Title: A Comparative Study of Multi-material Data Structures for Computational Physics Applications

Author(s): Garimella, Rao Veerabhadra
Robey, Robert W.

Intended for: Report

Issued: 2017-01-31 (rev.1)

A Comparative Study of Multi-material Data Structures for Computational Physics Applications

Garimella, R. (rao@lanl.gov), *T-5, Los Alamos National Laboratory*
Robey, R. (robey@lanl.gov), *XCP-2, Los Alamos National Laboratory*

Abstract

The data structures used to represent the multi-material state of a computational physics application can have a drastic impact on the performance of the application. We look at efficient data structures for sparse applications where there may be many materials, but only one or few in most computational cells. We develop simple performance models for use in selecting possible data structures and programming patterns. We verify the analytic models of performance through a small test program of the representative cases.

1 Introduction

A state manager in a multiphysics simulation helps physics kernels efficiently represent and query field or state variables pertinent to the simulation. The state variables are typically tied to a specific type of mesh entity (like cells, faces or nodes) or even discrete particles.

It is now relatively well established that the most efficient way to represent such data is in linear arrays so that the data is in contiguous memory and can be accessed efficiently without excessive cache misses. This is quite simple when each mesh entity has a single value of a variable type (density or temperature) associated with it. However, it is much more challenging in some multiphysics simulations where a mesh entity may have multiple materials or phases, each with its own value of a state variable such as density and only a few materials are present on most entities.

This is illustrated in Figure 1 where the different shaded regions represent material regions from which the mesh cells get material volume fractions. Note that not all materials are in all cells. Cells 0, 2 and 8 are pure cells (single material) with materials 1, 2 and 3 respectively. Cell 7 is the most complex with all four materials.

Assume that the physical state of materials is given by density $\rho_{C,m}$, temperature $t_{C,m}$, pressure $p_{C,m}$ and volume, $V_{C,m}$, where the subscript C refers to the mesh cell index and m refers to the material index. The state of a cell is described by these physical variables for as many materials as are present in the cell as well as the fractional volume V_f of each material. The fractional volume will be defined in cm^2 so that loops will have to divide by the volume of a cell, V_c or simply V where the meaning is clear, to get the average density. Thus, pure cell 8 would have state variables ($\rho_{8,3}$, $t_{8,3}$, $p_{8,3}$, $V_{f_{8,3}}$) but mixed cell 3 would have state variables ($\rho_{3,1}$, $t_{3,1}$, $p_{3,1}$, $V_{f_{3,1}}$, $\rho_{3,4}$, $t_{3,4}$, $p_{3,4}$,$V_{f_{3,4}}$). By the same token, mixed cell 7 would have these four variables for all four materials. Managing this complexity for large meshes where there are a lot of materials in the problem but each cell has only a small

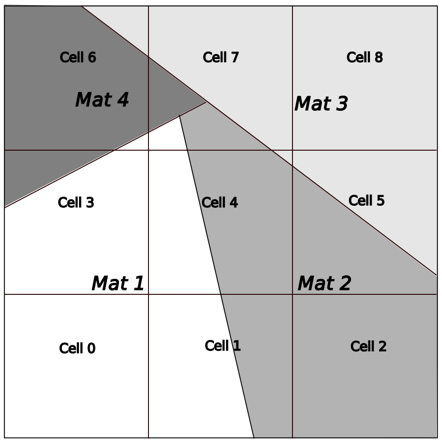


Figure 1: 3×3 mesh showing the presence of 4 materials in the mesh

subset of the materials (often just 1) and writing efficient computational kernels using these data structures is challenging.

To analyze storage schemes and how they may be used by computational scientists, we will consider three representative computational scenarios:

1. Compute, $\rho_{ave}[C]$, the weighted average density of materials in cells of a mesh (weighted by fractional volumes of the materials). This computation is a proxy for homogenization or material closure models in a cell.
2. Evaluate, $p[C][m]$, the pressure in each material contained in each cell using the ideal gas law $p(\rho, t) = nrt/v$. This computation is a proxy for more complex equation of state evaluations.
3. Evaluate, $\bar{\rho}[C][m]$, the weighted average density of each material over the node-connected neighbors of each cell, C. The weighting is inversely proportional to the square of the distance between the centroid of the cell and its neighbor. This computation explores more complex computations that access values over a neighborhood and is a proxy for a material-wise gradient calculation in each cell.

Performance Analysis

Even a cursory analysis of the first two of these computations shows that at best these will be one 8 byte memory load or store (referred to collectively as *memops*) per floating point operations (*flops*[1]). The first case will load the density plus the fractional volume (two loads) for each material in a cell and do one multiply and a reduction add. Also, there is a load of the cell volume and a divide by this volume but this is a small effect when there are lots of materials. This is a 1:1 memops to flops ratio. The second case must load material density, temperature, fractional volume and store pressure in each cell. The material constant has to be loaded for each material. This is just over four memops. There are two multiplications and a division in the ideal gas equation giving three flops. So the second case results in a 1.3:1 memops to flops ratio.

[1]Note that *flops* is the plural of flop but we will write *FLOPS* when we want to discuss floating point operations per second

This roughly 1:1 memops to flops ratio is pretty common in explicit finite difference loops. As a result these codes are almost always memory bandwidth limited. So to analyze the cost of each loop operation, we mainly count the loads and stores. In addition, we note whether memory accesses are contiguous or not.

To check the rough analysis given above, we test the average density loop for one million pure (single material) cells. We run this on a 2.7 Ghz MacBook Pro with 6 MB L3 Cache for a measured stream scale benchmark[2] of $13,375$ MB/sec. For two loads of eight bytes, one million element arrays, the memory size is 16 MB. The calculated performance should be 16 MB/ (13375 MB/sec) = 1.2 msec. The measured performance is 1.15 msec. For a second check, the pressure calculation loop with 4 memops has a 2.4 msec calculated time and a 2.66 msec measured performance.

The floating point cost for the average density loop is 2 flops per cell for a million cells or 2 Mflops. The theoretical peak performance for a single-threaded process without vectorization is 2.7 GHz times 2 flops for a multiply-add or 5.4 Gflops/sec. We then calculate 2 Mflops/1.2 msec = 1.67 Gflops/sec or about $1/3$ the floating point capability.

In addition to counting loads and stores, we will take into account some other costs associated with cache management and branch prediction. In accounting for cache misses, we have to account for the spatial and temporal locality (or lack thereof) of the accessed data. If the data has excellent spatial and temporal locality, we can expect the performance to match the stream performance. At the other extreme, if we have no locality, we may have to evict the entire cache for every load and it is as if we are loading 8 times the amount of data (assuming that the cache line is 64 bytes and each 'memop' in our calculation loads an 8 byte double). Thus, when there is a potential for cache misses, we must multiply the loads/stores by a cache miss cost, $C_p, 1 <= C_p <= 8$. To keep it simple, we will use $C_p = 8$, if there is no spatial or temporal locality and $C_p = 4$ if there is one or the other.

In algorithms that have branching, we add a branch prediction cost, B_c, for the cases where the 'if' statement code is executed and a cache miss cost, P_c, for a late prefetch operation. The hardware prefetch only occurs if the frequency of the branch is high and the instruction stream needs to be reset when the branch is true. From Agner [1], page 44, "*If the wrong branch is fed into the pipeline then the error is not detected until 10–20 clock cycles later and the work it has done by fetching, decoding and perhaps speculatively executing instructions during this time has been wasted*". For this processor architecture, a 16 cycle branch cost and a 112 cycle cache cost gives good match to measured performance. The processor frequency, ν, is 2.7 GHz for this architecture. We further tune the branch and cache miss penalties by multiplying by the branch miss frequency, B_f, to account for the higher miss rate of the branch prediction when the data lacks spatial locality. For the random data problem, we set the branch miss frequency to 1.0 and for the geometric shape initialization, we set it to 0.7. This accounts for the data already being in memory when traversing a nearly contiguous row of data for lower and upper regions of the rectangular shape. The selection of 0.7 is empirically determined to give a best fit to the performance data. Thus we can compute the branch penalty $B_p = N_b B_f (B_c + P_c)/\nu$, where N_b is number of times we encounter the branch.

Finally, if an algorithm executes small loops of unknown length (say 2-5), we assign a loop cost, L_c, taken to be about 20 cycles per exit. Then the penalty is $L_p = L_c/\nu$.

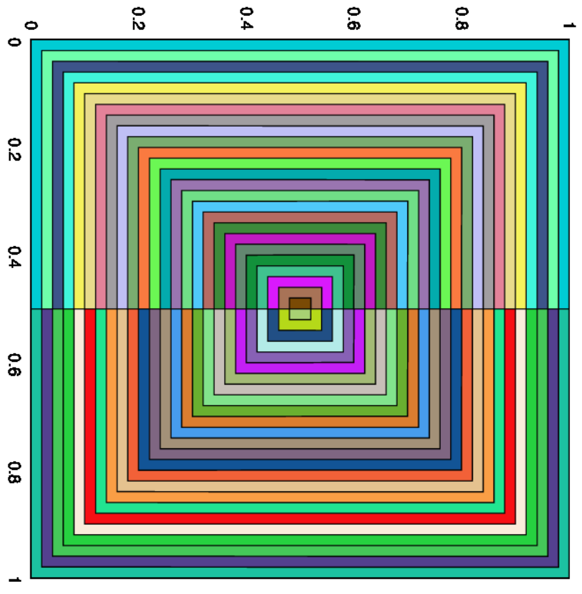


Figure 2: Material shapes used to generate fractional volumes for performance tests of various data structures. Note that corners of the condominant rectangles have been tweaked to force a larger number of mixed cells.

Test Cases

The test case used for the performance measurements is a 50 material (N_m), 1 million cell problem (N_c) with 4 state variables (N_v). This problem size is large enough to avoid being stored in L3 cache between iterations thereby forcing main memory accesses. This will more closely reflect real code performance. For systems with large caches, it may be necessary to increase the number of cells in the test. The stream benchmark recently increased their array size from 2 million to 10 million due to cache sizes getting larger. We also use a C memory allocator for two dimensional arrays that allocates a single contiguous block of memory.

We consider two test problems with different distributions of materials - the first is composed of 95% pure cells and 5% mixed cells initialized from geometric shapes as shown in Figure 2. Of the mixed cells, about 99.9% are 2 material cells (4.9% of the total), 0.06% (0.0032% of the total) are 3 material cells and 0.04% (0.0019% of the total) are 4 material cells. The number of variable values stored for mixed material cells, M_L, is $2(0.049N_c) + 3(0.000032N_c) + 4(0.000019N_c) \approx 0.098N_c$. The total number of variable values including mixed cell and pure cell values is $(0.95 + 0.098)N_c = 1.048N_c$.

The second test case is composed of 80% pure cells (P_f) and 20% mixed cells (M_f) initialized randomly. Of the mixed cells about 62.5% of the (12.5% of the total) have 2 materials, 25% (or 5% of the total) have 3 materials and 12.5% (or 2.5% of the total) have 4 materials. Therefore, the number of variable values stored for 2 material cells will be $2(0.125N_c)$, 3 material cells will be $3(0.05N_c)$ and 4 material cells will be $4(0.025N_c)$ or a total of $0.5N_c or N_c/2$ variable values for mixed material cells which we will refer to as M_L. The total number of variable values including mixed cell and pure cell values is $(0.80 + 0.5)N_c = 1.3N_c$.

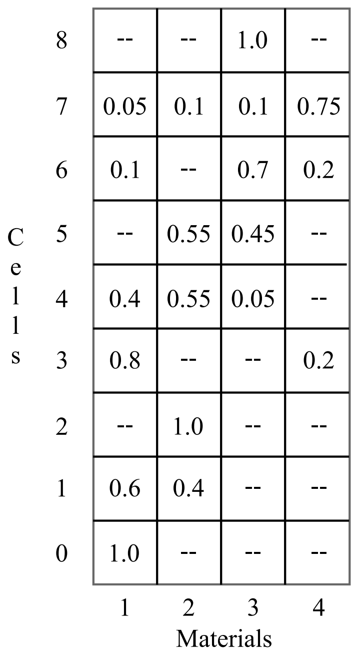
The two are end cases of a realistic simulation - materials are neatly distributed in geometric patterns in the domain only in the initial stages and become more deformed and fragmented as the simulation proceeds but they never quite reach a completely random distribution (Note that even with considerable tweaking of the geometric shapes, we could not generate more than 5% mixed cells of which 99.99% were 2 material cells). Therefore, one might expect the actual performance to be somewhere in between the two cases.

2 Full storage

The simplest approach is to assume that every material lives in every cell and just set the variables for the absent materials to be 0.0. This simplifies the storage of materials as well as the access of material states in a cell at the cost of overusing storage space for data. We will refer to this storage scheme as a as *full matrix* representation. It could also be called a *sparse matrix* but that can easily be confused with the *compressed sparse* form which we will introduce later.

There are two possible full matrix representations for the data based on which variable constitutes the outer loop of the proper storage access as the centric variable. This can be either cells or materials.

