My Project

Generated by Doxygen 1.9.4

Chapter 1

MATAR

MATAR is a C++ library that addresses the need for simple, fast, and memory-efficient multi-dimensional data representations for dense and sparse storage that arise with numerical methods and in software applications. The data representations are designed to perform well across multiple computer architectures, including CPUs and GPUs. MATAR allows users to easily create and use intricate data representations that are also portable across disparate architectures using Kokkos. The performance aspect is achieved by forcing contiguous memory layout (or as close to contiguous as possible) for multi-dimensional and multi-size dense or sparse MATrix and ARray (hence, MATAR) types. Results show that MATAR has the capability to improve memory utilization, performance, and programmer productivity in scientific computing. This is achieved by fitting more work into the available memory, minimizing memory loads required, and by loading memory in the most efficient order.

1.1 Examples

- ELEMENTS: MATAR is a part of the ELEMENTS Library (LANL C# C20058) and it underpins the routines implemented in ELEMENTS. MATAR is available in a stand-alone directory outside of the ELEMENTS directory because it can aid many code applications. The dense and sparse storage types in MATAR are the foundation for the ELEMENTS library, which contains mathematical functions to support a very broad range of element types including: linear, quadratic, and cubic serendipity elements in 2D and 3D; high-order spectral elements; and a linear 4D element. An unstructured high-order mesh class is available in ELEMENTS and it takes advantage of MATAR for efficient access of various mesh entities.
- Fierro: The MATAR library underpins the Fierro code that is designed to simulate quasi-static solid mechanics problems and material dynamics problems.
- Simple examples are in the /test folder

1.2 Descriptions

- All Array MATAR types (e.g., CArray, ViewCArray, FArray, RaggedRightArray, etc.) start with an index of 0 and stop at an index of N-1, where N is the number of entries.
- All Matrix MATAR types (e.g., CMatrix, ViewCMatrix, FMatrix, etc.) start with an index of 1 and stop at an index of N, where N is the number of entries.

- The MATAR View types (e.g., ViewCArray, ViewCMatrix, ViewFArray, etc.) are designed to accept a pointer to an existing 1D array and then access that 1D data as a multi-dimensional array. The MATAR View types can also be used to slice an existing View.

- The C dense storage and View types (e.g., CArray, ViewCArray, CMatrix, etc.) access the data following the C/C++ language convection of having the last index in a multi-dimensional array vary the quickest. In a 2D CArray A, the index j in A(i,j) varies first followed by the index i, so the optimal performance is achieved using the following loop ordering.

```
// Optimal use of CArray
for (i=0,i<N,i++){
    for (j=0,j<N,j++){
        A(i,j) = 0.0;
    }
}
```

- The F dense storage and View types (e.g., FArray, ViewFArray, FMatrix, etc.) access the data following the Fortran language convection of having the first index in a multi-dimensional array vary the quickest. In a 2D FMatrix M, the index i in M(i,j) varies first followed by the index j, so the optimal performance is achieved using the following loop ordering.

```
// Optimal use of FMatrix
for (j=1,j<=N,j++){
    for (i=1,i<=N,i++){
        M(i,j) = 0.0;
    }
}
```

- The ragged data types (e.g., RaggedRightArray, RaggedDownArray, etc) in MATAR are special sparse storage types. The Right access types are for R(i,j) where the number of column entries varies in width across the array. The Down access types are for D(i,j) where the number of row entries vary in length across the array.

- The SparseRowArray MATAR type is the idetical to the Compressed Sparse Row (CSR) or Compressed Row Storage (CSR) respresentation.

- The SparseColumnArray MATAR type is identical to the Compressed Sparse Column (CSC) or Compressed Column Storage (CCS) respresentation.

1.3 Usage

```
// create a 1D array of integers and then access as a 2D array
int A[9];
auto A_array = ViewCArray <int> (A, 3, 3); // access as A(i,j)
// create a 3D array of doubles
auto B = CArray <double> (3,3,3); // access as B(i,j,k)
// create a slice of the 3D array at index 1
auto C = ViewCArray <double> (&B(1,0,0),3,3); // access as C(j,k)
// create a 4D matrix of doubles, indices start at 1
auto D = CMatrix <double> (10,9,8,7); // access as D(i,j,k,l)
// create a 2D view of a standard array
std::array<int, 9> E1d;
auto E = ViewCArray<int> (&E1d[0], 3, 3);
E(0,0) = 1;  // and so on
// create a ragged-right array of integers
//
// [1, 2, 3]
// [4, 5]
// [6]
// [7, 8, 9, 10]
//
size_t my_strides[4] = {3, 2, 1, 4};
RaggedRightArray <int> ragged(my_strides, 4);

int value = 1;
for (int i=0; i<4; i++){
    for (int j=0; j<my_ragged.stride(i); j++){
        ragged(i,j) = value;
        value++;
    }
}
```

1.4 Cloning the code

If your SSH keys are set in github, then from the terminal type:

```
git clone --recursive ssh://git@github.com/lanl/MATAR.git
```

1.5 Basic build

The basic build is for users only interested in the serial CPU only MATAR data types. For this build, we recommend making a folder perhaps called build then go into the build folder and type

```
cmake ..
make
```

The compiled code will be in the build folder.

1.6 Debug basic build

To build serial CPU only MATAR data types in the debug mode, please use

```
cmake -DCMAKE_BUILD_TYPE=Debug ..
make
```

The debug flag includes checks on array and matrix dimensions and index bounds.

1.7 Building MATAR with Kokkos

A suite of build scripts are provided to build MATAR with Kokkos for performance portability across computer architectures (CPUs and GPUs). The scripts for various Kokkos backends (e.g., CUDA, HIP, OpenMP, and pthreads) are located within the scripts folder. The provided scripts are configured for particular hardware, the user will likely need to alter the inputs to reflect their hardware. There are three scripts in each folder that are sourced to build MATAR with Kokkos. The scripts are

```
sourceme-env.sh
kokkos-install.sh
backend-cmake-build.sh
```

The word backend denotes cuda, hip, openMP, and so forth. Scripts are also provided to build MATAR without Kokkos, and in that case there is no backend listed since it doesn't use Kokko. The backend-cmake-build.sh script will run cmake and make for the project. Afterwords, the user can just runs make inside the respective build directory to compile the project. For clarity, running all the scripts is only necessary to set up and compile the code the first time, afterwards, the use can compile the code using make in the build directory. The environment variables will need to be set when logging into a compute node or when changing to a different kokkos backend. For all builds, a single script is provided in each script folder to automate the entire build process, it runs the three aforementioned scripts sequentially.

```
build-it.sh
```

Before using the build-it.sh script, the user must verify that the settings in the other scripts that build MATAR with a Kokkos backend are correctly set. After running the build-it.sh script, the entire project is compiled and stored in a directory that is named with the respective Kokkos backend e.g., build-kokkos-cuda. Further details are provided on the three scripts to configure and build MATAR with a Kokkos backend.

1.7.1 Environment configuration script

To start, the environment variables and modules must be configured by sourcing the following script
```
source sourceme-env.sh
```

This script is where the user will load the necessary module files for their given machine/architecture combination. This script also creates the build directory for the project e.g., build-kokkos-cuda, build-kokkos-hip, build-kokkos-openmp, etc.

1.7.2 Install Kokkos script

The next step is to install Kokkos, using the version that was cloned recursively within MATAR, and configure the Kokkos build for specific hardware and a backend.

```
source kokkos-install.sh
```

Within this script, the user will need to set any Kokkos specific variables for their project. The architecture variables will need to be modified based on the architecture being used. The provided scripts are set for a particular hardware that might differ from what a user might be using. CPU architecture information needs to be listed if running with the Kokkos serial or OpenMP backends; GPU architecture information must be listed if using a Kokkos GPU backend. We refer the user to Kokkos compiling page to see the large list of compilation options,
```
https://github.com/kokkos/kokkos/wiki/Compiling
```

1.7.3 CUDA compilation script

To build the project with cuda, the last step is to type
```
source cuda-cmake-build.sh
```

1.7.4 HIP compilation script

To build the project with hip, the last step is to type
```
source hip-cmake-build.sh
```

1.7.5 openMP compilation script

To build the project with openMP, the last step is to type
```
source openmp-cmake-build.sh
```

The sourceme-env.sh script (the first step) sets the number of threads to 16 by default. Changing the number of threads used with openMP requires manually setting the environment variable OMP_NUM_THREADS.

1.7.6 pthreads compilation script

To build the project with ptheads, the last step is to type
```
source pthreads-cmake-build.sh
```

To specify number of threads when running a code with the Kokkos pthread backend, add the following command line arguments
```
--kokkos-threads=4
```

1.7.7 Automate build process

A build-it.sh script is provided that runs all scripts sequentially for the user. The build-it.sh script obviates the need to manually source each script. The user must verify the settings are correct in each script prior to using the build-it.sh script. If the build-it.sh script fails to build the project correctly, the user should carefully look at the loaded modules and settings for building Kokkos.

1.8 Contributing

Pull requests are welcome. For major changes, please open an issue first to discuss what you would like to change.

1.9 License

This program is open source under the BSD-3 License.

1.10 Citation

```
@article{MATAR,
title = "{MATAR: A Performance Portability and Productivity Implementation of Data-Oriented Design with
       Kokkos}",
journal = {Journal of Parallel and Distributed Computing},
pages = {86-104},
volume = {157},
year = {2021},
author = {Daniel J. Dunning and Nathaniel R. Morgan and Jacob L. Moore and Eappen Nelluvelil and Tanya V.
       Tafolla and Robert W. Robey},
keywords = {Performance, Portability, Productivity, Memory Efficiency, GPUs, dense and sparse storage}
```

Chapter 2

Class Index

2.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 3

File Index

3.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 4

Class Documentation

4.1 CArray< T > Class Template Reference

Public Member Functions

- **CArray** (size_t dim0)
- **CArray** (size_t dim0, size_t dim1)
- **CArray** (size_t dim0, size_t dim1, size_t dim2)
- **CArray** (size_t dim0, size_t dim1, size_t dim2, size_t dim3)
- **CArray** (size_t dim0, size_t dim1, size_t dim2, size_t dim3, size_t dim4)
- **CArray** (size_t dim0, size_t dim1, size_t dim2, size_t dim3, size_t dim4, size_t dim5)
- **CArray** (size_t dim0, size_t dim1, size_t dim2, size_t dim3, size_t dim4, size_t dim5, size_t dim6)
- **CArray** (const CArray &temp)
- T & **operator()** (size_t i) const
- T & **operator()** (size_t i, size_t j) const
- T & **operator()** (size_t i, size_t j, size_t k) const
- T & **operator()** (size_t i, size_t j, size_t k, size_t l) const
- T & **operator()** (size_t i, size_t j, size_t k, size_t l, size_t m) const
- T & **operator()** (size_t i, size_t j, size_t k, size_t l, size_t m, size_t n) const
- T & **operator()** (size_t i, size_t j, size_t k, size_t l, size_t m, size_t n, size_t o) const
- CArray & **operator=** (const CArray &temp)
- size_t **size** () const
- size_t **dims** (size_t i) const
- size_t **order** () const
- T * **pointer** () const

The documentation for this class was generated from the following file:

- /Users/calvinroth/paraNotes/MATAR/src/matar.h

4.2 CMatrix< T > Class Template Reference

Public Member Functions

- **CMatrix** (size_t dim1)
- **CMatrix** (size_t dim1, size_t dim2)
- **CMatrix** (size_t dim1, size_t dim2, size_t dim3)
- **CMatrix** (size_t dim1, size_t dim2, size_t dim3, size_t dim4)
- **CMatrix** (size_t dim1, size_t dim2, size_t dim3, size_t dim4, size_t dim5)
- **CMatrix** (size_t dim1, size_t dim2, size_t dim3, size_t dim4, size_t dim5, size_t dim6)
- **CMatrix** (size_t dim1, size_t dim2, size_t dim3, size_t dim4, size_t dim5, size_t dim6, size_t dim7)
- **CMatrix** (const CMatrix &temp)
- T & **operator()** (size_t i) const
- T & **operator()** (size_t i, size_t j) const
- T & **operator()** (size_t i, size_t j, size_t k) const
- T & **operator()** (size_t i, size_t j, size_t k, size_t l) const
- T & **operator()** (size_t i, size_t j, size_t k, size_t l, size_t m) const
- T & **operator()** (size_t i, size_t j, size_t k, size_t l, size_t m, size_t n) const
- T & **operator()** (size_t i, size_t j, size_t k, size_t l, size_t m, size_t n, size_t o) const
- CMatrix & **operator=** (const CMatrix &temp)
- size_t **size** () const
- size_t **dims** (size_t i) const
- size_t **order** () const
- T * **pointer** () const

The documentation for this class was generated from the following file:

- /Users/calvinroth/paraNotes/MATAR/src/matar.h

4.3 CSCArray< T > Class Template Reference

Public Member Functions

- **CSCArray** (CArray< T > data, CArray< T > row_ptrs, CArray< T > row_pts, size_t rows, size_t cols)
- T & **operator()** (size_t i, size_t j) const
- void **printer** ()
- size_t **getNcols** ()
- size_t **getNrows** ()
- T * **begin** (size_t i)
- T * **end** (size_t i)
- size_t **beginFlat** (size_t i)
- size_t **endFlat** (size_t i)
- size_t **nnz** (size_t i)
- size_t **nnz** ()
- T & **getValFlat** (size_t k)
- size_t **getColFlat** (size_t k)
- int **flatIndex** (size_t i, size_t j)
- int **toCSR** (CArray< T > &data, CArray< size_t > &row_ptrs, CArray< size_t > &col_ptrs)
- void **todense** (CArray< T > &A)

The documentation for this class was generated from the following file:

- /Users/calvinroth/paraNotes/MATAR/src/matar.h

4.4 CSRArray< T > Class Template Reference

Public Member Functions

- **CSRArray** (CArray< T > data, CArray< T > col_ptrs, CArray< T > row_ptrs, size_t rows, size_t cols)
- T & **operator()** (size_t i, size_t j) const
- void **printer** ()
- size_t **getNcols** ()
- size_t **getNrows** ()
- T * **begin** (size_t i)
- T * **end** (size_t i)
- size_t **beginFlat** (size_t i)
- size_t **endFlat** (size_t i)
- size_t **nnz** (size_t i)
- size_t **nnz** ()
- T & **getValFlat** (size_t k)
- size_t **getColFlat** (size_t k)
- int **flatIndex** (size_t i, size_t j)
- int **toCSC** (CArray< T > &data, CArray< size_t > &col_ptrs, CArray< size_t > &row_ptrs)
- void **todense** (CArray< T > &A)

The documentation for this class was generated from the following file:

- /Users/calvinroth/paraNotes/MATAR/src/matar.h

4.5 DynamicRaggedDownArray< T > Class Template Reference

Public Member Functions

- **DynamicRaggedDownArray** (size_t dim1, size_t dim2)
- **DynamicRaggedDownArray** (const DynamicRaggedDownArray &temp)
- size_t & **stride** (size_t j) const
- size_t **size** () const
- T & **operator()** (size_t i, size_t j) const
- DynamicRaggedDownArray & **operator=** (const DynamicRaggedDownArray &temp)
- T * **pointer** () const

The documentation for this class was generated from the following file:

- /Users/calvinroth/paraNotes/MATAR/src/matar.h

4.6 DynamicRaggedRightArray< T > Class Template Reference

Public Member Functions

- **DynamicRaggedRightArray** (size_t dim1, size_t dim2)
- **DynamicRaggedRightArray** (const DynamicRaggedRightArray &temp)
- size_t & **stride** (size_t i) const
- size_t **size** () const
- T * **pointer** () const
- T & **operator()** (size_t i, size_t j) const
- DynamicRaggedRightArray & **operator=** (const DynamicRaggedRightArray &temp)

The documentation for this class was generated from the following file:

- /Users/calvinroth/paraNotes/MATAR/src/matar.h

4.7 FArray< T > Class Template Reference

Public Member Functions

- **FArray** (size_t dim0)
- **FArray** (size_t dim0, size_t dim1)
- **FArray** (size_t dim0, size_t dim1, size_t dim2)
- **FArray** (size_t dim0, size_t dim1, size_t dim2, size_t dim3)
- **FArray** (size_t dim0, size_t dim1, size_t dim2, size_t dim3, size_t dim4)
- **FArray** (size_t dim0, size_t dim1, size_t dim2, size_t dim3, size_t dim4, size_t dim5)
- **FArray** (size_t dim0, size_t dim1, size_t dim2, size_t dim3, size_t dim4, size_t dim5, size_t dim6)
- **FArray** (const FArray &temp)
- T & **operator()** (size_t i) const
- T & **operator()** (size_t i, size_t j) const
- T & **operator()** (size_t i, size_t j, size_t k) const
- T & **operator()** (size_t i, size_t j, size_t k, size_t l) const
- T & **operator()** (size_t i, size_t j, size_t k, size_t l, size_t m) const
- T & **operator()** (size_t i, size_t j, size_t k, size_t l, size_t m, size_t n) const
- T & **operator()** (size_t i, size_t j, size_t k, size_t l, size_t m, size_t n, size_t o) const
- FArray & **operator=** (const FArray &temp)
- size_t **size** () const
- size_t **dims** (size_t i) const
- size_t **order** () const
- T * **pointer** () const

The documentation for this class was generated from the following file:

- /Users/calvinroth/paraNotes/MATAR/src/matar.h

4.8 FMatrix< T > Class Template Reference

Public Member Functions

- **FMatrix** (size_t dim1)
- **FMatrix** (size_t dim1, size_t dim2)
- **FMatrix** (size_t dim1, size_t dim2, size_t dim3)
- **FMatrix** (size_t dim1, size_t dim2, size_t dim3, size_t dim4)
- **FMatrix** (size_t dim1, size_t dim2, size_t dim3, size_t dim4, size_t dim5)
- **FMatrix** (size_t dim1, size_t dim2, size_t dim3, size_t dim4, size_t dim5, size_t dim6)
- **FMatrix** (size_t dim1, size_t dim2, size_t dim3, size_t dim4, size_t dim5, size_t dim6, size_t dim7)
- **FMatrix** (const FMatrix &temp)
- T & **operator()** (size_t i) const
- T & **operator()** (size_t i, size_t j) const
- T & **operator()** (size_t i, size_t j, size_t k) const
- T & **operator()** (size_t i, size_t j, size_t k, size_t l) const
- T & **operator()** (size_t i, size_t j, size_t k, size_t l, size_t m) const
- T & **operator()** (size_t i, size_t j, size_t k, size_t l, size_t m, size_t n) const
- T & **operator()** (size_t i, size_t j, size_t k, size_t l, size_t m, size_t n, size_t o) const
- FMatrix & **operator=** (const FMatrix &temp)
- size_t **size** () const
- size_t **dims** (size_t i) const
- size_t **order** () const
- T * **pointer** () const

The documentation for this class was generated from the following file:

- /Users/calvinroth/paraNotes/MATAR/src/matar.h

4.9 RaggedDownArray< T > Class Template Reference

Public Member Functions

- **RaggedDownArray** (CArray< size_t > &strides_array)
- **RaggedDownArray** (ViewCArray< size_t > &strides_array)
- **RaggedDownArray** (size_t *strides_array, size_t dome_dim1)
- **RaggedDownArray** (size_t some_dim2, size_t buffer)
- **RaggedDownArray** (const RaggedDownArray &temp)
- size_t **stride** (size_t j)
- void **push_back** (size_t j)
- T & **operator()** (size_t i, size_t j)
- size_t **size** ()
- T * **pointer** () const
- size_t * **get_starts** () const
- RaggedDownArray & **operator=** (const RaggedDownArray &temp)

The documentation for this class was generated from the following file:

- /Users/calvinroth/paraNotes/MATAR/src/matar.h

4.10 RaggedRightArray< T > Class Template Reference

Public Member Functions

- **RaggedRightArray** (CArray< size_t > &strides_array)
- **RaggedRightArray** (ViewCArray< size_t > &strides_array)
- **RaggedRightArray** (size_t *strides_array, size_t some_dim1)
- **RaggedRightArray** (size_t some_dim1, size_t buffer)
- **RaggedRightArray** (const RaggedRightArray &temp)
- size_t **stride** (size_t i) const
- void **push_back** (size_t i)
- T & **operator()** (size_t i, size_t j) const
- size_t **size** () const
- T * **pointer** () const
- size_t * **get_starts** () const
- RaggedRightArray & **operator+=** (const size_t i)
- RaggedRightArray & **operator=** (const RaggedRightArray &temp)

The documentation for this class was generated from the following file:

- /Users/calvinroth/paraNotes/MATAR/src/matar.h

4.11 RaggedRightArrayofVectors< T > Class Template Reference

Public Member Functions

- **RaggedRightArrayofVectors** (CArray< size_t > &strides_array, size_t vector_dim)
- **RaggedRightArrayofVectors** (ViewCArray< size_t > &strides_array, size_t vector_dim)
- **RaggedRightArrayofVectors** (size_t *strides_array, size_t some_dim1, size_t vector_dim)
- **RaggedRightArrayofVectors** (size_t some_dim1, size_t buffer, size_t vector_dim)
- **RaggedRightArrayofVectors** (const RaggedRightArrayofVectors &temp)
- size_t **stride** (size_t i) const
- size_t **vector_dim** () const
- void **push_back** (size_t i)
- T & **operator()** (size_t i, size_t j, size_t k) const
- size_t **size** () const
- T * **pointer** () const
- size_t * **get_starts** () const
- RaggedRightArrayofVectors & **operator+=** (const size_t i)
- RaggedRightArrayofVectors & **operator=** (const RaggedRightArrayofVectors &temp)

The documentation for this class was generated from the following file:

- /Users/calvinroth/paraNotes/MATAR/src/matar.h

4.12 SparseColArray< T > Class Template Reference

Public Member Functions

- **SparseColArray** (CArray< size_t > &strides_array)
- **SparseColArray** (ViewCArray< size_t > &strides_array)
- **SparseColArray** (size_t *strides_array, size_t some_dim1)
- **SparseColArray** (const SparseColArray &temp)
- size_t **stride** (size_t j) const
- size_t & **row_index** (size_t i, size_t j) const
- T & **value** (size_t i, size_t j) const
- size_t **size** () const
- T * **pointer** () const
- size_t * **get_starts** () const
- SparseColArray & **operator=** (const SparseColArray &temp)

The documentation for this class was generated from the following file:

- /Users/calvinroth/paraNotes/MATAR/src/matar.h

4.13 SparseRowArray< T > Class Template Reference

Public Member Functions

- **SparseRowArray** (CArray< size_t > &strides_array)
- **SparseRowArray** (ViewCArray< size_t > &strides_array)
- **SparseRowArray** (size_t *strides_array, size_t some_dim1)
- **SparseRowArray** (const SparseRowArray &temp)
- size_t **stride** (size_t i) const
- size_t & **column_index** (size_t i, size_t j) const
- T & **value** (size_t i, size_t j) const
- size_t **size** () const
- T * **pointer** () const
- size_t * **get_starts** () const
- SparseRowArray & **operator=** (const SparseRowArray &temp)

The documentation for this class was generated from the following file:

- /Users/calvinroth/paraNotes/MATAR/src/matar.h

4.14 ViewCArray< T > Class Template Reference

Public Member Functions

- **ViewCArray** (T *array, size_t dim0)
- **ViewCArray** (T *array, size_t dim0, size_t dim1)
- **ViewCArray** (T *some_array, size_t dim0, size_t dim1, size_t dim2)
- **ViewCArray** (T *some_array, size_t dim0, size_t dim1, size_t dim2, size_t dim3)
- **ViewCArray** (T *some_array, size_t dim0, size_t dim1, size_t dim2, size_t dim3, size_t dim4)
- **ViewCArray** (T *some_array, size_t dim0, size_t dim1, size_t dim2, size_t dim3, size_t dim4, size_t dim5)
- **ViewCArray** (T *some_array, size_t dim0, size_t dim1, size_t dim2, size_t dim3, size_t dim4, size_t dim5, size_t dim6)
- T & **operator()** (size_t i) const
- T & **operator()** (size_t i, size_t j) const
- T & **operator()** (size_t i, size_t j, size_t k) const
- T & **operator()** (size_t i, size_t j, size_t k, size_t l) const
- T & **operator()** (size_t i, size_t j, size_t k, size_t l, size_t m) const
- T & **operator()** (size_t i, size_t j, size_t k, size_t l, size_t m, size_t n) const
- T & **operator()** (size_t i, size_t j, size_t k, size_t l, size_t m, size_t n, size_t o) const
- template<typename M >
  void **operator=** (M do_this_math)
- size_t **size** () const
- size_t **dims** (size_t i) const
- size_t **order** () const
- T * **pointer** () const

The documentation for this class was generated from the following file:

- /Users/calvinroth/paraNotes/MATAR/src/matar.h

4.15 ViewCMatrix< T > Class Template Reference

Public Member Functions

- **ViewCMatrix** (T *matrix, size_t dim1)
- **ViewCMatrix** (T *matrix, size_t dim1, size_t dim2)
- **ViewCMatrix** (T *matrix, size_t dim1, size_t dim2, size_t dim3)
- **ViewCMatrix** (T *matrix, size_t dim1, size_t dim2, size_t dim3, size_t dim4)
- **ViewCMatrix** (T *matrix, size_t dim1, size_t dim2, size_t dim3, size_t dim4, size_t dim5)
- **ViewCMatrix** (T *matrix, size_t dim1, size_t dim2, size_t dim3, size_t dim4, size_t dim5, size_t dim6)
- **ViewCMatrix** (T *matrix, size_t dim1, size_t dim2, size_t dim3, size_t dim4, size_t dim5, size_t dim6, size_t dim7)
- T & **operator()** (size_t i) const
- T & **operator()** (size_t i, size_t j) const
- T & **operator()** (size_t i, size_t j, size_t k) const
- T & **operator()** (size_t i, size_t j, size_t k, size_t l) const
- T & **operator()** (size_t i, size_t j, size_t k, size_t l, size_t m) const
- T & **operator()** (size_t i, size_t j, size_t k, size_t l, size_t m, size_t n) const
- T & **operator()** (size_t i, size_t j, size_t k, size_t l, size_t m, size_t n, size_t o) const
- template<typename M >
  void **operator=** (M do_this_math)
- size_t **size** () const
- size_t **dims** (size_t i) const
- size_t **order** () const
- T * **pointer** () const

The documentation for this class was generated from the following file:

- /Users/calvinroth/paraNotes/MATAR/src/matar.h

4.16 ViewFArray< T > Class Template Reference

Public Member Functions

- **ViewFArray** (T *array, size_t dim0)
- **ViewFArray** (T *array, size_t dim0, size_t dim1)
- **ViewFArray** (T *array, size_t dim0, size_t dim1, size_t dim2)
- **ViewFArray** (T *array, size_t dim0, size_t dim1, size_t dim2, size_t dim3)
- **ViewFArray** (T *array, size_t dim0, size_t dim1, size_t dim2, size_t dim3, size_t dim4)
- **ViewFArray** (T *array, size_t dim0, size_t dim1, size_t dim2, size_t dim3, size_t dim4, size_t dim5)
- **ViewFArray** (T *array, size_t dim0, size_t dim1, size_t dim2, size_t dim3, size_t dim4, size_t dim5, size_t dim6)
- T & **operator()** (size_t i) const
- T & **operator()** (size_t i, size_t j) const
- T & **operator()** (size_t i, size_t j, size_t k) const
- T & **operator()** (size_t i, size_t j, size_t k, size_t l) const
- T & **operator()** (size_t i, size_t j, size_t k, size_t l, size_t m) const
- T & **operator()** (size_t i, size_t j, size_t k, size_t l, size_t m, size_t n) const
- T & **operator()** (size_t i, size_t j, size_t k, size_t l, size_t m, size_t n, size_t o) const
- template<typename M >
  void **operator=** (M do_this_math)
- size_t **size** () const
- size_t **dims** (size_t i) const
- size_t **order** () const
- T * **pointer** () const

The documentation for this class was generated from the following file:

- /Users/calvinroth/paraNotes/MATAR/src/matar.h

4.17 ViewFMatrix< T > Class Template Reference

Public Member Functions

- **ViewFMatrix** (T *matrix, size_t dim1)
- **ViewFMatrix** (T *some_matrix, size_t dim1, size_t dim2)
- **ViewFMatrix** (T *matrix, size_t dim1, size_t dim2, size_t dim3)
- **ViewFMatrix** (T *matrix, size_t dim1, size_t dim2, size_t dim3, size_t dim4)
- **ViewFMatrix** (T *matrix, size_t dim1, size_t dim2, size_t dim3, size_t dim4, size_t dim5)
- **ViewFMatrix** (T *matrix, size_t dim1, size_t dim2, size_t dim3, size_t dim4, size_t dim5, size_t dim6)
- **ViewFMatrix** (T *matrix, size_t dim1, size_t dim2, size_t dim3, size_t dim4, size_t dim5, size_t dim6, size_t dim7)
- T & **operator()** (size_t i) const
- T & **operator()** (size_t i, size_t j) const
- T & **operator()** (size_t i, size_t j, size_t k) const
- T & **operator()** (size_t i, size_t j, size_t k, size_t l) const
- T & **operator()** (size_t i, size_t j, size_t k, size_t l, size_t m) const
- T & **operator()** (size_t i, size_t j, size_t k, size_t l, size_t m, size_t n) const
- T & **operator()** (size_t i, size_t j, size_t k, size_t l, size_t m, size_t n, size_t o) const
- template<typename M >
  void **operator=** (M do_this_math)
- size_t **size** () const
- size_t **dims** (size_t i) const
- size_t **order** () const
- T * **pointer** () const

The documentation for this class was generated from the following file:

- /Users/calvinroth/paraNotes/MATAR/src/matar.h

Chapter 5

File Documentation

5.1 macros.h

```
/**********************************************************************************************
 This file has suite of MACROS to build serial and parallel loops that are more readable and
 are written with the same syntax. The parallel loops use kokkos (i.e., the MACROS hide the
 complexity) and the serial loops are done using functions located in this file. The goal is to
 help users add kokkos to their code projects for performance portability across architectures.

 The loop order with the MACRO enforces the inner loop varies the fastest and the outer most
 loop varies the slowest.  Optiminal performance will be achieved by ensureing the loop indices
 align with the access pattern of the MATAR datatype.

  1.  The syntax to use the FOR_ALL MACRO is as follows:

  // parallelization over a single loop
  FOR_ALL(k, 0, 10,
          { loop contents is here });

  // parallellization over two loops
  FOR_ALL(m, 0, 3,
           n, 0, 3,
          { loop contents is here });

  // parallellization over two loops
  FOR_ALL(i, 0, 3,
           j, 0, 3,
           k, 0, 3,
          { loop contents is here });

  2.  The syntax to use the FOR_REDUCE is as follows:

  // reduce over a single loop
  REDUCE_SUM(i, 0, 100,
             local_answer,
              { loop contents is here }, answer);

  REDUCE_SUM(i, 0, 100,
             j, 0, 100,
             local_answer,
            { loop contents is here }, answer);

  REDUCE_SUM(i, 0, 100,
             j, 0, 100,
             k, 0, 100,
             local_answer,
            { loop contents is here }, answer);

  // other reduces are: RDUCE_MAX and REDUCE_MIN
  **********************************************************************************************/


#include <stdio.h>
#include <iostream>




// ----------------------------------------
// MACROS used with both Kokkos and non-kokkos versions
```

```
59 // ----------------------------------------
60 // a macro to select the name of a macro based on the number of inputs
61 #define \
62     GET_MACRO(_1, _2, _3, _4, _5, _6, _7, _8, _9, _10, _11, _12, NAME,...) NAME
63
64
65 // ----------------------------------------
66 // MACROS for kokkos
67 // ----------------------------------------
68
69 #ifdef HAVE_KOKKOS
70
71 // CArray nested loop convention use Right, use Left for outermost loop first
72 #define LOOP_ORDER Kokkos::Iterate::Right
73
74 // FArray nested loop convention use Right
75 #define F_LOOP_ORDER Kokkos::Iterate::Right
76
77
78 // run once on the device
79 #define \
80     RUN(fcn) \
81     Kokkos::parallel_for( Kokkos::RangePolicy<> ( 0, 1), \
82                           KOKKOS_LAMBDA(const int ijkabc){fcn} )
83
84 // run once on the device inside a class
85 #define \
86     RUN_CLASS(fcn) \
87     Kokkos::parallel_for( Kokkos::RangePolicy<> ( 0, 1), \
88                           KOKKOS_CLASS_LAMBDA(const int ijkabc){fcn} )
89
90
91 // the FOR_ALL loop
92 #define \
93     FOR1D(i, x0, x1,fcn) \
94     Kokkos::parallel_for( Kokkos::RangePolicy<> ( (x0), (x1)), \
95                           KOKKOS_LAMBDA( const int (i) ){fcn} )
96
97 #define \
98     FOR2D(i, x0, x1, j, y0, y1,fcn) \
99     Kokkos::parallel_for( \
100         Kokkos::MDRangePolicy< Kokkos::Rank<2,LOOP_ORDER,LOOP_ORDER> > ( {(x0), (y0)}, {(x1), (y1)} ), \
101         KOKKOS_LAMBDA( const int (i), const int (j) ){fcn} )
102
103 #define \
104     FOR3D(i, x0, x1, j, y0, y1, k, z0, z1, fcn) \
105     Kokkos::parallel_for( \
106          Kokkos::MDRangePolicy< Kokkos::Rank<3,LOOP_ORDER,LOOP_ORDER> > ( {(x0), (y0), (z0)}, {(x1),
    (y1), (z1)} ), \
107          KOKKOS_LAMBDA( const int (i), const int (j), const int (k) ) {fcn} )
108
109 #define \
110     FOR_ALL(...) \
111     GET_MACRO(__VA_ARGS__, _12, _11, FOR3D, _9, _8, FOR2D, _6, _5, FOR1D)(__VA_ARGS__)
112
113
114 // the DO_ALL loop
115 #define \
116     DO1D(i, x0, x1,fcn) \
117     Kokkos::parallel_for( Kokkos::RangePolicy<> ( (x0), (x1)+1), \
118                           KOKKOS_LAMBDA( const int (i) ){fcn} )
119
120 #define \
121     DO2D(i, x0, x1, j, y0, y1,fcn) \
122     Kokkos::parallel_for( \
123         Kokkos::MDRangePolicy< Kokkos::Rank<2,F_LOOP_ORDER, F_LOOP_ORDER> > ( {(x0), (y0)}, {(x1)+1,
    (y1)+1} ), \
124         KOKKOS_LAMBDA( const int (i), const int (j) ){fcn} )
125
126 #define \
127     DO3D(i, x0, x1, j, y0, y1, k, z0, z1, fcn) \
128     Kokkos::parallel_for( \
129          Kokkos::MDRangePolicy< Kokkos::Rank<3,F_LOOP_ORDER,F_LOOP_ORDER> > ( {(x0), (y0), (z0)},
    {(x1)+1, (y1)+1, (z1)+1} ), \
130          KOKKOS_LAMBDA( const int (i), const int (j), const int (k) ) {fcn} )
131
132 #define \
133     DO_ALL(...) \
134     GET_MACRO(__VA_ARGS__, _12, _11, DO3D, _9, _8, DO2D, _6, _5, DO1D)(__VA_ARGS__)
135
136
137 // the REDUCE SUM loop
138 #define \
139     RSUM1D(i, x0, x1, var, fcn, result) \
140     Kokkos::parallel_reduce( Kokkos::RangePolicy<> ( (x0), (x1) ),  \
141                              KOKKOS_LAMBDA(const int (i), decltype(var) &(var)){fcn}, (result))
142
```

```
143 #define \
144     RSUM2D(i, x0, x1, j, y0, y1, var, fcn, result) \
145     Kokkos::parallel_reduce( \
146         Kokkos::MDRangePolicy< Kokkos::Rank<2,LOOP_ORDER,LOOP_ORDER> > ( {(x0), (y0)}, {(x1), (y1)} ), \
147         KOKKOS_LAMBDA( const int (i),const int (j), decltype(var) &(var) ){fcn}, \
148            (result) )
149
150 #define \
151     RSUM3D(i, x0, x1, j, y0, y1, k, z0, z1, var, fcn, result) \
152     Kokkos::parallel_reduce( \
153         Kokkos::MDRangePolicy< Kokkos::Rank<3,LOOP_ORDER,LOOP_ORDER> > ( {(x0), (y0), (z0)}, {(x1),
    (y1), (z1)} ), \
154         KOKKOS_LAMBDA( const int (i), const int (j), const int (k), decltype(var) &(var) ){fcn}, \
155             (result) )
156
157 #define \
158     REDUCE_SUM(...) \
159     GET_MACRO(__VA_ARGS__, RSUM3D, _11, _10, RSUM2D, _8, _7, RSUM1D)(__VA_ARGS__)
160
161
162 // the DO_REDUCE_SUM loop
163 #define \
164     DO_RSUM1D(i, x0, x1, var, fcn, result) \
165     Kokkos::parallel_reduce( Kokkos::RangePolicy<> ( (x0), (x1)+1 ),  \
166                              KOKKOS_LAMBDA(const int (i), decltype(var) &(var)){fcn}, (result))
167
168 #define \
169     DO_RSUM2D(i, x0, x1, j, y0, y1, var, fcn, result) \
170     Kokkos::parallel_reduce( \
171         Kokkos::MDRangePolicy< Kokkos::Rank<2,F_LOOP_ORDER,F_LOOP_ORDER> > ( {(x0), (y0)}, {(x1)+1,
    (y1)+1} ), \
172         KOKKOS_LAMBDA( const int (i),const int (j), decltype(var) &(var) ){fcn}, \
173            (result) )
174
175 #define \
176     DO_RSUM3D(i, x0, x1, j, y0, y1, k, z0, z1, var, fcn, result) \
177     Kokkos::parallel_reduce( \
178         Kokkos::MDRangePolicy< Kokkos::Rank<3,F_LOOP_ORDER,F_LOOP_ORDER> > ( {(x0), (y0), (z0)},
    {(x1)+1, (y1)+1, (z1)+1} ), \
179         KOKKOS_LAMBDA( const int (i), const int (j), const int (k), decltype(var) &(var) ){fcn}, \
180             (result) )
181
182 #define \
183     DO_REDUCE_SUM(...) \
184     GET_MACRO(__VA_ARGS__, DO_RSUM3D, _11, _10, DO_RSUM2D, _8, _7, DO_RSUM1D)(__VA_ARGS__)
185
186
187 // the REDUCE MAX loop
188 #define \
189     RMAX1D(i, x0, x1, var, fcn, result) \
190     Kokkos::parallel_reduce( \
191                          Kokkos::RangePolicy<> ( (x0), (x1) ),  \
192                          KOKKOS_LAMBDA(const int (i), decltype(var) &(var)){fcn}, \
193                          Kokkos::Max< decltype(result) > ( (result) ) )
194
195 #define \
196     RMAX2D(i, x0, x1, j, y0, y1, var, fcn, result) \
197     Kokkos::parallel_reduce( \
198                          Kokkos::MDRangePolicy< Kokkos::Rank<2,LOOP_ORDER,LOOP_ORDER> > ( {(x0), (y0)},
    {(x1), (y1)} ), \
199                          KOKKOS_LAMBDA( const int (i),const int (j), decltype(var) &(var) ){fcn}, \
200                          Kokkos::Max< decltype(result) > ( (result) ) )
201
202 #define \
203     RMAX3D(i, x0, x1, j, y0, y1, k, z0, z1, var, fcn, result) \
204     Kokkos::parallel_reduce( \
205                          Kokkos::MDRangePolicy< Kokkos::Rank<3,LOOP_ORDER,LOOP_ORDER> > ( {(x0), (y0),
    (z0)}, {(x1), (y1), (z1)} ), \
206                          KOKKOS_LAMBDA( const int (i), const int (j), const int (k), decltype(var) &(var)
    ){fcn}, \
207                          Kokkos::Max< decltype(result) > ( (result) ) )
208
209 #define \
210     REDUCE_MAX(...) \
211     GET_MACRO(__VA_ARGS__, RMAX3D, _11, _10, RMAX2D, _8, _7, RMAX1D)(__VA_ARGS__)
212
213
214 // the DO_REDUCE_MAX loop
215 #define \
216     DO_RMAX1D(i, x0, x1, var, fcn, result) \
217     Kokkos::parallel_reduce( \
218                          Kokkos::RangePolicy<> ( (x0), (x1)+1 ),  \
219                          KOKKOS_LAMBDA(const int (i), decltype(var) &(var)){fcn}, \
220                          Kokkos::Max< decltype(result) > ( (result) ) )
221
222 #define \
223     DO_RMAX2D(i, x0, x1, j, y0, y1, var, fcn, result) \
```

```
224     Kokkos::parallel_reduce( \
225                         Kokkos::MDRangePolicy< Kokkos::Rank<2,F_LOOP_ORDER,F_LOOP_ORDER> > ( {(x0),
    (y0)}, {(x1)+1, (y1)+1} ), \
226                         KOKKOS_LAMBDA( const int (i),const int (j), decltype(var) &(var) ){fcn}, \
227                         Kokkos::Max< decltype(result) > ( (result) ) )
228
229 #define \
230     DO_RMAX3D(i, x0, x1, j, y0, y1, k, z0, z1, var, fcn, result) \
231     Kokkos::parallel_reduce( \
232                         Kokkos::MDRangePolicy< Kokkos::Rank<3,F_LOOP_ORDER,F_LOOP_ORDER> > ( {(x0),
    (y0), (z0)}, {(x1)+1, (y1)+1, (z1)+1} ), \
233                         KOKKOS_LAMBDA( const int (i), const int (j), const int (k), decltype(var) &(var)
    ){fcn}, \
234                         Kokkos::Max< decltype(result) > ( (result) ) )
235
236 #define \
237     DO_REDUCE_MAX(...) \
238     GET_MACRO(__VA_ARGS__, DO_RMAX3D, _11, _10, DO_RMAX2D, _8, _7, DO_RMAX1D)(__VA_ARGS__)
239
240
241
242 // the REDUCE MIN loop
243 #define \
244     RMIN1D(i, x0, x1, var, fcn, result) \
245     Kokkos::parallel_reduce( \
246                         Kokkos::RangePolicy<> ( (x0), (x1) ),  \
247                         KOKKOS_LAMBDA( const int (i), decltype(var) &(var) ){fcn}, \
248                         Kokkos::Min< decltype(result) >(result))
249
250 #define \
251     RMIN2D(i, x0, x1, j, y0, y1, var, fcn, result) \
252     Kokkos::parallel_reduce( \
253                         Kokkos::MDRangePolicy< Kokkos::Rank<2,LOOP_ORDER,LOOP_ORDER> > ( {(x0), (y0)},
    {(x1), (y1)} ), \
254                         KOKKOS_LAMBDA( const int (i),const int (j), decltype(var) &(var) ){fcn}, \
255                         Kokkos::Min< decltype(result) >(result) )
256
257 #define \
258     RMIN3D(i, x0, x1, j, y0, y1, k, z0, z1, var, fcn, result) \
259     Kokkos::parallel_reduce( \
260                         Kokkos::MDRangePolicy< Kokkos::Rank<3,LOOP_ORDER,LOOP_ORDER> > ( {(x0), (y0),
    (z0)}, {(x1), (y1), (z1)} ), \
261                         KOKKOS_LAMBDA( const int (i), const int (j), const int (k), decltype(var) &(var)
    ){fcn}, \
262                         Kokkos::Min< decltype(result) >(result) )
263
264 #define \
265     REDUCE_MIN(...) \
266     GET_MACRO(__VA_ARGS__, RMIN3D, _11, _10, RMIN2D, _8, _7, RMIN1D)(__VA_ARGS__)
267
268
269 // the DO_REDUCE MIN loop
270 #define \
271     DO_RMIN1D(i, x0, x1, var, fcn, result) \
272     Kokkos::parallel_reduce( \
273                         Kokkos::RangePolicy<> ( (x0), (x1)+1 ),  \
274                         KOKKOS_LAMBDA( const int (i), decltype(var) &(var) ){fcn}, \
275                         Kokkos::Min< decltype(result) >(result))
276
277 #define \
278     DO_RMIN2D(i, x0, x1, j, y0, y1, var, fcn, result) \
279     Kokkos::parallel_reduce( \
280                         Kokkos::MDRangePolicy< Kokkos::Rank<2,F_LOOP_ORDER,F_LOOP_ORDER> > ( {(x0),
    (y0)}, {(x1)+1, (y1)+1} ), \
281                         KOKKOS_LAMBDA( const int (i),const int (j), decltype(var) &(var) ){fcn}, \
282                         Kokkos::Min< decltype(result) >(result) )
283
284 #define \
285     DO_RMIN3D(i, x0, x1, j, y0, y1, k, z0, z1, var, fcn, result) \
286     Kokkos::parallel_reduce( \
287                         Kokkos::MDRangePolicy< Kokkos::Rank<3,F_LOOP_ORDER,F_LOOP_ORDER> > ( {(x0),
    (y0), (z0)}, {(x1)+1, (y1)+1, (z1)+1} ), \
288                         KOKKOS_LAMBDA( const int (i), const int (j), const int (k), decltype(var) &(var)
    ){fcn}, \
289                         Kokkos::Min< decltype(result) >(result) )
290
291 #define \
292     DO_REDUCE_MIN(...) \
293     GET_MACRO(__VA_ARGS__, DO_RMIN3D, _11, _10, DO_RMIN2D, _8, _7, DO_RMIN1D)(__VA_ARGS__)
294
295
296
297 // the FOR_ALL loop with variables in a class
298 #define \
299 FORCLASS1D(i, x0, x1,fcn) \
300 Kokkos::parallel_for( Kokkos::RangePolicy<> ( (x0), (x1)), \
301                       KOKKOS_CLASS_LAMBDA( const int (i) ){fcn} )
```

```
302
303 #define \
304 FORCLASS2D(i, x0, x1, j, y0, y1,fcn) \
305 Kokkos::parallel_for( \
306                       Kokkos::MDRangePolicy< Kokkos::Rank<2,LOOP_ORDER,LOOP_ORDER> > ( {(x0), (y0)},
    {(x1), (y1)} ), \
307                       KOKKOS_CLASS_LAMBDA( const int (i), const int (j) ){fcn} )
308
309 #define \
310 FORCLASS3D(i, x0, x1, j, y0, y1, k, z0, z1, fcn) \
311 Kokkos::parallel_for( \
312                       Kokkos::MDRangePolicy< Kokkos::Rank<3,LOOP_ORDER,LOOP_ORDER> > ( {(x0), (y0),
    (z0)}, {(x1), (y1), (z1)} ), \
313                       KOKKOS_CLASS_LAMBDA( const int (i), const int (j), const int (k) ) {fcn} )
314
315 #define \
316 FOR_ALL_CLASS(...) \
317 GET_MACRO(__VA_ARGS__, _12, _11, FORCLASS3D, _9, _8, FORCLASS2D, _6, _5, FORCLASS1D)(__VA_ARGS__)
318
319
320 // the REDUCE SUM loop
321 #define \
322 RSUMCLASS1D(i, x0, x1, var, fcn, result) \
323 Kokkos::parallel_reduce( Kokkos::RangePolicy<> ( (x0), (x1) ),  \
324                          KOKKOS_CLASS_LAMBDA(const int (i), decltype(var) &(var)){fcn}, (result))
325
326 #define \
327 RSUMCLASS2D(i, x0, x1, j, y0, y1, var, fcn, result) \
328 Kokkos::parallel_reduce( \
329                          Kokkos::MDRangePolicy< Kokkos::Rank<2,LOOP_ORDER,LOOP_ORDER> > ( {(x0), (y0)},
    {(x1), (y1)} ), \
330                          KOKKOS_CLASS_LAMBDA( const int (i),const int (j), decltype(var) &(var) ){fcn}, \
331                          (result) )
332
333 #define \
334 RSUMCLASS3D(i, x0, x1, j, y0, y1, k, z0, z1, var, fcn, result) \
335 Kokkos::parallel_reduce( \
336                          Kokkos::MDRangePolicy< Kokkos::Rank<3,LOOP_ORDER,LOOP_ORDER> > ( {(x0), (y0),
    (z0)}, {(x1), (y1), (z1)} ), \
337                          KOKKOS_CLASS_LAMBDA( const int (i), const int (j), const int (k), decltype(var)
    &(var) ){fcn}, \
338                          (result) )
339
340 #define \
341 REDUCE_SUM_CLASS(...) \
342 GET_MACRO(__VA_ARGS__, RSUMCLASS3D, _11, _10, RSUMCLASS2D, _8, _7, RSUMCLASS1D)(__VA_ARGS__)
343
344
345
346 // the REDUCE MAX loop with variables in a class
347
348 #define \
349 RMAXCLASS1D(i, x0, x1, var, fcn, result) \
350 Kokkos::parallel_reduce( \
351                          Kokkos::RangePolicy<> ( (x0), (x1) ),  \
352                          KOKKOS_CLASS_LAMBDA(const int (i), decltype(var) &(var)){fcn}, \
353                          Kokkos::Max< decltype(result) > ( (result) ) )
354
355 #define \
356 RMAXCLASS2D(i, x0, x1, j, y0, y1, var, fcn, result) \
357 Kokkos::parallel_reduce( \
358                          Kokkos::MDRangePolicy< Kokkos::Rank<2,LOOP_ORDER,LOOP_ORDER> > ( {(x0), (y0)},
    {(x1), (y1)} ), \
359                          KOKKOS_CLASS_LAMBDA( const int (i),const int (j), decltype(var) &(var) ){fcn}, \
360                          Kokkos::Max< decltype(result) > ( (result) ) )
361
362 #define \
363 RMAXCLASS3D(i, x0, x1, j, y0, y1, k, z0, z1, var, fcn, result) \
364 Kokkos::parallel_reduce( \
365                          Kokkos::MDRangePolicy< Kokkos::Rank<3,LOOP_ORDER,LOOP_ORDER> > ( {(x0), (y0),
    (z0)}, {(x1), (y1), (z1)} ), \
366                          KOKKOS_CLASS_LAMBDA( const int (i), const int (j), const int (k), decltype(var)
    &(var) ){fcn}, \
367                          Kokkos::Max< decltype(result) > ( (result) ) )
368
369 #define \
370 REDUCE_MAX_CLASS(...) \
371 GET_MACRO(__VA_ARGS__, RMAXCLASS3D, _11, _10, RMAXCLASS2D, _8, _7, RMAXCLASS1D)(__VA_ARGS__)
372
373
374 // the REDUCE MIN loop with variables in a class
375 #define \
376 RMINCLASS1D(i, x0, x1, var, fcn, result) \
377 Kokkos::parallel_reduce( \
378                          Kokkos::RangePolicy<> ( (x0), (x1) ),  \
379                          KOKKOS_CLASS_LAMBDA( const int (i), decltype(var) &(var) ){fcn}, \
380                          Kokkos::Min< decltype(result) >(result))
```

```
381
382 #define \
383 RMINCLASS2D(i, x0, x1, j, y0, y1, var, fcn, result) \
384 Kokkos::parallel_reduce( \
385                         Kokkos::MDRangePolicy< Kokkos::Rank<2,LOOP_ORDER,LOOP_ORDER> > ( {(x0), (y0)},
    {(x1), (y1)} ), \
386                         KOKKOS_CLASS_LAMBDA( const int (i),const int (j), decltype(var) &(var) ){fcn}, \
387                         Kokkos::Min< decltype(result) >(result) )
388
389 #define \
390 RMINCLASS3D(i, x0, x1, j, y0, y1, k, z0, z1, var, fcn, result) \
391 Kokkos::parallel_reduce( \
392                         Kokkos::MDRangePolicy< Kokkos::Rank<3,LOOP_ORDER,LOOP_ORDER> > ( {(x0), (y0),
    (z0)}, {(x1), (y1), (z1)} ), \
393                         KOKKOS_CLASS_LAMBDA( const int (i), const int (j), const int (k), decltype(var)
    &(var) ){fcn}, \
394                         Kokkos::Min< decltype(result) >(result) )
395
396 #define \
397 REDUCE_MIN_CLASS(...) \
398 GET_MACRO(__VA_ARGS__, RMINCLASS3D, _11, _10, RMINCLASS2D, _8, _7, RMINCLASS1D)(__VA_ARGS__)
399
400 #endif
401
402
403 // end of KOKKOS routines
404
405
406
407
408 // ----------------------------------------
409 // The for_all and for_reduce functions that
410 // are used with the non-kokkos MACROS
411 // ----------------------------------------
412
413 #ifndef HAVE_KOKKOS
414 #include <limits>  // for the max and min values of a int, double, etc.
415
416 template <typename F>
417 void for_all (int i_start, int i_end,
418               const F &lambda_fcn){
419
420     for (int i=i_start; i<i_end; i++){
421         lambda_fcn(i);
422     }
423
424 }; // end for_all
425
426
427 template <typename F>
428 void for_all (int i_start, int i_end,
429               int j_start, int j_end,
430               const F &lambda_fcn){
431
432     for (int i=i_start; i<i_end; i++){
433         for (int j=j_start; j<j_end; j++){
434             lambda_fcn(i,j);
435         }
436     }
437
438 }; // end for_all
439
440
441 template <typename F>
442 void for_all (int i_start, int i_end,
443               int j_start, int j_end,
444               int k_start, int k_end,
445               const F &lambda_fcn){
446
447     for (int i=i_start; i<i_end; i++){
448         for (int j=j_start; j<j_end; j++){
449             for (int k=k_start; k<k_end; k++){
450                 lambda_fcn(i,j,k);
451             }
452         }
453     }
454
455 }; // end for_all
456
457
458 // SUM
459 template <typename T, typename F>
460 void reduce_sum (int i_start, int i_end,
461                  T var,
462                  const F &lambda_fcn, T &result){
463     var = 0;
464     for (int i=i_start; i<i_end; i++){
```

```
465         lambda_fcn(i, var);
466     }
467     result = var;
468 };  // end for_reduce
469
470
471 template <typename T, typename F>
472 void reduce_sum (int i_start, int i_end,
473                  int j_start, int j_end,
474                  T var,
475                  const F &lambda_fcn, T &result){
476     var = 0;
477     for (int i=i_start; i<i_end; i++){
478         for (int j=j_start; j<j_end; j++){
479             lambda_fcn(i,j,var);
480         }
481     }
482
483     result = var;
484 };  // end for_reduce
485
486
487 template <typename T, typename F>
488 void reduce_sum (int i_start, int i_end,
489                  int j_start, int j_end,
490                  int k_start, int k_end,
491                  T  var,
492                  const F &lambda_fcn,  T &result){
493     var = 0;
494     for (int i=i_start; i<i_end; i++){
495         for (int j=j_start; j<j_end; j++){
496             for (int k=k_start; k<k_end; k++){
497                 lambda_fcn(i,j,k,var);
498             }
499         }
500     }
501
502     result = var;
503 };  // end for_reduce
504
505
506 // MIN
507 template <typename T, typename F>
508 void reduce_min (int i_start, int i_end,
509                  T var,
510                  const F &lambda_fcn, T &result){
511     var = std::numeric_limits<T>::max(); //2147483647;
512     for (int i=i_start; i<i_end; i++){
513         lambda_fcn(i, var);
514     }
515     result = var;
516 };  // end for_reduce
517
518
519 template <typename T, typename F>
520 void reduce_min (int i_start, int i_end,
521                  int j_start, int j_end,
522                  T var,
523                  const F &lambda_fcn, T &result){
524     var = std::numeric_limits<T>::max(); //2147483647;
525     for (int i=i_start; i<i_end; i++){
526         for (int j=j_start; j<j_end; j++){
527             lambda_fcn(i,j,var);
528         }
529     }
530
531     result = var;
532 };  // end for_reduce
533
534
535 template <typename T, typename F>
536 void reduce_min (int i_start, int i_end,
537                  int j_start, int j_end,
538                  int k_start, int k_end,
539                  T  var,
540                  const F &lambda_fcn,  T &result){
541     var = std::numeric_limits<T>::max(); //2147483647;
542     for (int i=i_start; i<i_end; i++){
543         for (int j=j_start; j<j_end; j++){
544             for (int k=k_start; k<k_end; k++){
545                 lambda_fcn(i,j,k,var);
546             }
547         }
548     }
549
550     result = var;
551 };  // end for_reduce
```

```
552
553 // MAX
554 template <typename T, typename F>
555 void reduce_max (int i_start, int i_end,
556                  T var,
557                  const F &lambda_fcn, T &result){
558     var = std::numeric_limits<T>::min(); // -2147483647 - 1;
559     for (int i=i_start; i<i_end; i++){
560         lambda_fcn(i, var);
561     }
562     result = var;
563 };  // end for_reduce
564
565
566 template <typename T, typename F>
567 void reduce_max (int i_start, int i_end,
568                  int j_start, int j_end,
569                  T var,
570                  const F &lambda_fcn, T &result){
571     var = std::numeric_limits<T>::min(); //-2147483647 - 1;
572     for (int i=i_start; i<i_end; i++){
573         for (int j=j_start; j<j_end; j++){
574             lambda_fcn(i,j,var);
575         }
576     }
577
578     result = var;
579 };  // end for_reduce
580
581
582 template <typename T, typename F>
583 void reduce_max (int i_start, int i_end,
584                  int j_start, int j_end,
585                  int k_start, int k_end,
586                  T  var,
587                  const F &lambda_fcn,  T &result){
588     var = std::numeric_limits<T>::min(); // -2147483647 - 1;
589     for (int i=i_start; i<i_end; i++){
590         for (int j=j_start; j<j_end; j++){
591             for (int k=k_start; k<k_end; k++){
592                 lambda_fcn(i,j,k,var);
593             }
594         }
595     }
596
597     result = var;
598 };  // end for_reduce
599
600 #endif  // if not kokkos
601
602 // ----------------------------------------
603 // MACROS for none kokkos loops
604 // ----------------------------------------
605
606 #ifndef HAVE_KOKKOS
607
608 // replace the CLASS loops to be the nominal loops
609 #define FOR_ALL_CLASS FOR_ALL
610 #define REDUCE_SUM_CLASS REDUCE_SUM
611 #define REDUCE_MAX_CLASS REDUCE_MAX
612 #define REDUCE_MIN_CLASS REDUCE_MIN
613
614 // the FOR_ALL loop is chosen based on the number of inputs
615
616 // the FOR_ALL loop
617 // 1D FOR loop has 4 inputs
618 #define \
619     FOR1D(i, x0, x1, fcn) \
620     for_all( (x0), (x1), \
621              [&]( const int (i) ){fcn} )
622 // 2D FOR loop has 7 inputs
623 #define \
624     FOR2D(i, x0, x1, j, y0, y1, fcn)  \
625     for_all( (x0), (x1), (y0), (y1), \
626              [&]( const int (i), const int (j) ){fcn} )
627 // 3D FOR loop has 10 inputs
628 #define \
629     FOR3D(i, x0, x1, j, y0, y1, k, z0, z1, fcn) \
630     for_all( (x0), (x1), (y0), (y1), (z0), (z1), \
631              [&]( const int (i), const int (j), const int (k) ) {fcn} )
632 #define \
633     FOR_ALL(...) \
634     GET_MACRO(__VA_ARGS__, _12, _11, FOR3D, _9, _8, FOR2D, _6, _5, FOR1D)(__VA_ARGS__)
635
636
637 // the DO_ALL loop
638 // 1D DOloop has 4 inputs
```

```
639 #define \
640     DO1D(i, x0, x1, fcn) \
641     for_all( (x0), (x1)+1, \
642              [&]( const int (i) ){fcn} )
643 // 2D DO loop has 7 inputs
644 #define \
645     DO2D(i, x0, x1, j, y0, y1, fcn)  \
646     for_all( (x0), (x1)+1, (y0), (y1)+1, \
647              [&]( const int (i), const int (j) ){fcn} )
648 // 3D DO loop has 10 inputs
649 #define \
650     DO3D(i, x0, x1, j, y0, y1, k, z0, z1, fcn) \
651     for_all( (x0), (x1)+1, (y0), (y1)+1, (z0), (z1)+1, \
652              [&]( const int (i), const int (j), const int (k) ) {fcn} )
653 #define \
654     DO_ALL(...) \
655     GET_MACRO(__VA_ARGS__, _12, _11, DO3D, _9, _8, DO2D, _6, _5, DO1D)(__VA_ARGS__)
656
657
658 // the REDUCE loops, no kokkos
659 #define \
660     RSUM1D(i, x0, x1, var, fcn, result) \
661     reduce_sum( (x0), (x1), (var),  \
662                 [=]( const int (i), decltype(var) &(var) ){fcn}, \
663                 (result) )
664 #define \
665     RSUM2D(i, x0, x1, j, y0, y1, var, fcn, result) \
666     reduce_sum( (x0), (x1), (y0), (y1), (var),  \
667                 [=]( const int (i),const int (j), decltype(var) &(var) ){fcn}, \
668                 (result) )
669 #define \
670     RSUM3D(i, x0, x1, j, y0, y1, k, z0, z1, var, fcn, result) \
671     reduce_sum( (x0), (x1), (y0), (y1), (z0), (z1), (var),  \
672                 [=]( const int (i), const int (j), const int (k), decltype(var) &(var) ){fcn}, \
673                 (result) )
674
675 #define \
676     REDUCE_SUM(...) \
677     GET_MACRO(__VA_ARGS__, RSUM3D, _11, _10, RSUM2D, _8, _7, RSUM1D)(__VA_ARGS__)
678
679
680 // DO_REDUCE_SUM
681 #define \
682     DO_RSUM1D(i, x0, x1, var, fcn, result) \
683     reduce_sum( (x0), (x1)+1, (var),  \
684                 [=]( const int (i), decltype(var) &(var) ){fcn}, \
685                 (result) )
686 #define \
687     DO_RSUM2D(i, x0, x1, j, y0, y1, var, fcn, result) \
688     reduce_sum( (x0), (x1)+1, (y0), (y1)+1, (var),  \
689                 [=]( const int (i),const int (j), decltype(var) &(var) ){fcn}, \
690                 (result) )
691 #define \
692     DO_RSUM3D(i, x0, x1, j, y0, y1, k, z0, z1, var, fcn, result) \
693     reduce_sum( (x0), (x1)+1, (y0), (y1)+1, (z0), (z1)+1, (var),  \
694                 [=]( const int (i), const int (j), const int (k), decltype(var) &(var) ){fcn}, \
695                 (result) )
696
697 #define \
698     DO_REDUCE_SUM(...) \
699     GET_MACRO(__VA_ARGS__, DO_RSUM3D, _11, _10, DO_RSUM2D, _8, _7, DO_RSUM1D)(__VA_ARGS__)
700
701
702 // Reduce max
703 #define \
704     RMAX1D(i, x0, x1, var, fcn, result) \
705     reduce_max( (x0), (x1), (var),  \
706                 [=]( const int (i), decltype(var) &(var) ){fcn}, \
707                 (result) )
708 #define \
709     RMAX2D(i, x0, x1, j, y0, y1, var, fcn, result) \
710     reduce_max( (x0), (x1), (y0), (y1), (var),  \
711                 [=]( const int (i),const int (j), decltype(var) &(var) ){fcn}, \
712                 (result) )
713 #define \
714     RMAX3D(i, x0, x1, j, y0, y1, k, z0, z1, var, fcn, result) \
715     reduce_max( (x0), (x1), (y0), (y1), (z0), (z1), (var),  \
716                 [=]( const int (i), const int (j), const int (k), decltype(var) &(var) ){fcn}, \
717                 (result) )
718
719 #define \
720     REDUCE_MAX(...) \
721     GET_MACRO(__VA_ARGS__, RMAX3D, _11, _10, RMAX2D, _8, _7, RMAX1D)(__VA_ARGS__)
722
723
724 // DO_REDUCE_MAX
725 #define \
```

```
726     DO_RMAX1D(i, x0, x1, var, fcn, result) \
727     reduce_max( (x0), (x1)+1, (var),  \
728                 [=]( const int (i), decltype(var) &(var) ){fcn}, \
729                 (result) )
730 #define \
731     DO_RMAX2D(i, x0, x1, j, y0, y1, var, fcn, result) \
732     reduce_max( (x0), (x1)+1, (y0), (y1)+1, (var),  \
733                 [=]( const int (i),const int (j), decltype(var) &(var) ){fcn}, \
734                 (result) )
735 #define \
736     DO_RMAX3D(i, x0, x1, j, y0, y1, k, z0, z1, var, fcn, result) \
737     reduce_max( (x0), (x1)+1, (y0), (y1)+1, (z0), (z1)+1, (var),  \
738                 [=]( const int (i), const int (j), const int (k), decltype(var) &(var) ){fcn}, \
739                 (result) )
740
741 #define \
742     DO_REDUCE_MAX(...) \
743     GET_MACRO(__VA_ARGS__, DO_RMAX3D, _11, _10, DO_RMAX2D, _8, _7, DO_RMAX1D)(__VA_ARGS__)
744
745
746 // reduce min
747 #define \
748     RMIN1D(i, x0, x1, var, fcn, result) \
749     reduce_min( (x0), (x1), (var),  \
750                 [=]( const int (i), decltype(var) &(var) ){fcn}, \
751                 (result) )
752 #define \
753     RMIN2D(i, x0, x1, j, y0, y1, var, fcn, result) \
754     reduce_min( (x0), (x1), (y0), (y1), (var),  \
755                 [=]( const int (i),const int (j), decltype(var) &(var) ){fcn}, \
756                 (result) )
757 #define \
758     RMIN3D(i, x0, x1, j, y0, y1, k, z0, z1, var, fcn, result) \
759     reduce_min( (x0), (x1), (y0), (y1), (z0), (z1), (var),  \
760                 [=]( const int (i), const int (j), const int (k), decltype(var) &(var) ){fcn}, \
761                 (result) )
762
763 #define \
764     REDUCE_MIN(...) \
765     GET_MACRO(__VA_ARGS__, RMIN3D, _11, _10, RMIN2D, _8, _7, RMIN1D)(__VA_ARGS__)
766
767
768 // DO_REDUCE_MIN
769 #define \
770     DO_RMIN1D(i, x0, x1, var, fcn, result) \
771     reduce_min( (x0), (x1)+1, (var),  \
772                 [=]( const int (i), decltype(var) &(var) ){fcn}, \
773                 (result) )
774 #define \
775     DO_RMIN2D(i, x0, x1, j, y0, y1, var, fcn, result) \
776     reduce_min( (x0), (x1)+1, (y0), (y1)+1, (var),  \
777                 [=]( const int (i),const int (j), decltype(var) &(var) ){fcn}, \
778                 (result) )
779 #define \
780     DO_RMIN3D(i, x0, x1, j, y0, y1, k, z0, z1, var, fcn, result) \
781     reduce_min( (x0), (x1)+1, (y0), (y1)+1, (z0), (z1)+1, (var),  \
782                 [=]( const int (i), const int (j), const int (k), decltype(var) &(var) ){fcn}, \
783                 (result) )
784
785 #define \
786     DO_REDUCE_MIN(...) \
787     GET_MACRO(__VA_ARGS__, DO_RMIN3D, _11, _10, DO_RMIN2D, _8, _7, DO_RMIN1D)(__VA_ARGS__)
788
789
790 #endif  // if not kokkos
791
792
793
```

