MULTIMAT: AN API FOR MANAGING MULTIMATERIAL SIMULATION DATA

Raine Yeh[1] Kenneth Weiss[2] Arlie Capps[2] Xavier Tricoche[3]

[1] *Purdue University, West Lafayette, IN, U.S.A.*
now at: Google, New York, NY, U.S.A. raineyeh@google.com
[2] *Lawrence Livermore National Laboratory, Livermore, CA, U.S.A. {kweiss,capps2}@llnl.gov*
[3] *Purdue University, West Lafayette, IN, U.S.A. xmt@purdue.edu*

ABSTRACT

Multimaterial simulation codes model the flow of materials with differing physical properties over a computational domain. Due to the intrinsically complicated access and traversal patterns on the underlying material-based field data defined over its mesh cells, such codes require implementations to strike a careful balance between competing demands of usability and performance. For a successful multimaterial simulation code, the designs of space efficient data structures, performant implementations, and flexible, developer-friendly representations that can adapt to varying traversal patterns and computer architectures must all satisfy this balance. Towards this aim, we introduce `MultiMat`, an open source library designed for efficient interaction with multimaterial mesh data and clear, flexible expression of multimaterial algorithms. `MultiMat` provides an intuitive API for operating on multimaterial data, several concrete data structures for representing this data, and functions to easily convert between different representations. We include code-to-code comparisons against explicit implementations of several representative physics kernels. Our results indicate that `MultiMat` simplifies data access and increases code readability while achieving comparable performance.

Keywords: multimaterial simulations, mesh data structures, sparse encodings.

1. INTRODUCTION

Multimaterial problems are commonly used to model the flow of materials with differing physical properties through a problem domain, for example in computational fluid dynamics (CFD) and hydrodynamics [1, 2]. Typically, the domain is discretized into geometric cells and the materials within a cell are represented in terms of *volume fractions* indicating the percentage of the cell's volume occupied by each material [3].

Multiphysics problems, such as those modeling high energy density physics, can easily require tens of materials that are unevenly distributed over a problem domain discretized by millions of cells. Examples include modeling instabilities induced by laser-driven shocks [4] and Inertial Confinement Fusion (ICF) problems where we must also represent fields associated with different isotopes of each material [5].

The codes that simulate these problems can be large, ranging to hundreds of thousands or millions of lines of code. Due to the large investment of development and validation, these codes can be in active development and use for decades.

Operations on multimaterial mesh data typically occur within tight performance-critical loops, so considerable effort has gone into developing efficient data representations and layouts for operating on this data. But efficiency in laying out data is not the only concern. There are several tradeoffs that must be considered when developing representations to support performant multimaterial kernels within large multiphysics codes:

Data layout. Kernels within multiphysics codes can have vastly different data access patterns. For example, kernels that depend on local properties of a cell might benefit from a *cell-dominant* layout, with an outer loop that traverses the cells of the mesh and an inner loop traverses the cell's materials. In contrast, kernels that require expensive material-based lookups, such as for the material's equation of state (EOS), can benefit from a *material-dominant* layout, where the lookups can be performed once per material and the costs can be amortized over all cells containing that material. Further, simulations may need to run on machines with drastically differing computer architectures whose performance characteristics might warrant competing data layouts.

Data transformations. Multiphysics codes often depend on functionality from external libraries. Interfacing with such libraries can require transforming simulation data to another representation. Such transformations involve complex bookkeeping and are a common source of error. They can involve hundreds of lines of code, are rarely well-tested and can be difficult to modify, such as when new features are added to the code.

Sparsity and dynamic updates. A problem can be defined by many materials, but each cell of the mesh typically only contains a few materials. Several data representations have been proposed in the literature. The simplest storage scheme holds variables for all combinations of every material in each cell and sets the volume fraction to zero for materials that are not present in a cell. We refer to this as the *full matrix* layout, or *full* layout. This straightforward scheme provides fast access and easy modification. However, this comes at the cost of inefficient usage of space, which can limit the types of problems that a code can run, especially on memory-limited computing architectures such as GPUs. In contrast, *compact* layouts store state variables for a material only when it is present in a cell. This is typically implemented using extra indexing information to keep track of which materials are present in each cell. Compared to full matrix layout, compact representations can greatly reduce storage needs and calculation times, but data access is less straightforward due to the extra index indirection. Compact data can also be more difficult to dynamically update, such as when adding or removing a material from a cell.

Developer productivity and debugging. In an effort to achieve the greatest possible performance or to quickly add new capabilities, codes often use explicit (*native*) indexing into multimaterial fields, directly exposing programmers to details of the underlying data representation. Within small applications, native indexing does not typically impose an undue burden on developers. But in larger codes, the practice can have several drawbacks. Most prominently, bookkeeping errors, such as incorrect loop bounds, can be very difficult to track down when using native indexing. Native implementations can also increase the difficulty of developing new algorithms because developers must be intimately aware of the intricacies of the underlying data structures. In such cases, it is not uncommon for developers to bootstrap their development by copying code from similar kernels, which can lead to further bookkeeping issues [6]. Native implementations can also make it difficult to modify the underlying data structures since consistent changes must be made throughout the codebase. Manually switching between the data layouts to comply with different algorithmic data access pattern can be another source of errors. Developers recognize this and can be reluctant to work on code that uses extensive native indexing.

In summary, a flexible layout is critical for efficient data management in a codebase that must adapt to different layout patterns while staying memory efficient. As pointed out by Fogerty et al. [7, 8], there can be no single optimal choice of layout or sparsity for all situations. The best choice of data structure can vary between applications, between specific problems, and even within different sections of the code. This situation calls for an abstraction of the data storage structure. And, since multimaterial indexing is used heavily within tight inner loops, the design of an extra abstraction layer requires care to avoid performance overhead. The additional function calls and runtime checks necessary to support polymorphic behavior can impose an unwelcome penalty.

1.1 Contribution

In this paper, we propose `MultiMat`, a clean, lightweight API to access, transform and modify multimaterial data. `MultiMat` was developed with performance requirements in mind and provides transparent support for several important data layouts while hiding obscure implementation details behind a simple user interface.

It is important to note that the underlying representations used by multimaterial simulations codes (and abstracted by `MultiMat`) contain the same core ingredients as those in numerical processing and linear algebra libraries such as Eigen [9] and Armadillo [10]. As such, extremely efficient implementations of data structures for dense and sparse matrices are widely utilized across numerous fields. However, managing multimaterial field data is outside the design scope of such libraries. In particular, it is rare for collections of sparse matrices within a linear algebra libraries to share indexing data.

Alternatively, since material-based fields within a multimaterial simulation share the same indexing data, their computational kernels typically operate on several fields using the same indexing data. Furthermore, changes to the material distribution within these meshes, such as during remeshing, load balancing or mesh refinement requires careful coordination among all associated material-based fields.

The main contribution of this work lies in providing a performant, intuitive abstraction for accessing and operating on multimaterial mesh data. This lets application developers and code physicists focus not on the details of data structures and bookkeeping, but rather on the underlying physics in their application, without sacrificing performance. In this paper, we compare the performance of our API against native implementations. For a comprehensive comparison between the performance differences that can be obtained on the same kernel using different layouts and representations, see Fogerty et al. [7].

We review some basic terminology and data structures in Section 2 and describe our library design and API in Section 3. In Section 4, we evaluate `MultiMat` by comparing code and performance against native implementations. We conclude in Section 5 with a discussion of our results and an outline for future directions.