2.1 Cell-centric Full Matrix Representation



Cells	1	2	3	4
8	--	--	1.0	--
7	0.05	0.1	0.1	0.75
6	0.1	--	0.7	0.2
5	--	0.55	0.45	--
4	0.4	0.55	0.05	--
3	0.8	--	--	0.2
2	--	1.0	--	--
1	0.6	0.4	--	--
0	1.0	--	--	--

Materials

Figure 3: The Cell-centric full matrix data structure with materials stored contiguously for each cell.

The *cell-centric* data structure is shown in Figure 3. The diagram uses the C programming language notation of $\rho[C][m]$ with the convention of row data varying fastest so that materials 1 and 2 are closer in memory than cells 1 and 2.

Storage requirements

As before, assume that there are N_c cells in the mesh, N_m materials in the problem, N_{mv} material-wise variables and N_{cv} cell-wise variables. Then the storage requirements of the full matrix data structure, $Storage_{fm}$, are:

$$\begin{aligned} Storage_{fm} &= (N_{mv}N_cN_m + N_{mv}N_c + N_{cv}N_c) * 8 \\ &= [N_{mv}(N_m + 1) + N_{cv}] * N_c * 8 \end{aligned} \tag{1}$$

The first term is for the material-wise variable arrays such as ρ, t, p, and V_f. The second term is for the row pointers since we are using C. The last term is the storage for the cell-wise variables.

2.1.1 First Computational Loop – Average Cell Density

Then the first computational loop, written using *cell-dominant* logic, is shown in Algorithm 1.

Algorithm 1 *Cell-centric* algorithm to compute average density of cells using full matrix storage

```
 1: for all cells, $C$, in the mesh, up to $N_c$ do
 2:     $ave \leftarrow 0.0$                          # Not a main memory access, zero cost
 3:     for all material IDs, $m$, in the problem, up to $N_m$ do
 4:         $ave \leftarrow ave + \rho[C][m] * V_f[C][m]$
 5:                                                   # $2N_cN_m$ loads ($\rho$,$V_f$)
 6:                                                   # $2N_cN_m$ flops ($+, *$)
 7:     end for
 8:     $\rho_{ave}[C] \leftarrow ave/V[C]$           # $N_c$ loads (V)
 9:                                                   # $N_c$ stores ($\rho_{ave}$)
10:                                                   # $N_c$ flops (/)
11: end for
```

Performance analysis

We can also calculate the loads and stores for this computational loop. We will access the ρ and V_f arrays N_cN_m times each. There will be one load (V) and one store (ρ_{ave}) per cell for an additional $2N_c$ memops. Note that we do not account for the cost of local variables, such as ave, which are assumed to be in cache. Counting the flops, the first loop needs a multiply and an add for every material in every cell ($2N_cN_m$ flops) and then a division by V for each cell which is exactly one flop per word loaded. The number of arrays accessed is 4, namely, ρ, V_f, ρ_{ave} and V. The array data is accessed in contiguous order since the inner loop matches the second (row) index of the array. We summarize the performance as follows:

$$\begin{gathered} memops = 2N_c(N_m + 1), \quad flops = N_c(2N_m + 1) \\ PM = 2N_c(N_m + 1) * 8/\text{Stream} \end{gathered} \tag{2}$$

2.1.2 First Computational Loop - Average Cell Density - Conditional Variant

We can get some small improvement in the data loads and stores by testing the fractional volume for each cell and only add to the average where it is greater than zero as shown in Figure 2. This modification is shown in algorithm 2.

Algorithm 2 Modified *Cell-dominant* algorithm to compute average density of cells using full matrix storage

```
1: **for all** cells, $C$, in the mesh, up to $N_c$ **do**
2:     $ave \leftarrow 0.0$
3:     **for all** material IDs, $m$, in the problem, up to $N_m$ **do**
4:         **if** $V_f[C][m] > 0.0$ **then**                # $N_c N_m$ loads ($V_f$)
5:                                                   # $B_p N_c N_m$ branch penalty
6:             $ave \leftarrow ave + \rho[C][m] * f[C][m]$
7:                                                   # $F_f N_c N_m$ loads ($\rho$)
8:                                                   # $2F_f N_c N_m$ flops ($+, *$)
9:         **end if**
10:    **end for**
11:    $\rho_{ave}[C] \leftarrow ave/V[C]$               # $N_c$ stores ($\rho_{ave}$)
12:                                                  # $N_c$ loads (V)
13:                                                  # $N_c$ flops (/)
14: **end for**
```

Performance Analysis

For the modified algorithm performance analysis we need to define a sparsity of S_f corresponding to the frequency that the 'if' statement is false. The sparsity fraction S_f is the number of zero entries where $V_f(C,m) = 0.0$ (for the data shown in Figure 1 this is $17/36$). We define the complementary term, filled fraction, F_f, as $1 - S_f$. We can also express F_f as the average number of non-zero materials per cell or N_{mave}/N_m. For our test problems, since there are 1 million cells and 50 materials, the total number of variable entries is 50 million. However, in the first test problem only $1.048N_c = 1048000$ of those entries are non-zero (see description of the test problem in the introduction). This gives us a filled fraction, $F_f = 0.0209$ and a sparsity fraction, $S_f = 0.979$ for this problem. For the second problem with the random initialization, the number of non-zero variable values stored is $1.3N_c$ giving us a filled fraction of $F_f = 0.026$ and a sparsity fraction of $S_f = 0.974$. By testing for non-zero fractional materials the equations become:

$$\begin{gathered} memops = N_c(N_m + F_f N_m + 2), \quad flops = N_c(2F_f N_m + 1) \\ PM = N_c(N_m + F_f N_m + 2) * 8/\text{Stream} + B_p F_f N_c N_m \end{gathered} \tag{3}$$

Note that the performance model as stated above is not based only on memops. This is because accounting only for memops greatly underestimates the actual performance (as seen in the tables in the summary section). To get a more accurate model, we added a penalty for branch predicition and cache miss due to late prefetch as discussed earlier.

Testing with an example code shows, despite reducing the memory loads by a factor of two, the 'if' statement in this algorithm slows down the code a factor of three from the memory bandwidth capability. The reason for this is the branch misprediction cost and the resulting delay in memory fetching. The branch penalty is actually what we want even though the performance is slower than the earlier code because the alternative is that we unnecessarily saturate the memory bus. When there are multiple cores accessing data, this will cause a limit to performance as well as increasing the energy usage of the algorithm.

2.1.3 Second Computational Loop – Pressure for each cell and each material

The second computational loop is shown in Algorithm 3. In this case we use an 'if' test to avoid dividing by zero. This also avoids accessing all the array data where $V_f(C, m) = 0.0$ other than the fractional volume array itself.

Algorithm 3 *Cell-centric* algorithm to update material state using the full storage scheme

```
1:  for all cells, $C$, in the mesh, up to $N_c$ do
2:    for all material IDs, $m$, in the problem, up to $N_m$ do
3:      if $V_f[C][m] > 0.0$ then                  # $N_c N_m$ loads ($V_f$)
4:                                                  # $B_p N_c N_m$ branch penalty
5:        $n_m \leftarrow n(m)$                        # $F_f N_c N_m$ loads ($n$)
6:
7:        $p[C][m] \leftarrow (n_m * \rho[C][m] * t[C][m])/V_f[C][m]$
8:                                                  # $F_f N_c N_m$ stores ($p$)
9:                                                  # $2F_f N_c N_m$ loads ($\rho, t$)
10:                                                 # $3F_f N_c N_m$ flops ($*, *, /$)
11:     else
12:       $p[C][m] \leftarrow 0.0$                     # $S_f N_c N_m$ stores (p)
13:     end if
14:   end for
15: end for
```

Performance Analysis

The main loop will have $N_c N_m$ accesses for V_f and $2F_f N_c N_m$ for ρ and t. In addition, we will have $F_f N_c N_m$ accesses for the n array. Finally, there will also be $N_c N_m$ stores for the p array split across the two branches. The loop takes advantage of the sparsity for $3F_f N_m N_c$ flops.

$$\begin{aligned} memops &= N_c N_m (2 + 3F_f), \quad flops = 3F_f N_c N_m \\ PM &= N_c N_m (2 + 3F_f) * 8/\text{Stream} + B_p F_f N_c N_m \end{aligned} \tag{4}$$

In the second loop, the number of arrays accessed is 5 (n, ρ, t, V_f and p). The 'if' statement causes some difficulties, but otherwise the array data is mostly accessed in contiguous order since the inner loop, matches the second (row) index of the array. The exception is n

which has to be brought in multiple times (N_cN_m) using the *cell-centric* logic whereas it should only need to be brought in N_m times using a *material-dominant* logic. If the material data table represented by n is very large, this would be extremely wasteful and there would be a strong reason to use a *material-dominant* loop ordering to avoid the cost of the data loads.

2.1.4 Third Computational Loop – Average density of each material over neighborhood of each cell

A third computational loop that computes the average density of each material in the neighborhood of each cell is shown in Algorithm 4. We denote this average density by $\bar{\rho}$ to distinguish it from the average density of a materials in a cell, ρ_{ave}. Note that the neighborhood of a cell can be defined in any way we want (face-connected neighbors, node-connected neighbors or something else). Also, in the algorithm, it is assumed that the problem is 3D.

For simplicity we assume that the number of neighbors of a cell is constant ($\bar{N}_n$) and the neighbors of a cell are computed *a priori*, making their retrieval a constant cost operation.

The algorithm introduces a cost term, L_f, which indicates roughly what fraction of cells around a candidate cell contain a particular material if the candidate cell itself contains that material. Given that material generally stays together even under considerable stretching and formation of filamentary structures we hazard to guess that L_f is approximately 0.8. As before, we account for the conflicting costs from lack of spatial locality but presence of some temporal locality using the cache miss penalty term C_p whose values we take to be 4.

Performance Analysis

In computing the performance model, note that the cost of storing $\bar{\rho}$ is $F_fN_cN_m + S_fN_cN_m = N_cN_m$ since $F_f + S_f = 1.0$. Also, note, we do not add in a branch penalty for the inner 'if' statement in the algorithm because L_f is high (as opposed to F_f).

We can then write the following performance model:

$$\begin{aligned} memops &= N_c(3 + 2N_m + 12.5\bar{N}_n) + 4F_fN_cN_m\bar{N}_n(1 + L_f)), \\ flops &= 9N_c\bar{N}_n + 3F_fL_fN_cN_m\bar{N}_n \\ PM &= (N_c(3 + 2N_m + 12.5\bar{N}_n) + 4F_fN_cN_m\bar{N}_n(1 + L_f)) * 8/\text{Stream} + \\ &\quad B_pF_fN_cN_m \end{aligned} \tag{5}$$

2.2 *Material-centric* Full Matrix Representation

Now we look at the material-centric data structure as shown in Figure 4. The cells 1 and 2 are closer in memory than materials 1 and 2 which are a N_c stride apart in memory. The C notation for this data access is $\rho[m][C]$ and the outer loop should be over materials.

2.2.1 First Computational Loop – Average Cell Density

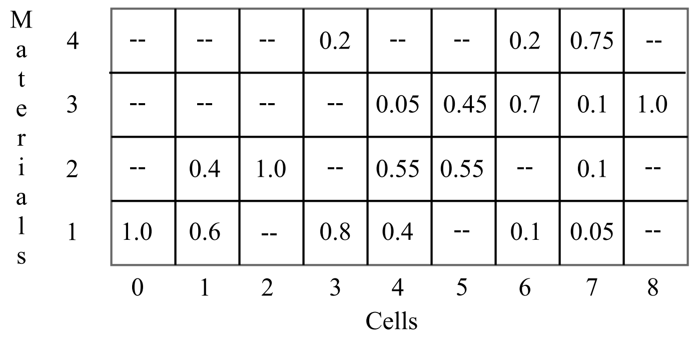
If we switch to *material-centric* data structure and loop logic (iterate over materials first), the algorithm will be as shown in Algorithm 5.

