LA-UR-20-27394
Approved for public release; distribution is unlimited.

Title: Parallelization and Performance Portability in Hydrodynamics Codes

Author(s): Dunning, Daniel Jeffrey

Intended for: PhD Dissertation

Issued: 2020-10-07 (rev.1)

Parallelization and Performance Portability in Hydrodynamics Codes

by

Daniel Dunning, B.S., M.S.

A Dissertation

In

Computer Science

Submitted to the Graduate Faculty
of Texas Tech University in
Partial Fulfillment of
the Requirements for
the Degree of

Doctor of Philosophy

Submitted to:

Dr. Yu Zhuang
Chair of Committee

Dr. Yong Chen

Dr. Sunho Lim

Dr. Nathaniel Morgan (LANL)

Dr. Mark A. Sheridan
Dean of the Graduate School

December 2020

This work has been released under LA-UR-20-27394.

Acknowledgements

I am thankful to all of the people at both Texas Tech and Los Alamos National Laboratory (LANL) that have helped me with the process of this degree. In particular I want to that Bob Robey at LANL for his continued support and guidance with the research. Nathaniel Morgan (LANL) was another huge contributor to my success and I am appreciative of his support. Also a thanks to my chair Dr. Yu Zhuang for his efforts in allowing me to pursue my research remotely at LANL. Of course a special thanks to my fiancé Madeline Whitacre for her continued support throughout the entire degree.

Table of Contents

Abstract

With the eve of Exascale computing, performance and portability are at the forefront of all scientific codes. Adding more cores and more energy to a system is no longer a sustainable way to achieve performance, and extra effort must now be made to improve performance in all areas of code and code development. Using hydrodynamic codes as a basis, this work explores numerous techniques to achieve performance in different ways. Adaptive mesh refinement (AMR) is a necessary technique to improve memory optimization in mesh-based simulations. However it is invasive and conventionally difficult to integrate into existing applications, so we present a new branch of AMR to create a smooth transition to these optimizations, which not only improves performance, but also greatly reduces developer effort. We introduce the concept of this improvement as Phantom-Cell AMR, and assess theoretically the improvements, as well as present an application of its use. Other work included involves and investigation into an efficient data structure that ensures optimal memory layout for cache performance, with a target of making codes performant and portable across all architectures. All of the work targets both performance and portability, not just on CPU hardware, but specifically across GPU architectures. Parallel performance is key to all of the methods presented, but the research makes a great effort to improve the portability of all applications to prepare for current high performance computing systems and those on the horizon.

List of Tables

List of Figures

Chapter 1

Introduction

The work presented in this dissertation has a strong foundation in numerical methods and grid-based simulations. As such, those numerical models are introduced here to represent the motivations for the research presented.

1.1 Computational Fluid Dynamics

The study of fluids is essential in many sciences and technologies including atmospheric sciences, oceanography, astrophysics, and propulsion systems. The Navier-Stokes equations of incompressible fluid flow are at the core fluid dynamics, representing simply the conservation of mass, Equation 1.1, and the balance of momentum, Equation 1.2.

$$\nabla \cdot U = 0 \tag{1.1}$$

$$\frac{DU}{Dt} = -\nabla P + \frac{1}{Re}\nabla^2 U \tag{1.2}$$

U here is the velocity vector, P represents pressure (divided by density), and Re Reynolds number. We are interested in predicting qualitative changes of a flow, with respect to Re. Although there is a large effort of experimental fluid dynamics, the research presented in this thesis focuses heavily on computation fluid dynamics (CFD). CFD is the computational approach to solving conservation of mass, energy, and momentum equations. These computations fall under the category of Navier-Stokes equations, either Lagrangian or Eulerian frame of reference, although some recent methods employ a combination of the two.

1.1.1 Lagrangian Method

Lagrangian views involve examining the movement of individual fluid parcels as they move from some initial position. Conceptually this method moves the co-

ordinate system along with the fluid and follow the trajectory of the parcels. The control volume moves with the fluid. Although the equations of motion from this approach are simple, the solutions only consist of the spatial locations at each time stamp for the fluid parcel. Furthermore, it is difficult to know which points in the field will display specific fluid properties beforehand. Another difficulty in terms of computation is that the Lagrangian method only follow individual parcels through the field. In order to represent the system completely, we need to track a large number of parcels, which is where the majority of the computation heaviness falls.

1.1.2 Eulerian Method

Eulerian views instead work with a fluid moving across a fixed spatial location. Individual fluid parcels are no longer the target of the view, rather, the fluid as a whole is measured directly at specific locations. For computational investigations, this approach is often simpler and more common than Lagrangian methods. However, there is a difficulty obtaining acceleration at single points in the space. The Lagrangian view has an ability which Eulerian lacks which is to give the total acceleration which is necessary for Newton's second law. This is because by following a fluid parcel we can obtain the acceleration at a single point and produce a formula for acceleration of the fluid elements. For this reason, Eulerian methods must express accelerations in terms of a Lagrangian system using the substantial (material) derivative. This is given in Equation 1.3

$$\frac{Df}{Dt} \equiv \frac{\partial f}{\partial t} + u\frac{\partial f}{\partial x} + v\frac{\partial f}{\partial y} + w\frac{\partial f}{\partial z} \tag{1.3}$$
$$\frac{Df}{Dt} = \frac{\partial f}{\partial t} + U \cdot \nabla f$$

Where U is a velocity vector $(u, v, w)^T$ and ∇ is a vector differential operator $\nabla \equiv (\partial/\partial x, \partial/\partial y, \partial/\partial z)^T$. This can be derived using the chain rule for any arbitrary fluid property. Consider a fluid parcel with property function f in Equation 1.4

$$f(x, y, z, t) = f(x(t), y(t), z(t), t) \tag{1.4}$$

Differentiating f with respect to t we get Equation 1.5

$$\begin{aligned} \frac{df}{dt} &= \frac{\partial f}{\partial x}\frac{dx}{dt} + \frac{\partial f}{\partial y}\frac{dy}{dt} + \frac{\partial f}{\partial z}\frac{dz}{dt} + \frac{\partial f}{\partial t}\frac{dt}{dt} \\ \frac{df}{dt} &= \frac{\partial f}{\partial t} + u\frac{\partial f}{\partial x} + v\frac{\partial f}{\partial y} + w\frac{\partial f}{\partial z} \end{aligned} \tag{1.5}$$

which is equivalent to Equation 1.3. As we discussed, our Eulerian acceleration can really be thought of as the Lagrangian acceleration. If we substitute the x component of velocity we say $f = u$ then we can show Equation 1.3 as Equation 1.6

$$\frac{Du}{Dt} = \frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} + w\frac{\partial u}{\partial z} \tag{1.6}$$

showing the total x-direction acceleration. $\frac{\partial u}{\partial t}$ is with respect to the Eulerian coordinates (local acceleration) and $u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} + w\frac{\partial u}{\partial z}$ is the Lagrangian contribution (convective acceleration). The convective acceleration is equal to $U \cdot \nabla u$ to give Equation 1.7

$$U \cdot \nabla u = u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} + w\frac{\partial u}{\partial z} \tag{1.7}$$

1.2 Adaptive Mesh Refinement

Adaptive mesh refinement (AMR) is a method of adapting a solution of numerical simulations to have higher accuracy in sensitive areas. Solutions often are confined to pre-determined grids on the Cartesian plane where the computation occurs. Several problems such as shock applications have areas where a higher degree of computation is required, possibly even a different numerical model entirely. AMR works to refine these areas of interest to increase the accuracy and consistency of the calculations. Berger et al. (??, 1989) first described a dynamic

grid adaptation they called *local adaptive mesh refinement*. This algorithm worked roughly by marking cells that needed refinement based on an application-specific criteria. Individual patches of those refined cells are created and computation is done on those during the timestep, with the result being present back to the original coarse grid interfaces. Of course if at any time a refined cell is no longer in an area of interest, the cell(s) can be coarsened back to their original state.

This refinement can be either dynamic or static based on the needs of the application. One problem may have an initial model that sets areas for refinement but those regions don't move much during the course of the simulation. In this case, a static, one-time refinement would be sufficient. Most problems, such as the dam break model discussed in Section subsection 2.1.1. Physical Model using the shallow water equations have a constantly changing area of interest. For this problem a dynamic refinement is necessary to maintain accuracy at the constantly moving regions. For this problem and others like it the dynamic refinement improve computational power as well as increased storage (when compared to a grid of the most refined level). One of the most beneficial aspects of AMR when compared to a static pre-determined mesh is that the adaptive one requires much less prior knowledge of the problem at the time of construction. If one were to create a single mesh that would represent the entire simulation it would have to be created with the knowledge of the how simulation was supposed to progress. If inputs or numerical models are changed the entire structure and construction of the mesh needs to be rebuilt.

1.3 Research

The remaining chapters of this document are largely based on papers published or submitted by myself. Some content has yet to be part of a publication, but the others are found from work as follows: Adaptive Mesh Refinement in the Fast Lane(Dunning, Marts, Robey, and Bridges, 2020), A Performance Analysis of Phantom-Cell Adaptive Mesh Refinement on CPUs and GPUs (Dunning, Robey, Kuehn, and Cook, 2020), and MATAR: A Performance Portability and Productivity Implementation of Data-Oriented Design with Kokkos (Dunning, Morgan, Moore, Nelluvelil, Tafolla, and Robey, 2020).

Chapter 2

CLAMR

CLAMR features two stencils for the finite difference calculations. The first is a cell-centric stencil, where the program loops over every cell of the mesh and computes the time-step from its neighbors in the cardinal directions. The second stencil is face-centric in which the program loops through every face of the mesh (both x and y-axis faces) and computes the time-step by querying the state from the two adjacent cells to the face.

2.1 Shallow Water Equations

The shallow water equations are currently the physics calculations implemented in CLAMR. A major reason for choosing them is because of there relative simplicity. This allows for ease of implementation in the code, helped by the low line count of the equations. In addition, the simplicity of the equations allow us to focus on optimizations within the code. These optimizations include vectorization, parallel implementations using OpenMP and OpenCL, and overall algorithm optimizations, talked about further in Section subsection 2.4.3. Mesh-based Methods. Another useful attribute of the shallow water equations is the high degree of symmetry that is presented in the initial timestep. When a cylindrical-type shock impacts the center of the mesh at the beginning of the simulation, it creates a circular wave. The wave has inherent symmetry because of the nature of the shock, and this symmetry because crucial to test to all iterations of the simulation. It creates something visually and numerically relevant that can be used as an easy "first pass" check to see whether or not the mesh is progressing as intended. The symmetry does not guarantee the correctness of the algorithm or the physics routine, but it is something that can immediately alert us to a problem in the mesh routines.

The shallow water equations, in their conservative form, are listed below:

$$\text{Conservation of mass}$$

$$\frac{\partial h}{\partial t} + \frac{\partial (hu)}{\partial x} + \frac{\partial (hv)}{\partial y} = 0$$

$$\text{Conservation of } x\text{-momentum}$$

$$\frac{\partial (hu)}{\partial t} + \frac{\partial}{\partial x}\left(hu^2 + \frac{1}{2}gh^2\right) + \frac{\partial}{\partial y}(huv) = 0$$

$$\text{Conservation of } y\text{-momentum}$$

$$\frac{\partial (hv)}{\partial t} + \frac{\partial}{\partial x}(hvu) + \frac{\partial}{\partial y}\left(hv^2 + \frac{1}{2}gh^2\right) = 0$$

where h is the height of a column of water, g is the acceleration due to gravity, u is the wave velocity in the x direction, and v is the wave velocity in the y direction. The speed of propagation as a result of these equations is $\sqrt{gh}$. This is the phase velocity of the wave, which is different from u and v, which are the velocities of the water molecules themselves. The phase velocity shows that this phase velocity is greatest at areas where the height (or depth) of the water is greatest. Tao (Tao, 2011) offers a discussion on the equations in regards to tsunamis.

It is important to note that for these equations mass is equal to height because water is an incompressible fluid. In fact, the incompressibility results in the pressure, here given as $gh^2/2$ to only vary the height of the water. The width and length of a particular column will remain constant for that given differential, and the density of the water in that column will stay constant. These equations all serve as the basic numerical method CLAMR is built on. However, adaptive mesh refinement makes the routines more complicated. The half-timestep equations must be adapted from regular grid equations because differences in area and distance to cell center now must be accounted for. In a regular grid all cells are the same size and have their timestep equations simplified as a result. The regular grid half-timestep equations are:

$$U^{n+1/2}_{i+1/2,j} = \frac{U^n_{i+1,j} + U^n_{i,j}}{2} - \frac{\Delta t}{2\Delta x}(F^n_{i+1,j} - F^n_{i,j}) \tag{2.1}$$

$$U^{n+1/2}_{i,j+1/2} = \frac{U^n_{i,j+1} + U^n_{i,j}}{2} - \frac{\Delta t}{2\Delta y}(G^n_{i,j+1} - G^n_{i,j}) \tag{2.2}$$

In accordance with standard notation the subscripts i and j are the spatial coordinates within the mesh, while the superscript variable n denotes a temporal coordinate. U represents a general state variable. For the shallow water equations it will take on mass, x-axis momentum, and y-axis momentum; in CLAMR these are labeled H (for height), U, and V. With this in mind, we can adapt the regular grid equations for adaptive mesh refinement as shown:

$$U^{n+1/2}_{i+1/2,j} = \frac{r_i U^n_{i+1,j} + r_{i+1} U^n_{i,j}}{2} - \Delta t \left(\frac{F^n_{i+1,j} A_{i+1} a_{i+1} - F^n_{i,j} A_i a_i}{V_{i+1} v_{i+1} + V_i v_i} \right) \tag{2.3}$$

$$U^{n+1/2}_{i,j+1/2} = \frac{r_j U^n_{j+1,j} + r_{j+1} U^n_{j,j}}{2} - \Delta t \left(\frac{F^n_{i,j+1} A_{j+1} a_{j+1} - F^n_{i,j} A_j a_j}{V_{j+1} v_{j+1} + V_j v_j} \right) \tag{2.4}$$

Here we add several variable not present in the regular grid equation. These include r, A, and V for the radius, area, and volume, respectively. In addition, we also include a and v which we call scaling variables for area and volume, respectively. These scaling variable are just a representation of the possible differing in size between two neighboring cells. It is clear that the the regular grid equations and adaptive mesh refinement equations are equal when considering two cells of the same size. The half-timestep calculations can all be thought of as the value at the face between two neighboring cells. The first expression in both equations is a linear interpolation of the state variable (U) at the face between the two cells. When working on a regular grid, this interpolation is simply an average of the state values from the two cells. However, the adaptive mesh equations must take into account the sizes of the cells, which is the reasoning for the inclusion of the

r term. The second expression in the equations is the term for the fluxes. The expression weights the fluxes, denoted by F and G, according to the area of the the cell interface, multiplied by the scaling factor a. In order for this expression to be computed for any arbitrary dimension, we must ensure that the scaling factor is always at a rate of less than or equal to one. As such, a will take the form:

$$a_i = min\left(1, \frac{a_{i+1}}{a_i}\right) a_{i+1} = min\left(1, \frac{a_i}{a_{i+1}}\right) \tag{2.5}$$

As mentioned, the denominator of the second expression is the volume. This also requires scaling factors for the adaptive mesh refinement equations, denoted with v. Similarly to the area scaling factor, the volume scaling factor must be modified to computed with any arbitrary dimension. It must be less than or equal to one half, and so the form is adapted to:

$$v_i = min\left(\frac{1}{2}, \frac{v_{i+1}}{v_i}\right) v_{i+1} = min\left(\frac{1}{2}, \frac{v_i}{v_{i+1}}\right) \tag{2.6}$$

We further extend the equations to their full-timestep calculation which takes the form:

$$U^{n+1/2}_{i+1/2,j} = \frac{U^n_{i+1,j} + U^n_{i,j}}{2} - \frac{\Delta t}{2\Delta x}(F^n_{i+1,j} - F^n_{i,j}) \tag{2.7}$$

$$U^{n+1/2}_{i,j+1/2} = \frac{U^n_{i,j+1} + U^n_{i,j}}{2} - \frac{\Delta t}{2\Delta y}(G^n_{i,j+1} - G^n_{i,j}) \tag{2.8}$$

Here we see that adapted half-timestep fluxes have been condensed into the single terms F and G. The equation presents the relation between the previous state value and the one calculated for the next timestep. Take note of the bar placed over both of the flux terms. This explains that in some cases this flux term is an average of neighboring flux terms, an in the case with a coarse cell having two refined neighbors. This average being present in the full-timestep calculation is important because it shows the additional complexity with calculating the state value in adaptive meshes. We would hope that this average could occur in the

half-timestep state calculation, and that the average would carry through the flux calculations and into the full-timestep calculation. However, the form of the flux terms for a variable do not match. Take the mass flux term in the x direction and x-momentum flux term, for example. For mass flux we use hu and the corresponding term for the flux from momentum is the expression $hu^2 + \frac{1}{2}gh^2$. To conserve the state variables, we must treat all cells equally and compute fluxes from the same perspective as the calculations done in the respective neighboring cells.

Because of the model that runs on CLAMR, there is a need for oscillation dampening within the state calculations. CLAMR uses a total variation diminishing (TVD) finite difference scheme which is mostly based on a two-step Lax-Wendroff method. The flux limiter is a minmod limiter which runs along the boundary. LeVeque's book on finite volume methods (Leveque, 2002) gives an in depth explanation into many TVD methods, as well as their numeric backgrounds and proofs of stability. The goal with these limiting terms is to be able to apply them so that at the discontinuous portions of the solution can have reduced or no oscillations while still keeping the smooth portion accurate. If not utilized properly, the TVD can improve the discontinuous section while imparting some inaccuracies, such as a phase shift, into the solution. The areas around steep shocks can impose oscillations in the methods which necessitates a need for these dampening terms. Here, this is done with a flux limiter creating a sort of artificial viscocity term. At a high level we would like the model to be aware of how the solution is behaving at neighboring cells and to choose a formula accordingly. Areas where the solution is smooth we can continue with the Lax-Wendroff slope. However, near discontinuities caused by shocks it is possible that we need to limit this slope to have a smaller magnitude so that oscillations are not formed. It is important to note here that it is not necessary that the total variation be constantly decreasing or diminishing, as the name suggests. Rather, it is crucial that the total variation be non-increasing.

A major takeaway from using this TVD scheme is the introduction of using a five-point stencil to capture the necessary information. The initial Lax-Wendroff scheme used to calculate fluxes is compressed in the sense that only the immediate

neighbors of each cell. The five-point stencil extends the request information from not only immediate neighbors, but of those neighbors' neighbors. The ratio of the corresponding gradient is examined and it is then determined whether there is a sign change, possibly indicating a oscillation, the flux limiter term can be applied. This is similar to the process of cell refinement in which the gradient of the cell and neighboring cells are examined and if it meets or exceeds a threshold the cell is refined. The final equation without oscillation stabilizers is:

$$U_{i,j}^{n+1} = U_{i,j}^{n} - \Delta t \left(\frac{\bar{F}_{i+1/2,j}^{n+1/2} - \bar{F}_{i-1/2,j}^{n+1/2}}{\Delta x} + \frac{\bar{G}_{i,j+1/2}^{n+1/2} - \bar{G}_{i,j-1/2}^{n+1/2}}{\Delta y} \right) \tag{2.9}$$

As an extension of the final state equation shown in Equation **??** we can show the corrections as:

$$U_{i,j}^{n+1} \stackrel{+}{=} \frac{v(1-v)}{2} \left[1 - \sigma(r^{+}, r^{-}) \right] \Delta U^{n} \tag{2.10}$$

σ is the flux limiter and ΔU represents the upwind difference in state variable. In addition, the Courant number, shown here in v, is a single variable representing the expression of the timestep divided by the grid spacing then multiplied by a state-corresponding eigenvalue of the system of equations. The quantifiable factors showing the change in gradient across the stencil are shown in the equation with r^{+} and r^{-}. These values are independent of dimension. They are calculated by taking the inner product of the finite differences and dividing it by the finite difference at the cell face. As with flux calculation, this is slightly complicated with cells of different refinement, but a simple interpolation of the inner product handles it.

2.1.1 Physical Model

The circular dam break problem is chosen in CLAMR because of its simplicity in demonstrating the shallow water equations. Toro (Toro, 2001) gives good insight into the problem as well as insight into the experimental analysis

of the model. The goal of the simulation is to solve the shallow water equations numerical to study the wave propagation caused by the dam break. The problem can be thought of as a simplified version of accompanying real life events. However, even with some conditions simplified, the model still possess crucial elements related to performing the numeric simulation. As mentioned, the shallow water equations are a good baseline for studying symmetry within a simulation, and the circular dam break problem is an excellent application for applying these equations. The problem follows the propagation of a wave as a result of a sudden collapse of a circular dam. Of course this isn't exactly the same as a real life event where a dam wall would break in steps as a opposed to the instantaneous disappearance. The simplicity of the model is helpful for CLAMR because it allows for a focus on mesh method design as opposed to an optimization effort strictly focused in the numerical computation. Wang, et al. (Wang, Liang, Kesserwani, and Hall, 2011) use a two-dimensional extension of a one-dimensional Godunov-type model. They propose a solution around wet-dry fronts which has the ability to be well balanced due to the simplified condition as a result of the dimensional extension. They compare their results with analytic benchmark solutions to assess the predictability in the event of a flood caused by an actual dam break. Alcrudo and Garcia-Navarro (Alcrudo and Garcia-Navarro, 1993) investigate an extension of another Godunov-type model. They also open the discussion regarding the common technique of applying two one-dimensional schemes individually as a representation of a two-dimensional scheme. Tseng et al. (Tseng, 1999) use extensions of common one-dimension Riemann solvers as non-oscillatory schemes imposed on the shocks from dam break-type problems. Lhomme and Gutierrez Andres, et al. (Lhomme, Gutierrez-Andres, Weisgerber, Davison, Mulet-Marti, Cooper, and Gouldby, 2010; Gutierrez Andres, Lhomme, Weisberger, Cooper, Gouldby, and Mullet-Marti, 2009) work on a methodology for testing new hydraulic model systems. The look to test the model with a combination of real-life data for broad correctness, and a suite of analytical tests which look to highlight shortcomings in specific areas of the model.

2.2 Adaptive Mesh Improvements

CLAMR was created to be a test-bed for mesh handling techniques. As a mini-application, it was built to mimic the properties of its parent application, but with a simpler numeric calculation, so that it may run easier tests and hopefully gain greater insight into AMR techniques in general. Mini-apps can be made to serve two different purposes. One train of thought is to precisely mirror the properties of the parent application, at both creation and advancement. The smaller "child" application can be used for application specific tests without the full capability of the overarching parent application. The other school of thought is to initialize the mini-app the same its parent application, but then look to advance methods inside of the application. In this way, the goal is that research into methods and optimizations in a smaller, simpler environment, and then work to transition those new improvements back into the parent application. CLAMR follows this later purpose. As such, it utilizes many state-of-the-art improvements to adaptive mesh codes. Some of these improvements are strictly performance based, adhering to recent research for improved neighbor look-ups, load balancing, etc. Others serve as improvements to code complexity. This is an achievement in and of itself, but it also helps to push methods towards a path that allows for that performance optimization.

2.2.1 Space Filling Curve

Spatial locality is crucial when working with multi-dimension meshes, such as AMR grids. Copious amounts of research in the past has been done to study efficient means of retaining locality when mapping to higher dimensions. A Z-order curve and Hilbert (??, 1891) curves are the most common, and both have implementations within CLAMR. However, previous tests found that for this particular application Hilbert curves were more efficient for maximizing spatial locality, so it will remain the target of this section.

Hilbert curves map one dimensional spaces to higher dimension spaces in a way that preserves locality. Locality here means that if two points are near each other in a one dimensional space, they will also be near each other in a two dimen-

sional space. This is extremely useful for an application like CLAMR because the two dimension regular grid can have a mapping to a one dimension data structure in a way that elements close to each other in the data structure are close to each other in the physical grid. This property is important for single node parallelism, but it is especially important when considering multi-node implementations of an application. With near perfect locality, we can assure that nodes need the minimal communication between them for certain neighbor calculations. each node knows which node neighboring cells lie because of the locality within the data structure. Without this locality, cells have to request information from multiple other nodes in an attempt to find the node that holds the information they need. This locality is trivial on a regular grid, but is greatly complicated with an AMR mesh, and this is reasoning behind having the best space filling curve implementation.

2.2.2 Neighbor Search

In addition to an efficient space filling curve, hashing provides an increased optimization in neighbor searching within CLAMR. The method used in CLAMR is an efficient hash based search algorithm developed by Robey et al. (Robey, Nicholaeff, and Robey, 2013). Considering an adaptive mesh, finding neighbors requires neighbor list for each direction of a cell. In creating these lists, a simple, naive approach would be to search through every other cell to determine if it is in fact a neighbor. This would be a $O(n^2)$ complex algorithm. A more efficient algorithm would be to use a *k–D tree* for neighbor searching. Robey et al. focused on an interative approach that would be more efficient and easily ported to a GPU. Using a bin-style sorting algorithm, a hash sized at the largest data value is created. Each element is then place in its own bin, and empty bins are removed. This hash table construction has $O(n)$ complexity and is also easily parallelized. OpenMP and GPU versions are implemented in CLAMR for efficient neighbor array creations as part of the mesh handling routines.

2.2.3 Face Method

In addition to the original cell-based finite difference calculation, CLAMR offers a face-based method. The method does not present different numeric cal-

culations. Instead, it calculates the state for each cell using the cell faces as the point of reference and indexing, instead of the cell itself. The idea is to reduce the redundant calculations that occur for state fluxes. Referring back to equation [EQUATION] each cell calculates a positive and negative flux for each state variable. We can imagine the common scenario in which neighboring cells calculate the same value for one's positive flux and the neighbor's negative flux. When we do these flux calculations per face as opposed to per cell, we eliminate this redundant calculation. We don't calculate a separate negative flux and positive flux. We calculate a "neutral" flux that occurs at that face and then each cell on the face can "use" that flux, with the corresponding sign. To index faces and use them for flux calculation we need to create data structures for these faces as well as data structures to map the relationships between cell to face and face to cell. These data structures hold information that is crucial for the state calculations. This includes the cells on either side of a face, the faces that are on either side of a cell, as well as the spatial coordinates of the faces similarly to the coordinates of the cells. This is a substantial increase in memory when comparing to the original cell method. An expected investigation is whether or not this different method is worth it. We do a look at this in later sections, but as a baseline it involves looking at the memory footprint necessary to even allow the new method, and then seeing whether or not the memory is used in such a way to speed up the calculations comparably.

2.3 Phantom-Cell AMR

Figure 2.1 shows an example of a 4x4 mesh in CLAMR after the initial refinement. The initial conditions represent the dam break problem discussed in Section subsection 2.1.1. Physical Model. This can be seen pretty clearly in the figure, despite the limited detail with the small size grid. Using CLAMR as the exploratory codebase, we developed a new AMR scheme that is a hybrid of existing cell-based and patched-based AMR schemes. We called this *phantom-cell* AMR. The method introduces a new class of cells titularly named *phantom* cells We have called these extra cells phantom cells because of the similar traits they share with the ghost cells used for MPI distributed domains. However, we also

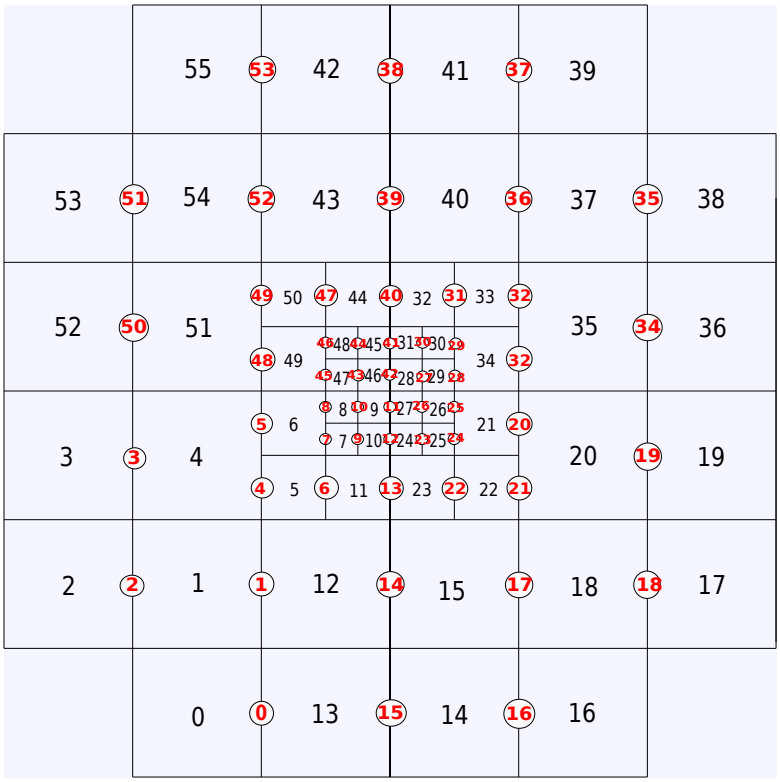

Figure 2.1. Starting three-level AMR mesh with cell numbers in center of cell and red face numbers in circles along faces. This will be used as a baseline to explain concepts throughout the paper.

want to acknowledge the difference between the two terms. In the case of phantom cells, the information carried is from another refinement level of the mesh rather than from another processor. For this reason, phantom cells are not present if the entire grid is the same level of refinement. Ghost cells are equivalent to the domain specific boundary cells called halo cells, except ghost cells border each processor's partition, as opposed to the entire mesh outline. Phantom cells are present in both serial and parallel implementations. We explored the feasibility of this new adaptive mesh refinement technique using a characteristic three-level AMR mesh. The existing AMR methods each have attractive properties, but they also each present unique implementation obstacles. Phantom-cell AMR aims to utilize the efficiency and minimal mesh imprinting of cell-based AMR while also utilizing the state-calculating simplicity offered by patch-based AMR. The net result of

this approach is a separation of mesh and physics codes, and allows the physics calculations to be entirely agnostic of the mesh complexity. The physics routines are simplified greatly regarding neighbor querying, but the mathematics remain the same. One big change that is seen in the routines is logical simplification in the step by step flow of the calculation. In the original cell-based method, and even in the face-based, each calculation can change based on the refinement of the cells surrounding the target location. In situations in which the equation itself remains the same, this introduces complexity in scenarios that don't require it. Relationships that are "simple", meaning the neighbors are the same refinement level, still must be treated as complex. The approach in the face-based method actually involves using two different functions for simple relationships and complex relationships. This is a perfectly valid mathematical approach, but it can certainly affect the debugging process, assuming that the two equations are in fact mathematically equivalent. This is not so much of a consideration in CLAMR where this branching occurs only at two individual places in the finite difference calculation. However in a more complex simulation, this branching situation can quickly overwhelm the code and become tedious to handle.

A key aspect of this technique is to give each level of the mesh to the physics code as a regular grid. The mesh and the physics state variables are modified when the mesh is passed to the physics code such that every cell has neighbors at its own level of refinement. This has some similarity to the technique used in patch-based methods but the areas of refinement can be of any shape. The difficulties of the clustering algorithm for patch-based schemes are seen to be overcome by keeping the core code cell-based. Relationships and refinement stay local, and the hybrid approach helps to overcome some of the shortcomings in simple parallelization from a cell-based approach by simplifying the physics routines. This showcases the goal of this hybrid approach in incorporating the strengths of existing AMR schemes to improve certain aspects of the other dogmas.

2.4 Phantom Cells

The base change in the routines involves offering additional relationships between cells to facilitate simpler finite difference calculations. In order to make

this work, additional cells that are called *phantom* cells are created at each refinement level. To transform our adaptive mesh into a pseudo-regular grid, we add these phantom cells and faces to our original cell and face lists. The phantom cells are at the same refinement level as their neighbors and are used to represent neighboring cells that are of a different refinement level. This is a key point. The original neighbors (from both refinement perspectives) are not removed or replaced. Simply they are represented to their neighbor as a single cell of the corresponding refinement. At every level of refinement we present a pseudo-regular grid. This can be thought of as treating treating each refinement level as one unified patch. The mesh acts as if it were a regular grid when it is presented to the physics routines. By allowing a way for the mesh to be perceived in this manner, we eliminate branching in the physics code involving logic based on the shape of the grid being given or on the refinement level of each cell's neighbors. Cells no longer need to check the refinement level of their neighbors or neighbors' neighbors. It is assumed in the finite difference calculations that for every cell its neighbors that are queried for state information will present themselves at the same level. This clearly reduces the complexity of the routines because all cells and be treated the same. The differences between equations acting on differently refined neighbors are removed from the physics routines. An important note that will be discussed in future sections is that this complexity of AMR is not eliminated from the code entirely. It is rather moved out of the physics routines and into the mesh routines, where it belongs.