5.2 matar.h

```
1 #ifndef MATAR_H
2 #define MATAR_H
3 /**********************************************************************************************
4  © 2020. Triad National Security, LLC. All rights reserved.
5  This program was produced under U.S. Government contract 89233218CNA000001 for Los Alamos
6  National Laboratory (LANL), which is operated by Triad National Security, LLC for the U.S.
7  Department of Energy/National Nuclear Security Administration. All rights in the program are
8  reserved by Triad National Security, LLC, and the U.S. Department of Energy/National Nuclear
9  Security Administration. The Government is granted for itself and others acting on its behalf a
10  nonexclusive, paid-up, irrevocable worldwide license in this material to reproduce, prepare
11  derivative works, distribute copies to the public, perform publicly and display publicly, and
12  to permit others to do so.
```

```
13  This program is open source under the BSD-3 License.
14  Redistribution and use in source and binary forms, with or without modification, are permitted
15  provided that the following conditions are met:
16
17  1.  Redistributions of source code must retain the above copyright notice, this list of
18  conditions and the following disclaimer.
19
20  2.  Redistributions in binary form must reproduce the above copyright notice, this list of
21  conditions and the following disclaimer in the documentation and/or other materials
22  provided with the distribution.
23
24  3.  Neither the name of the copyright holder nor the names of its contributors may be used
25  to endorse or promote products derived from this software without specific prior
26  written permission.
27  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS
28  IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
29  IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
30  PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR
31  CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
32  EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
33  PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS;
34  OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY,
35  WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR
36  OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF
37  ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
38  **********************************************************************************************/
39
40 // Order
41 //
42 //  Standard (non-Kokkos data structures)
43 //    1. FArray
44 //    2. ViewFArray
45 //    3. FMatrix
46 //    4. ViewFMatrix
47 //    5. CArray
48 //    6. ViewCArray
49 //    7. CMatrix
50 //    8. ViewCMatrix
51 //    9. RaggedRightArray
52 //    10. RaggedDownArray
53 //    11. DynamicRaggedRightArray
54 //    12. DynamicRaggedDownArray
55 //    13. SparseRowArray
56 //    14. SparseColArray
57 //    15. CSRArray
58 //    16. CSCArray //todo
59 //
60 //    Kokkos Data structures
61 //    17. FArrayKokkos
62 //    18. ViewFArrayKokkos
63 //    19. FMatrixKokkos
64 //    20. ViewFMatrixKokkos
65 //    21. CArrayKokkos
66 //    22. ViewCArrayKokkos
67 //    23. CMatrixKokkos
68 //    24. ViewCMatrixKokkos
69 //    25. RaggedRightArrayKokkos
70 //    26. RaggedDownArrayKokkos
71 //    27. DynamicRaggedRightArrayKokkos
72 //    28. DynamicRaggedDownArrayKokkos
73 //    29. SparseRowArrayKokkos
74 //    29. SparseColArrayKokkos
75
76
77 #include <stdio.h>
78 #include <stdlib.h>
79 #include <string>
80 #include <assert.h>
81 #include <memory> // for shared_ptr
82 #include "macros.h"
83
84 using real_t = double;
85 using u_int  = unsigned int;
86
87
88 #ifdef HAVE_KOKKOS
89 #include <Kokkos_Core.hpp>
90 #include <Kokkos_DualView.hpp>
91
92 using HostSpace    = Kokkos::HostSpace;
93 using MemoryUnmanaged = Kokkos::MemoryUnmanaged;
94
95 #ifdef HAVE_CUDA
96 //using UVMMemSpace     = Kokkos::CudaUVMSpace;
97 using DefaultMemSpace  = Kokkos::CudaSpace;
98 using DefaultExecSpace = Kokkos::Cuda;
99 using DefaultLayout    = Kokkos::LayoutLeft;
```

```
100 #elif HAVE_OPENMP
101 using DefaultMemSpace  = Kokkos::HostSpace;
102 using DefaultExecSpace = Kokkos::OpenMP;
103 using DefaultLayout    = Kokkos::LayoutRight;
104 #elif HAVE_THREADS
105 using DefaultMemSpace  = Kokkos::HostSpace;
106 using DefaultExecSpace = Kokkos::Threads;
107 using DefaultLayout    = Kokkos::LayoutLeft;
108 #elif HAVE_HIP
109 using DefaultMemSpace  = Kokkos::Experimental::HIPSpace;
110 using DefaultExecSpace = Kokkos::Experimental::HIP;
111 using DefaultLayout    = Kokkos::LayoutLeft;
112 #else
113 using DefaultMemSpace  = Kokkos::Serial;
114 using DefaultExecSpace = Kokkos::Serial;
115 using DefaultLayout    = Kokkos::LayoutLeft;
116 #endif
117
118 //MACROS to make the code less scary
119 #define kmalloc(size) ( Kokkos::kokkos_malloc<DefaultMemSpace>(size) )
120 #define kfree(pnt)        (  Kokkos::kokkos_free(pnt) )
121 #define ProfileRegionStart  ( Kokkos::Profiling::pushRegion )
122 #define ProfileRegionEnd  ( Kokkos::Profiling::popRegion )
123 #define DEFAULTSTRINGARRAY "array_"
124 #define DEFAULTSTRINGMATRIX "matrix_"
125
126 using policy1D = Kokkos::RangePolicy<DefaultExecSpace>;
127 using policy2D = Kokkos::MDRangePolicy< Kokkos::Rank<2> >;
128 using policy3D = Kokkos::MDRangePolicy< Kokkos::Rank<3> >;
129 using policy4D = Kokkos::MDRangePolicy< Kokkos::Rank<4> >;
130
131 using TeamPolicy = Kokkos::TeamPolicy<DefaultExecSpace>;
132 //using mdrange_policy2 = Kokkos::MDRangePolicy<Kokkos::Rank<2»;
133 //using mdrange_policy3 = Kokkos::MDRangePolicy<Kokkos::Rank<3»;
134
135 using RMatrix1D    = Kokkos::View<real_t *,DefaultLayout,DefaultExecSpace>;
136 using RMatrix2D    = Kokkos::View<real_t **,DefaultLayout,DefaultExecSpace>;
137 using RMatrix3D    = Kokkos::View<real_t ***,DefaultLayout,DefaultExecSpace>;
138 using RMatrix4D    = Kokkos::View<real_t ****,DefaultLayout,DefaultExecSpace>;
139 using RMatrix5D    = Kokkos::View<real_t *****,DefaultLayout,DefaultExecSpace>;
140 using IMatrix1D    = Kokkos::View<int *,DefaultLayout,DefaultExecSpace>;
141 using IMatrix2D    = Kokkos::View<int **,DefaultLayout,DefaultExecSpace>;
142 using IMatrix3D    = Kokkos::View<int ***,DefaultLayout,DefaultExecSpace>;
143 using IMatrix4D    = Kokkos::View<int ****,DefaultLayout,DefaultExecSpace>;
144 using IMatrix5D    = Kokkos::View<int *****,DefaultLayout,DefaultExecSpace>;
145 using SVar         = Kokkos::View<size_t,DefaultLayout,DefaultExecSpace>;
146 using SArray2D     = Kokkos::View<size_t **,DefaultLayout,DefaultExecSpace>;
147 using SArray3D     = Kokkos::View<size_t ***,DefaultLayout,DefaultExecSpace>;
148 using SArray4D     = Kokkos::View<size_t ****,DefaultLayout,DefaultExecSpace>;
149 using SArray5D     = Kokkos::View<size_t *****,DefaultLayout,DefaultExecSpace>;
150
151 using SHArray1D     = Kokkos::View<size_t *,DefaultLayout,Kokkos::HostSpace>;
152 #endif
153
154 //To disable asserts, uncomment the following line
155 //#define NDEBUG
156
157
158 //---Begin Standard Data Structures---
159
160 //1. FArray
161 // indicies are [0:N-1]
162 template <typename T>
163 class FArray {
164
165 private:
166     size_t dims_[7];
167     size_t length_;
168     size_t order_;  // tensor order (rank)
169     std::shared_ptr <T []> array_;
170
171 public:
172
173     // default constructor
174    FArray ();
175
176     //overload constructors from 1D to 7D
177
178    FArray(size_t dim0);
179
180    FArray(size_t dim0,
181           size_t dim1);
182
183    FArray(size_t dim0,
184           size_t dim1,
185           size_t dim2);
186
```

```
187     FArray(size_t dim0,
188            size_t dim1,
189            size_t dim2,
190            size_t dim3);
191
192     FArray(size_t dim0,
193            size_t dim1,
194            size_t dim2,
195            size_t dim3,
196            size_t dim4);
197
198     FArray(size_t dim0,
199            size_t dim1,
200            size_t dim2,
201            size_t dim3,
202            size_t dim4,
203            size_t dim5);
204
205     FArray(size_t dim0,
206            size_t dim1,
207            size_t dim2,
208            size_t dim3,
209            size_t dim4,
210            size_t dim5,
211            size_t dim6);
212
213      FArray (const FArray& temp);
214
215      // overload operator() to access data as array(i,....,n);
216      T& operator()(size_t i) const;
217
218      T& operator()(size_t i,
219                    size_t j) const;
220
221      T& operator()(size_t i,
222                    size_t j,
223                    size_t k) const;
224
225      T& operator()(size_t i,
226                    size_t j,
227                    size_t k,
228                    size_t l) const;
229
230      T& operator()(size_t i,
231                    size_t j,
232                    size_t k,
233                    size_t l,
234                    size_t m) const;
235
236      T& operator()(size_t i,
237                    size_t j,
238                    size_t k,
239                    size_t l,
240                    size_t m,
241                    size_t n) const;
242      T& operator()(size_t i,
243                    size_t j,
244                    size_t k,
245                    size_t l,
246                    size_t m,
247                    size_t n,
248                    size_t o) const;
249
250      //overload = operator
251      FArray& operator=(const FArray& temp);
252
253      //return array size
254      size_t size() const;
255
256      // return array dims
257      size_t dims(size_t i) const;
258
259      // return array order (rank)
260      size_t order() const;
261
262      //return pointer
263      T* pointer() const;
264
265      // deconstructor
266      ~FArray ();
267
268 }; // end of f_array_t
269
270 //---FArray class definnitions----
271
272 //constructors
273 template <typename T>
```

```
274 FArray<T>::FArray(){
275     array_ = NULL;
276     length_ = 0;
277 }
278
279 //1D
280 template <typename T>
281 FArray<T>::FArray(size_t dim0)
282 {
283     dims_[0] = dim0;
284     length_ = dim0;
285     order_ = 1;
286     array_ = std::shared_ptr <T []> (new T[length_]);
287 }
288
289 template <typename T>
290 FArray<T>::FArray(size_t dim0,
291                   size_t dim1)
292 {
293     dims_[0] = dim0;
294     dims_[1] = dim1;
295     order_ = 2;
296     length_ = dim0*dim1;
297     array_ = std::shared_ptr <T []> (new T[length_]);
298 }
299
300 //3D
301 template <typename T>
302 FArray<T>::FArray(size_t dim0,
303                   size_t dim1,
304                   size_t dim2)
305 {
306     dims_[0] = dim0;
307     dims_[1] = dim1;
308     dims_[2] = dim2;
309     order_ = 3;
310     length_ = dim0*dim1*dim2;
311     array_ = std::shared_ptr <T []> (new T[length_]);
312 }
313
314 //4D
315 template <typename T>
316 FArray<T>::FArray(size_t dim0,
317                   size_t dim1,
318                   size_t dim2,
319                   size_t dim3)
320 {
321     dims_[0] = dim0;
322     dims_[1] = dim1;
323     dims_[2] = dim2;
324     dims_[3] = dim3;
325     order_ = 4;
326     length_ = dim0*dim1*dim2*dim3;
327     array_ = std::shared_ptr <T []> (new T[length_]);
328 }
329
330 //5D
331 template <typename T>
332 FArray<T>::FArray(size_t dim0,
333                   size_t dim1,
334                   size_t dim2,
335                   size_t dim3,
336                   size_t dim4)
337 {
338     dims_[0] = dim0;
339     dims_[1] = dim1;
340     dims_[2] = dim2;
341     dims_[3] = dim3;
342     dims_[4] = dim4;
343     order_ = 5;
344     length_ = dim0*dim1*dim2*dim3*dim4;
345     array_ = std::shared_ptr <T []> (new T[length_]);
346 }
347
348 //6D
349 template <typename T>
350 FArray<T>::FArray(size_t dim0,
351                   size_t dim1,
352                   size_t dim2,
353                   size_t dim3,
354                   size_t dim4,
355                   size_t dim5)
356 {
357     dims_[0] = dim0;
358     dims_[1] = dim1;
359     dims_[2] = dim2;
360     dims_[3] = dim3;
```

```
361     dims_[4] = dim4;
362     dims_[5] = dim5;
363     order_ = 6;
364     length_ = dim0*dim1*dim2*dim3*dim4*dim5;
365     array_ = std::shared_ptr <T []> (new T[length_]);
366 }
367
368
369 //7D
370 template <typename T>
371 FArray<T>::FArray(size_t dim0,
372                   size_t dim1,
373                   size_t dim2,
374                   size_t dim3,
375                   size_t dim4,
376                   size_t dim5,
377                   size_t dim6)
378 {
379     dims_[0] = dim0;
380     dims_[1] = dim1;
381     dims_[2] = dim2;
382     dims_[3] = dim3;
383     dims_[4] = dim4;
384     dims_[5] = dim5;
385     dims_[6] = dim6;
386     order_ = 7;
387     length_ = dim0*dim1*dim2*dim3*dim4*dim5*dim6;
388     array_ = std::shared_ptr <T []> (new T[length_]);
389
390 }
391
392 //Copy constructor
393
394 template <typename T>
395 FArray<T>::FArray(const FArray& temp) {
396
397     // Do nothing if the assignment is of the form x = x
398
399     if (this != &temp) {
400         for (int iter = 0; iter < temp.order_; iter++){
401             dims_[iter] = temp.dims_[iter];
402         } // end for
403
404         order_  = temp.order_;
405         length_ = temp.length_;
406         array_ = temp.array_;
407     } // end if
408
409 } // end constructor
410
411 //overload operator () for 1D to 7D
412 //indices are from [0:N-1]
413
414 //1D
415 template <typename T>
416 T& FArray<T>::operator()(size_t i) const
417 {
418     assert(order_ == 1 && "Tensor order (rank) does not match constructor in FArray 1D!");
419     assert(i >= 0 && i < dims_[0] && "i is out of bounds in FArray 1D!");
420     return array_[i];
421 }
422
423 //2D
424 template <typename T>
425 T& FArray<T>::operator()(size_t i,
426                          size_t j) const
427 {
428     assert(order_ == 2 && "Tensor order (rank) does not match constructor in FArray 2D!");
429     assert(i >= 0 && i < dims_[0] && "i is out of bounds in FArray 2D!");
430     assert(j >= 0 && j < dims_[1] && "j is out of bounds in FArray 2D!");
431     return array_[i + j*dims_[0]];
432 }
433
434 //3D
435 template <typename T>
436 T& FArray<T>::operator()(size_t i,
437                          size_t j,
438                          size_t k) const
439 {
440     assert(order_ == 3 && "Tensor order (rank) does not match constructor in FArray 3D!");
441     assert(i >= 0 && i < dims_[0] && "i is out of bounds in FArray 3D!");
442     assert(j >= 0 && j < dims_[1] && "j is out of bounds in Farray 3D!");
443     assert(k >= 0 && k < dims_[2] && "k is out of bounds in FArray 3D!");
444     return array_[i + j*dims_[0]
445                    + k*dims_[0]*dims_[1]];
446 }
447
```

```
448 //4D
449 template <typename T>
450 T& FArray<T>::operator()(size_t i,
451                          size_t j,
452                          size_t k,
453                          size_t l) const
454 {
455     assert(order_ == 4 && "Tensor order (rank) does not match constructor in FArray 4D!");
456     assert(i >= 0 && i < dims_[0] && "i is out of bounds in FArray 4D!");
457     assert(j >= 0 && j < dims_[1] && "j is out of bounds in FArray 4D!");
458     assert(k >= 0 && k < dims_[2] && "k is out of bounds in FArray 4D!");
459     assert(l >= 0 && l < dims_[3] && "l is out of bounds in FArray 4D!");
460     return array_[i + j*dims_[0]
461                     + k*dims_[0]*dims_[1]
462                     + l*dims_[0]*dims_[1]*dims_[2]];
463 }
464
465 //5D
466 template <typename T>
467 T& FArray<T>::operator()(size_t i,
468                          size_t j,
469                          size_t k,
470                          size_t l,
471                          size_t m) const
472 {
473     assert(order_ == 5 && "Tensor order (rank) does not match constructor in FArray 5D!");
474     assert(i >= 0 && i < dims_[0] && "i is out of bounds in FArray 5D!");
475     assert(j >= 0 && j < dims_[1] && "j is out of bounds in FArray 5D!");
476     assert(k >= 0 && k < dims_[2] && "k is out of bounds in FArray 5D!");
477     assert(l >= 0 && l < dims_[3] && "l is out of bounds in FArray 5D!");
478     assert(m >= 0 && m < dims_[4] && "m is out of bounds in FArray 5D!");
479     return array_[i + j*dims_[0]
480                     + k*dims_[0]*dims_[1]
481                     + l*dims_[0]*dims_[1]*dims_[2]
482                     + m*dims_[0]*dims_[1]*dims_[2]*dims_[3]];
483 }
484
485 //6D
486 template <typename T>
487 T& FArray<T>::operator()(size_t i,
488                          size_t j,
489                          size_t k,
490                          size_t l,
491                          size_t m,
492                          size_t n) const
493 {
494     assert(order_ == 6 && "Tensor order (rank) does not match constructor in FArray 6D!");
495     assert(i >= 0 && i < dims_[0] && "i is out of bounds in FArray 6D!");
496     assert(j >= 0 && j < dims_[1] && "j is out of bounds in FArray 6D!");
497     assert(k >= 0 && k < dims_[2] && "k is out of bounds in FArray 6D!");
498     assert(l >= 0 && l < dims_[3] && "l is out of bounds in FArray 6D!");
499     assert(m >= 0 && m < dims_[4] && "m is out of bounds in FArray 6D!");
500     assert(n >= 0 && n < dims_[5] && "n is out of bounds in FArray 6D!");
501     return array_[i + j*dims_[0]
502                     + k*dims_[0]*dims_[1]
503                     + l*dims_[0]*dims_[1]*dims_[2]
504                     + m*dims_[0]*dims_[1]*dims_[2]*dims_[3]
505                     + n*dims_[0]*dims_[1]*dims_[2]*dims_[3]*dims_[4]];
506 }
507
508 //7D
509 template <typename T>
510 T& FArray<T>::operator()(size_t i,
511                          size_t j,
512                          size_t k,
513                          size_t l,
514                          size_t m,
515                          size_t n,
516                          size_t o) const
517 {
518     assert(order_ == 7 && "Tensor order (rank) does not match constructor in FArray 7D!");
519     assert(i >= 0 && i < dims_[0] && "i is out of bounds in FArray 7D!");
520     assert(j >= 0 && j < dims_[1] && "j is out of bounds in FArray 7D!");
521     assert(k >= 0 && k < dims_[2] && "k is out of bounds in FArray 7D!");
522     assert(l >= 0 && l < dims_[3] && "l is out of bounds in FArray 7D!");
523     assert(m >= 0 && m < dims_[4] && "m is out of bounds in FArray 7D!");
524     assert(n >= 0 && n < dims_[5] && "n is out of bounds in FArray 7D!");
525     assert(o >= 0 && o < dims_[6] && "o is out of bounds in FArray 7D!");
526     return array_[i + j*dims_[0]
527                     + k*dims_[0]*dims_[1]
528                     + l*dims_[0]*dims_[1]*dims_[2]
529                     + m*dims_[0]*dims_[1]*dims_[2]*dims_[3]
530                     + n*dims_[0]*dims_[1]*dims_[2]*dims_[3]*dims_[4]
531                     + o*dims_[0]*dims_[1]*dims_[2]*dims_[3]*dims_[4]*dims_[5]];
532 }
533
534 // = operator
```

```
535 //THIS = FArray <> TEMP(n,m,...)
536 template <typename T>
537 FArray<T>& FArray<T>::operator= (const FArray& temp)
538 {
539     if(this != & temp) {
540         for (int iter = 0; iter < temp.order_; iter++){
541             dims_[iter] = temp.dims_[iter];
542         } // end for
543 
544         order_  = temp.order_;
545         length_ = temp.length_;
546         array_  = temp.array_;
547     }
548     return *this;
549 }
550 
551 template <typename T>
552 inline size_t FArray<T>::size() const {
553     return length_;
554 }
555 
556 template <typename T>
557 inline size_t FArray<T>::dims(size_t i) const {
558     assert(i < order_ && "FArray order (rank) does not match constructor, dim[i] does not exist!");
559     assert(i >= 0 && dims_[i]>0 && "Access to FArray dims is out of bounds!");
560     return dims_[i];
561 }
562 
563 template <typename T>
564 inline size_t FArray<T>::order() const {
565     return order_;
566 }
567 
568 
569 template <typename T>
570 inline T* FArray<T>::pointer() const {
571     return array_.get();
572 }
573 
574 //delete FArray
575 template <typename T>
576 FArray<T>::~FArray(){}
577 
578 //---end of FArray class definitions----
579 
580 
581 //2. ViewFArray
582 // indicies are [0:N-1]
583 template <typename T>
584 class ViewFArray {
585 
586 private:
587     size_t dims_[7];
588     size_t length_; // Length of 1D array
589     size_t order_;  // tensor order (rank)
590     T * array_;
591 
592 public:
593 
594     // default constructor
595     ViewFArray ();
596 
597     //---1D to 7D array ---
598     ViewFArray(T *array,
599                size_t dim0);
600 
601     ViewFArray (T *array,
602                 size_t dim0,
603                 size_t dim1);
604 
605     ViewFArray (T *array,
606                 size_t dim0,
607                 size_t dim1,
608                 size_t dim2);
609 
610     ViewFArray (T *array,
611                 size_t dim0,
612                 size_t dim1,
613                 size_t dim2,
614                 size_t dim3);
615 
616     ViewFArray (T *array,
617                 size_t dim0,
618                 size_t dim1,
619                 size_t dim2,
620                 size_t dim3,
621                 size_t dim4);
```

```
622
623     ViewFArray (T *array,
624                 size_t dim0,
625                 size_t dim1,
626                 size_t dim2,
627                 size_t dim3,
628                 size_t dim4,
629                 size_t dim5);
630
631     ViewFArray (T *array,
632                 size_t dim0,
633                 size_t dim1,
634                 size_t dim2,
635                 size_t dim3,
636                 size_t dim4,
637                 size_t dim5,
638                 size_t dim6);
639
640     T& operator()(size_t i) const;
641
642     T& operator()(size_t i,
643                   size_t j) const;
644
645     T& operator()(size_t i,
646                   size_t j,
647                   size_t k) const;
648
649     T& operator()(size_t i,
650                   size_t j,
651                   size_t k,
652                   size_t l) const;
653
654     T& operator()(size_t i,
655                   size_t j,
656                   size_t k,
657                   size_t l,
658                   size_t m) const;
659
660     T& operator()(size_t i,
661                   size_t j,
662                   size_t k,
663                   size_t l,
664                   size_t m,
665                   size_t n) const;
666
667     T& operator()(size_t i,
668                   size_t j,
669                   size_t k,
670                   size_t l,
671                   size_t m,
672                   size_t n,
673                   size_t o) const;
674
675     // calculate C = math(A,B)
676     template <typename M>
677     void operator=(M do_this_math);
678
679     //return array size
680     size_t size() const;
681
682     //return array dims
683     size_t dims(size_t i) const;
684
685     // return array order (rank)
686     size_t order() const;
687
688     // return pointer
689     T* pointer() const;
690
691 }; // end of viewFArray
692
693 //class definitions for viewFArray
694
695 //~~~~constructors for viewFArray for 1D to 7D~~~~~~~
696
697 //no dimension
698 template <typename T>
699 ViewFArray<T>::ViewFArray(){
700   array_ = NULL;
701   length_ = 0;
702 }
703
704 //1D
705 template <typename T>
706 ViewFArray<T>::ViewFArray(T *array,
707                           size_t dim0)
708 {
```

```
709     dims_[0] = dim0;
710     order_ = 1;
711     length_ = dim0;
712     array_  = array;
713 }
714
715 //2D
716 template <typename T>
717 ViewFArray<T>::ViewFArray(T *array,
718                           size_t dim0,
719                           size_t dim1)
720 {
721     dims_[0] = dim0;
722     dims_[1] = dim1;
723     order_ = 2;
724     length_ = dim0*dim1;
725     array_  = array;
726 }
727
728 //3D
729 template <typename T>
730 ViewFArray<T>::ViewFArray(T *array,
731                           size_t dim0,
732                           size_t dim1,
733                           size_t dim2)
734 {
735     dims_[0] = dim0;
736     dims_[1] = dim1;
737     dims_[2] = dim2;
738     order_ = 3;
739     length_ = dim0*dim1*dim2;
740     array_  = array;
741 }
742
743 //4D
744 template <typename T>
745 ViewFArray<T>::ViewFArray(T *array,
746                           size_t dim0,
747                           size_t dim1,
748                           size_t dim2,
749                           size_t dim3)
750 {
751     dims_[0] = dim0;
752     dims_[1] = dim1;
753     dims_[2] = dim2;
754     dims_[3] = dim3;
755     order_ = 4;
756     length_ = dim0*dim1*dim2*dim3;
757     array_  = array;
758 }
759
760 //5D
761 template <typename T>
762 ViewFArray<T>::ViewFArray(T *array,
763                           size_t dim0,
764                           size_t dim1,
765                           size_t dim2,
766                           size_t dim3,
767                           size_t dim4)
768 {
769     dims_[0] = dim0;
770     dims_[1] = dim1;
771     dims_[2] = dim2;
772     dims_[3] = dim3;
773     dims_[4] = dim4;
774     order_ = 5;
775     length_ = dim0*dim1*dim2*dim3*dim4;
776     array_  = array;
777 }
778
779 //6D
780 template <typename T>
781 ViewFArray<T>::ViewFArray(T *array,
782                           size_t dim0,
783                           size_t dim1,
784                           size_t dim2,
785                           size_t dim3,
786                           size_t dim4,
787                           size_t dim5)
788 {
789     dims_[0] = dim0;
790     dims_[1] = dim1;
791     dims_[2] = dim2;
792     dims_[3] = dim3;
793     dims_[4] = dim4;
794     dims_[5] = dim5;
795     order_ = 6;
```

```
796     length_ = dim0*dim1*dim2*dim3*dim4*dim5;
797     array_  = array;
798 }
799
800 //7D
801 template <typename T>
802 ViewFArray<T>::ViewFArray(T *array,
803                           size_t dim0,
804                           size_t dim1,
805                           size_t dim2,
806                           size_t dim3,
807                           size_t dim4,
808                           size_t dim5,
809                           size_t dim6)
810 {
811     dims_[0] = dim0;
812     dims_[1] = dim1;
813     dims_[2] = dim2;
814     dims_[3] = dim3;
815     dims_[4] = dim4;
816     dims_[5] = dim5;
817     dims_[6] = dim6;
818     order_ = 7;
819     length_ = dim0*dim1*dim2*dim3*dim4*dim5*dim6;
820     array_  = array;
821 }
822
823 //~~~~~~operator () overload
824 //for dimensions 1D to 7D
825 //indices for array are from 0...N-1
826
827 //1D
828 template <typename T>
829 T& ViewFArray<T>::operator()(size_t i) const
830 {
831     assert(order_ == 1 && "Tensor order (rank) does not match constructor in ViewFArray 1D!");
832     assert(i >= 0 && i < dims_[0] && "i is out of bounds in ViewFArray 1D!");
833     return array_[i];
834 }
835
836 //2D
837 template <typename T>
838 T& ViewFArray<T>::operator()(size_t i,
839                              size_t j) const
840 {
841     assert(order_ == 2 && "Tensor order (rank) does not match constructor in ViewFArray 2D!");
842     assert(i >= 0 && i < dims_[0] && "i is out of bounds in ViewFArray 2D!");
843     assert(j >= 0 && j < dims_[1] && "j is out of bounds in ViewFArray 2D!");
844     return array_[i + j*dims_[0]];
845 }
846
847 //3D
848 template <typename T>
849 T& ViewFArray<T>::operator()(size_t i,
850                              size_t j,
851                              size_t k) const
852 {
853     assert(order_ == 3 && "Tensor order (rank) does not match constructor in ViewFArray 3D!");
854     assert(i >= 0 && i < dims_[0] && "i is out of bounds in ViewFArray 3D!");
855     assert(j >= 0 && j < dims_[1] && "j is out of bounds in ViewFArray 3D!");
856     assert(k >= 0 && k < dims_[2] && "k is out of bounds in ViewFArray 3D!");
857     return array_[i + j*dims_[0]
858                     + k*dims_[0]*dims_[1]];
859 }
860
861 //4D
862 template <typename T>
863 T& ViewFArray<T>::operator()(size_t i,
864                              size_t j,
865                              size_t k,
866                              size_t l) const
867 {
868     assert(order_ == 4 && "Tensor order (rank) does not match constructor in ViewFArray 4D!");
869     assert(i >= 0 && i < dims_[0] && "i is out of bounds in ViewFArray 4D!");
870     assert(j >= 0 && j < dims_[1] && "j is out of bounds in ViewFArray 4D!");
871     assert(k >= 0 && k < dims_[2] && "k is out of bounds in ViewFArray 4D!");
872     assert(l >= 0 && l < dims_[3] && "l is out of bounds in ViewFArray 4D!");
873     return array_[i + j*dims_[0]
874                     + k*dims_[0]*dims_[1]
875                     + l*dims_[0]*dims_[1]*dims_[2]];
876 }
877
878 //5D
879 template <typename T>
880 T& ViewFArray<T>::operator()(size_t i,
881                              size_t j,
882                              size_t k,
```

```
883                             size_t l,
884                             size_t m) const
885 {
886     assert(order_ == 5 && "Tensor order (rank) does not match constructor in ViewFArray 5D!");
887     assert(i >= 0 && i < dims_[0] && "i is out of bounds in ViewFArray 5D!");
888     assert(j >= 0 && j < dims_[1] && "j is out of bounds in ViewFArray 5D!");
889     assert(k >= 0 && k < dims_[2] && "k is out of bounds in ViewFArray 5D!");
890     assert(l >= 0 && l < dims_[3] && "l is out of bounds in ViewFArray 5D!");
891     assert(m >= 0 && m < dims_[4] && "m is out of bounds in ViewFArray 5D!");
892     return array_[i + j*dims_[0]
893                     + k*dims_[0]*dims_[1]
894                     + l*dims_[0]*dims_[1]*dims_[2]
895                     + m*dims_[0]*dims_[1]*dims_[2]*dims_[3]];
896 }
897
898 //6D
899 template <typename T>
900 T& ViewFArray<T>:: operator()(size_t i,
901                               size_t j,
902                               size_t k,
903                               size_t l,
904                               size_t m,
905                               size_t n) const
906 {
907     assert(order_ == 6 && "Tensor order (rank) does not match constructor in ViewFArray 6D!");
908     assert(i >= 0 && i < dims_[0] && "i is out of bounds in ViewFArray 6D!");
909     assert(j >= 0 && j < dims_[1] && "j is out of bounds in ViewFArray 6D!");
910     assert(k >= 0 && k < dims_[2] && "k is out of bounds in ViewFArray 6D!");
911     assert(l >= 0 && l < dims_[3] && "l is out of bounds in ViewFArray 6D!");
912     assert(m >= 0 && m < dims_[4] && "m is out of bounds in ViewFArray 6D!");
913     assert(n >= 0 && n < dims_[5] && "n is out of bounds in ViewFArray 6D!");
914     return array_[i + j*dims_[0]
915                     + k*dims_[0]*dims_[1]
916                     + l*dims_[0]*dims_[1]*dims_[2]
917                     + m*dims_[0]*dims_[1]*dims_[2]*dims_[3]
918                     + n*dims_[0]*dims_[1]*dims_[2]*dims_[3]*dims_[4]];
919 }
920
921 //7D
922 template <typename T>
923 T& ViewFArray<T>:: operator()(size_t i,
924                               size_t j,
925                               size_t k,
926                               size_t l,
927                               size_t m,
928                               size_t n,
929                               size_t o) const
930 {
931     assert(order_ == 7 && "Tensor order (rank) does not match constructor in ViewFArray 7D!");
932     assert(i >= 0 && i < dims_[0] && "i is out of bounds in ViewFArray 7D!");
933     assert(j >= 0 && j < dims_[1] && "j is out of bounds in ViewFArray 7D!");
934     assert(k >= 0 && k < dims_[2] && "k is out of bounds in ViewFArray 7D!");
935     assert(l >= 0 && l < dims_[3] && "l is out of bounds in ViewFArray 7D!");
936     assert(m >= 0 && m < dims_[4] && "m is out of bounds in ViewFArray 7D!");
937     assert(n >= 0 && n < dims_[5] && "n is out of bounds in ViewFArray 7D!");
938     assert(o >= 0 && o < dims_[6] && "n is out of bounds in ViewFArray 7D!");
939     return array_[i + j*dims_[0]
940                     + k*dims_[0]*dims_[1]
941                     + l*dims_[0]*dims_[1]*dims_[2]
942                     + m*dims_[0]*dims_[1]*dims_[2]*dims_[3]
943                     + n*dims_[0]*dims_[1]*dims_[2]*dims_[3]*dims_[4]
944                     + o*dims_[0]*dims_[1]*dims_[2]*dims_[3]*dims_[4]*dims_[5]];
945 }
946
947 // calculate this ViewFArray object = math(A,B)
948 template <typename T>
949 template <typename M>
950 void ViewFArray<T>::operator=(M do_this_math){
951     do_this_math(*this); // pass in this ViewFArray object
952 }// end of math opperation
953
954 template <typename T>
955 inline size_t ViewFArray<T>::dims(size_t i) const {
956     assert(i < order_ && "ViewFArray order (rank) does not match constructor, dim[i] does not exist!");
957     assert(i >= 0 && dims_[i]>0 && "Access to ViewFArray dims is out of bounds!");
958     return dims_[i];
959 }
960
961 template <typename T>
962 inline size_t ViewFArray<T>::order() const {
963     return order_;
964 }
965
966 template <typename T>
967 inline size_t ViewFArray<T>::size() const {
968     return length_;
969 }
```

```
970
971 template <typename T>
972 inline T* ViewFArray<T>::pointer() const {
973     return array_;
974 }
975
976 //---end of ViewFArray class definitions---
977
978
979 //3. FMatrix
980 // indicies are [1:N]
981 template <typename T>
982 class FMatrix {
983 private:
984     size_t dims_[7];
985     size_t length_; // Length of 1D array
986     size_t order_;  // tensor order (rank)
987     std::shared_ptr <T []> matrix_;
988
989 public:
990     // Default constructor
991     FMatrix ();
992
993     //---1D to 7D matrix ---
994     FMatrix (size_t dim1);
995
996     FMatrix (size_t dim1,
997              size_t dim2);
998
999     FMatrix (size_t dim1,
1000              size_t dim2,
1001              size_t dim3);
1002
1003     FMatrix (size_t dim1,
1004              size_t dim2,
1005              size_t dim3,
1006              size_t dim4);
1007
1008     FMatrix (size_t dim1,
1009              size_t dim2,
1010              size_t dim3,
1011              size_t dim4,
1012              size_t dim5);
1013
1014     FMatrix (size_t dim1,
1015              size_t dim2,
1016              size_t dim3,
1017              size_t dim4,
1018              size_t dim5,
1019              size_t dim6);
1020
1021     FMatrix (size_t dim1,
1022              size_t dim2,
1023              size_t dim3,
1024              size_t dim4,
1025              size_t dim5,
1026              size_t dim6,
1027              size_t dim7);
1028
1029     FMatrix (const FMatrix& temp);
1030
1031     T& operator() (size_t i) const;
1032
1033     T& operator() (size_t i,
1034                    size_t j) const;
1035
1036     T& operator() (size_t i,
1037                    size_t j,
1038                    size_t k) const;
1039
1040     T& operator() (size_t i,
1041                    size_t j,
1042                    size_t k,
1043                    size_t l) const;
1044
1045     T& operator() (size_t i,
1046                    size_t j,
1047                    size_t k,
1048                    size_t l,
1049                    size_t m) const;
1050
1051     T& operator() (size_t i,
1052                    size_t j,
1053                    size_t k,
1054                    size_t l,
1055                    size_t m,
1056                    size_t n) const;
```

```
1057
1058     T& operator() (size_t i,
1059                    size_t j,
1060                    size_t k,
1061                    size_t l,
1062                    size_t m,
1063                    size_t n,
1064                    size_t o) const;
1065
1066
1067     // Overload copy assignment operator
1068     FMatrix& operator=(const FMatrix& temp);
1069
1070     // the length of the 1D storage array
1071     size_t size() const;
1072
1073     // matrix dims
1074     size_t dims(size_t i) const;
1075
1076     // return matrix order (rank)
1077     size_t order() const;
1078
1079     //return pointer
1080     T* pointer() const;
1081
1082     // Deconstructor
1083     ~FMatrix ();
1084
1085 }; // End of FMatrix
1086
1087 //---FMatrix class definitions---
1088
1089 //constructors
1090 template <typename T>
1091 FMatrix<T>::FMatrix(){
1092     matrix_ = NULL;
1093     length_ = 0;
1094 }
1095
1096 //1D
1097 template <typename T>
1098 FMatrix<T>::FMatrix(size_t dim1)
1099 {
1100     dims_[0] = dim1;
1101     order_ = 1;
1102     length_ = dim1;
1103     matrix_ = std::shared_ptr <T []> (new T[length_]);
1104 }
1105
1106 //2D
1107 template <typename T>
1108 FMatrix<T>::FMatrix(size_t dim1,
1109                     size_t dim2)
1110 {
1111     dims_[0] = dim1;
1112     dims_[1] = dim2;
1113     order_ = 2;
1114     length_ = dim1 * dim2;
1115     matrix_ = std::shared_ptr <T []> (new T[length_]);
1116 }
1117
1118 //3D
1119 template <typename T>
1120 FMatrix<T>::FMatrix(size_t dim1,
1121                     size_t dim2,
1122                     size_t dim3)
1123 {
1124     dims_[0] = dim1;
1125     dims_[1] = dim2;
1126     dims_[2] = dim3;
1127     order_ = 3;
1128     length_ = dim1 * dim2 * dim3;
1129     matrix_ = std::shared_ptr <T []> (new T[length_]);
1130 }
1131
1132 //4D
1133 template <typename T>
1134 FMatrix<T>::FMatrix(size_t dim1,
1135                     size_t dim2,
1136                     size_t dim3,
1137                     size_t dim4)
1138 {
1139     dims_[0] = dim1;
1140     dims_[1] = dim2;
1141     dims_[2] = dim3;
1142     dims_[3] = dim4;
1143     order_ = 4;
```

```
1144     length_ = dim1 * dim2 * dim3 * dim4;
1145     matrix_ = std::shared_ptr <T []> (new T[length_]);
1146 }
1147
1148 //5D
1149 template <typename T>
1150 FMatrix<T>::FMatrix(size_t dim1,
1151                     size_t dim2,
1152                     size_t dim3,
1153                     size_t dim4,
1154                     size_t dim5)
1155 {
1156     dims_[0] = dim1;
1157     dims_[1] = dim2;
1158     dims_[2] = dim3;
1159     dims_[3] = dim4;
1160     dims_[4] = dim5;
1161     order_ = 5;
1162     length_ = dim1 * dim2 * dim3 * dim4 * dim5;
1163     matrix_ = std::shared_ptr <T []> (new T[length_]);
1164 }
1165
1166 //6D
1167 template <typename T>
1168 FMatrix<T>::FMatrix(size_t dim1,
1169                     size_t dim2,
1170                     size_t dim3,
1171                     size_t dim4,
1172                     size_t dim5,
1173                     size_t dim6)
1174 {
1175     dims_[0] = dim1;
1176     dims_[1] = dim2;
1177     dims_[2] = dim3;
1178     dims_[3] = dim4;
1179     dims_[4] = dim5;
1180     dims_[5] = dim6;
1181     order_ = 6;
1182     length_ = dim1 * dim2 * dim3 * dim4 * dim5 * dim6;
1183     matrix_ = std::shared_ptr <T []> (new T[length_]);
1184
1185 }
1186
1187 template <typename T>
1188 FMatrix<T>::FMatrix(size_t dim1,
1189                     size_t dim2,
1190                     size_t dim3,
1191                     size_t dim4,
1192                     size_t dim5,
1193                     size_t dim6,
1194                     size_t dim7)
1195 {
1196     dims_[0] = dim1;
1197     dims_[1] = dim2;
1198     dims_[2] = dim3;
1199     dims_[3] = dim4;
1200     dims_[4] = dim5;
1201     dims_[5] = dim6;
1202     dims_[6] = dim7;
1203     order_ = 7;
1204     length_ = dim1 * dim2 * dim3 * dim4 * dim5 * dim6 * dim7;
1205     matrix_ = std::shared_ptr <T []> (new T[length_]);
1206
1207 }
1208
1209 template <typename T>
1210 FMatrix<T>::FMatrix(const FMatrix& temp) {
1211
1212     // Do nothing if the assignment is of the form x = x
1213
1214     if (this != &temp) {
1215         for (int iter = 0; iter < temp.order_; iter++){
1216             dims_[iter] = temp.dims_[iter];
1217         } // end for
1218
1219         order_  = temp.order_;
1220         length_ = temp.length_;
1221         matrix_ = temp.matrix_;
1222     } // end if
1223
1224 } // end constructor
1225
1226
1227 //overload operators
1228
1229 //1D
1230 template <typename T>
```

```
1231 inline T& FMatrix<T>::operator() (size_t i) const
1232 {
1233     assert(order_ == 1 && "Tensor order (rank) does not match constructor in FMatrix 1D!");
1234     assert(i >= 1 && i <= dims_[0] && "i is out of bounds in FMatrix 1D!");
1235     return matrix_[i - 1];
1236 }
1237
1238 //2D
1239 template <typename T>
1240 inline T& FMatrix<T>::operator() (size_t i,
1241                                   size_t j) const
1242 {
1243     assert(order_ == 2 && "Tensor order (rank) does not match constructor in FMatrix 2D!");
1244     assert(i >= 1 && i <= dims_[0] && "i is out of bounds in FMatrix 2D!");
1245     assert(j >= 1 && j <= dims_[1] && "j is out of bounds in FMatrix 2D!");
1246     return matrix_[(i - 1) + ((j - 1) * dims_[0])];
1247 }
1248
1249 //3D
1250 template <typename T>
1251 inline T& FMatrix<T>::operator() (size_t i,
1252                                   size_t j,
1253                                   size_t k) const
1254 {
1255     assert(order_ == 3 && "Tensor order (rank) does not match constructor in FMatrix 3D!");
1256     assert(i >= 1 && i <= dims_[0] && "i is out of bounds in FMatrix 3D!");
1257     assert(j >= 1 && j <= dims_[1] && "j is out of bounds in FMatrix 3D!");
1258     assert(k >= 1 && k <= dims_[2] && "k is out of bounds in FMatrix 3D!");
1259     return matrix_[(i - 1) + ((j - 1) * dims_[0])
1260                            + ((k - 1) * dims_[0] * dims_[1])];
1261 }
1262
1263 //4D
1264 template <typename T>
1265 inline T& FMatrix<T>::operator() (size_t i,
1266                                   size_t j,
1267                                   size_t k,
1268                                   size_t l) const
1269 {
1270     assert(order_ == 4 && "Tensor order (rank) does not match constructor in FMatrix 4D!");
1271     assert(i >= 1 && i <= dims_[0] && "i is out of bounds in FMatrix 4D!");
1272     assert(j >= 1 && j <= dims_[1] && "j is out of bounds in FMatrix 4D!");
1273     assert(k >= 1 && k <= dims_[2] && "k is out of bounds in FMatrix 4D!");
1274     assert(l >= 1 && l <= dims_[3] && "l is out of bounds in FMatrix 4D!");
1275     return matrix_[(i - 1) + ((j - 1) * dims_[0])
1276                            + ((k - 1) * dims_[0] * dims_[1])
1277                            + ((l - 1) * dims_[0] * dims_[1] * dims_[2])];
1278 }
1279
1280 //5D
1281 template <typename T>
1282 inline T& FMatrix<T>::operator() (size_t i,
1283                                   size_t j,
1284                                   size_t k,
1285                                   size_t l,
1286                                   size_t m) const
1287 {
1288     assert(order_ == 5 && "Tensor order (rank) does not match constructor in FMatrix 5D!");
1289     assert(i >= 1 && i <= dims_[0] && "i is out of bounds in FMatrix 5D!");
1290     assert(j >= 1 && j <= dims_[1] && "j is out of bounds in FMatrix 5D!");
1291     assert(k >= 1 && k <= dims_[2] && "k is out of bounds in FMatrix 5D!");
1292     assert(l >= 1 && l <= dims_[3] && "l is out of bounds in FMatrix 5D!");
1293     assert(m >= 1 && m <= dims_[4] && "m is out of bounds in FMatrix 5D!");
1294     return matrix_[(i - 1) + ((j - 1) * dims_[0])
1295                            + ((k - 1) * dims_[0] * dims_[1])
1296                            + ((l - 1) * dims_[0] * dims_[1] * dims_[2])
1297                            + ((m - 1) * dims_[0] * dims_[1] * dims_[2] * dims_[3])];
1298 }
1299
1300 //6D
1301 template <typename T>
1302 inline T& FMatrix<T>::operator() (size_t i,
1303                                   size_t j,
1304                                   size_t k,
1305                                   size_t l,
1306                                   size_t m,
1307                                   size_t n) const
1308 {
1309     assert(order_ == 6 && "Tensor order (rank) does not match constructor in FMatrix 6D!");
1310     assert(i >= 1 && i <= dims_[0] && "i is out of bounds in FMatrix 6D!");
1311     assert(j >= 1 && j <= dims_[1] && "j is out of bounds in FMatrix 6D!");
1312     assert(k >= 1 && k <= dims_[2] && "k is out of bounds in FMatrix 6D!");
1313     assert(l >= 1 && l <= dims_[3] && "l is out of bounds in FMatrix 6D!");
1314     assert(m >= 1 && m <= dims_[4] && "m is out of bounds in FMatrix 6D!");
1315     assert(n >= 1 && n <= dims_[5] && "n is out of bounds in FMatrix 6D!");
1316     return matrix_[(i - 1) + ((j - 1) * dims_[0])
1317                            + ((k - 1) * dims_[0] * dims_[1])
```

```
1318                            + ((l - 1) * dims_[0] * dims_[1] * dims_[2])
1319                            + ((m - 1) * dims_[0] * dims_[1] * dims_[2] * dims_[3])
1320                            + ((n - 1) * dims_[0] * dims_[1] * dims_[2] * dims_[3] * dims_[4])];
1321 }
1322
1323 //7D
1324 template <typename T>
1325 inline T& FMatrix<T>::operator() (size_t i,
1326                                   size_t j,
1327                                   size_t k,
1328                                   size_t l,
1329                                   size_t m,
1330                                   size_t n,
1331                                   size_t o) const
1332 {
1333     assert(order_ == 7 && "Tensor order (rank) does not match constructor in FMatrix 7D!");
1334     assert(i >= 1 && i <= dims_[0] && "i is out of bounds in FMatrix 7D!");
1335     assert(j >= 1 && j <= dims_[1] && "j is out of bounds in FMatrix 7D!");
1336     assert(k >= 1 && k <= dims_[2] && "k is out of bounds in FMatrix 7D!");
1337     assert(l >= 1 && l <= dims_[3] && "l is out of bounds in FMatrix 7D!");
1338     assert(m >= 1 && m <= dims_[4] && "m is out of bounds in FMatrix 7D!");
1339     assert(n >= 1 && n <= dims_[5] && "n is out of bounds in FMatrix 7D!");
1340     assert(o >= 1 && o <= dims_[6] && "o is out of bounds in FMatrix 7D!");
1341     return matrix_[(i - 1) + ((j - 1) * dims_[0])
1342                            + ((k - 1) * dims_[0] * dims_[1])
1343                            + ((l - 1) * dims_[0] * dims_[1] * dims_[2])
1344                            + ((m - 1) * dims_[0] * dims_[1] * dims_[2] * dims_[3])
1345                            + ((n - 1) * dims_[0] * dims_[1] * dims_[2] * dims_[3] * dims_[4])
1346                            + ((o - 1) * dims_[0] * dims_[1] * dims_[2] * dims_[3] * dims_[4] *
      dims_[5])];
1347 }
1348
1349
1350 template <typename T>
1351 inline FMatrix<T>& FMatrix<T>::operator= (const FMatrix& temp)
1352 {
1353     // Do nothing if assignment is of the form x = x
1354     if (this != &temp) {
1355         for (int iter = 0; iter < temp.order_; iter++){
1356             dims_[iter] = temp.dims_[iter];
1357         } // end for
1358
1359         order_  = temp.order_;
1360         length_ = temp.length_;
1361     matrix_ = temp.matrix_;
1362     }
1363
1364     return *this;
1365 }
1366
1367 template <typename T>
1368 inline size_t FMatrix<T>::size() const {
1369     return length_;
1370 }
1371
1372 template <typename T>
1373 inline size_t FMatrix<T>::dims(size_t i) const {
1374     i--; // i starts at 1
1375     assert(i < order_ && "FMatrix order (rank) does not match constructor, dim[i] does not exist!");
1376     assert(i >= 0 && dims_[i]>0 && "Access to FMatrix dims is out of bounds!");
1377     return dims_[i];
1378 }
1379
1380 template <typename T>
1381 inline size_t FMatrix<T>::order() const {
1382     return order_;
1383 }
1384
1385 template <typename T>
1386 inline T* FMatrix<T>::pointer() const{
1387     return matrix_.get();
1388 }
1389
1390 template <typename T>
1391 FMatrix<T>::~FMatrix() {}
1392
1393 //----end of FMatrix class definitions----
1394
1395
1396 //4. ViewFMatrix
1397 //  indices are [1:N]
1398 template <typename T>
1399 class ViewFMatrix {
1400
1401 private:
1402     size_t dims_[7];
1403     size_t length_; // Length of 1D array
```

```
1404     size_t order_;  // tensor order (rank)
1405     T * matrix_;
1406
1407 public:
1408
1409     // Default constructor
1410     ViewFMatrix ();
1411
1412     //--- 1D to 7D matrix ---
1413
1414     ViewFMatrix(T *matrix,
1415                 size_t dim1);
1416
1417     ViewFMatrix(T *some_matrix,
1418                 size_t dim1,
1419                 size_t dim2);
1420
1421     ViewFMatrix(T *matrix,
1422                 size_t dim1,
1423                 size_t dim2,
1424                 size_t dim3);
1425
1426     ViewFMatrix(T *matrix,
1427                 size_t dim1,
1428                 size_t dim2,
1429                 size_t dim3,
1430                 size_t dim4);
1431
1432     ViewFMatrix (T *matrix,
1433                  size_t dim1,
1434                  size_t dim2,
1435                  size_t dim3,
1436                  size_t dim4,
1437                  size_t dim5);
1438
1439     ViewFMatrix (T *matrix,
1440                  size_t dim1,
1441                  size_t dim2,
1442                  size_t dim3,
1443                  size_t dim4,
1444                  size_t dim5,
1445                  size_t dim6);
1446
1447     ViewFMatrix (T *matrix,
1448                  size_t dim1,
1449                  size_t dim2,
1450                  size_t dim3,
1451                  size_t dim4,
1452                  size_t dim5,
1453                  size_t dim6,
1454                  size_t dim7);
1455
1456     T& operator()(size_t i) const;
1457
1458     T& operator()(size_t i,
1459                   size_t j) const;
1460
1461     T& operator()(size_t i,
1462                   size_t j,
1463                   size_t k) const;
1464
1465     T& operator()(size_t i,
1466                   size_t j,
1467                   size_t k,
1468                   size_t l) const;
1469
1470     T& operator() (size_t i,
1471                    size_t j,
1472                    size_t k,
1473                    size_t l,
1474                    size_t m) const;
1475
1476     T& operator()(size_t i,
1477                   size_t j,
1478                   size_t k,
1479                   size_t l,
1480                   size_t m,
1481                   size_t n) const;
1482
1483     T& operator()(size_t i,
1484                   size_t j,
1485                   size_t k,
1486                   size_t l,
1487                   size_t m,
1488                   size_t n,
1489                   size_t o) const;
1490
```

```
1491     // calculate C = math(A,B)
1492     template <typename M>
1493     void operator=(M do_this_math);
1494
1495     // length of 1D array
1496     size_t size() const;
1497
1498     // matrix dims
1499     size_t dims(size_t i) const;
1500
1501     // return matrix order (rank)
1502     size_t order() const;
1503
1504     // return pointer
1505     T* pointer() const;
1506
1507 }; // end of ViewFMatrix
1508
1509 //constructors
1510
1511 //no dimension
1512 template <typename T>
1513 ViewFMatrix<T>::ViewFMatrix() {
1514   matrix_ = NULL;
1515   length_ = 0;
1516 }
1517
1518 //1D
1519 template <typename T>
1520 ViewFMatrix<T>::ViewFMatrix(T *matrix,
1521                             size_t dim1)
1522 {
1523     dims_[0] = dim1;
1524     order_ = 1;
1525     length_ = dim1;
1526     matrix_ = matrix;
1527 }
1528
1529 //2D
1530 template <typename T>
1531 ViewFMatrix<T>::ViewFMatrix(T *matrix,
1532                             size_t dim1,
1533                             size_t dim2)
1534 {
1535     dims_[0] = dim1;
1536     dims_[1] = dim2;
1537     order_ = 2;
1538     length_ = dim1 * dim2;
1539     matrix_ = matrix;
1540 }
1541
1542 //3D
1543 template <typename T>
1544 ViewFMatrix<T>::ViewFMatrix (T *matrix,
1545                              size_t dim1,
1546                              size_t dim2,
1547                              size_t dim3)
1548 {
1549     dims_[0] = dim1;
1550     dims_[1] = dim2;
1551     dims_[2] = dim3;
1552     order_ = 3;
1553     length_ = dim1 * dim2 * dim3;
1554     matrix_ = matrix;
1555 }
1556
1557 //4D
1558 template <typename T>
1559 ViewFMatrix<T>::ViewFMatrix(T *matrix,
1560                             size_t dim1,
1561                             size_t dim2,
1562                             size_t dim3,
1563                             size_t dim4)
1564 {
1565     dims_[0] = dim1;
1566     dims_[1] = dim2;
1567     dims_[2] = dim3;
1568     dims_[3] = dim4;
1569     order_ = 4;
1570     length_ = dim1 * dim2 * dim3 * dim4;
1571     matrix_ = matrix;
1572 }
1573
1574 //5D
1575 template <typename T>
1576 ViewFMatrix<T>::ViewFMatrix(T *matrix,
1577                             size_t dim1,
```

```
1578                                size_t dim2,
1579                                size_t dim3,
1580                                size_t dim4,
1581                                size_t dim5)
1582 {
1583     dims_[0] = dim1;
1584     dims_[1] = dim2;
1585     dims_[2] = dim3;
1586     dims_[3] = dim4;
1587     dims_[4] = dim5;
1588     order_ = 5;
1589     length_ = dim1 * dim2 * dim3 * dim4 * dim5;
1590     matrix_ = matrix;
1591 }
1592
1593 //6D
1594 template <typename T>
1595 ViewFMatrix<T>::ViewFMatrix(T *matrix,
1596                                size_t dim1,
1597                                size_t dim2,
1598                                size_t dim3,
1599                                size_t dim4,
1600                                size_t dim5,
1601                                size_t dim6)
1602 {
1603     dims_[0] = dim1;
1604     dims_[1] = dim2;
1605     dims_[2] = dim3;
1606     dims_[3] = dim4;
1607     dims_[4] = dim5;
1608     dims_[5] = dim6;
1609     order_ = 6;
1610     length_ = dim1 * dim2 * dim3 * dim4 * dim5 * dim6;
1611     matrix_ = matrix;
1612 }
1613
1614 //6D
1615 template <typename T>
1616 ViewFMatrix<T>::ViewFMatrix(T *matrix,
1617                                size_t dim1,
1618                                size_t dim2,
1619                                size_t dim3,
1620                                size_t dim4,
1621                                size_t dim5,
1622                                size_t dim6,
1623                                size_t dim7)
1624 {
1625     dims_[0] = dim1;
1626     dims_[1] = dim2;
1627     dims_[2] = dim3;
1628     dims_[3] = dim4;
1629     dims_[4] = dim5;
1630     dims_[5] = dim6;
1631     dims_[6] = dim7;
1632     order_ = 7;
1633     length_ = dim1 * dim2 * dim3 * dim4 * dim5 * dim6 * dim7;
1634     matrix_ = matrix;
1635 }
1636
1637
1638 //overload operator ()
1639
1640 //1D
1641 template <typename T>
1642 inline T& ViewFMatrix<T>::operator()(size_t i) const
1643 {
1644     assert(order_ == 1 && "Tensor order (rank) does not match constructor in ViewFMatrix 1D!");
1645     assert(i >= 1 && i <= dims_[0] && "i is out of bounds in ViewFMatrix 1D");  // die if >= dim1
1646
1647     return matrix_[(i - 1)];
1648 }
1649
1650 //2D
1651 template <typename T>
1652 inline T& ViewFMatrix<T>::operator()(size_t i,
1653                                      size_t j) const
1654 {
1655     assert(order_ == 2 && "Tensor order (rank) does not match constructor in ViewFMatrix 2D!");
1656     assert(i >= 1 && i <= dims_[0] && "i is out of bounds in ViewFMatrix 2D");  // die if >= dim1
1657     assert(j >= 1 && j <= dims_[1] && "j is out of bounds in ViewFMatrix 2D");  // die if >= dim2
1658
1659     return matrix_[(i - 1) + ((j - 1) * dims_[0])];
1660 }
1661
1662 //3D
1663 template <typename T>
1664 inline T& ViewFMatrix<T>::operator()(size_t i,
```

```
1665                                    size_t j,
1666                                    size_t k) const
1667 {
1668     assert(order_ == 3 && "Tensor order (rank) does not match constructor in ViewFMatrix 3D!");
1669     assert(i >= 1 && i <= dims_[0] && "i is out of bounds in ViewFMatrix 3D");  // die if >= dim1
1670     assert(j >= 1 && j <= dims_[1] && "j is out of bounds in ViewFMatrix 3D");  // die if >= dim2
1671     assert(k >= 1 && k <= dims_[2] && "k is out of bounds in ViewFMatrix 3D");  // die if >= dim3
1672
1673     return matrix_[(i - 1) + ((j - 1) * dims_[0])
1674                            + ((k - 1) * dims_[0] * dims_[1])];
1675 }
1676
1677 //4D
1678 template <typename T>
1679 inline T& ViewFMatrix<T>::operator()(size_t i,
1680                                    size_t j,
1681                                    size_t k,
1682                                    size_t l) const
1683 {
1684     assert(order_ == 4 && "Tensor order (rank) does not match constructor in ViewFMatrix 4D!");
1685     assert(i >= 1 && i <= dims_[0] && "i is out of bounds in ViewFMatrix 4D");  // die if >= dim1
1686     assert(j >= 1 && j <= dims_[1] && "j is out of bounds in ViewFMatrix 4D");  // die if >= dim2
1687     assert(k >= 1 && k <= dims_[2] && "k is out of bounds in ViewFMatrix 4D");  // die if >= dim3
1688     assert(l >= 1 && l <= dims_[3] && "l is out of bounds in ViewFMatrix 4D");  // die if >= dim4
1689
1690     return matrix_[(i - 1) + ((j - 1) * dims_[0])
1691                            + ((k - 1) * dims_[0] * dims_[1])
1692                            + ((l - 1) * dims_[0] * dims_[1] * dims_[2])];
1693 }
1694
1695 //5D
1696 template <typename T>
1697 inline T& ViewFMatrix<T>::operator()(size_t i,
1698                                    size_t j,
1699                                    size_t k,
1700                                    size_t l,
1701                                    size_t m) const
1702 {
1703     assert(order_ == 5 && "Tensor order (rank) does not match constructor in ViewFMatrix 5D!");
1704     assert(i >= 1 && i <= dims_[0] && "i is out of bounds in ViewFMatrix 5D");  // die if >= dim1
1705     assert(j >= 1 && j <= dims_[1] && "j is out of bounds in ViewFMatrix 5D");  // die if >= dim2
1706     assert(k >= 1 && k <= dims_[2] && "k is out of bounds in ViewFMatrix 5D");  // die if >= dim3
1707     assert(l >= 1 && l <= dims_[3] && "l is out of bounds in ViewFMatrix 5D");  // die if >= dim4
1708     assert(m >= 1 && m <= dims_[4] && "m is out of bounds in ViewFMatrix 5D");  // die if >= dim5
1709
1710     return matrix_[(i - 1) + ((j - 1) * dims_[0])
1711                            + ((k - 1) * dims_[0] * dims_[1])
1712                            + ((l - 1) * dims_[0] * dims_[1] * dims_[2])
1713                            + ((m - 1) * dims_[0] * dims_[1] * dims_[2] * dims_[3])];
1714 }
1715
1716 //6D
1717 template <typename T>
1718 inline T& ViewFMatrix<T>::operator()(size_t i,
1719                                    size_t j,
1720                                    size_t k,
1721                                    size_t l,
1722                                    size_t m,
1723                                    size_t n) const
1724 {
1725     assert(order_ == 6 && "Tensor order (rank) does not match constructor in ViewFMatrix 6D!");
1726     assert(i >= 1 && i <= dims_[0] && "i is out of bounds in ViewFMatrix 6D");  // die if >= dim1
1727     assert(j >= 1 && j <= dims_[1] && "j is out of bounds in ViewFMatrix 6D");  // die if >= dim2
1728     assert(k >= 1 && k <= dims_[2] && "k is out of bounds in ViewFMatrix 6D");  // die if >= dim3
1729     assert(l >= 1 && l <= dims_[3] && "l is out of bounds in ViewFMatrix 6D");  // die if >= dim4
1730     assert(m >= 1 && m <= dims_[4] && "m is out of bounds in ViewFMatrix 6D");  // die if >= dim5
1731     assert(n >= 1 && n <= dims_[5] && "n is out of bounds in ViewFMatrix 6D");  // die if >= dim6
1732     return matrix_[(i - 1) + ((j - 1) * dims_[0])
1733                            + ((k - 1) * dims_[0] * dims_[1])
1734                            + ((l - 1) * dims_[0] * dims_[1] * dims_[2])
1735                            + ((m - 1) * dims_[0] * dims_[1] * dims_[2] * dims_[3])
1736                            + ((n - 1) * dims_[0] * dims_[1] * dims_[2] * dims_[3] * dims_[4])];
1737 }
1738
1739 //6D
1740 template <typename T>
1741 inline T& ViewFMatrix<T>::operator()(size_t i,
1742                                    size_t j,
1743                                    size_t k,
1744                                    size_t l,
1745                                    size_t m,
1746                                    size_t n,
1747                                    size_t o) const
1748 {
1749     assert(order_ == 7 && "Tensor order (rank) does not match constructor in ViewFMatrix 7D!");
1750     assert(i >= 1 && i <= dims_[0] && "i is out of bounds in ViewFMatrix 7D");  // die if >= dim1
1751     assert(j >= 1 && j <= dims_[1] && "j is out of bounds in ViewFMatrix 7D");  // die if >= dim2
```

```
1752     assert(k >= 1 && k <= dims_[2] && "k is out of bounds in ViewFMatrix 7D");  // die if >= dim3
1753     assert(l >= 1 && l <= dims_[3] && "l is out of bounds in ViewFMatrix 7D");  // die if >= dim4
1754     assert(m >= 1 && m <= dims_[4] && "m is out of bounds in ViewFMatrix 7D");  // die if >= dim5
1755     assert(n >= 1 && n <= dims_[5] && "n is out of bounds in ViewFMatrix 7D");  // die if >= dim6
1756     assert(o >= 1 && o <= dims_[6] && "o is out of bounds in ViewFMatrix 7D");  // die if >= dim7
1757
1758     return matrix_[(i - 1) + ((j - 1) * dims_[0])
1759                            + ((k - 1) * dims_[0] * dims_[1])
1760                            + ((l - 1) * dims_[0] * dims_[1] * dims_[2])
1761                            + ((m - 1) * dims_[0] * dims_[1] * dims_[2] * dims_[3])
1762                            + ((n - 1) * dims_[0] * dims_[1] * dims_[2] * dims_[3] * dims_[4])
1763                            + ((o - 1) * dims_[0] * dims_[1] * dims_[2] * dims_[3] * dims_[4] *
     dims_[5])];
1764 }
1765
1766 // calculate this ViewFMatrix object = math(A,B)
1767 template <typename T>
1768 template <typename M>
1769 void ViewFMatrix<T>::operator=(M do_this_math){
1770     do_this_math(*this); // pass in this ViewFArray object
1771 }// end of math opperation
1772
1773 template <typename T>
1774 inline size_t ViewFMatrix<T>::dims(size_t i) const {
1775     i--; // i starts at 1
1776     assert(i < order_ && "ViewFMatrix order (rank) does not match constructor, dim[i] does not
     exist!");
1777     assert(i >= 0 && dims_[i]>0 && "Access to ViewFMatrix dims is out of bounds!");
1778     return dims_[i];
1779 }
1780
1781 template <typename T>
1782 inline size_t ViewFMatrix<T>::order() const {
1783     return order_;
1784 }
1785
1786 template <typename T>
1787 inline T* ViewFMatrix<T>::pointer() const {
1788     return matrix_;
1789 }
1790 //-----end ViewFMatrix-----
1791
1792
1793 //5. CArray
1794 // indicies are [0:N-1]
1795 template <typename T>
1796 class CArray {
1797
1798 private:
1799     size_t dims_[7];
1800     size_t length_; // Length of 1D array
1801     size_t order_;  // tensor order (rank)
1802     std::shared_ptr <T []> array_;
1803
1804 public:
1805     // Default constructor
1806     CArray ();
1807
1808     // --- 1D to 7D array ---
1809
1810     CArray (size_t dim0);
1811
1812     CArray (size_t dim0,
1813             size_t dim1);
1814
1815     CArray (size_t dim0,
1816             size_t dim1,
1817             size_t dim2);
1818
1819     CArray (size_t dim0,
1820             size_t dim1,
1821             size_t dim2,
1822             size_t dim3);
1823
1824     CArray (size_t dim0,
1825             size_t dim1,
1826             size_t dim2,
1827             size_t dim3,
1828             size_t dim4);
1829
1830     CArray (size_t dim0,
1831             size_t dim1,
1832             size_t dim2,
1833             size_t dim3,
1834             size_t dim4,
1835             size_t dim5);
1836
```

```
1837     CArray (size_t dim0,
1838             size_t dim1,
1839             size_t dim2,
1840             size_t dim3,
1841             size_t dim4,
1842             size_t dim5,
1843             size_t dim6);
1844
1845     CArray (const CArray& temp);
1846
1847     // Overload operator()
1848     T& operator() (size_t i) const;
1849
1850     T& operator() (size_t i,
1851                    size_t j) const;
1852
1853     T& operator() (size_t i,
1854                    size_t j,
1855                    size_t k) const;
1856
1857     T& operator() (size_t i,
1858                    size_t j,
1859                    size_t k,
1860                    size_t l) const;
1861
1862     T& operator() (size_t i,
1863                    size_t j,
1864                    size_t k,
1865                    size_t l,
1866                    size_t m) const;
1867
1868     T& operator() (size_t i,
1869                    size_t j,
1870                    size_t k,
1871                    size_t l,
1872                    size_t m,
1873                    size_t n) const;
1874
1875     T& operator() (size_t i,
1876                    size_t j,
1877                    size_t k,
1878                    size_t l,
1879                    size_t m,
1880                    size_t n,
1881                    size_t o) const;
1882
1883     // Overload copy assignment operator
1884     CArray& operator= (const CArray& temp);
1885
1886      //return array size
1887     size_t size() const;
1888
1889     // return array dims
1890     size_t dims(size_t i) const;
1891
1892     // return array order (rank)
1893     size_t order() const;
1894
1895     //return pointer
1896     T* pointer() const;
1897
1898     // Deconstructor
1899     ~CArray ();
1900
1901 }; // End of CArray
1902
1903 //---carray class declarations---
1904
1905 //constructors
1906
1907 //no dim
1908 template <typename T>
1909 CArray<T>::CArray() {
1910     array_ = NULL;
1911     length_ = order_ = 0;
1912 }
1913
1914 //1D
1915 template <typename T>
1916 CArray<T>::CArray(size_t dim0)
1917 {
1918     dims_[0] = dim0;
1919     order_ = 1;
1920     length_ = dim0;
1921     array_ = std::shared_ptr <T[]> (new T[length_]);
1922 }
1923
```

```
1924 //2D
1925 template <typename T>
1926 CArray<T>::CArray(size_t dim0,
1927                   size_t dim1)
1928 {
1929     dims_[0] = dim0;
1930     dims_[1] = dim1;
1931     order_ = 2;
1932     length_ = dim0 * dim1;
1933     array_ = std::shared_ptr <T[]> (new T[length_]);
1934 }
1935
1936 //3D
1937 template <typename T>
1938 CArray<T>::CArray(size_t dim0,
1939                   size_t dim1,
1940                   size_t dim2)
1941 {
1942     dims_[0] = dim0;
1943     dims_[1] = dim1;
1944     dims_[2] = dim2;
1945     order_ = 3;
1946     length_ = dim0 * dim1 * dim2;
1947     array_ = std::shared_ptr <T[]> (new T[length_]);
1948 }
1949
1950 //4D
1951 template <typename T>
1952 CArray<T>::CArray(size_t dim0,
1953                   size_t dim1,
1954                   size_t dim2,
1955                   size_t dim3)
1956 {
1957     dims_[0] = dim0;
1958     dims_[1] = dim1;
1959     dims_[2] = dim2;
1960     dims_[3] = dim3;
1961     order_ = 4;
1962     length_ = dim0 * dim1 * dim2 * dim3;
1963     array_ = std::shared_ptr <T[]> (new T[length_]);
1964 }
1965
1966 //5D
1967 template <typename T>
1968 CArray<T>::CArray(size_t dim0,
1969                   size_t dim1,
1970                   size_t dim2,
1971                   size_t dim3,
1972                   size_t dim4) {
1973     dims_[0] = dim0;
1974     dims_[1] = dim1;
1975     dims_[2] = dim2;
1976     dims_[3] = dim3;
1977     dims_[4] = dim4;
1978     order_ = 5;
1979     length_ = dim0 * dim1 * dim2 * dim3 * dim4;
1980     array_ = std::shared_ptr <T[]> (new T[length_]);
1981 }
1982
1983 //6D
1984 template <typename T>
1985 CArray<T>::CArray(size_t dim0,
1986                   size_t dim1,
1987                   size_t dim2,
1988                   size_t dim3,
1989                   size_t dim4,
1990                   size_t dim5) {
1991     dims_[0] = dim0;
1992     dims_[1] = dim1;
1993     dims_[2] = dim2;
1994     dims_[3] = dim3;
1995     dims_[4] = dim4;
1996     dims_[5] = dim5;
1997     order_ = 6;
1998     length_ = dim0 * dim1 * dim2 * dim3 * dim4 * dim5;
1999     array_ = std::shared_ptr <T[]> (new T[length_]);
2000 }
2001
2002 //7D
2003 template <typename T>
2004 CArray<T>::CArray(size_t dim0,
2005                   size_t dim1,
2006                   size_t dim2,
2007                   size_t dim3,
2008                   size_t dim4,
2009                   size_t dim5,
2010                   size_t dim6) {
```

```
2011     dims_[0] = dim0;
2012     dims_[1] = dim1;
2013     dims_[2] = dim2;
2014     dims_[3] = dim3;
2015     dims_[4] = dim4;
2016     dims_[5] = dim5;
2017     dims_[6] = dim6;
2018     order_ = 7;
2019     length_ = dim0 * dim1 * dim2 * dim3 * dim4 * dim5 * dim6;
2020     array_ = std::shared_ptr <T[]> (new T[length_]);
2021 }
2022
2023 //Copy constructor
2024
2025 template <typename T>
2026 CArray<T>::CArray(const CArray& temp) {
2027
2028     // Do nothing if the assignment is of the form x = x
2029
2030     if (this != &temp) {
2031         for (int iter = 0; iter < temp.order_; iter++){
2032             dims_[iter] = temp.dims_[iter];
2033         } // end for
2034
2035         order_  = temp.order_;
2036         length_ = temp.length_;
2037         array_ = temp.array_;
2038     } // end if
2039
2040 } // end constructor
2041
2042
2043 //overload () operator
2044
2045 //1D
2046 template <typename T>
2047 inline T& CArray<T>::operator() (size_t i) const
2048 {
2049     assert(order_ == 1 && "Tensor order (rank) does not match constructor in CArray 1D!");
2050     assert(i >= 0 && i < dims_[0] && "i is out of bounds in CArray 1D!");
2051
2052     return array_[i];
2053 }
2054
2055 //2D
2056 template <typename T>
2057 inline T& CArray<T>::operator() (size_t i,
2058                                  size_t j) const
2059 {
2060     assert(order_ == 2 && "Tensor order (rank) does not match constructor in CArray 2D!");
2061     assert(i >= 0 && i < dims_[0] && "i is out of bounds in CArray 2D!");
2062     assert(j >= 0 && j < dims_[1] && "j is out of bounds in CArray 2D!");
2063
2064     return array_[j + (i *  dims_[1])];
2065 }
2066
2067 //3D
2068 template <typename T>
2069 inline T& CArray<T>::operator() (size_t i,
2070                                  size_t j,
2071                                  size_t k) const
2072 {
2073     assert(order_ == 3 && "Tensor order (rank) does not match constructor in CArray 3D!");
2074     assert(i >= 0 && i < dims_[0] && "i is out of bounds in CArray 3D!");
2075     assert(j >= 0 && j < dims_[1] && "j is out of bounds in Carray 3D!");
2076     assert(k >= 0 && k < dims_[2] && "k is out of bounds in CArray 3D!");
2077
2078     return array_[k + (j * dims_[2])
2079                     + (i * dims_[2] *  dims_[1])];
2080 }
2081
2082 //4D
2083 template <typename T>
2084 inline T& CArray<T>::operator() (size_t i,
2085                                  size_t j,
2086                                  size_t k,
2087                                  size_t l) const
2088 {
2089     assert(order_ == 4 && "Tensor order (rank) does not match constructor in CArray 4D!");
2090     assert(i >= 0 && i < dims_[0] && "i is out of bounds in CArray 4D");  // die if >= dim0
2091     assert(j >= 0 && j < dims_[1] && "j is out of bounds in CArray 4D");  // die if >= dim1
2092     assert(k >= 0 && k < dims_[2] && "k is out of bounds in CArray 4D");  // die if >= dim2
2093     assert(l >= 0 && l < dims_[3] && "l is out of bounds in CArray 4D");  // die if >= dim3
2094
2095     return array_[l + (k * dims_[3])
2096                     + (j * dims_[3] * dims_[2])
2097                     + (i * dims_[3] * dims_[2] *  dims_[1])];
```

```
2098 }
2099
2100 //5D
2101 template <typename T>
2102 inline T& CArray<T>::operator() (size_t i,
2103                                  size_t j,
2104                                  size_t k,
2105                                  size_t l,
2106                                  size_t m) const
2107 {
2108     assert(order_ == 5 && "Tensor order (rank) does not match constructor in CArray 5D!");
2109     assert(i >= 0 && i < dims_[0] && "i is out of bounds in CArray 5D!");
2110     assert(j >= 0 && j < dims_[1] && "j is out of bounds in CArray 5D!");
2111     assert(k >= 0 && k < dims_[2] && "k is out of bounds in CArray 5D!");
2112     assert(l >= 0 && l < dims_[3] && "l is out of bounds in CArray 5D!");
2113     assert(m >= 0 && m < dims_[4] && "m is out of bounds in CArray 5D!");
2114
2115     return array_[m + (l * dims_[4])
2116                     + (k * dims_[4] * dims_[3])
2117                     + (j * dims_[4] * dims_[3] * dims_[2])
2118                     + (i * dims_[4] * dims_[3] * dims_[2] *  dims_[1])];
2119 }
2120
2121 //6D
2122 template <typename T>
2123 inline T& CArray<T>::operator() (size_t i,
2124                                  size_t j,
2125                                  size_t k,
2126                                  size_t l,
2127                                  size_t m,
2128                                  size_t n) const
2129 {
2130     assert(order_ == 6 && "Tensor order (rank) does not match constructor in CArray 6D!");
2131     assert(i >= 0 && i < dims_[0] && "i is out of bounds in CArray 6D!");
2132     assert(j >= 0 && j < dims_[1] && "j is out of bounds in CArray 6D!");
2133     assert(k >= 0 && k < dims_[2] && "k is out of bounds in CArray 6D!");
2134     assert(l >= 0 && l < dims_[3] && "l is out of bounds in CArray 6D!");
2135     assert(m >= 0 && m < dims_[4] && "m is out of bounds in CArray 6D!");
2136     assert(n >= 0 && n < dims_[5] && "n is out of bounds in CArray 6D!");
2137
2138     return array_[n + (m * dims_[5])
2139                     + (l * dims_[5] * dims_[4])
2140                     + (k * dims_[5] * dims_[4] * dims_[3])
2141                     + (j * dims_[5] * dims_[4] * dims_[3] * dims_[2])
2142                     + (i * dims_[5] * dims_[4] * dims_[3] * dims_[2] *  dims_[1])];
2143 }
2144
2145 //7D
2146 template <typename T>
2147 inline T& CArray<T>::operator() (size_t i,
2148                                  size_t j,
2149                                  size_t k,
2150                                  size_t l,
2151                                  size_t m,
2152                                  size_t n,
2153                                  size_t o) const
2154 {
2155     assert(order_ == 7 && "Tensor order (rank) does not match constructor in CArray 7D!");
2156     assert(i >= 0 && i < dims_[0] && "i is out of bounds in CArray 7D!");
2157     assert(j >= 0 && j < dims_[1] && "j is out of bounds in CArray 7D!");
2158     assert(k >= 0 && k < dims_[2] && "k is out of bounds in CArray 7D!");
2159     assert(l >= 0 && l < dims_[3] && "l is out of bounds in CArray 7D!");
2160     assert(m >= 0 && m < dims_[4] && "m is out of bounds in CArray 7D!");
2161     assert(n >= 0 && n < dims_[5] && "n is out of bounds in CArray 7D!");
2162     assert(o >= 0 && o < dims_[6] && "o is out of bounds in CArray 7D!");
2163
2164     return array_[o + (n * dims_[6])
2165                     + (m * dims_[6] * dims_[5])
2166                     + (l * dims_[6] * dims_[5] * dims_[4])
2167                     + (k * dims_[6] * dims_[5] * dims_[4] * dims_[3])
2168                     + (j * dims_[6] * dims_[5] * dims_[4] * dims_[3] * dims_[2])
