LvArray Documentation

Randolph Settgast

Sep 06, 2023

Contents

LvArray is a collection of container classes designed for performance portability in that they are usable on the host and device and provide performance similar to direct pointer manipulation. It consists of six main template classes:

- `LvArray::Array`: A multidimensional array.
- `LvArray::SortedArray`: A sorted unique collection of values stored in a contiguous allocation.
- `LvArray::ArrayOfArrays`: Provides functionality similar to `std::vector< std::vector< T > >` except all the values are stored in one allocation.
- `LvArray::ArrayOfSets`: Provides functionality similar to `std::vector< std::set< T > >` except all the values are stored in one allocation.
- `LvArray::SparsityPattern`: A compressed row storage sparsity pattern, or equivalently a boolean CRS matrix.
- `LvArray::CRSMatrix`: A compressed row storage matrix.

These template classes all take three common template arguments:

- The type of the value stored in the container.
- The integral type used for indexing calculations, the recommended type is `std::ptrdiff_t`.
- The buffer type to use for allocation and de-allocation.

Like the standard template library containers the destructors destroy all the values in the container and release any acquired resources. Similarly the copy constructors and copy assignment operators perform a deep copy of the source container. In general the move constructors and move assignment operators perform a shallow copy and invalidate the source container.

The `LvArray` containers were designed specifically to integrate with RAJA although they would also work with other kernel launching abstractions or native device code. In RAJA, variables are passed to device kernels by lambda capture. When using CUDA a `__device__` or `__host__ __device__` lambda can only capture variables by value, which means the lambda's closure object contains a copy of each captured variable created by copy construction. As mentioned above the copy constructor for the `LvArray` containers always perform a deep copy. So when a container gets captured by value in a lambda any modifications to it or the values it contains won't be reflected in the source container. Even if this works for your use case it is not optimal because it requires a new allocation each time the container gets captured and each time the closure object gets copied. To remedy this each `LvArray` container has an associated view class. Unlike the containers the views do not own their resources and in general have shallow copy semantics. This makes them suitable to be captured by value in a lambda. Furthermore almost every view method is `const` which means the lambda does not need to be made `mutable`.

Note: Not all buffer types support shallow copying. When using a container with such a buffer the move constructor and move assignment operator will create a deep copy of the source. Similarly the view copy constructor, copy assignment operator, move constructor and move assignment operator will all be deep copies.

CHAPTER 1

Build Guide

LvArray uses a CMake based build system augmented with BLT. If you're not familiar with CMake, RAJA has a good introduction. LvArray has a dependency on RAJA and as such requires that the `RAJA_DIR` CMake variable defined and points to a RAJA installation.

1.1 CMake Options

In addition to the standard CMake and BLT options LvArray supports the following options.

- **Adding additional targets** The following variables add additional build targets but do not alter the usage or functionality of LvArray.
  - **`ENABLE_TESTS`** [default `ON`] Build the unit tests which can be run with `make test`. The unit tests take a long time to build with the IMB XL and Intel compilers.
  - **`ENABLE_EXAMPLES`** [default `ON`] Build the examples, `ENABLE_TESTS` must also be `ON`.
  - **`ENABLE_BENCHMARKS`** [default `ON`] Build the benchmarks, `ENABLE_TESTS` must also be `ON`.
  - **`DISABLE_UNIT_TESTS`** [default `OFF`] Use with `ENABLE_TESTS=ON` to disable building the unit tests but still allow the examples and benchmarks.
  - **`ENABLE_DOCS`** [default `ON`] Build the documentation.
- **Third party libraries** LvArray has a hard dependency on RAJA along with multiple optional dependencies. Of the following variables only `RAJA_DIR` is mandatory.
  - **`RAJA_DIR`** The path to the RAJA installation.
  - **`ENABLE_UMPIRE`** [default `OFF`] If Umpire is enabled. Currently no part of LvArray uses Umpire but it is required when using CHAI.
  - **`UMPIRE_DIR`** The path to the Umpire installation, must be specified when Umpire is enabled.
  - **`ENABLE_CHAI`** [default `OFF`] If CHAI is enabled, CHAI also requires Umpire. Enabling CHAI allows the usage of the `LvArray::ChaiBuffer`.

  - **CHAI_DIR** The path to the CHAI installation, must be specified when CHAI is enabled.
  - **ENABLE_CALIPER** [default `OFF`] If caliper is enabled. Currently caliper is only used to time the benchmarks.
  - **CALIPER_DIR** The path to the caliper installation, must be specified when caliper is enabled.
  - **ENABLE_ADDR2LINE** [default `ON`] If `addr2line` is enabled. This is used in `LvArray::system::stackTrace` to attempt to provide file and line locations for the stack frames.
  - **ADDR2LINE_EXEC** [default `/usr/bin/addr2line`] The path to the `addr2line` executable.

- **Debug options** The following options don't change the usage of LvArray but they are intended to make debugging easier.
  - **LVARRAY_BOUNDS_CHECK** [default `ON` iff in a CMake `Debug` build.] Enables bounds checks on container access along with checks for other invalid operations.
  - **ENABLE_TOTALVIEW_OUTPUT** [default `OFF`] Makes it easier to inspect the `LvArray::Array` in TotalView. This functionality is highly dependent on the version of TotalView used.

1.2 Using LvArray Your Application

Once LvArray has been installed if your application uses CMake importing LvArray is as simple as defining `LVARRAY_DIR` as the path to LvArray install directory and then adding `find_package(LVARRAY)`. This will export a `lvarray` target that can then be used by `target_link_libraries` and the like.

1.3 Host Configs

Host config files are a convenient way to group CMake options for a specific configuration together. There are a set of example host configs in the `host-configs` directory. Once you've created a host config file you can use `scripts/config-build.py` to create the build directory and run CMake for you. An example usage would be `python ./scripts/config-build.py --hc host-configs/LLNL/quartz-clang@10.0.0.cmake`.

```
> python scripts/config-build.py --help
usage: config-build.py [-h] [-bp BUILDPATH] [-ip INSTALLPATH]
                       [-bt {Release,Debug,RelWithDebInfo,MinSizeRel}] [-e]
                       [-x] [-ecc] -hc HOSTCONFIG

Configure cmake build. Unrecognized arguments are passed on to CMake.

optional arguments:
  -h, --help            show this help message and exit
  -bp BUILDPATH, --buildpath BUILDPATH
                        specify path for build directory. If not specified,
                        will create in current directory.
  -ip INSTALLPATH, --installpath INSTALLPATH
                        specify path for installation directory. If not
                        specified, will create in current directory.
  -bt {Release,Debug,RelWithDebInfo,MinSizeRel}, --buildtype {Release,Debug,
↪RelWithDebInfo,MinSizeRel}
                        build type.
  -e, --eclipse         create an eclipse project file.
  -x, --xcode           create an xcode project.
```

(continues on next page)

(continued from previous page)

```
  -ecc, --exportcompilercommands
                        generate a compilation database. Can be used by the
                        clang tools such as clang-modernize. Will create a
                        file called 'compile_commands.json' in build
                        directory.
  -hc HOSTCONFIG, --hostconfig HOSTCONFIG
                        select a specific host-config file to initalize
                        CMake's cache
```

1.4 Submodule usage

LvArray can also be used as a submodule. In this case the configuration is largely the same except that LvArray expects the parent project to have imported the third party libraries. For example if `ENABLE_UMPIRE` is `ON` then LvArray will depend on `umpire` but it will make no attempt to find these library (`UMPIRE_DIR` is unused).

1.5 Spack and Uberenv Builds

LvArray has an associated Spack package. For those unfamiliar with Spack the most important thing to understand is the spec syntax. For those interested the LvArray package implementation is here the important part of which is reproduced below.

```
class Lvarray(CMakePackage, CudaPackage):
    """LvArray portable HPC containers."""

    homepage = "https://github.com/GEOSX/lvarray"
    git      = "https://github.com/GEOSX/LvArray.git"

    version('develop', branch='develop', submodules='True')
    version('tribol', branch='temp/feature/corbett/tribol', submodules='True')

    variant('shared', default=True, description='Build Shared Libs')
    variant('umpire', default=False, description='Build Umpire support')
    variant('chai', default=False, description='Build Chai support')
    variant('caliper', default=False, description='Build Caliper support')
    variant('tests', default=True, description='Build tests')
    variant('benchmarks', default=False, description='Build benchmarks')
    variant('examples', default=False, description='Build examples')
    variant('docs', default=False, description='Build docs')
    variant('addr2line', default=True,
            description='Build support for addr2line.')

    depends_on('cmake@3.8:', type='build')
    depends_on('cmake@3.9:', when='+cuda', type='build')

    depends_on('raja')
    depends_on('raja+cuda', when='+cuda')

    depends_on('umpire', when='+umpire')
    depends_on('umpire+cuda', when='+umpire+cuda')

    depends_on('chai+raja', when='+chai')
```

(continues on next page)

(continued from previous page)

```
    depends_on('chai+raja+cuda', when='+chai+cuda')

    depends_on('caliper', when='+caliper')

    depends_on('doxygen@1.8.13:', when='+docs', type='build')
    depends_on('py-sphinx@1.6.3:', when='+docs', type='build')
```

LvArray also has an `uberenv` based build which simplifies building LvArray's dependencies along with optionally LvArray using spack.

```
> ./scripts/uberenv/uberenv.py --help
Usage: uberenv.py [options]

Options:
  -h, --help            show this help message and exit
  --install             Install `package_name`, not just its dependencies.
  --prefix=PREFIX       destination directory
  --spec=SPEC           spack compiler spec
  --mirror=MIRROR       spack mirror directory
  --create-mirror       Create spack mirror
  --upstream=UPSTREAM   add an external spack instance as upstream
  --spack-config-dir=SPACK_CONFIG_DIR
                        dir with spack settings files (compilers.yaml,
                        packages.yaml, etc)
  --package-name=PACKAGE_NAME
                        override the default package name
  --package-final-phase=PACKAGE_FINAL_PHASE
                        override the default phase after which spack should
                        stop
  --package-source-dir=PACKAGE_SOURCE_DIR
                        override the default source dir spack should use
  --project-json=PROJECT_JSON
                        uberenv project settings json file
  -k                    Ignore SSL Errors
  --pull                Pull if spack repo already exists
  --clean               Force uninstall of packages specified in project.json
  --run_tests           Invoke build tests during spack install
  --macos-sdk-env-setup
                        Set several env vars to select OSX SDK settings.This
                        was necessary for older versions of macOS  but can
                        cause issues with macOS versions >= 10.13.  so it is
                        disabled by default.
```

Two simple examples are provided below.

```
quartz2498 > ./scripts/uberenv/uberenv.py --install --spec="@develop %clang@10.0.1"
```

This will build RAJA (LvArray's only hard dependency) and LvArray and install them in `./uberenv_libs/linux-rhel7-ppc64le-clang@10.0.1`. By default libraries are built in the `RelWithDebInfo` CMake configuration.

```
quartz2498 > ./scripts/uberenv/uberenv.py --spec="@develop %gcc@8.3.1 ^raja@0.12.1␣
↪build_type=Release"
```

This will install RAJA in the same location but it will be built in the `Release` configuration and instead of building and installing LvArray a host-config will be generated and placed in the current directory. This can be useful for developing or debugging.

Currently `uberenv` only works on the LLNL `toss_3_x86_64_ib` and `blueos_3_ppc64le_ib_p9` systems. Further more only certain compilers are supported. On the TOSS systems `clang@10.0.1`, `gcc@8.3.1` and `intel@19.1.2` are supported. On BlueOS `clang-upstream-2019.08.15 (clang@9.0.0)`, `clang-ibm-10.0.1-gcc-8.3.1 (clang@10.0.1)`, `gcc@8.3.1` and `xl-2020.09.17-cuda-11.0.2 (xl@16.1.1)` are supported. Adding support for more compilers is as simple as adding them to the appropriate `compilers.yaml` file.

Adding support for a new system is easy too, you just need to create a directory with a `compilers.yaml` which specifies the available compilers and a `packages.yaml` for system packages and then pass this directory to uberenv with the `--spack-config-dir` option.

For reference two more complicated specs are shown below

```
lassen709 > ./scripts/uberenv/uberenv.py --install --run_tests --spec=
↪"@develop+umpire+chai+caliper+cuda %clang@10.0.1 cuda_arch=70 ^cuda@11.0.2 ^raja@0.
↪12.1~examples~exercises cuda_arch=70 ^umpire@4.0.1~examples cuda_arch=70 ^
↪chai@master~benchmarks~examples cuda_arch=70 ^caliper@2.4~adiak~mpi~dyninst~
↪callpath~papi~libpfm~gotcha~sampler~sosflow"
```

This will use `clang@10.0.1` and `cuda@11.0.2` to build and install RAJA v0.12.1 without examples or exercises, Umpire v4.0.1 without examples, the master branch of CHAI without benchmarks or examples, and caliper v2.4 without a bunch of options. Finally it will build and install LvArray after running the unit tests and verifying that they pass. Note that each package that depends on cuda gets the `cuda_arch=70` variable.

```
quartz2498 > ./scripts/uberenv/uberenv.py --spec="@tribol+umpire %intel@19.1.2 ^
↪raja@0.12.1 build_type=Release ^umpire@4.0.1 build_type=Release"
```

This will use `intel@19.1.2` to build and install RAJA V0.12.1 in release and Umpire v4.0.1 in release. Finally it will generate a host config that can be used to build LvArray.

CHAPTER 2

Buffer Classes

The buffer classes are the backbone of every LvArray class. A buffer class is responsible for allocating, reallocating and de-allocating a chunk of memory as well as moving it between memory spaces. A buffer is not responsible for managing the lifetime of the objects in their allocation. In general buffer classes have shallow copy semantics and do not de-allocate their allocations upon destruction. Buffer classes implement the copy and move constructors as well as the copy and move assignment operators. They also have a default constructor that leaves them in an uninitialized state. In general it is only safe to assign to an uninitialized buffer although different buffer implementations may allow other operations. To construct an initialized buffer pass a dummy boolean argument, this value of the parameter is not important and it only exists to differentiate it from the default constructor. Once created an initialized buffer must be free'd, either directly or though one of its copies. There are currently three buffer implementations: `LvArray::MallocBuffer`, `LvArray::ChaiBuffer` and `LvArray::StackBuffer`.

2.1 `LvArray::MallocBuffer`

As you might have guessed `LvArray::MallocBuffer` uses `malloc` and `free` to handle its allocation. Copying a `LvArray::MallocBuffer` does not copy the allocation. The allocation of a `LvArray::MallocBuffer` lives exclusively on the host and as such it will abort the program if you try to move it to or touch it in any space other than `MemorySpace::host`.

```
TEST( MallocBuffer, copy )
{
  constexpr std::ptrdiff_t size = 55;
  LvArray::MallocBuffer< int > buffer( true );
  buffer.reallocate( 0, LvArray::MemorySpace::host, size );

  for( int i = 0; i < size; ++i )
  {
    buffer[ i ] = i;
  }

  for( int i = 0; i < size; ++i )
  {
```

(continues on next page)

(continued from previous page)

```
    EXPECT_EQ( buffer[ i ], i );
  }

  // MallocBuffer has shallow copy semantics.
  LvArray::MallocBuffer< int > copy = buffer;
  EXPECT_EQ( copy.data(), buffer.data() );

  // Must be manually free'd.
  buffer.free();
}

TEST( MallocBuffer, nonPOD )
{
  constexpr std::ptrdiff_t size = 4;
  LvArray::MallocBuffer< std::string > buffer( true );
  buffer.reallocate( 0, LvArray::MemorySpace::host, size );

  // Buffers don't initialize data so placement new must be used.
  for( int i = 0; i < size; ++i )
  {
    new ( buffer.data() + i ) std::string( std::to_string( i ) );
  }

  for( int i = 0; i < size; ++i )
  {
    EXPECT_EQ( buffer[ i ], std::to_string( i ) );
  }

  // Buffers don't destroy the objects in free.
  // The using statement is needed to explicitly call the destructor
  using std::string;
  for( int i = 0; i < size; ++i )
  {
    buffer[ i ].~string();
  }

  buffer.free();
}
```

[Source: examples/exampleBuffers.cpp]

2.2 LvArray::ChaiBuffer

`LvArray::ChaiBuffer` uses CHAI to manage an allocation which can exist on both the host and device, it functions similarly to the `chai::ManagedArray`. Like the `LvArray::MallocBuffer` copying a `LvArray::ChaiBuffer` via the assignment operators or the move constructor do not copy the allocation. The unique feature of the `LvArray::ChaBuffer` is that when it is copy constructed if the CHAI execution space is set it will move its allocation to the appropriate space creating an allocation there if it did not already exist.

```
CUDA_TEST( ChaiBuffer, captureOnDevice )
{
  constexpr std::ptrdiff_t size = 55;
  LvArray::ChaiBuffer< int > buffer( true );
  buffer.reallocate( 0, LvArray::MemorySpace::host, size );
```

(continues on next page)

(continued from previous page)

```
  for( int i = 0; i < size; ++i )
  {
    buffer[ i ] = i;
  }

  // Capture buffer in a device kernel which creates an allocation on device
  // and copies the data there.
  RAJA::forall< RAJA::cuda_exec< 32 > >(
    RAJA::TypedRangeSegment< std::ptrdiff_t >( 0, size ),
    [buffer] __device__ ( std::ptrdiff_t const i )
  {
    buffer[ i ] += i;
  } );

  // Capture buffer in a host kernel moving the data back to the host allocation.
  RAJA::forall< RAJA::loop_exec >(
    RAJA::TypedRangeSegment< std::ptrdiff_t >( 0, size ),
    [buffer] ( std::ptrdiff_t const i )
  {
    EXPECT_EQ( buffer[ i ], 2 * i );
  } );

  buffer.free();
}
```

[Source: examples/exampleBuffers.cpp]

In order to prevent unnecessary memory motion if the type contained in the `LvArray::ChaiBuffer` is `const` then the data is not touched in any space it is moved to.

```
CUDA_TEST( ChaiBuffer, captureOnDeviceConst )
{
  constexpr std::ptrdiff_t size = 55;
  LvArray::ChaiBuffer< int > buffer( true );
  buffer.reallocate( 0, LvArray::MemorySpace::host, size );

  for( int i = 0; i < size; ++i )
  {
    buffer[ i ] = i;
  }

  // Create a const buffer and capture it in a device kernel which
  // creates an allocation on device and copies the data there.
  LvArray::ChaiBuffer< int const > const constBuffer( buffer );
  RAJA::forall< RAJA::cuda_exec< 32 > >(
    RAJA::TypedRangeSegment< std::ptrdiff_t >( 0, size ),
    [constBuffer] __device__ ( std::ptrdiff_t const i )
  {
    const_cast< int & >( constBuffer[ i ] ) += i;
  } );

  // Capture buffer in a host kernel moving the data back to the host allocation.
  // If constBuffer didn't contain "int const" then this check would fail because
  // the data would be copied back from device.
  RAJA::forall< RAJA::loop_exec >(
    RAJA::TypedRangeSegment< std::ptrdiff_t >( 0, size ),
```

(continues on next page)

(continued from previous page)

```
    [buffer] ( std::ptrdiff_t const i )
  {
    EXPECT_EQ( buffer[ i ], i );
  } );

  buffer.free();
}
```

[Source: examples/exampleBuffers.cpp]

`LvArray::ChaiBuffer` supports explicit movement and touching as well via the methods `move` and `registerTouch`.