The purpose of these phantom cells is to isolate the physics from the complexity of the AMR mesh. This is achieved by encompassing the state values of neighboring cells into a uniform cell of the same refinement level. Instead of directly indexing into an array, code written for our AMR method maintains a variety of mapping structures between mesh cells or faces and their respective neighbors. For example in CLAMR, to obtain the value of a right neighbor one would instead index into the right neighbor array e.g. ***nrht****[cellID]*. Working on a regular grid, this same action would be to use the spatial coordinates ***j****[cellID]****i****[cellID+1]*. It is important to note that these directional neighbor arrays used for mesh construction (***nlft****,* ***nrht****, etc.*) do not have their mapping modified

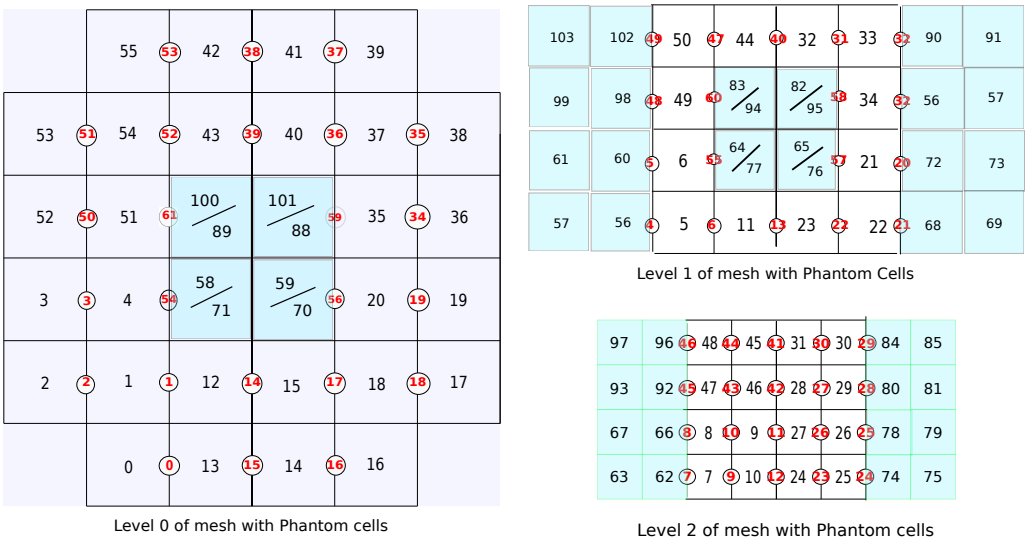

Figure 2.2. Phantom cells for faces in the x-axis. This figure shows where the phantom cells would fit if they were made into a regular grid with their level of refinement

for the original (i.e. non-phantom) cells. This is done to preserve the structure of the mesh itself for later rezoning. The new relationships between original and phantom cells are updated in separate mapping data structures. In fact, the mapping structures used to hold phantom cell relationships are the same ones that hold the mappings between cells and faces. These data structures are not used for refinement rezoning. They are initially created for the face-based method where it is necessary to index the faces between cells and conversely the cells on either side of the face where a flux calculation resides. These structures are updated to hold the phantom cells and phantom faces. The state-calculating routines that utilize the face relationships as well as the phantom cell relationships will index into these data structures to achieve the regular grid view. In this manner, refinement rezoning indexes the right neighbor as ***nrht****[cellID]* while the physics routines (except for the original cell-based method) will all query the cell's right neighbor by ***upper_cell****[****right_face****[cellID]]*. When we pass the mesh to the physics code it can perform its calculations as if it was on a regular grid because of the phantom cell neighbors. Every cell is treated the same and the computations are all performed as the simplest case i.e. the neighbor is at the same refinement level. This eliminates the need to worry about whether a neighbor is a different

refinement level or whether the neighbor's state value computations need to be weighted based on the area of the cell.

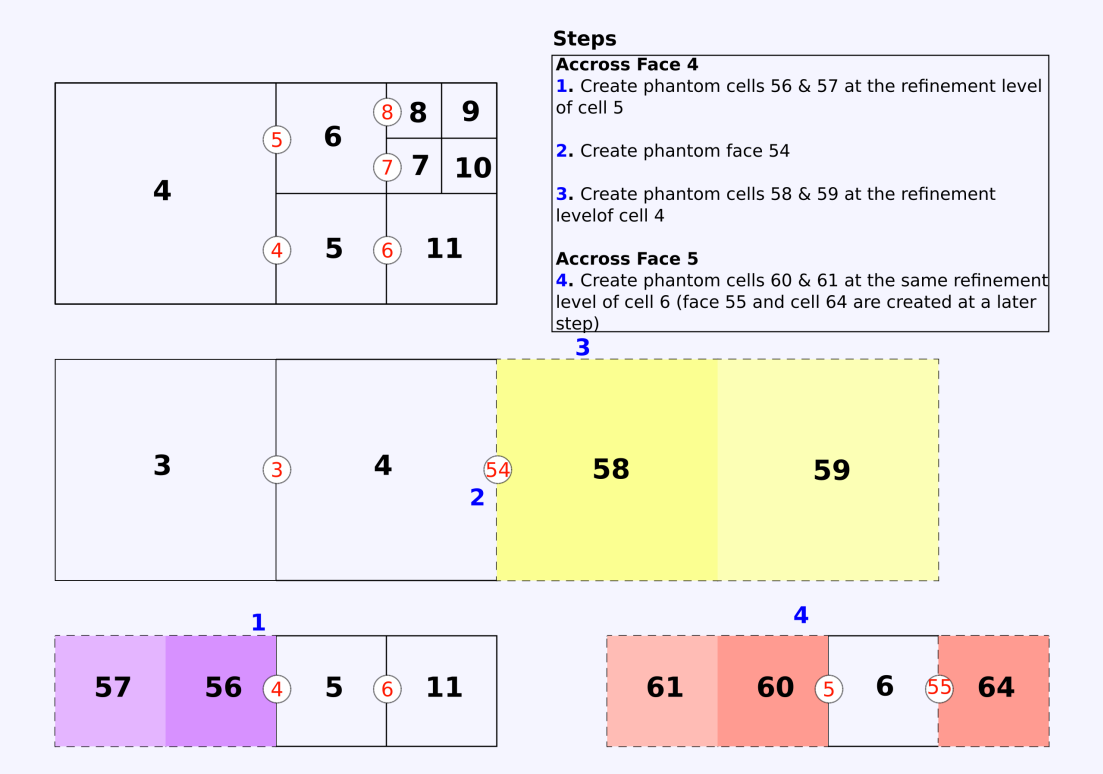

Figure 2.3. Example focusing on faces 4 and 5. This illustrates how the phantom cells and faces are created, and how they would appear to the underlying data structures.

2.4.1 Phantom Cell Construction Algorithm

A simple description of the method for adding phantom cells is shown in Algorithm 1. The process starts by looping through all faces in the mesh, first in the x-axis (left / right), then in the y-axis (up / down). Because these techniques are the same in each dimension, we will focus on the x-axis. Figure 2.1 shows the x-axis faces at the initial timestep. Using the bidirectional (cell to face and face to cell) mapping data structures, we obtain the two cells that share a face. If those cells are of the same level, we continue on as that is not a face where phantom cells are needed. If those cells differ in refinement level, we recognize that that is a place

add phantom cells and phantom faces. The data structures could be kept cleaner if we only needed a single phantom face at this refinement border. However, it became clear when working on the physics kernels that additional phantom faces between two phantom cells would indeed make the state calculation easier, and because that is the target of this work, the additional phantom faces are necessary. However, for diagram simplicity these extra phantom faces are omitted in the diagrams to avoid confusing in overlapping regions. Figure 2.2 gives a good representation of the phantom cells residing in the three-level mesh. Although the figure is broken up into three individual pieces, it is crucial to understand that this is purely for visual purposes. Each diagram represents the view of the individual cells of their respective similarly-refined neighbors. Cells colored light blue are the phantom cells and in areas where the same-leveled phantom cells overlap spatially we draw a diagonally partitioned cell (specifically looking for example at the level 0 mesh with cells 58 and 71). Note that the face immediately to the right of cell 4 is labeled face 54. If we compare this to the original mesh, we see this is the first phantom face because its face ID is one greater than the original largest face ID (i.e. face 53, located in the upper left corner between cells 55 and 42).

For a concrete example of this process in CLAMR, we will focus on faces 4 and 5 in Figure 2.3. At face 4 we can clearly see differing levels of refinement, and as such we will be adding four phantom cells and three phantom faces for around this single face. The first two phantom cells will be numbered 56 and 57, respectively, as shown in the figure, and they will be at the level of refinement of cell 5. Phantom face 54 is needed on the right edge of cell 4. This phantom face is the algorithmic mapping between cell 4 and the new coarse phantom cell 58. We add two phantom cells, 58 and 59, at the level of refinement of cell 4. In the same manner that the original faces are created between original cells, phantom faces would be added in between the new phantom cells. In this example this would occur, for example, between phantom cells 58 and 59. Two more phantom cells, 60 and 61 will be created across face 5 in a similar fashion to phantom cells of the same refinement in 56 and 57. This of course happens at a later step in the process because we must complete all phantom faces and cells for the given face (in the case initially face 4) before we move on to the next target face (face 5). Without going

into the redundant process, we can even look ahead in the diagram and imagine the similar process that will happen at faces 7 and 8 for cell 6 in which phantom cells need to be created on either side, two with a reduction in original refinement and two increasing the original neighbor refinement. The state values of these phantom cells are integrated for the coarse phantom cells based on the original fine cell, or interpolated from the original coarse cells for the finer phantom cells. This process is followed for all faces in the mesh, creating phantom neighbors at all refinement boundaries. The result is that for any given cell in the mesh, the physics finite difference calculation can request meaningful neighbor data regardless of its level of refinement. Looking ahead to parallel implementations, it is expected that ghost cells will have an overlap with phantom cells in the values they hold. Ghost cells can hold values of more refined neighbors present on another processor, and in this case phantom cells can aid that process by simply passing their own values to the ghost cells. This process has not been fully realized in CLAMR, but with improvements to MPI communication in progress these phantom cells may be key in facilitating neighbor state value communication.

Algorithm 1 *Adding Phantom cells and faces*

for all faces, if, in the mesh **do**
  if $level\_upper \neq level\_lower$ **then**
    Add one $face$ at end of face list that combines the two smaller faces
    For the larger level, add two $cells$ at the end of cell list and fill with $average$ for H, U, V
    For the smaller level, add two $cells$ at the end of the cell list and copy H, U, V from the coarser $cell$.
    Fill in neighbor lists and maps as needed
  end if
end for

CLAMR uses a *graded* mesh for its simulations. This means that there is a constraint on the jumps in refinement between two cells. If the mesh were ungraded two neighboring cells could have an arbitrary jump in refinement which can be useful in some applications where refinement is extremely variable. How-

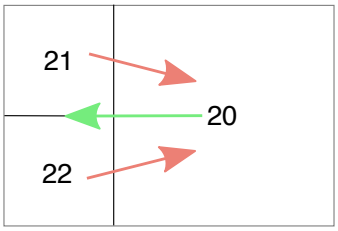

Figure 2.4. Flux passing. Positive (red) shown moving right and negative (green) moving left.

ever, it is obvious the complexity that this adds to the adaptive mesh refinement code and in this work it makes the most sense to use a graded mesh. CLAMR constrains the refinement ratio between cells to be a factor of two. This refinement constraint works well for compact stencil finite-difference or finite-volume problems. Other applications, however, may need a larger refinement, possibly a factor of ten or greater. For these applications octrees may be advantageous over hashing for neighbor connections because of the larger refinement jumps and vast number of neighbors a cell can have. An advantage of the phantom cells is that the construction would be almost exactly the same, even if this larger refinement jump is needed. Instead of querying for the two neighbors and getting an average value for those two cells, all neighbors would be indexed and a representative state value would be integrated from those cells. The change in mesh indexing does not affect the querying of the physics. This is a huge impact because of non-impact of the state calculation. If the mesh neighbor querying is coupled into the physics, the entire finite difference method is affected and must be modified in order to accommodate the different mesh it is built on. Phantom cells allow the physics to remain the same regardless of the mesh characteristics. The way that the mesh handles the AMR whether using octrees or whether it is even graded or not is invisible to the physics making this an attractive approach when developing an AMR code or modifying and existing regular grid application.

2.4.2 Phantom State Values and Conservation

To populate the phantom cells' state values, we need to either integrate the finer cells or interpolate from the coarser cells. Our basic integration scheme

is an average of the state values in the neighboring cells. Domain-specific schemes can easily be added and used in place of this simple scheme. To conserve mass throughout the system, it is crucial that the positive and negative fluxes of the cells, i.e. the fluxes moving in and out a of cell are equal. Calculating these fluxes in the physics routine without querying the refinement of neighbors is not trivial. Berger, et al. (Berger and LeVeque, 1998) used a technique for calculating the fluxes for each cell, and we implement a similar method. If the mesh class had previously flagged the cell, that cell passes on a directional flux to the corresponding neighbor. Figure 2.4 gives a visual representation of the process. This technique requires calculating the fluxes for each cell starting with the most refined, and working up to least refined. This is because it is the finer cells that will be passing their flux values to their coarser neighbors. When the coarse cell calculates its fluxes, it will use the average of the values it was given by its finer neighbors for that directional flux. We use the term directional flux here to describe the positive and negative flux in each x and y direction of a cell. An average of the state values being used by these equations does not yield the same flux as an average of the fluxes obtained by the original state values, which is a downside to this approach. It would be a huge improvement to the scheme overall if a technique was known to conserve these fluxes at the state level. As one is not known to my knowledge, averaging the fluxes in the state calculations is the only approach at this time. This is the reason we use a technique similar to Berger and LeVeque (Berger and LeVeque, 1998). In addition to the flux values of each cell, CLAMR includes corrector terms used for oscillation dampening. The integration and interpolation of state values in phantom cells will adjust these values to correspond with the phantom cells they are representing. These artificial viscosity terms must be be passed from more refined areas to coarse areas in the same way the fluxes are passed to guarantee mass conservation.

2.4.3 Mesh-based Methods

The original plan was to generate a regular grid at every refinement level and add the phantom cells onto the grids. This would leave a lot of gaps in each level and forced a path towards a way to compress this sparse mesh into a

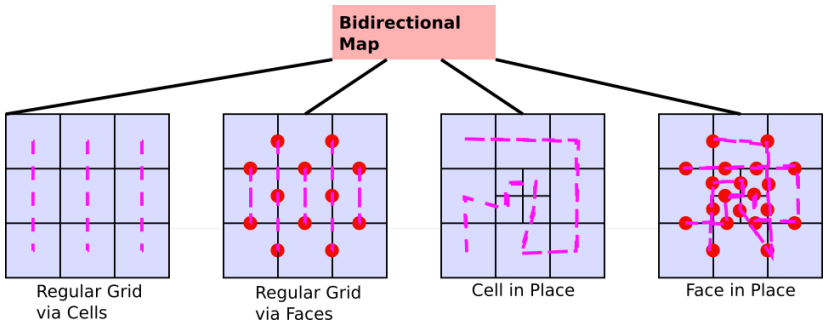

Figure 2.5. Various meshes generated give flexibility on adapting physics code to AMR.

more dense view. The breakthrough was that we the existing cell-based AMR data structure could be used and it would be a perfect compression the copy to a regular grid is eliminated. This method is what will be referred to as 'in-place' methods. Even though the memory is definitely inefficient, we will continue to look at the regular grid option because it allows the use of existing stencil optimizations. This is a large factor in maintaining the regular grids because the stencil optimizations are very efficient and if other factors can be improved the optimizations are highly valued. This is clear in the huge draw in existing research to work with patch-based AMR based on its closeness to regular grids. Using the bidirectional maps that are already presented in the original face-based method we are able to generate any of the meshes shown in Figure 2.5. Each method indexes through mesh in different ways, but the similarities are quite clear. This actually presents advantages when using the methods as a basis of testing different mesh strategies. The physics calculations can remain quite similar while allowing modifications to be mainly made in the mesh handling. This is an indirect but expected consequence of the investigation of phantom-cell AMR because the complexities of the mesh and grid indexing are removed from the physics, allowing the calculations to look similar in spite of drastically different mesh views.

Algorithm 2 *Regular Grid Algorithm*

map ← calculate bidirectional map with phantom
meshes ← generate regular cell meshes
for all levels, lev, from fine to coarse **do**
  for all cells j, i, from 2,2 to jmax-1,imax-1 **do**
    if $(mask_{j,i} \neq 1)$ **continue**
    // Halfstep in time and space
    $U^{1/2}_{j,i-1/2} \leftarrow (U_{j,i-1} + U_{j,i})/2 - \frac{\Delta t}{\Delta x} * (F_{j,i} - F_{j,i-1})$
    $U^{1/2}_{j,i+1/2} \leftarrow (U_{j,i} + U_{j,i+1})/2 - \frac{\Delta t}{\Delta x} * (F_{j,i+1} - F_{j,i})$
    $U^{1/2}_{j-1/2,i} \leftarrow (U_{j-1,i} + U_{j,i})/2 - \frac{\Delta t}{\Delta y} * (G_{j,i} - G_{j-1,i})$
    $U^{1/2}_{j+1/2,i} \leftarrow (U_{j,i} + U_{j+1,i})/2 - \frac{\Delta t}{\Delta y} * (G_{j+1,i} - G_{j,i})$
    // Flux corrector of some sort
    if (mesh interface) **then**
      $L^{1/2}_{lev-1,j/2,i/2} += F^{1/2}_{j,i\pm 1/2} + G^{1/2}_{j\pm 1/2,i}$ + Flux corrector
      $N^{1/2}_{lev-1,j/2,i/2} ++$
    end if
    // Fullstep using fluxes at cell interfaces
    $U'_{j,i} \leftarrow U_{j,i} - \frac{\Delta t}{\Delta x} * (F^{1/2}_{j,i+1/2} - F^{1/2}_{j,i-1/2} + G^{1/2}_{j+1/2,i} - G^{1/2}_{j-1/2,i} + L^{1/2}_{lev,j,i}/N^{1/2}_{lev,j,i})$
  end for
end for
destroy regular cell meshes(meshes)

2.4.3.1 Regular Grid Methods

In the regular grid code in Algorithm 2, the finite difference computation begins with generating a regular grid based bi-directional map from face to cell and cell to face. This is essentially a copy of the same mapping data structure used for phantom cells, but it has to be modified to accommodate i and j spatial coordinates. Then a set of regular meshes gets generated with the cells at the level filled in and a mask set for them. This mask will signal to the physics routine that the given cell is a cell it should be doing computations on, rather than a blank "island" due to cells of different refinement level physically present here.

This is not unlike the same way in which even the original cell-based method uses a mask to distinguish between real cells and boundary (ghost or domain based) cells. The phantom cells are then filled in so that there are two neighbor cells guaranteed (one immediate neighbor, one a cell away) in each direction. This allows the stencil calculation to be written for a simpler regular grid.

For a two-step predictor-corrector method, the first half-step is done for each face of the cell. Since we are guaranteed now to have a neighbor cell at the same level, there only needs to be four lines of code, one for each face. Most compressible fluid codes have some sort of flux corrector or slope limiter that is then applied at each face. CLAMR uses a simple, centered Total Variation Diminishing (TVD) scheme (Davis, Schemes, and Viscosity, 1984; Davis, 1987; Yee, 1987) which is discussed in detail in Section 2.1. Up to this point, the calculations are fairly standard for a regular grid. Now the same calculations are done for the mesh interface faces as if all the cells are the same size and the flux is summed to an accumulator for the level, L, and a count, N, is made for the number of writes to the accumulator. The last step loops across the cells to sum the contributions of the fluxes across the faces into the cell. The flux from a different level is treated as just another face flux from a "fourth dimension" that is automatically adjusted for the cell size. This is shown in Algorithm 3. There are a couple of optimizations that can be investigated further within these regular grid methods. When using the grid at the finest level of refinement, there is really no need for L fluxes to be added since there is no finer level. We can assume that these lowest level fluxes hold the correct values that will conserve the mass. In addition, assuming the refinement algorithm is working properly, the TVD flux correction does not need to be computed at every refinement level. The flux correction is for shocks and material discontinuities and they should all be in the finest level of the mesh. This could be a substantial impact because those oscillation dampening terms make up almost half of the computations within the finite difference calculation. While this is clearly important, it is not discussed further at this time because it is not an optimization specific to the use of phantom cells, it can be applied even at the original cell-based method.

The regular grid with faces, shown in Algorithm 3, does the first part of the

computation in a loop over the faces. This reduces the redundancy in calculating the advection from each cell's perspective, reducing the floating point operations by nearly a factor of two. Of course, this is the same improvement that is made when transitioning from the original cell-based method to the original face-based method. To accomplish this savings, additional arrays for both x-faces and y-faces must be allocated. Also, in fine-grained parallel loops, there must be a synchronization between the face and cell-based loops. This synchronization can be expected in all face-based methods because all of the fluxes must be calculated at the faces before the individual cells can use those fluxes.

Algorithm 3 *Regular Grid with Faces Algorithm*

map ← calculate bidirectional map with phantom
meshes ← generate regular cell meshes
faces ← allocate faces
for all levels, lev, from fine to coarse **do**
  for all faces, j, i, from 1,1 to jmax,imax **do**
    if $(mask_{j,i} \neq 1)$ **continue**
    // Halfstep in time and space
    if $mask_{j,i+1} = 1)$ **then**
      $U^{1/2}_{j,i+1/2} = (U_{j,i} + U_{j,i+1})/2 - \Delta t/\Delta x * (F_{j,i+1} - F_{j,i})$
    end if
    if $mask_{j+1,i} = 1)$ **then**
      $U^{1/2}_{j+1/2,i} = (U_{j,i} + U_{j+1,i})/2 - \Delta t/\Delta y * (G_{j+1,i} - G_{j,i})$
    end if
    Flux corrector of some sort
    if (mesh interface) **then**
      $L^{1/2}_{lev-1,j/2,i/2} += F^{1/2}_{j,i\pm 1/2} + G^{1/2}_{j\pm 1/2,i}$ + Flux corrector
      $N^{1/2}_{lev-1,j/2,i/2} ++$
    end if
  end for
  for all cells, j, i, from 2,2 to jmax-1,imax-1 **do**
    // Fullstep using fluxes at cell interfaces
    $U'_{j,i} = U_{j,i} - \Delta t/\Delta x * (F^{1/2}_{j,i+1/2} - F^{1/2}_{j,i-1/2} + G^{1/2}_{j+1/2,i} - G^{1/2}_{j-1/2,i})$
  end for
end for
destroy regular cell meshes(meshes)

2.4.3.2 In-place Methods

The in-place techniques contrast regular grid methods by being an extension of the original AMR methods. For both the cell in-place method and face in-place method there are no new data structures created in the mesh. The phantom cells and faces are simply added to the existing mapping lists, and neighbor

relationships within these structures are modified to reflect the addition of the phantom cells and faces. As noted in Section 2.3, the data structures that map the original cell neighbors are not changed. The phantom cells are added to the bidirectional data structures to facilitate mapping original cells to phantom cells through the original newly added phantom face. These are the data structures which are primarily used for the mapping to help the physics calculations, and as a result they are recomputed at the end of each timestep. Both cell in-place and face in-place method are shown in Algorithms 4 and 5, respectively.

Algorithm 4 *Cell In-place Algorithm*

map $\leftarrow$ calculate bidirectional map with phantom

// Halfstep in time and space

for all cells ic, from 1 to ncells **do**

  if $(mask_{ic} \neq 1)$ **continue**

  $nl \leftarrow map\_xface2cell\_lower[map\_xcell2face\_left1[ic]]$

  $nr \leftarrow map\_xface2cell\_upper[map\_xcell2face\_rht1[ic]]$

  $nb \leftarrow map\_yface2cell\_lower[map\_ycell2face\_bot1[ic]]$

  $nt \leftarrow map\_yface2cell\_upper[map\_ycell2face\_top1[ic]]$

  $U^{1/2}_{x-} \leftarrow (U_{nl} + U_{ic})/2 - \frac{\Delta t}{\Delta x[lev]} * (F_{ic} - F_{nl})$

  $U^{1/2}_{x+} \leftarrow (U_{ic} + U_{nr})/2 - \frac{\Delta t}{\Delta x[lev]} * (F_{nr} - F_{ic})$

  $U^{1/2}_{y-} \leftarrow (U_{nb} + U_{ic})/2 - \frac{\Delta t}{\Delta y[lev]} * (G_{ic} - G_{nb})$

  $U^{1/2}_{y+} \leftarrow (U_{ic} + U_{nt})/2 - \frac{\Delta t}{\Delta y[lev]} * (G_{nt} - G_{ic})$

  // Flux corrector of some sort

end for

// Flux fixup

for all fixup faces $ifacefixup$, from 1 to nxfixup,nyfixup **do**

  $F^{1/2}_{coarsecell} \leftarrow F^{1/2}_{finecell}$

  $G^{1/2}_{coarsecell} \leftarrow G^{1/2}_{finecell}$

end for

// Fullstep using fluxes at cell interfaces

for all cells ic, from 1 to ncells **do**

  $U'_{ic} \leftarrow U_{ic} - \frac{\Delta t}{\Delta x} * (F^{1/2}_{x+} - F^{1/2}_{x-} + G^{1/2}_{y+} - G^{1/2}_{y-})$

end for

Algorithm 5 *Face In-place Algorithm*

map $\leftarrow$ calculate bidirectional map with phantom
// Halfstep in time and space
for all face $iface$, from 1 to nxface **do**
  $c- \leftarrow map\_xface2cell\_lower[iface]$
  $c+ \leftarrow map\_xface2cell\_upper[iface]$
  $Ux^{1/2}_{iface} \leftarrow (U_{c+} + U_{c-})/2 - \frac{\Delta t}{\Delta x[lev]} * (F_{c+} - F_{c-})$
  $Fx^{1/2}_{iface} \leftarrow f(Ux^{1/2}_{iface})$
  // Flux corrector of some sort
end for
for all face $iface$, from 1 to nyface **do**
  $c- \leftarrow map\_yface2cell\_lower[iface]$
  $c+ \leftarrow map\_yface2cell\_upper[iface]$
  $Uy^{1/2}_{iface} \leftarrow (U_{c+} + U_{c-})/2 - \frac{\Delta t}{\Delta y[lev]} * (G_{c+} - G_{c-})$
  $Gy^{1/2}_{iface} \leftarrow f(Uy^{1/2}_{iface})$
  // Flux corrector of some sort
end for
// Flux fixup
for all fixup faces $iface$, from 1 to nxfacefixup **do**
  $F^{1/2}_{coarseface} \leftarrow F^{1/2}_{fineface}$
end for
for all fixup faces $iface$, from 1 to nyfacefixup **do**
  $G^{1/2}_{coarseface} \leftarrow G^{1/2}_{fineface}$
end for
// Fullstep using fluxes at cell interfaces
for all cells ic, from 1 to ncells **do**
  $U'_{ic} \leftarrow U_{ic} - \frac{\Delta t}{\Delta x} * (F^{1/2}_{x+} - F^{1/2}_{x-} + G^{1/2}_{y+} - G^{1/2}_{y-})$
end for

The in-place techniques have the advantage of keeping the same control flow as existing cell-based and face-based AMR, while only increasing the memory minimally. Each cell and face can query its neighboring cell using the bidirectional

maps. However, apart from the differences in indexing, in-place methods and regular grid techniques operate the same. The way in which the regular grid with cells method operates on cells and communicates with neighbors is equivalent to how the cell in-place technique operates (e.g. working through refinement levels separately). Similarly, face in-place performs calculations on faces and cells in the same manner as the regular grid with faces. This similarity maintains simplicity, allowing the methods to function the same way and keeping the physics computations largely unchanged.

These in-place methods are the main platform on which the phantom cells can be fully utilized. They show how phantom-cell adaptive mesh refinement can be incorporated into an existing AMR code with minimal intrusion into the base code. Many code bases shy away from an AMR implementation because of the complexity it adds to the numeric calculations. Even codes that have an adaptive mesh in place, modifying the corresponding methods can be a huge task and without a guarantee of substantial improvement the possible changes can be halted. A huge benefit of phantom-cell AMR is that the extreme simplification of physics state calculations does not come at the cost of equal complexity added to the mesh handling. Although at a baseline the refinement-based logic is moved out of the physics into the mesh, the methods involved in the logic are inherent to the mesh code. It makes sense that querying for different sized neighbors should be handled by the mesh itself that knows about the view of the mesh. When these uncertanties are presented to finite difference calculations the methods must start from the bottom and check for every refinement possibility. Obviously the same checks are done in the mesh, but they are already present because of other methods that required them. When we add faces to the mesh and the subsequent mappings to cells, we already must query the state of the mesh and the different neighbor relationships. The functions that handle this are written so that we only need to add some additional information as a result of learning of the different mesh relationships. The state calculations are formulated to try and work the same for each cell, so if it can be presented wit this view, it is already setup in the most optimal way.

2.4.4 Performance Portability

At its core, performance portability is the idea that codes be able to be run with adequate performance across many hardware architectures within many software environments. It is natural that certain hardware (in the same performance range) will optimize code slightly differently and there are bound to be different performance characteristics between software platforms. However, if we can keep the difference due to run system minimal, we can investigate the performance difference and draw conclusions on what areas of the code need broader optimizations. It is important to eliminate machine-dependent optimizations. This requires extra efforts to be made at both the building stage as well as running stage of the application. Build systems have to be more conscious of the vast array of variants in HPC machines so that the hardware on these machines can perform to the best of its abilities in any situation. This extra complexity is a reason few applications have made the effort to assess their performance portability. It is a huge task for full scale applications to focus on their application needs, let alone spending time on simulation-independent efforts. In addition, the complexities of portability should be transparent to the user. It is not feasible to force user's to adapt their runtime system variables to match the many options of the build system. In fact, it should be the opposite. The build system needs to handle any runtime system that is present which can be made easier by having the system be as automated as possible. CMake helps this process greatly, but work still is needed to allow CMake to build the makefiles to allow for performance portability. The goal is software reuse. Applications should be able to not only run across multiple HPC platforms, but be perform well on all of them with minimal to no expectation from the user as well as endure.

2.4.4.1 State of HPC

Before discussing performance portability in high performance computing environments, it is important to mention the current regime that computing is in. Current HPC systems are composed of several hardware devices and architectures. For a homogeneous systems these hardware devices and the software layers have stronger guarantees as to how each component communicates and how

computation and errors are handled. Heterogeneous systems do not have these guarantees. Initially heterogeneous computing simply differed in instruction set architectures (ISAs), but now areas of heterogeneity included memory distributions (i.e. shared or distributed), network interconnects, software APIs, among others. These systems use multiple, different processing devices to compute tasks that they should be able to perform the best at. Computing on heterogeneous systems is a relatively new research interest within the computing field. Experiments done by Anderson et al. (Anderson, Culler, and Patterson, 1995) and Becker et al. (Becker, Salmon, Sevarese, and et al., 1999) in the 90's were some of the first efforts done on multi-software, multi-hardware systems. Even in 1996 there were no commercially available heterogeneous systems, and the small number that did exist were for scientific calculation (Ekmecic, Tartalja, and Milutinovic, 1996). Allocation of resources and characterization of the system were among the challenges in programming for these early system. Ekmecic et al. argue that there is a need for heterogeneous computing, and that there are applications and problems that would benefit from the use of these heterogeneous systems. In today's world, these heterogeneous systems are not rare. In fact, they dominate the charts for top supercomputers, with Oak Ridge National Laboratory's Summit system at number one. Heterogeneous systems have been found to consistently outperform homogeneous systems (O'Brien, Pietri, Reddy, Lastovetsky, and Sakellariou, 2017), often times with less energy and power consumption. Performance measured as a correlation to energy consumption is an important metric to consider when dealing with Moore's law. More compute needs to be harnessed at the same or lower power level to continue to advance computing into the Exascale [1] regime.

The rapid development of parallel arithmetic units relative to the slower improvements in memory bandwidth has pushed floating point (FLOP) to memory load ratio to around 50 FLOPs/memory load (McCalpin, 2016) across almost every architecture. But most scientific applications are memory-bandwidth limited (or a similar limit) rather than compute bound. It also means that we have to come to terms with there theoretically being many extra FLOPs that can be done without any additional cost. Essentially, this suggests that FLOPs are free. As

[1] Exascale here referring to computing systems that are capable of performing *exaFLOPS* i.e. 10^{18} floating point operations per second.

will be discussed later, FLOPs rarely have a direct correlation to runtime because these systems are memory bound. The constant almost exponential increase in FLOPS that new systems present have caused some application designs to try and be more computation focused. However, this is entirely the wrong direction. Method design needs to be more data focused. The FLOPs will be acquired through the systematic improvements of the machine calculations, but the memory has not followed the same level of improvements. This is a huge bottleneck for most applications, and it needs to be a focus of developer effort in the future. This is the reason that heading forward, a lot of our focus is on memory bandwidth comparisons, in addition to (and sometimes in correlation with) runtime performance.

2.4.4.2 Performance Portability Methodology

Performance portability has been a topic of discussion within recent years, but there has been difficulty among the community to give it a clear, precise definition. Pennycook, Sewall, and Lee have made great efforts to put forth a common definition (Pennycook, Sewall, and Lee, 2016a). In addition, they take care to define certain aspects of performance portability, so that the language used when discussing it is clear to all. The research in CLAMR in performance portability truly utilizes these definitions, and so it is important to present them here for this work. The first two definitions serve as building blocks for future discussion:

> *platform* – a particular execution environment (i.e. hardware an operating system, some compilation and runtime tools)
>
> *application* – any suite of software that can accept a given problem as input and produce an output that can be validated against some existing measure of correctness (Pennycook, Sewall, and Lee, 2016a)

Pennycook et al. further give definitions for *performance* and *portability*. These build upon the previous definitions stated and I agree fully with them. The definitions are as such:

Performance – Any measurable property of an application's correct execution of a problem on a platform.