2169                     + (i * dims_[6] * dims_[5] * dims_[4] * dims_[3] * dims_[2] *  dims_[1])];
2170
2171 }
2172
2173
2174 //overload = operator
2175 template <typename T>
2176 inline CArray<T>& CArray<T>::operator= (const CArray& temp)
2177 {
2178
2179     // Do nothing if the assignment is of the form x = x
2180     if (this != &temp) {
2181         for (int iter = 0; iter < temp.order_; iter++){
2182             dims_[iter] = temp.dims_[iter];
2183         } // end for
2184
```

```
2185         order_  = temp.order_;
2186         length_ = temp.length_;
2187         array_  = temp.array_;
2188     }
2189     return *this;
2190 }
2191
2192
2193
2194 //return size
2195 template <typename T>
2196 inline size_t CArray<T>::size() const {
2197     return length_;
2198 }
2199
2200 template <typename T>
2201 inline size_t CArray<T>::dims(size_t i) const {
2202     assert(i < order_ && "CArray order (rank) does not match constructor, dim[i] does not exist!");
2203     assert(i >= 0 && dims_[i]>0 && "Access to CArray dims is out of bounds!");
2204     return dims_[i];
2205 }
2206
2207 template <typename T>
2208 inline size_t CArray<T>::order() const {
2209     return order_;
2210 }
2211
2212
2213 template <typename T>
2214 inline T* CArray<T>::pointer() const{
2215     return array_.get();
2216 }
2217
2218 //destructor
2219 template <typename T>
2220 CArray<T>::~CArray() {}
2221
2222 //----endof carray class definitions----
2223
2224
2225 //6. ViewCArray
2226 // indicies are [0:N-1]
2227 template <typename T>
2228 class ViewCArray {
2229
2230 private:
2231     size_t dims_[7];
2232     size_t length_; // Length of 1D array
2233     size_t order_;  // tensor order (rank)
2234     T * array_;
2235
2236 public:
2237
2238     // Default constructor
2239     ViewCArray ();
2240
2241     //--- 1D to 7D array ---
2242     ViewCArray(T *array,
2243                size_t dim0);
2244
2245     ViewCArray(T *array,
2246                size_t dim0,
2247                size_t dim1);
2248
2249     ViewCArray(T *some_array,
2250                size_t dim0,
2251                size_t dim1,
2252                size_t dim2);
2253
2254     ViewCArray(T *some_array,
2255                size_t dim0,
2256                size_t dim1,
2257                size_t dim2,
2258                size_t dim3);
2259
2260     ViewCArray (T *some_array,
2261                 size_t dim0,
2262                 size_t dim1,
2263                 size_t dim2,
2264                 size_t dim3,
2265                 size_t dim4);
2266
2267     ViewCArray (T *some_array,
2268                 size_t dim0,
2269                 size_t dim1,
2270                 size_t dim2,
2271                 size_t dim3,
```

```
2272                 size_t dim4,
2273                 size_t dim5);
2274
2275     ViewCArray (T *some_array,
2276                 size_t dim0,
2277                 size_t dim1,
2278                 size_t dim2,
2279                 size_t dim3,
2280                 size_t dim4,
2281                 size_t dim5,
2282                 size_t dim6);
2283
2284     T& operator()(size_t i) const;
2285
2286     T& operator()(size_t i,
2287                   size_t j) const;
2288
2289     T& operator()(size_t i,
2290                   size_t j,
2291                   size_t k) const;
2292
2293     T& operator()(size_t i,
2294                   size_t j,
2295                   size_t k,
2296                   size_t l) const;
2297     T& operator()(size_t i,
2298                   size_t j,
2299                   size_t k,
2300                   size_t l,
2301                   size_t m) const;
2302
2303     T& operator()(size_t i,
2304                   size_t j,
2305                   size_t k,
2306                   size_t l,
2307                   size_t m,
2308                   size_t n) const;
2309
2310     T& operator()(size_t i,
2311                   size_t j,
2312                   size_t k,
2313                   size_t l,
2314                   size_t m,
2315                   size_t n,
2316                   size_t o) const;
2317
2318     // calculate C = math(A,B)
2319     template <typename M>
2320     void operator=(M do_this_math);
2321
2322     //return array size
2323     size_t size() const;
2324
2325     // return array dims
2326     size_t dims(size_t i) const;
2327
2328     // return array order (rank)
2329     size_t order() const;
2330
2331     // return pointer
2332     T* pointer() const;
2333
2334 }; // end of ViewCArray
2335
2336 //class definitions
2337
2338 //constructors
2339
2340 //no dim
2341 template <typename T>
2342 ViewCArray<T>::ViewCArray() {
2343   array_ = NULL;
2344   length_ = order_ = 0;
2345 }
2346
2347 //1D
2348 template <typename T>
2349 ViewCArray<T>::ViewCArray(T *array,
2350                           size_t dim0)
2351 {
2352     dims_[0] = dim0;
2353     order_ = 1;
2354     length_ = dim0;
2355     array_ = array;
2356 }
2357
2358 //2D
```

```
2359 template <typename T>
2360 ViewCArray<T>::ViewCArray(T *array,
2361                           size_t dim0,
2362                           size_t dim1)
2363 {
2364     dims_[0] = dim0;
2365     dims_[1] = dim1;
2366     order_ = 2;
2367     length_ = dim0 * dim1;
2368     array_ = array;
2369 }
2370
2371 //3D
2372 template <typename T>
2373 ViewCArray<T>::ViewCArray (T *array,
2374                            size_t dim0,
2375                            size_t dim1,
2376                            size_t dim2)
2377 {
2378     dims_[0] = dim0;
2379     dims_[1] = dim1;
2380     dims_[2] = dim2;
2381     order_ = 3;
2382     length_ = dim0 * dim1 * dim2;
2383     array_ = array;
2384 }
2385
2386 //4D
2387 template <typename T>
2388 ViewCArray<T>::ViewCArray(T *array,
2389                           size_t dim0,
2390                           size_t dim1,
2391                           size_t dim2,
2392                           size_t dim3)
2393 {
2394     dims_[0] = dim0;
2395     dims_[1] = dim1;
2396     dims_[2] = dim2;
2397     dims_[3] = dim3;
2398     order_ = 4;
2399     length_ = dim0 * dim1 * dim2 * dim3;
2400     array_ = array;
2401 }
2402
2403 //5D
2404 template <typename T>
2405 ViewCArray<T>::ViewCArray(T *array,
2406                           size_t dim0,
2407                           size_t dim1,
2408                           size_t dim2,
2409                           size_t dim3,
2410                           size_t dim4)
2411 {
2412     dims_[0] = dim0;
2413     dims_[1] = dim1;
2414     dims_[2] = dim2;
2415     dims_[3] = dim3;
2416     dims_[4] = dim4;
2417     order_ = 5;
2418     length_ = dim0 * dim1 * dim2 * dim3 * dim4;
2419     array_ = array;
2420 }
2421
2422 //6D
2423 template <typename T>
2424 ViewCArray<T>::ViewCArray(T *array,
2425                           size_t dim0,
2426                           size_t dim1,
2427                           size_t dim2,
2428                           size_t dim3,
2429                           size_t dim4,
2430                           size_t dim5)
2431 {
2432     dims_[0] = dim0;
2433     dims_[1] = dim1;
2434     dims_[2] = dim2;
2435     dims_[3] = dim3;
2436     dims_[4] = dim4;
2437     dims_[5] = dim5;
2438     order_ = 6;
2439     length_ = dim0 * dim1 * dim2 * dim3 * dim4 * dim5;
2440     array_ = array;
2441 }
2442
2443 //7D
2444 template <typename T>
2445 ViewCArray<T>::ViewCArray(T *array,
```

```
2446                            size_t dim0,
2447                            size_t dim1,
2448                            size_t dim2,
2449                            size_t dim3,
2450                            size_t dim4,
2451                            size_t dim5,
2452                            size_t dim6)
2453 {
2454     dims_[0] = dim0;
2455     dims_[1] = dim1;
2456     dims_[2] = dim2;
2457     dims_[3] = dim3;
2458     dims_[4] = dim4;
2459     dims_[5] = dim5;
2460     dims_[6] = dim6;
2461     order_ = 7;
2462     length_ = dim0 * dim1 * dim2 * dim3 * dim4 * dim5 * dim6;
2463     array_ = array;
2464 }
2465
2466 //overload () operator
2467
2468 //1D
2469 template <typename T>
2470 inline T& ViewCArray<T>::operator()(size_t i) const
2471 {
2472     assert(order_ == 1 && "Tensor order (rank) does not match constructor in ViewCArray 1D!");
2473     assert(i >= 0 && i < dims_[0] && "i is out of bounds in ViewCArray 1D!");
2474
2475     return array_[i];
2476 }
2477
2478 /*
2479 //specification for CArray type
2480 //1D
2481 template <typename T>
2482 inline T& ViewCArray<CArray<T>>::operator()(size_t i) const
2483 {
2484     assert(i < dim1_ && "i is out of bounds in c_array 1D");  // die if >= dim1
2485
2486     return (*this_array_)(i);
2487 }
2488 */
2489
2490 //2D
2491 template <typename T>
2492 inline T& ViewCArray<T>::operator()(size_t i,
2493                                     size_t j) const
2494 {
2495
2496     assert(order_ == 2 && "Tensor order (rank) does not match constructor in ViewCArray 2D!");
2497     assert(i >= 0 && i < dims_[0] && "i is out of bounds in ViewCArray 2D!");
2498     assert(j >= 0 && j < dims_[1] && "j is out of bounds in ViewCArray 2D!");
2499
2500     return array_[j + (i *  dims_[1])];
2501 }
2502
2503 //3D
2504 template <typename T>
2505 inline T& ViewCArray<T>::operator()(size_t i,
2506                                     size_t j,
2507                                     size_t k) const
2508 {
2509     assert(order_ == 3 && "Tensor order (rank) does not match constructor in ViewCArray 3D!");
2510     assert(i >= 0 && i < dims_[0] && "i is out of bounds in ViewCArray 3D!");
2511     assert(j >= 0 && j < dims_[1] && "j is out of bounds in ViewCarray 3D!");
2512     assert(k >= 0 && k < dims_[2] && "k is out of bounds in ViewCArray 3D!");
2513
2514     return array_[k + (j * dims_[2])
2515                     + (i * dims_[2] *  dims_[1])];
2516 }
2517
2518 //4D
2519 template <typename T>
2520 inline T& ViewCArray<T>::operator()(size_t i,
2521                                     size_t j,
2522                                     size_t k,
2523                                     size_t l) const
2524 {
2525     assert(order_ == 4 && "Tensor order (rank) does not match constructor in ViewCArray 4D!");
2526     assert(i >= 0 && i < dims_[0] && "i is out of bounds in ViewCArray 4D");  // die if >= dim0
2527     assert(j >= 0 && j < dims_[1] && "j is out of bounds in ViewCArray 4D");  // die if >= dim1
2528     assert(k >= 0 && k < dims_[2] && "k is out of bounds in ViewCArray 4D");  // die if >= dim2
2529     assert(l >= 0 && l < dims_[3] && "l is out of bounds in ViewCArray 4D");  // die if >= dim3
2530
2531     return array_[l + (k * dims_[3])
2532                     + (j * dims_[3] * dims_[2])
```

```
2533                     + (i * dims_[3] * dims_[2] *  dims_[1])];
2534 }
2535
2536 //5D
2537 template <typename T>
2538 inline T& ViewCArray<T>::operator()(size_t i,
2539                                     size_t j,
2540                                     size_t k,
2541                                     size_t l,
2542                                     size_t m) const
2543 {
2544     assert(order_ == 5 && "Tensor order (rank) does not match constructor in ViewCArray 5D!");
2545     assert(i >= 0 && i < dims_[0] && "i is out of bounds in ViewCArray 5D!");
2546     assert(j >= 0 && j < dims_[1] && "j is out of bounds in ViewCArray 5D!");
2547     assert(k >= 0 && k < dims_[2] && "k is out of bounds in ViewCArray 5D!");
2548     assert(l >= 0 && l < dims_[3] && "l is out of bounds in ViewCArray 5D!");
2549     assert(m >= 0 && m < dims_[4] && "m is out of bounds in ViewCArray 5D!");
2550
2551     return array_[m + (l * dims_[4])
2552                     + (k * dims_[4] * dims_[3])
2553                     + (j * dims_[4] * dims_[3] * dims_[2])
2554                     + (i * dims_[4] * dims_[3] * dims_[2] *  dims_[1])];
2555 }
2556
2557 //6D
2558 template <typename T>
2559 inline T& ViewCArray<T>::operator()(size_t i,
2560                                     size_t j,
2561                                     size_t k,
2562                                     size_t l,
2563                                     size_t m,
2564                                     size_t n) const
2565 {
2566     assert(order_ == 6 && "Tensor order (rank) does not match constructor in ViewCArray 6D!");
2567     assert(i >= 0 && i < dims_[0] && "i is out of bounds in ViewCArray 6D!");
2568     assert(j >= 0 && j < dims_[1] && "j is out of bounds in ViewCArray 6D!");
2569     assert(k >= 0 && k < dims_[2] && "k is out of bounds in ViewCArray 6D!");
2570     assert(l >= 0 && l < dims_[3] && "l is out of bounds in ViewCArray 6D!");
2571     assert(m >= 0 && m < dims_[4] && "m is out of bounds in ViewCArray 6D!");
2572     assert(n >= 0 && n < dims_[5] && "n is out of bounds in ViewCArray 6D!");
2573
2574     return array_[n + (m * dims_[5])
2575                     + (l * dims_[5] * dims_[4])
2576                     + (k * dims_[5] * dims_[4] * dims_[3])
2577                     + (j * dims_[5] * dims_[4] * dims_[3] * dims_[2])
2578                     + (i * dims_[5] * dims_[4] * dims_[3] * dims_[2] *  dims_[1])];
2579 }
2580
2581 //7D
2582 template <typename T>
2583 inline T& ViewCArray<T>::operator()(size_t i,
2584                                     size_t j,
2585                                     size_t k,
2586                                     size_t l,
2587                                     size_t m,
2588                                     size_t n,
2589                                     size_t o) const
2590 {
2591     assert(order_ == 7 && "Tensor order (rank) does not match constructor in ViewCArray 7D!");
2592     assert(i >= 0 && i < dims_[0] && "i is out of bounds in ViewCArray 7D!");
2593     assert(j >= 0 && j < dims_[1] && "j is out of bounds in ViewCArray 7D!");
2594     assert(k >= 0 && k < dims_[2] && "k is out of bounds in ViewCArray 7D!");
2595     assert(l >= 0 && l < dims_[3] && "l is out of bounds in ViewCArray 7D!");
2596     assert(m >= 0 && m < dims_[4] && "m is out of bounds in ViewCArray 7D!");
2597     assert(n >= 0 && n < dims_[5] && "n is out of bounds in ViewCArray 7D!");
2598     assert(o >= 0 && o < dims_[6] && "o is out of bounds in ViewCArray 7D!");
2599
2600     return array_[o + (n * dims_[6])
2601                     + (m * dims_[6] * dims_[5])
2602                     + (l * dims_[6] * dims_[5] * dims_[4])
2603                     + (k * dims_[6] * dims_[5] * dims_[4] * dims_[3])
2604                     + (j * dims_[6] * dims_[5] * dims_[4] * dims_[3] * dims_[2])
2605                     + (i * dims_[6] * dims_[5] * dims_[4] * dims_[3] * dims_[2] *  dims_[1])];
2606 }
2607
2608
2609 // calculate this ViewFArray object = math(A,B)
2610 template <typename T>
2611 template <typename M>
2612 void ViewCArray<T>::operator=(M do_this_math){
2613     do_this_math(*this); // pass in this ViewFArray object
2614 }// end of math opperation
2615
2616 //return size
2617 template <typename T>
2618 inline size_t ViewCArray<T>::size() const {
2619     return length_;
```

```
2620 }
2621
2622 template <typename T>
2623 inline size_t ViewCArray<T>::dims(size_t i) const {
2624     assert(i < order_ && "ViewCArray order (rank) does not match constructor, dim[i] does not exist!");
2625     assert(i >= 0 && dims_[i]>0 && "Access to ViewCArray dims is out of bounds!");
2626     return dims_[i];
2627 }
2628
2629 template <typename T>
2630 inline size_t ViewCArray<T>::order() const {
2631     return order_;
2632 }
2633
2634 template <typename T>
2635 inline T* ViewCArray<T>::pointer() const {
2636     return array_;
2637 }
2638
2639 //---end of ViewCArray class definitions----
2640
2641
2642 //7. CMatrix
2643 template <typename T>
2644 class CMatrix {
2645
2646 private:
2647     size_t dims_[7];
2648     size_t length_; // Length of 1D array
2649     size_t order_;  // tensor order (rank)
2650     std::shared_ptr <T []> matrix_;
2651
2652 public:
2653
2654     // default constructor
2655     CMatrix();
2656
2657     CMatrix(size_t dim1);
2658
2659     CMatrix(size_t dim1,
2660             size_t dim2);
2661
2662     CMatrix(size_t dim1,
2663             size_t dim2,
2664             size_t dim3);
2665
2666     CMatrix(size_t dim1,
2667             size_t dim2,
2668             size_t dim3,
2669             size_t dim4);
2670
2671     CMatrix(size_t dim1,
2672             size_t dim2,
2673             size_t dim3,
2674             size_t dim4,
2675             size_t dim5);
2676
2677     CMatrix (size_t dim1,
2678             size_t dim2,
2679             size_t dim3,
2680             size_t dim4,
2681             size_t dim5,
2682             size_t dim6);
2683
2684     CMatrix (size_t dim1,
2685             size_t dim2,
2686             size_t dim3,
2687             size_t dim4,
2688             size_t dim5,
2689             size_t dim6,
2690             size_t dim7);
2691
2692     CMatrix(const CMatrix& temp);
2693
2694     //overload operators to access data
2695     T& operator()(size_t i) const;
2696
2697     T& operator()(size_t i,
2698                   size_t j) const;
2699
2700     T& operator()(size_t i,
2701                   size_t j,
2702                   size_t k) const;
2703
2704     T& operator()(size_t i,
2705                   size_t j,
2706                   size_t k,
```

```
2707                     size_t l) const;
2708
2709     T& operator()(size_t i,
2710                     size_t j,
2711                     size_t k,
2712                     size_t l,
2713                     size_t m) const;
2714
2715     T& operator()(size_t i,
2716                     size_t j,
2717                     size_t k,
2718                     size_t l,
2719                     size_t m,
2720                     size_t n) const;
2721
2722     T& operator()(size_t i,
2723                     size_t j,
2724                     size_t k,
2725                     size_t l,
2726                     size_t m,
2727                     size_t n,
2728                     size_t o) const;
2729
2730     //overload = operator
2731     CMatrix& operator= (const CMatrix &temp);
2732
2733     //return array size
2734     size_t size() const;
2735
2736     // return array dims
2737     size_t dims(size_t i) const;
2738
2739     // return array order (rank)
2740     size_t order() const;
2741
2742     //return pointer
2743     T* pointer() const;
2744
2745     // deconstructor
2746     ~CMatrix( );
2747
2748 }; // end of CMatrix
2749
2750 // CMatrix class definitions
2751
2752 //constructors
2753
2754 //no dim
2755
2756 //1D
2757 template <typename T>
2758 CMatrix<T>::CMatrix() {
2759     matrix_ = NULL;
2760     length_ = 0;
2761 }
2762
2763 //1D
2764 template <typename T>
2765 CMatrix<T>::CMatrix(size_t dim1)
2766 {
2767     dims_[0] = dim1;
2768     order_ = 1;
2769     length_ = dim1;
2770     matrix_ = std::shared_ptr <T[]> (new T[length_]);
2771 }
2772
2773 //2D
2774 template <typename T>
2775 CMatrix<T>::CMatrix(size_t dim1,
2776                     size_t dim2)
2777 {
2778     dims_[0] = dim1;
2779     dims_[1] = dim2;
2780     order_ = 2;
2781     length_ = dim1 * dim2;
2782     matrix_ = std::shared_ptr <T[]> (new T[length_]);
2783 }
2784
2785 //3D
2786 template <typename T>
2787 CMatrix<T>::CMatrix(size_t dim1,
2788                     size_t dim2,
2789                     size_t dim3)
2790 {
2791     dims_[0] = dim1;
2792     dims_[1] = dim2;
2793     dims_[2] = dim3;
```

```
2794     order_ = 3;
2795     length_ = dim1 * dim2 * dim3;
2796     matrix_ = std::shared_ptr <T[]> (new T[length_]);
2797 }
2798 
2799 //4D
2800 template <typename T>
2801 CMatrix<T>::CMatrix(size_t dim1,
2802                     size_t dim2,
2803                     size_t dim3,
2804                     size_t dim4)
2805 {
2806     dims_[0] = dim1;
2807     dims_[1] = dim2;
2808     dims_[2] = dim3;
2809     dims_[3] = dim4;
2810     order_ = 4;
2811     length_ = dim1 * dim2 * dim3 * dim4;
2812     matrix_ = std::shared_ptr <T[]> (new T[length_]);
2813 }
2814 
2815 //5D
2816 template <typename T>
2817 CMatrix<T>::CMatrix(size_t dim1,
2818                     size_t dim2,
2819                     size_t dim3,
2820                     size_t dim4,
2821                     size_t dim5)
2822 {
2823     dims_[0] = dim1;
2824     dims_[1] = dim2;
2825     dims_[2] = dim3;
2826     dims_[3] = dim4;
2827     dims_[4] = dim5;
2828     order_ = 5;
2829     length_ = dim1 * dim2 * dim3 * dim4 * dim5;
2830     matrix_ = std::shared_ptr <T[]> (new T[length_]);
2831 }
2832 
2833 //6D
2834 template <typename T>
2835 CMatrix<T>::CMatrix(size_t dim1,
2836                     size_t dim2,
2837                     size_t dim3,
2838                     size_t dim4,
2839                     size_t dim5,
2840                     size_t dim6)
2841 {
2842     dims_[0] = dim1;
2843     dims_[1] = dim2;
2844     dims_[2] = dim3;
2845     dims_[3] = dim4;
2846     dims_[4] = dim5;
2847     dims_[5] = dim6;
2848     order_ = 6;
2849     length_ = dim1 * dim2 * dim3 * dim4 * dim5 * dim6;
2850     matrix_ = std::shared_ptr <T[]> (new T[length_]);
2851 }
2852 
2853 //7D
2854 template <typename T>
2855 CMatrix<T>::CMatrix(size_t dim1,
2856                     size_t dim2,
2857                     size_t dim3,
2858                     size_t dim4,
2859                     size_t dim5,
2860                     size_t dim6,
2861                     size_t dim7)
2862 {
2863     dims_[0] = dim1;
2864     dims_[1] = dim2;
2865     dims_[2] = dim3;
2866     dims_[3] = dim4;
2867     dims_[4] = dim5;
2868     dims_[5] = dim6;
2869     dims_[6] = dim7;
2870     order_ = 7;
2871     length_ = dim1 * dim2 * dim3 * dim4 * dim5 * dim6 * dim7;
2872     matrix_ = std::shared_ptr <T[]> (new T[length_]);
2873 }
2874 
2875 template <typename T>
2876 CMatrix<T>::CMatrix(const CMatrix& temp) {
2877 
2878     // Do nothing if the assignment is of the form x = x
2879 
2880     if (this != &temp) {
```

```
2881         for (int iter = 0; iter < temp.order_; iter++){
2882             dims_[iter] = temp.dims_[iter];
2883         } // end for
2884 
2885         order_  = temp.order_;
2886         length_ = temp.length_;
2887         matrix_ = temp.matrix_;
2888     } // end if
2889 
2890 } // end constructor
2891 
2892 //overload () operator
2893 
2894 //1D
2895 template <typename T>
2896 T& CMatrix<T>::operator()(size_t i) const
2897 {
2898     assert(order_ == 1 && "Tensor order (rank) does not match constructor in CMatrix 1D!");
2899     assert(i >= 1 && i <= dims_[0] && "i is out of bounds in CMatrix 1D!");
2900 
2901     return matrix_[i-1];
2902 }
2903 
2904 //2D
2905 template <typename T>
2906 T& CMatrix<T>::operator()(size_t i,
2907                           size_t j) const
2908 {
2909     assert(order_ == 2 && "Tensor order (rank) does not match constructor in CMatrix 2D!");
2910     assert(i >= 1 && i <= dims_[0] && "i is out of bounds in CMatrix 2D!");
2911     assert(j >= 1 && j <= dims_[1] && "j is out of bounds in CMatrix 2D!");
2912 
2913     return matrix_[(j-1) + (i-1)*dims_[1]];
2914 }
2915 
2916 //3D
2917 template <typename T>
2918 T& CMatrix<T>::operator()(size_t i,
2919                           size_t j,
2920                           size_t k) const
2921 {
2922     assert(order_ == 3 && "Tensor order (rank) does not match constructor in CMatrix 3D!");
2923     assert(i >= 1 && i <= dims_[0] && "i is out of bounds in CMatrix 3D!");
2924     assert(j >= 1 && j <= dims_[1] && "j is out of bounds in CMatrix 3D!");
2925     assert(k >= 1 && k <= dims_[2] && "k is out of bounds in CMatrix 3D!");
2926 
2927     return matrix_[(k-1) + (j-1)*dims_[2]
2928                          + (i-1)*dims_[2]*dims_[1]];
2929 }
2930 
2931 //4D
2932 template <typename T>
2933 T& CMatrix<T>::operator()(size_t i,
2934                           size_t j,
2935                           size_t k,
2936                           size_t l) const
2937 {
2938     assert(order_ == 4 && "Tensor order (rank) does not match constructor in CMatrix 4D!");
2939     assert(i >= 1 && i <= dims_[0] && "i is out of bounds in CMatrix 4D");  // die if >= dim0
2940     assert(j >= 1 && j <= dims_[1] && "j is out of bounds in CMatrix 4D");  // die if >= dim1
2941     assert(k >= 1 && k <= dims_[2] && "k is out of bounds in CMatrix 4D");  // die if >= dim2
2942     assert(l >= 1 && l <= dims_[3] && "l is out of bounds in CMatrix 4D");  // die if >= dim3
2943 
2944     return matrix_[(l-1) + (k-1)*dims_[3]
2945                          + (j-1)*dims_[3]*dims_[2]
2946                          + (i-1)*dims_[3]*dims_[2]*dims_[1]];
2947 }
2948 
2949 //5D
2950 template <typename T>
2951 T& CMatrix<T>::operator()(size_t i,
2952                           size_t j,
2953                           size_t k,
2954                           size_t l,
2955                           size_t m) const
2956 {
2957     assert(order_ == 5 && "Tensor order (rank) does not match constructor in CMatrix 5D!");
2958     assert(i >= 1 && i <= dims_[0] && "i is out of bounds in CMatrix 5D!");
2959     assert(j >= 1 && j <= dims_[1] && "j is out of bounds in CMatrix 5D!");
2960     assert(k >= 1 && k <= dims_[2] && "k is out of bounds in CMatrix 5D!");
2961     assert(l >= 1 && l <= dims_[3] && "l is out of bounds in CMatrix 5D!");
2962     assert(m >= 1 && m <= dims_[4] && "m is out of bounds in CMatrix 5D!");
2963 
2964     return matrix_[(m-1) + (l-1)*dims_[4]
2965                          + (k-1)*dims_[4]*dims_[3]
2966                          + (j-1)*dims_[4]*dims_[3]*dims_[2]
2967                          + (i-1)*dims_[4]*dims_[3]*dims_[2]*dims_[1]];
```

```
2968 }
2969
2970 //6D
2971 template <typename T>
2972 T& CMatrix<T>::operator()(size_t i,
2973                           size_t j,
2974                           size_t k,
2975                           size_t l,
2976                           size_t m,
2977                           size_t n) const
2978 {
2979     assert(order_ == 6 && "Tensor order (rank) does not match constructor in CMatrix 6D!");
2980     assert(i >= 1 && i <= dims_[0] && "i is out of bounds in CMatrix 6D!");
2981     assert(j >= 1 && j <= dims_[1] && "j is out of bounds in CMatrix 6D!");
2982     assert(k >= 1 && k <= dims_[2] && "k is out of bounds in CMatrix 6D!");
2983     assert(l >= 1 && l <= dims_[3] && "l is out of bounds in CMatrix 6D!");
2984     assert(m >= 1 && m <= dims_[4] && "m is out of bounds in CMatrix 6D!");
2985     assert(n >= 1 && n <= dims_[5] && "n is out of bounds in CMatrix 6D!");
2986
2987     return matrix_[ (n-1) + (m-1)*dims_[5]
2988                           + (l-1)*dims_[5]*dims_[4]
2989                           + (k-1)*dims_[5]*dims_[4]*dims_[3]
2990                           + (j-1)*dims_[5]*dims_[4]*dims_[3]*dims_[2]
2991                           + (i-1)*dims_[5]*dims_[4]*dims_[3]*dims_[2]*dims_[1]];
2992 }
2993
2994 //7D
2995 template <typename T>
2996 T& CMatrix<T>::operator()(size_t i,
2997                           size_t j,
2998                           size_t k,
2999                           size_t l,
3000                           size_t m,
3001                           size_t n,
3002                           size_t o) const
3003 {
3004     assert(order_ == 7 && "Tensor order (rank) does not match constructor in CMatrix 7D!");
3005     assert(i >= 1 && i <= dims_[0] && "i is out of bounds in CMatrix 7D!");
3006     assert(j >= 1 && j <= dims_[1] && "j is out of bounds in CMatrix 7D!");
3007     assert(k >= 1 && k <= dims_[2] && "k is out of bounds in CMatrix 7D!");
3008     assert(l >= 1 && l <= dims_[3] && "l is out of bounds in CMatrix 7D!");
3009     assert(m >= 1 && m <= dims_[4] && "m is out of bounds in CMatrix 7D!");
3010     assert(n >= 1 && n <= dims_[5] && "n is out of bounds in CMatrix 7D!");
3011     assert(o >= 1 && o <= dims_[6] && "o is out of bounds in CMatrix 7D!");
3012
3013     return matrix_[(o-1) + (n-1)*dims_[6]
3014                          + (m-1)*dims_[6]*dims_[5]
3015                          + (l-1)*dims_[6]*dims_[5]*dims_[4]
3016                          + (k-1)*dims_[6]*dims_[5]*dims_[4]*dims_[3]
3017                          + (j-1)*dims_[6]*dims_[5]*dims_[4]*dims_[3]*dims_[2]
3018                          + (i-1)*dims_[6]*dims_[5]*dims_[4]*dims_[3]*dims_[2]*dims_[1]];
3019 }
3020
3021 //overload = operator
3022 //THIS = CMatrix<> temp
3023 template <typename T>
3024 CMatrix<T> &CMatrix<T>::operator= (const CMatrix &temp) {
3025     if(this != &temp) {
3026         for (int iter = 0; iter < temp.order_; iter++){
3027             dims_[iter] = temp.dims_[iter];
3028         } // end for
3029
3030         order_  = temp.order_;
3031         length_ = temp.length_;
3032         matrix_ = temp.matrix_;
3033     }
3034   return *this;
3035 }
3036
3037 template <typename T>
3038 inline size_t CMatrix<T>::size() const {
3039     return length_;
3040 }
3041
3042 template <typename T>
3043 inline size_t CMatrix<T>::dims(size_t i) const {
3044     i--; // i starts at 1
3045     assert(i < order_ && "CMatrix order (rank) does not match constructor, dim[i] does not exist!");
3046     assert(i >= 0 && dims_[i]>0 && "Access to CMatrix dims is out of bounds!");
3047     return dims_[i];
3048 }
3049
3050 template <typename T>
3051 inline size_t CMatrix<T>::order() const {
3052     return order_;
3053 }
3054
```

```
3055 template <typename T>
3056 inline T* CMatrix<T>::pointer() const{
3057     return matrix_.get();
3058 }
3059
3060 // Destructor
3061 template <typename T>
3062 CMatrix<T>::~CMatrix(){}
3063
3064 //----end of CMatrix class definitions----
3065
3066
3067 //8. ViewCMatrix
3068 //  indices [1:N]
3069 template <typename T>
3070 class ViewCMatrix {
3071
3072 private:
3073     size_t dims_[7];
3074     size_t length_; // Length of 1D array
3075     size_t order_;  // tensor order (rank)
3076      T * matrix_;
3077
3078 public:
3079
3080     // default constructor
3081     ViewCMatrix();
3082
3083
3084     //--- 1D array ---
3085     // overloaded constructor
3086     ViewCMatrix (T *matrix,
3087                  size_t dim1);
3088
3089     ViewCMatrix (T *matrix,
3090                  size_t dim1,
3091                  size_t dim2);
3092
3093     ViewCMatrix (T *matrix,
3094         size_t dim1,
3095         size_t dim2,
3096         size_t dim3);
3097
3098     ViewCMatrix (T *matrix,
3099         size_t dim1,
3100         size_t dim2,
3101         size_t dim3,
3102         size_t dim4);
3103
3104     ViewCMatrix (T *matrix,
3105         size_t dim1,
3106         size_t dim2,
3107         size_t dim3,
3108         size_t dim4,
3109         size_t dim5);
3110
3111     ViewCMatrix (T *matrix,
3112            size_t dim1,
3113            size_t dim2,
3114            size_t dim3,
3115            size_t dim4,
3116            size_t dim5,
3117            size_t dim6);
3118
3119     ViewCMatrix (T *matrix,
3120                  size_t dim1,
3121                  size_t dim2,
3122                  size_t dim3,
3123                  size_t dim4,
3124                  size_t dim5,
3125                  size_t dim6,
3126                  size_t dim7);
3127
3128     T& operator() (size_t i) const;
3129
3130     T& operator() (size_t i,
3131                    size_t j) const;
3132
3133     T& operator() (size_t i,
3134                    size_t j,
3135                    size_t k) const;
3136
3137     T& operator() (size_t i,
3138                    size_t j,
3139                    size_t k,
3140                    size_t l) const;
3141
```

```
3142     T& operator() (size_t i,
3143                    size_t j,
3144                    size_t k,
3145                    size_t l,
3146                    size_t m) const;
3147
3148     T& operator() (size_t i,
3149                    size_t j,
3150                    size_t k,
3151                    size_t l,
3152                    size_t m,
3153                    size_t n) const;
3154     T& operator() (size_t i,
3155                    size_t j,
3156                    size_t k,
3157                    size_t l,
3158                    size_t m,
3159                    size_t n,
3160                    size_t o) const;
3161
3162     // calculate C = math(A,B)
3163     template <typename M>
3164     void operator=(M do_this_math);
3165
3166     //return array size
3167     size_t size() const;
3168
3169     // return array dims
3170     size_t dims(size_t i) const;
3171
3172     // return array order (rank)
3173     size_t order() const;
3174
3175     // return pointer
3176     T* pointer() const;
3177
3178 }; // end of ViewCMatrix
3179
3180 //class definitions
3181
3182 //constructors
3183
3184 //no dim
3185 template <typename T>
3186 ViewCMatrix<T>::ViewCMatrix(){
3187   matrix_ = NULL;
3188   length_ = 0;
3189 }
3190
3191 //1D
3192 template <typename T>
3193 ViewCMatrix<T>::ViewCMatrix(T *matrix,
3194                             size_t dim1)
3195 {
3196     dims_[0] = dim1;
3197     order_ = 1;
3198     length_ = dim1;
3199     matrix_ = matrix;
3200 }
3201
3202 //2D
3203 template <typename T>
3204 ViewCMatrix<T>::ViewCMatrix(T *matrix,
3205                             size_t dim1,
3206                             size_t dim2)
3207 {
3208     dims_[0] = dim1;
3209     dims_[1] = dim2;
3210     order_ = 2;
3211     length_ = dim1 * dim2;
3212     matrix_ = matrix;
3213 }
3214
3215 //3D
3216 template <typename T>
3217 ViewCMatrix<T>::ViewCMatrix(T *matrix,
3218                             size_t dim1,
3219                             size_t dim2,
3220                             size_t dim3)
3221 {
3222     dims_[0] = dim1;
3223     dims_[1] = dim2;
3224     dims_[2] = dim3;
3225     order_ = 3;
3226     length_ = dim1 * dim2 * dim3;
3227     matrix_ = matrix;
3228 }
```

```
3229
3230 //4D
3231 template <typename T>
3232 ViewCMatrix<T>::ViewCMatrix(T *matrix,
3233                             size_t dim1,
3234                             size_t dim2,
3235                             size_t dim3,
3236                             size_t dim4)
3237 {
3238     dims_[0] = dim1;
3239     dims_[1] = dim2;
3240     dims_[2] = dim3;
3241     dims_[3] = dim4;
3242     order_ = 4;
3243     length_ = dim1 * dim2 * dim3 * dim4;
3244     matrix_ = matrix;
3245 }
3246
3247 //5D
3248 template <typename T>
3249 ViewCMatrix<T>::ViewCMatrix(T *matrix,
3250                             size_t dim1,
3251                             size_t dim2,
3252                             size_t dim3,
3253                             size_t dim4,
3254                             size_t dim5)
3255 {
3256     dims_[0] = dim1;
3257     dims_[1] = dim2;
3258     dims_[2] = dim3;
3259     dims_[3] = dim4;
3260     dims_[4] = dim5;
3261     order_ = 5;
3262     length_ = dim1 * dim2 * dim3 * dim4 * dim5;
3263     matrix_ = matrix;
3264 }
3265
3266 //6D
3267 template <typename T>
3268 ViewCMatrix<T>::ViewCMatrix(T *matrix,
3269                             size_t dim1,
3270                             size_t dim2,
3271                             size_t dim3,
3272                             size_t dim4,
3273                             size_t dim5,
3274                             size_t dim6) {
3275     dims_[0] = dim1;
3276     dims_[1] = dim2;
3277     dims_[2] = dim3;
3278     dims_[3] = dim4;
3279     dims_[4] = dim5;
3280     dims_[5] = dim6;
3281     order_ = 6;
3282     length_ = dim1 * dim2 * dim3 * dim4 * dim5 * dim6;
3283     matrix_ = matrix;
3284 }
3285
3286 //7D
3287 template <typename T>
3288 ViewCMatrix<T>::ViewCMatrix(T *matrix,
3289                             size_t dim1,
3290                             size_t dim2,
3291                             size_t dim3,
3292                             size_t dim4,
3293                             size_t dim5,
3294                             size_t dim6,
3295                             size_t dim7) {
3296     dims_[0] = dim1;
3297     dims_[1] = dim2;
3298     dims_[2] = dim3;
3299     dims_[3] = dim4;
3300     dims_[4] = dim5;
3301     dims_[5] = dim6;
3302     dims_[6] = dim7;
3303     order_ = 7;
3304     length_ = dim1 * dim2 * dim3 * dim4 * dim5 * dim6 * dim7;
3305     matrix = matrix_;
3306 }
3307
3308 //overload () operator
3309
3310 //1D
3311 template <typename T>
3312 T& ViewCMatrix<T>:: operator() (size_t i) const
3313 {
3314     assert(order_ == 1 && "Tensor order (rank) does not match constructor in ViewCMatrix 1D!");
3315     assert(i >= 1 && i <= dims_[0] && "i is out of bounds in ViewCMatrix 1D!");
```

```
3316
3317     return matrix_[i-1];
3318 }
3319
3320 //2D
3321 template <typename T>
3322 T& ViewCMatrix<T>::operator() (size_t i,
3323                                size_t j) const
3324 {
3325     assert(order_ == 2 && "Tensor order (rank) does not match constructor in ViewCMatrix 2D!");
3326     assert(i >= 1 && i <= dims_[0] && "i is out of bounds in ViewCMatrix 2D!");
3327     assert(j >= 1 && j <= dims_[1] && "j is out of bounds in ViewCMatrix 2D!");
3328
3329     return matrix_[(j-1) + (i-1)*dims_[1]];
3330 }
3331
3332 //3D
3333 template <typename T>
3334 T& ViewCMatrix<T>::operator () (size_t i,
3335                                 size_t j,
3336                                 size_t k) const
3337 {
3338     assert(order_ == 3 && "Tensor order (rank) does not match constructor in ViewCMatrix 3D!");
3339     assert(i >= 1 && i <= dims_[0] && "i is out of bounds in ViewCMatrix 3D!");
3340     assert(j >= 1 && j <= dims_[1] && "j is out of bounds in ViewCMatrix 3D!");
3341     assert(k >= 1 && k <= dims_[2] && "k is out of bounds in ViewCMatrix 3D!");
3342
3343     return matrix_[(k-1) + (j-1)*dims_[2]
3344                          + (i-1)*dims_[2]*dims_[1]];
3345 }
3346
3347 //4D
3348 template <typename T>
3349 T& ViewCMatrix<T>::operator()(size_t i,
3350                               size_t j,
3351                               size_t k,
3352                               size_t l) const
3353 {
3354     assert(order_ == 4 && "Tensor order (rank) does not match constructor in ViewCMatrix 4D!");
3355     assert(i >= 1 && i <= dims_[0] && "i is out of bounds in ViewCMatrix 4D");  // die if >= dim0
3356     assert(j >= 1 && j <= dims_[1] && "j is out of bounds in ViewCMatrix 4D");  // die if >= dim1
3357     assert(k >= 1 && k <= dims_[2] && "k is out of bounds in ViewCMatrix 4D");  // die if >= dim2
3358     assert(l >= 1 && l <= dims_[3] && "l is out of bounds in ViewCMatrix 4D");  // die if >= dim3
3359
3360     return matrix_[(l-1) + (k-1)*dims_[3]
3361                          + (j-1)*dims_[3]*dims_[2]
3362                          + (i-1)*dims_[3]*dims_[2]*dims_[1]];
3363 }
3364
3365 //5D
3366 template <typename T>
3367 T& ViewCMatrix<T>::operator()(size_t i,
3368                               size_t j,
3369                               size_t k,
3370                               size_t l,
3371                               size_t m) const
3372 {
3373     assert(order_ == 5 && "Tensor order (rank) does not match constructor in ViewCMatrix 5D!");
3374     assert(i >= 1 && i <= dims_[0] && "i is out of bounds in ViewCMatrix 5D!");
3375     assert(j >= 1 && j <= dims_[1] && "j is out of bounds in ViewCMatrix 5D!");
3376     assert(k >= 1 && k <= dims_[2] && "k is out of bounds in ViewCMatrix 5D!");
3377     assert(l >= 1 && l <= dims_[3] && "l is out of bounds in ViewCMatrix 5D!");
3378     assert(m >= 1 && m <= dims_[4] && "m is out of bounds in ViewCMatrix 5D!");
3379
3380     return matrix_[(m-1) + (l-1)*dims_[4]
3381                          + (k-1)*dims_[4]*dims_[3]
3382                          + (j-1)*dims_[4]*dims_[3]*dims_[2]
3383                          + (i-1)*dims_[4]*dims_[3]*dims_[2]*dims_[1]];
3384 }
3385
3386 //6D
3387 template <typename T>
3388 T& ViewCMatrix<T>::operator()(size_t i,
3389                               size_t j,
3390                               size_t k,
3391                               size_t l,
3392                               size_t m,
3393                               size_t n) const
3394 {
3395     assert(order_ == 6 && "Tensor order (rank) does not match constructor in ViewCMatrix 6D!");
3396     assert(i >= 1 && i <= dims_[0] && "i is out of bounds in ViewCMatrix 6D!");
3397     assert(j >= 1 && j <= dims_[1] && "j is out of bounds in ViewCMatrix 6D!");
3398     assert(k >= 1 && k <= dims_[2] && "k is out of bounds in ViewCMatrix 6D!");
3399     assert(l >= 1 && l <= dims_[3] && "l is out of bounds in ViewCMatrix 6D!");
3400     assert(m >= 1 && m <= dims_[4] && "m is out of bounds in ViewCMatrix 6D!");
3401     assert(n >= 1 && n <= dims_[5] && "n is out of bounds in ViewCMatrix 6D!");
3402
```

```
3403     return matrix_[(n-1) + (m-1)*dims_[5]
3404                          + (l-1)*dims_[5]*dims_[4]
3405                          + (k-1)*dims_[5]*dims_[4]*dims_[3]
3406                          + (j-1)*dims_[5]*dims_[4]*dims_[3]*dims_[2]
3407                          + (i-1)*dims_[5]*dims_[4]*dims_[3]*dims_[2]*dims_[1]];
3408 }
3409
3410 //7D
3411 template <typename T>
3412 T& ViewCMatrix<T>::operator()(size_t i,
3413                               size_t j,
3414                               size_t k,
3415                               size_t l,
3416                               size_t m,
3417                               size_t n,
3418                               size_t o) const
3419 {
3420     assert(order_ == 7 && "Tensor order (rank) does not match constructor in ViewCMatrix 7D!");
3421     assert(i >= 1 && i <= dims_[0] && "i is out of bounds in ViewCMatrix 7D!");
3422     assert(j >= 1 && j <= dims_[1] && "j is out of bounds in ViewCMatrix 7D!");
3423     assert(k >= 1 && k <= dims_[2] && "k is out of bounds in ViewCMatrix 7D!");
3424     assert(l >= 1 && l <= dims_[3] && "l is out of bounds in ViewCMatrix 7D!");
3425     assert(m >= 1 && m <= dims_[4] && "m is out of bounds in ViewCMatrix 7D!");
3426     assert(n >= 1 && n <= dims_[5] && "n is out of bounds in ViewCMatrix 7D!");
3427     assert(o >= 1 && o <= dims_[6] && "o is out of bounds in ViewCMatrix 7D!");
3428
3429     return matrix_[(o-1) + (n-1)*dims_[6]
3430                          + (m-1)*dims_[6]*dims_[5]
3431                          + (l-1)*dims_[6]*dims_[5]*dims_[4]
3432                          + (k-1)*dims_[6]*dims_[5]*dims_[4]*dims_[3]
3433                          + (j-1)*dims_[6]*dims_[5]*dims_[4]*dims_[3]*dims_[2]
3434                          + (i-1)*dims_[6]*dims_[5]*dims_[4]*dims_[3]*dims_[2]*dims_[1]];
3435 }
3436
3437 // calculate this ViewFArray object = math(A,B)
3438 template <typename T>
3439 template <typename M>
3440 void ViewCMatrix<T>::operator=(M do_this_math){
3441     do_this_math(*this); // pass in this ViewFArray object
3442 }// end of math opperation
3443
3444 template <typename T>
3445 inline size_t ViewCMatrix<T>::size() const {
3446     return length_;
3447 }
3448
3449 template <typename T>
3450 inline size_t ViewCMatrix<T>::dims(size_t i) const {
3451     i--; // i starts at 1
3452     assert(i < order_ && "ViewCMatrix order (rank) does not match constructor, dim[i] does not
      exist!");
3453     assert(i >= 0 && dims_[i]>0 && "Access to ViewCMatrix dims is out of bounds!");
3454     return dims_[i];
3455 }
3456
3457 template <typename T>
3458 inline size_t ViewCMatrix<T>::order() const {
3459     return order_;
3460 }
3461
3462 template <typename T>
3463 inline T* ViewCMatrix<T>::pointer() const {
3464     return matrix_;
3465 }
3466
3467
3468 //----end of ViewCMatrix class definitions----
3469
3470 //9. RaggedRightArray
3471 template <typename T>
3472 class RaggedRightArray {
3473 private:
3474     std::shared_ptr <size_t[]> start_index_;
3475     std::shared_ptr <T[]> array_;
3476
3477     size_t dim1_, length_;
3478     size_t num_saved_; // the number saved in the 1D array
3479
3480 public:
3481     // Default constructor
3482     RaggedRightArray ();
3483
3484     //--- 2D array access of a ragged right array ---
3485
3486     // Overload constructor for a CArray
3487     RaggedRightArray (CArray<size_t> &strides_array);
3488
```

```
3489     // Overload constructor for a ViewCArray
3490     RaggedRightArray (ViewCArray<size_t> &strides_array);
3491
3492     // Overloaded constructor for a traditional array
3493     RaggedRightArray (size_t *strides_array, size_t some_dim1);
3494
3495     // Overload constructor for a RaggedRightArray to
3496     // support a dynamically built stride_array
3497     RaggedRightArray (size_t some_dim1, size_t buffer);
3498
3499     // Copy constructor
3500     RaggedRightArray (const RaggedRightArray& temp);
3501
3502     // A method to return the stride size
3503     size_t stride(size_t i) const;
3504
3505     // A method to increase the number of column entries, i.e.,
3506     // the stride size. Used with the constructor for building
3507     // the stride_array dynamically.
3508     // DO NOT USE with the constructures with a strides_array
3509     void push_back(size_t i);
3510
3511     // Overload operator() to access data as array(i,j)
3512     // where i=[0:N-1], j=[stride(i)]
3513     T& operator()(size_t i, size_t j) const;
3514
3515     // method to return total size
3516     size_t size() const;
3517
3518     //return pointer
3519     T* pointer() const;
3520
3521     //get row starts array
3522     size_t* get_starts() const;
3523
3524     RaggedRightArray& operator+= (const size_t i);
3525
3526     RaggedRightArray& operator= (const RaggedRightArray &temp);
3527
3528     // Destructor
3529     ~RaggedRightArray ( );
3530 }; // End of RaggedRightArray
3531
3532 // Default constructor
3533 template <typename T>
3534 RaggedRightArray<T>::RaggedRightArray () {
3535     array_ = NULL;
3536     start_index_ = NULL;
3537     length_ = 0;
3538 }
3539
3540
3541 // Overloaded constructor with CArray
3542 template <typename T>
3543 RaggedRightArray<T>::RaggedRightArray (CArray<size_t> &strides_array){
3544     // The length of the stride array is some_dim1;
3545     dim1_  = strides_array.size();
3546
3547     // Create and initialize the starting index of the entries in the 1D array
3548     start_index_ = std::shared_ptr <size_t[]> (new size_t[(dim1_ + 1)]); // note the dim1+1
3549     start_index_[0] = 0; // the 1D array starts at 0
3550
3551     // Loop over to find the total length of the 1D array to
3552     // represent the ragged-right array and set the starting 1D index
3553     size_t count = 0;
3554     for (size_t i = 0; i < dim1_; i++){
3555         count += strides_array(i);
3556         start_index_[(i + 1)] = count;
3557     } // end for i
3558     length_ = count;
3559
3560     array_ = std::shared_ptr <T[]> (new T[length_]);
3561 } // End constructor
3562
3563 // Overloaded constructor with a view c array
3564 template <typename T>
3565 RaggedRightArray<T>::RaggedRightArray (ViewCArray<size_t> &strides_array) {
3566     // The length of the stride array is some_dim1;
3567     dim1_  = strides_array.size();
3568
3569     // Create and initialize the starting index of the entries in the 1D array
3570     start_index_ = std::shared_ptr <size_t[]> (new size_t[(dim1_ + 1)]); // note the dim1+1
3571     start_index_[0] = 0; // the 1D array starts at 0
3572
3573     // Loop over to find the total length of the 1D array to
3574     // represent the ragged-right array and set the starting 1D index
3575     size_t count = 0;
```

```
3576     for (size_t i = 0; i < dim1_; i++){
3577         count += strides_array(i);
3578         start_index_[(i + 1)] = count;
3579     } // end for i
3580     length_ = count;
3581 
3582     array_ = std::shared_ptr <T []> (new T[length_]);
3583 } // End constructor
3584 
3585 // Overloaded constructor with a regular cpp array
3586 template <typename T>
3587 RaggedRightArray<T>::RaggedRightArray (size_t *strides_array, size_t dim1){
3588     // The length of the stride array is some_dim1;
3589     dim1_ = dim1;
3590 
3591     // Create and initialize the starting index of the entries in the 1D array
3592     start_index_ = std::shared_ptr <size_t[]> (new size_t[(dim1_ + 1)]); // note the dim1+1
3593     start_index_[0] = 0; // the 1D array starts at 0
3594 
3595     // Loop over to find the total length of the 1D array to
3596     // represent the ragged-right array and set the starting 1D index
3597     size_t count = 0;
3598     for (size_t i = 0; i < dim1_; i++){
3599         count += strides_array[i];
3600         start_index_[(i + 1)] = count;
3601     } // end for i
3602     length_ = count;
3603 
3604     array_ = std::shared_ptr <T []> (new T[length_]);
3605 } // End constructor
3606 
3607 // overloaded constructor for a dynamically built strides_array.
3608 // buffer is the max number of columns needed
3609 template <typename T>
3610 RaggedRightArray<T>::RaggedRightArray (size_t some_dim1, size_t buffer){
3611 
3612     dim1_ = some_dim1;
3613 
3614     // create and initialize the starting index of the entries in the 1D array
3615     start_index_ = std::shared_ptr <size_t[]> (new size_t[(dim1_ + 1)]); // note the dim1+1
3616     //start_index_[0] = 0; // the 1D array starts at 0
3617 
3618     num_saved_ = 0;
3619 
3620     length_ = some_dim1*buffer;
3621     array_ = std::shared_ptr <T []> (new T[length_]);
3622 
3623 } // end constructor
3624 
3625 // Copy constructor
3626 template <typename T>
3627 RaggedRightArray<T>::RaggedRightArray (const RaggedRightArray& temp) {
3628 
3629     if (this != &temp) {
3630         dim1_ = temp.dim1_;
3631         length_ = temp.length_;
3632         num_saved_ = temp.num_saved_;
3633 
3634         // shared_ptr
3635         start_index_ = temp.start_index_;
3636         array_ = temp.array_;
3637     }
3638 }
3639 
3640 // A method to return the stride size
3641 template <typename T>
3642 inline size_t RaggedRightArray<T>::stride(size_t i) const {
3643     // Ensure that i is within bounds
3644     assert(i < dim1_ && "i is greater than dim1_ in RaggedRightArray");
3645 
3646     return start_index_[(i + 1)] - start_index_[i];
3647 }
3648 
3649 // A method to increase the stride size, in other words,
3650 // this is used to build the stride array dynamically
3651 // DO NOT USE with constructors that are given a stride array
3652 template <typename T>
3653 void RaggedRightArray<T>::push_back(size_t i){
3654     num_saved_ ++;
3655     start_index_[i+1] = num_saved_;
3656 }
3657 
3658 // Overload operator() to access data as array(i,j)
3659 // where i=[0:N-1], j=[0:stride(i)]
3660 template <typename T>
3661 inline T& RaggedRightArray<T>::operator()(size_t i, size_t j) const {
3662     // get the 1D array index
```

```
3663     size_t start = start_index_[i];
3664
3665     // asserts
3666     assert(i < dim1_ && "i is out of dim1 bounds in RaggedRightArray");  // die if >= dim1
3667     //assert(j < stride(i) && "j is out of stride bounds in RaggedRightArray");  // die if >= stride
3668     assert(j+start < length_ && "j+start is out of bounds in RaggedRightArray");  // die if >= 1D array
       length)
3669
3670     return array_[j + start];
3671 } // End operator()
3672
3673 //return size
3674 template <typename T>
3675 size_t RaggedRightArray<T>::size() const {
3676     return length_;
3677 }
3678
3679 template <typename T>
3680 RaggedRightArray<T> & RaggedRightArray<T>::operator+= (const size_t i) {
3681     this->num_saved_ ++;
3682     this->start_index_[i+1] = num_saved_;
3683     return *this;
3684 }
3685
3686 //overload = operator
3687 template <typename T>
3688 RaggedRightArray<T> & RaggedRightArray<T>::operator= (const RaggedRightArray &temp) {
3689
3690     if( this != &temp) {
3691         dim1_ = temp.dim1_;
3692         length_ = temp.length_;
3693         num_saved_ = temp.num_saved_;
3694
3695         // shared_ptr
3696         start_index_ = temp.start_index_;
3697         array_ = temp.array_;
3698     }
3699
3700     return *this;
3701 }
3702
3703 template <typename T>
3704 inline T* RaggedRightArray<T>::pointer() const{
3705     return array_.get();
3706 }
3707
3708 template <typename T>
3709 inline size_t* RaggedRightArray<T>::get_starts() const{
3710     return start_index_.get();
3711 }
3712
3713 // Destructor
3714 template <typename T>
3715 RaggedRightArray<T>::~RaggedRightArray () {}
3716
3717 //----end of RaggedRightArray class definitions----
3718
3719 //9. RaggedRightArrayofVectors
3720 template <typename T>
3721 class RaggedRightArrayofVectors {
3722 private:
3723     std::shared_ptr <T[]> start_index_;
3724     std::shared_ptr <T[]> array_;
3725
3726     size_t dim1_, length_, vector_dim_;
3727     size_t num_saved_; // the number saved in the 1D array
3728
3729 public:
3730     // Default constructor
3731     RaggedRightArrayofVectors ();
3732
3733     //--- 3D array access of a ragged right array storing a vector of size vector_dim_ at each (i,j)---
3734
3735     // Overload constructor for a CArray
3736     RaggedRightArrayofVectors (CArray<size_t> &strides_array, size_t vector_dim);
3737
3738     // Overload constructor for a ViewCArray
3739     RaggedRightArrayofVectors (ViewCArray<size_t> &strides_array, size_t vector_dim);
3740
3741     // Overloaded constructor for a traditional array
3742     RaggedRightArrayofVectors (size_t *strides_array, size_t some_dim1, size_t vector_dim);
3743
3744     // Overload constructor for a RaggedRightArray to
3745     // support a dynamically built stride_array
3746     RaggedRightArrayofVectors (size_t some_dim1, size_t buffer, size_t vector_dim);
3747
3748     // Copy constructor
```

```
3749     RaggedRightArrayofVectors (const RaggedRightArrayofVectors& temp);
3750
3751     // A method to return the stride size
3752     size_t stride(size_t i) const;
3753
3754     // A method to return the vector dim
3755     size_t vector_dim() const;
3756
3757     // A method to increase the number of column entries, i.e.,
3758     // the stride size. Used with the constructor for building
3759     // the stride_array dynamically.
3760     // DO NOT USE with the constructures with a strides_array
3761     void push_back(size_t i);
3762
3763     // Overload operator() to access data as array(i,j)
3764     // where i=[0:N-1], j=[stride(i)], k=[0,vector_dim_]
3765     T& operator()(size_t i, size_t j, size_t k) const;
3766
3767     // method to return total size
3768     size_t size() const;
3769
3770     //return pointer
3771     T* pointer() const;
3772
3773     //get row starts array
3774     size_t* get_starts() const;
3775
3776     RaggedRightArrayofVectors& operator+= (const size_t i);
3777
3778     RaggedRightArrayofVectors& operator= (const RaggedRightArrayofVectors &temp);
3779
3780     // Destructor
3781     ~RaggedRightArrayofVectors ( );
3782 }; // End of RaggedRightArray
3783
3784 // Default constructor
3785 template <typename T>
3786 RaggedRightArrayofVectors<T>::RaggedRightArrayofVectors () {
3787     array_ = NULL;
3788     start_index_ = NULL;
3789     length_ = 0;
3790 }
3791
3792
3793 // Overloaded constructor with CArray
3794 template <typename T>
3795 RaggedRightArrayofVectors<T>::RaggedRightArrayofVectors (CArray<size_t> &strides_array, size_t
      vector_dim){
3796     // The length of the stride array is some_dim1;
3797     dim1_  = strides_array.size();
3798     vector_dim_ = vector_dim;
3799
3800     // Create and initialize the starting index of the entries in the 1D array
3801     start_index_ = std::shared_ptr <size_t[]> (new size_t[(dim1_ + 1)]); // note the dim1+1
3802     start_index_[0] = 0; // the 1D array starts at 0
3803
3804     // Loop over to find the total length of the 1D array to
3805     // represent the ragged-right array and set the starting 1D index
3806     size_t count = 0;
3807     for (size_t i = 0; i < dim1_; i++){
3808         count += strides_array(i)*vector_dim_;
3809         start_index_[(i + 1)] = count;
3810     } // end for i
3811     length_ = count;
3812
3813     array_ = std::shared_ptr <T []> (new T[length_]);
3814 } // End constructor
3815
3816 // Overloaded constructor with a view c array
3817 template <typename T>
3818 RaggedRightArrayofVectors<T>::RaggedRightArrayofVectors (ViewCArray<size_t> &strides_array, size_t
      vector_dim) {
3819     // The length of the stride array is some_dim1;
3820     dim1_  = strides_array.size();
3821     vector_dim_ = vector_dim;
3822
3823     // Create and initialize the starting index of the entries in the 1D array
3824     start_index_ = std::shared_ptr <size_t[]> (new size_t[(dim1_ + 1)]); // note the dim1+1
3825     start_index_[0] = 0; // the 1D array starts at 0
3826
3827     // Loop over to find the total length of the 1D array to
3828     // represent the ragged-right array and set the starting 1D index
3829     size_t count = 0;
3830     for (size_t i = 0; i < dim1_; i++){
3831         count += strides_array(i)*vector_dim_;
3832         start_index_[(i + 1)] = count;
3833     } // end for i
```

```
3834     length_ = count;
3835
3836     array_ = std::shared_ptr <T []> (new T[length_]);
3837 } // End constructor
3838
3839 // Overloaded constructor with a regular cpp array
3840 template <typename T>
3841 RaggedRightArrayofVectors<T>::RaggedRightArrayofVectors (size_t *strides_array, size_t dim1, size_t
      vector_dim){
3842     // The length of the stride array is some_dim1;
3843     dim1_ = dim1;
3844     vector_dim_ = vector_dim;
3845
3846     // Create and initialize the starting index of the entries in the 1D array
3847     start_index_ = std::shared_ptr <size_t[]> (new size_t[(dim1_ + 1)]); // note the dim1+1
3848     start_index_[0] = 0; // the 1D array starts at 0
3849
3850     // Loop over to find the total length of the 1D array to
3851     // represent the ragged-right array of vectors and set the starting 1D index
3852     size_t count = 0;
3853     for (size_t i = 0; i < dim1_; i++){
3854         count += strides_array[i]*vector_dim_;
3855         start_index_[(i + 1)] = count;
3856     } // end for i
3857     length_ = count;
3858
3859     array_ = std::shared_ptr <T []> (new T[length_]);
3860 } // End constructor
3861
3862 // overloaded constructor for a dynamically built strides_array.
3863 // buffer is the max number of columns needed
3864 template <typename T>
3865 RaggedRightArrayofVectors<T>::RaggedRightArrayofVectors (size_t some_dim1, size_t buffer, size_t
      vector_dim){
3866
3867     dim1_ = some_dim1;
3868     vector_dim_ = vector_dim;
3869
3870     // create and initialize the starting index of the entries in the 1D array
3871     start_index_ = std::shared_ptr <size_t[]> (new size_t[(dim1_ + 1)]); // note the dim1+1
3872     //start_index_[0] = 0; // the 1D array starts at 0
3873
3874     num_saved_ = 0;
3875
3876     length_ = some_dim1*buffer*vector_dim;
3877     array_ = std::shared_ptr <T []> (new T[some_dim1*buffer]);
3878
3879 } // end constructor
3880
3881 // Copy constructor
3882 template <typename T>
3883 RaggedRightArrayofVectors<T>::RaggedRightArrayofVectors (const RaggedRightArrayofVectors& temp) {
3884
3885     if( this != &temp) {
3886         dim1_ = temp.dim1_;
3887         vector_dim_ = temp.vector_dim_;
3888         length_ = temp.length_;
3889         num_saved_ = temp.num_saved_;
3890
3891         // shared pointer
3892         start_index_ = temp.start_index_;
3893         array_ = temp.start_index_;
3894     }
3895 } // end copy constructor
3896
3897 // A method to return the stride size
3898 template <typename T>
3899 inline size_t RaggedRightArrayofVectors<T>::stride(size_t i) const {
3900     // Ensure that i is within bounds
3901     assert(i < dim1_ && "i is greater than dim1_ in RaggedRightArray");
3902
3903     return (start_index_[(i + 1)] - start_index_[i])/vector_dim_;
3904 }
3905
3906 // A method to increase the stride size, in other words,
3907 // this is used to build the stride array dynamically
3908 // DO NOT USE with constructors that are given a stride array
3909 template <typename T>
3910 void RaggedRightArrayofVectors<T>::push_back(size_t i){
3911     num_saved_ += vector_dim_;
3912     start_index_[i+1] = num_saved_;
3913 }
3914
3915 // Overload operator() to access data as array(i,j,k)
3916 // where i=[0:N-1], j=[0:stride(i)], k=[0:vector_dim_]
3917 template <typename T>
3918 inline T& RaggedRightArrayofVectors<T>::operator()(size_t i, size_t j, size_t k) const {
```

```
3919     // get the 1D array index
3920     size_t start = start_index_[i];
3921 
3922     // asserts
3923     assert(i < dim1_ && "i is out of dim1 bounds in RaggedRightArray");  // die if >= dim1
3924     //assert(j < stride(i) && "j is out of stride bounds in RaggedRightArray");  // die if >= stride
3925     assert(j*vector_dim_+start + k < length_ && "j+start is out of bounds in RaggedRightArray");  //
       die if >= 1D array length)
3926 
3927     return array_[j*vector_dim_ + start + k];
3928 } // End operator()
3929 
3930 //return size
3931 template <typename T>
3932 size_t RaggedRightArrayofVectors<T>::size() const {
3933     return length_;
3934 }
3935 
3936 template <typename T>
3937 RaggedRightArrayofVectors<T> & RaggedRightArrayofVectors<T>::operator+= (const size_t i) {
3938     this->num_saved_ += vector_dim_;
3939     this->start_index_[i+1] = num_saved_;
3940     return *this;
3941 }
3942 
3943 //overload = operator
3944 template <typename T>
3945 RaggedRightArrayofVectors<T> & RaggedRightArrayofVectors<T>::operator= (const RaggedRightArrayofVectors
       &temp) {
3946 
3947     if( this != &temp) {
3948         dim1_ = temp.dim1_;
3949         vector_dim_ = temp.vector_dim_;
3950         length_ = temp.length_;
3951         num_saved_ = temp.num_saved_;
3952 
3953         // shared pointer
3954         start_index_ = temp.start_index_;
3955         array_ = temp.start_index_;
3956     }
3957 
3958     return *this;
3959 }
3960 
3961 template <typename T>
3962 inline T* RaggedRightArrayofVectors<T>::pointer() const{
3963     return array_.get();
3964 }
3965 
3966 template <typename T>
3967 inline size_t* RaggedRightArrayofVectors<T>::get_starts() const{
3968     return start_index_.get();
3969 }
3970 
3971 // Destructor
3972 template <typename T>
3973 RaggedRightArrayofVectors<T>::~RaggedRightArrayofVectors () {}
3974 
3975 //----end of RaggedRightArrayofVectors class definitions----
3976 
3977 //10. RaggedDownArray
3978 template <typename T>
3979 class RaggedDownArray {
3980 private:
3981     std::shared_ptr <size_t[]> start_index_;
3982     std::shared_ptr <T[]> array_;
3983 
3984     size_t dim2_;
3985     size_t length_;
3986     size_t num_saved_; // the number saved in the 1D array
3987 
3988 public:
3989     //default constructor
3990     RaggedDownArray() ;
3991 
3992     //~~~~2D`~~~~
3993     //overload constructor with CArray
3994     RaggedDownArray(CArray<size_t> &strides_array);
3995 
3996     //overload with ViewCArray
3997     RaggedDownArray(ViewCArray <size_t> &strides_array);
3998 
3999     //overload with traditional array
4000     RaggedDownArray(size_t *strides_array, size_t dome_dim1);
4001 
4002     // Overload constructor for a RaggedDownArray to
4003     // support a dynamically built stride_array
```

```
4004     RaggedDownArray (size_t some_dim2, size_t buffer);
4005
4006     // Copy constructor
4007     RaggedDownArray (const RaggedDownArray& temp);
4008
4009     //method to return stride size
4010     size_t stride(size_t j);
4011
4012     // A method to increase the number of column entries, i.e.,
4013     // the stride size. Used with the constructor for building
4014     // the stride_array dynamically.
4015     // DO NOT USE with the constructures with a strides_array
4016     void push_back(size_t j);
4017
4018     //overload () operator to access data as array (i,j)
4019     T& operator()(size_t i, size_t j);
4020
4021     // method to return total size
4022     size_t size();
4023
4024     //return pointer
4025     T* pointer() const;
4026
4027     //get row starts array
4028     size_t* get_starts() const;
4029
4030     //overload = operator
4031     RaggedDownArray& operator= (const RaggedDownArray &temp);
4032
4033     //destructor
4034     ~RaggedDownArray();
4035
4036 }; //~~~~~end of RaggedDownArray class declarations~~~~~~~~
4037
4038 //no dims
4039 template <typename T>
4040 RaggedDownArray<T>::RaggedDownArray() {
4041     array_ = NULL;
4042     start_index_ = NULL;
4043     length_ = 0;
4044 }
4045
4046 //overload constructor with CArray
4047 template <typename T>
4048 RaggedDownArray<T>::RaggedDownArray( CArray <size_t> &strides_array) {
4049     // Length of stride array
4050     //dim2_ = strides_array.size();
4051
4052     // Create and initialize startding indices
4053     start_index_ = std::shared_ptr <size_t[]> (new size_t[(dim2_ + 1)]); // note the dim2+1
4054     start_index_[0] = 0; //1D array starts at 0
4055
4056
4057     //length of strides
4058     dim2_ = strides_array.size();
4059
4060     // Loop to find total length of 1D array
4061     size_t count = 0;
4062     for(size_t j = 0; j < dim2_ ; j++) {
4063         count += strides_array(j);
4064         start_index_[j+1] = count;
4065     }
4066     length_ = count;
4067
4068     array_ = std::shared_ptr <T[]> (new T[length_]);
4069
4070 } // End constructor
4071
4072 // Overload constructor with ViewCArray
4073 template <typename T>
4074 RaggedDownArray<T>::RaggedDownArray( ViewCArray <size_t> &strides_array) {
4075     // Length of strides
4076     //dim2_ = strides_array.size();
4077
4078     //create array for holding start indices
4079     start_index_ = std::shared_ptr <size_t[]> (new size_t[(dim2_ + 1)]); // note the dim2+1
4080     start_index_[0] = 0;
4081
4082     size_t count = 0;
4083     // Loop over to get total length of 1D array
4084     for(size_t j = 0; j < dim2_ ;j++ ) {
4085         count += strides_array(j);
4086         start_index_[j+1] = count;
4087     }
4088     length_ = count;
4089     array_ = std::shared_ptr <T []> (new T[length_]);
4090
```

```
4091 } // End constructor
4092
4093 // Overload constructor with regualar array
4094 template <typename T>
4095 RaggedDownArray<T>::RaggedDownArray( size_t *strides_array, size_t dim2){
4096     // Length of stride array
4097     dim2_ = dim2;
4098
4099     // Create and initialize starting index of entries
4100     start_index_ = std::shared_ptr <size_t[]> (new size_t[(dim2_ + 1)]); // note the dim2+1
4101     start_index_[0] = 0;
4102
4103     // Loop over to find length of 1D array
4104     // Represent ragged down array and set 1D index
4105     size_t count = 0;
4106     for(size_t j = 0; j < dim2_; j++) {
4107         count += strides_array[j];
4108         start_index_[j+1] = count;
4109     }
4110     length_ = count;
4111     array_ = std::shared_ptr <T[]> (new T[length_]);
4112
4113 } //end construnctor
4114
4115 // overloaded constructor for a dynamically built strides_array.
4116 // buffer is the max number of columns needed
4117 template <typename T>
4118 RaggedDownArray<T>::RaggedDownArray (size_t some_dim2, size_t buffer){
4119
4120     dim2_ = some_dim2;
4121
4122     // create and initialize the starting index of the entries in the 1D array
4123     start_index_ = std::shared_ptr <size_t[]> (new size_t[(dim2_ + 1)]); // note the dim2+1
4124     //start_index_[0] = 0; // the 1D array starts at 0
4125
4126     num_saved_ = 0;
4127
4128     length_ = some_dim2*buffer;
4129     array_ = std::shared_ptr <T[]> (new T[length_]);
4130
4131 } // end constructor
4132
4133 // Copy constructor
4134 template <typename T>
4135 RaggedDownArray<T>::RaggedDownArray (const RaggedDownArray& temp) {
4136     if( this != &temp) {
4137         dim2_ = temp.dim2_;
4138         length_ = temp.length_;
4139         num_saved_ = temp.num_saved_;
4140
4141         // shared pointer
4142         start_index_ = temp.start_index_;
4143         array_ = temp.array_;
4144     }
4145 } // end copy constructor
4146
4147 // Check the stride size
4148 template <typename T>
4149 size_t RaggedDownArray<T>::stride(size_t j) {
4150     assert(j < dim2_ && "j is greater than dim2_ in RaggedDownArray");
4151
4152     return start_index_[j+1] - start_index_[j];
4153 }
4154
4155 // A method to increase the stride size, in other words,
4156 // this is used to build the stride array dynamically
4157 // DO NOT USE with constructors that are given a stride array
4158 template <typename T>
4159 void RaggedDownArray<T>::push_back(size_t j){
4160     num_saved_ ++;
4161     start_index_[j+1] = num_saved_;
4162 }
4163
4164 //return size
4165 template <typename T>
4166 size_t RaggedDownArray<T>::size() {
4167     return length_;
4168 }
4169
4170 // overload operator () to access data as an array(i,j)
4171 // Note: i = 0:stride(j), j = 0:N-1
4172 template <typename T>
4173 T& RaggedDownArray<T>::operator()(size_t i, size_t j) {
4174     // Where is the array starting?
4175     // look at start index
4176     size_t start = start_index_[j];
4177
```

```
4178     // Make sure we are within array bounds
4179     assert(i < stride(j) && "i is out of bounds in RaggedDownArray");
4180     assert(j < dim2_ && "j is out of dim2_ bounds in RaggedDownArray");
4181     assert(i+start < length_ && "i+start is out of bounds in RaggedDownArray");  // die if >= 1D array
     length)
4182 
4183     return array_[i + start];
4184 
4185 } // End () operator
4186 
4187 //overload = operator
4188 template <typename T>
4189 RaggedDownArray<T> & RaggedDownArray<T>::operator= (const RaggedDownArray &temp) {
4190 
4191     if( this != &temp) {
4192         dim2_ = temp.dim2_;
4193         length_ = temp.length_;
4194         num_saved_ = temp.num_saved_;
4195 
4196         // shared pointer
4197         start_index_ = temp.start_index_;
4198         array_ = temp.array_;
4199     }
4200 
4201     return *this;
4202 }
4203 
4204 template <typename T>
4205 inline T* RaggedDownArray<T>::pointer() const{
4206     return array_.get();
4207 }
4208 
4209 
4210 template <typename T>
4211 inline size_t* RaggedDownArray<T>::get_starts() const{
4212     return start_index_.get();
4213 }
4214 
4215 // Destructor
4216 template <typename T>
4217 RaggedDownArray<T>::~RaggedDownArray() {}
4218 // End destructor
4219 
4220 
4221 //----end of RaggedDownArray----
4222 
4223 
4224 //11. DynamicRaggedRightArray
4225 
4226 template <typename T>
4227 class DynamicRaggedRightArray {
4228 private:
4229     std::shared_ptr <size_t[]> stride_;
4230     std::shared_ptr <T[]> array_;
4231 
4232     size_t dim1_;
4233     size_t dim2_;
4234     size_t length_;
4235 
4236 public:
4237     // Default constructor
4238     DynamicRaggedRightArray ();
4239 
4240     //--- 2D array access of a ragged right array ---
4241 
4242     // overload constructor
4243     DynamicRaggedRightArray (size_t dim1, size_t dim2);
4244 
4245     // Copy constructor
4246     DynamicRaggedRightArray (const DynamicRaggedRightArray& temp);
4247 
4248     // A method to return or set the stride size
4249     size_t& stride(size_t i) const;
4250 
4251     // A method to return the size
4252     size_t size() const;
4253 
4254     //return pointer
4255     T* pointer() const;
4256 
4257     // Overload operator() to access data as array(i,j),
4258     // where i=[0:N-1], j=[stride(i)]
4259     T& operator()(size_t i, size_t j) const;
4260 
4261     // Overload copy assignment operator
4262     DynamicRaggedRightArray& operator= (const DynamicRaggedRightArray &temp);
4263 
```

```
4264     // Destructor
4265     ~DynamicRaggedRightArray ();
4266 };
4267
4268 //nothing
4269 template <typename T>
4270 DynamicRaggedRightArray<T>::DynamicRaggedRightArray () {
4271     array_ = NULL;
4272     stride_ = NULL;
4273     length_ = 0;
4274 }
4275
4276 // Overloaded constructor
4277 template <typename T>
4278 DynamicRaggedRightArray<T>::DynamicRaggedRightArray (size_t dim1, size_t dim2) {
4279     // The dimensions of the array;
4280     dim1_  = dim1;
4281     dim2_  = dim2;
4282     length_ = dim1*dim2;
4283
4284     // Create memory on the heap for the values
4285     array_ = std::shared_ptr <T[]> (new T[dim1*dim2]);
4286
4287     // Create memory for the stride size in each row
4288     stride_ = std::shared_ptr <size_t[]> (new size_t[dim1]);
4289
4290     // Initialize the stride
4291     for (int i=0; i<dim1_; i++){
4292         stride_[i] = 0;
4293     }
4294
4295     // Start index is always = j + i*dim2
4296 }
4297
4298 // Copy constructor
4299 template <typename T>
4300 DynamicRaggedRightArray<T>::DynamicRaggedRightArray (const DynamicRaggedRightArray& temp) {
4301     if( this != &temp) {
4302         dim1_ = temp.dim1_;
4303         dim2_ = temp.dim2_;
4304         length_ = temp.length_;
4305
4306         // shared pointer
4307         stride_ = temp.stride_;
4308         array_ = temp.array_;
4309     }
4310 } // end copy constructor
4311
4312 // A method to set the stride size for row i
4313 template <typename T>
4314 size_t& DynamicRaggedRightArray<T>::stride(size_t i) const {
4315     return stride_[i];
4316 }
4317
4318 //return size
4319 template <typename T>
4320 size_t DynamicRaggedRightArray<T>::size() const{
4321     return length_;
4322 }
4323
4324 // Overload operator() to access data as array(i,j),
4325 // where i=[0:N-1], j=[0:stride(i)]
4326 template <typename T>
4327 inline T& DynamicRaggedRightArray<T>::operator()(size_t i, size_t j) const {
4328     // Asserts
4329     assert(i < dim1_ && "i is out of dim1 bounds in DynamicRaggedRight");  // die if >= dim1
4330     assert(j < dim2_ && "j is out of dim2 bounds in DynamicRaggedRight");  // die if >= dim2
4331     assert(j < stride_[i] && "j is out of stride bounds in DynamicRaggedRight");  // die if >= stride
4332
4333     return array_[j + i*dim2_];
4334 }
4335
4336 //overload = operator
4337 template <typename T>
4338 inline DynamicRaggedRightArray<T>& DynamicRaggedRightArray<T>::operator= (const DynamicRaggedRightArray
      &temp)
4339 {
4340
4341     if( this != &temp) {
4342         dim1_ = temp.dim1_;
4343         dim2_ = temp.dim2_;
4344         length_ = temp.length_;
4345
4346         // shared pointer
4347         stride_ = temp.stride_;
4348         array_ = temp.array_;
4349     }
```

```
4350
4351     return *this;
4352 }
4353
4354 template <typename T>
4355 inline T* DynamicRaggedRightArray<T>::pointer() const{
4356     return array_.get();
4357 }
4358
4359 // Destructor
4360 template <typename T>
4361 DynamicRaggedRightArray<T>::~DynamicRaggedRightArray() {}
4362
4363
4364
4365
4366 //----end DynamicRaggedRightArray class definitions----
4367
4368
4369 //12. DynamicRaggedDownArray
4370
4371 template <typename T>
4372 class DynamicRaggedDownArray {
4373 private:
4374     std::shared_ptr <size_t[]> stride_;
4375     std::shared_ptr <T[]> array_;
4376
4377     size_t dim1_;
4378     size_t dim2_;
