# LA-UR-22-24684

**Approved for public release; distribution is unlimited.**

**Title:** General purpose GPU programming made easy

**Author(s):** Morgan, Nathaniel Ray

**Intended for:** LANL presentation to staff and students

**Issued:** 2022-05-19

Los Alamos NATIONAL LABORATORY

# General purpose GPU programming made easy

**Nathaniel Morgan[1]**
**Contributors: Robert Robey[2], Caleb Yenusah[1], Adrian Diaz[1], Daniel Dunning[1,2], Jacob Moore[1], Tanya Tafolla[1], Eappen Nelluvelil[1], and Russel Marki [1]**

May 2022

[1]X-Computational Physics: Continuum Models and Numerical Methods

[2]X-Computational Physics: Eulerian Codes

LA-UR-22-YYYYYY

# Introduction

# GPUs are designed to perform mathematical operations in parallel

- CPUs are designed for intricate work flows and diverse applications
- GPUs have significantly more Arithmetic Logic Units (ALU) or float point units (FPUs) than CPUs, compare the green boxes in the Figure
- CPUs have more cache memory than GPUs

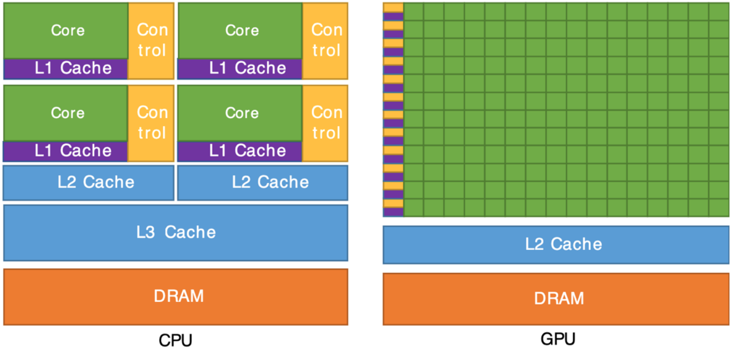
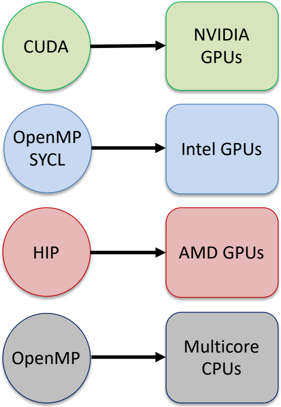



Figure: Comparison of GPU and CPU architectures[1]

[1] Image is from `https://cvw.cac.cornell.edu/GPUarch/gpu_characteristics`

# CPUs and GPUs have vendor specific fine-grained parallelism languages




Plus openCL and openACC

The Kokkos [2] performance portability library:

- Enables a single implementation to run on all architectures - replaces vendor specific languages
- Is based solely on C++11 (no special compiler language needed!)
- Has capabilities to manage access, allocation, and copying of data between CPU and GPU

**Goals for this work:**

- Enable easy adoption of data oriented programming (DOP) in codes
- Simplify the use of Kokkos in a code (C++ or Fortran) and extend it to support sparse data representations

# The C++ MATrix and ARray (MATAR) library is written using Kokkos for performance, portability, and productivity

- MATAR [1] is designed to support dense and sparse data representations using DOP following the C++17 standard.
- MATAR addresses:
  - Performance: contiguous dense and sparse data representation (i.e., use DOP).
  - Portable: run across CPUs and GPUs (using `Kokkos`) with a single implementation.
  - Productivity: easy to create, use, and integrate into codes (C++ and Fortran). For data allocating types,the memory is managed for the user.
- MATAR works in partnership with MPI or data management frameworks (e.g., FleCSI)

## MATAR design philosophy

- Enables a coding syntax that looks similar to C, python, and Fortran while adhering to C++
- Eliminates all parallel language syntax in a code (no pragmas, no Cuda, etc.)
- Enables existing codes to run on CPUs and GPUs after minor changes to the syntax

# MATAR data types

# MATAR has a rich set of capabilities to support diverse computational physics applications

**Dense** data representations include:

| | | Indexing pattern | |
|---|---|---|---|
| | | 0-indexed | 1-indexed |
| Access pattern | Column major | FArray<br>ViewFArray | FMatrix<br>ViewFMatrix |
| | Row major | CArray<br>ViewCArray | CMatrix<br>ViewCMatrix |