Whenever a `LvArray::ChaiBuffer` is moved between memory spaces it will print the size of the allocation, the type of the buffer and the name. Both the name and the type can be set with the `setName` method. If this behavior is not desired it can be disabled with `chai::ArrayManager::getInstance()->disableCallbacks()`.

```
TEST( ChaiBuffer, setName )
{
  LvArray::ChaiBuffer< int > buffer( true );
  buffer.reallocate( 0, LvArray::MemorySpace::host, 1024 );

  // Move to the device.
  buffer.move( LvArray::MemorySpace::cuda, true );

  // Give buffer a name and move back to the host.
  buffer.setName( "my_buffer" );
  buffer.move( LvArray::MemorySpace::host, true );

  // Rename buffer and override the default type.
  buffer.setName< double >( "my_buffer_with_a_nonsensical_type" );
  buffer.move( LvArray::MemorySpace::cuda, true );
}
```

[Source: examples/exampleBuffers.cpp]

Output

```
Moved    4.0 KB to the DEVICE: LvArray::ChaiBuffer<int>
Moved    4.0 KB to the HOST  : LvArray::ChaiBuffer<int> my_buffer
Moved    4.0 KB to the DEVICE: double my_buffer_with_a_nonsensical_type
```

2.3 LvArray::StackBuffer

The `LvArray::StackBuffer` is unique among the buffer classes because it wraps a c-array of objects whose size is fixed at compile time. It is so named because if you declare a `LvArray::StackBuffer` on the stack its allocation will also live on the stack. Unlike the other buffer classes by nature copying a `LvArray::StackBuffer` is a deep copy, furthermore a `LvArray::StackBuffer` can only contain trivially destructible types, so no putting a `std::string` in one. If you try to grow the allocation beyond the fixed size it will abort the program.

```
TEST( StackBuffer, example )
{
  constexpr std::ptrdiff_t size = 55;
  LvArray::StackBuffer< int, 55 > buffer( true );
```

(continues on next page)

(continued from previous page)

```
  static_assert( buffer.capacity() == size, "Capacity is fixed at compile time." );

  for( std::ptrdiff_t i = 0; i < size; ++i )
  {
    buffer[ i ] = i;
  }

  for( std::ptrdiff_t i = 0; i < size; ++i )
  {
    EXPECT_EQ( buffer[ i ], i );
  }

  EXPECT_DEATH_IF_SUPPORTED( buffer.reallocate( size, LvArray::MemorySpace::host, 2 *␣
↪size ), "" );

  // Not necessary with the StackBuffer but it's good practice.
  buffer.free();
}
```

[Source: examples/exampleBuffers.cpp]

2.4 Doxygen

- LvArray::MallocBuffer
- LvArray::ChaiBuffer
- LvArray::StackBuffer

CHAPTER 3

LvArray::Array

The LvArray::Array implements a multidimensional array of values. Unlike Kokkos mdspan or the RAJA View the LvArray::Array owns the allocation it is associated with and supports slicing with operator[] in addition to the standard operator(). Further more a one dimensional LvArray::Array supports operations such as emplace_back and erase with functionality similar to std::vector.

3.1 Template arguments

The LvArray::Array requires five template arguments.

1. T: The type of values stored in the array.
2. NDIM: The number of dimensionality of the array or the number of indices required to access a value.
3. PERMUTATION: A camp::idx_seq which describes the mapping from the multidimensional index space to a linear index. Must be of length NDIM and contain all the values between 0 and NDIM - 1. The way to read a permutation is that the indices go from the slowest on the left to the fastest on the right. Equivalently the left most index has the largest stride whereas the right most index has unit stride.
4. INDEX_TYPE: An integral type used in index calculations, the suggested type is std::ptrdiff_t.
5. BUFFER_TYPE: A template template parameter specifying the buffer type used for allocation and de-allocation, the LvArray::Array contains a BUFFER_TYPE< T >.

Note: LvArray uses the same permutation conventions as the RAJA::View, in fact it is recommended to use the types defined in RAJA/Permutations.hpp.

3.2 Creating and accessing a LvArray::Array

The LvArray::Array has two primary constructors a default constructor which creates an empty array and a constructor that takes the size of each dimension. When using the sized constructor the values of the array are default

initialized.

```
TEST( Array, constructors )
{
  {
    // Create an empty 2D array of integers.
    LvArray::Array< int,
                    2,
                    camp::idx_seq< 0, 1 >,
                    std::ptrdiff_t,
                    LvArray::MallocBuffer > array;
    EXPECT_TRUE( array.empty() );
    EXPECT_EQ( array.size(), 0 );
    EXPECT_EQ( array.size( 0 ), 0 );
    EXPECT_EQ( array.size( 1 ), 0 );
  }

  {
    // Create a 3D array of std::string of size 3 x 4 x 5.
    LvArray::Array< std::string,
                    3,
                    camp::idx_seq< 0, 1, 2 >,
                    std::ptrdiff_t,
                    LvArray::MallocBuffer > array( 3, 4, 5 );
    EXPECT_FALSE( array.empty() );
    EXPECT_EQ( array.size(), 3 * 4 * 5 );
    EXPECT_EQ( array.size( 0 ), 3 );
    EXPECT_EQ( array.size( 1 ), 4 );
    EXPECT_EQ( array.size( 2 ), 5 );

    // The values are default initialized.
    std::string const * const values = array.data();
    for( std::ptrdiff_t i = 0; i < array.size(); ++i )
    {
      EXPECT_EQ( values[ i ], std::string() );
    }
  }
}
```

[Source: examples/exampleArray.cpp]

`LvArray::Array` supports two indexing methods `operator()` which takes all of the indices to a value and `operator[]` which takes a single index at a time and can be chained together.

```
TEST( Array, accessors )
{
  // Create a 2D array of integers.
  LvArray::Array< int,
                  2,
                  camp::idx_seq< 0, 1 >,
                  std::ptrdiff_t,
                  LvArray::MallocBuffer > array( 3, 4 );

  // Access using operator().
  for( std::ptrdiff_t i = 0; i < array.size( 0 ); ++i )
  {
    for( std::ptrdiff_t j = 0; j < array.size( 1 ); ++j )
    {
```

(continues on next page)

(continued from previous page)

```
      array( i, j ) = array.size( 1 ) * i + j;
    }
  }

  // Access using operator[].
  for( std::ptrdiff_t i = 0; i < array.size( 0 ); ++i )
  {
    for( std::ptrdiff_t j = 0; j < array.size( 1 ); ++j )
    {
      EXPECT_EQ( array[ i ][ j ], array.size( 1 ) * i + j );
    }
  }
}
```

[Source: examples/exampleArray.cpp]

The two indexing methods work consistently regardless of how the data is layed out in memory.

```
TEST( Array, permutations )
{
  {
    // Create a 3D array of doubles in the standard layout.
    LvArray::Array< int,
                    3,
                    camp::idx_seq< 0, 1, 2 >,
                    std::ptrdiff_t,
                    LvArray::MallocBuffer > array( 3, 4, 5 );

    // Index 0 has the largest stride while index 2 has unit stride.
    EXPECT_EQ( array.strides()[ 0 ], array.size( 2 ) * array.size( 1 ) );
    EXPECT_EQ( array.strides()[ 1 ], array.size( 2 ) );
    EXPECT_EQ( array.strides()[ 2 ], 1 );

    int const * const pointer = array.data();
    for( std::ptrdiff_t i = 0; i < array.size( 0 ); ++i )
    {
      for( std::ptrdiff_t j = 0; j < array.size( 1 ); ++j )
      {
        for( std::ptrdiff_t k = 0; k < array.size( 2 ); ++k )
        {
          std::ptrdiff_t const offset = array.size( 2 ) * array.size( 1 ) * i +
                                        array.size( 2 ) * j + k;
          EXPECT_EQ( &array( i, j, k ), pointer + offset );
        }
      }
    }
  }

  {
    // Create a 3D array of doubles in a flipped layout.
    LvArray::Array< int,
                    3,
                    camp::idx_seq< 2, 1, 0 >,
                    std::ptrdiff_t,
                    LvArray::MallocBuffer > array( 3, 4, 5 );

    // Index 0 has the unit stride while index 2 has the largest stride.
```

(continues on next page)

(continued from previous page)

```
    EXPECT_EQ( array.strides()[ 0 ], 1 );
    EXPECT_EQ( array.strides()[ 1 ], array.size( 0 ) );
    EXPECT_EQ( array.strides()[ 2 ], array.size( 0 ) * array.size( 1 ) );

    int const * const pointer = array.data();
    for( std::ptrdiff_t i = 0; i < array.size( 0 ); ++i )
    {
      for( std::ptrdiff_t j = 0; j < array.size( 1 ); ++j )
      {
        for( std::ptrdiff_t k = 0; k < array.size( 2 ); ++k )
        {
          std::ptrdiff_t const offset = i + array.size( 0 ) * j +
                                        array.size( 0 ) * array.size( 1 ) * k;
          EXPECT_EQ( &array[ i ][ j ][ k ], pointer + offset );
        }
      }
    }
  }
}
```

[Source: examples/exampleArray.cpp]

3.3 Resizing a `LvArray::Array`

`LvArray::Array` supports a multitude of resizing options. You can resize all of the dimensions at once or specify a set of dimensions to resize. These methods default initialize newly created values and destroy any values no longer in the Array.

```
TEST( Array, resize )
{
  LvArray::Array< int,
                  3,
                  camp::idx_seq< 0, 1, 2 >,
                  std::ptrdiff_t,
                  LvArray::MallocBuffer > array;

  // Resize using a pointer
  std::ptrdiff_t const sizes[ 3 ] = { 2, 5, 6 };
  array.resize( 3, sizes );
  EXPECT_EQ( array.size(), 2 * 5 * 6 );
  EXPECT_EQ( array.size( 0 ), 2 );
  EXPECT_EQ( array.size( 1 ), 5 );
  EXPECT_EQ( array.size( 2 ), 6 );

  // Resizing using a variadic parameter pack.
  array.resize( 3, 4, 2 );
  EXPECT_EQ( array.size(), 3 * 4 * 2 );
  EXPECT_EQ( array.size( 0 ), 3 );
  EXPECT_EQ( array.size( 1 ), 4 );
  EXPECT_EQ( array.size( 2 ), 2 );

  // Resize the second and third dimensions
  array.resizeDimension< 1, 2 >( 3, 6 );
  EXPECT_EQ( array.size(), 3 * 3 * 6 );
```

(continues on next page)

(continued from previous page)

```
  EXPECT_EQ( array.size( 0 ), 3 );
  EXPECT_EQ( array.size( 1 ), 3 );
  EXPECT_EQ( array.size( 2 ), 6 );
}
```

[Source: examples/exampleArray.cpp]

`LvArray::Array` also has a method `resizeWithoutInitializationOrDestruction` that is only enabled if the value type of the array `T` is trivially destructible. This method does not initialize new values or destroy old values and as such it can be much faster for large allocations of trivial types.

It is important to note that unless the array being resized is one dimensional the resize methods above do not preserve the values in the array. That is if you have a two dimensional array `A` of size $M \times N$ and you resize it to $P \times Q$ using any of the methods above then you cannot rely on `A( i, j )` having the same value it did before the resize.

There is also a method `resize` which takes a single parameter and will resize the dimension given by `getSingleParameterResizeIndex`. Unlike the previous methods this will preserve the values in the array. By default the first dimension is resized but you can choose the dimension with `setSingleParameterResizeIndex`.

```
TEST( Array, resizeSingleDimension )
{
  LvArray::Array< int,
                  2,
                  camp::idx_seq< 1, 0 >,
                  std::ptrdiff_t,
                  LvArray::MallocBuffer > array( 5, 6 );

  for( std::ptrdiff_t i = 0; i < array.size( 0 ); ++i )
  {
    for( std::ptrdiff_t j = 0; j < array.size( 1 ); ++j )
    {
      array( i, j ) = 6 * i + j;
    }
  }

  // Grow the first dimension from 5 to 8.
  array.resize( 8 );
  for( std::ptrdiff_t i = 0; i < array.size( 0 ); ++i )
  {
    for( std::ptrdiff_t j = 0; j < array.size( 1 ); ++j )
    {
      if( i < 5 )
      {
        EXPECT_EQ( array( i, j ), 6 * i + j );
      }
      else
      {
        EXPECT_EQ( array( i, j ), 0 );
      }
    }
  }

  // Shrink the second dimension from 6 to 3;
  array.setSingleParameterResizeIndex( 1 );
  array.resize( 3 );
  for( std::ptrdiff_t i = 0; i < array.size( 0 ); ++i )
```

(continues on next page)

(continued from previous page)

```
  {
    for( std::ptrdiff_t j = 0; j < array.size( 1 ); ++j )
    {
      if( i < 5 )
      {
        EXPECT_EQ( array( i, j ), 6 * i + j );
      }
      else
      {
        EXPECT_EQ( array( i, j ), 0 );
      }
    }
  }
}
```

[Source: examples/exampleArray.cpp]

The single dimension resize should only be used when it is necessary to preserve the values as it is a much more complicated operation than the multi-dimension resize methods.

3.4 The one dimensional `LvArray::Array`

The one dimensional `LvArray::Array` supports a couple methods that are not available to multidimensional arrays. These methods are `emplace_back`, `emplace`, `insert`, `pop_back` and `erase`. They all behave exactly like their `std::vector` counter part, the only difference being that `emplace`, `insert` and `erase` take an integer specifying the position to perform the operation instead of an iterator.

3.5 Lambda capture and `LvArray::ArrayView`

`LvArray::ArrayView` is the parent class and the view class of `LvArray::Array`. It shares the same template parameters as `LvArray::Array` except that the `PERMUTATION` type is replaced by an integer corresponding to the unit stride dimension (the last entry in the permutation).

There are multiple ways to create an `LvArray::ArrayView` from an `LvArray::Array`. Because it is the parent class you can just create a reference to a `LvArray::ArrayView` from an existing `LvArray::Array` or by using the method `toView`. `LvArray::Array` also has a user defined conversion operator to a `LvArray::ArrayView` with a const value type or you can use the `toViewConst` method.

The `LvArray::ArrayView` has a copy constructor, move constructor, copy assignment operator and move assignment operator all of which perform the same operation on the `BUFFER_TYPE`. So `ArrayView( ArrayView const & )` calls `BUFFER_TYPE< T >( BUFFER_TYPE< T > const & )` and `ArrayView::operator=( ArrayView && )` calls `BUFFER_TYPE< T >::operator=( BUFFER_TYPE< T > && )`. With the exception of the `StackBuffer` all of these operations are shallow copies.

```
TEST( Array, arrayView )
{
  LvArray::Array< int,
                  2,
                  camp::idx_seq< 1, 0 >,
                  std::ptrdiff_t,
                  LvArray::MallocBuffer > array( 5, 6 );
```

(continues on next page)

(continued from previous page)

```
  // Create a view.
  LvArray::ArrayView< int,
                      2,
                      0,
                      std::ptrdiff_t,
                      LvArray::MallocBuffer > const view = array;
  EXPECT_EQ( view.data(), array.data() );

  // Create a view with const values.
  LvArray::ArrayView< int const,
                      2,
                      0,
                      std::ptrdiff_t,
                      LvArray::MallocBuffer > const viewConst = array.toViewConst();
  EXPECT_EQ( viewConst.data(), array.data() );

  // Copy a view.
  LvArray::ArrayView< int,
                      2,
                      0,
                      std::ptrdiff_t,
                      LvArray::MallocBuffer > const viewCopy = view;
  EXPECT_EQ( viewCopy.data(), array.data() );
}
```

[Source: examples/exampleArray.cpp]

An `LvArray::ArrayView` is almost always constructed from an existing `LvArray::Array` but it does has a default constructor which constructs an uninitialized `ArrayView`. The only valid use of an uninitialized `LvArray::ArrayView` is to assign to it from an existing `LvArray::ArrayView`. This behavior is primarily used when putting an `LvArray::ArrayView` into a container.

`LvArray::ArrayView` supports the subset of operations on `LvArray::Array` that do not require reallocation or resizing. Every method in `LvArray::ArrayView` is `const` except the assignment operators. Most methods of are callable on device.

3.6 ArraySlice

Up until now all the examples using `operator[]` have consumed all of the available indices, i.e. a two dimensional array has always had two immediate applications of `operator[]` to access a value. But what is the result of a single application of `operator[]` to a two dimensional array? Well it's a one dimensional `LvArray::ArraySlice`! `LvArray::ArraySlice` shares the same template parameters as `LvArray::ArrayView` except that it doesn't need the `BUFFER_TYPE` parameter. `LvArray::ArrayView` is a feather weight object that consists only of three pointers: a pointer to the values, a pointer to the dimensions of the array and a pointer to the strides of the dimensions. `LvArray::ArraySlice` has shallow copy semantics but no assignment operators and every method is `const`.

Unlike `LvArray::Array` and `LvArray::ArrayView` the `LvArray::ArraySlice` can refer to a non-contiguous set of values. As such it does not have a method `data` but instead has `dataIfContiguous` that performs a runtime check and aborts execution if the `LvArray::ArraySlice` does not refer to a contiguous set of values. Every method is `const` and callable on device.

```
TEST( Array, arraySlice )
{
  {
```

(continues on next page)

(continued from previous page)

```
    LvArray::Array< int,
                    2,
                    camp::idx_seq< 0, 1 >,
                    std::ptrdiff_t,
                    LvArray::MallocBuffer > array( 5, 6 );

    // The unit stride dimension of array is 1 so when we slice off
    // the first dimension the unit stride dimension of the slice is 0.
    LvArray::ArraySlice< int,
                         1,
                         0,
                         std::ptrdiff_t > const slice = array[ 2 ];
    EXPECT_TRUE( slice.isContiguous() );
    EXPECT_EQ( slice.size(), 6 );
    EXPECT_EQ( slice.size( 0 ), 6 );
    slice[ 3 ] = 1;
  }

  {
    LvArray::Array< int,
                    3,
                    camp::idx_seq< 2, 1, 0 >,
                    std::ptrdiff_t,
                    LvArray::MallocBuffer > array( 3, 5, 6 );

    // The unit stride dimension of array is 0 so when we slice off
    // the first dimension the unit stride dimension of the slice is -1.
    LvArray::ArraySlice< int,
                         2,
                         -1,
                         std::ptrdiff_t > const slice = array[ 2 ];
    EXPECT_FALSE( slice.isContiguous() );
    EXPECT_EQ( slice.size(), 5 * 6 );
    EXPECT_EQ( slice.size( 0 ), 5 );
    EXPECT_EQ( slice.size( 1 ), 6 );
    slice( 3, 4 ) = 1;
  }
}
```

[Source: examples/exampleArray.cpp]

Note: A `LvArray::ArraySlice` should not be captured in a device kernel, even if the slice comes from an array that has be moved to the device. This is because `LvArray::ArraySlice` only contains a pointer to the dimension sizes and strides. Therefore when the slice is `memcpy`'d to device the size and stride pointers will still point to host memory. This does not apply if you construct the slice manually and pass it device pointers but this is not a common use case.