4379     size_t length_;
4380
4381 public:
4382     // Default constructor
4383     DynamicRaggedDownArray ();
4384
4385     //--- 2D array access of a ragged right array ---
4386
4387     // overload constructor
4388     DynamicRaggedDownArray (size_t dim1, size_t dim2);
4389
4390     // Copy constructor
4391     DynamicRaggedDownArray (const DynamicRaggedDownArray& temp);
4392
4393     // A method to return or set the stride size
4394     size_t& stride(size_t j) const;
4395
4396     // A method to return the size
4397     size_t size() const;
4398
4399     // Overload operator() to access data as array(i,j),
4400     // where i=[stride(j)], j=[0:N-1]
4401     T& operator()(size_t i, size_t j) const;
4402
4403     // Overload copy assignment operator
4404     DynamicRaggedDownArray& operator= (const DynamicRaggedDownArray &temp);
4405
4406     //return pointer
4407     T* pointer() const;
4408
4409     // Destructor
4410     ~DynamicRaggedDownArray ();
4411 };
4412
4413 //nothing
4414 template <typename T>
4415 DynamicRaggedDownArray<T>::DynamicRaggedDownArray () {
4416     array_ = NULL;
4417     stride_ = NULL;
4418     length_ = 0;
4419 }
4420
4421 // Overloaded constructor
4422 template <typename T>
4423 DynamicRaggedDownArray<T>::DynamicRaggedDownArray (size_t dim1, size_t dim2) {
4424     // The dimensions of the array;
4425     dim1_  = dim1;
4426     dim2_  = dim2;
4427     length_ = dim1*dim2;
4428
4429     // Create memory on the heap for the values
4430     array_ = std::shared_ptr <T[]> (new T[dim1*dim2]);
4431
4432     // Create memory for the stride size in each row
4433     stride_ = std::shared_ptr <size_t[]> (new size_t[dim2]);
4434
4435     // Initialize the stride
4436     for (int j=0; j<dim2_; j++){
```

```
4437         stride_[j] = 0;
4438     }
4439
4440     // Start index is always = i + j*dim1
4441 }
4442
4443 // Copy constructor
4444 template <typename T>
4445 DynamicRaggedDownArray<T>::DynamicRaggedDownArray (const DynamicRaggedDownArray& temp) {
4446     if( this != &temp) {
4447         dim1_ = temp.dim1_;
4448         dim2_ = temp.dim2_;
4449         length_ = temp.length_;
4450
4451         // shared pointer
4452         stride_ = temp.stride_;
4453         array_ = temp.array_;
4454     }
4455 } // end copy constructor
4456
4457
4458 // A method to set the stride size for column j
4459 template <typename T>
4460 size_t& DynamicRaggedDownArray<T>::stride(size_t j) const {
4461     return stride_[j];
4462 }
4463
4464 //return size
4465 template <typename T>
4466 size_t DynamicRaggedDownArray<T>::size() const{
4467     return length_;
4468 }
4469
4470 // overload operator () to access data as an array(i,j)
4471 // Note: i = 0:stride(j), j = 0:N-1
4472
4473 template <typename T>
4474 inline T& DynamicRaggedDownArray<T>::operator()(size_t i, size_t j) const {
4475     // Asserts
4476     assert(i < dim1_ && "i is out of dim1 bounds in DynamicRaggedDownArray");  // die if >= dim1
4477     assert(j < dim2_ && "j is out of dim2 bounds in DynamicRaggedDownArray");  // die if >= dim2
4478     assert(i < stride_[j] && "i is out of stride bounds in DynamicRaggedDownArray");  // die if >=
       stride
4479
4480     return array_[i + j*dim1_];
4481 }
4482
4483 //overload = operator
4484 template <typename T>
4485 inline DynamicRaggedDownArray<T>& DynamicRaggedDownArray<T>::operator= (const DynamicRaggedDownArray
       &temp)
4486 {
4487
4488     if( this != &temp) {
4489         dim1_ = temp.dim1_;
4490         dim2_ = temp.dim2_;
4491         length_ = temp.length_;
4492
4493         // shared pointer
4494         stride_ = temp.stride_;
4495         array_ = temp.array_;
4496     }
4497
4498     return *this;
4499 }
4500
4501 template <typename T>
4502 inline T* DynamicRaggedDownArray<T>::pointer() const{
4503     return array_.get();
4504 }
4505
4506 // Destructor
4507 template <typename T>
4508 DynamicRaggedDownArray<T>::~DynamicRaggedDownArray() {}
4509
4510 //----end of DynamicRaggedDownArray class definitions-----
4511
4512
4513
4514 //13. SparseRowArray
4515 template <typename T>
4516 class SparseRowArray {
4517 private:
4518     std::shared_ptr <size_t[]> start_index_;
4519     std::shared_ptr <size_t[]> column_index_;
4520
4521     std::shared_ptr <T[]> array_;
```

```
4522
4523     size_t dim1_, length_;
4524
4525 public:
4526     // Default constructor
4527     SparseRowArray ();
4528
4529     //--- 2D array access of a ragged right array ---
4530
4531     // Overload constructor for a CArray
4532     SparseRowArray (CArray<size_t> &strides_array);
4533
4534     // Overload constructor for a ViewCArray
4535     SparseRowArray (ViewCArray<size_t> &strides_array);
4536
4537     // Overloaded constructor for a traditional array
4538     SparseRowArray (size_t *strides_array, size_t some_dim1);
4539
4540     // Copy constructor
4541     SparseRowArray (const SparseRowArray& temp);
4542
4543     // A method to return the stride size
4544     size_t stride(size_t i) const;
4545
4546     // A method to return the column index as array.column_index(i,j)
4547     size_t& column_index(size_t i, size_t j) const;
4548
4549     // A method to access data as array.value(i,j),
4550     // where i=[0:N-1], j=[stride(i)]
4551     T& value(size_t i, size_t j) const;
4552
4553     // A method to return the total size of the array
4554     size_t size() const;
4555
4556     //return pointer
4557     T* pointer() const;
4558
4559     //get row starts array
4560     size_t* get_starts() const;
4561
4562     // overloaded = operator
4563     SparseRowArray& operator=(const SparseRowArray& temp);
4564
4565     // Destructor
4566     ~SparseRowArray ();
4567 };
4568
4569 //Default Constructor
4570 template <typename T>
4571 SparseRowArray<T>::SparseRowArray (){
4572     array_ = NULL;
4573     start_index_ = NULL;
4574     column_index_ = NULL;
4575     length_ = 0;
4576 }
4577 // Overloaded constructor
4578 template <typename T>
4579 SparseRowArray<T>::SparseRowArray (CArray<size_t> &strides_array) {
4580     // The length of the stride array is some_dim1;
4581     dim1_  = strides_array.size();
4582
4583     // Create and initialize the starting index of the entries in the 1D array
4584     start_index_ = std::shared_ptr <size_t[]> (new size_t[dim1_+1]); // note the dim1+1
4585     start_index_[0] = 0; // the 1D array starts at 0
4586
4587     // Loop over to find the total length of the 1D array to
4588     // represent the ragged-right array and set the starting 1D index
4589     size_t count = 0;
4590     for (size_t i = 0; i < dim1_; i++){
4591         count += strides_array(i);
4592         start_index_[i+1] = count;
4593     } // end for i
4594
4595     length_ = count;
4596     array_ = std::shared_ptr <T[]> (new T[count]);
4597     column_index_ = std::shared_ptr <size_t[]> (new size_t[count]);
4598 }
4599
4600
4601 // Overloaded constructor
4602 template <typename T>
4603 SparseRowArray<T>::SparseRowArray (ViewCArray<size_t> &strides_array) {
4604     // The length of the stride array is some_dim1;
4605     dim1_  = strides_array.size();
4606
4607     // Create and initialize the starting index of the entries in the 1D array
4608     start_index_ = std::shared_ptr <size_t[]> (new size_t[dim1_+1]);  // note the dim1+1
```

```
4609     start_index_[0] = 0; // the 1D array starts at 0
4610
4611     // Loop over to find the total length of the 1D array to
4612     // represent the ragged-right array and set the starting 1D index
4613     size_t count = 0;
4614     for (size_t i = 0; i < dim1_; i++){
4615         count += strides_array(i);
4616         start_index_[i+1] = count;
4617     } // end for i
4618
4619     length_ = count;
4620     array_ = std::shared_ptr <T[]> (new T[count]);
4621     column_index_ = std::shared_ptr <size_t[]> (new size_t[count]);
4622 }
4623
4624 // Overloaded constructor
4625 template <typename T>
4626 SparseRowArray<T>::SparseRowArray (size_t *strides_array, size_t dim1) {
4627     // The length of the stride array is some_dim1;
4628     dim1_ = dim1;
4629
4630     // Create and initialize the starting index of the entries in the 1D array
4631     start_index_ = std::shared_ptr <size_t[]> (new size_t[dim1_+1]);  // note the dim1+1
4632     start_index_[0] = 0; // the 1D array starts at 0
4633
4634     // Loop over to find the total length of the 1D array to
4635     // represent the ragged-right array and set the starting 1D index
4636     size_t count = 0;
4637     for (size_t i = 0; i < dim1_; i++){
4638         count += strides_array[i];
4639         start_index_[i+1] = count;
4640     } // end for i
4641
4642     length_ = count;
4643     array_ = std::shared_ptr <T[]> (new T[count]);
4644     column_index_ = std::shared_ptr <size_t[]> (new size_t[count]);
4645 }
4646
4647 // Copy constructor
4648 template <typename T>
4649 SparseRowArray<T>::SparseRowArray (const SparseRowArray& temp) {
4650     if (this != &temp) {
4651         dim1_ = temp.dim1_;
4652         length_ = temp.length_;
4653
4654         // shared pointer
4655         start_index_ = temp.start_index_;
4656         column_index_ = temp.column_index_;
4657         array_ = temp.array_;
4658     }
4659 } // end copy constructor
4660
4661 // A method to return the stride size
4662 template <typename T>
4663 size_t SparseRowArray<T>::stride(size_t i) const {
4664     return start_index_[i+1] - start_index_[i];
4665 }
4666
4667 // A method to return the column index
4668 template <typename T>
4669 size_t& SparseRowArray<T>::column_index(size_t i, size_t j) const {
4670     // Get the 1D array index
4671     size_t start = start_index_[i];
4672
4673     // Asserts
4674     assert(i < dim1_ && "i is out of dim1 bounds in SparseRowArray");  // die if >= dim1
4675     assert(j < stride(i) && "j is out of stride bounds in SparseRowArray");  // die if >= stride
4676
4677     return column_index_[j + start];
4678 }
4679
4680 // Access data as array.value(i,j),
4681 // where i=[0:N-1], j=[0:stride(i)]
4682 template <typename T>
4683 inline T& SparseRowArray<T>::value(size_t i, size_t j) const {
4684     // Get the 1D array index
4685     size_t start = start_index_[i];
4686
4687     // Asserts
4688     assert(i < dim1_ && "i is out of dim1 bounds in sparseRowArray");  // die if >= dim1
4689     assert(j < stride(i) && "j is out of stride bounds in sparseRowArray");  // die if >= stride
4690
4691     return array_[j + start];
4692 }
4693
4694 // return size
4695 template <typename T>
```

```
4696 size_t SparseRowArray<T>::size() const{
4697     return length_;
4698 }
4699
4700 template <typename T>
4701 inline T* SparseRowArray<T>::pointer() const{
4702     return array_.get();
4703 }
4704
4705 template <typename T>
4706 inline size_t* SparseRowArray<T>::get_starts() const{
4707     return start_index_.get();
4708 }
4709
4710 template <typename T>
4711 SparseRowArray<T>& SparseRowArray<T>::operator= (const SparseRowArray& temp) {
4712     if (this != &temp) {
4713         dim1_ = temp.dim1_;
4714         length_ = temp.length_;
4715
4716         // shared pointer
4717         start_index_ = temp.start_index_;
4718         column_index_ = temp.column_index_;
4719         array_ = temp.array_;
4720     }
4721     return *this;
4722 }
4723
4724 // Destructor
4725 template <typename T>
4726 SparseRowArray<T>::~SparseRowArray() {}
4727
4728 //---- end of SparseRowArray class definitions-----
4729
4730
4731 //14. SparseColArray
4732 template <typename T>
4733 class SparseColArray {
4734
4735 private:
4736     std::shared_ptr <size_t[]> start_index_;
4737     std::shared_ptr <size_t[]> row_index_;
4738     std::shared_ptr <T[]> array_;
4739
4740     size_t dim2_, length_;
4741
4742 public:
4743
4744     //default constructor
4745     SparseColArray ();
4746
4747     //constructor with CArray
4748     SparseColArray(CArray<size_t> &strides_array);
4749
4750     //constructor with ViewCArray
4751     SparseColArray(ViewCArray<size_t> &strides_array);
4752
4753     //constructor with regular array
4754     SparseColArray(size_t *strides_array, size_t some_dim1);
4755
4756     // Copy constructor
4757     SparseColArray(const SparseColArray& temp);
4758
4759     //method return stride size
4760     size_t stride(size_t j) const;
4761
4762     //method return row index ass array.row_index(i,j)
4763     size_t& row_index(size_t i, size_t j) const;
4764
4765     //method access data as an array
4766     T& value(size_t i, size_t j) const;
4767
4768     // A method to return the total size of the array
4769     size_t size() const;
4770
4771     //return pointer
4772     T* pointer() const;
4773
4774     //get row starts array
4775     size_t* get_starts() const;
4776
4777     // Overload copy assignment operator
4778     SparseColArray& operator= (const SparseColArray& temp);
4779
4780
4781     //destructor
4782     ~SparseColArray();
```

```
4783 };
4784
4785 //Default Constructor
4786 template <typename T>
4787 SparseColArray<T>::SparseColArray (){
4788     array_ = NULL;
4789     start_index_ = NULL;
4790     row_index_ = NULL;
4791     length_ = 0;
4792 }
4793 //overload constructor with CArray
4794 template <typename T>
4795 SparseColArray<T>::SparseColArray(CArray<size_t> &strides_array) {
4796
4797     dim2_ = strides_array.size();
4798
4799     start_index_ = std::shared_ptr <size_t[]> (new size_t[dim2_+1]);
4800     start_index_[0] = 0;
4801
4802     //loop over to find total length of the 1D array
4803     size_t count = 0;
4804     for(size_t j = 0; j < dim2_; j++) {
4805         count+= strides_array(j);
4806         start_index_[j+1] = count;
4807     }
4808
4809     length_ = count;
4810     array_ = std::shared_ptr <T[]> (new T[count]);
4811     row_index_ = std::shared_ptr <size_t[]> (new size_t[count]);
4812
4813 } //end constructor with CArray
4814
4815
4816 //overload constructor with ViewCArray
4817 template <typename T>
4818 SparseColArray<T>::SparseColArray(ViewCArray<size_t> &strides_array) {
4819
4820     dim2_ = strides_array.size();
4821
4822     //create and initialize starting index of 1D array
4823     start_index_ = std::shared_ptr <size_t[]> (new size_t[dim2_+1]);
4824     start_index_[0] = 0;
4825
4826     //loop over to find total length of 1D array
4827     size_t count = 0;
4828     for(size_t j = 0; j < dim2_ ; j++) {
4829         count += strides_array(j);
4830         start_index_[j+1] = count;
4831     }
4832
4833     length_ = count;
4834     array_ = std::shared_ptr <T[]> (new T[count]);
4835     row_index_ = std::shared_ptr <size_t[]> (new size_t[count]);
4836
4837 } //end constructor
4838
4839 //overload constructor with traditional array
4840 template <typename T>
4841 SparseColArray<T>::SparseColArray(size_t *strides_array, size_t dim2) {
4842
4843     dim2_ = dim2;
4844
4845     //create and initialize the starting index
4846     start_index_ = std::shared_ptr <size_t[]> (new size_t[dim2_ +1]);
4847     start_index_[0] = 0;
4848
4849     //loop over to find the total length of the 1D array
4850     size_t count = 0;
4851     for(size_t j = 0; j < dim2_; j++) {
4852         count += strides_array[j];
4853         start_index_[j+1] = count;
4854     }
4855
4856     length_ = count;
4857     array_ = std::shared_ptr <T[]> (new T[count]);
4858     row_index_ = std::shared_ptr <size_t[]> (new size_t[count]);
4859
4860 } //end constructor
4861
4862 // copy constructor
4863 template <typename T>
4864 SparseColArray<T>::SparseColArray(const SparseColArray& temp) {
4865     if (this != &temp) {
4866         dim2_ = temp.dim2_;
4867         length_ = temp.length_;
4868
4869         // shared pointer
```

```
4870         start_index_ = temp.start_index_;
4871         row_index_ = temp.row_index_;
4872         array_ = temp.array_;
4873     }
4874 } // end copy constructor
4875
4876 //method to return stride size
4877 template <typename T>
4878 size_t SparseColArray<T>::stride(size_t j) const{
4879     return start_index_[j+1] - start_index_[j];
4880 }
4881
4882 //acces data ass arrow.row_index(i,j)
4883 // where i = 0:stride(j), j = 0:N-1
4884 template <typename T>
4885 size_t& SparseColArray<T>::row_index(size_t i, size_t j) const {
4886
4887     //get 1D array index
4888     size_t start = start_index_[j];
4889
4890     //asserts to make sure we are in bounds
4891     assert(i < stride(j) && "i is out of stride bounnds in SparseColArray!");
4892     assert(j < dim2_ && "j is out of dim1 bounds in SparseColArray");
4893
4894     return row_index_[i + start];
4895
4896 } //end row index method
4897
4898
4899 //access values as array.value(i,j)
4900 // where i = 0:stride(j), j = 0:N-1
4901 template <typename T>
4902 T& SparseColArray<T>::value(size_t i, size_t j) const {
4903
4904     size_t start = start_index_[j];
4905
4906     //asserts
4907     assert(i < stride(j) && "i is out of stride boundns in SparseColArray");
4908     assert(j < dim2_ && "j is out of dim1 bounds in SparseColArray");
4909
4910     return array_[i + start];
4911 }
4912
4913 //return size
4914 template <typename T>
4915 size_t SparseColArray<T>::size() const{
4916     return length_;
4917 }
4918
4919 template <typename T>
4920 inline T* SparseColArray<T>::pointer() const{
4921     return array_.get();
4922 }
4923
4924 template <typename T>
4925 inline size_t* SparseColArray<T>::get_starts() const{
4926     return start_index_.get();
4927 }
4928
4929 template <typename T>
4930 SparseColArray<T>& SparseColArray<T>::operator= (const SparseColArray& temp) {
4931     if (this != &temp) {
4932         dim2_ = temp.dim2_;
4933         length_ = temp.length_;
4934
4935         // shared pointer
4936         start_index_ = temp.start_index_;
4937         row_index_ = temp.row_index_;
4938         array_ = temp.array_;
4939     }
4940     return *this;
4941 }
4942
4943 //destructor
4944 template <typename T>
4945 SparseColArray<T>::~SparseColArray() {}
4946
4947
4948 //----end SparseColArray----
4949
4950 // 15 CSRArray
4951 template <typename T>
4952 class CSRArray {
4953   private: // What ought to be private ?
4954     size_t nrows_, ncols_;
4955     size_t nnz_;
4956     std::shared_ptr <T []> data_;
```

```
4957     std::shared_ptr <size_t[]> col_ptr_;
4958     std::shared_ptr <size_t[]> row_ptr_;
4959
4960   public:
4961     CSRArray(CArray<T> data, CArray<T> col_ptrs, CArray<T> row_ptrs, size_t rows, size_t cols);
4962
4963     T& operator()(size_t i, size_t j) const;
4964     void printer(); //debugging tool
4965
4966     size_t getNcols();
4967     size_t getNrows();
4968
4969     // Iterators for row i.
4970     T* begin(size_t i);
4971     T* end(size_t i);
4972
4973     // iterator for the raw data at row i
4974     // i.e. return the index each element is the index in the 1 array
4975     // This as the use of providing a reasonable way to get the column
4976     // index and data value in the case you need both
4977     size_t beginFlat(size_t i);
4978     size_t endFlat(size_t i);
4979
4980     // Get number of non zero elements in row i
4981     size_t nnz(size_t i);
4982     // Get total number of non zero elements
4983     size_t nnz();
4984
4985     // Use the index into the 1d array to get what value is stored there and what is the corresponding
    row
4986     T& getValFlat(size_t k);
4987     size_t getColFlat(size_t k);
4988     // reverse map function from A(i,j) to what element of data/col_pt_ it corersponds to
4989     int flatIndex(size_t i, size_t j);
4990     // Convertor
4991     int toCSC(CArray<T> &data, CArray<size_t> &col_ptrs, CArray<size_t> &row_ptrs);
4992     void todense(CArray<T>& A);
4993     //destructor
4994    ~CSRArray();
4995
4996
4997 };
4998
4999 template <typename T>
5000 CSRArray<T>::CSRArray(CArray<T> data, CArray<T> col_ptrs, CArray<T> row_ptrs, size_t rows, size_t cols
      ){
5001     nrows_ = rows;
5002     ncols_ = cols;
5003     size_t nnz = data.size();
5004     row_ptr_ = std::shared_ptr<size_t []> (new size_t[nrows_ + 1]);
5005     data_ = std::shared_ptr<T []> (new T[nnz+1]);
5006     col_ptr_ = std::shared_ptr<size_t []> (new size_t[nnz]);
5007     size_t i ;
5008     for(i = 0; i < nnz; i++){
5009         data_[i] = data(i);
5010         col_ptr_[i] = col_ptrs(i);
5011     }
5012     for(i = 0; i < nrows_ + 1; i++){
5013         row_ptr_[i] = row_ptrs(i);
5014     }
5015     nnz_ = nnz;
5016 }
5017
5018
5019 template<typename T>
5020 T& CSRArray<T>::operator()(size_t i, size_t j) const {
5021     size_t row_start = row_ptr_[i];
5022     size_t row_end = row_ptr_[i+1];
5023     size_t k;
5024     for(k = 0; k < row_end - row_start; k++){
5025         if(col_ptr_[row_start + k] == j){
5026             return data_[row_start + k];
5027         }
5028     }
5029     data_[nnz_] = (T) NULL;
5030     return data_[nnz_];
5031 }
5032
5033 //debugging tool primarily
5034 template <typename T>
5035 void CSRArray<T>::printer(){
5036     size_t i,j;
5037     for(i = 0; i < nrows_; i++){
5038         for(j = 0; j < ncols_; j++){
5039             printf(" %d ", (*this)(i,j));
5040         }
5041         printf("\n");
```

```
5042     }
5043 }
5044
5045 template<typename T>
5046 void CSRArray<T>::todense(CArray<T>& A){
5047     size_t i,j;
5048     /* use something like this if we assume A is initiliazed with 0s
5049      * size_t cur_row = 0;
5050      * for(i = 0; i < nnz_; i++){
5051         while(row_ptr_[cur_row + 1] <= i){
5052                 cur_row++;
5053         }
5054         A(cur_row, col_ptr_[i]) = data_[i];
5055
5056     } */
5057     for(i = 0; i < nrows_; i++){
5058         for(j = 0; j < ncols_; j++){
5059             A(i,j) = (*this)(i,j);
5060         }
5061     }
5062
5063 }
5064
5065 template<typename T>
5066 size_t CSRArray<T>::getNcols(){
5067     return ncols_;
5068 }
5069
5070 template<typename T>
5071 size_t CSRArray<T>::getNrows(){
5072     return nrows_;
5073 }
5074
5075 template<typename T>
5076 T* CSRArray<T>::begin(size_t i){
5077     if( i > nrows_){
5078         return  NULL; // Access check
5079     }
5080     size_t row_start = row_ptr_[i];
5081     return &data_[row_start];
5082 }
5083
5084 template<typename T>
5085 T* CSRArray<T>::end(size_t i){
5086     if( i > nrows_){
5087         return  NULL; // Access check
5088     }
5089     size_t row_start = row_ptr_[i+1];
5090     return &data_[row_start];
5091 }
5092
5093 template<typename T>
5094 size_t CSRArray<T>::beginFlat(size_t i){
5095     if( i < nrows_){
5096         return row_ptr_[i];
5097     }
5098     return 0;
5099 }
5100
5101 template<typename T>
5102 size_t CSRArray<T>::endFlat(size_t i){
5103     if( i < nrows_){
5104         return row_ptr_[i + 1];
5105     }
5106     return 0;
5107 }
5108
5109 template<typename T>
5110 size_t CSRArray<T>::nnz(){
5111     return row_ptr_[nrows_];
5112 }
5113
5114 template<typename T>
5115 size_t CSRArray<T>::nnz(size_t i){
5116     return row_ptr_[i+1] - row_ptr_[i];
5117 }
5118
5119
5120 template<typename T>
5121 T& CSRArray<T>::getValFlat(size_t k){
5122     return data_[k];
5123 }
5124
5125 template<typename T>
5126 size_t CSRArray<T>::getColFlat(size_t k){
5127     return col_ptr_[k];
5128 }
```

```
5129
5130
5131 template<typename T>
5132 int CSRArray<T>::flatIndex(size_t i, size_t j){
5133     size_t k;
5134     size_t row_start = row_ptr_[i];
5135     size_t row_end = row_ptr_[i+1];
5136     for(k = 0; k < row_end - row_start; k++){
5137         if(col_ptr_[row_start+k] == j){
5138             return row_start+k;
5139         }
5140     }
5141     return  -1;
5142 }
5143
5144 // Assumes that data, col_ptrs, and row_ptrs
5145 // have been allocated size already before this call
5146 // Returns the data in this csr format but as represented as the appropriatte vectors
5147 // for a csc format
5148 template<typename T>
5149 int CSRArray<T>::toCSC(CArray<T> &data, CArray<size_t> &col_ptrs, CArray<size_t> &row_ptrs ){
5150     int nnz_cols[ncols_ + 1];
5151     int col_counts[ncols_];
5152     int i = 0;
5153     // How many elements are each column
5154     for(i =0 ; i < ncols_; i++){
5155         nnz_cols[i] = 0;
5156         col_counts[i] = 0;
5157     }
5158     nnz_cols[ncols_] = 0;
5159     for(i =0; i < nnz_; i++){
5160         nnz_cols[col_ptr_[i] + 1] += 1;
5161     }
5162     // What we actually care about is how many elements are
5163     // in all the  columns preceeding this column.
5164     for(i = 1; i < ncols_; i++){
5165         nnz_cols[i] = nnz_cols[i-1] + nnz_cols[i];
5166     }
5167     size_t row = 1;
5168     // if b is at A(i,j)  stored in csr format
5169     // it needs to go where the where the ith column starts
5170     // + how many things we have put in the "window"
5171     // we allocated for this column already
5172     // For row we simply keep track of what row we are currently in
5173     // as we scan through the 1d array of data.
5174     for(i = 0; i < nnz_; i++){
5175         if(i >= row_ptr_[row]){
5176             row++;
5177         }
5178         int idx = nnz_cols[col_ptr_[i]] + col_counts[col_ptr_[i]];
5179         col_counts[col_ptr_[i]] += 1;
5180         data(idx) = data_[i];
5181         row_ptrs(idx) = row - 1;
5182     }
5183     // I return an int because I thought I might need to return an error code
5184     // Not sure that is true
5185     return 0;
5186 }
5187
5188 template <typename T>
5189 CSRArray<T>::~CSRArray() {}
5190
5191 // End CSRArray
5192
5193 // 16 CSCArray
5194 template <typename T>
5195 class CSCArray {
5196   private: // What ought to be private ?
5197     size_t nrows_, ncols_;
5198     size_t nnz_;
5199     std::shared_ptr <T []> data_;
5200     std::shared_ptr <size_t[]> col_ptr_;
5201     std::shared_ptr <size_t[]> row_ptr_;
5202
5203   public:
5204     CSCArray(CArray<T> data, CArray<T> row_ptrs, CArray<T> row_pts, size_t rows, size_t cols);
5205
5206     T& operator()(size_t i, size_t j) const;
5207     void printer(); //debugging tool
5208
5209     size_t getNcols();
5210     size_t getNrows();
5211
5212     // Iterators for row i.
5213     T* begin(size_t i);
5214     T* end(size_t i);
5215
```

```
5216     // iterator for the raw data at row i
5217     // i.e. return the index each element is the index in the 1 array
5218     // This as the use of providing a reasonable way to get the column
5219     // index and data value in the case you need both
5220     size_t beginFlat(size_t i);
5221     size_t endFlat(size_t i);
5222
5223     // Get number of non zero elements in row i
5224     size_t nnz(size_t i);
5225     // Get total number of non zero elements
5226     size_t nnz();
5227
5228     // Use the index into the 1d array to get what value is stored there and what is the corresponding
       row
5229     T& getValFlat(size_t k);
5230     size_t getColFlat(size_t k);
5231     // reverse map function from A(i,j) to what element of data/col_pt_ it corersponds to
5232     int flatIndex(size_t i, size_t j);
5233     // Convertor
5234     int toCSR(CArray<T> &data, CArray<size_t> &row_ptrs, CArray<size_t> &col_ptrs);
5235     void todense(CArray<T>& A);
5236     //destructor
5237    ~CSCArray();
5238
5239
5240 };
5241
5242 template <typename T>
5243 CSCArray<T>::CSCArray(CArray<T> data, CArray<T> row_ptrs, CArray<T> col_ptrs, size_t rows, size_t cols
      ){
5244     nrows_ = rows;
5245     ncols_ = cols;
5246     size_t nnz = data.size();
5247     col_ptr_ = std::shared_ptr<size_t []> (new size_t[ncols_ + 1]);
5248     data_ = std::shared_ptr<T []> (new T[nnz+1]);
5249     row_ptr_ = std::shared_ptr<size_t []> (new size_t[nnz]);
5250     size_t i ;
5251     for(i = 0; i < nnz; i++){
5252         data_[i] = data(i);
5253         row_ptr_[i] = row_ptrs(i);
5254     }
5255     for(i = 0; i < ncols_ + 1; i++){
5256         col_ptr_[i] = col_ptrs(i);
5257     }
5258     nnz_ = nnz;
5259 }
5260
5261
5262 template<typename T>
5263 T& CSCArray<T>::operator()(size_t i, size_t j) const {
5264     size_t col_start = col_ptr_[j];
5265     size_t col_end = col_ptr_[j+1];
5266     size_t k;
5267     for(k =0; k < col_end - col_start;k++){
5268         if(row_ptr_[col_start + k] == i){
5269                 return data_[col_start + k];
5270         }
5271     }
5272     data_[nnz_] = (T) NULL;
5273     return data_[nnz_];
5274 }
5275
5276 //debugging tool primarily
5277 template <typename T>
5278 void CSCArray<T>::printer(){
5279     size_t i,j;
5280     for(i = 0; i < nrows_; i++){
5281         for(j = 0; j < ncols_; j++){
5282             printf(" %d ", (*this)(i,j));
5283         }
5284         printf("\n");
5285     }
5286 }
5287
5288 template<typename T>
5289 void CSCArray<T>::todense(CArray<T>& A){
5290     size_t i,j;
5291     /* use something like this if we assume A is initiliazed with 0s
5292      * size_t cur_row = 0;
5293      * for(i = 0; i < nnz_; i++){
5294         while(row_ptr_[cur_row + 1] <= i){
5295                 cur_row++;
5296         }
5297         A(cur_row, col_ptr_[i]) = data_[i];
5298
5299     } */
5300     for(j = 0; j < nrows_; j++){
```

```
5301         for(i = 0; i < ncols_; i++){
5302             A(i,j) = (*this)(i,j);
5303         }
5304     }
5305 
5306 }
5307 
5308 template<typename T>
5309 size_t CSCArray<T>::getNcols(){
5310     return ncols_;
5311 }
5312 
5313 template<typename T>
5314 size_t CSCArray<T>::getNrows(){
5315     return nrows_;
5316 }
5317 
5318 template<typename T>
5319 T* CSCArray<T>::begin(size_t i){
5320     if( i > ncols_){
5321         return  NULL; // Access check
5322     }
5323     size_t col_start = col_ptr_[i];
5324     return &data_[col_start];
5325 }
5326 
5327 template<typename T>
5328 T* CSCArray<T>::end(size_t i){
5329     if( i > ncols_){
5330         return  NULL; // Access check
5331     }
5332     size_t col_start = col_ptr_[i+1];
5333     return &data_[col_start];
5334 }
5335 
5336 template<typename T>
5337 size_t CSCArray<T>::beginFlat(size_t i){
5338     if( i < ncols_){
5339         return col_ptr_[i];
5340     }
5341     return 0;
5342 }
5343 
5344 template<typename T>
5345 size_t CSCArray<T>::endFlat(size_t i){
5346     if( i < ncols_){
5347         return col_ptr_[i + 1];
5348     }
5349     return 0;
5350 }
5351 
5352 template<typename T>
5353 size_t CSCArray<T>::nnz(){
5354     return nnz_;
5355 }
5356 
5357 template<typename T>
5358 size_t CSCArray<T>::nnz(size_t i){
5359     return col_ptr_[i+1] - col_ptr_[i];
5360 }
5361 
5362 
5363 template<typename T>
5364 T& CSCArray<T>::getValFlat(size_t k){
5365     return data_[k];
5366 }
5367 
5368 template<typename T>
5369 size_t CSCArray<T>::getColFlat(size_t k){
5370     return col_ptr_[k];
5371 }
5372 
5373 
5374 template<typename T>
5375 int CSCArray<T>::flatIndex(size_t i, size_t j){
5376     size_t col_start = col_ptr_[j];
5377     size_t col_end = col_ptr_[j+1];
5378     size_t k;
5379     for(k =0; k < col_end - col_start;k++){
5380         if(row_ptr_[col_start + k] == i){
5381                 return col_start + k;
5382         }
5383     }
5384     return  -1;
5385 }
5386 
5387 // Assumes that data, col_ptrs, and row_ptrs
```

```
5388 // have been allocated size already before this call
5389 // Returns the data in this csr format but as represented as the appropriatte vectors
5390 // for a csc format
5391 template<typename T>
5392 int CSCArray<T>::toCSR(CArray<T> &data, CArray<size_t> &col_ptrs, CArray<size_t> &row_ptrs ){
5393     int nnz_rows[nrows_ + 1];
5394     int row_counts[nrows_];
5395     int i = 0;
5396     // How many elements are each column
5397     for(i =0 ; i < nrows_; i++){
5398         nnz_rows[i] = 0;
5399         row_counts[i] = 0;
5400     }
5401     nnz_rows[nrows_] = 0;
5402     for(i =0; i < nnz_; i++){
5403         nnz_rows[row_ptr_[i] + 1] += 1;
5404     }
5405     // What we actually care about is how many elements are
5406     // in all the columns preceeding this column.
5407     for(i = 1; i < nrows_; i++){
5408         nnz_rows[i] = nnz_rows[i-1] + nnz_rows[i];
5409     }
5410     size_t col = 1;
5411     // if b is at A(i,j)  stored in csr format
5412     // it needs to go where the where the ith column starts
5413     // + how many things we have put in the "window"
5414     // we allocated for this column already
5415     // For row we simply keep track of what row we are currently in
5416     // as we scan through the 1d array of data.
5417     for(i = 0; i < nnz_; i++){
5418         if(i >= col_ptr_[col]){
5419             col++;
5420         }
5421         int idx = nnz_rows[row_ptr_[i]] + row_counts[row_ptr_[i]];
5422         row_counts[row_ptr_[i]] += 1;
5423         data(idx) = data_[i];
5424         col_ptrs(idx) = col - 1;
5425     }
5426     // I return an int because I thought I might need to return an error code
5427     // Not sure that is true
5428     return 0;
5429 }
5430
5431 template <typename T>
5432 CSCArray<T>::~CSCArray() {}
5433
5434
5435 // End of CSCArray
5436 //==============================================================================
5437 //  end of standard MATAR data-types
5438 //==============================================================================
5439
5446 #ifdef HAVE_KOKKOS
5447 template <typename T, typename Layout = DefaultLayout, typename ExecSpace = DefaultExecSpace, typename
      MemoryTraits = void>
5448 class FArrayKokkos {
5449
5450     using TArray1D = Kokkos::View<T*, Layout, ExecSpace, MemoryTraits>;
5451
5452 private:
5453     size_t dims_[7];
5454     size_t order_;
5455     size_t length_;
5456     TArray1D this_array_;
5457
5458 public:
5459
5463     FArrayKokkos();
5464
5471     FArrayKokkos(size_t dim0, const std::string& tag_string = DEFAULTSTRINGARRAY);
5472
5480     FArrayKokkos(size_t dim0, size_t dim1, const std::string& tag_string = DEFAULTSTRINGARRAY);
5481
5490     FArrayKokkos(size_t dim0, size_t dim1, size_t dim2, const std::string& tag_string =
      DEFAULTSTRINGARRAY);
5491
5492     FArrayKokkos(size_t dim0, size_t dim1, size_t dim2,
5493                  size_t dim3, const std::string& tag_string = DEFAULTSTRINGARRAY);
5494
5495     FArrayKokkos(size_t dim0, size_t dim1, size_t dim2,
5496                  size_t dim3, size_t dim4, const std::string& tag_string = DEFAULTSTRINGARRAY);
5497
5498     FArrayKokkos(size_t dim0, size_t sone_dim2, size_t dim2,
5499                  size_t dim3, size_t dim4, size_t dim5, const std::string& tag_string =
      DEFAULTSTRINGARRAY);
5500
5501     FArrayKokkos(size_t dim0, size_t sone_dim2, size_t dim2,
```

```
5502                 size_t dim3, size_t dim4, size_t dim5,
5503                 size_t dim6, const std::string& tag_string = DEFAULTSTRINGARRAY);
5504
5505     // Overload operator() to acces data
5506     // from 1D to 6D
5507
5508     KOKKOS_INLINE_FUNCTION
5509     T& operator()(size_t i) const;
5510
5511     KOKKOS_INLINE_FUNCTION
5512     T& operator()(size_t i, size_t j) const;
5513
5514     KOKKOS_INLINE_FUNCTION
5515     T& operator() (size_t i, size_t j, size_t k) const;
5516
5517     KOKKOS_INLINE_FUNCTION
5518     T& operator() (size_t i, size_t j, size_t k,
5519                    size_t l) const;
5520
5521     KOKKOS_INLINE_FUNCTION
5522     T& operator() (size_t i, size_t j, size_t k,
5523                    size_t l, size_t m) const;
5524
5525     KOKKOS_INLINE_FUNCTION
5526     T& operator() (size_t i, size_t j, size_t k,
5527                    size_t l, size_t m, size_t n) const;
5528
5529     KOKKOS_INLINE_FUNCTION
5530     T& operator() (size_t i, size_t j, size_t k,
5531                    size_t l, size_t m, size_t n, size_t o) const;
5532
5533     // Overload = operator
5534     KOKKOS_INLINE_FUNCTION
5535     FArrayKokkos& operator= (const FArrayKokkos<T,Layout,ExecSpace,MemoryTraits> &temp);
5536
5537     KOKKOS_INLINE_FUNCTION
5538     size_t size() const;
5539
5540     KOKKOS_INLINE_FUNCTION
5541     size_t extent() const;
5542
5543     KOKKOS_INLINE_FUNCTION
5544     size_t dims(size_t i) const;
5545
5546     KOKKOS_INLINE_FUNCTION
5547     size_t order() const;
5548
5549     KOKKOS_INLINE_FUNCTION
5550     T* pointer() const;
5551
5552     //return kokkos view
5553     KOKKOS_INLINE_FUNCTION
5554     TArray1D get_kokkos_view() const;
5555
5556     // Destructor
5557     KOKKOS_INLINE_FUNCTION
5558     ~FArrayKokkos();
5559
5560 }; //end of FArrayKokkos declarations
5561
5562 // Default constructor
5563 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
5564 FArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::FArrayKokkos() {}
5565
5566 // Overloaded 1D constructor
5567 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
5568 FArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::FArrayKokkos(size_t dim0, const std::string&
       tag_string){
5569     using TArray1D = Kokkos::View<T*, Layout, ExecSpace>;
5570     dims_[0] = dim0;
5571     order_ = 1;
5572     length_ = dim0;
5573     this_array_ = TArray1D(tag_string, length_);
5574 }
5575
5576 // Overloaded 2D constructor
5577 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
5578 FArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::FArrayKokkos(size_t dim0, size_t dim1, const
       std::string& tag_string) {
5579
5580     using TArray1D = Kokkos::View<T*, Layout, ExecSpace>;
5581
5582     dims_[0] = dim0;
5583     dims_[1] = dim1;
5584     order_ = 2;
5585     length_ = (dim0 * dim1);
5586     this_array_ = TArray1D(tag_string, length_);
```

```
5587 }
5588
5589 // Overloaded 3D constructor
5590 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
5591 FArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::FArrayKokkos(size_t dim0, size_t dim1,
5592                                size_t dim2, const std::string& tag_string) {
5593
5594     using TArray1D = Kokkos::View<T*, Layout, ExecSpace>;
5595
5596     dims_[0] = dim0;
5597     dims_[1] = dim1;
5598     dims_[2] = dim2;
5599     order_ = 3;
5600     length_ = (dim0 * dim1 * dim2);
5601     this_array_ = TArray1D(tag_string, length_);
5602 }
5603
5604 // Overloaded 4D constructor
5605 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
5606 FArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::FArrayKokkos(size_t dim0, size_t dim1,
5607                                size_t dim2, size_t dim3, const std::string& tag_string) {
5608
5609     using TArray1D = Kokkos::View<T*, Layout, ExecSpace>;
5610
5611     dims_[0] = dim0;
5612     dims_[1] = dim1;
5613     dims_[2] = dim2;
5614     dims_[3] = dim3;
5615     order_ = 4;
5616     length_ = (dim0 * dim1 * dim2 * dim3);
5617     this_array_ = TArray1D(tag_string, length_);
5618 }
5619
5620 // Overloaded 5D constructor
5621 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
5622 FArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::FArrayKokkos(size_t dim0, size_t dim1,
5623                                size_t dim2, size_t dim3,
5624                                size_t dim4, const std::string& tag_string) {
5625
5626     using TArray1D = Kokkos::View<T*, Layout, ExecSpace>;
5627
5628     dims_[0] = dim0;
5629     dims_[1] = dim1;
5630     dims_[2] = dim2;
5631     dims_[3] = dim3;
5632     dims_[4] = dim4;
5633     order_ = 5;
5634     length_ = (dim0 * dim1 * dim2 * dim3 * dim4);
5635     this_array_ = TArray1D(tag_string, length_);
5636 }
5637
5638 // Overloaded 6D constructor
5639 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
5640 FArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::FArrayKokkos(size_t dim0, size_t dim1,
5641                                size_t dim2, size_t dim3,
5642                                size_t dim4, size_t dim5, const std::string& tag_string) {
5643
5644     using TArray1D = Kokkos::View<T*, Layout, ExecSpace>;
5645
5646     dims_[0] = dim0;
5647     dims_[1] = dim1;
5648     dims_[2] = dim2;
5649     dims_[3] = dim3;
5650     dims_[4] = dim4;
5651     dims_[5] = dim5;
5652     order_ = 6;
5653     length_ = (dim0 * dim1 * dim2 * dim3 * dim4 * dim5);
5654     this_array_ = TArray1D(tag_string, length_);
5655 }
5656
5657 // Overloaded 7D constructor
5658 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
5659 FArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::FArrayKokkos(size_t dim0, size_t dim1,
5660                                size_t dim2, size_t dim3,
5661                                size_t dim4, size_t dim5,
5662                                size_t dim6, const std::string& tag_string) {
5663
5664     using TArray1D = Kokkos::View<T*, Layout, ExecSpace>;
5665
5666     dims_[0] = dim0;
5667     dims_[1] = dim1;
5668     dims_[2] = dim2;
5669     dims_[3] = dim3;
5670     dims_[4] = dim4;
5671     dims_[5] = dim5;
5672     dims_[6] = dim6;
5673     order_ = 7;
```

```
5674     length_ = (dim0 * dim1 * dim2 * dim3 * dim4 * dim5 * dim6);
5675     this_array_ = TArray1D(tag_string, length_);
5676 }
5677
5678 // Definitions of overload operator()
5679 // for 1D to 7D
5680 // Note: the indices for array all start at 0
5681
5682 // 1D
5683 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
5684 KOKKOS_INLINE_FUNCTION
5685 T& FArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::operator()( size_t i) const {
5686     assert(order_ == 1 && "Tensor order (rank) does not match constructor in FArrayKokkos 1D!");
5687     assert( i >= 0 && i < dims_[0] && "i is out of bounds in FArrayKokkos 1D!");
5688     return this_array_(i);
5689 }
5690
5691 // 2D
5692 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
5693 KOKKOS_INLINE_FUNCTION
5694 T& FArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::operator()(size_t i, size_t j) const {
5695     assert(order_ == 2 && "Tensor order (rank) does not match constructor in FArrayKokkos 2D!");
5696     assert(i >= 0 && i < dims_[0] && "i is out of bounds in FArrayKokkos 2D!");
5697     assert(j >= 0 && j < dims_[1] && "j is out of bounds in FArrayKokkos 2D!");
5698     return this_array_(i + (j * dims_[0]));
5699 }
5700
5701 // 3D
5702 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
5703 KOKKOS_INLINE_FUNCTION
5704 T& FArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::operator()(size_t i, size_t j, size_t k) const {
5705     assert(order_ == 3 && "Tensor order (rank) does not match constructor in FArrayKokkos 3D!");
5706     assert(i >= 0 && i < dims_[0] && "i is out of bounds in FArrayKokkos 3D!");
5707     assert(j >= 0 && j < dims_[1] && "j is out of bounds in FArrayKokkos 3D!");
5708     assert(k >= 0 && k < dims_[2] && "k is out of bounds in FArrayKokkos 3D!");
5709     return this_array_(i + (j * dims_[0])
5710                         + (k * dims_[0] * dims_[1]));
5711 }
5712
5713 // 4D
5714 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
5715 KOKKOS_INLINE_FUNCTION
5716 T& FArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::operator()(size_t i, size_t j, size_t k, size_t l)
      const {
5717     assert(order_ == 4 && "Tensor order (rank) does not match constructor in FArrayKokkos 4D!");
5718     assert(i >= 0 && i < dims_[0] && "i is out of bounds in FArrayKokkos 4D!");
5719     assert(j >= 0 && j < dims_[1] && "j is out of bounds in FArrayKokkos 4D!");
5720     assert(k >= 0 && k < dims_[2] && "k is out of bounds in FArrayKokkos 4D!");
5721     assert(l >= 0 && l < dims_[3] && "l is out of bounds in FArrayKokkos 4D!");
5722     return this_array_(i + (j * dims_[0])
5723                         + (k * dims_[0] * dims_[1])
5724                         + (l * dims_[0] * dims_[1] * dims_[2]));
5725 }
5726
5727 // 5D
5728 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
5729 KOKKOS_INLINE_FUNCTION
5730 T& FArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::operator()(size_t i, size_t j, size_t k, size_t l,
5731                                 size_t m) const {
5732     assert(order_ == 5 && "Tensor order (rank) does not match constructor in FArrayKokkos 5D!");
5733     assert(i >= 0 && i < dims_[0] && "i is out of bounds in FArrayKokkos 5D!");
5734     assert(j >= 0 && j < dims_[1] && "j is out of bounds in FArrayKokkos 5D!");
5735     assert(k >= 0 && k < dims_[2] && "k is out of bounds in FArrayKokkos 5D!");
5736     assert(l >= 0 && l < dims_[3] && "l is out of bounds in FArrayKokkos 5D!");
5737     assert(m >= 0 && m < dims_[4] && "m is out of bounds in FArrayKokkos 5D!");
5738     return this_array_(i + (j * dims_[0])
5739                         + (k * dims_[0] * dims_[1])
5740                         + (l * dims_[0] * dims_[1] * dims_[2])
5741                         + (m * dims_[0] * dims_[1] * dims_[2] * dims_[3]));
5742 }
5743
5744 // 6D
5745 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
5746 KOKKOS_INLINE_FUNCTION
5747 T& FArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::operator()(size_t i, size_t j, size_t k, size_t l,
5748                                 size_t m, size_t n) const {
5749     assert(order_ == 6 && "Tensor order (rank) does not match constructor in FArrayKokkos 6D!");
5750     assert(i >= 0 && i < dims_[0] && "i is out of bounds in FArrayKokkos 6D!");
5751     assert(j >= 0 && j < dims_[1] && "j is out of bounds in FArrayKokkos 6D!");
5752     assert(k >= 0 && k < dims_[2] && "k is out of bounds in FArrayKokkos 6D!");
5753     assert(l >= 0 && l < dims_[3] && "l is out of bounds in FArrayKokkos 6D!");
5754     assert(m >= 0 && m < dims_[4] && "m is out of bounds in FArrayKokkos 6D!");
5755     assert(n >= 0 && n < dims_[5] && "n is out of bounds in FArrayKokkos 6D!");
5756     return this_array_(i + (j * dims_[0])
5757                         + (k * dims_[0] * dims_[1])
5758                         + (l * dims_[0] * dims_[1] * dims_[2])
5759                         + (m * dims_[0] * dims_[1] * dims_[2] * dims_[3])
```

```
5760                          + (n * dims_[0] * dims_[1] * dims_[2] * dims_[3] * dims_[4]));
5761 }
5762
5763 // 7D
5764 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
5765 KOKKOS_INLINE_FUNCTION
5766 T& FArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::operator()(size_t i, size_t j, size_t k, size_t l,
5767                                 size_t m, size_t n, size_t o) const {
5768     assert(order_ == 7 && "Tensor order (rank) does not match constructor in FArrayKokkos 7D!");
5769     assert(i >= 0 && i < dims_[0] && "i is out of bounds in FArrayKokkos 7D!");
5770     assert(j >= 0 && j < dims_[1] && "j is out of bounds in FArrayKokkos 7D!");
5771     assert(k >= 0 && k < dims_[2] && "k is out of bounds in FArrayKokkos 7D!");
5772     assert(l >= 0 && l < dims_[3] && "l is out of bounds in FArrayKokkos 7D!");
5773     assert(m >= 0 && m < dims_[4] && "m is out of bounds in FArrayKokkos 7D!");
5774     assert(n >= 0 && n < dims_[5] && "n is out of bounds in FArrayKokkos 7D!");
5775     assert(o >= 0 && o < dims_[6] && "o is out of bounds in FArrayKokkos 7D!");
5776     return this_array_(i + (j * dims_[0])
5777                          + (k * dims_[0] * dims_[1])
5778                          + (l * dims_[0] * dims_[1] * dims_[2])
5779                          + (m * dims_[0] * dims_[1] * dims_[2] * dims_[3])
5780                          + (n * dims_[0] * dims_[1] * dims_[2] * dims_[3] * dims_[4])
5781                          + (o * dims_[0] * dims_[1] * dims_[2] * dims_[3] * dims_[4] * dims_[5]));
5782 }
5783
5784 // Overload = operator
5785 // for object assingment THIS = FArrayKokkos<> TEMP(n,m,,,)
5786 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
5787 KOKKOS_INLINE_FUNCTION
5788 FArrayKokkos<T,Layout,ExecSpace,MemoryTraits>& FArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::operator=
      (const FArrayKokkos<T,Layout,ExecSpace,MemoryTraits>& temp) {
5789     using TArray1D = Kokkos::View<T*, Layout, ExecSpace>;
5790
5791     if (this != &temp) {
5792         for (int iter = 0; iter < temp.order_; iter++){
5793             dims_[iter] = temp.dims_[iter];
5794         } // end for
5795
5796         order_ = temp.order_;
5797         length_ = temp.length_;
5798         this_array_ = temp.this_array_;
5799     }
5800     return *this;
5801 }
5802
5803 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
5804 KOKKOS_INLINE_FUNCTION
5805 size_t FArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::size() const {
5806     return length_;
5807 }
5808
5809 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
5810 KOKKOS_INLINE_FUNCTION
5811 size_t FArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::extent() const {
5812     return length_;
5813 }
5814
5815 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
5816 KOKKOS_INLINE_FUNCTION
5817 size_t FArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::dims(size_t i) const {
5818     assert(i < order_ && "FArrayKokkos order (rank) does not match constructor, dim[i] does not
      exist!");
5819     assert(i >= 0 && dims_[i]>0 && "Access to FArrayKokkos dims is out of bounds!");
5820     return dims_[i];
5821 }
5822
5823 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
5824 KOKKOS_INLINE_FUNCTION
5825 size_t FArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::order() const {
5826     return order_;
5827 }
5828
5829 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
5830 KOKKOS_INLINE_FUNCTION
5831 T* FArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::pointer() const {
5832     return this_array_.data();
5833 }
5834
5835 //return the stored Kokkos view
5836 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
5837 KOKKOS_INLINE_FUNCTION
5838 Kokkos::View<T*, Layout, ExecSpace, MemoryTraits>
      FArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::get_kokkos_view() const {
5839     return this_array_;
5840 }
5841
5842 // Destructor
5843 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
```

```
5844 KOKKOS_INLINE_FUNCTION
5845 FArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::~FArrayKokkos() {}
5846
5848 // End of FArrayKokkos
5850
5854 template <typename T>
5855 class ViewFArrayKokkos {
5856
5857 private:
5858     size_t dims_[7];
5859     size_t order_;
5860     size_t length_;
5861     T* this_array_;
5862
5863 public:
5864     KOKKOS_INLINE_FUNCTION
5865     ViewFArrayKokkos();
5866
5867     KOKKOS_INLINE_FUNCTION
5868     ViewFArrayKokkos(T* some_array, size_t dim0);
5869
5870     KOKKOS_INLINE_FUNCTION
5871     ViewFArrayKokkos(T* some_array, size_t dim0, size_t dim1);
5872
5873     KOKKOS_INLINE_FUNCTION
5874     ViewFArrayKokkos(T* some_array, size_t dim0, size_t dim1, size_t dim2);
5875
5876     KOKKOS_INLINE_FUNCTION
5877     ViewFArrayKokkos(T* some_array, size_t dim0, size_t dim1, size_t dim2,
5878                      size_t dim3);
5879
5880     KOKKOS_INLINE_FUNCTION
5881     ViewFArrayKokkos(T* some_array, size_t dim0, size_t dim1, size_t dim2,
5882                      size_t dim3, size_t dim4);
5883
5884     KOKKOS_INLINE_FUNCTION
5885     ViewFArrayKokkos(T* some_array, size_t dim0, size_t dim1, size_t dim2,
5886                      size_t dim3, size_t dim4, size_t dim5);
5887
5888     KOKKOS_INLINE_FUNCTION
5889     ViewFArrayKokkos(T* some_array, size_t dim0, size_t dim1, size_t dim2,
5890                      size_t dim3, size_t dim4, size_t dim5, size_t dim6);
5891
5892     KOKKOS_INLINE_FUNCTION
5893     T& operator()(size_t i) const;
5894
5895     KOKKOS_INLINE_FUNCTION
5896     T& operator()(size_t i, size_t j) const;
5897
5898     KOKKOS_INLINE_FUNCTION
5899     T& operator()(size_t i, size_t j, size_t k) const;
5900
5901     KOKKOS_INLINE_FUNCTION
5902     T& operator()(size_t i, size_t j, size_t k,
5903                   size_t l) const;
5904
5905     KOKKOS_INLINE_FUNCTION
5906     T& operator()(size_t i, size_t j, size_t k,
5907                   size_t l, size_t m) const;
5908
5909     KOKKOS_INLINE_FUNCTION
5910     T& operator()(size_t i, size_t j, size_t k,
5911                   size_t l, size_t m, size_t n) const;
5912
5913     KOKKOS_INLINE_FUNCTION
5914     T& operator()(size_t i, size_t j, size_t k,
5915                   size_t l, size_t m, size_t n, size_t o) const;
5916
5917
5918     KOKKOS_INLINE_FUNCTION
5919     size_t size() const;
5920
5921     KOKKOS_INLINE_FUNCTION
5922     size_t extent() const;
5923
5924     KOKKOS_INLINE_FUNCTION
5925     size_t dims(size_t i) const;
5926
5927     KOKKOS_INLINE_FUNCTION
5928     size_t order() const;
5929
5930     KOKKOS_INLINE_FUNCTION
5931     T* pointer() const;
5932
5933     KOKKOS_INLINE_FUNCTION
5934     ~ViewFArrayKokkos();
5935
```

```
5936 }; // End of ViewFArrayKokkos declarations
5937
5938 // Default constructor
5939 template <typename T>
5940 KOKKOS_INLINE_FUNCTION
5941 ViewFArrayKokkos<T>::ViewFArrayKokkos() {}
5942
5943 // Overloaded 1D constructor
5944 template <typename T>
5945 KOKKOS_INLINE_FUNCTION
5946 ViewFArrayKokkos<T>::ViewFArrayKokkos(T *some_array, size_t dim0) {
5947     dims_[0] = dim0;
5948     order_ = 1;
5949     length_ = dim0;
5950     this_array_ = some_array;
5951 }
5952
5953 // Overloaded 2D constructor
5954 template <typename T>
5955 KOKKOS_INLINE_FUNCTION
5956 ViewFArrayKokkos<T>::ViewFArrayKokkos(T *some_array, size_t dim0, size_t dim1) {
5957     dims_[0] = dim0;
5958     dims_[1] = dim1;
5959     order_ = 2;
5960     length_ = (dim0 * dim1);
5961     this_array_ = some_array;
5962 }
5963
5964 // Overloaded 3D constructor
5965 template <typename T>
5966 KOKKOS_INLINE_FUNCTION
5967 ViewFArrayKokkos<T>::ViewFArrayKokkos(T *some_array, size_t dim0, size_t dim1,
5968                                       size_t dim2) {
5969     dims_[0] = dim0;
5970     dims_[1] = dim1;
5971     dims_[2] = dim2;
5972     order_ = 3;
5973     length_ = (dim0 * dim1 * dim2);
5974     this_array_ = some_array;
5975 }
5976
5977 // Overloaded 4D constructor
5978 template <typename T>
5979 KOKKOS_INLINE_FUNCTION
5980 ViewFArrayKokkos<T>::ViewFArrayKokkos(T *some_array, size_t dim0, size_t dim1,
5981                                       size_t dim2, size_t dim3) {
5982     dims_[0] = dim0;
5983     dims_[1] = dim1;
5984     dims_[2] = dim2;
5985     dims_[3] = dim3;
5986     order_ = 4;
5987     length_ = (dim0 * dim1 * dim2 * dim3);
5988     this_array_ = some_array;
5989 }
5990
5991 // Overloaded 5D constructor
5992 template <typename T>
5993 KOKKOS_INLINE_FUNCTION
5994 ViewFArrayKokkos<T>::ViewFArrayKokkos(T *some_array, size_t dim0, size_t dim1,
5995                                       size_t dim2, size_t dim3, size_t dim4) {
5996     dims_[0] = dim0;
5997     dims_[1] = dim1;
5998     dims_[2] = dim2;
5999     dims_[3] = dim3;
6000     dims_[4] = dim4;
6001     order_ = 5;
6002     length_ = (dim0 * dim1 * dim2 * dim3 * dim4);
6003     this_array_ = some_array;
6004 }
6005
6006 // Overloaded 6D constructor
6007 template <typename T>
6008 KOKKOS_INLINE_FUNCTION
6009 ViewFArrayKokkos<T>::ViewFArrayKokkos(T *some_array, size_t dim0, size_t dim1,
6010                                       size_t dim2, size_t dim3, size_t dim4,
6011                                       size_t dim5) {
6012     dims_[0] = dim0;
6013     dims_[1] = dim1;
6014     dims_[2] = dim2;
6015     dims_[3] = dim3;
6016     dims_[4] = dim4;
6017     dims_[5] = dim5;
6018     order_ = 6;
6019     length_ = (dim0 * dim1 * dim2 * dim3 * dim4 * dim5);
6020     this_array_ = some_array;
6021 }
6022
```

```
6023 // Overloaded 7D constructor
6024 template <typename T>
6025 KOKKOS_INLINE_FUNCTION
6026 ViewFArrayKokkos<T>::ViewFArrayKokkos(T *some_array, size_t dim0, size_t dim1,
6027                                       size_t dim2, size_t dim3, size_t dim4,
6028                                       size_t dim5, size_t dim6) {
6029     dims_[0] = dim0;
6030     dims_[1] = dim1;
6031     dims_[2] = dim2;
6032     dims_[3] = dim3;
6033     dims_[4] = dim4;
6034     dims_[5] = dim5;
6035     dims_[6] = dim6;
6036     order_ = 7;
6037     length_ = (dim0 * dim1 * dim2 * dim3 * dim4 * dim5 * dim6);
6038     this_array_ = some_array;
6039 }
6040
6041 // Overloaded operator() for 1D array access
6042 template <typename T>
6043 KOKKOS_INLINE_FUNCTION
6044 T& ViewFArrayKokkos<T>::operator()(size_t i) const {
6045     assert(order_ == 1 && "Tensor order (rank) does not match constructor in ViewFArrayKokkos 1D!");
6046     assert(i >= 0 && i < dims_[0] && "i is out of bounds in ViewFArrayKokkos 1D!");
6047     return this_array_[i];
6048 }
6049
6050 //2D
6051 template <typename T>
6052 KOKKOS_INLINE_FUNCTION
6053 T& ViewFArrayKokkos<T>::operator()(size_t i, size_t j) const {
6054     assert(order_ == 2 && "Tensor order (rank) does not match constructor in ViewFArrayKokkos 2D!");
6055     assert(i >= 0 && i < dims_[0] && "i is out of bounds in ViewFArrayKokkos 2D!");
6056     assert(j >= 0 && j < dims_[1] && "j is out of bounds in ViewFArrayKokkos 2D!");
6057     return this_array_[i + (j * dims_[0])];
6058 }
6059
6060 //3D
6061 template <typename T>
6062 KOKKOS_INLINE_FUNCTION
6063 T& ViewFArrayKokkos<T>::operator()(size_t i, size_t j, size_t k) const {
6064     assert(order_ == 3 && "Tensor order (rank) does not match constructor in ViewFArrayKokkos 3D!");
6065     assert(i >= 0 && i < dims_[0] && "i is out of bounds in ViewFArrayKokkos 3D!");
6066     assert(j >= 0 && j < dims_[1] && "j is out of bounds in ViewFArrayKokkos 3D!");
6067     assert(k >= 0 && k < dims_[2] && "k is out of bounds in ViewFArrayKokkos 3D!");
6068     return this_array_[i + (j * dims_[0])
6069                        + (k * dims_[0] * dims_[1])];
6070 }
6071
6072 //4D
6073 template <typename T>
6074 KOKKOS_INLINE_FUNCTION
6075 T& ViewFArrayKokkos<T>::operator()(size_t i, size_t j, size_t k,
6076                                    size_t l) const {
6077     assert(order_ == 4 && "Tensor order (rank) does not match constructor in ViewFArrayKokkos 4D!");
6078     assert(i >= 0 && i < dims_[0] && "i is out of bounds in ViewFArrayKokkos 4D!");
6079     assert(j >= 0 && j < dims_[1] && "j is out of bounds in ViewFArrayKokkos 4D!");
6080     assert(k >= 0 && k < dims_[2] && "k is out of bounds in ViewFArrayKokkos 4D!");
6081     assert(l >= 0 && l < dims_[3] && "l is out of bounds in ViewFArrayKokkos 4D!");
6082     return this_array_[i + (j * dims_[0])
6083                        + (k * dims_[0] * dims_[1])
6084                        + (l * dims_[0] * dims_[1] *dims_[2])];
6085 }
6086
6087 //5D
6088 template <typename T>
6089 KOKKOS_INLINE_FUNCTION
6090 T& ViewFArrayKokkos<T>::operator()(size_t i, size_t j, size_t k,
6091                                    size_t l, size_t m) const {
6092     assert(order_ == 5 && "Tensor order (rank) does not match constructor in ViewFArrayKokkos 5D!");
6093     assert(i >= 0 && i < dims_[0] && "i is out of bounds in ViewFArrayKokkos 5D!");
6094     assert(j >= 0 && j < dims_[1] && "j is out of bounds in ViewFArrayKokkos 5D!");
6095     assert(k >= 0 && k < dims_[2] && "k is out of bounds in ViewFArrayKokkos 5D!");
6096     assert(l >= 0 && l < dims_[3] && "l is out of bounds in ViewFArrayKokkos 5D!");
6097     assert(m >= 0 && m < dims_[4] && "m is out of bounds in ViewFArrayKokkos 5D!");
6098     return this_array_[i + (j * dims_[0])
6099                        + (k * dims_[0] * dims_[1])
6100                        + (l * dims_[0] * dims_[1] * dims_[2])
6101                        + (m * dims_[0] * dims_[1] * dims_[2] * dims_[3])];
6102 }
6103
6104 //6D
6105 template <typename T>
6106 KOKKOS_INLINE_FUNCTION
6107 T& ViewFArrayKokkos<T>::operator()(size_t i, size_t j, size_t k,
6108                                    size_t l, size_t m, size_t n) const {
6109     assert(order_ == 6 && "Tensor order (rank) does not match constructor in ViewFArrayKokkos 6D!");
```

```
6110     assert(i >= 0 && i < dims_[0] && "i is out of bounds in ViewFArrayKokkos 6D!");
6111     assert(j >= 0 && j < dims_[1] && "j is out of bounds in ViewFArrayKokkos 6D!");
6112     assert(k >= 0 && k < dims_[2] && "k is out of bounds in ViewFArrayKokkos 6D!");
6113     assert(l >= 0 && l < dims_[3] && "l is out of bounds in ViewFArrayKokkos 6D!");
6114     assert(m >= 0 && m < dims_[4] && "m is out of bounds in ViewFArrayKokkos 6D!");
6115     assert(n >= 0 && n < dims_[5] && "n is out of bounds in ViewFArrayKokkos 6D!");
6116     return this_array_[i + (j * dims_[0])
6117                         + (k * dims_[0] * dims_[1])
6118                         + (l * dims_[0] * dims_[1] * dims_[2])
6119                         + (m * dims_[0] * dims_[1] * dims_[2] * dims_[3])
6120                         + (n * dims_[0] * dims_[1] * dims_[2] * dims_[3] * dims_[4])];
6121 }
6122
6123 //7D
6124 template <typename T>
6125 KOKKOS_INLINE_FUNCTION
6126 T& ViewFArrayKokkos<T>::operator()(size_t i, size_t j, size_t k,
6127                                    size_t l, size_t m, size_t n,
6128                                    size_t o) const {
6129     assert(order_ == 7 && "Tensor order (rank) does not match constructor in ViewFArrayKokkos 7D!");
6130     assert(i >= 0 && i < dims_[0] && "i is out of bounds in ViewFArrayKokkos 7D!");
6131     assert(j >= 0 && j < dims_[1] && "j is out of bounds in ViewFArrayKokkos 7D!");
6132     assert(k >= 0 && k < dims_[2] && "k is out of bounds in ViewFArrayKokkos 7D!");
6133     assert(l >= 0 && l < dims_[3] && "l is out of bounds in ViewFArrayKokkos 7D!");
6134     assert(m >= 0 && m < dims_[4] && "m is out of bounds in ViewFArrayKokkos 7D!");
6135     assert(n >= 0 && n < dims_[5] && "n is out of bounds in ViewFArrayKokkos 7D!");
6136     assert(o >= 0 && o < dims_[6] && "o is out of bounds in ViewFArrayKokkos 7D!");
6137     return this_array_[i + (j * dims_[0])
6138                         + (k * dims_[0] * dims_[1])
6139                         + (l * dims_[0] * dims_[1] * dims_[2])
6140                         + (m * dims_[0] * dims_[1] * dims_[2] * dims_[3])
6141                         + (n * dims_[0] * dims_[1] * dims_[2] * dims_[3] * dims_[4])
6142                         + (o * dims_[0] * dims_[1] * dims_[2] * dims_[3] * dims_[4] * dims_[5])];
6143 }
6144
6145 template <typename T>
6146 KOKKOS_INLINE_FUNCTION
6147 size_t ViewFArrayKokkos<T>::size() const {
6148     return length_;
6149 }
6150
6151 template <typename T>
6152 KOKKOS_INLINE_FUNCTION
6153 size_t ViewFArrayKokkos<T>::extent() const {
6154     return length_;
6155 }
6156
6157 template <typename T>
6158 KOKKOS_INLINE_FUNCTION
6159 size_t ViewFArrayKokkos<T>::dims(size_t i) const {
6160     assert(i < order_ && "ViewFArrayKokkos order (rank) does not match constructor, dim[i] does not
      exist!");
6161     assert(i >= 0 && dims_[i]>0 && "Access to ViewFArrayKokkos dims is out of bounds!");
6162     return dims_[i];
6163 }
6164
6165 template <typename T>
6166 KOKKOS_INLINE_FUNCTION
6167 size_t ViewFArrayKokkos<T>::order() const {
6168     return order_;
6169 }
6170
6171 template <typename T>
6172 KOKKOS_INLINE_FUNCTION
6173 T* ViewFArrayKokkos<T>::pointer() const {
6174     return this_array_;
6175 }
6176
6177 template <typename T>
6178 KOKKOS_INLINE_FUNCTION
6179 ViewFArrayKokkos<T>::~ViewFArrayKokkos() {}
6180
6182 // End of ViewFArrayKokkos
6184
6188 template <typename T, typename Layout = DefaultLayout, typename ExecSpace = DefaultExecSpace, typename
      MemoryTraits = void>
6189 class FMatrixKokkos {
6190
6191     using TArray1D = Kokkos::View<T*, Layout, ExecSpace, MemoryTraits>;
6192
6193 private:
6194
6195     size_t dims_[7];
6196     size_t order_;
6197     size_t length_;
6198     TArray1D this_matrix_;
6199
```

```
6200 public:
6201     FMatrixKokkos();
6202 
6203     FMatrixKokkos(size_t dim1, const std::string& tag_string = DEFAULTSTRINGMATRIX);
6204 
6205     FMatrixKokkos(size_t dim1, size_t dim2, const std::string& tag_string = DEFAULTSTRINGMATRIX);
6206 
6207     FMatrixKokkos(size_t dim1, size_t dim2, size_t dim3, const std::string& tag_string =
      DEFAULTSTRINGMATRIX);
6208 
6209     FMatrixKokkos(size_t dim1, size_t dim2, size_t dim3,
6210                   size_t dim4, const std::string& tag_string = DEFAULTSTRINGMATRIX);
6211 
6212     FMatrixKokkos(size_t dim1, size_t dim2, size_t dim3,
6213                   size_t dim4, size_t dim5, const std::string& tag_string = DEFAULTSTRINGMATRIX);
6214 
6215     FMatrixKokkos(size_t dim1, size_t dim2, size_t dim3,
6216                   size_t dim4, size_t dim5, size_t dim6, const std::string& tag_string =
      DEFAULTSTRINGMATRIX);
6217 
6218     FMatrixKokkos(size_t dim1, size_t dim2, size_t dim3,
6219                   size_t dim4, size_t dim5, size_t dim6,
6220                   size_t dim7, const std::string& tag_string = DEFAULTSTRINGMATRIX);
6221 
6222     KOKKOS_INLINE_FUNCTION
6223     T& operator()(size_t i) const;
6224 
6225     KOKKOS_INLINE_FUNCTION
6226     T& operator()(size_t i, size_t j) const;
6227 
6228     KOKKOS_INLINE_FUNCTION
6229     T& operator()(size_t i, size_t j, size_t k) const;
6230 
6231     KOKKOS_INLINE_FUNCTION
6232     T& operator()(size_t i, size_t j, size_t k, size_t l) const;
6233 
6234     KOKKOS_INLINE_FUNCTION
6235     T& operator()(size_t i, size_t j, size_t k, size_t l, size_t m) const;
6236 
6237     KOKKOS_INLINE_FUNCTION
6238     T& operator()(size_t i, size_t j, size_t k, size_t l, size_t m,
6239                   size_t n) const;
6240 
6241     KOKKOS_INLINE_FUNCTION
6242     T& operator()(size_t i, size_t j, size_t k, size_t l, size_t m,
6243                   size_t n, size_t o) const;
6244 
6245     KOKKOS_INLINE_FUNCTION
6246     FMatrixKokkos& operator=(const FMatrixKokkos& temp);
6247 
6248     KOKKOS_INLINE_FUNCTION
6249     size_t size() const;
6250 
6251     KOKKOS_INLINE_FUNCTION
6252     size_t extent() const;
6253 
6254     KOKKOS_INLINE_FUNCTION
6255     size_t dims(size_t i) const;
6256 
6257     KOKKOS_INLINE_FUNCTION
6258     size_t order() const;
6259 
6260     KOKKOS_INLINE_FUNCTION
6261     T* pointer() const;
6262 
6263     //return kokkos view
6264     KOKKOS_INLINE_FUNCTION
6265     TArray1D get_kokkos_view() const;
6266 
6267     KOKKOS_INLINE_FUNCTION
6268     ~FMatrixKokkos();
6269 
6270 }; // End of FMatrixKokkos
6271 
6272 // Default constructor
6273 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
6274 FMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::FMatrixKokkos() {}
6275 
6276 // Overloaded 1D constructor
6277 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
6278 FMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::FMatrixKokkos(size_t dim1, const std::string&
      tag_string) {
6279     using TArray1D = Kokkos::View<T*, Layout, ExecSpace>;
6280 
6281     dims_[0] = dim1;
6282     order_ = 1;
6283     length_ = dim1;
```

```
6284     this_matrix_ = TArray1D(tag_string, length_);
6285 }
6286
6287 // Overloaded 2D constructor
6288 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
6289 FMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::FMatrixKokkos(size_t dim1, size_t dim2, const
      std::string& tag_string) {
6290     using TArray1D = Kokkos::View<T*, Layout, ExecSpace>;
6291
6292     dims_[0] = dim1;
6293     dims_[1] = dim2;
6294     order_ = 2;
6295     length_ = (dim1 * dim2);
6296     this_matrix_ = TArray1D(tag_string, length_);
6297 }
6298
6299 // Overloaded 3D constructor
6300 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
6301 FMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::FMatrixKokkos(size_t dim1, size_t dim2,
6302                                 size_t dim3, const std::string& tag_string) {
6303     using TArray1D = Kokkos::View<T*, Layout, ExecSpace>;
6304
6305     dims_[0] = dim1;
6306     dims_[1] = dim2;
6307     dims_[2] = dim3;
6308     order_ = 3;
6309     length_ = (dim1 * dim2 * dim3);
6310     this_matrix_ = TArray1D(tag_string, length_);
6311 }
6312
6313 // Overloaded 4D constructor
6314 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
6315 FMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::FMatrixKokkos(size_t dim1, size_t dim2,
6316                                 size_t dim3, size_t dim4, const std::string& tag_string) {
6317     using TArray1D = Kokkos::View<T*, Layout, ExecSpace>;
6318
6319     dims_[0] = dim1;
6320     dims_[1] = dim2;
6321     dims_[2] = dim3;
6322     dims_[3] = dim4;
6323     order_ = 4;
6324     length_ = (dim1 * dim2 * dim3 * dim4);
6325     this_matrix_ = TArray1D(tag_string, length_);
6326 }
6327
6328 // Overloaded 5D constructor
6329 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
6330 FMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::FMatrixKokkos(size_t dim1, size_t dim2,
6331                                 size_t dim3, size_t dim4,
6332                                 size_t dim5, const std::string& tag_string) {
6333     using TArray1D = Kokkos::View<T*, Layout, ExecSpace>;
6334
6335     dims_[0] = dim1;
6336     dims_[1] = dim2;
6337     dims_[2] = dim3;
6338     dims_[3] = dim4;
6339     dims_[4] = dim5;
6340     order_ = 5;
6341     length_ = (dim1 * dim2 * dim3 * dim4 * dim5);
6342     this_matrix_ = TArray1D(tag_string, length_);
6343 }
6344
6345 // Overloaded 5D constructor
6346 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
6347 FMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::FMatrixKokkos(size_t dim1, size_t dim2,
6348                                 size_t dim3, size_t dim4,
6349                                 size_t dim5, size_t dim6, const std::string& tag_string) {
6350     using TArray1D = Kokkos::View<T*, Layout, ExecSpace>;
6351
6352     dims_[0] = dim1;
6353     dims_[1] = dim2;
6354     dims_[2] = dim3;
6355     dims_[3] = dim4;
6356     dims_[4] = dim5;
6357     dims_[5] = dim6;
6358     order_ = 6;
6359     length_ = (dim1 * dim2 * dim3 * dim4 * dim5 * dim6);
6360     this_matrix_ = TArray1D(tag_string, length_);
6361 }
6362
6363 // Overloaded 5D constructor
6364 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
6365 FMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::FMatrixKokkos(size_t dim1, size_t dim2,
6366                                 size_t dim3, size_t dim4,
6367                                 size_t dim5, size_t dim6,
6368                                 size_t dim7, const std::string& tag_string) {
6369     using TArray1D = Kokkos::View<T*, Layout, ExecSpace>;
```

```
6370
6371     dims_[0] = dim1;
6372     dims_[1] = dim2;
6373     dims_[2] = dim3;
6374     dims_[3] = dim4;
6375     dims_[4] = dim5;
6376     dims_[5] = dim6;
6377     dims_[6] = dim7;
6378     order_ = 7;
6379     length_ = (dim1 * dim2 * dim3 * dim4 * dim5 * dim6 * dim7);
6380     this_matrix_ = TArray1D(tag_string, length_);
6381 }
6382
6383 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
6384 KOKKOS_INLINE_FUNCTION
6385 T& FMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::operator()(size_t i) const {
6386     assert(order_ == 1 && "Tensor order (rank) does not match constructor in FMatrixKokkos 1D!");
6387     assert(i >= 1 && i <= dims_[0] && "i is out of bounds in FMatrixKokkos in 1D!");
6388     return this_matrix_((i - 1));
6389 }
6390
6391 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
6392 KOKKOS_INLINE_FUNCTION
6393 T& FMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::operator()(size_t i, size_t j) const {
6394     assert(order_ == 2 && "Tensor order (rank) does not match constructor in FMatrixKokkos 2D!");
6395     assert(i >= 1 && i <= dims_[0] && "i is out of bounds in FMatrixKokkos in 2D!");
6396     assert(j >= 1 && j <= dims_[1] && "j is out of bounds in FMatrixKokkos in 2D!");
6397     return this_matrix_((i - 1) + ((j - 1) * dims_[0]));
6398 }
6399
6400 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
6401 KOKKOS_INLINE_FUNCTION
6402 T& FMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::operator()(size_t i, size_t j, size_t k) const {
6403     assert(order_ == 3 && "Tensor order (rank) does not match constructor in FMatrixKokkos 3D!");
6404     assert(i >= 1 && i <= dims_[0] && "i is out of bounds in FMatrixKokkos in 3D!");
6405     assert(j >= 1 && j <= dims_[1] && "j is out of bounds in FMatrixKokkos in 3D!");
6406     assert(k >= 1 && k <= dims_[2] && "k is out of bounds in FMatrixKokkos in 3D!");
6407     return this_matrix_((i - 1) + ((j - 1) * dims_[0])
6408                                 + ((k - 1) * dims_[0] * dims_[1]));
6409 }
6410
6411 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
6412 KOKKOS_INLINE_FUNCTION
6413 T& FMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::operator()(size_t i, size_t j, size_t k, size_t l)
      const {
6414     assert(order_ == 4 && "Tensor order (rank) does not match constructor in FMatrixKokkos 4D!");
6415     assert(i >= 1 && i <= dims_[0] && "i is out of bounds in FMatrixKokkos in 4D!");
6416     assert(j >= 1 && j <= dims_[1] && "j is out of bounds in FMatrixKokkos in 4D!");
6417     assert(k >= 1 && k <= dims_[2] && "k is out of bounds in FMatrixKokkos in 4D!");
6418     assert(l >= 1 && l <= dims_[3] && "l is out of bounds in FMatrixKokkos in 4D!");
6419     return this_matrix_((i - 1) + ((j - 1) * dims_[0])
6420                                 + ((k - 1) * dims_[0] * dims_[1])
6421                                 + ((l - 1) * dims_[0] * dims_[1] * dims_[2]));
6422 }
6423
6424 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
6425 KOKKOS_INLINE_FUNCTION
6426 T& FMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::operator()(size_t i, size_t j, size_t k, size_t l,
6427                                 size_t m) const {
6428     assert(order_ == 5 && "Tensor order (rank) does not match constructor in FMatrixKokkos 5D!");
6429     assert(i >= 1 && i <= dims_[0] && "i is out of bounds in FMatrixKokkos in 5D!");
6430     assert(j >= 1 && j <= dims_[1] && "j is out of bounds in FMatrixKokkos in 5D!");
6431     assert(k >= 1 && k <= dims_[2] && "k is out of bounds in FMatrixKokkos in 5D!");
6432     assert(l >= 1 && l <= dims_[3] && "l is out of bounds in FMatrixKokkos in 5D!");
6433     assert(m >= 1 && m <= dims_[4] && "m is out of bounds in FMatrixKokkos in 5D!");
6434     return this_matrix_((i - 1) + ((j - 1) * dims_[0])
6435                                 + ((k - 1) * dims_[0] * dims_[1])
6436                                 + ((l - 1) * dims_[0] * dims_[1] * dims_[2])
6437                                 + ((m - 1) * dims_[0] * dims_[1] * dims_[2] * dims_[3]));
6438 }
6439
6440 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
6441 KOKKOS_INLINE_FUNCTION
6442 T& FMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::operator()(size_t i, size_t j, size_t k, size_t l,
6443                                 size_t m, size_t n) const {
6444     assert(order_ == 6 && "Tensor order (rank) does not match constructor in FMatrixKokkos 6D!");
6445     assert(i >= 1 && i <= dims_[0] && "i is out of bounds in FMatrixKokkos in 6D!");
6446     assert(j >= 1 && j <= dims_[1] && "j is out of bounds in FMatrixKokkos in 6D!");
6447     assert(k >= 1 && k <= dims_[2] && "k is out of bounds in FMatrixKokkos in 6D!");
6448     assert(l >= 1 && l <= dims_[3] && "l is out of bounds in FMatrixKokkos in 6D!");
