# Contributing to cuDF

Contributions to cuDF fall into the following three categories.

1. To report a bug, request a new feature, or report a problem with

documentation, please file an [issue](https://github.com/rapidsai/cudf/issues/new/choose)

describing in detail the problem or new feature. The RAPIDS team evaluates

and triages issues, and schedules them for a release. If you believe the

issue needs priority attention, please comment on the issue to notify the

team.

2. To propose and implement a new Feature, please file a new feature request

[issue](https://github.com/rapidsai/cudf/issues/new/choose). Describe the

intended feature and discuss the design and implementation with the team and

community. Once the team agrees that the plan looks good, go ahead and

implement it, using the [code contributions](#code-contributions) guide below.

3. To implement a feature or bug-fix for an existing outstanding issue, please

Follow the [code contributions](#code-contributions) guide below. If you

need more context on a particular issue, please ask in a comment.

## Code contributions

### Your first issue

1. Follow the guide at the bottom of this page for [Setting Up Your Build Environment](#setting-up-your-build-environment).

2. Find an issue to work on. The best way is to look for the [good first issue](https://github.com/rapidsai/cudf/issues?q=is%3Aissue+is%3Aopen+label%3A%22good+first+issue%22)

or [help wanted](https://github.com/rapidsai/cudf/issues?q=is%3Aissue+is%3Aopen+label%3A%22help+wanted%22) labels.

3. Comment on the issue stating that you are going to work on it.

4. Code! Make sure to update unit tests!

5. When done, [create your pull request](https://github.com/rapidsai/cudf/compare).

6. Verify that CI passes all [status checks](https://help.github.com/articles/about-status-checks/). Fix if needed.

7. Wait for other developers to review your code and update code as needed.

8. Once reviewed and approved, a RAPIDS developer will merge your pull request.

Remember, if you are unsure about anything, don't hesitate to comment on issues

and ask for clarifications!

### Seasoned developers

Once you have gotten your feet wet and are more comfortable with the code, you

can look at the prioritized issues for our next release in our [project boards](https://github.com/rapidsai/cudf/projects).

> \*\*Pro Tip:\*\* Always look at the release board with the highest number for

issues to work on. This is where RAPIDS developers also focus their efforts.

Look at the unassigned issues, and find an issue to which you are comfortable

contributing. Start with \_Step 3\_ above, commenting on the issue to let

others know you are working on it. If you have any questions related to the

implementation of the issue, ask them in the issue instead of the PR.

## Setting Up Your Build Environment

The following instructions are for developers and contributors to cuDF OSS development. These instructions are tested on Linux Ubuntu 16.04 & 18.04. Use these instructions to build cuDF from source and contribute to its development. Other operating systems may be compatible, but are not currently tested.

### Code Formatting

#### Python

cuDF uses [Black](https://black.readthedocs.io/en/stable/),

[isort](https://readthedocs.org/projects/isort/), and

[flake8](http://flake8.pycqa.org/en/latest/) to ensure a consistent code format

throughout the project. `Black`, `isort`, and `flake8` can be installed with

`conda` or `pip`:

```bash

conda install black isort flake8

```

```bash

pip install black isort flake8

```

These tools are used to auto-format the Python code, as well as check the Cython

code in the repository. Additionally, there is a CI check in place to enforce

that committed code follows our standards. You can use the tools to

automatically format your python code by running:

```bash

isort --atomic python/\*\*/\*.py

black python

```

and then check the syntax of your Python and Cython code by running:

```bash

flake8 python

flake8 --config=python/cudf/.flake8.cython

```

Additionally, many editors have plugins that will apply `isort` and `Black` as

you edit files, as well as use `flake8` to report any style / syntax issues.

#### C++/CUDA

cuDF uses [`clang-format`](https://clang.llvm.org/docs/ClangFormat.html)

In order to format the C++/CUDA files, navigate to the root (`cudf`) directory and run:

```

python3 ./cpp/scripts/run-clang-format.py -inplace

```

Additionally, many editors have plugins or extensions that you can set up to automatically run `clang-format` either manually or on file save.

#### Pre-commit hooks

Optionally, you may wish to setup [pre-commit hooks](https://pre-commit.com/)

to automatically run `isort`, `Black`, `flake8` and `clang-format` when you make a git commit.

This can be done by installing `pre-commit` via `conda` or `pip`:

```bash

conda install -c conda-forge pre\_commit

```

```bash

pip install pre-commit

```

and then running:

```bash

pre-commit install

```

from the root of the cuDF repository. Now `isort`, `Black`, `flake8` and `clang-format` will be

run each time you commit changes.

