ADAPTIVE MESH REFINEMENT ON GRAPHICS PROCESSING UNITS FOR APPLICATIONS IN GAS DYNAMICS

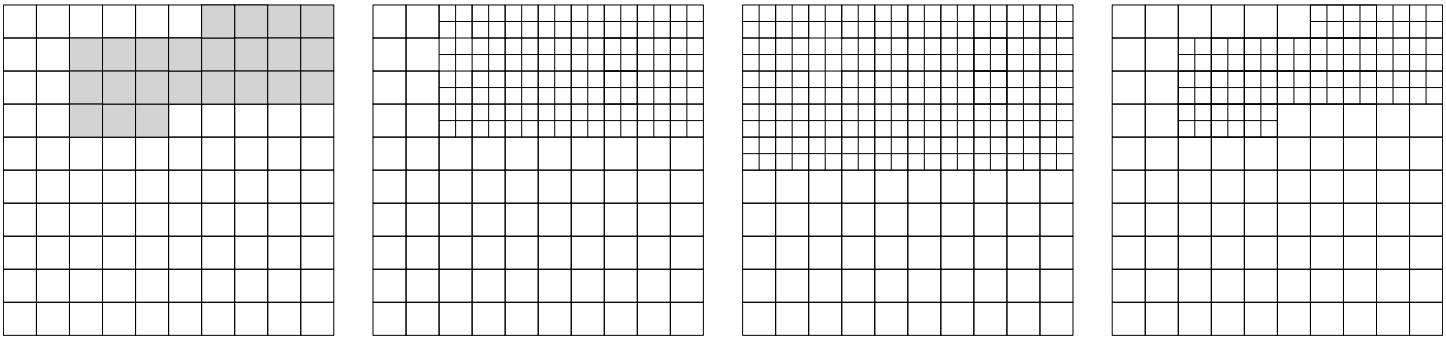
ANDREW GIULIANI AND LILIA KRIVODONOVA

Abstract. We present novel algorithms for cell-based adaptive mesh refinement on unstructured meshes of triangles on graphics processing units. Our implementation makes use of improved memory management techniques and a coloring algorithm for avoiding race conditions. The algorithm is entirely implemented on the GPU, with negligible communication between device and host. We show that the overhead of the AMR subroutines is small compared to the high order solver and that the proportion of total runtime spent adaptively refining the mesh decreases with the order of approximation. We apply our code to a number of benchmark problems as well as more recently proposed problems for the Euler equations that require extremely high resolution. We present the solution to a shock reflection problem that addresses the von Neumann triple point paradox with an accurately computed triple point location. Finally, we present the first solution on the full Euler equations to the problem of shock disappearance and self-similar diffraction of weak shocks around thin films.

1. Introduction

In recent years, graphics processing units (GPUs) have proven useful in accelerating numerical solvers for partial differential equations (PDEs) [1, 2, 3, 4]. They are popular due to their low cost and impressive compute capabilities and have been used in applications such as wave propagation [5], tsunami [6] and atmospheric modeling [7]. GPUs belong to the class of Single Instruction Multiple Data (SIMD) parallel architectures. For SIMD devices, a group of threads execute the same instruction on different data simultaneously. Not all algorithms are suitable for parallelization on this type of platform due to restrictive data access pattern requirements, race conditions, and warp divergence in control structures. Efficient computational fluid dynamics (CFD) solvers on GPUs must leverage the high floating point operation (FLOP) throughput available by optimizing memory transfers and reducing latencies [3, 8].

In this work, we discuss and implement code optimization techniques for high order finite element GPU codes that support runtime adaptive mesh refinement (AMR). Our implementation presents a number of novelties. First, it is entirely implemented on the GPU. Many AMR solvers in the literature are actually hybrid CPU-GPU solvers, whereby the main solver is implemented on the GPU and some algorithms that modify the adaptively refined mesh are offloaded onto the CPU. Second, we propose an efficient stream-compaction operation that ensures that data is contiguous in memory. Finally, we use a runtime edge



(A) Initial mesh. Shaded elements are flagged for refinement.

(B) Patch-based refinement.

(C) Block-based refinement.

(D) Cell-based refinement.

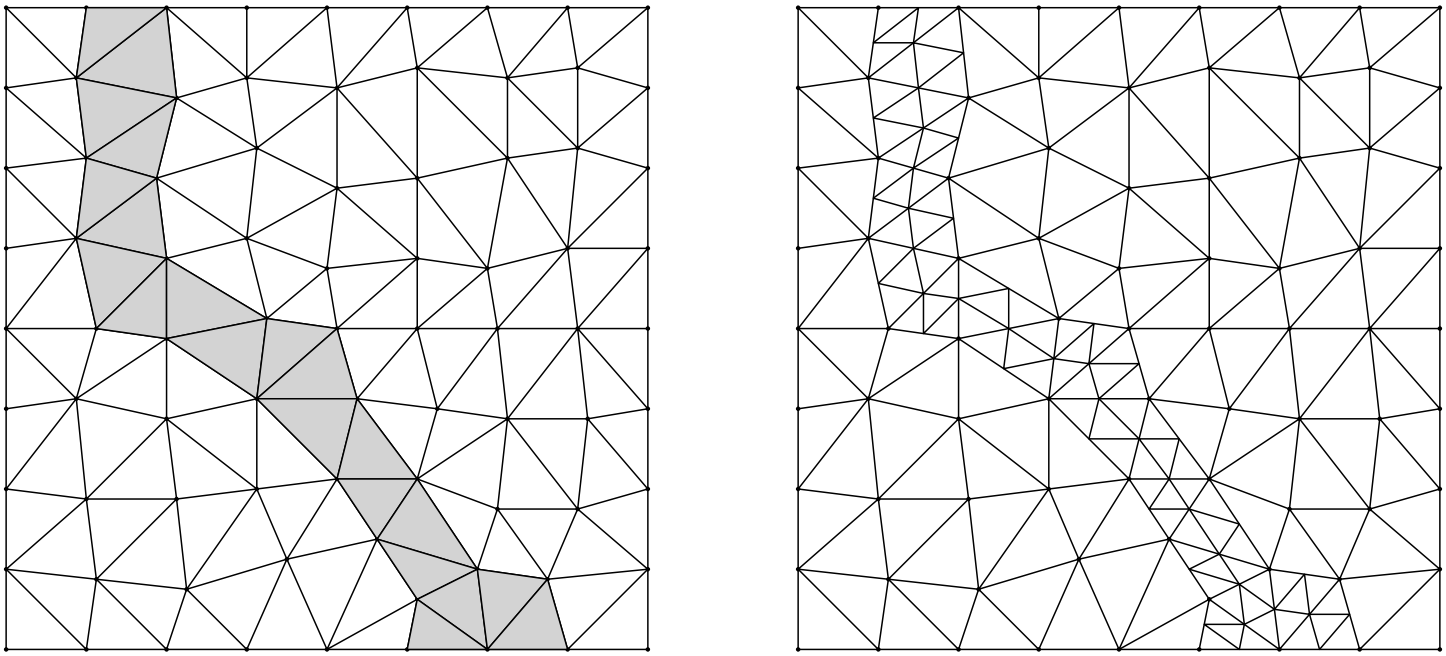
FIGURE 1. Patch-, block-, and cell-based refinement strategies on regular grids.

coloring operation for nonconforming meshes that avoids race conditions in the evaluation of an integral on all edges of the mesh. The runtime edge coloring algorithm allows the parallelization of mesh adaptation subroutines that are difficult to implement on the GPU. For example, mesh smoothing cannot efficiently be implemented on GPUs without the coloring algorithm. In fact, it is the most time consuming portion of the algorithm. We apply our optimized code to a number of computationally difficult problems in gas dynamics that are intractable without mesh adaptivity.

AMR is a technique that modifies the mesh in order to efficiently distribute computational resources over the domain. Common types of adaptivity include anisotropic adaptivity, p-, and h-refinement. Anisotropic adaptivity spatially relocates, or smooths, the geometrical nodes of the mesh [9]. P-refinement strategies aim to increase the local degree of approximation in smooth regions of the solution [10, 11]. Finally, h-refinement strategies enrich the mesh locally with new elements in order to capture fine structures of the solution or discontinuities.

Implementations of AMR are numerous and include PARAMESH [12], Chombo [13], deal.ii [14], and AMRClaw [15]. H-adaptivity has been implemented as patch-, block-, or cell-based refinement. The idea behind patch-based refinement, or component grids, is to superimpose progressively refined Cartesian grids until the desired accuracy is obtained [16] (Figure 1b). Subgrids communicate with one another and are advanced in time using local time stepping.

In block-based refinement, a predefined number of elements are grouped together into blocks [17, 18]. Refinement and coarsening operations execute on blocks of cells, rather than individual elements (Figure 1c). Only inter-block connectivity is required since the blocks are scaled versions of one another.



(A) Initial unstructured mesh of triangles. Shaded triangles are flagged for refinement.

(B) Cell-based refinement.

FIGURE 2. Cell-based refinement on an unstructured mesh of triangles.

In cell-based h-refinement, cells are refined independently of one another, which requires more connectivity data than block-based refinement (Figure 1d). Usually, parent-children relations between coarse and fine elements are organized into a quadtree or octree data structure for two- and three-dimensional codes [19]. The advantage of this approach is that fewer elements may be needed for a prescribed error tolerance and that it is well suited to unstructured meshes. We focus on cell based h-adaptivity on unstructured meshes of triangles in this work (Figure 2), though the work we present here generalizes to other AMR strategies.

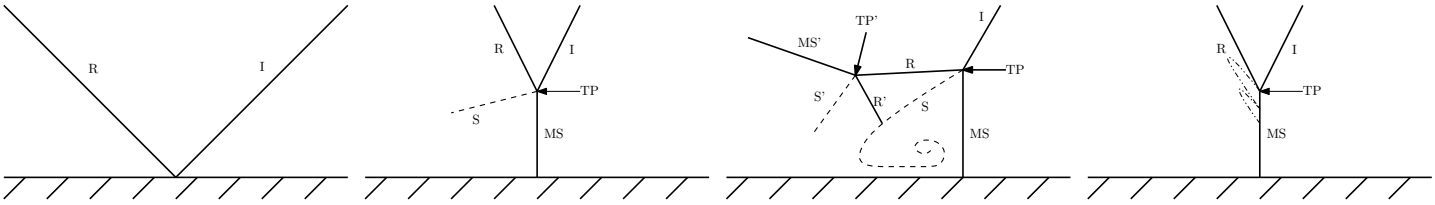
Cell-based h-adaptivity has been used extensively on serial and parallel CPU architectures in CFD codes, e.g., [20, 21, 22]. However, GPU architectures present their own challenges. During AMR, elements and their corresponding data may be added or removed from the mesh. Updating the data arrays can lead to memory management issues, e.g., ensuring that arrays contain contiguous information without excessive copying. We address this by developing a modified stream-compaction operation that results in the fewest possible number of memory transfers.

CFD codes on GPUs can also easily present race conditions when multiple threads attempt to write to the same memory location, e.g., in writing the surface contribution to the right-hand-side of two elements that share an edge or face. A suboptimal solution is extensive amounts of buffer memory [3]. In anisotropic adaptivity, a race condition can also occur when the code is modifying the position of the geometrical

vertices of the mesh. In both cases, a coloring algorithm for work scheduling is a suitable solution [8, 23, 24]. The edge coloring of the initial, conforming mesh is done in the preprocessing stage. Based on this initial coloring, we describe a fast runtime mapping from parent to children which extends the edge coloring to adaptively refined meshes (Section 3.5). The resulting edge coloring is also used in mesh smoothing subroutines that force adjacent elements to not differ by more than a prescribed difference in refinement level (Section 4.1).

An error estimator and refinement strategy are required to guide the AMR algorithm. There are few robust error estimators for transient hyperbolic problems, though a number of approaches have been proposed, e.g. Richardson extrapolation [25], solution slope [26], or residual [10]. Two possible refinement strategies are: (1) prescribing an error tolerance and attempting to attain that error using the smallest number of degrees of freedom (DOFs) (2) prescribing the number of DOFs to be used and modifying the mesh to minimize the error [27]. We note that computation of refinement indicators, as well as finding the optimal way of distributing the DOFs on the mesh can be very costly and comparable to the cost of the solver itself. In this work, we are not concerned with developing new error indicators, therefore we use one available in the literature.

We test our algorithm on a number of benchmarks as well as on two more challenging problems. The first problem is resolving Guderley Mach reflection and the second is resolving the shock disappearance point in a diffraction problem. These are less common, computationally difficult problems that are intractable without some form of mesh adaptation. Guderley Mach reflection has previously been solved on manually constructed, logically Cartesian grids. Here we present fully adaptive computations on an unstructured mesh, which allows us to obtain a more accurate position of the triple point and contribute to the body of numerical evidence for Guderley's solution. There are several self-similar reflection patterns that can result from the oblique reflection of a shock against a wedge. Regular reflection occurs when the incident (I) and reflected (R) shocks meet at the wall (Figure 3a). Single Mach reflection occurs when the point at which the incident and reflected shocks meet detaches from the wall (Figure 3b). This point is called the triple point (TP) and is connected to the wall via the Mach stem (MS). A slipline (S) also originates at the triple point. Under different wedge angles and shock strengths, a more complex reflection pattern is observed, called double Mach reflection. The reflected shock creates a second triple point (TP'), Mach stem, and slipline (S') (Figure 3c). A transient double Mach reflection is computed in Section 5.1.3. The theory of regular and Mach reflection was developed by von Neumann and allowed the prediction of the type of reflection pattern that occurs (regular or Mach reflection) based on the wedge angle and



(A) Regular reflection. (B) Single Mach reflection. (C) Double Mach reflection. (D) Guderley Mach reflection.

