rocFFT Documentation

Release 1.0.32

Advanced Micro Devices, Inc.

Jul 22, 2025

CONTENTS

The rocFFT library provides a fast and accurate implementation of the discrete Fast Fourier Transform (FFT) written in HIP for GPU devices. The rocFFT library calculates discrete Fourier transforms for one, two, and three-dimensional transforms, supporting various data types for real and complex values. To learn more, see *What is rocFFT?*

The rocFFT public repository is located at https://github.com/ROCm/rocFFT.

Install

- *Installation guide*

Conceptual

- *FFT computation*

How to

- *Work with rocFFT*
- *Use real data*
- *Load and store callbacks*
- *Use runtime compilation*
- *Distribute transforms*

Samples

- rocFFT GitHub client examples

API reference

- *API usage*
- *API reference*

To contribute to the documentation, see Contributing to ROCm.

You can find licensing information on the Licensing page.

CHAPTER

ONE

WHAT IS ROCFFT?

The rocFFT library implements the discrete Fast Fourier Transform (FFT) in HIP for GPU devices. It provides a fast and accurate platform for calculating discrete FFTs. The source code can be found at https://github.com/ROCm/rocFFT.

rocFFT supports the following features:

- Half (`FP16`), single, and double precision floating point formats
- 1D, 2D, and 3D transforms
- Computation of transforms in batches
- Real and complex FFTs
- Arbitrary lengths, with optimizations for combinations of powers of 2, 3, 5, 7, 11, 13, and 17

rocFFT also provides experimental support for:

- Distributing transforms across multiple GPU devices in a single process
- Distributing transforms across multiple MPI (Message Passing Interface) processes

For information about how rocFFT computes FFTs, see *FFT computation*.

CHAPTER

TWO

BUILDING AND INSTALLING ROCFFT

This topic explains how to install rocFFT from the prebuilt packages or build it from the source code.

2.1 Installing prebuilt packages

For information on downloading and installing ROCm, see the ROCm installation guide.

To install rocFFT, use the package manager for your Linux distribution.

For example, on the Ubuntu distribution, run the following command:

```
sudo apt update && sudo apt install rocfft
```

2.2 Building rocFFT from source

You can use the GitHub releases tab to download the source code. This might provide you with a more recent version than the prebuilt packages.

rocFFT uses the AMD clang++ compiler and CMake. You can specify several options to customize your build. Use the following commands to build a shared library for the supported AMD GPUs:

```
mkdir build && cd build
cmake -DCMAKE_CXX_COMPILER=amdclang++ -DCMAKE_C_COMPILER=amdclang ..
make -j
```

Note

To compile a static library, use the `-DBUILD_SHARED_LIBS=off` option.

In ROCm version 4.3 or higher, indirect function calls are enabled for rocFFT by default. Use `-DROCFFT_CALLBACKS_ENABLED=off` with CMake to prevent these calls on older ROCm versions.

Note

If rocFFT is built with this configuration, callbacks won't work correctly.

2.3 rocFFT clients

rocFFT includes the following clients and utilities:

- **rocfft-bench**: Runs general transforms and performance analysis
- **rocfft-test**: Runs a series of regression tests
- Various samples

The following table includes the CMake option to build each client and the client dependencies.

Client	CMake option	Dependencies
rocfft-bench	`-DBUILD_CLIENTS_BENCH=on`	hipRAND
rocfft-test	`-DBUILD_CLIENTS_TESTS=on`	hipRAND, FFTW, GoogleTest
samples	`-DBUILD_CLIENTS_SAMPLES=on`	none

The clients are not built by default. To build them, use `-DBUILD_CLIENTS=on`. The build process downloads and builds GoogleTest and FFTW if they are not already installed. rocFFT uses version 1.11 of GoogleTest.

You can build the clients separately from the main library. For example, to build all the clients with an existing rocFFT library, invoke CMake from within the `rocFFT-src/clients` folder using these commands:

```
mkdir build && cd build
cmake -DCMAKE_CXX_COMPILER=amdclang++ -DCMAKE_PREFIX_PATH=/path/to/rocFFT-libb ..
make -j
```

To install the client dependencies on Ubuntu, run the following command:

```
sudo apt install libgtest-dev libfftw3-dev libboost-dev
```

Note

`libboost-dev` is the Boost development package. On Red Hat-related distributions, these packages are named `gtest-devel`, `fftw-devel` and `boost-devel`.

CHAPTER

THREE

FFT COMPUTATION

rocFFT is an implementation of the Discrete Fourier Transform (DFT) that makes use of symmetries in the DFT definition to reduce the mathematical complexity from $O(N^2)$ to $O(N \log N)$.

3.1 How does the library compute DFTs?

Here are the formulas for 1D, 2D, and 3D complex DFTs:

For a 1D complex DFT:

$\tilde{x}_j = \sum_{k=0}^{n-1} x_k \exp\left(\pm i \frac{2\pi jk}{n}\right)$ for $j = 0, 1, \ldots, n-1$

Where x_k is the complex data to be transformed, $\tilde{x}_j$ is the transformed data, and the sign $\pm$ determines the direction of the transform: $-$ for forward and $+$ for backward.

For a 2D complex DFT:

$\tilde{x}_{jk} = \sum_{q=0}^{m-1} \sum_{r=0}^{n-1} x_{rq} \exp\left(\pm i \frac{2\pi jr}{n}\right) \exp\left(\pm i \frac{2\pi kq}{m}\right)$

For $j = 0, 1, \ldots, n-1$ and $k = 0, 1, \ldots, m-1$, where x_{rq} is the complex data to be transformed, $\tilde{x}_{jk}$ is the transformed data, and the sign $\pm$ determines the direction of the transform.

For a 3D complex DFT:

$\tilde{x}_{jkl} = \sum_{s=0}^{p-1} \sum_{q=0}^{m-1} \sum_{r=0}^{n-1} x_{rqs} \exp\left(\pm i \frac{2\pi jr}{n}\right) \exp\left(\pm i \frac{2\pi kq}{m}\right) \exp\left(\pm i \frac{2\pi ls}{p}\right)$

For $j = 0, 1, \ldots, n-1$ and $k = 0, 1, \ldots, m-1$ and $l = 0, 1, \ldots, p-1$, where x_{rqs} is the complex data to be transformed, $\tilde{x}_{jkl}$ is the transformed data, and the sign $\pm$ determines the direction of the transform.

CHAPTER

FOUR

WORKING WITH ROCFFT

This topic describes how to use rocFFT, including how to structure the workflow, set up and clean up the library, and use plans, buffers, and batches.

4.1 Workflow

To compute an FFT with rocFFT, first create a plan. A plan is a handle to an internal data structure that holds the details about the transform. After creating the plan, execute it with the specified data buffers using a separate API call. The execution step can be repeated with the same plan on different input/output buffers as needed. The plan is destroyed when it is no longer needed.

To perform a transform, follow these steps:

1. Initialize the library by calling *rocfft_setup()*.
2. Create a plan for each distinct type of FFT that is required:
   - If the plan specification is simple, call *rocfft_plan_create()* and specify the value of the fundamental parameters.
   - If the plan includes more details, first create a plan description using *rocfft_plan_description_create()*. Call additional APIs, such as *rocfft_plan_description_set_data_layout()*, to specify the plan details. Then call *rocfft_plan_create()*, passing the description handle to it along with the other details.
3. Optionally, allocate a work buffer for the plan:
   - Call *rocfft_plan_get_work_buffer_size()* to check the size of the work buffer required by the plan.
   - If a non-zero size is required:
     - Create an execution info object using *rocfft_execution_info_create()*.
     - Allocate a buffer using `hipMalloc` and pass the allocated buffer to *rocfft_execution_info_set_work_buffer()*.
4. Execute the plan:
   - Use the execution API *rocfft_execute()* to execute the actual computation on the data buffers specified.
   - Pass the extra execution information such as work buffers and compute streams to *rocfft_execute()* in the *rocfft_execution_info* object.
   - *rocfft_execute()* can be called repeatedly as needed for different data with the same plan.
   - If the plan requires a work buffer but one wasn't provided, *rocfft_execute()* automatically allocates a work buffer and frees it when execution is finished.
5. If a work buffer was allocated:

- Call `hipFree` to free the work buffer.
- Call *rocfft_execution_info_destroy()* to destroy the execution info object.