Portability – The ability of an application to execute a problem correctly on a given set of platforms. (Pennycook, Sewall, and Lee, 2016a)

With all of these pieces in mind, Pennycook et al. formulate a precise definition of performance portability, which will be the definition use for the remainder of this work:

Performance Portability – A measurement of an application's performance efficiency for a given problem that can be executed correctly on all platforms in a given set. (Pennycook, Sewall, and Lee, 2016a)

This definition serves to give a clear, but generalized, expectation for what we expect from performance portability.

2.4.4.3 Quantifying Application Performance Efficiency

We acknowledge that it is difficult for full production applications to implement aspects of performance portability. It was mentioned in Section section 2.2. Adaptive Mesh Improvements the roles that mini-apps can play in code development. In fact, investigations into performance portability are an excellent use of smaller applications that have their roots in a large scale parent application. A lot of the work in CLAMR has been done to better improve performance portability. CLAMR is a research-based code, as opposed to production level. This allows us to take a look at several options that maybe not end up being feasible, but the knowledge is accumulated so that future code development can benefit from choosing the most efficient approach. This will be discussed in later sections when we talk about using OpenCL as the GPU path and when we discuss performance portability as a means for choosing the best optimizations. Prior efforts have looked at the sweep algorithm in SNAP (Deakin, McIntosh-Smith, and Gaudin, 2016) and structured grids(McIntosh-Smith, Boulton, Curran, and Price, 2014).

This work builds on that earlier research and attempts to extend it to looking at adaptive mesh refinement applications. McIntosh et al. had concerns about the testing of performance portability on representative applications. They acknowledge, as we do here, that is in time consuming to present a code that has aspects of performance portability suitable for testing. By investigating several structured grid applications, they hoped to shed light onto the efforts needed for testing a more real-life application. For the analysis presented using CLAMR, it was important to use an application with characteristics of an irregular, unstructured grid. It would be expected that we could apply the performance portability methodology to this code and then extend those techniques to the even more complex applications. CLAMR uses unstructured neighbor lists for mesh refinement and it already can be built and run using most compilers on top of all of the major CPU and GPU architectures. It is fortunate that the research pertaining to performance portability can be tested in an environment where we can really make use of these runtime environments. The work is done on the Darwin REFERENCE cluster at Los Alamos National Laboratory. This cluster provides a single testing area that gives access to most of the applicable compilers, CPU hardware vendors and chips, and GPU cards (both AMD and Nvidia). The ability to build, run, and test the code with all of these build system variables in a single place is useful because it gives the ability to do a sort of "real world simulation" of moving code from machine to machine, without having to physically move the code away from the initial cluster. In addition to hardware portability, the mesh and state methods can be run across multiple parallel software frameworks such as OpenMP, MPI, and OpenCL (for GPU kernels). This is another advantage of investigating performance portability in CLAMR. It is difficult to implement all of these portability regimes for anything other than a simple example code. The fact that these are already implemented in CLAMR gives us a great baseline to test performance portability in a real application, where most other applications don't have this type of base to build upon.

In addition to providing useful definitions of performance portability, Pennycook et al. present the equation that will be used to evaluate the performance portability of CLAMR (and can be used for evaluating any other application). The

equation is shown in Equation 2.11. Without a specific metric for performance portability, it is difficult to quantify and analyze performance across diverse architectures in some objective manner. We mention that the applications need to run with comparable performance, but there needs to be some sort of quantitatively relevant value that can actually be compared from run to run. This performance portability metric, Φ is defined for a set of platforms, H, for solving a problem, p with application a. The equation quantifies the previously stated definition for performance portability. If an application doesn't run on any platform in the set, the overall performance portability value from this equation is 0. The choice of the set of platforms H that the application is run on is left to the performance analyst. It is important that this set is inclusive of the major software technology and hardware vendors. Leaving a common platforms out may leave a gap in research because there should be serious inquiries why that platform was not used. The equation uses the harmonic mean to weigh the individual values so that it can be more resistant to outliers, while at the same time have a greater influence from lower individual values.

$$\Phi = \begin{cases} \dfrac{|H|}{\sum_{i \in H} \frac{1}{e_i(a,p)}}, & \text{if } i \text{ is supported } \forall i \in H \\ 0, & \text{otherwise} \end{cases} \tag{2.11}$$

(Pennycook, Sewall, and Lee, 2016a)

The performance portability, Φ, of an application obviously relies on many factors. To control these factors and give this metric meaningful information, we must evaluate the application through different lenses. Our focus for each assessment comes from the *efficiencies* of the application. The efficiency term, $e_i(a, p)$, is located in the denominator of the lower term. The efficiency is defined for a problem, p run with application a. There are intentionally many definitions of efficiency that can be made, because there are many areas of performance portability we are interested in. Pennycook, et al. defined two possible efficiency terms:

1. *application efficiency* – achieved performance relative to the best implemen-

tation on that platform

2. *architectural efficiency* – achieved performance relative to the peak theoretical performance of the hardware

Application efficiency measures how well the different performance-based implementations perform across the platforms. In most cases, the application efficiency is feasible only for small benchmark codes that have multiple, optimized implementations of the methods. Specifically we can look at parallel implementations using MPI, OpenMP, Cuda, OpenCL, and many others, to investigate the applications performance relative to its optimization. The difficulty in testing this efficiency is that a lot of work is needed to get all different parallel implementations working in the code. Just as McIntosh et al. mention, we are lucky if there is even one parallel implementation that runs on a particular platform and so this efficiency is difficult to calculate for more complex applications. This is why CLAMR is a useful application to test performance portability; it is a complex application that already has several parallel implementations. The effort needed for the research is then to run performance analysis on these implementations, which can even help further optimize these routines.

Architectural efficiency measures how well the application performs on the hardware. It is obvious that there are many different performance characteristics that can be used when investigating hardware performance. For the work done in measuring this in CLAMR, we choose to primarily look at the memory bandwidth. Memory bandwidth is a useful characterstic to look at because it is so important in application performance. Applications are rarely limited by the FLOPs that can calculated on the hardware. Often it is the efficiency of the memory usage that holds back applications. So while investigations can target numerous hardware counters, bandwidth optimization is a solid direction to start at. One other performance characteristic that can be quite useful in current times is the energy consumption of applications. This is a sort of compilation of several factors, but with today's supercomputers it becomes more important. The ability to run large scale simulations as well as cool machines has become a cost point for many institutions. For many years man-hours has been the biggest contributing

factor to cost, but the need for extreme amounts of power in the wake of Moore's law, commonly peaking into the Megawatts range, has paved the way for another large contributor to cost.

When working with CLAMR, we wanted to add areas of investigation to performance portability. Through the work with phantom-cell AMR, many different finite difference calculation methods were created that opened up an avenue for possible performance portability research. We settled upon two new types of efficiency, *method efficiency* and *optimization efficiency*.

3. *method efficiency* – achieved performance of a method relative to the best implementation of that method

4. *optimization efficiency* – achieved performance of an implementation relative to the implementation effort

Method efficiency compares different implementations of a numerical calculation. As with all other types of efficiencies, there must be some sort of value that we can compare one run against another. For method efficiency we could give several possible choices for efficiency comparisons, as follows:

1. best method runtime/method runtime

2. method bandwidth/best method bandwidth

3. best method data volume/method data volume

4. best method FLOPs/method FLOPs

5. method FLOPs per sec/best method FLOPs per sec

All of these comparisons have a ratio of an method implementation versus the best method implementation. The best method may be in either the numerator or denominator, depending on whether a high value is good or bad. In fact, these comparisons can be used for any efficiency, not just method. Based on our investigation, it seems that runtime is often the most consistent factor when

comparing efficiencies. The essence of the problem is that each of the methods produces the same, or reasonably equivalent, solution. But they do it with different amounts of data or FLOPs. To complicate things, each method also gets different efficiencies of the use of the memory hierarchy and floating point units. We cannot disentangle the effect of the differences between the methods in the volumes or rates of each method. That leaves us with only the runtime as the only choice for the efficiency. The caveat for this is that it is only the best choice for comparison when looking at end results. It should be noted that it is common for other comparison factors, such as bandwidth, memory volume, etc. to be the best point of investigation when looking to optimize applications and methods. This is why we focus more on these choices in pursuit of studying optimization efficiency. This efficiency is quite difficult to quantify because it requires giving comparable values for developer effort. We do so in general terms, such as line counts, man hours, and cost, but these all lead to broad generalizations. For this reason the studies presented here cannot give clear answers for performance portability in reference to optimization efficiency. Rather, we can only look to provide insight into external factors that may not be considered in conventional performance portability calculations.

We tested the CLAMR code across a range of HPC architectures and software environments. For performance portability, we used 1) runtime for method efficiency and 2) measured memory bandwidth for architectural efficiency. There was not much effort in devoting a complete evaluation of application performance in CLAMR. Especially when looking at method efficiency, there is ample study in the performance of different parallel architectures, and this work can carry over as a sort of pseudo application efficiency study. For all tests the code was run for 500 cycles with a starting mesh of 1024 x 1024 cells.

2.4.4.4 Architectural Efficiency

We tested various CPU and compiler combinations for our architectural efficiencies. The CPU chips tested were: Intel Gold Xeon 6152, part of the Skylake family, the AMD EPYC 7551, ARM ThunderX2-B0, and an IBM POWER8. There is a common consensus that these are the major CPU vendors, and so

Method	Architecture: Intel	AMD	ARM	IBM	OpenMP	NVIDIA (TITANX)	NVIDIA (V100)
Cell	730	562	153	243	30645	5937	35185
Face	6521	6923	1933	2760	71428	17307	56250*
Cell In-place	4571	4900	1730	2891	39386	26750	131000
Face In-place	7318	6733	2028	3300	25560	3600	153000
Reggrid Cell	5848	4238	1667	3948	X	X	X
Reggrid Face	2879	1597	706	1668	X	X	X

Bandwidth (MBytes / s)

Table 2.1. Table showing bandwidths across all architectures for each method implementation. *Means calculated based on other GPU runtime

it was important to be able to run tests for all of them. For GPUs we ran the methods on Nvidia's Volta V100 and AMD's Radeon R9. In addition to the serial implementations, parallel implementations on the CPU using OpenMP were tested with the Intel Skylake CPU. The main reason for using the Intel CPU for these tests was that the performance counters that we could gather were the best for Intel. It is expected that other architectures would perform similarly with the OpenMP technology, so it is not necessary to run this exact test for all of them. The bandwidths for each method implementation are shown in Table 2.1. These were calculated using the total memory volume shown in Figure 2.6. Using Equation 2.11 described in Section 2.4.4.2, we were able to get a measure of the architectural efficiency for our physics calculation methods. Figure **??** shows the architectural efficiency for each method implementation. We calculated three separate groups of architectural efficiency. The *Serial* group is the efficiency across only the four CPU systems tested. *Heterogeneous* is the efficiency including both CPUs and GPUs. *Parallel* efficiency includes only parallel implementations, in this case OpenMP and GPU accelerations.

None of the efficiencies are where we would expect for a production level code. This is to be expected when we are working from an investigation, research standpoint as opposed to a production point of view. ARM and IBM chips in particular had poorer performance than would be expected, given the chips used. This lowers the overall architectural efficiency. For instance, the architectural efficiency for serial implementations of the face in-place method was 15%. However, if we don't include the efficiencies from IBM and ARM, we obtain an architectural efficiency of 42%. This shows us that more work has to be done in optimizing

methods for other architectures, in this case, ARM and IBM. This study already shows the importance of performance portability in applications. It is not feasible to have a code base that has this difference in performance across these two main CPU vendors. In some cases, the optimization required could even be as simple as adding compiler flags to tailor performance to those architectures. This makes a good case for tools such as CMake. In this situation, asking all users of the code to properly handle all optimization compiler flags for each architecture would not be possible. Passing of the code with a generalized build system allows the performance portability to be handled by the developer, not the user.

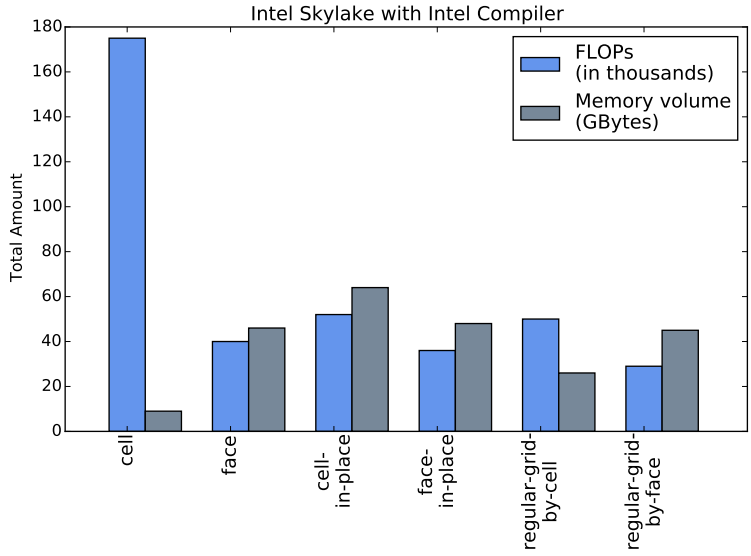

Figure 2.6. The total FLOPs (vectorized) and memory volume for each of the CLAMR computational methods.

2.4.4.5 Method Efficiency

The second type of efficiency we explore is *method efficiency*. We created several different methods for solving the physics calculation, and then create a substantial amount of parallel implementations of these specific methods. The next step is to test the method efficiency, using the best one as our "method benchmark" and comparing others against it. It is stated earlier that we would be using runtime to quantify the "best" method. Each method deals with a different

number of total flops and memory volume (see Figure 2.6) so the only constant that we could judge the methods against each other is runtime. Table **??** shows the runtimes for each method. Parallel methods performed the best on the GPU using OpenCL across all methods implemented, as expected. The best performing method on Nvidia architecture is the OpenCL implementation of the original cell method. This is the original method CLAMR was built on and has had the most optimization work, so the performance of this one did not come as a surprise. However, the best overall performing method is the OpenCL implementation of the regular-grid-by-faces method on the AMD GPU. Figure **??** displays the method efficiencies for each implementation.

The original cell method had the best parallel implementation (except across different GPU chips), so it has almost 100% efficiency for strictly parallel implementations. Face and face in-place methods often shared fastest runtimes for many of the implementations, except for parallel-only runs, so their efficiencies were often the highest. Face in-place is of particular interest in this regard. This method was written to help eliminate mesh-based conditionals in the physics code. This is useful for both the programmer and compiler. Obviously it is easier for a developer to optimize code that follows a single control flow. It is true for the compiler as well, which we saw with the vectorization from the Intel compiler. The compiler is able to vectorize loops and data movement when there is no break in control flow (i.e. if-else conditionals). Intel does this even when not explicitly told, but only if the code stays in the same control flow. The benefit of doing a study of method efficiency is the ability to see which methods present the ability for greater optimization beyond explicit developer changes.

Many questions about the CLAMR mini-app, the systems, the compilers, and the methods were spurred by this analysis of performance portability. We mention just a few highlights in the following analysis.

2.4.4.6 Programmer Productivity

Although programmer productivity or developer effort is closely tied performance portability, and I believe it deserves its own discussion, especially in the

context of phantom-cell AMR. All of the data collected for performance portability seek to give concrete values that can determine what aspects and implementations of an application are worthy of evaluation. These values can be compared and determine what implementations or methods are the best for the application. Often these come down to what can make the code run the fastest, and in rare cases, the most energy efficient. However, this actually doesn't include a large portion of the time and cost that goes into running an application. I believe that the runtime and energy cost of an application are the most highest end-goal for code development, but the pursuit of these must take in to account the programming effort needed to achieve these goals. It is clear that the quicker the application runs, the more simulations and tests can be done, allowing for more research to be accomplished. However, if the effort to speed the code up by a small amount takes exponentially longer to implement, it does not seem worth it. This is a key reason why many applications choose not to pursue an adaptive mesh approach to their simulation, and instead stick with a regular grid. It might be helpful to create higher resolution at certain areas, but the cost to achieve this might be a total change to the numerical methods needed to accommodate varying-sized cells. In fact, this is a huge benefit of phantom-cell AMR. It gives the ability to transform a regular grid code into an adaptively refining mesh with little programming effort. Suddenly a better simulation can be pursued because it is now feasible. The specific steps required for this are further discussed with the FIESTA code in Chapter chapter 3. **Wildfire Simulation**, but this transformation is made possible because of phantom-cell AMR.

It is clearly difficult to test full-size applications for performance portability because of the difficulties in getting multiple methods and method types implemented. For CLAMR, we tested six different methods across four CPUs, four separate compilers, along with parallel implementations for four of the methods. This is roughly 144 combinations of architecture, software, and methods that needed testing, and there are obviously numerous other CPU chips, compiler vendors, and parallel frameworks we did not test. It is not feasible for applications to achieve performance portability across all combinations of these, and so developers and researchers have to choose which ones to invest in. The targets

should be the best performing architectures and software with respect to the effort it takes to develop the methods for them. Harrell et al. (Harrell, Kitson, Bird, Pennycook, Sewall, Jacobsen, Asanza, Hsu, Carrillo, Kim, et al., 2018) discuss a supportive term to performance portability which they call *developer effort*. In a general sense, this is the effort contributed on the application developer side to implement features. These features can be any sort of contribution, ranging from domain specific calculations, to full parallel implementations of the algorithms. Obviously the goal of parallel frameworks and their utility is to speed up the code by considerable amounts. In this regard, many parallel technologies have made great efforts to make this implementation as easy as possible. There has been increasing interest in platforms such as Kokkos (Edwards, Trott, and Sunderland, 2014) because of the flexibility and relative simplicity it gives to creating portable, high-performance codes. Harrell et al. went further to define a more specific ratio of speedup to implementation effort which they labeled as *relative development time productivity (RDTP)*. This is notably used for parallel implementations, where the relative speedup of the code is compared with the developer effort to implement those parallel features. Ideally, the RDTP would be a high number. Obviously, this ratio can be improved upon by targeting the components, speedup and effort. Nvidia's CUDA and AMD's Hip technology work to target both components, but given the specific nature of these technologies coupled with their hardware, it makes sense that they would focus on the speedup of the code. Technologies like Kokkos work to improve the other aspect, the developer effort. Kokkos keeps the the performance stable in compared to those other technologies (naturally, as those frameworks are still the backend for Kokkos), but the developer effort is reduced greatly, improving the RDTP ratio.

2.4.4.7 Optimization Efficiency

Obviously the examples are taken from the perspective of the technology vendors who must make choices about what features of performance and portability are available to applications. They are not porting applications to utilize CUDA code, but rather improving CUDA itself. The responsibility falls on the developer to decide which of these technologies give the best performance and

the best portability. In this sense, we look at the new term called *optimization efficiency*. This term differs from other 'efficiencies' defined previously. There is no comparison to a best performing optimization. Rather, we investigate whether an optimization is worthwhile. In these contexts, we categorize an optimization as being worthwhile if it exhibits a good RDTP ratio. In most cases it can be a safe assumption that well-implemented thread parallelism is an efficient optimization. This can be both on node OpenMP or GPU implementations. Granted, a deeper look must be taken whether *both* are efficient (which is a powerful case for technologies like Kokkos).

This concept extends beyond parallel implementations, and is in fact a core design goal of phantom-cell AMR. Harrell et al. specified their developer effort metrics even further using an equation to quantify the effort required to port code to various platforms. Effort here is measure in 'person-months'. The equation is shown below in Equation 2.12

$$\text{effort } [\text{person} - \text{months}] = A \cdot M \cdot (\text{SIZE})^F \tag{2.12}$$

M here is a function for an effort multiplier. F serves as a function corresponding to a scaling factor, A is a calibration constant, and SIZE is the lines of code (in thousands). For this effort, there is a direct relationship between line count and the effort required, so it is clear that line count is a driving factor for porting effort. Simplifying physics and mesh components not only creates a more cohesive mesh library, but it also allows for optimizations of the physics code, independent of the mesh. The physics code is simplified, and at a bare minimum has a greatly reduced line count and conditional count.

Figure 2.7 shows the reduction in change from the original methods with the in-place and regular grid methods. Additionally, Listings **??** and **??** shows a comparison of part of the code used for each method. Using the equation described above, we can estimate the reduction in effort as a result of phantom-cell AMR. Based on line count alone, this reduces the effort of porting this code by roughly a factor of two, for each of the scaling factors (SF). Importantly, this is independent for each type of parallel framework that the code might be ported to.

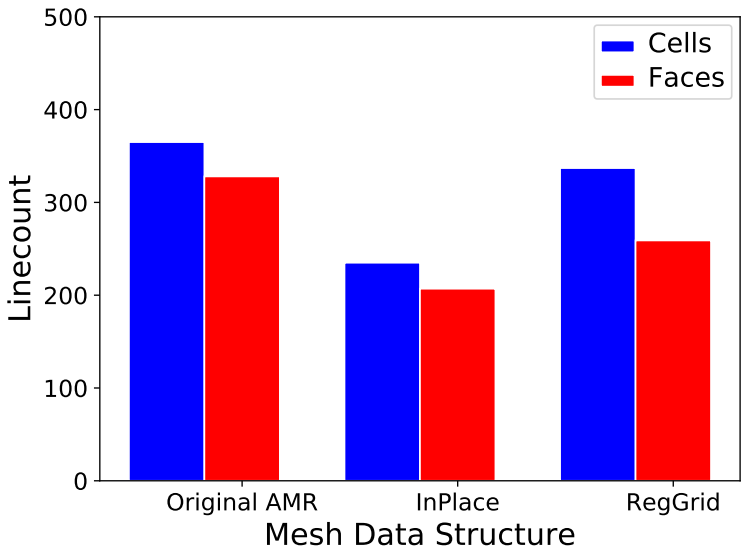

Figure 2.7. Line Counts for Phantom AMR versus Original AMR methods

This was experienced fairly accurately when porting the applications to OpenCL. The specifics the performance of the parallel technologies is discussed further in Section subsubsection 2.4.4.10. A Pathway to Parallel and GPU Computing. However, conversation about the implementation efforts are suitable here. The original cell method performs well in comparison to the phantom-cell based methods. However, the path to get the method in this state was longer and more difficult than the other method. This method underwent years of optimization for its implementations. On the other hand, the OpenCL implementation of our face in-place method had comparable performance, and was implemented in under a month. In addition, the original cell method needs to be entirely rewritten if it needs to perform a different physics calculation because the mesh handling is intertwined with the physics code. Face in-place only needs the physics model changed, the mesh querying will remain exactly the same. At this time this extended effort can not be evaluated because we are working with a single physics state calculation. If CLAMR had multiple physics models implemented, we could compare the effort it took to port the coupled cell-based method against the decoupled in-place methods. This would be an informative study to do in the future. This would give some sort of metric on whether this 'optimization', the reduction

in line count, would in fact be an improvement in regards to developer effort.

Another example of an optimization worthy of investigation is vectorization. Vectorization is supposed to be fairly easy to obtain in most scenarios with some special attention to loop structure, and the addition of the *#simd* command. In addition, the compiler flags for enabling vectorization should be relatively similar in turning on and off vectorization. However, through some in depth study on vectorization in CLAMR, we found that this was not always the case. The first investigation was to compare different compilers, and see if vectorization was achieved somewhat uniformly across them. For an easy comparison, we looked at the Intel compiler and compared it to GCC's compiler. These are two of the more common compilers used in scientific computing, and they are both able to give substantial compiler reports respective to vectorization. GCC's compiler was able to build CLAMR across all of the system architectures we tested, which allowed it to be a consistent basis of comparison in regards to many of the performance portability studies. The biggest discovery from comparing the two compilers was the that the GCC compiler was not able to vectorize any of the state calculation methods, while the Intel compiler was able to achieve substantial vectorization of the loops. Figure 2.10 shows the total FLOPS achieved on Intel's Skylake architecture with the GCC compiler, as well as the memory bandwidth we measured. Figure 2.9 shows the same measurements but this time using Intel's own compiler. The main outlier seems to be the FLOPS achieved in the original cell-based method. When looking at Figure 2.11, we can see that the cell-based method had a huge number of vectorized FLOPS. Although this is still an outlier for vectorization, it is clear that this was the contributing factor to the total number of FLOPS being so high for this method. Unfortunately we were not able to do a more in-depth study as to the reasoning behind the exceptionally high vector FLOP count. This would certainly be a worthy future study. Interesting of note, however, is that the method itself did not perform better than most other methods, so it is questionable whether or not these vector FLOPS were fully utilized. The more pressing question is why the GCC compiler was not able to vectorize the loops in these physics methods. What is the performance portability of vectorization if only one compiler shows the ability to achieve it in a real code?

If one compiler vectorizes and another one does not, our assumption about the supposed simplicity of adding vectorization is incorrect. A known reasoning for vectorization not happening is the presence of logic based conditionals inside of the loop. These conditionals prevent the compiler from vectorizing because it can no longer assume that all of the data in the vector unit will be performing the same operation. This is similar to performance problems on GPUs where a thread warp or wavefront is slowed substantially because it encounters a conditional where the threads don't follow the same logical tree of operations. This is in fact a huge advantage of phantom-cell AMR because our logic statements are reduced roughly 70%, as seen in Figure 2.8. With this in mind, we would expect the original cell and face-based methods to not vectorize, while the in-place methods should vectorize. However, even after removing *all* conditionals in the in-place methods, we were still unable to achieve vectorization. The compiler reports indicated that for all of the loops it perceived some sort of unauthorized aliasing of the data structures. This was not clearly happening on the developer side, so we were at a loss as to what needed to be changed. After a substantial period of trial and error, we found the solution. We needed to add an additional set of compiler flags no related to vectorize, and we needed to dereference the data structure pointers. The compiler flags in general were ones that would allow 'unsafe' math operations (operations on *INF* and *NAN*). The reasoning for the necessity of these is still unclear. Speculating for the deferencing, on the other hand, is easier. An example of the dereferencing is shown in Listing **??**. Even if that new pointer is not in use, it signals to the compiler that the data structure within the loop is in fact a valid pointer, and it can be loaded into the vector units as such. With these findings, we have sent reports to the GCC compiler team notifying them of this behavior. The hope would be that the loops would vectorize with the same ease as the Intel compiler, for instance. The extra effort needed for vectorization on the GCC compiler really showcases the need for considering optimization efficiency for developer efforts. Application developers who want to use the GCC compiler would have to make decisions as to whether the non-trivial process of getting vectorization to work would be worth the time devoted to it. Even with a minimal performance improvement the effort required to implement it might have a higher time or monetary cost. Further, at its current implementation in CLAMR

the vectorization does not present a suitable performance increase and in fact, for most of the methods built using GCC they performance decreased. To look at this further, we would have to come up with an equation similar to the one presented by Harrell et al. that could describe this effort in terms of performance optimization ability. At this time, it seems that we do not have the expected performance portability in the vectorization effort. This seriously undercuts putting effort into this form of parallel optimization. This is especially unfortunate because a lot of effort has been done on the hardware vendor side in improving the vector units on these high performance chipsets, but at this time I would doubt whether any real application achieves portable performance gains from vectorization.

Optimization efficiency shows the benefits of evaluating performance portability within an application. Evaluating the performance portability is necessary when looking ahead to future projects or optimizations. Time rarely permits for several method implementations such as the ones we tested, and the ability to choose the method that will give the best performance portability is valuable. Working on smaller mirrored codes, optimizing them, and then proceeding to a production-level application could be an interesting approach for applications. For this work, we learned that certain CPU hardware does not perform as well as others, and needs attention if we want to achieve greater portability. In addition, we observed that an optimization targets where the performance of one method's optimization did not match the performance of other implementations. This gives direction to future performance investigations. Our original cell method performed well with fully-optimized implementations. Based on this, as well as efficiencies for other some of the other methods, it seems like a worthwhile investment to fully optimize it in the other methods methods. Although these conclusions are specific to this CLAMR code, they could not be realized without the assessment of performance portability within the application, and shows the necessity of such assessment.

2.4.4.8 Difficulties with Evaluation

Testing performance portability on full applications can be difficult because of the complexities within numerous methods. There were several problems

we encountered that made performance portability testing considerably more difficult. The biggest challenge was profiling the methods for useful metrics other than runtime. Memory bandwidth, memory volume, FLOPS, and even energy/power consumed are all extremely valuable metrics when evaluating the efficiency of an application and its methods. Likwid (Treibig, Hager, and Wellein, 2010) was the best profiling tool for our work, but even then we were only able to get precise measurements on a handful of architectures. AMD's *rcprof* gave the best results for counters on AMD GPUs. Software version and dependency disagreements for GPU profilers made it difficult to gather exact metrics for our OpenCL methods on Nvidia GPUs other than the kernel runtimes and overall application runtimes. The tools we had available were helpful in gathering the data presented, but for future work it would be worthwhile to look into other tools or even look at development of profiling tools. There does not seem to be a common consensus within the high performance community as to the direction that performance profilers are going. This makes it difficult to enhance and optimize codes, specifically GPU-based codes. The Intel compiler is especially good at picking up on vectorization within the physics loops, but as mentioned, others need precise instructions and redundant instruction, or don't support vectorization at all. We think that performance portability testing could be greatly improved with a more consistent set of profiling tools. It is not reasonable that there should be a one size fits all tool for all architectures and compilers. However, it goes against the core of performance portability to have tools which can only be used with a particular few architecture and environment pairings.

2.4.4.9 Compiler Difference

The selection of compiler turned out to be an important consideration. The Intel compiler vectorized the physics calculation while GCC did not. But GCC worked across all the architectures, making it a more consistent basis of comparison. We studied differences in results from Intel's Skylake building with the GCC compiler and with Intel's own compiler. Figures 2.10 and 2.9 show each method between two separate runs built with the different compilers.

Adding a simple *#pragma omp simd* before each loop signals the compiler

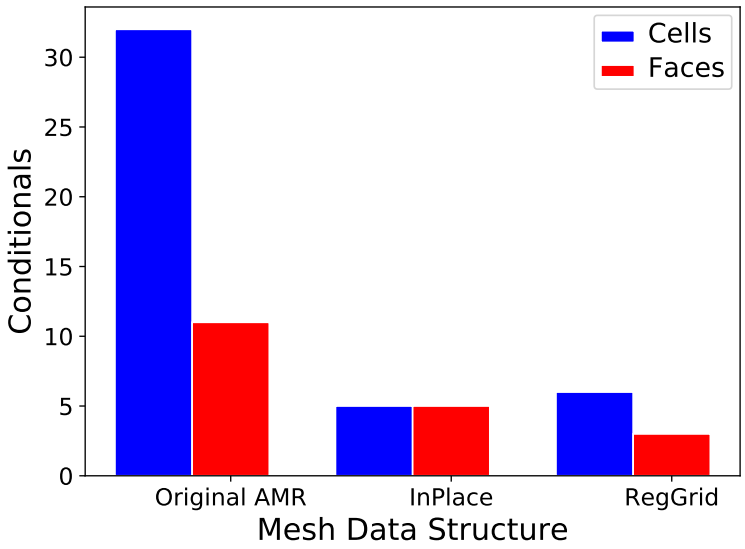

Figure 2.8. Logic conditionals counts for Phantom AMR versus Original AMR methods

to vectorize that section, which can make an impact on performance in the form of FLOPs per second and bandwidth. Figure 2.11 shows the difference in vectorized and non-vectorized FLOPs. With vectorization the Intel compiler was able to achieve about two times the FLOPs per second and one and a half times the bandwidth consistently, in comparison to GNU's compiler.

2.4.4.10 A Pathway to Parallel and GPU Computing

CLAMR was originally written to evaluate AMR performance on a variety of computing hardware and especially GPUs. From the start, the code was developed with OpenCL. A core feature of OpenCL is the ability to produce GPU code for both NVIDIA and AMD hardware. This was a main draw for us because a focus of the work as a mini-app is to evaluate multiple hardware architectures. This is often done by looking at performance on different CPUs, but the ability to look at performance across multiple GPU vendors is perhaps even more important.