FIGURE 3. Regular reflection, single and double Mach reflection. The incident (I), primary and secondary reflected shocks (R, R'), Mach stems (MS, MS') and sliplines (S, S') are indicated. The sonic line in the Guderley Mach reflection case is indicated by the dashed-dotted line.

shock strength. However, some difficulty was encountered when applying the theory to weak shocks. Early experimental evidence seemed to indicate that in some parameter regimes, Mach reflection was the observed reflection pattern even though this was not allowed in von Neumann's theory. There have been a number of proposed solutions to this paradox. For example, there could be a singularity behind the triple point, invalidating assumptions in von Neumann's theory. Another solution was proposed by Guderley, where there is an expansion fan and a supersonic region behind the triple point [28]. Early experimental and numerical studies were unable to determine the correct solution. The difficulty with confirming this theory is that the reflection pattern is very small, on the order of 10^{-4}. To properly resolve this flow feature, cell sizes on the order of 10^{-6} are required in the neighborhood of the triple point. Recently, numerical and experimental evidence has suggested that Guderley's solution is correct [29, 30, 31]. We compute a Guderley Mach reflection in self-similar coordinates in Section 5.2.1. We compute a more accurate location of the triple point than previously reported in the literature. This is due to our automatic mesh refinement algorithms which allow the accurate resolution of the incident shock and Mach stem not only near the triple point, but also at the domain boundaries.

We also apply our code to the problem of self-similar shock diffraction around a thin reflecting film. The objective of this problem is to determine the point where shock disappearance occurs. Numerical evidence has shown that in transonic flows over airfoils, shock disappearance occurs in supersonic regions [32]. In contrast, this problem seems to show that the shock disappears on the sonic line. This problem has previously been solved using the unsteady transonic disturbance equations (UTSDE) in [33], which is a simpler system of PDEs than the Euler equations. To our knowledge, we provide the first solution to the problem on the full Euler equations. In order to obtain a meaningful solution, the problem required high resolution near the sonic line.

2. The discontinuous Galerkin method

Two dimensional hyperbolic conservation laws are PDEs of the form

$$\mathbf{u}_t + \nabla \cdot \mathbf{F}(\mathbf{u}) = 0, \tag{1}$$

where the solution $\mathbf{u}(\mathbf{x}, t) = (u_1, u_2, ..., u_M)^\intercal$ is defined on the spatial domain Ω such that $\mathbf{x} = (x, y)$, $\mathbf{x} \in \Omega \subset \mathbb{R}^2$, $t \in [0, T]$ where T is a final time, M is the number of equations, and $\mathbf{F}(\mathbf{u}) = (\mathbf{F_1}(\mathbf{u}), \mathbf{F_2}(\mathbf{u}))$ is the flux function. We also subject the conservation law to the initial condition

$$\mathbf{u}(\mathbf{x}, 0) = \mathbf{u}_0(\mathbf{x}),$$

and boundary conditions.

We obtain the DG method by dividing the domain Ω into a mesh of unstructured elements, e.g. triangles, where $\Omega = \bigcup_{i=0}^{N-1} \Omega_i$ and N is the number of elements in the mesh. Typically, a mesh produced by a mesh generator is conforming (Figure 4, left). During adaptive refinement, an element is split into four smaller triangles by connecting edges' midpoints, which may produce a nonconforming mesh (Figure 4, right).

We multiply equation (1) by a test function $v \in H^1(\Omega_i)$ and integrate on Ω_i in order to obtain the weak form. After using the divergence theorem, we obtain

$$\int_{\Omega_i} \mathbf{u}_t v d\Omega_i - \int_{\Omega_i} \mathbf{F}(\mathbf{u}) \cdot \nabla v d\Omega_i + \int_{\partial\Omega_i} v \mathbf{F}(\mathbf{u}) \cdot \mathbf{n} dl = 0, \quad \forall v \in H^1(\Omega_i), \tag{2}$$

where $\mathbf{n}$ is the unit outward facing normal on $\partial\Omega_i$, the element's boundary.

Each element Ω_i is mapped to the canonical triangle Ω_c, having vertices at $(0,0), (1,0), (0,1)$, using the transformation

$$\begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} x_{i,1} & x_{i,2} & x_{i,3} \\ y_{i,1} & y_{i,2} & y_{i,3} \\ 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} 1-r-s \\ r \\ s \end{pmatrix}, \tag{3}$$

where $(x_i, y_i)_{1,2,3}$ are the vertices of Ω_i in the physical space. We label the edge defined by $(0,0)$ and $(1,0)$ of the canonical triangle edge 1, $(1,0)$ and $(0,1)$ edge 2, and $(0,1)$ and $(0,0)$ edge 3. The Jacobian of the transformation is

$$J_i = \begin{pmatrix} x_{i,2} - x_{i,1} & x_{i,3} - x_{i,1} \\ y_{i,2} - y_{i,1} & y_{i,3} - y_{i,1} \end{pmatrix}. \tag{4}$$

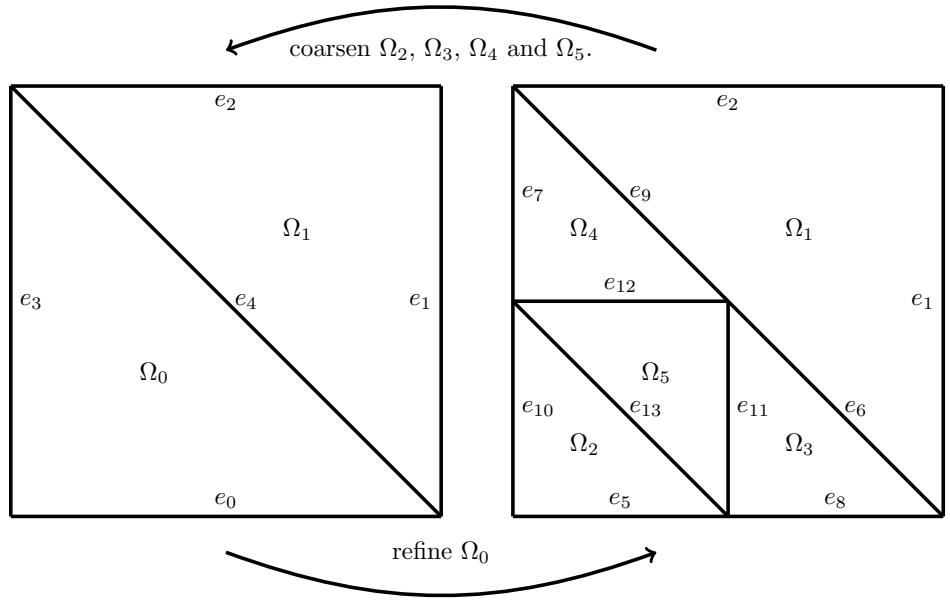


FIGURE 4. Initial mesh of two elements (left). Ω_0 is refined resulting in a nonconforming mesh (right).

We define $S^p(\Omega_c)$ to be the space of polynomials of order up to p on Ω_c, and $\{\varphi_k\}_{k=0,\cdots,N_p-1}$ to be the set of orthonormal basis functions on $S(\Omega_c)$ [34], where the number of basis functions N_p for the space of order p is $N_p = \frac{1}{2}(p+1)(p+2)$. The exact solution on element Ω_i is approximated by $\mathbf{U}_i$, which is a linear combination of the basis functions φ_k, i.e. $\mathbf{U}_i = \sum_{k=0}^{N_p^d-1} \mathbf{c}_{i,k}\varphi_k$, where $\mathbf{c}_{i,k} = [c_{i,k}^1, c_{i,k}^2, \ldots, c_{i,k}^M]^\intercal$ are referred to as the degrees of freedom. As continuity between elements is not imposed, the solution is multivalued in the boundary integral. We therefore introduce a numerical flux $\mathbf{F}^*(\mathbf{U}_i, \mathbf{U}_j)$ to allow information exchange between adjacent cells Ω_i and Ω_j. We assume that the numerical flux is consistent, monotone, and differentiable. With v chosen to be φ_k, equation (2) now becomes

$$\frac{d}{dt}\mathbf{c}_{i,k} = \frac{1}{\det J_i}\left(\int_{\Omega_c} \mathbf{F}(\mathbf{U}_i)\cdot{}^\top(J_i^{-1})\nabla\varphi_k \det J_i \, d\mathbf{x} - \sum_{j\in N_i^e}\int_{\partial\Omega_{i,j}} \varphi_k \mathbf{F}^*(\mathbf{U}_i,\mathbf{U}_j)\cdot\mathbf{n}_{i,j}\, dl\right), \quad k=0,\cdots,N_p-1, \tag{5}$$

where N_i^e is the set of indices of elements that share an interface with Ω_i, $\partial\Omega_{i,j}$ is the interface shared by Ω_i and Ω_j, and $\mathbf{n}_{i,j}$ is the outward pointing unit normal on that interface. $\partial\Omega_{i,j}$ is also referred to as e_s, where s is the index of the edge or face. The above is a system of ordinary differential equations (ODEs), which can be integrated in time using an ODE solver, e.g., a Runge-Kutta (RK) method.

Our h-adaptive DG-GPU algorithm is implemented in NVIDIA CUDA C and comprises a DG module and AMR module. In the DG module, we calculate the right-hand-side (RHS) of (5) on a static mesh and advance the solution in time. Every $\mathcal{N}$ time steps, the AMR module adapts the mesh to the solution

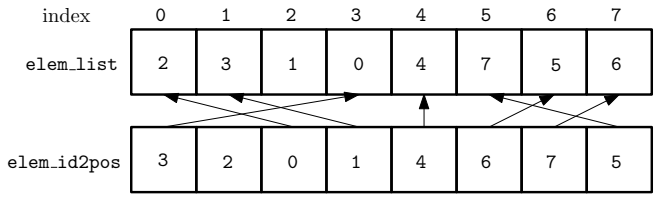


FIGURE 5. The value of `elem_id2pos[i]` indicates the location of Ω_i's ID in `elem_list`.

by refining and coarsening select elements. We show the organization of our AMR and right-hand-side evaluation subroutines in Algorithm 1.

Algorithm 1 Pseudocode for AMR solver

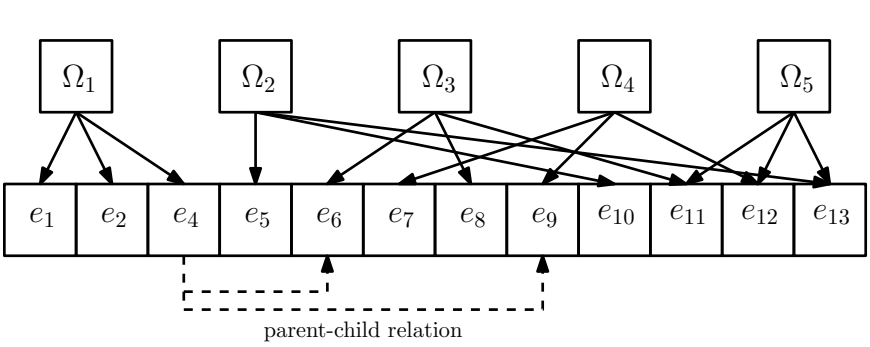
```
step = 1; t = 0;
while t < T do
    Advance c^n to c^{n+1} with the DG method and RK time stepping.   ▷ DG module
    if mod(step, N) == 0 then
        AMR module.
    end if
    step++
    t += Δt
end while
```

3. DG MODULE

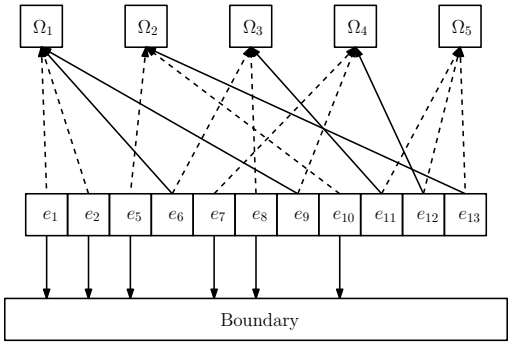
In this section we describe how the DG module is organized and give a brief overview of the subroutines that evaluate the right-hand-side of (5). For a more detailed treatment of these aspects of the solver, see [3, 8].

3.1. **Data ordering and ID numbers.** Every element and edge is assigned a unique identification integer (ID). The IDs of the elements and edges of the initial conforming mesh are given sequentially based on the output of the mesh generator. We store the IDs for elements and edges of the current mesh in arrays called `elem_list` and `edge_list` (Figure 5). After the AMR module is executed, the position an element or edge occupies in `elem_list` or `edge_list` will no longer correspond to its ID. This is because elements will be added and removed within the list due to refinement and coarsening subroutines. Therefore, we store the positions of IDs in `elem_id2pos` to avoid searching `elem_list`. In the example presented in Figure 5, the element with ID 7, Ω_7, is at the fifth index of `elem_list`, i.e. `elem_list[5] = 7`. Then, `elem_id2pos[7] = 5`. The same is done in `edge_id2pos` for edges in `edge_list`.

3.2. **Element information.** The DOFs are organized into arrays named `c0`, `c1`, ..., each of length N, which is the number of elements in the mesh. There is one array for every basis function and equation, i.e. the number of arrays is $N_p \times M$, where N_p is the number of basis functions and M is the number of



(A) Element-to-edge connectivity.



(B) Edge-to-element connectivity. Dashed and solid lines connect an edge to its left and right element, respectively.