**Sparse** data representations include:

| | | Indexing pattern |
|---|---|---|
| | | 0-indexed |
| Access pattern | Column major | RaggedDownArray<br>DynamicRaggedDownArray<br>SparseColArray |
| | Row major | RaggedRightArray<br>DynamicRaggedRightArray<br>SparseRowArray |

# MATAR offers many allocatable data types on GPUs

**MATAR 3D array example**

```
// allocate 10x10x10 array on the GPU
CArrayKokkos <int> array3D (10,10,10);
```

```
// use array3D on the GPU
array3D(i,j,k) = 1;
```

**MATAR 2D dual-array example**

```
// allocate a 4x5 array on the CPU and GPU
DCArrayKokkos <double> array2D (4,5);
```

```
// use array2D on the CPU (host)
array2D.host(i,j) = 3.45;
```

```
// use array2D on the GPU (device)
array2D(i,j) = 3.45;
```

```
// methods to copy data
array2D.update_device(); // update GPU
array2D.update_host(); // update CPU
```

# MATAR offers many data types to view data as multidimensional on GPUs

## MATAR view of a 2D array example

```
// ...
// coding run on the GPU
// ...

int A[9];
ViewCArrayKokkos <int> array2D (&A[0],3,3);

// use array2D
array2D(i,j) = 1;
```

# MATAR offers many types to view data as multidimensional on multi-core CPUs and GPUs

## MATAR dual view of a 2D array example

```
// on the CPU
int A[27]; // can be an array from an existing MPI code etc.
DViewCArrayKokkos <int> array3D (&A[0],3,3,3);

// use array3D on the CPU (host)
array3D.host(i,j,k) = 3;
```

```
// use array3D on the GPU (device)
array3D(i,j,k) = 4;
```

```
// methods exist to update GPU or CPU array3D
array3D.update_host();
```

# Simple syntax to slice or create a data access pattern with existing allocated arrays or matrices

```
// allocate num_cellsx3x3 stress
int num_cells = 100000;
CArrayKokkos <double> cell_stress (num_cells,3,3);
```

### MATAR slice of an array

```
// inside a routine run on a GPU
// slice out the stress in a cell
int cell_id=314;  // cell id
ViewCArrayKokkos <double> stress (&cell_stress(cell_id,0,0),3,3);

// Use stress
for (int i = 0; i < 3; i++) {
    stress(i,i) = -pressure;
} // end for loop
```

# MATAR parallel loops

# MATAR provides simple to use syntax to run loops in parallel on both multi-core CPUs and GPUs

## C++ MATAR 3D array example

```
// allocate 10x10x10 array
CArrayKokkos <int> array3D (10,10,10);

// DOP parallel loop using MATAR+Kokkos
FOR_ALL (i, 0, 10,
         j, 0, 10,
         k, 0, 10, {
            array3D(i,j,k) = 1;
}) // end parallel for loop

// runs in parallel on GPUs and CPUs
// leveraging Kokkos
```

## Classic 3D C++ array example

```
// allocate 10x10x10 array
int array3D [10][10][10];

// Classic 3D for loop
for (int i = 1; i < 10; i++) {
    for (int j = 1; j < 10; j++) {
         for (int k = 1; k < 10; k++){
              array3D[i][j][k] = 1;
         }
    }
} // end for loop
```

# Simple syntax is key to having maintainable coding

## C++ MATAR 2D array example

```
// allocate 10x10 array
CArrayKokkos <int> array2D(10,10);

// Initialize matrix3D in parallel
FOR_ALL (i, 0, 10,
         j, 0, 10, {
            array2D(i,j) = 1;
}); // end parallel for
// array2D is on the e.g., GPU
```

The Kokkos syntax is powerful but not straightforward to use. MATAR greatly simplifies using Kokkos in a code.

## Kokkos 2D array example

```
using LoopOrder = Kokkos::Iterate::Right;
using Layout = Kokkos::LayoutRight;
using ExecSpace = Kokkos::Cuda;
using MemoryTraits = void;
// allocate 10x10 array
Kokkos::View<int**, Layout, ExecSpace,
             MemoryTraits> Array2D(10,10);
// Initialize matrix3D
Kokkos::parallel_for(
        Kokkos::MDRangePolicy
        <Kokkos::Rank<2,LoopOrder,LoopOrder>>
        ( {0, 0}, {10, 10} ), KOKKOS_LAMBDA
        ( const int i, const int j ){
            array2D(i,j) = 1;
        });
// array2D is on the e.g., GPU
```