6. Destroy the plan by calling *rocfft_plan_destroy()*.
7. Terminate the library by calling *rocfft_cleanup()*.

4.1.1 Example

```
#include <iostream>
#include <vector>
#include "hip/hip_runtime_api.h"
#include "hip/hip_vector_types.h"
#include "rocfft/rocfft.h"

int main()
{
        // rocFFT gpu compute
        // ========================================

        rocfft_setup();

        size_t N = 16;
        size_t Nbytes = N * sizeof(float2);

        // Create HIP device buffer
        float2 *x;
        hipMalloc(&x, Nbytes);

        // Initialize data
        std::vector<float2> cx(N);
        for (size_t i = 0; i < N; i++)
        {
                cx[i].x = 1;
                cx[i].y = -1;
        }

        //  Copy data to device
        hipMemcpy(x, cx.data(), Nbytes, hipMemcpyHostToDevice);

        // Create rocFFT plan
        rocfft_plan plan = nullptr;
        size_t length = N;
        rocfft_plan_create(&plan, rocfft_placement_inplace,
             rocfft_transform_type_complex_forward, rocfft_precision_single,
             1, &length, 1, nullptr);

        // Check if the plan requires a work buffer
        size_t work_buf_size = 0;
        rocfft_plan_get_work_buffer_size(plan, &work_buf_size);
        void* work_buf = nullptr;
        rocfft_execution_info info = nullptr;
        if(work_buf_size)
        {
```

(continues on next page)

(continued from previous page)

```
                rocfft_execution_info_create(&info);
                hipMalloc(&work_buf, work_buf_size);
                rocfft_execution_info_set_work_buffer(info, work_buf, work_buf_size);
        }

        // Execute plan
        rocfft_execute(plan, (void**) &x, nullptr, info);

        // Wait for execution to finish
        hipDeviceSynchronize();

        // Clean up work buffer
        if(work_buf_size)
        {
                hipFree(work_buf);
                rocfft_execution_info_destroy(info);
        }

        // Destroy plan
        rocfft_plan_destroy(plan);

        // Copy result back to host
        std::vector<float2> y(N);
        hipMemcpy(y.data(), x, Nbytes, hipMemcpyDeviceToHost);

        // Print results
        for (size_t i = 0; i < N; i++)
        {
                std::cout << y[i].x << ", " << y[i].y << std::endl;
        }

        // Free device buffer
        hipFree(x);

        rocfft_cleanup();

        return 0;
}
```

4.2 Library setup and cleanup

At the beginning of the program, the function *rocfft_setup()* must be called before any of the library APIs. Similarly, the function *rocfft_cleanup()* must be called at the end of the program. These APIs properly allocate and free the resources.

4.3 Plans

A plan is a collection of most of the parameters needed to specify an FFT computation. A rocFFT plan includes the following information:

- The type of transform (complex or real)

- The dimension of the transform (1D, 2D, or 3D)
- The length or extent of data in each dimension
- The number of datasets that are transformed (batch size)
- The floating-point precision of the data
- Whether the transform is in-place or not in-place
- The format (array type) of the input/output buffer
- The layout of data in the input/output buffer
- The scaling factor to apply to the output of the transform

A rocFFT plan does not include the following parameters:

- The handles to the input and output data buffers.
- The handle to a temporary work buffer (if needed).
- Other information to control execution on the device.

These parameters are specified when the plan is executed.

4.4 Data

You must allocate, initialize, and specify the input/output buffers that hold the data for the transform. For larger transforms, temporary work buffers might be needed. Because the library tries to minimize its own allocation of memory regions on the device, it expects you to manage the work buffers. The size of the buffer that is needed can be queried using *rocfft_plan_get_work_buffer_size()*. After allocation, it can be passed to the library using *rocfft_execution_info_set_work_buffer()*. The GitHub repository provide some samples and examples.

4.5 Transform and array types

There are two main types of FFTs in the library:

- **Complex FFT**: Transformation of complex data (forward or backward). The library supports the following two array types to store complex numbers:
  - Planar format: The real and imaginary components are kept in two separate arrays:
    - Buffer 1: `RRRRR...`
    - Buffer 2: `IIIII...`
  - Interleaved format: The real and imaginary components are stored as contiguous pairs in the same array:
    - Buffer: `RIRIRIRIRI...`
- **Real FFT**: Transformation of real data. For transforms involving real data, there are two possibilities:
  - Real data being subject to a forward FFT that results in complex data (Hermitian).
  - Complex data (Hermitian) being subject to a backward FFT that results in real data.

Note

Real backward FFTs require Hermitian-symmetric input data, which would naturally happen in the output of a real forward FFT. rocFFT produces undefined results if this requirement is not met.

The library provides the *rocfft_transform_type* and *rocfft_array_type* enumerations to specify transform and array types, respectively.

4.6 Batches

The efficiency of the library is improved by batching the transforms. Sending as much data as possible in a single transform call leverages the parallel compute capabilities of the GPU devices and minimizes the control transfer penalty. It's best to think of the GPU as a high-throughput, high-latency device, similar to a high-bandwidth networking pipe with high ping response times. If the client is ready to send data to the device to compute, it should send it using as few API calls as possible, which can be accomplished by batching. rocFFT plans can use the `number_of_transforms` parameter (also referred to as the batch size) in *rocfft_plan_create()* to describe the number of transforms being requested. All 1D, 2D, and 3D transforms can be batched.

4.7 Result placement

The API supports both in-place and not in-place transforms through the *rocfft_result_placement* enumeration. With in-place transforms, only the input buffers are provided to the execution API. The resulting data is written to the same buffer, overwriting the input data. With not in-place transforms, distinct output buffers are provided, and the results are written into the output buffer.

> **Note**
>
> rocFFT can often overwrite the input buffers on real inverse (complex-to-real) transforms, even if they are requested as not in-place. By doing this, rocFFT is better able to optimize the FFT.

4.8 Strides and distances

Strides and distances enable users to specify a custom layout for the data using *rocfft_plan_description_set_data_layout()*.

For 1D data, if `strides[0] == strideX == 1`, successive elements in the first dimension (dimension index `0`) are stored contiguously in memory. If `strideX` is a value greater than 1, gaps in memory exist between each element of the vector. For multidimensional cases, if `strides[1] == strideY == LenX` for 2D data and `strides[2] == strideZ == LenX * LenY` for 3D data, no gaps exist in memory between each element, and all vectors are stored tightly packed in memory. Here, `LenX`, `LenY`, and `LenZ` denote the transform lengths `lengths[0]`, `lengths[1]`, and `lengths[2]`, respectively, which are used to set up the plan.

Distance is the stride between corresponding elements of successive FFT data instances (primitives) in a batch. Distance is measured in units of the memory type. Complex data is measured in complex units and real data in real units. For tightly packed data, the distance between FFT primitives is the size of the FFT primitive, such that `dist == LenX` for 1D data, `dist == LenX * LenY` for 2D data, and `dist == LenX * LenY * LenZ` for 3D data. It is possible to set the distance of a plan to be less than the size of the FFT vector, which is typically 1 when doing column (strided) access on packed data.

When computing a batch of 1D FFT vectors, if `distance == 1` and `strideX == length(vector)`, it means the data for each logical FFT is read along columns, in this case, along the batch. You must verify that the distance and strides are valid and confirm that each logical FFT instance does not overlap with any other in the output data. If this is not the case, undefined results can occur. Overlapping on input data is generally allowed.

A simple example of a column data access pattern would be a 1D transform of length 4096 on each row of an array of 1024 rows by 4096 columns of values stored in a column-major array, for example, from a Fortran program. (This would be equivalent to a C or C++ program that has an array of 4096 rows by 1024 columns stored in row-major format,

where you execute a 1D transform of length 4096 on each column.) In this case, specify the strides as `1024` and the distance as 1.

4.9 Overwriting non-contiguous buffers

rocFFT guarantees that both the reading of FFT input and the writing of FFT output respects the specified strides. However, temporary results can be written to these buffers contiguously, which might be unexpected if the strides are designed to avoid certain memory locations completely for reading and writing.

For example, a 1D FFT of length N with an input and output stride of 2 only transforms the even-indexed elements in the input and output buffers. But if temporary data needs to be written to the buffers, the odd-indexed elements might be overwritten.

However, rocFFT is guaranteed to respect the size of buffers. In the above example, the input/output buffers are $2N$ elements long, even if only N even-indexed elements are being transformed. No more than $2N$ elements of temporary data are written to the buffers during the transform.

These policies apply to both input and output buffers, because not in-place transforms might overwrite input data. See *Result placement* for more information.

4.10 Input and output fields

By default, the rocFFT inputs and outputs are on the same device and the layouts of each are described using a set of strides passed to *rocfft_plan_description_set_data_layout()*.

rocFFT optionally allows for inputs and outputs to be described as **fields**, each of which is decomposed into multiple **bricks**. Each brick can reside on a different device and have its own layout parameters.

> **Note**
>
> The rocFFT APIs for declaring fields and bricks are currently experimental and subject to change in future releases. To submit feedback, questions, and comments about these interfaces, use the rocFFT issue tracker.

The workflow for using fields is as follows:

1. Allocate a *rocfft_field* struct by calling *rocfft_field_create()*.
2. Add one or more bricks to the field:
    1. Allocate a *rocfft_brick* by calling *rocfft_brick_create()*. Define the brick dimensions in terms of the lower and upper coordinates in the field's index space.

       Note that the lower coordinate is inclusive (contained within the brick) and the upper coordinate is exclusive (the first index past the end of the brick).

       Specify the device on which the brick resides, along with the strides of the brick in device memory.

