hipTensor Documentation

Release 1.3.0

Advanced Micro Devices, Inc.

Sep 27, 2024

INSTALL

hipTensor is a work-in-progress (WIP) high-performance HIP library for tensor primitives. It is AMD's C++ library for accelerating tensor primitives which can leverage specialized GPU matrix cores on AMD's latest discrete GPUs. hipTensor is currently powered by the composable kernel library. The API is designed to be portable with the Nvidia cuTensor library, allowing those users to easily migrate to the AMD platform.

The hipTensor API offers functionality for defining tensor data objects and currently supports contraction and permutation operations on the tensor objects. Users may also control several available logging options. Under the hood, the hipTensor library is bundled with multitude of GPU kernels which are automatically selected and invoked to solve problems as defined by input parameters to the API. As hipTensor is currently a WIP, future tensor operation support may include reductions, element-wise operations and caching of selection instances.

Supporting host code is required for GPU device and memory management. The hipTensor code samples and tests provided are built and launched via the Heterogeneous-Compute Interface for Portability (HIP) ecosystem within ROCm.

You can access hipTensor code on our GitHub repository.

The documentation is structured as follows:

Install

- *Installation*

Conceptual

- *Programmer's guide*

API reference

- *API reference guide*

Contribution

- *Contributing to hipTensor*

To contribute to the documentation refer to Contributing to ROCm.

Licensing information can be found on the Licensing page.

CHAPTER

ONE

INSTALLATION

The quickest way to install is using prebuilt packages that are released with ROCm. Alternatively, there are instructions to build from source.

Available ROCm packages are:

- hiptensor (library + header files for development).
- hiptensor-dev (library development package).
- hiptensor-samples (sample executables).
- hiptensor-tests (test executables).
- hiptensor-clients (samples and test executables).

1.1 Prerequisites

- A ROCm 6.0 enabled platform. More information at ROCm Github.

1.2 Installing pre-built packages

To install hipTensor on Ubuntu or Debian, use:

```
sudo apt-get update
sudo apt-get install hiptensor hiptensor-dev hiptensor-samples hiptensor-tests
```

To install hipTensor on CentOS, use:

```
sudo yum update
sudo yum install hiptensor hiptensor-dev hiptensor-samples hiptensor-tests
```

To install hipTensor on SLES, use:

```
sudo dnf upgrade
sudo dnf install hiptensor hiptensor-dev hiptensor-samples hiptensor-tests
```

Once installed, hipTensor can be used just like any other library with a C++ API.

1.3 Building and installing hipTensor

For most users building from source is not necessary, as hipTensor can be used after installing the pre-built packages as described above. If still desired, here are the instructions to build hipTensor from source:

1.3.1 System requirements

As a general rule, 8GB of system memory is required for a full hipTensor build. This value can be lower if hipTensor is built without tests. This value may also increase in the future as more functions are added.

1.3.2 GPU support

AMD CDNA class GPU featuring matrix core support: *gfx908*, *gfx90a*, *gfx940*, *gfx941*, *gfx942* labeled as *gfx9*.

Note: Double precision FP64 datatype support requires *gfx90a*, *gfx940*, *gfx941* or *gfx942*.

1.3.3 Dependencies

hipTensor is designed to have minimal external dependencies such that it is light-weight and portable.

- Minimum ROCm version support is 6.0.
- Minimum cmake version support is 3.14.
- Minimum ROCm-cmake version support is 0.8.0.
- Minimum Composable Kernel version support is composable_kernel 1.1.0 for ROCm 6.0.2 (or ROCm package composablekernel-dev).
- Minimum HIP runtime version support is 4.3.0 (or ROCm package ROCm hip-runtime-amd).
- Minimum LLVM dev package version support is 7.0 (available as ROCm package rocm-llvm-dev).

Note: It is best to use available ROCm packages from the same release where applicable.

1.3.4 Download hipTensor

The hipTensor source code is available on hipTensor Github. hipTensor has a minimum ROCm support version 6.0. To check the ROCm Version on your system, use:

```
apt show rocm-libs -a
```

For Centos use

```
yum info rocm-libs
```

The ROCm version has major, minor, and patch fields, possibly followed by a build specific identifier. For example, a ROCm version 4.0.0.40000-23 corresponds to major = 4, minor = 0, patch = 0, and build identifier 40000-23. There are GitHub branches at the hipTensor site with names `rocm-major.minor.x` where major and minor are the same as in the ROCm version. To download hipTensor on ROCm version 4.0.0.40000-23, use:

```
git clone -b release/rocm-rel-x.y https://github.com/ROCmSoftwarePlatform/hipTensor.git
cd hipTensor
```

Replace `x.y` in the above command with the version of ROCm installed on your machine. For example, if you have ROCm 5.0 installed, then replace release/rocm-rel-x.y with release/rocm-rel-5.0.

1.3.5 Build documentation

To build documentation locally, run:

```
cd docs

sudo apt-get update
sudo apt-get install doxygen
sudo apt-get install texlive-latex-base texlive-latex-extra

pip3 install -r sphinx/requirements.txt

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

cd _build/latex

pdflatex hiptensor.tex
```

Running the above commands generates `hiptensor.pdf`. Alternatively, the latest docs build can be found at hipTensor docs.

1.3.6 Build configuration

You can choose to build any of the following:

- library only
- library and samples
- library and tests
- library, samples and tests

You only need the hipTensor library for calling and linking to hipTensor API from your code. The clients contain the tests and sample codes.

Below are the project options available to build hipTensor library with or without clients.

Option	Description	Default Value
AMDGPU_TARGETS	Build code for specific GPU target(s)	gfx908:xnack-; gfx90a:xnack-; gfx90a:xnack+; gfx940; gfx941; gfx942
HIPTENSOR_BUILD_TESTS	Build Tests	ON
HIPTENSOR_BUILD_SAMPLES	Build Samples	ON
HIPTENSOR_BUILD_COMPRESSED	Enable compressed debug symbols	ON
HIPTENSOR_DATA_LAYOUT_COL_	Set hiptensor default data layout to column major	ON

Here are some example project configurations:

Configuration	Command
Basic	`CC=/opt/rocm/bin/amdclang CXX=/opt/rocm/bin/amdclang++ cmake -B<build_dir> .`
Targeting gfx908	`CC=/opt/rocm/bin/amdclang CXX=/opt/rocm/bin/amdclang++ cmake -B<build_dir> . -DAMDGPU_TARGETS=gfx908:xnack-`
Debug build	`CC=/opt/rocm/bin/amdclang CXX=/opt/rocm/bin/amdclang++ cmake -B<build_dir> . -DCMAKE_BUILD_TYPE=Debug`

1.3.7 Build library

By default, the project is configured in Release mode.