# MATAR supports a myriad of parallel loop types on a GPU

### C++ parallel loop examples

```
// all loops go from i=a to i<N
FOR_ALL();
REDUCE_SUM();
REDUCE_MAX();
REDUCE_MIN();
```

### Fortran-like parallel loop examples

```
// all loops go from i=a to i<=N
DO_ALL();
DO_REDUCE_SUM();
DO_REDUCE_MAX();
DO_REDUCE_MIN();
```

### Serial execution

```
// run coding serially on a GPU
RUN();
```

# Using MATAR with Fortran codes

# MATAR supports Fortran multidimensional dense data access patterns and indexing

| | j = 0 | j = 1 | j = 2 | j = 3 |
|---|---|---|---|---|
| i = 0 | 10.9 | 0.9 | 11.3 | 90 |
| i = 1 | 0.5 | 110.1 | 0.125 | 81 |
| i = 2 | 1.2 | 7.3 | 8.19 | 14.12 |
| i = 3 | 8.9 | 45.1 | 1 | 11.9 |

| Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Data | 10.9 | 0.5 | 1.2 | 8.9 | 0.9 | 110.1 | 7.3 | 45.1 | 11.3 | 0.125 | 8.19 | 1 | 90 | 81 | 14.12 | 11.9 |

Figure: The data inside MATAR is 1D but accessed as $A(i, j)$ contiguously.

# The goal with MATAR is to make GPU coding as simple as possible

## C++ MATAR 3D matrix example

```cpp
// allocate 10x10x10 matrix
FMatrixKokkos <int> matrix3D (10,10,10);

// Initialize matrix3D in parallel
DO_ALL (k, 1, 10,
        j, 1, 10,
        i, 1, 10, {
            matrix3D(i,j,k) = 1;
}); // end parallel do
// matrix3D is on the device, e.g., GPU
```

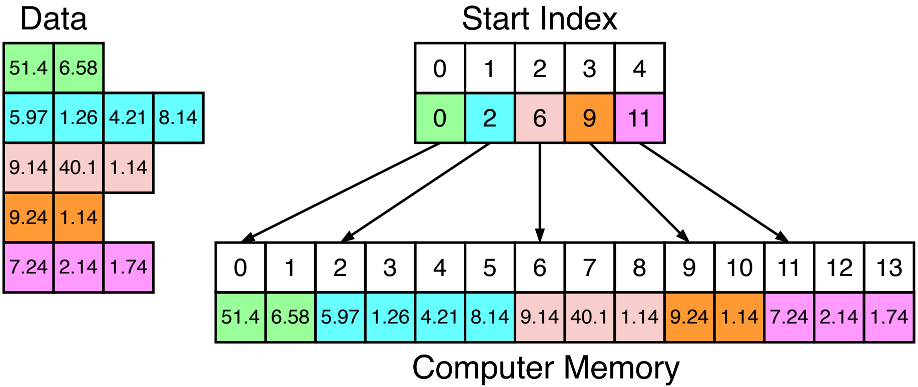
## Fortran 3D matrix example

```fortran
! allocate 10x10x10 matrix
INTEGER :: matrix3D (10,10,10)

! Initialize matrix3D
DO k = 1, 10
    DO j = 1, 10
        DO i = 1, 10
            matrix3D(i,j,k) = 1
        END DO
    END DO
END DO
```

# MATAR sparse types

# MATAR supports a ragged-right sparsity type




The layout of a ragged-right array where the column size for each row varies. The user accesses the data as a 2D array, $A(i,j)$, but the data in memory is stored as a contiguous 1D array by rows.

# Simple-to-use sparse data representations benefit computational physics codes

Consider the ragged-right array,

$$\mathbf{R}(i,j) = \begin{matrix} \begin{pmatrix} 1 & 1 & 1 \end{pmatrix} \\ \begin{pmatrix} 2 & 2 \end{pmatrix} \\ \begin{pmatrix} 3 \end{pmatrix} \\ \begin{pmatrix} 4 & 4 & 4 & 4 \end{pmatrix} \end{matrix}$$

**C++ MATAR ragged-right example**

```
CArrayKokkos <int> set_strides(4);
RUN({set_strides(0)=3;
     set_strides(1)=2;
     set_strides(2)=1;
     set_strides(3)=4;
    }); // serial execution on a GPU

RaggedRightArrayKokkos <int> my_ragged(set_strides);

FOR_ALL (i, 0, 4, {
    for (int j = 0; j < my_ragged.stride(i); j++){
        my_ragged(i,j) = i;
    }
}); // end parallel for loop
```