       The coordinates and strides provided here include the batch dimensions unless the batch is 1.
    2. Add the brick to the field by calling *rocfft_field_add_brick()*.
    3. Deallocate the brick by calling *rocfft_brick_destroy()*.
3. Set the field as an input or output for the transform by calling either *rocfft_plan_description_add_infield()* or *rocfft_plan_description_add_outfield()* on a plan description that has already been allocated. The plan description must then be provided to *rocfft_plan_create()*.

The offsets, strides, and distances specified by *rocfft_plan_description_set_data_layout()* for input or output are ignored when a field is set for the corresponding input or output.

If the same field layout is used for both input and output, the same *rocfft_field* struct can be passed to both *rocfft_plan_description_add_infield()* and *rocfft_plan_description_add_outfield()*.

For in-place transforms, only call *rocfft_plan_description_add_infield()*. Do not call *rocfft_plan_description_add_outfield()*.

4. Deallocate the field by calling *rocfft_field_destroy()*.
5. Create the plan by calling *rocfft_plan_create()*. Pass the plan description that has already been allocated.
6. Execute the plan by calling *rocfft_execute()*. This function takes arrays of pointers for input and output. If fields have been set for input or output, then the arrays must contain pointers to each brick in the input or output.

   The pointers must be provided in the same order in which the bricks were added to the field (using calls to *rocfft_field_add_brick()*) and must point to memory on the device that was specified at that time.

Important

For in-place transforms, pass a non-empty array of input pointers and an empty array of output pointers.

4.11 Transforms of real data

See *Real data* for more information.

4.12 Reproducibility of results

The results of an FFT computation generated by rocFFT are bitwise reproducible. For example, deterministic behavior is expected between runs and every run generates the exact same results. Bitwise reproducibility is achieved provided the following attributes are kept constant between runs:

- FFT parameters
- rocFFT library version
- GPU model

A valid FFT plan is a requirement for reproducibility. In particular, the *rules for overlapping of FFT data* must be followed.

4.13 Result scaling

The output of a forward or backward FFT often needs to be multiplied by a scaling factor before the data can be passed to the next step of a computation. While you can launch a separate GPU kernel to do this work, rocFFT provides a *rocfft_plan_description_set_scale_factor()* function to more efficiently combine this scaling multiplication with the FFT work.

The scaling factor is set as part of the plan description before plan creation.

4.14 Loading and storing callbacks

See *Loading and storing callbacks* for more information.

4.15 Runtime compilation

See *Runtime compilation* for more information.

CHAPTER

FIVE

REAL DATA

When real data serves as input to a DFT, the resulting complex output data follows a special property, which is that about half of the output is redundant because it consists of complex conjugates of the other half. This is called Hermitian redundancy. So it's only necessary to store the non-redundant part of the data. Most FFT libraries use this property to offer specific storage layouts for FFTs involving real data. rocFFT provides three enumeration values for *rocfft_array_type* to deal with real data FFTs:

- `REAL`: rocfft_array_type_real
- `HERMITIAN_INTERLEAVED`: rocfft_array_type_hermitian_interleaved
- `HERMITIAN_PLANAR`: rocfft_array_type_hermitian_planar

The `REAL` (rocfft_array_type_real) enumeration specifies that the data is purely real. This can be used to feed real input or get back real output. The `HERMITIAN_INTERLEAVED` (rocfft_array_type_hermitian_interleaved) and `HERMITIAN_PLANAR` (rocfft_array_type_hermitian_planar) enumerations are similar to the corresponding full complex enumerations in the way they store real and imaginary components but store only about half of the complex output. Client applications can perform a forward transform and analyze the output or they can process the output and do a backward transform to get real data back. This is illustrated in the following figure.

Note

Real backward FFTs require that the input data be Hermitian-symmetric, which would naturally happen in the output of a real forward FFT. rocFFT will produce undefined results if this requirement is not met.

Consider the full output of a 1D real FFT of length N, as shown in following figure:

Here, `C*` denotes the complex conjugate. Because the values at indices greater than $N/2$ can be deduced from the first half of the array, rocFFT only stores the data up to the index $N/2$. This means that the output contains only $1 + N/2$ complex elements, where the division $N/2$ is rounded down. Examples for even and odd lengths are given below.

An example for $N = 8$ is shown in following figure.

An example for $N = 7$ is shown in following figure.

For a length of 8, only $(1+8/2) = 5$ of the output complex numbers are stored, with the index ranging from 0 through 4. Similarly, for a length of 7, only $(1 + 7/2) = 4$ of the output complex numbers are stored, with the index ranging from 0 through 3. For 2D and 3D FFTs, the FFT length along the innermost dimension is used to compute the $(1+N/2)$ value. This is because the FFT along the innermost dimension is computed first and is logically a real-to-Hermitian transform. The FFTs that are along other dimensions are computed next and are complex-to-complex transforms. For example, assuming $Lengths[2]$ is used to set up a 2D real FFT, let $N1 = Lengths[1]$ and $N0 = Lengths[0]$. The output FFT has $N1 * (1 + N0/2)$ complex elements. Similarly, for a 3D FFT with $Lengths[3]$, $N2 = Lengths[2]$, $N1 = Lengths[1]$, and $N0 = Lengths[0]$, the output has $N2 * N1 * (1 + N0/2)$ complex elements.

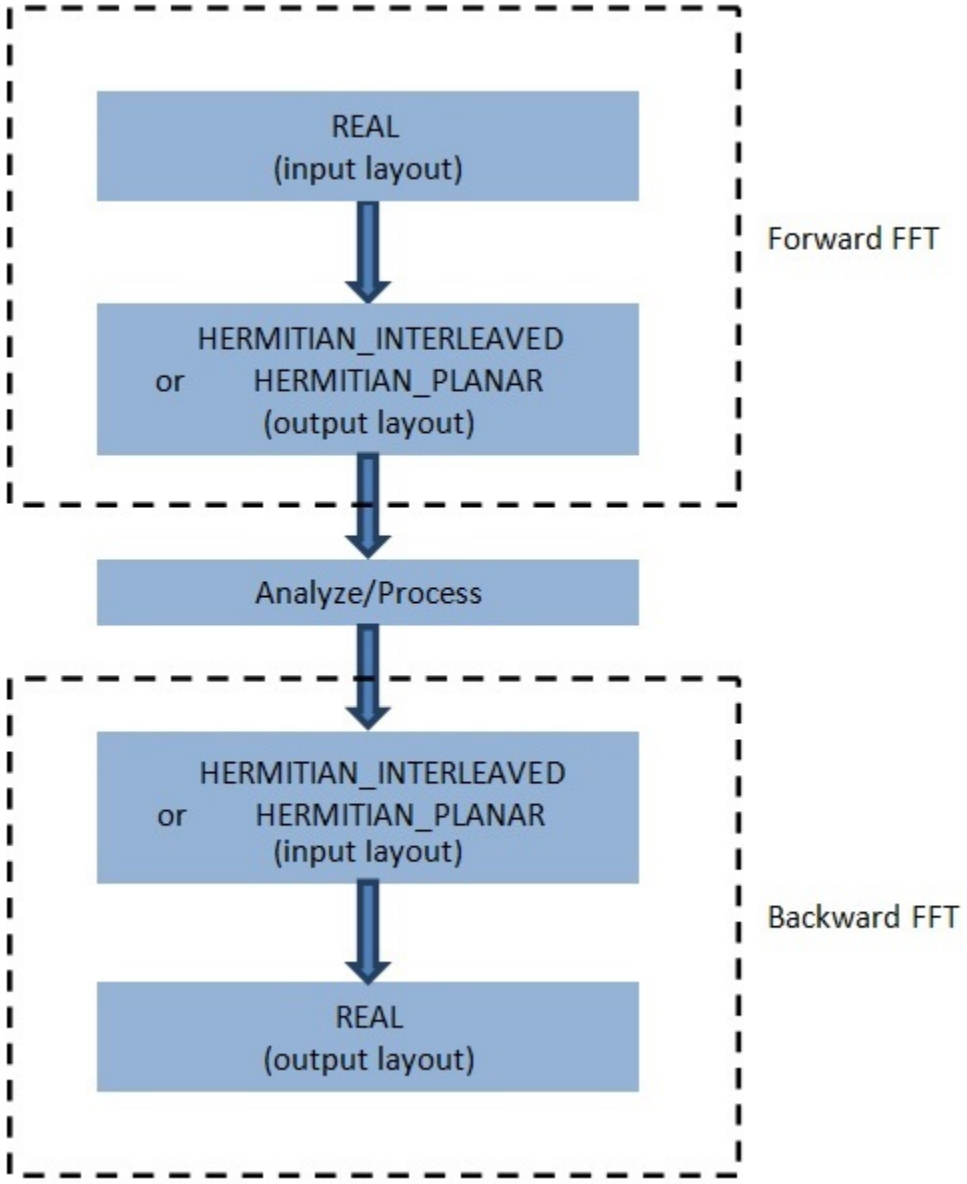

Fig. 5.1: **Forward and backward real FFTs**

C(0)	**C(1)**	**C(2)**	**…**	**C(N/2)**	**…**	**C(N-2) = C*(2)**	**C(N-1) = C*(1)**

Fig. 5.2: **1D real FFT of length N**

C(0)	**C(1)**	**C(2)**	**C(3)**	**C(4)**	C(5) = C*(3)	C(6) = C*(2)	C(7) = C*(1)