To build the library alone, run:

```
CC=/opt/rocm/bin/amdclang CXX=/opt/rocm/bin/amdclang++ cmake -B <build_dir> . -
↪DHIPTENSOR_BUILD_TESTS=OFF -DHIPTENSOR_BUILD_SAMPLES=OFF
```

After configuration, build using:

```
cmake --build <build_dir> -- -j<nproc>
```

Note: We recommend using a minimum of 16 threads to build hipTensor with any tests (-j16).

1.3.8 Build library and samples

To build library and samples, run:

```
CC=/opt/rocm/bin/amdclang CXX=/opt/rocm/bin/amdclang++ cmake -B <build_dir> . -
↪DHIPTENSOR_BUILD_TESTS=OFF -DHIPTENSOR_BUILD_SAMPLES=ON
```

After configuration, build using:

```
cmake --build <build_dir> -- -j<nproc>
```

The samples folder in <build_dir> contains executables in the table below.

Executable Name	Description
simple_bilinear_contraction_bf16_bf16_bf16_b	A simple bilinear contraction [D = alpha * (A x B) + beta * C] using half-precision brain float inputs, output and compute types
simple_bilinear_contraction_f16_f16_f16_f16_	A simple bilinear contraction [D = alpha * (A x B) + beta * C] using half-precision floating point inputs, output and compute types
simple_bilinear_contraction_f32_f32_f32_f32_	A simple bilinear contraction [D = alpha * (A x B) + beta * C] using single-precision floating point input and output, half-precision brain float compute types
simple_bilinear_contraction_f32_f32_f32_f32_	A simple bilinear contraction [D = alpha * (A x B) + beta * C] using single-precision floating point input and output, half-precision floating point compute types
simple_bilinear_contraction_f32_f32_f32_f32_	A simple bilinear contraction [D = alpha * (A x B) + beta * C] using single-precision floating point input, output and compute types
simple_bilinear_contraction_cf32_cf32_cf32_c	A simple bilinear contraction [D = alpha * (A x B) + beta * C] using complex single-precision floating point input, output and compute types
simple_bilinear_contraction_f64_f64_f64_f64_	A simple bilinear contraction [D = alpha * (A x B) + beta * C] using double-precision floating point input, output and single precision floating point compute types
simple_bilinear_contraction_f64_f64_f64_f64_	A simple bilinear contraction [D = alpha * (A x B) + beta * C] using double-precision floating point input, output and compute types
simple_scale_contraction_bf16_bf16_bf16_comp	A simple scale contraction [D = alpha * (A x B)] using half-precision brain float inputs, output and compute types
simple_scale_contraction_f16_f16_f16_compute	A simple scale contraction [D = alpha * (A x B)] using half-precision floating point inputs, output and compute types
simple_scale_contraction_f32_f32_f32_compute	A simple scale contraction [D = alpha * (A x B)] using single-precision floating point input and output, half-precision brain float compute types
simple_scale_contraction_f32_f32_f32_compute	A simple scale contraction [D = alpha * (A x B)] using single-precision floating point input and output, half-precision floating point compute types
simple_scale_contraction_f32_f32_f32_compute	A simple scale contraction [D = alpha * (A x B)] using single-precision floating point input, output and compute types
simple_scale_contraction_cf32_cf32_cf32_comp	A simple scale contraction [D = alpha * (A x B)] using complex single-precision floating point input, output and compute types
simple_scale_contraction_f64_f64_f64_compute	A simple scale contraction [D = alpha * (A x B)] using double-precision floating point input, output and single precision floating point compute types
simple_scale_contraction_f64_f64_f64_compute	A simple scale contraction [D = alpha * (A x B)] using double-precision floating point input, output and compute types
simple_permutation	A simple permutation using single-precision floating point input and output types

1.3.9 Build library and tests

To build library and tests, run:

```
CC=/opt/rocm/bin/amdclang CXX=/opt/rocm/bin/amdclang++ cmake -B <build_dir> . -
↪DHIPTENSOR_BUILD_TESTS=ON -DHIPTENSOR_BUILD_SAMPLES=OFF
```

After configuration, build using:

```
cmake --build <build_dir> -- -j<nproc>
```

The tests in <build_dir> contain executables as given in the table below.

Executable name	Description
logger_test	Unit test to validate hipTensor Logger APIs
yaml_test	Unit test to validate the YAML functionality used to bundle and run test suites
bilinear_contraction_test_m1n1k1	Bilinear contraction test [D = alpha * (A x B) + beta * C] with half, single and mixed precision datatypes of rank 2
bilinear_contraction_test_m2n2k2	Bilinear contraction test [D = alpha * (A x B) + beta * C] with half, single and mixed precision datatypes of rank 4
bilinear_contraction_test_m3n3k3	Bilinear contraction test [D = alpha * (A x B) + beta * C] with half, single and mixed precision datatypes of rank 6
bilinear_contraction_test_m4n4k4	Bilinear contraction test [D = alpha * (A x B) + beta * C] with half, single and mixed precision datatypes of rank 8
bilinear_contraction_test_m5n5k5	Bilinear contraction test [D = alpha * (A x B) + beta * C] with half, single and mixed precision datatypes of rank 10
bilinear_contraction_test_m6n6k6	Bilinear contraction test [D = alpha * (A x B) + beta * C] with half, single and mixed precision datatypes of rank 12
complex_bilinear_contraction_test_m1n2k1	Bilinear contraction test [D = alpha * (A x B) + beta * C] with complex single and double precision datatypes of rank 2
complex_bilinear_contraction_test_m2n2k2	Bilinear contraction test [D = alpha * (A x B) + beta * C] with complex single and double precision datatypes of rank 4
complex_bilinear_contraction_test_m3n3k3	Bilinear contraction test [D = alpha * (A x B) + beta * C] with complex single and double precision datatypes of rank 6
complex_bilinear_contraction_test_m4n4k4	Bilinear contraction test [D = alpha * (A x B) + beta * C] with complex single and double precision datatypes of rank 8
complex_bilinear_contraction_test_m5n5k5	Bilinear contraction test [D = alpha * (A x B) + beta * C] with complex single and double precision datatypes of rank 10