3.7 Usage with `LvArray::ChaiBuffer`

When using the `LvArray::ChaiBuffer` as the buffer type the `LvArray::Array` can exist in multiple memory spaces. It can be explicitly moved between spaces with the method `move` and when the RAJA execution context is set the `LvArray::ArrayView` copy constructor will ensure that the newly constructed view's allocation is in the associated memory space.

```
CUDA_TEST( Array, chaiBuffer )
{
  LvArray::Array< int,
                  2,
                  camp::idx_seq< 1, 0 >,
                  std::ptrdiff_t,
                  LvArray::ChaiBuffer > array( 5, 6 );

  // Move the array to the device.
  array.move( LvArray::MemorySpace::cuda );
  int * const devicePointer = array.data();

  RAJA::forall< RAJA::cuda_exec< 32 > >(
    RAJA::TypedRangeSegment< std::ptrdiff_t >( 0, array.size() ),
    [devicePointer] __device__ ( std::ptrdiff_t const i )
  {
    devicePointer[ i ] = i;
  }
    );

  LvArray::ArrayView< int,
                      2,
                      0,
                      std::ptrdiff_t,
                      LvArray::ChaiBuffer > const & view = array;

  // Capture the view in a host kernel which moves the data back to the host.
  RAJA::forall< RAJA::loop_exec >(
    RAJA::TypedRangeSegment< std::ptrdiff_t >( 0, view.size() ),
    [view] ( std::ptrdiff_t const i )
  {
    EXPECT_EQ( view.data()[ i ], i );
  }
    );
}
```

[Source: examples/exampleArray.cpp]

Like the `LvArray::ChaiBuffer` a new allocation is created the first time the array is moved to a new space. Every time the array is moved between spaces it is touched in the new space unless the value type `T` is `const` or the array was moved with the `move` method and the optional `touch` parameter was set to `false`. The data is only copied between memory spaces if the last space the array was touched in is different from the space to move to. The name associated with the Array can be set with `setName`.

```
TEST( Array, setName )
{
  LvArray::Array< int,
                  2,
                  camp::idx_seq< 1, 0 >,
                  std::ptrdiff_t,
                  LvArray::ChaiBuffer > array( 1024, 1024 );

  // Move the array to the device.
  array.move( LvArray::MemorySpace::cuda );

  // Provide a name and move the array to the host.
  array.setName( "my_array" );
```

(continues on next page)

(continued from previous page)

```
  array.move( LvArray::MemorySpace::host );
}
```

[Source: examples/exampleArray.cpp]

Output

```
Moved    4.0 MB to the DEVICE: LvArray::Array<int, 2, camp::int_seq<long, 1l, 0l>,␣
↪long, LvArray::ChaiBuffer>
Moved    4.0 MB to the HOST  : LvArray::Array<int, 2, camp::int_seq<long, 1l, 0l>,␣
↪long, LvArray::ChaiBuffer> my_array
```

3.8 Interacting with an `LvArray::Array` of arbitrary dimension.

Often it can be useful to write a template function that can operate on an `LvArray::Array` with an arbitrary number of dimension. If the order the data is accessed in is not important you can use `data()` or the iterator interface `begin` and `end`. For example you could write a function to sum up the values in an array as follows

```
template< int NDIM, int USD >
int sum( LvArray::ArraySlice< int const, NDIM, USD, std::ptrdiff_t > const slice )
{
  int value = 0;
  for( int const val : slice )
  {
    value += val;
  }

  return value;
}
```

[Source: examples/exampleArray.cpp]

However this same approach might not work as intended for floating point arrays because the order of additions and hence the answer would change with the data layout. To calculate the sum consistently you need to iterate over the multidimensional index space of the array. For an array with a fixed number of dimensions you could just nest the appropriate number of for loops. `LvArray::forValuesInSlice` defined in `sliceHelpers.hpp` solves this problem, it iterates over the values in a `LvArray::ArraySlice` in a consistent order regardless of the permutation. Using this `sum` could be written as

```
template< int NDIM, int USD >
double sum( LvArray::ArraySlice< double const, NDIM, USD, std::ptrdiff_t > const␣
↪slice )
{
  double value = 0;
  LvArray::forValuesInSlice( slice, [&value] ( double const val )
  {
    value += val;
  } );

  return value;
}
```

[Source: examples/exampleArray.cpp]

`sliceHelpers.hpp` also provides a function `forAllValuesInSliceWithIndices` which calls the user provided callback with the indicies to each value in addition to the value. This can be used to easily copy data between arrays with different layouts.

```
template< int NDIM, int DST_USD, int SRC_USD >
void copy( LvArray::ArraySlice< int, NDIM, DST_USD, std::ptrdiff_t > const dst,
           LvArray::ArraySlice< int const, NDIM, SRC_USD, std::ptrdiff_t > const src )
{
  for( int dim = 0; dim < NDIM; ++dim )
  {
    LVARRAY_ERROR_IF_NE( dst.size( dim ), src.size( dim ) );
  }

  LvArray::forValuesInSliceWithIndices( dst,
                                        [src] ( int & val, auto const ... indices )
  {
    val = src( indices ... );
  }
                                        );
}
```

[Source: examples/exampleArray.cpp]

Finally you can write a recursive function that operates on an `LvArray::ArraySlice`. An example of this is the stream output method for `LvArray::Array`.

```
template< typename T, int NDIM, int USD, typename INDEX_TYPE >
std::ostream & operator<<( std::ostream & stream,
                           ::LvArray::ArraySlice< T, NDIM, USD, INDEX_TYPE > const␣
↪slice )
{
  stream << "{ ";

  if( slice.size( 0 ) > 0 )
  {
    stream << slice[ 0 ];
  }

  for( INDEX_TYPE i = 1; i < slice.size( 0 ); ++i )
  {
    stream << ", " << slice[ i ];
  }

  stream << " }";
  return stream;
}
```

[Source: src/output.hpp]

3.9 Usage with `LVARRAY_BOUNDS_CHECK`

When `LVARRAY_BOUNDS_CHECK` is defined all accesses via `operator[]` and `operator()` is checked to make sure the indices are valid. If invalid indices are detected an error message is printed to standard out and the program is aborted. It should be noted that access via `operator()` is able to provide a more useful error message upon an invalid access because it has access to all of the indices whereas `operator[]` only has access to a single index at

a time. `size( int dim )`, `linearIndex`, `emplace` and `insert` will also check that their arguments are in bounds.

```
TEST( Array, boundsCheck )
{
#if defined(ARRAY_USE_BOUNDS_CHECK)
  LvArray::Array< int, 3, camp::idx_seq< 0, 1, 2 >, std::ptrdiff_t,␣
↪LvArray::MallocBuffer > x( 3, 4, 5 );

  // Out of bounds access aborts the program.
  EXPECT_DEATH_IF_SUPPORTED( x( 2, 3, 4 ), "" );
  EXPECT_DEATH_IF_SUPPORTED( x( -1, 4, 6 ), "" );
  EXPECT_DEATH_IF_SUPPORTED( x[ 0 ][ 10 ][ 2 ], "" );

  // Out of bounds emplace
  LvArray::Array< int, 1, camp::idx_seq< 0 >, std::ptrdiff_t, LvArray::MallocBuffer >␣
↪x( 10 );
  EXPECT_DEATH_IF_SUPPORTED( x.emplace( -1, 5 ) );
#endif
}
```

[Source: examples/exampleArray.cpp]

3.10 Guidelines

In general you should opt to pass around the most restrictive array possible. If a function takes in an array and only needs to read and write the values then it should take in an `LvArray::ArraySlice`. If it needs to read the values and captures the array in a kernel then it should take in an `LvArray::ArrayView< T const, ... >`. Only when a function needs to resize or reallocate the array should it accept an `LvArray::Array`.

Our benchmarks have shown no significant performance difference between using `operator()` and nested applications of `operator[]`, if you do see a difference let us know!

3.11 Doxygen

- LvArray::Array
- LvArray::ArrayView
- LvArray::ArraySlice

CHAPTER 4

LvArray::SortedArray

The LvArray::SortedArray functions similarly to a std::set except that unlike a std::set the values are stored contiguously in memory like a std::vector. Like the std::set the cost of seeing if a LvArray::SortedArray a contains a value is $O(log(N))$ however the const of inserting or removing a value is $O(N)$ where N = a.size().

4.1 Template arguments

The LvArray::SortedArray requires three template arguments.

1. T: The type of values stored in the set.
2. INDEX_TYPE: An integral type used in index calculations, the suggested type is std::ptrdiff_t.
3. BUFFER_TYPE: A template template parameter specifying the buffer type used for allocation and de-allocation, the LvArray::SortedArray contains a BUFFER_TYPE< T >.

Note: Unlike std::set, LvArray::SortedArray does not yet support a custom comparator, it us currently hard coded to operator< to sort values from least to greatest.

4.2 Creating and accessing a LvArray::SortedArray

The LvArray::SortedArray has a single default constructor which creates an empty set. The only way to modify the set is through the methods insert and remove. These methods are similar to the std::set methods of the same name except that when inserting or removing multiple values at once the values need to be sorted and unique. LvArray::SortedArray also has an operator[] and data method that provide read only access to the values.

```
TEST( SortedArray, construction )
{
  // Construct an empty set.
  LvArray::SortedArray< std::string, std::ptrdiff_t, LvArray::MallocBuffer > set;
  EXPECT_TRUE( set.empty() );

  // Insert two objects one at a time.
  EXPECT_TRUE( set.insert( "zebra" ) );
  EXPECT_TRUE( set.insert( "aardvark" ) );

  // "zebra" is already in the set so it won't be inserted again.
  EXPECT_FALSE( set.insert( "zebra" ) );

  // Query the contents of the set.
  EXPECT_EQ( set.size(), 2 );
  EXPECT_TRUE( set.contains( "zebra" ) );
  EXPECT_FALSE( set.contains( "whale" ) );

  // Insert two objects at once.
  std::string const moreAnimals[ 2 ] = { "cat", "dog" };
  EXPECT_EQ( set.insert( moreAnimals, moreAnimals + 2 ), 2 );

  // Remove a single object.
  set.remove( "aardvark" );

  EXPECT_EQ( set.size(), 3 );
  EXPECT_EQ( set[ 0 ], "cat" );
  EXPECT_EQ( set[ 1 ], "dog" );
  EXPECT_EQ( set[ 2 ], "zebra" );
}
```

[Source: examples/exampleSortedArray.cpp]

4.3 `LvArray::SortedArrayView`

`LvArray::SortedArrayView` is the view class of `LvArray::SortedArray` and it shares all the same template parameters. To construct a `LvArray::SortedArrayView` you must call `toView()` on an existing `LvArray::SortedArray` or create a copy of an existing `LvArray::SortedArrayView`. The `LvArray::SortedArrayView` is not allowed to insert or remove values. As such the return type of `LvArray::SortedArray< T, ... >::toView()` is an `LvArray::SortedArrayView< T const, ... >`

Note: Unlike `LvArray::ArrayView` the `LvArray::SortedArrayView` does not yet support default construction.

4.4 Usage with `LvArray::ChaiBuffer`

When using the `LvArray::ChaiBuffer` as the buffer type the `LvArray::SortedArray` can exist in multiple memory spaces. It can be explicitly moved between spaces with the method `move`. Because the `SortedArrayView` cannot modify the values the data is never touched when moving to device, even if the optional `touch` parameter is set to false.

It is worth noting that after a `LvArray::SortedArray` is moved to the device it must be explicitly moved back to the host by calling `move( MemorySpace::host )` before it can be safely modified. This won't actually trigger a memory copy since the values weren't touched on device, its purpose is to let CHAI know that the values were touched on the host so that the next time it is moved to device it will copy the values back over.

```
CUDA_TEST( SortedArray, ChaiBuffer )
{
  // Construct an empty set consisting of the even numbers { 0, 2, 4 }.
  LvArray::SortedArray< int, std::ptrdiff_t, LvArray::ChaiBuffer > set;
  int const values[ 4 ] = { 0, 2, 4 };
  EXPECT_EQ( set.insert( values, values + 3 ), 3 );
  EXPECT_EQ( set.size(), 3 );

  // Create a view and capture it on device, this will copy the data to the device.
  RAJA::forall< RAJA::cuda_exec< 32 > >(
    RAJA::TypedRangeSegment< std::ptrdiff_t >( 0, set.size() ),
    [view = set.toView()] __device__ ( std::ptrdiff_t const i )
  {
    LVARRAY_ERROR_IF_NE( view[ i ], 2 * i );
    LVARRAY_ERROR_IF( view.contains( 2 * i + 1 ), "The set should only contain odd␣
↪numbers!" );
  }
    );

  // Move the set back to the CPU and modify it.
  set.move( LvArray::MemorySpace::host );
  set.insert( 6 );

  // Verify that the modification is seen on device.
  RAJA::forall< RAJA::cuda_exec< 32 > >(
    RAJA::TypedRangeSegment< std::ptrdiff_t >( 0, 1 ),
    [view = set.toView()] __device__ ( std::ptrdiff_t const )
  {
    LVARRAY_ERROR_IF( !view.contains( 6 ), "The set should contain 6!" );
  }
    );
}
```

[Source: examples/exampleSortedArray.cpp]

4.5 Usage with `LVARRAY_BOUNDS_CHECK`

Like `LvArray::Array` when `LVARRAY_BOUNDS_CHECK` is defined access via `operator[]` is checked for invalid access. If an out of bounds access is detected the program is aborted. In addition calls to insert and remove multiple values will error out if the values to insert or remove aren't sorted and unique.

```
TEST( SortedArray, boundsCheck )
{
  // Create a set containing {2, 4}
  LvArray::SortedArray< int, std::ptrdiff_t, LvArray::MallocBuffer > set;
  set.insert( 4 );
  set.insert( 2 );

  // Invalid access.
  EXPECT_DEATH_IF_SUPPORTED( set[ 5 ], "" );
```

(continues on next page)

(continued from previous page)

```
  // Attempt to insert unsorted values.
  int const unsortedInsert[ 2 ] = { 4, 0 };
  EXPECT_DEATH_IF_SUPPORTED( set.insert( unsortedInsert, unsortedInsert + 2 ), "" );

  // Attempt to insert nonUnique values.
  int const notUnique[ 2 ] = { 5, 5 };
  EXPECT_DEATH_IF_SUPPORTED( set.insert( notUnique, notUnique + 2 ), "" );

  // Attempt to remove unsorted values.
  int const unsortedRemove[ 2 ] = { 4, 2 };
  EXPECT_DEATH_IF_SUPPORTED( set.remove( unsortedRemove, unsortedRemove + 2 ), "" );
}
```

[Source: examples/exampleSortedArray.cpp]

4.6 Guidelines

Batch insertion and removal is much faster than inserting or removing each value individually. For example calling `a.insert( 5 )` has complexity `O( a.size() )` but calling `a.insert( first, last )` only has complexity `O( a.size() + std::distance( first, last ) )`. The function `LvArray::sortedArrayManipulation::makeSortedUnique` can help with this as it takes a range, sorts it and removes any duplicate values. When possible it is often faster to append values to a temporary container, sort the values, remove duplicates and then perform the operation.

```
TEST( SortedArray, fastConstruction )
{
  LvArray::SortedArray< int, std::ptrdiff_t, LvArray::MallocBuffer > set;

  // Create a temporary list of 100 random numbers between 0 and 99.
  std::vector< int > temporarySpace( 100 );
  std::mt19937 gen;
  std::uniform_int_distribution< int > dis( 0, 99 );
  for( int i = 0; i < 100; ++i )
  {
    temporarySpace[ i ] = dis( gen );
  }

  // Sort the random numbers and move any duplicates to the end.
  std::ptrdiff_t const numUniqueValues =␣
↪LvArray::sortedArrayManipulation::makeSortedUnique( temporarySpace.begin(),
                                                                              ␣
↪       temporarySpace.end() );

  // Insert into the set.
  set.insert( temporarySpace.begin(), temporarySpace.begin() + numUniqueValues );
}
```

[Source: examples/exampleSortedArray.cpp]

4.7 Doxygen

- LvArray::SortedArray

- LvArray::SortedArrayView

CHAPTER 5

LvArray::ArrayOfArrays

The LvArray::ArrayOfArrays provides functionality similar to a std::vector< std::vector< T > > but with just three allocations. The primary purpose is to reduce the number of memory transfers when moving between memory spaces but it also comes with the added benefit of reduced memory fragmentation.

5.1 Template arguments

The LvArray::ArrayOfArrays requires three template arguments.

1. T: The type of values stored in the inner arrays.
2. INDEX_TYPE: An integral type used in index calculations, the suggested type is std::ptrdiff_t.
3. BUFFER_TYPE: A template template parameter specifying the buffer type used for allocation and de-allocation, the LvArray::ArrayOfArrays contains a BUFFER_TYPE< T > along with two BUFFER_TYPE< INDEX_TYPE >.

5.2 Usage

LvArray::ArrayOfArrays has a single constructor which takes two optional parameters; the number of inner arrays and the capacity to allocate for each inner array. Both values default to zero.

Given an ArrayOfArrays< T, ... > array, an equivalent std::vector< std::vector< T > > vector, a non negative integer n, integer i such that 0 <= i < vector.size(), integer j such that 0 <= j < vector[i].size(), two iterators first and last and a variadic pack of parameters ... args then following are equivalent:

Getting information about the outer array or a specific inner array	
`LvArray::ArrayOfArrays< T, ... >`	`std::vector< std::vector< T > >`
`array.size()`	`vector.size()`
`array.capacity()`	`vector.capacity()`
`array.sizeOfArray( i )`	`vector[ i ].size()`
`array.capacityOfArray( i )`	`vector[ i ].capacity()`

Modifying the outer array	
`LvArray::ArrayOfArrays< T, ... >`	`std::vector< std::vector< T > >`
`array.reserve( n )`	`vector.reserve( n )`
`array.resize( n )`	`vector.resize( n )`
`array.appendArray( n )`	`vector.push_back( std::vector< T >( n ) )`
`array.appendArray( first, last )`	`vector.push_back( std::vector< T >( first, last ) )`
`array.insertArray( i, first, last )`	`vector.insert( vector.begin() + i, std::vector< T >( first, last ) )`
`array.eraseArray( i )`	`vector.erase( vector.first() + i )`

Modifying an inner array	
`LvArray::ArrayOfArrays< T, ... >`	`std::vector< std::vector< T > >`
`array.resizeArray( i, n, args ... )`	`vector[ i ].resize( n, T( args ... ) )`
`array.clearArray( i )`	`vector[ i ].clear()`
`array.emplaceBack( i, args ... )`	`vector[ i ].emplace_back( args ... )`
`array.appendToArray( i, first, last )`	`vector[ i ].insert( vector[ i ].end(), first, last )`
`array.emplace( i, j , args ... )`	`vector[i].emplace(j, args ...)``
`array.insertIntoArray( i, j , first, last )`	`vector[ i ].insert( vector[ i ].begin() + j, first, last )`
`eraseFromArray( i, j, n )`	`vector[ i ].erase( vector[ i ].begin() + j, vector[ i ].begin() + j + n ).`

Accessing the data	
`LvArray::ArrayOfArrays< T, ... >`	`std::vector< std::vector< T > >`
`array( i, j )`	`vector[ i ][ j ]`
`array[ i ][ j ]`	`vector[ i ][ j ]`

It is worth noting that `operator[]` returns a one dimensional `LvArray::ArraySlice`.

```
TEST( ArrayOfArrays, construction )
{
  // Create an empty ArrayOfArrays.
  LvArray::ArrayOfArrays< std::string, std::ptrdiff_t, LvArray::MallocBuffer >␣
↪arrayOfArrays;
  EXPECT_EQ( arrayOfArrays.size(), 0 );
```

(continues on next page)