FIGURE 6. Connectivity information stored for the refined mesh in Figure 4.

equations in (1). The right-hand-side of (5) is stored in a similar manner in arrays named `rhs0`, `rhs1`, ... This guarantees coalesced reads and writes in computing the volume integral [3]. The DOFs are organized in the same order as the element IDs in `elem_list`. For example, $c_{7,0}$ is at index 5 of `c0` (Figure 5).

Additional data required for the computation of (5) such as element vertices, coordinate transformation Jacobians, and precomputed basis function values are also stored. Element connectivity is stored as the ID numbers of a triangle's three edges (element-to-edge connectivity data, Figure 6a). A triangle may have more than three edges if one or more of its neighbors have been refined, e.g. Ω_1 has four edges in Figure 4. For such nonconforming triangles, the parent ID of the refined edges is stored instead, e.g., edge ID 4 is stored rather than 6 and 9. The IDs 6 and 9 are found using the edge tree structure (Section 4.2).

3.3. **Edge information.** Edge normals and lengths are required for the computation of (5). Refining or coarsening an edge does not introduce new edge normals. Therefore, we only store the normals and lengths of edges in the original mesh. Edges in the current mesh store their refinement level. To find the current edge length, simple arithmetic is done when evaluating the surface contribution term by dividing the original edge length by 2^r, where r is the edge refinement level. An edge points to its left and right element, i.e. the two elements that share it (Figure 6b); the ID of an edge's left and right elements are stored as integers in the arrays `left_elem` and `right_elem`.

3.4. **Right-hand-side evaluation kernels.** A standard RK time integrator requires the evaluation of the right-hand-side of (5). This time stepping module consists of two kernels that evaluate the volume

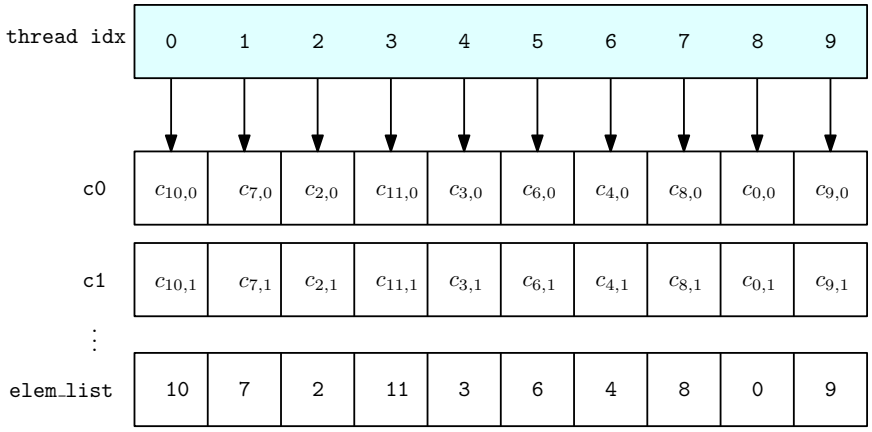


FIGURE 7. `eval_volume` coalesced read access pattern.

terms

$$\frac{1}{\det J_i}\int_{\Omega_c} \mathbf{F}(\mathbf{U}_i)\cdot {}^{\top}(J_i^{-1})\nabla\varphi_k \det J_i d\mathbf{x} \tag{6}$$

and surface terms

$$-\frac{1}{\det J_i}\sum_{j\in N_i^e}\int_{\partial\Omega_{i,j}} \varphi_k \mathbf{F}(\mathbf{U}_i,\mathbf{U}_j)\cdot\mathbf{n}_{i,j}\, dl. \tag{7}$$

One thread per element is launched for the first kernel `eval_volume`. Thread t_i computes the volume integral terms for Ω_i in (6). The thread then stores the volume contribution in `rhs0`, `rhs1`, ... The data for this kernel is accessed in a coalesced fashion. We illustrate this in Figure 7 where thread s accesses the sth positions of arrays `c0`, `c1`, ...

Similarly, one thread per edge $\partial\Omega_{i,j}$, i.e. e_s, of the computational mesh is launched for the second kernel, `eval_surface`. Thread t_s loads the solution coefficients of its edge's left and right elements, then it computes the surface integral terms for edge e_s in (7). The thread then adds the surface contribution to the right-hand-side of its edge's left and right element in `rhs0`, `rhs1`, ... In this kernel, memory is not guaranteed to be accessed in a coalesced fashion. This is because consecutive edges may not necessarily have consecutive left and right elements due to the unstructured nature of the mesh.

3.5. **Coloring.** Thread t_s in the `eval_surface` kernel evaluates the surface integral along the sth edge in `edge_list`. Then, t_s adds its surface contribution to the right-hand-side of the edge's left and right element. The race condition can arise if two threads simultaneously attempt to write their surface contribution to the same element (Figure 8). An edge coloring algorithm that partitions the edges of a conforming mesh was proposed in [8]. The race condition can be avoided by executing `eval_surface` over all edges of the same color in separate kernel launches (Figure 9). In [8], a simple mapping of the colors between coarse

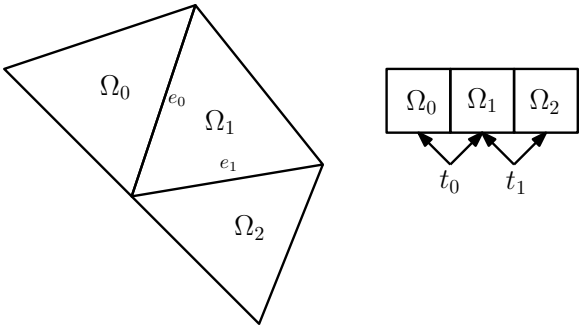
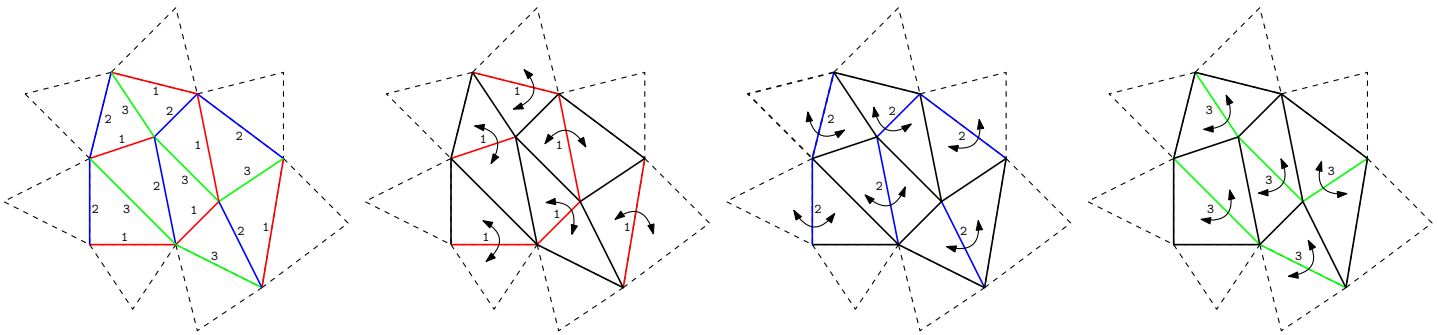


FIGURE 8. Race condition arises when threads t_0 and t_1 write simultaneously to the memory location of Ω_1.



(A) Edge coloring with colors 1, 2, and 3.

(B) Kernel launch over edges of color 1.

(C) Kernel launch over edges of color 2.

(D) Kernel launch over edges of color 3.

FIGURE 9. Avoiding the race condition with coloring. The arrows indicate the elements to which each thread writes its surface contribution.

and fine elements is proposed as it is impractical to recolor the entire mesh when it is adaptively refined (Figure 10). The first child edge retains the color of its parent and the second child takes the parent's color incremented by three. If the parent's color is c, then the children's colors are c and $\mathrm{mod}(c+2,6)+1$, e.g., if the parent's edge color is 1, then its children's colors are 1 and 4. Each new interior edge takes the color of the edge on the parent element to which it is parallel. This process is easily reversible during the coarsening operation. These algorithms result in the minimum number of colors used, i.e., 3 and 6 colors on conforming and nonconforming meshes of triangles, respectively.

4. Adaptive mesh refinement

We discuss in this section the implementation details of the adaptive mesh refinement module of the code. First, we compute an indicator on each cell from which we determine which elements to flag for refinement or coarsening. During one execution of the AMR subroutines, we allow the refinement level of a cell to be adjusted by at most one. The exception is the initial condition where the AMR module is executed a number of times until a predefined maximum refinement level is reached.

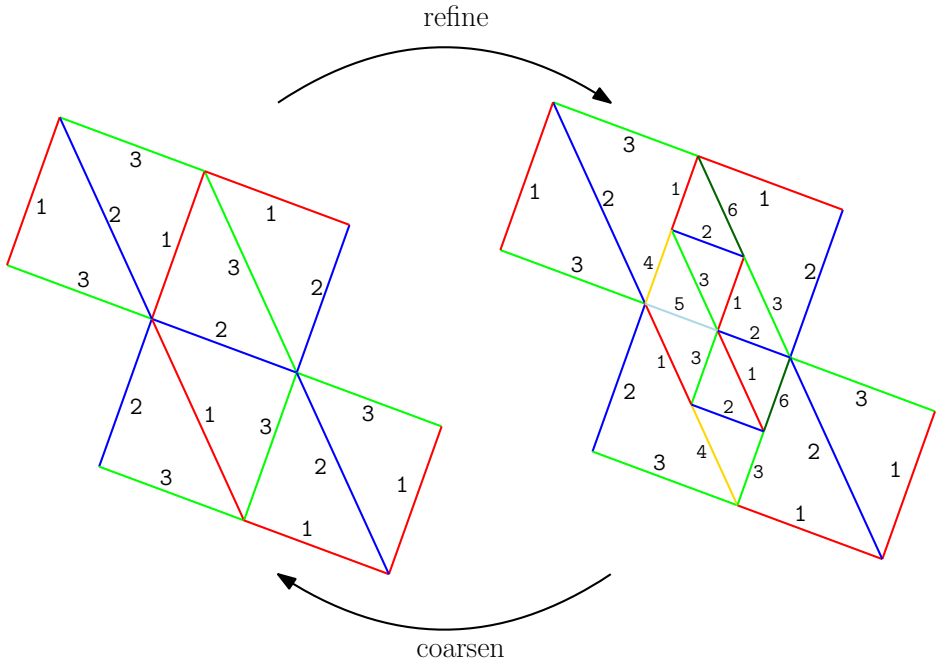


FIGURE 10. Mapping the initial coloring to refined triangles and back.

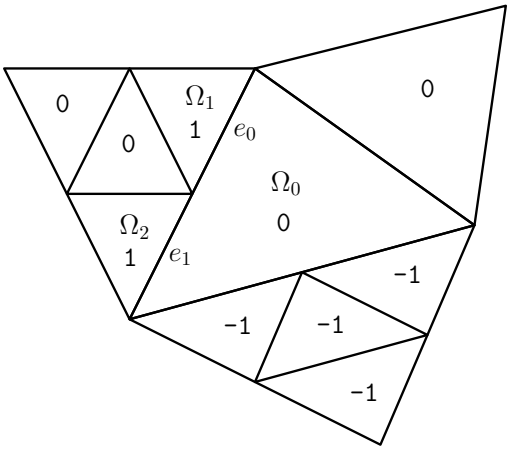
4.1. **Mesh smoothing.** After elements are flagged for refinement or coarsening (Section 4.8), we perform mesh smoothing to avoid creating a mesh where the refinement levels of neighboring elements differ by more than one (Figure 4, right), i.e., we require that

$$|r_i - r_j| \leq 1, \tag{8}$$

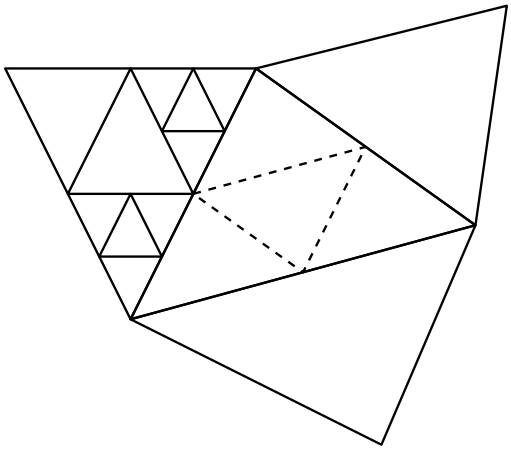
where r_i and r_j are the refinement levels of adjacent elements Ω_i and Ω_j, respectively. For example, in Figure 11a we display a mesh with AMR flags '`-1`', '`1`', or '`0`' on each element, indicating whether it should be coarsened, refined, or neither. These AMR flags are stored in the array `edr`, '**e**lement **d**ifference in **r**efinement'. When Ω_1 and Ω_2 are refined, the refinement level of their children and Ω_0 will differ by two, which violates the nonconformity constraint (8). We therefore must smooth the mesh to reduce this jump.

We implement this operation by launching one thread per edge in a kernel called `smooth` (Algorithm 2). Each thread loads the left and right element of its designated edge. Using the elements' current refinement levels and `edr` value, the element refinement levels after the AMR operation are computed. If their updated refinement levels do not satisfy the nonconformity constraint (8), then the AMR flag in `edr` of either the left or right element is modified such that a more refined mesh is obtained. For example, on edge e_0 in Figure 11a, the refinement levels after the AMR operation will be 2 on Ω_1's children and 0 on Ω_0. Consequently, we refine Ω_0 once (Figure 11b).

We execute `smooth` on all edges of the same color in separate kernel launches. This is done in order to avoid memory contention when modifying the AMR flag array `edr`. Without coloring, threads operating



(A) '1' indicates refinement, '-1' indicates coarsening, and '0' indicates that the element is neither coarsened nor refined.