6449     assert(m >= 1 && m <= dims_[4] && "m is out of bounds in FMatrixKokkos in 6D!");
6450     assert(n >= 1 && n <= dims_[5] && "n is out of bounds in FMatrixKokkos in 6D!");
6451     return this_matrix_[(i - 1) + ((j - 1) * dims_[0])
6452                                 + ((k - 1) * dims_[0] * dims_[1])
6453                                 + ((l - 1) * dims_[0] * dims_[1] * dims_[2])
6454                                 + ((m - 1) * dims_[0] * dims_[1] * dims_[2] * dims_[3])
6455                                 + ((n - 1) * dims_[0] * dims_[1] * dims_[2] * dims_[3] * dims_[4])];
```

```
6456 }
6457
6458 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
6459 KOKKOS_INLINE_FUNCTION
6460 T& FMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::operator()(size_t i, size_t j, size_t k, size_t l,
6461                                 size_t m, size_t n, size_t o) const {
6462     assert(order_ == 7 && "Tensor order (rank) does not match constructor in FMatrixKokkos 7D!");
6463     assert(i >= 1 && i <= dims_[0] && "i is out of bounds in FMatrixKokkos in 7D!");
6464     assert(j >= 1 && j <= dims_[1] && "j is out of bounds in FMatrixKokkos in 7D!");
6465     assert(k >= 1 && k <= dims_[2] && "k is out of bounds in FMatrixKokkos in 7D!");
6466     assert(l >= 1 && l <= dims_[3] && "l is out of bounds in FMatrixKokkos in 7D!");
6467     assert(m >= 1 && m <= dims_[4] && "m is out of bounds in FMatrixKokkos in 7D!");
6468     assert(n >= 1 && n <= dims_[5] && "n is out of bounds in FMatrixKokkos in 7D!");
6469     assert(o >= 1 && o <= dims_[6] && "o is out of bounds in FMatrixKokkos in 7D!");
6470     return this_matrix_[(i - 1) + ((j - 1) * dims_[0])
6471                                 + ((k - 1) * dims_[0] * dims_[1])
6472                                 + ((l - 1) * dims_[0] * dims_[1] * dims_[2])
6473                                 + ((m - 1) * dims_[0] * dims_[1] * dims_[2] * dims_[3])
6474                                 + ((n - 1) * dims_[0] * dims_[1] * dims_[2] * dims_[3] * dims_[4])
6475                                 + ((o - 1) * dims_[0] * dims_[1] * dims_[2] * dims_[3] * dims_[4] *
       dims_[5])];
6476 }
6477
6478 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
6479 KOKKOS_INLINE_FUNCTION
6480 FMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>&
      FMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::operator=(const
      FMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>& temp) {
6481     // Do nothing if the assignment is of the form x = x
6482     if (this != &temp) {
6483         for (int iter = 0; iter < temp.order_; iter++){
6484             dims_[iter] = temp.dims_[iter];
6485         } // end for
6486
6487         order_ = temp.order_;
6488         length_ = temp.length_;
6489         this_matrix_ = temp.this_matrix_;
6490     }
6491     return *this;
6492 }
6493
6494
6495
6496 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
6497 KOKKOS_INLINE_FUNCTION
6498 size_t FMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::size() const {
6499     return length_;
6500 }
6501
6502 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
6503 KOKKOS_INLINE_FUNCTION
6504 size_t FMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::extent() const {
6505     return length_;
6506 }
6507
6508 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
6509 KOKKOS_INLINE_FUNCTION
6510 size_t FMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::dims(size_t i) const {
6511     i--;
6512     assert(i < order_ && "FMatrixKokkos order (rank) does not match constructor, dim[i] does not
      exist!");
6513     assert(i >= 0 && dims_[i]>0 && "Access to FMatrixKokkos dims is out of bounds!");
6514     return dims_[i];
6515 }
6516
6517 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
6518 KOKKOS_INLINE_FUNCTION
6519 size_t FMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::order() const {
6520     return order_;
6521 }
6522
6523 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
6524 KOKKOS_INLINE_FUNCTION
6525 T* FMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::pointer() const {
6526     return this_matrix_.data();
6527 }
6528
6529 //return the stored Kokkos view
6530 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
6531 KOKKOS_INLINE_FUNCTION
6532 Kokkos::View<T*, Layout, ExecSpace, MemoryTraits>
      FMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::get_kokkos_view() const {
6533     return this_matrix_;
6534 }
6535
6536 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
6537 KOKKOS_INLINE_FUNCTION
```

```
6538 FMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::~FMatrixKokkos() {}
6539
6541 // End of FMatrixKokkos
6543
6547 template <typename T>
6548 class ViewFMatrixKokkos {
6549
6550 private:
6551
6552     size_t dims_[7];
6553     size_t order_;
6554     size_t length_;
6555     T* this_matrix_;
6556
6557 public:
6558
6559     KOKKOS_INLINE_FUNCTION
6560     ViewFMatrixKokkos();
6561
6562     KOKKOS_INLINE_FUNCTION
6563     ViewFMatrixKokkos(T* some_matrix, size_t dim1);
6564
6565     KOKKOS_INLINE_FUNCTION
6566     ViewFMatrixKokkos(T* some_matrix, size_t dim1, size_t dim2);
6567
6568     KOKKOS_INLINE_FUNCTION
6569     ViewFMatrixKokkos(T* some_matrix, size_t dim1, size_t dim2,
6570                       size_t dim3);
6571
6572     KOKKOS_INLINE_FUNCTION
6573     ViewFMatrixKokkos(T* some_matrix, size_t dim1, size_t dim2,
6574                       size_t dim3, size_t dim4);
6575
6576     KOKKOS_INLINE_FUNCTION
6577     ViewFMatrixKokkos(T* some_matrix, size_t dim1, size_t dim2,
6578                       size_t dim3, size_t dim4, size_t dim5);
6579
6580     KOKKOS_INLINE_FUNCTION
6581     ViewFMatrixKokkos(T* some_matrix, size_t dim1, size_t dim2,
6582                       size_t dim3, size_t dim4, size_t dim5,
6583                       size_t dim6);
6584
6585     KOKKOS_INLINE_FUNCTION
6586     ViewFMatrixKokkos(T* some_matrix, size_t dim1, size_t dim2,
6587                       size_t dim3, size_t dim4, size_t dim5,
6588                       size_t dim6, size_t dim7);
6589
6590     KOKKOS_INLINE_FUNCTION
6591     T& operator()(size_t i) const;
6592
6593     KOKKOS_INLINE_FUNCTION
6594     T& operator()(size_t i, size_t j) const;
6595
6596     KOKKOS_INLINE_FUNCTION
6597     T& operator()(size_t i, size_t j, size_t k) const;
6598
6599     KOKKOS_INLINE_FUNCTION
6600     T& operator()(size_t i, size_t j, size_t k, size_t l) const;
6601
6602     KOKKOS_INLINE_FUNCTION
6603     T& operator()(size_t i, size_t j, size_t k, size_t l, size_t m) const;
6604
6605     KOKKOS_INLINE_FUNCTION
6606     T& operator()(size_t i, size_t j, size_t k, size_t l, size_t m,
6607                   size_t n) const;
6608
6609     KOKKOS_INLINE_FUNCTION
6610     T& operator()(size_t i, size_t j, size_t k, size_t l, size_t m,
6611                   size_t n, size_t o) const;
6612
6613     KOKKOS_INLINE_FUNCTION
6614     size_t size() const;
6615
6616     KOKKOS_INLINE_FUNCTION
6617     size_t extent() const;
6618
6619     KOKKOS_INLINE_FUNCTION
6620     size_t dims(size_t i) const;
6621
6622     KOKKOS_INLINE_FUNCTION
6623     size_t order() const;
6624
6625     KOKKOS_INLINE_FUNCTION
6626     T* pointer() const;
6627
6628     KOKKOS_INLINE_FUNCTION
6629     ~ViewFMatrixKokkos();
```

```
6630
6631 }; // end of ViewFMatrixKokkos
6632
6633 // Default constructor
6634 template <typename T>
6635 KOKKOS_INLINE_FUNCTION
6636 ViewFMatrixKokkos<T>::ViewFMatrixKokkos() {}
6637
6638 // Overloaded 1D constructor
6639 template <typename T>
6640 KOKKOS_INLINE_FUNCTION
6641 ViewFMatrixKokkos<T>::ViewFMatrixKokkos(T* some_matrix, size_t dim1) {
6642     dims_[0] = dim1;
6643     order_ = 1;
6644     length_ = dim1;
6645     this_matrix_ = some_matrix;
6646 }
6647
6648 // Overloaded 2D constructor
6649 template <typename T>
6650 KOKKOS_INLINE_FUNCTION
6651 ViewFMatrixKokkos<T>::ViewFMatrixKokkos(T* some_matrix, size_t dim1,
6652                                         size_t dim2) {
6653     dims_[0] = dim1;
6654     dims_[1] = dim2;
6655     order_ = 2;
6656     length_ = (dim1 * dim2);
6657     this_matrix_ = some_matrix;
6658 }
6659
6660 // Overloaded 3D constructor
6661 template <typename T>
6662 KOKKOS_INLINE_FUNCTION
6663 ViewFMatrixKokkos<T>::ViewFMatrixKokkos(T* some_matrix, size_t dim1,
6664                                         size_t dim2, size_t dim3) {
6665     dims_[0] = dim1;
6666     dims_[1] = dim2;
6667     dims_[2] = dim3;
6668     order_ = 3;
6669     length_ = (dim1 * dim2 * dim3);
6670     this_matrix_ = some_matrix;
6671 }
6672
6673 // Overloaded 4D constructor
6674 template <typename T>
6675 KOKKOS_INLINE_FUNCTION
6676 ViewFMatrixKokkos<T>::ViewFMatrixKokkos(T* some_matrix, size_t dim1,
6677                                         size_t dim2, size_t dim3,
6678                                         size_t dim4) {
6679     dims_[0] = dim1;
6680     dims_[1] = dim2;
6681     dims_[2] = dim3;
6682     dims_[3] = dim4;
6683     order_ = 4;
6684     length_ = (dim1 * dim2 * dim3 * dim4);
6685     this_matrix_ = some_matrix;
6686 }
6687
6688 // Overloaded 5D constructor
6689 template <typename T>
6690 KOKKOS_INLINE_FUNCTION
6691 ViewFMatrixKokkos<T>::ViewFMatrixKokkos(T* some_matrix, size_t dim1,
6692                                         size_t dim2, size_t dim3,
6693                                         size_t dim4, size_t dim5) {
6694     dims_[0] = dim1;
6695     dims_[1] = dim2;
6696     dims_[2] = dim3;
6697     dims_[3] = dim4;
6698     dims_[4] = dim5;
6699     order_ = 5;
6700     length_ = (dim1 * dim2 * dim3 * dim4 * dim5);
6701     this_matrix_ = some_matrix;
6702 }
6703
6704 // Overloaded 6D constructor
6705 template <typename T>
6706 KOKKOS_INLINE_FUNCTION
6707 ViewFMatrixKokkos<T>::ViewFMatrixKokkos(T* some_matrix, size_t dim1,
6708                                         size_t dim2, size_t dim3,
6709                                         size_t dim4, size_t dim5,
6710                                         size_t dim6) {
6711     dims_[0] = dim1;
6712     dims_[1] = dim2;
6713     dims_[2] = dim3;
6714     dims_[3] = dim4;
6715     dims_[4] = dim5;
6716     dims_[5] = dim6;
```

```
6717     order_ = 6;
6718     length_ = (dim1 * dim2 * dim3 * dim4 * dim5 * dim6);
6719     this_matrix_ = some_matrix;
6720 }
6721
6722 // Overloaded 6D constructor
6723 template <typename T>
6724 KOKKOS_INLINE_FUNCTION
6725 ViewFMatrixKokkos<T>::ViewFMatrixKokkos(T* some_matrix, size_t dim1,
6726                                         size_t dim2, size_t dim3,
6727                                         size_t dim4, size_t dim5,
6728                                         size_t dim6, size_t dim7) {
6729     dims_[0] = dim1;
6730     dims_[1] = dim2;
6731     dims_[2] = dim3;
6732     dims_[3] = dim4;
6733     dims_[4] = dim5;
6734     dims_[5] = dim6;
6735     dims_[6] = dim7;
6736     order_ = 7;
6737     length_ = (dim1 * dim2 * dim3 * dim4 * dim5 * dim6 * dim7);
6738     this_matrix_ = some_matrix;
6739 }
6740
6741
6742 template <typename T>
6743 KOKKOS_INLINE_FUNCTION
6744 T& ViewFMatrixKokkos<T>::operator()(size_t i) const {
6745     assert(order_ == 1 && "Tensor order (rank) does not match constructor in ViewFMatrixKokkos 1D!");
6746     assert(i >= 1 && i <= dims_[0] && "i is out of bounds in ViewFMatrixKokkos 1D!");
6747     return this_matrix_[(i - 1)];
6748 }
6749
6750 template <typename T>
6751 KOKKOS_INLINE_FUNCTION
6752 T& ViewFMatrixKokkos<T>::operator()(size_t i, size_t j) const {
6753     assert(order_ == 2 && "Tensor order (rank) does not match constructor in ViewFMatrixKokkos 2D!");
6754     assert(i >= 1 && i <= dims_[0] && "i is out of bounds in ViewFMatrixKokkos 2D!");
6755     assert(j >= 1 && j <= dims_[1] && "j is out of bounds in ViewFMatrixKokkos 2D!");
6756     return this_matrix_[(i - 1) + ((j - 1) * dims_[0])];
6757 }
6758
6759 template <typename T>
6760 KOKKOS_INLINE_FUNCTION
6761 T& ViewFMatrixKokkos<T>::operator()(size_t i, size_t j, size_t k) const
6762 {
6763     assert(order_ == 3 && "Tensor order (rank) does not match constructor in ViewFMatrixKokkos 3D!");
6764     assert(i >= 1 && i <= dims_[0] && "i is out of bounds in ViewFMatrixKokkos 3D!");
6765     assert(j >= 1 && j <= dims_[1] && "j is out of bounds in ViewFMatrixKokkos 3D!");
6766     assert(k >= 1 && k <= dims_[2] && "k is out of bounds in ViewFMatrixKokkos 3D!");
6767
6768     return this_matrix_[(i - 1) + ((j - 1) * dims_[0])
6769                                 + ((k - 1) * dims_[0] * dims_[1])];
6770 }
6771
6772 template <typename T>
6773 KOKKOS_INLINE_FUNCTION
6774 T& ViewFMatrixKokkos<T>::operator()(size_t i, size_t j, size_t k,
6775                                     size_t l) const {
6776     assert(order_ == 4 && "Tensor order (rank) does not match constructor in ViewFMatrixKokkos 4D!");
6777     assert(i >= 1 && i <= dims_[0] && "i is out of bounds in ViewFMatrixKokkos 4D!");
6778     assert(j >= 1 && j <= dims_[1] && "j is out of bounds in ViewFMatrixKokkos 4D!");
6779     assert(k >= 1 && k <= dims_[2] && "k is out of bounds in ViewFMatrixKokkos 4D!");
6780     assert(l >= 1 && l <= dims_[3] && "l is out of bounds in ViewFMatrixKokkos 4D!");
6781     return this_matrix_[(i - 1) + ((j - 1) * dims_[0])
6782                                 + ((k - 1) * dims_[0] * dims_[1])
6783                                 + ((l - 1) * dims_[0] * dims_[1] * dims_[2])];
6784 }
6785
6786 template <typename T>
6787 KOKKOS_INLINE_FUNCTION
6788 T& ViewFMatrixKokkos<T>::operator()(size_t i, size_t j, size_t k, size_t l,
6789                                     size_t m) const {
6790     assert(order_ == 5 && "Tensor order (rank) does not match constructor in ViewFMatrixKokkos 5D!");
6791     assert(i >= 1 && i <= dims_[0] && "i is out of bounds in ViewFMatrixKokkos 5D!");
6792     assert(j >= 1 && j <= dims_[1] && "j is out of bounds in ViewFMatrixKokkos 5D!");
6793     assert(k >= 1 && k <= dims_[2] && "k is out of bounds in ViewFMatrixKokkos 5D!");
6794     assert(l >= 1 && l <= dims_[3] && "l is out of bounds in ViewFMatrixKokkos 5D!");
6795     assert(m >= 1 && m <= dims_[4] && "m is out of bounds in ViewFMatrixKokkos 5D!");
6796     return this_matrix_[(i - 1) + ((j - 1) * dims_[0])
6797                                 + ((k - 1) * dims_[0] * dims_[1])
6798                                 + ((l - 1) * dims_[0] * dims_[1] * dims_[2])
6799                                 + ((m - 1) * dims_[0] * dims_[1] * dims_[2] * dims_[3])];
6800 }
6801
6802 template <typename T>
6803 KOKKOS_INLINE_FUNCTION
```

```
6804 T& ViewFMatrixKokkos<T>::operator()(size_t i, size_t j, size_t k, size_t l,
6805                                     size_t m, size_t n) const
6806 {
6807     assert(order_ == 6 && "Tensor order (rank) does not match constructor in ViewFMatrixKokkos 6D!");
6808     assert(i >= 1 && i <= dims_[0] && "i is out of bounds in ViewFMatrixKokkos 6D!");
6809     assert(j >= 1 && j <= dims_[1] && "j is out of bounds in ViewFMatrixKokkos 6D!");
6810     assert(k >= 1 && k <= dims_[2] && "k is out of bounds in ViewFMatrixKokkos 6D!");
6811     assert(l >= 1 && l <= dims_[3] && "l is out of bounds in ViewFMatrixKokkos 6D!");
6812     assert(m >= 1 && m <= dims_[4] && "m is out of bounds in ViewFMatrixKokkos 6D!");
6813     assert(n >= 1 && n <= dims_[5] && "n is out of bounds in ViewFMatrixKokkos 6D!");
6814     return this_matrix_[(i - 1) + ((j - 1) * dims_[0])
6815                                 + ((k - 1) * dims_[0] * dims_[1])
6816                                 + ((l - 1) * dims_[0] * dims_[1] * dims_[2])
6817                                 + ((m - 1) * dims_[0] * dims_[1] * dims_[2] * dims_[3])
6818                                 + ((n - 1) * dims_[0] * dims_[1] * dims_[2] * dims_[3] * dims_[4])];
6819 }
6820
6821 template <typename T>
6822 KOKKOS_INLINE_FUNCTION
6823 T& ViewFMatrixKokkos<T>::operator()(size_t i, size_t j, size_t k, size_t l,
6824                                     size_t m, size_t n, size_t o) const
6825 {
6826     assert(order_ == 7 && "Tensor order (rank) does not match constructor in ViewFMatrixKokkos 7D!");
6827     assert(i >= 1 && i <= dims_[0] && "i is out of bounds in ViewFMatrixKokkos 7D!");
6828     assert(j >= 1 && j <= dims_[1] && "j is out of bounds in ViewFMatrixKokkos 7D!");
6829     assert(k >= 1 && k <= dims_[2] && "k is out of bounds in ViewFMatrixKokkos 7D!");
6830     assert(l >= 1 && l <= dims_[3] && "l is out of bounds in ViewFMatrixKokkos 7D!");
6831     assert(m >= 1 && m <= dims_[4] && "m is out of bounds in ViewFMatrixKokkos 7D!");
6832     assert(n >= 1 && n <= dims_[5] && "n is out of bounds in ViewFMatrixKokkos 7D!");
6833     assert(o >= 1 && o <= dims_[6] && "o is out of bounds in ViewFMatrixKokkos 7D!");
6834     return this_matrix_[(i - 1) + ((j - 1) * dims_[0])
6835                                 + ((k - 1) * dims_[0] * dims_[1])
6836                                 + ((l - 1) * dims_[0] * dims_[1] * dims_[2])
6837                                 + ((m - 1) * dims_[0] * dims_[1] * dims_[2] * dims_[3])
6838                                 + ((n - 1) * dims_[0] * dims_[1] * dims_[2] * dims_[3] * dims_[4])
6839                                 + ((o - 1) * dims_[0] * dims_[1] * dims_[2] * dims_[3] * dims_[4] *
       dims_[5])];
6840 }
6841
6842 template <typename T>
6843 KOKKOS_INLINE_FUNCTION
6844 size_t ViewFMatrixKokkos<T>::size() const {
6845     return length_;
6846 }
6847
6848 template <typename T>
6849 KOKKOS_INLINE_FUNCTION
6850 size_t ViewFMatrixKokkos<T>::extent() const {
6851     return length_;
6852 }
6853
6854 template <typename T>
6855 KOKKOS_INLINE_FUNCTION
6856 size_t ViewFMatrixKokkos<T>::dims(size_t i) const {
6857     i--;
6858     assert(i < order_ && "ViewFMatrixKokkos order (rank) does not match constructor, dim[i] does not
      exist!");
6859     assert(i >= 0 && dims_[i]>0 && "Access to ViewFMatrixKokkos dims is out of bounds!");
6860     return dims_[i];
6861 }
6862
6863 template <typename T>
6864 KOKKOS_INLINE_FUNCTION
6865 size_t ViewFMatrixKokkos<T>::order() const {
6866     return order_;
6867 }
6868
6869 template <typename T>
6870 KOKKOS_INLINE_FUNCTION
6871 T* ViewFMatrixKokkos<T>::pointer() const {
6872     return this_matrix_;
6873 }
6874
6875 template <typename T>
6876 KOKKOS_INLINE_FUNCTION
6877 ViewFMatrixKokkos<T>::~ViewFMatrixKokkos() {}
6878
6880 // End of ViewFMatrixKokkos
6882
6883
6885 // DFArrayKokkos:  Dual type for managing data on both CPU and GPU.
6887 template <typename T, typename Layout = DefaultLayout, typename ExecSpace = DefaultExecSpace, typename
       MemoryTraits = void>
6888 class DFArrayKokkos {
6889
6890     // this is manage
6891     using TArray1D = Kokkos::DualView<T*, Layout, ExecSpace, MemoryTraits>;
```

```
6892
6893 private:
6894     size_t dims_[7];
6895     size_t length_;
6896     size_t order_;  // tensor order (rank)
6897     TArray1D this_array_;
6898
6899 public:
6900     DFArrayKokkos();
6901
6902     DFArrayKokkos(size_t dim0, const std::string& tag_string = DEFAULTSTRINGARRAY);
6903
6904     DFArrayKokkos(size_t dim0, size_t dim1, const std::string& tag_string = DEFAULTSTRINGARRAY);
6905
6906     DFArrayKokkos (size_t dim0, size_t dim1, size_t dim2, const std::string& tag_string =
    DEFAULTSTRINGARRAY);
6907
6908     DFArrayKokkos(size_t dim0, size_t dim1, size_t dim2,
6909                   size_t dim3, const std::string& tag_string = DEFAULTSTRINGARRAY);
6910
6911     DFArrayKokkos(size_t dim0, size_t dim1, size_t dim2,
6912                   size_t dim3, size_t dim4, const std::string& tag_string = DEFAULTSTRINGARRAY);
6913
6914     DFArrayKokkos(size_t dim0, size_t dim1, size_t dim2,
6915                   size_t dim3, size_t dim4, size_t dim5, const std::string& tag_string =
    DEFAULTSTRINGARRAY);
6916
6917     DFArrayKokkos(size_t dim0, size_t dim1, size_t dim2,
6918                   size_t dim3, size_t dim4, size_t dim5,
6919                   size_t dim6, const std::string& tag_string = DEFAULTSTRINGARRAY);
6920
6921     KOKKOS_INLINE_FUNCTION
6922     T& operator()(size_t i) const;
6923
6924     KOKKOS_INLINE_FUNCTION
6925     T& operator()(size_t i, size_t j) const;
6926
6927     KOKKOS_INLINE_FUNCTION
6928     T& operator()(size_t i, size_t j, size_t k) const;
6929
6930     KOKKOS_INLINE_FUNCTION
6931     T& operator()(size_t i, size_t j, size_t k, size_t l) const;
6932
6933     KOKKOS_INLINE_FUNCTION
6934     T& operator()(size_t i, size_t j, size_t k, size_t l, size_t m) const;
6935
6936     KOKKOS_INLINE_FUNCTION
6937     T& operator()(size_t i, size_t j, size_t k, size_t l, size_t m,
6938                   size_t n) const;
6939
6940     KOKKOS_INLINE_FUNCTION
6941     T& operator()(size_t i, size_t j, size_t k, size_t l, size_t m,
6942                   size_t n, size_t o) const;
6943
6944     KOKKOS_INLINE_FUNCTION
6945     DFArrayKokkos& operator=(const DFArrayKokkos& temp);
6946
6947     // GPU Method
6948     // Method that returns size
6949     KOKKOS_INLINE_FUNCTION
6950     size_t size() const;
6951
6952     // Host Method
6953     // Method that returns size
6954     KOKKOS_INLINE_FUNCTION
6955     size_t extent() const;
6956
6957     KOKKOS_INLINE_FUNCTION
6958     size_t dims(size_t i) const;
6959
6960     KOKKOS_INLINE_FUNCTION
6961     size_t order() const;
6962
6963     // Method returns the raw device pointer of the Kokkos DualView
6964     KOKKOS_INLINE_FUNCTION
6965     T* device_pointer() const;
6966
6967     // Method returns the raw host pointer of the Kokkos DualView
6968     KOKKOS_INLINE_FUNCTION
6969     T* host_pointer() const;
6970
6971     // Data member to access host view
6972     ViewFArray <T> host;
6973
6974     // Method that update host view
6975     void update_host();
6976
```

```
6977     // Method that update device view
6978     void update_device();
6979 
6980     // Deconstructor
6981     KOKKOS_INLINE_FUNCTION
6982     ~DFArrayKokkos ();
6983 
6984 }; // End of DFArrayKokkos declarations
6985 
6986 // Default constructor
6987 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
6988 DFArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::DFArrayKokkos() {}
6989 
6990 // Overloaded 1D constructor
6991 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
6992 DFArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::DFArrayKokkos(size_t dim0, const std::string&
      tag_string) {
6993 
6994     dims_[0] = dim0;
6995     order_ = 1;
6996     length_ = dim0;
6997     this_array_ = TArray1D(tag_string, length_);
6998     // Create host ViewFArray
6999     host = ViewFArray <T> (this_array_.h_view.data(), dim0);
7000 }
7001 
7002 // Overloaded 2D constructor
7003 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
7004 DFArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::DFArrayKokkos(size_t dim0, size_t dim1, const
      std::string& tag_string) {
7005 
7006     dims_[0] = dim0;
7007     dims_[1] = dim1;
7008     order_ = 2;
7009     length_ = (dim0 * dim1);
7010     this_array_ = TArray1D(tag_string, length_);
7011     // Create host ViewFArray
7012     host = ViewFArray <T> (this_array_.h_view.data(), dim0, dim1);
7013 }
7014 
7015 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
7016 DFArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::DFArrayKokkos(size_t dim0, size_t dim1,
7017                                size_t dim2, const std::string& tag_string) {
7018 
7019     dims_[0] = dim0;
7020     dims_[1] = dim1;
7021     dims_[2] = dim2;
7022     order_ = 3;
7023     length_ = (dim0 * dim1 * dim2);
7024     this_array_ = TArray1D(tag_string, length_);
7025     // Create host ViewFArray
7026     host = ViewFArray <T> (this_array_.h_view.data(), dim0, dim1, dim2);
7027 }
7028 
7029 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
7030 DFArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::DFArrayKokkos(size_t dim0, size_t dim1,
7031                                size_t dim2, size_t dim3, const std::string& tag_string) {
7032 
7033     dims_[0] = dim0;
7034     dims_[1] = dim1;
7035     dims_[2] = dim2;
7036     dims_[3] = dim3;
7037     order_ = 4;
7038     length_ = (dim0 * dim1 * dim2 * dim3);
7039     this_array_ = TArray1D(tag_string, length_);
7040     // Create host ViewFArray
7041     host = ViewFArray <T> (this_array_.h_view.data(), dim0, dim1, dim2, dim3);
7042 }
7043 
7044 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
7045 DFArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::DFArrayKokkos(size_t dim0, size_t dim1,
7046                                size_t dim2, size_t dim3,
7047                                size_t dim4, const std::string& tag_string) {
7048 
7049     dims_[0] = dim0;
7050     dims_[1] = dim1;
7051     dims_[2] = dim2;
7052     dims_[3] = dim3;
7053     dims_[4] = dim4;
7054     order_ = 5;
7055     length_ = (dim0 * dim1 * dim2 * dim3 * dim4);
7056     this_array_ = TArray1D(tag_string, length_);
7057     // Create host ViewFArray
7058     host = ViewFArray <T> (this_array_.h_view.data(), dim0, dim1, dim2, dim3, dim4);
7059 }
7060 
7061 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
```

```
7062 DFArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::DFArrayKokkos(size_t dim0, size_t dim1,
7063                                size_t dim2, size_t dim3,
7064                                size_t dim4, size_t dim5, const std::string& tag_string) {
7065
7066     dims_[0] = dim0;
7067     dims_[1] = dim1;
7068     dims_[2] = dim2;
7069     dims_[3] = dim3;
7070     dims_[4] = dim4;
7071     dims_[5] = dim5;
7072     order_ = 6;
7073     length_ = (dim0 * dim1 * dim2 * dim3 * dim4 * dim5);
7074     this_array_ = TArray1D(tag_string, length_);
7075     // Create host ViewFArray
7076     host = ViewFArray <T> (this_array_.h_view.data(), dim0, dim1, dim2, dim3, dim4, dim5);
7077 }
7078
7079 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
7080 DFArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::DFArrayKokkos(size_t dim0, size_t dim1,
7081                                size_t dim2, size_t dim3,
7082                                size_t dim4, size_t dim5,
7083                                size_t dim6, const std::string& tag_string) {
7084
7085     dims_[0] = dim0;
7086     dims_[1] = dim1;
7087     dims_[2] = dim2;
7088     dims_[3] = dim3;
7089     dims_[4] = dim4;
7090     dims_[5] = dim5;
7091     dims_[6] = dim6;
7092     order_ = 7;
7093     length_ = (dim0 * dim1 * dim2 * dim3 * dim4 * dim5 * dim6);
7094     this_array_ = TArray1D(tag_string, length_);
7095     // Create host ViewFArray
7096     host = ViewFArray <T> (this_array_.h_view.data(), dim0, dim1, dim2, dim3, dim4, dim5, dim6);
7097 }
7098
7099 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
7100 KOKKOS_INLINE_FUNCTION
7101 T& DFArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::operator()(size_t i) const {
7102     assert(order_ == 1 && "Tensor order (rank) does not match constructor in DFArrayKokkos 1D!");
7103     assert(i >= 0 && i < dims_[0] && "i is out of bounds in DFArrayKokkos 1D!");
7104     return this_array_.d_view(i);
7105 }
7106
7107 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
7108 KOKKOS_INLINE_FUNCTION
7109 T& DFArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::operator()(size_t i, size_t j) const {
7110     assert(order_ == 2 && "Tensor order (rank) does not match constructor in DFArrayKokkos 2D!");
7111     assert(i >= 0 && i < dims_[0] && "i is out of bounds in DFArrayKokkos 2D!");
7112     assert(j >= 0 && j < dims_[1] && "j is out of bounds in DFArrayKokkos 2D!");
7113     return this_array_.d_view(i + (j * dims_[0]));
7114 }
7115
7116 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
7117 KOKKOS_INLINE_FUNCTION
7118 T& DFArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::operator()(size_t i, size_t j, size_t k) const {
7119     assert(order_ == 3 && "Tensor order (rank) does not match constructor in DFArrayKokkos 3D!");
7120     assert(i >= 0 && i < dims_[0] && "i is out of bounds in DFArrayKokkos 3D!");
7121     assert(j >= 0 && j < dims_[1] && "j is out of bounds in DFArrayKokkos 3D!");
7122     assert(k >= 0 && k < dims_[2] && "k is out of bounds in DFArrayKokkos 3D!");
7123     return this_array_.d_view(i + (j * dims_[0])
7124                               + (k * dims_[0] * dims_[1]));
7125 }
7126
7127 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
7128 KOKKOS_INLINE_FUNCTION
7129 T& DFArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::operator()(size_t i, size_t j, size_t k, size_t l)
      const {
7130     assert(order_ == 4 && "Tensor order (rank) does not match constructor in DFArrayKokkos 4D!");
7131     assert(i >= 0 && i < dims_[0] && "i is out of bounds in DFArrayKokkos 4D!");
7132     assert(j >= 0 && j < dims_[1] && "j is out of bounds in DFArrayKokkos 4D!");
7133     assert(k >= 0 && k < dims_[2] && "k is out of bounds in DFArrayKokkos 4D!");
7134     assert(l >= 0 && l < dims_[3] && "l is out of bounds in DFArrayKokkos 4D!");
7135     return this_array_.d_view(i + (j * dims_[0])
7136                               + (k * dims_[0] * dims_[1])
7137                               + (l * dims_[0] * dims_[1] * dims_[2]));
7138 }
7139
7140 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
7141 KOKKOS_INLINE_FUNCTION
7142 T& DFArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::operator()(size_t i, size_t j, size_t k, size_t l,
7143                               size_t m) const {
7144     assert(order_ == 5 && "Tensor order (rank) does not match constructor in DFArrayKokkos 5D!");
7145     assert(i >= 0 && i < dims_[0] && "i is out of bounds in DFArrayKokkos 5D!");
7146     assert(j >= 0 && j < dims_[1] && "j is out of bounds in DFArrayKokkos 5D!");
7147     assert(k >= 0 && k < dims_[2] && "k is out of bounds in DFArrayKokkos 5D!");
```

```
7148     assert(l >= 0 && l < dims_[3] && "l is out of bounds in DFArrayKokkos 5D!");
7149     assert(m >= 0 && m < dims_[4] && "m is out of bounds in DFArrayKokkos 5D!");
7150     return this_array_.d_view(i + (j * dims_[0])
7151                               + (k * dims_[0] * dims_[1])
7152                               + (l * dims_[0] * dims_[1] * dims_[2])
7153                               + (m * dims_[0] * dims_[1] * dims_[2] * dims_[3]));
7154 }
7155
7156 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
7157 KOKKOS_INLINE_FUNCTION
7158 T& DFArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::operator()(size_t i, size_t j, size_t k, size_t l,
7159                               size_t m, size_t n) const {
7160     assert(order_ == 6 && "Tensor order (rank) does not match constructor in DFArrayKokkos 6D!");
7161     assert(i >= 0 && i < dims_[0] && "i is out of bounds in DFArrayKokkos 6D!");
7162     assert(j >= 0 && j < dims_[1] && "j is out of bounds in DFArrayKokkos 6D!");
7163     assert(k >= 0 && k < dims_[2] && "k is out of bounds in DFArrayKokkos 6D!");
7164     assert(l >= 0 && l < dims_[3] && "l is out of bounds in DFArrayKokkos 6D!");
7165     assert(m >= 0 && m < dims_[4] && "m is out of bounds in DFArrayKokkos 6D!");
7166     assert(n >= 0 && n < dims_[5] && "n is out of bounds in DFArrayKokkos 6D!");
7167     return this_array_.d_view(i + (j * dims_[0])
7168                               + (k * dims_[0] * dims_[1])
7169                               + (l * dims_[0] * dims_[1] * dims_[2])
7170                               + (m * dims_[0] * dims_[1] * dims_[2] * dims_[3])
7171                               + (n * dims_[0] * dims_[1] * dims_[2] * dims_[3] * dims_[4]));
7172 }
7173
7174 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
7175 KOKKOS_INLINE_FUNCTION
7176 T& DFArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::operator()(size_t i, size_t j, size_t k, size_t l,
7177                               size_t m, size_t n, size_t o) const {
7178     assert(order_ == 7 && "Tensor order (rank) does not match constructor in DFArrayKokkos 7D!");
7179     assert(i >= 0 && i < dims_[0] && "i is out of bounds in DFArrayKokkos 7D!");
7180     assert(j >= 0 && j < dims_[1] && "j is out of bounds in DFArrayKokkos 7D!");
7181     assert(k >= 0 && k < dims_[2] && "k is out of bounds in DFArrayKokkos 7D!");
7182     assert(l >= 0 && l < dims_[3] && "l is out of bounds in DFArrayKokkos 7D!");
7183     assert(m >= 0 && m < dims_[4] && "m is out of bounds in DFArrayKokkos 7D!");
7184     assert(n >= 0 && n < dims_[5] && "n is out of bounds in DFArrayKokkos 7D!");
7185     assert(o >= 0 && o < dims_[6] && "o is out of bounds in DFArrayKokkos 7D!");
7186     return this_array_.d_view(i + (j * dims_[0])
7187                               + (k * dims_[0] * dims_[1])
7188                               + (l * dims_[0] * dims_[1] * dims_[2])
7189                               + (m * dims_[0] * dims_[1] * dims_[2] * dims_[3])
7190                               + (n * dims_[0] * dims_[1] * dims_[2] * dims_[3] * dims_[4])
7191                               + (o * dims_[0] * dims_[1] * dims_[2] * dims_[3] * dims_[4] *
       dims_[5]));
7192 }
7193
7194 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
7195 KOKKOS_INLINE_FUNCTION
7196 DFArrayKokkos<T,Layout,ExecSpace,MemoryTraits>&
      DFArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::operator= (const DFArrayKokkos& temp) {
7197
7198     // Do nothing if the assignment is of the form x = x
7199     if (this != &temp) {
7200         for (int iter = 0; iter < temp.order_; iter++){
7201             dims_[iter] = temp.dims_[iter];
7202         } // end for
7203
7204         order_ = temp.order_;
7205         length_ = temp.length_;
7206         this_array_ = temp.this_array_;
7207     host = temp.host;
7208     }
7209
7210     return *this;
7211 }
7212
7213 // Return size
7214 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
7215 KOKKOS_INLINE_FUNCTION
7216 size_t DFArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::size() const {
7217     return length_;
7218 }
7219
7220 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
7221 KOKKOS_INLINE_FUNCTION
7222 size_t DFArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::extent() const {
7223     return length_;
7224 }
7225
7226 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
7227 KOKKOS_INLINE_FUNCTION
7228 size_t DFArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::dims(size_t i) const {
7229     assert(i < order_ && "DFArrayKokkos order (rank) does not match constructor, dim[i] does not
      exist!");
7230     assert(i >= 0 && dims_[i]>0 && "Access to DFArrayKokkos dims is out of bounds!");
7231     return dims_[i];
```

```
7232 }
7233
7234 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
7235 KOKKOS_INLINE_FUNCTION
7236 size_t DFArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::order() const {
7237     return order_;
7238 }
7239
7240 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
7241 KOKKOS_INLINE_FUNCTION
7242 T* DFArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::device_pointer() const {
7243     return this_array_.d_view.data();
7244 }
7245
7246 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
7247 KOKKOS_INLINE_FUNCTION
7248 T* DFArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::host_pointer() const {
7249     return this_array_.h_view.data();
7250 }
7251
7252 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
7253 void DFArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::update_host() {
7254
7255     this_array_.template modify<typename TArray1D::execution_space>();
7256     this_array_.template sync<typename TArray1D::host_mirror_space>();
7257 }
7258
7259 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
7260 void DFArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::update_device() {
7261
7262     this_array_.template modify<typename TArray1D::host_mirror_space>();
7263     this_array_.template sync<typename TArray1D::execution_space>();
7264 }
7265
7266 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
7267 KOKKOS_INLINE_FUNCTION
7268 DFArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::~DFArrayKokkos() {}
7269 // End DFArrayKokkos
7270
7271
7273 // DViewFArrayKokkos:  The DView means dual view of the data, where data is on both CPU and GPU.
7274 //
7275 // This MATAR type is for accepting a pointer to data on the CPU via the constructor and then it copies
       the data
7276 // data to the GPU where the member functions and overloads access the data on the GPU. The
       corresponding
7277 // FArrayKokkos type creates memory on the GPU; likewise, the viewFArrayKokkos accesses data already on
       the GPU.
7278 // To emphasize, the data must be on the CPU prior to calling the constructor for the DView data type.
7280 template <typename T, typename Layout = DefaultLayout, typename ExecSpace = DefaultExecSpace, typename
       MemoryTraits = void>
7281 class DViewFArrayKokkos {
7282
7283     // this is always unmanaged
7284     using TArray1DHost = Kokkos::View<T*, Layout, HostSpace, MemoryUnmanaged>;
7285     // this is manage
7286     using TArray1D     = Kokkos::View<T*, Layout, ExecSpace, MemoryTraits>;
7287
7288 private:
7289     size_t dims_[7];
7290     size_t length_;
7291     size_t order_;  // tensor order (rank)
7292     TArray1D this_array_;
7293     TArray1DHost this_array_host_;
7294     T * temp_inp_array_;
7295
7296 public:
7297     DViewFArrayKokkos();
7298
7299     DViewFArrayKokkos(T * inp_array, size_t dim0);
7300
7301     DViewFArrayKokkos(T * inp_array, size_t dim0, size_t dim1);
7302
7303     DViewFArrayKokkos(T * inp_array, size_t dim0, size_t dim1, size_t dim2);
7304
7305     DViewFArrayKokkos(T * inp_array, size_t dim0, size_t dim1, size_t dim2,
7306                 size_t dim3);
7307
7308     DViewFArrayKokkos(T * inp_array, size_t dim0, size_t dim1, size_t dim2,
7309                 size_t dim3, size_t dim4);
7310
7311     DViewFArrayKokkos(T * inp_array, size_t dim0, size_t dim1, size_t dim2,
7312                 size_t dim3, size_t dim4, size_t dim5);
7313
7314     DViewFArrayKokkos(T * inp_array, size_t dim0, size_t dim1, size_t dim2,
7315                 size_t dim3, size_t dim4, size_t dim5,
7316                 size_t dim6);
```

```
7317
7318     KOKKOS_INLINE_FUNCTION
7319     T& operator()(size_t i) const;
7320
7321     KOKKOS_INLINE_FUNCTION
7322     T& operator()(size_t i, size_t j) const;
7323
7324     KOKKOS_INLINE_FUNCTION
7325     T& operator()(size_t i, size_t j, size_t k) const;
7326
7327     KOKKOS_INLINE_FUNCTION
7328     T& operator()(size_t i, size_t j, size_t k, size_t l) const;
7329
7330     KOKKOS_INLINE_FUNCTION
7331     T& operator()(size_t i, size_t j, size_t k, size_t l, size_t m) const;
7332
7333     KOKKOS_INLINE_FUNCTION
7334     T& operator()(size_t i, size_t j, size_t k, size_t l, size_t m,
7335                   size_t n) const;
7336
7337     KOKKOS_INLINE_FUNCTION
7338     T& operator()(size_t i, size_t j, size_t k, size_t l, size_t m,
7339                   size_t n, size_t o) const;
7340
7341     KOKKOS_INLINE_FUNCTION
7342     DViewFArrayKokkos& operator=(const DViewFArrayKokkos& temp);
7343
7344     // GPU Method
7345     // Method that returns size
7346     KOKKOS_INLINE_FUNCTION
7347     size_t size() const;
7348
7349     // Host Method
7350     // Method that returns size
7351     KOKKOS_INLINE_FUNCTION
7352     size_t extent() const;
7353
7354     KOKKOS_INLINE_FUNCTION
7355     size_t dims(size_t i) const;
7356
7357     KOKKOS_INLINE_FUNCTION
7358     size_t order() const;
7359
7360     // Method returns the raw device pointer of the Kokkos View
7361     KOKKOS_INLINE_FUNCTION
7362     T* device_pointer() const;
7363
7364     // Method returns the raw host pointer of the Kokkos View
7365     KOKKOS_INLINE_FUNCTION
7366     T* host_pointer() const;
7367
7368     // Data member to access host view
7369     ViewFArray <T> host;
7370
7371     // Method that update host view
7372     void update_host();
7373
7374     // Method that update device view
7375     void update_device();
7376
7377     // Deconstructor
7378     KOKKOS_INLINE_FUNCTION
7379     ~DViewFArrayKokkos ();
7380 }; // End of DViewFArrayKokkos
7381
7382
7383 // Default constructor
7384 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
7385 DViewFArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::DViewFArrayKokkos() {}
7386
7387 // Overloaded 1D constructor
7388 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
7389 DViewFArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::DViewFArrayKokkos(T * inp_array, size_t dim0) {
7390     //using TArray1DHost = Kokkos::View<T*, Layout, HostSpace, MemoryUnmanaged>;
7391     //using TArray1D = Kokkos::View<T*, Layout, ExecSpace>;
7392
7393     dims_[0] = dim0;
7394     order_ = 1;
7395     length_ = dim0;
7396     // Create a 1D host view of the external allocation
7397     this_array_host_ = TArray1DHost(inp_array, length_);
7398     // Assign temp point to inp_array pointer that is passed in
7399     temp_inp_array_ = inp_array;
7400     // Create a device copy of that host view
7401     this_array_ = create_mirror_view_and_copy(ExecSpace(), this_array_host_);
7402     // Create host ViewFArray. Note: inp_array and this_array_host_.data() are the same pointer
7403     host = ViewFArray <T> (inp_array, dim0);
```

```
7404 }
7405
7406 // Overloaded 2D constructor
7407 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
7408 DViewFArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::DViewFArrayKokkos(T * inp_array, size_t dim0,
       size_t dim1) {
7409     //using TArray1DHost = Kokkos::View<T*, Layout, HostSpace, MemoryUnmanaged>;
7410     //using TArray1D = Kokkos::View<T*, Layout, ExecSpace>;
7411     //using TArray1Dtemp = TArray1D::HostMirror;
7412
7413     dims_[0] = dim0;
7414     dims_[1] = dim1;
7415     order_ = 2;
7416     length_ = (dim0 * dim1);
7417     // Create a 1D host view of the external allocation
7418     this_array_host_ = TArray1DHost(inp_array, length_);
7419     // Assign temp point to inp_array pointer that is passed in
7420     temp_inp_array_ = inp_array;
7421     // Create a device copy of that host view
7422     this_array_ = create_mirror_view_and_copy(ExecSpace(), this_array_host_);
7423     // Create host ViewFArray
7424     host = ViewFArray <T> (inp_array, dim0, dim1);
7425 }
7426
7427 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
7428 DViewFArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::DViewFArrayKokkos(T * inp_array, size_t dim0,
       size_t dim1,
7429                               size_t dim2) {
7430     //using TArray1D = Kokkos::View<T*, Layout, ExecSpace>;
7431
7432     dims_[0] = dim0;
7433     dims_[1] = dim1;
7434     dims_[2] = dim2;
7435     order_ = 3;
7436     length_ = (dim0 * dim1 * dim2);
7437     // Create a 1D host view of the external allocation
7438     this_array_host_ = TArray1DHost(inp_array, length_);
7439     // Assign temp point to inp_array pointer that is passed in
7440     temp_inp_array_ = inp_array;
7441     // Create a device copy of that host view
7442     this_array_ = create_mirror_view_and_copy(ExecSpace(), this_array_host_);
7443     // Create host ViewFArray
7444     host = ViewFArray <T> (inp_array, dim0, dim1, dim2);
7445 }
7446
7447 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
7448 DViewFArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::DViewFArrayKokkos(T * inp_array, size_t dim0,
       size_t dim1,
7449                               size_t dim2, size_t dim3) {
7450     //using TArray1D = Kokkos::View<T *,Layout,ExecSpace>;
7451
7452     dims_[0] = dim0;
7453     dims_[1] = dim1;
7454     dims_[2] = dim2;
7455     dims_[3] = dim3;
7456     order_ = 4;
7457     length_ = (dim0 * dim1 * dim2 * dim3);
7458     // Create a 1D host view of the external allocation
7459     this_array_host_ = TArray1DHost(inp_array, length_);
7460     // Assign temp point to inp_array pointer that is passed in
7461     temp_inp_array_ = inp_array;
7462     // Create a device copy of that host view
7463     this_array_ = create_mirror_view_and_copy(ExecSpace(), this_array_host_);
7464     // Create host ViewFArray
7465     host = ViewFArray <T> (inp_array, dim0, dim1, dim2, dim3);
7466 }
7467
7468 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
7469 DViewFArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::DViewFArrayKokkos(T * inp_array, size_t dim0,
       size_t dim1,
7470                               size_t dim2, size_t dim3,
7471                               size_t dim4) {
7472
7473     //using TArray1D = Kokkos::View<T *,Layout,ExecSpace>;
7474
7475     dims_[0] = dim0;
7476     dims_[1] = dim1;
7477     dims_[2] = dim2;
7478     dims_[3] = dim3;
7479     dims_[4] = dim4;
7480     order_ = 5;
7481     length_ = (dim0 * dim1 * dim2 * dim3 * dim4);
7482     // Create a 1D host view of the external allocation
7483     this_array_host_ = TArray1DHost(inp_array, length_);
7484     // Assign temp point to inp_array pointer that is passed in
7485     temp_inp_array_ = inp_array;
7486     // Create a device copy of that host view
```

```
7487     this_array_ = create_mirror_view_and_copy(ExecSpace(), this_array_host_);
7488     // Create host ViewFArray
7489     host = ViewFArray <T> (inp_array, dim0, dim1, dim2, dim3, dim4);
7490 }
7491
7492 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
7493 DViewFArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::DViewFArrayKokkos(T * inp_array, size_t dim0,
       size_t dim1,
7494                                size_t dim2, size_t dim3,
7495                                size_t dim4, size_t dim5) {
7496     //using TArray1D = Kokkos::View<T *,Layout,ExecSpace>;
7497
7498     dims_[0] = dim0;
7499     dims_[1] = dim1;
7500     dims_[2] = dim2;
7501     dims_[3] = dim3;
7502     dims_[4] = dim4;
7503     dims_[5] = dim5;
7504     order_ = 6;
7505     length_ = (dim0 * dim1 * dim2 * dim3 * dim4 * dim5);
7506     // Create a 1D host view of the external allocation
7507     this_array_host_ = TArray1DHost(inp_array, length_);
7508     // Assign temp point to inp_array pointer that is passed in
7509     temp_inp_array_ = inp_array;
7510     // Create a device copy of that host view
7511     this_array_ = create_mirror_view_and_copy(ExecSpace(), this_array_host_);
7512     // Create host ViewFArray
7513     host = ViewFArray <T> (inp_array, dim0, dim1, dim2, dim3, dim4, dim5);
7514 }
7515
7516 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
7517 DViewFArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::DViewFArrayKokkos(T * inp_array, size_t dim0,
       size_t dim1,
7518                                size_t dim2, size_t dim3,
7519                                size_t dim4, size_t dim5,
7520                                size_t dim6) {
7521     //using TArray1D = Kokkos::View<T *,Layout,ExecSpace>;
7522
7523     dims_[0] = dim0;
7524     dims_[1] = dim1;
7525     dims_[2] = dim2;
7526     dims_[3] = dim3;
7527     dims_[4] = dim4;
7528     dims_[5] = dim5;
7529     dims_[6] = dim6;
7530     order_ = 7;
7531     length_ = (dim0 * dim1 * dim2 * dim3 * dim4 * dim5 * dim6);
7532     // Create a 1D host view of the external allocation
7533     this_array_host_ = TArray1DHost(inp_array, length_);
7534     // Assign temp point to inp_array pointer that is passed in
7535     temp_inp_array_ = inp_array;
7536     // Create a device copy of that host view
7537     this_array_ = create_mirror_view_and_copy(ExecSpace(), this_array_host_);
7538     // Create host ViewFArray
7539     host = ViewFArray <T> (inp_array, dim0, dim1, dim2, dim3, dim4, dim5, dim6);
7540 }
7541
7542 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
7543 KOKKOS_INLINE_FUNCTION
7544 T& DViewFArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::operator()(size_t i) const {
7545     assert(order_ == 1 && "Tensor order (rank) does not match constructor in DViewFArrayKokkos 1D!");
7546     assert(i >= 0 && i < dims_[0] && "i is out of bounds in DViewFArrayKokkos 1D!");
7547     return this_array_(i);
7548 }
7549
7550 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
7551 KOKKOS_INLINE_FUNCTION
7552 T& DViewFArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::operator()(size_t i, size_t j) const {
7553     assert(order_ == 2 && "Tensor order (rank) does not match constructor in DViewFArrayKokkos 2D!");
7554     assert(i >= 0 && i < dims_[0] && "i is out of bounds in DViewFArrayKokkos 2D!");
7555     assert(j >= 0 && j < dims_[1] && "j is out of bounds in DViewFArrayKokkos 2D!");
7556     return this_array_(i + (j * dims_[0]));
7557 }
7558
7559 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
7560 KOKKOS_INLINE_FUNCTION
7561 T& DViewFArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::operator()(size_t i, size_t j, size_t k) const {
7562     assert(order_ == 3 && "Tensor order (rank) does not match constructor in DViewFArrayKokkos 3D!");
7563     assert(i >= 0 && i < dims_[0] && "i is out of bounds in DViewFArrayKokkos 3D!");
7564     assert(j >= 0 && j < dims_[1] && "j is out of bounds in DViewFArrayKokkos 3D!");
7565     assert(k >= 0 && k < dims_[2] && "k is out of bounds in DViewFArrayKokkos 3D!");
7566     return this_array_(i + (j * dims_[0])
7567                        + (k * dims_[0] * dims_[1]));
7568 }
7569
7570 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
7571 KOKKOS_INLINE_FUNCTION
```

```
7572 T& DViewFArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::operator()(size_t i, size_t j, size_t k, size_t
       l) const {
7573     assert(order_ == 4 && "Tensor order (rank) does not match constructor in DViewFArrayKokkos 4D!");
7574     assert(i >= 0 && i < dims_[0] && "i is out of bounds in DViewFArrayKokkos 4D!");
7575     assert(j >= 0 && j < dims_[1] && "j is out of bounds in DViewFArrayKokkos 4D!");
7576     assert(k >= 0 && k < dims_[2] && "k is out of bounds in DViewFArrayKokkos 4D!");
7577     assert(l >= 0 && l < dims_[3] && "l is out of bounds in DViewFArrayKokkos 4D!");
7578     return this_array_(i + (j * dims_[0])
7579                         + (k * dims_[0] * dims_[1])
7580                         + (l * dims_[0] * dims_[1] * dims_[2]));
7581 }
7582
7583 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
7584 KOKKOS_INLINE_FUNCTION
7585 T& DViewFArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::operator()(size_t i, size_t j, size_t k, size_t
       l,
7586                                 size_t m) const {
7587     assert(order_ == 5 && "Tensor order (rank) does not match constructor in DViewFArrayKokkos 5D!");
7588     assert(i >= 0 && i < dims_[0] && "i is out of bounds in DViewFArrayKokkos 5D!");
7589     assert(j >= 0 && j < dims_[1] && "j is out of bounds in DViewFArrayKokkos 5D!");
7590     assert(k >= 0 && k < dims_[2] && "k is out of bounds in DViewFArrayKokkos 5D!");
7591     assert(l >= 0 && l < dims_[3] && "l is out of bounds in DViewFArrayKokkos 5D!");
7592     assert(m >= 0 && m < dims_[4] && "m is out of bounds in DViewFArrayKokkos 5D!");
7593     return this_array_(i + (j * dims_[0])
7594                         + (k * dims_[0] * dims_[1])
7595                         + (l * dims_[0] * dims_[1] * dims_[2])
7596                         + (m * dims_[0] * dims_[1] * dims_[2] * dims_[3]));
7597 }
7598
7599 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
7600 KOKKOS_INLINE_FUNCTION
7601 T& DViewFArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::operator()(size_t i, size_t j, size_t k, size_t
       l,
7602                                 size_t m, size_t n) const {
7603     assert(order_ == 6 && "Tensor order (rank) does not match constructor in DViewFArrayKokkos 6D!");
7604     assert(i >= 0 && i < dims_[0] && "i is out of bounds in DViewFArrayKokkos 6D!");
7605     assert(j >= 0 && j < dims_[1] && "j is out of bounds in DViewFArrayKokkos 6D!");
7606     assert(k >= 0 && k < dims_[2] && "k is out of bounds in DViewFArrayKokkos 6D!");
7607     assert(l >= 0 && l < dims_[3] && "l is out of bounds in DViewFArrayKokkos 6D!");
7608     assert(m >= 0 && m < dims_[4] && "m is out of bounds in DViewFArrayKokkos 6D!");
7609     assert(n >= 0 && n < dims_[5] && "n is out of bounds in DViewFArrayKokkos 6D!");
7610     return this_array_(i + (j * dims_[0])
7611                         + (k * dims_[0] * dims_[1])
7612                         + (l * dims_[0] * dims_[1] * dims_[2])
7613                         + (m * dims_[0] * dims_[1] * dims_[2] * dims_[3])
7614                         + (n * dims_[0] * dims_[1] * dims_[2] * dims_[3] * dims_[4]));
7615 }
7616
7617 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
7618 KOKKOS_INLINE_FUNCTION
7619 T& DViewFArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::operator()(size_t i, size_t j, size_t k, size_t
       l,
7620                                 size_t m, size_t n, size_t o) const {
7621     assert(order_ == 7 && "Tensor order (rank) does not match constructor in DViewFArrayKokkos 7D!");
7622     assert(i >= 0 && i < dims_[0] && "i is out of bounds in DViewFArrayKokkos 7D!");
7623     assert(j >= 0 && j < dims_[1] && "j is out of bounds in DViewFArrayKokkos 7D!");
7624     assert(k >= 0 && k < dims_[2] && "k is out of bounds in DViewFArrayKokkos 7D!");
7625     assert(l >= 0 && l < dims_[3] && "l is out of bounds in DViewFArrayKokkos 7D!");
7626     assert(m >= 0 && m < dims_[4] && "m is out of bounds in DViewFArrayKokkos 7D!");
7627     assert(n >= 0 && n < dims_[5] && "n is out of bounds in DViewFArrayKokkos 7D!");
7628     assert(o >= 0 && o < dims_[6] && "o is out of bounds in DViewFArrayKokkos 7D!");
7629     return this_array_(i + (j * dims_[0])
7630                         + (k * dims_[0] * dims_[1])
7631                         + (l * dims_[0] * dims_[1] * dims_[2])
7632                         + (m * dims_[0] * dims_[1] * dims_[2] * dims_[3])
7633                         + (n * dims_[0] * dims_[1] * dims_[2] * dims_[3] * dims_[4])
7634                         + (o * dims_[0] * dims_[1] * dims_[2] * dims_[3] * dims_[4] * dims_[5]));
7635 }
7636
7637 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
7638 KOKKOS_INLINE_FUNCTION
7639 DViewFArrayKokkos<T,Layout,ExecSpace,MemoryTraits>&
      DViewFArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::operator= (const DViewFArrayKokkos& temp) {
7640
7641     // Do nothing if the assignment is of the form x = x
7642     if (this != &temp) {
7643         for (int iter = 0; iter < temp.order_; iter++){
7644             dims_[iter] = temp.dims_[iter];
7645         } // end for
7646
7647         order_ = temp.order_;
7648         length_ = temp.length_;
7649         temp_inp_array_ = temp.temp_inp_array_;
7650         this_array_host_ = temp.this_array_host_;
7651         this_array_ = temp.this_array_;
7652     host = temp.host;
7653     }
```

```
7654
7655     return *this;
7656 }
7657
7658 // Return size
7659 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
7660 KOKKOS_INLINE_FUNCTION
7661 size_t DViewFArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::size() const {
7662     return length_;
7663 }
7664
7665 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
7666 KOKKOS_INLINE_FUNCTION
7667 size_t DViewFArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::extent() const {
7668     return length_;
7669 }
7670
7671 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
7672 KOKKOS_INLINE_FUNCTION
7673 size_t DViewFArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::dims(size_t i) const {
7674     assert(i < order_ && "DViewFArrayKokkos order (rank) does not match constructor, dim[i] does not
      exist!");
7675     assert(i >= 0 && dims_[i]>0 && "Access to DViewFArrayKokkos dims is out of bounds!");
7676     return dims_[i];
7677 }
7678
7679 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
7680 KOKKOS_INLINE_FUNCTION
7681 size_t DViewFArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::order() const {
7682     return order_;
7683 }
7684
7685 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
7686 KOKKOS_INLINE_FUNCTION
7687 T* DViewFArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::device_pointer() const {
7688     return this_array_.data();
7689 }
7690
7691 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
7692 KOKKOS_INLINE_FUNCTION
7693 T* DViewFArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::host_pointer() const {
7694     return this_array_host_.data();
7695 }
7696
7697 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
7698 void DViewFArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::update_host() {
7699     // Deep copy of device view to host view
7700     deep_copy(this_array_host_, this_array_);
7701 }
7702
7703 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
7704 void DViewFArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::update_device() {
7705     // Deep copy of host view to device view
7706     deep_copy(this_array_, this_array_host_);
7707 }
7708
7709 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
7710 KOKKOS_INLINE_FUNCTION
7711 DViewFArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::~DViewFArrayKokkos() {}
7712 // End DViewFArrayKokkos
7713
7714
7716 // DFMatrixKokkos
7718 template <typename T, typename Layout = DefaultLayout, typename ExecSpace = DefaultExecSpace, typename
      MemoryTraits = void>
7719 class DFMatrixKokkos {
7720
7721     // this is manage
7722     using TArray1D = Kokkos::DualView<T*, Layout, ExecSpace, MemoryTraits>;
7723
7724 private:
7725     size_t dims_[7];
7726     size_t length_;
7727     size_t order_;  // tensor order (rank)
7728     TArray1D this_matrix_;
7729
7730 public:
7731     DFMatrixKokkos();
7732
7733     DFMatrixKokkos(size_t dim1, const std::string& tag_string = DEFAULTSTRINGMATRIX);
7734
7735     DFMatrixKokkos(size_t dim1, size_t dim2, const std::string& tag_string = DEFAULTSTRINGMATRIX);
7736
7737     DFMatrixKokkos (size_t dim1, size_t dim2, size_t dim3, const std::string& tag_string =
      DEFAULTSTRINGMATRIX);
7738
7739     DFMatrixKokkos(size_t dim1, size_t dim2, size_t dim3,
```

```
7740                 size_t dim4, const std::string& tag_string = DEFAULTSTRINGMATRIX);
7741
7742     DFMatrixKokkos(size_t dim1, size_t dim2, size_t dim3,
7743                 size_t dim4, size_t dim5, const std::string& tag_string = DEFAULTSTRINGMATRIX);
7744
7745     DFMatrixKokkos(size_t dim1, size_t dim2, size_t dim3,
7746                 size_t dim4, size_t dim5, size_t dim6, const std::string& tag_string =
      DEFAULTSTRINGMATRIX);
7747
7748     DFMatrixKokkos(size_t dim1, size_t dim2, size_t dim3,
7749                 size_t dim4, size_t dim5, size_t dim6,
7750                 size_t dim7, const std::string& tag_string = DEFAULTSTRINGMATRIX);
7751
7752     KOKKOS_INLINE_FUNCTION
7753     T& operator()(size_t i) const;
7754
7755     KOKKOS_INLINE_FUNCTION
7756     T& operator()(size_t i, size_t j) const;
7757
7758     KOKKOS_INLINE_FUNCTION
7759     T& operator()(size_t i, size_t j, size_t k) const;
7760
7761     KOKKOS_INLINE_FUNCTION
7762     T& operator()(size_t i, size_t j, size_t k, size_t l) const;
7763
7764     KOKKOS_INLINE_FUNCTION
7765     T& operator()(size_t i, size_t j, size_t k, size_t l, size_t m) const;
7766
7767     KOKKOS_INLINE_FUNCTION
7768     T& operator()(size_t i, size_t j, size_t k, size_t l, size_t m,
7769                   size_t n) const;
7770
7771     KOKKOS_INLINE_FUNCTION
7772     T& operator()(size_t i, size_t j, size_t k, size_t l, size_t m,
7773                   size_t n, size_t o) const;
7774
7775     KOKKOS_INLINE_FUNCTION
7776     DFMatrixKokkos& operator=(const DFMatrixKokkos& temp);
7777
7778     // GPU Method
7779     // Method that returns size
7780     KOKKOS_INLINE_FUNCTION
7781     size_t size() const;
7782
7783     // Host Method
7784     // Method that returns size
7785     KOKKOS_INLINE_FUNCTION
7786     size_t extent() const;
7787
7788     KOKKOS_INLINE_FUNCTION
7789     size_t dims(size_t i) const;
7790
7791     KOKKOS_INLINE_FUNCTION
7792     size_t order() const;
7793
7794     // Method returns the raw device pointer of the Kokkos DualView
7795     KOKKOS_INLINE_FUNCTION
7796     T* device_pointer() const;
7797
7798     // Method returns the raw host pointer of the Kokkos DualView
7799     KOKKOS_INLINE_FUNCTION
7800     T* host_pointer() const;
7801
7802     // Data member to access host view
7803     ViewFMatrix <T> host;
7804
7805     // Method that update host view
7806     void update_host();
7807
7808     // Method that update device view
7809     void update_device();
7810
7811     // Deconstructor
7812     KOKKOS_INLINE_FUNCTION
7813     ~DFMatrixKokkos ();
7814 }; // End of DFMatrixKokkos declarations
7815
7816 // Default constructor
7817 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
7818 DFMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::DFMatrixKokkos() {}
7819
7820 // Overloaded 1D constructor
7821 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
7822 DFMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::DFMatrixKokkos(size_t dim1, const std::string&
      tag_string) {
7823
7824     dims_[0] = dim1;
```

```
7825     order_ = 1;
7826     length_ = dim1;
7827     this_matrix_ = TArray1D(tag_string, length_);
7828     // Create host ViewFMatrix
7829     host = ViewFMatrix <T> (this_matrix_.h_view.data(), dim1);
7830 }
7831
7832 // Overloaded 2D constructor
7833 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
7834 DFMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::DFMatrixKokkos(size_t dim1, size_t dim2, const
      std::string& tag_string) {
7835
7836     dims_[0] = dim1;
7837     dims_[1] = dim2;
7838     order_ = 2;
7839     length_ = (dim1 * dim2);
7840     this_matrix_ = TArray1D(tag_string, length_);
7841     // Create host ViewFMatrix
7842     host = ViewFMatrix <T> (this_matrix_.h_view.data(), dim1, dim2);
7843 }
7844
7845 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
7846 DFMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::DFMatrixKokkos(size_t dim1, size_t dim2,
7847                                 size_t dim3, const std::string& tag_string) {
7848
7849     dims_[0] = dim1;
7850     dims_[1] = dim2;
7851     dims_[2] = dim3;
7852     order_ = 3;
7853     length_ = (dim1 * dim2 * dim3);
7854     this_matrix_ = TArray1D(tag_string, length_);
7855     // Create host ViewFMatrix
7856     host = ViewFMatrix <T> (this_matrix_.h_view.data(), dim1, dim2, dim3);
7857 }
7858
7859 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
7860 DFMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::DFMatrixKokkos(size_t dim1, size_t dim2,
7861                                 size_t dim3, size_t dim4, const std::string& tag_string) {
7862
7863     dims_[0] = dim1;
7864     dims_[1] = dim2;
7865     dims_[2] = dim3;
7866     dims_[3] = dim4;
7867     order_ = 4;
7868     length_ = (dim1 * dim2 * dim3 * dim4);
7869     this_matrix_ = TArray1D(tag_string, length_);
7870     // Create host ViewFMatrix
7871     host = ViewFMatrix <T> (this_matrix_.h_view.data(), dim1, dim2, dim3, dim4);
7872 }
7873
7874 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
7875 DFMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::DFMatrixKokkos(size_t dim1, size_t dim2,
7876                                 size_t dim3, size_t dim4,
7877                                 size_t dim5, const std::string& tag_string) {
7878
7879     dims_[0] = dim1;
7880     dims_[1] = dim2;
7881     dims_[2] = dim3;
7882     dims_[3] = dim4;
7883     dims_[4] = dim5;
7884     order_ = 5;
7885     length_ = (dim1 * dim2 * dim3 * dim4 * dim5);
7886     this_matrix_ = TArray1D(tag_string, length_);
7887     // Create host ViewFMatrix
7888     host = ViewFMatrix <T> (this_matrix_.h_view.data(), dim1, dim2, dim3, dim4, dim5);
7889 }
7890
7891 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
7892 DFMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::DFMatrixKokkos(size_t dim1, size_t dim2,
7893                                 size_t dim3, size_t dim4,
7894                                 size_t dim5, size_t dim6, const std::string& tag_string) {
7895
7896     dims_[0] = dim1;
7897     dims_[1] = dim2;
7898     dims_[2] = dim3;
7899     dims_[3] = dim4;
7900     dims_[4] = dim5;
7901     dims_[5] = dim6;
7902     order_ = 6;
7903     length_ = (dim1 * dim2 * dim3 * dim4 * dim5 * dim6);
7904     this_matrix_ = TArray1D(tag_string, length_);
7905     // Create host ViewFMatrix
7906     host = ViewFMatrix <T> (this_matrix_.h_view.data(), dim1, dim2, dim3, dim4, dim5, dim6);
7907 }
7908
7909 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
7910 DFMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::DFMatrixKokkos(size_t dim1, size_t dim2,
```

```
7911                                size_t dim3, size_t dim4,
7912                                size_t dim5, size_t dim6,
7913                                size_t dim7, const std::string& tag_string) {
7914
7915     dims_[0] = dim1;
7916     dims_[1] = dim2;
7917     dims_[2] = dim3;
7918     dims_[3] = dim4;
7919     dims_[4] = dim5;
7920     dims_[5] = dim6;
7921     dims_[6] = dim7;
7922     order_ = 7;
7923     length_ = (dim1 * dim2 * dim3 * dim4 * dim5 * dim6 * dim7);
7924     this_matrix_ = TArray1D(tag_string, length_);
7925     // Create host ViewFMatrix
7926     host = ViewFMatrix <T> (this_matrix_.h_view.data(), dim1, dim2, dim3, dim4, dim5, dim6, dim7);
7927 }
7928
7929 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
7930 KOKKOS_INLINE_FUNCTION
7931 T& DFMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::operator()(size_t i) const {
7932     assert(order_ == 1 && "Tensor order (rank) does not match constructor in DFMatrixKokkos 1D!");
7933     assert(i >= 1 && i <= dims_[0] && "i is out of bounds in DFMatrixKokkos 1D!");
7934     return this_matrix_.d_view((i - 1));
7935 }
7936
7937 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
7938 KOKKOS_INLINE_FUNCTION
7939 T& DFMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::operator()(size_t i, size_t j) const {
7940     assert(order_ == 2 && "Tensor order (rank) does not match constructor in DFMatrixKokkos 2D!");
7941     assert(i >= 1 && i <= dims_[0] && "i is out of bounds in DFMatrixKokkos 2D!");
7942     assert(j >= 1 && j <= dims_[1] && "j is out of bounds in DFMatrixKokkos 2D!");
7943     return this_matrix_.d_view((i - 1) + ((j - 1) * dims_[0]));
7944 }
7945
7946 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
7947 KOKKOS_INLINE_FUNCTION
7948 T& DFMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::operator()(size_t i, size_t j, size_t k) const {
7949     assert(order_ == 3 && "Tensor order (rank) does not match constructor in DFMatrixKokkos 3D!");
7950     assert(i >= 1 && i <= dims_[0] && "i is out of bounds in DFMatrixKokkos 3D!");
7951     assert(j >= 1 && j <= dims_[1] && "j is out of bounds in DFMatrixKokkos 3D!");
7952     assert(k >= 1 && k <= dims_[2] && "k is out of bounds in DFMatrixKokkos 3D!");
7953     return this_matrix_.d_view((i - 1) + ((j - 1) * dims_[0])
7954                                        + ((k - 1) * dims_[0] * dims_[1]));
7955 }
7956
7957 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
7958 KOKKOS_INLINE_FUNCTION
7959 T& DFMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::operator()(size_t i, size_t j, size_t k, size_t l)
      const {
7960     assert(order_ == 4 && "Tensor order (rank) does not match constructor in DFMatrixKokkos 4D!");
7961     assert(i >= 1 && i <= dims_[0] && "i is out of bounds in DFMatrixKokkos 4D!");
7962     assert(j >= 1 && j <= dims_[1] && "j is out of bounds in DFMatrixKokkos 4D!");
7963     assert(k >= 1 && k <= dims_[2] && "k is out of bounds in DFMatrixKokkos 4D!");
7964     assert(l >= 1 && l <= dims_[3] && "l is out of bounds in DFMatrixKokkos 4D!");
7965     return this_matrix_.d_view((i - 1) + ((j - 1) * dims_[0])
7966                                        + ((k - 1) * dims_[0] * dims_[1])
7967                                        + ((l - 1) * dims_[0] * dims_[1] * dims_[2]));
7968 }
7969
7970 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
7971 KOKKOS_INLINE_FUNCTION
7972 T& DFMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::operator()(size_t i, size_t j, size_t k, size_t l,
7973                                size_t m) const {
7974     assert(order_ == 5 && "Tensor order (rank) does not match constructor in DFMatrixKokkos 5D!");
7975     assert(i >= 1 && i <= dims_[0] && "i is out of bounds in DFMatrixKokkos 5D!");
7976     assert(j >= 1 && j <= dims_[1] && "j is out of bounds in DFMatrixKokkos 5D!");
7977     assert(k >= 1 && k <= dims_[2] && "k is out of bounds in DFMatrixKokkos 5D!");
7978     assert(l >= 1 && l <= dims_[3] && "l is out of bounds in DFMatrixKokkos 5D!");
7979     assert(m >= 1 && m <= dims_[4] && "m is out of bounds in DFMatrixKokkos 5D!");
7980     return this_matrix_.d_view((i - 1) + ((j - 1) * dims_[0])
7981                                        + ((k - 1) * dims_[0] * dims_[1])
7982                                        + ((l - 1) * dims_[0] * dims_[1] * dims_[2])
7983                                        + ((m - 1) * dims_[0] * dims_[1] * dims_[2] * dims_[3]));
7984 }
7985
7986 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
7987 KOKKOS_INLINE_FUNCTION
7988 T& DFMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::operator()(size_t i, size_t j, size_t k, size_t l,
7989                                size_t m, size_t n) const {
7990     assert(order_ == 6 && "Tensor order (rank) does not match constructor in DFMatrixKokkos 6D!");
7991     assert(i >= 1 && i <= dims_[0] && "i is out of bounds in DFMatrixKokkos 6D!");
7992     assert(j >= 1 && j <= dims_[1] && "j is out of bounds in DFMatrixKokkos 6D!");
7993     assert(k >= 1 && k <= dims_[2] && "k is out of bounds in DFMatrixKokkos 6D!");
7994     assert(l >= 1 && l <= dims_[3] && "l is out of bounds in DFMatrixKokkos 6D!");
7995     assert(m >= 1 && m <= dims_[4] && "m is out of bounds in DFMatrixKokkos 6D!");
7996     assert(n >= 1 && n <= dims_[5] && "n is out of bounds in DFMatrixKokkos 6D!");
```

```
7997     return this_matrix_.d_view((i - 1) + ((j - 1) * dims_[0])
7998                                        + ((k - 1) * dims_[0] * dims_[1])
7999                                        + ((l - 1) * dims_[0] * dims_[1] * dims_[2])
8000                                        + ((m - 1) * dims_[0] * dims_[1] * dims_[2] * dims_[3])
8001                                        + ((n - 1) * dims_[0] * dims_[1] * dims_[2] * dims_[3] *
       dims_[4]));
8002 }
8003
8004 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
8005 KOKKOS_INLINE_FUNCTION
8006 T& DFMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::operator()(size_t i, size_t j, size_t k, size_t l,
8007                                 size_t m, size_t n, size_t o) const {
8008     assert(order_ == 7 && "Tensor order (rank) does not match constructor in DFMatrixKokkos 7D!");
8009     assert(i >= 1 && i <= dims_[0] && "i is out of bounds in DFMatrixKokkos 7D!");
8010     assert(j >= 1 && j <= dims_[1] && "j is out of bounds in DFMatrixKokkos 7D!");
8011     assert(k >= 1 && k <= dims_[2] && "k is out of bounds in DFMatrixKokkos 7D!");
8012     assert(l >= 1 && l <= dims_[3] && "l is out of bounds in DFMatrixKokkos 7D!");
8013     assert(m >= 1 && m <= dims_[4] && "m is out of bounds in DFMatrixKokkos 7D!");
8014     assert(n >= 1 && n <= dims_[5] && "n is out of bounds in DFMatrixKokkos 7D!");
8015     assert(o >= 1 && o <= dims_[6] && "o is out of bounds in DFMatrixKokkos 7D!");
8016     return this_matrix_.d_view((i - 1) + ((j - 1) * dims_[0])
8017                                        + ((k - 1) * dims_[0] * dims_[1])
8018                                        + ((l - 1) * dims_[0] * dims_[1] * dims_[2])
8019                                        + ((m - 1) * dims_[0] * dims_[1] * dims_[2] * dims_[3])
8020                                        + ((n - 1) * dims_[0] * dims_[1] * dims_[2] * dims_[3] *
       dims_[4])
8021                                        + ((o - 1) * dims_[0] * dims_[1] * dims_[2] * dims_[3] *
       dims_[4] * dims_[5]));
8022 }
8023
8024 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
8025 KOKKOS_INLINE_FUNCTION
8026 DFMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>&
      DFMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::operator= (const DFMatrixKokkos& temp) {
8027
8028     // Do nothing if the assignment is of the form x = x
8029     if (this != &temp) {
8030         for (int iter = 0; iter < temp.order_; iter++){
8031             dims_[iter] = temp.dims_[iter];
8032         } // end for
8033
8034         order_ = temp.order_;
8035         length_ = temp.length_;
8036         this_matrix_ = temp.this_matrix_;
8037     host = temp.host;
8038     }
8039
8040     return *this;
8041 }
8042
8043 // Return size
8044 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
8045 KOKKOS_INLINE_FUNCTION
8046 size_t DFMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::size() const {
8047     return length_;
8048 }
8049
8050 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
8051 KOKKOS_INLINE_FUNCTION
8052 size_t DFMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::extent() const {
8053     return length_;
8054 }
8055
8056 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
8057 KOKKOS_INLINE_FUNCTION
8058 size_t DFMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::dims(size_t i) const {
8059     i--;
8060     assert(i < order_ && "DFMatrixKokkos order (rank) does not match constructor, dim[i] does not
       exist!");
8061     assert(i >= 0 && dims_[i]>0 && "Access to DFMatrixKokkos dims is out of bounds!");
8062     return dims_[i];
8063 }
8064
8065 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
8066 KOKKOS_INLINE_FUNCTION
8067 size_t DFMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::order() const {
8068     return order_;
8069 }
8070
8071 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
8072 KOKKOS_INLINE_FUNCTION
8073 T* DFMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::device_pointer() const {
8074     return this_matrix_.d_view.data();
8075 }
8076
8077 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
8078 KOKKOS_INLINE_FUNCTION
```

```
8079 T* DFMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::host_pointer() const {
8080     return this_matrix_.h_view.data();
8081 }
8082
8083 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
8084 void DFMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::update_host() {
8085
8086     this_matrix_.template modify<typename TArray1D::execution_space>();
