

Chapter 2. 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.

2.1. 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
    CUFFT_NOT_IMPLEMENTED = 14, // Function does not implement functionality for
↪parameters given.
    CUFFT_LICENSE_ERROR  = 15, // Used in previous versions.
    CUFFT_NOT_SUPPORTED  = 16  // Operation is not supported for parameters given.
} cufftResult;
```

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

2.2. cuFFT Basic Plans

These API routines take care of initializing the cufftHandle. Any already-initialized handle attributes passed to the planning functions will be ignored.

2.2.1. 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.

This call can only be used once for a given handle. It will fail and return `CUFFT_INVALID_PLAN` if the plan is locked, i.e. the handle was previously used with a different `cufftPlan` or `cufftMakePlan` call.

Parameters

- **`plan[In]`** – Pointer to an uninitialized `cufftHandle` object.
- **`nx[In]`** – The transform size (e.g. 256 for a 256-point FFT).
- **`type[In]`** – The transform data type (e.g., `CUFFT_C2C` for single precision complex to complex).
- **`batch[In]`** – Number of transforms of size `nx`. Please consider using `cufftPlanMany` for multiple transforms.
- **`plan[Out]`** – Contains a cuFFT 1D plan handle value.

Return values

- **`CUFFT_SUCCESS`** – cuFFT successfully created the FFT plan.
- **`CUFFT_INVALID_PLAN`** – The `plan` parameter is not a valid handle. Handle is not valid when the plan is locked.
- **`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.

2.2.2. cufftPlan2d()

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

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

This call can only be used once for a given handle. It will fail and return `CUFFT_INVALID_PLAN` if the plan is locked, i.e. the handle was previously used with a different `cufftPlan` or `cufftMakePlan` call.

Parameters

- **`plan[In]`** – Pointer to an uninitialized `cufftHandle` object.
- **`nx[In]`** – The transform size in the *x* dimension This is slowest changing dimension of a transform (strided in memory).
- **`ny[In]`** – The transform size in the *y* dimension. This is fastest changing dimension of a transform (contiguous in memory).
- **`type[In]`** – The transform data type (e.g., `CUFFT_C2R` for single precision complex to real).
- **`plan[Out]`** – Contains a cuFFT 2D plan handle value.

Return values

- **`CUFFT_SUCCESS`** – cuFFT successfully created the FFT plan.
- **`CUFFT_INVALID_PLAN`** – The `plan` parameter is not a valid handle. Handle is not valid when the plan is locked.
- **`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.

2.2.3. 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`.

This call can only be used once for a given handle. It will fail and return `CUFFT_INVALID_PLAN` if the plan is locked, i.e. the handle was previously used with a different `cufftPlan` or `cufftMakePlan` call.

Parameters

- **`plan[In]`** – Pointer to an uninitialized `cufftHandle` object.

- **nx[In]** – The transform size in the *x* dimension. This is slowest changing dimension of a transform (strided in memory).
- **ny[In]** – The transform size in the *y* dimension.
- **nz[In]** – The transform size in the *z* dimension. This is fastest changing dimension of a transform (contiguous in memory).
- **type[In]** – The transform data type (e.g., CUFFT_R2C for single precision real to complex).
- **plan[Out]** – Contains a cuFFT 3D plan handle value.

Return values

- **CUFFT_SUCCESS** – cuFFT successfully created the FFT plan.
- **CUFFT_INVALID_PLAN** – The plan parameter is not a valid handle. Handle is not valid when the plan is locked.
- **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.

2.2.4. 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.

If inembed and onembed are set to NULL, all other stride information is ignored, and default strides are used. The default assumes contiguous data arrays.

All arrays are assumed to be in CPU memory.

Please note that behavior of cufftPlanMany function when inembed and onembed is NULL is different than corresponding function in FFTW library fftw_plan_many_dft.

This call can only be used once for a given handle. It will fail and return CUFFT_INVALID_PLAN if the plan is locked, i.e. the handle was previously used with a different cufftPlan or cufftMakePlan call.

Parameters

- **plan[In]** – Pointer to an uninitialized cufftHandle object.

- **rank[In]** – Dimensionality of the transform (1, 2, or 3).
- **n[In]** – Array of size `rank`, describing the size of each dimension, `n[0]` being the size of the outermost and `n[rank-1]` innermost (contiguous) dimension of a transform.
- **inembed[In]** – 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[In]** – Indicates the distance between two successive input elements in the least significant (i.e., innermost) dimension.
- **idist[In]** – Indicates the distance between the first element of two consecutive signals in a batch of the input data.
- **onembed[In]** – 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[In]** – Indicates the distance between two successive output elements in the output array in the least significant (i.e., innermost) dimension.
- **odist[In]** – Indicates the distance between the first element of two consecutive signals in a batch of the output data.
- **type[In]** – The transform data type (e.g., `CUFFT_R2C` for single precision real to complex).
- **batch[In]** – Batch size for this transform.
- **plan[Out]** – Contains a cuFFT plan handle.

Return values

- **CUFFT_SUCCESS** – cuFFT successfully created the FFT plan.
- **CUFFT_INVALID_PLAN** – The `plan` parameter is not a valid handle. Handle is not valid when the plan is locked.
- **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.

2.3. cuFFT Extensible Plans

These API routines 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.

2.3.1. 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.

Parameters

- **plan[In]** – Pointer to a `cufftHandle` object.
- **plan[Out]** – 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.

2.3.2. 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. In the case of multi-GPU plans, the plan created first should be destroyed last.

Parameters

- **plan[In]** – 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.

2.3.3. 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.

This call can only be used once for a given handle. It will fail and return `CUFFT_INVALID_PLAN` if the plan is locked, i.e. the handle was previously used with a different `cufftPlan` or `cufftMakePlan` call.

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

Parameters

- **`plan[In]`** – `cufftHandle` returned by `cufftCreate`.
- **`nx[In]`** – The transform size (e.g. 256 for a 256-point FFT). For multiple GPUs, this must be a power of 2.
- **`type[In]`** – The transform data type (e.g., `CUFFT_C2C` for single precision complex to complex). For multiple GPUs this must be a complex to complex transform.
- **`batch[In]`** – Number of transforms of size `nx`. Please consider using `cufftMakePlanMany` for multiple transforms.
- **`*workSize[In]`** – Pointer to the size(s), in bytes, of the work areas. For example for two GPUs worksize must be declared to have two elements.
- **`*workSize[Out]`** – 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. Handle is not valid when the plan is locked or multi-GPU restrictions are not met.
- **`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.

2.3.4. 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.

This call can only be used once for a given handle. It will fail and return `CUFFT_INVALID_PLAN` if the plan is locked, i.e. the handle was previously used with a different `cufftPlan` or `cufftMakePlan` call.

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

Parameters

- **`plan[In]`** – `cufftHandle` returned by `cufftCreate`.
- **`nx[In]`** – The transform size in the *x* dimension. This is slowest changing dimension of a transform (strided in memory). For multiple GPUs, this must be factorable into primes less than or equal to 127.
- **`ny[In]`** – The transform size in the *y* dimension. This is fastest changing dimension of a transform (contiguous in memory). For 2 GPUs, this must be factorable into primes less than or equal to 127.
- **`type[In]`** – The transform data type (e.g., `CUFFT_C2R` for single precision complex to real).
- **`workSize[In]`** – Pointer to the size(s), in bytes, of the work areas. For example for two GPUs worksize must be declared to have two elements.
- **`*workSize[Out]`** – 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.

2.3.5. 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`.

