# CHAI Documentation

***Release 2.2.1***

**CHAI Developers**

**Sep 02, 2020**

# Basics


**1 Getting Started** **3**
- 1.1 Installation . . . 3
- 1.2 Basic Usage . . . 4

**2 Tutorial** **5**

**3 Configuring CHAI** **7**

**4 Code Documentation** **9**

**5 Contribution Guide** **11**
- 5.1 Forking CHAI . . . 11
- 5.2 Developing a New Feature . . . 11
- 5.3 Developing a Bug Fix . . . 12
- 5.4 Creating a Pull Request . . . 12
- 5.5 Tests . . . 12

**6 Developer Guide** **13**
- 6.1 Generating CHAI host-config files . . . 13

CHAI is a C++ libary providing an array object that can be used transparently in multiple memory spaces. Data is automatically migrated based on copy-construction, allowing for correct data access regardless of location. CHAI can be used standalone, but is best when paired with the RAJA library, which has built-in CHAI integration that takes care of everything.

- If you want to get and install CHAI, take a look at our getting started guide.
- If you are looking for documentation about a particular CHAI function, see the code documentation.
- Want to contribute? Take a look at our developer and contribution guides.

Any questions? Contact chai-dev@llnl.gov

CHAPTER 1

# Getting Started

This page provides information on how to quickly get up and running with CHAI.

## 1.1 Installation

CHAI is hosted on GitHub here. To clone the repo into your local working space, type:

```
$ git clone --recursive git@github.com:LLNL/CHAI.git
```

The `--recursive` argument is required to ensure that the *BLT* submodule is also checked out. BLT is the build system we use for CHAI.

### 1.1.1 Building CHAI

CHAI uses CMake and BLT to handle builds. Make sure that you have a modern compiler loaded and the configuration is as simple as:

```
$ mkdir build && cd build
$ cmake -DCUDA_TOOLKIT_ROOT_DIR=/path/to/cuda ../
```

By default, CHAI will attempt to build with CUDA. CMake will provide output about which compiler is being used, and what version of CUDA was detected. Once CMake has completed, CHAI can be built with Make:

```
$ make
```

For more advanced configuration, see *Configuring CHAI*.

## 1.2 Basic Usage

Let's take a quick tour through CHAI's most important features. A complete listing you can compile is included at the bottom of the page. First, let's create a new ManagedArray object. This is the interface through which you will want to access data:

```
chai::ManagedArray<double> a(100);
```

This creates a ManagedArray storing elements of type double, with 100 elements allocated in the CPU memory.

Next, let's assign some data to this array. We'll use CHAI's forall helper function for this, since it interacts with the ArrayManager for us to ensure the data is in the appropriate ExecutionSpace:

```
forall(sequential(), 0, 100, [=] (int i) {
  a[i] = 3.14 * i;
});
```

CHAI's ArrayManager can copy this array to another ExecutionSpace transparently. Let's use the GPU to double the contents of this array:

```
forall(cuda(), 0, 100, [=] __device__ (int i) {
  a[i] = 2.0 * a[i];
});
```

We can access the array again on the CPU, and the ArrayManager will handle copying the modified data back:

```
forall(sequential(), 0, 100, [=] (int i) {
  std::cout << "a[" << i << "] = " << a[i] << std::endl;
});
```

CHAPTER 2

# Tutorial

The file `src/examples/example.cpp` contains a brief program that shows how CHAI can be used. Let's walk through this example, line-by-line:

# CHAPTER 3

# Configuring CHAI

In addition to the normal options provided by CMake, CHAI uses some additional configuration arguments to control optional features and behavior. Each argument is a boolean option, and can be turned on or off:

