

---

# CHAI Documentation

*Release 2.2.2*

**CHAI Developers**

**Oct 12, 2020**



<b>1</b>	<b>Getting Started</b>	<b>3</b>
1.1	Installation . . . . .	3
1.2	Basic Usage . . . . .	4
<b>2</b>	<b>Tutorial</b>	<b>5</b>
<b>3</b>	<b>User Guide</b>	<b>7</b>
3.1	A Portable Pattern for Polymorphism . . . . .	7
<b>4</b>	<b>Configuring CHAI</b>	<b>13</b>
<b>5</b>	<b>Code Documentation</b>	<b>15</b>
<b>6</b>	<b>Contribution Guide</b>	<b>17</b>
6.1	Forking CHAI . . . . .	17
6.2	Developing a New Feature . . . . .	17
6.3	Developing a Bug Fix . . . . .	18
6.4	Creating a Pull Request . . . . .	18
6.5	Tests . . . . .	18
<b>7</b>	<b>Developer Guide</b>	<b>19</b>
7.1	Generating CHAI host-config files . . . . .	19



CHAI is a C++ library 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](mailto:chai-dev@llnl.gov)



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:



### 3.1 A Portable Pattern for Polymorphism

CHAI provides a data structure to help handle cases where it is desirable to call virtual functions on the device. If you only call virtual functions on the host, this pattern is unnecessary. But for those who do want to use virtual functions on the device without a painstaking amount of refactoring, we begin with a short, albeit admittedly contrived example.

```
class MyBaseClass {
public:
    MyBaseClass() {}
    virtual ~MyBaseClass() {}
    virtual int getValue() const = 0;
};

class MyDerivedClass : public MyBaseClass {
public:
    MyDerivedClass(int value) : MyBaseClass(), m_value(value) {}
    ~MyDerivedClass() {}
    int getValue() const { return m_value; }

private:
    int m_value;
};

int main(int argc, char** argv) {
    MyBaseClass* myBaseClass = new MyDerivedClass(0);
    myBaseClass->getValue();
    delete myBaseClass;
    return 0;
}
```

It is perfectly fine to call `myBaseClass->getValue()` in host code, since `myBaseClass` was created on the host. However, what if you want to call this virtual function on the device?

```
__global__ void callVirtualFunction(MyBaseClass* myBaseClass) {
    myBaseClass->getValue();
}

int main(int argc, char** argv) {
    MyBaseClass* myBaseClass = new MyDerivedClass(0);
    callVirtualFunction<<<1, 1>>>(myBaseClass);
    delete myBaseClass;
    return 0;
}
```

At best, calling this code will result in a crash. At worst, it will access garbage and happily continue while giving incorrect results. It is illegal to access host pointers on the device and produces undefined behavior. So what is our next attempt? Why not pass the argument by value rather than by a pointer?

```
__global__ void callVirtualFunction(MyBaseClass myBaseClass) {
    myBaseClass.getValue();
}

int main(int argc, char** argv) {
    MyBaseClass* myBaseClass = new MyDerivedClass(0);
    callVirtualFunction<<<1, 1>>>(*myBaseClass); // This will not compile
    delete myBaseClass;
    return 0;
}
```

At first glance, this may seem like it would work, but this is not supported by nvidia: “It is not allowed to pass as an argument to a `__global__` function an object of a class with virtual functions” (<https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#virtual-functions>). Also: “It is not allowed to pass as an argument to a `__global__` function an object of a class derived from virtual base classes” (<https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#virtual-base-classes>). You could refactor to use the curiously recurring template pattern, but that would likely require a large development effort and also limits the programming patterns you can use. Also, there is a limitation on the size of the arguments passed to a global kernel, so if you have a very large class this is simply impossible. So we make another attempt.

```
__global__ void callVirtualFunction(MyBaseClass* myBaseClass) {
    myBaseClass->getValue();
}

int main(int argc, char** argv) {
    MyBaseClass* myBaseClass = new MyDerivedClass(0);
    MyBaseClass* d_myBaseClass;
    cudaMalloc(&d_myBaseClass, sizeof(MyBaseClass));
    cudaMemcpy(d_myBaseClass, myBaseClass, sizeof(MyBaseClass),
    ↪ cudaMemcpyHostToDevice);

    callVirtualFunction<<<1, 1>>>(d_myBaseClass);

    cudaFree(d_myBaseClass);
    delete myBaseClass;

    return 0;
}
```

We are getting nearer, but there is still a flaw. The bits of `myBaseClass` contain the virtual function table that allows virtual function lookups on the host, but that virtual function table is not valid for lookups on the device since it contains pointers to host functions. It will not work any better to cast to `MyDerivedClass` and copy the bits. The only

option is to call the constructor on the device and keep that device pointer around.

```
__global__ void make_on_device(MyBaseClass** myBaseClass, int argument) {
    *myBaseClass = new MyDerivedClass(argument);
}

__global__ void destroy_on_device(MyBaseClass* myBaseClass) {
    delete myBaseClass;
}

__global__ void callVirtualFunction(MyBaseClass* myBaseClass) {
    myBaseClass->getValue();
}

int main(int argc, char** argv) {
    MyBaseClass** d_temp;
    cudaMalloc(&d_temp, sizeof(MyBaseClass*));
    make_on_device<<<1, 1>>>>(d_temp, 0);

    MyBaseClass** temp = (MyBaseClass**) malloc(sizeof(MyBaseClass*));
    cudaMemcpy(temp, d_temp, sizeof(MyBaseClass*), cudaMemcpyDeviceToHost);
    MyBaseClass d_myBaseClass = *temp;

    callVirtualFunction<<<1, 1>>>>(d_myBaseClass);

    free(temp);
    destroy_on_device<<<1, 1>>>>(d_myBaseClass);
    cudaFree(d_temp);

    return 0;
}
```