This call can only be used once for a given handle. It will fail and return `CUFFT_INVALID_PLAN` if the plan is locked, i.e. the handle was previously used with a different `cufftPlan` or `cufftMakePlan` call.

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

Parameters

- **plan[In]** – `cufftHandle` returned by `cufftCreate`.
- **nx[In]** – The transform size in the *x* dimension. This is slowest changing dimension of a transform (strided in memory). For multiple GPUs, this must be factorable into primes less than or equal to 127.
- **ny[In]** – The transform size in the *y* dimension. For multiple GPUs, this must be factorable into primes less than or equal to 127.
- **nz[In]** – The transform size in the *z* dimension. This is fastest changing dimension of a transform (contiguous in memory). For multiple GPUs, this must be factorable into primes less than or equal to 127.
- **type[In]** – The transform data type (e.g., `CUFFT_R2C` for single precision real to complex).
- **workSize[In]** – Pointer to the size(s), in bytes, of the work areas. For example for two GPUs worksize must be declared to have two elements.
- ***workSize[Out]** – 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.

2.3.6. 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 `inembed` and `onembed` are set to `NULL`, all other stride information is ignored, and default strides are used. The default assumes contiguous data arrays.

This call can only be used once for a given handle. It will fail and return `CUFFT_INVALID_PLAN` if the plan is locked, i.e. the handle was previously used with a different `cufftPlan` or `cufftMakePlan` call.

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

All arrays are assumed to be in CPU memory.

Parameters

- **`plan[In]`** – `cufftHandle` returned by `cufftCreate`.
- **`rank[In]`** – Dimensionality of the transform (1, 2, or 3)
- **`n[In]`** – Array of size `rank`, describing the size of each dimension, `n[0]` being the size of the outermost and `n[rank-1]` innermost (contiguous) dimension of a transform. For multiple GPUs and rank equal to 1, the sizes must be a power of 2. For multiple GPUs and rank equal to 2 or 3, the sizes must be factorable into primes less than or equal to 127.
- **`inembed[In]`** – Pointer of size `rank` that indicates the storage dimensions of the input data in memory, `inembed[0]` being the storage dimension of the outermost dimension. If set to NULL all other advanced data layout parameters are ignored.
- **`istride[In]`** – Indicates the distance between two successive input elements in the least significant (i.e., innermost) dimension
- **`idist[In]`** – Indicates the distance between the first element of two consecutive signals in a batch of the input data
- **`onembed[In]`** – Pointer of size `rank` that indicates the storage dimensions of the output data in memory, `onembed[0]` being the storage dimension of the outermost dimension. If set to NULL all other advanced data layout parameters are ignored.
- **`ostride[In]`** – Indicates the distance between two successive output elements in the output array in the least significant (i.e., innermost) dimension
- **`odist[In]`** – Indicates the distance between the first element of two consecutive signals in a batch of the output data
- **`type[In]`** – 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[In]** – Batch size for this transform.
- ***workSize[In]** – Pointer to the size(s), in bytes, of the work areas. For example for two GPUs worksize must be declared to have two elements.
- ***workSize[Out]** – 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. Handle is not valid when the plan is locked or multi-GPU restrictions are not met.
- **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.

2.3.7. cufftMakePlanMany64()

cufftResult **cufftMakePlanMany64**(*cufftHandle* plan, int rank, long long int *n, long long int *inembed, long long int istride, long long int idist, long long int *onembed, long long int ostride, long long int odist, cufftType type, long long 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.

This API is identical to `cufftMakePlanMany` except that the arguments specifying sizes and strides are 64 bit integers. This API makes very large transforms possible. cuFFT includes kernels that use 32 bit indexes, and kernels that use 64 bit indexes. cuFFT planning selects 32 bit kernels whenever possible to avoid any overhead due to 64 bit arithmetic.

All sizes and types of transform are supported by this interface, with two exceptions. For transforms whose size exceeds 4G elements, the dimensions specified in the array `n` must be factorable into primes that are less than or equal to 127. For real to complex and complex to real transforms whose size exceeds 4G elements, the fastest changing dimension must be even.

The `cufftPlanMany64()` API supports more complicated input and output data layouts via the advanced data layout parameters: `inembed`, `istride`, `idist`, `onembed`, `ostride`, and `odist`.

If `inembed` and `onembed` are set to `NULL`, all other stride information is ignored, and default strides are used. The default assumes contiguous data arrays.

This call can only be used once for a given handle. It will fail and return `CUFFT_INVALID_PLAN` if the plan is locked, i.e. the handle was previously used with a different `cufftPlan` or `cufftMakePlan` call.

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

All arrays are assumed to be in CPU memory.

Parameters

- **plan[In]** – `cufftHandle` returned by `cufftCreate`.
- **rank[In]** – Dimensionality of the transform (1, 2, or 3).
- **n[In]** – Array of size `rank`, describing the size of each dimension. For multiple GPUs and rank equal to 1, the sizes must be a power of 2. For multiple GPUs and rank equal to 2 or 3, the sizes must be factorable into primes less than or equal to 127.
- **inembed[In]** – 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[In]** – Indicates the distance between two successive input elements in the least significant (i.e., innermost) dimension.
- **idist[In]** – Indicates the distance between the first element of two consecutive signals in a batch of the input data.
- **onembed[In]** – 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[In]** – Indicates the distance between two successive output elements in the output array in the least significant (i.e., innermost) dimension.
- **odist[In]** – Indicates the distance between the first element of two consecutive signals in a batch of the output data.
- **type[In]** – 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[In]** – Batch size for this transform.
- ***workSize[In]** – Pointer to the size(s), in bytes, of the work areas. For example for two GPUs worksize must be declared to have two elements.
- ***workSize[Out]** – 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. Handle is not valid when the plan is locked or multi-GPU restrictions are not met.
- **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.

2.3.8. cufftXtMakePlanMany()

cufftResult **cufftXtMakePlanMany**(*cufftHandle* plan, int rank, long long int *n, long long int *inembed, long long int istride, long long int idist, cudaDataType inputtype, long long int *onembed, long long int ostride, long long int odist, cudaDataType outputtype, long long int batch, size_t *workSize, cudaDataType executiontype);

Following a call to `cufftCreate()` makes an 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.

Type specifiers `inputtype`, `outputtype` and `executiontype` dictate type and precision of transform to be performed. Not all combinations of parameters are supported. Currently all three parameters need to match precision. Parameters `inputtype` and `outputtype` need to match transform type complex-to-complex, real-to-complex or complex-to-real. Parameter `executiontype` needs to match precision and be of a complex type. Example: for a half-precision real-to-complex transform, parameters `inputtype`, `outputtype` and `executiontype` would have values of `CUDA_R_16F`, `CUDA_C_16F` and `CUDA_C_16F` respectively. Similarly, a bfloat16 complex-to-real transform would use `CUDA_C_16BF` for `inputtype` and `executiontype`, and `CUDA_R_16BF` for `outputtype`.

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

If `inembed` and `onembed` are set to `NULL`, all other stride information is ignored, and default strides are used. The default assumes contiguous data arrays.

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

All arrays are assumed to be in CPU memory.

Parameters

- **`plan[In]`** – `cufftHandle` returned by `cufftCreate`.
- **`rank[In]`** – Dimensionality of the transform (1, 2, or 3).