With the phantom-cell approach, the physics code becomes much simpler to port to the GPUs. Loops that iterate across the cell and face lists are easy to

Listing 1 Face in place CPU method.

```
for (int iface = 0; iface < mesh->nxface; iface++){
   int cell_lower = mesh->map_xface2cell_lower[iface];
   int cell_upper = mesh->map_xface2cell_upper[iface];
   //if (cell_lower >= mesh->ncells && cell_upper >= mesh->ncells) continue

   // set the two faces
   int fl = mesh->map_xcell2face_left1[cell_lower];
   int fr = mesh->map_xcell2face_right1[cell_upper];

   //if (fl == -1 || fr == -1) continue;
   real_t Hx, Ux, Vx;

   // set the two cells away
   int nll = mesh->map_xface2cell_lower[fl];
   int nrr = mesh->map_xface2cell_upper[fr];

   uchar_t lev = mesh->level[cell_lower];
   real_t dxic    = mesh->lev_deltax[lev];
   real_t Cxhalf = 0.5*deltaT/dxic;

   real_t Hic = H[cell_lower];
   real_t Hr  = H[cell_upper];
   real_t Hl  = H[nll];
   real_t Hrr = H[nrr];
   real_t Uic = U[cell_lower];
   real_t Ur  = U[cell_upper];
   real_t Ul  = U[nll];
   real_t Urr = U[nrr];

   Hx=HALF*(H[cell_upper]+H[cell_lower]) - Cxhalf*( HXFLUX(cell_upper)-HXFLU
   Ux=HALF*(U[cell_upper]+U[cell_lower]) - Cxhalf*( UXFLUX(cell_upper)-UXFLU
   Vx=HALF*(V[cell_upper]+V[cell_lower]) - Cxhalf*( UVFLUX(cell_upper)-UVFLU

   real_t U_eigen = fabs(Ux/Hx) + sqrt(g*Hx);

   Wx_H[iface] = w_corrector(deltaT, dxic, U_eigen, Hr-Hic, Hic-Hl, Hrr-Hr)
   Wx_U[iface] = w_corrector(deltaT, dxic, U_eigen, Ur-Uic, Uic-Ul, Urr-Ur)

   HxFlux[iface] = HXFLUXFACE;
   UxFlux[iface] = UXFLUXFACE;
   VxFlux[iface] = VXFLUXFACE;
}
```

Listing 2 Face in place GPU method.

```
    if (giX < nfaces[0]) {
        int cell_lower = map_xface2cell_lower[iface];
        int cell_upper = map_xface2cell_upper[iface];

        int fl = map_xcell2face_left1[cell_lower];
        int fr = map_xcell2face_right1[cell_upper];
        real_t Hx, Ux, Vx;

        int nll = map_xface2cell_lower[fl];
        int nrr = map_xface2cell_upper[fr];
        //if (cell_lower != nll && cell_upper != nrr) {
        //if (fl > -1 || fr > -1) {

        uchar_t lev;
        if (cell_lower < ncells)
            lev = level[cell_lower];
        else
            lev = level[cell_upper];
        real_t dxic    = lev_dx[lev];
        real_t Cxhalf = 0.5*deltaT/dxic;

        real_t Hic = H[cell_lower];
        real_t Hr  = H[cell_upper];
        real_t Hl  = H[nll];
        real_t Hrr = H[nrr];
        real_t Uic = U[cell_lower];
        real_t Ur  = U[cell_upper];
        real_t Ul  = U[nll];
        real_t Urr = U[nrr];

        Hx=HALF*(H[cell_upper]+H[cell_lower]) - Cxhalf*( HXFLUX(cell_upper)-HXF
        Ux=HALF*(U[cell_upper]+U[cell_lower]) - Cxhalf*( UXFLUX(cell_upper)-UXF
        Vx=HALF*(V[cell_upper]+V[cell_lower]) - Cxhalf*( UVFLUX(cell_upper)-UVF

        real_t U_eigen = fabs(Ux/Hx) + sqrt(g*Hx);

        Wx_H[iface] = w_corrector(deltaT, dxic, U_eigen, Hr-Hic, Hic-Hl, Hrr-Hr
        Wx_U[iface] = w_corrector(deltaT, dxic, U_eigen, Ur-Uic, Uic-Ul, Urr-Ur

        HxFlux[iface] = HXFLUXFACE;
        UxFlux[iface] = UXFLUXFACE;
        VxFlux[iface] = VXFLUXFACE;
        //}
    }
```

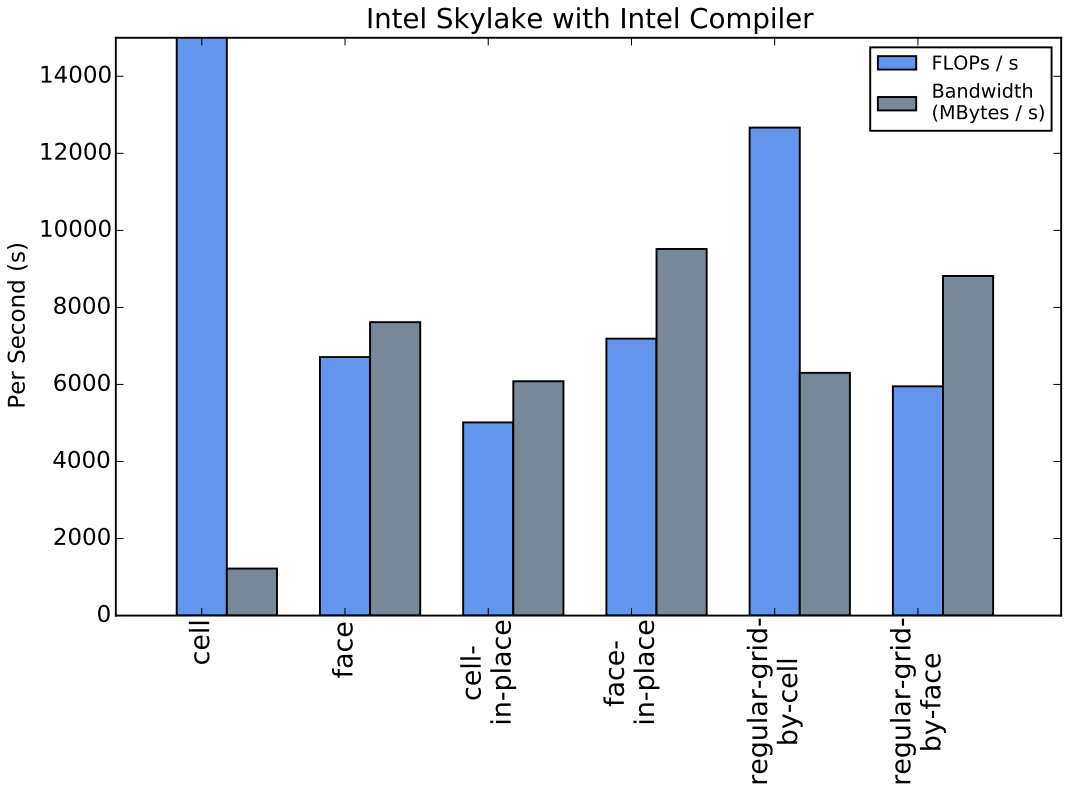

Figure 2.9. FLOPs and Bandwidth Rates for Skylake processor across CLAMR computational methods using the Intel compiler.

port. There are no intra-loop dependencies and most of the conditionals have been removed. The removed conditionals dealt with refinement checks of neighbors, but with a regular grid view, the refinement is known. For example, in Figure 2.12 for the cells to the right of the center cell, c, need references such as *nrht* which gets the lower cell to the right. Then *ntop* gets the cell above. We do this again for the next cells to the right to get a stencil width of two cells on the right side. Each reference requires a conditional test to determine whether there are one or two cells on that side. It isn't hard to see that mistakes in the logic are easy to make. The phantom cell version to the right of the figure guarantees that the neighbors are at the same level, greatly simplifying the logic. To see how straightforward the porting process is, we can take a look at the CPU code in Listing 1. The initial conversion to GPU code is mostly done by stripping off the *for* loop and replacing

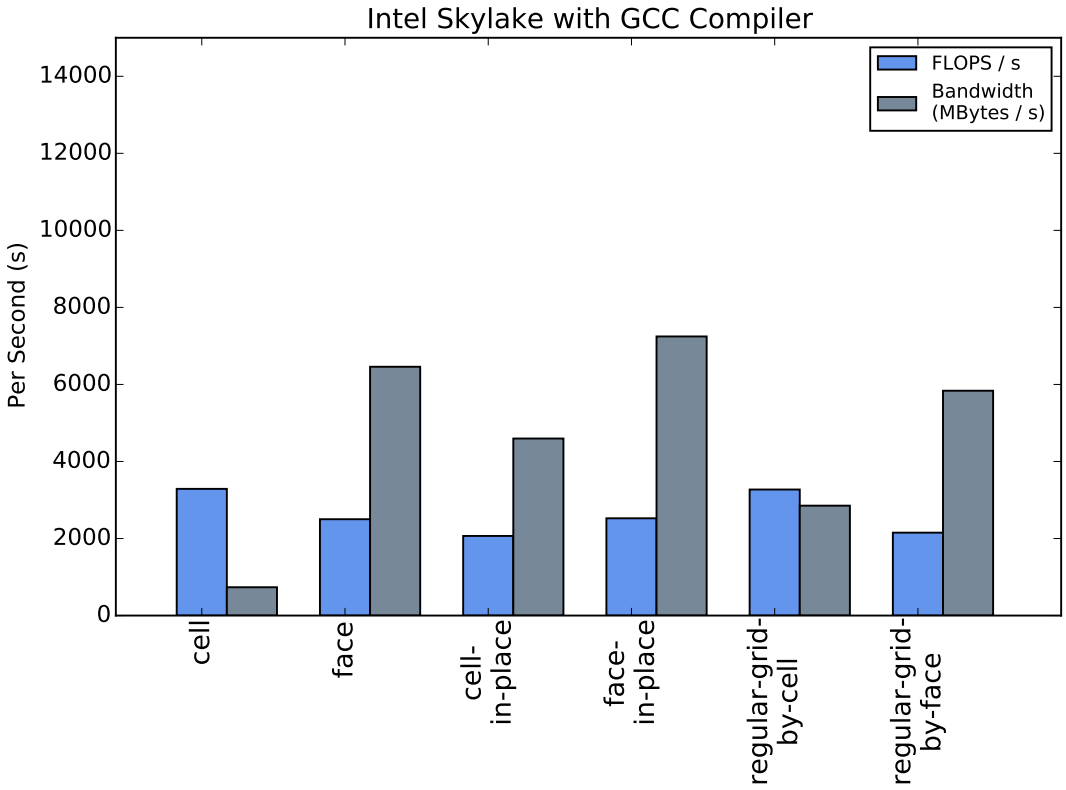

Figure 2.10. FLOPs and Bandwidth Rates for Skylake processor across CLAMR computational methods using the GCC compiler.

it with a thread index as shown in Listing 2. There are some other minor changes that are mostly due to keeping the two versions in sync.

We can speed up the physics kernels further by utilizing local memory. We have not yet done this for any but the original cell method because this changes the code and requires substantially more work. However, these changes have a clear implementation path. The difficult code to optimize on the GPU is in the mesh operations. The cell comparisons and mesh refinement logic require a lot of pre-counts because the serial version assumes that cells earlier in a loop have calculated their neighbors, and thus later cells piggyback on those neighbor calculations. This assumption cannot be made with parallel threads and as such starting indices of neighbors have to be calculated with a prefix scan operation so that when the threads are executed, they will assign their neighbor indices with

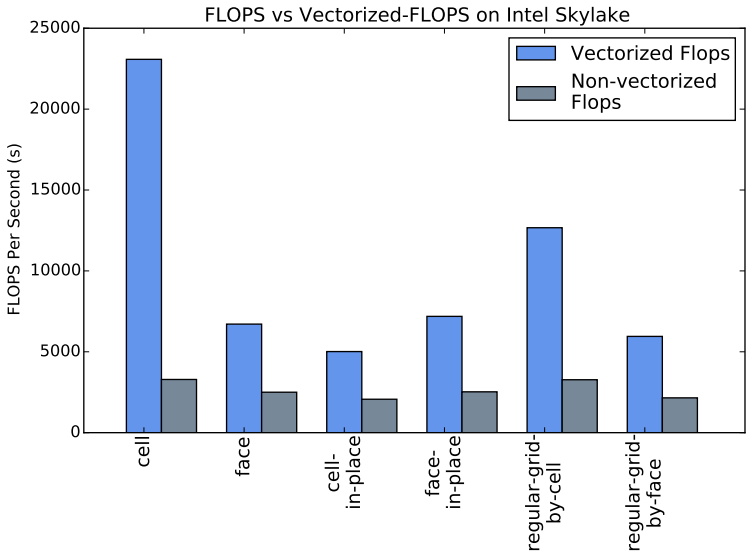

Figure 2.11. The vectorization by the Intel compiler greatly increases the floating point computation rate for all of the CLAMR computational methods.

the correct value in the global sequence. These types of complexities within the mesh handling are definite areas for future optimization, but they also illustrate why it is important to separate it from the physics routines. This is also the great advantage of writing this mesh code once and using it for many applications.

2.4.5 Performance Metric

To assess the performance of the methods, we collect three separate types of performance data: whole-code timers, routine-specific timers within the code, and hardware performance event counter information collected on the whole code (separated by individual routines). While the implementation of the timers is relatively straightforward, the collection and use of the hardware performance counter information deserves a few additional comments. For a test of each method, we collected several hundred different performance counters, by multiplexing across the collected set during a single long test for each code. The multiplexed collection is accomplished by a modified `mpirun` wrapper based on the Linux `perf` command. This allows us to non-intrusively collect data without requiring code modifications to the tested software units. Once the data is collected, it is post-

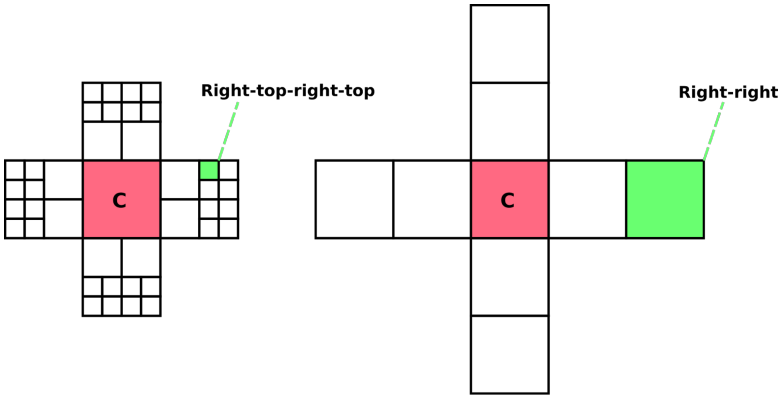

Figure 2.12. Neighbors for the physics in the phantom-cell are always at the same level as the cell being calculated like the mesh on the right. In the original AMR approach, the physics has to consider the possibility of refinement on each face, causing the code to have many conditionals.

processed through python scripts for eventual ingestion into a spreadsheet where it can be analyzed and further used to develop *signature vectors* representing each application. While the signature method is resilient against systemic biases or errors in the implementation of the hardware event counters, other portions of the analysis rely on basic validity of the event counters. Thus, the first task after ingestion is cross-validation of the counter-data, to catch instances in which the selected event counters can be unreliable or produce unexpected results. Because we collect a large set of event counters, we have the opportunity to cross-check the data being analyzed for self-consistency, based on knowledge of the codes, the processor, and the counters themselves. While this process is slow and time consuming, it need be performed only once for a processor model to determine for which counters there are issues. Once a set of valid event counters is obtained, we can convert the event counter values into normalized coordinates in a high-dimensional vector space, essentially using this vector as a quantified signature for each application tested (Kuehn, 2019). Once a vector for each application is

	cell	face	cell-in-place	face-in-place	regular-grid	regular-grid-by-faces
cell	1.000	0.996	0.983	0.988	0.961	0.981
face	0.996	1.000	0.993	0.995	0.977	0.974
cell-in-place	0.983	0.993	1.000	0.996	0.994	0.949
face-in-place	0.988	0.995	0.996	1.000	0.989	0.963
regular-grid	0.961	0.977	0.994	0.989	1.000	0.925
regular-grid-by-faces	0.981	0.974	0.949	0.963	0.925	1.000
sum	3.913	3.938	3.940	3.948	3.908	3.837

Figure 2.13. Cosine similarity of the methods on the CPU.

constructed, we can characterize the distance between two vectors in this space by calculating the angle (or cosine of the angle) between the vectors. This angle can then be used to indicate the similarity between application signatures. The angle can also be used to elucidate which counters of a large set are most important for a comparison between two or more applications. Analyses can then proceed using a small subset of less than 20%, focusing on the key event counters rather than the full set.

Our run-time results feature runs from two different GPU vendors, Nvidia and AMD. Nvidia runs operated through Cuda version 10.1 and AMD run used ROCM version 2.5. Both GPU tests were compiled with GCC 9.2.0 and OpenCL 2.0. Due to struggles with `nvprof` and limitations of Nvidia tools with current versions of OpenCL, we were only able to gather hardware counters from AMD's `rcprof`. While we plan to gather performance data from other GPUs in the future, the AMD data should provide us with a starting point to understand GPU performance characteristics.

	cell	face	cell-in-place	face-in-place	regular-grid	regular-grid-by-faces
cell	0.00	4.91	10.69	8.76	16.07	11.05
face	4.91	0.00	6.97	5.67	12.43	13.09
cell-in-place	10.69	6.97	0.00	4.88	6.27	18.29
face-in-place	8.76	5.67	4.88	0.00	8.34	15.70
regular-grid	16.07	12.43	6.27	8.34	0.00	22.37
regular-grid-by-faces	11.05	13.09	18.29	15.70	22.37	0.00
sum	51.47	43.06	47.10	43.35	65.49	80.50

Figure 2.14. Cosine similarity of the methods on the CPU, in terms of degrees.

2.5 Results

In this work, we want to understand the potential performance differences in the six different methods across a broad range of architectures. The six methods are: cell, face, cell in-place, face in-place, regular grid, regular grid by faces. The first two are the original cell-based AMR methods. The in-place methods build upon and simplify the original methods leveraging phantom cells. The regular grid methods use phantom cells to create separate regular grids at each level of refinement. We know some of the properties of each method. For example:

1. The face-based methods eliminate the redundant calculation of the fluxes at each face, reducing the floating point work by an estimated 40%.

2. The reduction in computations comes at the cost of additional physics state arrays at each face. These additional arrays and the mapping arrays from face-to-cell and back will increase memory by 40%.

3. The four phantom-cell methods eliminate 80% of the conditionals from the

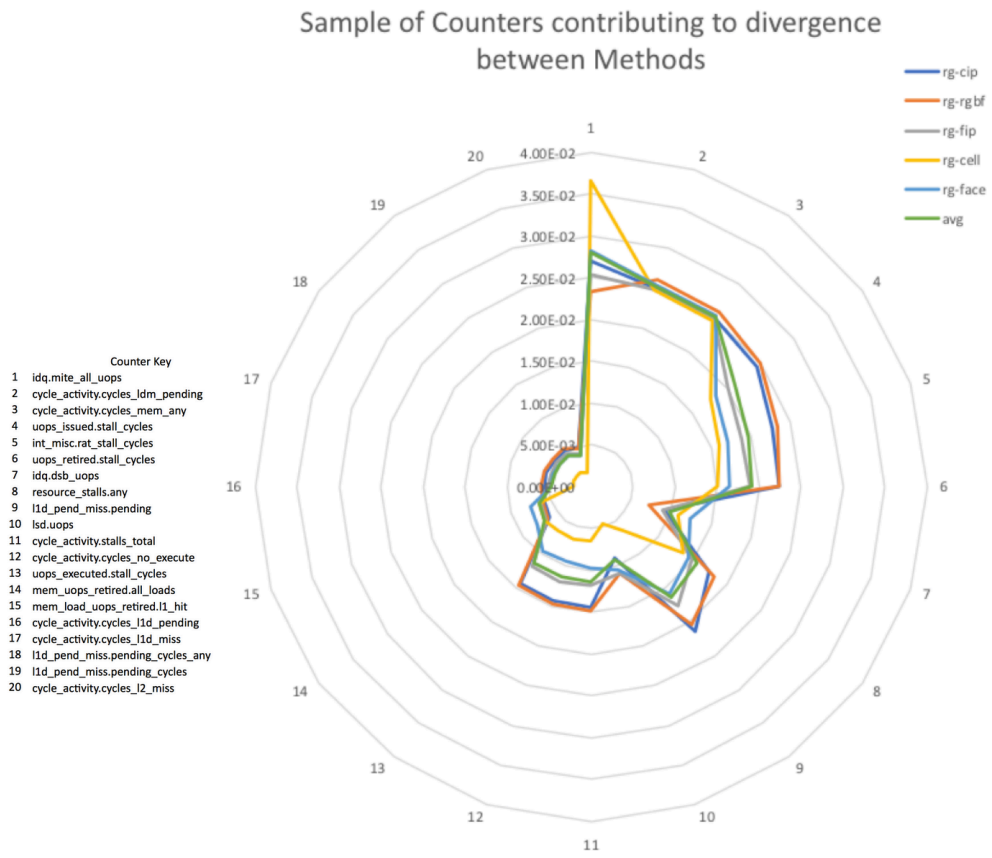

Figure 2.15. Subset of counters possibly contributing to divergence between CPU implemented methods.

original cell-based AMR.

Given these improved properties, we had questions that we wanted to answer which would help identify the pay-off of the improvements. The questions are as follows:

1. Which method performs the best across all of the computing hardware and why?

2. How does each method stress the computing hardware and what would improve its performance?

	cell	cell in-place	face	face in-place	regular-grid	regular-grid-by-faces
cell	1.0000	0.8871	0.9492	0.9596	0.8847	0.9206
cell in-place	0.8871	1.0000	0.9260	0.9520	0.9997	0.7961
face	0.9492	0.9260	1.0000	0.9961	0.9178	0.9653
face in-place	0.9596	0.9520	0.9961	1.0000	0.9460	0.9426
regular-grid	0.8847	0.9997	0.9178	0.9460	1.0000	0.7837
regular-grid-by-faces	0.9206	0.7961	0.9653	0.9426	0.7837	1.0000

Figure 2.16. Cosine similarity of the methods on the GPU.

3. Does the simplification of the physics kernel enable more compiler optimizations?

4. How does the performance of GPU accelerators vary with these different method characteristics?

We note that the code for each of the methods for each device is not heavily optimized (with the exception of the original cell method). Indeed, one of the questions is which kernel has the most promising attributes and should receive the most attention for optimization.

	cell	cell in-place	face	face in-place	regular-grid	regular-grid-by-faces
cell	0.00	27.48	18.35	16.34	27.79	22.99
cell in-place	27.48	0.00	22.17	17.82	1.51	37.24
face	18.35	22.17	0.00	5.05	23.39	15.14
face in-place	16.34	17.82	5.05	0.00	18.92	19.51
regular-grid	27.79	1.51	23.39	18.92	0.00	38.40
regular-grid-by-faces	22.99	37.24	15.14	19.51	38.40	0.00

Figure 2.17. Cosine similarity of the methods on the GPU, in terms of degrees.

2.5.1 Insights from Hardware Counters

A comparison of similarity provides the best analysis tool to explore the relative difference across all methods. Figures 2.13 and 2.14 show these similarities for the CPU implementations of the methods. These are standard direct comparison matrices, with method signatures compared in each cell. Of course, a similarity of 1.0 or angle difference of 0.0 appears along the diagonal when comparing a method to itself. The regular grid by faces method is the most unique of all methods, showing the largest difference in signatures.

Figure 2.15 shows an example set of counters to investigate when looking at divergence between methods. This subset of counters is the top 20 in terms of discrepancies between the methods, and can be used as a starting base for investi-

Run	Time	V/S ALU Inst	Scalar ALU/fetch Inst	Vector ALU/Mem Inst	Vector ALU Utilization	V/S ALU Busy	FetchSize	WriteSize	L1CacheHit	L2CacheHit	MemUnit Busy	MemUnit Stalled	WriteUnit Stalled	LDS Bank Conflict
cell	1.48	6.64	37.45	32.71	95.59	5.44	23417	12455	67.12	43.22	15.01	0.21	0.01	2.45
face	0.80	12.20	2.86	9.25	98.13	8.35	59837	28164	72.14	36.86	64.32	6.05	0.15	0.00
cell in-place	1.22	48.33	1.41	13.29	99.32	30.27	54097	59704	62.90	29.19	57.41	18.58	0.29	0.00
face in-place	0.74	15.01	3.20	10.36	99.37	10.25	49989	28892	70.60	32.63	60.59	12.83	0.14	0.00
regular-grid	1.16	27.35	2.80	8.88	98.54	19.18	49538	57355	71.09	39.88	75.43	24.46	0.29	0.00
regular-grid-by-faces	0.42	13.02	1.58	4.80	97.19	2.65	55312	9455	66.19	24.13	75.96	29.91	0.01	0.00

Figure 2.18. Hardware counters for the GPU.

gating what each method does well, and what needs to be optimized. Particularly note counters 2 and 3, showing that all of the methods are similarly sensitive to memory performance, spending much of their time awaiting the completion of memory references. The similarity of these counters can be used as a sanity check. For comparison, consider counters 4, 5, and 6, which show each method varies in its sensitivity to stalls on the front-end of the processor, spanning the issue of instructions and allocation/release of resources. Finally, examine counters 18 and 19, which demonstrate that for each of the methods, the time spent waiting on L1 data cache loads is a relatively smaller proportion, suggesting the cache utilization is already fairly efficient. Figures 2.16 and 2.17 show corresponding data for the GPU implementations. Following the CPU trend, we find that the regular grid by faces method is the most unique among the GPU implementations.

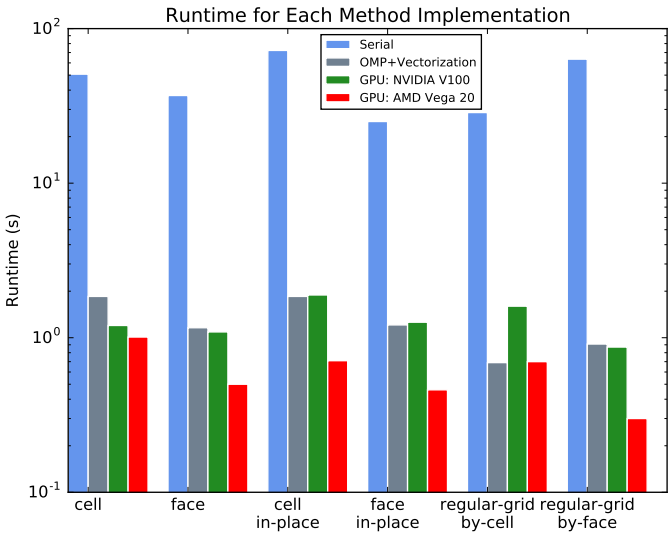

Figure 2.19. Run-times for all method implementations.

Figure 2.19 shows run-times for all method implementations. These runs were done with a 1024 x 1024 starting mesh and run for 500 cycles. Although our OpenMP implementations are slightly out of scope for this paper, we include them as a comparison to the GPU accelerators. The run-times appear as expected, with GPU performance being the best. One note is the performance difference between the AMD Vega and the Nvidia V100 for some methods. This could be due to the fact that these computations are significantly memory bound, as opposed to FLOP bound, giving the Vega the edge on this particular problem. The V100 can produce significantly more FLOPs per second, but the memory clock speed of the Vega is significantly higher. This is only a speculation, but draws attention to something that may be investigated with further work. Figure 2.18 shows a set of counters to look at when analyzing the performance of the GPU. Note the relatively poor ratios of VALU-to-SALU instructions and VALU-to-SALU Busy, as well as a the relatively smaller fetch and write sizes, resulting in low memory unit utilization. Faster methods include face, face-in-place, and regular-grid-by-faces. The later runs the fastest, likely because of the reduction of overall operation count as measured on the CPU, but note the combinations of higher VALU utilization, better ratio of vector to scalar instructions, larger fetch/write

sizes, better cache utilization, and better use of the memory unit. None of the methods needed to be best on all of these, rather the most successful methods pushed all of them into a more reasonable range. It is unclear at this time is exactly the cause of discrepancies in the counters for each method, and we leave a more in depth study for future work. However, we speculate on some differences that we can see at hand. Vectorization for instance, is affected by control flow and dependence within a loop. By removing almost all logic checks from the original methods in the in-place methods, we allow the compiler to vectorize more easily or more completely. This is a high level speculation, but it gives thought to kind of conclusions we can draw from these counters. As an example, the regular-grid-by-faces method is one of the lower performing serial versions, but its GPU performance is among the best. More in depth examination needs to be done to determine possible reasons for this, but this shows that performance on different compute devices may not exhibit the same relative ranking, thus requiring some device specific measurement and analysis.

2.5.2 Discussion

Chapter 3

Wildfire Simulation

Wildfire simulations have been of increasing interest as of late, but their complexity makes creating models difficult. Often multiple models are required to capture aspect groups. HIGRAD (high gradient flow solver) is an atmospheric hydrodynamic model developed at LANL which has evolved to study cloud physics and nonlinear solvers. Coupled with FIRETEC, its goal is to be a representative model of wildfire behavior. FIESTA (Fast Interfaces and Transport in the Atmosphere) is a C++ implementation of the HIGRAD model. The motivation for this was to incorporate C++ technologies, such as Kokkos, into the original model to increase lifetime as well as promote easier coupling to other current research projects. In addition to a strict portage protocol, FIESTA also aims to serve as a test-bed for different numerical approaches for the hydrodynamic model. Because of the coupled usage of HIGRAD and FIRETEC, throughout the remainder of the chapter I will refer to this code as only HIGRAD. Furthermore, the research presented in this section is through FIESTA (Fast Interface Evolution Shocks and Transition in the Atmosphere) specifically, so unless explicitly stated, the code being referenced will be the FIESTA code.

3.1 Governing Equations

The code work presented is all done in the FIESTA code. However, the main objectives and extended research on projects are concurrent with the models present in the HIGRAD and FIRETEC models. For this reason it is pertinent to discuss the equations governing the models. Since HIGRAD and furthermore FIESTA are built off of the core of FIRETEC, only the equations for this code will be discussed.

3.1.1 FIRETEC

FIRETEC aims to capture the complex chemical reactions occurring in a wildfire. The reactions involve a high number of variables because of the numerous intermediate short lived reactions. To simplify matters, all of the chemical reactions are reduced to a smaller set of reactions some solid–gas and gas–gas reactions. Additionally, wood pyrolysis is used and the reaction is simplified further to a single reaction, shown in Equation 3.1

$$N_{\text{wood}}(wood) + N_{0_2}(oxygen) \rightarrow (products) + (heat) \tag{3.1}$$

The coefficients N_{wood} and N_{O2} used here are stoichiometric and describe the net amount of wood and oxygen used as a result of pyrolysis. The reaction rate for this model is shown in Equation 3.2

$$F_{\text{wood}} = \rho_{\text{wood}} \rho_{O_2} \sigma \Pi \tag{3.2}$$

F_{wood} represents the rate of change of wood within a resolved volume. ρ_{wood} and ρ_{02} are the density of wood and oxygen, respectively, within that resolved volume. Here, σ represents the turbulent diffusion coefficient. Also of note, the coefficient Π is actually also a function of stoichiometric coefficients, probability distribution for temperature, and relative densities of reactants. The idea behind this specific model is that the rate of pyrolysis is related to the heat flux of the wood. If the wood or oxygen is not present then the chain is broken. As with all models, conservation is a necessary evaluation. Equations 3.3 and 3.4 describe the conservation of the average masses of wood and moisture (water).

$$\frac{\partial \rho_{\text{wood}}}{\partial t} = -N_{\text{wood}} F_{\text{wood}} \tag{3.3}$$

$$\frac{\partial \rho_{\text{water}}}{\partial t} = -F_{\text{water}} \tag{3.4}$$

Additionally, the temperature represented by Equation 3.5, which is thought

of as evolution of internal energy of the solid.

$$\mathbf{P}\frac{\partial T_s}{\partial t} = Q_{\text{rad}} + ha_v(T_{\text{gas}} - T_s) + \mathbf{F} \tag{3.5}$$

where

$$\mathbf{P} = (c_{p_{\text{water}}}\rho_{\text{water}} + c_{p_{\text{wood}}}\rho_{\text{wood}})$$

and

$$\mathbf{F} = -F_{\text{water}}(H_{\text{water}} + c_{p_{\text{water}}}T_{\text{boil}}) + F_{\text{wood}}(\Theta H_{\text{wood}} - c_{p_{\text{wood}}}T_{\text{wood}}N_{\text{wood}})$$

This equation has several redundant terms, so for the sake of simplicity I will only define each variable type once. $c_{p_{water}}$ represents the isobaric heat capacity of water. $\frac{\partial T_s}{\partial t}$ is an expression of the change in average temperature for the solid components within the volume with respect to time. Q_{rad} is the net thermal radiation heat flux to the solid at the specific location. ha_v is the contact area, per unit volume, between the gas and the solid, which contains the convective heat exchange coefficient. $T_{gas,s}$ is the average temperature of the gas and solids, respectively, within the volume. Finally, Θ expresses the fraction of heat being released from the gas phase combustion that is *deposited directly* back to the solid phase. This equation can be seen to describe the increasing change in the internal energy of the solid based on the temperature and densities of the wood and water.

The density of the gas is calculated using Equation 3.6 and conservation of momentum is shown in Equation 3.7

$$\frac{\partial \rho_{\text{gas}}}{\partial t} + \frac{\partial u_i \rho_{\text{gas}}}{\partial x_i} = F_{\text{wood}}N_{\text{wood}} + F_{\text{water}} \tag{3.6}$$

$$\frac{\partial \rho_{\text{gas}} u_i}{\partial t} + \frac{\partial u_i u_j \rho_{\text{gas}}}{\partial x_j} = -\frac{\partial R_{ij}}{\partial x_j} - \frac{\partial p}{\partial x_i} + p_{\text{gas}} g_i - \frac{3 C_D \rho_{\text{wood}} \rho_{\text{gas}} |u| u_i}{8 s_s \rho_{\text{wood(micro)}}} \tag{3.7}$$

These equations are largely based on the evolution of the components, often within the density variable and with respect to time and space. $x_{i,j}$ are the spatial units and g_i is the acceleration due to gravity (in the i direction). R_{ij} represents the Reynolds stress tensor while C_D simply represents the drag coefficient. Here, u is the average velocity of the combined gases (at a given location) and s_s is the size scale representing the average size of each of the fuel elements.