2. DATA STRUCTURE BACKGROUND

In this section, we describe several data structures that underlie `MultiMat` and introduce concepts and terminology used in the rest of this paper. We illustrate our discussion with a running example of a multimaterial mesh of four cells and three materials, shown in Figure 1a. The shaded regions in the figure represent material regions occupying the mesh cells. For example, cell 1 contains only a single material and cell 2 contains all three materials. Following the *Volume of Fluid (VOF)* framework [3], the mesh cells implicitly encode the material interfaces through *volume fractions*, encoding the percentages of each cell's volume occupied by each material. Note that while we are demonstrating only a 2D mesh, `MultiMat` works identically with 3D meshes since it encodes fields as indexed arrays.

Multimaterial meshes contain fields that describe the simulation's state variables, such as volume, temperature, and pressure. Depending on the attribute in question, a field can map to each cell, to each material, or to specific materials within each cell. For example, fields for "cell volume" or "number of neighboring cells" might be per-cell variables, as there is one value stored for each cell, whereas "material density" would be a per-material variable. For the mesh in Figure 1a, a per-cell field would need to store four variables, while a per-material field would need to store three. We refer to these as *1D fields*. Using a 1D field is fairly straightforward. The field values are stored in a contiguous array that is directly indexed (e.g. `field[i]` accesses the value associated with index i).

In contrast, we refer to fields that associate values with materials within a cell as *2D fields*. "Volume fraction" is an example of a 2D field since it is defined for each material of a mesh's cell. Returning to Figure 1a, cell 1 would have a single non-zero volume fraction (for material B), while cell 2 would have three associated volume fractions (i.e. for materials A, B, and C).

Since we are dealing with the Cartesian product of materials and cells, it is convenient to conceptualize 2D fields as matrices, whose rows and columns correspond to materials and cells. Entries are uniquely identified by a material ID and a cell ID, which are equivalent to row and column indices in a matrix. Note that layout dimension is distinct from the spatial dimension of the simulation mesh.

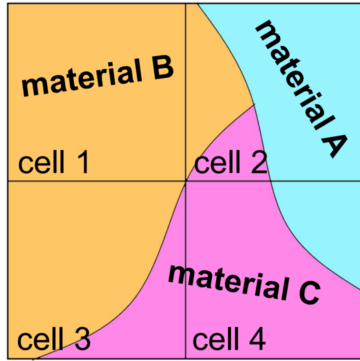
Data structures for 2D fields allow for more optimization opportunities along with greater complexity. In the remainder of this section, we will discuss the full matrix data structure, followed by compact alternatives that can reduce storage costs.

2.1 Full Matrix Layout

The simplest way to store a 2D field is the *full matrix layout*, or *full layout*, where every material in the problem has an entry for each cell of the mesh. In this case, the absence of a material from a cell can be inferred when its corresponding volume fraction is zero.

The data in a full matrix layout can be organized in one of two ways: A multimaterial data structure is *cell-dominant* if it is organized as a list of cells composed of varying sets of materials, and *material-dominant* if it is organized by the materials. Figure 1b shows our example mesh in a cell-dominant full matrix layout while Figure 1c shows the same mesh in a material-dominant full matrix layout. We use the row-major convention in this paper, where consecutive entries of a row reside next to each other. For a cell-dominant layout, we call the cell ID the *outer-index*, and the material ID the *inner-index*. Similarly, a material-dominant layout would consider the material ID as the outer-index, and the cell ID as the inner-index.

Since the full 2D matrix uses a contiguous array to store these fields, the actual array data is accessed with a single index, which we refer to as a *flat-index*. Fig. 2 shows the flat-index for our example mesh in relation to its cell and material IDs. Accessing a state variable for cell `c` and material `m` is done by calculating the flat-index as `field[c * nM + m]` for a cell-dominant lay-



(a) Example mesh

Cells \ Materials	A	B	C
1	-	1B	-
2	2A	2B	2C
3	-	3B	3C
4	4A	-	4C

(b) Full cell-dominant layout

Materials \ Cells	1	2	3	4
A	-	2A	-	4A
B	1B	2B	3B	-
C	-	2C	3C	4C

(c) Full material-dominant layout

	1B	2A	2B	2C	3B	3C	4A	4C
Flat-index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*

(d) Compact cell-dominant layout

	2A	4A	1B	2B	3B	2C	3C	4C
Flat-index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*

(e) Compact material-dominant layout

Figure 1: An example mesh with four cells and three materials (a) along with several cell-dominant (b,d) and material-dominant (c,e) layouts. Full layouts (b,c) have storage for all materials and cells, while compact layouts (d,e) reorganize the data into a 1D array.

Cell index \ *Material index*	*0*	*1*	*2*
0	0	1	2
1	3	4	5
2	6	7	8
3	9	10	11

Figure 2: Flat-index for a cell-dominant full matrix layout of the example mesh.

out, or `field[m * nC + c]` for a material-dominant layout. In both cases, the storage space used is the same: A 2D field with `nC` cells and `nM` materials will use an array of size `nC * nM` in full matrix layout.

The full matrix layout provides a simple storage scheme and straightforward access to state variables at the cost of inefficient usage of storage space due to explicitly storing state variables for materials that are not present in a cell. The extra storage usage is especially excessive in simulations with many materials but low material mixture, where most cells contain only a few materials, effectively making the 2D field a sparse matrix.

2.2 Compact Layout

One can improve on the storage space used by the full matrix layout by omitting the zeroed entries. We refer to this as a *compact layout*, which stores a field variable for a material only when it is actually present in the cell (i.e. its volume fraction is non-zero). Figure 1d shows our example mesh in compact cell-dominant layout, where only the non-zero values are stored, and Figure 1e shows the material-dominant equivalent. In addition to storing the values, a compact layout must keep track of additional index information in order to reconstruct the equivalent full matrix data.

There have been many compact storage schemes developed by the scientific computing community for encoding sparse matrices [11]. `MultiMat` uses the compressed sparse row (CSR) layout for our compact layout. This is a common representation in linear algebra libraries [9, 10]. as well as in mesh processing, where it can be used to compactly encode topological connectivity relations [12, 13] and for efficient spatial indexing [14, 15], among numerous other applications [16].

To store a 2D field with `M` rows and `N` columns, CSR uses three (one-dimensional) arrays, `A`, `IA`, and `JA`. The first array `A` stores the non-zero values, and has length `nnz`, the number of non-zeros in the field. The second array `IA` is of length `M+1` and contains the offsets into `A` of the first element from each row. The first `M` elements of `IA` store the index into `A` of the first nonzero element in each row, while the last element

A: non-zero values	2A	4A	1B	2B	3B	2C	3C	4C
Flat-index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*

IA: extent of row	0	2	5	8

JA: column index	1	3	0	1	2	1	2	3

Figure 3: Compact material-dominant layout of the example mesh, stored in the CSR format.

`IA[M]` stores the value `nnz`, which is also the number of elements in `A`. If a row `i` has no elements, `IA[i]` stores the same value as `IA[i-1]`. The values of the $\mathtt{i}^{th}$ row of the original matrix can be found in entries `A[IA[i]]` to `A[IA[i+1]-1]`. The third array, `JA`, contains the column index of each element of `A`, and hence is of length `nnz` as well.

Figure 3 shows the CSR layout for our example mesh in compact material-dominant layout (see Figure 1e). In this example, array `A` stores the field values, array `JA` stores the cell ID associated with each of the field values in `A` and array `IA` stores the flat-index of the first field value of each material, with the last entry storing the size of the `A` array.