continues on next page

Table 1.1 – continued from previous page

Executable name	Description
complex_bilinear_contraction_test_m6n6k6	Bilinear contraction test [D = alpha * (A x B) + beta * C] with complex single and double precision datatypes of rank 12
scale_contraction_test_m1n1k1	Scale contraction test [D = alpha * (A x B)] with half, single and mixed precision datatypes of rank 2
scale_contraction_test_m2n2k2	Scale contraction test [D = alpha * (A x B)] with half, single and mixed precision datatypes of rank 4
scale_contraction_test_m3n3k3	Scale contraction test [D = alpha * (A x B)] with half, single and mixed precision datatypes of rank 6
scale_contraction_test_m4n4k4	Scale contraction test [D = alpha * (A x B)] with half, single and mixed precision datatypes of rank 8
scale_contraction_test_m5n5k5	Scale contraction test [D = alpha * (A x B)] with half, single and mixed precision datatypes of rank 10
scale_contraction_test_m6n6k6	Scale contraction test [D = alpha * (A x B)] with half, single and mixed precision datatypes of rank 12
complex_scale_contraction_test_m1n1k1	Scale contraction test [D = alpha * (A x B)] with complex single and double precision datatypes of rank 2
complex_scale_contraction_test_m2n2k2	Scale contraction test [D = alpha * (A x B)] with complex single and double precision datatypes of rank 4
complex_scale_contraction_test_m3n3k3	Scale contraction test [D = alpha * (A x B)] with complex single and double precision datatypes of rank 6
complex_scale_contraction_test_m4n4k4	Scale contraction test [D = alpha * (A x B)] with complex single and double precision datatypes of rank 8
complex_scale_contraction_test_m5n5k5	Scale contraction test [D = alpha * (A x B)] with complex single and double precision datatypes of rank 10
complex_scale_contraction_test_m6n6k6	Scale contraction test [D = alpha * (A x B)] with complex single and double precision datatypes of rank 12
rank2_permutation_test	Permutation test with half and single precision datatypes of rank 2
rank3_permutation_test	Permutation test with half and single precision datatypes of rank 3
rank4_permutation_test	Permutation test with half and single precision datatypes of rank 4
rank5_permutation_test	Permutation test with half and single precision datatypes of rank 5
rank6_permutation_test	Permutation test with half and single precision datatypes of rank 6

1.3.10 Make targets list

When building hipTensor during the `make` step, we can specify make targets instead of defaulting `make all`. The following table highlights relationships between high level grouped targets and individual targets.

Group Target	Individual Targets
hiptensor_samples	simple_bilinear_contraction_bf16_bf16_bf16_bf16_compute_bf16
	simple_bilinear_contraction_f16_f16_f16_f16_compute_f16
	simple_bilinear_contraction_f32_f32_f32_f32_compute_bf16
	simple_bilinear_contraction_f32_f32_f32_f32_compute_f16

continues on next page

Table 1.2 – continued from previous page

Group Target	Individual Targets
	simple_bilinear_contraction_f32_f32_f32_f32_compute_f32
	simple_bilinear_contraction_cf32_cf32_cf32_cf32_compute_cf32
	simple_bilinear_contraction_f64_f64_f64_f64_compute_f32
	simple_bilinear_contraction_f64_f64_f64_f64_compute_f64
	simple_scale_contraction_bf16_bf16_bf16_compute_bf16
	simple_scale_contraction_f16_f16_f16_compute_f16
	simple_scale_contraction_f32_f32_f32_compute_bf16
	simple_scale_contraction_f32_f32_f32_compute_f16
	simple_scale_contraction_f32_f32_f32_compute_f32
	simple_scale_contraction_cf32_cf32_cf32_compute_cf32
	simple_scale_contraction_f64_f64_f64_compute_f32
	simple_scale_contraction_f64_f64_f64_compute_f64
	simple_permutation
hiptensor_tests	logger_test
	yaml_test
	bilinear_contraction_test_m1n1k1
	bilinear_contraction_test_m2n2k2
	bilinear_contraction_test_m3n3k3
	bilinear_contraction_test_m4n4k4
	bilinear_contraction_test_m5n5k5
	bilinear_contraction_test_m6n6k6
	complex_bilinear_contraction_test_m1n1k1
	complex_bilinear_contraction_test_m2n2k2
	complex_bilinear_contraction_test_m3n3k3
	complex_bilinear_contraction_test_m4n4k4
	complex_bilinear_contraction_test_m5n5k5
	complex_bilinear_contraction_test_m6n6k6
	scale_contraction_test_m1n1k1
	scale_contraction_test_m2n2k2
	scale_contraction_test_m3n3k3
	scale_contraction_test_m4n4k4
	scale_contraction_test_m5n5k5
	scale_contraction_test_m6n6k6
	complex_scale_contraction_test_m1n1k1
	complex_scale_contraction_test_m2n2k2
	complex_scale_contraction_test_m3n3k3
	complex_scale_contraction_test_m4n4k4
	complex_scale_contraction_test_m5n5k5
	complex_scale_contraction_test_m6n6k6
	rank2_permutation_test
	rank3_permutation_test
	rank4_permutation_test
	rank5_permutation_test
	rank6_permutation_test

1.3.11 Build performance

Depending on the resources available to the build machine and the build configuration selected, hipTensor build times can be on the order of an hour or more. Here are some things you can do to reduce build times:

- Target a specific GPU (e.g., `-D AMDGPU_TARGETS=gfx908:xnack-`)
- Use lots of threads (e.g., `-j32`)
- If they aren't needed, specify either `HIPTENSOR_BUILD_TESTS` or `HIPTENSOR_BUILD_SAMPLES` as OFF to disable client builds.
- During the `make` command, build a specific target, e.g: `logger_test`.

1.3.12 Test run lengths

Depending on the resources available to the machine running the selected tests, hipTensor test runtimes can be on the order of an hour or more. Here are some things you can do to reduce run-times:

- CTest will invoke the entire test suite. You may invoke tests individually by name.
- Use GoogleTest filters, targeting specific test cases:

```
<test_exe> --gtest_filter=*name_filter*
```

- Manually adjust the test cases coverage. Using your favorite text editor, you can modify test YAML configs to affect the test parameter coverage.
- Alternatively, use your own testing YAML config with a reduced parameter set.
- For tests with large tensor ranks, avoid using larger lengths to reduce computational load.