- **`n[In]`** – Array of size `rank`, describing the size of each dimension, `n[0]` being the size of the outermost and `n[rank-1]` innermost (contiguous) dimension of a transform. For multiple GPUs and rank equal to 1, the sizes must be a power of 2. For multiple GPUs and rank equal to 2 or 3, the sizes must be factorable into primes less than or equal to 127.
- **`inembed[In]`** – Pointer of size `rank` that indicates the storage dimensions of the input data in memory, `inembed[0]` being the storage dimension of the outermost dimension. If set to NULL all other advanced data layout parameters are ignored.
- **`istride[In]`** – Indicates the distance between two successive input elements in the least significant (i.e., innermost) dimension.
- **`idist[In]`** – Indicates the distance between the first element of two consecutive signals in a batch of the input data.
- **`inputtype[In]`** – Type of input data.

- **onembed[In]** – Pointer of size `rank` that indicates the storage dimensions of the output data in memory, `onembed[0]` being the storage dimension of the outermost dimension. If set to NULL all other advanced data layout parameters are ignored.
- **ostride[In]** – Indicates the distance between two successive output elements in the output array in the least significant (i.e., innermost) dimension.
- **odist[In]** – Indicates the distance between the first element of two consecutive signals in a batch of the output data.
- **outputtype[In]** – Type of output data.
- **batch[In]** – Batch size for this transform.
- ***workSize[In]** – Pointer to the size(s), in bytes, of the work areas. For example for two GPUs worksize must be declared to have two elements.
- **executiontype[In]** – Type of data to be used for computations.
- ***workSize[Out]** – 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. Handle is not valid when multi-GPU restrictions are not met.
- **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.

2.4. cuFFT Plan Properties

Users can further customize cuFFT plans using plan properties. These properties can be set, queried and reset on a per-plan basis as needed, using the routines listed in this section.

The current supported properties are listed below:

Property	Underlying Type	Description	Behavior
NVFFT_PLAN_PROPERTY_INT64_PATIENT_JIT	long long int	▶ Runtime LTO kernels are enabled when set to not-zero value. See Link-Time Optimized Kernels ▶ Runtime LTO kernles are disabled when set to zero (default)	▶ Can be set / reset before planning ▶ Cannot be set / reset after planning

2.4.1. cufftSetPlanPropertyInt64()

cufftResult **cufftSetPlanPropertyInt64**(*cufftHandle* plan, cufftProperty property, const long long int propertyValueInt64);

Associates a cuFFT plan with a property identified by the key `property`. The value for the property is given by value `propertyValueInt64`, which is a signed long long integer.

Parameters

- **plan[In]** – `cufftHandle` returned by `cufftCreate`.
- **property[In]** – The property identifier, of type `cufftPlanProperty`.
- **propertyValueInt64[In]** – Value to set for the property, a long long signed integer.

Return values

- **CUFFT_SUCCESS** – cuFFT successfully set the property.
- **CUFFT_INVALID_PLAN** – The `plan` parameter is not a valid handle.
- **CUFFT_NOT_SUPPORTED** – The property is not supported, or it cannot be set at the time (e.g. some properties cannot be set after calling a planning routine for the plan, see cuFFT Plan Properties).
- **CUFFT_INVALID_VALUE** – Invalid property or value with which to set the property

2.4.2. cufftGetPlanPropertyInt64()

cufftResult **cufftGetPlanPropertyInt64**(*cufftHandle* plan, cufftProperty property, long long int *propertyValueInt64);

Retrieves the property value identified by the key `property` associated with the cuFFT plan `plan`. The value for the property, which is a signed long long integer, is set in the address space pointed by `propertyValueInt64`.

Parameters

- **plan[In]** – `cufftHandle` returned by `cufftCreate`.
- **property[In]** – The property identifier, of type `cufftPlanProperty`.
- **propertyValueInt64[In]** – Pointer to the value to be set with the value of the property.

Return values

- **CUFFT_SUCCESS** – cuFFT successfully retrieved the property value.
- **CUFFT_INVALID_PLAN** – The `plan` parameter is not a valid handle.
- **CUFFT_NOT_SUPPORTED** – The property is not supported.
- **CUFFT_INVALID_VALUE** – Invalid property, or pointer `propertyValueInt64` is null

2.4.3. cufftResetPlanProperty()

cufftResult **cufftResetPlanProperty**(*cufftHandle* plan, cufftProperty property);

Resets the value of the property identified by the key `property`, associated with the cuFFT plan `plan`, to its default value.

Parameters

- **plan[In]** – `cufftHandle` returned by `cufftCreate`.
- **property[In]** – The property identifier, of type `cufftPlanProperty`.

Return values

- **CUFFT_SUCCESS** – cuFFT successfully reset the property value.
- **CUFFT_INVALID_PLAN** – The `plan` parameter is not a valid handle.
- **CUFFT_NOT_SUPPORTED** – The property is not supported for `plan`, or cannot be reset at present time (see Behavior column on cuFFT Plan Properties).
- **CUFFT_INVALID_VALUE** – Invalid property

2.5. 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.

2.5.1. 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.

Parameters

- **`nx[In]`** – The transform size (e.g. 256 for a 256-point FFT).
- **`type[In]`** – The transform data type (e.g., CUFFT_C2C for single precision complex to complex).
- **`batch[In]`** – Number of transforms of size nx. Please consider using `cufftEstimateMany` for multiple transforms.
- **`*workSize[In]`** – Pointer to the size, in bytes, of the work space.
- **`*workSize[Out]`** – 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.

2.5.2. 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.

Parameters

- **nx[In]** – The transform size in the *x* dimension (number of rows).
- **ny[In]** – The transform size in the *y* dimension (number of columns).
- **type[In]** – The transform data type (e.g., CUFFT_C2R for single precision complex to real).
- ***workSize[In]** – Pointer to the size, in bytes, of the work space.
- ***workSize[Out]** – 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.

2.5.3. 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.

Parameters

- **nx[In]** – The transform size in the *x* dimension.
- **ny[In]** – The transform size in the *y* dimension.
- **nz[In]** – The transform size in the *z* dimension.
- **type[In]** – The transform data type (e.g., CUFFT_R2C for single precision real to complex).
- ***workSize[In]** – Pointer to the size, in bytes, of the work space.
- ***workSize[Out]** – 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.

2.5.4. 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.

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.

Parameters

- **rank[In]** – Dimensionality of the transform (1, 2, or 3).
- **n[In]** – Array of size `rank`, describing the size of each dimension.
- **inembed[In]** – 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[In]** – Indicates the distance between two successive input elements in the least significant (i.e., innermost) dimension.
- **idist[In]** – Indicates the distance between the first element of two consecutive signals in a batch of the input data.
- **onembed[In]** – 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[In]** – Indicates the distance between two successive output elements in the output array in the least significant (i.e., innermost) dimension.
- **odist[In]** – Indicates the distance between the first element of two consecutive signals in a batch of the output data.
- **type[In]** – The transform data type (e.g., `CUFFT_R2C` for single precision real to complex).

- **batch[In]** – Batch size for this transform.
- ***workSize[In]** – Pointer to the size, in bytes, of the work space.
- ***workSize[Out]** – 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.

2.6. 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.

2.6.1. 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.

Parameters

- **plan[In]** – `cufftHandle` returned by `cufftCreate`.
- **nx[In]** – The transform size (e.g. 256 for a 256-point FFT).
- **type[In]** – The transform data type (e.g., `CUFFT_C2C` for single precision complex to complex).
- **batch[In]** – Number of transforms of size `nx`. Please consider using `cufftGetSizeMany` for multiple transforms.
- ***workSize[In]** – Pointer to the size(s), in bytes, of the work areas. For example for two GPUs worksize must be declared to have two elements.
- ***workSize[Out]** – 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.

2.6.2. 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.

Parameters

- **plan[In]** – `cufftHandle` returned by `cufftCreate`.