(continued from previous page)

```
  // Append an array of length 2.
  arrayOfArrays.appendArray( 2 );
  EXPECT_EQ( arrayOfArrays.size(), 1 );
  EXPECT_EQ( arrayOfArrays.sizeOfArray( 0 ), 2 );
  EXPECT_EQ( arrayOfArrays.capacityOfArray( 0 ), 2 );
  arrayOfArrays( 0, 0 ) = "First array, first entry.";
  arrayOfArrays( 0, 1 ) = "First array, second entry.";

  // Append another array of length 3.
  arrayOfArrays.appendArray( 3 );
  EXPECT_EQ( arrayOfArrays.size(), 2 );
  EXPECT_EQ( arrayOfArrays.sizeOfArray( 1 ), 3 );
  EXPECT_EQ( arrayOfArrays.capacityOfArray( 1 ), 3 );
  arrayOfArrays( 1, 0 ) = "Second array, first entry.";
  arrayOfArrays( 1, 2 ) = "Second array, third entry.";

  EXPECT_EQ( arrayOfArrays[ 0 ][ 1 ], "First array, second entry." );
  EXPECT_EQ( arrayOfArrays[ 1 ][ 2 ], "Second array, third entry." );

  // Values are default initialized.
  EXPECT_EQ( arrayOfArrays[ 1 ][ 1 ], std::string() );
}

TEST( ArrayOfArrays, modification )
{
  LvArray::ArrayOfArrays< std::string, std::ptrdiff_t, LvArray::MallocBuffer >␣
↪arrayOfArrays;

  // Append an array.
  arrayOfArrays.appendArray( 3 );
  arrayOfArrays( 0, 0 ) = "First array, first entry.";
  arrayOfArrays( 0, 1 ) = "First array, second entry.";
  arrayOfArrays( 0, 2 ) = "First array, third entry.";

  // Insert a new array at the beginning.
  std::array< std::string, 2 > newFirstArray = { "New first array, first entry.",
                                                 "New first array, second entry." };
  arrayOfArrays.insertArray( 0, newFirstArray.begin(), newFirstArray.end() );

  EXPECT_EQ( arrayOfArrays.size(), 2 );
  EXPECT_EQ( arrayOfArrays.sizeOfArray( 0 ), 2 );
  EXPECT_EQ( arrayOfArrays.sizeOfArray( 1 ), 3 );

  EXPECT_EQ( arrayOfArrays( 0, 1 ), "New first array, second entry." );
  EXPECT_EQ( arrayOfArrays[ 1 ][ 1 ], "First array, second entry." );

  // Erase the values from what is now the second array.
  arrayOfArrays.clearArray( 1 );
  EXPECT_EQ( arrayOfArrays.sizeOfArray( 1 ), 0 );

  // Append a value to the end of each array.
  arrayOfArrays.emplaceBack( 1, "Second array, first entry." );
  arrayOfArrays.emplaceBack( 0, "New first array, third entry." );

  EXPECT_EQ( arrayOfArrays.sizeOfArray( 0 ), 3 );
  EXPECT_EQ( arrayOfArrays.sizeOfArray( 1 ), 1 );
```

(continues on next page)

(continued from previous page)

```
  EXPECT_EQ( arrayOfArrays[ 1 ][ 0 ], "Second array, first entry." );
  EXPECT_EQ( arrayOfArrays( 0, 2 ), "New first array, third entry." );
}
```

[Source: examples/exampleArrayOfArrays.cpp]

5.3 LvArray::ArrayOfArraysView

`LvArray::ArrayOfArraysView` is the view counterpart to `LvArray::ArrayOfArrays`. The `LvArray::ArrayOfArraysView` shares the three template parameters of `LvArray::ArrayOfArraysView` but it has an additional boolean parameter `CONST_SIZES` which specifies if the view can change the sizes of the inner arrays. If `CONST_SIZES` is true the `sizes` buffer has type `BUFFER_TYPE< INDEX_TYPE const >` otherwise it has type `BUFFER_TYPE< std::remove_const_t< INDEX_TYPE > >`.

The `LvArray::ArrayOfArraysView` doesn't have a default constructor, it must always be created from an existing `LvArray::ArrayOfArrays`. From a `LvArray::ArrayOfArrays< T, INDEX_TYPE, BUFFER_TYPE >` you can get three different view types.

- `toView()` returns a `LvArray::ArrayOfArraysView< T, INDEX_TYPE const, false, BUFFER_TYPE >` which can modify existing values and change the size of the inner arrays as long as their size doesn't exceed their capacity. -
- `toViewConstSizes()` returns a `LvArray::ArrayOfArraysView< T, INDEX_TYPE const, true, BUFFER_TYPE >` which can modify existing values but cannot change the size of the inner arrays.
- `toViewConst()` returns a `LvArray::ArrayOfArraysView< T const, INDEX_TYPE const, true, BUFFER_TYPE >` which provides read only access.

```
TEST( ArrayOfArrays, view )
{
  // Create an ArrayOfArrays with 10 inner arrays each with capacity 9.
  LvArray::ArrayOfArrays< int, std::ptrdiff_t, LvArray::MallocBuffer > arrayOfArrays(␣
↪10, 9 );

  {
    // Then create a view.
    LvArray::ArrayOfArraysView< int,
                                std::ptrdiff_t const,
                                false,
                                LvArray::MallocBuffer > const view = arrayOfArrays.
↪toView();
    EXPECT_EQ( view.size(), 10 );
    EXPECT_EQ( view.capacityOfArray( 7 ), 9 );

    // Modify every inner array in parallel
    RAJA::forall< RAJA::omp_parallel_for_exec >(
      RAJA::TypedRangeSegment< std::ptrdiff_t >( 0, view.size() ),
      [view] ( std::ptrdiff_t const i )
    {
      for( std::ptrdiff_t j = 0; j < i; ++j )
      {
        view.emplaceBack( i, 10 * i + j );
      }
    }
```

(continues on next page)

(continued from previous page)

```
    );

  EXPECT_EQ( view.sizeOfArray( 9 ), view.capacityOfArray( 9 ) );

  // The last array is at capacity. Therefore any attempt at increasing its size␣
↪through
  // a view is invalid and may lead to undefined behavior!
  // view.emplaceBack( 9, 0 );
}

{
  // Create a view which cannot modify the sizes of the inner arrays.
  LvArray::ArrayOfArraysView< int,
                              std::ptrdiff_t const,
                              true,
                              LvArray::MallocBuffer > const viewConstSizes =␣
↪arrayOfArrays.toViewConstSizes();
  for( std::ptrdiff_t i = 0; i < viewConstSizes.size(); ++i )
  {
    for( int & value : viewConstSizes[ i ] )
    {
      value *= 2;
    }

    // This would be a compilation error
    // viewConstSizes.emplaceBack( i, 0 );
  }
}

{
  // Create a view which has read only access.
  LvArray::ArrayOfArraysView< int const,
                              std::ptrdiff_t const,
                              true,
                              LvArray::MallocBuffer > const viewConst =␣
↪arrayOfArrays.toViewConst();

  for( std::ptrdiff_t i = 0; i < viewConst.size(); ++i )
  {
    for( std::ptrdiff_t j = 0; j < viewConst.sizeOfArray( i ); ++j )
    {
      EXPECT_EQ( viewConst( i, j ), 2 * ( 10 * i + j ) );

      // This would be a compilation error
      // viewConst( i, j ) = 0;
    }
  }
}

// Verify that all the modifications are present in the parent ArrayOfArrays.
EXPECT_EQ( arrayOfArrays.size(), 10 );
for( std::ptrdiff_t i = 0; i < arrayOfArrays.size(); ++i )
{
  EXPECT_EQ( arrayOfArrays.sizeOfArray( i ), i );
  for( std::ptrdiff_t j = 0; j < arrayOfArrays.sizeOfArray( i ); ++j )
  {
    EXPECT_EQ( arrayOfArrays( i, j ), 2 * ( 10 * i + j ) );
```

(continues on next page)

(continued from previous page)

```
    }
  }
}
```

[Source: examples/exampleArrayOfArrays.cpp]

`LvArray::ArrayView` also has a method `emplaceBackAtomic` which is allows multiple threads to append to a single inner array in a thread safe manner.

```
TEST( ArrayOfArrays, atomic )
{
  // Create an ArrayOfArrays with 1 array with a capacity of 100.
  LvArray::ArrayOfArrays< int, std::ptrdiff_t, LvArray::MallocBuffer > arrayOfArrays(␣
↪1, 100 );
  EXPECT_EQ( arrayOfArrays.sizeOfArray( 0 ), 0 );

  // Then create a view.
  LvArray::ArrayOfArraysView< int,
                              std::ptrdiff_t const,
                              false,
                              LvArray::MallocBuffer > const view = arrayOfArrays.
↪toView();

  // Append to the single inner array in parallel
  RAJA::forall< RAJA::omp_parallel_for_exec >(
    RAJA::TypedRangeSegment< std::ptrdiff_t >( 0, view.capacityOfArray( 0 ) ),
    [view] ( std::ptrdiff_t const i )
  {
    view.emplaceBackAtomic< RAJA::builtin_atomic >( 0, i );
  }
    );

  EXPECT_EQ( arrayOfArrays.sizeOfArray( 0 ), 100 );

  // Sort the entries in the array since they were appended in an arbitrary order
  LvArray::sortedArrayManipulation::makeSorted( arrayOfArrays[ 0 ].begin(),␣
↪arrayOfArrays[ 0 ].end() );

  for( std::ptrdiff_t i = 0; i < arrayOfArrays.sizeOfArray( 0 ); ++i )
  {
    EXPECT_EQ( arrayOfArrays( 0, i ), i );
  }
}
```

[Source: examples/exampleArrayOfArrays.cpp]

5.4 Data structure

In order to use the `LvArray::ArrayOfArrays` efficiently it is important to understand the underlying data structure. A `LvArray::ArrayOfArrays< T, INDEX_TYPE, BUFFER_TYPE > array` contains three buffers:

1. `BUFFER_TYPE< T > values`: Contains the values of each inner array.
2. `BUFFER_TYPE< INDEX_TYPE > sizes`: Of length `array.size()`, `sizes[ i ]` contains the size of inner array `i`.

3. `BUFFER_TYPE< INDEX_TYPE > offsets`: Of length `array.size() + 1`, inner array `i` begins at `values[ offsets[ i ] ]` and has capacity `offsets[ i + 1 ] - offsets[ i ]`.

Given `M = offsets[ array.size() ]` which is the sum of the capacities of the inner arrays then

- `array.appendArray( n )` is O(n) because it entails appending a new entry to `sizes` and `offsets` and then appending the `n` new values to `values`.
- `array.insertArray( i, first, last )` is `O( array.size() + M + std::distance( first, last ) )`. It involves inserting an entry into `sizes` and `offsets` which is `O( array.size() )`, making room in `values` for the new array which is `O( M )` and finally copying over the new values which is `O( std::distance( first, last ) )`.
- `array.eraseArray( i )` is `O( array.size() + M )`. It involves removing an entry from `sizes` and `offsets` which is `O( array.size() )` and then it involves shifting the entries in `values` down which is `O( M )`.
- The methods which modify an inner array have the same complexity as their `std::vector` counterparts **if** the capacity of the inner array won't be exceeded by the operation. Otherwise they have an added cost of `O( M )` because new space will have to be made in `values`. So for example `array.emplaceBack( i, args... )` is `O( 1 )` if `array.sizeOfArray( i ) < array.capacityOfArray( i )` and `O( M )` otherwise.

`LvArray::ArrayOfArrays` also supports two methods that don't have an `std::vector< std::vector >` counterpart. The first is `compress` which shrinks the capacity of each inner array to match it's size. This ensures that the inner arrays are contiguous in memory with no extra space between them.

```
TEST( ArrayOfArrays, compress )
{
  // Create an ArrayOfArrays with three inner arrays each with capacity 5.
  LvArray::ArrayOfArrays< int, std::ptrdiff_t, LvArray::MallocBuffer > arrayOfArrays(␣
↪3, 5 );
  EXPECT_EQ( arrayOfArrays.sizeOfArray( 1 ), 0 );
  EXPECT_EQ( arrayOfArrays.capacityOfArray( 1 ), 5 );

  // Append values to the inner arrays.
  std::array< int, 5 > values = { 0, 1, 2, 3, 4 };
  arrayOfArrays.appendToArray( 0, values.begin(), values.begin() + 3 );
  arrayOfArrays.appendToArray( 1, values.begin(), values.begin() + 4 );
  arrayOfArrays.appendToArray( 2, values.begin(), values.begin() + 5 );

  EXPECT_EQ( arrayOfArrays.sizeOfArray( 0 ), 3 );
  EXPECT_EQ( arrayOfArrays.capacityOfArray( 0 ), 5 );

  // Since the first array has some extra capacity it is not contiguous in memory
  // with the second array.
  EXPECT_NE( &arrayOfArrays( 0, 2 ) + 1, &arrayOfArrays( 1, 0 ) );

  // Compress the ArrayOfArrays. This doesn't change the sizes or values of the
  // inner arrays, only their capacities.
  arrayOfArrays.compress();

  EXPECT_EQ( arrayOfArrays.sizeOfArray( 0 ), 3 );
  EXPECT_EQ( arrayOfArrays.capacityOfArray( 0 ), 3 );

  // The values are preserved.
  EXPECT_EQ( arrayOfArrays( 2, 4 ), 4 );

  // And the inner arrays are now contiguous.
```

(continues on next page)

(continued from previous page)

```
  EXPECT_EQ( &arrayOfArrays( 0, 2 ) + 1, &arrayOfArrays( 1, 0 ) );
}
```

[Source: examples/exampleArrayOfArrays.cpp]

The second is `resizeFromCapacities` which takes in a number of inner arrays and the capacity of each inner array. It clears the `ArrayOfArrays` and reconstructs it with the given number of empty inner arrays each with the provided capacity. It also takes a RAJA execution policy as a template parameter which specifies how to compute the offsets array from the capacities. Currently only host execution policies are supported.

```
TEST( ArrayOfArrays, resizeFromCapacities )
{
  // Create an ArrayOfArrays with two inner arrays
  LvArray::ArrayOfArrays< int, std::ptrdiff_t, LvArray::MallocBuffer > arrayOfArrays(␣
↪2 );

  // Append values to the inner arrays.
  std::array< int, 5 > values = { 0, 1, 2, 3, 4 };
  arrayOfArrays.appendToArray( 0, values.begin(), values.begin() + 3 );
  arrayOfArrays.appendToArray( 1, values.begin(), values.begin() + 4 );

  EXPECT_EQ( arrayOfArrays.sizeOfArray( 0 ), 3 );

  // Resize the ArrayOfArrays from a new list of capacities.
  std::array< std::ptrdiff_t, 3 > newCapacities = { 3, 5, 2 };
  arrayOfArrays.resizeFromCapacities< RAJA::loop_exec >( newCapacities.size(),␣
↪newCapacities.data() );

  // This will clear any existing arrays.
  EXPECT_EQ( arrayOfArrays.size(), 3 );
  EXPECT_EQ( arrayOfArrays.sizeOfArray( 1 ), 0 );
  EXPECT_EQ( arrayOfArrays.capacityOfArray( 1 ), newCapacities[ 1 ] );
}
```

[Source: examples/exampleArrayOfArrays.cpp]

5.5 Usage with `LvArray::ChaiBuffer`

The three types of `LvArray::ArrayOfArrayView` obtainable from an `LvArray::ArrayOfArrays` all act differently when moved to a new memory space.

- `LvArray::ArrayOfArraysView< T, INDEX_TYPE const, false, LvArray::ChaiBuffer >`, obtained by calling `toView()`. When it is moved to a new space the values are touched as well as the sizes. The offsets are not touched.
- `LvArray::ArrayOfArraysView< T, INDEX_TYPE const, true, LvArray::ChaiBuffer >`, obtained by calling `toViewConstSizes()`. When it is moved to a new space the values are touched but the sizes and offsets aren't.
- `LvArray::ArrayOfArraysView< T const, INDEX_TYPE const, true, LvArray::ChaiBuffer >`, obtained by calling `toViewConst()`. None of the buffers are touched in the new space.

Calling the explicit `move` method with the touch parameter set to `true` on a view type has the behavior described above. However calling `move( MemorySpace::host )` on an `LvArray::ArrayOfArrays` will also touch the offsets (if moving to the GPU the offsets aren't touched). This is the only way to touch the offsets so if an

`LvArray::ArrayOfArrays` was previously on the device then it must be explicitly moved and touched on the host before any modification to the offsets can safely take place.

```
CUDA_TEST( ArrayOfArrays, ChaiBuffer )
{
  LvArray::ArrayOfArrays< int, std::ptrdiff_t, LvArray::ChaiBuffer > arrayOfArrays(␣
↪10, 9 );

  {
    // Create a view.
    LvArray::ArrayOfArraysView< int,
                                std::ptrdiff_t const,
                                false,
                                LvArray::ChaiBuffer > const view = arrayOfArrays.
↪toView();

    // Capture the view on device. This will copy the values, sizes and offsets.
    // The values and sizes will be touched.
    RAJA::forall< RAJA::cuda_exec< 32 > >(
      RAJA::TypedRangeSegment< std::ptrdiff_t >( 0, view.size() ),
      [view] __device__ ( std::ptrdiff_t const i )
    {
      for( std::ptrdiff_t j = 0; j < i; ++j )
      {
        view.emplace( i, 0, 10 * i + j );
      }
    }
      );
  }

  {
    // Create a view which cannot modify the sizes of the inner arrays.
    LvArray::ArrayOfArraysView< int,
                                std::ptrdiff_t const,
                                true,
                                LvArray::ChaiBuffer > const viewConstSizes =␣
↪arrayOfArrays.toViewConstSizes();

    // Capture the view on the host. This will copy back the values and sizes since␣
↪they were previously touched
    // on device. It will only touch the values on host.
    RAJA::forall< RAJA::loop_exec >(
      RAJA::TypedRangeSegment< std::ptrdiff_t >( 0, viewConstSizes.size() ),
      [viewConstSizes] ( std::ptrdiff_t const i )
    {
      for( int & value : viewConstSizes[ i ] )
      {
        value *= 2;
      }
    }
      );
  }

  {
    // Create a view which has read only access.
    LvArray::ArrayOfArraysView< int const,
                                std::ptrdiff_t const,
                                true,
```

(continues on next page)

(continued from previous page)

```
                                LvArray::ChaiBuffer > const viewConst = arrayOfArrays.
↪toViewConst();

    // Capture the view on device. Since the values were previously touched on host␣
↪it will copy them over.
    // Both the sizes and offsets are current on device so they are not copied over.␣
↪Nothing is touched.
    RAJA::forall< RAJA::loop_exec >(
      RAJA::TypedRangeSegment< std::ptrdiff_t >( 0, viewConst.size() ),
      [viewConst] ( std::ptrdiff_t const i )
    {
      for( std::ptrdiff_t j = 0; j < viewConst.sizeOfArray( i ); ++j )
      {
        LVARRAY_ERROR_IF_NE( viewConst( i, j ), 2 * ( 10 * i + i - j - 1 ) );
      }
    }
      );
  }

  // This won't copy any data since everything is current on host. It will however␣
↪touch the values,
  // sizes and offsets.
  arrayOfArrays.move( LvArray::MemorySpace::host );

  // Verify that all the modifications are present in the parent ArrayOfArrays.
  EXPECT_EQ( arrayOfArrays.size(), 10 );
  for( std::ptrdiff_t i = 0; i < arrayOfArrays.size(); ++i )
  {
    EXPECT_EQ( arrayOfArrays.sizeOfArray( i ), i );
    for( std::ptrdiff_t j = 0; j < arrayOfArrays.sizeOfArray( i ); ++j )
    {
      EXPECT_EQ( arrayOfArrays( i, j ), 2 * ( 10 * i + i - j - 1 ) );
    }
  }
}
```

[Source: examples/exampleArrayOfArrays.cpp]

5.6 Usage with `LVARRAY_BOUNDS_CHECK`

When `LVARRAY_BOUNDS_CHECK` is defined access via `operator[]` and `operator()` is checked. If an invalid access is detected the program is aborted. Methods such as `sizeOfArray`, `insertArray` and `emplace` are also checked.

```
TEST( ArrayOfArrays, boundsCheck )
{
#if defined(LVARRAY_BOUNDS_CHECK)
  LvArray::ArrayOfArrays< int, std::ptrdiff_t, LvArray::MallocBuffer > arrayOfArrays;

  // Append an array.
  std::array< int, 5 > values = { 0, 1, 2, 3, 4 };
  arrayOfArrays.appendArray( values.begin(), values.end() );

  EXPECT_EQ( arrayOfArrays.size(), 1 );
```

(continues on next page)

(continued from previous page)

```
  EXPECT_EQ( arrayOfArrays.sizeOfArray( 0 ), 5 );

  // Out of bounds access aborts the program.
  EXPECT_DEATH_IF_SUPPORTED( arrayOfArrays( 0, -1 ), "" );
  EXPECT_DEATH_IF_SUPPORTED( arrayOfArrays[ 0 ][ 6 ], "" );
  EXPECT_DEATH_IF_SUPPORTED( arrayOfArrays[ 1 ][ 5 ], "" );

  EXPECT_DEATH_IF_SUPPORTED( arrayOfArrays.capacityOfArray( 5 ), "" );
  EXPECT_DEATH_IF_SUPPORTED( arrayOfArrays.insertArray( 5, values.begin(), values.
↪end() ), "" );
  EXPECT_DEATH_IF_SUPPORTED( arrayOfArrays.emplace( 0, 44, 4 ), "" );
  EXPECT_DEATH_IF_SUPPORTED( arrayOfArrays.emplace( 1, 44, 4 ), "" );
#endif
}
```

[Source: examples/exampleArrayOfArrays.cpp]

5.7 Guidelines

Like with the `LvArray::Array` it is a good idea to pass around the most restrictive `LvArray::ArrayOfArrays` object possible. If a function only reads the values of an array it should accept an `LvArray::ArrayOfArraysView< T const, INDEX_TYPE const, true, BUFFER_TYPE >`. If it only reads and writes to the values of the inner arrays it should accept an `LvArray::ArrayOfArraysView< T, INDEX_TYPE const, true, BUFFER_TYPE >`. If it will modify the size of the inner arrays and can guarantee that their capacity won't be exceeded it should accept a `LvArray::ArrayOfArraysView< T, INDEX_TYPE const, false, BUFFER_TYPE >`. Only when the function needs to modify the outer array or can't guarantee that the capacity of an inner array won't be exceeded should it accept an `LvArray::ArrayOfArrays`.