8087     this_matrix_.template sync<typename TArray1D::host_mirror_space>();
8088 }
8089
8090 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
8091 void DFMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::update_device() {
8092
8093     this_matrix_.template modify<typename TArray1D::host_mirror_space>();
8094     this_matrix_.template sync<typename TArray1D::execution_space>();
8095 }
8096
8097 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
8098 KOKKOS_INLINE_FUNCTION
8099 DFMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::~DFMatrixKokkos() {}
8100 // End DFMatrixKokkos
8101
8102
8104 // DViewFMatrixKokkos
8106 template <typename T, typename Layout = DefaultLayout, typename ExecSpace = DefaultExecSpace, typename
       MemoryTraits = void>
8107 class DViewFMatrixKokkos {
8108
8109     // this is always unmanaged
8110     using TArray1DHost = Kokkos::View<T*, Layout, HostSpace, MemoryUnmanaged>;
8111     // this is manage
8112     using TArray1D     = Kokkos::View<T*, Layout, ExecSpace, MemoryTraits>;
8113
8114 private:
8115     size_t dims_[7];
8116     size_t length_;
8117     size_t order_;  // tensor order (rank)
8118     TArray1D this_matrix_;
8119     TArray1DHost this_matrix_host_;
8120     T * temp_inp_matrix_;
8121
8122 public:
8123     DViewFMatrixKokkos();
8124
8125     DViewFMatrixKokkos(T * inp_matrix, size_t dim1);
8126
8127     DViewFMatrixKokkos(T * inp_matrix, size_t dim1, size_t dim2);
8128
8129     DViewFMatrixKokkos(T * inp_matrix, size_t dim1, size_t dim2, size_t dim3);
8130
8131     DViewFMatrixKokkos(T * inp_matrix, size_t dim1, size_t dim2, size_t dim3,
8132                 size_t dim4);
8133
8134     DViewFMatrixKokkos(T * inp_matrix, size_t dim1, size_t dim2, size_t dim3,
8135                 size_t dim4, size_t dim5);
8136
8137     DViewFMatrixKokkos(T * inp_matrix, size_t dim1, size_t dim2, size_t dim3,
8138                 size_t dim4, size_t dim5, size_t dim6);
8139
8140     DViewFMatrixKokkos(T * inp_matrix, size_t dim1, size_t dim2, size_t dim3,
8141                 size_t dim4, size_t dim5, size_t dim6,
8142                 size_t dim7);
8143
8144     KOKKOS_INLINE_FUNCTION
8145     T& operator()(size_t i) const;
8146
8147     KOKKOS_INLINE_FUNCTION
8148     T& operator()(size_t i, size_t j) const;
8149
8150     KOKKOS_INLINE_FUNCTION
8151     T& operator()(size_t i, size_t j, size_t k) const;
8152
8153     KOKKOS_INLINE_FUNCTION
8154     T& operator()(size_t i, size_t j, size_t k, size_t l) const;
8155
8156     KOKKOS_INLINE_FUNCTION
8157     T& operator()(size_t i, size_t j, size_t k, size_t l, size_t m) const;
8158
8159     KOKKOS_INLINE_FUNCTION
8160     T& operator()(size_t i, size_t j, size_t k, size_t l, size_t m,
8161                   size_t n) const;
8162
8163     KOKKOS_INLINE_FUNCTION
8164     T& operator()(size_t i, size_t j, size_t k, size_t l, size_t m,
8165                   size_t n, size_t o) const;
8166
```

```
8167     KOKKOS_INLINE_FUNCTION
8168     DViewFMatrixKokkos& operator=(const DViewFMatrixKokkos& temp);
8169
8170     // GPU Method
8171     // Method that returns size
8172     KOKKOS_INLINE_FUNCTION
8173     size_t size() const;
8174
8175     // Host Method
8176     // Method that returns size
8177     KOKKOS_INLINE_FUNCTION
8178     size_t extent() const;
8179
8180     KOKKOS_INLINE_FUNCTION
8181     size_t dims(size_t i) const;
8182
8183     KOKKOS_INLINE_FUNCTION
8184     size_t order() const;
8185
8186     // Method returns the raw device pointer of the Kokkos View
8187     KOKKOS_INLINE_FUNCTION
8188     T* device_pointer() const;
8189
8190     // Method returns the raw host pointer of the Kokkos View
8191     KOKKOS_INLINE_FUNCTION
8192     T* host_pointer() const;
8193
8194     // Data member to access host view
8195     ViewFMatrix <T> host;
8196
8197     // Method that update host view
8198     void update_host();
8199
8200     // Method that update device view
8201     void update_device();
8202
8203     // Deconstructor
8204     KOKKOS_INLINE_FUNCTION
8205     ~DViewFMatrixKokkos ();
8206 }; // End of DViewFMatrixKokkos
8207
8208
8209 // Default constructor
8210 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
8211 DViewFMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::DViewFMatrixKokkos() {}
8212
8213 // Overloaded 1D constructor
8214 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
8215 DViewFMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::DViewFMatrixKokkos(T * inp_matrix, size_t dim1) {
8216
8217     dims_[0] = dim1;
8218     order_ = 1;
8219     length_ = dim1;
8220     // Create a 1D host view of the external allocation
8221     this_matrix_host_ = TArray1DHost(inp_matrix, length_);
8222     // Assign temp point to inp_matrix pointer that is passed in
8223     temp_inp_matrix_ = inp_matrix;
8224     // Create a device copy of that host view
8225     this_matrix_ = create_mirror_view_and_copy(ExecSpace(), this_matrix_host_);
8226     // Create host ViewFMatrix. Note: inp_matrix and this_matrix_host_.data() are the same pointer
8227     host = ViewFMatrix <T> (inp_matrix, dim1);
8228 }
8229
8230 // Overloaded 2D constructor
8231 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
8232 DViewFMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::DViewFMatrixKokkos(T * inp_matrix, size_t dim1,
       size_t dim2) {
8233
8234     dims_[0] = dim1;
8235     dims_[1] = dim2;
8236     order_ = 2;
8237     length_ = (dim1 * dim2);
8238     // Create a 1D host view of the external allocation
8239     this_matrix_host_ = TArray1DHost(inp_matrix, length_);
8240     // Assign temp point to inp_matrix pointer that is passed in
8241     temp_inp_matrix_ = inp_matrix;
8242     // Create a device copy of that host view
8243     this_matrix_ = create_mirror_view_and_copy(ExecSpace(), this_matrix_host_);
8244     // Create host ViewFMatrix
8245     host = ViewFMatrix <T> (inp_matrix, dim1, dim2);
8246 }
8247
8248 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
8249 DViewFMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::DViewFMatrixKokkos(T * inp_matrix, size_t dim1,
       size_t dim2,
8250                                 size_t dim3) {
8251
```

```
8252     dims_[0] = dim1;
8253     dims_[1] = dim2;
8254     dims_[2] = dim3;
8255     order_ = 3;
8256     length_ = (dim1 * dim2 * dim3);
8257     // Create a 1D host view of the external allocation
8258     this_matrix_host_ = TArray1DHost(inp_matrix, length_);
8259     // Assign temp point to inp_matrix pointer that is passed in
8260     temp_inp_matrix_ = inp_matrix;
8261     // Create a device copy of that host view
8262     this_matrix_ = create_mirror_view_and_copy(ExecSpace(), this_matrix_host_);
8263     // Create host ViewFMatrix
8264     host = ViewFMatrix <T> (inp_matrix, dim1, dim2, dim3);
8265 }
8266
8267 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
8268 DViewFMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::DViewFMatrixKokkos(T * inp_matrix, size_t dim1,
       size_t dim2,
8269                                size_t dim3, size_t dim4) {
8270
8271     dims_[0] = dim1;
8272     dims_[1] = dim2;
8273     dims_[2] = dim3;
8274     dims_[3] = dim4;
8275     order_ = 4;
8276     length_ = (dim1 * dim2 * dim3 * dim4);
8277     // Create a 1D host view of the external allocation
8278     this_matrix_host_ = TArray1DHost(inp_matrix, length_);
8279     // Assign temp point to inp_matrix pointer that is passed in
8280     temp_inp_matrix_ = inp_matrix;
8281     // Create a device copy of that host view
8282     this_matrix_ = create_mirror_view_and_copy(ExecSpace(), this_matrix_host_);
8283     // Create host ViewFMatrix
8284     host = ViewFMatrix <T> (inp_matrix, dim1, dim2, dim3, dim4);
8285 }
8286
8287 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
8288 DViewFMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::DViewFMatrixKokkos(T * inp_matrix, size_t dim1,
       size_t dim2,
8289                                size_t dim3, size_t dim4,
8290                                size_t dim5) {
8291
8292     dims_[0] = dim1;
8293     dims_[1] = dim2;
8294     dims_[2] = dim3;
8295     dims_[3] = dim4;
8296     dims_[4] = dim5;
8297     order_ = 5;
8298     length_ = (dim1 * dim2 * dim3 * dim4 * dim5);
8299     // Create a 1D host view of the external allocation
8300     this_matrix_host_ = TArray1DHost(inp_matrix, length_);
8301     // Assign temp point to inp_matrix pointer that is passed in
8302     temp_inp_matrix_ = inp_matrix;
8303     // Create a device copy of that host view
8304     this_matrix_ = create_mirror_view_and_copy(ExecSpace(), this_matrix_host_);
8305     // Create host ViewFMatrix
8306     host = ViewFMatrix <T> (inp_matrix, dim1, dim2, dim3, dim4, dim5);
8307 }
8308
8309 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
8310 DViewFMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::DViewFMatrixKokkos(T * inp_matrix, size_t dim1,
       size_t dim2,
8311                                size_t dim3, size_t dim4,
8312                                size_t dim5, size_t dim6) {
8313
8314     dims_[0] = dim1;
8315     dims_[1] = dim2;
8316     dims_[2] = dim3;
8317     dims_[3] = dim4;
8318     dims_[4] = dim5;
8319     dims_[5] = dim6;
8320     order_ = 6;
8321     length_ = (dim1 * dim2 * dim3 * dim4 * dim5 * dim6);
8322     // Create a 1D host view of the external allocation
8323     this_matrix_host_ = TArray1DHost(inp_matrix, length_);
8324     // Assign temp point to inp_matrix pointer that is passed in
8325     temp_inp_matrix_ = inp_matrix;
8326     // Create a device copy of that host view
8327     this_matrix_ = create_mirror_view_and_copy(ExecSpace(), this_matrix_host_);
8328     // Create host ViewFMatrix
8329     host = ViewFMatrix <T> (inp_matrix, dim1, dim2, dim3, dim4, dim5, dim6);
8330 }
8331
8332 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
8333 DViewFMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::DViewFMatrixKokkos(T * inp_matrix, size_t dim1,
       size_t dim2,
8334                                size_t dim3, size_t dim4,
```

```
8335                                size_t dim5, size_t dim6,
8336                                size_t dim7) {
8337
8338     dims_[0] = dim1;
8339     dims_[1] = dim2;
8340     dims_[2] = dim3;
8341     dims_[3] = dim4;
8342     dims_[4] = dim5;
8343     dims_[5] = dim6;
8344     dims_[6] = dim7;
8345     order_ = 7;
8346     length_ = (dim1 * dim2 * dim3 * dim4 * dim5 * dim6 * dim7);
8347     // Create a 1D host view of the external allocation
8348     this_matrix_host_ = TArray1DHost(inp_matrix, length_);
8349     // Assign temp point to inp_matrix pointer that is passed in
8350     temp_inp_matrix_ = inp_matrix;
8351     // Create a device copy of that host view
8352     this_matrix_ = create_mirror_view_and_copy(ExecSpace(), this_matrix_host_);
8353     // Create host ViewFMatrix
8354     host = ViewFMatrix <T> (inp_matrix, dim1, dim2, dim3, dim4, dim5, dim6, dim7);
8355 }
8356
8357 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
8358 KOKKOS_INLINE_FUNCTION
8359 T& DViewFMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::operator()(size_t i) const {
8360     assert(order_ == 1 && "Tensor order (rank) does not match constructor in DViewFMatrixKokkos 1D!");
8361     assert(i >= 1 && i <= dims_[0] && "i is out of bounds in DViewFMatrixKokkos 1D!");
8362     return this_matrix_((i - 1));
8363 }
8364
8365 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
8366 KOKKOS_INLINE_FUNCTION
8367 T& DViewFMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::operator()(size_t i, size_t j) const {
8368     assert(order_ == 2 && "Tensor order (rank) does not match constructor in DViewFMatrixKokkos 2D!");
8369     assert(i >= 1 && i <= dims_[0] && "i is out of bounds in DViewFMatrixKokkos 2D!");
8370     assert(j >= 1 && j <= dims_[1] && "j is out of bounds in DViewFMatrixKokkos 2D!");
8371     return this_matrix_((i - 1) + ((j - 1) * dims_[0]));
8372 }
8373
8374 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
8375 KOKKOS_INLINE_FUNCTION
8376 T& DViewFMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::operator()(size_t i, size_t j, size_t k) const
      {
8377     assert(order_ == 3 && "Tensor order (rank) does not match constructor in DViewFMatrixKokkos 3D!");
8378     assert(i >= 1 && i <= dims_[0] && "i is out of bounds in DViewFMatrixKokkos 3D!");
8379     assert(j >= 1 && j <= dims_[1] && "j is out of bounds in DViewFMatrixKokkos 3D!");
8380     assert(k >= 1 && k <= dims_[2] && "k is out of bounds in DViewFMatrixKokkos 3D!");
8381     return this_matrix_((i - 1) + ((j - 1) * dims_[0])
8382                                 + ((k - 1) * dims_[0] * dims_[1]));
8383 }
8384
8385 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
8386 KOKKOS_INLINE_FUNCTION
8387 T& DViewFMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::operator()(size_t i, size_t j, size_t k, size_t
      l) const {
8388     assert(order_ == 4 && "Tensor order (rank) does not match constructor in DViewFMatrixKokkos 4D!");
8389     assert(i >= 1 && i <= dims_[0] && "i is out of bounds in DViewFMatrixKokkos 4D!");
8390     assert(j >= 1 && j <= dims_[1] && "j is out of bounds in DViewFMatrixKokkos 4D!");
8391     assert(k >= 1 && k <= dims_[2] && "k is out of bounds in DViewFMatrixKokkos 4D!");
8392     assert(l >= 1 && l <= dims_[3] && "l is out of bounds in DViewFMatrixKokkos 4D!");
8393     return this_matrix_((i - 1) + ((j - 1) * dims_[0])
8394                                 + ((k - 1) * dims_[0] * dims_[1])
8395                                 + ((l - 1) * dims_[0] * dims_[1] * dims_[2]));
8396 }
8397
8398 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
8399 KOKKOS_INLINE_FUNCTION
8400 T& DViewFMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::operator()(size_t i, size_t j, size_t k, size_t
      l,
8401                                 size_t m) const {
8402     assert(order_ == 5 && "Tensor order (rank) does not match constructor in DViewFMatrixKokkos 5D!");
8403     assert(i >= 1 && i <= dims_[0] && "i is out of bounds in DViewFMatrixKokkos 5D!");
8404     assert(j >= 1 && j <= dims_[1] && "j is out of bounds in DViewFMatrixKokkos 5D!");
8405     assert(k >= 1 && k <= dims_[2] && "k is out of bounds in DViewFMatrixKokkos 5D!");
8406     assert(l >= 1 && l <= dims_[3] && "l is out of bounds in DViewFMatrixKokkos 5D!");
8407     assert(m >= 1 && m <= dims_[4] && "m is out of bounds in DViewFMatrixKokkos 5D!");
8408     return this_matrix_((i - 1) + ((j - 1) * dims_[0])
8409                                 + ((k - 1) * dims_[0] * dims_[1])
8410                                 + ((l - 1) * dims_[0] * dims_[1] * dims_[2])
8411                                 + ((m - 1) * dims_[0] * dims_[1] * dims_[2] * dims_[3]));
8412 }
8413
8414 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
8415 KOKKOS_INLINE_FUNCTION
8416 T& DViewFMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::operator()(size_t i, size_t j, size_t k, size_t
      l,
8417                                 size_t m, size_t n) const {
```

```
8418     assert(order_ == 6 && "Tensor order (rank) does not match constructor in DViewFMatrixKokkos 6D!");
8419     assert(i >= 1 && i <= dims_[0] && "i is out of bounds in DViewFMatrixKokkos 6D!");
8420     assert(j >= 1 && j <= dims_[1] && "j is out of bounds in DViewFMatrixKokkos 6D!");
8421     assert(k >= 1 && k <= dims_[2] && "k is out of bounds in DViewFMatrixKokkos 6D!");
8422     assert(l >= 1 && l <= dims_[3] && "l is out of bounds in DViewFMatrixKokkos 6D!");
8423     assert(m >= 1 && m <= dims_[4] && "m is out of bounds in DViewFMatrixKokkos 6D!");
8424     assert(n >= 1 && n <= dims_[5] && "n is out of bounds in DViewFMatrixKokkos 6D!");
8425     return this_matrix_((i - 1) + ((j - 1) * dims_[0])
8426                                 + ((k - 1) * dims_[0] * dims_[1])
8427                                 + ((l - 1) * dims_[0] * dims_[1] * dims_[2])
8428                                 + ((m - 1) * dims_[0] * dims_[1] * dims_[2] * dims_[3])
8429                                 + ((n - 1) * dims_[0] * dims_[1] * dims_[2] * dims_[3] * dims_[4]));
8430 }
8431
8432 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
8433 KOKKOS_INLINE_FUNCTION
8434 T& DViewFMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::operator()(size_t i, size_t j, size_t k, size_t
      l,
8435                                 size_t m, size_t n, size_t o) const {
8436     assert(order_ == 7 && "Tensor order (rank) does not match constructor in DViewFMatrixKokkos 7D!");
8437     assert(i >= 1 && i <= dims_[0] && "i is out of bounds in DViewFMatrixKokkos 7D!");
8438     assert(j >= 1 && j <= dims_[1] && "j is out of bounds in DViewFMatrixKokkos 7D!");
8439     assert(k >= 1 && k <= dims_[2] && "k is out of bounds in DViewFMatrixKokkos 7D!");
8440     assert(l >= 1 && l <= dims_[3] && "l is out of bounds in DViewFMatrixKokkos 7D!");
8441     assert(m >= 1 && m <= dims_[4] && "m is out of bounds in DViewFMatrixKokkos 7D!");
8442     assert(n >= 1 && n <= dims_[5] && "n is out of bounds in DViewFMatrixKokkos 7D!");
8443     assert(o >= 1 && o <= dims_[6] && "o is out of bounds in DViewFMatrixKokkos 7D!");
8444     return this_matrix_((i - 1) + ((j - 1) * dims_[0])
8445                                 + ((k - 1) * dims_[0] * dims_[1])
8446                                 + ((l - 1) * dims_[0] * dims_[1] * dims_[2])
8447                                 + ((m - 1) * dims_[0] * dims_[1] * dims_[2] * dims_[3])
8448                                 + ((n - 1) * dims_[0] * dims_[1] * dims_[2] * dims_[3] * dims_[4])
8449                                 + ((o - 1) * dims_[0] * dims_[1] * dims_[2] * dims_[3] * dims_[4] *
      dims_[5]));
8450 }
8451
8452 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
8453 KOKKOS_INLINE_FUNCTION
8454 DViewFMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>&
      DViewFMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::operator= (const DViewFMatrixKokkos& temp) {
8455
8456     // Do nothing if the assignment is of the form x = x
8457     if (this != &temp) {
8458         for (int iter = 0; iter < temp.order_; iter++){
8459             dims_[iter] = temp.dims_[iter];
8460         } // end for
8461
8462         order_ = temp.order_;
8463         length_ = temp.length_;
8464         temp_inp_matrix_ = temp.temp_inp_matrix_;
8465         this_matrix_host_ = temp.this_matrix_host_;
8466         this_matrix_ = temp.this_matrix_;
8467     host = temp.host;
8468     }
8469
8470     return *this;
8471 }
8472
8473 // Return size
8474 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
8475 KOKKOS_INLINE_FUNCTION
8476 size_t DViewFMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::size() const {
8477     return length_;
8478 }
8479
8480 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
8481 KOKKOS_INLINE_FUNCTION
8482 size_t DViewFMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::extent() const {
8483     return length_;
8484 }
8485
8486 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
8487 KOKKOS_INLINE_FUNCTION
8488 size_t DViewFMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::dims(size_t i) const {
8489     i--;
8490     assert(i < order_ && "DViewFMatrixKokkos order (rank) does not match constructor, dim[i] does not
      exist!");
8491     assert(i >= 0 && dims_[i]>0 && "Access to DViewFMatrixKokkos dims is out of bounds!");
8492     return dims_[i];
8493 }
8494
8495 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
8496 KOKKOS_INLINE_FUNCTION
8497 size_t DViewFMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::order() const {
8498     return order_;
8499 }
8500
```

```
8501 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
8502 KOKKOS_INLINE_FUNCTION
8503 T* DViewFMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::device_pointer() const {
8504     return this_matrix_.data();
8505 }
8506
8507 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
8508 KOKKOS_INLINE_FUNCTION
8509 T* DViewFMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::host_pointer() const {
8510     return this_matrix_host_.data();
8511 }
8512
8513 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
8514 void DViewFMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::update_host() {
8515     // Deep copy of device view to host view
8516     deep_copy(this_matrix_host_, this_matrix_);
8517 }
8518
8519 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
8520 void DViewFMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::update_device() {
8521     // Deep copy of host view to device view
8522     deep_copy(this_matrix_, this_matrix_host_);
8523 }
8524
8525 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
8526 KOKKOS_INLINE_FUNCTION
8527 DViewFMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::~DViewFMatrixKokkos() {}
8528 // End DViewFMatrixKokkos
8529
8530
8534 template <typename T, typename Layout = DefaultLayout, typename ExecSpace = DefaultExecSpace, typename
       MemoryTraits = void>
8535 class CArrayKokkos {
8536
8537     using TArray1D = Kokkos::View<T*, Layout, ExecSpace, MemoryTraits>;
8538
8539 private:
8540     size_t dims_[7];
8541     size_t order_;
8542     size_t length_;
8543     TArray1D this_array_;
8544
8545 public:
8546     CArrayKokkos();
8547
8548     CArrayKokkos(size_t dim0, const std::string& tag_string = DEFAULTSTRINGARRAY);
8549
8550     CArrayKokkos(size_t dim0, size_t dim1, const std::string& tag_string = DEFAULTSTRINGARRAY);
8551
8552     CArrayKokkos (size_t dim0, size_t dim1, size_t dim2, const std::string& tag_string =
       DEFAULTSTRINGARRAY);
8553
8554     CArrayKokkos(size_t dim0, size_t dim1, size_t dim2,
8555                  size_t dim3, const std::string& tag_string = DEFAULTSTRINGARRAY);
8556
8557     CArrayKokkos(size_t dim0, size_t dim1, size_t dim2,
8558                  size_t dim3, size_t dim4, const std::string& tag_string = DEFAULTSTRINGARRAY);
8559
8560     CArrayKokkos(size_t dim0, size_t dim1, size_t dim2,
8561                  size_t dim3, size_t dim4, size_t dim5, const std::string& tag_string =
       DEFAULTSTRINGARRAY);
8562
8563     CArrayKokkos(size_t dim0, size_t dim1, size_t dim2,
8564                  size_t dim3, size_t dim4, size_t dim5,
8565                  size_t dim6, const std::string& tag_string = DEFAULTSTRINGARRAY);
8566
8567     KOKKOS_INLINE_FUNCTION
8568     T& operator()(size_t i) const;
8569
8570     KOKKOS_INLINE_FUNCTION
8571     T& operator()(size_t i, size_t j) const;
8572
8573     KOKKOS_INLINE_FUNCTION
8574     T& operator()(size_t i, size_t j, size_t k) const;
8575
8576     KOKKOS_INLINE_FUNCTION
8577     T& operator()(size_t i, size_t j, size_t k, size_t l) const;
8578
8579     KOKKOS_INLINE_FUNCTION
8580     T& operator()(size_t i, size_t j, size_t k, size_t l, size_t m) const;
8581
8582     KOKKOS_INLINE_FUNCTION
8583     T& operator()(size_t i, size_t j, size_t k, size_t l, size_t m,
8584                   size_t n) const;
8585
8586     KOKKOS_INLINE_FUNCTION
8587     T& operator()(size_t i, size_t j, size_t k, size_t l, size_t m,
```

```
8588                   size_t n, size_t o) const;
8589
8590     KOKKOS_INLINE_FUNCTION
8591     CArrayKokkos& operator=(const CArrayKokkos& temp);
8592
8593     // GPU Method
8594     // Method that returns size
8595     KOKKOS_INLINE_FUNCTION
8596     size_t size() const;
8597
8598     // Host Method
8599     // Method that returns size
8600     KOKKOS_INLINE_FUNCTION
8601     size_t extent() const;
8602
8603     KOKKOS_INLINE_FUNCTION
8604     size_t dims(size_t i) const;
8605
8606     KOKKOS_INLINE_FUNCTION
8607     size_t order() const;
8608
8609     // Methods returns the raw pointer (most likely GPU) of the Kokkos View
8610     KOKKOS_INLINE_FUNCTION
8611     T* pointer() const;
8612
8613     //return the view
8614     KOKKOS_INLINE_FUNCTION
8615     TArray1D get_kokkos_view() const;
8616
8617     // Deconstructor
8618     KOKKOS_INLINE_FUNCTION
8619     ~CArrayKokkos ();
8620 }; // End of CArrayKokkos
8621
8622 // Default constructor
8623 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
8624 CArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::CArrayKokkos() {}
8625
8626 // Overloaded 1D constructor
8627 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
8628 CArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::CArrayKokkos(size_t dim0, const std::string& tag_string)
       {
8629     using TArray1D = Kokkos::View<T*, Layout, ExecSpace>;
8630
8631     dims_[0] = dim0;
8632     order_ = 1;
8633     length_ = dim0;
8634     this_array_ = TArray1D(tag_string, length_);
8635 }
8636
8637 // Overloaded 2D constructor
8638 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
8639 CArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::CArrayKokkos(size_t dim0, size_t dim1, const
       std::string& tag_string) {
8640     using TArray1D = Kokkos::View<T*, Layout, ExecSpace>;
8641
8642     dims_[0] = dim0;
8643     dims_[1] = dim1;
8644     order_ = 2;
8645     length_ = (dim0 * dim1);
8646     this_array_ = TArray1D(tag_string, length_);
8647 }
8648
8649 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
8650 CArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::CArrayKokkos(size_t dim0, size_t dim1,
8651                                size_t dim2, const std::string& tag_string) {
8652     using TArray1D = Kokkos::View<T*, Layout, ExecSpace>;
8653
8654     dims_[0] = dim0;
8655     dims_[1] = dim1;
8656     dims_[2] = dim2;
8657     order_ = 3;
8658     length_ = (dim0 * dim1 * dim2);
8659     this_array_ = TArray1D(tag_string, length_);
8660 }
8661
8662 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
8663 CArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::CArrayKokkos(size_t dim0, size_t dim1,
8664                                size_t dim2, size_t dim3, const std::string& tag_string) {
8665     using TArray1D = Kokkos::View<T *,Layout,ExecSpace>;
8666
8667     dims_[0] = dim0;
8668     dims_[1] = dim1;
8669     dims_[2] = dim2;
8670     dims_[3] = dim3;
8671     order_ = 4;
8672     length_ = (dim0 * dim1 * dim2 * dim3);
```

```
8673     this_array_ = TArray1D(tag_string, length_);
8674 }
8675
8676 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
8677 CArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::CArrayKokkos(size_t dim0, size_t dim1,
8678                                size_t dim2, size_t dim3,
8679                                size_t dim4, const std::string& tag_string) {
8680
8681     using TArray1D = Kokkos::View<T *,Layout,ExecSpace>;
8682
8683     dims_[0] = dim0;
8684     dims_[1] = dim1;
8685     dims_[2] = dim2;
8686     dims_[3] = dim3;
8687     dims_[4] = dim4;
8688     order_ = 5;
8689     length_ = (dim0 * dim1 * dim2 * dim3 * dim4);
8690     this_array_ = TArray1D(tag_string, length_);
8691 }
8692
8693 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
8694 CArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::CArrayKokkos(size_t dim0, size_t dim1,
8695                                size_t dim2, size_t dim3,
8696                                size_t dim4, size_t dim5, const std::string& tag_string) {
8697     using TArray1D = Kokkos::View<T *,Layout,ExecSpace>;
8698
8699     dims_[0] = dim0;
8700     dims_[1] = dim1;
8701     dims_[2] = dim2;
8702     dims_[3] = dim3;
8703     dims_[4] = dim4;
8704     dims_[5] = dim5;
8705     order_ = 6;
8706     length_ = (dim0 * dim1 * dim2 * dim3 * dim4 * dim5);
8707     this_array_ = TArray1D(tag_string, length_);
8708 }
8709
8710 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
8711 CArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::CArrayKokkos(size_t dim0, size_t dim1,
8712                                size_t dim2, size_t dim3,
8713                                size_t dim4, size_t dim5,
8714                                size_t dim6, const std::string& tag_string) {
8715     using TArray1D = Kokkos::View<T *,Layout,ExecSpace>;
8716
8717     dims_[0] = dim0;
8718     dims_[1] = dim1;
8719     dims_[2] = dim2;
8720     dims_[3] = dim3;
8721     dims_[4] = dim4;
8722     dims_[5] = dim5;
8723     dims_[6] = dim6;
8724     order_ = 7;
8725     length_ = (dim0 * dim1 * dim2 * dim3 * dim4 * dim5 * dim6);
8726     this_array_ = TArray1D(tag_string, length_);
8727 }
8728
8729 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
8730 KOKKOS_INLINE_FUNCTION
8731 T& CArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::operator()(size_t i) const {
8732     assert(order_ == 1 && "Tensor order (rank) does not match constructor in CArrayKokkos 1D!");
8733     assert(i >= 0 && i < dims_[0] && "i is out of bounds in CArrayKokkos 1D!");
8734     return this_array_(i);
8735 }
8736
8737 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
8738 KOKKOS_INLINE_FUNCTION
8739 T& CArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::operator()(size_t i, size_t j) const {
8740     assert(order_ == 2 && "Tensor order (rank) does not match constructor in CArrayKokkos 2D!");
8741     assert(i >= 0 && i < dims_[0] && "i is out of bounds in CArrayKokkos 2D!");
8742     assert(j >= 0 && j < dims_[1] && "j is out of bounds in CArrayKokkos 2D!");
8743     return this_array_(j + (i * dims_[1]));
8744 }
8745
8746 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
8747 KOKKOS_INLINE_FUNCTION
8748 T& CArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::operator()(size_t i, size_t j, size_t k) const {
8749     assert(order_ == 3 && "Tensor order (rank) does not match constructor in CArrayKokkos 3D!");
8750     assert(i >= 0 && i < dims_[0] && "i is out of bounds in CArrayKokkos 3D!");
8751     assert(j >= 0 && j < dims_[1] && "j is out of bounds in CArrayKokkos 3D!");
8752     assert(k >= 0 && k < dims_[2] && "k is out of bounds in CArrayKokkos 3D!");
8753     return this_array_(k + (j * dims_[2])
8754                        + (i * dims_[2] * dims_[1]));
8755 }
8756
8757 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
8758 KOKKOS_INLINE_FUNCTION
8759 T& CArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::operator()(size_t i, size_t j, size_t k, size_t l)
```

```
    const {
8760    assert(order_ == 4 && "Tensor order (rank) does not match constructor in CArrayKokkos 4D!");
8761    assert(i >= 0 && i < dims_[0] && "i is out of bounds in CArrayKokkos 4D!");
8762    assert(j >= 0 && j < dims_[1] && "j is out of bounds in CArrayKokkos 4D!");
8763    assert(k >= 0 && k < dims_[2] && "k is out of bounds in CArrayKokkos 4D!");
8764    assert(l >= 0 && l < dims_[3] && "l is out of bounds in CArrayKokkos 4D!");
8765    return this_array_(l + (k * dims_[3])
8766                        + (j * dims_[3] * dims_[2])
8767                        + (i * dims_[3] * dims_[2] * dims_[1]));
8768 }
8769
8770 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
8771 KOKKOS_INLINE_FUNCTION
8772 T& CArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::operator()(size_t i, size_t j, size_t k, size_t l,
8773                                size_t m) const {
8774    assert(order_ == 5 && "Tensor order (rank) does not match constructor in CArrayKokkos 5D!");
8775    assert(i >= 0 && i < dims_[0] && "i is out of bounds in CArrayKokkos 5D!");
8776    assert(j >= 0 && j < dims_[1] && "j is out of bounds in CArrayKokkos 5D!");
8777    assert(k >= 0 && k < dims_[2] && "k is out of bounds in CArrayKokkos 5D!");
8778    assert(l >= 0 && l < dims_[3] && "l is out of bounds in CArrayKokkos 5D!");
8779    assert(m >= 0 && m < dims_[4] && "m is out of bounds in CArrayKokkos 5D!");
8780    return this_array_(m + (l * dims_[4])
8781                        + (k * dims_[4] * dims_[3])
8782                        + (j * dims_[4] * dims_[3] * dims_[2])
8783                        + (i * dims_[4] * dims_[3] * dims_[2] * dims_[1]));
8784 }
8785
8786 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
8787 KOKKOS_INLINE_FUNCTION
8788 T& CArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::operator()(size_t i, size_t j, size_t k, size_t l,
8789                                size_t m, size_t n) const {
8790    assert(order_ == 6 && "Tensor order (rank) does not match constructor in CArrayKokkos 6D!");
8791    assert(i >= 0 && i < dims_[0] && "i is out of bounds in CArrayKokkos 6D!");
8792    assert(j >= 0 && j < dims_[1] && "j is out of bounds in CArrayKokkos 6D!");
8793    assert(k >= 0 && k < dims_[2] && "k is out of bounds in CArrayKokkos 6D!");
8794    assert(l >= 0 && l < dims_[3] && "l is out of bounds in CArrayKokkos 6D!");
8795    assert(m >= 0 && m < dims_[4] && "m is out of bounds in CArrayKokkos 6D!");
8796    assert(n >= 0 && n < dims_[5] && "n is out of bounds in CArrayKokkos 6D!");
8797    return this_array_(n + (m * dims_[5])
8798                        + (l * dims_[5] * dims_[4])
8799                        + (k * dims_[5] * dims_[4] * dims_[3])
8800                        + (j * dims_[5] * dims_[4] * dims_[3] * dims_[2])
8801                        + (i * dims_[5] * dims_[4] * dims_[3] * dims_[2] * dims_[1]));
8802 }
8803
8804 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
8805 KOKKOS_INLINE_FUNCTION
8806 T& CArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::operator()(size_t i, size_t j, size_t k, size_t l,
8807                                size_t m, size_t n, size_t o) const {
8808    assert(order_ == 7 && "Tensor order (rank) does not match constructor in CArrayKokkos 7D!");
8809    assert(i >= 0 && i < dims_[0] && "i is out of bounds in CArrayKokkos 7D!");
8810    assert(j >= 0 && j < dims_[1] && "j is out of bounds in CArrayKokkos 7D!");
8811    assert(k >= 0 && k < dims_[2] && "k is out of bounds in CArrayKokkos 7D!");
8812    assert(l >= 0 && l < dims_[3] && "l is out of bounds in CArrayKokkos 7D!");
8813    assert(m >= 0 && m < dims_[4] && "m is out of bounds in CArrayKokkos 7D!");
8814    assert(n >= 0 && n < dims_[5] && "n is out of bounds in CArrayKokkos 7D!");
8815    assert(o >= 0 && o < dims_[6] && "o is out of bounds in CArrayKokkos 7D!");
8816    return this_array_(o + (n * dims_[6])
8817                        + (m * dims_[6] * dims_[5])
8818                        + (l * dims_[6] * dims_[5] * dims_[4])
8819                        + (k * dims_[6] * dims_[5] * dims_[4] * dims_[3])
8820                        + (j * dims_[6] * dims_[5] * dims_[4] * dims_[3] * dims_[2])
8821                        + (i * dims_[6] * dims_[5] * dims_[4] * dims_[3] * dims_[2] * dims_[1]));
8822 }
8823
8824 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
8825 KOKKOS_INLINE_FUNCTION
8826 CArrayKokkos<T,Layout,ExecSpace,MemoryTraits>& CArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::operator=
    (const CArrayKokkos<T,Layout,ExecSpace,MemoryTraits>& temp) {
8827    using TArray1D = Kokkos::View<T *,Layout,ExecSpace>;
8828
8829    // Do nothing if the assignment is of the form x = x
8830    if (this != &temp) {
8831        for (int iter = 0; iter < temp.order_; iter++){
8832            dims_[iter] = temp.dims_[iter];
8833        } // end for
8834
8835        order_ = temp.order_;
8836        length_ = temp.length_;
8837        this_array_ = temp.this_array_;
8838    }
8839
8840    return *this;
8841 }
8842
8843 // Return size
8844 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
```

```
8845 KOKKOS_INLINE_FUNCTION
8846 size_t CArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::size() const {
8847     return length_;
8848 }
8849
8850 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
8851 KOKKOS_INLINE_FUNCTION
8852 size_t CArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::extent() const {
8853     return length_;
8854 }
8855
8856 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
8857 KOKKOS_INLINE_FUNCTION
8858 size_t CArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::dims(size_t i) const {
8859     assert(i < order_ && "CArrayKokkos order (rank) does not match constructor, dim[i] does not
      exist!");
8860     assert(i >= 0 && dims_[i]>0 && "Access to CArrayKokkos dims is out of bounds!");
8861     return dims_[i];
8862 }
8863
8864 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
8865 KOKKOS_INLINE_FUNCTION
8866 size_t CArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::order() const {
8867     return order_;
8868 }
8869
8870 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
8871 KOKKOS_INLINE_FUNCTION
8872 T* CArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::pointer() const {
8873     return this_array_.data();
8874 }
8875
8876 //return the stored Kokkos view
8877 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
8878 KOKKOS_INLINE_FUNCTION
8879 Kokkos::View<T*, Layout, ExecSpace, MemoryTraits>
      CArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::get_kokkos_view() const {
8880     return this_array_;
8881 }
8882
8883 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
8884 KOKKOS_INLINE_FUNCTION
8885 CArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::~CArrayKokkos() {}
8886
8888 // End of CArrayKokkos
8890
8894 template <typename T>
8895 class ViewCArrayKokkos {
8896
8897 private:
8898     size_t dims_[7];
8899     size_t order_;
8900     size_t length_;  // Length of 1D array
8901     T* this_array_;
8902
8903 public:
8904     KOKKOS_INLINE_FUNCTION
8905     ViewCArrayKokkos();
8906
8907     KOKKOS_INLINE_FUNCTION
8908     ViewCArrayKokkos(T* some_array, size_t dim0);
8909
8910     KOKKOS_INLINE_FUNCTION
8911     ViewCArrayKokkos(T* some_array, size_t dim0, size_t dim1);
8912
8913     KOKKOS_INLINE_FUNCTION
8914     ViewCArrayKokkos(T* some_array, size_t dim0, size_t dim1,
8915                      size_t dim2);
8916
8917     KOKKOS_INLINE_FUNCTION
8918     ViewCArrayKokkos(T* some_array, size_t dim0, size_t dim1,
8919                      size_t dim2, size_t dim3);
8920
8921     KOKKOS_INLINE_FUNCTION
8922     ViewCArrayKokkos(T* some_array, size_t dim0, size_t dim1,
8923                      size_t dim2, size_t dim3, size_t dim4);
8924
8925     KOKKOS_INLINE_FUNCTION
8926     ViewCArrayKokkos(T* some_array, size_t dim0, size_t dim1,
8927                      size_t dim2, size_t dim3, size_t dim4,
8928                      size_t dim5);
8929
8930     KOKKOS_INLINE_FUNCTION
8931     ViewCArrayKokkos(T* some_array, size_t dim0, size_t dim1,
8932                      size_t dim2, size_t dim3, size_t dim4,
8933                      size_t dim5, size_t dim6);;
8934
```

```
8935     KOKKOS_INLINE_FUNCTION
8936     T& operator()(size_t i) const;
8937
8938     KOKKOS_INLINE_FUNCTION
8939     T& operator()(size_t i, size_t j) const;
8940
8941     KOKKOS_INLINE_FUNCTION
8942     T& operator()(size_t i, size_t j, size_t k) const;
8943
8944     KOKKOS_INLINE_FUNCTION
8945     T& operator()(size_t i, size_t j, size_t k, size_t l) const;
8946
8947     KOKKOS_INLINE_FUNCTION
8948     T& operator() (size_t i, size_t j, size_t k, size_t l, size_t m) const;
8949
8950     KOKKOS_INLINE_FUNCTION
8951     T& operator()(size_t i, size_t j, size_t k, size_t l, size_t m,
8952                   size_t n) const;
8953
8954     KOKKOS_INLINE_FUNCTION
8955     T& operator()(size_t i, size_t j, size_t k, size_t l, size_t m,
8956                   size_t n, size_t o) const;
8957
8958     KOKKOS_INLINE_FUNCTION
8959     size_t size() const;
8960
8961     KOKKOS_INLINE_FUNCTION
8962     size_t extent() const;
8963
8964     KOKKOS_INLINE_FUNCTION
8965     size_t dims(size_t i) const;
8966
8967     KOKKOS_INLINE_FUNCTION
8968     size_t order() const;
8969
8970     KOKKOS_INLINE_FUNCTION
8971     T* pointer() const;
8972
8973     KOKKOS_INLINE_FUNCTION
8974     ~ViewCArrayKokkos();
8975
8976 }; // end of ViewCArrayKokkos
8977
8978 // Default constructor
8979 template <typename T>
8980 KOKKOS_INLINE_FUNCTION
8981 ViewCArrayKokkos<T>::ViewCArrayKokkos() {}
8982
8983 // Overloaded 1D constructor
8984 template <typename T>
8985 KOKKOS_INLINE_FUNCTION
8986 ViewCArrayKokkos<T>::ViewCArrayKokkos(T* some_array, size_t dim0) {
8987     dims_[0] = dim0;
8988     order_ = 1;
8989     length_ = dim0;
8990     this_array_ = some_array;
8991 }
8992
8993 // Overloaded 2D constructor
8994 template <typename T>
8995 KOKKOS_INLINE_FUNCTION
8996 ViewCArrayKokkos<T>::ViewCArrayKokkos(T* some_array, size_t dim0,
8997                                       size_t dim1) {
8998     dims_[0] = dim0;
8999     dims_[1] = dim1;
9000     order_ = 2;
9001     length_ = (dim0 * dim1);
9002     this_array_ = some_array;
9003 }
9004
9005 // Overloaded 3D constructor
9006 template <typename T>
9007 KOKKOS_INLINE_FUNCTION
9008 ViewCArrayKokkos<T>::ViewCArrayKokkos(T* some_array, size_t dim0,
9009                                       size_t dim1, size_t dim2) {
9010     dims_[0] = dim0;
9011     dims_[1] = dim1;
9012     dims_[2] = dim2;
9013     order_ = 3;
9014     length_ = (dim0 * dim1 * dim2);
9015     this_array_ = some_array;
9016 }
9017
9018 // Overloaded 4D constructor
9019 template <typename T>
9020 KOKKOS_INLINE_FUNCTION
9021 ViewCArrayKokkos<T>::ViewCArrayKokkos(T* some_array, size_t dim0,
```

```
9022                                        size_t dim1, size_t dim2,
9023                                        size_t dim3) {
9024     dims_[0] = dim0;
9025     dims_[1] = dim1;
9026     dims_[2] = dim2;
9027     dims_[3] = dim3;
9028     order_ = 4;
9029     length_ = (dim0 * dim1 * dim2 * dim3);
9030     this_array_ = some_array;
9031 }
9032
9033 // Overloaded 5D constructor
9034 template <typename T>
9035 KOKKOS_INLINE_FUNCTION
9036 ViewCArrayKokkos<T>::ViewCArrayKokkos(T* some_array, size_t dim0,
9037                                        size_t dim1, size_t dim2,
9038                                        size_t dim3, size_t dim4) {
9039     dims_[0] = dim0;
9040     dims_[1] = dim1;
9041     dims_[2] = dim2;
9042     dims_[3] = dim3;
9043     dims_[4] = dim4;
9044     order_ = 5;
9045     length_ = (dim0 * dim1 * dim2 * dim3 * dim4);
9046     this_array_ = some_array;
9047 }
9048
9049 // Overloaded 6D constructor
9050 template <typename T>
9051 KOKKOS_INLINE_FUNCTION
9052 ViewCArrayKokkos<T>::ViewCArrayKokkos(T* some_array, size_t dim0,
9053                                        size_t dim1, size_t dim2,
9054                                        size_t dim3, size_t dim4,
9055                                        size_t dim5) {
9056     dims_[0] = dim0;
9057     dims_[1] = dim1;
9058     dims_[2] = dim2;
9059     dims_[3] = dim3;
9060     dims_[4] = dim4;
9061     dims_[5] = dim5;
9062     order_ = 6;
9063     length_ = (dim0 * dim1 * dim2 * dim3 * dim4 * dim5);
9064     this_array_ = some_array;
9065 }
9066
9067 // Overloaded 7D constructor
9068 template <typename T>
9069 KOKKOS_INLINE_FUNCTION
9070 ViewCArrayKokkos<T>::ViewCArrayKokkos(T* some_array, size_t dim0,
9071                                        size_t dim1, size_t dim2,
9072                                        size_t dim3, size_t dim4,
9073                                        size_t dim5, size_t dim6) {
9074     dims_[0] = dim0;
9075     dims_[1] = dim1;
9076     dims_[2] = dim2;
9077     dims_[3] = dim3;
9078     dims_[4] = dim4;
9079     dims_[5] = dim5;
9080     dims_[6] = dim6;
9081     order_ = 7;
9082     length_ = (dim0 * dim1 * dim2 * dim3 * dim4 * dim5 * dim6);
9083     this_array_ = some_array;
9084 }
9085
9086 template <typename T>
9087 KOKKOS_INLINE_FUNCTION
9088 T& ViewCArrayKokkos<T>::operator()(size_t i) const {
9089     assert(order_ == 1 && "Tensor order (rank) does not match constructor in ViewCArrayKokkos 1D!");
9090     assert(i >= 0 && i < dims_[0] && "i is out of bounds in ViewCArrayKokkos 1D!");
9091     return this_array_[i];
9092 }
9093
9094 template <typename T>
9095 KOKKOS_INLINE_FUNCTION
9096 T& ViewCArrayKokkos<T>::operator()(size_t i, size_t j) const {
9097     assert(order_ == 2 && "Tensor order (rank) does not match constructor in ViewCArrayKokkos 2D!");
9098     assert(i >= 0 && i < dims_[0] && "i is out of bounds in ViewCArrayKokkos 2D!");
9099     assert(j >= 0 && j < dims_[1] && "j is out of bounds in ViewCArrayKokkos 2D!");
9100     return this_array_[j + (i * dims_[1])];
9101 }
9102
9103 template <typename T>
9104 KOKKOS_INLINE_FUNCTION
9105 T& ViewCArrayKokkos<T>::operator()(size_t i, size_t j, size_t k) const {
9106     assert(order_ == 3 && "Tensor order (rank) does not match constructor in ViewCArrayKokkos 3D!");
9107     assert(i >= 0 && i < dims_[0] && "i is out of bounds in ViewCArrayKokkos 3D!");
9108     assert(j >= 0 && j < dims_[1] && "j is out of bounds in ViewCArrayKokkos 3D!");
```

```
9109     assert(k >= 0 && k < dims_[2] && "k is out of bounds in ViewCArrayKokkos 3D!");
9110     return this_array_[k + (j * dims_[2])
9111                        + (i * dims_[2] * dims_[1])];
9112 }
9113
9114 template <typename T>
9115 KOKKOS_INLINE_FUNCTION
9116 T& ViewCArrayKokkos<T>::operator()(size_t i, size_t j, size_t k,
9117                                    size_t l) const {
9118     assert(order_ == 4 && "Tensor order (rank) does not match constructor in ViewCArrayKokkos 4D!");
9119     assert(i >= 0 && i < dims_[0] && "i is out of bounds in ViewCArrayKokkos 4D!");
9120     assert(j >= 0 && j < dims_[1] && "j is out of bounds in ViewCArrayKokkos 4D!");
9121     assert(k >= 0 && k < dims_[2] && "k is out of bounds in ViewCArrayKokkos 4D!");
9122     assert(l >= 0 && l < dims_[3] && "l is out of bounds in ViewCArrayKokkos 4D!");
9123     return this_array_[l + (k * dims_[3])
9124                        + (j * dims_[3] * dims_[2])
9125                        + (i * dims_[3] * dims_[2] * dims_[1])];
9126 }
9127
9128 template <typename T>
9129 KOKKOS_INLINE_FUNCTION
9130 T& ViewCArrayKokkos<T>::operator()(size_t i, size_t j, size_t k, size_t l,
9131                                    size_t m) const {
9132     assert(order_ == 5 && "Tensor order (rank) does not match constructor in ViewCArrayKokkos 5D!");
9133     assert(i >= 0 && i < dims_[0] && "i is out of bounds in ViewCArrayKokkos 5D!");
9134     assert(j >= 0 && j < dims_[1] && "j is out of bounds in ViewCArrayKokkos 5D!");
9135     assert(k >= 0 && k < dims_[2] && "k is out of bounds in ViewCArrayKokkos 5D!");
9136     assert(l >= 0 && l < dims_[3] && "l is out of bounds in ViewCArrayKokkos 5D!");
9137     assert(m >= 0 && m < dims_[4] && "m is out of bounds in ViewCArrayKokkos 5D!");
9138     return this_array_[m + (l * dims_[4])
9139                        + (k * dims_[4] * dims_[3])
9140                        + (j * dims_[4] * dims_[3] * dims_[2])
9141                        + (i * dims_[4] * dims_[3] * dims_[2] * dims_[1])];
9142 }
9143
9144 template <typename T>
9145 KOKKOS_INLINE_FUNCTION
9146 T& ViewCArrayKokkos<T>::operator()(size_t i, size_t j, size_t k, size_t l,
9147                                    size_t m, size_t n) const {
9148     assert(order_ == 6 && "Tensor order (rank) does not match constructor in ViewCArrayKokkos 6D!");
9149     assert(i >= 0 && i < dims_[0] && "i is out of bounds in ViewCArrayKokkos 6D!");
9150     assert(j >= 0 && j < dims_[1] && "j is out of bounds in ViewCArrayKokkos 6D!");
9151     assert(k >= 0 && k < dims_[2] && "k is out of bounds in ViewCArrayKokkos 6D!");
9152     assert(l >= 0 && l < dims_[3] && "l is out of bounds in ViewCArrayKokkos 6D!");
9153     assert(m >= 0 && m < dims_[4] && "m is out of bounds in ViewCArrayKokkos 6D!");
9154     assert(n >= 0 && n < dims_[5] && "n is out of bounds in ViewCArrayKokkos 6D!");
9155     return this_array_[n + (m * dims_[5])
9156                        + (l * dims_[5] * dims_[4])
9157                        + (k * dims_[5] * dims_[4] * dims_[3])
9158                        + (j * dims_[5] * dims_[4] * dims_[3] * dims_[2])
9159                        + (i * dims_[5] * dims_[4] * dims_[3] * dims_[2] * dims_[1])];
9160 }
9161
9162 template <typename T>
9163 KOKKOS_INLINE_FUNCTION
9164 T& ViewCArrayKokkos<T>::operator()(size_t i, size_t j, size_t k, size_t l,
9165                                    size_t m, size_t n, size_t o) const {
9166     assert(order_ == 7 && "Tensor order (rank) does not match constructor in ViewCArrayKokkos 7D!");
9167     assert(i >= 0 && i < dims_[0] && "i is out of bounds in ViewCArrayKokkos 7D!");
9168     assert(j >= 0 && j < dims_[1] && "j is out of bounds in ViewCArrayKokkos 7D!");
9169     assert(k >= 0 && k < dims_[2] && "k is out of bounds in ViewCArrayKokkos 7D!");
9170     assert(l >= 0 && l < dims_[3] && "l is out of bounds in ViewCArrayKokkos 7D!");
9171     assert(m >= 0 && m < dims_[4] && "m is out of bounds in ViewCArrayKokkos 7D!");
9172     assert(n >= 0 && n < dims_[5] && "n is out of bounds in ViewCArrayKokkos 7D!");
9173     assert(o >= 0 && o < dims_[6] && "o is out of bounds in ViewCArrayKokkos 7D!");
9174     return this_array_[o + (n * dims_[6])
9175                        + (m * dims_[6] * dims_[5])
9176                        + (l * dims_[6] * dims_[5] * dims_[4])
9177                        + (k * dims_[6] * dims_[5] * dims_[4] * dims_[3])
9178                        + (j * dims_[6] * dims_[5] * dims_[4] * dims_[3] * dims_[2])
9179                        + (i * dims_[6] * dims_[5] * dims_[4] * dims_[3] * dims_[2] * dims_[1])];
9180 }
9181
9182 template <typename T>
9183 KOKKOS_INLINE_FUNCTION
9184 size_t ViewCArrayKokkos<T>::size() const {
9185     return length_;
9186 }
9187
9188 template <typename T>
9189 KOKKOS_INLINE_FUNCTION
9190 size_t ViewCArrayKokkos<T>::extent() const {
9191     return length_;
9192 }
9193
9194 template <typename T>
9195 KOKKOS_INLINE_FUNCTION
```

```
9196 size_t ViewCArrayKokkos<T>::dims(size_t i) const {
9197     assert(i < order_ && "ViewCArrayKokkos order (rank) does not match constructor, dim[i] does not
       exist!");
9198     assert(i >= 0 && dims_[i]>0 && "Access to ViewCArrayKokkos dims is out of bounds!");
9199     return dims_[i];
9200 }
9201
9202 template <typename T>
9203 KOKKOS_INLINE_FUNCTION
9204 size_t ViewCArrayKokkos<T>::order() const {
9205     return order_;
9206 }
9207
9208 template <typename T>
9209 KOKKOS_INLINE_FUNCTION
9210 T* ViewCArrayKokkos<T>::pointer() const {
9211     return this_array_;
9212 }
9213
9214 template <typename T>
9215 KOKKOS_INLINE_FUNCTION
9216 ViewCArrayKokkos<T>::~ViewCArrayKokkos() {}
9217
9219 // End of ViewCArrayKokkos
9221
9225 template <typename T, typename Layout = DefaultLayout, typename ExecSpace = DefaultExecSpace, typename
       MemoryTraits = void>
9226 class CMatrixKokkos {
9227
9228     using TArray1D = Kokkos::View<T*, Layout, ExecSpace, MemoryTraits>;
9229
9230 private:
9231     size_t dims_[7];
9232     size_t order_;
9233     size_t length_;
9234     TArray1D this_matrix_;
9235
9236 public:
9237     CMatrixKokkos();
9238
9239     CMatrixKokkos(size_t dim1, const std::string& tag_string = DEFAULTSTRINGMATRIX);
9240
9241     CMatrixKokkos(size_t dim1, size_t dim2, const std::string& tag_string = DEFAULTSTRINGMATRIX);
9242
9243     CMatrixKokkos(size_t dim1, size_t dim2, size_t dim3, const std::string& tag_string =
       DEFAULTSTRINGMATRIX);
9244
9245     CMatrixKokkos(size_t dim1, size_t dim2, size_t dim3,
9246                   size_t dim4, const std::string& tag_string = DEFAULTSTRINGMATRIX);
9247
9248     CMatrixKokkos(size_t dim1, size_t dim2, size_t dim3,
9249                   size_t dim4, size_t dim5, const std::string& tag_string = DEFAULTSTRINGMATRIX);
9250
9251     CMatrixKokkos(size_t dim1, size_t dim2, size_t dim3,
9252                   size_t dim4, size_t dim5, size_t dim6, const std::string& tag_string =
       DEFAULTSTRINGMATRIX);
9253
9254     CMatrixKokkos(size_t dim1, size_t dim2, size_t dim3,
9255                   size_t dim4, size_t dim5, size_t dim6,
9256                   size_t dim7, const std::string& tag_string = DEFAULTSTRINGMATRIX);
9257
9258     KOKKOS_INLINE_FUNCTION
9259     T& operator()(size_t i) const;
9260
9261     KOKKOS_INLINE_FUNCTION
9262     T& operator()(size_t i, size_t j) const;
9263
9264     KOKKOS_INLINE_FUNCTION
9265     T& operator()(size_t i, size_t j, size_t k) const;
9266
9267     KOKKOS_INLINE_FUNCTION
9268     T& operator()(size_t i, size_t j, size_t k, size_t l) const;
9269
9270     KOKKOS_INLINE_FUNCTION
9271     T& operator()(size_t i, size_t j, size_t k, size_t l, size_t m) const;
9272
9273     KOKKOS_INLINE_FUNCTION
9274     T& operator()(size_t i, size_t j, size_t k, size_t l, size_t m,
9275                   size_t n) const;
9276
9277     KOKKOS_INLINE_FUNCTION
9278     T& operator()(size_t i, size_t j, size_t k, size_t l, size_t m,
9279                   size_t n, size_t o) const;
9280
9281     KOKKOS_INLINE_FUNCTION
9282     CMatrixKokkos& operator=(const CMatrixKokkos &temp);
9283
```

```
9284     KOKKOS_INLINE_FUNCTION
9285     size_t size() const;
9286
9287     KOKKOS_INLINE_FUNCTION
9288     size_t extent() const;
9289
9290     KOKKOS_INLINE_FUNCTION
9291     size_t dims(size_t i) const;
9292
9293     KOKKOS_INLINE_FUNCTION
9294     size_t order() const;
9295
9296     KOKKOS_INLINE_FUNCTION
9297     T* pointer() const;
9298
9299     //return the view
9300     KOKKOS_INLINE_FUNCTION
9301     TArray1D get_kokkos_view() const;
9302
9303     KOKKOS_INLINE_FUNCTION
9304     ~CMatrixKokkos();
9305
9306 }; // End of CMatrixKokkos
9307
9308 // Default constructor
9309 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
9310 CMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::CMatrixKokkos() {}
9311
9312 // Overloaded 1D constructor
9313 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
9314 CMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::CMatrixKokkos(size_t dim1, const std::string&
       tag_string) {
9315     using TArray1D = Kokkos::View<T*, Layout, ExecSpace>;
9316
9317     dims_[0] = dim1;
9318     order_ = 1;
9319     length_ = dim1;
9320     this_matrix_ = TArray1D(tag_string, length_);
9321 }
9322
9323 // Overloaded 2D constructor
9324 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
9325 CMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::CMatrixKokkos(size_t dim1, size_t dim2, const
       std::string& tag_string) {
9326     using TArray1D = Kokkos::View<T*, Layout, ExecSpace>;
9327
9328     dims_[0] = dim1;
9329     dims_[1] = dim2;
9330     order_ = 2;
9331     length_ = (dim1 * dim2);
9332     this_matrix_ = TArray1D(tag_string, length_);
9333 }
9334
9335 // Overloaded 3D constructor
9336 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
9337 CMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::CMatrixKokkos(size_t dim1, size_t dim2,
9338                                size_t dim3, const std::string& tag_string) {
9339     using TArray1D = Kokkos::View<T*, Layout, ExecSpace>;
9340
9341     dims_[0] = dim1;
9342     dims_[1] = dim2;
9343     dims_[2] = dim3;
9344     order_ = 3;
9345     length_ = (dim1 * dim2 * dim3);
9346     this_matrix_ = TArray1D(tag_string, length_);
9347 }
9348
9349 // Overloaded 4D constructor
9350 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
9351 CMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::CMatrixKokkos(size_t dim1, size_t dim2,
9352                                size_t dim3, size_t dim4, const std::string& tag_string) {
9353     using TArray1D = Kokkos::View<T*, Layout, ExecSpace>;
9354
9355     dims_[0] = dim1;
9356     dims_[1] = dim2;
9357     dims_[2] = dim3;
9358     dims_[3] = dim4;
9359     order_ = 4;
9360     length_ = (dim1 * dim2 * dim3 * dim4);
9361     this_matrix_ = TArray1D(tag_string, length_);
9362 }
9363
9364 // Overloaded 5D constructor
9365 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
9366 CMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::CMatrixKokkos(size_t dim1, size_t dim2,
9367                                size_t dim3, size_t dim4,
9368                                size_t dim5, const std::string& tag_string) {
```

```
9369
9370     using TArray1D = Kokkos::View<T*, Layout, ExecSpace>;
9371
9372     dims_[0] = dim1;
9373     dims_[1] = dim2;
9374     dims_[2] = dim3;
9375     dims_[3] = dim4;
9376     dims_[4] = dim5;
9377     order_ = 5;
9378     length_ = (dim1 * dim2 * dim3 * dim4 * dim5);
9379     this_matrix_ = TArray1D(tag_string, length_);
9380 }
9381
9382 // Overloaded 6D constructor
9383 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
9384 CMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::CMatrixKokkos(size_t dim1, size_t dim2,
9385                                 size_t dim3, size_t dim4,
9386                                 size_t dim5, size_t dim6, const std::string& tag_string) {
9387     using TArray1D = Kokkos::View<T*, Layout, ExecSpace>;
9388
9389     dims_[0] = dim1;
9390     dims_[1] = dim2;
9391     dims_[2] = dim3;
9392     dims_[3] = dim4;
9393     dims_[4] = dim5;
9394     dims_[5] = dim6;
9395     order_ = 6;
9396     length_ = (dim1 * dim2 * dim3 * dim4 * dim5 * dim6);
9397     this_matrix_ = TArray1D(tag_string, length_);
9398 }
9399
9400 // Overloaded 7D constructor
9401 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
9402 CMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::CMatrixKokkos(size_t dim1, size_t dim2,
9403                                 size_t dim3, size_t dim4,
9404                                 size_t dim5, size_t dim6,
9405                                 size_t dim7, const std::string& tag_string) {
9406     using TArray1D = Kokkos::View<T*, Layout, ExecSpace>;
9407
9408     dims_[0] = dim1;
9409     dims_[1] = dim2;
9410     dims_[2] = dim3;
9411     dims_[3] = dim4;
9412     dims_[4] = dim5;
9413     dims_[5] = dim6;
9414     dims_[6] = dim7;
9415     order_ = 7;
9416     length_ = (dim1 * dim2 * dim3 * dim4 * dim5 * dim6 * dim7);
9417     this_matrix_ = TArray1D(tag_string, length_);
9418 }
9419
9420 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
9421 KOKKOS_INLINE_FUNCTION
9422 T& CMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::operator()(size_t i) const {
9423     assert(order_ == 1 && "Tensor order (rank) does not match constructor in CMatrixKokkos 1D!");
9424     assert(i >= 1 && i <= dims_[0] && "i is out of bounds in CMatrixKokkos 1D!");
9425     return this_matrix_((i - 1));
9426 }
9427
9428 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
9429 KOKKOS_INLINE_FUNCTION
9430 T& CMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::operator()(size_t i, size_t j) const {
9431     assert(order_ == 2 && "Tensor order (rank) does not match constructor in CMatrixKokkos 2D!");
9432     assert(i >= 1 && i <= dims_[0] && "i is out of bounds in CMatrixKokkos 2D!");
9433     assert(j >= 1 && j <= dims_[1] && "j is out of bounds in CMatrixKokkos 2D!");
9434     return this_matrix_((j - 1) + ((i - 1) * dims_[1]));
9435 }
9436
9437 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
9438 KOKKOS_INLINE_FUNCTION
9439 T& CMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::operator()(size_t i, size_t j, size_t k) const {
9440     assert(order_ == 3 && "Tensor order (rank) does not match constructor in CMatrixKokkos 3D!");
9441     assert(i >= 1 && i <= dims_[0] && "i is out of bounds in CMatrixKokkos 3D!");
9442     assert(j >= 1 && j <= dims_[1] && "j is out of bounds in CMatrixKokkos 3D!");
9443     assert(k >= 1 && k <= dims_[2] && "k is out of bounds in CMatrixKokkos 3D!");
9444     return this_matrix_((k - 1) + ((j - 1) * dims_[2])
9445                                 + ((i - 1) * dims_[2] * dims_[1]));
9446 }
9447
9448 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
9449 KOKKOS_INLINE_FUNCTION
9450 T& CMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::operator()(size_t i, size_t j, size_t k, size_t l)
      const {
9451     assert(order_ == 4 && "Tensor order (rank) does not match constructor in CMatrixKokkos 4D!");
9452     assert(i >= 1 && i <= dims_[0] && "i is out of bounds in CMatrixKokkos 4D!");
9453     assert(j >= 1 && j <= dims_[1] && "j is out of bounds in CMatrixKokkos 4D!");
9454     assert(k >= 1 && k <= dims_[2] && "k is out of bounds in CMatrixKokkos 4D!");
```

```
9455     assert(l >= 1 && l <= dims_[3] && "l is out of bounds in CMatrixKokkos 4D!");
9456     return this_matrix_((l - 1) + ((k - 1) * dims_[3])
9457                                 + ((j - 1) * dims_[3] * dims_[2])
9458                                 + ((i - 1) * dims_[3] * dims_[2] * dims_[1]));
9459 }
9460
9461 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
9462 KOKKOS_INLINE_FUNCTION
9463 T& CMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::operator()(size_t i, size_t j, size_t k, size_t l,
9464                                 size_t m) const {
9465     assert(order_ == 5 && "Tensor order (rank) does not match constructor in CMatrixKokkos 5D!");
9466     assert(i >= 1 && i <= dims_[0] && "i is out of bounds in CMatrixKokkos 5D!");
9467     assert(j >= 1 && j <= dims_[1] && "j is out of bounds in CMatrixKokkos 5D!");
9468     assert(k >= 1 && k <= dims_[2] && "k is out of bounds in CMatrixKokkos 5D!");
9469     assert(l >= 1 && l <= dims_[3] && "l is out of bounds in CMatrixKokkos 5D!");
9470     assert(m >= 1 && m <= dims_[4] && "m is out of bounds in CMatrixKokkos 5D!");
9471     return this_matrix_((m - 1) + ((l - 1) * dims_[4])
9472                                 + ((k - 1) * dims_[4] * dims_[3])
9473                                 + ((j - 1) * dims_[4] * dims_[3] * dims_[2])
9474                                 + ((i - 1) * dims_[4] * dims_[3] * dims_[2] * dims_[1]));
9475 }
9476
9477 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
9478 KOKKOS_INLINE_FUNCTION
9479 T& CMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::operator()(size_t i, size_t j, size_t k, size_t l,
9480                                 size_t m, size_t n) const {
9481     assert(order_ == 6 && "Tensor order (rank) does not match constructor in CMatrixKokkos 6D!");
9482     assert(i >= 1 && i <= dims_[0] && "i is out of bounds in CMatrixKokkos 6D!");
9483     assert(j >= 1 && j <= dims_[1] && "j is out of bounds in CMatrixKokkos 6D!");
9484     assert(k >= 1 && k <= dims_[2] && "k is out of bounds in CMatrixKokkos 6D!");
9485     assert(l >= 1 && l <= dims_[3] && "l is out of bounds in CMatrixKokkos 6D!");
9486     assert(m >= 1 && m <= dims_[4] && "m is out of bounds in CMatrixKokkos 6D!");
9487     assert(n >= 1 && n <= dims_[5] && "n is out of bounds in CMatrixKokkos 6D!");
9488     return this_matrix_((n - 1) + ((m - 1) * dims_[5])
9489                                 + ((l - 1) * dims_[5] * dims_[4])
9490                                 + ((k - 1) * dims_[5] * dims_[4] * dims_[3])
9491                                 + ((j - 1) * dims_[5] * dims_[4] * dims_[3] * dims_[2])
9492                                 + ((i - 1) * dims_[5] * dims_[4] * dims_[3] * dims_[2] * dims_[1]));
9493 }
9494
9495 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
9496 KOKKOS_INLINE_FUNCTION
9497 T& CMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::operator()(size_t i, size_t j, size_t k, size_t l,
9498                                 size_t m, size_t n, size_t o) const {
9499     assert(order_ == 7 && "Tensor order (rank) does not match constructor in CMatrixKokkos 7D!");
9500     assert(i >= 1 && i <= dims_[0] && "i is out of bounds in CMatrixKokkos 7D!");
9501     assert(j >= 1 && j <= dims_[1] && "j is out of bounds in CMatrixKokkos 7D!");
9502     assert(k >= 1 && k <= dims_[2] && "k is out of bounds in CMatrixKokkos 7D!");
9503     assert(l >= 1 && l <= dims_[3] && "l is out of bounds in CMatrixKokkos 7D!");
9504     assert(m >= 1 && m <= dims_[4] && "m is out of bounds in CMatrixKokkos 7D!");
9505     assert(n >= 1 && n <= dims_[5] && "n is out of bounds in CMatrixKokkos 7D!");
9506     assert(o >= 1 && o <= dims_[6] && "o is out of bounds in CMatrixKokkos 7D!");
9507     return this_matrix_((o-1) + ((n - 1) * dims_[6])
9508                               + ((m - 1) * dims_[6] * dims_[5])
9509                               + ((l - 1) * dims_[6] * dims_[5] * dims_[4])
9510                               + ((k - 1) * dims_[6] * dims_[5] * dims_[4] * dims_[3])
9511                               + ((j - 1) * dims_[6] * dims_[5] * dims_[4] * dims_[3] * dims_[2])
9512                               + ((i - 1) * dims_[6] * dims_[5] * dims_[4] * dims_[3] * dims_[2] *
   dims_[1]));
9513 }
9514
9515 // Overload = operator
9516 // for object assignment THIS = CMatrixKokkos <> temp
9517 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
9518 KOKKOS_INLINE_FUNCTION
9519 CMatrixKokkos<T,Layout,ExecSpace,MemoryTraits> &
     CMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::operator=(const
     CMatrixKokkos<T,Layout,ExecSpace,MemoryTraits> &temp) {
9520   using TArray1D = Kokkos::View<T*, Layout, ExecSpace>;
9521
9522     if( this != &temp) {
9523         for (int iter = 0; iter < temp.order_; iter++){
9524             dims_[iter] = temp.dims_[iter];
9525         } // end for
9526
9527         order_ = temp.order_;
9528         length_ = temp.length_;
9529         this_matrix_ = temp.this_matrix_;
9530     }
9531
9532     return *this;
9533 }
9534
9535 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
9536 KOKKOS_INLINE_FUNCTION
9537 size_t CMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::size() const {
9538     return length_;
```

```
9539 }
9540
9541 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
9542 KOKKOS_INLINE_FUNCTION
9543 size_t CMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::extent() const {
9544     return length_;
9545 }
9546
9547 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
9548 KOKKOS_INLINE_FUNCTION
9549 size_t CMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::dims(size_t i) const {
9550     i--;
9551     assert(i < order_ && "CMatrixKokkos order (rank) does not match constructor, dim[i] does not
      exist!");
9552     assert(i >= 0 && dims_[i]>0 && "Access to CMatrixKokkos dims is out of bounds!");
9553     return dims_[i];
9554 }
9555
9556 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
9557 KOKKOS_INLINE_FUNCTION
9558 size_t CMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::order() const {
9559     return order_;
9560 }
9561
9562 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
9563 KOKKOS_INLINE_FUNCTION
9564 T* CMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::pointer() const {
9565     return this_matrix_.data();
9566 }
9567
9568 //return the stored Kokkos view
9569 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
9570 KOKKOS_INLINE_FUNCTION
9571 Kokkos::View<T*, Layout, ExecSpace, MemoryTraits>
     CMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::get_kokkos_view() const {
9572     return this_matrix_;
9573 }
9574
9575 // Deconstructor
9576 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
9577 KOKKOS_INLINE_FUNCTION
9578 CMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::~CMatrixKokkos() {}
9579
9581 // End of CMatrixKokkos
9583
9587 template <typename T>
9588 class ViewCMatrixKokkos {
9589
9590 private:
9591     size_t dims_[7];
9592     size_t order_;
9593     size_t length_;
9594     T* this_matrix_;
9595
9596 public:
9597     KOKKOS_INLINE_FUNCTION
9598     ViewCMatrixKokkos();
9599
9600     KOKKOS_INLINE_FUNCTION
9601     ViewCMatrixKokkos(T* some_matrix, size_t dim1);
9602
9603     KOKKOS_INLINE_FUNCTION
9604     ViewCMatrixKokkos(T* some_matrix, size_t dim1, size_t dim2);
9605
9606     KOKKOS_INLINE_FUNCTION
9607     ViewCMatrixKokkos(T* some_matrix, size_t dim1, size_t dim2, size_t dim3);
9608
9609     KOKKOS_INLINE_FUNCTION
9610     ViewCMatrixKokkos(T* some_matrix, size_t dim1, size_t dim2, size_t dim3,
9611                       size_t dim4);
9612
9613     KOKKOS_INLINE_FUNCTION
9614     ViewCMatrixKokkos(T* some_matrix, size_t dim1, size_t dim2, size_t dim3,
9615                       size_t dim4, size_t dim5);
9616
9617     KOKKOS_INLINE_FUNCTION
9618     ViewCMatrixKokkos(T* some_matrix, size_t dim1, size_t dim2, size_t dim3,
9619                       size_t dim4, size_t dim5, size_t dim6);
9620
9621     KOKKOS_INLINE_FUNCTION
9622     ViewCMatrixKokkos(T* some_matrix, size_t dim1, size_t dim2, size_t dim3,
9623                       size_t dim4, size_t dim5, size_t dim6, size_t dim7);
9624
9625     KOKKOS_INLINE_FUNCTION
9626     T& operator()(size_t i) const;
9627
9628     KOKKOS_INLINE_FUNCTION
```

```
9629     T& operator()(size_t i, size_t j) const;
9630
9631     KOKKOS_INLINE_FUNCTION
9632     T& operator()(size_t i, size_t j , size_t k) const;
9633
9634     KOKKOS_INLINE_FUNCTION
9635     T& operator()(size_t i, size_t j, size_t k , size_t l) const;
9636
9637     KOKKOS_INLINE_FUNCTION
9638     T& operator()(size_t i, size_t j, size_t k, size_t l, size_t m) const;
9639
9640     KOKKOS_INLINE_FUNCTION
9641     T& operator()(size_t i, size_t j, size_t k, size_t l, size_t m, size_t n) const;
9642
9643     KOKKOS_INLINE_FUNCTION
9644     T& operator()(size_t i, size_t j, size_t k, size_t l, size_t m, size_t n, size_t o) const;
9645
9646     KOKKOS_INLINE_FUNCTION
9647     size_t size() const;
9648
9649     KOKKOS_INLINE_FUNCTION
9650     size_t extent() const;
9651
9652     KOKKOS_INLINE_FUNCTION
9653     size_t dims(size_t i) const;
9654
9655     KOKKOS_INLINE_FUNCTION
9656     size_t order() const;
9657
9658     KOKKOS_INLINE_FUNCTION
9659     T* pointer() const;
9660
9661     KOKKOS_INLINE_FUNCTION
9662     ~ViewCMatrixKokkos();
9663
9664 }; // End of ViewCMatrixKokkos
9665
9666 // Default constructor
9667 template <typename T>
9668 KOKKOS_INLINE_FUNCTION
9669 ViewCMatrixKokkos<T>::ViewCMatrixKokkos(){ }
9670
9671 // Overloaded 1D constructor
9672 template <typename T>
9673 KOKKOS_INLINE_FUNCTION
9674 ViewCMatrixKokkos<T>::ViewCMatrixKokkos(T* some_matrix, size_t dim1) {
9675     dims_[0] = dim1;
9676     order_ = 1;
9677     length_ = dim1;
9678     this_matrix_ = some_matrix;
9679 }
9680
9681 // Overloaded 2D constructor
9682 template <typename T>
9683 KOKKOS_INLINE_FUNCTION
9684 ViewCMatrixKokkos<T>::ViewCMatrixKokkos(T* some_matrix, size_t dim1,
9685                                         size_t dim2) {
9686     dims_[0] = dim1;
9687     dims_[1] = dim2;
9688     order_ = 2;
9689     length_ = (dim1 * dim2);
9690     this_matrix_ = some_matrix;
9691 }
9692
9693 // Overloaded 3D constructor
9694 template <typename T>
9695 KOKKOS_INLINE_FUNCTION
9696 ViewCMatrixKokkos<T>::ViewCMatrixKokkos(T* some_matrix, size_t dim1, size_t dim2,
9697                                         size_t dim3) {
9698     dims_[0] = dim1;
9699     dims_[1] = dim2;
9700     dims_[2] = dim3;
9701     order_ = 3;
9702     length_ = (dim1 * dim2 * dim3);
9703     this_matrix_ = some_matrix;
9704 }
9705
9706 // Overloaded 4D constructor
9707 template <typename T>
9708 KOKKOS_INLINE_FUNCTION
9709 ViewCMatrixKokkos<T>::ViewCMatrixKokkos(T* some_matrix, size_t dim1, size_t dim2,
9710                                         size_t dim3, size_t dim4) {
9711     dims_[0] = dim1;
9712     dims_[1] = dim2;
9713     dims_[2] = dim3;
9714     dims_[3] = dim4;
9715     order_ = 4;
```

```
9716     length_ = (dim1 * dim2 * dim3 * dim4);
9717     this_matrix_ = some_matrix;
9718 }
9719
9720 // Overloaded 5D constructor
9721 template <typename T>
9722 KOKKOS_INLINE_FUNCTION
9723 ViewCMatrixKokkos<T>::ViewCMatrixKokkos(T* some_matrix, size_t dim1, size_t dim2,
9724                                         size_t dim3, size_t dim4, size_t dim5) {
9725     dims_[0] = dim1;
9726     dims_[1] = dim2;
9727     dims_[2] = dim3;
9728     dims_[3] = dim4;
9729     dims_[4] = dim5;
9730     order_ = 5;
9731     length_ = (dim1 * dim2 * dim3 * dim4 * dim5);
9732     this_matrix_ = some_matrix;
9733 }
9734
9735 // Overloaded 6D constructor
9736 template <typename T>
9737 KOKKOS_INLINE_FUNCTION
9738 ViewCMatrixKokkos<T>::ViewCMatrixKokkos(T* some_matrix, size_t dim1, size_t dim2,
9739                                         size_t dim3, size_t dim4, size_t dim5,
9740                                         size_t dim6) {
9741     dims_[0] = dim1;
9742     dims_[1] = dim2;
9743     dims_[2] = dim3;
9744     dims_[3] = dim4;
9745     dims_[4] = dim5;
9746     dims_[5] = dim6;
9747     order_ = 6;
9748     length_ = (dim1 * dim2 * dim3 * dim4 * dim5 * dim6);
9749     this_matrix_ = some_matrix;
9750 }
9751
9752 // Overloaded 7D constructor
9753 template <typename T>
9754 KOKKOS_INLINE_FUNCTION
9755 ViewCMatrixKokkos<T>::ViewCMatrixKokkos(T* some_matrix, size_t dim1, size_t dim2,
9756                                         size_t dim3, size_t dim4, size_t dim5,
9757                                         size_t dim6, size_t dim7) {
9758     dims_[0] = dim1;
9759     dims_[1] = dim2;
9760     dims_[2] = dim3;
9761     dims_[3] = dim4;
9762     dims_[4] = dim5;
9763     dims_[5] = dim6;
9764     dims_[6] = dim7;
9765     order_ = 7;
9766     length_ = (dim1 * dim2 * dim3 * dim4 * dim5 * dim6 * dim7);
9767     this_matrix_ = some_matrix;
9768 }
9769
9770 template <typename T>
9771 KOKKOS_INLINE_FUNCTION
9772 T& ViewCMatrixKokkos<T>::operator()(size_t i) const {
9773     assert(order_ == 1 && "Tensor order (rank) does not match constructor in ViewCMatrixKokkos 1D!");
9774     assert(i >= 1 && i <= dims_[0] && "i is out of bounds in ViewCMatrixKokkos 1D!");
9775     return this_matrix_[(i - 1)];
9776 }
9777
9778 template <typename T>
9779 KOKKOS_INLINE_FUNCTION
9780 T& ViewCMatrixKokkos<T>::operator()(size_t i, size_t j) const {
9781     assert(order_ == 2 && "Tensor order (rank) does not match constructor in ViewCMatrixKokkos 2D!");
9782     assert(i >= 1 && i <= dims_[0] && "i is out of bounds in ViewCMatrixKokkos 2D!");
9783     assert(j >= 1 && j <= dims_[1] && "j is out of bounds in ViewCMatrixKokkos 2D!");
9784     return this_matrix_[(j - 1) + ((i - 1) * dims_[1])];
9785 }
9786
9787 template <typename T>
9788 KOKKOS_INLINE_FUNCTION
9789 T& ViewCMatrixKokkos<T>::operator()(size_t i, size_t j, size_t k) const {
9790     assert(order_ == 3 && "Tensor order (rank) does not match constructor in ViewCMatrixKokkos 3D!");
9791     assert(i >= 1 && i <= dims_[0] && "i is out of bounds in ViewCMatrixKokkos 3D!");
9792     assert(j >= 1 && j <= dims_[1] && "j is out of bounds in ViewCMatrixKokkos 3D!");
9793     assert(k >= 1 && k <= dims_[2] && "k is out of bounds in ViewCMatrixKokkos 3D!");
9794     return this_matrix_[(k - 1) + ((j - 1) * dims_[2])
9795                                 + ((i - 1) * dims_[2] * dims_[1])];
9796 }
9797
9798 template <typename T>
9799 KOKKOS_INLINE_FUNCTION
9800 T& ViewCMatrixKokkos<T>::operator()(size_t i, size_t j , size_t k, size_t l) const {
9801     assert(order_ == 4 && "Tensor order (rank) does not match constructor in ViewCMatrixKokkos 4D!");
9802     assert(i >= 1 && i <= dims_[0] && "i is out of bounds in ViewCMatrixKokkos 4D!");
```

```