(B) Mesh after refinement, coarsening, and smoothing operations. The dashed line shows the elements added due to smoothing.

FIGURE 11. Mesh smoothing operation.

on edges e_0 and e_1 in Figure 11a could write their result simultaneously to the position in memory corresponding to Ω_0 in `edr`. `smooth` is launched multiple times until the nonconformity condition (8) is satisfied for all elements.

After mesh smoothing, it is finalized which elements are to be refined or coarsened. Thus, we may proceed to executing the refinement and coarsening subroutines.

Algorithm 2 Mesh smoothing operation

```
procedure SMOOTH(edr)
    le, re← positions of left and right element.
    left_after, right_after← level after refinement of left and right elements
    if left_after-right_after > 1 then
        edr[re] = 1
    else if right_after-left_after > 1 then
        edr[le] = 1
    end if
end procedure
```

4.2. **Tree structures.** The parent-children relations of elements and edges are stored in tree data structures, where parents and children point to one another. The tree structure for the elements in the refined mesh of Figure 4 are shown in Figure 12. The elements and edges that are active in the refined mesh are highlighted in blue and do not have children.

4.3. **Connectivity.** In order to evaluate the surface contributions in (7), each thread of the kernel `eval_surface` needs the IDs of the left and right elements sharing its assigned edge, i.e., edge-to-element

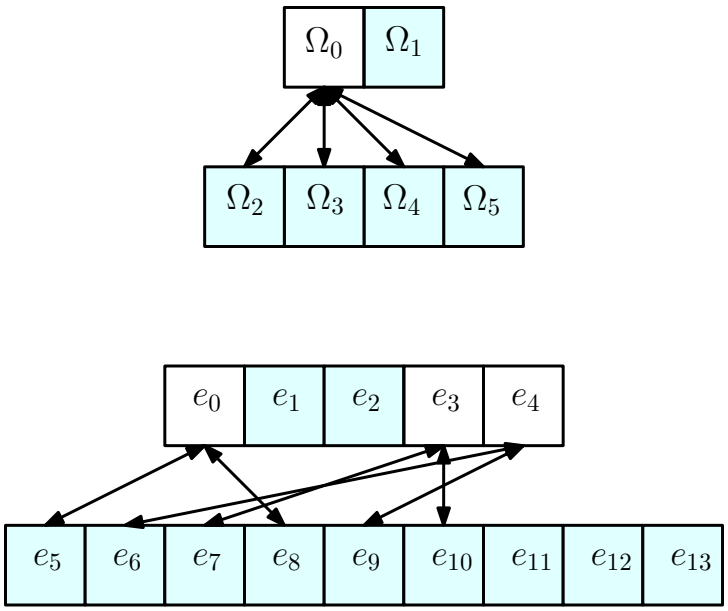


FIGURE 12. Tree for the refined mesh in Figure 4. Elements and edges shaded in blue are active in the current mesh.

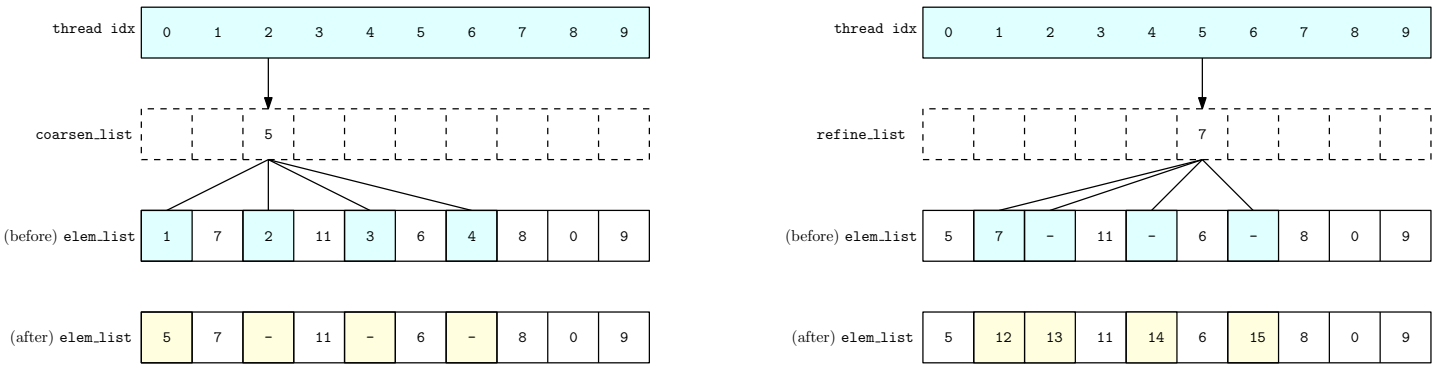
connectivity data (Figure 6b). After the mesh is refined and coarsened, it is necessary to update mesh connectivity to reflect the addition and removal of elements, e.g., find the new left and right elements sharing an edge. We do this by using the tree structure described above.

First, we refine the edge tree by splitting edges flagged for refinement, assign new IDs to child edges, and update the IDs in `edge_list`. Similarly, we refine the element tree and update the IDs in `elem_list`. From the edge and element trees, we can compute the connectivity between new elements and edges.

As an example, consider the refined mesh in Figure 4. First the edges of Ω_0 are refined and the new IDs for the child edges of e_0, e_3, and e_4 are assigned (Figure 12, bottom). Next, Ω_0 is refined and the new IDs for the children are assigned. The IDs of the child elements' outer edges can be found from the already updated edge tree, e.g. e_7 and e_{10}. Then, the edge IDs of the interior child triangle Ω_5 must be created, e_{11}, e_{12}, and e_{13}. Now that the new elements know the IDs of their edges, i.e. we have the updated element-to-edge connectivity (Figure 6a), the left and right elements of the new edges in `edge_list` can be updated.

By our convention, a triangle points to three edges that have the same refinement level. For elements that have more than three edges, e.g. Ω_1 in Figure 4, the ID of the refined edges' parent is stored instead. In our example, Ω_1 points to e_1, e_2, and e_4, which are all of refinement level 0. From the edge tree, we can find that e_4 points to e_6 and e_9 (Figure 6a).

4.4. **Coarsening.** Coarsening is done 'in-place', i.e., this modification to the mesh does not require a buffer and avoids large amounts of memory transfers (Figure 13a). `coarsen_list` contains a list of parent



(A) Coarsening. (B) Refining.

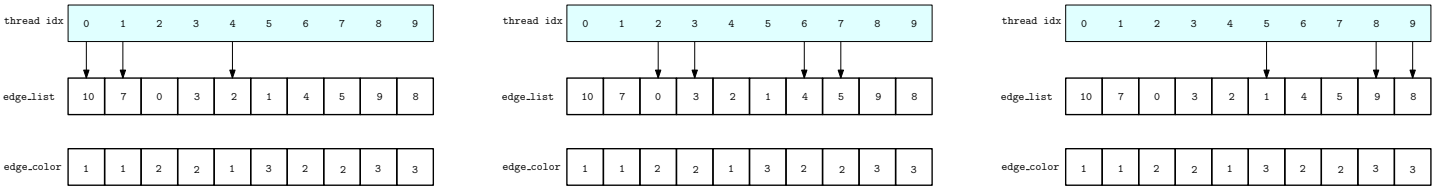
FIGURE 13. Access pattern for refinement and coarsening kernels.

element IDs that are to replace their children. We place the parent ID in the first child's position in `elem_list` and flag the other children's memory locations as unused. We illustrate in Figure 13a how `elem_list` is updated to reflect the coarsening of elements Ω_1, Ω_2, Ω_3, and Ω_4. Thread 2 places the parent element ID of the cluster, i.e. Ω_5, in the list position of its first child element ID, i.e. in the position of element Ω_1. The freed memory spaces of the three other children are indicated by dashes. All the data associated with elements are dealt with in a similar fashion, e.g., the DOFs of the parent element Ω_5 are placed in the old memory location of the first child, Ω_1.

Coarsening four children leads to the removal of a section of the element and edge trees. We keep track of the memory and ID numbers freed during the coarsening operation in order for them to be reused during a refinement operation. For this reason, coarsening, if required, is always executed before refinement.

The solution coefficients on a parent element are obtained using an L_2 projection on the four child elements. The projection is implemented as a dot product of the solution coefficients on the children and weights that have been precomputed for fast execution.

4.5. **Refinement.** Refinement is also done 'in-place' (Figure 13b). `refine_list` contains a list of element IDs that are to be refined. The position of the parent element in `elem_list` is taken by its first child. The IDs for the three remaining children are placed in free memory locations, if any were made available during coarsening. If there are no free locations within the list, then the additional IDs are concatenated to the end of the array. All the data associated with elements are dealt with in a similar fashion. For example, in Figure 13b, Ω_7 in `refine_list` is refined by thread 5. The first child ID Ω_{12} overwrites Ω_7, the rest (Ω_{13}, Ω_{14}, and Ω_{15}) are placed in the remaining available memory locations of `elem_list` indicated by dashes. The DOFs of the four child elements are obtained with an L_2 projection, which is equivalent to a dot product of the parent DOFs and weights, which have been precomputed for fast execution.

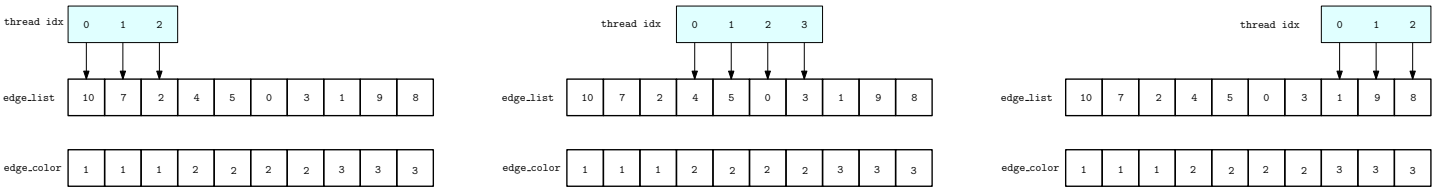


(A) Launch over edges of color 1.

(B) Launch over edges of color 2.

(C) Launch over edges of color 3.

FIGURE 14. Without ordering of the edge IDs, `eval_surface` will have inactive threads, reducing parallelism.



(A) Launch over edges of color 1.

(B) Launch over edges of color 2.

(C) Launch over edges of color 3.

FIGURE 15. With ordering of the edge IDs, all threads of `eval_surface` will be active.

4.6. **Edge reordering.** The surface integral kernel, `eval_surface`, is executed on the edges of a particular color in separate launches. If the edges in `edge_list` are not ordered by color, then the parallelism of `eval_surface` will be reduced. This is because some threads will be inactive during the kernel execution (Figure 14). The edges in `edge_list` are not guaranteed to be ordered by color after the mesh is refined and coarsened. Therefore, after the AMR subroutines complete, the edges in `edge_list` must be reordered by color. This will guarantee all threads in the launches of `eval_surface` are active (Figure 15).

4.7. **Memory management.** In Sections 4.4 and 4.5, we described how the coarsening and refinement operations are done in-place. This is to avoid using buffers and unnecessary memory transfers. However, if the total number of elements in the mesh is reduced by the refinement and coarsening operations, there will be 'holes' in the the data arrays. This will reduce the efficiency of memory accesses by making some reads and writes uncoalesced. Therefore, these 'holes' must be filled to make the data contiguous in memory.

Algorithm 3 Compaction

```
procedure COMPACTION(out, in, out_flag, in_flag)
    idx ← thread index
    if in_flag[idx] then
        out[out_flag[idx]] ← in[idx]
    end if
end procedure
```

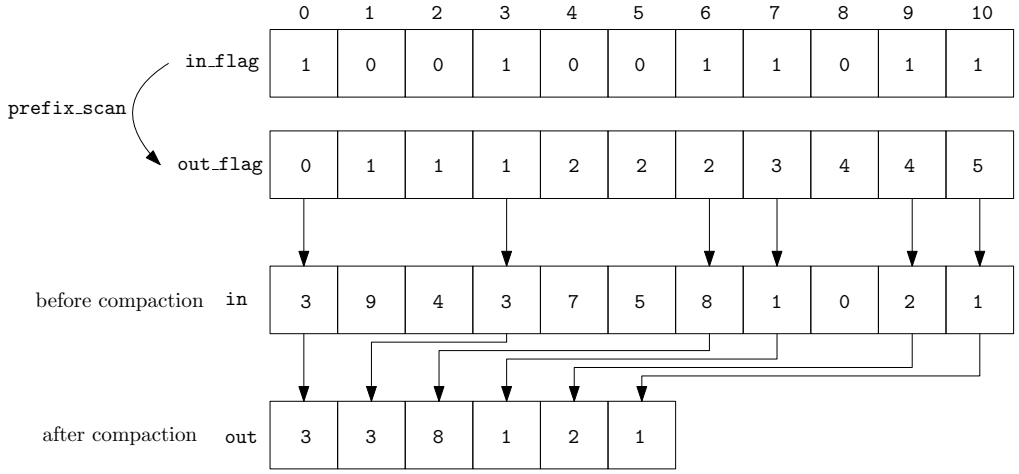


FIGURE 16. Example of a standard compaction algorithm.