We can see right away from the additional complexity of the loop structures that the first computational case has become inefficient due to the multiple loads/stores of $\rho_{ave}[C]$. The number of memops will include N_c stores for initialization of ρ_{ave}, plus loads of $3N_cN_m$

Algorithm 4 *Cell-dominant* algorithm to compute average density of each material over a neighborhood of each cell (Assume problem is 3D)

```
 1: for all cells, $C$, in the mesh, up to $N_c$ do
 2:    $\bar{\mathbf{x}}_c \leftarrow \bar{\mathbf{x}}[C]$                              # $3N_c$ loads ($\bar{x}[C]$)
 3:    $\{c_{nbrs}\} \leftarrow nbrs[C]$                         # $N_c\bar{N}_n$ integer loads
 4:                                                # Integer loads, multiply by 0.5
 5:    for all neighbors, $i$, up to $\bar{N}_n$ do
 6:       $C_i \leftarrow c_{nbrs}[i]$
 7:       $d_{sqr}[i] \leftarrow 0.0$
 8:       for $j \leftarrow 0, 2$ do
 9:          $d_{sqr}[i] \leftarrow d_{sqr}[i] + (\bar{x}_c[j] - \bar{x}[C_i][j])^2$
10:                                                # $3N_c\bar{N}_n$ loads ($\bar{\mathbf{x}}$)
11:                                                # Partial reuse, multiply by $C_p = 4$
12:                                                # $9N_c\bar{N}_n$ flops (-,*,+)
13:       end for
14:    end for
15:    for all material IDs, $m$, in the problem, up to $N_m$ do
16:       if $V_f[C][m] > 0.0$ then                  # $N_cN_m$ loads ($V_f$)
17:                                                # $B_pN_cN_m$ branch penalty
18:          $\rho_{sum} \leftarrow 0$
19:          $N_n \leftarrow 0$                          # local variable $N_n$, not global variable $\bar{N}_n$
20:          for all neighbors, $i$ up to $\bar{N}_n$ do
21:             $C_i \leftarrow c_{nbrs}[i]$
22:             if $V_f[C_i][m] > 0.0$ then           # $F_fN_cN_m\bar{N}_n$ loads ($V_f$)
23:                                                # Partial reuse, multiply by $C_p = 4$
24:                $\rho_{sum} \leftarrow \rho_{sum} + \rho[C_i][m]/d_{sqr}[i]$
25:                                                # $F_fN_cN_mL_f\bar{N}_n$ loads ($\rho$)
26:                                                # Partial reuse, multiply by $C_p = 4$
27:                                                # $2F_fN_cN_mL_f\bar{N}_n$ flops (/,+)
28:                $N_n \leftarrow N_n + 1$                 # $F_fN_cN_mL_f\bar{N}_n$ flops (+)
29:             end if
30:          end for
31:          $\bar{\rho}[C][m] \leftarrow \rho_{sum}/N_n$               # $F_fN_cN_m$ stores ($\bar{\rho}$)
32:                                                # $F_fN_cN_m$ flops (/)
33:       else
34:          $\bar{\rho}[C][m] \leftarrow 0.0$                  # $S_fN_cN_m$ stores
35:       end if
36:    end for
37: end for
```

for ρ, and V_f and N_cN_m stores of ρ_{ave} in the main loop. Note that we count the load and store of ρ_{ave} as a single memop instead of two. Then in the final loop there are N_c loads for V and ρ_{ave} plus N_c stores of ρ_{ave}. We get contiguous access of variables at the cost of



Materials									
4	--	--	--	0.2	--	--	0.2	0.75	--
3	--	--	--	--	0.05	0.45	0.7	0.1	1.0
2	--	0.4	1.0	--	0.55	0.55	--	0.1	--
1	1.0	0.6	--	0.8	0.4	--	0.1	0.05	--
	0	1	2	3	4	5	6	7	8
	Cells								

Figure 4: The Material-centric full data structure has cells stored contiguously for each material.

Algorithm 5 *Material-dominant* algorithm to compute average density of cells using full storage

```
1:  for all cells, $C$, in the mesh, up to $N_c$ do
2:      $\rho_{ave}[C] \leftarrow 0.0$                         # $N_c$ stores ($\rho_{ave}$)
3:  end for
4:  for all material IDs, $m$, in the problem, up to $N_m$ do
5:      for all cells, $C$, in the mesh, up to $N_c$ do
6:          $\rho_{ave}[C] \leftarrow \rho_{ave}[C] + \rho[m][C] * V_f[m][C]$
7:                                                        # $N_c N_m$ stores ($\rho_{ave}$)
8:                                                        # $2N_c N_m$ loads ($\rho, V_f$)
9:                                                        # $2N_c N_m$ flops ($+, *$)
10:     end for
11: end for
12: for all cells, $C$, in the mesh, up to $N_c$ do
13:     $\rho_{ave}[C] \leftarrow \rho_{ave}[C]/V[C]$          # $2N_c$ loads/stores ($\rho_{ave}, V$)
14:                                                       # $N_c$ flops (/)
15: end for
```

loading them multiple times. The resulting performance equations are:

$$\begin{aligned} memops = 3N_c(N_m+1), \quad flops = 2N_cN_m + N_c, \\ PM = 3N_c(N_m+1) * 8/\text{Stream} \end{aligned} \tag{6}$$

We could get some better performance by ”blocking” loops. In this case, we could block the cell loop for a cache line size of about 8 doubles and with memory aligned properly. Most codes, however, do not write loops this way as it is more tedious.

Another optimization would be to only do the operation for $V_F > 0.0$, as before. The algorithm for the modified loop and its performance analysis is included in the appendix (See Algorithm 21). Although the memops to flops ratios degrade for this problem it does not affect the performance of the algorithm appreciably. The performance statistics for the two large problems are presented in tables in the summary section.

2.2.2 Second Computational Loop – Pressure for each cell and each material

A reordering of loops and data accesses also can be made for the second computational loop as shown in Algorithm 6.

Algorithm 6 *Material-dominant* algorithm to update material state using the full storage scheme

```
1:  for all material IDs, $m$, in the problem, up to $N_m$ do
2:      $n_m \leftarrow n(m)$                              # $N_m$ loads ($n_m$)
3:      for all cells, $C$, in the mesh, up to $N_c$ do
4:          if $V_f[m][C] > 0.0$ then                      # $N_c N_m$ loads ($V_f$)
5:                                                         # $B_p N_c N_m$ branch penalty
6:
7:              $p[m][C] \leftarrow n_m * \rho[m][C] * t[m][C]/V_f[m][C]$
8:                                                         # $2F_f N_c N_m$ loads ($\rho, t$)
9:                                                         # $F_f N_c N_m$ stores (p)
10:                                                        # $3F_f N_c N_m$ flops ($*, *, /$)
11:         else
12:             $p[m][C] \leftarrow 0.0$                   # $S_f N_c N_m$ stores (p)
13:         end if
14:     end for
15: end for
```

Performance Analysis

The multiple material table data loads represented by n will be far more ideal and the second loop becomes more efficient. The material loop loads the material constant N_m times. Then there are $N_c N_m$ loads of V_f for the 'if' test and $F_f N_c N_m$ stores of p and $2F_f N_c N_m$ loads in the main loop with $3F_f N_c N_m$ flops. Finally, there are $S_f N_c N_m$ stores of zero into p. We account for the 'if' statement as before with a penalty due to branching.

$$\begin{aligned} memops &= N_m(1 + 2F_f N_c + 2N_c), \quad flops = 3F_f N_c N_m \\ PM &= N_m(1 + 2F_f N_c + 2N_c) * 8/\text{Stream} + B_p F_f N_c N_m \end{aligned} \tag{7}$$

2.2.3 Third Computational Loop – Average density of each material over neighborhood of each cell

The third computational loop computing the average density of each material in the neighborhood of each cell is shown in Algorithm 7. The only difference between this algorithm and the cell-centric one (Algorithm 4) is that the outer loop is on materials.

Performance Analysis

The performance model for this case is similar to that of the cell-centric case except that the loads of $\bar{x}[C]$ and $nbrs[C]$ have moved further inside the loops multiplying their costs by N_m. Thus the performance model can be stated as shown below:

Algorithm 7 *Material-dominant* algorithm to compute average density of each material over a neighborhood of each cell (Assume problem is 3D)

```
for all material IDs, $m$, in the problem, up to $N_m$ do
    for all cells, $C$, in the mesh, up to $N_c$ do
        if $V_f[m][C] > 0.0$ then                      # $N_cN_m$ loads ($V_f$)
                                                       # $B_pN_cN_m$ branch penalty
            $x_c \leftarrow \bar{x}[C]$                 # $3F_fN_cN_m$ loads ($\bar{x}[C]$)
            $\rho_{sum} \leftarrow 0$
            $N_n \leftarrow 0.0$                        # Local var. $N_n$, not global var. $\bar{N}_n$
            for neighbor cell index $i \leftarrow 0, \bar{N}_n - 1$ do
                $C_i \leftarrow nbrs[C][i]$             # $F_fN_cN_m\bar{N}_n$ integer loads ($nbrs$)
                                                       # Integer loads, multiply by 0.5
                if $V_f[m][C_i] > 0.0$ then            # $F_fN_cN_m\bar{N}_n$ loads ($V_f$)
                                                       # Partial reuse, multiply by $C_p = 4$
                    $d_{sqr} \leftarrow 0.0$
                    for $j \leftarrow 0, 2$ do
                        $d_{sqr} \leftarrow d_{sqr} + (x_c[j] - \bar{x}[C_i][j])^2$
                                                       # $3F_fN_cN_mL_f\bar{N}_n$ loads ($\bar{x}[C_i]$)
                                                       # Partial reuse, multiply by $C_p = 4$
                                                       # $9F_fN_cN_mL_f\bar{N}_n$ flops (-,*,+)
                    end for
                    $\rho_{sum} \leftarrow \rho_{sum}$ + $\rho[m][C_i]/d_{sqr}$
                                                       # $F_fN_cN_mL_f\bar{N}_n$ loads ($\rho$)
                                                       # Partial reuse, multiply by $C_p = 4$
                                                       # $2F_fN_cN_mL_f\bar{N}_n$ flops (/,+)
                    $N_n \leftarrow N_n + 1$            # $F_fN_cN_mL_f\bar{N}_n$ flops (+)
                end if
            end for
            $\bar{\rho}[m][C] \leftarrow \rho_{sum}/N_n$   # $F_fN_cN_m$ stores ($\bar{\rho}$)
                                                       # $F_fN_cN_m$ flops (/)
        else
            $\bar{\rho}[m][C] \leftarrow 0.0$           # $S_fN_cN_m$ stores ($\bar{\rho}$)
        end if
    end for
end for
```

$$\begin{aligned} memops &= N_cN_m(2 + F_f(3 + \bar{N}_n(4.5 + 16L_f))), \\ flops &= F_fN_cN_m + 12F_fL_fN_cN_m\bar{N}_n \\ PM &= (N_cN_m(2 + F_f(3 + \bar{N}_n(4.5 + 16L_f))) * 8/\text{Stream} + \\ &\quad B_pF_fN_cN_m \end{aligned} \tag{8}$$

3 Compact storage

Compact storage representations store the variable data for a material in a cell only if the material is present in the cell (i.e., its volume fraction is non-zero). We also refer to this as a *compressed sparse* representation since that is the terminology used in the matrix community for their compact storage, such as *compressed sparse row* (CSR) and similar terms.

3.1 *Cell-centric* Compact Storage

Here we present a compact storage scheme for multimaterial data that is based on that currently used in the Roxane code[2]. The general strategy is to use a linked-list of materials. As shown in Figure 5, the empty data cells are first squeezed out so that only those with information are retained. Then the pure cells are removed since we already have that information in the cell state arrays. The remaining data could then be accessed in ragged right form such as $\rho[C][m]$. But for short lists, this adds an 8 byte pointer for each list for each variable, which is too high an overhead. So we concatenate the list data and use a special linked list that is in array form where the pointer to the start of the list for each cell is an index into an array. The next set of fractional material immediately follows it. This gives better cache behavior than the typical linked-list as the data is accessed in contiguous order. The offset into all the mixed-material lists is the same for all the state variables, so we only need a single offset value for all of them and the addressing is now a single-dimensioned array accessor in the form $\rho[mstart]$.

The resulting data structure with the *abs(imaterial)* and *nmats* pointing into the mixed-material lists is shown in Figure 6. The length of each material list can be found either from the *nmats* array or looping over the *nextfrac* array until it hits a -1, indicating the end of the list. The advantage of the *nextfrac* array is that each entry points to the next item in the list, allowing added materials to be placed at the end of the entire mixed material list instead of copying all the existing materials in the cell to be moved there.

Storage costs

To compute the storage costs for this representation we use the length of the mixed material arrays M_L in the first test problem (approximately $0.5N_c$ as derived in the introduction). The storage costs for this compact representation is M_L for four doubles and three integers, plus N_c for four double and two integers. For the million cell mesh with 50 materials and a mixed material length, $500,000$, the storage would be 62.0 MB for a reduction of $\approx$96% from the full matrix representation. The memory savings approaches that of the sparsity of the data as would be expected. The net impact of this storage reduction is the ability to run larger problems on a node and because these applications are bandwidth limited, we expect to see a corresponding decrease in runtime.