Fig. 5.3: **Example for N = 8**

C(0)	C(1)	C(2)	C(3)	C(4) = C*(3)	C(5) = C*(2)	C(6) = C*(1)

Fig. 5.4: **Example for N = 7**

5.1 Supported array type combinations

Not in-place transforms:

- Forward: `REAL` to `HERMITIAN_INTERLEAVED`
- Forward: `REAL` to `HERMITIAN_PLANAR`
- Backward: `HERMITIAN_INTERLEAVED` to `REAL`
- Backward: `HERMITIAN_PLANAR` to `REAL`

In-place transforms:

- Forward: `REAL` to `HERMITIAN_INTERLEAVED`
- Backward: `HERMITIAN_INTERLEAVED` to `REAL`

5.2 Setting strides

The library requires the user to explicitly set input and output strides for real transforms for non-simple cases. See the following examples to understand which values to use for input and output strides under different scenarios. These examples show typical use cases, but you can allocate the buffers and choose a data layout according to your needs.

The following figures and examples explain the real FFT features of this library in detail.

This schematic illustrates the forward 1D FFT (real to Hermitian).

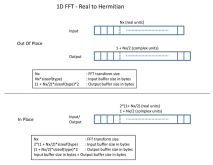

Fig. 5.5: **1D FFT - Real to Hermitian**

This schematic shows an example of a not in-place transform with an even N and how strides and distances are set.

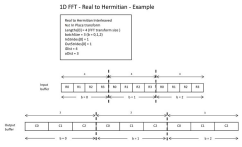

Fig. 5.6: **1D FFT - Real to Hermitian: example 1**

This schematic shows an example of an in-place transform with an even N and how strides and distances are set. Even though this example only deals with one buffer (in-place), the output strides/distance can take different values compared to the input strides/distance.

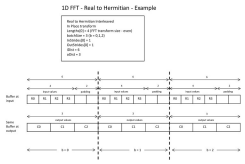

Fig. 5.7: **1D FFT - Real to Hermitian: example 2**

Here is an example of an in-place transform with an odd N and how strides and distances are set. Even though this example only deals with one buffer (in-place), the output strides/distance can take different values than the input strides/distance.

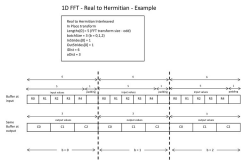

Fig. 5.8: **1D FFT - Real to Hermitian: example 3**

This schematic illustrates the backward 1D FFT (Hermitian to real).

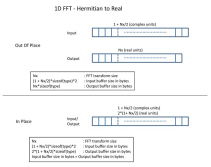

Fig. 5.9: **1D FFT - Hermitian to real**

Here is an example of an in-place transform with an even N and how strides and distances are set. Even though this example only deals with one buffer (in-place), the output strides/distance can take different values compared to the input strides/distance.

This schematic illustrates the in-place forward 2D FFT for real to Hermitian.

Here is an example of an in-place 2D transform and how strides and distances are set. Even though this example only deals with one buffer (in-place), the output strides/distance can take different values compared to the input strides/distance.

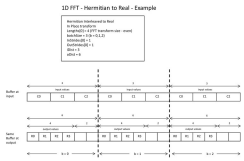

Fig. 5.10: **1D FFT - Hermitian to real example**

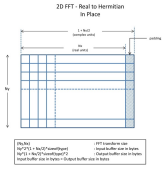

Fig. 5.11: **2D FFT - Real to Hermitian in-place**

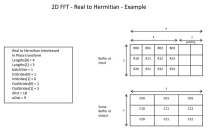

Fig. 5.12: **2D FFT - Real to Hermitian example**

CHAPTER

SIX

LOAD AND STORE CALLBACKS

rocFFT includes experimental functionality to call user-defined device functions when loading input from global memory at the transform start or when storing output to global memory at the transform end.

These optional user-defined callback functions can be supplied to the library using *rocfft_execution_info_set_load_callback()* and *rocfft_execution_info_set_store_callback()*.

Device functions supplied as callbacks must load and store element data types appropriate for the transform being executed.

Transform type	Load element type	Store element type
Complex-to-complex, half-precision	`_Float16_2`	`_Float16_2`
Complex-to-complex, single-precision	`float2`	`float2`
Complex-to-complex, double-precision	`double2`	`double2`
Real-to-complex, single-precision	`float`	`float2`
Real-to-complex, half-precision	`_Float16`	`_Float16_2`
Real-to-complex, double-precision	`double`	`double2`
Complex-to-real, half-precision	`_Float16_2`	`_Float16`
Complex-to-real, single-precision	`float2`	`float`
Complex-to-real, double-precision	`double2`	`double`

The callback function signatures must match the specifications below.

```
Tdata load_callback(Tdata* buffer, size_t offset, void* callback_data, void* shared_
↪memory);
void store_callback(Tdata* buffer, size_t offset, Tdata element, void* callback_data,␣
↪void* shared_memory);
```

The parameters for the functions are as follows:

- `Tdata`: The data type of each element being loaded or stored from the input or output.
- `buffer`: Pointer to the input (for load callbacks) or output (for store callbacks) in device memory that was passed to *rocfft_execute()*.
- `offset`: The offset of the location being read from or written to. This counts by elements from the `buffer` pointer.
- `element`: For store callbacks only, the element to be stored.
- `callback_data`: A pointer value accepted by *rocfft_execution_info_set_load_callback()* and *rocfft_execution_info_set_store_callback()* which is passed through to the callback function.
- `shared_memory`: A pointer to an amount of shared memory requested when the callback is set. Shared memory is not supported, so this parameter is always null.

Callback functions are called exactly once for each element being loaded or stored in a transform. Multiple kernels can be launched to decompose a transform, which means that separate kernels might call the load and store callbacks for a transform if both are specified.

Callbacks functions are only supported for transforms that do not use planar format for input or output.

CHAPTER

SEVEN

RUNTIME COMPILATION

rocFFT includes many kernels for common FFT problems. Many plans require additional kernels aside from the ones built into the library. In these cases, rocFFT compiles optimized kernels for the plan when the plan is created.

Compiled kernels are stored in memory by default. They will be reused if they are required again for plans in the same process.

If the `ROCFFT_RTC_CACHE_PATH` environment variable is set to a writable file location, rocFFT writes the compiled kernels to this location. rocFFT reads the kernels from this location for plans in other processes that need runtime-compiled kernels. rocFFT will create the specified file if it does not already exist.

CHAPTER

EIGHT

DISTRIBUTED TRANSFORMS

rocFFT can optionally distribute FFTs across multiple devices in a single process or across multiple Message Passing Interface (MPI) ranks. To perform distributed transforms, describe the input and output data layouts as *fields*.

8.1 Multiple devices in a single process

A transform can be distributed across multiple devices in a single process by passing distinct device IDs to *rocfft_brick_create()* to create bricks in the input and output fields.

Support for single-process multi-device transforms was introduced in ROCm 6.0 with rocFFT 1.0.25.

8.2 Message Passing Interface

MPI lets you distribute the transform across multiple processes, organized into MPI ranks.

To turn on rocFFT MPI support, enable the `ROCFFT_MPI_ENABLE` CMake option when building the library. By default, this option is off. To use Cray MPI, enable the `ROCFFT_CRAY_MPI_ENABLE` CMake option.

Additionally, rocFFT MPI support requires a GPU-aware MPI library that supports transferring data to and from HIP devices.

Support for MPI transforms was introduced in ROCm 6.3 with rocFFT 1.0.29.

> **Note**
>
> rocFFT API calls made on different ranks might return different values. Application developers must ensure that all ranks have successfully created their plans before attempting to execute a distributed transform. One rank can fail to create or execute a plan while the others succeed.

To distribute a transform across multiple MPI ranks, the following additional steps are required:

1. Each rank calls *rocfft_plan_description_set_comm()* to add an MPI communicator to an allocated plan description. rocFFT distributes the computation across all ranks in the communicator.
2. Each rank allocates the same fields and calls *rocfft_plan_description_add_infield()* and *rocfft_plan_description_add_outfield()* on the plan description. However, each rank must only call *rocfft_brick_create()* and *rocfft_field_add_brick()* for bricks that reside on that rank. A brick resides on exactly one rank. Each rank can have zero or more bricks associated to it.
3. Each rank in the communicator calls *rocfft_plan_create()*. rocFFT then uses this information to distribute the supplied brick information between all of the ranks.
4. Each rank in the communicator calls *rocfft_execute()*. This function accepts arrays of pointers for input and output. The arrays contain pointers to each brick in the input or output of the current rank.