- **nx[In]** – The transform size in the *x* dimension (number of rows).
- **ny[In]** – The transform size in the *y* dimension (number of columns).
- **type[In]** – The transform data type (e.g., `CUFFT_C2R` for single precision complex to real).
- ***workSize[In]** – Pointer to the size(s), in bytes, of the work areas. For example for two GPUs worksize must be declared to have two elements.
- ***workSize[Out]** – 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.

2.6.3. 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.

Parameters

- **plan[In]** – `cufftHandle` returned by `cufftCreate`.
- **nx[In]** – The transform size in the *x* dimension.
- **ny[In]** – The transform size in the *y* dimension.
- **nz[In]** – The transform size in the *z* dimension.
- **type[In]** – The transform data type (e.g., `CUFFT_R2C` for single precision real to complex).
- ***workSize[In]** – Pointer to the size(s), in bytes, of the work areas. For example for two GPUs worksize must be declared to have two elements.
- ***workSize[Out]** – 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.

2.6.4. 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.

Parameters

- **plan[In]** – `cufftHandle` returned by `cufftCreate`.

- **rank[In]** – Dimensionality of the transform (1, 2, or 3).
- **n[In]** – Array of size rank, describing the size of each dimension.
- **inembed[In]** – 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[In]** – Indicates the distance between two successive input elements in the least significant (i.e., innermost) dimension.
- **idist[In]** – Indicates the distance between the first element of two consecutive signals in a batch of the input data.
- **onembed[In]** – 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[In]** – Indicates the distance between two successive output elements in the output array in the least significant (i.e., innermost) dimension.
- **odist[In]** – Indicates the distance between the first element of two consecutive signals in a batch of the output data.
- **type[In]** – The transform data type (e.g., CUFFT_R2C for single precision real to complex).
- **batch[In]** – Batch size for this transform.
- ***workSize[In]** – Pointer to the size(s), in bytes, of the work areas. For example for two GPUs worksize must be declared to have two elements.
- ***workSize[Out]** – 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.

2.6.5. cufftGetSizeMany64()

cufftResult **cufftGetSizeMany64**(*cufftHandle* plan, int rank, long long int *n, long long int *inembed, long long int istride, long long int idist, long long int *onembed, long long int ostride, long long int odist, cufftType type, long long 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.

This API is identical to `cufftMakePlanMany` except that the arguments specifying sizes and strides are 64 bit integers. This API makes very large transforms possible. cuFFT includes kernels that use 32 bit indexes, and kernels that use 64 bit indexes. cuFFT planning selects 32 bit kernels whenever possible to avoid any overhead due to 64 bit arithmetic.

All sizes and types of transform are supported by this interface, with two exceptions. For transforms whose total size exceeds 4G elements, the dimensions specified in the array `n` must be factorable into primes that are less than or equal to 127. For real to complex and complex to real transforms whose total size exceeds 4G elements, the fastest changing dimension must be even.

Parameters

- **`plan[In]`** – `cufftHandle` returned by `cufftCreate`.
- **`rank[In]`** – Dimensionality of the transform (1, 2, or 3).
- **`n[In]`** – Array of size `rank`, describing the size of each dimension.
- **`inembed[In]`** – 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[In]`** – Indicates the distance between two successive input elements in the least significant (i.e., innermost) dimension.
- **`idist[In]`** – Indicates the distance between the first element of two consecutive signals in a batch of the input data.
- **`onembed[In]`** – 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[In]`** – Indicates the distance between two successive output elements in the output array in the least significant (i.e., innermost) dimension.
- **`odist[In]`** – Indicates the distance between the first element of two consecutive signals in a batch of the output data.
- **`type[In]`** – The transform data type (e.g., `CUFFT_R2C` for single precision real to complex).
- **`batch[In]`** – Batch size for this transform.
- **`*workSize[In]`** – Pointer to the size(s), in bytes, of the work areas. For example for two GPUs worksize must be declared to have two elements.
- **`*workSize[Out]`** – 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.

2.6.6. cufftXtGetSizeMany()

cufftResult **cufftXtGetSizeMany**(*cufftHandle* plan, int rank, long long int *n, long long int *inembed, long long int istride, long long int idist, cudaDataType inputtype, long long int *onembed, long long int ostride, long long int odist, cudaDataType outputtype, long long int batch, size_t *workSize, cudaDataType executiontype);

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

For more information about valid combinations of `inputtype`, `outputtype` and `executiontype` parameters please refer to documentation of `cufftXtMakePlanMany` function.

Parameters

- **plan[In]** – `cufftHandle` returned by `cufftCreate`.
- **rank[In]** – Dimensionality of the transform (1, 2, or 3).
- **n[In]** – Array of size `rank`, describing the size of each dimension.
- **inembed[In]** – 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[In]** – Indicates the distance between two successive input elements in the least significant (i.e., innermost) dimension.
- **idist[In]** – Indicates the distance between the first element of two consecutive signals in a batch of the input data.
- **inputtype[In]** (`cudaDataType`) – Type of input data.
- **onembed[In]** – 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[In]** – Indicates the distance between two successive output elements in the output array in the least significant (i.e., innermost) dimension.
- **odist[In]** – Indicates the distance between the first element of two consecutive signals in a batch of the output data.
- **outputtype[In]** (`cudaDataType`) – Type of output data.
- **batch[In]** – Batch size for this transform.
- ***workSize[In]** – Pointer to the size(s), in bytes, of the work areas. For example for two GPUs worksize must be declared to have two elements.
- **executiontype[In]** (`cudaDataType`) – Type of data to be used for computations.
- ***workSize[Out]** – 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.

2.7. 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.

Parameters

- **plan[In]** – `cufftHandle` returned by `cufftCreate`.
- ***workSize[In]** – Pointer to the size(s), in bytes, of the work areas. For example for two GPUs worksize must be declared to have two elements.
- ***workSize[Out]** – 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_INTERNAL_ERROR** – An internal driver error was detected.

2.8. cuFFT Caller Allocated Work Area Support

2.8.1. 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.

Parameters

- **plan[In]** – `cufftHandle` returned by `cufftCreate`.
- **autoAllocate[In]** – Indicates whether to allocate 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_INTERNAL_ERROR** – An internal driver error was detected.

2.8.2. 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.

Parameters

- **plan[In]** – `cufftHandle` returned by `cufftCreate`.
- ***workArea[In]** – Pointer to `workArea`. For multiple GPUs, multiple work area pointers must be given.

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.
- **CUFFT_SETUP_FAILED** – The cuFFT library failed to initialize.

2.8.3. cufftXtSetWorkAreaPolicy()

cufftResult **cufftXtSetWorkAreaPolicy**(*cufftHandle* plan, cufftXtWorkAreaPolicy policy, size_t *workSize);

`cufftXtSetWorkAreaPolicy()` indicates that the caller intends to change work area size for a given plan handle. cuFFT's default behavior is to allocate the work area at plan generation time with a default size that depends on the plan type and other parameters. If `cufftXtSetWorkAreaPolicy()` has been called with the `policy` parameter set to `CUFFT_WORKAREA_MINIMAL`, cuFFT will attempt to re-plan the handle to use zero bytes of work area memory. If the `cufftXtSetWorkAreaPolicy()` call is successful the auto-allocated work area memory is released.

Currently the policies `CUFFT_WORKAREA_PERFORMANCE`, `CUFFT_WORKAREA_USER` and the `workSize` parameter are not supported and reserved for use in future cuFFT releases.

This function can be called once per lifetime of a plan handle.