The cell-dominant version of the CSR layout is the transpose of the material-dominant version. The sizes of `A` and `JA` do not change, whereas the size of `IA` is now one more than the number of cells, rather than the number of materials. The CSR layout uses `2*nnz + nr + 1` storage space and is more compact than the full matrix when `(2*nnz + nr + 1) < M*N`.

We note that, in contrast to typical sparse matrix representations, multimaterial fields can reuse the indexing arrays for multiple fields. Specifically, if we already have at least one compact 2D field in our multimaterial state, the overhead associated with adding a new field is only a single array of size `nnz` rather than an entire new sparse matrix.

The CSR format allows fast row access and more efficient space usage by not storing the zeros explicitly. However, accessing an element requires a level of indirection since we must find the index of the data before actually accessing the data. Additionally, dynamic modification (adding or removing an entry) requires extra operations and bookkeeping.

We conclude this section with some additional indexing terminology. We refer to a single row of a 2D field as a *subfield*. For example, a subfield in a cell-dominant field contains all data associated with the materials in a specific cell. A *subindex* is a zero-based index into a subfield. This is equivalent to the column index for the full matrix representation, but can be different in compact layouts.

3. MULTIMAT API

We followed several guiding principles in the design of our multimaterial data management library:

One interface to rule them all. `MultiMat` should be a unified API for many different data layouts and representations. Layout changes should require minimal changes to user code.

Intuitive methods. Our API should be easy to use and understand. Function calls should have intuitive names to make their semantics clear. Users should not be required to be aware of the underlying data structures to use our API.

Versatility. Our API should be general enough to operate with different data layouts. The API should also provide access methods to let users exploit specific underlying data layouts for efficiency.

Covers most needs. Our API should allow for a variety of computational use cases that may arise. It should provide efficient traversal of the data as well as random access capability for algorithms with that need (when supported by the underlying representation). When all else fails, provide a means for users to retrieve their data.

Robust error checking. All internal structures should provide functions to check their internal validity, such as that pointers are not `nullptr` and that indexes are in-bounds.

Efficient. When used correctly, there should be negligible overhead cost to use our API in comparison to explicitly programming against the underlying data structure.

`MultiMat` is a `C++` library for managing multimaterial data. It stores some metadata about the problem (e.g. number of cells and materials) as well as fields for the state variables pertinent to the simulation. Each field is stored as a `C++` object that records the field mapping (to mesh entities or materials) and the data type of the field (e.g. `double`, `int`).

We will now discuss the specific API our library uses, and give a few usage examples for each method. For simplicity, we assume the data type of the fields to be double floating point (`double` in `C++`). Because our library is templated on the data type, the actual data can be any primitive or user defined data type.

Field access using `operator()`, as shown in the following subsections, can retrieve a value, a reference, or a `const` reference. Brackets indicate optional use of `const` or `&`.

3.1 1D Field

Using `MultiMat`, access to per-cell or per-material 1D fields is straightforward since the data is stored in a contiguous array. Given an index into the array, its memory location can be calculated and accessed in constant time using the `C++` parenthesis operator. The API to access the data in field `field` at index `idx` is:

```
double [const][&] val = field(idx);
```

A typical traversal can be done by incrementing an index in a for-loop. For example to compute the total volume from the 1D `Volume` field:

```
double totalVolume = 0;
for (int c = 0; c < ncells; ++c)
  totalVolume += Volume(c);
```

3.2 Direct Access with findValue()

For 2D fields that are mapped to each material in each cell, each entry of the field is indexed by a material ID and a cell ID. We provide the general `findValue()` function to query a full or compact 2D field, using an arbitrary cell ID and material ID:

```
double* val = field2D.findValue(cellId, matId);
```

This function returns a `nullptr` when cell `cellId` does not contain material `matId`.

We note that compact layouts do not provide $\mathcal{O}(1)$ random access to the data, so each such call may incur a search for the specific entry. This API is therefore often not the most efficient way to access variables sequentially, but can be used when the application requires this or when performance is not a concern.

Below is an example snippet using `findValue` to calculate the total volume of each material using 2D `VolFrac` field, which contains the material volume fractions, and 1D `Volume` field, which contains the per-cell volume.

```
double* MatVol = new double[ nmats ];
for (int c = 0; c < ncells; ++c)
{
  for (int m = 0; m < nmats; ++m)
  {
    double* val = VolFrac.findValue(c,m);
    if (val)
      MatVol[m] += *val * Volume(c);
  }
}
```

3.3 Row Access with Subfields

In many cases, instead of random access by material and cell ID, the user wants to traverse all the variables in a field sequentially. `MultiMat` provides several API options for efficient traversals in both full and compact layouts. Given a cell-dominant 2D field, a user can call

```
auto subfield = field2D(cellId);
```

to acquire the subfield associated with the materials contained in cell `cellId`. The user can then access this subfield as if it were a 1D field:

```
double [const][&] val = subfield(k);
```

where subindex `k` is an index into the subfield.

In a full matrix layout, `k` would be equivalent to the material ID. Users will have to check if individual `VolFrac` subfield entry is nonzero to see if a material is present in a cell. In the compact layout scenario, only the materials present in a cell would have their variables stored. The number of cells in a subfield is the same as the number of materials present in the cell. One can retrieve the material index with a `matId()` call:

```
int matId = subfield.matId(k);
```

Similarly, use `cellId()` to retrieve the cell ID:

```
int cellId = subfield.cellId(k);
```

although it would be redundant as the subfield is acquired using an existing cell ID.

Below is an example of code to calculate `MatVol`, the total volume of each material, using `VolFrac` field, which contains the material volume fractions, and `Volume`, which contains the per-cell volume:

```
double* MatVol = new double[ nmats ];
for (int c = 0; c < ncells; ++c)
{
  auto subField = VolFrac(c);                    //#1
  for (int k = 0; k < subField.size(); ++k)  //#2
  {
    int m = subField.matId(k);                   //#3
    MatVol[m] += subField(k) * Volume(c);    //#4
  }
}
```

There is a lot to unpack in this snippet. We first access the subfield associated with each cell in the outer loop (`#1`). We use the subfield's `size()` function as the inner loop bounds (`#2`) and, for each subindex `k` in the inner loop, we use the `matId()` (`#3`) and value (via the parenthesis operator) to update the material-based 1D field (`#4`).

This code snippet works for both full and compact layouts. For the material-dominant version, the subfield is indexed by material ID, and contains variables associated with all the cells that contain the specific material

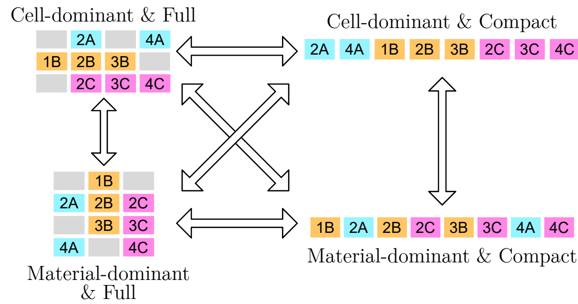


Figure 4: One-step conversion between layouts

```
auto subfield = matDomField(matId);
auto value = subfield(k); //for some index k
```

Similarly, the cell ID of an entry in the subfield can be retrieved with a `cellId()` call.

```
int cellId = subfield.cellId(k);
```

In the above example, layout-independent calls `cellId()` and `matId()` are used to retrieve the cell and material ID, which are equivalent to the row/column index of an entry, depending on the layout used. To retrieve the column and row index irrespective of the layout that is in-use, user can call `innerIndex()` for the column index in a subfield (equivalent to retrieving a material ID for a cell-dominant subfield entry), or `outerIndex()` for the row index (equivalent to retrieving the cell ID in a cell-dominant entry).

```
//cell-dominant layout only
int colIdx = subfield.matId(k);
//any layout
int colIdxAgain = subfield.innerIndex(k);
```

3.4 Iterator Access

We have also provided data access and traversal through STL-compliant iterators. This can be especially useful for representations that do not support random access traversals within a subfield as in the array-based linked list data structure used by the Silo I/O library [17], and implemented as the core cell-dominant compact layout in several multiphysics codes (see [7, 8] for more details).