The pointers must be provided in the same order in which the bricks were added to the field (using calls to *rocfft_field_add_brick()*) and must point to the memory on the device that was specified at that time.

For in-place transforms, only pass the input pointers and use an empty array of output pointers.

CHAPTER

NINE

ROCFFT API REFERENCE GUIDE

The rocFFT API reference information is divided into two sections. The *API usage* topic describes the datatypes and functions used to set up and implement the FFT in the hardware. The *API reference* topic provides a comprehensive listing of the classes and methods in the rocFFT library.

- *API usage*
- *API reference*

9.1 API usage

This section describes how to use the rocFFT library API.

9.1.1 Types

There are a few data structures that are internal to the library. The pointer types to these structures are listed below. Use these types to create handles and pass them between different library functions.

typedef struct rocfft_plan_t ***rocfft_plan**

Pointer type to plan structure.

This type is used to declare a plan handle that can be initialized with *rocfft_plan_create*.

typedef struct rocfft_plan_description_t ***rocfft_plan_description**

Pointer type to plan description structure.

This type is used to declare a plan description handle that can be initialized with *rocfft_plan_description_create*.

typedef struct rocfft_execution_info_t ***rocfft_execution_info**

Pointer type to execution info structure.

This type is used to declare an execution info handle that can be initialized with *rocfft_execution_info_create*.

9.1.2 Library setup and cleanup

The following functions handle initialization and cleanup of the library.

rocfft_status **rocfft_setup**()

Library setup function, called once in program before start of library use.

rocfft_status **rocfft_cleanup**()

Library cleanup function, called once in program after end of library use.

9.1.3 Plan

The following functions are used to create and destroy plan objects.

rocfft_status **rocfft_plan_create**(*rocfft_plan* *plan, *rocfft_result_placement* placement, *rocfft_transform_type* transform_type, *rocfft_precision* precision, size_t dimensions, const size_t *lengths, size_t number_of_transforms, const *rocfft_plan_description* description)

Create an FFT plan.

This API creates a plan, which the user can execute subsequently. This function takes many of the fundamental parameters needed to specify a transform.

The dimensions parameter can take a value of 1, 2, or 3. The 'lengths' array specifies the size of data in each dimension. Note that lengths[0] is the size of the innermost dimension, lengths[1] is the next higher dimension and so on (column-major ordering).

The 'number_of_transforms' parameter specifies how many transforms (of the same kind) needs to be computed. By specifying a value greater than 1, a batch of transforms can be computed with a single API call.

Additionally, a handle to a plan description can be passed for more detailed transforms. For simple transforms, this parameter can be set to NULL.

The plan must be destroyed with a call to *rocfft_plan_destroy*.

Parameters

- **plan** – **[out]** plan handle
- **placement** – **[in]** placement of result
- **transform_type** – **[in]** type of transform
- **precision** – **[in]** precision
- **dimensions** – **[in]** dimensions
- **lengths** – **[in]** dimensions-sized array of transform lengths
- **number_of_transforms** – **[in]** number of transforms
- **description** – **[in]** description handle created by rocfft_plan_description_create; can be NULL for simple transforms

rocfft_status **rocfft_plan_destroy**(*rocfft_plan* plan)

Destroy an FFT plan.

This API frees the plan after it is no longer needed.

Parameters

plan – **[in]** plan handle

The following functions are used to query for information after a plan is created.

rocfft_status **rocfft_plan_get_work_buffer_size**(const *rocfft_plan* plan, size_t *size_in_bytes)

Get work buffer size.

Get the work buffer size required for a plan.

Parameters

- **plan** – **[in]** plan handle
- **size_in_bytes** – **[out]** size of needed work buffer in bytes

rocfft_status **rocfft_plan_get_print**(const *rocfft_plan* plan)

Print all plan information.

Prints plan details to stdout, to aid debugging

Parameters
plan – **[in]** plan handle

9.1.4 Plan description

Most of the time, *rocfft_plan_create()* is able to fully specify a transform. However, advanced plan details such as strides and offsets require the creation of a plan description object, which is configured and passed to the *rocfft_plan_create()* function.

The plan description object can be safely destroyed after it is passed to the *rocfft_plan_create()* function.

rocfft_status **rocfft_plan_description_create**(*rocfft_plan_description* *description)

Create plan description.

This API creates a plan description with which the user can set extra plan properties. The plan description must be freed with a call to *rocfft_plan_description_destroy*.

Parameters
description – **[out]** plan description handle

rocfft_status **rocfft_plan_description_destroy**(*rocfft_plan_description* description)

Destroy a plan description.

This API frees the plan description. A plan description can be freed any time after it is passed to *rocfft_plan_create*.

Parameters
description – **[in]** plan description handle

rocfft_status **rocfft_plan_description_set_scale_factor**(*rocfft_plan_description* description, const double scale_factor)

Set scaling factor.

rocFFT multiplies each element of the result by the given factor at the end of the transform.

The supplied factor must be a finite number. That is, it must neither be infinity nor NaN.

Parameters

- **description** – **[in]** description handle
- **scale_factor** – **[in]** scaling factor

rocfft_status **rocfft_plan_description_set_data_layout**(*rocfft_plan_description* description, const *rocfft_array_type* in_array_type, const *rocfft_array_type* out_array_type, const size_t *in_offsets, const size_t *out_offsets, const size_t in_strides_size, const size_t *in_strides, const size_t in_distance, const size_t out_strides_size, const size_t *out_strides, const size_t out_distance)

Set advanced data layout parameters on a plan description.

This API specifies advanced layout of input/output buffers for a plan description.

The following parameters are supported for inputs and outputs:

- Array type (real, hermitian, or complex data, in either interleaved or planar format).
  - Real forward transforms require real input and hermitian output.
  - Real inverse transforms require hermitian input and real output.
  - Complex transforms require complex input and output.
  - Hermitian and complex data defaults to interleaved if a specific format is not specified.
- Offset of first data element in the data buffer. Defaults to 0 if unspecified.
- Stride between consecutive elements in each dimension. Defaults to contiguous data in all dimensions if unspecified.
- Distance between consecutive batches. Defaults to contiguous batches if unspecified.

Not all combinations of array types are supported and error codes will be returned for unsupported cases.

Offset, stride, and distance for either input or output provided here is ignored if a field is set for the corresponding input or output.

Parameters

- **description** – **[inout]** description handle
- **in_array_type** – **[in]** array type of input buffer
- **out_array_type** – **[in]** array type of output buffer
- **in_offsets** – **[in]** offsets, in element units, to start of data in input buffer
- **out_offsets** – **[in]** offsets, in element units, to start of data in output buffer
- **in_strides_size** – **[in]** size of in_strides array (must be equal to transform dimensions)
- **in_strides** – **[in]** array of strides, in each dimension, of input buffer; if set to null ptr library chooses defaults
- **in_distance** – **[in]** distance between start of each data instance in input buffer
- **out_strides_size** – **[in]** size of out_strides array (must be equal to transform dimensions)
- **out_strides** – **[in]** array of strides, in each dimension, of output buffer; if set to null ptr library chooses defaults
- **out_distance** – **[in]** distance between start of each data instance in output buffer

9.1.5 Execution

After creating a plan, execute it using the *rocfft_execute()* function. This function computes a transform on the specified data. It provides control over the execution and returns useful information.

rocfft_status **rocfft_execute**(const *rocfft_plan* plan, void *in_buffer[], void *out_buffer[], *rocfft_execution_info* info)

Execute an FFT plan.

This API executes an FFT plan on buffers given by the user.

If the transform is in-place, only the input buffer is needed and the output buffer parameter can be set to NULL. For not in-place transforms, output buffers have to be specified.

Input and output buffers are arrays of pointers. Interleaved array formats are the default, and require just one pointer per input or output buffer. Planar array formats require two pointers per input or output buffer - real and imaginary pointers, in that order.

If fields have been set for transform input or output, these arrays have one pointer per brick in the input or output field, provided in the order that the bricks were added to the field.

Note that input buffers may still be overwritten during execution of a transform, even if the transform is not in-place.

The final parameter in this function is a rocfft_execution_info handle. This optional parameter serves as a way for the user to control execution streams and work buffers.

Parameters

- **plan** – **[in]** plan handle
- **in_buffer** – **[inout]** array (of size 1 for interleaved data, of size 2 for planar data, or one per brick if an input field is set) of input buffers
- **out_buffer** – **[inout]** array (of size 1 for interleaved data, of size 2 for planar data, or one per brick if an output field is set) of output buffers, ignored for in-place transforms
- **info** – **[in]** execution info handle created by rocfft_execution_info_create

Execution info -==========

rocfft_execute() takes an optional *rocfft_execution_info* parameter. This parameter encapsulates information such as the work buffer and compute stream for the transform.

rocfft_status **rocfft_execution_info_create**(*rocfft_execution_info* *info)

Create execution info.