Assuming, as before, that we have to represent N_{mv} material based variables and N_{cv} cell based variables we can state the storage estimate for the cell-centric representation as

$$Storage_{cc} = 4(3M_L + 2N_c) + 8(M_L + N_c)N_{mv} + 8N_cN_{cv} \tag{9}$$

[2]Roxane is a LANL cell-based AMR Eulerian hydrocode

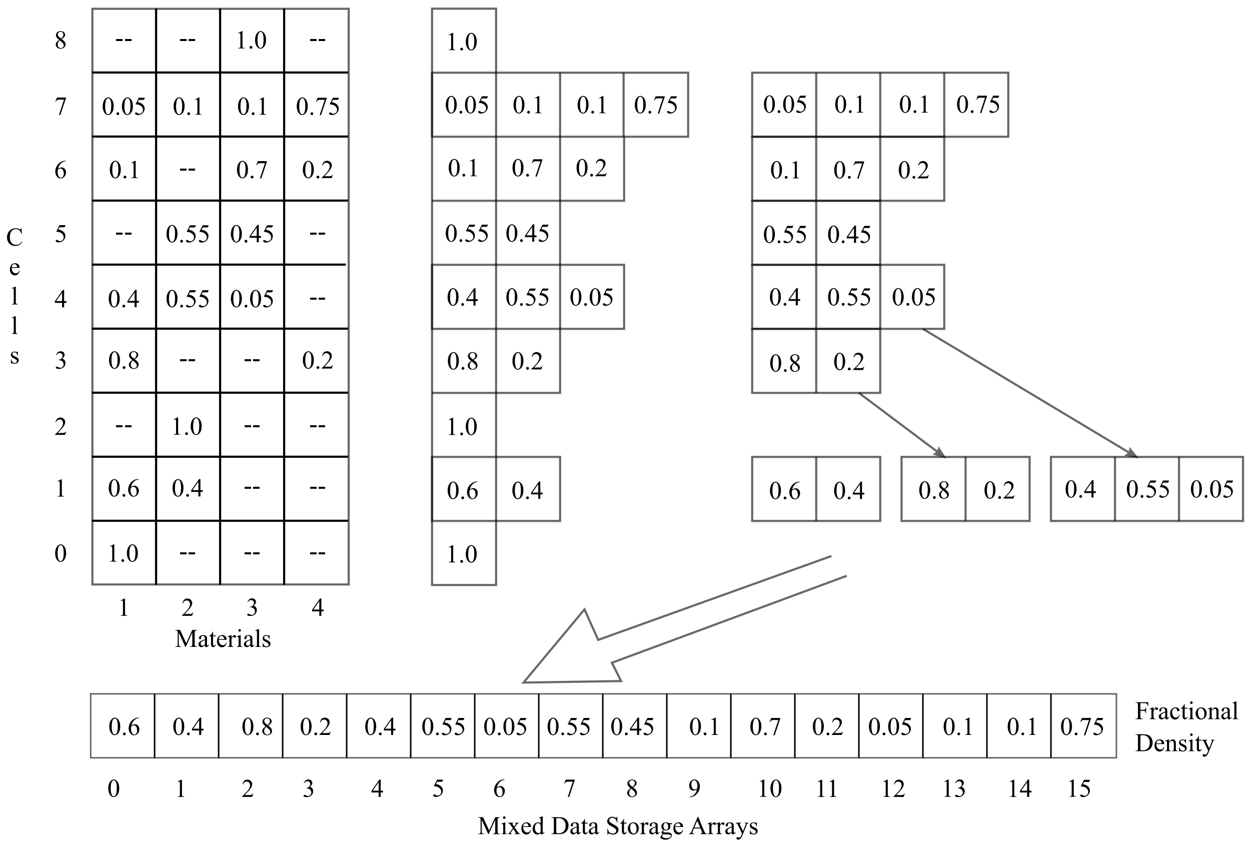


Figure 5: The cell-centric compact data structure has a linked-list for each cell that has multiple materials.

3.1.1 First Computational Loop – Average Cell Density

For the C/C++ data structure, we use the $\rho[abs(imaterial)]$ offset into the linear linked-list data structure. Then we loop over the fractional materials, summing the fractional densities times the fractional volumes, until we hit a -1 value as shown in Algorithm 8.

Note the loop over cells with an inner loop over materials that only loops for as many materials as there are in each cell. This shows just how much of an advantage the compact storage representation can achieve. First, if there are no fractional materials in a cell, we do nothing because the stored density is already the average density. If there are fractional materials, we average them by their weighted volume in the cell[3].

Performance analysis

The number of loads is N_c for the integer array imaterial plus M_L loads of nextfrac and $2M_L$ of ρ and V_f. After the material loop, there are $M_f N_c$ stores of ρ and $M_f N_c$ loads of V. In the material loop, there are $2M_L$ flops plus $M_f N_c$ flops after the loop. This gives us the following performance equations:

[3]See Appendix for algorithm and performance analysis when the average density is not a stored variable but must be created on the fly

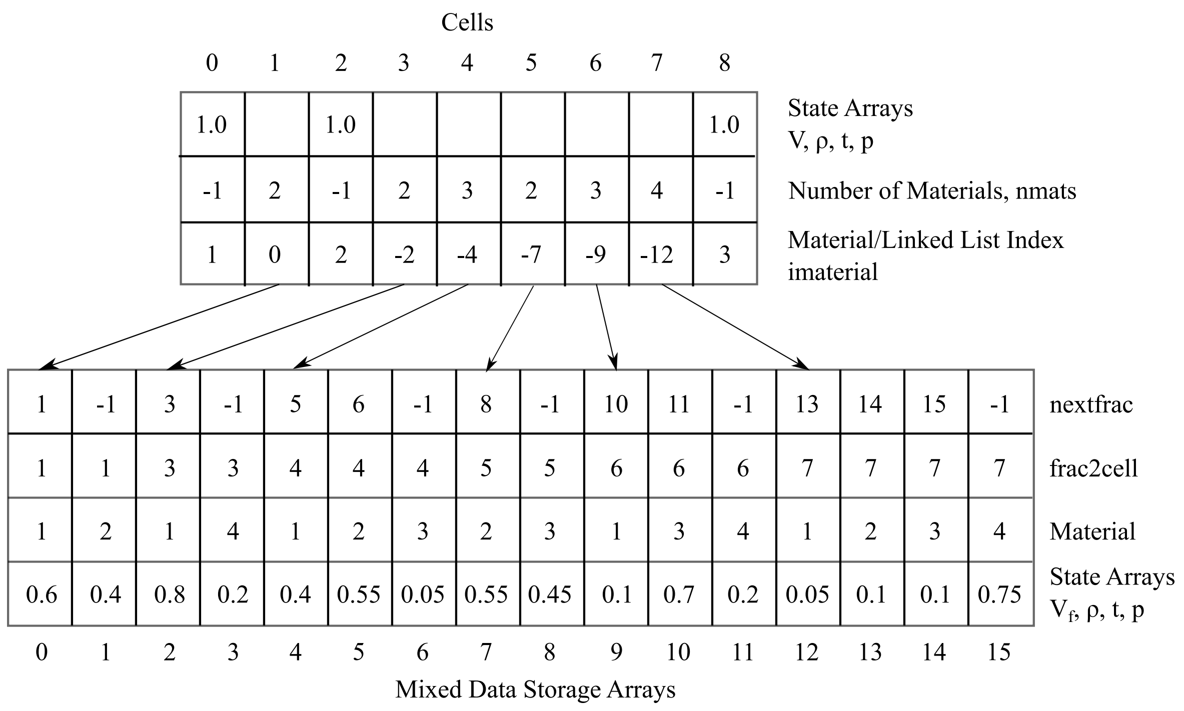


Figure 6: The mixed material arrays for the *cell-centric* compact data structure.

$$\begin{aligned} memops &= N_c(1 + 2M_f) + 3M_L, \quad membytes = (4 + 2M_f * 8)N_c + 20M_L, \\ flops &= 2M_L + M_f N_c, \quad PM = membytes/\text{Stream} + L_p M_f N_c \end{aligned} \tag{10}$$

For this analysis we have to separate out 4 byte integer and 8 byte memory loads. Note that, the material loop has an unknown but small length thereby incurring a complex loop cost, L_c, of 20 cycles per exit.

If the fractional material data are ordered in the same access pattern as the cell data, the loads are pretty much contiguous. A careful reader will note that one of the advantages of this data structure is that the cell average densities never have to be calculated since as a pure cell, they are already stored in the ρ cell array. Also, we could use the $nmats$ array to determine how many iterations for the inner material loop. The modified algorithm and performance analysis is shown in the appendix as Algorithm 22 and the estimated performance and actual performance are included in the tables in the summary section.

3.1.2 Second Computational Loop – Pressure for each cell and each material

The loop to calculate pressure for each cell and each material for the *cell-centric* compact storage is shown in Algorithm 9.

Most of the data is loaded in a contiguous order except for the material property n. This will be loaded multiple times more for each material than necessary.

$$\begin{aligned} memops &= (1 + 5P_f)N_c + 6M_L, \quad membytes = (4 + 5P_f * 8)N_c + 40M_L, \\ flops &= 3M_L + 3P_f Nc, \quad PM = membytes/\text{Stream} + L_p M_f N_c \end{aligned} \tag{11}$$

If the material property is something like an ideal gas law, this would not be too bad. But for a table-based EOS with a couple of hundred data values, it would perform poorly.

Algorithm 8 *Cell-dominant* algorithm to compute average density of cells using compact storage

```
for all cells, $C$, in the mesh, up to $N_c$ do
    $ave \leftarrow 0.0$
    $ix \leftarrow imaterial[C]$                         # $N_c$ loads ($imaterial$)
    if $ix <= 0$ then
        for $ix \leftarrow -ix$, until $ix < 0$ do
                                                   # $M_L$16 cycle branch miss
            $ave \leftarrow ave + \rho[ix] * V_f[ix]$      # $2M_L$ loads ($\rho, V_f$)
                                                   # $M_L$ flops ($+, *$)
            $ix \leftarrow nextfrac[ix]$                 # $M_L$ loads ($nextfrac$)
        end for
        $\rho[C] \leftarrow ave/V[C]$                     # $M_f N_c$ stores ($\rho_{ave}$)
                                                   # $M_f N_c$ loads ($V$)
                                                   # $M_f N_c$ flops ($/$)
    end if
end for
```

3.1.3 Third Computational Loop - Average density of each material over neighborhood of each cell

Performance Analysis

The complex logic of the algorithm to compute average material density over a neighborhood using the cell centric algorithm makes it challenging to account for its memops and flops. Still, we can reduce the complexity by combining terms from the two branches using the recognition that $F_f N_c N_m = P_f N_c + M_L$ - in other words, the number of non-zero entries for any variable is the number of of mixed cell entries in the mesh (M_L) plus the number of pure cell entries ($P_f N_c$). Then we can write the performance model as:

$$\begin{aligned} memops &= N_c(3.5 + 12.5\bar{N}_n) + 0.5M_L + N_c N_m(1 + 1.5F_f) + \\ &\quad 2F_f N_c N_m \bar{N}_n(1 + 2\bar{N}_{mc} + 2L_f) \\ flops &= 9N_c\bar{N}_n + 3F_f N_c N_m \bar{N}_n L_f + F_f N_c N_m \\ PM &= (N_c(3.5 + 12.5\bar{N}_n) + 0.5M_L + N_c N_m(1 + 1.5F_f) + \\ &\quad 2F_f N_c N_m \bar{N}_n(1 + 2\bar{N}_{mc} + 2L_f)) * 8/\text{Stream} + B_p M_f \end{aligned} \tag{12}$$

3.1.4 Adding to the Cell-centric Compact Data Structures

Adding to the mixed material data arrays is made simpler by having extra array length of 10–20%. We first test to see if the material exists in the cell and then if it doesn't, we move the existing material data to the end of the mixed material array and add the new material as shown in Algorithm 11. When threading, either with OpenMP or OpenCL/Cuda, we have to use an atomic operation to extend the array so that another thread does not get the same storage.

Algorithm 9 *Cell-centric* algorithm to update material state using the compact storage scheme

```
1:  for all cells, $C$, in the mesh, up to $N_c$ do
2:     $ix \leftarrow imaterial[C])$                        # $N_c$ loads ($imaterial$)
3:     if $ix <= 0$ then
4:        for $ix \leftarrow -ix$, until $nextfrac[ix] < 0$ do
5:                                                          # $M_L$ loads ($nextfrac$)
6:           $n_m \leftarrow n(matids[ix])$                 # $M_L$ loads ($matids$)
7:           $p[ix] \leftarrow (n_m * \rho[ix] * t[ix])/V_f[ix]$   # $3M_L$ loads ($\rho, t, V_f$)
8:                                                          # $M_L$ store ($p$)
9:                                                          # $3M_L$ flops ($*, *, /$)
10:       end for
11:    else
12:       $p[C] \leftarrow$                                 # $P_f N_c$ stores ($p$)
13:          $n(imaterial[C]) * \rho[C] * t[C]/V[C]$        # $4P_f N_c$ loads ($n, \rho, t, V$)
14:                                                         # $3P_f N_c$ flops ($*, *, /$)
15:    end if
16: end for
```

The operation of adding a new material to a cell is also shown in Figure 7. Material 4 has just moved into cell 4 in the problem from Figure 1. The new material has been added to the end of the array and the nextfrac entry is set to point to it.

When removing a material, we simply remove the link to the data and zero it out. Thus it will never be accessed.

At the end of each cycle or few cycles, we reorder the arrays by walking through them, copying the data in order to the new array as shown in Algorithm 12. This reordering causes the data access patterns to be in sequence and cache friendly as well as reclaiming holes in the mixed-material arrays.