Users can call function `begin()` for an iterator at the start of the field, traverse the data one-by-one by incrementing the iterator with `iter++`, and acquire the value of the field by dereferencing with `*iter`. Similar to a subfield, users can call utility functions `cellId()` to get the cell ID, and `matId()` to get the material ID. Alternatively, users can access the column and row indices by calling `iter.outerIndex()` and `iter.innerIndex()`, respectively. Using the same convention as described in the Subfield section, for a cell-dominant 2D field, `iter.outerIndex()` returns the cell ID, and `iter.innerIndex()` returns the material ID. And vice versa for a material-dominant field.

We provide a 1-level iterator that traverses both 1D and 2D fields from start to finish. The following is a (not particularly efficient) cell-dominant example:

```
double* MatVol = new double[ nmats ];
auto iter = VolFrac.begin();    // 1-level
while( iter != VolFrac.end() )
{
  int cellId = iter.cellId();
  int matId = iter.matId();
  MatVol[matId] += *iter * Volume(cellId);
  ++iter;
}
```

It can often be more efficient to have a nested loop, for example, when we have computations on the outer loop. In this case, subfield iterators can be used by passing the outer loop index to the 2D field's `begin()` and `end()` functions, and the inner index can be acquired with the `index()` function:

```
double* MatVol = new double[ nmats ];
for (int c = 0; c < ncells; c++)
{
  auto iter = VolFrac.begin(c);  // subfield c
  while( iter != VolFrac.end(c) )
  {
    int matId = iter.index();
    MatVol[matId] += *iter * Volume(c);
    ++iter;
  }
}
```

Either iterator style can be used for both full and compact layouts.

3.5 Converting Between Layouts

Currently our library supports full matrix layout and CSR for compact layout, as well as cell-dominant and material-dominant layouts. To simplify data management, users can convert all fields between the different layouts with a single function call:

```
convertToFullLayout()
convertToCompactLayout()
convertToCellDominant()
convertToMaterialDominant()
```

or individual layouts by specifying the field index:

```
convertFieldToFullLayout(fieldIdx)
convertFieldToCompactLayout(fieldIdx)
convertFieldToCellDominant(fieldIdx)
convertFieldToMaterialDominant(fieldIdx)
```

A user can query the current data layout by calling

```
getLayoutSparsity()
```

to check if the current layouts of all fields are full or compact, and

```
getLayoutOrdering()
```

to check if the current layouts are cell- or material-dominant. To query the layout of individual fields, the user can call field-specific functions with a field index:

```
getFieldLayoutSparsity(fieldIdx)
getFieldLayoutOrdering(fieldIdx)
```

Figure 4 shows the possible conversions between the different layouts.

Our current implementation stores each field in only one layout at a give time. In the future we plan to support multiple concurrent layouts for a given field, as this may benefit some algorithms.

3.6 Dynamic Access

Multiphysics simulations often have well-defined phases where the material decomposition within cells is static, and phases where dynamic changes can occur, for example materials can advect through cells during the remap stage of an ALE (Arbitrary Lagrangian-Eulerian) simulation [2]. In a full matrix layout, it is trivial to add or remove a material from a cell simply by writing to the location of the entry specific to that material and cell. In contrast, for a compact layout, it is non-trivial to add an entry in the data array or remove an existing entry while keeping good performance.

Using our library, a user can convert data from static to dynamic layout with

```
dynamicMode()
```

To add a new material to a cell and set a value in a 2D field, the user can call:

```
multiMatObj.addEntry(cellId, matId); //add entry
field2D(cellId)(matId) = new_value;  //set value
```

Similarly, to remove a material from a cell:

```
multiMatObj.removeEntry(cellId, matId);
```

When the dynamic phase is completed the user can call

```
staticMode()
```

Depending on the layout, this call might trigger compaction of the internal representation.

We currently do not have a specific implementation for a dynamic layout that optimizes performance while handing dynamic data modification; adding such an implementation is part of our future work. In the mean time, our implementation achieves dynamic modification by temporarily converting to a full matrix layout.

3.7 Other Implementation Details

Our library's internal implementation makes heavy use of `C++` templates to create and define objects of different data types and layouts. In many cases, this allows the compiler to inline functions and operations such that calling access operations will result in little to no overhead.

We have taken great care to expose an API to users that does not require (excessive) templates in user code. In many cases, however, a code has information on data layout at compile time. The optimal layout may have been determined by profiling, or a kernel may be written to use a specific data layout. In such cases, the user can provide this information to the `MultiMat` library in the form of template parameters to achieve even better performance.

For example, to get the "Density" field of type `double` without templating on the layout type, one would call

```
getField2D<double>("Density");
```

In the type-templated version, the user can provide the current layout as an additional template parameter:

```
getField2D<double,LayoutType>("Density");
```

where `LayoutType` indicates whether the field is stored in full matrix or a compact layout, such as CSR. In either case, the rest of the `MultiMat` access code is exactly the same. The only change is in the template parameter types passed to `getField2D()`.

The `MultiMat` data management library is part of Axom project developed at Lawrence Livermore National Laboratory [18]. Axom hosts a number of software infrastructure components for the development of multi-physics applications and computational tools. The Axom project is open-source under a permissive BSD 3-clause license, and is available at `https://github.com/LLNL/axom`.

Our library also provides support for simulations that involve multiple components per 1D- or 2D-field value.

For example, a vector field in 3D could contain three components per value. In general, those components are accessed with a component index in addition to the cell and material ID (i.e. `findValue(cellId, matId, comp)`). For simplicity's sake, we omitted this in our API overview. We refer the interested reader to our library project page for more information.

4. EVALUATION

In this section, we use a suite of representative computational scenarios to evaluate the usability and performance of our `MultiMat` library against *native* implementations that directly use the multimaterial field data structures.

Our benchmark suite is composed of several multi-material physics kernels proposed by Garimella and Robey [8].

The three scenarios are proxies for typical multimaterial kernels and each have different computational access patterns:

Average cell density. Compute the average density of a cell from those of its materials. The result is a 1D density field on the cells of the mesh. For each cell, the material-based density terms are weighted by the material volume fractions.

Material-dependent pressure. Evaluate the material-dependent pressure in each cell using the ideal gas law $p = nrt/v$. This kernel requires material-based data for each term and is typically more expensive than the simple array lookup implemented in this kernel.

Material-dependent neighborhood density. This is the most complex kernel in our benchmark. It computes the per-material average density from the cell's precomputed list of neighbors. Each term is weighted by the inverse squared distance to the neighbor.