### Get libcudf Dependencies

Compiler requirements:

\* `gcc` version 5.4+

\* `nvcc` version 10.0+

\* `cmake` version 3.14.0+

CUDA/GPU requirements:

\* CUDA 10.0+

\* NVIDIA driver 410.48+

\* Pascal architecture or better

You can obtain CUDA from [https://developer.nvidia.com/cuda-downloads](https://developer.nvidia.com/cuda-downloads).

## Script to build cuDF from source

### Build from Source

To install cuDF from source, ensure the dependencies are met and follow the steps below:

- Clone the repository and submodules

```bash

CUDF\_HOME=$(pwd)/cudf

git clone https://github.com/rapidsai/cudf.git $CUDF\_HOME

cd $CUDF\_HOME

git submodule update --init --remote --recursive

```

- Create the conda development environment `cudf\_dev`:

```bash

# create the conda environment (assuming in base `cudf` directory)

# note: RAPIDS currently doesn't support `channel\_priority: strict`; use `channel\_priority: flexible` instead

conda env create --name cudf\_dev --file conda/environments/cudf\_dev\_cuda10.0.yml

# activate the environment

conda activate cudf\_dev

```

- If using CUDA 10.0, create the environment with `conda env create --name cudf\_dev --file conda/environments/cudf\_dev\_cuda10.0.yml` instead.

- For other CUDA versions, check the corresponding cudf\_dev\_cuda\*.yml file in conda/environments

- Build and install `libcudf` after its dependencies. CMake depends on the `nvcc` executable being on your path or defined in `$CUDACXX`.

```bash

$ cd $CUDF\_HOME/cpp # navigate to C/C++ CUDA source root directory

$ mkdir build # make a build directory

$ cd build # enter the build directory

# CMake options:

# -DCMAKE\_INSTALL\_PREFIX set to the install path for your libraries or $CONDA\_PREFIX if you're using Anaconda, i.e. -DCMAKE\_INSTALL\_PREFIX=/install/path or -DCMAKE\_INSTALL\_PREFIX=$CONDA\_PREFIX

# -DCMAKE\_CXX11\_ABI set to ON or OFF depending on the ABI version you want, defaults to ON. When turned ON, ABI compability for C++11 is used. When OFF, pre-C++11 ABI compability is used.

$ cmake .. -DCMAKE\_INSTALL\_PREFIX=$CONDA\_PREFIX -DCMAKE\_CXX11\_ABI=ON # configure cmake ...

$ make -j # compile the libraries librmm.so, libcudf.so ... '-j' will start a parallel job using the number of physical cores available on your system

$ make install # install the libraries librmm.so, libcudf.so to the CMAKE\_INSTALL\_PREFIX

```

- As a convenience, a `build.sh` script is provided in `$CUDF\_HOME`. To execute the same build commands above, run the script as shown below. Note that the libraries will be installed to the location set in `$INSTALL\_PREFIX` if set (i.e. `export INSTALL\_PREFIX=/install/path`), otherwise to `$CONDA\_PREFIX`.

```bash

$ cd $CUDF\_HOME

$ ./build.sh # To build both C++ and Python cuDF versions with their dependencies

```

- To build only the C++ component with the script

```bash

$ ./build.sh libcudf # Build only the cuDF C++ components and install them to $INSTALL\_PREFIX if set, otherwise $CONDA\_PREFIX

```

- To run tests (Optional):

```bash

$ make test

```

- Build the `cudf` python package, in the `python/cudf` folder:

```bash

$ cd $CUDF\_HOME/python/cudf

$ python setup.py build\_ext --inplace

$ python setup.py install

```

- Like the `libcudf` build step above, `build.sh` can also be used to build the `cudf` python package, as shown below:

```bash

$ cd $CUDF\_HOME

$ ./build.sh cudf

```

- Additionally to build the `dask-cudf` python package, in the `python/dask\_cudf` folder:

```bash

$ cd $CUDF\_HOME/python/dask\_cudf

$ python setup.py install

```

- The `build.sh` script can also be used to build the `dask-cudf` python package, as shown below:

```bash

$ cd $CUDF\_HOME

$ ./build.sh dask\_cudf

```

- To run Python tests (Optional):

```bash

$ cd $CUDF\_HOME/python

$ py.test -v # run python tests on cudf and dask-cudf python bindings

```

- Other `build.sh` options:

```bash

$ cd $CUDF\_HOME

$ ./build.sh clean # remove any prior build artifacts and configuration (start over)

$ ./build.sh libcudf -v # compile and install libcudf with verbose output

$ ./build.sh libcudf -g # compile and install libcudf for debug

$ PARALLEL\_LEVEL=4 ./build.sh libcudf # compile and install libcudf limiting parallel build jobs to 4 (make -j4)

$ ./build.sh libcudf -n # compile libcudf but do not install

```

- The `build.sh` script can be customized to support other features:

- \*\*ABI version:\*\* The cmake `-DCMAKE\_CXX11\_ABI` option can be set to ON or OFF depending on the ABI version you want, defaults to ON. When turned ON, ABI compability for C++11 is used. When OFF, pre-C++11 ABI compability is used.