This API creates an execution info with which the user can control plan execution and work buffers. The execution info must be freed with a call to *rocfft_execution_info_destroy*.

Parameters

info – **[out]** execution info handle

rocfft_status **rocfft_execution_info_destroy**(*rocfft_execution_info* info)

Destroy an execution info.

This API frees the execution info. An execution info object can be freed any time after it is passed to *rocfft_execute*.

Parameters

info – **[in]** execution info handle

rocfft_status **rocfft_execution_info_set_work_buffer**(*rocfft_execution_info* info, void *work_buffer, const size_t size_in_bytes)

Set work buffer in execution info.

This is one of the execution info functions to specify optional additional information to control execution. This API provides a work buffer for the transform. It must be called before *rocfft_execute*.

When a non-zero value is obtained from *rocfft_plan_get_work_buffer_size*, that means the library needs a work buffer to compute the transform. In this case, the user should allocate the work buffer and pass it to the library via this API.

If a work buffer is required for the transform but is not specified using this function, *rocfft_execute* will automatically allocate the required buffer and free it when execution is finished.

Users should allocate their own work buffers if they need precise control over the lifetimes of those buffers, or if multiple plans need to share the same buffer.

Parameters

- **info** – **[in]** execution info handle

- **work_buffer** – **[in]** work buffer
- **size_in_bytes** – **[in]** size of work buffer in bytes

rocfft_status **rocfft_execution_info_set_stream**(*rocfft_execution_info* info, void *stream)

Set stream in execution info.

Associates an existing compute stream to a plan. This must be called before the call to *rocfft_execute*.

Once the association is made, execution of the FFT will run the computation through the specified stream.

The stream must be of type hipStream_t. It is an error to pass the address of a hipStream_t object.

Parameters

- **info** – **[in]** execution info handle
- **stream** – **[in]** underlying compute stream

9.1.6 HIP graph support for rocFFT

rocFFT supports capturing kernels launched by *rocfft_execute()* into HIP graph nodes. This approach captures the FFT execution and other work into a HIP graph and launches the work in the graph multiple times.

Graph capture is only supported for single-process transforms. Multi-process transforms, such as those that use Message Passing Interface, cannot use graph capture because rocFFT performs inter-process communication in addition to launching kernels.

Each launch of a HIP graph provides the same arguments to the kernels in the graph. In particular, this implies that all of the parameters to *rocfft_execute()* remain valid while the HIP graph is in use, including the following:

- The rocFFT plan
- The input and output buffers
- The *rocfft_execution_info* object, if provided

rocFFT does not support capturing work performed by other API functions, aside from *rocfft_execute()*, into HIP graphs.

9.1.7 Enumerations

This section lists all the enumerations that are used.

enum **rocfft_status**

rocFFT status/error codes

Values:

enumerator **rocfft_status_success**

enumerator **rocfft_status_failure**

enumerator **rocfft_status_invalid_arg_value**

enumerator **rocfft_status_invalid_dimensions**

enumerator **rocfft_status_invalid_array_type**

enumerator **rocfft_status_invalid_strides**

enumerator **rocfft_status_invalid_distance**

enumerator **rocfft_status_invalid_offset**

enumerator **rocfft_status_invalid_work_buffer**

enum **rocfft_transform_type**

Type of transform.

Values:

enumerator **rocfft_transform_type_complex_forward**

enumerator **rocfft_transform_type_complex_inverse**

enumerator **rocfft_transform_type_real_forward**

enumerator **rocfft_transform_type_real_inverse**

enum **rocfft_precision**

Precision.

Values:

enumerator **rocfft_precision_single**

enumerator **rocfft_precision_double**

enumerator **rocfft_precision_half**

enum **rocfft_result_placement**

Result placement.

Declares where the output of the transform should be placed. Note that input buffers may still be overwritten during execution of a transform, even if the transform is not in-place.

Values:

enumerator **rocfft_placement_inplace**

enumerator **rocfft_placement_notinplace**

enum **rocfft_array_type**

Array type.

Values:

enumerator **rocfft_array_type_complex_interleaved**

enumerator **rocfft_array_type_complex_planar**

enumerator **rocfft_array_type_real**

enumerator **rocfft_array_type_hermitian_interleaved**

enumerator **rocfft_array_type_hermitian_planar**

enumerator **rocfft_array_type_unset**

9.2 API reference

This topic includes a comprehensive listing of the classes and methods in the rocFFT library.

file **rocfft.h**

rocfft.h defines all the public interfaces and types

Defines

ROCFFT_EXPORT

Typedefs

typedef struct rocfft_plan_t ***rocfft_plan**

Pointer type to plan structure.

This type is used to declare a plan handle that can be initialized with *rocfft_plan_create*.

typedef struct rocfft_plan_description_t ***rocfft_plan_description**

Pointer type to plan description structure.

This type is used to declare a plan description handle that can be initialized with *rocfft_plan_description_create*.

typedef struct rocfft_execution_info_t ***rocfft_execution_info**

Pointer type to execution info structure.

This type is used to declare an execution info handle that can be initialized with *rocfft_execution_info_create*.

typedef struct rocfft_field_t *rocfft_field

Pointer type to a rocFFT field structure.

rocFFT fields are used to hold data decomposition information which is then passed to a *rocfft_plan* via a *rocfft_plan_description*

> **Warning**
>
> Experimental! This feature is part of an experimental API preview.

typedef struct rocfft_brick_t *rocfft_brick

Pointer type to a rocFFT brick structure.

rocFFT bricks are used to describe the data decomposition of fields.

> **Warning**
>
> Experimental! This feature is part of an experimental API preview.

Enums

enum rocfft_status

rocFFT status/error codes

Values:

enumerator rocfft_status_success

enumerator rocfft_status_failure

enumerator rocfft_status_invalid_arg_value

enumerator rocfft_status_invalid_dimensions

enumerator rocfft_status_invalid_array_type

enumerator rocfft_status_invalid_strides

enumerator rocfft_status_invalid_distance

enumerator rocfft_status_invalid_offset

enumerator rocfft_status_invalid_work_buffer

enum **rocfft_transform_type**

Type of transform.

Values:

enumerator **rocfft_transform_type_complex_forward**

enumerator **rocfft_transform_type_complex_inverse**

enumerator **rocfft_transform_type_real_forward**

enumerator **rocfft_transform_type_real_inverse**

enum **rocfft_precision**

Precision.

Values:

enumerator **rocfft_precision_single**

enumerator **rocfft_precision_double**

enumerator **rocfft_precision_half**

enum **rocfft_result_placement**

Result placement.

Declares where the output of the transform should be placed. Note that input buffers may still be overwritten during execution of a transform, even if the transform is not in-place.

Values:

enumerator **rocfft_placement_inplace**

enumerator **rocfft_placement_notinplace**

enum **rocfft_array_type**

Array type.

Values:

enumerator **rocfft_array_type_complex_interleaved**

enumerator **rocfft_array_type_complex_planar**

enumerator **rocfft_array_type_real**

enumerator **rocfft_array_type_hermitian_interleaved**

enumerator **rocfft_array_type_hermitian_planar**

enumerator **rocfft_array_type_unset**

enum **rocfft_comm_type**

Communicator type for distributed transforms.

Values:

enumerator **rocfft_comm_none**

enumerator **rocfft_comm_mpi**

Functions

rocfft_status **rocfft_setup**()

Library setup function, called once in program before start of library use.

rocfft_status **rocfft_cleanup**()

Library cleanup function, called once in program after end of library use.

rocfft_status **rocfft_plan_create**(*rocfft_plan* *plan, *rocfft_result_placement* placement, *rocfft_transform_type* transform_type, *rocfft_precision* precision, size_t dimensions, const size_t *lengths, size_t number_of_transforms, const *rocfft_plan_description* description)

Create an FFT plan.

This API creates a plan, which the user can execute subsequently. This function takes many of the fundamental parameters needed to specify a transform.

The dimensions parameter can take a value of 1, 2, or 3. The 'lengths' array specifies the size of data in each dimension. Note that lengths[0] is the size of the innermost dimension, lengths[1] is the next higher dimension and so on (column-major ordering).

The 'number_of_transforms' parameter specifies how many transforms (of the same kind) needs to be computed. By specifying a value greater than 1, a batch of transforms can be computed with a single API call.

Additionally, a handle to a plan description can be passed for more detailed transforms. For simple transforms, this parameter can be set to NULL.

The plan must be destroyed with a call to *rocfft_plan_destroy*.

Parameters

- **plan** – **[out]** plan handle
- **placement** – **[in]** placement of result
- **transform_type** – **[in]** type of transform
- **precision** – **[in]** precision
- **dimensions** – **[in]** dimensions

- **lengths** – **[in]** dimensions-sized array of transform lengths
- **number_of_transforms** – **[in]** number of transforms
- **description** – **[in]** description handle created by rocfft_plan_description_create; can be NULL for simple transforms

rocfft_status **rocfft_execute**(const *rocfft_plan* plan, void *in_buffer[], void *out_buffer[], *rocfft_execution_info* info)

Execute an FFT plan.