We compare cell-dominant and material-dominant variants of each computational kernel for the full and compact layouts of the native and `MultiMat` implementations. We note that, for the remainder of this section, we are interested in comparing each `MultiMat` variant against its native (direct-indexing) counterpart, rather than comparing the different layouts against each other.

4.1 Code Comparison

Our API is designed with ease of use for the end users in mind. We want them to focus on their physics applications rather than on low level data structure and bookkeeping details. Reading and writing `MultiMat`-based code should be natural and intuitive once users get accustomed to a few concepts, like subfields and subindexes. With that in mind, we present some code comparisons. For easier comparisons across different implementations, we use color highlights for related lines of code.

We begin by comparing several implementations of the "Average cell density" physics kernel (see Listings 1, 2, 3 and 4). Listing 1a shows a cell-dominant full matrix layout implementation. Within the nested for-loop, the flat-index is explicitly calculated and used to access the array at the specific data location. Listing 1b shows the same kernel but using the CSR layout, which involves a level of indirection when accessing the values by first getting the flat-index with the `begin_idx` (the CSR's `IA` array described in Section 2) and using the flat-index to access the actual data array. In both cases, explicit indexing calculation has to be done in order to access the data. Changing the data structure layout would also mean rewriting the calculation kernel, even if the algorithm itself does not change.

Using our `MultiMat` library, the previous two implementations for full and compact layouts can be implemented with the exact same code due to our unified API (see Listing 2). The outer loop goes through each cell and acquires the subfield using the parenthesis operator `field2D(c)`. The inner for-loop then iterates through the subfield. In the full matrix layout, the subfield would contain the field data for all materials, while in the compact layout, the subfield would contain only data associated with materials that are actually present in cell `c`. Using the `MultiMat` subfield API, manual indexing work is kept to a minimum.

The previous examples were cell-dominant algorithms. The next two listings show material-dominant implementations of this algorithm. A key difference in this implementation is that there are separate pre- and post-processing loops to initialize the data and to normalize the averages, respectively. Listing 3a shows the physics kernel using a full matrix material-dominant layout. Similar to the cell-dominant equivalent, the outer loop goes through each material, and the inner loop through each cell using an explicit flat-index. Listing 3b shows the same kernel using the CSR layout. An additional array, `colIdx`, containing the column index is used to retrieve the cell ID in the material-dominant version. This is the `JA` array for CSR as described in Section 2.

Listing 4 shows the material-dominant algorithm using our `MultiMat` subfield access method. The cell ID is retrieved with the `cellId()` call. Aside from the pre- and post-processing loops and the `cellId()` call, the material-dominant code is quite similar to the cell-dominant equivalent. The same code is used for both

```
void AverageDensity_CellDom_full(
  int ncells, int nmats,
  vector<double>& VolFrac,
  vector<double>& DensityFrac,
  vector<double>& Volume,

  vector<double>& DensityAvg )
{
  for (int c = 0; c < ncells; ++c)
  {
    double densitySum = 0.0;
    for (int m = 0;
             m < nmats; ++m)
    {
      densitySum += DensityFrac[c*nmats + m]
                  * VolFrac[c*nmats + m];
    }
    DensityAvg[c] = densitySum / Volume[c];
  }
}
```

(a) Cell-dominant Full layout

```
void AverageDensity_CellDom_CSR(
  int ncells,
  vector<double>& VolFrac,
  vector<double>& DensityFrac,
  vector<double>& Volume,
  vector<int>& beginIdx,
  vector<double>& DensityAvg )
{
  for (int c = 0; c < ncells; ++c)
  {
    double densitySum = 0.0;
    for (int k = beginIdx[c];
             k < beginIdx[c + 1]; ++k)
    {
      densitySum += DensityFrac[k]
                  * VolFrac[k];
    }
    DensityAvg[c] = densitySum / Volume[c];
  }
}
```

(b) Cell-dominant CSR layout

Listing 1: Native implementation for Scenario 1: average cell density, in cell-dominant layout

```
void AverageDensity_CellDom_MM(
  int ncells,
  Field2D& VolFrac,
  Field2D& DensityFrac,
  Field1D& Volume,

  Field1D& DensityAvg )
{
  for (int c = 0; c < ncells; ++c)
  {
    double densitySum = 0.0;
    auto DensityFracRow = DensityFrac(c);
    auto VolFracRow = VolFrac(c);
    for (int k = 0;
             k < DensityFracRow.size(); ++k)
    {
      densitySum += DensityFracRow(k)
                  * VolFracRow(k);
    }
    DensityAvg(c) = densitySum / Volume(c);
  }
}
```

Listing 2: Scenario 1 for full or compact cell-dominant layout using `MultiMat` subfields.

full and compact layouts for the material-dominant loop, and there are few changes in index calculation or field access.

Listings 5 and 6 present implementations of the full and compact cell-dominant "Material-dependent neighbor density" benchmark, respectively. For each cell and material, this kernel loops through the neighboring cells and computes the sum of distance-weighted densities. Since the full variant checks volume fractions to see if materials are present (highlighted in green), while the compact variant performs indirection to find the material index (highlighted in pink), we present separate `MultiMat` implementations in Listings 5b and 6b, respectively, to better compare against their native counterparts.

For the sake of brevity, we present only a selected few layout and algorithm combinations to demonstrate our code readability. The "Material-dependent pressure" kernel comparisons have similar characteristics as the presented examples.

We hope that the above listings have demonstrated a more meaningful usage of our API. In general, native code developed in full layout tends to be simpler since the cell and material IDs can be used directly, but array access involves manual calculation of the flat-indices and this representation can be inefficient for sparse materials. Conversely, compact layouts, such as CSR, allow for more efficient code, but can be less intuitive due to the array indirection required to access the cell/material ID and their respective values (such as the `beginIdx` and `colIdx` arrays in CSR). With our `MultiMat` API, users can benefit from the efficiency of a compact layout while directly indexing into arrays, without explicit array indirection. Even in cases where our API results in more lines of code, the code is easier to develop and interpret since the underlying implementation details are hidden away. The resulting code is more straightforward, with as few details unrelated to the physics calculation as possible.

For demonstration purposes, the example listings shown above are shorter and simpler than real world multimaterial calculations with little computational work in each kernel. In large multimaterial codebases, having to keep track of bookkeeping details about the underlying data layout can quickly become a burden in the development process. In these situations, a clear and comprehensive API that simplifies understanding of the code can be highly valuable.

```
double AverageDensity_MatDom_full(
  int ncells, int nmats,
  vector<double>& VolFrac,
  vector<double>& DensityFrac,
  vector<double>& Volume,


  vector<double>& DensityAvg )
{
  for (int c = 0; c < ncells; ++c)
  {
    DensityAvg[c] = 0.0;
  }

  for (int m = 0; m < nmats; ++m)
  {
    for (int c = 0;
             c < ncells; ++c)
    {

      DensityAvg[c]
        += DensityFrac[m*ncells + c]
         * VolFrac[m*ncells + c];
    }
  }

  for (int c = 0; c < ncells; ++c) {
    DensityAvg[c] /= Volume[c];
  }
}
```