1.3.13 Test verbosity and file redirection

Tests support logging arguments that can be used to control verbosity and output redirection.

```
<test_exe> -y "testing_params.yaml" -o "output.csv" --omit 1
```

Compact	Verbose	Description
-y <input_file>.yaml		override read testing parameters from input file
-o <output_file>.csv		redirect gtest output to file
	–omit <code>	code = 1: Omit gtest SKIPPED tests
		code = 2: Omit gtest FAILED tests
		code = 4: Omit gtest PASSED tests
		code = 8: Omit all gtest output
		code = <N>: OR combination of 1, 2, 4

CHAPTER

TWO

PROGRAMMER'S GUIDE

This document provides insight into the library source code organization, design implementation details, helpful information for new development, and testing and benchmarking details.

2.1 Infrastructure

- Doxygen and Sphinx are used to generate the project's documentation.
- Jenkins is used to automate Continuous Integration (CI) testing (`.jenkins` folder has configurations).
- hipTensor is hosted and maintained by AMD on Github.
- The hipTensor project is organized and configured via `CMake` and the collection of `CMakeLists.txt` in the base of each directory.
- `clang-format` is used to format C++ code. `.githooks/install` ensures that a clang-format pass will run on each committed file.
- `GTest` is used to implement test suite organization and execution.
- `CTest` is used to consolidate and invoke multiple test targets. In the `<hipTensor_install_dir>/CTestTestfile.cmake` file, testing targets are listed that will be run when `ctest` is invoked.
- The preferred compiler for hipTensor is `CC=<path_to_rocm>/bin/amdclang and CXX=<path_to_rocm>/bin/amdclang++`. `hipcc` is also supported, however may be deprecated in future ROCm releases.

2.2 General Design Concepts

hipTensor is developed with the `C++17` language standard. The library takes advantage of several meta-programming techniques that help to statically optimize code at compile time and generate more efficient GPU kernels. hipTensor employs Composable Kernel as a functional backend, therefore the library is written in different layers.

The outer API layer serves as a functional interface for the user to define tensor data abstractions and manipulations. The second layer is the hipTensor solution interface that bridges the communication gap between the API objects and the desired functionality. The solution layer encompasses the translation of the input problem parameters into solution candidates, candidate selection, resource management and logging. Solution candidates provide interface abstractions into functional backends such as Composable Kernel objects which may be invoked and whose results are passed back up through the API. The Composable Kernel library is consumed as a header library where all kernel instances are customized by hipTensor and statically bundled which is managed by the hipTensor functional backend layer. This way if additional backends were to be considered in the future, the backends could be isolated into their own modules as they are now. The hipTensor solution layer is also split up into functional components, such as permutation and contraction. Each component contains a registry of backend instances which are held as potential solution candidates

to a given set of input parameters. These instances go through selection processing as directed with hints from the API, and are populated with appropriate arguments and readied for invocation by the API.

hipTensor tests and samples are consumers of the hipTensor library and demonstrate the usages of the API in different contexts, such as tensor contractions and permutations.

2.3 Nomenclature

2.3.1 Tensor contraction

In general, a tensor contraction is a multiply-accumulate problem over elements between two multi-dimensional tensors. hipTensor will use the Einstein notation for the contraction notation. Repeated indices are summed over, and each index may appear a maximum of twice in each mathematical term. In the process of accumulating over summation dimensions, they are effectively collapsed, or contracted.

2.3.2 Tensor permutation

Tensor permutation is essentially the re-ordering of the stride indices such that the data dimensional locality relationships are changed.

2.3.3 Tensor rank

In terms of tensor rank, this is considered the dimensionality of the data. For example this would be the number of modes. We consider Einstein notation in contractions such as repeated modes are the dimensions which are contracted.

In contractions, we may differentiate modes in terms of M's, N's and K's, in which:

- Tensor A modes [M0, ..., Mn, K0, ..., Kn]
- Tensor B modes [N0, ..., Nn, K0, ..., Kn]
- Tensor C/D modes [M0, ..., Mn, N0, ..., Nn]

and repeated indices K0, ..., Kn are indices that are contracted. Contractions currently support up to M6N6K6 which means we may have up to 6 dimensions for each M, N and K. A tensor contraction with A = [M0, ..., M5, K0, ..., K5] B = [N0, ..., N5, K0, ..., K5] and C/D [M0, ..., M5, N0, ..., N5] would be considered a rank 12 contraction.

2.3.4 Tensor mode

Tensor modes are a way for the user to easily specify the ordering or labeling of the input strides that define the dimensional data relationship. This is used to describe how the data is laid out in memory and how they are related to each other spatially.

2.4 Library source code organization

The hipTensor code is split into four major parts:

- The `library` directory contains the library source code.
- The `samples` directory contains real-world use-cases of the hipTensor API.
- The `test` directory contains validation tests for hipTensor API.
- Infrastructure

2.4.1 `library` directory

The `library` directory contains the following include and source files:

- `library/include/hiptensor/`: C++ include files for the hipTensor API. These files also contain Doxygen comments that document the API.
- `library/include/hiptensor/internal`: Include files for utility code and generate tensor utility.
- `library/src/`: Source files for Logger, device, and performance functions.
- `library/src/contraction/`: Source files for core initialization and management of contraction module.
- `library/src/contraction/device`: Source files for composable kernel backend bilinear and scale instances.
- `library/src/permutation/`: Source files for core initialization and management of permutation module.
- `library/src/permutation/device`: Source files for composable kernel backend permute instances.
- `library/src/include`: Infrastructure support for backend and logging management.

2.4.2 `samples` directory

The `samples` directory contains the sample codes for the following simple demonstrations:

- `01_contraction/simple_bilinear_contraction`: Abstract base test for bilinear contractions.
- `01_contraction/simple_scale_contraction`: Abstract base test for scale contractions.
- `01_contraction/simple_bilinear_contraction_*`: Specialized bilinear contraction tests per data type.
- `01_contraction/simple_scale_contraction_*`: Specialized bilinear contraction tests per data type.
- `02_permutation/permutation`: Simple permutation demonstration.

2.4.3 `test` directory

The `test` directory contains the test codes for testing the following functionalities:

- `00_unit/logger_test`: Tests logger API functions of hipTensor.
- `00_unit/yaml_test`: Tests the YAML serialization / de-serialization for testing parameters.
- `01_contraction/contraction_test`: Testing harness for the bilinear and scale contractions.
- `01_contraction/complex_*_contraction`: Testing harness for the bilinear and scale contractions with complex data types.
- `01_contraction/contraction_resource`: Shared resource infrastructure for testing contractions.