# MATAR extends Kokkos to support performance portability with diverse sparse data types

## C++ MATAR example using Kokkos

```cpp
int i_max = 3, j_max = 5;

DynamicRaggedRightArrayKokkos <int> my_dyn_ragged(i_max, j_max);

// parallel for loop on a CPU or GPU using MATAR+Kokkos
FOR_ALL(i, 0, i_max, {
    for (int j = 0; j <= (i % j_max); j++) {
        my_dyn_ragged.stride(i)++;
        my_dyn_ragged(i,j) = j;
    } // end for
}); // end parallel for
Kokkos::fence();  // wait for all threads to finish
```

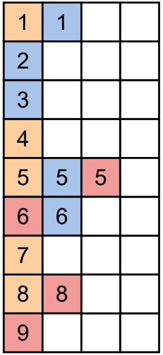
# MATAR supports a large suite of data representations, only a few are presented in this talk

**Dense data types**

| | |
|---|---|
| FArray | ViewFArray |
| FMatrix | ViewFMatrix |
| CArray | ViewCArray |
| CMatrix | ViewCMatrix |

**Sparse data types**

| | |
|---|---|
| RaggedRightArray | SparseRowArray |
| RaggedDownArray | SparseColumnArray |
| DynamicRaggedRightArray | SparseRowAccessor |
| DynamicRaggedDownArray | SparseColumnAccessor |




The dynamic ragged arrays have memory buffers. The dynamic ragged-right is shown.

# General examples

# Example: passing MATAR data types to functions

coding

```
int calculate_sum(const FMatrixKokkos<int> &matrix){
    int local_sum;
    int sum;

    // do summation in parallel on GPU
    DO_REDUCE_SUM(k, 1, 6,
                  j, 1, 5,
                  i, 1, 4,
                  local_sum, {
                      local_sum += matrix(i,j,k);
                  }, sum);
     return sum;
} // end function
```

# Example: Enumerated lists on a GPU

header file

```
// enum
namespace choices
{
    enum myChoice
    {
        METHOD_A = 1,
        METHOD_B = 2,
        METHOD_C = 3
    };
}
```

coding

```
CArrayKokkos <choices::myChoice> my_choices(2);

// set the method on the GPU
RUN({
        my_choices(0) = choices::METHOD_A;
        my_choices(1) = choices::METHOD_B;
});

FOR_ALL(i,1,2,{
    switch (my_choices(i)) {
        case choices::METHOD_A: { // do stuff
            break;}
        case choices::METHOD_B: { // do stuff
            break;}
    };  // end switch
});
```

# Example: Using MATAR with structs and classes

header file

```
// a struct with dual arrays inside
struct cell_data_t{

    DCArrayKokkos <double> den;
    DCArrayKokkos <double> pres;

    KOKKOS_FUNCTION
    void initialize(const int i,
                    const int j,
                    const int k) const{
        den(i,j,k)  = 0.0;
        pres(i,j,k) = 0.0;
    };
};
```

coding

```
cell_data_t cell_data;

cell_data.den =
    DCArrayKokkos <double> (10,10,10);
cell_data.pres =
    DCArrayKokkos <double> (10,10,10);

// initialize cell variables on GPU
FOR_ALL(i, 0, 10,
        j, 0, 10,
        k, 0, 10, {
        // remember KOKKOS_FUNCTION
        cell_data.initialize(i,j,k);
    });
```

# Example: Using function pointers on a GPU for object-based programming

header file

```
// function pointer
template <typename T>
struct method_ptrs{
  void (*fcn_ptr)(const T);
};

template <typename T>
KOKKOS_FUNCTION
void sum(const T){ };

template <typename T>
KOKKOS_FUNCTION
void multiply(const T){ };
```

coding

```
CArrayKokkos <method_ptrs<FMatrixKokkos<int>>>
                                        Array_ptrs(2);
// set the pointer on the device e.g., GPU
RUN({
    Array_ptrs(0).fcn_ptr = sum;
    Array_ptrs(1).fcn_ptr = multiply;
});
Kokkos::fence();

// use the function
FOR_ALL(i,0,2, {
    Array_ptrs(i).fcn_ptr(matrix2D);
});
```