Parameters

- **`plan[In]`** – `cufftHandle` returned by `cufftCreate`.
- **`policy[In]`** – Type of work area policy to apply.
- **`*workSize[In]`** – Reserved for future use.

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_INVALID_SIZE`** – FFT size does not allow use of the selected policy.
- **`CUFFT_INTERNAL_ERROR`** – An internal driver error was detected.
- **`CUFFT_SETUP_FAILED`** – The cuFFT library failed to initialize.

2.9. cuFFT Execution

2.9.1. 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.

Parameters

- **plan[In]** – cufftHandle returned by cufftCreate.
- **idata[In]** – Pointer to the complex input data (in GPU memory) to transform.
- **odata[In]** – Pointer to the complex output data (in GPU memory).
- **direction[In]** – The transform direction: CUFFT_FORWARD or CUFFT_INVERSE.
- **odata[Out]** – ontains the complex Fourier coefficients.

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_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.

2.9.2. 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.

Parameters

- **plan[In]** – cufftHandle returned by cufftCreate.
- **idata[In]** – Pointer to the real input data (in GPU memory) to transform.
- **odata[In]** – Pointer to the complex output data (in GPU memory).
- **odata[Out]** – Contains the complex Fourier coefficients.

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_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.

2.9.3. cufftExecC2R() and cufftExecZ2D()

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

cufftResult **cufftExecZ2D**(*cufftHandle* plan, cufftDoubleComplex *idata, cufftDoubleReal *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.

Parameters

- **plan[In]** – `cufftHandle` returned by `cufftCreate`.
- **idata[In]** – Pointer to the complex input data (in GPU memory) to transform.
- **odata[In]** – Pointer to the real output data (in GPU memory).
- **odata[Out]** – 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.

2.9.4. cufftXtExec()

cufftResult **cufftXtExec**(*cufftHandle* plan, void *input, void *output, int direction);

Function `cufftXtExec` executes any cuFFT transform regardless of precision and type. In case of complex-to-real and real-to-complex transforms `direction` parameter is ignored. cuFFT uses the GPU memory pointed to by the `input` parameter as input data. This function stores the Fourier coefficients in the `output` array. If `input` and `output` are the same, this method does an in-place transform.

Parameters

- **`plan[In]`** – `cufftHandle` returned by `cufftCreate`.
- **`input[In]`** – Pointer to the input data (in GPU memory) to transform.
- **`output[In]`** – Pointer to the output data (in GPU memory).
- **`direction[In]`** – The transform direction: `CUFFT_FORWARD` or `CUFFT_INVERSE`. Ignored for complex-to-real and real-to-complex transforms.
- **`output[Out]`** – 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.

2.9.5. cufftXtExecDescriptor()

cufftResult **cufftXtExecDescriptor**(*cufftHandle* plan, cudaLibXtDesc *input, cudaLibXtDesc *output, int direction);

Function `cufftXtExecDescriptor()` executes any cuFFT transform regardless of precision and type. In case of complex-to-real and real-to-complex transforms `direction` parameter is ignored. cuFFT uses the GPU memory pointed to by `cudaLibXtDesc *input` descriptor as input data and `cudaLibXtDesc *output` as output data.

Parameters

- **`plan[In]`** – `cufftHandle` returned by `cufftCreate`.
- **`input[In]`** – Pointer to the complex input data (in GPU memory) to transform.
- **`output[In]`** – Pointer to the complex output data (in GPU memory).

- **direction[In]** – The transform direction: CUFFT_FORWARD or CUFFT_INVERSE. Ignored for complex-to-real and real-to-complex transforms.
- **idata[Out]** – 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.

2.10. cuFFT and Multiple GPUs

2.10.1. cufftXtSetGPUs()

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

`cufftXtSetGPUs()` identifies 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. In cuFFT prior to 10.4.0, 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*()`. Parameter `whichGPUs` of `cufftXtSetGPUs()` function determines ordering of the GPUs with respect to data decomposition (first data chunk is placed on GPU denoted by first element of `whichGPUs`).

Parameters

- **plan[In]** – `cufftHandle` returned by `cufftCreate`.
- **nGPUs[In]** – Number of GPUs to use.
- **whichGPUs[In]** – The GPUs to use.

Return values

- **CUFFT_SUCCESS** – cuFFT successfully set 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 in cuFFT prior to 10.4.0.
- **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_VALUE** – The requested number of GPUs was less than 2 or more than 8.
- **CUFFT_INVALID_DEVICE** – An invalid GPU index was specified.
- **CUFFT_INVALID_SIZE** – Transform size that `plan` was created for does not meet minimum size criteria.

2.10.2. 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.

Parameters

- **plan[In]** – `cufftHandle` returned by `cufftCreate`.
- **workArea[In]** – Pointer to the pointers to workArea.

Return values

- **CUFFT_SUCCESS** – cuFFT successfully set the GPUs to use.
- **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.

2.10.3. cuFFT Multiple GPU Execution

2.10.3.1 cufftXtExecDescriptorC2C() and cufftXtExecDescriptorZ2Z()

cufftResult **cufftXtExecDescriptorC2C**(*cufftHandle* plan, cudaLibXtDesc *input, cudaLibXtDesc *output, int direction);

cufftResult **cufftXtExecDescriptorZ2Z**(*cufftHandle* plan, cudaLibXtDesc *input, cudaLibXtDesc *output, 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 *input` as input data. Since only in-place multiple GPU functionality is supported, this function also stores the result in the `cudaLibXtDesc *input` arrays.

Parameters

- **plan[In]** – `cufftHandle` returned by `cufftCreate`.

- ***input[In]** – Pointer to the complex input data (in GPU memory) to transform.
- ***output[In]** – Pointer to the complex output data (in GPU memory).
- **direction[In]** – The transform direction: CUFFT_FORWARD or CUFFT_INVERSE.
- **input[Out]** – 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 input 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.

2.10.3.2 cufftXtExecDescriptorR2C() and cufftXtExecDescriptorD2Z()

cufftResult **cufftXtExecDescriptorR2C**(*cufftHandle* plan, cudaLibXtDesc *input, cudaLibXtDesc *output);

cufftResult **cufftXtExecDescriptorD2Z**(*cufftHandle* plan, cudaLibXtDesc *input, cudaLibXtDesc *output);

cufftXtExecDescriptorR2C() (cufftXtExecDescriptorD2Z()) executes a single-precision (double-precision) real-to-complex transform plan. cuFFT uses the GPU memory pointed to by cudaLibXtDesc *input as input data. Since only in-place multiple GPU functionality is supported, this function also stores the result in the cudaLibXtDesc *input arrays.

Parameters

- **plan[In]** – cufftHandle returned by cufftCreate.
- ***input[In]** – Pointer to the complex input data (in GPU memory) to transform.
- ***output[In]** – Pointer to the complex output data (in GPU memory).
- **input[Out]** – 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 input 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.

2.10.3.3 cufftXtExecDescriptorC2R() and cufftXtExecDescriptorZ2D()

cufftResult **cufftXtExecDescriptorC2R**(*cufftHandle* plan, cudaLibXtDesc *input, cudaLibXtDesc *output);

cufftResult **cufftXtExecDescriptorZ2D**(*cufftHandle* plan, cudaLibXtDesc *input, cudaLibXtDesc *output);

`cufftXtExecDescriptorC2R()` (`cufftXtExecDescriptorZ2D()`) executes a single-precision (double-precision) complex-to-real transform plan in the transform direction as specified by `direction` parameter. cuFFT uses the GPU memory pointed to by `cudaLibXtDesc *input` as input data. Since only in-place multiple GPU functionality is supported, this function also stores the result in the `cudaLibXtDesc *input` arrays.

Parameters

- **plan[In]** – `cufftHandle` returned by `cufftCreate`.
- ***input[In]** – Pointer to the complex input data (in GPU memory) to transform.
- ***output[In]** – Pointer to the complex output data (in GPU memory).
- **input[Out]** – 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 `input` 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.

2.10.4. 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*()`.

2.10.4.1 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.

Parameters

- **plan[In]** – `cufftHandle` returned by `cufftCreate`.
- ****descriptor[In]** – Pointer to a pointer to a `cudaLibXtDesc` object.
- **format[In]** – cufftXtSubFormat`` value.
- ****descriptor[Out]** – 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.

2.10.4.1.1 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;
```

