# Contributor Guidelines

## Make Tips

When building HIP, you will likely want to build and install to a local user-accessible directory (rather than /opt/rocm).

This can be easily be done by setting the -DCMAKE\_INSTALL\_PREFIX variable when running cmake. Typical use case is to

set CMAKE\_INSTALL\_PREFIX to your HIP git root, and then ensure HIP\_PATH points to this directory. For example

```

cmake .. -DCMAKE\_INSTALL\_PREFIX=..

make install

export HIP\_PATH=

```

After making HIP, don't forget the "make install" step !

## Adding a new HIP API

- Add a translation to the hipify-clang tool ; many examples abound.

- For stat tracking purposes, place the API into an appropriate stat category ("dev", "mem", "stream", etc).

- Add a inlined NVCC implementation for the function in include/hip/nvcc\_detail/hip\_runtime\_api.h.

- These are typically headers

- Add an HCC definition and Doxygen comments for the function in include/hcc\_detail/hip\_runtime\_api.h

- Source implementation typically go in src/hcc\_detail/hip\_hcc.cpp. The implementation may involve

calls to HCC runtime or HSA runtime, or interact with other pieces of the HIP runtime (ie for

hipStream\_t).

#### Testing HCC version

In some cases new HIP features are tied to specified releases of HCC, and it can be useful to determine at compile-time

if the current HCC compiler is sufficiently new enough to support the desired feature. The `\_\_hcc\_workweek\_\_` compiler

define is a monotonically increasing integer value that combines the year + workweek + day-of-week (0-6, Sunday is 0)

(ie 15403, 16014, etc).

The granularity is one day, so \_\_hcc\_workweek\_\_ can only be used to distinguish compiler builds that are at least one day apart.

```

#ifdef \_\_hcc\_workweek\_ > 16014

// use cool new HCC feature here

#endif

```

Additionally, hcc binary can print the work-week to stdout: ("16014" in the version info below.)4

```

> /opt/rocm/hcc/bin/hcc -v

HCC clang version 3.5.0 (based on HCC 0.8.16014-81f8a3f-f155163-5a1009a LLVM 3.5.0svn)

Target: x86\_64-unknown-linux-gnu

Thread model: posix

Found candidate GCC installation: /usr/lib/gcc/x86\_64-linux-gnu/4.8

Found candidate GCC installation: /usr/lib/gcc/x86\_64-linux-gnu/4.8.4

Found candidate GCC installation: /usr/lib/gcc/x86\_64-linux-gnu/4.9

Found candidate GCC installation: /usr/lib/gcc/x86\_64-linux-gnu/4.9.1

Selected GCC installation: /usr/lib/gcc/x86\_64-linux-gnu/4.8

Candidate multilib: .;@m64

Candidate multilib: 32;@m32

Candidate multilib: x32;@mx32

Selected multilib: .;@m64

```

The unix `date` command can print the HCC-format work-week for a specific date , ie:

```

> date --utc +%y%U%w -d 2015-11-09

15451

```

## Unit Testing Environment

HIP includes unit tests in the tests/src directory.

When adding a new HIP feature, add a new unit test as well.

See [tests/README.md](README.md) for more information.

## Development Flow

It is recommended that developers set the flag HIP\_BUILD\_LOCAL=1 so that the unit testing environment automatically rebuilds libhip\_hcc.a and the tests when a change it made to the HIP source.

Directed tests provide a great place to develop new features alongside the associated test.

For applications and benchmarks outside the directed test environment, developments should use a two-step development flow:

- #1. Compile, link, and install HCC. See [Installation](README.md#Installation) notes.

- #2. Relink the target application to include changes in the libhip\_hcc.a file.

## Environment Variables

- \*\*HIP\_PATH\*\* : Location of HIP include, src, bin, lib directories.

- \*\*HCC\_HOME\*\* : Path to HCC compiler. Default /opt/rocm/hcc.

- \*\*HSA\_PATH\*\* : Path to HSA include, lib. Default /opt/rocm/hsa.

- \*\*CUDA\_PATH\* : On nvcc system, this points to root of CUDA installation.

### Contribution guidelines ###

Features (ie functions, classes, types) defined in hip\*.h should resemble CUDA APIs.

The HIP interface is designed to be very familiar for CUDA programmers.

Differences or limitations of HIP APIs as compared to CUDA APIs should be clearly documented and described.

## Coding Guidelines (in brief)

- Code Indentation:

- Tabs should be expanded to spaces.

- Use 4 spaces indentation.

- Capitalization and Naming

- Prefer camelCase for HIP interfaces and internal symbols. Note HCC uses \_ for separator.

This guideline is not yet consistently followed in HIP code - eventual compliance is aspirational.

- Member variables should begin with a leading "\_". This allows them to be easily distinguished from other variables or functions.

- {} placement

- For functions, the opening { should be placed on a new line.

- For if/else blocks, the opening { is placed on same line as the if/else. Use a space to separate {/" from if/else. Example

'''

if (foo) {

doFoo()

} else {

doFooElse();

}

'''

- namespace should be on same line as { and separated by a space.

- Single-line if statement should still use {/} pair (even though C++ does not require).

- Miscellaneous

- All references in function parameter lists should be const.

- "ihip" = internal hip structures. These should not be exposed through the HIP API.

- Keyword TODO refers to a note that should be addressed in long-term. Could be style issue, software architecture, or known bugs.

- FIXME refers to a short-term bug that needs to be addressed.

- HIP\_INIT\_API() should be placed at the start of each top-level HIP API. This function will make sure the HIP runtime is initialized,

and also constructs an appropriate API string for tracing and CodeXL marker tracing. The arguments to HIP\_INIT\_API should match

those of the parent function.

- ihipLogStatus should only be called from top-level HIP APIs,and should be called to log and return the error code. The error code

is used by the GetLastError and PeekLastError functions - if a HIP API simply returns, then the error will not be logged correctly.

- All HIP environment variables should begin with the keyword HIP\_

Environment variables should be long enough to describe their purpose but short enough so they can be remembered - perhaps 10-20 characters, with 3-4 parts separated by underscores.

To see the list of current environment variables, along with their values, set HIP\_PRINT\_ENV and run any hip applications on ROCm platform .

HIPCC or other tools may support additional environment variables which should follow the above convention.

#### Presubmit Testing:

Before checking in or submitting a pull request, run all directed tests (see tests/README.md) and all Rodinia tests.

Ensure pass results match starting point:

```shell

> cd examples/

> ./run\_all.sh

```

#### Checkin messages

Follow existing best practice for writing a good Git commit message. Some tips:

http://chris.beams.io/posts/git-commit/

https://robots.thoughtbot.com/5-useful-tips-for-a-better-commit-message

In particular :

- Use imperative voice, ie "Fix this bug", "Refactor the XYZ routine", "Update the doc".

Not : "Fixing the bug", "Fixed the bug", "Bug fix", etc.

- Subject should summarize the commit. Do not end subject with a period. Use a blank line

after the subject.

## Doxygen Editing Guidelines

- bugs should be marked with @bugs near the code where the bug might be fixed. The @bug message will appear in the API description and also in the

doxygen bug list.

## Other Tips:

### Markdown Editing

Recommended to use an offline Markdown viewer to review documentation, such as Markdown Preview Plus extension in Chrome browser, or Remarkable.