3.2 Material-centric Compact Storage

The *material-centric* compact storage, as the name indicates, is organized around subsets of mesh cells corresponding to each material. Each mesh subset carries two lists:

1. *subset2mesh* - A list of mesh cell IDs in the subset (represents a map from the subset to the mesh)
2. *mesh2subset* - A list that maps mesh cell IDs to their local index in the subset (reverse map from the mesh to the subset).

If we don't use a smart array that knows its own size to represent the *subset2mesh* array, we must also have a an array *ncellsmat* to represent the number of cells in a subset. However, this is a negligible cost since it is only as many integers as there are materials.

The concept of subsets is illustrated in Figure 8 for the multi-material example of Figure 1. Referring to material 4 in the figure, it can be seen that the *subset2mesh* for material subset 4 contains cells 3, 6 and 7 of the mesh and the reverse map, *mesh2subset*, indicates that mesh cells 3, 6 and 7 correspond to the entries 0, 1 and 2 in the subset.

Algorithm 10 *Cell-centric* algorithm to compute average material density over a neighborhood using compact storage

```
for all cells, $C$, in the mesh, up to $N_c$ do
    Initialize $\bar{\rho}$ to 0                         # $N_c N_m$ stores from Alg. 4
    $\bar{\mathbf{x}}_c \leftarrow \bar{\mathbf{x}}[C]$                              # $3N_c$ loads ($\bar{x}[C]$). Assume 3D
    $\{c_{nbrs}\} \leftarrow nbrs[C]$                     # $N_c \bar{N}_n$ integer loads
                                                  # Integer loads, multiply by 0.5
    Initialize $d_{sqr}$ w.r.t. neighbors          # $3N_c \bar{N}_n$ loads ($\bar{\mathbf{x}}$) from Alg. 4
                                                  # Partial reuse, multiply by $C_p = 4$
                                                  # $9N_c \bar{N}_n$ flops (-,*,+)

    $ix \leftarrow imaterial[C]$                        # $N_c$ loads ($imaterial$)
                                                  # Integer loads, multiply by 0.5

    if $ix <= 0$ then                             # Multiple materials in cell, $C$
                                                  # $B_p M_f$ branch penalty
        for $ix \leftarrow -ix$, until $nextfrac[ix] < 0$ do
                                                  # $M_L$ loads ($nextfrac$)
                                                  # Integer loads, multiply by 0.5
            $m \leftarrow matids[ix]$                     # $M_L$ loads $matids$
                                                  # Integer loads, multiply by 0.5
            $\rho_{sum} \leftarrow 0.0$
            $N_n \leftarrow 0$                            # Local var. $N_n$, not global var. $\bar{N}_n$
            for all neighbors, $j$ up to $\bar{N}_n$ do
                $C_j \leftarrow c_{nbrs}[j]$
                $jx \leftarrow imaterial[C_j]$            # $M_L \bar{N}_n$ loads $imaterial$
                                                  # Integer loads, multiply by 0.5
                                                  # Partial reuse, multiply by $C_p = 4$

                if $jx <= 0$ then                 # Multiple materials in cell, $C_i$
                    for $jx \leftarrow -jx$, until $nextfrac[jx] < 0$ and not $found$ do
                                                  # $M_L \bar{N}_n \bar{N}_{mc}$ loads ($nextfrac$)
                                                  # Integer loads, multiply by 0.5
                                                  # Partial reuse, multiply by $C_p = 4$
                        if $matids[jx] == m$ then # $M_L \bar{N}_n \bar{N}_{mc}$ loads ($matids$)
                                                  # Integer loads, multiply by 0.5
                                                  # Partial reuse, multiply by $C_p = 4$
                            $\rho_{sum} \leftarrow \rho_{sum} + \rho[jx]/d_{sqr}[j]$
                                                  # $M_L \bar{N}_n L_f$ loads ($\rho$)
                                                  # Partial reuse, multiply by $C_p = 4$
                                                  # $2M_L \bar{N}_n L_f$ flops (/,+)
                            $N_n \leftarrow N_n + 1$              # $M_L \bar{N}_n L_f$ flops (+)
                        end if
                    end for
                else                              # Single material in cell $C_j$ (see next page)
            if $matids[jx] == m$ then          # Assume loads of $matids$ are roughly
                                               # accounted for in other branch
               $\rho_{sum} \leftarrow \rho_{sum} + \rho[jx]/d_{sqr}[j]$
               $N_n \leftarrow N_n + 1$                  # Loads, flops accounted for in other branch
            end if
         end if
      end for
      $\bar{\rho}[C][m] \leftarrow \rho_{sum}/N_n$                # $M_L$ stores, $M_L$ flops (/)
   end for
else                                           # Pure cell $C$
   $m \leftarrow matids[ix]$                          # $P_f N_c$ loads $matids$
                                               # Integer loads, multiply by 0.5
   $\rho_{sum} \leftarrow 0.0$
   $N_n \leftarrow 0$                                 # Local var. $N_n$, not global var. $\bar{N}_n$
   for all neighbors, $j$ up to $\bar{N}_n$ do
      $C_j \leftarrow c_{nbrs}[j]$
      $jx \leftarrow imaterial[C_j]$                  # $P_f N_c \bar{N}_n$ loads $imaterial$
                                               # Integer loads, multiply by 0.5
                                               # Partial reuse, multiply by $C_p = 4$
      if $jx <= 0$ then
         for $jx \leftarrow -jx$, until $nextfrac[jx] < 0$ and not $found$ do
                                               # $P_f N_c \bar{N}_n \bar{N}_{mc}$ loads ($nextfrac$)
                                               # Integer loads, multiply by 0.5
                                               # Partial reuse, multiply by $C_p = 4$
            if $matids[jx] == m$ then          # $P_f N_c \bar{N}_n \bar{N}_{mc}$ loads ($matids$)
                                               # Integer loads, multiply by 0.5
                                               # Partial reuse, multiply by $C_p = 4$
               $\rho_{sum} \leftarrow \rho_{sum} + \rho[jx]/d_{sqr}[j]$
                                               # $P_f N_c \bar{N}_n L_f$ loads ($\rho$)
                                               # Partial reuse, multiply by $C_p = 4$
                                               # $2P_f N_c \bar{N}_n L_f$ flops (/,+)
               $N_n \leftarrow N_n + 1$                  # $P_f N_c \bar{N}_n L_f$ flops (+)
            end if
         end for
      else
         if $matids[jx] == m$ then
            $\rho_{sum} \leftarrow \rho_{sum} + \rho[jx]/d_{sqr}[j]$
            $N_n \leftarrow N_n + 1$                     # Loads, flops accounted for in other branch
         end if
      end if
   end for
   $\bar{\rho}[C][m] \leftarrow \rho_{sum}/N_n$                   # $P_f N_c$ stores ($\bar{\rho}$), $P_f N_c$ flops (/)
end if
end for
```

Algorithm 11 Add material to cell, check for existence and expand mixed material if not

```
$\rho_{new}[C][matid]$                              # add material to cell
$match \leftarrow true$
**for all** material IDs, $m$, in the cell, up to $nmats[C]$ **do**
  **if** $matid_{new} == matid[C][m]$ **then**
    $r[C][m] \leftarrow r[C][m] + \rho_{new}[C][matid]$ break
  **end if**
  $match \leftarrow false$
**end for**
**if** match == false **then**
  $matinsert \leftarrow atomic\_compxchange(oldvalue, maxmat + nmats[C] + 1)$
  **for all** material IDs, $m$, in the cell, up to $nmats[C]$ **do**
    $r[maxmat + m] \leftarrow r[C][m]$
  **end for**
  $r[maxmat + m + 1] \leftarrow \rho_{new}[C][matid]$
**end if**
```

The representation of material-based variables in this scheme is the equivalent of a ragged right array where an array of pointers points into the values for the cells of the individual materials. The values for each material mirror the *subset2mesh* array for the material and therefore, store only as much information as needed. Thus, for material 4, only 3 values each of f_4, ρ_4, p_4 and t_4 corresponding to mesh cells 3, 6 and 7 are stored (the figure only shows f_m and ρ_m for each material m). It is possible to condense this ragged right array into one integer and one double array but introducing new cells into a subset would then require expensive memory movement.

In addition to these data structures, we define two integer arrays to allow us to access the IDs of materials in any given cell. The first array called *nmats* contains the number of materials in each cell. Since we expect the maximum number of materials in any cell, N_{mc}^{max}, to be at most 4 this could even be a character array. The second array contains the IDs of materials in each cell and is of length ($N_c N_{mc}^{max}$) with a dummy value like -1 for empty slots. For example, for cell 1, *nmats* is 2 and the *matids* entries are 1, 2, -1, -1 as shown in 8. One could make this a compact array which stores only the materials that are in a cell but once again introduction of new materials to a cell requires expensive memory motion.

Finally, we store one array of double values for the volume of cells, V.

Storage costs

Assuming that the probability of any material being present in any cell is equal, the average number of cells containing a material, $\bar{N}_{mc}$ can be computed as the ratio of the total number of non-zero variable values $F_f N_c N_m$ to the number of materials N_m, or in other words, $\bar{N}_{mc} = F_f N_c$. For the first test problem, this works out to be 26000 and for the second problem, 20900.

Proceeding with the accounting of storage, we have:

1. N_c integers for the *mesh2subset* list for each material or a total of $N_c N_m$ integers

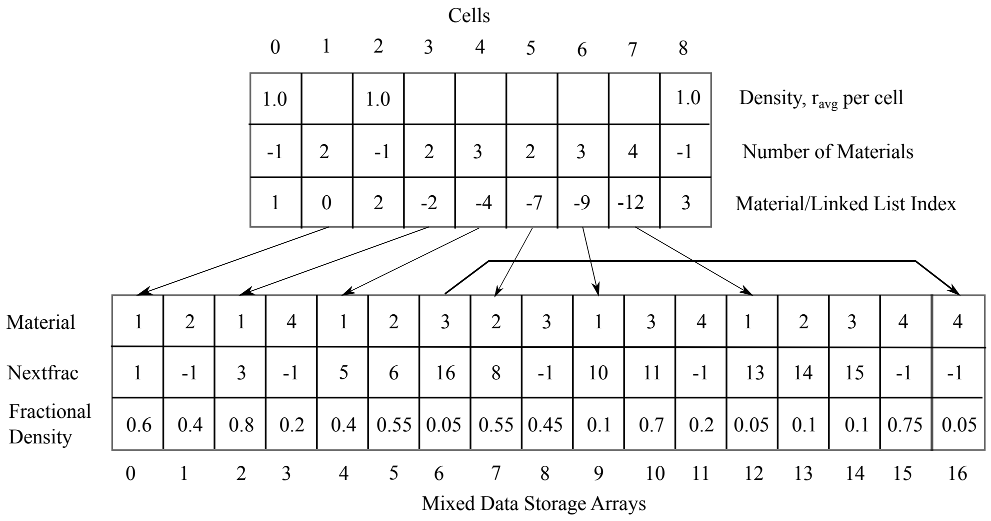


Figure 7: Adding a material to the mixed material data structure

Algorithm 12 Reordering mixed material arrays

```
$matindex \leftarrow 0$
**for all** cells, $C$, in the mesh, up to $N_c$ **do**
  **if** $nmats[C] > 1$ **then**
    $matid[C] \leftarrow -matindex$                # Setting offset into mixed data array
  **end if**
  **for all** material IDs, $m$, in the cell, up to $nmats[C]$ **do**
    $\rho_{new}[matindex] \leftarrow r[C][m]$
    $matindex + 1$
  **end for**
**end for**
```

2. $\bar{N}_{mc}$ integers for the *subset2mesh* list for each material or a total of $\bar{N}_{mc}N_m = F_f N_c N_m$ integers
3. N_c integers for the *nmats* array
4. $N_{mc}^{max} N_c$ integers for the *matids* array

or a total of $(N_m + F_f N_m + 1 + N_{mc}^{max})N_c$ integers or $4(N_m + F_f N_m + 1 + N_{mc}^{max})N_c$ bytes.