- `01_contraction/configs`: YAML files with actual contraction testing parameters.
- `02_permutation/permutation*`: Testing infrastructure for permutation tests.
- `02_permutation/rank*`: Testing harnesses for permutation of a particular rank.
- `02_permutation/configs`: YAML files with actual permutation testing parameters.

2.4.4 Contributing

For those wishing to contribute to the project, please see *Contributing to hipTensor*.

CHAPTER

THREE

API REFERENCE GUIDE

This document provides information about hipTensor APIs, data types, and other programming constructs.

3.1 Supported GPU architectures

List of supported CDNA architectures:

- gfx908
- gfx90a
- gfx940
- gfx941
- gfx942

Note: gfx9 = gfx908, gfx90a, gfx940, gfx941, gfx942

gfx940+ = gfx940, gfx941, gfx942

3.2 Supported data types

hipTensor supports the following datatype combinations in API functionality.

Data Types **<Ti / To / Tc>** = <Input type / Output Type / Compute Type>, where:

- Input Type = Matrix A / B
- Output Type = Matrix C / D
- Compute Type = Math / accumulation type
- f16 = half-precision floating point
- bf16 = half-precision brain floating point
- f32 = single-precision floating point
- cf32 = complex single-precision floating point
- f64 = double-precision floating point
- cf64 = complex double-precision floating point

Note: f16 represents equivalent support for both _Float16 and __half types.

API context	Datatype Support <Ti / To / Tc>	GPU Support	Tensor Rank Support
Contraction (Scale, bilinear)	f16 / f16 / f32	gfx908 gfx90a	2m2n2k (Rank4)
	bf16 / bf16 / f32	gfx940+	3m3n3k (Rank6)
	f32 / f32 / f32		4m4n4k (Rank8)
	f32 / f32 / f16		5m5n5k (Rank10)
	f32 / f32 / bf16		6m6n6k (Rank12)
	cf32 / cf32 / cf32		
	f64 / f64 / f64	gfx940+	
	f64 / f64 / f32		
	cf64 / cf64 / cf64		
Permutation	f16 / f16 / -	gfx908 gfx90a	Rank2 - Rank6
	f16 / f32 / -	gfx940+	
	f32 / f32 / -		

3.3 Limitations

- hipTensor currently supports tensors up to 2GB in size due to backend address-space limitations.

3.3.1 hipTensor API objects

3.4 hiptensorStatus_t

enum **hiptensorStatus_t**

hipTensor status type enumeration

The type is used to indicate the resulting status of hipTensor library function calls

Values:

enumerator **HIPTENSOR_STATUS_SUCCESS**

The operation is successful.

enumerator **HIPTENSOR_STATUS_NOT_INITIALIZED**

The handle was not initialized.

enumerator **HIPTENSOR_STATUS_ALLOC_FAILED**

Resource allocation failed inside the hipTensor library.

enumerator **HIPTENSOR_STATUS_INVALID_VALUE**

Invalid value or parameter was passed to the function (indicates a user error).

enumerator **HIPTENSOR_STATUS_ARCH_MISMATCH**

Indicates that the target architecure is not supported, or the device is not ready.

enumerator **HIPTENSOR_STATUS_EXECUTION_FAILED**

Indicates the failure of a GPU program or a kernel, which can be caused by multiple reasons.

enumerator **HIPTENSOR_STATUS_INTERNAL_ERROR**

An internal error has occurred.

enumerator **HIPTENSOR_STATUS_NOT_SUPPORTED**

The requested operation is not supported.

enumerator **HIPTENSOR_STATUS_CK_ERROR**

A call to Composable Kernels did not succeed.

enumerator **HIPTENSOR_STATUS_HIP_ERROR**

Unknown hipTensor error has occurred.

enumerator **HIPTENSOR_STATUS_INSUFFICIENT_WORKSPACE**

The provided workspace was insufficient.

enumerator **HIPTENSOR_STATUS_INSUFFICIENT_DRIVER**

Indicates that the driver version is insufficient.

enumerator **HIPTENSOR_STATUS_IO_ERROR**

Indicates an error related to file I/O.

3.5 hiptensorComputeType_t

enum **hiptensorComputeType_t**

hipTensor compute type enumeration

Values:

enumerator **HIPTENSOR_COMPUTE_32F**

Single precision floating point.

enumerator **HIPTENSOR_COMPUTE_64F**

Double precision floating point.

enumerator **HIPTENSOR_COMPUTE_16F**

Half precision floating point.

enumerator **HIPTENSOR_COMPUTE_16BF**

Brain float half precision floating point.

enumerator **HIPTENSOR_COMPUTE_C32F**

Complex single precision floating point.

enumerator **HIPTENSOR_COMPUTE_C64F**

Complex double precision floating point.

enumerator **HIPTENSOR_COMPUTE_NONE**

No type.

3.6 hiptensorOperator_t

enum **hiptensorOperator_t**

Element-wise operations.

Values:

enumerator **HIPTENSOR_OP_IDENTITY**

Identity operator.

enumerator **HIPTENSOR_OP_SQRT**

Square root operator.

enumerator **HIPTENSOR_OP_UNKNOWN**

Reserved.

3.7 hiptensorAlgo_t

enum **hiptensorAlgo_t**

Tensor contraction kernel selection algorithm.

Values:

enumerator **HIPTENSOR_ALGO_ACTOR_CRITIC**

Uses novel actor-critic selection model.

enumerator **HIPTENSOR_ALGO_DEFAULT**

Lets the internal heuristic choose.

enumerator **HIPTENSOR_ALGO_DEFAULT_PATIENT**

Uses the more accurate and time-consuming model.

3.8 hiptensorWorksizePreference_t

enum **hiptensorWorksizePreference_t**

Workspace size selection.

Values:

enumerator **HIPTENSOR_WORKSPACE_MIN**

At least one algorithm will be available.

enumerator **HIPTENSOR_WORKSPACE_RECOMMENDED**

The most suitable algorithm will be available.

enumerator **HIPTENSOR_WORKSPACE_MAX**

All algorithms will be available.

3.9 hiptensorLogLevel_t

enum **hiptensorLogLevel_t**

Logging context.

The logger output of certain contexts maybe constrained to these levels

Values:

enumerator **HIPTENSOR_LOG_LEVEL_OFF**

No logging.

enumerator **HIPTENSOR_LOG_LEVEL_ERROR**

Log errors.

enumerator **HIPTENSOR_LOG_LEVEL_PERF_TRACE**

Log performance messages.

enumerator **HIPTENSOR_LOG_LEVEL_PERF_HINT**

Log performance hints.

enumerator **HIPTENSOR_LOG_LEVEL_HEURISTICS_TRACE**

Log selection messages.

enumerator **HIPTENSOR_LOG_LEVEL_API_TRACE**

Log a trace of API calls.