2.10.4.2 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()`.

Parameters

- **`*descriptor[In]`** – 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.

2.10.4.3 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. Calling `cufftXtMemcpy` function for multi-GPU batched FFT plans with `CUFFT_COPY_DEVICE_TO_DEVICE` transfer type is not supported.

Note that starting from CUDA 11.2 (cuFFT 10.4.0), `cufftSetStream()` is supported on multi-GPU plans. When associating a stream with a plan, `cufftXtMemcpy()` remains synchronous across the multiple GPUs.

Parameters

- **`plan[In]`** – `cufftHandle` returned by `cufftCreate`.
- **`dstPointer[In]`** – Pointer to the destination address(es).
- **`srcPointer[In]`** – Pointer to the source address(es).
- **`type[In]`** – `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.

2.10.4.3.1 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. Note that device_to_device `cufftXtMemcpy()` for 2D and 3D data is not currently supported.

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

2.10.5. General Multiple GPU Descriptor Types

2.10.5.1 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;
```

2.10.5.2 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
```

(continues on next page)

(continued from previous page)

```
};
typedef struct cudaLibXtDesc_t cudaLibXtDesc;
```

2.11. cuFFT Callbacks

2.11.1. cufftXtSetJITCallback()

cufftResult **cufftXtSetJITCallback**(*cufftHandle* plan, const char *callbackSymbolName, const void *callbackFatbin, size_t callbackFatbinSize, cufftXtCallbackType type, void **caller_info)

`cufftXtSetJITCallback()` specifies a load or store LTO callback to be used with the plan.

This call is valid only after a call to `cufftCreate()`, but before calling `cufftMakePlan*()`, which does the plan generation.

If there was already an LTO 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 the callback function needs. cuFFT will not retain the amount of shared memory associated with the previous callback if the callback function is changed.

Parameters

- **`plan[In]`** – `cufftHandle` returned by `cufftCreate`.
- **`callbackSymbolName[In]`** – null-terminated C string containing the (unmangled) callback symbol name (i.e. the name of the LTO callback routine). This symbol name will be runtime-compiled, and modifiers such as `extern "C"` or `namespace` are not supported.
- **`callbackFatbin[In]`** – Pointer to the location in host memory where the callback device function is located, after being compiled into LTO-IR with nvcc or NVRTC.
- **`callbackFatbinSize[In]`** – Size in bytes of the data pointed at by `callbackFatbin`.
- **`type[In]`** – Type of callback routine.
- **`callerInfo[In]`** – 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 valid (e.g. the handle was already used to make a plan).
- **`CUFFT_INVALID_TYPE`** – The callback type is not valid.

- **CUFFT_INVALID_VALUE** – The pointer to the callback device function is invalid or the size is 0.
- **CUFFT_NOT_SUPPORTED** – The functionality is not supported yet (e.g. multi-GPU with LTO callbacks).
- **CUFFT_INTERNAL_ERROR** – cuFFT encountered an unexpected error, likely in the runtime linking process; error codes will be expanded in a future release.
- **CUFFT_SETUP_FAILED** – The cuFFT library failed to initialize.

2.11.2. cufftXtSetCallback()

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

`cufftXtSetCallback()` specifies a load or store legacy 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 legacy 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.

Parameters

- **plan[In]** – `cufftHandle` returned by `cufftCreate`.
- **callbackRoutine[In]** – Array of callback routine pointers, one per GPU.
- **type[In]** – Type of callback routine.
- **callerInfo[In]** – 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 in cuFFT prior to 10.4.0.
- **CUFFT_INTERNAL_ERROR** – An internal driver error was detected.
- **CUFFT_SETUP_FAILED** – The cuFFT library failed to initialize.

2.11.3. cufftXtClearCallback()

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

`cufftXtClearCallback()` instructs cuFFT to stop invoking the specified legacy 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`.

Note that this method **does not work** with LTO callbacks.

Parameters

- **plan[In]** – cufftHandle returned by cufftCreate.
- **type[In]** – 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 in cuFFT prior to 10.4.0.
- **CUFFT_INTERNAL_ERROR** – An internal driver error was detected.

2.11.4. 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.

Parameters

- **plan[In]** – cufftHandle returned by cufftCreate.
- **type[In]** – Type of callback routine.
- **sharedSize[In]** – 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 in cuFFT prior to 10.4.0.
- **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.

2.12. 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()`.

Note that starting from CUDA 11.2 (cuFFT 10.4.0), `cufftSetStream()` is supported on multi-GPU plans. When associating a stream with a plan, `cufftXtMemcpy()` remains synchronous across the multiple GPUs. For previous versions of cuFFT, `cufftSetStream()` will return an error in multiple GPU plans.