This operation on GPUs is called a 'stream compaction' and it does not have a simple solution. Typically, a compaction is done using an operation called a prefix sum. A prefix sum takes as input an array of integers `in_flag` and outputs another array `out_flag`. The element at the nth index of the output array is given by the formula

$$\texttt{out\_flag}[n] = \sum_{i=0}^{n-1} \texttt{in\_flag}[i] \text{ for } n > 0, \tag{9}$$

and `out_flag`[0] is set to 0 (Figure 16). Now, assume that we wish to copy selected data from the array `in` into the array `out`. `in_flag` is an integer array of `0`'s and `1`'s, which indicates whether the data at the corresponding position in `in` must be preserved ('`1`') or removed ('`0`'). After computing the prefix sum for `in_flag`, a compaction kernel (Algorithm 3) is launched that creates a new array `out` without the unwanted data. This procedure is illustrated in Figure 16. There are application programming interfaces (APIs) that provide an implementation of the prefix sum and compaction operations. Unfortunately, the standard implementations of the stream compaction operation are not suitable for our purposes. This is because we may have GBs of data where only a small fraction of the elements in those arrays require removal. Executing the compaction kernel will always result in all the data of the compacted array being moved to a new location, regardless of the number of elements being removed. A more efficient solution is to implement an in-place compaction, which we will now describe.

Suppose we wish to remove the elements from the array of integers in Figure 17 using `in_flag`. According to `in_flag`, the compacted array will have a length of 6. Let us refer to the position between indices 5 and 6 as the 'pivot'. The idea is to fill the holes to the left of the pivot with elements from the right of the pivot. First, we find the elements to the right of the pivot for which `in_flag` is '`1`' and store their position in the array `from`. Next, we find the elements located to the left of the pivot for which `in_flag` is '`0`' and

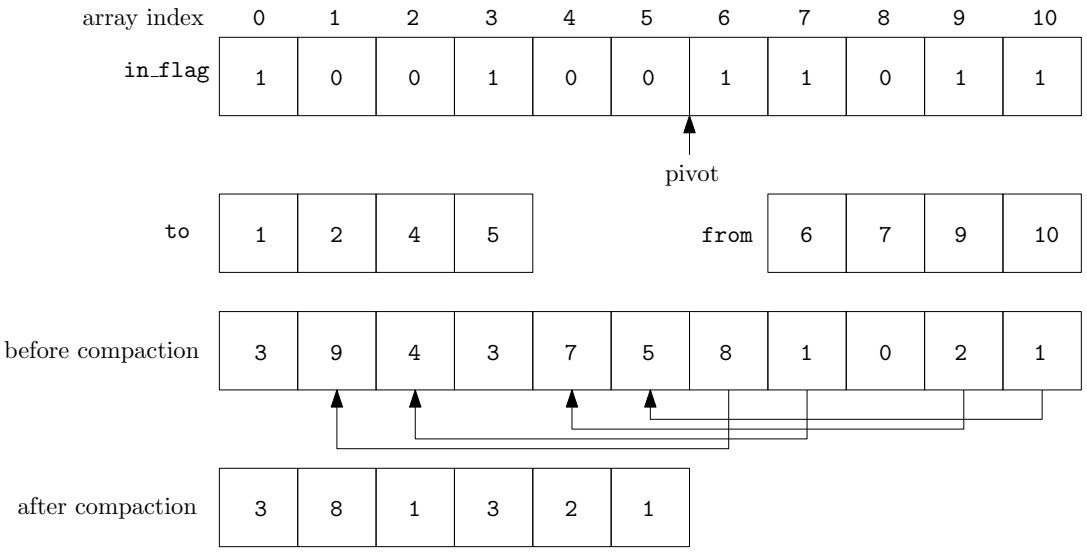


FIGURE 17. In-place compaction implementation.

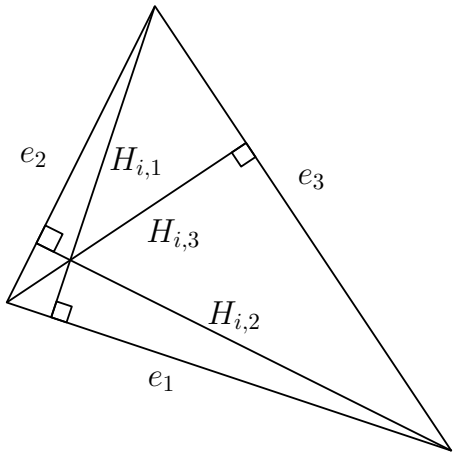


FIGURE 18. Minimum cell height $h_i = \min(H_{i,1}, H_{i,2}, H_{i,3})$ [36].

store their positions in `to`. Populating `to` and `from` is done using the prefix-scan from CUB [35]. The final step is to launch a kernel which completes the data transfer in-place (Algorithm 4). This operation does not preserve the initial ordering of the data, but this is not important in our application.

Algorithm 4 In-place compaction

```
procedure IN_PLACE_COMPACTION(in, to, from)
    idx ← thread index
    in[to[idx]] ← in[from[idx]]
end procedure
```

4.8. **Refinement strategy.** In this paper, we are not concerned with the optimal way to flag an element for coarsening or refinement. Therefore, we adopt the following simple strategy reported in the literature [26]. We compute on Ω_i the refinement indicator ϵ_i. Then, we compare ϵ_i to reference values $\epsilon_r = \epsilon\delta$ and $\epsilon_c = \epsilon/\delta$, where ϵ and δ are prescribed constants. Ω_i is refined if $\epsilon_i > \epsilon_r$. A cluster of four elements, Ω_i,

Ω_j, Ω_k, Ω_l, with the same parent element, is coarsened if $\epsilon_i, \epsilon_j, \epsilon_k, \epsilon_l < \epsilon_c$. We choose the simple refinement indicator

$$\epsilon_i = h_i ||\nabla U_i||_2, \tag{10}$$

where ∇U_i is the gradient evaluated at the cell centroid and h_i is the minimum cell height (Figure 18, [36]).

5. Computed examples

5.1. **Initial-boundary value problems.** We solve the two-dimensional Euler equations, which can be written in form (1) with the fluxes

$$\mathbf{F}_1(\mathbf{U}) = \begin{pmatrix} \rho u \\ \rho u^2 + p \\ \rho uv \\ (E+p)u \end{pmatrix} \quad \text{and} \quad \mathbf{F}_2(\mathbf{U}) = \begin{pmatrix} \rho v \\ \rho uv \\ \rho v^2 + p \\ (E+p)v \end{pmatrix}, \tag{11}$$

where $\mathbf{U} = [\rho, \rho u, \rho v, E]$. The system is closed with the equation of state

$$p = (\gamma - 1)\left(E - \frac{\rho}{2}(u^2 + v^2)\right),$$

where $\gamma = 1.4$ is the adiabatic constant for air, ρ is the density, u and v are components of the velocity vector, and E is the energy.

5.1.1. *Smooth isentropic vortex.* We use this example to illustrate the runtime performance of our AMR algorithm. This example was solved on an NVIDIA Titan X Pascal. The problem with initial conditions stated in Table 1 has the exact solution $\mathbf{U}(x, y, t) = \mathbf{U}_0(x, y - t)$, i.e. the initial vortex is advected in the y direction with speed 1 [37]. It is solved until the final time $T = 2$ on the domain $[-10, 10]^2$ with the exact solution used as boundary conditions. The initial mesh is coarse and composed of 180 unstructured triangles.

First, we perform an initial mesh adaptation to accurately capture the initial conditions. Then, we run the mesh adaptation subroutines every time step. In this example, we allow at most six levels of refinement. The size of the mesh for all orders of approximation was about 30,000 elements and did not vary substantially during the simulation. This is expected since the solution is a translation of the initial conditions.

ρ	$\left(1-(\gamma-1)\frac{(SM)^2}{8\pi^2}e^r\right)^{\frac{1}{\gamma-1}}$
u	$\frac{Sy}{2\pi R}e^{\frac{1}{2}r}$
v	$1-\frac{Sx}{2\pi R}e^{\frac{1}{2}r}$
p	$\frac{1}{\gamma M^2}\rho^\gamma$

TABLE 1. Initial conditions for the smooth isentropic vortex problem (Example 5.1.1), where $r=\frac{1}{R^2}(1-x^2-y^2)$, $S=13.5$, $M=0.4$, and $R=1.5$.

The breakdown of compute time in seconds for solutions with $p=1...4$ is reported in Table 2. The same data, but as a percentage of total AMR time, are shown in Figure 19. We report the timings in terms of seven subroutines: `determineParents`, `smooth`, `coarsen`, `refine`, `compaction`, `determineSideOrder`, `reorderSides`, and `other`. The procedures grouped in `other` include computing the refinement indicator (10), updating mesh connectivity and tree structures. `determineParents` looks up the parent IDs of the elements flagged for coarsening and stores them in `coarsen_list` (Figure 13a). `determineSideOrder` determines the new order of the edge IDs based on their color and `reorderSides` reorders them and edge data to avoid race conditions in `eval_surface`.

In Table 2, we notice that the time spent in the AMR subroutines increases with the order of approximation as the number of calls to the time stepping and mesh adaptation modules increases with p. This is because the time step size scales inversely with the order of the method for the DG method of order approximation p paired with an RK scheme of order $p+1$. In Table 3, we list the data from Table 2 normalized by the number of time steps. For subroutines that only depend on the number of elements in the mesh, we observe that the normalized timings are similar for all polynomial orders.

We also note from Table 2 that the fraction of the total runtime spent in the AMR module of the code decreases with p. This is because the number of RK stages and the cost of computing the RHS of (5) increases with p. For orders $p=1$ to 3, the `smooth` subroutine is the most time consuming operation because it is difficult to parallelize efficiently on the GPU (Section 4.1).

Note that refinement is usually performed less frequently than after every time step as is done for this example, e.g., we may refine when the solution moves two cell widths. For the RK-DG method where $p=1,2,3,4$, this means every 8, 12, 16, 20 time steps, respectively [38]. Therefore, for practical applications, the overhead associated with the AMR subroutines is small compared to the total runtime of the solver, especially for high order simulations.

5.1.2. *Kelvin-Helmholtz instability.* Next, we solve the Kelvin-Helmholtz instability problem on the domain $[-1,1]^2$, with the initial conditions given in Table 4 and illustrated in Figure 20 [39]. The initial conditions

Subroutine	1	2	3	4
`determineParents`	0.44	0.64	0.84	1.03
`smooth`	1.21	1.75	2.29	2.81
`coarsen`	0.49	0.91	1.71	2.90
`refine`	0.76	1.24	2.03	3.39
`compaction`	0.49	0.78	1.16	1.67
`determineSideOrder`	0.70	1.00	1.32	1.62
`reorderSides`	0.18	0.25	0.33	0.41
`other`	0.49	0.73	0.96	1.21
AMR time	4.76 (0.45)	7.30 (0.20)	10.63 (0.09)	15.05 (0.04)
Total solver runtime	10.54	36.83	121.33	380.67

TABLE 2. Break-down of time spent in each AMR subroutine in seconds for Example 5.1.1. The number in parentheses is the fraction of the total runtime spent in the AMR module of the code, when refining every time step.

Subroutine	1	2	3	4
`determineParents`	195.93	196.18	198.41	198.27
`smooth`	539.08	537.41	542.00	541.03
`coarsen`	216.95	279.58	404.79	558.14
`refine`	335.65	381.98	480.29	652.52
`compaction`	216.66	239.40	275.16	321.12
`determineSideOrder`	310.73	309.06	312.01	311.20
`reorderSides`	78.13	77.82	78.42	79.02
`other`	218.06	224.95	226.57	233.06
AMR time	2111.19	2246.38	2517.65	2894.35

TABLE 3. Timings in Table 2 divided by the number of time steps. Values are reported in microseconds per time step. The grayed rows correspond to operations with runtimes that are independent of the order of approximation.

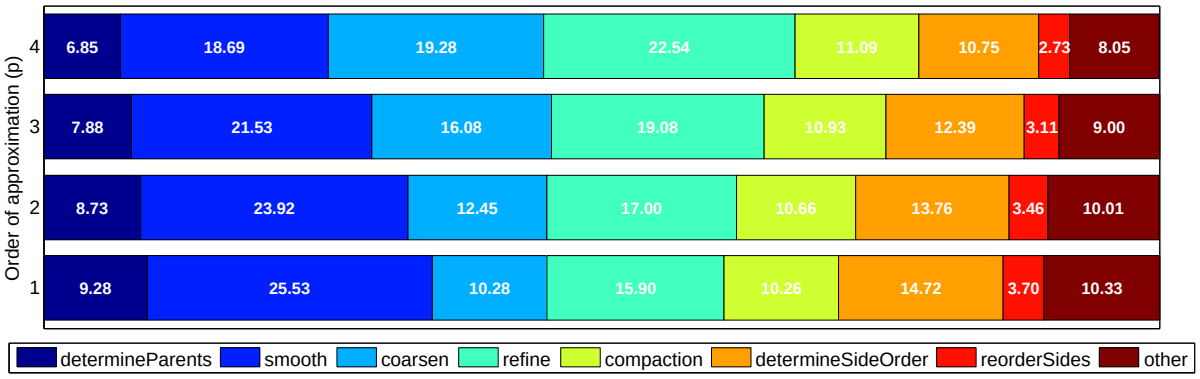


FIGURE 19. Percentage of the AMR runtime spent in each subroutine.

describe three fluid layers. The outer layers are dense, rightward moving fluids that are sandwiching a less dense leftward moving fluid. In the neighborhood of the interfaces, the fluids are perturbed with an oscillatory vertical velocity. These interfaces are sliplines, which will lead to a rich production of vortices

	$\lvert y\rvert \leq 0.5$	elsewhere
ρ	2	1
u	0.5	-0.5
v	$w\sin(4\pi x)\left[e^{-\frac{1}{2s^2}(y+0.5)^2}+e^{-\frac{1}{2s^2}(y-0.5)^2)}\right]$	
p	2.5	

TABLE 4. Density, velocity, and pressure of the three layers of fluid where $w = 0.1$ and $s = 0.05/\sqrt{2}$ [39] (Example 5.1.2).