(a) Material-dominant Full layout

```
double AverageDensity_MatDom_CSR(
  int ncells, int nmats,
  vector<double>& VolFrac,
  vector<double>& DensityFrac,
  vector<double>& Volume,
  vector<int>& beginIdx,
  vector<int>& colIdx,
  vector<double>& DensityAvg )
{
  for (int c = 0 ; c < ncells ; ++c)
  {
    DensityAvg[c] = 0.0;
  }

  for (int m = 0; m < nmats; ++m)
  {
    for (int i = beginIdx[m];
             i < beginIdx[m + 1]; ++i)
    {
      int c = colIdx[i];
      DensityAvg[c]
        += DensityFrac[i]
         * VolFrac[i];
    }
  }

  for (int c = 0; c < ncells; ++c) {
    DensityAvg[c] /= Volume[c];
  }
}
```

(b) Material-dominant CSR layout

Listing 3: Native implementation for Scenario 1: average cell density, in material-dominant layout

```
void AverageDensity_MatDom_MM(
  int ncells, int nmats
  Field2D& DensityFrac,
  Field2D& VolFrac,
  Field1D& Volume,
  Field1D& DensityAvg )
{
  for (int c = 0; c < ncells; ++c)
  {
    DensityAvg(c) = 0.0;
  }

  for (int m = 0; m < nmats; ++m)
  {
    auto DensityFracRow = DensityFrac(m);
    auto VolFracRow = VolFrac(m);
    for (int k = 0 ;
             k < DensityFracRow.size() ; ++k)
    {
      int c = DensityFracRow.cellId(k);
      DensityAvg(c)
        += DensityFracRow(k)
         * VolFracRow(k);
    }
  }

  for (int c = 0; c < ncells; ++c)
  {
    DensityAvg(c) /= Volume(c);
  }
}
```

Listing 4: Scenario 1 with full or compact material-dominant layout using `MultiMat` subfield

One additional feature that is not highlighted in our code samples relates to internal validity checks that we have implemented throughout the `MultiMat` library. Every class has an internal `isValid()` function that checks the internal validity of the overall structure. For example, the function checks that none of the pointers are `nullptr` and that the internal indices are within their proper ranges. Furthermore, since bookkeeping and indirection errors are developer errors, debug builds of the code guard all indirection accesses (such as the subfield parenthesis operator) with asserts that validate the provided indices to ensure they are in the proper range. If this check fails, an error message and complete stacktrace are printed out to help the developer pinpoint the problem. These checks are disabled in release builds and so have no runtime costs.

As noted above, powerful, popular linear algebra packages exist that implement the same sparsity and storage schemes However, `MultiMat`'s function goes beyond that of a linear algebra package. Our library is designed to track the materials that interact in a computational mesh. The overall mesh index space is shared by all the materials, and all the fields of a particular material share the same index set. When transitions are needed between sparsity or material- vs. cell-dominant storage, `MultiMat` takes care of the tedious bookkeeping that is needed to maintain the interrelation of the materials. `MultiMat` is also designed for portability across platforms and interoperability

```cpp
void AvgDensityOverNeighbor_CellDom_Full(
  int ncells, int nmats,
  vector<double>& VolFrac,
  vector<double>& DensityFrac,
  vector<int>& numNbrsInCell,
  vector<int>& cellNbrs,
  vector<double>& cen, //cell centroid

  vector<double> MatDensityAvg )
{
  for (int c = 0; c < ncells; ++c)
  {
    // Get the neighbors for this cell
    int* nbrs = &(cellNbrs[c * MAX_NBR]);
    int nn = numNbrsInCell[c];

    //center of this cell
    double xc[2] = {cen[c * 2], cen[c * 2 + 1]};



    for (int m = 0;
             m < nmats; ++m)
    {
      if (VolFrac[c*nmats + m] > 0.0)
      {
        int nnm = 0; //material neighbor count
        double den = 0.0;

        //loop through each neighboring cell
        for (int n = 0; n < nn; ++n)
        {
          int jc = nbrs[n]; //neighbor cellID
          if (VolFrac[jc*nmats + m] > 0.0)
          {
            double dx = xc[0] - cen[jc * 2];
            double dy = xc[1] - cen[jc * 2 + 1];
            double dsqr += dx*dx + dy*dy;

            den += DensityFrac[jc*nmats+m]/dsqr;
            ++nnm;
          }
        }

        if (nnm > 0)
          MatDensityAvg[c*nmats + m] = den / nnm;
        else
          MatDensityAvg[c*nmats + m] = 0.0;
      }
      else
      {
        MatDensityAvg[c*nmats + m] = 0.0;
      }
    }
  }
}
```

(a) Using native implementation

```cpp
void AvgDensityOverNeighbor_CellDom_Full_MM(
  int ncells,
  Field2D& VolFrac,
  Field2D& DensityFrac,

  Relation& cellNbrs,
  Field1D& cen, //cell centroid

  Field2D& MatDensityAvg)
{
  for (int c = 0; c < ncells; ++c)
  {
    // Get the neighbors for this cell
    auto nbrs = cellNbrs[c];
    int nn = nbrs.size();

    //center of this cell
    double xc[2] = {cen(c,0), cen(c,1)};

    auto MatDensityAvgRow = MatDensityAvg(c);

    for (int m = 0;
             m < MatDensityAvgRow.size(); ++m)
    {
      if ( VolFrac(c, m) > 0.0)
      {
        int nnm = 0; //material neighbor count
        double den = 0.0;

        //loop through each neighboring cell
        for (int n = 0; n < nn; ++n)
        {
          int jc = nbrs[n]; //neighbor cellID
          if( VolFrac(jc, m) > 0.0)
          {
            double dx = xc[0] - cen(jc,0);
            double dy = xc[1] - cen(jc,1);
            double dsqr = dx*dx + dy*dy;

            den += DensityFrac(jc, m) / dsqr;
            ++nnm;
          }
        }

        if( nnm > 0)
          MatDensityAvgRow(m) = den / nnm;
        else
          MatDensityAvgRow(m) = 0.0;
      }
      else
      {
        MatDensityAvgRow(m) = 0.0;
      }
    }
  }
}
```

(b) Using `MultiMat` subfield

Listing 5: Code for scenario 3: material-dependent neighbor density, for full cell-dominant layout

```
void AvgDensityOverNeighbor_CellDom_Compact_CSR(
  int ncells,
  vector<double>& DensityFrac,
  vector<int>& numNbrsInCell,
  vector<int>& cellNbrs,
  vector<double>& cen, //cell centroid
  vector<int>& beginIdx,
  vector<int>& colIdx,
  vector<double> MatDensityAvg )
{
  for (int c = 0; c < ncells; ++c)
  {
    // Get the neighbors for this cell
    int* nbrs = &(cellNbrs[ic * MAX_NBR]);
    int nn = numNbrsInCell[c];

    //center of this cell
    double xc[2] = {cen[c * 2], cen[c * 2 + 1]};



    for (int ii = beginIdx[c];
             ii < beginIdx[c + 1]; ++ii)
    {
      int m = colIdx[ii];
      int nnm = 0; //material neighbor count
      double den = 0.0;

      //loop through each neighboring cell
      for (int n = 0; n < nn; ++n)
      {
        int jc = nbrs[n]; //neighbor cellID
        for (int jj = beginIdx[jc];
                 jj < beginIdx[jc + 1]; ++jj)
        {
          if (colIdx[jj] == m)
          {
            double dx = xc[0] - cen[jc*2];
            double dy = xc[1] - cen[jc*2 + 1];
            double dsqr = dx*dx + dy*dy;

            den += DensityFrac[jj] / dsqr;
            ++nnm;
            break;
          }
        }
      }
      if(nnm > 0)
        MatDensityAvg[ii] = den / nnm;
      else
        MatDensityAvg[ii] = 0.0;
    }
  }
}
```