Examining the computational complexities listed above resizing the inner arrays of an `LvArray::ArrayOfArrays` can be as fast as the equivalent operations on a `std::vector< std::vector< T > >` only if the capacity of the inner arrays is not exceeded. Whenever possible preallocate space for the inner arrays. The following examples demonstrate the impact this can have.

One use case for the `LvArray::ArrayOfArrays` is to represent the node-to-element map of a computational mesh. Usually this is constructed from a provided element-to-node map which is a map from an element index to a list of node indices that make up the element. For a structured mesh or an unstructured mesh made up of a single element type this map can be represented as a two dimensional `LvArray::Array` since every element has the same number of nodes. However the inverse node-to-element map cannot be easily represented this way since in general not all nodes are a part of the same number of elements. In a two dimensional structured mesh an interior node is part of four elements while the four corner nodes are only part of a single element.

This is an example of how to construct the node-to-element map represented as a `std::vector< std::vector > >` from the element-to-node map.

```
void NaiveNodeToElemMapConstruction::
  vector( ArrayView< INDEX_TYPE const, 2, 1, INDEX_TYPE, DEFAULT_BUFFER > const &␣
↪elementToNodeMap,
          std::vector< std::vector< INDEX_TYPE > > & nodeToElementMap,
          INDEX_TYPE const numNodes )
{
  nodeToElementMap.resize( numNodes );

  for( INDEX_TYPE elementIndex = 0; elementIndex < elementToNodeMap.size( 0 );␣
↪++elementIndex )
```

(continues on next page)

(continued from previous page)

```
  {
    for( INDEX_TYPE const nodeIndex : elementToNodeMap[ elementIndex ] )
    {
      nodeToElementMap[ nodeIndex ].emplace_back( elementIndex );
    }
  }
}
```

[Source: benchmarks/benchmarkArrayOfArraysNodeToElementMapConstructionKernels.cpp]

For a mesh with `N` nodes this construction has complexity `O( N )` since there are `N` calls to `std::vector::emplace_back` each of which is `O( 1 )`. The same approach works when the node-to-element map is a `LvArray::ArrayOfArrays` however the complexity is much higher.

```
void NaiveNodeToElemMapConstruction::
  naive( ArrayView< INDEX_TYPE const, 2, 1, INDEX_TYPE, DEFAULT_BUFFER > const &␣
↪elementToNodeMap,
         ArrayOfArrays< INDEX_TYPE, INDEX_TYPE, DEFAULT_BUFFER > & nodeToElementMap,
         INDEX_TYPE const numNodes )
{
  nodeToElementMap.resize( numNodes );

  for( INDEX_TYPE elementIndex = 0; elementIndex < elementToNodeMap.size( 0 );␣
↪++elementIndex )
  {
    for( INDEX_TYPE const nodeIndex : elementToNodeMap[ elementIndex ] )
    {
      nodeToElementMap.emplaceBack( nodeIndex, elementIndex );
    }
  }
}
```

[Source: benchmarks/benchmarkArrayOfArraysNodeToElementMapConstructionKernels.cpp]

Since nothing is done to preallocate space for each inner array every call to `LvArray::ArrayOfArrays::emplaceBack` has complexity `O( M )` where `M` is the sum of the capacities of the inner arrays. `M` is proportional to the number of nodes in the mesh `N` so the entire algorithm runs in `O( N * N )`.

This can be sped up considerably with some preallocation.

```
template< typename POLICY >
void NodeToElemMapConstruction< POLICY >::
overAllocation( ArrayView< INDEX_TYPE const, 2, 1, INDEX_TYPE, DEFAULT_BUFFER > const␣
↪& elementToNodeMap,
                ArrayOfArrays< INDEX_TYPE, INDEX_TYPE, DEFAULT_BUFFER > &␣
↪nodeToElementMap,
                INDEX_TYPE const numNodes,
                INDEX_TYPE const maxNodeElements )
{
  using ATOMIC_POLICY = typename RAJAHelper< POLICY >::AtomicPolicy;

  // Resize the node to element map allocating space for each inner array.
  nodeToElementMap.resize( numNodes, maxNodeElements );

  // Create an ArrayOfArraysView
  ArrayOfArraysView< INDEX_TYPE, INDEX_TYPE const, false, DEFAULT_BUFFER > const␣
↪nodeToElementMapView =
```

(continues on next page)

(continued from previous page)

```
    nodeToElementMap.toView();

  // Launch a RAJA kernel that populates the ArrayOfArraysView.
  RAJA::forall< POLICY >(
    RAJA::TypedRangeSegment< INDEX_TYPE >( 0, elementToNodeMap.size( 0 ) ),
    [elementToNodeMap, nodeToElementMapView] ( INDEX_TYPE const elementIndex )
  {
    for( INDEX_TYPE const nodeIndex : elementToNodeMap[ elementIndex ] )
    {
      nodeToElementMapView.emplaceBackAtomic< ATOMIC_POLICY >( nodeIndex,␣
↪elementIndex );
    }
  }
    );
}
```

[Source: benchmarks/benchmarkArrayOfArraysNodeToElementMapConstructionKernels.cpp]

Since this method guarantees that the capacity of each inner arrays won't be exceeded the complexity is reduced back down to `O( N )`. In addition the loop appending to the inner arrays can be parallelized.

One problem with this approach is that it may allocate significantly more memory than is necessary to store the map since most of the inner arrays may not have the maximal length. Another potential issue is that the array is not compressed since some inner arrays will have a capacity that exceeds their size. Precomputing the size of each inner array (the number of elements each node is a part of) and using `resizeFromCapacities` solves both of these problems and is almost as fast as over allocating.

```
template< typename POLICY >
void NodeToElemMapConstruction< POLICY >::
resizeFromCapacities( ArrayView< INDEX_TYPE const, 2, 1, INDEX_TYPE, DEFAULT_BUFFER >␣
↪const & elementToNodeMap,
                      ArrayOfArrays< INDEX_TYPE, INDEX_TYPE, DEFAULT_BUFFER > &␣
↪nodeToElementMap,
                      INDEX_TYPE const numNodes )
{
  using ATOMIC_POLICY = typename RAJAHelper< POLICY >::AtomicPolicy;

  // Create an Array containing the size of each inner array.
  Array< INDEX_TYPE, 1, RAJA::PERM_I, INDEX_TYPE, DEFAULT_BUFFER > elementsPerNode(␣
↪numNodes );

  // Calculate the size of each inner array.
  RAJA::forall< POLICY >(
    RAJA::TypedRangeSegment< INDEX_TYPE >( 0, elementToNodeMap.size( 0 ) ),
    [elementToNodeMap, &elementsPerNode] ( INDEX_TYPE const elementIndex )
  {
    for( INDEX_TYPE const nodeIndex : elementToNodeMap[ elementIndex ] )
    {
      RAJA::atomicInc< ATOMIC_POLICY >( &elementsPerNode[ nodeIndex ] );
    }
  }
    );

  // Resize the node to element map with the inner array sizes.
  nodeToElementMap.resizeFromCapacities< POLICY >( elementsPerNode.size(),␣
↪elementsPerNode.data() );
```

(continues on next page)

(continued from previous page)

```
  // Create an ArrayOfArraysView
  ArrayOfArraysView< INDEX_TYPE, INDEX_TYPE const, false, DEFAULT_BUFFER > const␣
↪nodeToElementMapView =
    nodeToElementMap.toView();

  // Launch a RAJA kernel that populates the ArrayOfArraysView.
  RAJA::forall< POLICY >(
    RAJA::TypedRangeSegment< INDEX_TYPE >( 0, elementToNodeMap.size( 0 ) ),
    [elementToNodeMap, nodeToElementMapView] ( INDEX_TYPE const elementIndex )
  {
    for( INDEX_TYPE const nodeIndex : elementToNodeMap[ elementIndex ] )
    {
      nodeToElementMapView.emplaceBackAtomic< ATOMIC_POLICY >( nodeIndex,␣
↪elementIndex );
    }
  } );
}
```

[Source: benchmarks/benchmarkArrayOfArraysNodeToElementMapConstructionKernels.cpp]

The following timings are from a clang 10 release build on LLNL's Quartz system run with a `200 x 200 x 200` element structured mesh. The reported time is the best of ten iterations.

Function	RAJA Policy	Time
`vector`	N/A	0.99s
`overAllocation`	`loop_exec`	0.49s
`resizeFromCapacities`	`loop_exec`	0.58s
`overAllocation`	`omp_parallel_for_exec`	0.11s
`resizeFromCapacities`	`omp_parallel_for_exec`	0.17s

The `naive` method is much to slow to run on this size mesh. However on a `30 x 30 x 30` mesh it takes 1.28 seconds.

5.8 Doxygen

- LvArray::ArrayOfArrays
- LvArray::ArrayOfArraysView

CHAPTER 6

LvArray::ArrayOfSets

The LvArray::ArrayOfSets is very similar to the LvArray::ArrayOfArrays except that the values of the inner arrays are sorted an unique like the LvArray::SortedArray. If you are familiar with both of these classes the functionality of the LvArray::ArrayOfSets should be pretty straightforward.

6.1 Template arguments

The LvArray::ArrayOfArrays requires three template arguments.

1. T: The type of values stored in the inner arrays.
2. INDEX_TYPE: An integral type used in index calculations, the suggested type is std::ptrdiff_t.
3. BUFFER_TYPE: A template template parameter specifying the buffer type used for allocation and de-allocation, the LvArray::ArrayOfArrays contains a BUFFER_TYPE< T > along with two BUFFER_TYPE< INDEX_TYPE >.

6.2 Usage

All of the functionality for modifying the outer array from LvArray::ArrayOfArrays is present in the LvArray::ArrayOfSets although it might go by a different name. For example instead of appendArray there is appendSet. However like LvArray::SortedArray the only options for modifying the inner sets are through either insertIntoSet` or ``removeFromSet.

```
TEST( ArrayOfSets, examples )
{
  LvArray::ArrayOfSets< std::string, std::ptrdiff_t, LvArray::MallocBuffer >␣
↪arrayOfSets;

  // Append a set with capacity 2.
  arrayOfSets.appendSet( 2 );
```

(continues on next page)

(continued from previous page)

```
  arrayOfSets.insertIntoSet( 0, "oh" );
  arrayOfSets.insertIntoSet( 0, "my" );

  // Insert a set at the beginning with capacity 3.
  arrayOfSets.insertSet( 0, 3 );
  arrayOfSets.insertIntoSet( 0, "lions" );
  arrayOfSets.insertIntoSet( 0, "tigers" );
  arrayOfSets.insertIntoSet( 0, "bears" );

  // "tigers" is already in the set.
  EXPECT_FALSE( arrayOfSets.insertIntoSet( 0, "tigers" ) );

  EXPECT_EQ( arrayOfSets( 0, 0 ), "bears" );
  EXPECT_EQ( arrayOfSets( 0, 1 ), "lions" );
  EXPECT_EQ( arrayOfSets( 0, 2 ), "tigers" );

  EXPECT_EQ( arrayOfSets[ 1 ][ 0 ], "my" );
  EXPECT_EQ( arrayOfSets[ 1 ][ 1 ], "oh" );
}
```

[Source: examples/exampleArrayOfSets.cpp]

`LvArray::ArrayOfSets` also has a method `assimilate` which takes an rvalue-reference to an `LvArray::ArrayOfArrays` and converts it to a `LvArray::ArrayOfSets`. `LvArray::ArrayOfArrays` also has a similar method.

```
TEST( ArrayOfSets, assimilate )
{
  LvArray::ArrayOfSets< int, std::ptrdiff_t, LvArray::MallocBuffer > arrayOfSets;

  // Create an ArrayOfArrays and populate the inner arrays with sorted unique values.
  LvArray::ArrayOfArrays< int, std::ptrdiff_t, LvArray::MallocBuffer > arrayOfArrays(␣
↪3 );

  // The first array is empty, the second is {0} and the third is {0, 1}.
  arrayOfArrays.emplaceBack( 1, 0 );
  arrayOfArrays.emplaceBack( 2, 0 );
  arrayOfArrays.emplaceBack( 2, 1 );

  // Assimilate arrayOfArrays into arrayOfSets.
  arrayOfSets.assimilate< RAJA::loop_exec >( std::move( arrayOfArrays ),
                                             ␣
↪LvArray::sortedArrayManipulation::Description::SORTED_UNIQUE );

  // After being assimilated arrayOfArrays is empty.
  EXPECT_EQ( arrayOfArrays.size(), 0 );

  // arrayOfSets now contains the values.
  EXPECT_EQ( arrayOfSets.size(), 3 );
  EXPECT_EQ( arrayOfSets.sizeOfSet( 0 ), 0 );
  EXPECT_EQ( arrayOfSets.sizeOfSet( 1 ), 1 );
  EXPECT_EQ( arrayOfSets.sizeOfSet( 2 ), 2 );
  EXPECT_EQ( arrayOfSets( 1, 0 ), 0 );
  EXPECT_EQ( arrayOfSets( 2, 0 ), 0 );
  EXPECT_EQ( arrayOfSets( 2, 1 ), 1 );

  // Resize arrayOfArrays and populate it the inner arrays with values that are␣
↪neither sorted nor unique.
```

(continues on next page)

(continued from previous page)

```
  arrayOfArrays.resize( 2 );

  // The first array is {4, -1} and the second is {4, 4}.
  arrayOfArrays.emplaceBack( 0, 3 );
  arrayOfArrays.emplaceBack( 0, -1 );
  arrayOfArrays.emplaceBack( 1, 4 );
  arrayOfArrays.emplaceBack( 1, 4 );

  // Assimilate the arrayOfArrays yet again.
  arrayOfSets.assimilate< RAJA::loop_exec >( std::move( arrayOfArrays ),
↪LvArray::sortedArrayManipulation::Description::UNSORTED_WITH_DUPLICATES );

  EXPECT_EQ( arrayOfSets.size(), 2 );
  EXPECT_EQ( arrayOfSets.sizeOfSet( 0 ), 2 );
  EXPECT_EQ( arrayOfSets.sizeOfSet( 1 ), 1 );
  EXPECT_EQ( arrayOfSets( 0, 0 ), -1 );
  EXPECT_EQ( arrayOfSets( 0, 1 ), 3 );
  EXPECT_EQ( arrayOfSets( 1, 0 ), 4 );
}
```

[Source: examples/exampleArrayOfSets.cpp]

6.3 LvArray::ArrayOfSetsView

The `LvArray::ArrayOfSetsView` is the view counterpart to `LvArray::ArrayOfSets`. Functionally it is very similar to the `LvArray::ArrayOfArraysView` in that it cannot modify the outer array but it can modify the inner sets as long as their capacities aren't exceeded. Unlike `LvArray::ArrayOfArraysView` however it doesn't have the `CONST_SIZES` template parameter. This is because if you can't change the size of the inner arrays then you also can't modify their values. Specifically an `LvArray::ArrayOfSetsView< T, INDEX_TYPE, BUFFER_TYPE >` contains the following buffers:

1. `BUFFER_TYPE< T > values`: Contains the values of each inner set.
2. `BUFFER-TYPE< std::conditional_t< std::is_const< T >::value, INDEX_TYPE const, INDEX_TYPE > >`: Of length `array.size()`, `sizes[ i ]` contains the size of the inner set `i`.
3. `BUFFER_TYPE< INDEX_TYPE > offsets`: Of length `array.size() + 1`, inner set `i` begins at `values[ offsets[ i ] ]` and has capacity `offsets[ i + 1 ] - offsets[ i ]`.

From an `LvArray::ArrayOfSets< T, INDEX_TYPE, BUFFER_TYPE >` you can get three view types by calling the following methods

- `toView()` returns an `LvArray::ArrayOfSetsView< T, INDEX_TYPE const, BUFFER_TYPE >`.
- `toViewConst()` returns an `LvArray::ArrayOfSetsView< T const, INDEX_TYPE const, BUFFER_TYPE >`.
- `toArrayOfArraysView()` returns an `LvArray::ArrayOfArraysView< T const, INDEX_TYPE const, true, BUFFER_TYPE >`.

6.4 Usage with `LvArray::ChaiBuffer`

The two types of `LvArray::ArrayOfSetsView` obtainable from an `LvArray::ArrayOfSets` act differently when moved to a new memory space.

- `LvArray::ArrayOfSetsView< T, INDEX_TYPE const, LvArray::ChaiBuffer >`, obtained by calling `toView()`. When it is moved to a new space the values are touched as well as the sizes. The offsets are not touched.
- `LvArray::ArrayOfSetsView< T const, INDEX_TYPE const, LvArray::ChaiBuffer >`, obtained by calling `toViewConst()`. None of the buffers are touched in the new space.

Calling the explicit `move` method with the touch parameter set to `true` on a view type has the behavior described above. However calling `move( MemorySpace::host )` on an `LvArray::ArrayOfSets` will also touch the offsets (if moving to the GPU the offsets aren't touched). This is the only way to touch the offsets so if an `LvArray::ArrayOfSets` was previously on the device then it must be explicitly moved and touched on the host before any modification to the offsets can safely take place.

6.5 Usage with `LVARRAY_BOUNDS_CHECK`

When `LVARRAY_BOUNDS_CHECK` is defined access via `operator[]` and `operator()` is checked. If an invalid access is detected the program is aborted. Methods such as `sizeOfArray`, `insertArray` and `emplace` are also checked. The values passed to `insertIntoSet` and `removeFromSet` are also checked to ensure they are sorted and contain no duplicates.

6.6 Guidelines

Like `LvArray::SortedArray` batch insertion and removal from an inner set is much faster than inserting or removing each value individually.