OK, this is finally correct, but super tedious. So we took care of all the boilerplate and underlying details for you. The final result is at least recognizable when compared to the original code. The added benefit is that you can use a `chai::managed_ptr` on the host AND the device.

```
__global__ void callVirtualFunction(chai::managed_ptr<MyBaseClass> myBaseClass) {
    myBaseClass->getValue();
}

int main(int argc, char** argv) {
    chai::managed_ptr<MyBaseClass> myBaseClass = chai::make_managed<MyDerivedClass>(0);
    myBaseClass->getValue(); // Accessible on the host
    callVirtualFunction<<<1, 1>>>>(myBaseClass); // Accessible on the device
    myBaseClass.free();
    return 0;
}
```

OK, so we didn't do all the work for you, but we definitely gave you a leg up. What's left for you to do? You just need to make sure the functions accessed on the device have the `__device__` specifier (including constructors and destructors). We use the `CHAI_HOST_DEVICE` macro in this example, which actually annotates the functions as `__host__ __device__` so we can call the virtual method on both the host and the device. You also need to make sure the destructors of all base classes are virtual so the object gets cleaned up properly on the device.

```
class MyBaseClass {
public:
    CARE_HOST_DEVICE MyBaseClass() {}
}
```

(continues on next page)

(continued from previous page)

```

    CARE_HOST_DEVICE virtual ~MyBaseClass() {}
    CARE_HOST_DEVICE virtual int getValue() const = 0;
};

class MyDerivedClass : public MyBaseClass {
public:
    CARE_HOST_DEVICE MyDerivedClass(int value) : MyBaseClass(), m_value(value) {}
    CARE_HOST_DEVICE ~MyDerivedClass() {}
    CARE_HOST_DEVICE int getValue() const { return m_value; }

private:
    int m_value;
};

```

Now you may rightfully ask, what happens when this class contains raw pointers? There is a convenient solution for this case and we demonstrate with a more interesting example.

```

class MyBaseClass {
public:
    CARE_HOST_DEVICE MyBaseClass() {}
    CARE_HOST_DEVICE virtual ~MyBaseClass() {}
    CARE_HOST_DEVICE virtual int getScalarValue() const = 0;
    CARE_HOST_DEVICE virtual int getArrayValue(int index) const = 0;
};

class MyDerivedClass : public MyBaseClass {
public:
    CARE_HOST_DEVICE MyDerivedClass(int scalarValue, int* arrayValue)
        : MyBaseClass(), m_scalarValue(scalarValue), m_arrayValue(arrayValue) {}
    CARE_HOST_DEVICE ~MyDerivedClass() {}
    CARE_HOST_DEVICE int getScalarValue() const { return m_scalarValue; }
    CARE_HOST_DEVICE int getArrayValue() const { return m_arrayValue; }

private:
    int m_scalarValue;
    int* m_arrayValue;
};

__global__ void callVirtualFunction(chai::managed_ptr<MyBaseClass> myBaseClass) {
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    myBaseClass->getScalarValue();
    myBaseClass->getArrayValue(i);
}

int main(int argc, char** argv) {
    chai::ManagedArray<int> arrayValue(10);
    chai::managed_ptr<MyBaseClass> myBaseClass
        = chai::make_managed<MyDerivedClass>(0, chai::unpack(arrayValue));
    callVirtualFunction<<<1, 10>>>(myBaseClass);
    myBaseClass.free();
    arrayValue.free();
    return 0;
}

```

The respective host and device pointers contained in the *chai::ManagedArray* can be extracted and passed to the host and device instance of *MyDerivedClass* using *chai::unpack*. Of course, if you never dereference *m\_arrayValue* on the device, you could simply pass a raw pointer to *chai::make\_managed*. If the class contains a *chai::ManagedArray*, a

*chai::ManagedArray* can simply be passed to the constructor. The same rules apply for passing a *chai::managed\_ptr*, calling *chai::unpack* on a *chai::managed\_ptr*, or passing a raw pointer and not accessing it on the device.

More complicated rules apply for keeping the data in sync between the host and device instances of an object, but it is possible to do so to a limited extent. It is also possible to control the lifetimes of objects passed to *chai::make\_managed*.





## 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	De- fault	Meaning
<code>ENABLE_CUDA</code>	Off	Enable CUDA support.
<code>ENABLE_HIP</code>	Off	Enable HIP support.
<code>ENABLE_GPU_SIMULATION_MODE</code>	Off	Simulates GPU execution.
<code>ENABLE_UM</code>	Off	Enable support for CUDA Unified Memory.
<code>ENABLE_IMPLICIT_CONVERSIONS</code>	On	Enable implicit conversions between <code>ManagedArray</code> and raw pointers
<code>DISABLE_RM</code>	Off	Disable the <code>ArrayManager</code> and make <code>ManagedArray</code> a thin wrapper around a pointer.
<code>ENABLE_TESTS</code>	On	Build test executables.
<code>ENABLE_BENCHMARKS</code>	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 corresponding 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 5

---

### Code Documentation

---



This document is intended 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.

### 6.1 Forking CHAI

If you aren't a CHAI developer 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.

### 6.2 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.

## 6.3 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.

## 6.4 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.

## 6.5 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.

## 7.1 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**.

### 7.1.1 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.

### 7.1.2 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: `srn -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
```

### 7.1.3 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
```

### 7.1.4 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
```