To store the N_{mv} material-based scalar fields, we need:

1. $\bar{N}_{mc}$ doubles for each field for each material or a total of $F_f N_c N_m N_{mv}$ doubles.
2. $N_m N_{mv}$ pointers for pointing into the field arrays (but this can be ignored for large meshes as it does not depend on N_c)

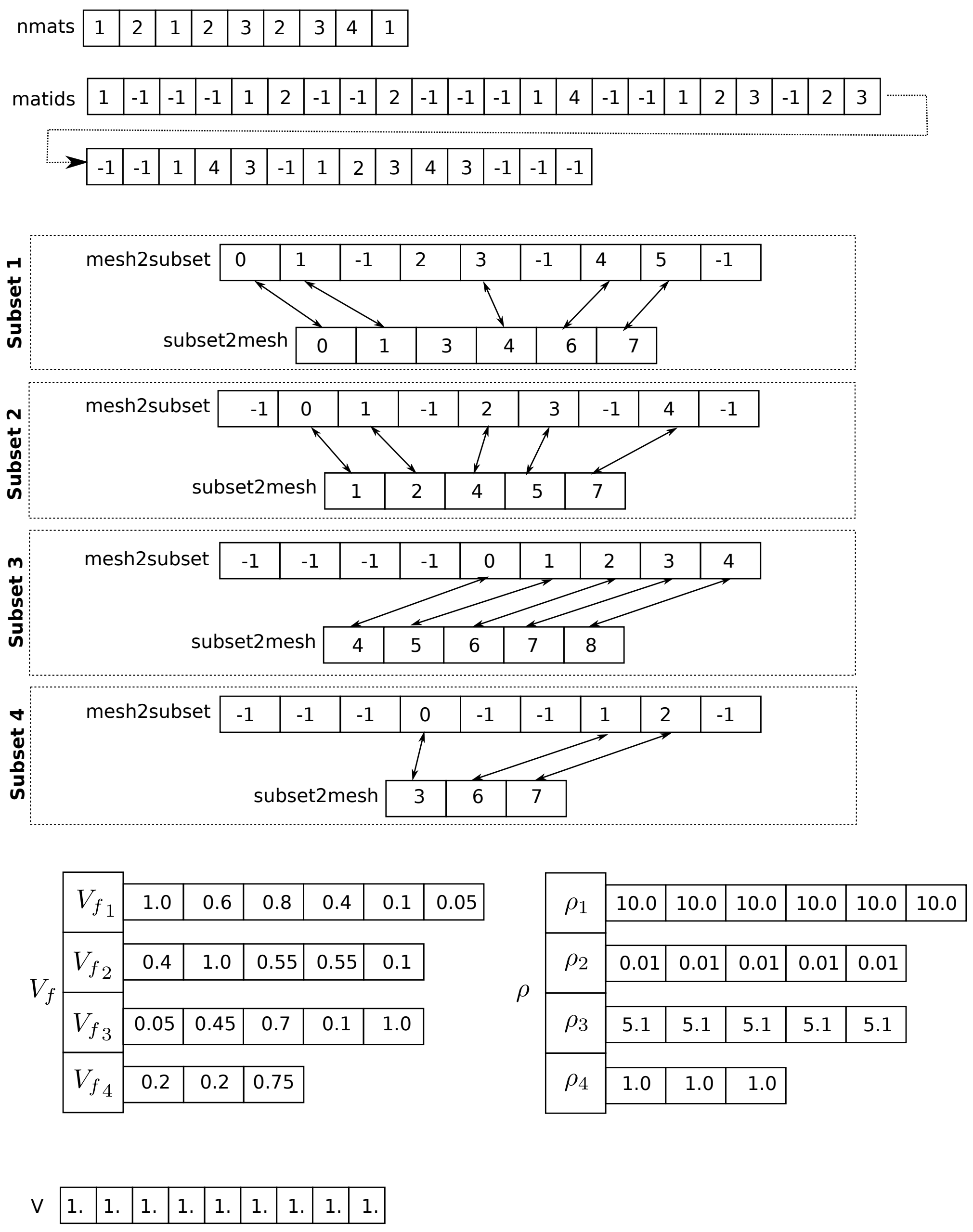


Figure 8: The material-centric compact data representation of material specific fields ρ and V_f for the example shown in Figure 1.

Finally, to store N_{cv} cell-based scalar fields (such as volumes of cells), we need $N_{cv}N_c$ doubles. This gives us a total of $(F_f N_m N_{mv} + N_{cv})N_c$ doubles or $8(F_f N_{mv} + N_{cv})N_c$ bytes.

Thus, the total storage for the material-centric compact representation, can be written as:

$$Storage_{cm} = [4(N_m + F_f N_m + 1 + N_{mc}^{max}) + 8(F_f N_m N_{mv} + N_{cv})]N_c \qquad (13)$$

One can see that the first part of the expression is the cost of storing the subset representation and the second the fields. Therefore, the relative cost of storing the subset representation decreases with the number of fields one needs to represent. For the first test problem, the cost of storing the subsets dominates (80%) as we are storing only 4 material-based fields. However, in a representative problem with say 100 material based fields and 10 cell-based fields this drops to 16% of the cost.

Several optimizations are possible to reduce some of the fixed costs for storing subsets. If we chose not to store a fixed number of entries for each cell in the *matids* it would make the length of the *matids* array $N_c/2$ (see estimation of M_L array in the cell-centric compact storage discussion). This optimization will drop the storage by about 15 MB for the first test problem. However, it makes it much more expensive to introduce materials into cells because all the elements in the array after the cell's entries will have to be moved. An easier optimization is to use character arrays (1 byte per entry) for *nmats* and *matids* arrays which will support a maximum material ID of 127 and also save 15 MB in storage.

3.2.1 First Computational Loop – Average Cell Density

With the material-centric compact data structure, the average cell density calculation shown in Algorithm 13 is similar to the one for the material-centric full matrix storage Algorithm 5 except that we loop over subset cell indices c, not mesh cell indices C, in the main loop. Consequently, we have to translate the subset cell index c to mesh cell index C for accessing the correct cell-based average density.

The *cell-centric* algorithm for doing the same operation is shown below in Algorithm 14. Note that in this algorithm we have to translate the mesh cell index, C, to a local index, c, in the particular subset using the reverse map, *mesh2subset*.

It is useful to point out how close the algorithms using the compact material-centric data structure are to the full matrix algorithm. In fact, in a feature-rich language like C++, the algorithms can be made to look just like the full matrix algorithms with the conversion from mesh index to subset index or vice versa being done under the hood.

Performance analysis

Since there are no conditional statements in the above two algorithms for computing average density, the compact storage scheme will result in as many accesses of material specific variables as there are values ($\bar{N_{mc}}N_m = F_f N_c N_m$) whether we loop over materials and cells in each material or cells and materials in each cell.

Considering the material-dominant algorithm for computing average densities, we see that it performs $(2F_f N_m + 1)N_c$ flops. We also see that it has $N_c + F_f N_c N_m$ integer loads, N_c pointer loads, $3F_f N_c N_m + 2N_c$ double loads and $F_f N_c N_m + 2N_c$ double stores (Cost of integer load/store will be half of that of a double). The cache penalty in this algorithm

Algorithm 13 *Material-dominant* algorithm to compute average density of cells using material-centric compact storage

```
for all cells, $C$, in the mesh, up to $N_c$ do
    $\rho_{ave}[C] \leftarrow 0.0$                          # $N_c$ stores
end for
for all material IDs, $m$, in the problem, up to $N_m$ do
    $ncmat \leftarrow ncellsmat[m]$                         # $N_c$ loads
    $subset \leftarrow subset2mesh[m]$                      # $N_c$ pointer loads
    for all cells, $c$, in the material upto $ncmat$ do
        $C \leftarrow subset[c]$                            # $F_f N_c N_m$ loads
        $\rho_{ave}[C] \leftarrow \rho_{ave}[C] + \rho[m][c] * V_f[m][c]$
                                                            # $3F_f N_c N_m$ loads ($\rho_{ave}$, $\rho$, $V_f$)
                                                            # $F_f N_c N_m$ stores ($\rho_{ave}$)
                                                            # $2F_f N_c N_m$ flops ($+, *$)
                                                            # Likely cache miss for $\rho_{ave}$
                                                            # Multiply by $C_p = 8$
    end for
end for
for all cells, $C$, in the mesh, up to $N_c$ do
    $\rho_{ave}[C] \leftarrow \rho_{ave}[C]/V[C]$           # $2N_c$ loads ($\rho_{ave}$, $V$)
                                                            # $N_c$ stores ($\rho_{ave}$)
                                                            # $N_c$ flops (/)
end for
```

Algorithm 14 *Cell-dominant* algorithm to compute average density of cells using the material-centric compact storage scheme

```
for all cells, $C$, in the mesh, up to $N_c$ do
    $ave \leftarrow 0.0$
    for all material indices, $i$, in the cell, up to $nmats(C)$ do
        $m \leftarrow matids(i)$                            # $F_f N_c N_m$ integer loads ($matids$)
                                                            # Integer loads, multiply by 0.5
        $c \leftarrow mesh2subset(m, C)$                    # $F_f N_c N_m$ integer loads ($mesh2subset$)
                                                            # Likely cache miss, multiply by $C_p = 8$
                                                            # Integer loads, multiply by 0.5
        $ave \leftarrow ave + \rho[m][c] * V_f[m][c]$       # $2F_f N_c N_m$ loads ($\rho$, $V_f$)
                                                            # Likely cache miss, multiply by $C_p = 8$
                                                            # $2F_f N_c N_m$ flops ($*, +$)
    end for
    $\rho_{ave}[C] \leftarrow ave/V[C]$                     # $N_c$ loads ($V$)
                                                            # $N_c$ stores ($\rho_{ave}$)
                                                            # $N_c$ flops
end for
```

comes from accessing a mesh cell-based variable (ρ_{ave}) in a loop that is stepping through cells of a material subset. Therefore, performance equations for the material-dominant loop for average density calculations using this storage scheme are:

$$\begin{aligned} memops &= (3.5F_f N_m + 5.5)N_c + 8F_f N_m N_c \\ flops &= (2F_f N_m + 1)N_c \\ PM &= [(3.5F_f N_m + 5.5)N_c + 8F_f N_m N_c] * 8/Stream \end{aligned}$$

The memory access pattern in the material-dominant algorithm with this storage scheme is quite favorable because only 4 arrays, $subset2mesh$, ρ, V_f and ρ_{ave} are needed for the inner loop and except ρ_{ave}, all the others are referenced by subset cell ID, c which is stored contiguously.

In the cell-dominant algorithm for computing average densities using this storage scheme, the number of flops will be $(2F_f N_m + 1)N_c$. Accounting for memory ops, we will have $2F_f N_c N_m$ integer loads ($matids$, $mesh2subset$) and $(2F_f N_m + 1)N_c$ double loads (ρ, V_f). There will be 1 double store (ρ_{ave}) in the outer loop for a total of N_c stores. Cache penalties are factored because we access material arrays $mesh2subset$, ρ and V_f arrays while looping through mesh cells.

$$\begin{aligned} memops &= 2N_c + 20.5F_f N_c N_m \\ flops &= (2F_f N_m + 1)N_c \\ PM &= [2N_c + 20.5F_f N_m N_c] * 8/Stream \end{aligned}$$

The memory accesses needed for the cell-dominant average density algorithm varies in each iteration of the inner loop, depending on $nmats(C)$. If the cell is a single material cell, then one has to access 6 arrays, ρ_{ave}, $nmats$, $matids$, $mesh2subset$, ρ and V_f (as mentioned before, this can be cut to 5 by bundling $nmats$ and $matids$). With each extra material, however, this increases by 3 more arrays ($mesh2subset$, ρ and V_f for the extra material). Thus this data structure and this loop combination are less advantageous to use than the others in multi-material cells. *However, it should be noted that a majority of the cells in a large mesh will be pure cells and therefore, this disadvantage is minor and its effect rapidly diminishes with increasing mesh resolution.*

3.2.2 Second Computational Loop – Pressure for each cell and each material

The second computational loop (which is naturally *material-dominant*) can be written like this:

Performance analysis

In the computational loop to update material properties using this loop, we have 3 double loads (r, t, V_f) and 1 integer load ($subset2mesh$) in the inner loop which is $3.5F_f N_c N_m$ 8 byte memops. There is 1 double load in the outer loop (n) which contributes a paltry N_m 8 byte memops. There is also 1 double store in the inner loop of the variable p for total of $F_f N_c N_m$ stores. So the total number of 8 byte memops is $4.5F_f N_c N_m + N_m$. The number of flops is 4 per inner loop giving a total of $4F_f N_c N_m$. Thus the number of loads to flops is nearly 1:1 which is very good.

Algorithm 15 *Material-dominant* algorithm to update material state using the material-centric compact storage scheme

```
for all material IDs, $m$, in the problem, up to $N_m$ do
  $n_m \leftarrow n(m)$                                # $N_m$ loads
  for all cells, $c$, in the material do
    $C \leftarrow subset2mesh[c]$                      # $F_f N_c N_m$ integer loads ($subset2mesh$)
                                                       # Integer loads, multiply by 0.5
    $p[m][c] \leftarrow (n_m * \rho[m][c] * t[m][c])/V_f[c]$
                                                       # $3F_f N_c N_m$ loads ($\rho$, $t$, $V_f$, $V$)
                                                       # $F_f N_c N_m$ stores ($p$)
                                                       # $4F_f N_c N_m$ flops ($*$, $/$)
  end for
end for
```

Therefore, the performance equations for these algorithms using the material-centric compact storage are:

$$\begin{aligned}
memops &= (4.5F_fN_c + 1)Nm\\
flops &= 4F_fN_cN_m\\
PM &= (4.5F_fN_c + 1)N_m * 8/Stream
\end{aligned}$$

Each loop in the material update algorithm requires access to 6 arrays n, $subset2mesh$, V, r, t, V_f and p. Except for V which is indexed by mesh cell ID C, all arrays are accessed in a contiguous fashion making this an efficient access pattern.

3.2.3 Third Computational Loop - Average density of each material over neighborhood of each cell

The third computational loop computing the average density of each material in the neighborhood of each cell is shown in Algorithm 16.