Finally, the evolution of potential temperature of the gas is shown in Equation 3.8. It uses the advection, turbulent diffusion, as well as several source terms.

$$\frac{\partial \rho_{\text{gas}} \theta}{\partial t} + \frac{\partial \theta u_j \rho_{\text{gas}}}{\partial x_j} = \frac{\partial}{\partial x_j}\left(\sigma \frac{\partial \theta}{\partial x_j}\right) + \frac{\theta}{c_p} \left[\frac{h a_v (T_s - T_{\text{gas}}) + Q_{\text{rad,gas}} + F_{\text{wood}}(1-\Theta) H_{\text{wood}}}{T_{\text{gas}}}\right] \tag{3.8}$$

The only new terms in this equation are θ, which is the average potential temperature of the combined gas at a given location, and $Q_{rad,gas}$. We defined Q_{rad} previously, but this new term involving gas represents the net gain of energy by the gas resulting from thermal radiation. It is calculated based on a two-field thermal radiation scheme. We can see that energy is emitted and absorbed. Emitted energy is diffused through the gas material and is absorbed by the solid objects. It should be noted that this transport formulation is not as accurate as other discrete transport methods. However, it is importantly less computationally expensive.

3.1.2 HIGRAD

3.1.3 FIESTA

3.2 Porting to AMR

A huge challenge for the simulations that are being run is the scale of detail we need. A lot of the high resolution physics has to be approximated or estimated because the grid can not be a small enough scale. For example, the current wildfire simulations in 2D would hope to be run on a 200 kilometer by 200 kilometer regular grid. Even with a pretty poor detail, where every cell is 10 meters by 10 meters, would result in a grid with $4x10^8$ cells. If each cell only carries four doubles of data, we are already needing 3.2 Gigabytes of data. This is already a sizeable simulation, and even then it might not capture all off the detail necessary. If we were to refine the cells further, allow for more data on each cell, and extend the problem to three dimensions, its easy to see that the problem quickly gets out of hand in terms. In fact, particle calculations cannot be feasibly done on this scale, and they are one of the important models to follow in combustion simulation. Adaptive mesh refinement can greatly help in this realm. In fact, wildfire models such as the ones presented here are perfect candidates for the level of AMR that I have done throughout these projects. It is a sizeable problem in which only a few specific areas of interest exist.

3.2.1 Feasibility

The reason many codes stay in regular grid form is because of the effort it takes to port them to an adaptive mesh form. Calculations that loop through the mesh are simple on a regular grid because the indexes are simple spatial *i,j,k* indexes. With the addition of adaptive grid properties, all of the loop calculations and loops must be rewritten to accommodate the newly formed grid. Neighbors can no longer be accessed spatially because cells can be of varying sizes. Furthermore, the grid is now dynamic at each step. A regular grid has a static structure and maintains its shape throughout the entire computation. An adaptive grid always has the possibility of changing. Accessing cells needs to be done with a scheme utilizing unique cell IDs. In addition, physics methods and routines must

be rewritten to accommodate cells of varying sizes and cells with more than one neighbor on a given side.

In fact, all of the detail mentioned above is under the assumption that the developers had access to a plug and play AMR library. These libraries are not common, and they are rarely so simple to attach to an existing code. In that case it is the computational scientists job to write these AMR routines. They are not trivial, and often take years of dedicated effort to write efficient algorithms to handle all the components of adaptive meshes. For this reason it is helpful that we have access to a code like CLAMR. The complex algorithms exist (obviously due to previous research and efforts) with the benefit of having been designed with portability in mind.

3.2.2 Preliminary Steps

The first efforts in converting a regular grid code to AMR is to change the indexing scheme from spatial `i, j, k` indexes to a singular index based on cell ID. The easiest first step for this is to change array dimensions from 3 dimensional to 1 dimensional and make the size be $\texttt{size}_\texttt{i}$ `*` $\texttt{size}_\texttt{j}$ `*` $\texttt{size}_\texttt{k}$. Naturally, any loops accompanying these arrays that loop through the spatial indexes will need to have the loop bounds converted in a similar fashion. As a first pass, this can as trivial as looping through `i, j, k` indexes, but converting these into a single index where

$$Index = k + j * size_k + i * size_j * size_k \tag{3.9}$$

so that a single unique index is created from the dimensional spatial indexes. Assuming this works as intended, the next step would be to simple change the nested loops to a single loop with the array size as the range and omit the index conversion.

The next step is simple, but an easy area to mess up the relationships, especially for more complicated functions. Accessing neighbors must now be done using left, right, etc. arrays as opposed to an incremental change in a spatial

coordinate. This is trivial for immediate neighbors, and naturally gets slightly more complex when having to call neighbors of neighbors. This can be trivial in many cases, but when neighbors are accessed via a loop special care has to be take to ensure the neighbors are being indexed correctly. For example, examine the regular grid loop in code Listing **??** below:

```
int index = my_index;
for (int shift = 1; shift <= right_neighbors; shift++) {
    state[index] = state[index] + state[index + shift];
}
```

When converting this to use neighbor relationship arrays, we cannot do a single substitute. At each iteration of the loop the change in neighbor goes one step further, and we need to be more clever about how we substitute our changes in. In this case, we have to change the 'base' index at each iteration so that it points to our previous neighbor, allowing us to keep the code the same throughout the loop. An example of this is shown in Listing **??**:

```
int index = my_index;
int start_idx = index;    // starting index; won't change
for (int shift = 0;       // shift now will start at 0
shift < right_neighbors; // change shift bound
shift++)                  // increment remains the same
{
    state[start_idx] = state[start_idx] +
                       state[neighbor_right[index]];
    index = neighbor_right[index]; // new index of neighbor
}
```

There are two main differences in this loop. First, our index we 'shift' is shifted by using the neighbor lists. The for loop changes from being an incremental shift amount to simply being a known-bounded loop. Second, in a related manner

we now must keep track of our main target index because we still want to be doing computation on that state value.

Changing the converting indexing from multi-dimensional to a single dimension system is a tedious task of porting regular grid applications. However, it is an important step because it simplifies the code and makes future modifications much easier. As we see in future sections, this conversion also makes optimization for efficient and easier for a user to implement. Parallel tasks work only on a single loop, and users can work through the problem with more direct relationships between cells. For example, when working on a physics state calculation, it can be mentally consuming thinking about cell neighbor relationships in terms of `i, j, k`. One has to calculate spatially what these indexes mean, and how they relate to a cell. By using specific relationship data structures (e.g. right neighbor, top neighbor, etc.) it is easier to spatially quantify where that cell resides. Of course this is a small gain in the broader picture, but it certainly helps with more complex algorithms and more in-depth levels of loop optimization.

After changing the index structure of the application, we have to run a full test suite to ensure that the results remain the same. It is obvious that after any change we should be making sure our results are correct. However, there are large code design shifts that are prone to disrupting the results, and since this is one of those situations, it is crucial that extra effort is made at this step (and others like it) to retain validity of results.

3.2.3 Addition of AMR

Once the loops have been converted to use neighbor relationships, we are ready to incorporate adaptive mesh refinement into the application. This project made use of the CLAMR application we mentioned in the first chapter. The huge benefit of using CLAMR for this task is the portability it brings for applications using the mesh refinement routines. The library can be included in the build system, and the only other changes necessary to the code at this point are to add the corresponding routines before each time step calculation. CLAMR uses the neighbor data structures to handle all mesh refinement routines, so they can be

passed back and forth between the two applications without compromising any functionality. The FIESTA application handles the physics computation aspect while CLAMR works on the underlying mesh.

The key aspect of CLAMR that allows for the easy portability of applications is the addition of phantom-cell AMR. As mentioned, a major roadblock for applications trying to use adaptive mesh refinement is incorporating mesh handling into their physics routines. Algorithms have to be rewritten to work under the premise that there can be more than one neighbor on either side, breaking the assumption previously held on a regular grid. Phantom cells in CLAMR pull out this mesh complexity from the physics routines and keep it integrated with other mesh routines. As such, mesh functions are now responsible for the complexity of the mesh, not the physics routines. AMR can be added to the state calculations without needed knowledge of how the other component operates. This removes the biggest source of work when incorporating AMR in an existing application.

The other difference when incorporating AMR into an existing regular grid applications is checking for certain groups of cells, for instance boundary cells. With a regular grid application the appropriate cells can be identified by their spatial positioning, for instance the cells at index 0 and `row_size - 1` can be easily identified as boundary cells. Any other grouping can be labeled with the appropriate coordinates. However when using a single one dimensional array, this is clearly not an option. Cells have to be identified by who their neighbors are. Often (and in our case the logic in CLAMR), cells which have no neighbors on a particular side are boundary cells. Furthermore, cells with two missing neighbors are corners (unless of course we are working in three dimensions, then they are edges). For a regular grid routine, loops can have their boundaries set to the specific spatial indexes, depending on what types of cells are supposed to be included in the calculation.

It should be noted that the implementation of phantom-cell AMR is not fully complete withing the FIESTA code. As will be discussed in the next section, FIESTA uses MPI as one framework for parallelization. Phantom-cell AMR has been tested on a range of CPU and GPU architectures, each with their own implementation. However, the MPI version of the routine is still in progress and

undergoing bug finding. As such, the current AMR implemented in FIESTA is less accurate than is expected once we have fully implemented CLAMR's phantom-cell AMR. For the time being, cells request the state information from their neighbors. However, with the present state these neighbors are not phantom cells, they are simply a single neighbor of the possible two neighbors for that particular side. For many cells this does not change the results, however for cells that would normally have multiple neighbors in one direction this creates some inaccuracies at this stage. Once CLAMR has the bug(s) found, the routines within FIESTA will not actually need to be changed as their code is "correct." The results will be accurate at that time because the underlying mesh handling will be correct for this implementation.

3.3 Parallelization

As with all similar applications, performance is a high priority for producing results. As such, two parallel frameworks have been employed to assist the process. The core one is Kokkos. Kokkos is helpful for allowing codes to be run in parallel on the CPU or, especially, the GPU, without compromising portability. For FIESTA, the focus of Kokkos is to port the code to modern GPU architectures. Most of the loops are simple for-loops that can be easily converted to Kokkos parallel loops. However, a main challenge is that CLAMR does not currently offer support for Kokkos. Several different parallel frameworks are supported, but Kokkos has not been implemented for algorithmic purposes (although it has been incorporated into the build system). As such, for the initial implementation of using CLAMR withing FIESTA, a sort of bridge has to be made between the two applications. For most cases this is as simple as creating a double set of data structures to transition from Kokkos data to mesh handling data structures. Unfortunately, at this stage this transition adds a considerable overhead to the overall calculation. However, this overhead is masked by the efficiency and accuracy that is brought as a part of the included AMR. With a normal regular grid application, to get the refinement scale needed for some calculations the entire grid size must be increase several orders of magnitude, even in areas that are largely computaionally unimportant. Adaptive refinement allows the overall grid

size to be smaller, so even with added routines and additional data copies, the overall runtime of the code is improved because of the drastic reduction in data and computation. This is an area where future work is definitely planned out to include Kokkos within CLAMR. If the data structures in CLAMR were already set to be targeting Kokkos, they could be accessed directly between the two applications. There would be no need to do any sort of deep copy to and from the GPU twice, once for the mesh handling to the FIESTA calculation and once the other direction. In fact, have Kokkos support within CLAMR would further the performance gains because those routines can be run on the GPU with these same data structures being used in FIESTA. The improvement is two fold, removing deep copies between the two applications and their execution spaces, as well as speeding up the mesh handling itself within CLAMR.

A more difficult adaptation was incorporating MPI into FIESTA as part of the transition to using AMR. FIESTA actually currently runs with MPI used in combination with Kokkos. Cross-node and multi-GPU parallelism has been tested and the performance was found to be very good. CLAMR itself has several MPI implementations, so the hope was that the coupling of the two applications would be quite simple. Unfortunately, this was not the case because the MPI implementations in CLAMR currently reside in the base cell-based method. Methods involving phantom-cell AMR still have not produced the desired results for all boundary cases when using MPI. The implementations are close, but without full confidence, we can not use these phantom-cell methods within FIESTA as part of the transition to AMR. As mentioned previously, this is why the computation has slight inaccuracies, because of the incorrect state value that is calculated from all neighbors. Working on the full implementation of MPI in CLAMR has been moved to a priority, because of the necessity for this transition. Adequate results can be produced at the current level of work. However, with the addition of AMR we will be looking to improve accuracy and optimization. The optimization portion has been achieved, but the accuracy needs to be maintained at the same level.

3.3.1 Visual Validation

Of course the work presented in the chapter is not exposing a new numerical model. The purpose of the research is to show how the phantom-cell AMR implemented in CLAMR can easily be used to drastically reduce the amount of work in porting regular grid applications to adaptive mesh applications. The main validation for this work is using real-time graphics to verify that the mesh is doing what we think we have designed it to do. The original regular grid application in FIESTA has been mathematically validated, but when we introduce AMR, we change the mesh. There is no longer a one to one comparison with the original application on any level. Ideally we would compare the state values between the two grids, but AMR meshes do not match their regular grid counterpart. We have to be able to compare the application in a different way than a direct matching of values in cells. Real-time graphics are the key to doing this because even if state values and the number of cells are slightly different, we can still watch the application progress. If the flow of the application is matched between the two different methods, then we can say it accurately represents the model or simulation. In fact, even with the knowledge that the data at refinement level boundaries is not as precise as we would like, we still see from the graphics that the model acts identical to its original production. This shows the increased accuracy we can achieve with AMR, and promises even greater precision in the future.

For our real-time graphics we use OpenGL. There are several libraries that all would have been adequate for this type of showcase. However past work in a similar area had some related OpenGL functionality, so it was an easy choice moving forward. OpenGL works by taking receiving values for the edges of the display box, and then drawing the scene within those bounds. Mesh applications are perfect for this type of showing because the work is entirely shown with squares. Each level of refinement can have the square size specified, and each square is placed at the floating point precisioned spatial coordinates. CLAMR keeps track of these coordinates (for display measures such as this), so these values did not need any extra computation. Each cell passes in its state value, and OpenGL will color each cell based on that state value. This calculation can either be a static number block or a dynamic comparison function based on the other cells' state

values. The progression of our three level CLAMR mesh in FIESTA is show in Figure 3.1.

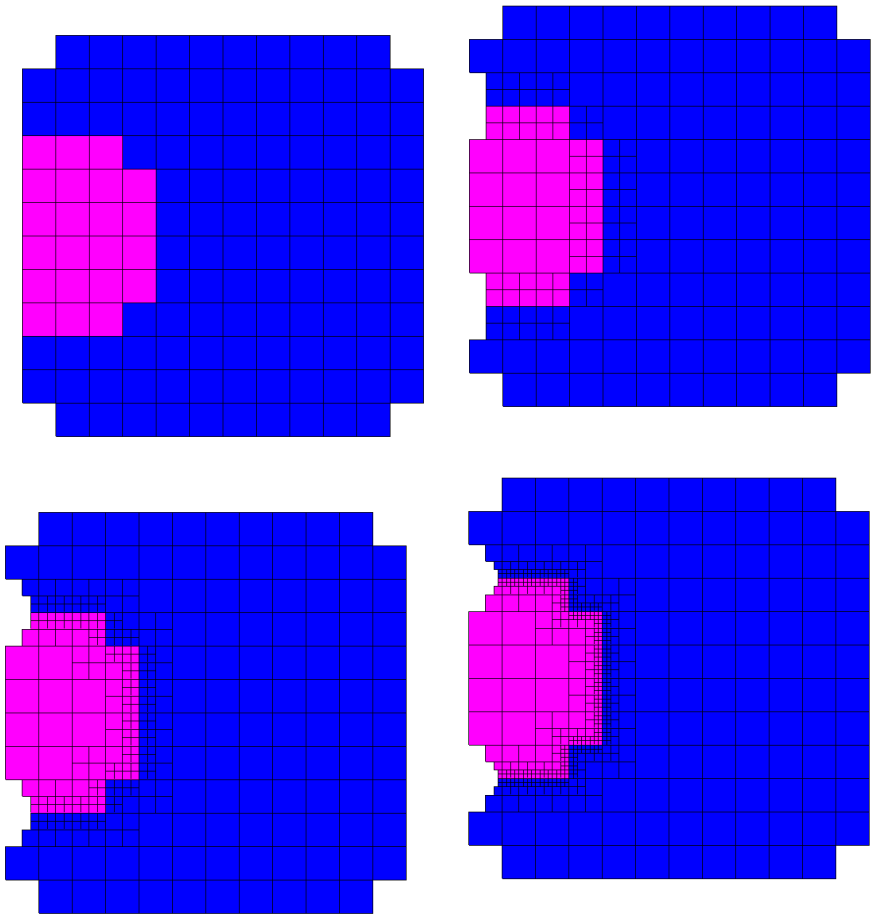

Figure 3.1. AMR progression for the the rising gas bubble problem in FIESTA. Notice the refinement occurring along the gas boundary where more precise information is needed for calculations.

3.3.2 Future Work

The future work for this application is quite straightforward. Many tests and problems need to be run to evaluate the precision of the AMR work in conjunction with the original state methods. The first problem of note will be a two dimensional grass fire problem to show how AMR will give greater precision at

areas of interest. From there, work has to be done to incorporate other physics/numerical methods into the code to accommodate the dimensionality (or lack there of) within CLAMR. This will transition into work with CLAMR, as mentioned, adding a third dimension component to the mesh routines so that more physics problems can be handled for a more thorough testing of the methods applied here.

Chapter 4

Matar

There are three main problems in code design in computational physics. They are: making the code easy to write and upkeep (e.g. making the programmer productive), making the code performant, and making the code portable across modern computer architectures. These problems all arise when working through the low-level data layout for a given code. To help simplify this process, we have developed a software library, MATAR, that provides the flexible data structures needed by scientific applications and delivers high performance code even within complex routines. In many of our applications, the layout of data is a major hurdle when it comes to the performance of the code. Sparse data and unstructured data are commonplace and add complexity to our data structures. Furthermore, common array data structure implementations, especially multi-dimensional, are often inadequate for achieving desired performance. MATAR looks to improve the performance of these data structures within a simple interface that is portable across multiple architectures.

4.1 Background

Interest in proper multi-dimensional array support has long existed in the C and C++ programming community. There has always been the capability for using the Standard Template Library (STL) vector of vectors to provide a two-dimensional array, but is not a performant method, nor does it scale reasonably to larger n-dimensional arrays much larger than two dimensions.

The Boost Multidimensional Array Library (Garcia, 2006) has provided a multi-dimensional array for C++ programmers since around 2002. The library provide a dense n-dimensional array with either C or Fortran ordering and support for subviews. Some of the design for the Boost library came from the well-known Blitz++ array library (Veldhuizen, 1998) that provides many powerful multi-dimensional array capabilities. The Matrix Template Library (Siek and Lumsdaine, 1998) develops a flexible multi-dimensional data structure for use in

matrix operations for numerical solvers.

The latest multi-dimensional array implementation to come to C++ is the mdspan proposal that has been accepted into the C++ standard (Hollman, Lelbach, Edwards, Hoemmen, Sunderland, and Trott, 2019). Mdspan originated as the multi-dimensional array object in the Kokkos performance portability library (Edwards, Trott, and Sunderland, 2014).

Many of these libraries cite their motivation being easier and more powerful to use in C++ applications and the better performance of contiguous memory implementations over the vector of vectors approach.

4.2 Design and Goals

One of the most difficult parts of code design for computational physics is deciding on how data will be laid out in memory, and how to make accessing that data efficient. Contiguous memory access is generally the most performant on any hardware since it allows for data streaming and vectorization, but contiguous memory is rarely the most efficient from the programmers perspective since the data structures for contiguous memory become more challenging as the complexity of the problem being solved increases, and the problems themselves are rarely set up to make contiguous memory access easy to achieve.

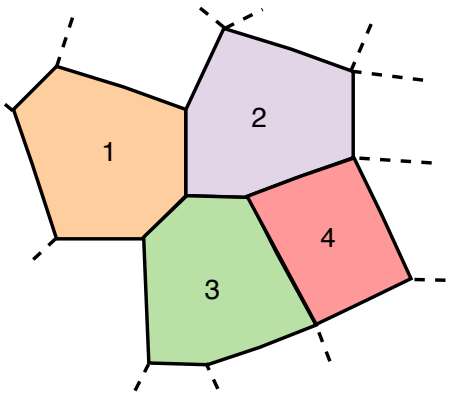

Figure 4.1. Polytopal meshes do not lend themselves to easily implementable contiguous connectivity structures since the amount of information required to represent the connectivity between each cell is not the same.

One example of where these problems arise is in methods that involve a polytopal meshes like the one discussed in (Lipnikov, Shashkov, and Svyatskiy, 2006). Polytopal meshes like the one shown in Fig. 4.1 are difficult to build since the connectivity structure is entirely based on the initial point cloud used to generate the cells and no symmetry or repeated structure can be used to simplify the mesh building process. For example, cell 1 in Fig. 4.1 has 6 nodes and 6 neighbors, cell 2 has 6 nodes and 9 neighbors, cell 3 has 7 nodes and 7 neighbors, and cell 4 has 4 nodes and 7 neighbors. A data structure to hold the information for this will necessarily be "ragged" since amount of information required to be stored for each cell is varied. There are few tools for building these meshes (Pouderoux, Charest, Kenamond, and Shashkov, 2017) because of the complexity in their construction, but they show great promise for meshing complex geometry.

For another example, it is often the case in Eulerian codes that multiple different materials exists inside each cell, and since the materials move through each cell the number of materials changes throughout the runtime of the code. A few methods that take this approach are (Barlow, Morgan, and Shashkov, 2019; **?**) and an example of this type of problem is shown in Fig. 4.2. To make these multi-material codes performant the information required needs to be laid out and accessed contiguously in memory. There are two main ways to lay out memory for these applications, cell centric or material centric. The performance of each of these is analyzed in (Garimella and Robey, 2017), but each still result in a ragged data layout.

Also, numerical methods are derived and designed using multi-dimensional matrices for the calculation, and most people think about these methods as object manipulation (e.g. matrix-matrix multiply) instead of direct data manipulation (e.g contiguous-access, fused multiply-add). The goal of MATAR is to build data structures that are easy to use and give the feeling of object manipulation, but lay out and access the data contiguously in memory so the end-user gets the performance benefits of data-oriented design with the productivity benefits of object-oriented design. But what do we mean by data-oriented design and the better known term, object-oriented design? We first review the definitions of the two terms.

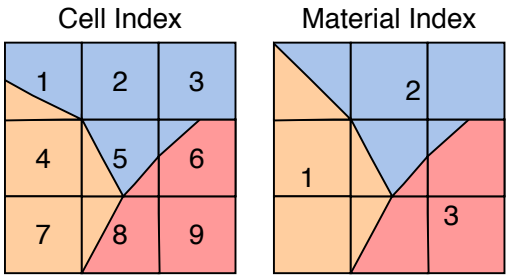

Figure 4.2. In some applications there can be multiple materials per cell, and for Eulerian or ALE methods these materials move through the cells. This results in a ragged memory layout to describe either the materials in a cell, or the cells which have a material depending on if your code is cell centric or material centric (Garimella and Robey, 2017).

4.2.1 Object-Oriented Programming and Data-Oriented Design

Introduced around the late 1980s, object-oriented programming (OOP) continues to be a prominent programming paradigm and has become a standard way of programming in many fields (Stroustrup, 1988). object-oriented programming solves problems by breaking down the problem into its constitutive parts and their relations. These objects are then manipulated through the process of solving a problem instead of manipulating all of the bits of information individually.

Listing 3 An example of Object-Oriented Programming. The object is the circle and the method manipulates traits of the circle object.

```
class Circle{
private:
  float x_, y_, r_;
public:

  // initializer
  void init(float x, float y){
      x_ = x;
      y_ = y;
  }

  // method
  void calc_radius(){
      r_ = sqrt(x_*x_ + y_*y_);
  }

  // access private variable r
  float radius(){return r_;}
};

//in main
Circle my_circles[100000];

//do calculation with my_circles
```

Although object-oriented programming is a great mental map for the relationship of objects and their capabilities, there are some performance limitations that are often overlooked. The most significant drawback of object-oriented programming is that it is typically not a scalable paradigm. For example, in Listing 3, we create a `Circle` class with some variables and functions that will be used. It

is then possible to create many circle objects. Having millions of `Circle` objects will lead to poor performance because of the sub-optimal layout of the data in memory. The non-contiguous memory layout for each variable, i.e. `x, y, radius` will lead to more memory loads for each object and cache misses if the data is not used sequentially (Robey and Zamora,). The memory loads become a bottleneck, making it more difficult to optimize the code for performance.

Figure 4.3 shows how the `Circle` class is laid out in memory. If you want to use the radius of each circle sequentially in a calculation, then you will generally only be getting one value per cache load. If the memory is laid out so that the radius value is contiguous (sequential) in memory, then you would load multiple values into cache that can all be used, which is both faster and allows for vectorization. While this object-oriented programming example creates many circle objects, the object-oriented programming approach can be applied to to multi-physics and multi-material implementations (Barlow, Morgan, and Shashkov, 2019; **?**; **?**) by creating many cell or material objects.

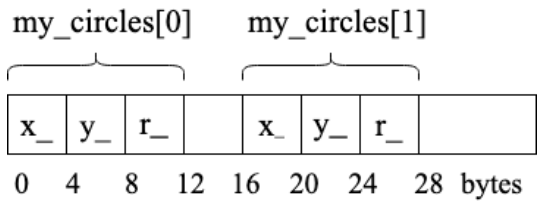

Figure 4.3. The memory layout for the array of classes `Circle`. Notice how the data are loaded for each object in memory instead of having the data, i.e variable `r` contiguously in memory. If the data is not used sequentially, more memory loads are needed leading to a memory bottleneck.

If we want performance on large problems the focus needs to shift from the objects themselves to the data layout and access patterns required to solve the problem in order to reduce memory loads. As others have suggested in (Llopis, 2009) and more in-depth in (Fabian, 2018), a data-oriented design (DOD) approach would be more performant than the programmer-oriented object-oriented programming paradigm because it places the focus on *how* the data is being used and *where* it is in memory. Shifting the focus onto how the data is being used and accessed will lead to better memory management for the data. Laying out

the data contiguously in memory so that it matches the access patterns used by the algorithm will lead to sequential data access, maximizing cache usage and opportunity for vectorization (Llopis, 2009). Also, for the case of ragged data structures, the amount of memory required to describe the problem is minimized.

Having the data laid out contiguous in memory in a bandwidth-limited code will also yield high performance benefits from getting full use of a cache line. For example, changing from the use of only half a cache line to a full cache line will generally double performance. Going from only using one value in a cache line to using all eight double precision values will roughly increase performance by 8x. Even when getting mostly full cache lines, getting memory on the same pages of memory will avoid additional page loads and their associated costs. Penalties are high if a data is not in the page cache and it has to walk all the pages to find the right page to load.

4.2.2 Design of dynamically allocated multi-dimensional arrays

It is often desirable to work with dynamically allocated multi-dimensional arrays, and C++ offers its end-users the ability to manage such arrays via the keywords `new` and `delete`. Listing 4 shows one of the conventional ways for dynamically allocating memory on the heap for a two-dimensional array in C++:

Listing 4 An example of dynamically allocating and deleting a two-dimensional array.

```
int main() {

  size_t rows = 4;  // Number of rows
  size_t cols = 5;  // Number of columns

  double** array = new double*[rows];

  for (size_t i = 0; i < rows; i++) {
    array[i] = new double[cols];
  }

  // Use array, array[i][j]

  // Deallocate the memory
  for (size_t i = 0; i < rows; i++) {
    delete[] array[i];
  }

  delete[] array;
  return 0;
}
```

In instances where programmers want to create a small multi-dimensional array in C++ for one-off usage, they can use the above approach. However, when programmers want to create a conventionally allocated, large multi-dimensional array, e.g., a five-dimensional array, they will find that writing the corresponding code is cumbersome and error-prone. A common error is forgetting to deallocate the memory upon termination of the application, which leads to memory leaks during runtime and can crash the code. In addition, there are deeper problems with the above approach, with one of the most pressing issues being that

multi-dimensional arrays that are conventionally allocated do not necessarily reside contiguously in memory. An example of what this can look like in memory is given in Fig. 4.4, and this can lead to a high performance cost.

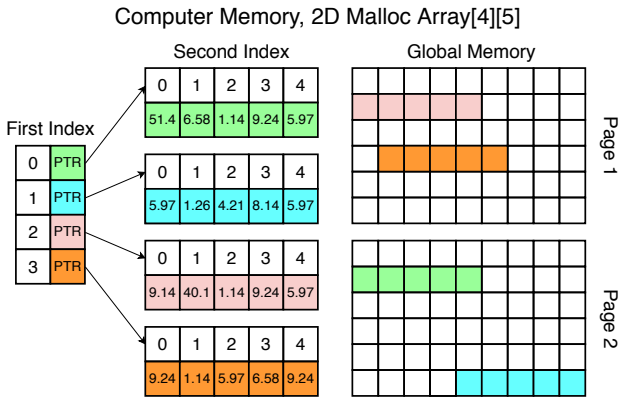

Figure 4.4. Generally, when using `malloc` or `new` to allocate multiple dimensional arrays in C/C++ you are not necessarily getting memory that is contiguously laid out on hardware. This can severely limit performance as memory has to be moved from random places instead of streaming continuously.

Having the data in the array scattered in global memory impacts how many flops can be done per load and lowers overall cache efficiency. This impacts the array's cache efficiency, but more importantly, any code that uses such arrays is not necessarily portable and parallelizable. Arrays allocated this way cannot be trivially passed between programming languages, GPUs cannot work with the array, the array cannot be written in block format to a file, and multiple processors cannot easily work with different portions of the array (Robey and Zamora,).

Listing 5 This code allocates a 2D array contiguously in C++. As you increase the number of dimensions the complexity required to allocate the array contiguously increases, and the number of times the memory must be deleted grows. This generally leads to more errors in a code.

```
int main() {

  size_t rows = 4;    // Number of rows
  size_t cols = 5;    // Number of columns

  // Contiguously allocate memory
  double** array = new double*[rows];
  array[0] = new double[(rows * cols)];

  for (size_t i = 1; i < rows; i++) {
    array[i] = array[(i - 1)] + cols;
  }

  // Use array, array[i][j]

  // Deallocate the memory
  delete[] array[0];
  delete[] array;

  return 0;
}
```

Of course, programmers do not have to dynamically allocate multi-dimensional arrays row by row, as described above; they can allocate multi-dimensional arrays contiguously in memory in C++, and one such way is given in Listing 5.

The above method is certainly not the only way to contiguously allocate memory for multi-dimensional arrays in C++, but we see that the programmer is able to obtain better memory allocation efficiency and cache efficiency even

with this rudimentary approach, and the portability problems that were identified earlier are now addressed. However, this approach is error-prone; the programmer still has to exercise great caution when allocating and deallocating the array in the above example, and this process is made more difficult when the programmer wants to work with larger multi-dimensional arrays. These are just a few of the common issues that motivated the creation of MATAR.

4.2.3 Goals of MATAR

MATAR is a C++ header library in development at Los Alamos National Laboratory that aims to address the issues identified in previous sections regarding contiguous memory layout for multi-dimensional and sparse data. MATAR aims to provide the end-user data structures that are:

1. easy-to-use, i.e., for each data structure, the library exposes a clean interface for intuitive data access and modification;
2. portable, i.e., each data structure is able to be used in both serial and parallel settings, and on multiple CPU and GPU architectures, via the Kokkos programming model, and
3. performant, i.e., each data structure, regardless of the data sparsity use case, maps, accesses, and modifies data as efficiently as possible in memory.

MATAR's data structures use a minimum amount of memory to describe complex data layouts and allows programmers to access and manipulate the data easily. It is also designed to give good performance with today's deep-cache hierarchies.

4.3 Methodology

The high level view of MATAR data structures combines the organization and object focus of object-oriented programming and data allocation and management of data-oriented design. MATAR data structures are pseudo-objects

that allocate multi-dimensional, dense, and sparse data in one contiguous block of memory and access the elements in memory as a one-dimensional array.

From a high level, every MATAR data structure consists of the following:

1. a one-dimensional array allocated contiguously in memory;
2. meta-data that
   (a) specifies the array's properties and
   (b) how the array is to be interpreted by the programmer, e.g., whether the array is to be treated as a two-dimensional array, a three-dimensional array, etc.; and
3. methods that allow the end-user to access and modify the elements of the array in a way that is consistent with the mental model of the MATAR data structure, i.e., if the end-user is expecting to work with a two-dimensional array, then the class exposes a method to access the array with two indices rather than forcing the user to do index calculations. MATAR also prevents the end-user from working with the two-dimensional array as an one-dimensional array or a higher-dimensional array when compiled with a debug flag. This allows for ease of debugging, but does not effect performance during a production run.

In addition, MATAR takes care of additional low-level details, including deciding where the memory for the underlying arrays and associated meta-data is allocated, i.e., on the CPU or GPU.

4.3.1 How MATAR's data structures are organized

MATAR's data structures are categorized by the following criteria:

1. access pattern, i.e., whether the data is laid out in column-major format or row-major format;
2. indexing pattern, i.e., whether the data is 0-indexed or 1-indexed; and

3. data sparsity, i.e., whether the data is densely packed or sparsely packed.

Tables 4.5 and 4.6 indicate how the various MATAR data structures are grouped by the above criteria.