Note that starting from CUDA 12.2 (cuFFT 11.0.8), on multi-GPU plans, `stream` can be associated with any context on any GPU. However, repeated calls to `cufftSetStream()` with streams from different contexts incur a small time penalty. Optimal performance is obtained when repeated calls to `cufftSetStream` use streams from the same CUDA context.

Parameters

- **`plan[In]`** – The `cufftHandle` object to associate with the stream.
- **`stream[In]`** – A valid CUDA stream created with `cudaStreamCreate()`; `0` for the default stream.

Return values

- **`CUFFT_SUCCESS`** – The stream was associated with the plan.
- **`CUFFT_INVALID_PLAN`** – The `plan` parameter is not a valid handle, or plan is multi-gpu in cuFFT version prior to 10.4.0.

2.13. cufftGetVersion()

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

Returns the version number of cuFFT.

Parameters

- **`*version[In]`** – Pointer to the version number.
- **`*version[Out]`** – Contains the version number.

Return values

`CUFFT_SUCCESS` – cuFFT successfully returned the version number.

2.14. cufftGetProperty()

cufftResult **cufftGetProperty**(libraryPropertyType type, int *value);

Return in `*value` the number for the property described by `type` of the dynamically linked CUFFT library.

Parameters

- **type[In]** – CUDA library property.
- **value[Out]** – Contains the integer value for the requested property.

Return values

- **CUFFT_SUCCESS** – The property value was successfully returned.
- **CUFFT_INVALID_TYPE** – The property type is not recognized.
- **CUFFT_INVALID_VALUE** – `value` is NULL.

2.15. cuFFT Types

2.15.1. 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;
```

2.15.2. Parameters for Transform Direction

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

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

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

2.15.3. 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;
```

2.15.3.1 Type definitions for LTO callbacks

The LTO callback function prototypes and pointer type definitions are as follows:

```
typedef cufftComplex (*cufftJITCallbackLoadC)(void *dataIn, unsigned long long offset,
↪ void *callerInfo, void *sharedPointer);

typedef cufftDoubleComplex (*cufftJITCallbackLoadZ)(void *dataIn, unsigned long long
↪offset, void *callerInfo, void *sharedPointer);

typedef cufftReal (*cufftJITCallbackLoadR)(void *dataIn, unsigned long long offset,
↪void *callerInfo, void *sharedPointer);

typedef cufftDoubleReal(*cufftJITCallbackLoadD)(void *dataIn, unsigned long long
↪offset, void *callerInfo, void *sharedPointer);


typedef void (*cufftJITCallbackStoreC)(void *dataOut, unsigned long long offset,
↪cufftComplex element, void *callerInfo, void *sharedPointer);

typedef void (*cufftJITCallbackStoreZ)(void *dataOut, unsigned long long offset,
↪cufftDoubleComplex element, void *callerInfo, void *sharedPointer);

typedef void (*cufftJITCallbackStoreR)(void *dataOut, unsigned long long offset,
↪cufftReal element, void *callerInfo, void *sharedPointer);

typedef void (*cufftJITCallbackStoreD)(void *dataOut, unsigned long long offset,
↪cufftDoubleReal element, void *callerInfo, void *sharedPointer);
```

Notice the difference in the type of the `offset` parameter (`unsigned long long`) vs. legacy callbacks (which use `size_t`).

2.15.3.2 Type definitions for legacy callbacks

The legacy callback 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);
```

2.15.4. Other cuFFT Types

2.15.4.1 cufftHandle

type **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;
```

2.15.4.2 cufftReal

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

```
typedef float cufftReal;
```

2.15.4.3 cufftDoubleReal

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

```
typedef double cufftDoubleReal;
```

2.15.4.4 cufftComplex

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

```
typedef cuComplex cufftComplex;
```

2.15.4.5 cufftDoubleComplex

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

```
typedef cuDoubleComplex cufftDoubleComplex;
```

2.16. Common types

2.16.1. cudaDataType

The `cudaDataType` data type is an enumeration of the types supported by CUDA libraries.

```
typedef enum cudaDataType_t
{
        CUDA_R_16F= 2, // 16 bit real
        CUDA_C_16F= 6, // 16 bit complex
        CUDA_R_32F= 0, // 32 bit real
        CUDA_C_32F= 4, // 32 bit complex
        CUDA_R_64F= 1, // 64 bit real
        CUDA_C_64F= 5, // 64 bit complex
        CUDA_R_8I= 3,  // 8 bit real as a signed integer
        CUDA_C_8I= 7,  // 8 bit complex as a pair of signed integers
        CUDA_R_8U= 8,  // 8 bit real as an unsigned integer
        CUDA_C_8U= 9   // 8 bit complex as a pair of unsigned integers
} cudaDataType;
```

2.16.2. libraryPropertyType

The `libraryPropertyType` data type is an enumeration of library property types. (ie. CUDA version X.Y.Z would yield `MAJOR_VERSION=X`, `MINOR_VERSION=Y`, `PATCH_LEVEL=Z`)

```
typedef enum libraryPropertyType_t
{
        MAJOR_VERSION,
        MINOR_VERSION,
        PATCH_LEVEL
} libraryPropertyType;
```

Chapter 3. 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.

3.1. Multiple GPU Data Organization for Batched Transforms

For batches of transforms, each individual transform is executed on a single GPU. If possible the batches are evenly distributed among the GPUs. For a batch of size `m` performed on `n` GPUs, where `m` is not divisible by `n`, the first `m % n` GPUs will perform $\lfloor \frac{m}{n} \rfloor + 1$ transforms. The remaining GPUs will perform $\lfloor \frac{m}{n} \rfloor$ transforms. For example, in a batch of 15 transforms performed on 4 GPUs, the first three GPUs would perform 4 transforms, and the last GPU would perform 3 transforms. This approach removes the need for data exchange between the GPUs, and results in nearly perfect scaling for cases where the batch size is divisible by the number of GPUs.

3.2. 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. For example with 2 GPUs, for 2D and 3D transforms with even sized dimensions, 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.

Since 2D and 3D transforms support sizes other than powers of 2, it is possible that the data can not be evenly distributed among the GPUs. In general for the case of `n` GPUs, a dimension of size `m` that is not a multiple of `n` would be distributed such that the first `m % n` GPUs would get one extra row for 2D transforms, one extra plane for 3D transforms.

Take for example, a 2D transform on 4 GPUs, using an array declared in C as `data[x][y]`, where `x` is 65 and `y` is 99. The surface is distributed prior to the transform such that GPU 0 receives a surface with dimensions `[17][99]`, and GPUs 1...3 receive surfaces with dimensions `[16][99]`. After the

transform, each GPU again has a portion of the surface, but divided in the y dimension. GPUs 0...2 have surfaces with dimensions `[65][25]`. GPU 3 has a surface with dimensions `[65][24]`

For a 3D transform on 4 GPUs consider an array declared in C as `data[x][y][z]`, where `x` is 103, `y` is 122, and `z` is 64. The volume is distributed prior to the transform such that each GPUs 0...2 receive volumes with dimensions `[26][122][64]`, and GPU 3 receives a volume with dimensions `[25][122][64]`. After the transform, each GPU again has a portion of the surface, but divided in the y dimension. GPUs 0 and 1 have a volumes with dimensions `[103][31][64]`, and GPUs 2 and 3 have volumes with dimensions `[103][30][64]`.