All the tips for efficiently constructing an `LvArray::ArrayOfArrays` apply to constructing an `LvArray::ArrayOfSets`. The main difference is that `LvArray::ArrayOfSets` doesn't support concurrent modification of an inner set. Often if the sorted-unique properties of the inner sets aren't used during construction it can be faster to first construct a `LvArray::ArrayOfArrays` where each inner array can contain duplicates and doesn't have to be sorted and then create the `LvArray::ArrayOfSets` via a call to `assimilate`.

6.7 Doxygen

- LvArray::ArrayOfSets
- LvArray::ArrayOfSetsView

CHAPTER 7

LvArray::SparsityPattern and LvArray::CRSMatrix

LvArray::SparsityPattern represents just the sparsity pattern of a matrix while the LvArray::CRSMatrix represents a sparse matrix. Both use a slightly modified version of the compressed row storage format. The modifications to the standard format are as follows:

1. The columns of each row are sorted.
2. Each row can have a capacity in addition to a size which means that the rows aren't necessarily adjacent in memory.

7.1 Template arguments

The LvArray::SparsityPattern has three template arguments

1. COL_TYPE: The integral type used to enumerate the columns of the matrix.
2. INDEX_TYPE: An integral type used in index calculations, the suggested type is std::ptrdiff_t.
3. BUFFER_TYPE: A template template parameter specifying the buffer type used for allocation and deallocation, the LvArray::SparsityPattern contains a BUFFER_TYPE< COL_TYPE > along with two BUFFER_TYPE< INDEX_TYPE >.

The LvArray::CRSMatrix adds an additional template argument T which is the type of the entries in the matrix. It also has an addition member of type BUFER_TYPE< T >.

7.2 Usage

The LvArray::SparsityPattern is just a LvArray::ArrayOfSets by a different name. The only functional difference is that it doesn't support inserting or removing rows from the matrix (inserting or removing inner sets). Both the LvArray::SparsityPattern and LvArray::CRSMatrix support insertNonZero and insertNonZeros for inserting entries into a row as well as removeNonZero and removeNonZeros for removing entries from a row. LvArray::CRSMatrix also supports various addToRow methods which will add to existing entries in a specific row.

It is worth noting that neither `LvArray::SparsityPattern` nor `LvArray::CRSMatrix` have an `operator()` or `operator[]`. Instead they both support `getColumns` which returns a `LvArray::ArraySlice< COL_TYPE const, 1, 0, INDEX_TYPE >` with the columns of the row and `LvArray::CRSMatrix` supports `getEntries` which returns a `LvArray::ArraySlice< T, 1, 0, INDEX_TYPE >` with the entries of the row.

```
TEST( CRSMatrix, examples )
{
  // Create a matrix with three rows and three columns.
  LvArray::CRSMatrix< double, int, std::ptrdiff_t, LvArray::MallocBuffer > matrix( 2, 3 );
  EXPECT_EQ( matrix.numRows(), 2 );
  EXPECT_EQ( matrix.numColumns(), 3 );

  // Insert two entries into the first row.
  int const row0Columns[ 2 ] = { 0, 2 };
  double const row0Values[ 2 ] = { 4, 3 };
  matrix.insertNonZeros( 0, row0Columns, row0Values, 2 );
  EXPECT_EQ( matrix.numNonZeros( 0 ), 2 );

  // Insert three entries into the second row.
  int const row1Columns[ 3 ] = { 0, 1, 2 };
  double const row1Values[ 3 ] = { 55, -1, 4 };
  matrix.insertNonZeros( 1, row1Columns, row1Values, 3 );
  EXPECT_EQ( matrix.numNonZeros( 1 ), 3 );

  // The entire matrix has five non zero entries.
  EXPECT_EQ( matrix.numNonZeros(), 5 );

  // Row 0 does not have an entry for column 1.
  EXPECT_TRUE( matrix.empty( 0, 1 ) );

  // Row 1 does have an entry for column 1.
  EXPECT_FALSE( matrix.empty( 1, 1 ) );

  LvArray::ArraySlice< int const, 1, 0, std::ptrdiff_t > columns = matrix.getColumns( 0 );
  LvArray::ArraySlice< double, 1, 0, std::ptrdiff_t > entries = matrix.getEntries( 0 );

  // Check the entries of the matrix.
  EXPECT_EQ( columns.size(), 2 );
  EXPECT_EQ( columns[ 0 ], row0Columns[ 0 ] );
  EXPECT_EQ( entries[ 0 ], row0Values[ 0 ] );

  EXPECT_EQ( columns[ 1 ], row0Columns[ 1 ] );
  entries[ 1 ] += 10;
  EXPECT_EQ( entries[ 1 ], 10 + row0Values[ 1 ] );
}
```

[Source: examples/exampleSparsityPatternAndCRSMatrix.cpp]

Third party packages such as MKL or cuSparse expect the rows of the CRS matrix to be adjacent in memory. Specifically they don't accept a pointer that gives the size of each row they only accept an offsets pointer and calculate the sizes from that. To make an `LvArray::CRSMatrix` conform to this layout you can call `compress` which will set the capacity of each row equal to its size making the rows adjacent in memory.

```
TEST( CRSMatrix, compress )
{
  // Create a matrix with two rows and four columns where each row has capacity 3.
  LvArray::CRSMatrix< double, int, std::ptrdiff_t, LvArray::MallocBuffer > matrix( 2,␣
↪4, 3 );

  // Insert two entries into the first row.
  int const row0Columns[ 2 ] = { 0, 2 };
  double const row0Values[ 2 ] = { 4, 3 };
  matrix.insertNonZeros( 0, row0Columns, row0Values, 2 );
  EXPECT_EQ( matrix.numNonZeros( 0 ), 2 );
  EXPECT_EQ( matrix.nonZeroCapacity( 0 ), 3 );

  // Insert two entries into the second row.
  int const row1Columns[ 3 ] = { 0, 1 };
  double const row1Values[ 3 ] = { 55, -1 };
  matrix.insertNonZeros( 1, row1Columns, row1Values, 2 );
  EXPECT_EQ( matrix.numNonZeros( 1 ), 2 );
  EXPECT_EQ( matrix.nonZeroCapacity( 1 ), 3 );

  // The rows are not adjacent in memory.
  EXPECT_NE( &matrix.getColumns( 0 )[ 0 ] + matrix.numNonZeros( 0 ), &matrix.
↪getColumns( 1 )[ 0 ] );
  EXPECT_NE( &matrix.getEntries( 0 )[ 0 ] + matrix.numNonZeros( 0 ), &matrix.
↪getEntries( 1 )[ 0 ] );

  // After compression the rows are adjacent in memroy.
  matrix.compress();
  EXPECT_EQ( &matrix.getColumns( 0 )[ 0 ] + matrix.numNonZeros( 0 ), &matrix.
↪getColumns( 1 )[ 0 ] );
  EXPECT_EQ( &matrix.getEntries( 0 )[ 0 ] + matrix.numNonZeros( 0 ), &matrix.
↪getEntries( 1 )[ 0 ] );
  EXPECT_EQ( matrix.numNonZeros( 0 ), matrix.nonZeroCapacity( 0 ) );
  EXPECT_EQ( matrix.numNonZeros( 1 ), matrix.nonZeroCapacity( 1 ) );

  // The entries in the matrix are unchanged.
  EXPECT_EQ( matrix.getColumns( 0 )[ 0 ], row0Columns[ 0 ] );
  EXPECT_EQ( matrix.getEntries( 0 )[ 0 ], row0Values[ 0 ] );
  EXPECT_EQ( matrix.getColumns( 0 )[ 1 ], row0Columns[ 1 ] );
  EXPECT_EQ( matrix.getEntries( 0 )[ 1 ], row0Values[ 1 ] );

  EXPECT_EQ( matrix.getColumns( 1 )[ 0 ], row1Columns[ 0 ] );
  EXPECT_EQ( matrix.getEntries( 1 )[ 0 ], row1Values[ 0 ] );
  EXPECT_EQ( matrix.getColumns( 1 )[ 1 ], row1Columns[ 1 ] );
  EXPECT_EQ( matrix.getEntries( 1 )[ 1 ], row1Values[ 1 ] );
}
```

[Source: examples/exampleSparsityPatternAndCRSMatrix.cpp]

`LvArray::CRSMatrix` also has an `assimilate` method which takes an r-values reference to a `LvArray::SparsityPattern` and converts it into a `LvArray::CRSMatrix`. It takes a RAJA execution policy as a template parameter.

```
TEST( CRSMatrix, assimilate )
{
  // Create a sparsity pattern with two rows and four columns where each row has␣
↪capacity 3.
```

(continues on next page)

(continued from previous page)

```
  LvArray::SparsityPattern< int, std::ptrdiff_t, LvArray::MallocBuffer > sparsity( 2,␣
↪4, 3 );

  // Insert two entries into the first row.
  int const row0Columns[ 2 ] = { 0, 2 };
  sparsity.insertNonZeros( 0, row0Columns, row0Columns + 2 );

  // Insert two entries into the second row.
  int const row1Columns[ 3 ] = { 0, 1 };
  sparsity.insertNonZeros( 1, row1Columns, row1Columns + 2 );

  // Create a matrix with the sparsity pattern
  LvArray::CRSMatrix< double, int, std::ptrdiff_t, LvArray::MallocBuffer > matrix;
  matrix.assimilate< RAJA::loop_exec >( std::move( sparsity ) );

  // The sparsity is empty after being assimilated.
  EXPECT_EQ( sparsity.numRows(), 0 );
  EXPECT_EQ( sparsity.numColumns(), 0 );

  // The matrix has the same shape as the sparsity pattern.
  EXPECT_EQ( matrix.numRows(), 2 );
  EXPECT_EQ( matrix.numColumns(), 4 );

  EXPECT_EQ( matrix.numNonZeros( 0 ), 2 );
  EXPECT_EQ( matrix.nonZeroCapacity( 0 ), 3 );

  EXPECT_EQ( matrix.numNonZeros( 1 ), 2 );
  EXPECT_EQ( matrix.nonZeroCapacity( 1 ), 3 );

  // The entries in the matrix are zero initialized.
  EXPECT_EQ( matrix.getColumns( 0 )[ 0 ], row0Columns[ 0 ] );
  EXPECT_EQ( matrix.getEntries( 0 )[ 0 ], 0 );
  EXPECT_EQ( matrix.getColumns( 0 )[ 1 ], row0Columns[ 1 ] );
  EXPECT_EQ( matrix.getEntries( 0 )[ 1 ], 0 );

  EXPECT_EQ( matrix.getColumns( 1 )[ 0 ], row1Columns[ 0 ] );
  EXPECT_EQ( matrix.getEntries( 1 )[ 0 ], 0 );
  EXPECT_EQ( matrix.getColumns( 1 )[ 1 ], row1Columns[ 1 ] );
  EXPECT_EQ( matrix.getEntries( 1 )[ 1 ], 0 );
}
```

[Source: examples/exampleSparsityPatternAndCRSMatrix.cpp]

7.3 `LvArray::CRSMatrixView`

The `LvArray::SparsityPatternView` and `LvArray::CRSMatrixView` are the view counterparts to `LvArray::SparsityPattern` and `LvArray::CRSMatrix`. They both have the same template arguments as their counterparts. The `LvArray::SparsityPatternView` behaves exactly like an `LvArray::ArrayOfSetsView`. From a `LvArray::CRSMatrix< T, COL_TYPE, INDEX_TYPE, BUFFER_TYPE >` there are four view types you can create

- `toView()` returns a `LvArray::CRSMatrixView< T, COL_TYPE, INDEX_TYPE const, BUFFER_TYPE >` which can modify the entries as well as insert and remove columns from the rows as long as the size of each row doesn't exceed its capacity.

- `toViewConstSizes()` returns a `LvArray::CRSMatrixView< T, COL_TYPE const, INDEX_TYPE const, BUFFER_TYPE >` which can modify existing values but cannot add or remove columns from the rows.
- `toViewConst()` returns a `LvArray::CRSMatrixView< T const, COL_TYPE const, INDEX_TYPE const, BUFFER_TYPE >` which provides read only access to both the columns and values of the rows.
- `toSparsityPatternView()` returns a `LvArray::SparsityPatternView< COL_TYPE const, INDEX_TYPE const, BUFFER_TYPE >` which provides read only access to the columns of the rows.

```
TEST( CRSMatrix, views )
{
  // Create a 100x100 tri-diagonal sparsity pattern.
  LvArray::SparsityPattern< int, std::ptrdiff_t, LvArray::MallocBuffer > sparsity(␣
↪100, 100, 3 );

  // Since enough space has been preallocated in each row we can insert into the␣
↪sparsity
  // pattern in parallel.
  LvArray::SparsityPatternView< int,
                                std::ptrdiff_t const,
                                LvArray::MallocBuffer > const sparsityView = sparsity.
↪toView();

  RAJA::forall< RAJA::omp_parallel_for_exec >(
    RAJA::TypedRangeSegment< std::ptrdiff_t >( 0, sparsityView.numRows() ),
    [sparsityView] ( std::ptrdiff_t const row )
  {
    int const columns[ 3 ] = { int( row - 1 ), int( row ), int( row + 1 ) };
    int const begin = row == 0 ? 1 : 0;
    int const end = row == sparsityView.numRows() - 1 ? 2 : 3;
    sparsityView.insertNonZeros( row, columns + begin, columns + end );
  }
    );

  // Create a matrix from the sparsity pattern
  LvArray::CRSMatrix< double, int, std::ptrdiff_t, LvArray::MallocBuffer > matrix;
  matrix.assimilate< RAJA::omp_parallel_for_exec >( std::move( sparsity ) );

  // Assemble into the matrix.
  LvArray::CRSMatrixView< double,
                          int const,
                          std::ptrdiff_t const,
                          LvArray::MallocBuffer > const matrixView = matrix.
↪toViewConstSizes();
  RAJA::forall< RAJA::omp_parallel_for_exec >(
    RAJA::TypedRangeSegment< std::ptrdiff_t >( 0, matrixView.numRows() ),
    [matrixView] ( std::ptrdiff_t const row )
  {
    // Some silly assembly where each diagonal entry ( r, r ) has a contribution to␣
↪the row above ( r-1, r )
    // The current row (r, r-1), (r, r), (r, r+1) and the row below (r+1, r).
    int const columns[ 3 ] = { int( row - 1 ), int( row ), int( row + 1 ) };
    double const contribution[ 3 ] = { double( row ), double( row ), double( row ) };

    // Contribution to the row above
```

(continues on next page)

(continued from previous page)

```
    if( row > 0 )
    {
      matrixView.addToRow< RAJA::builtin_atomic >( row - 1, columns + 1, contribution,
↪ 1 );
    }

    // Contribution to the current row
    int const begin = row == 0 ? 1 : 0;
    int const end = row == matrixView.numRows() - 1 ? 2 : 3;
    matrixView.addToRow< RAJA::builtin_atomic >( row, columns + begin, contribution,␣
↪end - begin );

    // Contribution to the row below
    if( row < matrixView.numRows() - 1 )
    {
      matrixView.addToRow< RAJA::builtin_atomic >( row + 1, columns + 1, contribution,
↪ 1 );
    }
  }
    );

  // Check every row except for the first and the last.
  for( std::ptrdiff_t row = 1; row < matrix.numRows() - 1; ++row )
  {
    EXPECT_EQ( matrix.numNonZeros( row ), 3 );

    LvArray::ArraySlice< int const, 1, 0, std::ptrdiff_t > const rowColumns = matrix.
↪getColumns( row );
    LvArray::ArraySlice< double const, 1, 0, std::ptrdiff_t > const rowEntries =␣
↪matrix.getEntries( row );

    for( std::ptrdiff_t i = 0; i < matrix.numNonZeros( row ); ++i )
    {
      // The first first entry is the sum of the contributions of the row above and␣
↪the current row.
      EXPECT_EQ( rowColumns[ 0 ], row - 1 );
      EXPECT_EQ( rowEntries[ 0 ], row + row - 1 );

      // The second entry is the from the current row alone.
      EXPECT_EQ( rowColumns[ 1 ], row );
      EXPECT_EQ( rowEntries[ 1 ], row );

      // The third entry is the sum of the contributions of the row below and the␣
↪current row.
      EXPECT_EQ( rowColumns[ 2 ], row + 1 );
      EXPECT_EQ( rowEntries[ 2 ], row + row + 1 );
    }
  }

  // Check the first row.
  EXPECT_EQ( matrix.numNonZeros( 0 ), 2 );
  EXPECT_EQ( matrix.getColumns( 0 )[ 0 ], 0 );
  EXPECT_EQ( matrix.getEntries( 0 )[ 0 ], 0 );
  EXPECT_EQ( matrix.getColumns( 0 )[ 1 ], 1 );
  EXPECT_EQ( matrix.getEntries( 0 )[ 1 ], 1 );

  // Check the last row.
```

(continues on next page)

(continued from previous page)

```
  EXPECT_EQ( matrix.numNonZeros( matrix.numRows() - 1 ), 2 );
  EXPECT_EQ( matrix.getColumns( matrix.numRows() - 1 )[ 0 ], matrix.numRows() - 2 );
  EXPECT_EQ( matrix.getEntries( matrix.numRows() - 1 )[ 0 ], matrix.numRows() - 1  +␣
↪matrix.numRows() - 2 );
  EXPECT_EQ( matrix.getColumns( matrix.numRows() - 1 )[ 1 ], matrix.numRows() - 1 );
  EXPECT_EQ( matrix.getEntries( matrix.numRows() - 1 )[ 1 ], matrix.numRows() - 1 );
}
```

[Source: examples/exampleSparsityPatternAndCRSMatrix.cpp]

7.4 Usage with `LvArray::ChaiBuffer`

The three types of `LvArray::CRSMatrixView` obtainable from an `LvArray::CRSMatrixs` all act differently when moved to a new memory space.

- `LvArray::CRSMatrixsView< T, COL_TYPE, INDEX_TYPE const, LvArray::ChaiBuffer >`, obtained by calling `toView()`. When it is moved to a new space the values, columns and sizes are all touched. The offsets are not touched.
- `LvArray::CRSMatrixsView< T, COL_TYPE const, INDEX_TYPE const, LvArray::ChaiBuffer >`, obtained by calling `toViewConstSizes()`. When it is moved to a new space the values are touched but the columns, sizes and offsets aren't.
- `LvArray::CRSMatrixsView< T const, COL_TYPE const, INDEX_TYPE const, LvArray::ChaiBuffer >`, obtained by calling `toViewConst()`. None of the buffers are touched in the new space.

Calling the explicit `move` method with the touch parameter set to `true` on a view type has the behavior described above. However calling `move( MemorySpace::host )` on an `LvArray::CRSMatrix` or `LvArray::SparsityPattern` will also touch the offsets (if moving to the GPU the offsets aren't touched). This is the only way to touch the offsets so if an `LvArray::CRSMatrix` was previously on the device then it must be explicitly moved and touched on the host before any modification to the offsets can safely take place.

7.5 Usage with `LVARRAY_BOUNDS_CHECK`

When `LVARRAY_BOUNDS_CHECK` is defined access all row and column access is checked. Methods which expect a sorted unique set of columns check that the columns are indeed sorted and unique. In addition if `addToRow` checks that all the given columns are present in the row.

7.6 Guidelines

As with all the `LvArray` containers it is important to pass around the most restrictive form. A function should only accept a `LvArray::CRSMatrix` if it needs to resize the matrix or might bust the capacity of a row. If a function only needs to be able to modify existing entries it should accept a `LvArray::CRSMatrixView< T, COL_TYPE const, INDEX_TYPE const, BUFFER_TYPE >`. If a function only needs to examine the sparsity pattern of the matrix it should accept a `LvArray::SparsityPatternView< COL_TYPE const, INDEX_TYPE const, BUFFER_TYPE >`.