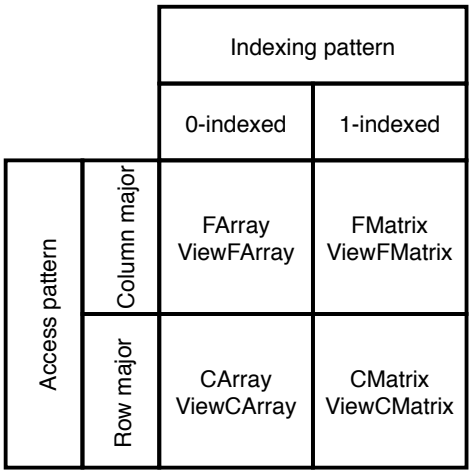

Figure 4.5. How MATAR's data structures for dense data access and modification are organized by memory access and indexing patterns

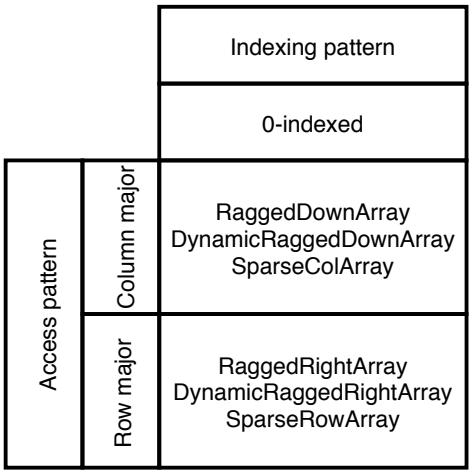

Figure 4.6. How MATAR's data structures for sparse data access and modification are organized by memory access and indexing patterns (all the sparse data structures are 0-indexed)

In the following sub-sections we will will go into greater detail about the importance of these criteria and how each of these structures are allocated and accessed.

4.3.1.1 Row-major and column-major data layout

The high-performance computing (HPC) community extensively uses Fortran, C, and C++, and many HPC practitioners call Fortran code, e.g., numerical linear algebra subroutines provided by LAPACK, from C and C++ code bases. One of the difficulties in getting existing Fortran code to work correctly within C and C++ code bases is ensuring that existing C and C++ access and modify multi-dimensional arrays appropriately with respect to Fortran code, and this difficulty arises from how Fortran and C/C++ lay out data in memory and how memory is allocated.

In Fortran, multi-dimensional arrays are laid out in memory in column-major layout, and Fortran gives end-users the ability to allocate arrays contiguously in memory via the `contiguous` attribute keyword, but in practice, most heavily-used Fortran compilers allocate arrays contiguously without the programmer explicitly requiring it to (Robey and Zamora,). In C and C++, multi-dimensional arrays are laid out in memory in row-major layout, and both languages allow programmers wide leeway in determining how the memory is allocated, as seen in earlier sections. As shown in Fig. 4.7, `Fortran` stores the multi-dimensional data as a one dimensional array column wise while `C/C++` stores a one dimensional array row wise.

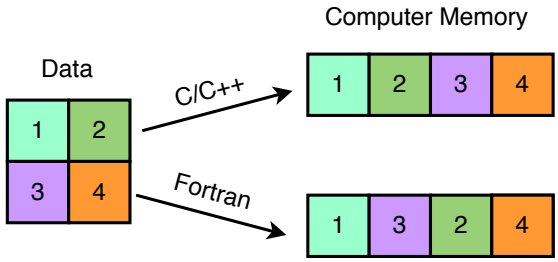

Figure 4.7. Memory layout pattern for a 2D array in `Fortran` and `C/C++`. Notice how `Fortran` data is laid out traversing the columns while `C/C++` data is laid out by row. MATAR considers both memory layout patterns for optimal performance.

It is important to consider the memory layout pattern of the language because better performance is achieved when data access patterns match how the data is laid out in memory (eg. the loop structure must walk over the data the

way it is laid out in machine memory). Listing 6 shows the fastest and slowest loop structures in C/C++ as an example. The loop structure will be reversed for Fortran code.

Listing 6 C/C++ stores memory in a row major order, so the index that is contiguous in memory is always the last one. Data is accessed contiguously by looping over this index at the lowest level of the loop structure, and if the loop is reversed, the memory will be fetched in jumps that are the size of the second index of the array. This kills cache performance and vectorization.

```
int main() {

  // Some int
  int a=1;

  // Make some array
  int some_array[10][10];

  // Fastest loop structure
  for(int i=0; i<10; i++){
    for(int j=0; j<10; j++){
      some_array[i][j]=a*i*j;
    }
  }

  // Slowest loop structure
  for(int i=0; i<10; i++){
    for(int j=0; j<10; j++){
      some_array[j][i]=a*i*j;
    }
  }
  return 0;
}
```

Accessing the data in the same pattern as how it is laid out in memory

ensures a cache line is fully utilized. MATAR has dense data structures `Matrix`, `Array`, `Views`, that take care of the contiguous memory allocation for each respected language for up to six dimensional data. Another benefit of including both row major and column major is it simplifies the process of converting scientific Fortran code to C/C++ code.

4.3.2 Dense Data Structures

MATAR data types for dense data are `Array`, `Matrix`, and `Views`. The data is allocated contiguously in memory (either column or row major) and accessed with the () operator. The only difference between an `Array` and `Matrix` is the index start value (0 for `Array`, and 1 for `Matrix`). Looking at Fig. 4.8 , an `Array` follows the traditional `C` indexing where j_0 and $i_0 = 0$ and `Matrix` indices are from $j_0 = i_0 = 1$ in order to be consistent with the how indices are normally defined in mathematics.

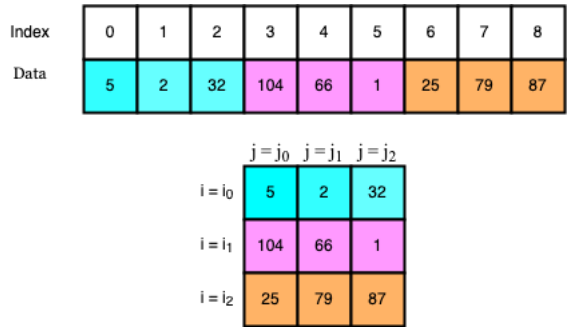

Figure 4.8. MATAR C-style (row-ordered) 2-dimensional data object. The 3×3 square is how the user will interpret the object while the top shows how the object is laid out contiguous in memory.

In Listing 7, we show some examples of how to create a `CArray` and `CMatrix` objects with different dimensions. When creating a MATAR data structure, the desired data type is specified in the chevron brackets <> and the size for each dimension in parenthesis () separated by a comma. MATAR currently supports one to six dimensional data.

Listing 7 A few examples of how to create and access CArrays and CMatrixs in MATAR. For each of these structures the last index is contiguous in memory. Creation of FArrays and FMatrixs is exactly the same, but with the notable exception that the first index is contiguous in memory.

```
int main() {
  // Creating examples of multi-dimensional
  // array and matrix objects

  // 2D 10x10 array
  auto array2D = CArray<int>(10,10);

  // Use the 2D CArray
  for (int i = 0; i < 10; i++) {
    for (int j = 0; j < 10; j++) {
      array2D(i,j) = 1;
    }
  }

  // 3D 70x70x70 array
  auto array3D = CArray<double>(70,70,70);

  // Use the 3D CArray
  for (int i = 0; i < 70; i++) {
    for (int j = 0; j < 70; j++) {
      for (int k = 0; k < 70; k++) {
        array3D(i,j,k) = 0.0;
      }
    }
  }

  // 2D CMatrix 50x50
  auto matrix2D = CMatrix<float>(50,50);

  // Use the 2D CMatrix
  for (int i = 1; i <= 50; i++) {
    for (int j = 1; j <= 50; j++) {
      matrix2D(i,j) = 0.0;
    }
  }
}
```

If you are working on a legacy code where it is unfeasible to go through a large code and replace *all* the traditional arrays with MATAR data types, then `Views` are a great alternative. `Views` allow the programmer to take a pre-existing array of data and access it as its own object. The `Views` allow the programmer to walk over an existing array that is contiguously allocated in memory and treat it as a multi-dimensional array for manipulation. One of the main benefits of `Views` is that they allow the programmer to pass sections of pre-existing blocks of memory into functions by reference. That view of the memory is then accessed contiguously in memory and manipulated in place. `Views` work similarly to arrays when it comes to accessing memory, and the main difference is that the `Views` do not allocate memory. They simply take in the pointer to the first index in memory of some pre-existing data and manipulate that data as if it is its own object. Figure 4.9 gives an example of how a contiguous block of memory can be accessed as a 3D array using `Views`, and Fig. 4.10 shows how the programmer can start a view at any point in memory. One thing to note is all memory is modified in place. This means that if you make a view of a pre-existing array and modify the data in the view, then the data in the pre-existing array is also modified. Also, when making views it is important to understand that while the views let you treat the underlying data as any size and dimension as long as the multiplicative combination is less than the difference between the starting index and the end of the pre-existing array, the user must understand the underlying structure of the pre-existing data to make sure the views are numerically consistent with whatever operation is being done on the data.

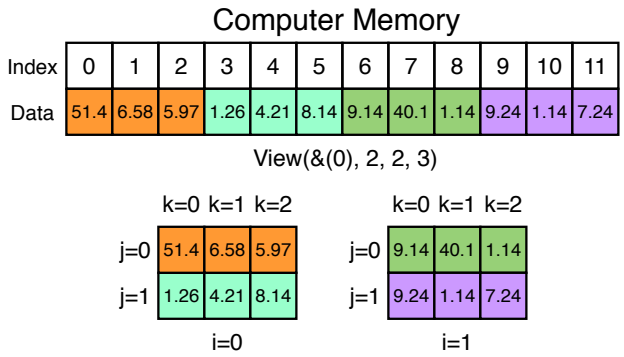

Figure 4.9. Here a `View` is created by specifying the desired index of some pre-existing memory and "viewing" it as a 3D array of size 2x2x3. This view can be accessed the same way as a CArray.

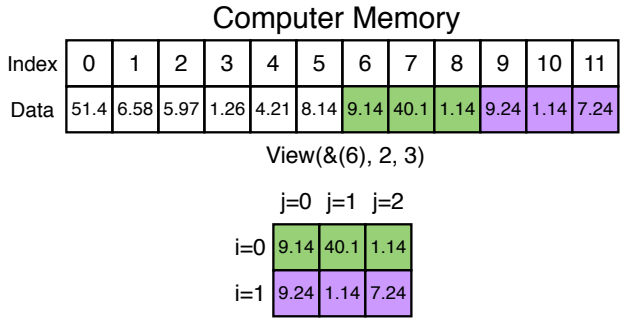

Figure 4.10. `Views` do not necessarily have to start at the first index of a pre-existing array. The user defines the start point, dimension, and size of each dimension when creating the view.

Listing 8 shows how views can be created using some pre-exisitng array. One benefit of using `Views` is you can treat the same data as different size arrays and each view will still access and modify data contiguously in memory. It is also possible to make `Views` of `Views`, as shown in Listing 8. To reiterate, the views are not “slices" that one might expect from python of Matlab, they are simply ways of accessing underlying contiguously allocated data. This means if you view a 3x3 matrix as a 1x9 array you will be walking over the 9 values in whatever order the 3x3 matrix was stored in memory. Listing 8 should clarify this further.

Listing 8 A `View` allows the programmer to access an already existing contiguous array and treat it as its own object. `Views` can be treated as having a higher, lower, or the same dimension as the underlying data depending on what is needed.

```
int main()
{
  // Creating examples of Views from fig. 9 and 10
  // 1. have a traditional contiguous array
  int arr[12];

  // 2. create view from figure 9
  // "view" array arr as a 3-dimensional object
  //with dimensions 2x2x3
  auto v_arr1 = ViewCArray<int>(&arr[0],2,2,3);

  // 3. create slice from figure 10
  auto v_arr2 = ViewCArray<int>(&arr[6],2,3);

  // View last 3 values of v_arr2 as a 1D array
  auto v_1d = ViewCArray<int>(&v_arr2(1,0), 3);

}
```

4.3.3 Sparse data structures

MATAR also provides data structures for sparse data access and modification, and these data structures fall under one of the following categories:

1. ragged data structures,
2. dynamic ragged data structures, and
3. compressed sparse arrays.

We will provide an overview of these three categories of data structures.

4.3.3.1 Ragged data structures

In computational physics codes, it is common to work with data that is be grouped by rows (or columns), where each row (column) may not be the same size. Figure 4.11 gives an example of the type of data layout where memory issues arise. The two-dimensional array is laid out in row-major order, but each row is a different size.

51.4	6.58	-	-	-	-	-	-
5.97	1.26	4.21	8.14	-	-	-	-
9.14	40.1	1.14	-	-	-	-	-
9.24	1.14	-	-	-	-	-	-
7.24	2.14	1.74	-	-	-	-	-

Figure 4.11. How a contiguous array might have unevenly populated, i.e., “ragged", rows. It is non-trivial to allocate memory efficiently for these types of structures with the standard C/C++ approach. The programmer can either allocate each row individually (which is rather tricky), allocate it as a dense array where large sections are unpopulated, or as a linked list, which is not necessarily contiguous in memory.

It is inefficient and expensive with respect to run time and memory usage to treat such sparse arrays as dense arrays, and while programmers can create a sparse array by allocating memory row-by-row for such data in C and C++, to gain the full benefits of this approach, they will have to ensure that the underlying memory is contiguously allocated, which is non-trivial. In these instances, MATAR’s ragged data structures offer programmers the benefits of working with such sparse data intuitively and efficiently without the pitfalls of standard methods of allocation. Assuming the sparse array shown in Fig. 4.11 is 0-indexed, this is how the end-user can visualize the data from Fig. 4.11 as a `RaggedRightArray` object, a MATAR data structure that lays out its associated data in row-major layout and is 0-indexed.

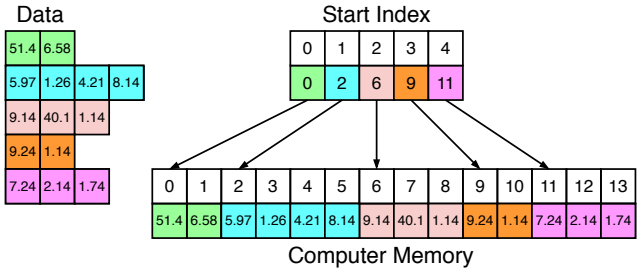

Figure 4.12. How the end-user can visualize the data from the previously shown sparse array as a `RaggedRightArray` object and how MATAR lays out the associated data in memory. The underlying memory is contiguously allocated, and an extra array is used to store the start and stop index.

If the end-user wants to traverse a given sparse, two dimensional array assuming that it is 0-indexed and that the array's data are laid out in column-major format, MATAR also provides a `RaggedDownArray` data structure, which is identical to the `RaggedRightArray` structure except that it allows end-users to access and modify data as if it were laid out in column-major format.

4.3.3.2 Dynamic, ragged data structures

MATAR also provides end-users with dynamic, i.e., "growable", ragged data structures, which are similar to the ragged data structures discussed in the earlier sub-section except that every row (or column) has an additional buffer that is not touched until the end-user chooses to do so. Thus, the end-user is able to increase the size of an individual row (or column), whenever the need arises, up to the buffer length that is specified upon the object's construction. For example, consider the dense array in Fig. 4.11. This dense array can be treated as a dynamic, ragged right array as shown in Fig. 4.13. If for this application the programmer knows that each row will be resized often they can use MATAR's dynamic ragged right array. This buffer addition to the ragged data structure leads to a slight performance hit, but it will be faster overall than if the memory had to be re-allocated and propagated each time the row size is changed (assuming this happens often. Figure 4.13 shows how the memory is laid out for a dynamic ragged data structure.

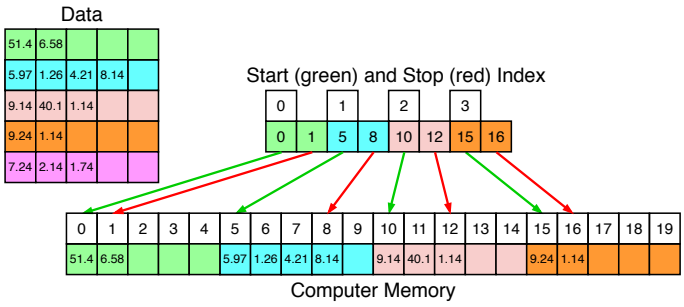

Figure 4.13. How the end-user can visualize the data from the previously shown sparse array as a `DynamicRaggedRightArray` object and how MATAR lays out the associated data in memory. The underlying memory is contiguously allocated, as in the case with a regular `RaggedRightArray` object, and an extra array is used the store the start and stop indices for each row. Furthermore, the `DynamicRaggedRightArray` allows the end-user to append additional data to a given row, up to the provided buffer size.

If the end-user wants to traverse the same data in column-major format, they can use MATAR's dynamic ragged down array data structure.

4.3.3.3 Compressed sparse arrays

It is common to work with large arrays that have very few non-zero entries (for example, finite difference matrices), and it is typically desirable to work only with the non-zero entries of such arrays. Two commonly used formats for compressing and working with such sparse, two-dimensional, and (typically) non-symmetric arrays are the following:

1. compressed row storage (CRS) format (also known as the Yale format), where the data is "compressed" row-wise, and
2. compressed column storage (CCS) format (also known as the Harwell-Boeing format), where the data is "compressed" column-wise.

For brevity's sake, we will discuss only the first format. Given a sparse, two-dimensional, and (typically) non-symmetric array `A` that is $m \times n$ and has ℓ non-zero entries (where ℓ is typically much smaller than $(m \times n)$), common CSR

implementations store `A`'s non-zero data via the following three, one-dimensional arrays:

1. `val`, an ℓ long array that stores `A`'s non-zero values as `A` is traversed row-by-row;
2. `row_ptr`, an $n+1$ long array (where the last entry of `row_ptr` is $\ell+1$) that stores the indices of `val`'s entries that start rows in `A`; and
3. `col_ind`, an ℓ long array that stores the column indices of `A` that correspond to `val`'s entries.

For example, consider the following sparse, two-dimensional array:

10	0	0	0	-2	0
3	9	0	0	0	3
0	7	8	7	0	0
3	0	8	7	5	0
0	8	0	9	9	13
0	4	0	0	2	-1

Figure 4.14. An example of a sparse, 6×6 array with 17 non-zero entries

The above array would be stored in CSR format as follows:

$$\begin{aligned} \texttt{val} &= [10, -2, 3, 9, 3, 7, 8, 7, 3, \ldots, 9, 13, 4, 2, -1] \\ \texttt{col\_ind} &= [1, 5, 1, 2, 6, 2, 3, 4, 1, \ldots, 5, 6, 2, 5, 6] \\ \texttt{row\_ptr} &= [1, 3, 6, 9, 13, 17, 20] \end{aligned}$$

MATAR provides both CPU and GPU implementations of both formats.

4.3.3.4 Motivation for ragged data structures

A linked list (LL) is a basic data structure in which objects are arranged in linear order, where the order is determined by a pointer in each object (Cormen, Leiserson, Rivest, and Stein, 2009). Typical linked list implementations provide simple and flexible representations of dynamic sets; new elements can be deleted and added unlike a traditional array where the size is set. Although LL are flexible in terms of their size, there are significant drawbacks such as

1. Accessing the elements must be done in order, therefore a search can quickly become expensive if the list is not sorted.
2. No guarantee the LL is contiguous in memory (i.e poor data locality). The nodes may be scattered throughout global memory, similarly to figure 4.4.
3. They can not be ported to the GPU.
4. Repeatedly adding elements to an list without re-allocating memory on the heap for these additional elements; a prohibitively memory- and time-intensive operation.

Thus, linked lists are generally avoided in parallel code bases, especially so when it is expected that they will be heavily modified, i.e., elements will be inserted at the end of lists and deleted from lists.

MATAR addresses the portable, accessing pattern, and data locality drawbacks from LL and offers the dynamic ragged data structures. The sparse dynamic data structures allow end-users to leverage the flexibility of working with linked lists while being usable in parallel settings(CPUs and GPUs) and offers better data locality through contiguous memory allocation. The biggest difference between a MATAR dynamic data structure and a LL is where the new elements are added. While in a LL the nodes can be added throughout the list, for a dynamic ragged structure elements can only be appended in the buffer at the *end* of a row (`DynamicRaggedRight` in figure 4.13) or column (`DynamicRaggedDown`).

To verify the potential gains of using MATAR's dynamic ragged data structures over linked lists, we looked at the production code HOSS (Knight, Rougier,

Lei, Euser, Chau, Boyce, Gao, Okubo, and Froment, 2020), specifically within the contact detection algorithm (Rougier and Munjiza, 2010). Within HOSS, every element within a simulation has it's own linked list that contain the neighbours of that element. As the simulations progress, nodes from the LL are inserted if elements come into contact or removed if they are no longer in contact. For a more in-depth understanding of the contact detection algorithm, we refer the reader to (Rougier and Munjiza, 2010). Two separate code bases were maintained, (1) the existing code base that used the LL, and (2) a modified code base that used MATAR's `DynamicRaggedRight` array (DRRA) data structure. The size of the test problem is 10 million elements, with the input file containing the cartesian coordinates for the elements that form the cells and the points that are allowed to freely move.

To compare the performance of the contact detection algorithm with the two data structures we used the command line tool `likwid` to gather information about the bandwidth, data volume, and vectorization. More information about `likwid` and how it was used can be found in section 5.Both codes were compiled with `gcc-9.2.0` and ran on an Intel Xeon Gold 6152 CPU (2.10 GHz), without any accelerators.

The performance statistics will capture the contact detection algorithm and the assembly of the respective data structure (a linked list in the first code base and a dynamic ragged right array in the second code base) for two examples. One is just one run of HOSS, and the second test is a stress test- a section that computes row statistics (row average, row minimum, row maximum, and row sum) over each "row" of the respective data structure after assembly (this set of computations was performed 10000 times).

Table 4.3 and figure 4.15 show the results for one run of the HOSS code while table 4.4 and figure 4.16.

Table 4.1. LIKWID results from running the CDA on competing HOSS implementations. The DDRA is about 3% faster.

Performance Metric	LL	DRRA
Run time (RDTSC) [s]	16.1768	15.7079
DP MFLOP/s	119.9842	123.7221
AVX DP MFLOP/s	0.7669	0.8489

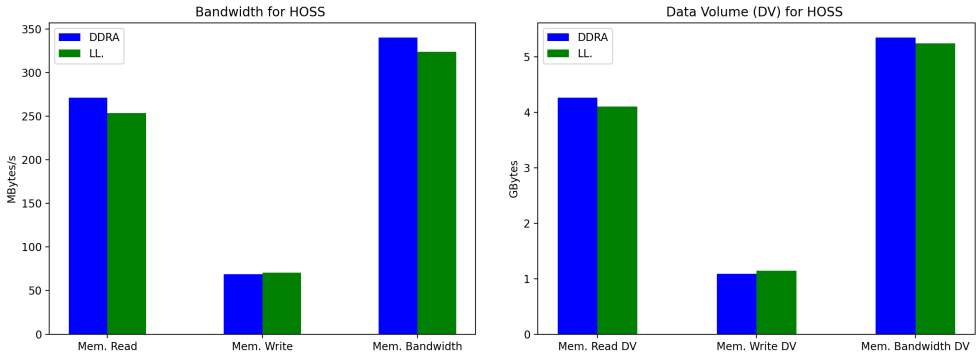

Figure 4.15. Memory bandwidth and data volume for the HOSS after one run. The MATAR DDRA has higher bandwidth and data volume for memory read and overall. The DDRA was still faster even though more data fetches were needed.

From the results in table 4.3 MATAR's `DynamicRaggedRight` array is about 3% faster than the traditional LL. However figure 4.15 shows the DRRA had higher data volume than the linked list.

Table 4.2. LIKWID results from computing row statistics on competing HOSS implementations (over $10,000$ iterations). We see a drastic improvement in run time with the DDRA. The increased performance is due to the contiguous memory access.

Performance Metric	LL	DRRA
Run time (RDTSC) [s]	580.4707	**325.6728**
DP MFLOP/s	453.9550	568.5682
AVX DP MFLOP/s	0	0

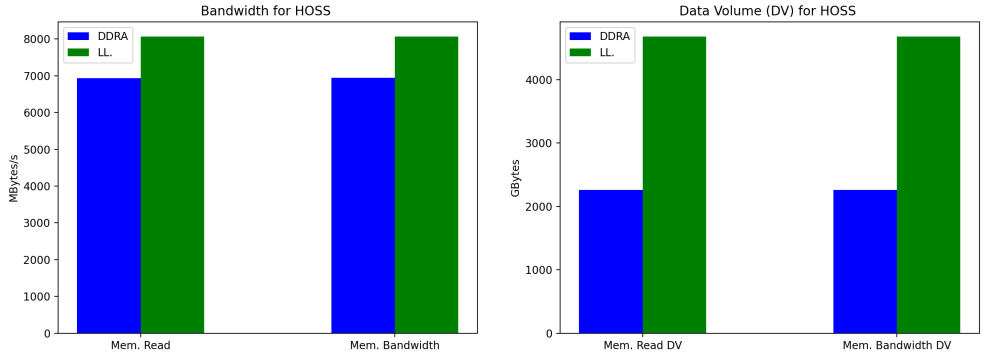

Figure 4.16. Memory bandwidth and data volume for the HOSS stress test. The linked list has higher bandwidth and data volume for the stress test (10,000) runs. The greater data volume is attributed to the linked-list data being scattered throughout global memory. While the DRRA is allocated contiguous, less memory loads are needed. Note, the memory write bandwidth and data volume is not included because the value is significantly smaller than the read and overall making it not visible in the plots.

Performance metric	LL	DRRA
Run time (RDTSC) [s]	16.1768	15.7079
DP MFLOP/s	119.9842	123.7221
AVX DP MFLOP/s	0.7669	0.8489
Memory read bandwidth [MBytes/s]	253.6222	271.2838
Memory read data volume [GBytes]	4.1028	4.2613
Memory write bandwidth [MBytes/s]	70.4443	68.9248
Memory write data volume [GBytes]	1.1396	1.0827
Memory bandwidth [MBytes/s]	324.0665	340.2086
Memory data volume [GBytes]	5.2424	5.3440
Operational intensity	0.3702	0.3637

Table 4.3. LIKWID results from running the CDA on competing HOSS implementations

Performance metric	LL	DRRA
Run time (RDTSC) [s]	580.4707	**325.6728**
DP MFLOP/s	453.9550	568.5682
AVX DP MFLOP/s	0	0
Memory read bandwidth [MBytes/s]	8060.5569	6936.2213
Memory read data volume [GBytes]	4678.9171	2258.9386
Memory write bandwidth [MBytes/s]	3.7774	2.9379
Memory write data volume [GBytes]	2.1927	0.9568
Memory bandwidth [MBytes/s]	8064.3344	6939.1593
Memory data volume [GBytes]	4681.1098	2259.8954
Operational intensity	0.0563	0.0819

Table 4.4. LIKWID results from computing row statistics on competing HOSS implementations (over $10,000$ iterations)

The generated LIKWID report shows the run time slightly decreased for the DRRA over the LL for one run, and had a 44% reduced run time over the LL for the stress test. If multiple simulations will be performed, the DRRA is a clear choice over the traditional LL. These preliminary results indicate that the overall HOSS code and similar algorithms that need to keep track of neighbors/elements (refer to figure 4.1 and 4.2) base will potentially benefit from the replacement of existing LL instances with DRRAs. More importantly, DRRAs offer features that LL can not handle. LLs are not portable compared to MATAR's DRRAs, i.e., general LL implementations do not work correctly, if they do work at all, on GPUs, whereas MATAR's DRRA implementation is designed to work on GPUs. The elements of a DRRA can be accessed as a 2D array `(i,j)`, eliminating the sorting overhead that most LL do.

4.4 Parallelization (KOKKOS)

MATAR is build with performance in mind, and with ease of use for developers and scientists alike. In following with that trend, we wanted to pair it with a technology that also focused on performance as well as portability. Kokkos (Edwards, Trott, and Sunderland, 2014; **?**) was the obvious choice for this coupling because of its focus on providing a programming model that focuses on similar concepts that we focus on in MATAR. The objective of Kokkos is to create an environment where the same code (or nearly the same) can be run with high performance on multiple CPU and GPU architectures. For MATAR specifically, we incorporate it as a means to target many different GPU architectures on diverse HPC systems. By leveraging Kokkos, MATAR is able to provide complex data structures to users with a simple interface.

4.4.1 Adding Kokkos to MATAR

Extensive collaboration with Kokkos developers was done to incorporate Kokkos inside of MATAR, while still providing a simple interface to users. A large portion of this work involved providing MATAR class data and functions to a user that hides all GPU pointers and memory management, as well as complex indexing, from the inheriting libraries. GPU memory does not have to be managed by a MATAR user, and even some costly initialization are parallelized and optimized inside of the MATAR library. Specific class functions must be labeled with the Kokkos macro `KOKKOS_FUNCTION` for class functions to return data members on the GPU. The data structures themselves are allocated and initialized almost exactly as they are for their non-Kokkos counterparts, using Kokkos `Views` (not to be confused with our `MATAR Views`) instead of the `C++` arrays.

The key difference between the MATAR classes containing Kokkos and those that do not occurs within the Ragged data structures. Here, the constructor has preliminary calculations that are necessary for keeping track of the ragged array start indexes. Ideally this is a simple conversion of for-loops to Kokkos parallel loops. However only the latest Cuda version supports lambda capturing inside of class functions, and even then these lambdas (extending further to

Kokkos parallel for) are not allowed in a class constructor. In order to preserve the functionality and appearance we present in the other MATAR classes, extra background work within MATAR had to be done such that initialization is shown to the user in the same way as the other classes. Furthermore, to get the latest Kokkos class functionality, we need to use GCC 9 coupled with Cuda 11.0 to work with C++ 17.

4.4.2 Virtual Functions

MATAR has the ability to hold data types that differ from a simple integer or double type. When creating a library of data structures, as we have done here, it is important to demonstrate the ability to contain more complex (not to be confused with a complex number) data types. We show the ability of MATAR to to hold class data types, including those involving virtual functions. This is fairly trivial on a CPU, but is a complicated process when we involve a GPU. Kokkos views have the ability to hold class data types by copying all of the class attributes and functions, including constructors and destructors, at the time of instantiantion. However, virtual functions are not able to be copied at this time. Extra steps need to be taken to ensure that the virtual functions and their derived counterparts are seen on the device (GPU), rather than on the host (CPU). The Kokkos framework gives an adequate explanation for this problem specific to Kokkos, which we echo here for this simplification.

The problem is that the class is initialized on the host, so the object points to a host VTable. When these objects are accessed on a GPU, an error obviously occurs because the device has no knowledge of that host pointer. The class needs to be initialize on the GPU itself so that the virtual function pointers hold a GPU pointer value. Unfortunately, this does not occur when creating a Kokkos view of that class type, because the class is being initialize on the host for that Kokkos GPU view. The situation becomes even more complex when we are working a derived class that lies within a different class. The solutions accommodating this situation is obviously out of the scope of most applications, even though it is a common application design. We have made the process as simple as possible for a user doing this with MATAR. This is especially helpful when the abstraction is

extended and we are working with derived classes inside other classes, a common situation in multiphysics applications. Algorithm 6 shows the steps to ensure correct pointer location.

Algorithm 6 *Allocating polymorphic classes with virtual function classes in MATAR*

```
// Assuming a hierarchy of Class A containing a pointer to Class B, which can be derived to B1 or B2.

// Host Array
Create CArray h_array of type A with size elements
// Device Array
Create CArrayKokkos d_array of type A with size elements

// Loop through host array
for all elements elem in h_array do
   // GPU pointer in the host array
   Cuda or Hip Malloc h_array[elem].B of size parent class
end for

// "move" GPU pointers to the device
Kokkos deep copy h_array to d_array

// Loop through device array
// This is a Kokkos parallel loop
for all elements elem in d_array do
   // Initialize object on device in allocated space
   new ((B1 or B2 *) d_array[elem].B) with constructor parameters
end for

// d_array is now ready for use on the GPU
```

4.5 Results (Scaling Studies)

One of the goals of MATAR is to be as or more performant than standard C/C++ data structures (such as linked list, arrays, vectors). "More performant" can be measured by a variety of metrics. For example, we could see an improvement in run time of the application or lower data volume. Here we show some performance results between MATAR and dynamically allocated C++ arrays with the performance metrics: memory read bandwidth, memory write bandwidth, memory bandwidth, advanced vector extensions (AVX), and double precision (DP) MFlops/s. To obtain the performance metrics mentioned, our performance tool of our choice is `likwid` (**L**ike **I** **K**now **W**hat **I**'m **D**oing) (Treibig, Hager, and Wellein, 2010), a Linux-specific suite of command line tools that utilizes hardware counters to let end-users measure and report information about an application. Specifically `likwid-perfctr` version 4.3.4. We use the marker API mode, were we section of and gather data only on the kernels; parts of the code such as the initialization is not measured.

We run timing studies and the performance diagnostics on the Babel STREAM Benchmark, a matrix-matrix multiply (MMM) a matrix-vector product (MVP), and an a contact detection algorithm in a geophysical application-HOSS. All CPU tests are run on an Intel `Skylake Gold` CPU with a `2.10 GHz` clock speed using `gcc/9.2.0`. Details for each test are given in their respected subsection and detailed examples of how to reproduce the results are on the github.

With these test, we are testing if (1) MATAR multi-dimensional data structures that are contiguously allocated in memory are faster than traditional, dynamically allocated `C++` arrays, and (2) complex, sparse MATAR data structures are more efficient than traditional sparse representations in real a application.