3.10 hiptensorHandle_t

struct **hiptensorHandle_t**

hipTensor's library context

3.11 hiptensorTensorDescriptor_t

struct **hiptensorTensorDescriptor_t**

Structure representing a tensor descriptor.

Represents a descriptor for the tensor with the given properties of data type, lengths, strides and element-wise unary operation. Constructed with *hiptensorInitTensorDescriptor()* function.

Public Members

hipDataType **mType**

Data type of the tensors enum selection.

std::vector<std::size_t> **mLengths**

Lengths of the tensor.

std::vector<std::size_t> **mStrides**

Strides of the tensor.

hiptensorOperator_t **mUnaryOp**

Unary operator applied to the tensor.

3.12 hiptensorContractionDescriptor_t

struct **hiptensorContractionDescriptor_t**

Structure representing a tensor contraction descriptor.

Represents contraction descriptor with the given properties of internal contraction op (either scale or bilinear), the internal compute type, as well as all of the input tensor descriptors, their alignment requirements and modes. Constructed with *hiptensorInitContractionDescriptor()* function.

Public Members

int32_t **mContractionOpId**

Enum that differentiates the internal contraction operation.

hiptensorComputeType_t **mComputeType**

Compute type for the contraction.

std::vector<*hiptensorTensorDescriptor_t*> **mTensorDesc**

Cache of tensor descriptors.

std::vector<uint32_t> **mAlignmentReq**

Cache of alignment requirements.

std::vector<std::vector<int32_t>> **mTensorMode**

Tensor modes.

3.13 hiptensorContractionFind_t

struct **hiptensorContractionFind_t**

hipTensor structure representing the contraction selection algorithm and candidates.

Public Members

hiptensorAlgo_t **mSelectionAlgorithm**

Id of the selection algorithm.

std::vector<void*> **mCandidates**

A vector of the solver candidates.

3.14 hiptensorContractionPlan_t

struct **hiptensorContractionPlan_t**

hipTensor structure representing a contraction plan. Constructed with the *hiptensorInitContractionPlan()* function.

Public Members

void *mSolution

Final solution candidate.

hiptensorContractionDescriptor_t mContractionDesc

Contraction parameters.

3.14.1 Helper functions

3.15 hiptensorCreate

hiptensorStatus_t **hiptensorCreate**(*hiptensorHandle_t* **handle)

Allocates an instance of *hiptensorHandle_t* on the heap and updates the handle pointer.

Creates hipTensor handle for the associated device. In order for the hipTensor library to use a different device, set the new device to be used by calling hipInit(0) and then create another hipTensor handle, which will be associated with the new device, by calling *hiptensorCreate()*.

Parameters
handle – **[out]** Pointer to *hiptensorHandle_t* pointer

Returns
HIPTENSOR_STATUS_SUCCESS on success and an error code otherwise

3.16 hiptensorDestroy

hiptensorStatus_t **hiptensorDestroy**(*hiptensorHandle_t* *handle)

De-allocates the instance of *hiptensorHandle_t*.

Parameters
handle – **[out]** Pointer to *hiptensorHandle_t*

Returns
HIPTENSOR_STATUS_SUCCESS on success and an error code otherwise

3.17 hiptensorInitTensorDescriptor

hiptensorStatus_t **hiptensorInitTensorDescriptor**(const *hiptensorHandle_t* *handle, *hiptensorTensorDescriptor_t* *desc, const uint32_t numModes, const int64_t lens[], const int64_t strides[], hipDataType dataType, *hiptensorOperator_t* unaryOp)

Initializes a tensor descriptor.

Parameters

- **handle** – **[in]** Opaque handle holding hipTensor's library context.
- **desc** – **[out]** Pointer to the allocated tensor descriptor object.
- **numModes** – **[in]** Number of modes.

- **lens** – **[in]** Extent of each mode(lengths) (must be larger than zero).
- **strides** – **[in]** stride[i] denotes the displacement (stride) between two consecutive elements in the ith-mode. If stride is NULL, generalized packed column-major memory layout is assumed (i.e., the strides increase monotonically from left to right).
- **dataType** – **[in]** Data type of the stored entries.
- **unaryOp** – **[in]** Unary operator that will be applied to the tensor.

Return values

- **HIPTENSOR_STATUS_SUCCESS** – The operation completed successfully.
- **HIPTENSOR_STATUS_NOT_INITIALIZED** – if the handle is not initialized.

3.18 hiptensorGetAlignmentRequirement

hiptensorStatus_t **hiptensorGetAlignmentRequirement**(const *hiptensorHandle_t* *handle, const void *ptr, const *hiptensorTensorDescriptor_t* *desc, uint32_t *alignmentRequirement)

Computes the alignment requirement for a given pointer and descriptor.

Parameters

- **handle** – **[in]** Opaque handle holding hipTensor's library context.
- **ptr** – **[in]** Pointer to the respective tensor data.
- **desc** – **[in]** Tensor descriptor for ptr data.
- **alignmentRequirement** – **[out]** Largest alignment requirement that ptr can fulfill (in bytes).

Return values

- **HIPTENSOR_STATUS_SUCCESS** – The operation completed successfully.
- **HIPTENSOR_STATUS_NOT_INITIALIZED** – if the handle is not initialized.
- **HIPTENSOR_STATUS_INVALID_VALUE** – if the unsupported parameter is passed.

3.19 hiptensorGetErrorString

const char ***hiptensorGetErrorString**(const *hiptensorStatus_t* error)

Returns the description string for an error code.

Parameters

error – **[in]** Error code to convert to string.

Return values

the – error string.

3.19.1 Contraction operations

3.20 hiptensorInitContractionDescriptor

hiptensorStatus_t **hiptensorInitContractionDescriptor**(const *hiptensorHandle_t* *handle, *hiptensorContractionDescriptor_t* *desc, const *hiptensorTensorDescriptor_t* *descA, const int32_t modeA[], const uint32_t alignmentRequirementA, const *hiptensorTensorDescriptor_t* *descB, const int32_t modeB[], const uint32_t alignmentRequirementB, const *hiptensorTensorDescriptor_t* *descC, const int32_t modeC[], const uint32_t alignmentRequirementC, const *hiptensorTensorDescriptor_t* *descD, const int32_t modeD[], const uint32_t alignmentRequirementD, *hiptensorComputeType_t* typeCompute)

Initializes a contraction descriptor for the tensor contraction problem.