Like the `LvArray::ArrayOfArrays` and `LvArray::ArrayOfSets` when constructing an `LvArray::SparsityPattern` or `LvArray::CRSMatrix` it is important to preallocate space for each row in order to achieve decent performance.

A common pattern with sparse matrices is that the sparsity pattern need only be assembled once but the matrix is used multiple times with different values. For example when using the finite element method on an unstructured mesh you can generate the sparsity pattern once at the beginning of the simulation and then each time step you repopulate the entries of the matrix. When this is the case it is usually best to do the sparsity generation with a `LvArray::SparsityPattern` and then assimilate that into a `LvArray::CRSMatrix`. Once you have a matrix with the proper sparsity pattern create a `LvArray::CRSMatrixView< T, COL_TYPE const, INDEX_TYPE const, BUFFER_TYPE >` via `toViewConstSizes()` and you can then assemble into the matrix in parallel with the `addToRow` methods. Finally before beginning the next iteration you can zero out the entries in the matrix by calling `setValues`.

7.7 Doxygen

- LvArray::SparsityPattern
- LvArray::SparsityPatternView
- LvArray::CRSMatrix
- LvArray::CRSMatrixView

CHAPTER 8

LvArray::tensorOps

LvArray::tensorOps is a collection of free template functions which perform common linear algebra operations on compile time sized matrices and vectors. All of the functions work on device.

8.1 Object Representation

The LvArray::tensorOps methods currently operate on four different types of object; scalars, vectors, matrices, and symmetric matrices.

Vectors are represented by either a one dimensional c-array or a one dimensional LvArray::Array object. Examples of acceptable vector types are

- double[4]
- int[55]
- LvArray::Array< float, 1, camp::idx_seq< 0 >, std::ptrdiff_t, LvArray::MallocBuffer >
- LvArray::ArrayView< long, 1, 0, std::ptrdiff_t, LvArray::MallocBuffer >
- LvArray::ArraySlice< double, 1, 0, std::ptrdiff_t >
- LvArray::ArraySlice< double, 1, -1, std::ptrdiff_t >

Matrices are represented by either a two dimensional c-array or a two dimensional LvArray::Array object. Examples of acceptable matrix types

- double[3][3]
- int[5][2]
- LvArray::Array< float, 2, camp::idx_seq< 0, 1 >, std::ptrdiff_t, LvArray::MallocBuffer >
- LvArray::ArrayView< long, 2, 0, std::ptrdiff_t, LvArray::MallocBuffer >:
- LvArray::ArraySlice< double, 2, 1, std::ptrdiff_t >

- `LvArray::ArraySlice< double, 2, 0, std::ptrdiff_t >`
- `LvArray::ArraySlice< double, 2, -1, std::ptrdiff_t >`

Symmetric matrices are represented in Voigt notation as a vector. This means that a 2×2 symmetric matrix is represented as vector of length three as $[a_{00}, a_{11}, a_{01}]$ and that a 3×3 symmetric matrix is represented as vector of length six as $[a_{00}, a_{11}, a_{22}, a_{12}, a_{02}, a_{01}]$. One consequence of this is that you can use the vector methods to for example add one symmetric matrix to another.

8.2 Common operations

Variables	
α: A scalar.	
x: A vector in $\mathbb{R}^m \times \mathbb{R}^1$.	
y: A vector in $\mathbb{R}^m \times \mathbb{R}^1$.	
z: A vector in $\mathbb{R}^m \times \mathbb{R}^1$.	
$\mathbf{A}$: A matrix in $\mathbb{R}^m \times \mathbb{R}^n$.	
$\mathbf{B}$: A matrix in $\mathbb{R}^m \times \mathbb{R}^n$.	
Operation	Function
$x \leftarrow y$	`tensorOps::copy< m >( x, y )`
$\mathbf{A} \leftarrow \mathbf{B}$	`tensorOps::copy< m, n >( A, B )`
$x \leftarrow \alpha x$	`tensorOps::scale< m >( x, alpha )`
$\mathbf{A} \leftarrow \alpha \mathbf{A}$	`tensorOps::scale< m, n >( A, alpha )`
$x \leftarrow \alpha y$	`tensorOps::scaledCopy< m >( x, y, alpha )`
$\mathbf{A} \leftarrow \alpha \mathbf{B}$	`tensorOps::scaledCopy< m, n >( A, B, alpha )`
$x \leftarrow x + y$	`tensorOps::add< m >( x, y )`
$\mathbf{A} \leftarrow \mathbf{A} + \mathbf{B}$	`tensorOps::add< m, n >( A, B, alpha )`
$x \leftarrow x - y$	`tensorOps::subtract< m >( x, y )`
$x \leftarrow x + \alpha y$	`tensorOps::scaledAdd< m >( x, y, alpha )`
$x \leftarrow y \circ z$	`tensorOps::hadamardProduct< m >( x, y, z )`

There is also a function `fill` that will set all the entries of the object to a specific value. For a vector it is called as `tensorOps::fill< n >( x )` and for a matrix as `tensorOps::fill< m, n >( A )`.

8.3 Vector operations

Variables	
α: A scalar.	
x: A vector in $\mathbb{R}^m \times \mathbb{R}^1$.	
y: A vector in $\mathbb{R}^m \times \mathbb{R}^1$.	
z: A vector in $\mathbb{R}^m \times \mathbb{R}^1$.	
Operation	Function
$\|x\|_\infty$	`tensorOps::maxAbsoluteEntry< m >( x )`
$\|x\|_2^2$	`tensorOps::l2NormSquared< m >( x )`
$\|x\|_2$	`tensorOps::l2Norm< m >( x )`
$x \leftarrow \hat{x}$	`tensorOps::normalize< m >( x )`
$x^T y$	`tensorOps::AiBi( x, y )`

If x, y and z are all in $\mathbb{R}^3 \times \mathbb{R}^1$ then you can perform the operation $x \leftarrow y \times z$ as `tensorOps::crossProduct( x, y, z )`.

8.4 Matrix vector operations

Variables	
x: A vector in $\mathbb{R}^m \times \mathbb{R}^1$.	
y: A vector in $\mathbb{R}^n \times \mathbb{R}^1$.	
$\mathbf{A}$: A matrix in $\mathbb{R}^m \times \mathbb{R}^n$.	
Operation	Function
$\mathbf{A} \leftarrow xy^T$	`tensorOps::Rij_eq_AiBj< m, n >( A, x, y )`
$\mathbf{A} \leftarrow \mathbf{A} + xy^T$	`tensorOps::Rij_add_AiBj< m, n >( A, x, y )`
$x \leftarrow \mathbf{A}y$	`tensorOps::Ri_eq_AijBj< m, n >( x, A, y )`
$x \leftarrow x + \mathbf{A}y$	`tensorOps::Ri_add_AijBj< m, n >( x, A, y )`
$y \leftarrow \mathbf{A}^T x$	`tensorOps::Ri_eq_AjiBj< n, m >( y, A, x )`
$y \leftarrow y + \mathbf{A}^T x$	`tensorOps::Ri_add_AjiBj< n, m >( y, A, x )`

8.5 Matrix operations

Variables	
$\mathbf{A}$: A matrix in $\mathbb{R}^m \times \mathbb{R}^n$.	
$\mathbf{B}$: A matrix in $\mathbb{R}^m \times \mathbb{R}^p$.	
$\mathbf{C}$: A matrix in $\mathbb{R}^p \times \mathbb{R}^n$.	
$\mathbf{D}$: A matrix in $\mathbb{R}^n \times \mathbb{R}^m$.	
$\mathbf{E}$: A matrix in $\mathbb{R}^m \times \mathbb{R}^m$.	
Operation Function	
$\mathbf{A} \leftarrow \mathbf{D}^T$	`tensorOps::transpose< m, n >( A, D )`
$\mathbf{A} \leftarrow \mathbf{BC}$	`tensorOps::Rij_eq_AikBkj< m, n, p >( A, B, C )`
$\mathbf{A} \leftarrow \mathbf{A} + \mathbf{BC}$	`tensorOps::Rij_add_AikBkj< m, n, p >( A, B, C )`
$\mathbf{B} \leftarrow \mathbf{AC}^T$	`tensorOps::Rij_eq_AikBjk< m, p, n >( B, A, C )`
$\mathbf{B} \leftarrow \mathbf{B} + \mathbf{AC}^T$	`tensorOps::Rij_add_AikBjk< m, p, n >( B, A, C )`
$\mathbf{E} \leftarrow \mathbf{E} + \mathbf{AA}^T$	`tensorOps::Rij_add_AikAjk< m, n >( E, A )`
$\mathbf{C} \leftarrow \mathbf{B}^T\mathbf{A}$	`tensorOps::Rij_eq_AkiBkj< p, n, m >( C, B, A )`
$\mathbf{C} \leftarrow \mathbf{C} + \mathbf{B}^T\mathbf{A}$	`tensorOps::Rij_add_AkiBkj< p, n, m >( C, B, A )`

8.6 Square matrix operations

Variables	
α: A scalar.	
$\mathbf{A}$: A matrix in $\mathbb{R}^m \times \mathbb{R}^m$.	
$\mathbf{B}$: A matrix in $\mathbb{R}^m \times \mathbb{R}^m$.	
Operation	Function
$\mathbf{A} \leftarrow \mathbf{A}^T$	`tensorOps::transpose< m >( A )`
$\mathbf{A} \leftarrow \mathbf{A} + \alpha\mathbf{I}$	`tensorOps::tensorOps::addIdentity< m >( A, alpha )`
$tr(\mathbf{A})$	`tensorOps::trace< m >( A )`
$\|\mathbf{A}\|$	`tensorOps::determinant< m >( A )`
$\mathbf{A} \leftarrow \mathbf{B}^-1$	`tensorOps::invert< m >( A, B )`
$\mathbf{A} \leftarrow \mathbf{A}^-1$	`tensorOps::invert< m >( A )`

Note: Apart from `tensorOps::determinant` and `tensorOps::invert` are only implemented for matrices of size 2×2 and 3×3.

8.7 Symmetric matrix operations

Variables	
α: A scalar.	
x: A vector in $\mathbb{R}^m \times \mathbb{R}^1$.	
y: A vector in $\mathbb{R}^m \times \mathbb{R}^1$.	
$\mathbf{A}$: A matrix in $\mathbb{R}^m \times \mathbb{R}^m$.	
$\mathbf{B}$: A matrix in $\mathbb{R}^m \times \mathbb{R}^m$.	
$\mathbf{S}$: A symmetric matrix in $\mathbb{R}^m \times \mathbb{R}^m$.	
$\mathbf{Q}$: A symmetric matrix in $\mathbb{R}^m \times \mathbb{R}^m$.	
Operation	Function
$\mathbf{S} \leftarrow \mathbf{S} + \alpha\mathbf{I}$	`tensorOps::symAddIdentity< m >( S, alpha )`
$tr(\mathbf{S})$	`tensorOps::symTrace< m >( S )`
$x \leftarrow \mathbf{S}y$	`tensorOps::Ri_eq_symAijBj< m >( x, S ,y )`
$x \leftarrow x + \mathbf{S}y$	`tensorOps::Ri_add_symAijBj< m >( x, S ,y )`
$\mathbf{A} \leftarrow \mathbf{S}\mathbf{B}^T$	`tensorOps::Rij_eq_symAikBjk< m >( A, S, B )`
$\mathbf{S} \leftarrow \mathbf{A}\mathbf{Q}\mathbf{A}^T$	`tensorOps::Rij_eq_AikSymBklAjl< m >( S, A, Q )`
$\|\mathbf{S}\|$	`tensorOps::symDeterminant< m >( S )`
$\mathbf{S} \leftarrow \mathbf{Q}^-1$	`tensorOps::symInvert< m >( S, Q )`
$\mathbf{S} \leftarrow \mathbf{S}^-1$	`tensorOps::symInvert< m >( S )`
$x \leftarrow \mathbf{S}\mathbf{A}^T = diag(x)\mathbf{A}^T$	`tensorOps::symEigenvalues< M >( x, S )`
$x, \mathbf{A} \leftarrow \mathbf{S}\mathbf{A}^T = diag(x)\mathbf{A}^T$	`tensorOps::symEigenvectors< M >( x, S )`

There are also two function `tensorOps::denseToSymmetric` and `tensorOps::symmetricToDense` which convert between dense and symmetric matrix representation.

Note: Apart from `tensorOps::symAddIdentity` and `tensorOps::symTrace` the symmetric matrix operations are only implemented for matrices of size 2×2 and 3×3.

8.8 Examples

```
TEST( tensorOps, AiBi )
{
  double const x[ 3 ] = { 0, 1, 2 };
  double const y[ 3 ] = { 1, 2, 3 };

  // Works with c-arrays
  EXPECT_EQ( LvArray::tensorOps::AiBi< 3 >( x, y ), 8 );

  LvArray::Array< double,
                  1,
                  camp::idx_seq< 0 >,
                  std::ptrdiff_t,
                  LvArray::MallocBuffer > xArray( 3 );

  xArray( 0 ) = 0;
  xArray( 1 ) = 1;
  xArray( 2 ) = 2;

  // Works with LvArray::Array.
  EXPECT_EQ( LvArray::tensorOps::AiBi< 3 >( xArray, y ), 8 );

  // Works with LvArray::ArrayView.
  EXPECT_EQ( LvArray::tensorOps::AiBi< 3 >( xArray.toView(), y ), 8 );
  EXPECT_EQ( LvArray::tensorOps::AiBi< 3 >( xArray.toViewConst(), y ), 8 );

  // Create a 2D array with fortran layout.
  LvArray::Array< double,
                  2,
                  camp::idx_seq< 1, 0 >,
                  std::ptrdiff_t,
                  LvArray::MallocBuffer > yArray( 1, 3 );

  yArray( 0, 0 ) = 1;
  yArray( 0, 1 ) = 2;
  yArray( 0, 2 ) = 3;

  // Works with non contiguous slices.
  EXPECT_EQ( LvArray::tensorOps::AiBi< 3 >( x, yArray[ 0 ] ), 8 );
  EXPECT_EQ( LvArray::tensorOps::AiBi< 3 >( xArray, yArray[ 0 ] ), 8 );
}
```

[Source: examples/exampleTensorOps.cpp]

You can mix and match the data types of the objects and also call the `tensorOps` methods on device.

```
CUDA_TEST( tensorOps, device )
{
  // Create an array of 5 3x3 symmetric matrices.
  LvArray::Array< int,
                  2,
                  camp::idx_seq< 1, 0 >,
                  std::ptrdiff_t,
                  LvArray::ChaiBuffer > symmetricMatrices( 5, 6 );

  int offset = 0;
```

(continues on next page)

(continued from previous page)

```
  for( int & value : symmetricMatrices )
  {
    value = offset++;
  }

  LvArray::ArrayView< int const,
                      2,
                      0,
                      std::ptrdiff_t,
                      LvArray::ChaiBuffer > symmetricMatricesView = symmetricMatrices.
↪toViewConst();

  // The tensorOps methods work on device.
  RAJA::forall< RAJA::cuda_exec< 32 > >(
    RAJA::TypedRangeSegment< std::ptrdiff_t >( 0, symmetricMatricesView.size( 0 ) ),
    [symmetricMatricesView] __device__ ( std::ptrdiff_t const i )
  {
    double result[ 3 ];
    float x[ 3 ] = { 1.3f, 2.2f, 5.3f };

    // You can mix value types.
    LvArray::tensorOps::Ri_eq_symAijBj< 3 >( result, symmetricMatricesView[ i ], x );

    LVARRAY_ERROR_IF_NE( result[ 0 ], x[ 0 ] * symmetricMatricesView( i, 0 ) +
                         x[ 1 ] * symmetricMatricesView( i, 5 ) +
                         x[ 2 ] * symmetricMatricesView( i, 4 ) );
  }
    );
}
```

[Source: examples/exampleTensorOps.cpp]

8.9 Bounds checking

Whatever the argument type the number of dimensions is checked at compile time. For example if you pass a `double[ 3 ][ 3 ]` or a three dimensional `LvArray::ArraySlice` to `LvArray::tensorOps::crossProduct` you will get a compilation error since that function is only implemented for vectors. When passing a c-array as an argument the size of the array is checked at compile time. For example if you pass `int[ 2 ][ 3 ]` to `LvArray::tensorOps::addIdentity` you will get a compilation error because that function only operates on square matrices. However when passing an `LvArray::Array*` object the size is only checked at runtime if `LVARRAY_BOUNDS_CHECK` is defined.

```
TEST( tensorOps, boundsCheck )
{
  double x[ 3 ] = { 0, 1, 2 };
  LvArray::tensorOps::normalize< 3 >( x );

  // This would fail to compile since x is not length 4.
  // LvArray::tensorOps::normalize< 4 >( x );

  LvArray::Array< double,
                  1,
                  camp::idx_seq< 0 >,
                  std::ptrdiff_t,
```

(continues on next page)

(continued from previous page)

```
                  LvArray::MallocBuffer > xArray( 8 );

  xArray( 0 ) = 10;

#if defined(LVARRAY_BOUNDS_CHECK)
  // This will fail at runtime.
  EXPECT_DEATH_IF_SUPPORTED( LvArray::tensorOps::normalize< 3 >( xArray ), "" );
#endif

  int matrix[ 2 ][ 2 ] = { { 1, 0 }, { 0, 1 } };
  LvArray::tensorOps::normalize< 2 >( matrix[ 0 ] );

  // This would fail to compile since normalize expects a vector
  // LvArray::tensorOps::normalize< 4 >( matrix );

  LvArray::Array< double,
                  2,
                  camp::idx_seq< 0, 1 >,
                  std::ptrdiff_t,
                  LvArray::MallocBuffer > matrixArray( 2, 2 );

  matrixArray( 0, 0 ) = 1;
  matrixArray( 0, 1 ) = 1;

  LvArray::tensorOps::normalize< 2 >( matrixArray[ 0 ] );

  // This will also fail to compile
  // LvArray::tensorOps::normalize< 2 >( matrixArray );
}
```

[Source: examples/exampleTensorOps.cpp]

8.10 Doxygen

- LvArray::tensorOps

CHAPTER 9

Extra Goodies (Coming soon)

Comming soon: documentation for the following

- SFINAE template helpers
- string utilities
- stack trace
- macros
- sorting
- printers
- portable basic math functions

CHAPTER 10

Development Aids (Coming soon)

Coming soon: documentation for

- `LvArray::arrayManipulation`
- `LvArray::sortedArrayManipulation`
- `LvArray::bufferManipulation`
- Creating a new buffer type

10.1 Debugging helpers

10.1.1 TotalView

10.1.2 GDB

`LvArray` comes with a collection of custom GDB pretty printers for buffer and array classes that simplify inspection of data in a debug session by adding extra member(s) to the object display that "views" the data in a convenient format recognizable by GDB and IDEs. This eliminates the need for users of these classes to understand the particular class structure and spend time obtaining and casting the actual data pointer to the right type.

Note: In order to allow GDB to load the pretty printer script (`scripts/gdb-printers.py`), the following lines must be added to `.gdbinit` (with the proper path substituted):

```
add-auto-load-safe-path /path/to/LvArray/
directory /path/to/LvArray/
```

The first line allows GDB to load python scripts located under `LvArray` source tree. The second line adds `LvArray` source tree to the source path used by GDB to lookup file names, which allows it to find the pretty printer script by relative path (which is how it is referenced from the compiled binary). The `.gdbinit` file may be located under the user's home directory or current working directory where GDB is invoked.