This API executes an FFT plan on buffers given by the user.

If the transform is in-place, only the input buffer is needed and the output buffer parameter can be set to NULL. For not in-place transforms, output buffers have to be specified.

Input and output buffers are arrays of pointers. Interleaved array formats are the default, and require just one pointer per input or output buffer. Planar array formats require two pointers per input or output buffer - real and imaginary pointers, in that order.

If fields have been set for transform input or output, these arrays have one pointer per brick in the input or output field, provided in the order that the bricks were added to the field.

Note that input buffers may still be overwritten during execution of a transform, even if the transform is not in-place.

The final parameter in this function is a rocfft_execution_info handle. This optional parameter serves as a way for the user to control execution streams and work buffers.

Parameters

- **plan** – **[in]** plan handle
- **in_buffer** – **[inout]** array (of size 1 for interleaved data, of size 2 for planar data, or one per brick if an input field is set) of input buffers
- **out_buffer** – **[inout]** array (of size 1 for interleaved data, of size 2 for planar data, or one per brick if an output field is set) of output buffers, ignored for in-place transforms
- **info** – **[in]** execution info handle created by rocfft_execution_info_create

rocfft_status **rocfft_plan_destroy**(*rocfft_plan* plan)

Destroy an FFT plan.

This API frees the plan after it is no longer needed.

Parameters

plan – **[in]** plan handle

rocfft_status **rocfft_plan_description_set_scale_factor**(*rocfft_plan_description* description, const double scale_factor)

Set scaling factor.

rocFFT multiplies each element of the result by the given factor at the end of the transform.

The supplied factor must be a finite number. That is, it must neither be infinity nor NaN.

Parameters

- **description** – **[in]** description handle
- **scale_factor** – **[in]** scaling factor

rocfft_status **rocfft_plan_description_set_data_layout**(*rocfft_plan_description* description, const *rocfft_array_type* in_array_type, const *rocfft_array_type* out_array_type, const size_t *in_offsets, const size_t *out_offsets, const size_t in_strides_size, const size_t *in_strides, const size_t in_distance, const size_t out_strides_size, const size_t *out_strides, const size_t out_distance)

Set advanced data layout parameters on a plan description.

This API specifies advanced layout of input/output buffers for a plan description.

The following parameters are supported for inputs and outputs:

- Array type (real, hermitian, or complex data, in either interleaved or planar format).
  - Real forward transforms require real input and hermitian output.
  - Real inverse transforms require hermitian input and real output.
  - Complex transforms require complex input and output.
  - Hermitian and complex data defaults to interleaved if a specific format is not specified.
- Offset of first data element in the data buffer. Defaults to 0 if unspecified.
- Stride between consecutive elements in each dimension. Defaults to contiguous data in all dimensions if unspecified.
- Distance between consecutive batches. Defaults to contiguous batches if unspecified.

Not all combinations of array types are supported and error codes will be returned for unsupported cases.

Offset, stride, and distance for either input or output provided here is ignored if a field is set for the corresponding input or output.

Parameters

- **description** – **[inout]** description handle
- **in_array_type** – **[in]** array type of input buffer
- **out_array_type** – **[in]** array type of output buffer
- **in_offsets** – **[in]** offsets, in element units, to start of data in input buffer
- **out_offsets** – **[in]** offsets, in element units, to start of data in output buffer
- **in_strides_size** – **[in]** size of in_strides array (must be equal to transform dimensions)
- **in_strides** – **[in]** array of strides, in each dimension, of input buffer; if set to null ptr library chooses defaults
- **in_distance** – **[in]** distance between start of each data instance in input buffer
- **out_strides_size** – **[in]** size of out_strides array (must be equal to transform dimensions)
- **out_strides** – **[in]** array of strides, in each dimension, of output buffer; if set to null ptr library chooses defaults
- **out_distance** – **[in]** distance between start of each data instance in output buffer

rocfft_status **rocfft_field_create**(*rocfft_field* *field)

Create a rocfft field struct.

> **Warning**
>
> Experimental! This feature is part of an experimental API preview.

rocfft_status **rocfft_field_destroy**(*rocfft_field* field)

Destroy a rocfft field struct.

The field struct can be destroyed after being added to the plan description; it is not used for plan execution.

> **Warning**
>
> Experimental! This feature is part of an experimental API preview.

rocfft_status **rocfft_get_version_string**(char *buf, size_t len)

Get library version string.

Parameters

- **buf** – **[inout]** buffer that receives the version string
- **len** – **[in]** length of buf, minimum 30 characters

rocfft_status **rocfft_plan_description_set_comm**(*rocfft_plan_description* description, *rocfft_comm_type* comm_type, void *comm_handle)

Set the communication library for distributed transforms.

Set the multi-processing communication library for a plan.

Multi-processing communication libraries require library-specific handle to also be specified. For MPI libraries, this is a pointer to an MPI communicator.

Parameters

- **description** – **[in]** description handle
- **comm_type** – **[in]** communicator type
- **comm_handle** – **[in]** handle to communication-library-specific state

rocfft_status **rocfft_brick_create**(*rocfft_brick* *brick, const size_t *field_lower, const size_t *field_upper, const size_t *brick_stride, size_t dim_with_batch, int deviceID)

Define a brick as part of a decomposition of a field.

Fields can contain a full-dimensional data distribution. The decomposition is specified by providing a lower coordinate and an upper coordinate in the field's index space. The lower coordinate is inclusive (contained within the brick) and the upper coordinate is exclusive (first index past the end of the brick).

One must also provide a stride for the brick data which specifies how the brick's data is arranged in memory.

All coordinates and strides must include batch dimensions, and are in column-major order (fastest-moving dimension first).

A HIP device ID is also provided - each brick may reside on a different device.

All arrays may be re-used or freed immediately after the function returns.

> **Warning**
>
> Experimental! This feature is part of an experimental API preview.

Parameters

- **brick** – **[out]** brick structure
- **field_lower** – **[in]** array of length dim specifying the lower index (inclusive) for the brick in the field's index space.
- **field_upper** – **[in]** array of length dim specifying the upper index (exclusive) for the brick in the field's index space.
- **brick_stride** – **[in]** array of length dim specifying the brick's stride in memory
- **dim_with_batch** – **[in]** length of the arrays; this must match the dimension of the FFT plus one for the batch dimension.
- **deviceID** – **[in]** HIP device ID for the device on which the brick's data is resident.

rocfft_status **rocfft_brick_destroy**(*rocfft_brick* brick)

Deallocate a brick created with rocfft_brick_create.

> **Warning**
>
> Experimental! This feature is part of an experimental API preview.

rocfft_status **rocfft_field_add_brick**(*rocfft_field* field, *rocfft_brick* brick)

Add a brick to a field.

Note that the order in which the bricks are added is significant; the pointers provided for each brick to *rocfft_execute* are in the same order that the bricks were added to the field.

The brick may be added to another field or destroyed any time after this function returns.

> **Warning**
>
> Experimental! This feature is part of an experimental API preview.

Parameters

- **field** – **[inout]** *rocfft_field* struct which holds the brick decomposition.
- **brick** – **[in]** *rocfft_brick* struct to add to the field.

rocfft_status **rocfft_plan_description_add_infield**(*rocfft_plan_description* description, *rocfft_field* field)

Add a *rocfft_field* to a *rocfft_plan_description* as an input.

The field may be reused or freed immediately after the function returns.

> **Warning**
>
> Experimental! This feature is part of an experimental API preview.

Parameters

- **description** – **[inout]** *rocfft_plan_description* that will pass the field information to plan creation
- **field** – **[in]** *rocfft_field* struct added as an input field

rocfft_status **rocfft_plan_description_add_outfield**(*rocfft_plan_description* description, *rocfft_field* field)

Add a *rocfft_field* to a *rocfft_plan_description* as an output.

The field may be reused or freed immediately after the function returns.

> **Warning**
>
> Experimental! This feature is part of an experimental API preview.

Parameters

- **description** – **[inout]** *rocfft_plan_description* that will pass the field information to plan creation
- **field** – **[in]** *rocfft_field* struct added as an output field

rocfft_status **rocfft_plan_get_work_buffer_size**(const *rocfft_plan* plan, size_t *size_in_bytes)

Get work buffer size.

Get the work buffer size required for a plan.

Parameters

- **plan** – **[in]** plan handle
- **size_in_bytes** – **[out]** size of needed work buffer in bytes

rocfft_status **rocfft_plan_get_print**(const *rocfft_plan* plan)

Print all plan information.

Prints plan details to stdout, to aid debugging

Parameters

plan – **[in]** plan handle

rocfft_status **rocfft_plan_description_create**(*rocfft_plan_description* *description)

Create plan description.

This API creates a plan description with which the user can set extra plan properties. The plan description must be freed with a call to *rocfft_plan_description_destroy*.

Parameters

description – **[out]** plan description handle

rocfft_status **rocfft_plan_description_destroy**(*rocfft_plan_description* description)

Destroy a plan description.