Parameters

- **handle** – **[in]** Opaque handle holding hipTensor's library context.
- **desc** – **[out]** Tensor contraction problem descriptor.
- **descA** – **[in]** A descriptor that holds information about tensor A.
- **modeA** – **[in]** Array with 'nmodeA' entries that represent the modes of A.
- **alignmentRequirementA** – **[in]** Alignment reqirement for A's pointer (in bytes);
- **descB** – **[in]** A descriptor that holds information about tensor B.
- **modeB** – **[in]** Array with 'nmodeB' entries that represent the modes of B.
- **alignmentRequirementB** – **[in]** Alignment reqirement for B's pointer (in bytes);
- **modeC** – **[in]** Array with 'nmodeC' entries that represent the modes of C.
- **descC** – **[in]** A descriptor that holds information about tensor C.
- **alignmentRequirementC** – **[in]** Alignment requirement for C's pointer (in bytes);
- **modeD** – **[in]** Array with 'nmodeD' entries that represent the modes of D (must be identical to modeC).
- **descD** – **[in]** A descriptor that holds information about tensor D (must be identical to descC).
- **alignmentRequirementD** – **[in]** Alignment requirement for D's pointer (in bytes);
- **typeCompute** – **[in]** Datatype for the intermediate computation T = A * B.

Return values

- **HIPTENSOR_STATUS_SUCCESS** – Successful completion of the operation.
- **HIPTENSOR_STATUS_NOT_INITIALIZED** – if the handle or tensor descriptors are not initialized.

3.21 hiptensorInitContractionFind

hiptensorStatus_t **hiptensorInitContractionFind**(const *hiptensorHandle_t* *handle, *hiptensorContractionFind_t* *find, const *hiptensorAlgo_t* algo)

Narrows down the candidates for the contraction problem.

This function gives the user finer control over the candidates that the subsequent call to *hiptensorInitContractionPlan* is allowed to evaluate. Currently, the backend provides few set of algorithms(DEFAULT).

Parameters

- **handle** – **[in]** Opaque handle holding hipTensor's library context.
- **find** – **[out]** Narrowed set of candidates for the contraction problem.
- **algo** – **[in]** Allows users to select a specific algorithm.

Return values

- **HIPTENSOR_STATUS_SUCCESS** – The operation completed successfully.
- **HIPTENSOR_STATUS_NOT_SUPPORTED** – If a specified algorithm is not supported
- **HIPTENSOR_STATUS_NOT_INITIALIZED** – if the handle or find is not initialized.

3.22 hiptensorInitContractionPlan

hiptensorStatus_t **hiptensorInitContractionPlan**(const *hiptensorHandle_t* *handle, *hiptensorContractionPlan_t* *plan, const *hiptensorContractionDescriptor_t* *desc, const *hiptensorContractionFind_t* *find, const uint64_t workspaceSize)

Initializes the contraction plan for a given tensor contraction problem.

This function creates a contraction plan for the problem by applying hipTensor's heuristics to select a candidate. The creaated plan can be reused multiple times for the same tensor contraction problem. The plan is created for the active HIP device.

Parameters

- **handle** – **[in]** Opaque handle holding hipTensor's library context.
- **plan** – **[out]** Opaque handle holding the contraction plan (i.e., the algorithm that will be executed, its runtime parameters for the given tensor contraction problem).
- **desc** – **[in]** Tensor contraction descriptor.
- **find** – **[in]** Narrows down the candidates for the contraction problem.
- **workspaceSize** – **[in]** Available workspace size (in bytes).

Return values

- **HIPTENSOR_STATUS_SUCCESS** – If a viable candidate has been found.
- **HIPTENSOR_STATUS_NOT_INITIALIZED** – if the handle or find or desc is not initialized.

3.23 hiptensorContraction

hiptensorStatus_t **hiptensorContraction**(const *hiptensorHandle_t* *handle, const *hiptensorContractionPlan_t* *plan, const void *alpha, const void *A, const void *B, const void *beta, const void *C, void *D, void *workspace, uint64_t workspaceSize, hipStream_t stream)

Computes the tensor contraction.

$$D = alpha * A * B + beta * C$$

Parameters

- **handle** – **[in]** Opaque handle holding hipTensor's library context. HIP Device associated with the handle must be same/active at the time,0 the plan was created.
- **plan** – **[in]** Opaque handle holding the contraction plan (i.e., the algorithm that will be executed, its runtime parameters for the given tensor contraction problem).
- **alpha** – **[in]** Scaling parameter for A*B of data type 'typeCompute'.
- **A** – **[in]** Pointer to A's data in device memory.
- **B** – **[in]** Pointer to B's data in device memory.
- **beta** – **[in]** Scaling parameter for C of data type 'typeCompute'.
- **C** – **[in]** Pointer to C's data in device memory.
- **D** – **[out]** Pointer to D's data in device memory.
- **workspace** – **[out]** Workspace pointer in device memory
- **workspaceSize** – **[in]** Available workspace size.
- **stream** – **[in]** HIP stream to perform all operations.

Return values

- **HIPTENSOR_STATUS_SUCCESS** – Successful completion of the operation.
- **HIPTENSOR_STATUS_NOT_INITIALIZED** – if the handle or pointers are not initialized.
- **HIPTENSOR_STATUS_CK_ERROR** – if some unknown composable_kernel (CK) error has occurred (e.g., no instance supported by inputs).

3.24 hiptensorContractionGetWorkspaceSize

hiptensorStatus_t **hiptensorContractionGetWorkspaceSize**(const *hiptensorHandle_t* *handle, const *hiptensorContractionDescriptor_t* *desc, const *hiptensorContractionFind_t* *find, const *hiptensorWorksizePreference_t* pref, uint64_t *workspaceSize)

Computes the size of workspace for a given tensor contraction.

Parameters

- **handle** – **[in]** Opaque handle holding hipTensor's library context.
- **desc** – **[in]** Tensor contraction descriptor.

- **find** – **[in]** Narrowed set of candidates for the contraction problem.
- **pref** – **[in]** Preference to choose the workspace size.
- **workspaceSize** – **[out]** Size of the workspace (in bytes).

Return values

- **HIPTENSOR_STATUS_SUCCESS** – Successful completion of the operation.
- **HIPTENSOR_STATUS_NOT_INITIALIZED** – if the handle is not initialized.
- **HIPTENSOR_STATUS_INVALID_VALUE** – if some input data is invalid (this typically indicates an user error).