-DENABLE_CUDA=Off

Here is a summary of the configuration options, their default value, and meaning:

| Variable | Default | Meaning |
|---|---|---|
| ENABLE_CUDA | Off | Enable CUDA support. |
| ENABLE_HIP | Off | Enable HIP support. |
| ENABLE_GPU_SIMULATION_MODE | Off | Simulates GPU execution. |
| ENABLE_UM | Off | Enable support for CUDA Unified Memory. |
| ENABLE_IMPLICIT_CONVERSIONS | On | Enable implicit conversions between ManagedArray and raw pointers |
| DISABLE_RM | Off | Disable the ArrayManager and make ManagedArray a thin wrapper around a pointer. |
| ENABLE_TESTS | On | Build test executables. |
| ENABLE_BENCHMARKS | On | Build benchmark programs. |

These arguments are explained in more detail below:

- ENABLE_CUDA This option enables support for GPUs using CUDA. If CHAI is built without CUDA, HIP, or GPU_SIMULATION_MODE support, then only the `CPU` execution space is available for use.
- ENABLE_HIP This option enables support for GPUs using HIP. If CHAI is built without CUDA, HIP, or GPU_SIMULATION_MODE support, then only the `CPU` execution space is available for use.
- ENABLE_GPU_SIMULATION_MODE This option simulates GPU support by enabling the GPU execution space, backed by a HOST umpire allocator. If CHAI is built without CUDA, HIP, or GPU_SIMULATION_MODE support, then only the `CPU` execution space is available for use.

- ENABLE_UM This option enables support for Unified Memory as an optional execution space. When a `ManagedArray` is allocated in the `UM` space, CHAI will not manually copy data. Data movement in this case is handled by the CUDA driver and runtime.
- ENABLE_IMPLICIT_CONVERSIONS This option will allow implicit casting between an object of type `ManagedArray<T>` and the correpsonding raw pointer type `T*`. This option is disabled by default, and should be used with caution.
- DISABLE_RM This option will remove all usage of the `ArrayManager` class and let the `ManagedArray` objects function as thin wrappers around a raw pointer. This option can be used with CPU-only allocations, or with CUDA Unified Memory.
- ENABLE_TESTS This option controls whether or not test executables will be built.
- ENABLE_BENCHMARKS This option will build the benchmark programs used to test `ManagedArray` performance.

CHAPTER 4

# Code Documentation

CHAPTER 5

# Contribution Guide

This document is intented for developers who want to add new features or bugfixes to CHAI. It assumes you have some familiarity with git and GitHub. It will discuss what a good pull request (PR) looks like, and the tests that your PR must pass before it can be merged into CHAI.

## 5.1 Forking CHAI

If you aren't a CHAI deveolper at LLNL, then you won't have permission to push new branches to the repository. First, you should create a fork. This will create a copy of the CHAI repository that you own, and will ensure you can push your changes up to GitHub and create pull requests.

## 5.2 Developing a New Feature

New features should be based on the `develop` branch. When you want to create a new feature, first ensure you have an up-to-date copy of the `develop` branch:

```
$ git checkout develop
$ git pull origin develop
```

You can now create a new branch to develop your feature on:

```
$ git checkout -b feature/<name-of-feature>
```

Proceed to develop your feature on this branch, and add tests that will exercise your new code. If you are creating new methods or classes, please add Doxygen documentation.

Once your feature is complete and your tests are passing, you can push your branch to GitHub and create a PR.

## 5.3 Developing a Bug Fix

First, check if the change you want to make has been fixed in `develop`. If so, we suggest you either start using the `develop` branch, or temporarily apply the fix to whichever version of CHAI you are using.

Assuming there is an unsolved bug, first make sure you have an up-to-date copy of the develop branch:

```
$ git checkout develop
$ git pull origin develop
```

Then create a new branch for your bugfix:

```
$ git checkout -b bugfix/<name-of-bug>
```

First, add a test that reproduces the bug you have found. Then develop your bugfix as normal, and ensure to `make test` to check your changes actually fix the bug.

Once you are finished, you can push your branch to GitHub, then create a PR.

## 5.4 Creating a Pull Request

You can create a new PR here. GitHub has a good guide to PR basics if you want some more information. Ensure that your PR base is the `develop` branch of CHAI.

Add a descriptive title explaining the bug you fixed or the feature you have added, and put a longer description of the changes you have made in the comment box.