(a) Using native implementation of CSR

```
void AvgDensityOverNeighbor_CellDom_Compact_MM(
  int ncells,

  Field2D& DensityFrac,

  Relation& cellNbrs,
  Field1D& cen, //cell centroid

  Field2D& MatDensityAvg )
{
  for (int c = 0; c < ncells; ++c)
  {
    // Get the neighbors for this cell
    auto nbrs = cellNbrs[c];
    int nn = nbrs.size();

    //center of this cell
    double xc[2] = {cen(c,0), cen(c,1)};

    auto MatDensityAvgRow = MatDensityAvg(c);

    for (int k = 0;
             k < MatDensityAvgRow.size(); ++k)
    {
      int m = MatDensityAvgRow.matId(k);
      int nnm = 0; //material neighbor count
      double den = 0.0;

      //loop through each neighboring cell
      for (int n = 0; n < nn; ++n)
      {
        int jc = nbrs[n]; //neighbor cellID

        auto* val = DensityFrac.findValue(jc, m);

        if ( val != nullptr)
        {
          double dx = xc[0] - cen(jc, 0);
          double dy = xc[1] - cen(jc, 1);
          double dsqr = dx*dx + dy * dy;

          den += *val / dsqr;
          ++nnm;

        }
      }

      if( nnm > 0)
        MatDensityAvgRow(k) = den / nnm;
      else
        MatDensityAvgRow(k) = 0.0;
    }
  }
}
```

(b) Using `MultiMat` subfield and findValue

Listing 6: Code for scenario 3: material-dependent neighbor density, in compact cell-dominant layout

between packages, and to allow a user control over the data format. With care, a user could add such features to the matrix classes of their linear algebra libraries to achieve a multimaterial data management API, but that is the burden `MultiMat` was designed to lift from the user.

4.2 Performance Comparison

For our performance analysis, we compare `MultiMat` against native full and compact (CSR) implementations on our benchmark scenarios. For each benchmark, we have implemented cell-dominant and material-dominant variants for the full and compact representations, leading to twelve samples.

Furthermore, to compare the code on different material distributions, we use the two test data sets proposed in [7] for our timing benchmarks. Both datasets are 2D meshes with one million quadrilateral cells and fifty materials. The materials in the first dataset are defined by concentric geometric square disks whose boundaries are not aligned with the cell boundaries. In this case, about 5% of the cells contain more than one material. For the second dataset, the materials were initialized randomly such that about 80% of the cells contain a single material, about 12.5% contain two materials, about 5% contain three materials and the remaining 2.5% contain four materials. As described in [7, 8], the sizes of these datasets were chosen so that the dataset does not completely fit in L1 cache, and the distributions were selected to match initial and later distributions in multimaterial simulations. As such, the expected performance of the benchmark kernels should lie somewhere within the range of the obtained results.

Figures 5, 6 and 7 show the three test algorithms for both data sets. Our tests were executed on a 2.1 GHz Intel Xeon E5-2695 CPU computer on a linux-based operating system and were compiled in release mode with `gcc-8.1.0`. We ran each test 20 times and present the median run-time of the 20 runs.

From the plot, we can see that for the three algorithms, the templated `MultiMat` version has comparable performance to the native implementation in all cases, with little overhead. The non-templated version of `MultiMat` is slower in most cases due to reduced optimization opportunities, such as virtual function calls that the compiler is unable to inline. The non-templated version of our API can be useful for cases when users would like to focus on the development and less on performance.

It is worth noting that these micro-benchmarks are very simple, and as such exaggerate the expected overhead of `MultiMat`. Typical kernels will have significantly more computational work, effectively lowering the overhead imposed by multimaterial indirection.

5. CONCLUDING REMARKS

We have presented `MultiMat`, an open source library for managing multimaterial simulation data that can flexibly convert between different underlying data structures and layouts. Our API is easy to use and only requires users to be familiar with a few simple concepts, following the matrix-based metaphor. `MultiMat` users do not need to be aware of the underlying implementation details or bookkeeping and their code can transparently support new underlying implementations. `MultiMat` is part of the Axom project [18] which provides Computer Science infrastructure components for HPC applications.

We compared physics kernels implemented using our API with "native" implementations that directly index into multimaterial data structures. We showed that code written using our API is more straightforward, and avoids explicit index calculations and array indirections used in native implementations. We have demonstrated that, in many cases, our API affords a layout-agnostic implementation for physics kernels without sacrificing performance, as in our first and second benchmark kernels. In cases where the algorithm depends on characteristics of the layout, such as our third benchmark, it is easy to port the code using a few API calls, without adding explicit layout-specific bookkeeping or indexing details.

Our API is still undergoing development and we are planning to add support for more features. In particular, we plan to implement more compact layout options. Of particular interest is an array-based linked list representation for cell-dominant field data from the Silo I/O library [17] since this is incorporated into several multiphysics codes. A nice feature of this representation is its support for dynamic updates, e.g. adding and removing materials from fields. This representation will also help us assess and improve our iterator API since it does not support random access within a subfield. We are also investigating options for dynamic updates on GPUs and parallel architectures, such as a CSR-based dynamic data structure [19].

Our current implementation stores a field in one layout at a time. Based on user request, we are planning to allow for a field to have more than one layout if necessary. This will reduce the overhead in converting between different layouts on demand.

We are looking into incorporating `MultiMat` into Marbl, a multiphysics code at LLNL based on high order discretizations [20], and anticipate incorporating design feedback from its developers into `MultiMat`.