# Example: Adding MATAR to an existing MPI code

## Struct in existing code

```
// The data in an existing code
struct code_t{
    double data1[100];
    double data2[100];
};
```

## header file

```
// MATAR view of the existing data
struct code_matar_t{
   DViewCArrayKokkos <double> data1;
   DViewCArrayKokkos <double> data2;
};
```

## coding

```
code_matar_t mtr;

mtr.data1 = DViewCArrayKokkos
      <double> (&code.data1[0],10,10);
mtr.data2 = DViewCArrayKokkos
      <double> (&code.data2[0],10,10);

// set the code values on the GPU
FOR_ALL(i, 0, 10,
        j, 0, 10, {
            mtr.data1(i,j) = 5.6;
            mtr.data2(i,j) = 9.2;
        });
Kokkos::fence();
mtr.data1.update_device(); // code_t data1
mtr.data2.update_device(); // code_t data2
```

# Many examples are provided in the MATAR and Fierro GitHub repositories

## 1D hydro code snippet from the Fierro repository

```
 // v_new = v_n + alpha*dt/mass*Sum(forces)
FOR_ALL (node_id, 1, num_nodes-1, {

    int corner_id_0 = get_corners_in_node(node_id, 0); // left corner id
    int corner_id_1 = get_corners_in_node(node_id, 1); // right corner id

    double force_tally = corner_force(corner_id_0) +
                         corner_force(corner_id_1);


    // update velocity
    node_vel(node_id) = node_vel_n(node_id) +
                        rk_alpha*dt/node_mass(node_id)*force_tally;

}); // end parallel for on device
```

# The Fierro code has a 3D explicit finite element hydrodynamic method written with MATAR that runs on CPUs and GPUs

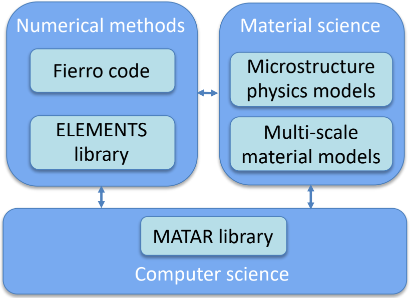
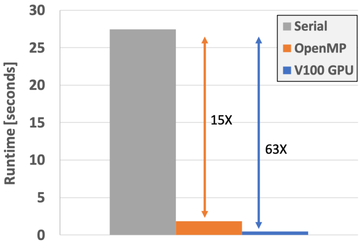
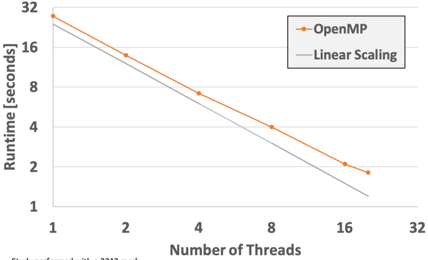



Figure: The code structure




Study performed with a 32^3 mesh

# Conclusions

# MATAR is designed to help people write codes that run in parallel across CPUs and GPUs

- MATAR uses DOP for **P**erformance, Kokkos for **P**ortability over architectures, and uses simple interfaces for **P**roductivity —3P's in writing software.
  - Contains many data types including dense and sparse types
  - Can benefit existing or new C++ codes, and aids conversion of existing Fortran routines to C++ with Kokkos
  - Successfully used in diverse applications including contact surface detection, phase-field modeling, scale-bridging material models, finite element solvers, discontinuous Galerkin methods, and more.
- MATAR (for parallelism on CPUs or GPUs): https://github.com/lanl/MATAR
- ELEMENTS (finite element library): https://github.com/lanl/ELEMENTS
- Fierro (Implicit and explicit FE code): https://github.com/lanl/Fierro

# References I


[1] Daniel J. Dunning, Nathaniel R. Morgan, Jacob L. Moore, Eappen Nelluvelil, Tanya V. Tafolla, and Robert W. Robey.

MATAR: A Performance Portability and Productivity Implementation of Data-Oriented Design with Kokkos.

*Journal of Parallel and Distributed Computing*, 157:86–104, 2021.

▸ Link https://github.com/lanl/MATAR.

[2] H. Carter Edwards, Christian R. Trott, and Daniel Sunderland.

Kokkos: Enabling manycore performance portability through polymorphic memory access patterns.

*Journal of Parallel and Distributed Computing*, 74(12):3202 – 3216, 2014.

Domain-Specific Languages and High-Level Frameworks for High-Performance Computing.