Performance Analysis

We can then write the following performance model as before:

$$\begin{aligned}
memops &= N_c(3 + 12.5\bar{N}_n) + N_cN_m + F_fN_cN_m(5.5 + 4\bar{N}_n(1 + 2L_f)),\\
flops &= 11F_fL_fN_cN_m\bar{N}_n + F_fN_cN_m\\
PM &= (N_c(3 + 12.5\bar{N}_n) + N_cN_m + F_fN_cN_m(5.5 + 4\bar{N}_n(1 + 2L_f))) * 8/\text{Stream}
\end{aligned} \tag{14}$$

The same process using a material-dominant logic is shown below in Algorithm 17). The only difference between this algorithm and the cell-centric one (Algorithm 16) is that the outer loop is on materials.

Algorithm 16 *Cell-dominant* algorithm to compute average density of each material over a neighborhood of each cell

```
1:  for all cells, $C$, in the mesh, up to $N_c$ do
2:      Initialize $\bar{\rho}$ to 0                      # $N_c N_m$ stores from Alg. 4
3:      $\bar{\mathbf{x}}_c \leftarrow \bar{\mathbf{x}}[C]$                        # $3N_c$ loads ($\bar{x}[C]$). Assume 3D
4:      $\{c_{nbrs}\} \leftarrow nbrs[C]$                 # $N_c \bar{N}_n$ integer loads
5:                                             # Integer loads, multiply by 0.5
6:      Initialize $d_{sqr}$ w.r.t. neighbors        # $3N_c \bar{N}_n$ loads ($\bar{\mathbf{x}}$) from Alg. 4
7:                                             # Partial reuse, multiply by $C_p = 4$
8:                                             # $9N_c \bar{N}_n$ flops (-,*,+)
9:      for all material IDs, $k$, in the cell, up to $nmats(C)$ do
10:         $m \leftarrow matids(k)$                     # $F_f N_c N_m$ loads ($matids$)
11:                                            # Integer loads, multiply by 0.5
12:         $c \leftarrow mesh2subset(m, C)$             # $F_f N_c N_m$ loads ($mesh2subset$)
13:                                            # Integer loads, multiply by 0.5
14:                                            # Likely cache miss, multiply by $C_p = 8$
15:         $\rho_{sum} \leftarrow 0$
16:         $N_n \leftarrow 0$                           # Local var. $N_n$, not global var. $\bar{N}_n$
17:         for neighbor cell index $i \leftarrow 0, \bar{N}_n - 1$ do
18:             $C_i \leftarrow c_{nbrs}[i]$
19:             $c_i \leftarrow mesh2subset(m, C_i)$     # $F_f N_c N_m \bar{N}_n$ loads ($mesh2subset$)
20:                                            # Likely cache miss, multiply by $C_p = 8$
21:                                            # Integer load, multiply by 0.5
22:             if $c_i >= 0$ then
23:                 $\rho_{sum} \leftarrow \rho_{sum} + \rho[c_i][m]/d_{sqr}[i]$
24:                                            # $F_f N_c N_m L_f \bar{N}_n$ loads ($\rho$)
25:                                            # Likely cache miss, multiply by $C_p = 8$
26:                                            # $2F_f N_c N_m L_f \bar{N}_n$ flops (/,+)
27:                 $N_n \leftarrow N_n + 1$                 # $F_f N_c N_m L_f \bar{N}_n$ flops (+)
28:             end if
29:         end for
30:         $\bar{\rho}[C][m] \leftarrow \rho_{sum}/N_n$            # $F_f N_c N_m$ stores ($\bar{\rho}$)
31:                                            # $F_f N_c N_m$ flops (/)
32:     end for
33: end for
```

Algorithm 17 *Material-dominant* algorithm to compute average density of each material over a neighborhood of each cell using a material-centric compact representation

```
for all material IDs, $m$, in the problem, up to $N_m$ do
    for all cells, $C$, in the mesh, up to $N_c$ do
        $\bar{\rho}[C][m] \leftarrow 0.0$                        # $N_c N_m$ stores
    end for
    for all cells, $c$, in the material do
        $C = subset2mesh(m, c)$                     # $F_f N_c N_m$ loads ($subset2mesh$)
                                                    # Integer loads, multiply by 0.5
        $x_c \leftarrow \bar{x}[C]$                           # $3 F_f N_c N_m$ loads ($\bar{x}$)
                                                    # Assume problem is 3D
                                                    # Likely cache miss, multiply by $C_p = 8$
        $\rho_{sum} \leftarrow 0.0$
        for neighbor cell index $i = 0, \bar{N}_n - 1$ do
            $C_i = nbrs[C][i]$                      # $F_f N_c N_m \bar{N}_n$ integer loads ($nbrs$)
                                                    # Likely cache miss, multiply by $C_p = 8$
                                                    # Integer loads, multiply by 0.5
            $c_i = mesh2subset(m, C_i)$             # $F_f N_c N_m \bar{N}_n$ integer loads ($mesh2subset$)
                                                    # Partial reuse, multiply by $C_p = 4$
                                                    # Integer loads, multiply by 0.5
            if $c_i >= 0$ then
                $d_{sqr} = 0.0$
                for $j = 0, 2$ do
                    $d_{sqr} = d_{sqr} + (x_c[j] - \bar{x}[C_i][j])^2$
                                                    # $3 F_f N_c N_m L_f \bar{N}_n$ loads ($\bar{x}$)
                                                    # Partial reuse, multiply by $C_p = 4$
                                                    # $9 F_f N_c N_m L_f \bar{N}_n$ flops (-,*,+)
                end for
                $\rho_{sum} = \rho_{sum} + \rho[m][c_i]/d_{sqr}$      # $F_f N_c N_m L_f \bar{N}_n$ loads ($\rho$)
                                                    # $2 F_f N_c N_m L_f \bar{N}_n$ flops (/,+)
            end if
        end for
        $\bar{\rho}[C][m] = \rho_{sum}/N_n$                   # $F_f N_c N_m$ stores ($\bar{\rho}$)
                                                    # $F_f N_c N_m$ flops (/)
    end for
end for
```

Performance Analysis

The performance model for this case is similar to that of the cell-centric case except that the loads of $\bar{x}[C]$ and $nbrs[C]$ have moved further inside the loops multiplying their costs by N_m. Thus the performance model from the cell-centric case can be modified to appear as shown below:

$$\begin{aligned} memops &= N_c N_m + F_f N_c N_m (25.5 + \bar{N}_n(6 + 13L_f))), \\ flops &= 11 F_f L_f N_c N_m \bar{N}_n + F_f N_c N_m \\ PM &= (N_c N_m + F_f N_c N_m (25.5 + \bar{N}_n(6 + 13L_f))) * 8/\text{Stream} \end{aligned} \tag{15}$$

3.2.4 Adding to the Material-Centric Compact Data Structures

Modification of the material-centric compact data structures to accommodate a new material appearing in a cell is quite trivial. The new cell is simply appended to the *subset2mesh* array in the particular material subset and the corresponding field values appended to the appropriate field arrays. At the same time the material ID is added to the *matids* array at a location corresponding to the cell and the *nmats* array value is incremented. One could reorganize these arrays from time to time so that values of cells that are spatially close are also close in the array and result in fewer cache misses.

The algorithms become only a little more complicated when a material disappears from a cell. Then the entry in the *subset2mesh* array must be nullified (set to -1) and the algorithms modified to check for this. This doesn't significantly impact the actual performance since the branch prediction will detect that the code inside the 'if' statement will be executed more often than not. Alternatively, we can allocate an extra element in each cell-based variable array and set the $subset2mesh$ array value to $N_c + 1$. Then, we will write into a dummy location without the penalty of an 'if' statement when processing a cell which has been 'removed' from a material subset. Periodically, but infrequently, the holes in the subset can be squeezed out restoring efficiency to the process.

4 Conversions

4.1 Converting from Cell-Centric to Material-Centric

We can convert from *cell-centric* to *material-centric* compact data structures for some arrays so that the data access pattern is more contiguous. This process is made easier if the number of cells for each material tracked. This can be done at the same time that the material adding operation is being done. An atomic operation will also be needed for this since multiple threads could add at the same time. By keeping track of $ncells[m]$, we can then allocate the arrays without worrying about getting them too large or small. If the number of cells is not tracked, a prepass through the data to get the sizes is probably best.

4.2 Converting from Material-Centric to Cell-Centric

The conversion of the material-centric compact structure to the cell-centric compact structure requires knowing the size of the mixed material arrays. This size is calculated as shown in Algorithm 19

Algorithm 18 Converting from Cell-Centric to Material-Centric Compact Structure

```
for all material IDs, $m$, in the mesh, up to $N_m$ do
    $alloc(\rho_n ew[m](ncells[m] * sizeof(double)))$
    $nextcell[m] \leftarrow 0$
end for
for all cells, $C$, in the mesh, up to $N_c$ do
    for all material IDs, $m$, in the cell, from mstart and for $nmats[C]$ do
        $mm \leftarrow matid[m]$
        $\rho_{new}[mm][nextcell[C]] \leftarrow r[m]$
        $nextcell[m] \leftarrow nextcell[m] + 1$
    end for
end for
```

Algorithm 19 Counting Mix Cells for Converting from Material-Centric to Cell-Centric Compact Structure

```
$ix \leftarrow 0$
for all cells, $ic$, in the mesh, up to $N_c$ do
    if $nmatscell[ic] > 1$ then
        $mxsize \leftarrow mxsize + nmatscell[ic]$
    end if
end for
```

Now all the mix cell arrays can be allocated and the rest of the conversion process can go forward as shown in Algorithm 20. The material-centric arrays are on the right-hand side and the cell-based arrays are distinguished by variables starting with a capital C and are on the left-hand side of the statements.

The actual performance of the conversion is about 60 msec for Algorithms 19 and 20. The material data is accessed in an irregular pattern that will have cache misses, but the cell-centric data is written out in contiguous order which should give good performance.

Algorithm 20 Converting from Material-Centric to Cell-Centric Compact Structure

```
$ix \leftarrow 0$
for all cells, $ic$, in the mesh, up to $N_c$ do
  $nmats \leftarrow nmatscell[ic]$
  if $nmats == 1$ then
    $m \leftarrow matids[4 * ic]$
    $c \leftarrow mesh2subset[m][ic]$
    $Cimaterial[ic] \leftarrow m$
  else
    for all materials, $im$, in each cell, up to $nmats$ do
      $m \leftarrow matids[4 * ic + im]$
      $c \leftarrow mesh2subset[m][ic]$
      $Cimaterialfrac[ix] \leftarrow m$
      $Cnextfrac[ix] \leftarrow ix + 1$
      $Cfrac2cell[ix] \leftarrow ic$
      $CVolfrac[ix] \leftarrow Volfrac[m][c]$
      $CDensityfrac[ix] \leftarrow Densityfrac[m][c]$
      $CTemperaturefrac[ix] \leftarrow Temperaturefrac[m][c]$
      $CPressurefrac[ix] \leftarrow Pressurefrac[m][c]$
      $ix \leftarrow ix + 1$
    end for
    $Cnextfrac[ix - 1] \leftarrow -1$
  end if
end for
```

5 Results and Discussion

We have written a test code which evaluates the performance of the data structures for the two computational loops as applied to the two test problems presented in the discussion. The results of the tests are discussed here.

First we present the growth of the storage with the number of material variables for the various storage schemes. The compact storage schemes reduce the memory usage over the full storage scheme by over 95% as shown in Table 1. This table also shows how the data storage costs increase with numbers of field variables where there is not much difference between the two compact schemes at higher numbers of material variables. This alone is enough to adopt the compact schemes, but with a bandwidth-limited algorithm, we would also expect better performance. This is discussed next.

Tables 2 and 3 show the performance of the various data structures for average cell density and material pressure loops with fractional volumes initialized from material shapes. Tables 4 and 5 show the performance data for randomized fractional volume evaluation.

From the above data it is clear that the compact data structures present a real advantage in memory footprint, memory accesses and predicted computational time over the full data structures.

Between the compact data structures, the cell-dominant compact storage affords a lower

Scheme ↓	Storage MB →		
	4 mat. vars	10 mat. vars	100 mat. vars
Single Material	32	80	800
Full Matrix	1663	4160	41600
Compact, Cell	62	134	1214
Compact, Material	268	332	1297

Table 1: Data Storage increase in MB for increasing numbers of material variables (data for cell variables is not counted here)

		Memops		Flops	Memops	Estimated	Actual	Error
Description	Alg.	M	MB	M	to Flops	msecs	msecs	%
Single Material		2	16	2	1:1	1.20	1.03	16.3
Full, Cell	1	102	816	100	1:1	61.0	53.4	14.3
Full, Cell, with if	2	53.0	424.4	3.1	17.1:1	66.5	67.9	-2.0
Full, Mat.	4	153	1224	101	1.5:1	91.5	79.9	14.6
Full, Mat., with if	5	55.1	440.8	3.1	17.8:1	42.9	35.8	19.9
Compact, Cell	7	2.9	17.2	1.2	2.4:1	1.43	1.08	32.3
Compact, Cell, with nmats	8	2.6	16.0	1.2	2.2:1	1.34	1.24	8.5
Compact, Cell, with rho_ave	9	4.5	30.0	1.2	3.8:1	2.39	1.91	25.2
Compact, Cell, divide by V	10†	5.3	36.4	2	2.7:1	2.87	2.51	14.2
Compact, Mat, Mat-dominant	14†	10.7	81.5	3.1	3.46:1	6.09	5.98	1.84
Compact, Mat, Cell-dominant	15†	13.3	81.3	3.1	4.29:1	6.08	6.04	0.6

Table 2: Performance models for average cell density calculation with fractional volumes initialized from geometric shapes (†closest in assumptions).