Once your PR has been created, it will be run through our automated tests and also be reviewed by RAJA team members. Providing the branch passes both the tests and reviews, it will be merged into RAJA.

## 5.5 Tests

CHAI's tests are all in the `test` directory, and your PR must pass all these tests before it is merged. If you are adding a new feature, add new tests.

CHAPTER 6

# Developer Guide

## 6.1 Generating CHAI host-config files

This mechanism will generate a cmake configuration file that reproduces the configuration *Spack <https://github.com/spack/spack>* would have generated in the same context. It will contain all the information necessary to build CHAI with the described toolchain.

In particular, the host config file will setup: * flags corresponding with the target required (Release, Debug). * compilers path, and other toolkits (cuda if required), etc. * paths to installed dependencies.

This provides an easy way to build CHAI based on *Spack <https://github.com/spack/spack>* itself driven by Uberenv.

### 6.1.1 Uberenv role

Uberenv helps by doing the following:

- Pulls a blessed version of Spack locally
- If you are on a known operating system (like TOSS3), we have defined compilers and system packages so you don't have to rebuild the world (CMake typically in CHAI).
- Overrides CHAI Spack packages with the local one if it exists. (see `scripts/uberenv/packages`).
- Covers both dependencies and project build in one command.

Uberenv will create a directory `uberenv_libs` containing a Spack instance with the required CHAI dependencies installed. It then generates a host-config file (`<config_dependent_name>.cmake`) at the root of CHAI repository.

### 6.1.2 Using Uberenv to generate the host-config file

```
$ python scripts/uberenv/uberenv.py
```

**Note:** On LC machines, it is good practice to do the build step in parallel on a compute node. Here is an example command: `srun -ppdebug -N1 --exclusive python scripts/uberenv/uberenv.py`

Unless otherwise specified Spack will default to a compiler. It is recommended to specify which compiler to use: add the compiler spec to the `--spec` Uberenv command line option.

On blessed systems, compiler specs can be found in the Spack compiler files in our repository: `scripts/uberenv/spack_configs/<System type>/compilers.yaml`.

Some examples uberenv options:

- `--spec=%clang@4.0.0`
- `--spec=%clang@4.0.0+cuda`
- `--prefix=<Path to uberenv build directory (defaults to ./uberenv_libs)>`

It is also possible to use the CI script outside of CI:

```
$ SPEC="%clang@9.0.0 +cuda" scripts/gitlab/build_and_test.sh --deps-only
```

Building dependencies can take a long time. If you already have a Spack instance you would like to reuse (in supplement of the local one managed by Uberenv), you can do so changing the uberenv command as follow:

```
$ python scripts/uberenv/uberenv.py --upstream=<path_to_my_spack>/opt/spack
```

### 6.1.3 Using host-config files to build CHAI

When a host-config file exists for the desired machine and toolchain, it can easily be used in the CMake build process:

```
$ mkdir build && cd build
$ cmake -C  <path_to>/lassen-blueos_3_ppc64le_ib_p9-clang@9.0.0.cmake ..
$ cmake --build -j .
$ ctest --output-on-failure -T test
```

It is also possible to use the CI script outside of CI:

```
$ HOST_CONFIG=<path_to>/<host-config>.cmake scripts/gitlab/build_and_test.sh
```

### 6.1.4 Testing new dependencies versions

CHAI depends on Umpire, and optionally RAJA. Testing with newer versions of both is made straightforward with Uberenv and Spack:

- `$ python scripts/uberenv/uberenv.py --spec=%clang@9.0.0 ^umpire@develop`
- `$ python scripts/uberenv/uberenv.py --spec=%clang@9.0.0+raja ^raja@develop`

Those commands will install respectively *umpire@develop* and *raja@develop* locally, and generate host-config files with the corresponding paths.

Again, the CI script can be used directly to install, build and test in one command:

```
$ SPEC="%clang@9.0.0 ^umpire@develop" scripts/gitlab/build_and_test.sh
```