9803     assert(j >= 1 && j <= dims_[1] && "j is out of bounds in ViewCMatrixKokkos 4D!");
9804     assert(k >= 1 && k <= dims_[2] && "k is out of bounds in ViewCMatrixKokkos 4D!");
9805     assert(l >= 1 && l <= dims_[3] && "l is out of bounds in ViewCMatrixKokkos 4D!");
9806     return this_matrix_[(l - 1) + ((k - 1) * dims_[3])
9807                                 + ((j - 1) * dims_[3] * dims_[2])
9808                                 + ((i - 1) * dims_[3] * dims_[2] * dims_[1])];
9809 }
9810
9811 template <typename T>
9812 KOKKOS_INLINE_FUNCTION
9813 T& ViewCMatrixKokkos<T>::operator()(size_t i, size_t j, size_t k, size_t l,
9814                                     size_t m) const {
9815     assert(order_ == 5 && "Tensor order (rank) does not match constructor in ViewCMatrixKokkos 5D!");
9816     assert(i >= 1 && i <= dims_[0] && "i is out of bounds for ViewCMatrixKokkos 5D!");
9817     assert(j >= 1 && j <= dims_[1] && "j is out of bounds for ViewCMatrixKokkos 5D!");
9818     assert(k >= 1 && k <= dims_[2] && "k is out of bounds for ViewCMatrixKokkos 5D!");
9819     assert(l >= 1 && l <= dims_[3] && "l is out of bounds for ViewCMatrixKokkos 5D!");
9820     assert(m >= 1 && m <= dims_[4] && "m is out of bounds for ViewCMatrixKokkos 5D!");
9821     return this_matrix_[(m - 1) + ((l - 1) * dims_[4])
9822                                 + ((k - 1) * dims_[4] * dims_[3])
9823                                 + ((j - 1) * dims_[4] * dims_[3] * dims_[2])
9824                                 + ((i - 1) * dims_[4] * dims_[3] * dims_[2] * dims_[1])];
9825 }
9826
9827 template <typename T>
9828 KOKKOS_INLINE_FUNCTION
9829 T& ViewCMatrixKokkos<T>::operator()(size_t i, size_t j, size_t k, size_t l,
9830                                     size_t m, size_t n) const {
9831     assert(order_ == 6 && "Tensor order (rank) does not match constructor in ViewCMatrixKokkos 6D!");
9832     assert(i >= 1 && i <= dims_[0] && "i is out of bounds for ViewCMatrixKokkos 6D!");
9833     assert(j >= 1 && j <= dims_[1] && "j is out of bounds for ViewCMatrixKokkos 6D!");
9834     assert(k >= 1 && k <= dims_[2] && "k is out of bounds for ViewCMatrixKokkos 6D!");
9835     assert(l >= 1 && l <= dims_[3] && "l is out of bounds for ViewCMatrixKokkos 6D!");
9836     assert(m >= 1 && m <= dims_[4] && "m is out of bounds for ViewCMatrixKokkos 6D!");
9837     assert(n >= 1 && n <= dims_[5] && "n is out of bounds for ViewCMatrixKokkos 6D!");
9838     return this_matrix_[(n - 1) + ((m - 1) * dims_[5])
9839                                 + ((l - 1) * dims_[5] * dims_[4])
9840                                 + ((k - 1) * dims_[5] * dims_[4] * dims_[3])
9841                                 + ((j - 1) * dims_[5] * dims_[4] * dims_[3] * dims_[2])
9842                                 + ((i - 1) * dims_[5] * dims_[4] * dims_[3] * dims_[2] * dims_[1])];
9843 }
9844
9845 template <typename T>
9846 KOKKOS_INLINE_FUNCTION
9847 T& ViewCMatrixKokkos<T>::operator()(size_t i, size_t j, size_t k, size_t l,
9848                                     size_t m, size_t n, size_t o) const {
9849     assert(order_ == 7 && "Tensor order (rank) does not match constructor in ViewCMatrixKokkos 7D!");
9850     assert(i >= 1 && i <= dims_[0] && "i is out of bounds for ViewCMatrixKokkos 7D!");
9851     assert(j >= 1 && j <= dims_[1] && "j is out of bounds for ViewCMatrixKokkos 7D!");
9852     assert(k >= 1 && k <= dims_[2] && "k is out of bounds for ViewCMatrixKokkos 7D!");
9853     assert(l >= 1 && l <= dims_[3] && "l is out of bounds for ViewCMatrixKokkos 7D!");
9854     assert(m >= 1 && m <= dims_[4] && "m is out of bounds for ViewCMatrixKokkos 7D!");
9855     assert(n >= 1 && n <= dims_[5] && "n is out of bounds for ViewCMatrixKokkos 7D!");
9856     assert(o >= 1 && o <= dims_[6] && "o is out of bounds for ViewCMatrixKokkos 7D!");
9857     return this_matrix_[o + ((n - 1) * dims_[6])
9858                           + ((m - 1) * dims_[6] * dims_[5])
9859                           + ((l - 1) * dims_[6] * dims_[5] * dims_[4])
9860                           + ((k - 1) * dims_[6] * dims_[5] * dims_[4] * dims_[3])
9861                           + ((j - 1) * dims_[6] * dims_[5] * dims_[4] * dims_[3] * dims_[2])
9862                           + ((i - 1) * dims_[6] * dims_[5] * dims_[4] * dims_[3] * dims_[2] *
      dims_[1])];
9863 }
9864
9865
9866 template <typename T>
9867 KOKKOS_INLINE_FUNCTION
9868 size_t ViewCMatrixKokkos<T>::size() const {
9869     return length_;
9870 }
9871
9872 template <typename T>
9873 KOKKOS_INLINE_FUNCTION
9874 size_t ViewCMatrixKokkos<T>::extent() const {
9875     return length_;
9876 }
9877
9878 template <typename T>
9879 KOKKOS_INLINE_FUNCTION
9880 size_t ViewCMatrixKokkos<T>::dims(size_t i) const {
9881     i--;
9882     assert(i < order_ && "ViewCMatrixKokkos order (rank) does not match constructor, dim[i] does not
    exist!");
9883     assert(i >= 0 && dims_[i]>0 && "Access to ViewCMatrixKokkos dims is out of bounds!");
9884     return dims_[i];
9885 }
9886
9887 template <typename T>
```

```
9888 KOKKOS_INLINE_FUNCTION
9889 size_t ViewCMatrixKokkos<T>::order() const {
9890     return order_;
9891 }
9892
9893 template <typename T>
9894 KOKKOS_INLINE_FUNCTION
9895 T* ViewCMatrixKokkos<T>::pointer() const {
9896     return this_matrix_;
9897 }
9898
9899 template <typename T>
9900 KOKKOS_INLINE_FUNCTION
9901 ViewCMatrixKokkos<T>::~ViewCMatrixKokkos() {}
9902
9904 // End of ViewCMatrixKokkos
9906
9908 // DCArrayKokkos:  Dual type for managing data on both CPU and GPU.
9910 template <typename T, typename Layout = DefaultLayout, typename ExecSpace = DefaultExecSpace, typename
       MemoryTraits = void>
9911 class DCArrayKokkos {
9912
9913     // this is manage
9914     using TArray1D = Kokkos::DualView <T*, Layout, ExecSpace, MemoryTraits>;
9915
9916 private:
9917     size_t dims_[7];
9918     size_t length_;
9919     size_t order_;  // tensor order (rank)
9920     TArray1D this_array_;
9921
9922 public:
9923     DCArrayKokkos();
9924
9925     DCArrayKokkos(size_t dim0, const std::string& tag_string = DEFAULTSTRINGARRAY);
9926
9927     DCArrayKokkos(size_t dim0, size_t dim1, const std::string& tag_string = DEFAULTSTRINGARRAY);
9928
9929     DCArrayKokkos (size_t dim0, size_t dim1, size_t dim2, const std::string& tag_string =
       DEFAULTSTRINGARRAY);
9930
9931     DCArrayKokkos(size_t dim0, size_t dim1, size_t dim2,
9932                   size_t dim3, const std::string& tag_string = DEFAULTSTRINGARRAY);
9933
9934     DCArrayKokkos(size_t dim0, size_t dim1, size_t dim2,
9935                   size_t dim3, size_t dim4, const std::string& tag_string = DEFAULTSTRINGARRAY);
9936
9937     DCArrayKokkos(size_t dim0, size_t dim1, size_t dim2,
9938                   size_t dim3, size_t dim4, size_t dim5, const std::string& tag_string =
       DEFAULTSTRINGARRAY);
9939
9940     DCArrayKokkos(size_t dim0, size_t dim1, size_t dim2,
9941                   size_t dim3, size_t dim4, size_t dim5,
9942                   size_t dim6, const std::string& tag_string = DEFAULTSTRINGARRAY);
9943
9944     KOKKOS_INLINE_FUNCTION
9945     T& operator()(size_t i) const;
9946
9947     KOKKOS_INLINE_FUNCTION
9948     T& operator()(size_t i, size_t j) const;
9949
9950     KOKKOS_INLINE_FUNCTION
9951     T& operator()(size_t i, size_t j, size_t k) const;
9952
9953     KOKKOS_INLINE_FUNCTION
9954     T& operator()(size_t i, size_t j, size_t k, size_t l) const;
9955
9956     KOKKOS_INLINE_FUNCTION
9957     T& operator()(size_t i, size_t j, size_t k, size_t l, size_t m) const;
9958
9959     KOKKOS_INLINE_FUNCTION
9960     T& operator()(size_t i, size_t j, size_t k, size_t l, size_t m,
9961                   size_t n) const;
9962
9963     KOKKOS_INLINE_FUNCTION
9964     T& operator()(size_t i, size_t j, size_t k, size_t l, size_t m,
9965                   size_t n, size_t o) const;
9966
9967     KOKKOS_INLINE_FUNCTION
9968     DCArrayKokkos& operator=(const DCArrayKokkos& temp);
9969
9970     // GPU Method
9971     // Method that returns size
9972     KOKKOS_INLINE_FUNCTION
9973     size_t size() const;
9974
9975     // Host Method
```

```
9976     // Method that returns size
9977     KOKKOS_INLINE_FUNCTION
9978     size_t extent() const;
9979 
9980     KOKKOS_INLINE_FUNCTION
9981     size_t dims(size_t i) const;
9982 
9983     KOKKOS_INLINE_FUNCTION
9984     size_t order() const;
9985 
9986     // Method returns the raw device pointer of the Kokkos DualView
9987     KOKKOS_INLINE_FUNCTION
9988     T* device_pointer() const;
9989 
9990     // Method returns the raw host pointer of the Kokkos DualView
9991     KOKKOS_INLINE_FUNCTION
9992     T* host_pointer() const;
9993 
9994     // Method returns kokkos dual view
9995     KOKKOS_INLINE_FUNCTION
9996     TArray1D get_kokkos_dual_view() const;
9997 
9998     // Data member to access host view
9999     ViewCArray <T> host;
10000 
10001      // Method that update host view
10002      void update_host();
10003 
10004      // Method that update device view
10005      void update_device();
10006 
10007      // Deconstructor
10008      KOKKOS_INLINE_FUNCTION
10009      ~DCArrayKokkos ();
10010 }; // End of DCArrayKokkos
10011 
10012 
10013 // Default constructor
10014 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
10015 DCArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::DCArrayKokkos() {}
10016 
10017 // Overloaded 1D constructor
10018 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
10019 DCArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::DCArrayKokkos(size_t dim0, const std::string&
       tag_string) {
10020 
10021     dims_[0] = dim0;
10022     order_ = 1;
10023     length_ = dim0;
10024     this_array_ = TArray1D(tag_string, length_);
10025     // Create host ViewCArray
10026     host = ViewCArray <T> (this_array_.h_view.data(), dim0);
10027 }
10028 
10029 // Overloaded 2D constructor
10030 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
10031 DCArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::DCArrayKokkos(size_t dim0, size_t dim1, const
       std::string& tag_string) {
10032 
10033     dims_[0] = dim0;
10034     dims_[1] = dim1;
10035     order_ = 2;
10036     length_ = (dim0 * dim1);
10037     this_array_ = TArray1D(tag_string, length_);
10038     // Create host ViewCArray
10039     host = ViewCArray <T> (this_array_.h_view.data(), dim0, dim1);
10040 }
10041 
10042 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
10043 DCArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::DCArrayKokkos(size_t dim0, size_t dim1,
10044                                size_t dim2, const std::string& tag_string) {
10045 
10046     dims_[0] = dim0;
10047     dims_[1] = dim1;
10048     dims_[2] = dim2;
10049     order_ = 3;
10050     length_ = (dim0 * dim1 * dim2);
10051     this_array_ = TArray1D(tag_string, length_);
10052     // Create host ViewCArray
10053     host = ViewCArray <T> (this_array_.h_view.data(), dim0, dim1, dim2);
10054 }
10055 
10056 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
10057 DCArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::DCArrayKokkos(size_t dim0, size_t dim1,
10058                                size_t dim2, size_t dim3, const std::string& tag_string) {
10059 
10060     dims_[0] = dim0;
```

```
10061     dims_[1] = dim1;
10062     dims_[2] = dim2;
10063     dims_[3] = dim3;
10064     order_ = 4;
10065     length_ = (dim0 * dim1 * dim2 * dim3);
10066     this_array_ = TArray1D(tag_string, length_);
10067     // Create host ViewCArray
10068     host = ViewCArray <T> (this_array_.h_view.data(), dim0, dim1, dim2, dim3);
10069 }
10070
10071 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
10072 DCArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::DCArrayKokkos(size_t dim0, size_t dim1,
10073                                size_t dim2, size_t dim3,
10074                                size_t dim4, const std::string& tag_string) {
10075
10076     dims_[0] = dim0;
10077     dims_[1] = dim1;
10078     dims_[2] = dim2;
10079     dims_[3] = dim3;
10080     dims_[4] = dim4;
10081     order_ = 5;
10082     length_ = (dim0 * dim1 * dim2 * dim3 * dim4);
10083     this_array_ = TArray1D(tag_string, length_);
10084     // Create host ViewCArray
10085     host = ViewCArray <T> (this_array_.h_view.data(), dim0, dim1, dim2, dim3, dim4);
10086 }
10087
10088 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
10089 DCArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::DCArrayKokkos(size_t dim0, size_t dim1,
10090                                size_t dim2, size_t dim3,
10091                                size_t dim4, size_t dim5, const std::string& tag_string) {
10092
10093     dims_[0] = dim0;
10094     dims_[1] = dim1;
10095     dims_[2] = dim2;
10096     dims_[3] = dim3;
10097     dims_[4] = dim4;
10098     dims_[5] = dim5;
10099     order_ = 6;
10100     length_ = (dim0 * dim1 * dim2 * dim3 * dim4 * dim5);
10101     this_array_ = TArray1D(tag_string, length_);
10102     // Create host ViewCArray
10103     host = ViewCArray <T> (this_array_.h_view.data(), dim0, dim1, dim2, dim3, dim4, dim5);
10104 }
10105
10106 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
10107 DCArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::DCArrayKokkos(size_t dim0, size_t dim1,
10108                                size_t dim2, size_t dim3,
10109                                size_t dim4, size_t dim5,
10110                                size_t dim6, const std::string& tag_string) {
10111
10112     dims_[0] = dim0;
10113     dims_[1] = dim1;
10114     dims_[2] = dim2;
10115     dims_[3] = dim3;
10116     dims_[4] = dim4;
10117     dims_[5] = dim5;
10118     dims_[6] = dim6;
10119     order_ = 7;
10120     length_ = (dim0 * dim1 * dim2 * dim3 * dim4 * dim5 * dim6);
10121     this_array_ = TArray1D(tag_string, length_);
10122     // Create host ViewCArray
10123     host = ViewCArray <T> (this_array_.h_view.data(), dim0, dim1, dim2, dim3, dim4, dim5, dim6);
10124 }
10125
10126 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
10127 KOKKOS_INLINE_FUNCTION
10128 T& DCArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::operator()(size_t i) const {
10129     assert(order_ == 1 && "Tensor order (rank) does not match constructor in DCArrayKokkos 1D!");
10130     assert(i >= 0 && i < dims_[0] && "i is out of bounds in DCArrayKokkos 1D!");
10131     return this_array_.d_view(i);
10132 }
10133
10134 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
10135 KOKKOS_INLINE_FUNCTION
10136 T& DCArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::operator()(size_t i, size_t j) const {
10137     assert(order_ == 2 && "Tensor order (rank) does not match constructor in DCArrayKokkos 2D!");
10138     assert(i >= 0 && i < dims_[0] && "i is out of bounds in DCArrayKokkos 2D!");
10139     assert(j >= 0 && j < dims_[1] && "j is out of bounds in DCArrayKokkos 2D!");
10140     return this_array_.d_view(j + (i * dims_[1]));
10141 }
10142
10143 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
10144 KOKKOS_INLINE_FUNCTION
10145 T& DCArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::operator()(size_t i, size_t j, size_t k) const {
10146     assert(order_ == 3 && "Tensor order (rank) does not match constructor in DCArrayKokkos 3D!");
10147     assert(i >= 0 && i < dims_[0] && "i is out of bounds in DCArrayKokkos 3D!");
```

```
10148     assert(j >= 0 && j < dims_[1] && "j is out of bounds in DCArrayKokkos 3D!");
10149     assert(k >= 0 && k < dims_[2] && "k is out of bounds in DCArrayKokkos 3D!");
10150     return this_array_.d_view(k + (j * dims_[2])
10151                               + (i * dims_[2] * dims_[1]));
10152 }
10153
10154 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
10155 KOKKOS_INLINE_FUNCTION
10156 T& DCArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::operator()(size_t i, size_t j, size_t k, size_t l)
       const {
10157     assert(order_ == 4 && "Tensor order (rank) does not match constructor in DCArrayKokkos 4D!");
10158     assert(i >= 0 && i < dims_[0] && "i is out of bounds in DCArrayKokkos 4D!");
10159     assert(j >= 0 && j < dims_[1] && "j is out of bounds in DCArrayKokkos 4D!");
10160     assert(k >= 0 && k < dims_[2] && "k is out of bounds in DCArrayKokkos 4D!");
10161     assert(l >= 0 && l < dims_[3] && "l is out of bounds in DCArrayKokkos 4D!");
10162     return this_array_.d_view(l + (k * dims_[3])
10163                               + (j * dims_[3] * dims_[2])
10164                               + (i * dims_[3] * dims_[2] * dims_[1]));
10165 }
10166
10167 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
10168 KOKKOS_INLINE_FUNCTION
10169 T& DCArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::operator()(size_t i, size_t j, size_t k, size_t l,
10170                                 size_t m) const {
10171     assert(order_ == 5 && "Tensor order (rank) does not match constructor in DCArrayKokkos 5D!");
10172     assert(i >= 0 && i < dims_[0] && "i is out of bounds in DCArrayKokkos 5D!");
10173     assert(j >= 0 && j < dims_[1] && "j is out of bounds in DCArrayKokkos 5D!");
10174     assert(k >= 0 && k < dims_[2] && "k is out of bounds in DCArrayKokkos 5D!");
10175     assert(l >= 0 && l < dims_[3] && "l is out of bounds in DCArrayKokkos 5D!");
10176     assert(m >= 0 && m < dims_[4] && "m is out of bounds in DCArrayKokkos 5D!");
10177     return this_array_.d_view(m + (l * dims_[4])
10178                               + (k * dims_[4] * dims_[3])
10179                               + (j * dims_[4] * dims_[3] * dims_[2])
10180                               + (i * dims_[4] * dims_[3] * dims_[2] * dims_[1]));
10181 }
10182
10183 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
10184 KOKKOS_INLINE_FUNCTION
10185 T& DCArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::operator()(size_t i, size_t j, size_t k, size_t l,
10186                                 size_t m, size_t n) const {
10187     assert(order_ == 6 && "Tensor order (rank) does not match constructor in DCArrayKokkos 6D!");
10188     assert(i >= 0 && i < dims_[0] && "i is out of bounds in DCArrayKokkos 6D!");
10189     assert(j >= 0 && j < dims_[1] && "j is out of bounds in DCArrayKokkos 6D!");
10190     assert(k >= 0 && k < dims_[2] && "k is out of bounds in DCArrayKokkos 6D!");
10191     assert(l >= 0 && l < dims_[3] && "l is out of bounds in DCArrayKokkos 6D!");
10192     assert(m >= 0 && m < dims_[4] && "m is out of bounds in DCArrayKokkos 6D!");
10193     assert(n >= 0 && n < dims_[5] && "n is out of bounds in DCArrayKokkos 6D!");
10194     return this_array_.d_view(n + (m * dims_[5])
10195                               + (l * dims_[5] * dims_[4])
10196                               + (k * dims_[5] * dims_[4] * dims_[3])
10197                               + (j * dims_[5] * dims_[4] * dims_[3] * dims_[2])
10198                               + (i * dims_[5] * dims_[4] * dims_[3] * dims_[2] * dims_[1]));
10199 }
10200
10201 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
10202 KOKKOS_INLINE_FUNCTION
10203 T& DCArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::operator()(size_t i, size_t j, size_t k, size_t l,
10204                                 size_t m, size_t n, size_t o) const {
10205     assert(order_ == 7 && "Tensor order (rank) does not match constructor in DCArrayKokkos 7D!");
10206     assert(i >= 0 && i < dims_[0] && "i is out of bounds in DCArrayKokkos 7D!");
10207     assert(j >= 0 && j < dims_[1] && "j is out of bounds in DCArrayKokkos 7D!");
10208     assert(k >= 0 && k < dims_[2] && "k is out of bounds in DCArrayKokkos 7D!");
10209     assert(l >= 0 && l < dims_[3] && "l is out of bounds in DCArrayKokkos 7D!");
10210     assert(m >= 0 && m < dims_[4] && "m is out of bounds in DCArrayKokkos 7D!");
10211     assert(n >= 0 && n < dims_[5] && "n is out of bounds in DCArrayKokkos 7D!");
10212     assert(o >= 0 && o < dims_[6] && "o is out of bounds in DCArrayKokkos 7D!");
10213     return this_array_.d_view(o + (n * dims_[6])
10214                               + (m * dims_[6] * dims_[5])
10215                               + (l * dims_[6] * dims_[5] * dims_[4])
10216                               + (k * dims_[6] * dims_[5] * dims_[4] * dims_[3])
10217                               + (j * dims_[6] * dims_[5] * dims_[4] * dims_[3] * dims_[2])
10218                               + (i * dims_[6] * dims_[5] * dims_[4] * dims_[3] * dims_[2] *
       dims_[1]));
10219 }
10220
10221 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
10222 KOKKOS_INLINE_FUNCTION
10223 DCArrayKokkos<T,Layout,ExecSpace,MemoryTraits>&
       DCArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::operator= (const DCArrayKokkos& temp) {
10224
10225     // Do nothing if the assignment is of the form x = x
10226     if (this != &temp) {
10227         for (int iter = 0; iter < temp.order_; iter++){
10228             dims_[iter] = temp.dims_[iter];
10229         } // end for
10230
10231         order_ = temp.order_;
```

```
10232         length_ = temp.length_;
10233         this_array_ = temp.this_array_;
10234         host = temp.host;
10235     }
10236 
10237     return *this;
10238 }
10239 
10240 // Return size
10241 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
10242 KOKKOS_INLINE_FUNCTION
10243 size_t DCArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::size() const {
10244     return length_;
10245 }
10246 
10247 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
10248 KOKKOS_INLINE_FUNCTION
10249 size_t DCArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::extent() const {
10250     return length_;
10251 }
10252 
10253 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
10254 KOKKOS_INLINE_FUNCTION
10255 size_t DCArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::dims(size_t i) const {
10256     assert(i < order_ && "DCArrayKokkos order (rank) does not match constructor, dim[i] does not
      exist!");
10257     assert(i >= 0 && dims_[i]>0 && "Access to DCArrayKokkos dims is out of bounds!");
10258     return dims_[i];
10259 }
10260 
10261 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
10262 KOKKOS_INLINE_FUNCTION
10263 size_t DCArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::order() const {
10264     return order_;
10265 }
10266 
10267 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
10268 KOKKOS_INLINE_FUNCTION
10269 T* DCArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::device_pointer() const {
10270     return this_array_.d_view.data();
10271 }
10272 
10273 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
10274 KOKKOS_INLINE_FUNCTION
10275 T* DCArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::host_pointer() const {
10276     return this_array_.h_view.data();
10277 }
10278 
10279 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
10280 KOKKOS_INLINE_FUNCTION
10281 Kokkos::DualView <T*, Layout, ExecSpace, MemoryTraits>
       DCArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::get_kokkos_dual_view() const {
10282   return this_array_;
10283 }
10284 
10285 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
10286 void DCArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::update_host() {
10287 
10288     this_array_.template modify<typename TArray1D::execution_space>();
10289     this_array_.template sync<typename TArray1D::host_mirror_space>();
10290 }
10291 
10292 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
10293 void DCArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::update_device() {
10294 
10295     this_array_.template modify<typename TArray1D::host_mirror_space>();
10296     this_array_.template sync<typename TArray1D::execution_space>();
10297 }
10298 
10299 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
10300 KOKKOS_INLINE_FUNCTION
10301 DCArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::~DCArrayKokkos() {}
10302 // End DCArrayKokkos
10303 
10304 
10306 // DViewCArrayKokkos
10308 template <typename T, typename Layout = DefaultLayout, typename ExecSpace = DefaultExecSpace, typename
       MemoryTraits = void>
10309 class DViewCArrayKokkos {
10310 
10311     // this is always unmanaged
10312     using TArray1DHost = Kokkos::View<T*, Layout, HostSpace, MemoryUnmanaged>;
10313     // this is manage
10314     using TArray1D     = Kokkos::View<T*, Layout, ExecSpace, MemoryTraits>;
10315 
10316 private:
10317     size_t dims_[7];
```

```
10318     size_t length_;
10319     size_t order_;  // tensor order (rank)
10320     TArray1D this_array_;
10321     TArray1DHost this_array_host_;
10322     T * temp_inp_array_;
10323     //typename Kokkos::View<T*, Layout, ExecSpace>::HostMirror  h_this_array_;
10324
10325 public:
10326     DViewCArrayKokkos();
10327
10328     DViewCArrayKokkos(T * inp_array, size_t dim0);
10329
10330     DViewCArrayKokkos(T * inp_array, size_t dim0, size_t dim1);
10331
10332     DViewCArrayKokkos(T * inp_array, size_t dim0, size_t dim1, size_t dim2);
10333
10334     DViewCArrayKokkos(T * inp_array, size_t dim0, size_t dim1, size_t dim2,
10335                 size_t dim3);
10336
10337     DViewCArrayKokkos(T * inp_array, size_t dim0, size_t dim1, size_t dim2,
10338                 size_t dim3, size_t dim4);
10339
10340     DViewCArrayKokkos(T * inp_array, size_t dim0, size_t dim1, size_t dim2,
10341                 size_t dim3, size_t dim4, size_t dim5);
10342
10343     DViewCArrayKokkos(T * inp_array, size_t dim0, size_t dim1, size_t dim2,
10344                 size_t dim3, size_t dim4, size_t dim5,
10345                 size_t dim6);
10346
10347     KOKKOS_INLINE_FUNCTION
10348     T& operator()(size_t i) const;
10349
10350     KOKKOS_INLINE_FUNCTION
10351     T& operator()(size_t i, size_t j) const;
10352
10353     KOKKOS_INLINE_FUNCTION
10354     T& operator()(size_t i, size_t j, size_t k) const;
10355
10356     KOKKOS_INLINE_FUNCTION
10357     T& operator()(size_t i, size_t j, size_t k, size_t l) const;
10358
10359     KOKKOS_INLINE_FUNCTION
10360     T& operator()(size_t i, size_t j, size_t k, size_t l, size_t m) const;
10361
10362     KOKKOS_INLINE_FUNCTION
10363     T& operator()(size_t i, size_t j, size_t k, size_t l, size_t m,
10364                   size_t n) const;
10365
10366     KOKKOS_INLINE_FUNCTION
10367     T& operator()(size_t i, size_t j, size_t k, size_t l, size_t m,
10368                   size_t n, size_t o) const;
10369
10370     KOKKOS_INLINE_FUNCTION
10371     DViewCArrayKokkos& operator=(const DViewCArrayKokkos& temp);
10372
10373     // GPU Method
10374     // Method that returns size
10375     KOKKOS_INLINE_FUNCTION
10376     size_t size() const;
10377
10378     // Host Method
10379     // Method that returns size
10380     KOKKOS_INLINE_FUNCTION
10381     size_t extent() const;
10382
10383     KOKKOS_INLINE_FUNCTION
10384     size_t dims(size_t i) const;
10385
10386     KOKKOS_INLINE_FUNCTION
10387     size_t order() const;
10388
10389     // Method returns the raw device pointer of the Kokkos View
10390     KOKKOS_INLINE_FUNCTION
10391     T* device_pointer() const;
10392
10393     // Method returns the raw host pointer of the Kokkos View
10394     KOKKOS_INLINE_FUNCTION
10395     T* host_pointer() const;
10396
10397     // Data member to access host view
10398     ViewCArray <T> host;
10399
10400     // Method that update host view
10401     void update_host();
10402
10403     // Method that update device view
10404     void update_device();
```

```
10405
10406     // Deconstructor
10407     KOKKOS_INLINE_FUNCTION
10408     ~DViewCArrayKokkos ();
10409 }; // End of DViewCArrayKokkos
10410
10411
10412 // Default constructor
10413 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
10414 DViewCArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::DViewCArrayKokkos() {}
10415
10416 // Overloaded 1D constructor
10417 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
10418 DViewCArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::DViewCArrayKokkos(T * inp_array, size_t dim0) {
10419     //using TArray1DHost = Kokkos::View<T*, Layout, HostSpace, MemoryUnmanaged>;
10420     //using TArray1D = Kokkos::View<T*, Layout, ExecSpace>;
10421
10422     dims_[0] = dim0;
10423     order_ = 1;
10424     length_ = dim0;
10425     // Create a 1D host view of the external allocation
10426     this_array_host_ = TArray1DHost(inp_array, length_);
10427     // Assign temp point to inp_array pointer that is passed in
10428     temp_inp_array_ = inp_array;
10429     // Create a device copy of that host view
10430     this_array_ = create_mirror_view_and_copy(ExecSpace(), this_array_host_);
10431     // Create host ViewCArray. Note: inp_array and this_array_host_.data() are the same pointer
10432     host = ViewCArray <T> (inp_array, dim0);
10433 }
10434
10435 // Overloaded 2D constructor
10436 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
10437 DViewCArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::DViewCArrayKokkos(T * inp_array, size_t dim0,
      size_t dim1) {
10438     //using TArray1DHost = Kokkos::View<T*, Layout, HostSpace, MemoryUnmanaged>;
10439     //using TArray1D = Kokkos::View<T*, Layout, ExecSpace>;
10440     //using TArray1Dtemp = TArray1D::HostMirror;
10441
10442     dims_[0] = dim0;
10443     dims_[1] = dim1;
10444     order_ = 2;
10445     length_ = (dim0 * dim1);
10446     // Create a 1D host view of the external allocation
10447     this_array_host_ = TArray1DHost(inp_array, length_);
10448     // Assign temp point to inp_array pointer that is passed in
10449     temp_inp_array_ = inp_array;
10450     // Create a device copy of that host view
10451     this_array_ = create_mirror_view_and_copy(ExecSpace(), this_array_host_);
10452     // Create host ViewCArray
10453     host = ViewCArray <T> (inp_array, dim0, dim1);
10454 }
10455
10456 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
10457 DViewCArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::DViewCArrayKokkos(T * inp_array, size_t dim0,
      size_t dim1,
10458                                 size_t dim2) {
10459     //using TArray1D = Kokkos::View<T*, Layout, ExecSpace>;
10460
10461     dims_[0] = dim0;
10462     dims_[1] = dim1;
10463     dims_[2] = dim2;
10464     order_ = 3;
10465     length_ = (dim0 * dim1 * dim2);
10466     // Create a 1D host view of the external allocation
10467     this_array_host_ = TArray1DHost(inp_array, length_);
10468     // Assign temp point to inp_array pointer that is passed in
10469     temp_inp_array_ = inp_array;
10470     // Create a device copy of that host view
10471     this_array_ = create_mirror_view_and_copy(ExecSpace(), this_array_host_);
10472     // Create host ViewCArray
10473     host = ViewCArray <T> (inp_array, dim0, dim1, dim2);
10474 }
10475
10476 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
10477 DViewCArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::DViewCArrayKokkos(T * inp_array, size_t dim0,
      size_t dim1,
10478                                 size_t dim2, size_t dim3) {
10479     //using TArray1D = Kokkos::View<T *,Layout,ExecSpace>;
10480
10481     dims_[0] = dim0;
10482     dims_[1] = dim1;
10483     dims_[2] = dim2;
10484     dims_[3] = dim3;
10485     order_ = 4;
10486     length_ = (dim0 * dim1 * dim2 * dim3);
10487     // Create a 1D host view of the external allocation
10488     this_array_host_ = TArray1DHost(inp_array, length_);
```

```
10489     // Assign temp point to inp_array pointer that is passed in
10490     temp_inp_array_ = inp_array;
10491     // Create a device copy of that host view
10492     this_array_ = create_mirror_view_and_copy(ExecSpace(), this_array_host_);
10493     // Create host ViewCArray
10494     host = ViewCArray <T> (inp_array, dim0, dim1, dim2, dim3);
10495 }
10496
10497 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
10498 DViewCArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::DViewCArrayKokkos(T * inp_array, size_t dim0,
      size_t dim1,
10499                               size_t dim2, size_t dim3,
10500                               size_t dim4) {
10501
10502     //using TArray1D = Kokkos::View<T *,Layout,ExecSpace>;
10503
10504     dims_[0] = dim0;
10505     dims_[1] = dim1;
10506     dims_[2] = dim2;
10507     dims_[3] = dim3;
10508     dims_[4] = dim4;
10509     order_ = 5;
10510     length_ = (dim0 * dim1 * dim2 * dim3 * dim4);
10511     // Create a 1D host view of the external allocation
10512     this_array_host_ = TArray1DHost(inp_array, length_);
10513     // Assign temp point to inp_array pointer that is passed in
10514     temp_inp_array_ = inp_array;
10515     // Create a device copy of that host view
10516     this_array_ = create_mirror_view_and_copy(ExecSpace(), this_array_host_);
10517     // Create host ViewCArray
10518     host = ViewCArray <T> (inp_array, dim0, dim1, dim2, dim3, dim4);
10519 }
10520
10521 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
10522 DViewCArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::DViewCArrayKokkos(T * inp_array, size_t dim0,
      size_t dim1,
10523                               size_t dim2, size_t dim3,
10524                               size_t dim4, size_t dim5) {
10525     //using TArray1D = Kokkos::View<T *,Layout,ExecSpace>;
10526
10527     dims_[0] = dim0;
10528     dims_[1] = dim1;
10529     dims_[2] = dim2;
10530     dims_[3] = dim3;
10531     dims_[4] = dim4;
10532     dims_[5] = dim5;
10533     order_ = 6;
10534     length_ = (dim0 * dim1 * dim2 * dim3 * dim4 * dim5);
10535     // Create a 1D host view of the external allocation
10536     this_array_host_ = TArray1DHost(inp_array, length_);
10537     // Assign temp point to inp_array pointer that is passed in
10538     temp_inp_array_ = inp_array;
10539     // Create a device copy of that host view
10540     this_array_ = create_mirror_view_and_copy(ExecSpace(), this_array_host_);
10541     // Create host ViewCArray
10542     host = ViewCArray <T> (inp_array, dim0, dim1, dim2, dim3, dim4, dim5);
10543 }
10544
10545 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
10546 DViewCArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::DViewCArrayKokkos(T * inp_array, size_t dim0,
      size_t dim1,
10547                               size_t dim2, size_t dim3,
10548                               size_t dim4, size_t dim5,
10549                               size_t dim6) {
10550     //using TArray1D = Kokkos::View<T *,Layout,ExecSpace>;
10551
10552     dims_[0] = dim0;
10553     dims_[1] = dim1;
10554     dims_[2] = dim2;
10555     dims_[3] = dim3;
10556     dims_[4] = dim4;
10557     dims_[5] = dim5;
10558     dims_[6] = dim6;
10559     order_ = 7;
10560     length_ = (dim0 * dim1 * dim2 * dim3 * dim4 * dim5 * dim6);
10561     // Create a 1D host view of the external allocation
10562     this_array_host_ = TArray1DHost(inp_array, length_);
10563     // Assign temp point to inp_array pointer that is passed in
10564     temp_inp_array_ = inp_array;
10565     // Create a device copy of that host view
10566     this_array_ = create_mirror_view_and_copy(ExecSpace(), this_array_host_);
10567     // Create host ViewCArray
10568     host = ViewCArray <T> (inp_array, dim0, dim1, dim2, dim3, dim4, dim5, dim6);
10569 }
10570
10571 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
10572 KOKKOS_INLINE_FUNCTION
```

```
10573 T& DViewCArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::operator()(size_t i) const {
10574     assert(order_ == 1 && "Tensor order (rank) does not match constructor in DViewCArrayKokkos 1D!");
10575     assert(i >= 0 && i < dims_[0] && "i is out of bounds in DViewCArrayKokkos 1D!");
10576     return this_array_(i);
10577 }
10578
10579 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
10580 KOKKOS_INLINE_FUNCTION
10581 T& DViewCArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::operator()(size_t i, size_t j) const {
10582     assert(order_ == 2 && "Tensor order (rank) does not match constructor in DViewCArrayKokkos 2D!");
10583     assert(i >= 0 && i < dims_[0] && "i is out of bounds in DViewCArrayKokkos 2D!");
10584     assert(j >= 0 && j < dims_[1] && "j is out of bounds in DViewCArrayKokkos 2D!");
10585     return this_array_(j + (i * dims_[1]));
10586 }
10587
10588 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
10589 KOKKOS_INLINE_FUNCTION
10590 T& DViewCArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::operator()(size_t i, size_t j, size_t k) const
       {
10591     assert(order_ == 3 && "Tensor order (rank) does not match constructor in DViewCArrayKokkos 3D!");
10592     assert(i >= 0 && i < dims_[0] && "i is out of bounds in DViewCArrayKokkos 3D!");
10593     assert(j >= 0 && j < dims_[1] && "j is out of bounds in DViewCArrayKokkos 3D!");
10594     assert(k >= 0 && k < dims_[2] && "k is out of bounds in DViewCArrayKokkos 3D!");
10595     return this_array_(k + (j * dims_[2])
10596                          + (i * dims_[2] * dims_[1]));
10597 }
10598
10599 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
10600 KOKKOS_INLINE_FUNCTION
10601 T& DViewCArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::operator()(size_t i, size_t j, size_t k, size_t
       l) const {
10602     assert(order_ == 4 && "Tensor order (rank) does not match constructor in DViewCArrayKokkos 4D!");
10603     assert(i >= 0 && i < dims_[0] && "i is out of bounds in DViewCArrayKokkos 4D!");
10604     assert(j >= 0 && j < dims_[1] && "j is out of bounds in DViewCArrayKokkos 4D!");
10605     assert(k >= 0 && k < dims_[2] && "k is out of bounds in DViewCArrayKokkos 4D!");
10606     assert(l >= 0 && l < dims_[3] && "l is out of bounds in DViewCArrayKokkos 4D!");
10607     return this_array_(l + (k * dims_[3])
10608                          + (j * dims_[3] * dims_[2])
10609                          + (i * dims_[3] * dims_[2] * dims_[1]));
10610 }
10611
10612 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
10613 KOKKOS_INLINE_FUNCTION
10614 T& DViewCArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::operator()(size_t i, size_t j, size_t k, size_t
       l,
10615                                size_t m) const {
10616     assert(order_ == 5 && "Tensor order (rank) does not match constructor in DViewCArrayKokkos 5D!");
10617     assert(i >= 0 && i < dims_[0] && "i is out of bounds in DViewCArrayKokkos 5D!");
10618     assert(j >= 0 && j < dims_[1] && "j is out of bounds in DViewCArrayKokkos 5D!");
10619     assert(k >= 0 && k < dims_[2] && "k is out of bounds in DViewCArrayKokkos 5D!");
10620     assert(l >= 0 && l < dims_[3] && "l is out of bounds in DViewCArrayKokkos 5D!");
10621     assert(m >= 0 && m < dims_[4] && "m is out of bounds in DViewCArrayKokkos 5D!");
10622     return this_array_(m + (l * dims_[4])
10623                          + (k * dims_[4] * dims_[3])
10624                          + (j * dims_[4] * dims_[3] * dims_[2])
10625                          + (i * dims_[4] * dims_[3] * dims_[2] * dims_[1]));
10626 }
10627
10628 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
10629 KOKKOS_INLINE_FUNCTION
10630 T& DViewCArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::operator()(size_t i, size_t j, size_t k, size_t
       l,
10631                                size_t m, size_t n) const {
10632     assert(order_ == 6 && "Tensor order (rank) does not match constructor in DViewCArrayKokkos 6D!");
10633     assert(i >= 0 && i < dims_[0] && "i is out of bounds in DViewCArrayKokkos 6D!");
10634     assert(j >= 0 && j < dims_[1] && "j is out of bounds in DViewCArrayKokkos 6D!");
10635     assert(k >= 0 && k < dims_[2] && "k is out of bounds in DViewCArrayKokkos 6D!");
10636     assert(l >= 0 && l < dims_[3] && "l is out of bounds in DViewCArrayKokkos 6D!");
10637     assert(m >= 0 && m < dims_[4] && "m is out of bounds in DViewCArrayKokkos 6D!");
10638     assert(n >= 0 && n < dims_[5] && "n is out of bounds in DViewCArrayKokkos 6D!");
10639     return this_array_(n + (m * dims_[5])
10640                          + (l * dims_[5] * dims_[4])
10641                          + (k * dims_[5] * dims_[4] * dims_[3])
10642                          + (j * dims_[5] * dims_[4] * dims_[3] * dims_[2])
10643                          + (i * dims_[5] * dims_[4] * dims_[3] * dims_[2] * dims_[1]));
10644 }
10645
10646 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
10647 KOKKOS_INLINE_FUNCTION
10648 T& DViewCArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::operator()(size_t i, size_t j, size_t k, size_t
       l,
10649                                size_t m, size_t n, size_t o) const {
10650     assert(order_ == 7 && "Tensor order (rank) does not match constructor in DViewCArrayKokkos 7D!");
10651     assert(i >= 0 && i < dims_[0] && "i is out of bounds in DViewCArrayKokkos 7D!");
10652     assert(j >= 0 && j < dims_[1] && "j is out of bounds in DViewCArrayKokkos 7D!");
10653     assert(k >= 0 && k < dims_[2] && "k is out of bounds in DViewCArrayKokkos 7D!");
10654     assert(l >= 0 && l < dims_[3] && "l is out of bounds in DViewCArrayKokkos 7D!");
```

```
10655     assert(m >= 0 && m < dims_[4] && "m is out of bounds in DViewCArrayKokkos 7D!");
10656     assert(n >= 0 && n < dims_[5] && "n is out of bounds in DViewCArrayKokkos 7D!");
10657     assert(o >= 0 && o < dims_[6] && "o is out of bounds in DViewCArrayKokkos 7D!");
10658     return this_array_(o + (n * dims_[6])
10659                         + (m * dims_[6] * dims_[5])
10660                         + (l * dims_[6] * dims_[5] * dims_[4])
10661                         + (k * dims_[6] * dims_[5] * dims_[4] * dims_[3])
10662                         + (j * dims_[6] * dims_[5] * dims_[4] * dims_[3] * dims_[2])
10663                         + (i * dims_[6] * dims_[5] * dims_[4] * dims_[3] * dims_[2] * dims_[1]));
10664 }
10665
10666 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
10667 KOKKOS_INLINE_FUNCTION
10668 DViewCArrayKokkos<T,Layout,ExecSpace,MemoryTraits>&
       DViewCArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::operator= (const DViewCArrayKokkos& temp) {
10669     //using TArray1D = Kokkos::View<T *,Layout,ExecSpace>;
10670
10671     // Do nothing if the assignment is of the form x = x
10672     if (this != &temp) {
10673         for (int iter = 0; iter < temp.order_; iter++){
10674             dims_[iter] = temp.dims_[iter];
10675         } // end for
10676
10677         order_ = temp.order_;
10678         length_ = temp.length_;
10679         temp_inp_array_ = temp.temp_inp_array_;
10680         this_array_host_ = temp.this_array_host_;
10681         this_array_ = temp.this_array_;
10682         host = temp.host;
10683     }
10684
10685     return *this;
10686 }
10687
10688 // Return size
10689 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
10690 KOKKOS_INLINE_FUNCTION
10691 size_t DViewCArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::size() const {
10692     return length_;
10693 }
10694
10695 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
10696 KOKKOS_INLINE_FUNCTION
10697 size_t DViewCArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::extent() const {
10698     return length_;
10699 }
10700
10701 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
10702 KOKKOS_INLINE_FUNCTION
10703 size_t DViewCArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::dims(size_t i) const {
10704     assert(i < order_ && "DViewCArrayKokkos order (rank) does not match constructor, dim[i] does not
       exist!");
10705     assert(i >= 0 && dims_[i]>0 && "Access to DViewCArrayKokkos dims is out of bounds!");
10706     return dims_[i];
10707 }
10708
10709 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
10710 KOKKOS_INLINE_FUNCTION
10711 size_t DViewCArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::order() const {
10712     return order_;
10713 }
10714
10715 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
10716 KOKKOS_INLINE_FUNCTION
10717 T* DViewCArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::device_pointer() const {
10718     return this_array_.data();
10719 }
10720
10721 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
10722 KOKKOS_INLINE_FUNCTION
10723 T* DViewCArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::host_pointer() const {
10724     return this_array_host_.data();
10725 }
10726
10727 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
10728 void DViewCArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::update_host() {
10729     // Deep copy of device view to host view
10730     deep_copy(this_array_host_, this_array_);
10731 }
10732
10733 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
10734 void DViewCArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::update_device() {
10735     // Deep copy of host view to device view
10736     deep_copy(this_array_, this_array_host_);
10737 }
10738
10739 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
```

```
10740 KOKKOS_INLINE_FUNCTION
10741 DViewCArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::~DViewCArrayKokkos() {}
10742 // End DViewCArrayKokkos
10743
10744
10746 // DCMatrixKokkos
10748 template <typename T, typename Layout = DefaultLayout, typename ExecSpace = DefaultExecSpace, typename
       MemoryTraits = void>
10749 class DCMatrixKokkos {
10750
10751     // this is manage
10752     using TArray1D = Kokkos::DualView<T*, Layout, ExecSpace, MemoryTraits>;
10753
10754 private:
10755     size_t dims_[7];
10756     size_t length_;
10757     size_t order_;  // tensor order (rank)
10758     TArray1D this_matrix_;
10759
10760 public:
10761     DCMatrixKokkos();
10762
10763     DCMatrixKokkos(size_t dim1, const std::string& tag_string = DEFAULTSTRINGMATRIX);
10764
10765     DCMatrixKokkos(size_t dim1, size_t dim2, const std::string& tag_string = DEFAULTSTRINGMATRIX);
10766
10767     DCMatrixKokkos (size_t dim1, size_t dim2, size_t dim3, const std::string& tag_string =
       DEFAULTSTRINGMATRIX);
10768
10769     DCMatrixKokkos(size_t dim1, size_t dim2, size_t dim3,
10770                    size_t dim4, const std::string& tag_string = DEFAULTSTRINGMATRIX);
10771
10772     DCMatrixKokkos(size_t dim1, size_t dim2, size_t dim3,
10773                    size_t dim4, size_t dim5, const std::string& tag_string = DEFAULTSTRINGMATRIX);
10774
10775     DCMatrixKokkos(size_t dim1, size_t dim2, size_t dim3,
10776                    size_t dim4, size_t dim5, size_t dim6, const std::string& tag_string =
       DEFAULTSTRINGMATRIX);
10777
10778     DCMatrixKokkos(size_t dim1, size_t dim2, size_t dim3,
10779                    size_t dim4, size_t dim5, size_t dim6,
10780                    size_t dim7, const std::string& tag_string = DEFAULTSTRINGMATRIX);
10781
10782     KOKKOS_INLINE_FUNCTION
10783     T& operator()(size_t i) const;
10784
10785     KOKKOS_INLINE_FUNCTION
10786     T& operator()(size_t i, size_t j) const;
10787
10788     KOKKOS_INLINE_FUNCTION
10789     T& operator()(size_t i, size_t j, size_t k) const;
10790
10791     KOKKOS_INLINE_FUNCTION
10792     T& operator()(size_t i, size_t j, size_t k, size_t l) const;
10793
10794     KOKKOS_INLINE_FUNCTION
10795     T& operator()(size_t i, size_t j, size_t k, size_t l, size_t m) const;
10796
10797     KOKKOS_INLINE_FUNCTION
10798     T& operator()(size_t i, size_t j, size_t k, size_t l, size_t m,
10799                   size_t n) const;
10800
10801     KOKKOS_INLINE_FUNCTION
10802     T& operator()(size_t i, size_t j, size_t k, size_t l, size_t m,
10803                   size_t n, size_t o) const;
10804
10805     KOKKOS_INLINE_FUNCTION
10806     DCMatrixKokkos& operator=(const DCMatrixKokkos& temp);
10807
10808     // GPU Method
10809     // Method that returns size
10810     KOKKOS_INLINE_FUNCTION
10811     size_t size() const;
10812
10813     // Host Method
10814     // Method that returns size
10815     KOKKOS_INLINE_FUNCTION
10816     size_t extent() const;
10817
10818     KOKKOS_INLINE_FUNCTION
10819     size_t dims(size_t i) const;
10820
10821     KOKKOS_INLINE_FUNCTION
10822     size_t order() const;
10823
10824     // Method returns the raw device pointer of the Kokkos DualView
10825     KOKKOS_INLINE_FUNCTION
```

```
10826     T* device_pointer() const;
10827
10828     // Method returns the raw host pointer of the Kokkos DualView
10829     KOKKOS_INLINE_FUNCTION
10830     T* host_pointer() const;
10831
10832     // Data member to access host view
10833     ViewCMatrix <T> host;
10834
10835     // Method that update host view
10836     void update_host();
10837
10838     // Method that update device view
10839     void update_device();
10840
10841     // Deconstructor
10842     KOKKOS_INLINE_FUNCTION
10843     ~DCMatrixKokkos ();
10844
10845 }; // End of DCMatrixKokkos declarations
10846
10847 // Default constructor
10848 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
10849 DCMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::DCMatrixKokkos() {}
10850
10851 // Overloaded 1D constructor
10852 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
10853 DCMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::DCMatrixKokkos(size_t dim1, const std::string&
       tag_string) {
10854
10855     dims_[0] = dim1;
10856     order_ = 1;
10857     length_ = dim1;
10858     this_matrix_ = TArray1D(tag_string, length_);
10859     // Create host ViewCMatrix
10860     host = ViewCMatrix <T> (this_matrix_.h_view.data(), dim1);
10861 }
10862
10863 // Overloaded 2D constructor
10864 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
10865 DCMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::DCMatrixKokkos(size_t dim1, size_t dim2, const
       std::string& tag_string) {
10866
10867     dims_[0] = dim1;
10868     dims_[1] = dim2;
10869     order_ = 2;
10870     length_ = (dim1 * dim2);
10871     this_matrix_ = TArray1D(tag_string, length_);
10872     // Create host ViewCMatrix
10873     host = ViewCMatrix <T> (this_matrix_.h_view.data(), dim1, dim2);
10874 }
10875
10876 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
10877 DCMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::DCMatrixKokkos(size_t dim1, size_t dim2,
10878                                size_t dim3, const std::string& tag_string) {
10879
10880     dims_[0] = dim1;
10881     dims_[1] = dim2;
10882     dims_[2] = dim3;
10883     order_ = 3;
10884     length_ = (dim1 * dim2 * dim3);
10885     this_matrix_ = TArray1D(tag_string, length_);
10886     // Create host ViewCMatrix
10887     host = ViewCMatrix <T> (this_matrix_.h_view.data(), dim1, dim2, dim3);
10888 }
10889
10890 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
10891 DCMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::DCMatrixKokkos(size_t dim1, size_t dim2,
10892                                size_t dim3, size_t dim4, const std::string& tag_string) {
10893
10894     dims_[0] = dim1;
10895     dims_[1] = dim2;
10896     dims_[2] = dim3;
10897     dims_[3] = dim4;
10898     order_ = 4;
10899     length_ = (dim1 * dim2 * dim3 * dim4);
10900     this_matrix_ = TArray1D(tag_string, length_);
10901     // Create host ViewCMatrix
10902     host = ViewCMatrix <T> (this_matrix_.h_view.data(), dim1, dim2, dim3, dim4);
10903 }
10904
10905 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
10906 DCMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::DCMatrixKokkos(size_t dim1, size_t dim2,
10907                                size_t dim3, size_t dim4,
10908                                size_t dim5, const std::string& tag_string) {
10909
10910     dims_[0] = dim1;
```

```
10911     dims_[1] = dim2;
10912     dims_[2] = dim3;
10913     dims_[3] = dim4;
10914     dims_[4] = dim5;
10915     order_ = 5;
10916     length_ = (dim1 * dim2 * dim3 * dim4 * dim5);
10917     this_matrix_ = TArray1D(tag_string, length_);
10918     // Create host ViewCMatrix
10919     host = ViewCMatrix <T> (this_matrix_.h_view.data(), dim1, dim2, dim3, dim4, dim5);
10920 }
10921
10922 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
10923 DCMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::DCMatrixKokkos(size_t dim1, size_t dim2,
10924                               size_t dim3, size_t dim4,
10925                               size_t dim5, size_t dim6, const std::string& tag_string) {
10926
10927     dims_[0] = dim1;
10928     dims_[1] = dim2;
10929     dims_[2] = dim3;
10930     dims_[3] = dim4;
10931     dims_[4] = dim5;
10932     dims_[5] = dim6;
10933     order_ = 6;
10934     length_ = (dim1 * dim2 * dim3 * dim4 * dim5 * dim6);
10935     this_matrix_ = TArray1D(tag_string, length_);
10936     // Create host ViewCMatrix
10937     host = ViewCMatrix <T> (this_matrix_.h_view.data(), dim1, dim2, dim3, dim4, dim5, dim6);
10938 }
10939
10940 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
10941 DCMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::DCMatrixKokkos(size_t dim1, size_t dim2,
10942                               size_t dim3, size_t dim4,
10943                               size_t dim5, size_t dim6,
10944                               size_t dim7, const std::string& tag_string) {
10945
10946     dims_[0] = dim1;
10947     dims_[1] = dim2;
10948     dims_[2] = dim3;
10949     dims_[3] = dim4;
10950     dims_[4] = dim5;
10951     dims_[5] = dim6;
10952     dims_[6] = dim7;
10953     order_ = 7;
10954     length_ = (dim1 * dim2 * dim3 * dim4 * dim5 * dim6 * dim7);
10955     this_matrix_ = TArray1D(tag_string, length_);
10956     // Create host ViewCMatrix
10957     host = ViewCMatrix <T> (this_matrix_.h_view.data(), dim1, dim2, dim3, dim4, dim5, dim6, dim7);
10958 }
10959
10960 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
10961 KOKKOS_INLINE_FUNCTION
10962 T& DCMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::operator()(size_t i) const {
10963     assert(order_ == 1 && "Tensor order (rank) does not match constructor in DCMatrixKokkos 1D!");
10964     assert(i >= 1 && i <= dims_[0] && "i is out of bounds in DCMatrixKokkos 1D!");
10965     return this_matrix_.d_view((i - 1));
10966 }
10967
10968 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
10969 KOKKOS_INLINE_FUNCTION
10970 T& DCMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::operator()(size_t i, size_t j) const {
10971     assert(order_ == 2 && "Tensor order (rank) does not match constructor in DCMatrixKokkos 2D!");
10972     assert(i >= 1 && i <= dims_[0] && "i is out of bounds in DCMatrixKokkos 2D!");
10973     assert(j >= 1 && j <= dims_[1] && "j is out of bounds in DCMatrixKokkos 2D!");
10974     return this_matrix_.d_view((j - 1) + ((i - 1) * dims_[1]));
10975 }
10976
10977 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
10978 KOKKOS_INLINE_FUNCTION
10979 T& DCMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::operator()(size_t i, size_t j, size_t k) const {
10980     assert(order_ == 3 && "Tensor order (rank) does not match constructor in DCMatrixKokkos 3D!");
10981     assert(i >= 1 && i <= dims_[0] && "i is out of bounds in DCMatrixKokkos 3D!");
10982     assert(j >= 1 && j <= dims_[1] && "j is out of bounds in DCMatrixKokkos 3D!");
10983     assert(k >= 1 && k <= dims_[2] && "k is out of bounds in DCMatrixKokkos 3D!");
10984     return this_matrix_.d_view((k - 1) + ((j - 1) * dims_[2])
10985                                        + ((i - 1) * dims_[2] * dims_[1]));
10986 }
10987
10988 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
10989 KOKKOS_INLINE_FUNCTION
10990 T& DCMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::operator()(size_t i, size_t j, size_t k, size_t l)
       const {
10991     assert(order_ == 4 && "Tensor order (rank) does not match constructor in DCMatrixKokkos 4D!");
10992     assert(i >= 1 && i <= dims_[0] && "i is out of bounds in DCMatrixKokkos 4D!");
10993     assert(j >= 1 && j <= dims_[1] && "j is out of bounds in DCMatrixKokkos 4D!");
10994     assert(k >= 1 && k <= dims_[2] && "k is out of bounds in DCMatrixKokkos 4D!");
10995     assert(l >= 1 && l <= dims_[3] && "l is out of bounds in DCMatrixKokkos 4D!");
10996     return this_matrix_.d_view((l - 1) + ((k - 1) * dims_[3])
```

```
10997                                       + ((j - 1) * dims_[3] * dims_[2])
10998                                       + ((i - 1) * dims_[3] * dims_[2] * dims_[1]));
10999 }
11000
11001 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
11002 KOKKOS_INLINE_FUNCTION
11003 T& DCMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::operator()(size_t i, size_t j, size_t k, size_t l,
11004                                 size_t m) const {
11005     assert(order_ == 5 && "Tensor order (rank) does not match constructor in DCMatrixKokkos 5D!");
11006     assert(i >= 1 && i <= dims_[0] && "i is out of bounds in DCMatrixKokkos 5D!");
11007     assert(j >= 1 && j <= dims_[1] && "j is out of bounds in DCMatrixKokkos 5D!");
11008     assert(k >= 1 && k <= dims_[2] && "k is out of bounds in DCMatrixKokkos 5D!");
11009     assert(l >= 1 && l <= dims_[3] && "l is out of bounds in DCMatrixKokkos 5D!");
11010     assert(m >= 1 && m <= dims_[4] && "m is out of bounds in DCMatrixKokkos 5D!");
11011     return this_matrix_.d_view((m - 1) + ((l - 1) * dims_[4])
11012                                       + ((k - 1) * dims_[4] * dims_[3])
11013                                       + ((j - 1) * dims_[4] * dims_[3] * dims_[2])
11014                                       + ((i - 1) * dims_[4] * dims_[3] * dims_[2] * dims_[1]));
11015 }
11016
11017 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
11018 KOKKOS_INLINE_FUNCTION
11019 T& DCMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::operator()(size_t i, size_t j, size_t k, size_t l,
11020                                 size_t m, size_t n) const {
11021     assert(order_ == 6 && "Tensor order (rank) does not match constructor in DCMatrixKokkos 6D!");
11022     assert(i >= 1 && i <= dims_[0] && "i is out of bounds in DCMatrixKokkos 6D!");
11023     assert(j >= 1 && j <= dims_[1] && "j is out of bounds in DCMatrixKokkos 6D!");
11024     assert(k >= 1 && k <= dims_[2] && "k is out of bounds in DCMatrixKokkos 6D!");
11025     assert(l >= 1 && l <= dims_[3] && "l is out of bounds in DCMatrixKokkos 6D!");
11026     assert(m >= 1 && m <= dims_[4] && "m is out of bounds in DCMatrixKokkos 6D!");
11027     assert(n >= 1 && n <= dims_[5] && "n is out of bounds in DCMatrixKokkos 6D!");
11028     return this_matrix_.d_view((n - 1) + ((m - 1) * dims_[5])
11029                                       + ((l - 1) * dims_[5] * dims_[4])
11030                                       + ((k - 1) * dims_[5] * dims_[4] * dims_[3])
11031                                       + ((j - 1) * dims_[5] * dims_[4] * dims_[3] * dims_[2])
11032                                       + ((i - 1) * dims_[5] * dims_[4] * dims_[3] * dims_[2] *
       dims_[1]));
11033 }
11034
11035 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
11036 KOKKOS_INLINE_FUNCTION
11037 T& DCMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::operator()(size_t i, size_t j, size_t k, size_t l,
11038                                 size_t m, size_t n, size_t o) const {
11039     assert(order_ == 7 && "Tensor order (rank) does not match constructor in DCMatrixKokkos 7D!");
11040     assert(i >= 1 && i <= dims_[0] && "i is out of bounds in DCMatrixKokkos 7D!");
11041     assert(j >= 1 && j <= dims_[1] && "j is out of bounds in DCMatrixKokkos 7D!");
11042     assert(k >= 1 && k <= dims_[2] && "k is out of bounds in DCMatrixKokkos 7D!");
11043     assert(l >= 1 && l <= dims_[3] && "l is out of bounds in DCMatrixKokkos 7D!");
11044     assert(m >= 1 && m <= dims_[4] && "m is out of bounds in DCMatrixKokkos 7D!");
11045     assert(n >= 1 && n <= dims_[5] && "n is out of bounds in DCMatrixKokkos 7D!");
11046     assert(o >= 1 && o <= dims_[6] && "o is out of bounds in DCMatrixKokkos 7D!");
11047     return this_matrix_.d_view((o-1) + ((n - 1) * dims_[6])
11048                                     + ((m - 1) * dims_[6] * dims_[5])
11049                                     + ((l - 1) * dims_[6] * dims_[5] * dims_[4])
11050                                     + ((k - 1) * dims_[6] * dims_[5] * dims_[4] * dims_[3])
11051                                     + ((j - 1) * dims_[6] * dims_[5] * dims_[4] * dims_[3] *
       dims_[2])
11052                                     + ((i - 1) * dims_[6] * dims_[5] * dims_[4] * dims_[3] * dims_[2]
       * dims_[1]));
11053 }
11054
11055 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
11056 KOKKOS_INLINE_FUNCTION
11057 DCMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>&
       DCMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::operator= (const DCMatrixKokkos& temp) {
11058
11059     // Do nothing if the assignment is of the form x = x
11060     if (this != &temp) {
11061         for (int iter = 0; iter < temp.order_; iter++){
11062             dims_[iter] = temp.dims_[iter];
11063         } // end for
11064
11065         order_ = temp.order_;
11066         length_ = temp.length_;
11067         this_matrix_ = temp.this_matrix_;
11068         host = temp.host;
11069     }
11070
11071     return *this;
11072 }
11073
11074 // Return size
11075 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
11076 KOKKOS_INLINE_FUNCTION
11077 size_t DCMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::size() const {
11078     return length_;
11079 }
```

```
11080
11081 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
11082 KOKKOS_INLINE_FUNCTION
11083 size_t DCMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::extent() const {
11084     return length_;
11085 }
11086
11087 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
11088 KOKKOS_INLINE_FUNCTION
11089 size_t DCMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::dims(size_t i) const {
11090     i--;
11091     assert(i < order_ && "DCMatrixKokkos order (rank) does not match constructor, dim[i] does not
       exist!");
11092     assert(i >= 0 && dims_[i]>0 && "Access to DCMatrixKokkos dims is out of bounds!");
11093     return dims_[i];
11094 }
11095
11096 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
11097 KOKKOS_INLINE_FUNCTION
11098 size_t DCMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::order() const {
11099     return order_;
11100 }
11101
11102 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
11103 KOKKOS_INLINE_FUNCTION
11104 T* DCMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::device_pointer() const {
11105     return this_matrix_.d_view.data();
11106 }
11107
11108 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
11109 KOKKOS_INLINE_FUNCTION
11110 T* DCMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::host_pointer() const {
11111     return this_matrix_.h_view.data();
11112 }
11113
11114 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
11115 void DCMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::update_host() {
11116
11117     this_matrix_.template modify<typename TArray1D::execution_space>();
11118     this_matrix_.template sync<typename TArray1D::host_mirror_space>();
11119 }
11120
11121 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
11122 void DCMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::update_device() {
11123
11124     this_matrix_.template modify<typename TArray1D::host_mirror_space>();
11125     this_matrix_.template sync<typename TArray1D::execution_space>();
11126 }
11127
11128 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
11129 KOKKOS_INLINE_FUNCTION
11130 DCMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::~DCMatrixKokkos() {}
11131 // End DCMatrixKokkos
11132
11133
11135 // DViewCMatrixKokkos
11137 template <typename T, typename Layout = DefaultLayout, typename ExecSpace = DefaultExecSpace, typename
       MemoryTraits = void>
11138 class DViewCMatrixKokkos {
11139
11140     // this is always unmanaged
11141     using TArray1DHost = Kokkos::View<T*, Layout, HostSpace, MemoryUnmanaged>;
11142     // this is manage
11143     using TArray1D     = Kokkos::View<T*, Layout, ExecSpace, MemoryTraits>;
11144
11145 private:
11146     size_t dims_[7];
11147     size_t length_;
11148     size_t order_;  // tensor order (rank)
11149     TArray1D this_matrix_;
11150     TArray1DHost this_matrix_host_;
11151     T * temp_inp_matrix_;
11152
11153 public:
11154     DViewCMatrixKokkos();
11155
11156     DViewCMatrixKokkos(T * inp_matrix, size_t dim1);
11157
11158     DViewCMatrixKokkos(T * inp_matrix, size_t dim1, size_t dim2);
11159
11160     DViewCMatrixKokkos(T * inp_matrix, size_t dim1, size_t dim2, size_t dim3);
11161
11162     DViewCMatrixKokkos(T * inp_matrix, size_t dim1, size_t dim2, size_t dim3,
11163                 size_t dim4);
11164
11165     DViewCMatrixKokkos(T * inp_matrix, size_t dim1, size_t dim2, size_t dim3,
11166                 size_t dim4, size_t dim5);
```

```
11167
11168     DViewCMatrixKokkos(T * inp_matrix, size_t dim1, size_t dim2, size_t dim3,
11169                 size_t dim4, size_t dim5, size_t dim6);
11170
11171     DViewCMatrixKokkos(T * inp_matrix, size_t dim1, size_t dim2, size_t dim3,
11172                 size_t dim4, size_t dim5, size_t dim6,
11173                 size_t dim7);
11174
11175     KOKKOS_INLINE_FUNCTION
11176     T& operator()(size_t i) const;
11177
11178     KOKKOS_INLINE_FUNCTION
11179     T& operator()(size_t i, size_t j) const;
11180
11181     KOKKOS_INLINE_FUNCTION
11182     T& operator()(size_t i, size_t j, size_t k) const;
11183
11184     KOKKOS_INLINE_FUNCTION
11185     T& operator()(size_t i, size_t j, size_t k, size_t l) const;
11186
11187     KOKKOS_INLINE_FUNCTION
11188     T& operator()(size_t i, size_t j, size_t k, size_t l, size_t m) const;
11189
11190     KOKKOS_INLINE_FUNCTION
11191     T& operator()(size_t i, size_t j, size_t k, size_t l, size_t m,
11192                   size_t n) const;
11193
11194     KOKKOS_INLINE_FUNCTION
11195     T& operator()(size_t i, size_t j, size_t k, size_t l, size_t m,
11196                   size_t n, size_t o) const;
11197
11198     KOKKOS_INLINE_FUNCTION
11199     DViewCMatrixKokkos& operator=(const DViewCMatrixKokkos& temp);
11200
11201     // GPU Method
11202     // Method that returns size
11203     KOKKOS_INLINE_FUNCTION
11204     size_t size() const;
11205
11206     // Host Method
11207     // Method that returns size
11208     KOKKOS_INLINE_FUNCTION
11209     size_t extent() const;
11210
11211     KOKKOS_INLINE_FUNCTION
11212     size_t dims(size_t i) const;
11213
11214     KOKKOS_INLINE_FUNCTION
11215     size_t order() const;
11216
11217     // Method returns the raw device pointer of the Kokkos View
11218     KOKKOS_INLINE_FUNCTION
11219     T* device_pointer() const;
11220
11221     // Method returns the raw host pointer of the Kokkos View
11222     KOKKOS_INLINE_FUNCTION
11223     T* host_pointer() const;
11224
11225     // Data member to access host view
11226     ViewCMatrix <T> host;
11227
11228     // Method that update host view
11229     void update_host();
11230
11231     // Method that update device view
11232     void update_device();
11233
11234     // Deconstructor
11235     KOKKOS_INLINE_FUNCTION
11236     ~DViewCMatrixKokkos ();
11237 }; // End of DViewCMatrixKokkos
11238
11239
11240 // Default constructor
11241 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
11242 DViewCMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::DViewCMatrixKokkos() {}
11243
11244 // Overloaded 1D constructor
11245 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
11246 DViewCMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::DViewCMatrixKokkos(T * inp_matrix, size_t dim1) {
11247
11248     dims_[0] = dim1;
11249     order_ = 1;
11250     length_ = dim1;
11251     // Create a 1D host view of the external allocation
11252     this_matrix_host_ = TArray1DHost(inp_matrix, length_);
11253     // Assign temp point to inp_matrix pointer that is passed in
```

```
11254     temp_inp_matrix_ = inp_matrix;
11255     // Create a device copy of that host view
11256     this_matrix_ = create_mirror_view_and_copy(ExecSpace(), this_matrix_host_);
11257     // Create host ViewCMatrix. Note: inp_matrix and this_matrix_host_.data() are the same pointer
11258     host = ViewCMatrix <T> (inp_matrix, dim1);
11259 }
11260
11261 // Overloaded 2D constructor
11262 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
11263 DViewCMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::DViewCMatrixKokkos(T * inp_matrix, size_t dim1,
       size_t dim2) {
11264
11265     dims_[0] = dim1;
11266     dims_[1] = dim2;
11267     order_ = 2;
11268     length_ = (dim1 * dim2);
11269     // Create a 1D host view of the external allocation
11270     this_matrix_host_ = TArray1DHost(inp_matrix, length_);
11271     // Assign temp point to inp_matrix pointer that is passed in
11272     temp_inp_matrix_ = inp_matrix;
11273     // Create a device copy of that host view
11274     this_matrix_ = create_mirror_view_and_copy(ExecSpace(), this_matrix_host_);
11275     // Create host ViewCMatrix
11276     host = ViewCMatrix <T> (inp_matrix, dim1, dim2);
11277 }
11278
11279 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
11280 DViewCMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::DViewCMatrixKokkos(T * inp_matrix, size_t dim1,
       size_t dim2,
11281                                 size_t dim3) {
11282
11283     dims_[0] = dim1;
11284     dims_[1] = dim2;
11285     dims_[2] = dim3;
11286     order_ = 3;
11287     length_ = (dim1 * dim2 * dim3);
11288     // Create a 1D host view of the external allocation
11289     this_matrix_host_ = TArray1DHost(inp_matrix, length_);
11290     // Assign temp point to inp_matrix pointer that is passed in
11291     temp_inp_matrix_ = inp_matrix;
11292     // Create a device copy of that host view
11293     this_matrix_ = create_mirror_view_and_copy(ExecSpace(), this_matrix_host_);
11294     // Create host ViewCMatrix
11295     host = ViewCMatrix <T> (inp_matrix, dim1, dim2, dim3);
11296 }
11297
11298 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
11299 DViewCMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::DViewCMatrixKokkos(T * inp_matrix, size_t dim1,
       size_t dim2,
11300                                 size_t dim3, size_t dim4) {
11301
11302     dims_[0] = dim1;
11303     dims_[1] = dim2;
11304     dims_[2] = dim3;
11305     dims_[3] = dim4;
11306     order_ = 4;
11307     length_ = (dim1 * dim2 * dim3 * dim4);
11308     // Create a 1D host view of the external allocation
11309     this_matrix_host_ = TArray1DHost(inp_matrix, length_);
11310     // Assign temp point to inp_matrix pointer that is passed in
11311     temp_inp_matrix_ = inp_matrix;
11312     // Create a device copy of that host view
11313     this_matrix_ = create_mirror_view_and_copy(ExecSpace(), this_matrix_host_);
11314     // Create host ViewCMatrix
11315     host = ViewCMatrix <T> (inp_matrix, dim1, dim2, dim3, dim4);
11316 }
11317
11318 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
11319 DViewCMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::DViewCMatrixKokkos(T * inp_matrix, size_t dim1,
       size_t dim2,
11320                                 size_t dim3, size_t dim4,
11321                                 size_t dim5) {
11322
11323     dims_[0] = dim1;
11324     dims_[1] = dim2;
11325     dims_[2] = dim3;
11326     dims_[3] = dim4;
11327     dims_[4] = dim5;
11328     order_ = 5;
11329     length_ = (dim1 * dim2 * dim3 * dim4 * dim5);
11330     // Create a 1D host view of the external allocation
11331     this_matrix_host_ = TArray1DHost(inp_matrix, length_);
11332     // Assign temp point to inp_matrix pointer that is passed in
11333     temp_inp_matrix_ = inp_matrix;
11334     // Create a device copy of that host view
11335     this_matrix_ = create_mirror_view_and_copy(ExecSpace(), this_matrix_host_);
11336     // Create host ViewCMatrix
```

```
11337     host = ViewCMatrix <T> (inp_matrix, dim1, dim2, dim3, dim4, dim5);
11338 }
11339
11340 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
11341 DViewCMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::DViewCMatrixKokkos(T * inp_matrix, size_t dim1,
       size_t dim2,
11342                               size_t dim3, size_t dim4,
11343                               size_t dim5, size_t dim6) {
11344
11345     dims_[0] = dim1;
11346     dims_[1] = dim2;
11347     dims_[2] = dim3;
11348     dims_[3] = dim4;
11349     dims_[4] = dim5;
11350     dims_[5] = dim6;
11351     order_ = 6;
11352     length_ = (dim1 * dim2 * dim3 * dim4 * dim5 * dim6);
11353     // Create a 1D host view of the external allocation
11354     this_matrix_host_ = TArray1DHost(inp_matrix, length_);
11355     // Assign temp point to inp_matrix pointer that is passed in
11356     temp_inp_matrix_ = inp_matrix;
11357     // Create a device copy of that host view
11358     this_matrix_ = create_mirror_view_and_copy(ExecSpace(), this_matrix_host_);
11359     // Create host ViewCMatrix
11360     host = ViewCMatrix <T> (inp_matrix, dim1, dim2, dim3, dim4, dim5, dim6);
11361 }
11362
11363 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
11364 DViewCMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::DViewCMatrixKokkos(T * inp_matrix, size_t dim1,
       size_t dim2,
11365                               size_t dim3, size_t dim4,
11366                               size_t dim5, size_t dim6,
11367                               size_t dim7) {
11368
11369     dims_[0] = dim1;
11370     dims_[1] = dim2;
11371     dims_[2] = dim3;
11372     dims_[3] = dim4;
11373     dims_[4] = dim5;
11374     dims_[5] = dim6;
11375     dims_[6] = dim7;
11376     order_ = 7;
11377     length_ = (dim1 * dim2 * dim3 * dim4 * dim5 * dim6 * dim7);
11378     // Create a 1D host view of the external allocation
11379     this_matrix_host_ = TArray1DHost(inp_matrix, length_);
11380     // Assign temp point to inp_matrix pointer that is passed in
11381     temp_inp_matrix_ = inp_matrix;
11382     // Create a device copy of that host view
11383     this_matrix_ = create_mirror_view_and_copy(ExecSpace(), this_matrix_host_);
11384     // Create host ViewCMatrix
11385     host = ViewCMatrix <T> (inp_matrix, dim1, dim2, dim3, dim4, dim5, dim6, dim7);
11386 }
11387
11388 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
11389 KOKKOS_INLINE_FUNCTION
11390 T& DViewCMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::operator()(size_t i) const {
11391     assert(order_ == 1 && "Tensor order (rank) does not match constructor in DViewCMatrixKokkos 1D!");
11392     assert(i >= 1 && i <= dims_[0] && "i is out of bounds in DViewCMatrixKokkos 1D!");
11393     return this_matrix_((i - 1));
11394 }
11395
11396 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
11397 KOKKOS_INLINE_FUNCTION
11398 T& DViewCMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::operator()(size_t i, size_t j) const {