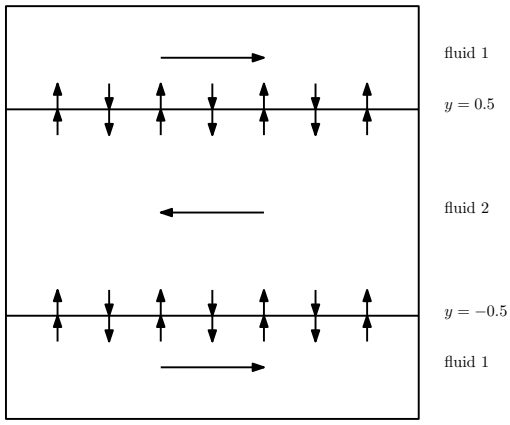
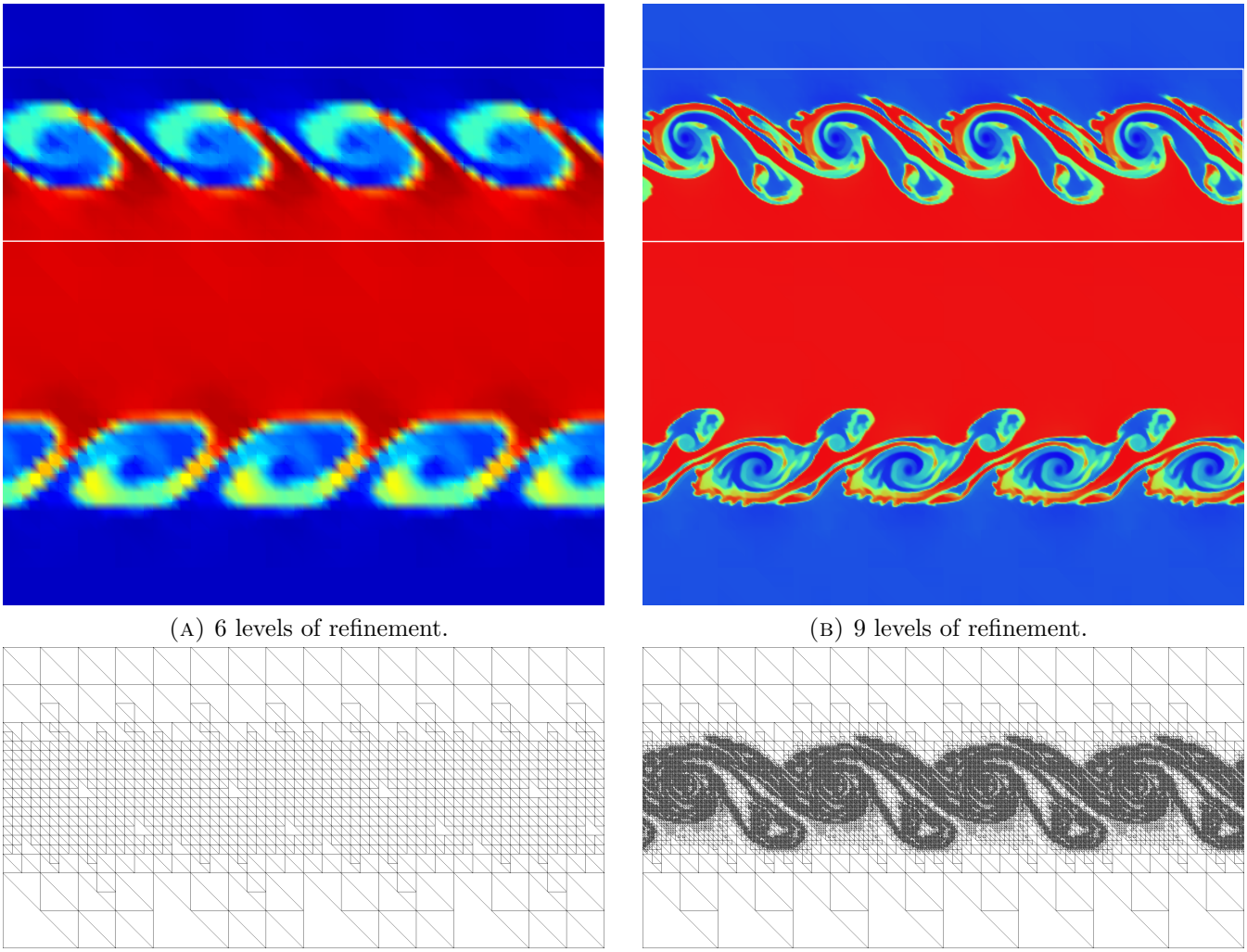


FIGURE 20. Setup of the Kelvin-Helmholtz test problem (Example 5.1.2). The arrows indicate the direction of fluid flow.

in the numerical solution. We prescribe the initial state at the horizontal boundaries of the domain and impose periodic boundary conditions at the vertical boundaries. The initial mesh is composed of two triangles.

The density along with the adaptively refined meshes at $T = 2$ with 6, 9, and 12 levels of refinement are plotted in Figures 21 and 22. The number of elements in the final meshes is approximately 4,000, 120,000, and 4,000,000. The AMR algorithms act predominantly in the neighborhood of the sliplines. As we increase the number of refinement levels, the vortices become more pronounced due to the reduction in numerical viscosity.

5.1.3. *Double Mach reflection.* We use this example to demonstrate the algorithm's performance on a transient shock reflection problem. We solve double Mach reflection problem on the domain $[0, 3.5] \times [0, 1]$ with the initial condition of a rightward-moving shock that propagates into a quiescent gas [40]. The incident, Mach 10 shock forms a 60° angle with a reflecting boundary, which results in the reflection pattern shown in Figure 3c. We provide the setup in Figure 23 along with the incident $\mathbf{U}_I$ and quiescent $\mathbf{U}_Q$ states in Table 5. We solve the problem until the final time of $T = 0.2$, allowing 3, 6, and 9 levels of refinement from an initial unstructured mesh composed of 1,776 triangles. The adaptively refined meshes at the final time are shown in Figure 24 along with the density isolines on the finest mesh in Figure 25.



(A) 6 levels of refinement.

(B) 9 levels of refinement.

(C) Mesh on white rectangle in Figure 21a.

(D) Mesh on white rectangle in Figure 21b

FIGURE 21. Final solutions and adaptively refined meshes of the Kelvin-Helmholtz test problem allowing 6 and 9 levels of refinement.

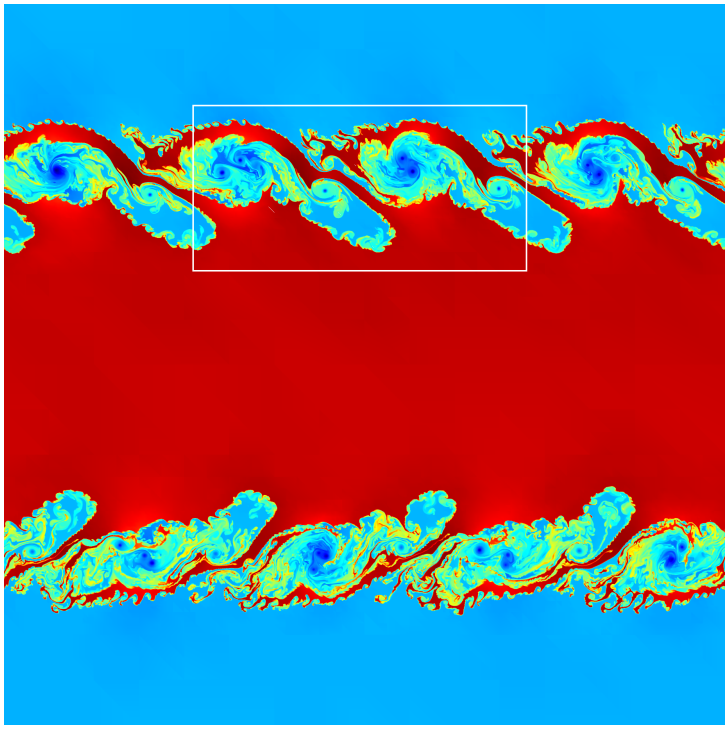
	$\mathbf{U}_I$	$\mathbf{U}_Q$
ρ	8	1.4
s	8.25	0
p	116.5	1

TABLE 5. Density, normal speed (s), and pressure of the incident $\mathbf{U}_I$ and quiescent $\mathbf{U}_q$ states in Example 5.1.3.

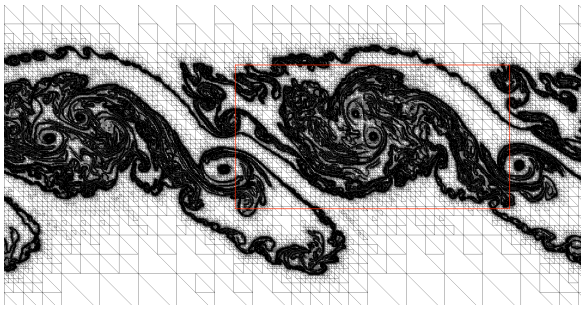
5.2. **Boundary value problems.** The initial-boundary value problem (1) can be restated as a boundary value problem in the self-similar coordinates $\xi = \frac{x}{t}$, $\eta = \frac{y}{t}$, and $\tau = \ln t$, as follows

$$\frac{\partial}{\partial \tau}\mathbf{U} + \frac{\partial}{\partial \xi}(\mathbf{F}_1(\mathbf{U}) - \xi\mathbf{U}) + \frac{\partial}{\partial \eta}(\mathbf{F}_2(\mathbf{U}) - \eta\mathbf{U}) + 2\mathbf{U} = 0. \tag{12}$$

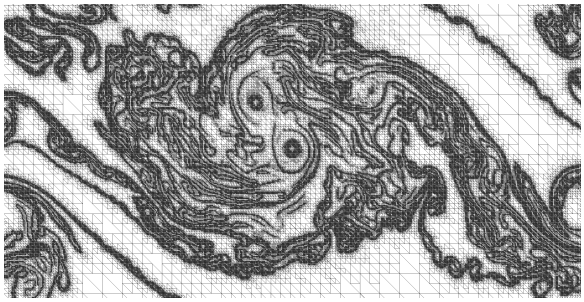
A steady state solution of (12) corresponds to a self-similar solution of (1) in (ξ, η) coordinates. Self-similar form (12) is useful because adapting the mesh for steady state solutions is simpler than for transient ones.



(A) 12 levels of refinement.



(B) Mesh on white square in Figure 22a.



(C) Zoom on red rectangle in Figure 22b.

FIGURE 22. Final solution and adaptively refined mesh of the Kelvin-Helmholtz test problem allowing 12 levels of refinement.

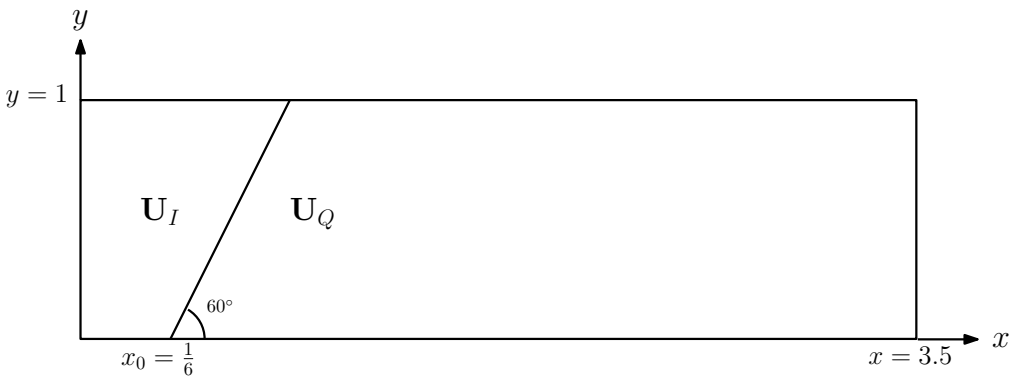
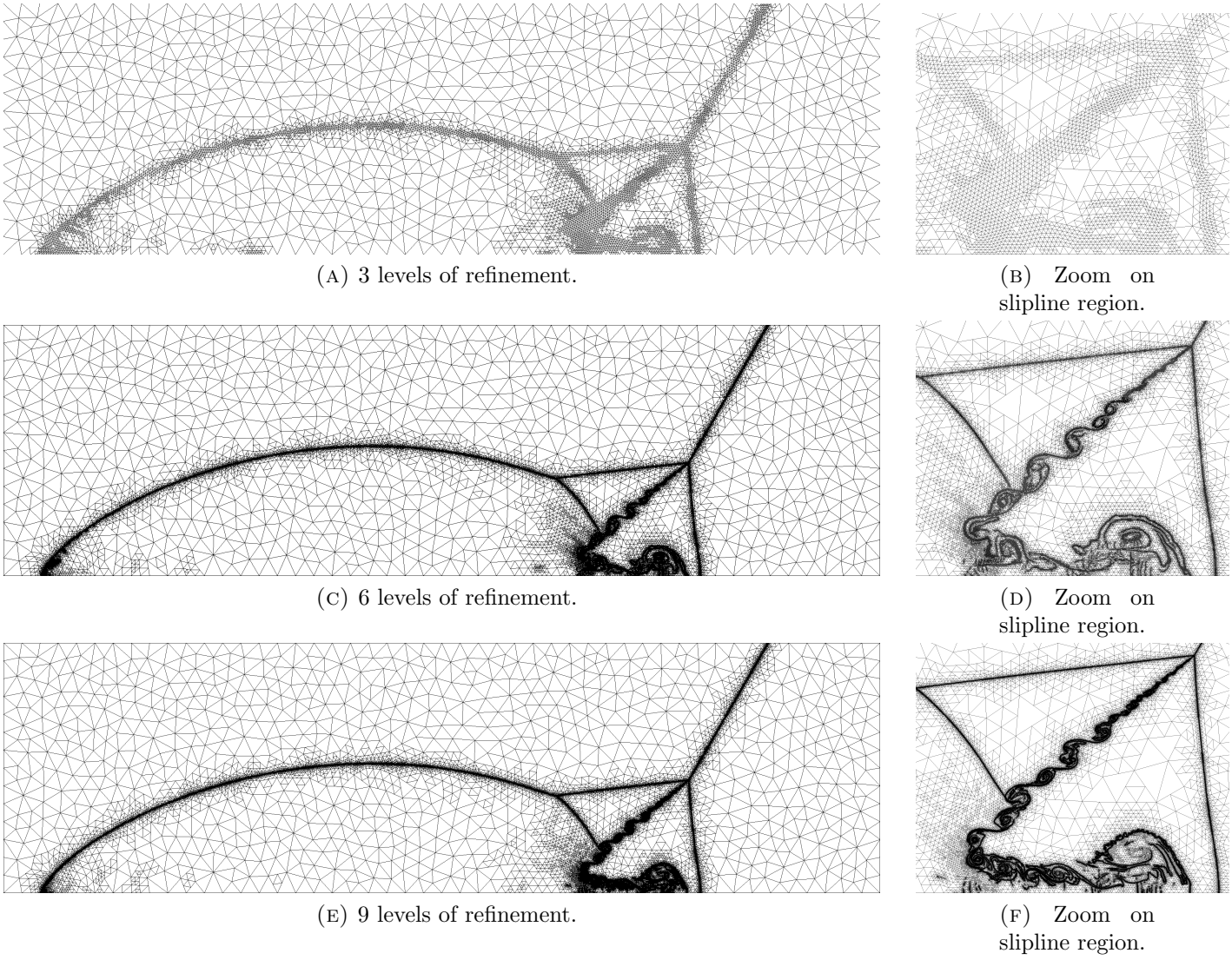