The following pretty printers are available:

- For buffer types (`StackBuffer`, `MallocBuffer` and `ChaiBuffer`) the extra member `gdb_view` is a C-array of the appropriate type and size equal to the buffer size.
- For multidimensional arrays and slices (`ArraySlice`, `ArrayView` and `Array`) the extra member `gdb_view` is a multidimensional C-array with the sizes equal to current runtime size of the array or slice. This only works correctly for arrays with default (unpermuted) layout.
- For `ArrayOfArrays` (and consequently all of its descendants) each contained sub-array is added as a separate child that is again a C-array of size equal to that of the sub-array.

Example below demonstrates the difference between pretty printer output and raw output:

```
(gdb) p dofNumber
$1 = const geos::arrayView1d & of size [10] = {gdb_view = {0, 5, 10, 15, 20, 25, 30,␣
↪35, 40, 45}}
(gdb) p /r dofNumber
$2 = (const geos::arrayView1d &) @0x7ffcda2a8ab0: {static NDIM = 1, static USD = 0, m_
↪dims = {data = {10}}, m_strides = {data = {1}}, m_dataBuffer = {static␣
↪hasShallowCopy = <optimized out>, m_pointer = 0x55bec1de4860, m_capacity = 10, m_
↪pointerRecord = 0x55bec1dfa5c0}, m_singleParameterResizeIndex = 0}
```

This is how the variable is viewed in a debugging session in CLion IDE:

```
dofNumber = {const geos::arrayView1d &}
 gdb_view = {const long long [10]}
  [0] = {const long long} 0
  [1] = {const long long} 5
  [2] = {const long long} 10
  [3] = {const long long} 15
  [4] = {const long long} 20
  [5] = {const long long} 25
  [6] = {const long long} 30
  [7] = {const long long} 35
  [8] = {const long long} 40
  [9] = {const long long} 45
 NDIM = {const int} 1
 USD = {const int} 0
 m_dims = {LvArray::typeManipulation::CArray<long, 1>}
 m_strides = {LvArray::typeManipulation::CArray<long, 1>}
 m_dataBuffer = {LvArray::ChaiBuffer<long long const>}
 m_singleParameterResizeIndex = {int} 0
```

Warning: GDB printers are for host-only data. Attempting to display an array or buffer whose active pointer is a device pointer may have a range of outcomes, from incorrect data being displayed, to debugger crashes. The printer script is yet to be tested with `cuda-gdb`.

CHAPTER 11

Testing

Testing is a crucial component of writing quality software and the nature LvArray lends itself nicely to unit tests.

11.1 Building and Running the Tests

Tests are built by default, to disable the tests set the CMake variable `ENABLE_TESTS` to `OFF`. The tests are output in the `tests` folder of the build directory.

To run all the tests run `make test` in the build directory. To run a specific set of tests that match the regular expression `REGEX` run `ctest -V -R REGEX`, to run just `testCRSMatrix` run `./tests/testCRSMatrix`. LvArray uses Google Test for the testing framework and each test accepts a number of command line arguments.

```
> ./tests/testCRSMatrix --help
This program contains tests written using Google Test. You can use the
following command line flags to control its behavior:

Test Selection:
  --gtest_list_tests
      List the names of all tests instead of running them. The name of
      TEST(Foo, Bar) is "Foo.Bar".
  --gtest_filter=POSTIVE_PATTERNS[-NEGATIVE_PATTERNS]
      Run only the tests whose name matches one of the positive patterns but
      none of the negative patterns. '?' matches any single character; '*'
      matches any substring; ':' separates two patterns.
  --gtest_also_run_disabled_tests
      Run all disabled tests too.

Test Execution:
  --gtest_repeat=[COUNT]
      Run the tests repeatedly; use a negative count to repeat forever.
  --gtest_shuffle
      Randomize tests' orders on every iteration.
  --gtest_random_seed=[NUMBER]
```

(continues on next page)

(continued from previous page)

```
        Random number seed to use for shuffling test orders (between 1 and
        99999, or 0 to use a seed based on the current time).

  Test Output:
    --gtest_color=(yes|no|auto)
        Enable/disable colored output. The default is auto.
    --gtest_print_time=0
        Don't print the elapsed time of each test.
    --gtest_output=(json|xml)[:DIRECTORY_PATH/|:FILE_PATH]
        Generate a JSON or XML report in the given directory or with the given
        file name. FILE_PATH defaults to test_detail.xml.
    --gtest_stream_result_to=HOST:PORT
        Stream test results to the given server.

  Assertion Behavior:
    --gtest_death_test_style=(fast|threadsafe)
        Set the default death test style.
    --gtest_break_on_failure
        Turn assertion failures into debugger break-points.
    --gtest_throw_on_failure
        Turn assertion failures into C++ exceptions for use by an external
        test framework.
    --gtest_catch_exceptions=0
        Do not report exceptions as test failures. Instead, allow them
        to crash the program or throw a pop-up (on Windows).

Except for --gtest_list_tests, you can alternatively set the corresponding
environment variable of a flag (all letters in upper-case). For example, to
disable colored text output, you can either specify --gtest_color=no or set
the GTEST_COLOR environment variable to no.

For more information, please read the Google Test documentation at
https://github.com/google/googletest/. If you find a bug in Google Test
(not one in your own code or tests), please report it to
<googletestframework@googlegroups.com>.
```

The most useful of these is `gtest_filter` which lets you run a subset of tests in the file, this can be very useful when running a test through a debugger.

11.2 Test structure

The source for all the tests are all located in the `unitTests` directory, each tests consists of a `cpp` file whose name begins with `test` followed by the name of the class or namespace that is tested. For example the tests for `CRSMatrix` and `CRSMatrixView` are in `unitTests/testCRSMatrix.cpp` and the tests for `sortedArrayManipulation` are in `unitTests/testSortedArrayManipulation.cpp`.

Note: The tests for `LvArray::Array`, `LvArray::ArrayView` and `LvArray::tensorOps` are spread across multiple `cpp` files in order to speed up compilation on multithreaded systems.

11.3 Adding a New Test

Any time new functionality is added it should be tested. Before writing any test code it is highly recommended you familiarize yourself with the Google Test framework, see the Google Test primer and Google Test advanced documentation.

As an example say you add a new class `Foo`

```
class Foo
{
public:
  Foo( int const x ):
    m_x( x )
  {}

  int get() const
  { return m_x; }

  void set( int const x )
  { m_x = x; }

private:
  int m_x;
};
```

[Source: examples/Foo.hpp]

You'll also want to create a file `unitTests/testFoo.cpp` and add it to `unitTests/CMakeLists.txt`. A basic set of tests might look something like this

```
TEST( Foo, get )
{
  Foo foo( 5 );
  EXPECT_EQ( foo.get(), 5 );
}

TEST( Foo, set )
{
  Foo foo( 3 );
  EXPECT_EQ( foo.get(), 3 );
  foo.set( 8 );
  EXPECT_EQ( foo.get(), 8 );
}
```

[Source: examples/exampleTestFoo.cpp]

Note: These tests aren't very thorough. They don't test any of the implicit constructors and operators that the compiler defines such as the copy constructor and the move assignment operator and `get` and `set` are only tested with a single value.

Now you decide you want to generalize `Foo` to support types other than `int` so you define the template class `FooTemplate`

```
template< typename T >
class FooTemplate
{
```

(continues on next page)

(continued from previous page)

```
public:
  FooTemplate( T const & x ):
    m_x( x )
  {}

  T const & get() const
  { return m_x; }

  void set( T const & x )
  { m_x = x; }

private:
  T m_x;
};
```

[Source: examples/Foo.hpp]

Naturally you should test more than just a single instantiation of FooTemplate so you modify your tests as such

```
TEST( FooTemplate, get )
{
  {
    FooTemplate< int > foo( 5 );
    EXPECT_EQ( foo.get(), 5 );
  }

  {
    FooTemplate< double > foo( 5 );
    EXPECT_EQ( foo.get(), 5 );
  }
}

TEST( FooTemplate, set )
{
  {
    FooTemplate< int > foo( 3 );
    EXPECT_EQ( foo.get(), 3 );
    foo.set( 8 );
    EXPECT_EQ( foo.get(), 8 );
  }

  {
    FooTemplate< double > foo( 3 );
    EXPECT_EQ( foo.get(), 3 );
    foo.set( 8 );
    EXPECT_EQ( foo.get(), 8 );
  }
}
```

[Source: examples/exampleTestFoo.cpp]

In this example the code duplication isn't too bad because the tests are simple and only two instantiations are being tested. But using this style to write the tests for `LvArray::Array` which has five different template arguments would be unmaintainable. Luckily Google Test has an excellent solution: typed tests. Using typed tests the tests can be restructured as

```
template< typename T >
class FooTemplateTest : public ::testing::Test
{
public:
  void get()
  {
    FooTemplate< T > foo( 5 );
    EXPECT_EQ( foo.get(), 5 );
  }

  void set()
  {
    FooTemplate< T > foo( 3 );
    EXPECT_EQ( foo.get(), 3 );
    foo.set( 8 );
    EXPECT_EQ( foo.get(), 8 );
  }
};

using FooTemplateTestTypes = ::testing::Types<
  int
  , double
  >;

TYPED_TEST_SUITE( FooTemplateTest, FooTemplateTestTypes, );

TYPED_TEST( FooTemplateTest, get )
{
  this->get();
}

TYPED_TEST( FooTemplateTest, set )
{
  this->set();
}
```

[Source: examples/exampleTestFoo.cpp]

The benefits of using typed tests are many. In addition to the reduction in code duplication it makes it easy to run the tests associated with a single instantiation via `gtest_filter` and it lets you quickly add and remove types. Almost every test in LvArray is built using typed tests.

Note: When modifying a typed tests the compilation errors can be particularly painful to parse because usually an error in one instantiation means there will be errors in every instantiation. To decrease the verbosity you can simply limit the types used to instantiate the tests. For instance in the example above instead of testing both `int` and `double` comment out the `, double` and fix the `int` instantiation first.

One of the limitations of typed tests is that the class that gtest instantiates can only have a single template parameter and that parameter must be a type (not a value or a template). To get around this the type you pass in can be a `std::pair` or `std::tuple` when you need more than one type. For example the class `CRSMatrixViewTest` is defined as

```
template< typename CRS_MATRIX_POLICY_PAIR >
class CRSMatrixViewTest : public CRSMatrixTest< typename CRS_MATRIX_POLICY_
↪PAIR::first_type >
```

[Source: unitTests/testCRSMatrix.cpp]

where `CRS_MATRIX_POLICY_PAIR` is intended to be a `std::pair` where the first type is the `CRSMatrix` type to test and the second type is the RAJA policy to use. It is instantiated as follows

```
using CRSMatrixViewTestTypes = ::testing::Types<
  std::pair< CRSMatrix< int, int, std::ptrdiff_t, MallocBuffer >, serialPolicy >
```

[Source: unitTests/testCRSMatrix.cpp]

Another hurdle in writing typed tests is writing them in such a way that they compile for all the types. For example `FooTemplate< std::string >` is a perfectly valid instantiation but `FooTemplateTest< std::string >` is not because `FooTemplate< std::string > foo( 3 )` is invalid. You get an error like the following

```
../examples/exampleTestFoo.cpp:85:22: error: no matching constructor for␣
↪initialization of 'FooTemplate<std::basic_string<char> >'
  FooTemplate< T > foo( 5 );
                   ^    ~
```

However instantiating with `std::string` is very important for many LvArray classes because it behaves very differently from the built in types. For that reason `unitTests/testUtils.hpp` defines a class `TestString` which wraps a `std::string` and `Tensor` which wraps a `double[ 3 ]` both of which have constructors from integers.

11.4 Best practices

- Whenever possible use typed tests.
- Whenever possible do not write CUDA (or OpenMP) specific tests. Instead write tests a typed test that is templated on the RAJA policy and use a typed test to instantiate it with the appropriate policies.
- When linking to gtest it is not necessary to include the `main` function in the executable because if it is not there `gtest` will link in its own `main`. However you should include `main` in each test file to ease debugging. Furthermore if the executable needs some setup or cleanup such as initializing MPI it should be done in main. Note that while it is certainly possible to write tests which take command line arguments it is discouraged because then `./tests/testThatTakesCommandLineArguments` no longer works.
- For commonly called functions define a macro which first calls `SCOPED_TRACE` and then the the function. This helps illuminate exactly where errors are occurring.
- Prefer the `EXPECT_` family of macros to the `ASSERT_` family.
- Use the most specific `EXPECT_` macro applicable. So don't do `EXPECT_TRUE( bar() == 5 )` instead use `EXPECT_EQ( bar(), 5 )`

CHAPTER 12

Benchmarking (Coming soon)

Coming soon: documentation for the benchmarks.

CHAPTER 13

pylvarray — LvArray in Python

Many of the LvArray classes can be accessed and manipulated from Python. However, they cannot be created from Python.

Warning: The pylvarray module provides plenty of opportunites to crash Python. See the Segmentation Faults section below.

Only Python 3 is supported.

13.1 Module Constants

13.1.1 Space Constants

The following constants are used to set the space in which an LvArray object lives. The object pylvarray.GPU will only be defined if it is a valid space for the current system.

pylvarray.**CPU**

pylvarray.**GPU**

13.1.2 Permissions Constants

The following constants are used to set permissions for an array instance.

pylvarray.**READ_ONLY**
: No modification of the underlying data is allowed.

pylvarray.**MODIFIABLE**
: Allows Numpy views to be modified, but the object itself cannot be resized (or otherwise have its buffer reallocated, such as by inserting new elements).

pylvarray.**RESIZEABLE**
Allows Numpy views to be modified, and the object to be resized.

13.2 Module Classes

All of the objects documented below have an attribute, `dtype`, which returns the `numpy.dtype` of the object's data, and therefore the datatype of any Numpy view of the object.

13.2.1 Array and SortedArray

class pylvarray.**Array**
Represents an LvArray::Array, a multidimensional array.

get_single_parameter_resize_index()

set_single_parameter_resize_index(*dim*)
Set the dimension resized by a call to `resize()`.

resize(*new_size*)
Resize the array in the default dimension to `new_size`.

resize_all(*new_dims*)
Resize all the dimensions of the array in-place, discarding values.

to_numpy()
Return a Numpy view of the array.

set_access_level(*new_level*)
Set read/modify/resize permissions for the instance.

get_access_level()
Return the read/modify/resize permissions for the instance.

class pylvarray.**SortedArray**
Represents an LvArray::SortedArray, a one-dimensional sorted array.

to_numpy()
Return a read-only Numpy view of the array.

set_access_level(*new_level*)
Set read/modify/resize permissions for the instance.

get_access_level()
Return the read/modify/resize permissions for the instance.

insert(*values*)
Insert one or more values into the array. The object passed in will be converted to a 1D numpy array of the same dtype as the underlying instance, raising an exception if the conversion cannot be made safely.

remove(*values*)
Remove one or more values from the array. The object passed in will be converted to a 1D numpy array of the same dtype as the underlying instance, raising an exception if the conversion cannot be made safely.

13.2.2 ArrayOfArrays and ArrayOfSets

class pylvarray.**ArrayOfArrays**
Represents an LvArray::ArrayOfArrays, a two-dimensional ragged array.

Supports Python's sequence protocol, with the addition of deleting subarrays with `del arr[i]` syntax. An array fetched with `[]` is returned as a Numpy view. The built-in `len()` function will return the number of arrays in the instance. Iterating over an instance will yield a Numpy view of each array.

set_access_level(*new_level*)
Set read/modify/resize permissions for the instance.

get_access_level()
Return the read/modify/resize permissions for the instance.

insert(*index*, *values*)
Insert a new array consisting of `values` at the given index.

insert_into(*index*, *subindex*, *values*)
Insert `values` into the subarray given by `index` at position `subindex`. `values` will be converted to a 1D numpy array of the same dtype as the underlying instance, raising an exception if the conversion cannot be made safely.

erase_from(*index*, *subindex*)
Remove the value at `subindex` in the subarray `index`.

Conceptually equivalent to `del array_of_sets[index][subindex]`.

class pylvarray.**ArrayOfSets**
Represents an LvArray::ArrayOfSets, a collection of sets.

Behaves very similarly to the `ArrayOfArrays`, with differences outlined below.

set_access_level(*new_level*)
Set read/modify/resize permissions for the instance.

get_access_level()
Return the read/modify/resize permissions for the instance.

insert(*position*, *capacity=0*)
Insert a new set with a given capacity at `position`.

insert_into(*set_index*, *values*)
Insert values into a specific set.

`values` will be converted to a 1D numpy array of the same dtype as the underlying instance, raising an exception if the conversion cannot be made safely.

erase_from(*set_index*, *values*)
Remove values from a specific set.

`values` will be converted to a 1D numpy array of the same dtype as the underlying instance, raising an exception if the conversion cannot be made safely.

13.2.3 CRSMatrix

class pylvarray.**CRSMatrix**
Represents an LvArray::CRSMatrix, a sparse matrix.

to_scipy()
Return a scipy.sparse.csr_matrix representing the matrix.

Note that many methods of `scipy.sparse.csr_matrix` will, without raising an exception, generate deep copies of the LvArray::CRSMatrix's data. For instance, assigning a new value to an element in the `csr_matrix` may or may not modify the `CRSMatrix` data. Other `csr_matrix` methods will raise exceptions—for instance, when resizing. It is therefore in your best interest to be very careful about what

methods and operations you perform on the `crs_matrix`. To be safe, do not attempt to modify the matrix at all.

set_access_level(*new_level*)
Set read/modify/resize permissions for the instance.

get_access_level()
Return the read/modify/resize permissions for the instance.

num_rows()
Return the number of rows in the matrix.

num_columns()
Return the number of columns in the matrix.

get_entries(*row*)
Return a Numpy array representing the entries in the given row.

resize(*num_rows*, *num_cols*, *initial_row_capacity=0*)
Set the dimensions of the matrix, and the row capacity for any newly-created rows.

compress()
Compress the matrix.

insert_nonzeros(*row*, *columns*, *entries*)
Insert new nonzero entries to the matrix.

`columns` and `entries` should be iterables of equal length; both will be converted to Numpy arrays and a `TypeError` will be raised if the conversion cannot be made safely.

remove_nonzeros(*row*, *columns*)
Remove nonzero entries from the matrix.

`columns` should be an iterable identifying the columns of the given row to remove nonzero entries from. It will be converted to a Numpy array and a `TypeError` will be raised if the conversion cannot be made safely.

add_to_row(*row*, *columns*, *values*)
Add values to already-existing entries in a row.

`columns` and `values` should be iterables of equal length; both will be converted to Numpy arrays and a `TypeError` will be raised if the conversion cannot be made safely.

13.3 Segmentation Faults

Improper use of this module and associated programs can easily cause Python to crash. There are two main causes of crashes.

13.3.1 Stale Numpy Views

The `pylvarray` classes provide various ways to get Numpy views of their data. However, those views are only valid as long as the LvArray object's buffer is not reallocated. The buffer may be reallocated by invoking methods (the ones that require the `RESIZEABLE` permission) or by calls into a C++ program with access to the underlying C++ LvArray object.

```
view = my_array.to_numpy()
my_array.resize(1000)
print(view)  # segfault
```

13.3.2 Destroyed LvArray C++ objects

As mentioned earlier, the classes defined in this module cannot be created in Python; some external C++ program/library must create an LvArray object in C++, then create a `pylvarray` view of it. However, the Python view will only be valid as long as the underlying LvArray C++ object is kept around. If that is destroyed, the Python object will be left holding an invalid pointer and subsequent attempts to use the Python object will cause undefined behavior. The only way to avoid this is to know under what circumstances the external C++ program/library will destroy LvArray objects. To be safe, however, do not hold onto `pylvarray` object references after calling functions that have access to the underlying `LvArray` objects.

CHAPTER 14

Indices and tables

- genindex
- modindex
- search

CHAPTER 15

Doxygen

Doxygen

Python Module Index

p

Index

T