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

By default 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 on two GPUs, GPU 0 receives data ranging from 0...(x/2-1). GPU 1 receives data ranging from (x/2)...(x-1). Similarly, with 4 GPUs, the data are evenly distributed among all 4 GPUs.

Before computation can begin, data are redistributed among the GPUs. It is possible to perform this redistribution in the copy from host memory, in cases where the application does not need to pre-process the data prior to the transform. To do this, the application can create the data descriptor with `cufftXtMalloc` using the sub-format `CUFFT_XT_FORMAT_1D_INPUT_SHUFFLED`. This can significantly reduce the time it takes to execute the transform.

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
```

(continues on next page)

(continued from previous page)

```
...
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
```

(continues on next page)

(continued from previous page)

```
 *                          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;
```

(continues on next page)

(continued from previous page)

```
    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 4. 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 5. 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
Using Plans	`fftw_execute()`, `fftw_destroy_plan()`, `fftw_cleanup()`
Basic Interface	
Complex DFTs	`fftw_plan_dft_1d()`, `fftw_plan_dft_2d()`, `fftw_plan_dft_3d()`, `f`
Planner Flags	
Real-data DFTs	`fftw_plan_dft_r2c_1d()`, `fftw_plan_dft_r2c_2d()`, `fftw_plan_dft`
Read-data DFT Array Format	
Read-to-Real Transform	
Read-to-Real Transform Kinds	
Advanced Interface	
Advanced Complex DFTs	`fftw_plan_many_dft()` with multiple 1D, 2D, 3D transforms
Advanced Real-data DFTs	`fftw_plan_many_dft_r2c()`, `fftw_plan_many_dft_c2r()` with multiple
Advanced Real-to-Real Transforms	
Guru Interface	
Interleaved and split arrays	Interleaved format
Guru vector and transform sizes	`fftw_iodim` struct
Guru Complex DFTs	`fftw_plan_guru_dft()`, `fftw_plan_guru_dft_r2c()`, `fftw_plan_gur`
Guru Real-data DFTs	
Guru Real-to-real Transforms	
64-bit Guru Interface	`fftw_plan_guru64_dft()`, `fftw_plan_guru64_dft_r2c()`, `fftw_plan`
New-array Execute Functions	`fftw_execute_dft()`, `fftw_execute_dft_r2c()`, `fftw_execute_dft_`
Wisdom	

Chapter 6. Deprecated Functionality

Starting from CUDA 12.0:

- GPU architectures SM35 and SM37 are no longer supported. The minimum required architecture is SM50.

Starting from CUDA 11.8:

- CUDA Graphs capture is no longer supported for legacy callback routines that load data in out-of-place mode transforms. Starting from CUDA 12.6 Update 2, LTO callbacks can be used as a replacement for legacy callbacks without this limitation.

Starting from CUDA 11.4:

- Support for callback functionality using separately compiled device code (legacy callbacks) is deprecated on all GPU architectures. Callback functionality will continue to be supported for all GPU architectures.

Starting from CUDA 11.0:

- GPU architecture SM30 is no longer supported. The minimum required architecture is SM35.
- Support for GPU architectures SM35, SM37 (Kepler), and SM50, SM52 (Maxwell) is deprecated.

Function `cufftSetCompatibilityMode` was removed in version 9.1.

Chapter 7. Notices

7.1. Notice

This document is provided for information purposes only and shall not be regarded as a warranty of a certain functionality, condition, or quality of a product. NVIDIA Corporation (“NVIDIA”) makes no representations or warranties, expressed or implied, as to the accuracy or completeness of the information contained in this document and assumes no responsibility for any errors contained herein. NVIDIA shall have no liability for the consequences or use of such information or for any infringement of patents or other rights of third parties that may result from its use. This document is not a commitment to develop, release, or deliver any Material (defined below), code, or functionality.

NVIDIA reserves the right to make corrections, modifications, enhancements, improvements, and any other changes to this document, at any time without notice.

Customer should obtain the latest relevant information before placing orders and should verify that such information is current and complete.

NVIDIA products are sold subject to the NVIDIA standard terms and conditions of sale supplied at the time of order acknowledgement, unless otherwise agreed in an individual sales agreement signed by authorized representatives of NVIDIA and customer (“Terms of Sale”). NVIDIA hereby expressly objects to applying any customer general terms and conditions with regards to the purchase of the NVIDIA product referenced in this document. No contractual obligations are formed either directly or indirectly by this document.

NVIDIA products are not designed, authorized, or warranted to be suitable for use in medical, military, aircraft, space, or life support equipment, nor in applications where failure or malfunction of the NVIDIA product can reasonably be expected to result in personal injury, death, or property or environmental damage. NVIDIA accepts no liability for inclusion and/or use of NVIDIA products in such equipment or applications and therefore such inclusion and/or use is at customer’s own risk.

NVIDIA makes no representation or warranty that products based on this document will be suitable for any specified use. Testing of all parameters of each product is not necessarily performed by NVIDIA. It is customer’s sole responsibility to evaluate and determine the applicability of any information contained in this document, ensure the product is suitable and fit for the application planned by customer, and perform the necessary testing for the application in order to avoid a default of the application or the product. Weaknesses in customer’s product designs may affect the quality and reliability of the NVIDIA product and may result in additional or different conditions and/or requirements beyond those contained in this document. NVIDIA accepts no liability related to any default, damage, costs, or problem which may be based on or attributable to: (i) the use of the NVIDIA product in any manner that is contrary to this document or (ii) customer product designs.

No license, either expressed or implied, is granted under any NVIDIA patent right, copyright, or other NVIDIA intellectual property right under this document. Information published by NVIDIA regarding third-party products or services does not constitute a license from NVIDIA to use such products or

services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property rights of the third party, or a license from NVIDIA under the patents or other intellectual property rights of NVIDIA.

Reproduction of information in this document is permissible only if approved in advance by NVIDIA in writing, reproduced without alteration and in full compliance with all applicable export laws and regulations, and accompanied by all associated conditions, limitations, and notices.

THIS DOCUMENT AND ALL NVIDIA DESIGN SPECIFICATIONS, REFERENCE BOARDS, FILES, DRAWINGS, DIAGNOSTICS, LISTS, AND OTHER DOCUMENTS (TOGETHER AND SEPARATELY, "MATERIALS") ARE BEING PROVIDED "AS IS." NVIDIA MAKES NO WARRANTIES, EXPRESSED, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE MATERIALS, AND EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES OF NONINFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE. TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL NVIDIA BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF NVIDIA HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. Notwithstanding any damages that customer might incur for any reason whatsoever, NVIDIA's aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the Terms of Sale for the product.

7.2. OpenCL

OpenCL is a trademark of Apple Inc. used under license to the Khronos Group Inc.

7.3. Trademarks

NVIDIA and the NVIDIA logo are trademarks or registered trademarks of NVIDIA Corporation in the U.S. and other countries. Other company and product names may be trademarks of the respective companies with which they are associated.

Copyright

Index

C