FIGURE 23. Double Mach reflection setup (Example 5.1.3).

This is because the relevant features in the solution do not move after the initial transient phase passes. In this section, we solve (12) where $\mathbf{F}_1(\mathbf{U})$ and $\mathbf{F}_2(\mathbf{U})$ are the fluxes in (11).

5.2.1. *Von Neumann triple point paradox.* In this example, we present numerical evidence that supports Guderley's solution to the von Neumann triple point paradox. The problem setup consists of a weak, rightward moving incident shock impinging obliquely on a wedge on a computational domain in the shape of a circular sector (Figure 26a). The incident (I) and reflected (R) shocks detach from the wedge and meet at a triple point (TP). The triple point is connected to the wedge via a Mach stem (MS). We are interested in a very small portion of the reflection interaction shown in Figures 26b and 26c. The supersonic patches predicted by Guderley Mach reflection are illustrated with dash-dotted lines in Figure 26c.



(A) 3 levels of refinement.

(B) Zoom on slipline region.

(C) 6 levels of refinement.

(D) Zoom on slipline region.

(E) 9 levels of refinement.

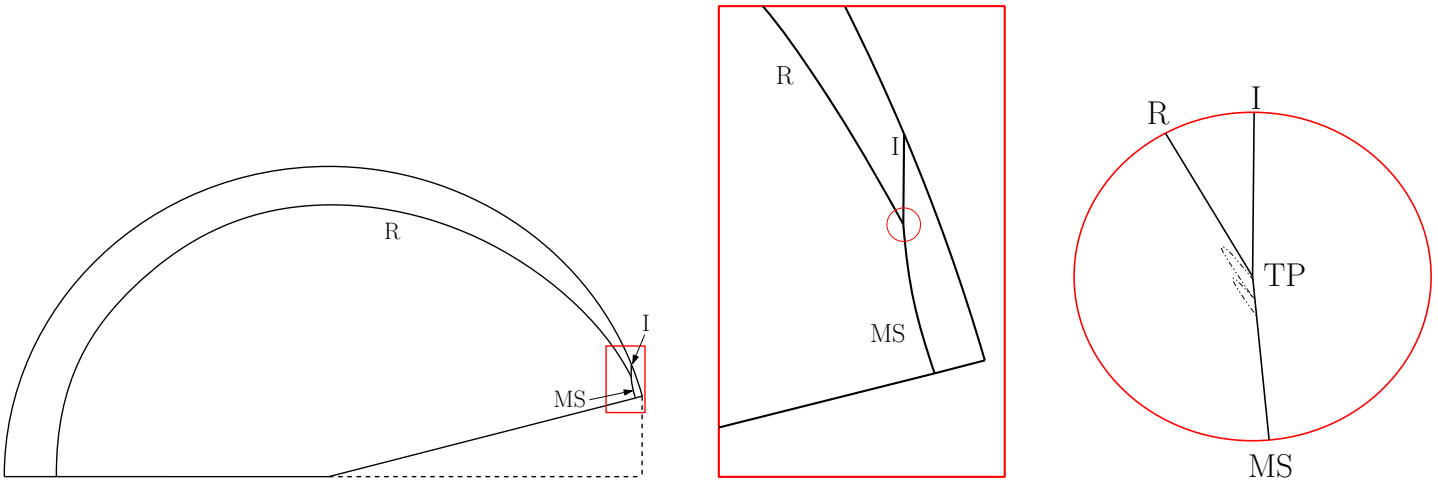
(F) Zoom on slipline region.

FIGURE 24. Final meshes of the double Mach reflection problem allowing 3, 6, and 9 levels of refinement.

(A) Density isolines.

(B) Zoom on slipline region.

FIGURE 25. Density isolines at final time on the mesh with 9 levels of refinement in Figures 24e and 24f.



(A) Shock reflection pattern on the domain.

(B) Zoom on the red rectangle in Figure 26a.

(C) Zoom on the red circle in Figure 26b. The dashed-dotted line behind the triple point is the sonic line.

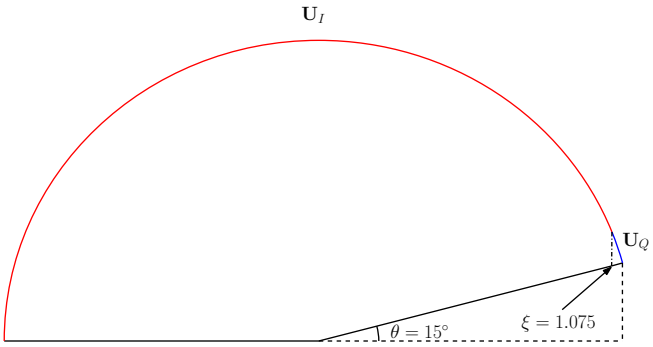
FIGURE 26. Incident (I) and reflected (R) shocks, with the Mach stem (MS) on the solution domain, with zooms on the neighborhood of the triple point (TP).

A number of numerical investigations of this problem have been executed on block adaptively refined grids [41] or distorted conforming grids [29, 42]. Here we present results on an unstructured mesh of triangles. The initial conforming mesh of the circular sector-shaped domain in Figure 27a is shown in Figure 27b. The initial mesh is constructed such that the elements are aligned with the incident shock.

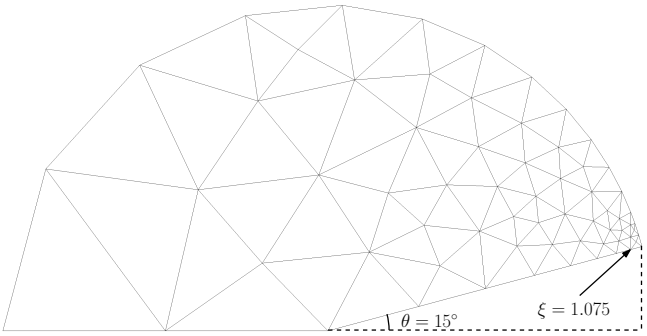
The boundary conditions are given by a weak incident shock traveling at Mach 1.075 into a quiescent gas [29]. The incident $\mathbf{U}_I$ and quiescent $\mathbf{U}_Q$ states are reported in Table 6, the boundaries on which they are imposed are shown in Figure 27a. With the goal of determining an accurate position of the triple point, we resolve the full length of the incident and Mach stem as opposed to only in a neighborhood of the triple point. Additionally, we explicitly prescribe that elements lying on a half disk centered on the triple point are refined to the maximum level (Figure 29c). This is because the solution varies little in that region and the refinement indicator we use has difficulty detecting the subtle reflection pattern. This small refined region moves with increasing pseudotime τ, following the triple point along the vertical line $\xi = 1.075$ to its final position in self-similar coordinates $(1.075, 0.4111)$. We note that this is more accurate than previously reported in the literature due to better resolution the shocks away from the triple point. This is the advantage of the automatic mesh adaptation algorithms. We plot in Figure 28 the η coordinate as a function of pseudotime τ.

	$\mathbf{U}_I$	$\mathbf{U}_Q$
ρ	1.57697	1.4
u	0.12064	0
v	0	0
p	1.18156	1

TABLE 6. The incident $\mathbf{U}_I$ and quiescent $\mathbf{U}_Q$ states imposed as boundary conditions in Example 5.2.1.



(A) Initial domain. The incident $\mathbf{U}_I$ and quiescent $\mathbf{U}_Q$ states are imposed on the red and blue boundaries, respectively. The bottom boundary is reflecting.



(B) Initial, conforming mesh of domain in Figure 27a.

FIGURE 27. Initial domain and mesh for the von Neumann triple point paradox problem in Example 5.2.1.

In Figure 29, we provide an adapted mesh, composed of $\sim$ 800,000 elements, with $\sim$ 120,000 elements of minimum cell width $h \approx 4.6 \cdot 10^{-6}$ in the neighborhood of the triple point, where h is defined in Figure 18. We compute the self-similar Mach number

$$\tilde{M} = \frac{\sqrt{(u-\xi)^2 + (v-\eta)^2}}{c},$$

where c is the speed of sound, and plot the isolines of $\tilde{M}$ in Figure 30. The solution in Figure 30b was obtained by further refining the elements in the neighborhood of the triple point in Figure 30a by a factor of 8. On the coarser mesh, only one supersonic patch is discernible. However, two patches become visible with additional resolution. The finer mesh is composed of $\sim$ 6,200,000 elements, with $\sim$ 5,000,000 elements of minimum cell width $h \approx 5.8 \cdot 10^{-7}$ in the neighborhood of the triple point.

5.2.2. *Shock diffraction around a thin film.* We now consider the problem of a shock interacting with a thin, reflecting film on a square domain $[-1.125, 1.125]^2$. The thin film is located on the line $\xi = 0$ between $\eta = -1.125$ and $\eta = 0$. The incident, horizontally oriented shock propagates downward to the thin film. Once it hits the film, it reflects and diffracts around the obstacle. The incident shock is weak and propagates at Mach 1.075. As the shock diffracts, it transforms into an expansion wave in a self-similar

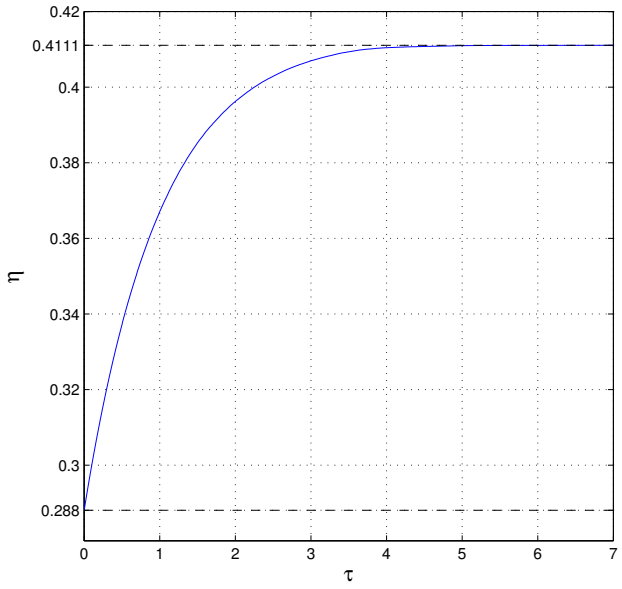
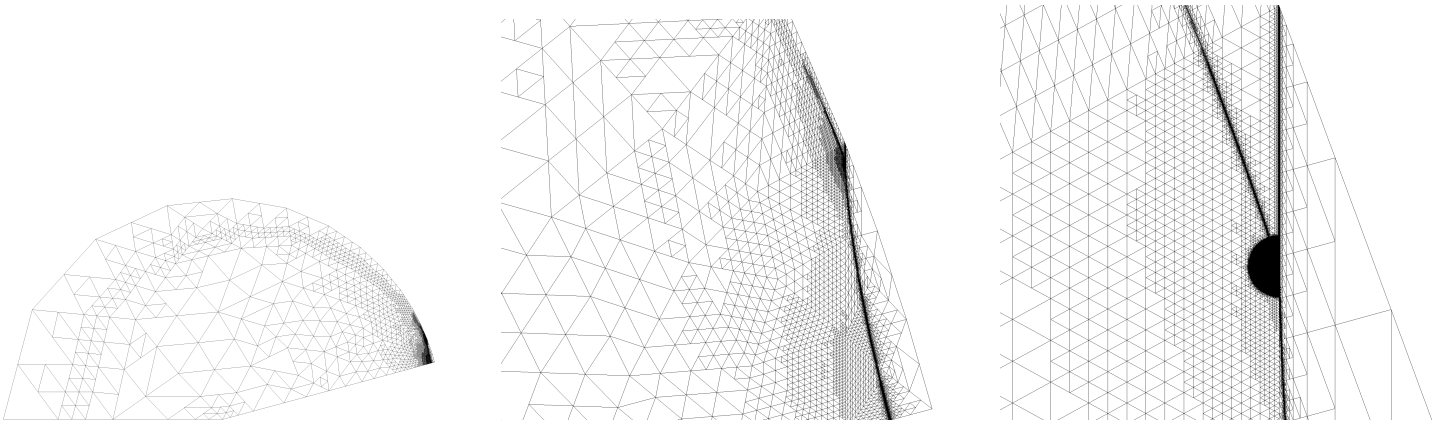


FIGURE 28. The vertical coordinate of the triple point vs. pseudotime.