11399     assert(order_ == 2 && "Tensor order (rank) does not match constructor in DViewCMatrixKokkos 2D!");
11400     assert(i >= 1 && i <= dims_[0] && "i is out of bounds in DViewCMatrixKokkos 2D!");
11401     assert(j >= 1 && j <= dims_[1] && "j is out of bounds in DViewCMatrixKokkos 2D!");
11402     return this_matrix_((j - 1) + ((i - 1) * dims_[1]));
11403 }
11404
11405 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
11406 KOKKOS_INLINE_FUNCTION
11407 T& DViewCMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::operator()(size_t i, size_t j, size_t k) const
       {
11408     assert(order_ == 3 && "Tensor order (rank) does not match constructor in DViewCMatrixKokkos 3D!");
11409     assert(i >= 1 && i <= dims_[0] && "i is out of bounds in DViewCMatrixKokkos 3D!");
11410     assert(j >= 1 && j <= dims_[1] && "j is out of bounds in DViewCMatrixKokkos 3D!");
11411     assert(k >= 1 && k <= dims_[2] && "k is out of bounds in DViewCMatrixKokkos 3D!");
11412     return this_matrix_((k - 1) + ((j - 1) * dims_[2])
11413                                 + ((i - 1) * dims_[2] * dims_[1]));
11414 }
11415
11416 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
11417 KOKKOS_INLINE_FUNCTION
11418 T& DViewCMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::operator()(size_t i, size_t j, size_t k,
       size_t l) const {
11419     assert(order_ == 4 && "Tensor order (rank) does not match constructor in DViewCMatrixKokkos 4D!");
```

```
11420     assert(i >= 1 && i <= dims_[0] && "i is out of bounds in DViewCMatrixKokkos 4D!");
11421     assert(j >= 1 && j <= dims_[1] && "j is out of bounds in DViewCMatrixKokkos 4D!");
11422     assert(k >= 1 && k <= dims_[2] && "k is out of bounds in DViewCMatrixKokkos 4D!");
11423     assert(l >= 1 && l <= dims_[3] && "l is out of bounds in DViewCMatrixKokkos 4D!");
11424     return this_matrix_((l - 1) + ((k - 1) * dims_[3])
11425                                 + ((j - 1) * dims_[3] * dims_[2])
11426                                 + ((i - 1) * dims_[3] * dims_[2] * dims_[1]));
11427 }
11428
11429 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
11430 KOKKOS_INLINE_FUNCTION
11431 T& DViewCMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::operator()(size_t i, size_t j, size_t k,
       size_t l,
11432                                 size_t m) const {
11433     assert(order_ == 5 && "Tensor order (rank) does not match constructor in DViewCMatrixKokkos 5D!");
11434     assert(i >= 1 && i <= dims_[0] && "i is out of bounds in DViewCMatrixKokkos 5D!");
11435     assert(j >= 1 && j <= dims_[1] && "j is out of bounds in DViewCMatrixKokkos 5D!");
11436     assert(k >= 1 && k <= dims_[2] && "k is out of bounds in DViewCMatrixKokkos 5D!");
11437     assert(l >= 1 && l <= dims_[3] && "l is out of bounds in DViewCMatrixKokkos 5D!");
11438     assert(m >= 1 && m <= dims_[4] && "m is out of bounds in DViewCMatrixKokkos 5D!");
11439     return this_matrix_((m - 1) + ((l - 1) * dims_[4])
11440                                 + ((k - 1) * dims_[4] * dims_[3])
11441                                 + ((j - 1) * dims_[4] * dims_[3] * dims_[2])
11442                                 + ((i - 1) * dims_[4] * dims_[3] * dims_[2] * dims_[1]));
11443 }
11444
11445 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
11446 KOKKOS_INLINE_FUNCTION
11447 T& DViewCMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::operator()(size_t i, size_t j, size_t k,
       size_t l,
11448                                 size_t m, size_t n) const {
11449     assert(order_ == 6 && "Tensor order (rank) does not match constructor in DViewCMatrixKokkos 6D!");
11450     assert(i >= 1 && i <= dims_[0] && "i is out of bounds in DViewCMatrixKokkos 6D!");
11451     assert(j >= 1 && j <= dims_[1] && "j is out of bounds in DViewCMatrixKokkos 6D!");
11452     assert(k >= 1 && k <= dims_[2] && "k is out of bounds in DViewCMatrixKokkos 6D!");
11453     assert(l >= 1 && l <= dims_[3] && "l is out of bounds in DViewCMatrixKokkos 6D!");
11454     assert(m >= 1 && m <= dims_[4] && "m is out of bounds in DViewCMatrixKokkos 6D!");
11455     assert(n >= 1 && n <= dims_[5] && "n is out of bounds in DViewCMatrixKokkos 6D!");
11456     return this_matrix_((n - 1) + ((m - 1) * dims_[5])
11457                                 + ((l - 1) * dims_[5] * dims_[4])
11458                                 + ((k - 1) * dims_[5] * dims_[4] * dims_[3])
11459                                 + ((j - 1) * dims_[5] * dims_[4] * dims_[3] * dims_[2])
11460                                 + ((i - 1) * dims_[5] * dims_[4] * dims_[3] * dims_[2] * dims_[1]));
11461 }
11462
11463 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
11464 KOKKOS_INLINE_FUNCTION
11465 T& DViewCMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::operator()(size_t i, size_t j, size_t k,
       size_t l,
11466                                 size_t m, size_t n, size_t o) const {
11467     assert(order_ == 7 && "Tensor order (rank) does not match constructor in DViewCMatrixKokkos 7D!");
11468     assert(i >= 1 && i <= dims_[0] && "i is out of bounds in DViewCMatrixKokkos 7D!");
11469     assert(j >= 1 && j <= dims_[1] && "j is out of bounds in DViewCMatrixKokkos 7D!");
11470     assert(k >= 1 && k <= dims_[2] && "k is out of bounds in DViewCMatrixKokkos 7D!");
11471     assert(l >= 1 && l <= dims_[3] && "l is out of bounds in DViewCMatrixKokkos 7D!");
11472     assert(m >= 1 && m <= dims_[4] && "m is out of bounds in DViewCMatrixKokkos 7D!");
11473     assert(n >= 1 && n <= dims_[5] && "n is out of bounds in DViewCMatrixKokkos 7D!");
11474     assert(o >= 1 && o <= dims_[6] && "o is out of bounds in DViewCMatrixKokkos 7D!");
11475     return this_matrix_(o + ((n - 1) * dims_[6])
11476                           + ((m - 1) * dims_[6] * dims_[5])
11477                           + ((l - 1) * dims_[6] * dims_[5] * dims_[4])
11478                           + ((k - 1) * dims_[6] * dims_[5] * dims_[4] * dims_[3])
11479                           + ((j - 1) * dims_[6] * dims_[5] * dims_[4] * dims_[3] * dims_[2])
11480                           + ((i - 1) * dims_[6] * dims_[5] * dims_[4] * dims_[3] * dims_[2] *
       dims_[1]));
11481 }
11482
11483 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
11484 KOKKOS_INLINE_FUNCTION
11485 DViewCMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>&
       DViewCMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::operator= (const DViewCMatrixKokkos& temp) {
11486
11487     // Do nothing if the assignment is of the form x = x
11488     if (this != &temp) {
11489         for (int iter = 0; iter < temp.order_; iter++){
11490             dims_[iter] = temp.dims_[iter];
11491         } // end for
11492
11493         order_ = temp.order_;
11494         length_ = temp.length_;
11495         temp_inp_matrix_ = temp.temp_inp_matrix_;
11496         this_matrix_host_ = temp.this_matrix_host_;
11497         this_matrix_ = temp.this_matrix_;
11498         host = temp.host;
11499     }
11500
11501     return *this;
```

```
11502 }
11503
11504 // Return size
11505 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
11506 KOKKOS_INLINE_FUNCTION
11507 size_t DViewCMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::size() const {
11508     return length_;
11509 }
11510
11511 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
11512 KOKKOS_INLINE_FUNCTION
11513 size_t DViewCMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::extent() const {
11514     return length_;
11515 }
11516
11517 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
11518 KOKKOS_INLINE_FUNCTION
11519 size_t DViewCMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::dims(size_t i) const {
11520     i--;
11521     assert(i < order_ && "DViewCMatrixKokkos order (rank) does not match constructor, dim[i] does not
       exist!");
11522     assert(i >= 0 && dims_[i]>0 && "Access to DViewCMatrixKokkos dims is out of bounds!");
11523     return dims_[i];
11524 }
11525
11526 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
11527 KOKKOS_INLINE_FUNCTION
11528 size_t DViewCMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::order() const {
11529     return order_;
11530 }
11531
11532 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
11533 KOKKOS_INLINE_FUNCTION
11534 T* DViewCMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::device_pointer() const {
11535     return this_matrix_.data();
11536 }
11537
11538 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
11539 KOKKOS_INLINE_FUNCTION
11540 T* DViewCMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::host_pointer() const {
11541     return this_matrix_host_.data();
11542 }
11543
11544 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
11545 void DViewCMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::update_host() {
11546     // Deep copy of device view to host view
11547     deep_copy(this_matrix_host_, this_matrix_);
11548 }
11549
11550 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
11551 void DViewCMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::update_device() {
11552     // Deep copy of host view to device view
11553     deep_copy(this_matrix_, this_matrix_host_);
11554 }
11555
11556 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
11557 KOKKOS_INLINE_FUNCTION
11558 DViewCMatrixKokkos<T,Layout,ExecSpace,MemoryTraits>::~DViewCMatrixKokkos() {}
11559 // End DViewCMatrixKokkos
11560
11561
11565 template <typename T, typename Layout = DefaultLayout, typename ExecSpace = DefaultExecSpace,
11566  typename MemoryTraits = void, typename ILayout = Layout>
11567 class RaggedRightArrayKokkos {
11568
11569     using TArray1D = Kokkos::View<T*, Layout, ExecSpace, MemoryTraits>;
11570     using SArray1D = Kokkos::View<size_t *,Layout, ExecSpace, MemoryTraits>;
11571     using Strides1D = Kokkos::View<size_t *,ILayout, ExecSpace, MemoryTraits>;
11572
11573 private:
11574     TArray1D array_;
11575
11576     size_t dim1_;
11577     size_t length_;
11578
11579 public:
11580     // Default constructor
11581     RaggedRightArrayKokkos();
11582
11583     //--- 2D array access of a ragged right array ---
11584
11585     // Overload constructor for a CArrayKokkos
11586     RaggedRightArrayKokkos(CArrayKokkos<size_t,ILayout,ExecSpace,MemoryTraits> &strides_array, const
       std::string& tag_string = DEFAULTSTRINGARRAY);
11587
11588     // Overload constructor for a DCArrayKokkos
11589     RaggedRightArrayKokkos(DCArrayKokkos<size_t,ILayout,ExecSpace,MemoryTraits> &strides_array, const
```

```
      std::string& tag_string = DEFAULTSTRINGARRAY);
11590
11591     // Overload constructor for a ViewCArray
11592     RaggedRightArrayKokkos(ViewCArray<size_t> &strides_array, const std::string& tag_string =
      DEFAULTSTRINGARRAY);
11593
11594     // Overloaded constructor for a traditional array
11595     RaggedRightArrayKokkos(size_t* strides_array, size_t some_dim1, const std::string& tag_string =
      DEFAULTSTRINGARRAY);
11596
11597     // A method to return the stride size
11598     KOKKOS_INLINE_FUNCTION
11599     size_t stride(size_t i) const;
11600
11601     // Host method to return the stride size
11602     size_t stride_host(size_t i) const;
11603
11604     // A method to increase the number of column entries, i.e.,
11605     // the stride size. Used with the constructor for building
11606     // the stride_array dynamically.
11607     // DO NOT USE with the constructures with a strides_array
11608     KOKKOS_INLINE_FUNCTION
11609     size_t& build_stride(const size_t i) const;
11610
11611     KOKKOS_INLINE_FUNCTION
11612     void stride_finalize() const;
11613
11614     // Overload operator() to access data as array(i,j)
11615     // where i=[0:N-1], j=[stride(i)]
11616     KOKKOS_INLINE_FUNCTION
11617     T& operator()(size_t i, size_t j) const;
11618
11619     // method to return total size
11620     KOKKOS_INLINE_FUNCTION
11621     size_t size(){
11622       return length_;
11623     }
11624
11625     //setup start indices
11626     void data_setup(const std::string& tag_string);
11627
11628     KOKKOS_INLINE_FUNCTION
11629     T* pointer();
11630
11631     //return the view
11632     KOKKOS_INLINE_FUNCTION
11633     TArray1D get_kokkos_view();
11634
11635     // Kokkos views of strides and start indices
11636     Strides1D mystrides_;
11637     SArray1D start_index_;
11638
11639     KOKKOS_INLINE_FUNCTION
11640     RaggedRightArrayKokkos& operator= (const RaggedRightArrayKokkos &temp);
11641
11642     //initialize start indices view
11643     class init_start_indices_functor{
11644       public:
11645       SArray1D mystart_index_;
11646       init_start_indices_functor(SArray1D tempstart_index_){
11647         mystart_index_ = tempstart_index_;
11648       }
11649       KOKKOS_INLINE_FUNCTION void operator()(const int index) const {
11650         mystart_index_(index) = 0;
11651       }
11652     };
11653
11654     //setup start indices view
11655     class setup_start_indices_functor{
11656         public:
11657         SArray1D mystart_index_;
11658         Strides1D mytemp_strides_;
11659         setup_start_indices_functor(SArray1D tempstart_index_, Strides1D temp_strides_){
11660           mystart_index_ = tempstart_index_;
11661           mytemp_strides_ = temp_strides_;
11662         }
11663         KOKKOS_INLINE_FUNCTION void operator()(const int index, int& update, bool final) const {
11664           // Load old value in case we update it before accumulating
11665             const size_t count = mytemp_strides_(index);
11666             update += count;
11667             if (final) {
11668                 mystart_index_((index+1)) = update;
11669             }
11670         }
11671     };
11672
11673     //setup length of view
```

```
11674     class setup_length_functor{
11675         public:
11676         //kokkos needs this typedef named
11677         typedef size_t value_type;
11678         // This is helpful for determining the right index type,
11679         // especially if you expect to need a 64-bit index.
11680         //typedef Kokkos::View<size_t*>::size_type size_type;
11681         Strides1D mytemp_strides_;
11682         setup_length_functor(Strides1D temp_strides_){
11683           mytemp_strides_ = temp_strides_;
11684         }
11685         KOKKOS_INLINE_FUNCTION void operator()(const int index, size_t& update) const {
11686             //const size_t count = mytemp_strides_(index);
11687             update += mytemp_strides_(index);
11688         }
11689     };
11690
11691     //sets final 1D array size
11692     class finalize_stride_functor{
11693         public:
11694         SArray1D mystart_index_;
11695         finalize_stride_functor(SArray1D tempstart_index_){
11696           mystart_index_ = tempstart_index_;
11697         }
11698         KOKKOS_INLINE_FUNCTION void operator()(const int index, int& update, bool final) const {
11699           // Load old value in case we update it before accumulating
11700             const size_t count = mystart_index_(index+1);
11701             update += count;
11702             if (final) {
11703                 mystart_index_((index+1)) = update;
11704             }
11705         }
11706     };
11707
11708     // Destructor
11709     KOKKOS_INLINE_FUNCTION
11710     ~RaggedRightArrayKokkos ( );
11711 }; // End of RaggedRightArray
11712
11713 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits, typename ILayout>
11714 RaggedRightArrayKokkos<T,Layout,ExecSpace,MemoryTraits,ILayout>::RaggedRightArrayKokkos() {}
11715
11716 // Overloaded constructor
11717 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits, typename ILayout>
11718
       RaggedRightArrayKokkos<T,Layout,ExecSpace,MemoryTraits,ILayout>::RaggedRightArrayKokkos(CArrayKokkos<size_t,ILayout,Exe
       &strides_array,
11719                                                                                                const
       std::string& tag_string) {
11720     mystrides_ = strides_array.get_kokkos_view();
11721     dim1_ = strides_array.extent();
11722     data_setup(tag_string);
11723 } // End constructor
11724
11725 // Overloaded constructor
11726 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits, typename ILayout>
11727
       RaggedRightArrayKokkos<T,Layout,ExecSpace,MemoryTraits,ILayout>::RaggedRightArrayKokkos(DCArrayKokkos<size_t,ILayout,Ex
       &strides_array,
11728                                                                                                const
       std::string& tag_string) {
11729     mystrides_ = strides_array.get_kokkos_dual_view().d_view;
11730     dim1_ = strides_array.extent();
11731     data_setup(tag_string);
11732 } // End constructor
11733
11734 // Overloaded constructor
11735 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits, typename ILayout>
11736
       RaggedRightArrayKokkos<T,Layout,ExecSpace,MemoryTraits,ILayout>::RaggedRightArrayKokkos(ViewCArray<size_t>
       &strides_array,
11737                                                                                                 const
       std::string& tag_string) {
11738 } // End constructor
11739
11740 // Overloaded constructor
11741 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits, typename ILayout>
11742 RaggedRightArrayKokkos<T,Layout,ExecSpace,MemoryTraits,ILayout>::RaggedRightArrayKokkos(size_t*
       strides_array,  size_t some_dim1,
11743                                                                                                const
       std::string& tag_string) {
11744     mystrides_.assign_data(strides_array);
11745     dim1_ = some_dim1;
11746     data_setup(tag_string);
11747 } // End constructor
11748
11749 //setup start indices
```

```
11750 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits, typename ILayout>
11751 void RaggedRightArrayKokkos<T,Layout,ExecSpace,MemoryTraits,ILayout>::data_setup(const std::string&
      tag_string) {
11752     //allocate start indices
11753     std::string append_indices_string("start_indices");
11754     std::string append_array_string("array");
11755     std::string temp_copy_string = tag_string;
11756     std::string start_index_tag_string = temp_copy_string.append(append_indices_string);
11757     temp_copy_string = tag_string;
11758     std::string array_tag_string = temp_copy_string.append(append_array_string);
11759 
11760     start_index_ = SArray1D(start_index_tag_string,dim1_ + 1);
11761     #ifdef HAVE_CLASS_LAMBDA
11762     Kokkos::parallel_for("StartValuesInit", dim1_+1, KOKKOS_CLASS_LAMBDA(const int i) {
11763       start_index_((i) = 0;
11764     });
11765     #else
11766     init_start_indices_functor execution_functor(start_index_);
11767     Kokkos::parallel_for("StartValuesInit", dim1_+1,execution_functor);
11768     #endif
11769 
11770     #ifdef HAVE_CLASS_LAMBDA
11771     Kokkos::parallel_scan("StartValuesSetup", dim1_, KOKKOS_CLASS_LAMBDA(const int i, int& update,
      const bool final) {
11772             // Load old value in case we update it before accumulating
11773             const size_t count = mystrides_(i);
11774             update += count;
11775             if (final) {
11776                 start_index_((i+1)) = update;
11777             }
11778 
11779         });
11780     #else
11781     setup_start_indices_functor setup_execution_functor(start_index_, mystrides_);
11782     Kokkos::parallel_scan("StartValuesSetup", dim1_,setup_execution_functor);
11783     #endif
11784 
11785     //compute length of the storage
11786     #ifdef HAVE_CLASS_LAMBDA
11787     Kokkos::parallel_reduce("LengthSetup", dim1_, KOKKOS_CLASS_LAMBDA(const int i, int& update) {
11788             // Load old value in case we update it before accumulating
11789             update += mystrides_(i);
11790         }, length_);
11791     #else
11792     setup_length_functor length_functor(mystrides_);
11793     Kokkos::parallel_reduce("LengthSetup", dim1_, length_functor, length_);
11794     #endif
11795 
11796     //allocate view
11797     array_ = TArray1D(array_tag_string, length_);
11798 }
11799 
11800 // A method to return the stride size
11801 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits, typename ILayout>
11802 KOKKOS_INLINE_FUNCTION
11803 size_t RaggedRightArrayKokkos<T,Layout,ExecSpace,MemoryTraits,ILayout>::stride(size_t i) const {
11804     // Ensure that i is within bounds
11805     assert(i < (dim1_) && "i is greater than dim1_ in RaggedRightArray");
11806     return mystrides_(i);
11807 }
11808 
11809 // Method to build the stride (non-Kokkos push back)
11810 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits, typename ILayout>
11811 KOKKOS_INLINE_FUNCTION
11812 size_t& RaggedRightArrayKokkos<T,Layout,ExecSpace,MemoryTraits,ILayout>::build_stride(const size_t i)
      const {
11813     return start_index_(i+1);
11814 }
11815 
11816 // Method to finalize stride
11817 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits, typename ILayout>
11818 KOKKOS_INLINE_FUNCTION
11819 void RaggedRightArrayKokkos<T,Layout,ExecSpace,MemoryTraits,ILayout>::stride_finalize() const {
11820 
11821     #ifdef HAVE_CLASS_LAMBDA
11822     Kokkos::parallel_scan("StartValues", dim1_, KOKKOS_CLASS_LAMBDA(const int i, int& update, const
      bool final) {
11823             // Load old value in case we update it before accumulating
11824             const size_t count = start_index_(i+1);
11825             update += count;
11826             if (final) {
11827                 start_index_((i+1)) = update;
11828             }
11829 
11830         });
11831     #else
11832     finalize_stride_functor execution_functor(start_index_);
```

```
11833     Kokkos::parallel_scan("StartValues", dim1_,execution_functor);
11834     #endif
11835     Kokkos::fence();
11836 }
11837
11838
11839 // Overload operator() to access data as array(i,j)
11840 // where i=[0:N-1], j=[0:stride(i)]
11841 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits, typename ILayout>
11842 KOKKOS_INLINE_FUNCTION
11843 T& RaggedRightArrayKokkos<T,Layout,ExecSpace,MemoryTraits,ILayout>::operator()(size_t i, size_t j)
       const {
11844     // Get the 1D array index
11845     size_t start = start_index_(i);
11846
11847     // asserts
11848     assert(i < dim1_ && "i is out of dim1 bounds in RaggedRightArrayKokkos");  // die if >= dim1
11849     assert(j < stride(i) && "j is out of stride bounds in RaggedRightArrayKokkos");  // die if >=
       stride
11850
11851     return array_(j + start);
11852 } // End operator()
11853
11854 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits, typename ILayout>
11855 KOKKOS_INLINE_FUNCTION
11856 T* RaggedRightArrayKokkos<T,Layout,ExecSpace,MemoryTraits,ILayout>::pointer() {
11857     return array_.data();
11858 }
11859
11860
11861 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits, typename ILayout>
11862 KOKKOS_INLINE_FUNCTION
11863 RaggedRightArrayKokkos<T,Layout,ExecSpace,MemoryTraits,ILayout> &
       RaggedRightArrayKokkos<T,Layout,ExecSpace,MemoryTraits,ILayout>::
11864   operator= (const RaggedRightArrayKokkos<T,Layout,ExecSpace,MemoryTraits,ILayout> &temp) {
11865
11866   if (this != &temp) {
11867       /*
11868     SArray1D tempdim = SArray1D("tempdim", 1);
11869     auto h_tempdim = HostMirror(tempdim);
11870     Kokkos::parallel_for("StrideDim", 1, KOKKOS_CLASS_LAMBDA(const int&) {
11871             tempdim(0)  = strides_array.size();
11872             //dim1_  = strides_array.size();
11873         });
11874     Kokkos::fence();
11875     deep_copy(h_tempdim, tempdim);
11876     dim1_ = h_tempdim(0);
11877     */
11878     dim1_ = temp.dim1_;
11879
11880     // Create and initialize the starting index of the entries in the 1D array
11881     start_index_ = temp.start_index_;
11882     //start_index_(0) = 0; // the 1D array starts at 0
11883
11884     /*
11885     size_t * h_start_index = new size_t [dim1_+1];
11886     h_start_index[0] = 0;
11887     size_t * herenow = new size_t [2];
11888     herenow[0] = 1;
11889     herenow[1] = 2;
11890     size_t count = 0;
11891     for (size_t i = 0; i < dim1_; i++){
11892         count += herenow[i];
11893         h_start_index[(i + 1)] = count;
11894         printf("%d) Start check %ld\n", i, h_start_index[i]);
11895     } // end for i
11896     */
11897     /*
11898     SArray1D templen = SArray1D("templen", 1);
11899     auto h_templen = Kokkos::create_mirror_view(templen);
11900     #ifdef HAVE_CLASS_LAMBDA
11901     Kokkos::parallel_for("ArrayLength", 1, KOKKOS_CLASS_LAMBDA(const int&) {
11902             templen(0) = start_index_(dim1_);
11903             //length_ = start_index_(dim1_);
11904         });
11905     #else
11906     templen_functor templen_execution_functor(templen);
11907     Kokkos::parallel_for("ArrayLength", 1, templen_execution_functor);
11908     #endif
11909     Kokkos::fence();
11910     Kokkos::deep_copy(h_templen, templen);
11911     if (h_templen(0) != 0)
11912         length_ = h_templen(0);
11913     else
11914     */
11915     length_ = temp.length_;
11916
```

```
11917
11918     //printf("Length %ld\n", length_);
11919
11920     //Kokkos::parallel_for("StartCheck", dim1_+1, KOKKOS_CLASS_LAMBDA(const int i) {
11921     //        printf("%d) Start %ld\n", i, start_index_(i));
11922     //    });
11923     //Kokkos::fence();
11924
11925     array_ = temp.array_;
11926     mystrides_ = temp.mystrides_;
11927
11928     /*
11929         dim1_ = temp.dim1_;
11930         length_ = temp.length_;
11931         start_index_ = SArray1D("start_index_", dim1_ + 1);
11932         Kokkos::parallel_for("EqualOperator", dim1_+1, KOKKOS_CLASS_LAMBDA(const int j) {
11933                 start_index_(j) = temp.start_index_(j);
11934             });
11935         //for (int j = 0; j < dim1_; j++) {
11936         //    start_index_(j) = temp.start_index_(j);
11937         //}
11938         array_ = TArray1D("array_", length_);
11939     */
11940   }
11941
11942     return *this;
11943 }
11944
11945 //return the stored Kokkos view
11946 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits, typename ILayout>
11947 KOKKOS_INLINE_FUNCTION
11948 Kokkos::View<T*, Layout, ExecSpace, MemoryTraits>
       RaggedRightArrayKokkos<T,Layout,ExecSpace,MemoryTraits,ILayout>::get_kokkos_view() {
11949     return array_;
11950 }
11951
11952 // Destructor
11953 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits, typename ILayout>
11954 KOKKOS_INLINE_FUNCTION
11955 RaggedRightArrayKokkos<T,Layout,ExecSpace,MemoryTraits,ILayout>::~RaggedRightArrayKokkos() { }
11956
11958 // End of RaggedRightArrayKokkos
11960
11964 template <typename T, typename Layout = DefaultLayout, typename ExecSpace = DefaultExecSpace, typename
       MemoryTraits = void, typename ILayout = Layout>
11965 class RaggedRightArrayofVectorsKokkos {
11966
11967     using TArray1D = Kokkos::View<T*, Layout, ExecSpace, MemoryTraits>;
11968     using SArray1D = Kokkos::View<size_t *,Layout, ExecSpace, MemoryTraits>;
11969     using Strides1D = Kokkos::View<size_t *,ILayout, ExecSpace, MemoryTraits>;
11970
11971 private:
11972     TArray1D array_;
11973
11974     size_t dim1_, vector_dim_;
11975     size_t length_;
11976
11977 public:
11978     // Default constructor
11979     RaggedRightArrayofVectorsKokkos();
11980
11981     //--- 2D array access of a ragged right array ---
11982
11983     // Overload constructor for a CArrayKokkos
11984     RaggedRightArrayofVectorsKokkos(CArrayKokkos<size_t,ILayout,ExecSpace,MemoryTraits>
       &strides_array, size_t vector_dim,
11985                                     const std::string& tag_string = DEFAULTSTRINGARRAY );
11986
11987     // Overload constructor for a ViewCArray
11988     RaggedRightArrayofVectorsKokkos(ViewCArray<size_t> &strides_array, size_t vector_dim, const
       std::string& tag_string = DEFAULTSTRINGARRAY);
11989
11990     // Overloaded constructor for a traditional array
11991     RaggedRightArrayofVectorsKokkos(size_t* strides_array, size_t some_dim1, size_t vector_dim, const
       std::string& tag_string = DEFAULTSTRINGARRAY);
11992
11993     // A method to return the stride size
11994     KOKKOS_INLINE_FUNCTION
11995     size_t stride(size_t i) const;
11996
11997     // A method to increase the number of column entries, i.e.,
11998     // the stride size. Used with the constructor for building
11999     // the stride_array dynamically.
12000     // DO NOT USE with the constructures with a strides_array
12001     KOKKOS_INLINE_FUNCTION
12002     size_t& build_stride(const size_t i) const;
12003
```

```
12004     KOKKOS_INLINE_FUNCTION
12005     void stride_finalize() const;
12006
12007     // Overload operator() to access data as array(i,j)
12008     // where i=[0:N-1], j=[stride(i)]
12009     KOKKOS_INLINE_FUNCTION
12010     T& operator()(size_t i, size_t j, size_t k) const;
12011
12012     // method to return total size
12013     KOKKOS_INLINE_FUNCTION
12014     size_t size(){
12015       return length_;
12016     }
12017
12018     //setup start indices
12019     void data_setup(const std::string& tag_string);
12020
12021     KOKKOS_INLINE_FUNCTION
12022     T* pointer();
12023
12024     //return the view
12025     KOKKOS_INLINE_FUNCTION
12026     TArray1D get_kokkos_view();
12027
12028     // Kokkos views of strides and start indices
12029     Strides1D mystrides_;
12030     SArray1D start_index_;
12031
12032     KOKKOS_INLINE_FUNCTION
12033     RaggedRightArrayofVectorsKokkos& operator= (const RaggedRightArrayofVectorsKokkos &temp);
12034
12035     //functors for kokkos execution policies
12036     //initialize start indices view
12037     class init_start_indices_functor{
12038       public:
12039       SArray1D mystart_index_;
12040       init_start_indices_functor(SArray1D tempstart_index_){
12041         mystart_index_ = tempstart_index_;
12042       }
12043       KOKKOS_INLINE_FUNCTION void operator()(const int index) const {
12044         mystart_index_(index) = 0;
12045       }
12046     };
12047
12048     //setup start indices view
12049     class setup_start_indices_functor{
12050         public:
12051         SArray1D mystart_index_;
12052         Strides1D mytemp_strides_;
12053         size_t myvector_dim_;
12054         setup_start_indices_functor(SArray1D tempstart_index_, Strides1D temp_strides_, size_t
      myvector_dim){
12055           mystart_index_ = tempstart_index_;
12056           mytemp_strides_ = temp_strides_;
12057           myvector_dim_ = myvector_dim;
12058         }
12059         KOKKOS_INLINE_FUNCTION void operator()(const int index, int& update, bool final) const {
12060           // Load old value in case we update it before accumulating
12061             const size_t count = mytemp_strides_(index)*myvector_dim_;
12062             update += count;
12063             if (final) {
12064                 mystart_index_((index+1)) = update;
12065             }
12066         }
12067     };
12068
12069     //setup length of view
12070     class setup_length_functor{
12071         public:
12072         //kokkos needs this typedef named
12073         typedef size_t value_type;
12074         // This is helpful for determining the right index type,
12075         // especially if you expect to need a 64-bit index.
12076         //typedef Kokkos::View<size_t*>::size_type size_type;
12077
12078         Strides1D mytemp_strides_;
12079         size_t myvector_dim_;
12080
12081         setup_length_functor(Strides1D temp_strides_, size_t myvector_dim){
12082           mytemp_strides_ = temp_strides_;
12083           myvector_dim_ = myvector_dim;
12084         }
12085         KOKKOS_INLINE_FUNCTION void operator()(const int index, size_t& update) const {
12086             //const size_t count = mytemp_strides_(index)*myvector_dim_;
12087             update += mytemp_strides_(index)*myvector_dim_;;
12088         }
12089     };
```

```
12090
12091     //sets final 1D array size
12092     class finalize_stride_functor{
12093         public:
12094         SArray1D mystart_index_;
12095         finalize_stride_functor(SArray1D tempstart_index_){
12096           mystart_index_ = tempstart_index_;
12097         }
12098         KOKKOS_INLINE_FUNCTION void operator()(const int index, int& update, bool final) const {
12099           // Load old value in case we update it before accumulating
12100             const size_t count = mystart_index_(index+1);
12101             update += count;
12102             if (final) {
12103                 mystart_index_((index+1)) = update;
12104             }
12105         }
12106     };
12107
12108     // Destructor
12109     KOKKOS_INLINE_FUNCTION
12110     ~RaggedRightArrayofVectorsKokkos ( );
12111 }; // End of RaggedRightArrayofVectorsKokkos
12112
12113 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits, typename ILayout>
12114
      RaggedRightArrayofVectorsKokkos<T,Layout,ExecSpace,MemoryTraits,ILayout>::RaggedRightArrayofVectorsKokkos()
      {}
12115
12116 // Overloaded constructor
12117 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits, typename ILayout>
12118
      RaggedRightArrayofVectorsKokkos<T,Layout,ExecSpace,MemoryTraits,ILayout>::RaggedRightArrayofVectorsKokkos(CArrayKokkos<

12119
         &strides_array, size_t vector_dim,
12120
         const std::string& tag_string) {
12121     //mystrides_.assign_data(strides_array.pointer());
12122     vector_dim_ = vector_dim;
12123     mystrides_ = strides_array.get_kokkos_view();
12124     dim1_ = strides_array.extent();
12125     data_setup(tag_string);
12126 } // End constructor
12127
12128 /*
12129 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits, typename ILayout>
12130
      RaggedRightArrayofVectorsKokkos<T,Layout,ExecSpace,MemoryTraits>::RaggedRightArrayofVectorsKokkos(CArrayKokkos<size_t,

12131
      &strides_array, size_t vector_dim) {
12132     //mystrides_.assign_data(strides_array.pointer());
12133     vector_dim_ = vector_dim;
12134     mystrides_ = strides_array;
12135     dim1_ = strides_array.extent();
12136 } // End constructor
12137 */
12138
12139 // Overloaded constructor
12140 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits, typename ILayout>
12141
      RaggedRightArrayofVectorsKokkos<T,Layout,ExecSpace,MemoryTraits,ILayout>::RaggedRightArrayofVectorsKokkos(ViewCArray<s
      &strides_array, size_t vector_dim,
12142
         const std::string& tag_string) {
12143 } // End constructor
12144
12145 // Overloaded constructor
12146 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits, typename ILayout>
12147
      RaggedRightArrayofVectorsKokkos<T,Layout,ExecSpace,MemoryTraits,ILayout>::RaggedRightArrayofVectorsKokkos(size_t*
      strides_array, size_t some_dim1, size_t vector_dim,
12148
         const std::string& tag_string) {
12149     vector_dim_ = vector_dim;
12150     mystrides_.assign_data(strides_array);
12151     dim1_ = some_dim1;
12152     data_setup(tag_string);
12153 } // End constructor
12154
12155 //setup start indices
12156 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits, typename ILayout>
12157 void RaggedRightArrayofVectorsKokkos<T,Layout,ExecSpace,MemoryTraits,ILayout>::data_setup(const
      std::string& tag_string) {
12158
12159     //allocate start indices
12160     std::string append_indices_string("start_indices");
```

```
12161     std::string append_array_string("array");
12162     std::string temp_copy_string = tag_string;
12163     std::string start_index_tag_string = temp_copy_string.append(append_indices_string);
12164     temp_copy_string = tag_string;
12165     std::string array_tag_string = temp_copy_string.append(append_array_string);
12166 
12167     start_index_ = SArray1D(start_index_tag_string,dim1_ + 1);
12168     #ifdef HAVE_CLASS_LAMBDA
12169     Kokkos::parallel_for("StartValuesInit", dim1_+1, KOKKOS_CLASS_LAMBDA(const int i) {
12170       start_index_((i) = 0;
12171     });
12172     #else
12173     init_start_indices_functor execution_functor(start_index_);
12174     Kokkos::parallel_for("StartValuesInit", dim1_+1,execution_functor);
12175     #endif
12176 
12177     #ifdef HAVE_CLASS_LAMBDA
12178     Kokkos::parallel_scan("StartValuesSetup", dim1_, KOKKOS_CLASS_LAMBDA(const int i, int& update,
      const bool final) {
12179             // Load old value in case we update it before accumulating
12180             const size_t count = mystrides_(i)*vector_dim_;
12181             update += count;
12182             if (final) {
12183                 start_index_((i+1)) = update;
12184             }
12185 
12186         });
12187     #else
12188     setup_start_indices_functor setup_execution_functor(start_index_, mystrides_, vector_dim_);
12189     Kokkos::parallel_scan("StartValuesSetup", dim1_,setup_execution_functor);
12190     #endif
12191 
12192     //compute length of the storage
12193     #ifdef HAVE_CLASS_LAMBDA
12194     Kokkos::parallel_reduce("LengthSetup", dim1_, KOKKOS_CLASS_LAMBDA(const int i, int& update) {
12195             // Load old value in case we update it before accumulating
12196             update += mystrides_(i)*vector_dim_;
12197         }, length_);
12198     #else
12199     setup_length_functor length_functor(mystrides_, vector_dim_);
12200     Kokkos::parallel_reduce("LengthSetup", dim1_, length_functor,length_);
12201     #endif
12202 
12203     //allocate view
12204     array_ = TArray1D(array_tag_string, length_);
12205 }
12206 
12207 // A method to return the stride size
12208 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits, typename ILayout>
12209 KOKKOS_INLINE_FUNCTION
12210 size_t RaggedRightArrayofVectorsKokkos<T,Layout,ExecSpace,MemoryTraits,ILayout>::stride(size_t i)
      const {
12211     // Ensure that i is within bounds
12212     assert(i < (dim1_) && "i is greater than dim1_ in RaggedRightArray");
12213     return mystrides_(i);
12214 }
12215 
12216 // Method to build the stride (non-Kokkos push back)
12217 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits, typename ILayout>
12218 KOKKOS_INLINE_FUNCTION
12219 size_t& RaggedRightArrayofVectorsKokkos<T,Layout,ExecSpace,MemoryTraits,ILayout>::build_stride(const
      size_t i) const {
12220     return start_index_(i+1);
12221 }
12222 
12223 // Method to finalize stride
12224 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits, typename ILayout>
12225 KOKKOS_INLINE_FUNCTION
12226 void RaggedRightArrayofVectorsKokkos<T,Layout,ExecSpace,MemoryTraits,ILayout>::stride_finalize() const
      {
12227 
12228     #ifdef HAVE_CLASS_LAMBDA
12229     Kokkos::parallel_scan("StartValues", dim1_, KOKKOS_CLASS_LAMBDA(const int i, int& update, const
      bool final) {
12230             // Load old value in case we update it before accumulating
12231             const size_t count = start_index_(i+1);
12232             update += count;
12233             if (final) {
12234                 start_index_((i+1)) = update;
12235             }
12236 
12237         });
12238     #else
12239     finalize_stride_functor execution_functor(start_index_);
12240     Kokkos::parallel_scan("StartValues", dim1_,execution_functor);
12241     #endif
12242     Kokkos::fence();
```

```
12243 }
12244
12245
12246 // Overload operator() to access data as array(i,j)
12247 // where i=[0:N-1], j=[0:stride(i)]
12248 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits, typename ILayout>
12249 KOKKOS_INLINE_FUNCTION
12250 T& RaggedRightArrayofVectorsKokkos<T,Layout,ExecSpace,MemoryTraits,ILayout>::operator()(size_t i,
      size_t j, size_t k) const {
12251     // Get the 1D array index
12252     size_t start = start_index_(i);
12253
12254     // asserts
12255     assert(i < dim1_ && "i is out of dim1 bounds in RaggedRightArrayKokkos");  // die if >= dim1
12256     assert(j < stride(i) && "j is out of stride bounds in RaggedRightArrayKokkos");  // die if >=
      stride
12257     assert(j < vector_dim_ && "k is out of vector_dim bounds in RaggedRightArrayKokkos");  // die if
      >= vector_dim
12258
12259     return array_(j*vector_dim_ + start + k);
12260 } // End operator()
12261
12262 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits, typename ILayout>
12263 KOKKOS_INLINE_FUNCTION
12264 T* RaggedRightArrayofVectorsKokkos<T,Layout,ExecSpace,MemoryTraits,ILayout>::pointer() {
12265     return array_.data();
12266 }
12267
12268
12269 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits, typename ILayout>
12270 KOKKOS_INLINE_FUNCTION
12271 RaggedRightArrayofVectorsKokkos<T,Layout,ExecSpace,MemoryTraits,ILayout> &
      RaggedRightArrayofVectorsKokkos<T,Layout,ExecSpace,MemoryTraits,ILayout>::
12272   operator= (const RaggedRightArrayofVectorsKokkos<T,Layout,ExecSpace,MemoryTraits,ILayout> &temp) {
12273
12274   if (this != &temp) {
12275     dim1_ = temp.dim1_;
12276     vector_dim_ = temp.vector_dim_;
12277
12278     // Create and initialize the starting index of the entries in the 1D array
12279     start_index_ = temp.start_index_;
12280     length_ = temp.length_;
12281
12282     array_ = temp.array_;
12283     mystrides_ = temp.mystrides_;
12284   }
12285
12286     return *this;
12287 }
12288
12289 //return the stored Kokkos view
12290 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits, typename ILayout>
12291 KOKKOS_INLINE_FUNCTION
12292 Kokkos::View<T*, Layout, ExecSpace, MemoryTraits>
      RaggedRightArrayofVectorsKokkos<T,Layout,ExecSpace,MemoryTraits,ILayout>::get_kokkos_view() {
12293     return array_;
12294 }
12295
12296 // Destructor
12297 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits, typename ILayout>
12298 KOKKOS_INLINE_FUNCTION
12299
      RaggedRightArrayofVectorsKokkos<T,Layout,ExecSpace,MemoryTraits,ILayout>::~RaggedRightArrayofVectorsKokkos()
      { }
12300
12302 // End of RaggedRightArrayofVectorsKokkos
12304
12308 template <typename T, typename Layout = DefaultLayout, typename ExecSpace = DefaultExecSpace,
12309           typename MemoryTraits = void, typename ILayout = Layout>
12310 class RaggedDownArrayKokkos {
12311
12312     using TArray1D = Kokkos::View<T*, Layout, ExecSpace, MemoryTraits>;
12313     using SArray1D = Kokkos::View<size_t *, Layout, ExecSpace, MemoryTraits>;
12314     using Strides1D = Kokkos::View<size_t *, ILayout, ExecSpace, MemoryTraits>;
12315
12316 private:
12317     TArray1D array_;
12318
12319     size_t dim2_;
12320     size_t length_;
12321
12322 public:
12323     // Default constructor
12324     RaggedDownArrayKokkos();
12325
12326     //--- 2D array access of a ragged right array ---
12327
```

```
12328     // Overload constructor for a CArray
12329     RaggedDownArrayKokkos(CArrayKokkos<size_t, Layout, ExecSpace, MemoryTraits> &strides_array, const
      std::string& tag_string = DEFAULTSTRINGARRAY);
12330
12331     // Overload constructor for a ViewCArray
12332     RaggedDownArrayKokkos(ViewCArray<size_t> &strides_array, const std::string& tag_string =
      DEFAULTSTRINGARRAY);
12333
12334     // Overloaded constructor for a traditional array
12335     RaggedDownArrayKokkos(size_t* strides_array, size_t some_dim2, const std::string& tag_string =
      DEFAULTSTRINGARRAY);
12336
12337     // A method to return the stride size
12338     KOKKOS_INLINE_FUNCTION
12339     size_t stride(size_t j) const;
12340
12341     //setup start indices
12342     void data_setup(const std::string& tag_string);
12343
12344     // Overload operator() to access data as array(i,j)
12345     // where i=[0:N-1], j=[stride(i)]
12346     KOKKOS_INLINE_FUNCTION
12347     T& operator()(size_t i, size_t j) const;
12348
12349     KOKKOS_INLINE_FUNCTION
12350     T* pointer();
12351
12352     //return the view
12353     KOKKOS_INLINE_FUNCTION
12354     TArray1D get_kokkos_view();
12355
12356     KOKKOS_INLINE_FUNCTION
12357     RaggedDownArrayKokkos& operator= (const RaggedDownArrayKokkos &temp);
12358
12359     // Kokkos views of strides and start indices
12360     Strides1D mystrides_;
12361     SArray1D start_index_;
12362
12363     //functors for kokkos execution policies
12364     //initialize start indices view
12365     class init_start_indices_functor{
12366       public:
12367       SArray1D mystart_index_;
12368       init_start_indices_functor(SArray1D tempstart_index_){
12369         mystart_index_ = tempstart_index_;
12370       }
12371       KOKKOS_INLINE_FUNCTION void operator()(const int index) const {
12372         mystart_index_(index) = 0;
12373       }
12374     };
12375
12376     //setup start indices view
12377     class setup_start_indices_functor{
12378         public:
12379         SArray1D mystart_index_;
12380         Strides1D mytemp_strides_;
12381         setup_start_indices_functor(SArray1D tempstart_index_, Strides1D temp_strides_){
12382           mystart_index_ = tempstart_index_;
12383           mytemp_strides_ = temp_strides_;
12384         }
12385         KOKKOS_INLINE_FUNCTION void operator()(const int index, int& update, bool final) const {
12386           // Load old value in case we update it before accumulating
12387             const size_t count = mytemp_strides_(index);
12388             update += count;
12389             if (final) {
12390                 mystart_index_((index+1)) = update;
12391             }
12392         }
12393     };
12394
12395     //setup length of view
12396     class setup_length_functor{
12397         public:
12398         //kokkos needs this typedef named
12399         typedef size_t value_type;
12400         // This is helpful for determining the right index type,
12401         // especially if you expect to need a 64-bit index.
12402         //typedef Kokkos::View<size_t*>::size_type size_type;
12403         Strides1D mytemp_strides_;
12404         setup_length_functor(Strides1D temp_strides_){
12405           mytemp_strides_ = temp_strides_;
12406         }
12407         KOKKOS_INLINE_FUNCTION void operator()(const int index, size_t& update) const {
12408             //const size_t count = mytemp_strides_(index);
12409             update += mytemp_strides_(index);
12410         }
12411     };
```

```
12412
12413     // Destructor
12414     KOKKOS_INLINE_FUNCTION
12415     ~RaggedDownArrayKokkos ( );
12416 }; // End of RaggedDownArray
12417
12418 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits, typename ILayout>
12419 RaggedDownArrayKokkos<T,Layout,ExecSpace,MemoryTraits,ILayout>::RaggedDownArrayKokkos() {}
12420
12421 // Overloaded constructor
12422 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits, typename ILayout>
12423
       RaggedDownArrayKokkos<T,Layout,ExecSpace,MemoryTraits,ILayout>::RaggedDownArrayKokkos(CArrayKokkos<size_t,
       Layout, ExecSpace, MemoryTraits> &strides_array,
12424                                                                                         const std::string&
       tag_string) {
12425     mystrides_ = strides_array.get_kokkos_view();
12426     dim2_ = strides_array.extent();
12427     data_setup(tag_string);
12428 } // End constructor
12429
12430 // Overloaded constructor
12431 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits, typename ILayout>
12432
       RaggedDownArrayKokkos<T,Layout,ExecSpace,MemoryTraits,ILayout>::RaggedDownArrayKokkos(ViewCArray<size_t>
       &strides_array, const std::string& tag_string) {
12433 } // End constructor
12434
12435 // Overloaded constructor
12436 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits, typename ILayout>
12437 RaggedDownArrayKokkos<T,Layout,ExecSpace,MemoryTraits,ILayout>::RaggedDownArrayKokkos(size_t*
       strides_array, size_t some_dim2,
12438                                                                                         const std::string&
       tag_string) {
12439     mystrides_.assign_data(strides_array);
12440     dim2_ = some_dim2;
12441     data_setup(tag_string);
12442 } // End constructor
12443
12444 //setup start indices
12445 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits, typename ILayout>
12446 void RaggedDownArrayKokkos<T,Layout,ExecSpace,MemoryTraits,ILayout>::data_setup(const std::string&
       tag_string) {
12447     //allocate start indices
12448     std::string append_indices_string("start_indices");
12449     std::string append_array_string("array");
12450     std::string temp_copy_string = tag_string;
12451     std::string start_index_tag_string = temp_copy_string.append(append_indices_string);
12452     temp_copy_string = tag_string;
12453     std::string array_tag_string = temp_copy_string.append(append_array_string);
12454
12455     start_index_ = SArray1D(start_index_tag_string,dim2_ + 1);
12456     #ifdef HAVE_CLASS_LAMBDA
12457     Kokkos::parallel_for("StartValuesInit", dim2_+1, KOKKOS_CLASS_LAMBDA(const int i) {
12458       start_index_((i) = 0;
12459     });
12460     #else
12461     init_start_indices_functor execution_functor(start_index_);
12462     Kokkos::parallel_for("StartValuesInit", dim2_+1,execution_functor);
12463     #endif
12464
12465     #ifdef HAVE_CLASS_LAMBDA
12466     Kokkos::parallel_scan("StartValuesSetup", dim2_, KOKKOS_CLASS_LAMBDA(const int i, int& update,
       const bool final) {
12467             // Load old value in case we update it before accumulating
12468             const size_t count = mystrides_(i);
12469             update += count;
12470             if (final) {
12471                 start_index_((i+1)) = update;
12472             }
12473
12474         });
12475     #else
12476     setup_start_indices_functor setup_execution_functor(start_index_, mystrides_);
12477     Kokkos::parallel_scan("StartValuesSetup", dim2_,setup_execution_functor);
12478     #endif
12479
12480     //compute length of the storage
12481     #ifdef HAVE_CLASS_LAMBDA
12482     Kokkos::parallel_reduce("LengthSetup", dim2_, KOKKOS_CLASS_LAMBDA(const int i, int& update) {
12483             // Load old value in case we update it before accumulating
12484             update += mystrides_(i);
12485         }, length_);
12486     #else
12487     setup_length_functor length_functor(mystrides_);
12488     Kokkos::parallel_reduce("LengthSetup", dim2_, length_functor, length_);
12489     #endif
```

```
12490
12491     //allocate view
12492     array_ = TArray1D(array_tag_string, length_);
12493 }
12494
12495 // A method to return the stride size
12496 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits, typename ILayout>
12497 KOKKOS_INLINE_FUNCTION
12498 size_t RaggedDownArrayKokkos<T,Layout,ExecSpace,MemoryTraits,ILayout>::stride(size_t j) const {
12499     // Ensure that j is within bounds
12500     assert(j < (dim2_) && "j is greater than dim1_ in RaggedDownArray");
12501
12502     return mystrides_(j);
12503 }
12504
12505 // Overload operator() to access data as array(i,j)
12506 // where i=[0:N-1], j=[0:stride(i)]
12507 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits, typename ILayout>
12508 KOKKOS_INLINE_FUNCTION
12509 T& RaggedDownArrayKokkos<T,Layout,ExecSpace,MemoryTraits,ILayout>::operator()(size_t i, size_t j)
      const {
12510     // Get the 1D array index
12511     size_t start = start_index_(j);
12512
12513     // asserts
12514     assert(i < stride(j) && "i is out of stride bounds in RaggedDownArrayKokkos");  // die if >=
      stride
12515     assert(j < dim2_ && "j is out of dim1 bounds in RaggedDownArrayKokkos");  // die if >= dim1
12516
12517     return array_(i + start);
12518 } // End operator()
12519
12520 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits, typename ILayout>
12521 KOKKOS_INLINE_FUNCTION
12522 RaggedDownArrayKokkos<T,Layout,ExecSpace,MemoryTraits,ILayout>&
      RaggedDownArrayKokkos<T,Layout,ExecSpace,MemoryTraits,ILayout>::
12523 operator= (const RaggedDownArrayKokkos<T,Layout,ExecSpace,MemoryTraits,ILayout> &temp) {
12524
12525   if (this != &temp) {
12526       /*
12527     SArray1D tempdim = SArray1D("tempdim", 1);
12528     auto h_tempdim = HostMirror(tempdim);
12529     Kokkos::parallel_for("StrideDim", 1, KOKKOS_CLASS_LAMBDA(const int&) {
12530             tempdim(0)  = strides_array.size();
12531             //dim1_  = strides_array.size();
12532         });
12533     Kokkos::fence();
12534     deep_copy(h_tempdim, tempdim);
12535     dim1_ = h_tempdim(0);
12536     */
12537     dim2_ = temp.dim2_;
12538
12539     // Create and initialize the starting index of the entries in the 1D array
12540     start_index_ = temp.start_index_;
12541     /*
12542     //start_index_(0) = 0; // the 1D array starts at 0
12543     #ifdef HAVE_CLASS_LAMBDA
12544     Kokkos::parallel_for("StartFirst", 1, KOKKOS_CLASS_LAMBDA(const int&) {
12545             start_index_(0) = 0;
12546         });
12547     #else
12548     assignment_init_functor init_execution_functor;
12549     Kokkos::parallel_for("StartFirst", 1, init_execution_functor);
12550     #endif
12551     Kokkos::fence();
12552
12553     // Loop over to find the total length of the 1D array to
12554     // represent the ragged-right array and set the starting 1D index
12555     #ifdef HAVE_CLASS_LAMBDA
12556     Kokkos::parallel_scan("StartValues", dim2_, KOKKOS_CLASS_LAMBDA(const int j, double& update, const
      bool final) {
12557             // Load old value in case we update it before accumulating
12558             const size_t count = temp.mystrides_[j];
12559             update += count;
12560             if (final) {
12561                 start_index_((j+1)) = update;
12562             }
12563
12564         });
12565     #else
12566     assignment_scan_functor scan_execution_functor(temp);
12567     Kokkos::parallel_scan("StartValues", dim2_, scan_execution_functor);
12568     #endif
12569     Kokkos::fence();
12570     */
12571     /*
12572     size_t * h_start_index = new size_t [dim1_+1];
```

```
12573     h_start_index[0] = 0;
12574     size_t * herenow = new size_t [2];
12575     herenow[0] = 1;
12576     herenow[1] = 2;
12577     size_t count = 0;
12578     for (size_t i = 0; i < dim1_; i++){
12579         count += herenow[i];
12580         h_start_index[(i + 1)] = count;
12581         printf("%d) Start check %ld\n", i, h_start_index[i]);
12582     } // end for i
12583     */
12584     /*
12585     SArray1D templen = SArray1D("templen", 1);
12586     auto h_templen = Kokkos::create_mirror_view(templen);
12587     #ifdef HAVE_CLASS_LAMBDA
12588     Kokkos::parallel_for("ArrayLength", 1, KOKKOS_CLASS_LAMBDA(const int&) {
12589             templen(0) = start_index_(dim2_);
12590             //length_ = start_index_(dim2_);
12591         });
12592     #else
12593     templen_functor templen_execution_functor(templen);
12594     Kokkos::parallel_for("ArrayLength", 1, templen_execution_functor);
12595     #endif
12596     Kokkos::fence();
12597     deep_copy(h_templen, templen);
12598     length_ = h_templen(0);
12599
12600     printf("Length %ld\n", length_);
12601
12602     #ifdef HAVE_CLASS_LAMBDA
12603     Kokkos::parallel_for("StartCheck", dim2_+1, KOKKOS_CLASS_LAMBDA(const int j) {
12604             printf("%d) Start %ld\n", j, start_index_(j));
12605         });
12606     #else
12607     stride_check_functor check_execution_functor;
12608     Kokkos::parallel_for("StartCheck", dim2_+1, check_execution_functor);
12609     #endif
12610     Kokkos::fence();
12611     */
12612     length_ = temp.length_;
12613     array_ = temp.length_;
12614     mystrides_ = temp.mystrides_;
12615
12616     /*
12617         dim1_ = temp.dim1_;
12618         length_ = temp.length_;
12619         start_index_ = SArray1D("start_index_", dim1_ + 1);
12620         Kokkos::parallel_for("EqualOperator", dim1_+1, KOKKOS_CLASS_LAMBDA(const int j) {
12621                 start_index_(j) = temp.start_index_(j);
12622             });
12623         //for (int j = 0; j < dim1_; j++) {
12624         //    start_index_(j) = temp.start_index_(j);
12625         //}
12626         array_ = TArray1D("array_", length_);
12627     */
12628   }
12629
12630     return *this;
12631 }
12632
12633 //return the stored Kokkos view
12634 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits, typename ILayout>
12635 KOKKOS_INLINE_FUNCTION
12636 Kokkos::View<T*, Layout, ExecSpace, MemoryTraits>
      RaggedDownArrayKokkos<T,Layout,ExecSpace,MemoryTraits,ILayout>::get_kokkos_view() {
12637     return array_;
12638 }
12639
12640 // Destructor
12641 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits, typename ILayout>
12642 KOKKOS_INLINE_FUNCTION
12643 RaggedDownArrayKokkos<T,Layout,ExecSpace,MemoryTraits,ILayout>::~RaggedDownArrayKokkos() { }
12644
12646 // End of RaggedDownArrayKokkos
12648
12649 //11. DynamicRaggedRightArray
12650 template <typename T, typename Layout = DefaultLayout, typename ExecSpace = DefaultExecSpace, typename
      MemoryTraits = void>
12651 class DynamicRaggedRightArrayKokkos {
12652
12653     using TArray1D = Kokkos::View<T*, Layout, ExecSpace, MemoryTraits>;
12654     using SArray1D = Kokkos::View<size_t *,Layout, ExecSpace, MemoryTraits>;
12655
12656 private:
12657     // THIS WILL BE A GPU POINTER!
12658     SArray1D stride_;
12659     TArray1D array_;
```

```
12660
12661     size_t dim1_;
12662     size_t dim2_;
12663     size_t length_;
12664
12665 public:
12666     // Default constructor
12667     DynamicRaggedRightArrayKokkos ();
12668
12669     //--- 2D array access of a ragged right array ---
12670
12671     // overload constructor
12672     DynamicRaggedRightArrayKokkos (size_t dim1, size_t dim2, const std::string& tag_string =
      DEFAULTSTRINGARRAY);
12673
12674     // A method to return or set the stride size
12675     KOKKOS_INLINE_FUNCTION
12676     size_t& stride(size_t i) const;
12677
12678     // A method to return the size
12679     KOKKOS_INLINE_FUNCTION
12680     size_t size() const;
12681
12682     //return the view
12683     KOKKOS_INLINE_FUNCTION
12684     TArray1D get_kokkos_view();
12685
12686     // Overload operator() to access data as array(i,j),
12687     // where i=[0:N-1], j=[stride(i)]
12688     KOKKOS_INLINE_FUNCTION
12689     T& operator()(size_t i, size_t j) const;
12690
12691     // Overload copy assignment operator
12692     KOKKOS_INLINE_FUNCTION
12693     DynamicRaggedRightArrayKokkos& operator= (const DynamicRaggedRightArrayKokkos &temp);
12694
12695     //kokkos policy functors
12696
12697     //functors for kokkos execution policies
12698     //set strides to a constant value
12699     class set_strides_functor{
12700       public:
12701       SArray1D functor_strides_;
12702       size_t init_stride_;
12703       set_strides_functor(size_t init_stride, SArray1D temp_strides_){
12704         init_stride_ = init_stride;
12705         functor_strides_ = temp_strides_;
12706       }
12707       KOKKOS_INLINE_FUNCTION void operator()(const int index) const {
12708         functor_strides_(index) = init_stride_;
12709       }
12710     };
12711
12712     // Destructor
12713     KOKKOS_INLINE_FUNCTION
12714     ~DynamicRaggedRightArrayKokkos ();
12715 };
12716
12717 //nothing
12718 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
12719 DynamicRaggedRightArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::DynamicRaggedRightArrayKokkos () {}
12720
12721 // Overloaded constructor
12722 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
12723 DynamicRaggedRightArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::DynamicRaggedRightArrayKokkos (size_t
      dim1, size_t dim2, const std::string& tag_string) {
12724     // The dimensions of the array;
12725     dim1_  = dim1;
12726     dim2_  = dim2;
12727     length_ = dim1*dim2;
12728
12729     std::string append_stride_string("strides");
12730     std::string append_array_string("array");
12731     std::string temp_copy_string = tag_string;
12732     std::string strides_tag_string = temp_copy_string.append(append_stride_string);
12733     temp_copy_string = tag_string;
12734     std::string array_tag_string = temp_copy_string.append(append_array_string);
12735
12736     stride_ = SArray1D(strides_tag_string, dim1_);
12737     #ifdef HAVE_CLASS_LAMBDA
12738     Kokkos::parallel_for("StridesInit", dim1_, KOKKOS_CLASS_LAMBDA(const int i) {
12739       strides_((i) = 0;
12740     });
12741     #else
12742     set_strides_functor execution_functor(0, stride_);
12743     Kokkos::parallel_for("StridesInit", dim1_,execution_functor);
12744     #endif
```

```
12745
12746     //allocate view
12747     array_ = TArray1D(array_tag_string, length_);
12748 }
12749
12750 // A method to set the stride size for row i
12751 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
12752 KOKKOS_INLINE_FUNCTION
12753 size_t& DynamicRaggedRightArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::stride(size_t i) const {
12754     return stride_(i);
12755 }
12756
12757 //return size
12758 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
12759 KOKKOS_INLINE_FUNCTION
12760 size_t DynamicRaggedRightArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::size() const{
12761     return length_;
12762 }
12763
12764 // Overload operator() to access data as array(i,j),
12765 // where i=[0:N-1], j=[0:stride(i)]
12766 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
12767 KOKKOS_INLINE_FUNCTION
12768 T& DynamicRaggedRightArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::operator()(size_t i, size_t j)
       const {
12769     // Asserts
12770     assert(i < dim1_ && "i is out of dim1 bounds in DynamicRaggedRight");  // die if >= dim1
12771     assert(j < stride_(i) && "j is out of stride bounds in DynamicRaggedRight");  // die if >= dim2
12772     // Cannot assert on Kokkos View
12773     //assert(j < stride_[i] && "j is out of stride bounds in DynamicRaggedRight");  // die if >=
       stride
12774
12775     return array_(j + i*dim2_);
12776 }
12777
12778 //overload = operator
12779 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
12780 KOKKOS_INLINE_FUNCTION
12781 DynamicRaggedRightArrayKokkos<T,Layout,ExecSpace,MemoryTraits>&
12782        DynamicRaggedRightArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::operator= (const
       DynamicRaggedRightArrayKokkos<T,Layout,ExecSpace,MemoryTraits> &temp)
12783 {
12784
12785     if( this != &temp) {
12786         dim1_ = temp.dim1_;
12787         dim2_ = temp.dim2_;
12788         length_ = temp.length_;
12789         stride_ = temp.stride_;
12790         array_ = temp.array_;
12791         /*
12792         #ifdef HAVE_CLASS_LAMBDA
12793         Kokkos::parallel_for("StrideZeroOut", dim1_, KOKKOS_CLASS_LAMBDA(const int i) {
12794             stride_(i) = 0;
12795         });
12796         #else
12797         stride_zero_functor execution_functor;
12798         Kokkos::parallel_for("StrideZeroOut", dim1_, execution_functor);
12799         #endif
12800         */
12801     }
12802
12803     return *this;
12804 }
12805
12806 //return the stored Kokkos view
12807 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
12808 KOKKOS_INLINE_FUNCTION
12809 Kokkos::View<T*, Layout, ExecSpace, MemoryTraits>
       DynamicRaggedRightArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::get_kokkos_view() {
12810     return array_;
12811 }
12812
12813 // Destructor
12814 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
12815 KOKKOS_INLINE_FUNCTION
12816 DynamicRaggedRightArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::~DynamicRaggedRightArrayKokkos() {
12817 }
12818
12819
12820
12821
12822 //----end DynamicRaggedRightArray class definitions----
12823
12824
12825 //12. DynamicRaggedDownArray
12826
12827 template <typename T, typename Layout = DefaultLayout, typename ExecSpace = DefaultExecSpace, typename
```

```
      MemoryTraits = void>
12828 class DynamicRaggedDownArrayKokkos {
12829
12830     using TArray1D = Kokkos::View<T*, Layout, ExecSpace, MemoryTraits>;
12831     using SArray1D = Kokkos::View<size_t *,Layout, ExecSpace, MemoryTraits>;
12832
12833 private:
12834     SArray1D stride_;
12835     TArray1D array_;
12836
12837     size_t dim1_;
12838     size_t dim2_;
12839     size_t length_;
12840
12841 public:
12842     // Default constructor
12843     DynamicRaggedDownArrayKokkos ();
12844
12845     //--- 2D array access of a ragged right array ---
12846
12847     // overload constructor
12848     DynamicRaggedDownArrayKokkos (size_t dim1, size_t dim2, const std::string& tag_string =
      DEFAULTSTRINGARRAY);
12849
12850     // A method to return or set the stride size
12851     KOKKOS_INLINE_FUNCTION
12852     size_t& stride(size_t j) const;
12853
12854     // A method to return the size
12855     KOKKOS_INLINE_FUNCTION
12856     size_t size() const;
12857
12858     //return the view
12859     KOKKOS_INLINE_FUNCTION
12860     TArray1D get_kokkos_view();
12861
12862     // Overload operator() to access data as array(i,j),
12863     // where i=[stride(j)], j=[0:N-1]
12864     KOKKOS_INLINE_FUNCTION
12865     T& operator()(size_t i, size_t j) const;
12866
12867     // Overload copy assignment operator
12868     KOKKOS_INLINE_FUNCTION
12869     DynamicRaggedDownArrayKokkos& operator= (const DynamicRaggedDownArrayKokkos &temp);
12870
12871     //kokkos policy functors
12872     //set strides to 0 functor
12873     //set strides to a constant value
12874     class set_strides_functor{
12875       public:
12876       SArray1D functor_strides_;
12877       size_t init_stride_;
12878       set_strides_functor(size_t init_stride, SArray1D temp_strides_){
12879         init_stride_ = init_stride;
12880         functor_strides_ = temp_strides_;
12881       }
12882       KOKKOS_INLINE_FUNCTION void operator()(const int index) const {
12883         functor_strides_(index) = init_stride_;
12884       }
12885     };
12886
12887     // Destructor
12888     KOKKOS_INLINE_FUNCTION
12889     ~DynamicRaggedDownArrayKokkos ();
12890 };
12891
12892 //nothing
12893 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
12894 DynamicRaggedDownArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::DynamicRaggedDownArrayKokkos () {}
12895
12896 // Overloaded constructor
12897 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
12898 DynamicRaggedDownArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::DynamicRaggedDownArrayKokkos (size_t
      dim1, size_t dim2, const std::string& tag_string) {
12899     // The dimensions of the array;
12900     dim1_  = dim1;
12901     dim2_  = dim2;
12902     length_ = dim1*dim2;
12903
12904     std::string append_stride_string("strides");
12905     std::string append_array_string("array");
12906     std::string temp_copy_string = tag_string;
12907     std::string strides_tag_string = temp_copy_string.append(append_stride_string);
12908     temp_copy_string = tag_string;
12909     std::string array_tag_string = temp_copy_string.append(append_array_string);
12910
12911     stride_ = SArray1D(strides_tag_string, dim2_);
```

```
12912     #ifdef HAVE_CLASS_LAMBDA
12913     Kokkos::parallel_for("StridesInit", dim2_, KOKKOS_CLASS_LAMBDA(const int i) {
12914       strides_((i) = 0;
12915     });
12916     #else
12917     set_strides_functor execution_functor(0, stride_);
12918     Kokkos::parallel_for("StridesInit", dim2_,execution_functor);
12919     #endif
12920 
12921     //allocate view
12922     array_ = TArray1D(array_tag_string, length_);
12923 }
12924 
12925 // A method to set the stride size for column j
12926 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
12927 KOKKOS_INLINE_FUNCTION
12928 size_t& DynamicRaggedDownArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::stride(size_t j) const {
12929     return stride_(j);
12930 }
12931 
12932 //return size
12933 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
12934 KOKKOS_INLINE_FUNCTION
12935 size_t DynamicRaggedDownArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::size() const{
12936     return length_;
12937 }
12938 
12939 // overload operator () to access data as an array(i,j)
12940 // Note: i = 0:stride(j), j = 0:N-1
12941 
12942 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
12943 KOKKOS_INLINE_FUNCTION
12944 T& DynamicRaggedDownArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::operator()(size_t i, size_t j) const
       {
12945     // Asserts
12946     assert(j < dim2_ && "j is out of dim2 bounds in DynamicRaggedDownArrayKokkos");  // die if >= dim2
12947     assert(i < stride(j) && "i is out of stride bounds in DynamicRaggedDownArrayKokkos");  // die if
       >= stride(j)
12948     // Can't do this assert with a Kokkos View
12949     //assert(i < stride_[j] && "i is out of stride bounds in DynamicRaggedDownArrayKokkos");  // die
       if >= stride
12950 
12951     return array_(i + j*dim1_);
12952 }
12953 
12954 //overload = operator
12955 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
12956 KOKKOS_INLINE_FUNCTION
12957 DynamicRaggedDownArrayKokkos<T,Layout,ExecSpace,MemoryTraits>&
12958   DynamicRaggedDownArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::operator= (const
       DynamicRaggedDownArrayKokkos<T,Layout,ExecSpace,MemoryTraits> &temp)
12959 {
12960 
12961     if( this != &temp) {
12962         dim1_ = temp.dim1_;
12963         dim2_ = temp.dim2_;
12964         length_ = temp.length_;
12965         stride_ = temp.stride_;
12966         array_ = temp.array_;
12967         /*
12968         #ifdef HAVE_CLASS_LAMBDA
12969         Kokkos::parallel_for("StrideZeroOut", dim2_, KOKKOS_CLASS_LAMBDA(const int j) {
12970             stride_(j) = 0;
12971         });
12972         #else
12973         stride_zero_functor execution_functor;
12974         Kokkos::parallel_for("StrideZeroOut", dim2_, execution_functor);
12975         #endif
12976         */
12977     }
12978 
12979     return *this;
12980 }
12981 
12982 //return the stored Kokkos view
12983 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
12984 KOKKOS_INLINE_FUNCTION
12985 Kokkos::View<T*, Layout, ExecSpace, MemoryTraits>
       DynamicRaggedDownArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::get_kokkos_view() {
12986     return array_;
12987 }
12988 
12989 // Destructor
12990 template <typename T, typename Layout, typename ExecSpace, typename MemoryTraits>
12991 KOKKOS_INLINE_FUNCTION
12992 DynamicRaggedDownArrayKokkos<T,Layout,ExecSpace,MemoryTraits>::~DynamicRaggedDownArrayKokkos() {
12993 }
```

```
12994
12995
12996
12998 // Inherited Class Array
13000
13001 /*
13002 //template<class T, class Layout, class ExecSpace>
13003 template<typename T>
13004 class InheritedArray2L {
13005
13006     using TArray1D = Kokkos::View<T*, Layout, ExecSpace>;
13007
13008 private:
13009     size_t dim1_, length_;
13010
13011 public:
13012     TArray1D this_array_;
13013     typename Kokkos::View<T*, Layout, ExecSpace>::HostMirror  h_this_array_;
13014
13015     InheritedArray2L();
13016
13017     InheritedArray2L(size_t some_dim1);
13018
13019     KOKKOS_INLINE_FUNCTION
13020     T& operator()(size_t i, size_t dest) const;
13021
13022     template <typename U>
13023     void AllocateHost(size_t size, U *obj);
13024
13025     void AllocateGPU();
13026
13027     template <typename U, typename V>
13028     void InitModels(U *obj, V input);
13029
13030     template <typename U>
13031     void ClearModels(U obj);
13032
13033     InheritedArray2L& operator=(const InheritedArray2L& temp);
13034
13035     // GPU Method
13036     // Method that returns size
13037     KOKKOS_INLINE_FUNCTION
13038     size_t size();
13039
13040     // Host Method
13041     // Method that returns size
13042     size_t extent();
13043
13044     // Methods returns the raw pointer (most likely GPU) of the Kokkos View
13045     T* pointer();
13046
13047     // Deconstructor
13048     KOKKOS_INLINE_FUNCTION
13049     ~InheritedArray2L ();
13050 }; // End of InheritedArray2L
13051
13052 // Default constructor
13053 template <typename T>
13054 InheritedArray2L<T>::InheritedArray2L() {}
13055
13056 // Overloaded 1D constructor
13057 template <typename T>
13058 InheritedArray2L<T>::InheritedArray2L(size_t some_dim1) {
13059     using TArray1D = Kokkos::View<T*, Layout, ExecSpace>;
13060
13061     dim1_ = some_dim1;
13062     length_ = dim1_;
13063     this_array_ = TArray1D("this_array_", length_);
13064     h_this_array_ = Kokkos::create_mirror_view(this_array_);
13065 }
13066
13067 template <typename T>
13068 KOKKOS_INLINE_FUNCTION
13069 T& InheritedArray2L<T>::operator()(size_t i, size_t dest) const {
13070     assert(i < dim1_ && "i is out of bounds in InheritedArray2L 1D!");
13071     assert(dest < 2 && "dest is out of bounds in InheritedArray2L 1D!");
13072     if (dest == 0)
13073         return h_this_array_(i);
13074     else
13075         return this_array_(i);
13076 }
13077
13078 template <typename T>
13079 template <typename U>
13080 void InheritedArray2L<T>::AllocateHost(size_t size, U *obj) {
13081     obj = (U *) kmalloc(size);
13082 }
```

```
13083
13084 template <typename T>
13085 void InheritedArray2L<T>::AllocateGPU() {
13086     Kokkos::deep_copy(this_array_, h_this_array_);
13087 }
13088
13089 template <typename T>
13090 template <typename U, typename V>
13091 void InheritedArray2L<T>::InitModels(U *obj, V input) {
13092     Kokkos::parallel_for(
13093             "CreateObjects", 1, KOKKOS_CLASS_LAMBDA(const int&) {
13094                 new ((V *)obj) V{input};
13095             });
13096 }
13097
13098 template <typename T>
13099 template <typename U>
13100 void InheritedArray2L<T>::ClearModels(U obj) {
13101     Kokkos::parallel_for(
13102             "DestroyObjects", 1, KOKKOS_LAMBDA(const int&) {
13103               this_array_(0).obj->~U();
13104               this_array_(1).obj->~U();
13105             });
13106 }
13107
13108 template <typename T>
13109 InheritedArray2L<T>& InheritedArray2L<T>::operator= (const InheritedArray2L& temp) {
13110     using TArray1D = Kokkos::View<T *,Layout,ExecSpace>;
13111
13112     // Do nothing if the assignment is of the form x = x
13113     if (this != &temp) {
13114         dim1_ = temp.dim1_;
13115         length_ = temp.length_;
13116         this_array_ = TArray1D("this_array_", length_);
13117     }
13118
13119     return *this;
13120 }
13121
13122 // Return size
13123 template <typename T>
13124 KOKKOS_INLINE_FUNCTION
13125 size_t InheritedArray2L<T>::size() {
13126     return length_;
13127 }
13128
13129 template <typename T>
13130 size_t InheritedArray2L<T>::extent() {
13131     return length_;
13132 }
13133
13134 template <typename T>
13135 T* InheritedArray2L<T>::pointer() {
13136     return this_array_.data();
13137 }
13138
13139 template <typename T>
13140 KOKKOS_INLINE_FUNCTION
13141 InheritedArray2L<T>::~InheritedArray2L() {}
13142 */
13143
13145 // End of InheritedArray2L
13147
13148
13149 #endif
13150
13151
13152
13153
13154
13155
13156
13157 #endif // MATAR_H
```

Index