4.5.1 CPU

In the CPU performance portion, we focus on understanding and comparing the baseline run time performance, bandwidth, data volume and vectorization of MATAR `CArrays` and `ViewCArray` against a traditional dynamically allocated C++ array (denoted by `Trad. Array` in figures and tables following) using the

Babel Stream Benchmark. As well as testing a matrix-matrix multiply (MMM) and matrix-vector product (MVP) because these operations are heavily used in most numerical methods. These operators are simple yet suited to validate the effectiveness of MATAR's data-oriented design methodology. The `matrix` data structure is not presented because the only difference between MATAR `arrays` and `matrix` is the indexing (refer to section 3); the `array` and `matrix` are allocated identically in memory. The `Fortran`-style is also not presented in the Babel Stream benchmark. We believe the results from `CArray` is sufficient in understanding how data-oriented design compares against traditional arrays. Contiguous memory, whether it be `Fortran`-style (column-wise) or `C`-style (row-wise) is expected to perform similarly. Therefore the results of the C-style array types are extended to `matrix` types and `Fortran` style data structures.

4.5.1.1 BabelStream Benchmark

The STREAM benchmark is a set of simple kernels (copy, scale, sum, triad) to test the theoretical sustainable memory bandwidth in MB/s. The arrays in question are allocated on the stack and the size must be large enough to be in main memory. In our case, since we want to compare the differences of contiguous memory and dynamically allocated memory, we followed the modified Babel STREAM, which includes a dot product kernel and arrays allocated on the heap (Deakin, Price, Martineau, and McIntosh-Smith, 2016; **?**).

We present the results for the Babel STREAM on a MATAR `CArray`, a `ViewCArray` of a traditional array, and traditionally dynamically allocated `C++` array in one-dimension (1D) and three-dimensions (3D). The one-dimensional is meant to serve as a proof-of-concept that we are at least up to par with the traditional arrays. Meanwhile the three-dimensional case will give a better overview and extension of MATAR's capabilities since the intended design is for multi-dimensional data structures in multi-physics simulations. The array size for the 1D is 16,777,216 which is large enough to be in main memory and divisible by 8 (ideal for vectorization) and for the 3D case, each dimension is 256 ($256^3 = 16,777,216$).

The run time results for both the 1D and 3D case are shown in tables 4.5

and 4.6. The test were ran 101 times, but the average was computed from the 2^{nd} to the 101^{st} time; the first timing is ignored since the cache is warming up. The reported times are using the standard `chrono::steady_clock` and is truncated after five significant digits.

Table 4.5. Run time (in milliseconds) of the 1-dimensional BabelStream Benchmark Test (size 16,777,216). The results show some variation between the MATAR `CArray` and `ViewCArray`. Both are slower in the copy and scale kernel, but the `ViewCArray` is faster than the `CArray` in the dot product. Overall, we see do not see significant overhead from the OOP/classes implementation.

Data Structure	Tests	Avg.	Min.	Max.
CArray	Copy	20.648	20.433	22.572
	Scale	20.399	20.195	23.213
	Sum	30.069	29.911	33.061
	Triad	30.557	30.430	32.135
	Dot Prod.	23.847	23.636	25.169
Trad. Array	Copy	20.064	19.905	20.164
	Scale	20.262	20.079	20.425
	Sum	30.263	29.988	31.880
	Triad	30.370	30.305	31.5794
	Dot Prod.	23.474	23.275	23.766
ViewCArray	Copy	20.573	20.426	20.733
	Scale	20.743	20.606	21.028
	Sum	30.438	30.285	31.659
	Triad	30.359	30.311	30.514
	Dot Prod.	23.331	23.117	24.573

Table 4.6. Run time (in milliseconds) of the 3-dimensional Babel STREAM Benchmark Test (size 256^3 total). The table shows MATAR excels in higher dimensional data structures. Every kernel excluding the dot product is about 40% faster. The decrease in run time is most likely attributed to the contiguous memory layout and maximizing cache usage. In general, `C++` multi-dimensional arrays are not guaranteed to be continuous and are often scattered throughout global memory.

Data Structure	Tests	Avg.	Min.	Max.
	Copy	56.812	56.765	57.185
	Scale	57.160	57.111	58.313
CArray	Sum	60.867	60.759	61.967
	Triad	58.370	58.318	58.679
	Dot Prod.	23.655	23.460	23.972
	Copy	94.319	94.013	94.812
	Scale	96.192	95.584	113.412
Trad. Array	Sum	96.087	95.449	129.506
	Triad	100.803	99.700	101.518
	Dot Prod.	31.222	31.443	31.465
	Copy	55.459	55.344	56.305
	Scale	55.902	55.864	56.294
ViewCArray	Sum	58.298	58.218	59.493
	Triad	58.809	58.756	59.244
	Dot Prod.	23.686	23.526	23.938

From table 4.5, MATAR `ViewCArray` and `CArray` are slightly under performing the traditional `C++` array in the copy, scale and individual vary in the

sum, triad and dot product kernel. The copy kernel is the slowest for both the `CArray` and `ViewCArray`. Yet, we can see that there is no significant overhead from the object-oriented implementation of the MATAR data structures.

Table 4.6 shows the timings for the 3D test, and we can see MATAR `CArray` and `ViewCArray` are the fastest for all kernels. Both MATAR `CArray` and `ViewCArray` are 40% faster over all the tests except the dot product, which is only about 20% faster. We expect the speed-up in run time to be more significant for higher dimensional data structures because in `C++`, as shown in figure 4.4, dynamically allocated multi-dimensional arrays are not guaranteed to be contiguous and are often scattered throughout global memory. The performance benefit we are seeing is likely due to the contiguous memory allocation. The data is rolled out in memory in the order it will be used therefore maximizing cache usage, needing less memory loads per operation and hence, we see a decrease in run time.

To summarize the run time results, tables **??** and **??** are the percent difference for the average run time for each tests. The percent differences are taken with respect to the traditional arrays, therefore a positive value would signify a slow down while a negative value indicate a speed up.

We now use the performance profiling tool `likwid` to gain some insight on the run time results by looking at the the double precision flops (DP), advanced vector extensions (AVX), bandwidth (read, write and overall), and data volume (read, write and overall). `likwid` markers are put at the beginning and end of each test and only ran once. Explicit `#pragma omp` directives were included for all tests in each dimension and vectorization flags are added automatically using a `FindVector.cmake` module. This vector module optimizes the compiler flags based on the architecture (Robey and Zamora,). Tables 4.8 and 4.9 show the vectorization for each of the 5 kernels. All of the double precision flops for each of the 1D data structure were able to vectorize for the scale, sum and triad kernel, however we did not see the same vectorization for the 3D case (4.9. We would expect to see vectorization for the all dimensions of the MATAR `CArray`'s and `ViewCArray`'s since multi-dimensional data is unrolled and allocated as a 1D contiguous array (see Fig. 4.8).

Table 4.7. Table describing percent difference for 1D and 3D with respect to the traditional arrays. A positive number signifies a slow down while a negative number is a speed up. We can see both MATAR data structures are more performant than the traditional array. The 1D data structures were slower but up-to-par with the traditional arrays.

	1D		3D	
	CArray	View	CArray	View
Copy	2.90	2.53	-39.76	-41.20
Scale	0.678	2.37	-40.57	-41.88
Sum	-0.640	0.579	-36.66	-39.32
Triad	0.616	-0.35	-42.09	-43.65
Dot Prod.	1.58	-0.67	-24.23	-24.13

We experiment further with the high-dimensional data to understand the vectorization inconsistencies. In table 4.10, the triad kernel is the tested for 3D and four-dimensions (4D) on the `gcc/9.2.0` and `Intel/19.0.5` compiler. The size of the 4D tests is of size 64 in each dimension ($64^4 = 16,777,216$). The triad was selected to re-run on different compilers because it is the most arithmetic intensive test that was able to vectorize (as seen in table 4.5). For the test in table 4.10, we experimented with the `#pragma omp` directives and found that vectorization *is* achievable on the `gcc` compiler *without* the `pragma` whereas the `Intel` compiler vectorized more `with` the `pragma` directives.

It is still unclear as to why MATAR and traditional arrays are compiler sensitive to the `pragma` directives. Obtaining consistent vectorization is a complicated task and merits its own research. The scope of this paper will not cover why we see the vectorization and compiler sensitivities. More research is required and a deeper analysis of the individual compilers and vectorization flags are left for future studies.

Table 4.8. Double Precision Flops and Advanced Vectorization (something) for the 1D Babel Stream Benchmark. All data structures were able to vectorize the entire DP Flops for the scale, sum and triad kernels.

Data Structure	Tests	DP	AVX
CArray	Copy	0.007	0
	Scale	339.1739	339.1732
	Sum	279.2456	279.2451
	Triad	893.6371	893.6362
	Dot Prod.	1109.2731	0
Trad. Array	Copy	0.007	0
	Scale	346.3716	346.3715
	Sum	275.6272	275.6267
	Triad	937.6608	937.6599
	Dot Prod.	1105.8774	0
ViewCArray	Copy	0.0012	0
	Scale	643.5925	643.5913
	Sum	452.9831	452.9829
	Triad	934.6927	943.6929
	Dot Prod.	1110.9600	0

Table 4.9. Double Precision Flops and Advanced Vectorization (something) for the 3D BabelStream Benchmark. None of the data structures were able to vectorize, even though explicit `pragma` directives were included in the tests. More studies need to be conducted to better understand vectorization and see consistent results.

Data Structure	Tests	DP	AVX
CArray	Copy	0.002	0
	Scale	191.9354	0
	Sum	189.8068	0
	Triad	282.4736	0
	Dot Prod.	1125.5502	0
Trad. Array	Copy	0.001	0
	Scale	120.9522	0
	Sum	138.2932	0
	Triad	282.4736	0
	Dot Prod.	835.2483	0
ViewCArray	Copy	0.003	0
	Scale	248.1745	0
	Sum	138.2932	0
	Triad	475.6680	0
	Dot Prod.	461.5282	0

Table 4.10. Table comparing the `Intel` and `gcc` compiler on a 3D and 4D triad example. All data structures were able to vectorize using the `gcc` compiler *without* the `#pragma omp` directives. Meanwhile with `Intel`, we see more vectorization on the same problem *with* the `#pragma omp` directives. Future research need to be conducted on the compiler and vectorization flag sensitivities for both MATAR and multi-dimensional traditional arrays.

		`gcc/9.2.0`		`Intel/19.0.5`	
		DP	AVX	DP	AVX
3D	CArray	469.69	469.69	565.52	565.52
	Trad. Array	595.12	595.12	4409.12	4409.12
	View	466.08	466.08	514.93	514.93
4D	CArray	520.10	520.10	5532.28	5532.28
	Trad. Array	664.04	664.04	4548.79	4548.79
	View	811.85	811.85	6087.59	6087.59

Now we look at the bandwidth and data volume for each of the kernels. As a note to the reader, for both the 1D and the 3D dot product kernel (figures 4.21 and 4.26), the memory write bandwidth and data volume is excluded from the graph. This is because the value was significantly lower than the memory read and overall and was not visible in the plot.

For all figures 4.17 - 4.25, the memory read, write and overall bandwidth (MBytes/s) and memory read data volume, write data volume, and overall memory data volume (GBytes) are presented. The raw data for the bandwidth and data volume is shown in the appendix and in the github repository along with the tests.

For the 1D case, the `CArray` performed the slowest in the copy kernel. However figure 4.17 shows the `CArray` and the traditional array have nearly identical bandwidth rates and data volume for read and write. The `ViewCArray`'s bandwidth and data volume is significantly higher than either the traditional and `CArray`, yet it was faster than the `CArray`. It is difficult determining the rela-

tionship between the bandwidth, data volume and run time for the Babel Stream benchmark; for this test, there is no obvious correlation between higher bandwidth or data volume and an increase in run time. In figure 4.19 where the `CArray` was faster by 0.6%, we see the same pattern as the copy kernel (figure 4.17) where the bandwidth and data volume compared to the traditional array is nearly identical. In the dot product kernel, the `CArray` slowed down by 1.5% while the `ViewCArray` saw a 0.6% speed up, yet figure 4.21 shows the bandwidth and data volume are all nearly identical. Perhaps the slow down we see is the slight overhead from the object-oriented class structure implementation of MATAR.

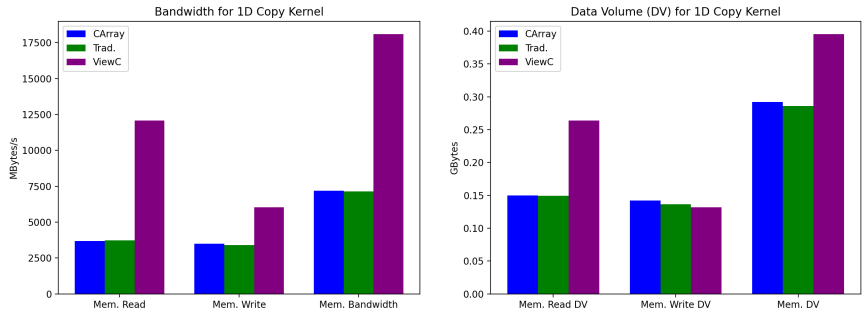

Figure 4.17. Memory bandwidth and data volume for the 1D copy kernel. Both `CArray` and `ViewCArray` perform slower than the traditional `C++` array (2.9% and 2.5%). The `CArray` is the slowest yet had identical bandwidth and data volume. The `ViewCArray` has the highest bandwidth and data volume, yet the run time performance ordering is `CArray` < `ViewCArray` < traditional array.

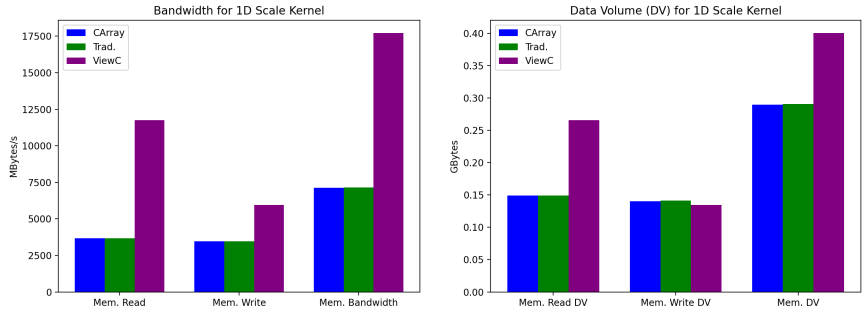

Figure 4.18. Memory bandwidth and data volume for the 1D scale kernel.Both `CArray` and `ViewCArray` perform slower than the traditional `C++` array (0.6% and 2.3%). The `CArray` has identical bandwidth and data volume to the traditional array and the `ViewCArray` maintains the highest bandwidth and data volume.

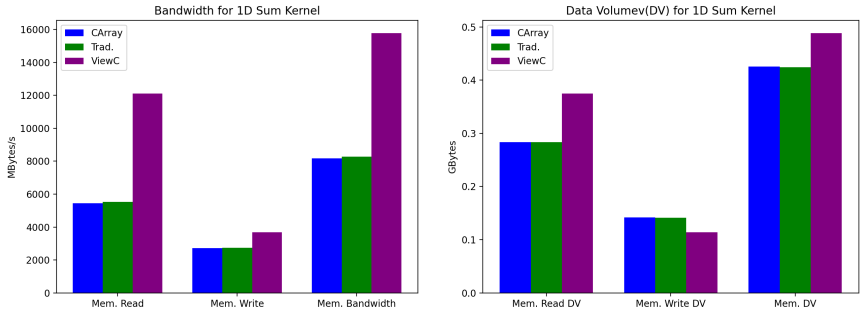

Figure 4.19. Memory bandwidth and data volume for the 1D sum kernel. The `CArray` is 0.6% faster but maintains identical bandwidths and data volume to the traditional array. The `ViewCArray` is 0.5% slower and has higher bandwidth and data volume.

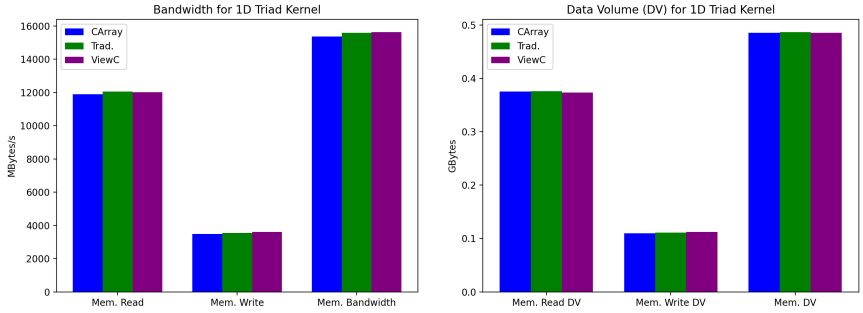

Figure 4.20. Memory bandwidth and data volume for the 1D triad kernel. Here all three data structures have nearly identical read and overall bandwidths and data volume. However the `CArray` performed 0.6% slower while the `ViewCArray` performed 0.3% faster.

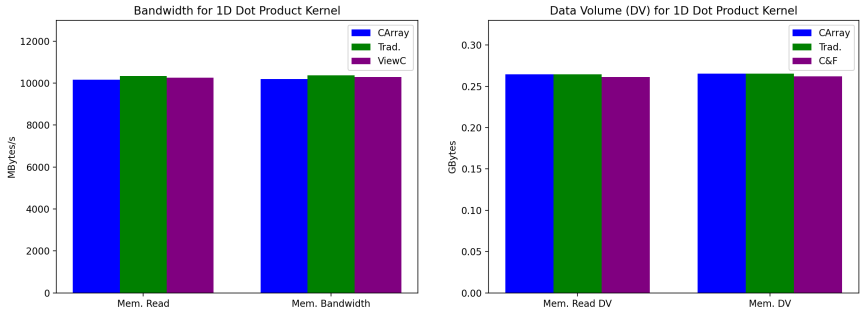

Figure 4.21. Memory bandwidth and data volume for the 1D dot product kernel. Here all three data structures have nearly identical read and overall bandwidths and data volume. However the `CArray` performed 1.5% slower while the `ViewCArray` performed 0.6% faster.

The 3D tests showed MATAR `CArray` and `View` to be faster by about 40% for every kernel , expect the dot product where they are only 20% faster (recall table 4.7). The bandwidth is more varied for this dimension compared to the 1D case. Although we were unable to draw a clear correlation between the bandwidth, data volume and run time, the 1D case showed correlation between having higher bandwidth and more data volume. In figures 4.17 - 4.20, the `ViewCArray` data structure had the highest bandwidth *and* the most data volume for the copy, scale, and sum kernel. However, the results for the 3D benchmark (figures 4.22 - 4.25) show no correlation between the bandwidth and data volume. For example, in the copy kernel in figure 4.22 the `ViewCArray` had the highest rate of memory bandwidth for all cases (read, write, and overall), yet the traditional `C++` array had the highest data volume for read and overall. For all tests cases, figures (4.22 - 4.25) report the the traditional array has the most amount of data volume in read and overall. The higher data volume could be a strong indicator as to why the traditional C++ array is significantly slower to its counterparts. For higher dimensional data structures, the data has a greater chance of scattering throughout memory (recall **??**), therefore increasing the amount of memory loads per operation. The increase in memory loads per operation.

The variation in the bandwidth rates between the `ViewCArray` and `CArray` is also unexpected. In figures 4.22 - 4.19, the `CArray` has the *lowest* bandwidth rate while the `ViewCArray` has the *highest* rate. Meanwhile *both* were around 40%

faster than the traditional array (whose bandwidth lays in the middle).

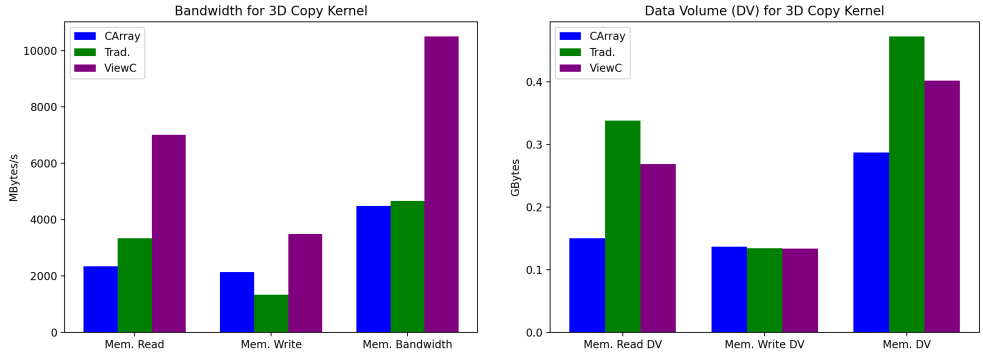

Figure 4.22. Memory bandwidth and data volume for the 3D copy kernel. Both the `CArray` and the `ViewCArray` outperformed the traditional `C++` array by about 40%, yet the bandwidths rate are on the opposite side of the spectrum. The `ViewCArray` has the highest bandwidth while the `CArray` has the lowest. The traditional array has the highest data volume overall.

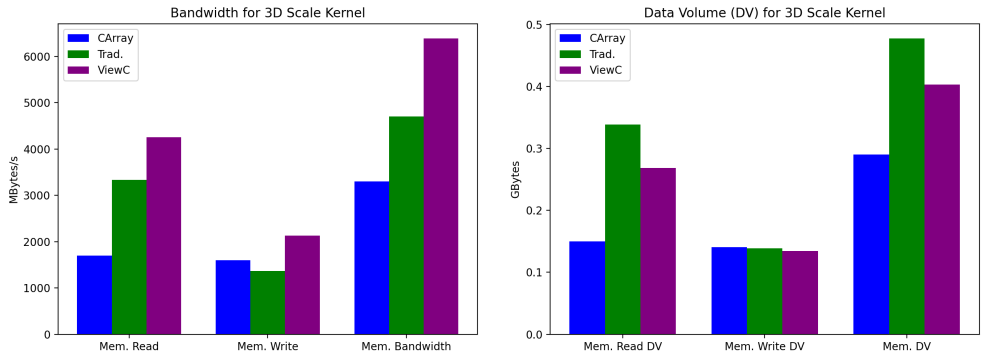

Figure 4.23. Memory bandwidth and data volume for the 3D scale kernel. Both the `CArray` and the `ViewCArray` outperformed the traditional `C++` array by about 40%, yet the bandwidths rate are on the opposite side of the spectrum. The `ViewCArray` has the highest bandwidth while the `CArray` has the lowest. The traditional array has the highest data volume overall.

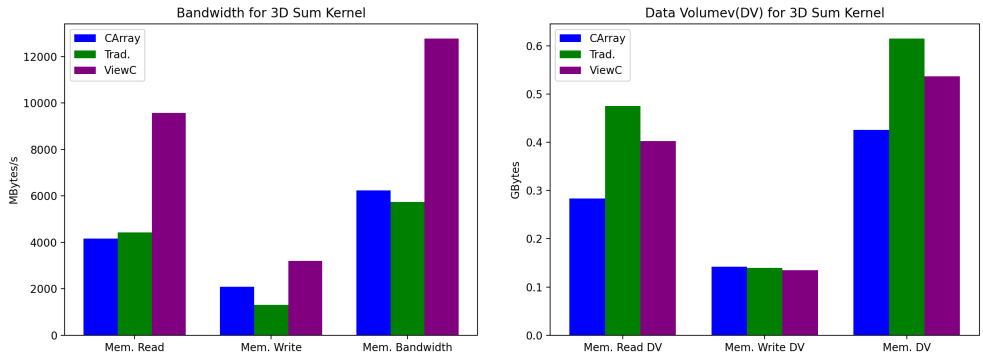

Figure 4.24. Memory bandwidth and data volume for the 3D sum kernel. Both the `CArray` and the `ViewCArray` outperformed the traditional `C++` array by about 40%, yet the bandwidths rate are on the opposite side of the spectrum. The `ViewCArray` has the highest bandwidth while the `CArray` has the lowest. The traditional array has the highest data volume overall.

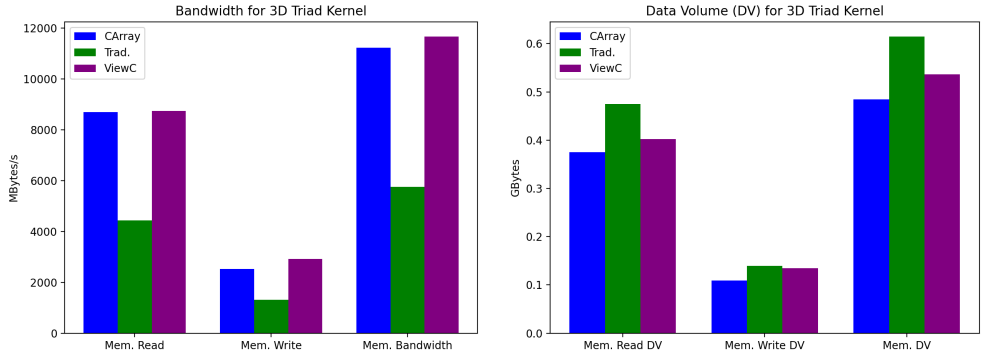

Figure 4.25. Memory bandwidth and data volume for the 3D triad kernel. Both the `CArray` and the `ViewCArray` outperformed the traditional `C++` array by about 40%, yet the bandwidths rate are on the opposite side of the spectrum. The `ViewCArray` has the highest bandwidth while the `CArray` has the lowest. The traditional array has the highest data volume overall.

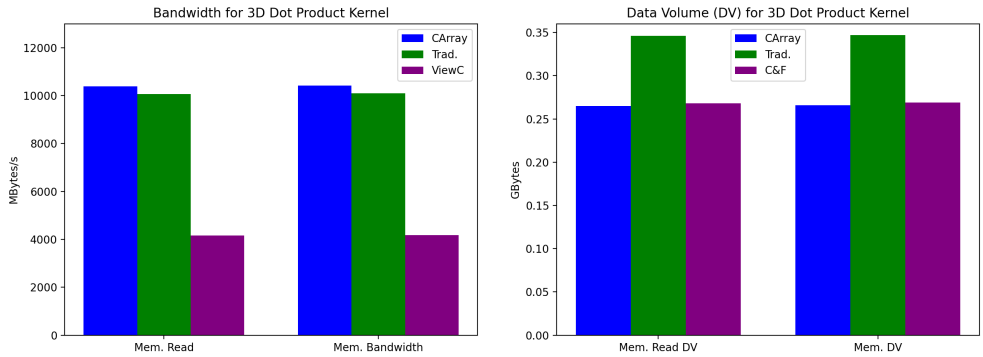

Figure 4.26. Memory bandwidth and data volume for the 2D dot product. Here the 3d dot product. Both the MATAR `View` and `CArray` have nearly identical data volume but significantly varying bandwidth. In this kernel both performed about 24% than the traditional array.

4.5.1.2 Matrix Operations Performance

Tensor products are a common computation in many numerical methods (Morgan, Kenamond, Burton, Carney, and Ingraham, 2013; **?**) therefore a timing study on a simple yet heavily used operation such as a matrix-matrix multiply (MMM) and matrix-vector (MVP) product is suitable. The matrix is size 3200×3200 ($3200^2 = 10{,}240{,}000$), and the vector is 3200×1.

Both the MMM amd MVP test includes an extra example: a MATAR `CArray` $\times$ `FArray`. In matrix multiplication, the right matrix is traversed column-wise as shown in figure 4.27, which matches the access pattern of the matrix matches the memory layout of a `Fortran` array.

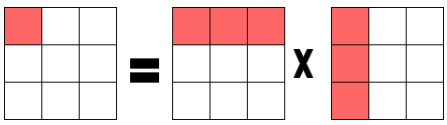

Figure 4.27. Figure showing the order of a matrix-matrix multiply. Each element of the resulting matrix is the inner product on a row in the left matrix and column in the right matrix.

Table 4.11 shows the run time for the MMM test. The combination of a `CArray` and `FArray` performed the fastest while the `CArray` performed the slowest.

Although the `CArrays` are contiguous in memory, it was expected that the `CArray` performed poorly. In order to maximize performance with MATAR `CArrays`, the access patters should match how the `CArray` is unrolled in memory - row wise. However, the right matrix in a MMM is traversed column wise (recall figure 4.27.) Elements already loaded into cache are not used and more memory loads are needed in order to obtain the column elements needed for each matrix product computation. Figure 4.29 illustrates the actual access pattern for the right matrix *does not* match the expected access pattern of the `CArray` memory layout. In fact, the access pattern is identically the memory layout pattern for the `FArray`. We can see from figure 4.28 that the `CArray` test had the highest amount of read data volume and overall data volume, validating the extra memory fetches needed to obtain for the correct elements. More memory loads are needed per operation leading to a slower run time. The significant slow down in the `CArray` test raises a key point that contiguous memory alone does not guarantee better performance; contiguous memory *and* matching access pattern will be the most performant. The traditional C++ arrays are scattered throughout global memory, but perhaps the values that were loaded into cache were the ones needed for the MMM.

Continuing the analysis of the bandwidth and data volume from figure 4.28, we see the `C` $\times$ `F` had the highest bandwidth, but the data volume closely matched the traditional array.

Table 4.11. Timing results for Matrix-Matrix Multiply (3200x3200) in seconds. The combination matrices are the fastest while the `CArray` ended up being the slowest. This is likely due to the expected row-access that the `CArray` layout expects. This test showed that accessing pattern and contiguous memory layout is key for the best performance.

Data Structure	Avg.	Min	Max.
Trad. Array	88.5457	88.3022	89.7648
CArray	145.7483	145.7303	145.7796
C x F	39.7292	39.6534	39.9351

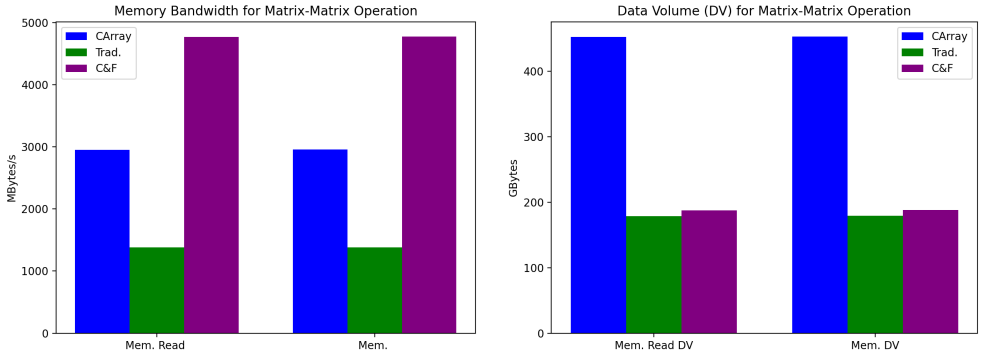

Figure 4.28. Memory bandwidth and data volume for the matrix-matrix multiply. This figure supports the claim that matching access pattern and contiguous memory layout is vital for a performance benefit. The `CArray` has the highest data volume (read and overall). The access pattern for the right matrix *does not* match the memory layout of the `CArray` and in turn, more memory loads were needed in order to obtain the correct values for the operation.

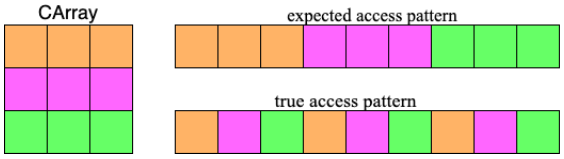

Figure 4.29. The figure shows a 2D `CArray`. As explained in section 3, the memory layout for a `CArray` is contiguous in memory row-wise. In order to maximize cache usage, the access pattern should be row by row. However, for a matrix-matrix multiply operation, the right matrix traverses the columns. Elements already loaded into cache are ignored and more memory loaded are needed per calculation. The bottom array shows the actual access pattern for the latter matrix.

Now we look at the MVP product example. For the MATAR examples, the matrix is a `CArray` and we test the vector as a `FArray` and `CArray`. In reality we don't expect to see a significant difference (unlike the MMM test) because the memory allocation and access pattern for a 1 dimensional `FArray` and `CArray` is the same. In tables 4.12 - 4.13, traditional array stands as a dynamically allocated 2D array (matrix) and 1D array (vector) `CArray` results are for the `CArray` vector, `C×F` is the `CArray` 2D matrix and the `FArray` vector.

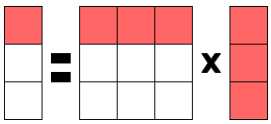

Figure 4.30. Figure showing the order of a matrix-vector product. Each element of the resulting vector is the result of an inner product with each row of the matrix and the column vector. With the vector being a column, we are inclined to think a `FArray` is more suitable. But for a 1D case both the memory allocation and access pattern for the `FArray` and `CArray` are the same.

Table 4.12. Timing results for Matrix-Vector Product (matrix=3200x3200) in milliseconds. The `CArray` and `FArray` vector combination ended up performing the fastest. Both MATAR data structures were faster than the traditional 2D `C++` array.

Data Structure	Avg.	Min	Max.	% Diff.
Trad. Array	12.91	12.83	13.20	-
CArray	12.24	12.10	13.66	-5.18
C x F	12.03	11.98	13.20	-6.81

Table 4.13. Timing results for Matrix-Vector Product (matrix=4096x4096) in milliseconds. The `CArray` and `FArray` vector combination ended up performing the fastest. Here we see a greater difference in run time between the `CArray` and `FArray` vector. Both MATAR data structures were faster than the traditional 2D `C++` array.