Done! You are ready to develop for the cuDF OSS project.

## Debugging cuDF

### Building Debug mode from source

Follow the [above instructions](#build-from-source) to build from source and add `-DCMAKE\_BUILD\_TYPE=Debug` to the `cmake` step.

For example:

```bash

$ cmake .. -DCMAKE\_INSTALL\_PREFIX=/install/path -DCMAKE\_BUILD\_TYPE=Debug # configure cmake ... use -DCMAKE\_INSTALL\_PREFIX=$CONDA\_PREFIX if you're using Anaconda

```

This builds `libcudf` in Debug mode which enables some `assert` safety checks and includes symbols in the library for debugging.

All other steps for installing `libcudf` into your environment are the same.

### Debugging with `cuda-gdb` and `cuda-memcheck`

When you have a debug build of `libcudf` installed, debugging with the `cuda-gdb` and `cuda-memcheck` is easy.

If you are debugging a Python script, simply run the following:

#### `cuda-gdb`

```bash

cuda-gdb -ex r --args python <program\_name>.py <program\_arguments>

```

#### `cuda-memcheck`

```bash

cuda-memcheck python <program\_name>.py <program\_arguments>

```

### Building and Testing on a gpuCI image locally

Before submitting a pull request, you can do a local build and test on your machine that mimics our gpuCI environment using the `ci/local/build.sh` script.

For detailed information on usage of this script, see [here](ci/local/README.md).

## Automated Build in Docker Container

A Dockerfile is provided with a preconfigured conda environment for building and installing cuDF from source based off of the master branch.

### Prerequisites

\* Install [nvidia-docker2](https://github.com/nvidia/nvidia-docker/wiki/Installation-(version-2.0)) for Docker + GPU support

\* Verify NVIDIA driver is `410.48` or higher

\* Ensure CUDA 10.0+ is installed

### Usage

From cudf project root run the following, to build with defaults:

```bash

$ docker build --tag cudf .

```

After the container is built run the container:

```bash

$ docker run --runtime=nvidia -it cudf bash

```

Activate the conda environment `cudf` to use the newly built cuDF and libcudf libraries:

```

root@3f689ba9c842:/# source activate cudf

(cudf) root@3f689ba9c842:/# python -c "import cudf"

(cudf) root@3f689ba9c842:/#

```

### Customizing the Build

Several build arguments are available to customize the build process of the

container. These are specified by using the Docker [build-arg](https://docs.docker.com/engine/reference/commandline/build/#set-build-time-variables---build-arg)

flag. Below is a list of the available arguments and their purpose:

| Build Argument | Default Value | Other Value(s) | Purpose |

| --- | --- | --- | --- |

| `CUDA\_VERSION` | 10.0 | 10.1, 10.2 | set CUDA version |

| `LINUX\_VERSION` | ubuntu16.04 | ubuntu18.04 | set Ubuntu version |

| `CC` & `CXX` | 5 | 7 | set gcc/g++ version; \*\*NOTE:\*\* gcc7 requires Ubuntu 18.04 |

| `CUDF\_REPO` | This repo | Forks of cuDF | set git URL to use for `git clone` |

| `CUDF\_BRANCH` | master | Any branch name | set git branch to checkout of `CUDF\_REPO` |

| `NUMBA\_VERSION` | newest | >=0.40.0 | set numba version |

| `NUMPY\_VERSION` | newest | >=1.14.3 | set numpy version |

| `PANDAS\_VERSION` | newest | >=0.23.4 | set pandas version |

| `PYARROW\_VERSION` | 0.17.1 | Not supported | set pyarrow version |

| `CMAKE\_VERSION` | newest | >=3.14 | set cmake version |

| `CYTHON\_VERSION` | 0.29 | Not supported | set Cython version |

| `PYTHON\_VERSION` | 3.6 | 3.7 | set python version |

---

## Attribution

Portions adopted from https://github.com/pytorch/pytorch/blob/master/CONTRIBUTING.md

Portions adopted from https://github.com/dask/dask/blob/master/docs/source/develop.rst