3.24.1 Logging functions

3.25 hiptensorLoggerSetCallback

hiptensorStatus_t **hiptensorLoggerSetCallback**(hiptensorLoggerCallback_t callback)

Registers a callback function that will be invoked by logger calls.

Parameters

callback – **[in]** This parameter is the callback function pointer provided to the logger.

Return values

- **HIPTENSOR_STATUS_SUCCESS** – if the operation completed successfully.
- **HIPTENSOR_STATUS_INVALID_VALUE** – if the given callback is invalid.

3.26 hiptensorLoggerSetFile

hiptensorStatus_t **hiptensorLoggerSetFile**(FILE *file)

Registers a file output stream to redirect logging output to.

Note: File stream must be open and writable in text mode.

Parameters

file – **[in]** This parameter is a file stream pointer provided to the logger.

Return values

- **HIPTENSOR_STATUS_SUCCESS** – if the operation completed successfully.
- **HIPTENSOR_STATUS_IO_ERROR** – if the output file is not valid (defaults back to stdout).

3.27 hiptensorLoggerOpenFile

hiptensorStatus_t **hiptensorLoggerOpenFile**(const char *logFile)

Redirects log output to a file given by the user.

Parameters

logFile – **[in]** This parameter is a file name (relative to binary) or full path to redirect logger output.

Return values

- **HIPTENSOR_STATUS_SUCCESS** – if the operation completed successfully.
- **HIPTENSOR_STATUS_IO_ERROR** – if the output file is not valid (defaults back to stdout).

3.28 hiptensorLoggerSetLevel

hiptensorStatus_t **hiptensorLoggerSetLevel**(*hiptensorLogLevel_t* level)

User-specified logging level. Logs in other contexts will not be recorded.

Parameters

level – **[in]** This parameter is the logging level to be enforced.

Return values

- **HIPTENSOR_STATUS_SUCCESS** – if the operation completed successfully.
- **HIPTENSOR_STATUS_INVALID_VALUE** – if the given log level is invalid.

3.29 hiptensorLoggerSetMask

hiptensorStatus_t **hiptensorLoggerSetMask**(int32_t mask)

User-specified logging mask. A mask may be a binary OR combination of several log levels together. Logs in other contexts will not be recorded.

Parameters

mask – **[in]** This parameter is the logging mask to be enforced.

Return values

- **HIPTENSOR_STATUS_SUCCESS** – if the operation completed successfully.
- **HIPTENSOR_STATUS_INVALID_VALUE** – if the given log mask is invalid.

3.30 hiptensorLoggerForceDisable

hiptensorStatus_t **hiptensorLoggerForceDisable**()

Disables logging.

Return values

HIPTENSOR_STATUS_SUCCESS – if the operation completed successfully.

CHAPTER

FOUR

CONTRIBUTING TO HIPTENSOR

This document provides the coding guidelines to be followed while contributing to the hipTensor APIs.

4.1 License agreement

1. The code I am contributing is mine and I have the right to license it.
2. By submitting a pull request for this project I am granting you a license to distribute said code under the MIT License for the project.

4.2 Pull-request guidelines

Our code contribution guidelines closely follows the model of GitHub pull-requests. The hipTensor repository follows a workflow which dictates a /master branch where releases are cut, and a /develop branch which serves as an integration branch for new code. Follow the guidelines below while creating a pull request:

- Target the **develop** branch for integration
- Ensure that the code builds successfully
- Do not break existing test cases
- Be informed that a new functionality is only merged with new unit tests
- Ensure that new unit tests integrate within the existing GoogleTest framework
- Design the tests with good code coverage
- Ensure that the code contains benchmark tests and performance approaches the compute bound limit or memory bound limit

4.3 Style guide

This project follows the CPP Core guidelines, with few modifications and additions as given below. We encourage you to follow the below-mentioned guidelines while creating the pull requests.

4.3.1 Interface

- Use C++17 for the library code
- Avoid camel case
- The above-given rules apply specifically to publicly visible APIs, but is also encouraged (not mandated) for internal code

4.3.2 Philosophy

- Write in ISO Standard C++14 (especially to support Windows, Linux and macOS platforms)
- Prefer compile-time check to run-time check

4.3.3 Implementation

- Use a `.cpp` suffix for code files and an `.hpp` suffix for interface files if your project doesn't already follow another convention
- Ensure that a `.cpp` file mandatorily includes the `.hpp` file(s) that defines its interface
- Don't put a global `using` -directive in a header file
- Use `#include` guards for all `.hpp` files
- Don't use an unnamed (anonymous) `namespace` in a header
- Prefer using `std::array` or `std::vector` instead of a C array
- Minimize the exposure of class members
- Keep functions short and simple
- To return multiple 'out' values, prefer returning a `std::tuple`
- Manage resources automatically using RAII (includes `std::unique_ptr` and `std::shared_ptr`)
- Use `auto` to avoid redundant repetition of type names
- Always initialize an object
- Prefer the `{}` initializer syntax
- Expect your code to run as part of a multi-threaded program
- Avoid global variables

4.3.4 Format

C++ code is formatted using `clang-format`. To run clang-format, use the version in the `/opt/rocm/llvm/bin` directory. Don't use your system's built-in `clang-format`, which could be an older version leading to different results.

To format a file, use:

```
/opt/rocm/llvm/bin/clang-format -style=file -i <path-to-source-file>
```

To format all files, run the following script in the hipTensor directory:

```
#!/bin/bash
git ls-files -z *.cc *.cpp *.h *.hpp *.cl *.h.in *.hpp.in *.cpp.in | xargs -0 /opt/rocm/
↪llvm/bin/clang-format -style=file -i
```

Also, to install githooks to format the code per-commit, use:

```
./.githooks/install
```

4.3.5 Example

1. Create and track a hipTensor fork.
2. Clone your fork:

```
git clone -b develop https://github.com/<your_fork>/hipTensor.git .
.githooks/install
git checkout -b <new_branch>
...
git add <new_work>
git commit -m "What was changed"
git push origin <new_branch>
...
```

3. Create a pull request to ROCmSoftwarePlatform/hipTensor targeting develop branch.
4. Make sure to respond to code reviews.
5. Await CI and approval feedback.
6. Once approved, await for dev team to merge!

Note: Please don't forget to install the githooks via `.githooks/install` as there are triggers for clang formatting in commits.

CHAPTER

FIVE

LICENSE

INDEX

H