Data Structure	Avg.	Min	Max.	% Diff.
Trad. Array	20.86	20.45	38.18	-
CArray	20.05	20.00	21.16	-3.88
C x F	19.54	19.44	20.43	-6.32

Table 4.12 shows the results for for a matrix size 3200 $\times$ 3200 and table 4.13 for matrix size 4096 $\times$ 4096. Both MATAR data structures are faster than the

traditional array, but we see a deviance between the `CArray` and `FArray` vector. The `C× F` stays about 6% faster while the `CArray` drops from a 5% to a 3%. As mentioned, we should not expect a difference between them as a 1D vector because the memory layout and access pattern is the same. In practice, if the data access pattern is known, there should be some consideration for creating the appropriate data structure (i.e `FArray` compared to a `CArray`).

Figure 4.31 shows the bandwidth and data volume for the 3200 size MVP operation. We could see that although the combination `C×F` has a higher bandwidth, it has nearly identical data volume to the `CArray`. The data transfer rate is the same for the 1D MATAR data structures, which is expected since a 1D array is unrolled and allocated identically. Unfortunately the figures do not offer an explanation into the different run time between the `CArray` and `C×F`.

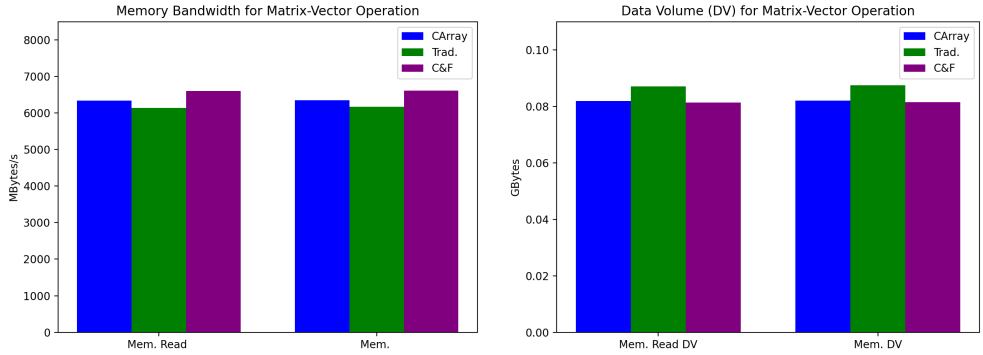

Figure 4.31. Memory bandwidth and data volume for the matrix-vector product (matrix 3200x3200). We see the traditional array had the lowest bandwidth but highest data volume. The `CArray` and `C×F` had identical data volume transfer rates for read and overall, which is expected since a 1D MATAR `FArray` and `CArray` data structure is allocated the same.

4.5.1.3 Sparse Data Structures

One of MATAR's main contributions is the ability to efficiently represent sparse data structures. The need for sparse data structures is shown in numerical methods such as (Chiravalle, Barlow, and Morgan, 2020; **?**; **?**). We see that not every cell has the same amount of nodes. Therefore, having storing the number of nodes will lead to a "ragged" structure.

We perform a STREAM benchmark on a MATAR `Ragged-Right` array and a ragged 2 dimensional traditional `C++` array. The rows are set at 64,000 elements while the columns were set to vary randomly using `rand()` from 6 to 20,004; the average number of columns each row had was 10023.7 for this particular test. Table 4.14 displays the run time and table 4.15 displays the double precision flops and vectorization. The `Trad. Array` is a dynamically allocated 2D array with varying columns.

Table 4.14. Ragged-Right (RR) STREAM Benchmark Test. Units are in seconds.

Data Structure	Tests	Avg.	Min.	Max.	% Diff.
RR MATAR	Copy	0.8187	0.8181	0.8194	-1.94
	Scale	0.8183	0.8178	0.8190	-2.13
	Sum	1.1754	1.1750	1.176	1.87
	Triad	1.190	1.189	1.192	-3.34
RR Trad. Array	Copy	0.8404	0.8397	- 0.8412	
	Scale	0.8399	0.8393	0.8406	-
	Sum	1.165	1.156	1.242	-
	Triad	1.240	1.170	1.308	-

Table 4.15. Double Precision Flops and Advanced Vectorization Extension for the Ragged-Right Stream Benchmark. We see both the traditional array and the `Ragged-Right` were able to vectorize their flops.

Data Structure	Tests	DP	AVX
RR MATAR	Copy	2.583e-05	0
	Scale	356.6631	356.6102
	Sum	280.7025	280.6606
	Triad	886.3353	886.2030
RR Trad. Array	Copy	1.273e-06	0
	Scale	237.2493	237.2138
	Sum	463.4686	463.3993
	Triad	423.3059	423.2427

The MATAR `Ragged-Right (RR)` data structure is faster than the traditional array in the copy, scale and triad kernel showing an increase of about 2%. Since the traditional array is dynamically allocated and every row has a varied column size, the data may be more scattered throughout memory leading to a slower performance.

Both the traditional array and the MATAR `RR` were able to vectorize all the double precision flops for the the scale, sum and triad kernel. `Ragged-Right`s are initialized and accessed like a 2D object `(i,j)` where `i` is the fixed number of rows and `j` is the varying column size for each row. In order to collapse two for loops using pragma directives, the dimensions need to be constant. Therefore a `#pragma omp` directive is place in the inner loop.

Figures 4.32 - 4.35 show the memory bandwidth and data volume plots. The figures show some inconsistency between the bandwidth, data volume and run time, similar to the 3D stream benchmark. As we saw in table 4.14, the `Ragged-Right` performed about 2% slower, and it is the only kernel where the `Ragged-Right` had a lower bandwidth than the traditional array (see figure 4.34). Yet, the data volume for the sum kernel in lower than the traditional array. In

the copy and scale kernels,(figures 4.32 - 4.33) , the bandwidth is higher for the `Ragged-Right` but the data volume is lower. We associated a lower data volume with a decrease in run time in the MMM and MVP, but in this test case for the sum and triad kernel, the assumption does not hold. It is unclear why the sum kernel is the only test where the `Ragged-Right` is slower. Especially since the `Ragged-Right` showed the best speed up in the triad kernel, which is similar to the sum kernel.

Overall, the `Ragged-Right` array were faster and the in-depth analysis regarding the data volume and bandwidth is limited by how the profiling tool is using the hardware counters.

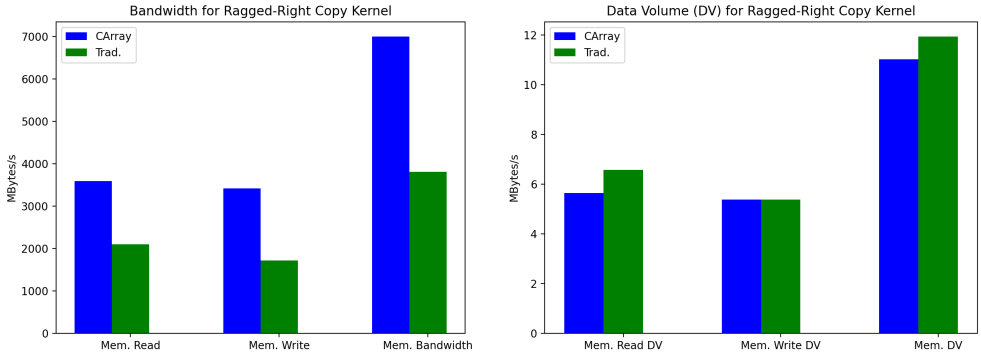

Figure 4.32. Memory bandwidth and data volume for the Ragged-Right(RR) copy kernel. The RR is 2% faster than the traditional array. We see that the RR has higher bandwidth for all read, write and overall but lower data volume.

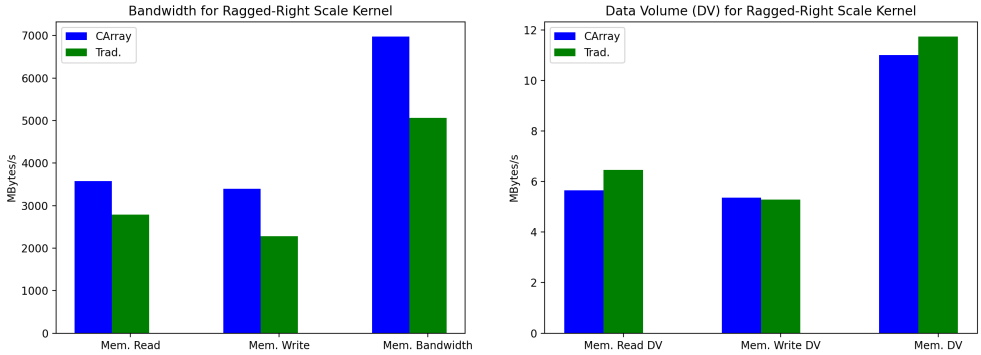

Figure 4.33. Memory bandwidth and data volume for the Ragged-Right(RR) scale kernel. The RR is 2% faster than the traditional array. We see the bandwidth and data volume follow the same trend as the copy kernel. Where the RR was also 2% faster.

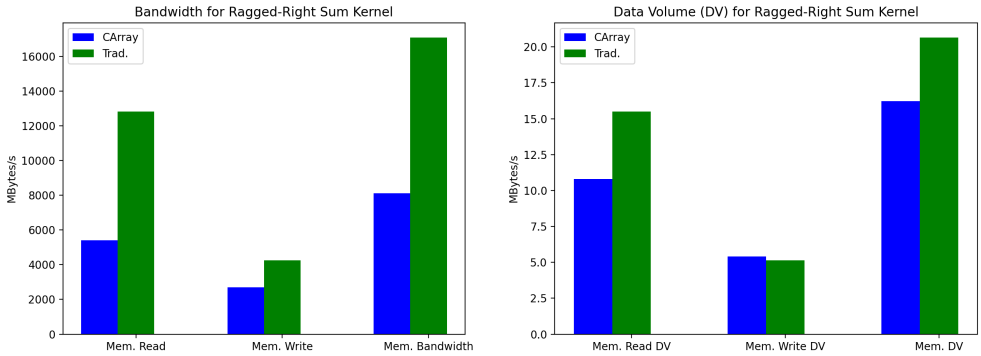

Figure 4.34. Memory bandwidth and data volume for the Ragged-Right(RR) sum kernel. This is the only test where the RR is 2% slower. The bandwidth is significantly higher for the traditional array, but the data volume remained lower for the RR. It is unclear why this kernel was the only one slower.

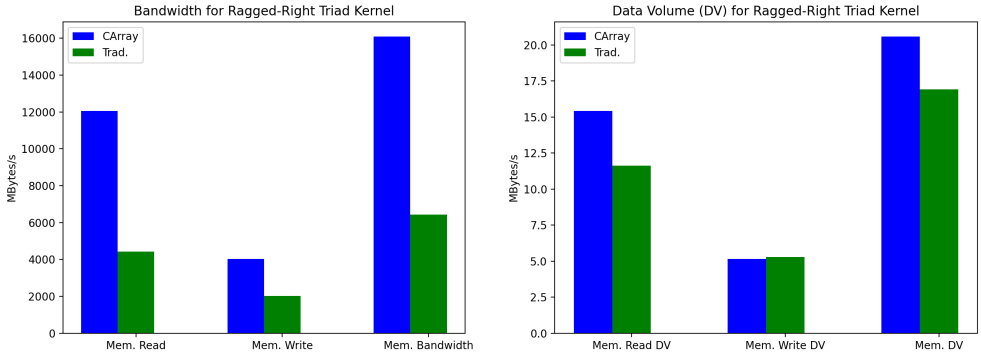

Figure 4.35. Memory bandwidth and data volume for the Ragged-Right(RR) triad kernel. The RR is about 3% faster- the largest speed up compared to the other test. However the bandwidth and data volume are higher for RR than the traditional array.

4.5.2 KOKKOS Performance

Tables 4.21 and 4.22 show the results when running the STREAM benchmark tests with our MATAR Kokkos data structures on the GPU and CPU. We show that our performance results are almost identical to using a traditional Kokkos View. For one dimensional views with is obvious because the data structures being compared are identical. In fact, this is also true for our 3D example because Kokkos does a similar data conversion to one dimension with their view allocation. Our breakthrough in doing these performance tests is discovering the importance of data layouts and loop indexing on the GPU. Kokkos provides many ways to control the data layout of the Kokkos Views on the GPU. Additionally, Kokkos allows the users to handle the order in which loop lambda arguments are oriented. All of these play a factor in performance on the GPU, and MATAR adds additional options with by choosing `CArrays` versus `FArrays`. Even with optimal GPU data layout, loop indexing matters. A Kokkos lambda of `(int i, int j, int k)` will not give the same performance as `(int k, int j, int i)` (assuming access to the same data structure). Using MATAR, we can keep this optimization complexity hidden by simply allowing the user to choose between `Arrays` and `FArrays` for performance gains, depending on the target architecture. The Kokkos optimization within MATAR will be specific to whichever data layout

is being chosen.

As a result of our investigation into the optimal layout combinations, we discovered that mathematical operations cannot be optimized under a single data layout formula. For example, simple multiplications of array elements perform best with our `FArrays` (assuming default Kokkos settings for the GPU). However, a matrix multiplication performs best multiplying a `FArray` by a `FArray`. MATAR views are key in facilitation this optimization because a data structure can be declared as one layout, but have a view of the same data in a different layout that can be used in differing operations, depending on which layout gives better performance. The results for a 2 dimensional matrix multiplication are shown in tables REF REF. The caveat to this is that the user has to take special care to ensure the operations are still valid with the "new" access pattern. For example, a matrix multiply of AxB would need to become $(B^T x A^T)^T$ if A and B are not symmetric to ensure the same result with the "opposite" layout since the opposite layout of A is A^T.

Table 4.16. 1-dimensional BabelStream benchmark results with Kokkos (CPU) (size 16,777,216). Time units are seconds. *Benchmark was run on an Intel Skylake Gold.*

Data Structure	Tests	Avg.	Min.	Max.
	Copy	0.00176	0.00181	0.00177
	Scale	0.00184	0.00187	0.00186
FArrayKokkos	Sum	0.00250	0.00257	0.00252
	Triad	0.00241	0.00999	0.00250
	Dot Prod.	0.00128	0.00159	0.00130
	Copy	0.00177	0.00893	0.00186
	Scale	0.00185	0.00199	0.00257
ViewFArrayKokkos	Sum	0.00241	0.00285	0.00247
	Triad	0.00241	0.00285	0.00247
	Dot Prod.	0.00129	0.00137	0.00133
	Copy	0.00176	0.00188	0.00178
	Scale	0.00186	0.00190	0.00187
CArrayKokkos	Sum	0.00250	0.00261	0.00252
	Triad	0.00245	0.00248	0.00246
	Dot Prod.	0.00129	0.00791	0.00137
	Copy	0.00176	0.00188	0.00178
	Scale	0.00186	0.00828	0.00194
ViewCArrayKokkos	Sum	0.00250	0.00269	0.00252
	Triad	0.00245	0.00249	0.00246
	Dot Prod.	0.00129	0.00136	0.00130
	Copy	0.00174	0.00177	0.00175
	Scale	0.00186	0.00190	0.00188
Kokkos View	Sum	0.00250	0.00254	0.00252
	Triad[141]	0.00243	0.00833	0.00250
	Dot Prod.	0.00129	0.00146	0.00130

Table 4.17. 3-dimensional BabelStream benchmark results with Kokkos (CPU) (size 134,217,728, i.e., 512 elements in each dimension). Time units are seconds. *Benchmark was run on an Intel Skylake Gold.*

Data Structure	Tests	Avg.	Min.	Max.
FArrayKokkos	Copy	0.01589	0.01598	0.01593
	Scale	0.00298	0.00298	0.00298
	Sum	0.00388	0.00428	0.00389
	Triad	0.00388	0.00390	0.00388
	Dot Prod.	0.01257	0.01280	0.01267
CArrayKokkos	Copy	0.00298	0.00298	0.00298
	Scale	0.00298	0.00299	0.00298
	Sum	0.00387	0.00388	0.00387
	Triad	0.00387	0.00388	0.00387
	Dot Prod.	0.01254	0.01278	0.01263
Kokkos View	Copy	0.00298	0.00298	0.00298
	Scale	0.00298	0.00298	0.00298
	Sum	0.00384	0.00384	0.00384
	Triad	0.00384	0.00385	0.00384
	Dot Prod.	0.01247	0.01264	0.01253

Table 4.18. 1-dimensional BabelStream benchmark results with Kokkos (CPU) (size 16,777,216). Time units are seconds. *Benchmark was run on a POWER9.*

Data Structure	Tests	Avg.	Min.	Max.
	Copy	0.00176	0.00181	0.00177
	Scale	0.00184	0.00187	0.00186
FArrayKokkos	Sum	0.00250	0.00257	0.00252
	Triad	0.00241	0.00999	0.00250
	Dot Prod.	0.00128	0.00159	0.00130
	Copy	0.00177	0.00893	0.00186
	Scale	0.00185	0.00199	0.00257
ViewFArrayKokkos	Sum	0.00241	0.00285	0.00247
	Triad	0.00241	0.00285	0.00247
	Dot Prod.	0.00129	0.00137	0.00133
	Copy	0.00176	0.00188	0.00178
	Scale	0.00186	0.00190	0.00187
CArrayKokkos	Sum	0.00250	0.00261	0.00252
	Triad	0.00245	0.00248	0.00246
	Dot Prod.	0.00129	0.00791	0.00137
	Copy	0.00176	0.00188	0.00178
	Scale	0.00186	0.00828	0.00194
ViewCArrayKokkos	Sum	0.00250	0.00269	0.00252
	Triad	0.00245	0.00249	0.00246
	Dot Prod.	0.00129	0.00136	0.00130
	Copy	0.00174	0.00177	0.00175
	Scale	0.00186	0.00190	0.00188
Kokkos View	Sum	0.00250	0.00254	0.00252
	Triad	0.00243	0.00833	0.00250
	Dot Prod.	0.00129	0.00146	0.00130

Table 4.19. 1-dimensional BabelStream benchmark results with Kokkos (GPU) (size 16,777,216). Time units are seconds.

Data Structure	Tests	Avg.	Min.	Max.
FArrayKokkos	Copy	0.00041	0.00045	0.00043
	Scale	0.00041	0.00045	0.00043
	Sum	0.00049	0.00049	0.00049
	Triad	0.00049	0.00049	0.00049
	Dot Prod.	0.00043	0.00045	0.00044
ViewFArrayKokkos	Copy	0.00038	0.00041	0.00039
	Scale	0.00038	0.00041	0.00039
	Sum	0.00049	0.00049	0.00049
	Triad	0.00049	0.00049	0.00049
	Dot Prod.	0.00043	0.00044	0.00043
CArrayKokkos	Copy	0.00039	0.00039	0.00039
	Scale	0.00039	0.00039	0.00039
	Sum	0.00049	0.00050	0.00049
	Triad	0.00049	0.00050	0.00049
	Dot Prod.	0.00043	0.00044	0.00043
ViewCArrayKokkos	Copy	0.00038	0.00039	0.00039
	Scale	0.00038	0.00039	0.00039
	Sum	0.00038	0.00039	0.00039
	Triad	0.00049	0.00049	0.00049
	Dot Prod.	0.00043	0.00044	0.00043
Kokkos View	Copy	0.00038	0.00039	0.00039
	Scale	0.00038	0.00039	0.00039
	Sum	0.00049	0.00049	0.00049
	Triad	0.00049	0.00050	0.00049
	Dot Prod.	0.00043	0.00044	0.00043

Table 4.20. 3-dimensional BabelStream benchmark results with Kokkos (GPU) (size 134,217,728, i.e., 512 elements in each dimension). Time units are seconds.

Data Structure	Tests	Avg.	Min.	Max.
FArrayKokkos	Copy	0.00298	0.00298	0.00298
	Scale	0.00298	0.00298	0.00298
	Sum	0.00388	0.00428	0.00389
	Triad	0.00388	0.00390	0.00388
	Dot Prod.	0.01257	0.01280	0.01267
CArrayKokkos	Copy	0.00298	0.00298	0.00298
	Scale	0.00298	0.00299	0.00298
	Sum	0.00387	0.00388	0.00387
	Triad	0.00387	0.00388	0.00387
	Dot Prod.	0.01254	0.01278	0.01263
Kokkos View	Copy	0.00298	0.00298	0.00298
	Scale	0.00298	0.00298	0.00298
	Sum	0.00384	0.00384	0.00384
	Triad	0.00384	0.00385	0.00384
	Dot Prod.	0.01247	0.01264	0.01253

Table 4.21. 1-dimensional STREAM Benchmark Test with Kokkos (GPU) (size 16,777,216). Units are milliseconds.

Data Structure	Tests	Avg.	Min.	Max.
FArrayKokkos	Copy	0.41	0.39	0.41
	Scale	0.36	0.39	0.39
	Sum	0.50.	0.50	0.50
	Triad	0.50	0.50	0.50
	Dot Prod.	0.29	0.29	0.30
Trad. Kokkos View	Copy	0.41	0.39	0.41
	Scale	0.36	0.39	0.39
	Sum	0.50.	0.50	0.50
	Triad	0.50	0.50	0.50
	Dot Prod.	0.29	0.29	0.30
ViewFArrayKokkos	Copy	21.0009	20.819	22.252
	Scale	21.532	21.404	21.834
	Sum	30.548	30.470	30.834
	Triad	30.710	30.666	30.850
	Dot Prod.	24.400	24.304	25.400

Table 4.22. 3-dimensional STREAM Benchmark Test with Kokkos (GPU) (size 256) Units are milliseconds.

Data Structure	Tests	Avg.	Min.	Max.
FArrayKokkos	Copy	3.00	2.98	3.15
	Scale	2.98	2.98	2.99
	Sum	3.88	3.87	3.88
	Triad	3.88	3.87	3.88
	Dot Prod.	12.02	11.96	12.09
Trad. Kokkos View	Copy	3.00	2.98	3.15
	Scale	2.98	2.98	2.98
	Sum	3.84	3.83	3.84
	Triad	3.84	3.83	3.84
	Dot Prod.	11.80	11.67	11.73
ViewFArrayKokkos	Copy	3.00	2.98	3.15
	Scale	2.98	2.98	2.98
	Sum	3.84	3.83	3.84
	Triad	3.84	3.83	3.84
	Dot Prod.	11.80	11.67	11.73

4.6 Summary and Conclusion

We have introduced a new `C++` library for data oriented data structures for dense and sparse representation. These data structures are allocated contiguous in memory and multi-dimensional data is unrolled as a one-dimensional array

based on `Fortran` (i.e. `FArray`) or `C++` memory layout (i.e. `CArray`). Various test are presented in order to validate the idea that contiguous memory layout paired with sequential data access improves run time performance. On the CPU, MATAR data structures were competitive to traditionally dynamically allocated arrays, especially for higher-dimensional data. The MMM test raised an important observation that if the access pattern does not match the data layout the application may run slower. This observation was later reinforced in the GPU section with the layout order for `Kokkos`' views, `CArrayKokkos`, and `FArrayKokkos` where it the run time significantly varied with different loop indexing. Moreover, it was shown that MATAR dynamic sparse data structures, the `DynamicRaggedRight`, are nearly twice as fast when being used compared to a linked-list in the HOSS contact-detection algorithm. A simpler version of the RaggaedRight MATAR data structure was implemented in the main HOSS code, resulting in a 40% reduction in run-time in one of the most expensive functions and a 20% reduction in run-time overall for a 2D test case. Finally, MATAR data structures present a competitive advantage over traditional arrays and linked-list due to the performance gain from contiguous memory, portability to other architectures through `Kokkos`, and the addition of sparse data representation.

Although the presented paper focuses on simple benchmarks to validate the theory behind data-oriented design, further work remains on obtaining consistent vectorization for MATAR array dimension (1D to 6D) and compilers. A theoretical study and a deeper focus on the performance portability aspect can be explored with quantitative measurements such as the Pennycook metric (Pennycook, Sewall, and Lee, 2016b).

Chapter 5

Conclusion

Presented in this dissertation is the culmination of several publications over the course of my PhD career. The work shown focuses on parallel and scientific computing, specifically in the hydrodynamics and computational fluids discipline. Extensive efforts have been made to improve current methods for Eulerian-based algorithms in adaptive mesh refinement. Furthermore, studies have been done upon these codes to investigate areas of high performance computing that are not fully realized and have not fully been exploited. These areas included vectorization, performance portability, and memory utilization. Additional work has been performed to present software libraries that are intended to give users all of the capabilities of high performance computing, without the expertise in all areas. A common theme is a separation of duties, allowing each component to be optimized to increase developer efforts when achieving performant codes.

References

(1891). Ueber die stetige abbildung einer line auf ein flächenstück. *Mathematische Annalen 38*, 459–460.

(1989). Local adaptive mesh refinement for shock hydrodynamics. *Journal of Computational Physics 82*, 64–84.

Alcrudo, F. and P. Garcia-Navarro (1993). A high-resolution Godunov-type scheme in finite volumes for the 2D shallow-water equations. *International Journal for Numerical Methods in Fluids 16*, 489–505.

Anderson, T., D. Culler, and D. Patterson (1995). A case for NOW (networks of workstations). *Micro IEEE 15*(1), 54–64.

Barlow, A., N. Morgan, and M. Shashkov (2019). Constrained optimization framework for interface-aware sub-scale dynamics discrete closure model for multimaterial cells in lagrangian cell-centered hydrodynamics. *Computers & Mathematics with Applications 78*(2), 541–564.

Becker, D., J. Salmon, D. Sevarese, and et al. (1999). *How to Build a Beowulf: A Guide to the Implementation and Application of PC Clusters*. MIT Press.

Berger, M. J. and R. J. LeVeque (1998). Adaptive mesh refinement using wave-propagation algorithms for hyperbolic systems. *SIAM Journal on Numerical Analysis 35*(6), 2298–2316.

Chiravalle, V. P., A. Barlow, and N. R. Morgan (2020). 3d cell-centered hydrodynamics with subscale closure model and multi-material remap. *Computers & Fluids*, 104592.

Cormen, T. H., C. E. Leiserson, R. L. Rivest, and C. Stein (2009). *Introduction to algorithms*. MIT press.

Davis, S., T. F. D. Schemes, and A. Viscosity (1984). ICASE report no. 84-20. *National Aeronautics and Space Administration. USA*.

Davis, S. F. (1987). A simplified TVD finite difference scheme via artificial viscosity. *SIAM journal on scientific and statistical computing 8*(1), 1–18.

Deakin, T., S. McIntosh-Smith, and W. Gaudin (2016). Many-core acceleration of a discrete ordinates transport mini-app at extreme scale. In *International Conference on High Performance Computing*, pp. 429–448. Springer.

Deakin, T., J. Price, M. Martineau, and S. McIntosh-Smith (2016). Gpu-stream v2. 0: Benchmarking the achievable memory bandwidth of many-core processors across diverse parallel programming models. In *International Conference on High Performance Computing*, pp. 489–507. Springer.

Dunning, D., W. Marts, R. Robey, and P. Bridges (2020). Adaptive mesh refinement in the fast lane. *Journal of Computation Physics 406*.

Dunning, D., N. Morgan, J. Moore, E. Nelluvelil, T. Tafolla, and R. Robey (2020). Matar: A performance portability and productivity implementation of data-oriented design with kokkos. *Tech. Rep. LA-UR-20-27630*.

Dunning, D. J., R. Robey, J. A. Kuehn, and J. Cook (2020). A performance analysis of phantom-cell adaptive mesh refinement on cpus and gpus. *Journal of Computing and Information Science in Engineering*.

Edwards, H. C., C. R. Trott, and D. Sunderland (2014). Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of Parallel and Distributed Computing 74*(12), 3202 – 3216. Domain-Specific Languages and High-Level Frameworks for High-Performance Computing.

Ekmecic, I., I. Tartalja, and V. Milutinovic (1996). A survey of heterogeneous computing: concepts and systems. *Proceedings of the IEEE 84*(8), 1127–1144.

Fabian, R. (2018). *Data-oriented design: software engineering for limited resources and short schedules*. Richard Fabian.

Garcia, R. (2006). Boost multidimensional array library.

Garimella, R. V. and R. W. Robey (2017). A comparative study of multi-material data structures for computational physics applications. Technical report, Los Alamos National Lab.(LANL), Los Alamos, NM (United States).

Gutierrez Andres, J., J. Lhomme, A. Weisberger, A. Cooper, B. Gouldby, and

J. Mullet-Marti (2009). Testing and application of a practical new 2D hydrodynamic model. *Proceedings of the FloodRisk 2008 Conference*.

Harrell, S. L., J. Kitson, R. Bird, S. J. Pennycook, J. Sewall, D. Jacobsen, D. N. Asanza, A. Hsu, H. C. Carrillo, H. Kim, et al. (2018). Effective performance portability. In *2018 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, pp. 24–36. IEEE.

Hollman, D., B. Lelbach, H. C. Edwards, M. Hoemmen, D. Sunderland, and C. R. Trott (2019). mdspan in c++: A case study in the integration of performance portable features into international language standards. In *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, pp. 60–70. IEEE.

Knight, E. E., E. Rougier, Z. Lei, B. Euser, V. Chau, S. H. Boyce, K. Gao, K. Okubo, and M. Froment (2020). Hoss: an implementation of the combined finite-discrete element method. *Computational Particle Mechanics*, 1–23.

Kuehn, J. A. (2019, July 15-18,). An approach to quantifying representativeness of workload proxies. In *JOWOG-34*, Albuquerque, NM. Sandia National Laboratories.

Leveque, R. (2002). *Finite volume methods for hyperbolic problems*. Cambridge University Press.

Lhomme, J., J. Gutierrez-Andres, A. Weisgerber, M. Davison, J. Mulet-Marti, A. Cooper, and B. Gouldby (2010). Testing a new two-dimensional flood modelling system: analytical tests and application to a flood event. *Journal of Flood Risk Management 1*(3), 33–51.

Lipnikov, K., M. Shashkov, and D. Svyatskiy (2006). The mimetic finite difference discretization of diffusion problem on unstructured polyhedral meshes. *Journal of Computational Physics 211*(2), 473–491.

Llopis, N. (2009, Dec). Data-oriented design (or why you might be shooting yourself in the foot with oop).

McCalpin, J. D. (2016). Memory bandwidth and system balance in HPC systems. *Invited talk, Supercomputing*.

McIntosh-Smith, S., M. Boulton, D. Curran, and J. Price (2014, 1). On the performance portability of structured grid codes on many-core computer architectures. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, Volume 8488 LNCS of *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, Germany, pp. 53–75. Springer Verlag.

Morgan, N. R., M. A. Kenamond, D. E. Burton, T. C. Carney, and D. J. Ingraham (2013). An approach for treating contact surfaces in lagrangian cell-centered hydrodynamics. *Journal of Computational Physics 250*, 527–554.

O'Brien, K., I. Pietri, R. Reddy, A. Lastovetsky, and R. Sakellariou (2017, June). A survey of power and energy predictive models in HPC systems and applications. *ACM Comput. Surv. 50*(3), 37:1–37:38.

Pennycook, S. J., J. D. Sewall, and V. W. Lee (2016a). A metric for performance portability. *CoRR abs/1611.07409*.

Pennycook, S. J., J. D. Sewall, and V. W. Lee (2016b). A metric for performance portability. *arXiv preprint arXiv:1611.07409*.

Pouderoux, J., M. Charest, M. Kenamond, and M. Shashkov (2017). 2d & 3d voronoi meshes generation with shapo. In *The 8th international conference on numerical methods for multi-material fluid flow (MULTIMAT 2017)*.

Robey, R. and Y. Zamora. *Parallel and High Performance Computing*. Manning Publications.

Robey, R. N., D. Nicholaeff, and R. W. Robey (2013). Hash-based algorithms for discretized data. *SIAM Journal on Scientific Computing 35*(4), C346–C368.

Rougier, E. and A. Munjiza (2010). Mrck_3d contact detection algorithm. In *Proceedings of 5th international conference on discrete element methods, London, UK*.

Siek, J. and A. Lumsdaine (1998, 11). The matrix template library: A generic programming approach to high performance numerical linear algebra.

Stroustrup, B. (1988). What is object-oriented programming? *IEEE software 5*(3), 10–20.

Tao, T. (2011). The shallow water wave equation and tsunami propagation. https://terrytao.wordpress.com/2011/03/13/.

Toro, E. F. (2001). *Shock-Capturing Methods for Free-Surface Shallow Flows*. John Wiley & Sons, Ltd.

Treibig, J., G. Hager, and G. Wellein (2010). Likwid: A lightweight performance-oriented tool suite for x86 multicore environments. In *Proceedings of PSTI2010, the First International Workshop on Parallel Software Tools and Tool Infrastructures*, San Diego CA.

Tseng, M.-H. (1999). Explicit finite volume non-oscillatory schemes for 2D transient free-surface flows. *International Journal for Numerical Methods in Fluids 30*, 831–843.

Veldhuizen, T. L. (1998). Arrays in blitz++. In D. Caromel, R. R. Oldehoeft, and M. Tholburn (Eds.), *Computing in Object-Oriented Parallel Environments*, Berlin, Heidelberg, pp. 223–230. Springer Berlin Heidelberg.

Wang, Y., Q. Liang, G. Kesserwani, and J. W. Hall (2011). A 2D shallow flow model for practical dam-break simulations. *Journal of Hydraulic Research 49*(3), 307–316.

Yee, H. (1987). Construction of explicit and implicit symmetric TVD schemes and their applications. *Journal of Computational Physics 68*(1), 151 – 179.