		Memops		Flops	Memops	Estimated	Actual	Error
Description	Alg.	M	MB	M	to Flops	msecs	msecs	%
Single Material		4	32	3	1.3:1	2.39	2.62	-8.8
Full, Cell	3	103.1	825.2	3.1	32.8:1	96.5	97.6	-1.1
Full, Material	6	102.1	816.8	3.1	32.4:1	71.0	63.7	11.5
Compact, Cell	11	8.0	56.0	3.9	2.1:1	4.34	2.90	49.5
Compact, Mat	16	4.2	33.6	3.1	1.33:1	2.51	2.80	-10.2

Table 3: Performance models for material pressure calculation with fractional volumes initialized from geometric shapes.

		Memops		Flops	Memops	Estimated	Actual	Error
Description	Alg.	M	MB	M	to Flops	msecs	msecs	%
Single Material		2	16	2	1:1	1.20	1.00	19.7
Full, Cell	1	102	816	100	1:1	61.0	53.4	14.2
Full, Cell, with if	2	53.3	426.4	3.6	14.8:1	93.5	96.4	-3.0
Full, Mat.	4	153	1224	101	1.5:1	91.5	79.1	15.7
Full, Mat., with if	5	55.6	444.8	3.6	15.4:1	94.9	84.0	13.0
Compact, Cell	7	2.9	17.2	1.2	2.4:1	2.77	3.27	-15.5
Compact, Cell, with nmats	8	2.6	16.0	1.2	2.2:1	2.68	3.29	-18.5
Compact, Cell, with rho_ave	9	4.5	30.0	1.2	3.75:1	3.72	3.56	4.5
Compact, Cell, divide by V	10†	5.3	36.4	2	2.65:1	4.20	3.65	15.1
Compact, Mat, Mat-dominant	14†	19.6	151.6	3.6	5.44:1	11.33	11.29	0.4
Compact, Mat, Cell-dominant	15†	45.9	274.8	3.6	12.75:1	20.54	23.08	-11.0

Table 4: Performance Models for Average Cell Density Calculation with random initialization of volume fractions (†closest in assumptions).

		Memops		Flops	Memops	Estimated	Actual	Error
Description	Alg.	M	MB	M	to Flops	msecs	msecs	%
Single Material		4	32	3	1.3:1	2.4	2.57	-6.8
Full, Cell	3	103.9	831.2	3.9	26.6:1	123.8	135.6	-8.7
Full, Material	6	102.6	820.8	3.9	26.3:1	123.0	116.2	5.8
Compact, Cell	11	8.0	61.0	3.9	2.1:1	5.7	4.59	23.4
Compact, Mat	16	6.03	48.2	5.4	1.12:1	3.11	3.59	-13.4

Table 5: Performance Models for Material Pressure Calculation with random initialization of volume fractions.

storage cost and faster compute times particularly for updating existing state variable vectors (See Algorithm 8 in Table 4) in cell-dominant loops. On the other hand, the material-centric compact storage scheme does very well for a material-dominant loop and this advantage would increase if the simulation had very disparate material models that were much more compute and memory intensive than an ideal gas law.

Between the two test cases, the cell-dominant data structures were not greatly affected by the random nature of the material distribution but the material-centric data structures were. The reason for the poorer performance of the material-centric data structures in the randomized case is that contiguous cells of a material are quite unlikely to be contiguous in the mesh for this type of distribution. As expected this storage scheme performed better when the material distribution was closer to reality.

The analysis of the compact data structures perfectly illustrates the tension between programmability and efficiency of data access and ease of coding in complex codes. In principle, both data structures offer similar memops to flops ratios. However, while the compact material-centric data structure offers a much more intuitive and simple programming model for the application developer, the compact cell-dominant data structure offers superior cache performance and therefore, better execution times.

So we can see that there is no perfect data structure and that the choice of which should be the centric variable (cells or materials) is dependent on the most common data access demands of a code. A cell-based Eulerian code may need outer loops by cells and a Lagrangian-based method may benefit some from a material-based outer loop, though many loops in that approach are cell-dominant. But in any code, there will be cases where data will likely have to be accessed in a less than ideal manner. The goal is to minimize where this occurs. Alternately, one could either switch between the two schemes as needed or even maintain both data structures in the code and seamlessly access one or the other by the use of modern programming mechanisms. The cost of such a transformation will be the focus of our ongoing work.

6 Acknowledgments

This work was performed under the auspices of the National Nuclear Security Administration of the US Department of Energy at Los Alamos National Laboratory under Contract No. DE-AC52-06NA25396 and supported by the DOE Advanced Simulation and Computing (ASC) program. Report number LA-UR-16-23889.

A Appendix

Algorithm 21 *Material-centric* algorithm to compute average density of cells using full storage

```
 1: for all cells, $C$, in the mesh, up to $N_c$ do
 2:     $\rho_{ave}[C] \leftarrow 0.0$                          # $N_c$ stores ($\rho_{ave}$)
 3: end for
 4: for all material IDs, $m$, in the problem, up to $N_m$ do
 5:     for all cells, $C$, in the mesh, up to $N_c$ do
 6:         if $V_f[m][C] > 0.0$ then                # $N_c N_m$ loads ($V_f$)
 7:                                                  # 16 cycle branch $+112$ cache miss
 8:             $\rho_{ave}[C] \leftarrow$                  # $F_f N_c N_m$ stores ($\rho_{ave}$)
 9:                 $\rho_{ave}[C] + \rho[m][C] * V_f[m][C]$     # $F_f N_c N_m$ loads ($\rho$)
10:                                                  # $2F_f N_c N_m$ flops ($+, *$)
11:         end if
12:     end for
13: end for
14: for all cells, $C$, in the mesh, up to $N_c$ do
15:     $\rho_{ave}[C] \leftarrow \rho_{ave}[C]/V[C]$             # $2N_c$ loads/stores ($\rho_{ave}, V$)
16:                                                  # $N_c$ flops (/)
17: end for
```

The number of loads will be N_c for initialization of ρ_{ave} plus $N_m N_c$ for the fractional volume test. In the main loop there are $F_f N_c N_m$ loads of ρ and $F_f NcNm$ stores of ρ_{ave}. There will be $2F_f N_c N_m$ flops in the main loop. The final loop has $2N_c$ loads and stores and an additional N_c flops. The performance equations become:

$$\begin{aligned} memops &= N_c + N_c N_m + 2F_f N_c N_m + 2N_c = N_c(N_m + 2F_f N_m + 3), \\ flops &= 2F_f N_c N_m + N_c \\ PM &= N_c(N_m + 2F_f N_m + 3) * 8/\text{Stream} + B_p F_f N_c N_m \end{aligned} \tag{16}$$

This will be 55.6 M memops or 444.8 MB. The flops will be 3.6 Mflops. The memops to flop ratio will be 15.4. Our estimated performance is 94.9 msecs. The measured performance is 89.4 msecs.

The algorithm would be the following:

$$\begin{aligned} memops &= N_c + M_f N_c + 2M_L + 2M_f N_c = N_c(1 + 3M_f) + 2M_L, \\ membytes &= N_c * 4 + M_f N_c * 4 + 2M_L * 8 + 2M_f N_c * 8 = (4 + 4 * M_f + 2M_f * 8)N_c + 16M_L, \\ flops &= 2M_L + M_f N_c \\ PM &= membytes/\text{Stream} + L_p M_f N_c \end{aligned} \tag{17}$$

The memops are 2.6 M memops and 16.0 MB. The flops are 1.2 M flops for a 2.2 memops to flops ratio. The expected performance is 2.68 msecs and the actual is 3.13 msecs.

Algorithm 22 *Cell-centric* algorithm to compute average density of cells using compact storage

```
for all cells, $C$, in the mesh, up to $N_c$ do
    $ave \leftarrow 0.0$
    $ix \leftarrow imaterial[C]$                          # $N_c$ loads ($imaterial$)
    if $ix <= 0$ then
        for $ix \leftarrow -ix$, for $nmats[C]$ do          # $M_f N_c$ loads ($nmats$)
                                                          # $M_f$ 16 cycle branch miss
            $ave \leftarrow ave + \rho[ix] * V_f[ix]$            # $2M_L$ loads ($\rho, V_f$)
                                                          # $M_L$ flops ($+, *$)
        end for
        $\rho[C] \leftarrow ave/V[C]$                        # $M_f N_c$ stores $\rho_{ave}$
                                                          # $M_f N_c$ loads ($V$)
                                                          # $M_f N_c$ flops (/)
    end if
end for
```

Algorithm 23 *Cell-centric* algorithm to compute average density of cells using compact storage

```
for all cells, $C$, in the mesh, up to $N_c$ do
    $ave \leftarrow 0.0$
    $ix \leftarrow imaterial[C]$                          # $N_c$ loads ($imaterial$)
    if $ix <= 0$ then
        for $ix \leftarrow -ix$, until $nextfrac[ix] < 0$ do                              #
    $M_L$ loads ($nextfrac$)
            $ave \leftarrow ave + \rho[ix] * V_f[ix]$            # $2M_L$ loads ($\rho, V_f$)
                                                          # $M_L$ flops ($+, *$)
        end for
        $\rho_{ave}[C] \leftarrow ave/V[C]$                    # $M_f N_c$ stores $\rho_{ave}$
                                                          # $M_f N_c$ loads ($V$)
                                                          # $M_f N_c$ flops (/)
    else
        $\rho_{ave}[C] = \rho[C]$                          # $P_f N_c$ stores ($\rho_{ave}$)
                                                          # $P_f N_c$ loads ($\rho$)
    end if
end for
```

We look at performance if a new ρ_{ave} array needed to be calculated.

$$\begin{aligned} memops &= N_c + M_L + 2M_L + 2M_f N_c + 2P_f N_c = 3N_c + 3M_L, \\ membytes &= N_c * 4 + M_L * 4 + 2M_L * 8 + 2M_f N_c * 8 + 2P_f N_c * 8 = 20N_c + 20M_L, \\ flops &= 2M_L + M_f Nc \\ PM &= membytes/\text{Stream} + L_p M_f N_c \end{aligned} \tag{18}$$

The memops are 4.5 M memops and 30 MB. The flops are 1.2 M flops for a 3.75 memops to flops ratio. The expected performance is 3.72 msecs and the actual is 3.49 msecs.

We look at another where the pure cells are divided by volume just to see what the performance numbers come out to:

Algorithm 24 *Cell-centric* algorithm to compute average density of cells using compact storage

```
1:  for all cells, $C$, in the mesh, up to $N_c$ do
2:      $ave \leftarrow 0.0$
3:      $ix \leftarrow imaterial[C]$                      # $N_c$ loads ($imaterial$)
4:      if $ix <= 0$ then
5:          for $ix \leftarrow -ix$, until $nextfrac[ix] < 0$ do          #
    $M_L$ loads ($nextfrac$)
6:              $ave \leftarrow ave + \rho[ix] * V_f[ix]$       # $2M_L$ loads ($\rho, V_f$)
7:                                                    # $M_L$ flops ($+, *$)
8:          end for
9:          $\rho_{ave}[C] \leftarrow ave/V[C]$                 # $M_f N_c$ stores $\rho_{ave}$
10:                                                   # $M_f N_c$ loads ($V$)
11:                                                   # $M_f N_c$ flops (/)
12:     else
13:         $\rho_{ave}[C] = \rho[C]/V$                      # $P_f N_c$ stores ($\rho_{ave}$)
14:                                                   # $2P_f N_c$ loads ($\rho, V$)
15:                                                   # $P_f N_c$ flops (/)
16:     end if
17: end for
```

$$\begin{aligned} memops &= N_c + M_L + 2M_L + 2M_f N_c + 3P_f N_c = (3 + P_f)N_c + 3M_L, \\ membytes &= N_c * 4 + M_L * 4 + 2M_L * 8 + 2M_f N_c * 8 + 3P_f N_c * 8 = (20 + P_f * 8)N_c + 20M_L, \\ flops &= 2M_L + Nc \\ PM &= membytes/\text{Stream} + L_p M_f N_c \end{aligned} \tag{19}$$

The memops are 5.3 M memops and 36.4 MB. The flops are 2 M flops for a 2.65 memops to flops ratio. The expected performance is 4.20 msecs and the actual is 3.78 msecs.

References

[1] F Agner. Optimizing software in C++ an optimization guide for Windows, Linux and Mac platforms, 2011.

[2] John D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, December 1995.