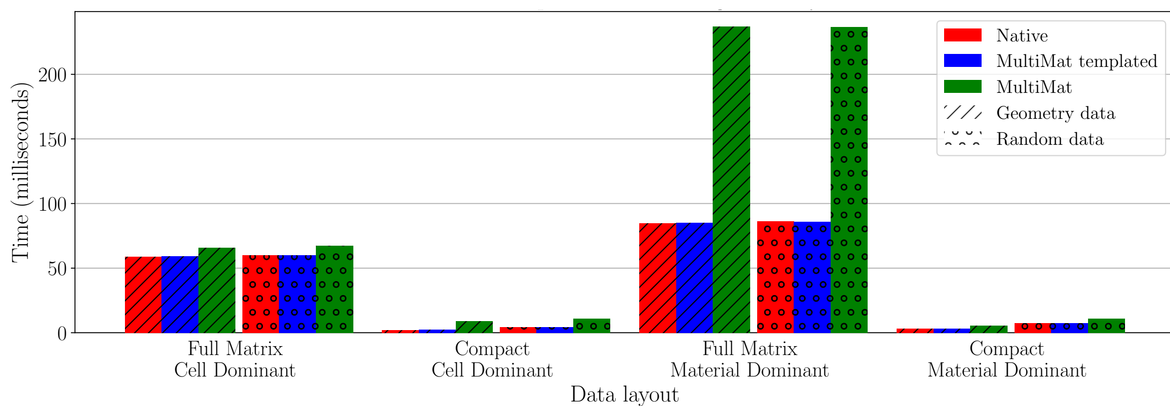


Figure 5: Timing for average density algorithm in scenario 1

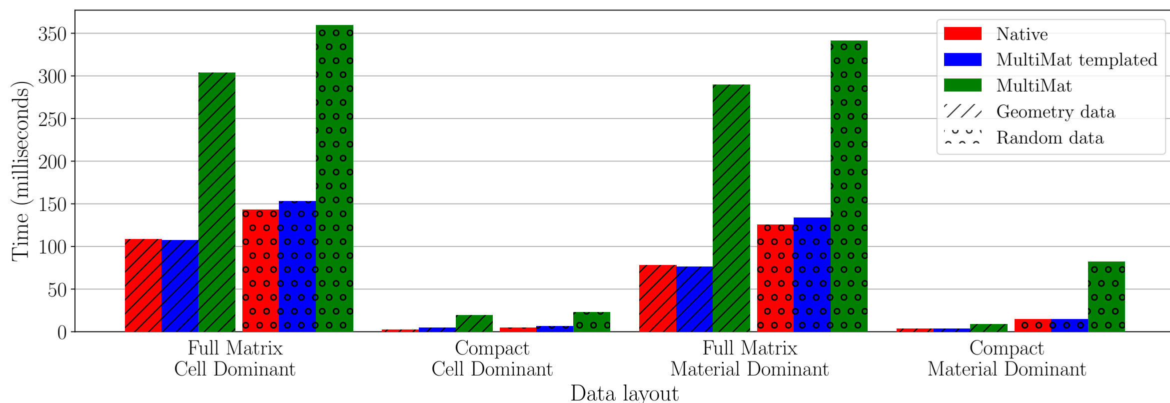


Figure 6: Timing for pressure calculated from ideal gas law in scenario 2

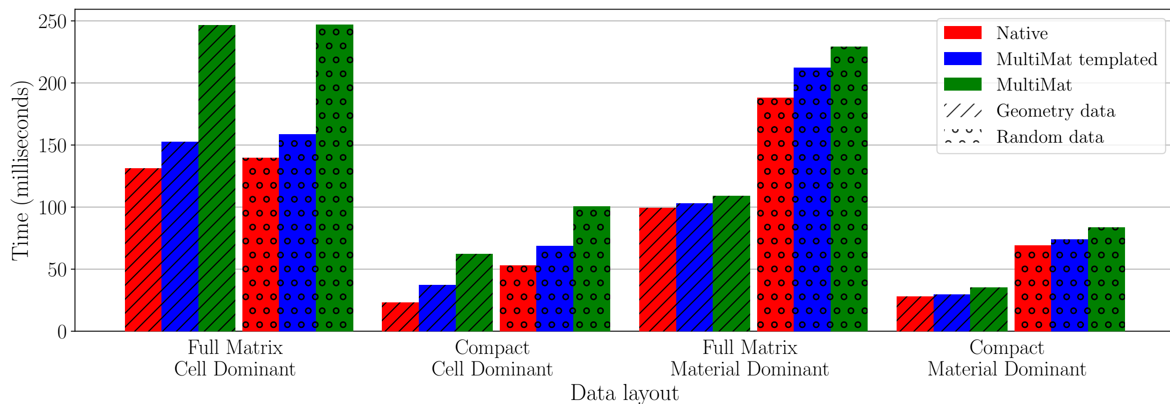


Figure 7: Timing for material density of neighboring cells algorithm in scenario 3

ACKNOWLEDGEMENTS

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344.

References

[1] Galera S., Maire P., Breil J. "A two-dimensional unstructured cell-centered multi-material ALE scheme using VOF interface reconstruction." *Journal of Computational Physics*, vol. 229, no. 16, 5755–5787, 2010

[2] Barlow A.J., Maire P.H., Rider W.J., Rieben R.N., Shashkov M.J. "Arbitrary Lagrangian–Eulerian methods for modeling high-speed compressible multimaterial flows." *Journal of Computational Physics*, vol. 322, 603–665, 2016

[3] Hirt C.W., Nichols B. "Volume of fluid (VOF) method for the dynamics of free boundaries." *Journal of Computational Physics*, vol. 39, no. 1, 201–225, 1981

[4] Raman K.S., Hurricane O.A., Park H.S., Remington B.A., Robey H., Smalyuk V.A., Drake R.P., Krauland C.M., Kuranz C.C., Hansen J.F., Harding E.C. "Three-dimensional modeling and analysis of a high energy density Kelvin-Helmholtz experiment." *Physics of Plasmas*, vol. 19, no. 9, 092112, 2012

[5] Marinak M.M., Kerbel G.D., Gentile N.A., Jones O., Munro D., Pollaine S., Dittrich T.R., Haan S.W. "Three-dimensional HYDRA simulations of National Ignition Facility targets." *Physics of Plasmas*, vol. 8, no. 5, 2275–2280, 2001

[6] Mann Z.A. "Three public enemies: cut, copy, and paste." *Computer*, vol. 39, no. 7, 31–35, 2006

[7] Fogerty S., Martineau M., Garimella R., Robey R. "A comparative study of multi-material data structures for computational physics applications." *Computers & Mathematics with Applications*, vol. 78, no. 2, 565–581, July 2018

[8] Garimella R., Robey R. "A Comparative Study of Multi-material Data Structures for Computational Physics Applications." Tech. Rep. LA-UR-16-23889, Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2017. Example code at `https://github.com/lanl/MultiMatTest`

[9] Guennebaud G., Jacob B., et al. "Eigen v3." http://eigen.tuxfamily.org, 2010

[10] Sanderson C., Curtin R. "Armadillo: A template-based C++ library for linear algebra." *Journal of Open Source Software*, vol. 1, no. 2, 26, 2016

[11] Saad Y. *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, 2nd edn., 2003

[12] Botsch M., Kobbelt L., Pauly M., Alliez P., Lévy B. *Polygon mesh processing*. AK Peters/CRC Press, 2010

[13] De Floriani L., Hui A. "Shape Representations Based on Simplicial and Cell Complexes." D. Schmalstieg, J. Bittner, editors, *Eurographics State of the Art Reports*, pp. 63–87. Prague, 2007

[14] Andrysco N., Tricoche X. "Matrix trees." *Computer Graphics Forum*, vol. 29, pp. 963–972. Wiley Online Library, 2010

[15] Sauer F., Xie J., Ma K.L. "A combined Eulerian-Lagrangian data representation for large-scale applications." *IEEE transactions on visualization and computer graphics*, vol. 23, no. 10, 2248–2261, 2016

[16] Jacobson A., Panozzo D., et al. "libigl: A simple C++ geometry processing library." https://libigl.github.io, 2018

[17] Whitlock B. "Getting data into VisIt." Tech. Rep. LLNL-SM-446033, Lawrence Livermore National Laboratory, 2010

[18] Hornung R.D., Black A., Capps A., Corbett B., Elliott N., Harrison C., Settgast R., Taylor L., Weiss K., White C., Zagaris G. "Axom: A computer science infrastructure toolkit for high performance computing.", 2017. URL `https://github.com/llnl/axom`

[19] King J., Gilray T., Kirby R.M., Might M. "Dynamic sparse-matrix allocation on GPUs." *International Conference on High Performance Computing*, pp. 61–80. Springer, 2016

[20] Anderson R., Black A., Blakeley B., Bleile R., Busby L., Camier J.S., Ciurej J., Cook A., Dobrev V., Elliott N., Grondalski J., Harrison C., Hornung R., Kolev T., Legendre M., Liu W., Nissen W., Olson B., Osawe M., Papadimitriou G., Pearce O., Pember R., Skinner A., Stevens D., Stitt T., Taylor L., Tomov V., Rieben R., Vargas A., Weiss K., White D. "The Multiphysics on Advanced Platforms Project." Tech. Rep. LLNL-TR-815869, LLNL, Oct 2020