This API frees the plan description. A plan description can be freed any time after it is passed to *rocfft_plan_create*.

Parameters
description – **[in]** plan description handle

rocfft_status **rocfft_execution_info_create**(*rocfft_execution_info* *info)

Create execution info.

This API creates an execution info with which the user can control plan execution and work buffers. The execution info must be freed with a call to *rocfft_execution_info_destroy*.

Parameters
info – **[out]** execution info handle

rocfft_status **rocfft_execution_info_destroy**(*rocfft_execution_info* info)

Destroy an execution info.

This API frees the execution info. An execution info object can be freed any time after it is passed to *rocfft_execute*.

Parameters
info – **[in]** execution info handle

rocfft_status **rocfft_execution_info_set_work_buffer**(*rocfft_execution_info* info, void *work_buffer, const size_t size_in_bytes)

Set work buffer in execution info.

This is one of the execution info functions to specify optional additional information to control execution. This API provides a work buffer for the transform. It must be called before *rocfft_execute*.

When a non-zero value is obtained from *rocfft_plan_get_work_buffer_size*, that means the library needs a work buffer to compute the transform. In this case, the user should allocate the work buffer and pass it to the library via this API.

If a work buffer is required for the transform but is not specified using this function, *rocfft_execute* will automatically allocate the required buffer and free it when execution is finished.

Users should allocate their own work buffers if they need precise control over the lifetimes of those buffers, or if multiple plans need to share the same buffer.

Parameters

- **info** – **[in]** execution info handle
- **work_buffer** – **[in]** work buffer
- **size_in_bytes** – **[in]** size of work buffer in bytes

rocfft_status **rocfft_execution_info_set_stream**(*rocfft_execution_info* info, void *stream)

Set stream in execution info.

Associates an existing compute stream to a plan. This must be called before the call to *rocfft_execute*.

Once the association is made, execution of the FFT will run the computation through the specified stream.

The stream must be of type hipStream_t. It is an error to pass the address of a hipStream_t object.

Parameters

- **info** – **[in]** execution info handle

- **stream** – **[in]** underlying compute stream

rocfft_status **rocfft_execution_info_set_load_callback**(*rocfft_execution_info* info, void **cb_functions, void **cb_data, size_t shared_mem_bytes)

Set a load callback for a plan execution (experimental)

This function specifies a user-defined callback function that is run to load input from global memory at the start of the transform. Callbacks are an experimental feature in rocFFT.

Callback function pointers/data are given as arrays, with one function/data pointer per device executing this plan. Currently, plans can only use one device.

The provided function pointers replace any previously-specified load callback for this execution info handle.

Load callbacks have the following signature:

```
Tdata load_cb(Tdata* data, size_t offset, void* cbdata, void* sharedMem);
```

'Tdata' is the type of a single element of the input buffer. It is the caller's responsibility to ensure that the function type is appropriate for the plan (for example, a single-precision real-to-complex transform would load single-precision real elements).

A null value for 'cb' may be specified to clear any previously registered load callback.

Currently, 'shared_mem_bytes' must be 0. Callbacks are not supported on transforms that use planar formats for either input or output.

Parameters

- **info** – **[in]** execution info handle
- **cb_functions** – **[in]** callback function pointers
- **cb_data** – **[in]** callback function data, passed to the function pointer when it is called
- **shared_mem_bytes** – **[in]** amount of shared memory to allocate for the callback function to use

rocfft_status **rocfft_execution_info_set_store_callback**(*rocfft_execution_info* info, void **cb_functions, void **cb_data, size_t shared_mem_bytes)

Set a store callback for a plan execution (experimental)

This function specifies a user-defined callback function that is run to store output to global memory at the end of the transform. Callbacks are an experimental feature in rocFFT.

Callback function pointers/data are given as arrays, with one function/data pointer per device executing this plan. Currently, plans can only use one device.

The provided function pointers replace any previously-specified store callback for this execution info handle.

Store callbacks have the following signature:

```
void store_cb(Tdata* data, size_t offset, Tdata element, void* cbdata, void*␣
↪sharedMem);
```

'Tdata' is the type of a single element of the output buffer. It is the caller's responsibility to ensure that the function type is appropriate for the plan (for example, a single-precision real-to-complex transform would store single-precision complex elements).

A null value for 'cb' may be specified to clear any previously registered store callback.

Currently, 'shared_mem_bytes' must be 0. Callbacks are not supported on transforms that use planar formats for either input or output.

Parameters

- **info** – **[in]** execution info handle
- **cb_functions** – **[in]** callbacks function pointers
- **cb_data** – **[in]** callback function data, passed to the function pointer when it is called
- **shared_mem_bytes** – **[in]** amount of shared memory to allocate for the callback function to use

rocfft_status **rocfft_cache_serialize**(void **buffer, size_t *buffer_len_bytes)

Serialize compiled kernel cache.

Serialize rocFFT's cache of compiled kernels into a buffer. This buffer is allocated by rocFFT and must be freed with a call to *rocfft_cache_buffer_free*. The length of the buffer in bytes is written to 'buffer_len_bytes'.

rocfft_status **rocfft_cache_buffer_free**(void *buffer)

Free cache serialization buffer.

Deallocate a buffer allocated by *rocfft_cache_serialize*.

rocfft_status **rocfft_cache_deserialize**(const void *buffer, size_t buffer_len_bytes)

Deserialize a buffer into the compiled kernel cache.

Kernels in the buffer that match already-cached kernels will replace those kernels that are in the cache. Already-cached kernels that do not match those in the buffer are unmodified by this operation. The cache is unmodified if either a null buffer pointer or a zero length is passed.

file **README.md**

page **rocFFT**

rocFFT is a software library for computing fast Fourier transforms (FFTs) written in the HIP programming language. It's part of AMD's software ecosystem based on ROCm. The rocFFT library can be used with AMD and NVIDIA GPUs.

9.2.1 Documentation

[!NOTE] The published rocFFT documentation is available at rocFFT in an organized, easy-to-read format, with search and a table of contents. The documentation source files reside in the rocFFT/docs folder of this repository. As with all ROCm projects, the documentation is open source. For more information, see Contribute to ROCm documentation.

To build our documentation locally, use the following code:

```
cd docs

pip3 install -r sphinx/requirements.txt

python3 -m sphinx -T -E -b html -d _build/doctrees -D language=en . _build/html
```

9.2.2 Build and install

You can install rocFFT using pre-built packages or building from source.

- Installing pre-built packages:
  1. Download the pre-built packages from the ROCm package servers or use the GitHub releases tab to download the source (this may give you a more recent version than the pre-built packages).
  2. Run: `sudo apt update && sudo apt install rocfft`
- Building from source:

  rocFFT is compiled with AMD's clang++ and uses CMake. You can specify several options to customize your build. The following commands build a shared library for supported AMD GPUs:

```
mkdir build && cd build
cmake -DCMAKE_CXX_COMPILER=amdclang++ -DCMAKE_C_COMPILER=amdclang ..
make -j
```

  You can compile a static library using the `-DBUILD_SHARED_LIBS=off` option.

  With rocFFT, you can use indirect function calls by default; this requires ROCm 4.3 or higher. You can use `-DROCFFT_CALLBACKS_ENABLED=off` with CMake to prevent these calls on older ROCm compilers. Note that with this configuration, callbacks won't work correctly.

  rocFFT includes the following clients:

  - `rocfft-bench`: Runs general transforms and is useful for performance analysis
  - `rocfft-test`: Runs various regression tests
  - Various small samples

Client	CMake option	Dependencies
`rocfft-bench`	`-DBUILD_CLIENTS_BENCH=on`	hipRAND
`rocfft-test`	`-DBUILD_CLIENTS_TESTS=on`	hipRAND, FFTW, GoogleTest
samples	`-DBUILD_CLIENTS_SAMPLES=on`	None

Clients are not built by default. To build them, use `-DBUILD_CLIENTS=on`. The build process downloads and builds GoogleTest and FFTW if they are not already installed.

Clients can be built separately from the main library. For example, you can build all the clients with an existing rocFFT library by invoking CMake from within the `rocFFT-src/clients` folder:

```
mkdir build && cd build
cmake -DCMAKE_CXX_COMPILER=amdclang++ -DCMAKE_PREFIX_PATH=/path/to/rocFFT-lib ..
make -j
```

To install client dependencies on Ubuntu, run:

```
bash sudo apt install libgtest-dev libfftw3-dev libboost-dev
```

We use version 1.11 of GoogleTest.

9.2.3 Examples

A summary of the latest functionality and workflow to compute an FFT with rocFFT is available on the rocFFT documentation portal.

You can find additional examples in the `clients/samples` subdirectory.

9.2.4 Support

You can report bugs and feature requests through the GitHub issue tracker.

9.2.5 Contribute

If you want to contribute to rocFFT, you must follow our contribution guidelines.

CHAPTER

TEN

LICENSE

INDEX

R