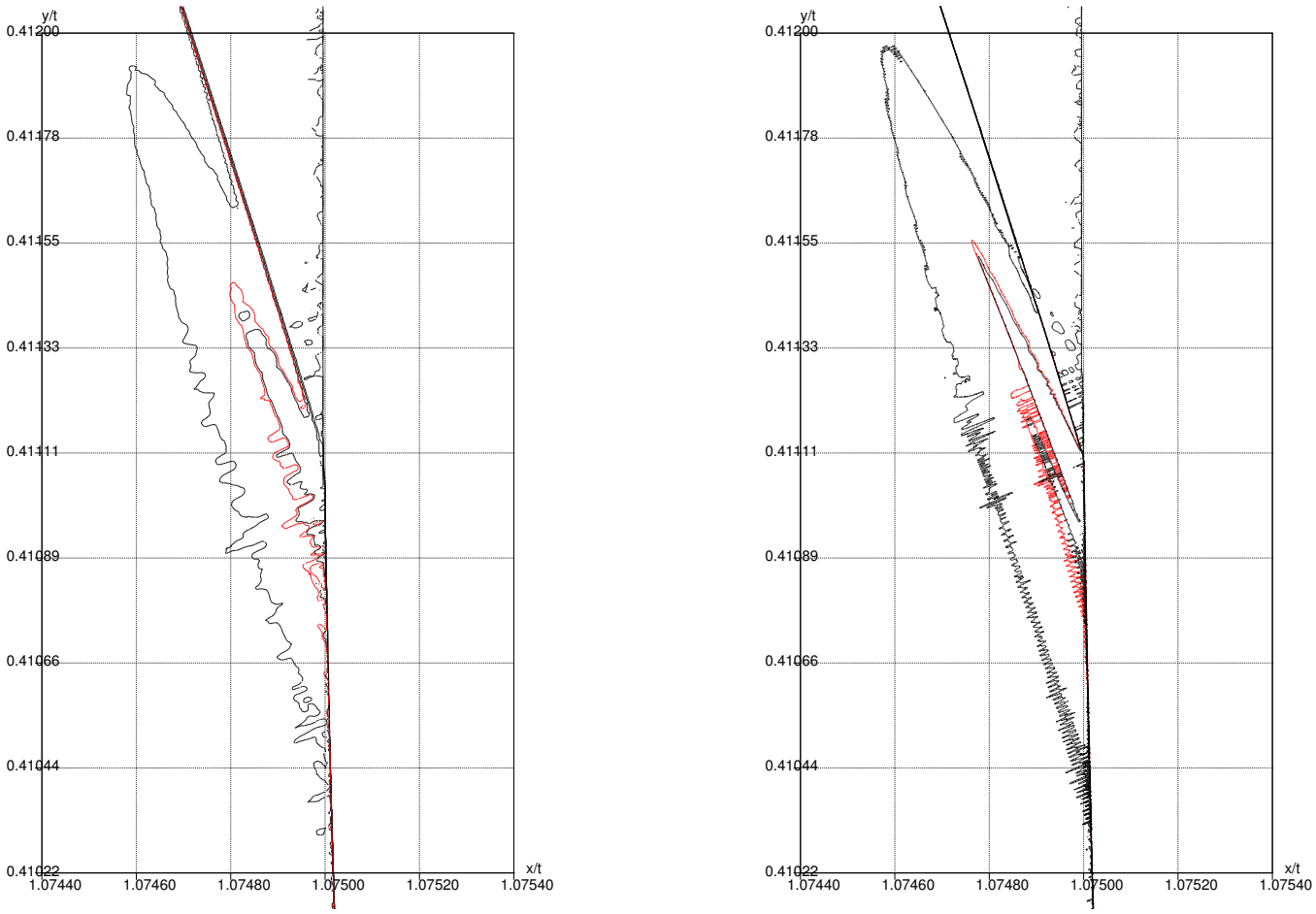
(A) Adaptively refined mesh for the solution in Figure 30a.

(B) First zoom of Mach stem and triple point region in Figure 29a.

(C) Second zoom of Mach stem and triple point region in Figure 29b.

FIGURE 29. The adaptively refined mesh for the solution in Figure 30a.

fashion. Another characteristic of the flow is the development of a vortex at the corner of the thin film. The incident (I) and reflected (R) shocks as well as the point (P) where the shock disappears are illustrated in Figure 31a, after the incident shock has passed the thin film. Since this interaction is self-similar, this figure also illustrates the boundary conditions applied in (ξ, η) coordinates, which are given by three constant states $\mathbf{U}_R$, $\mathbf{U}_I$, $\mathbf{U}_Q$, i.e., the reflected, incident, and quiescent states (Table 7). The thin film is modeled by a reflecting internal boundary condition, indicated on the initial mesh in Figure 31b by a bold line. The reflected shock state $\mathbf{U}_R$ is computed from the incident shock state $\mathbf{U}_I$ by solving a one-dimensional Riemann problem about the thin film (reflecting boundary).



(A) $h \approx 4.3 \cdot 10^{-6}$ (B) $h \approx 5.8 \cdot 10^{-7}$

FIGURE 30. 6 isolines for self-similar Mach numbers $\tilde{M}$ in the range 0.996 to 1.02. The red isoline corresponds to the sonic line, i.e., $\tilde{M} = 1$. The h is the approximate minimum cell width in the neighborhood of the supersonic patches.

We compute the sonic function

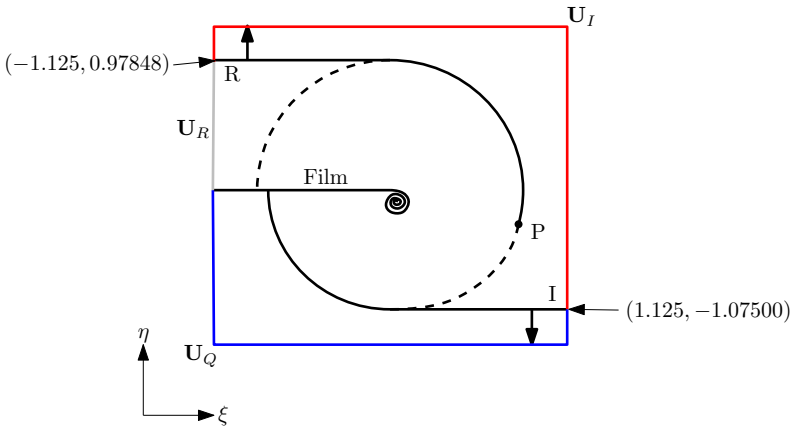
$$S = \sqrt{(u-\xi)^2 + (v-\eta)^2} - c.$$

The flow is subsonic when $S < 0$, supersonic when $S > 0$, and the sonic line is located where $S = 0$. For this problem, refinement is driven by proximity to the sonic line. The final mesh is comprised of $\sim$ 2,800,000 elements where the smallest resolution is $h \approx 4.8 \cdot 10^{-5}$ in the neighborhood of the sonic line.

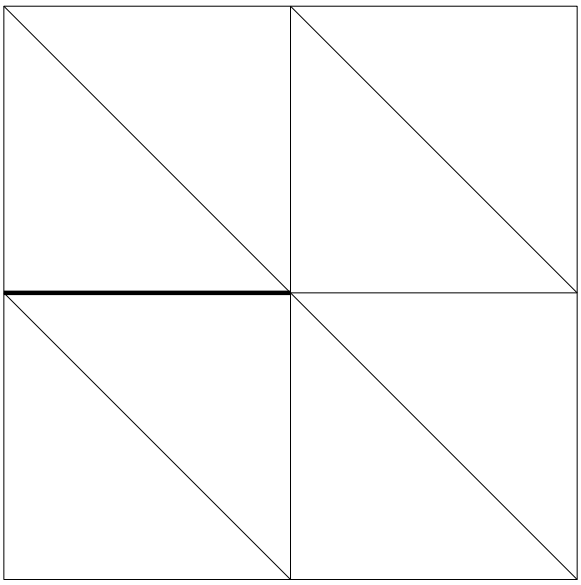
The sonic function of the solution is plotted in Figure 32a. In Figures 32b and 32c, cross sections of S are provided in the neighborhood of the shock disappearance point on $\eta = 0, -0.05, ..., -0.25$. On these cross sections, the shock is visible at the coordinates (ξ, η): $(0, 1.0173)$, $(-0.05, 1.0219)$, and $(-0.1, 1.0240)$. The location of shock disappearance seems to be about $\sim (-0.15, 1.0237)$, a more precise location is difficult to determine. Finally, shock disappearance seems to occur on the sonic line as indicated in [33] for the UTSDE.

	$\mathbf{U}_I$	$\mathbf{U}_R$	$\mathbf{U}_Q$
ρ	1.57697	1.77139	1.4
u	0	0	0
v	-0.12064	0	0
p	1.18156	1.39066	1

TABLE 7. The incident $\mathbf{U}_I$, reflected $\mathbf{U}_R$, and quiescent $\mathbf{U}_Q$ states imposed as boundary conditions in Example 5.2.2.



(A) The incident (I) shock propagates into quiescent (Q) gas, diffracts around, and reflects (R) off the thin film. The shock disappears at P. The incident $\mathbf{U}_I$, reflected $\mathbf{U}_R$, and quiescent $\mathbf{U}_Q$ states are imposed as boundary conditions on the red, gray, and blue boundaries, respectively.

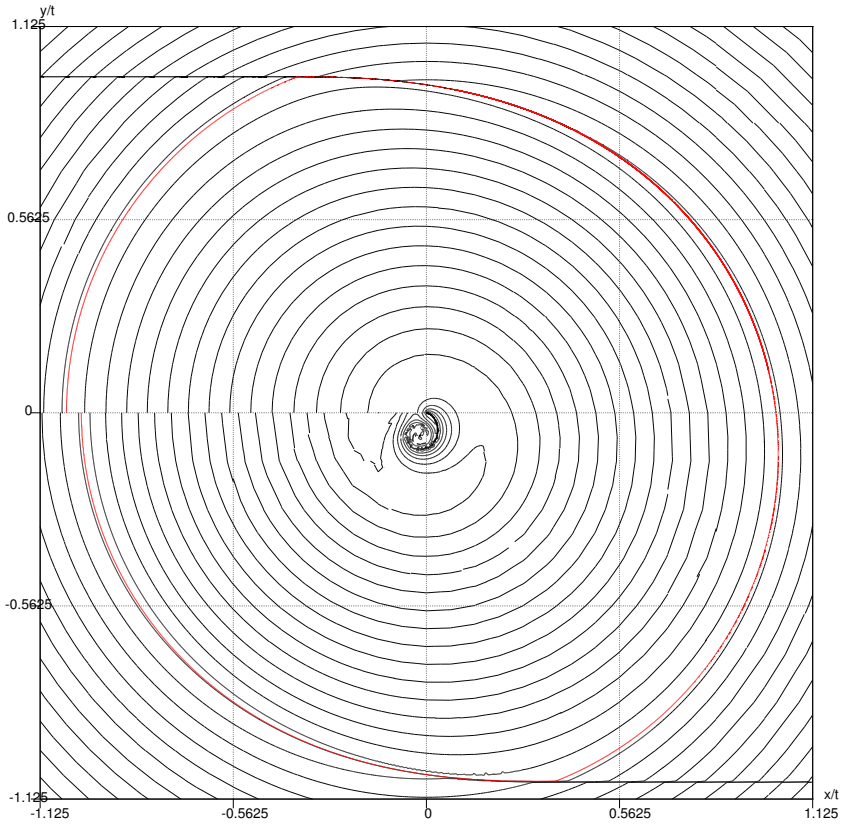


(B) Initial mesh for the shock diffraction problem. The bolded line corresponds to the thin film.

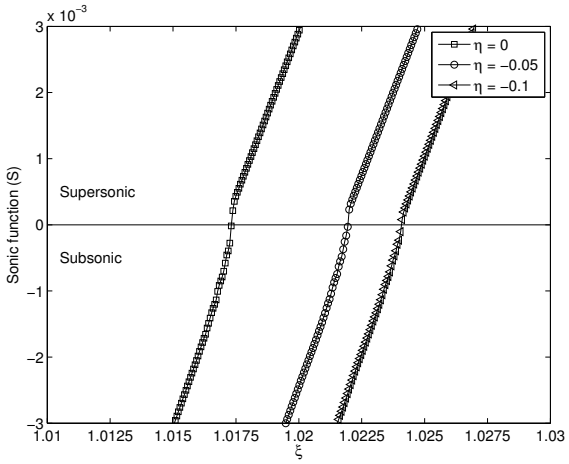
FIGURE 31. Initial setup and mesh for the shock diffraction problem in Example 5.2.2.

6. CONCLUSION

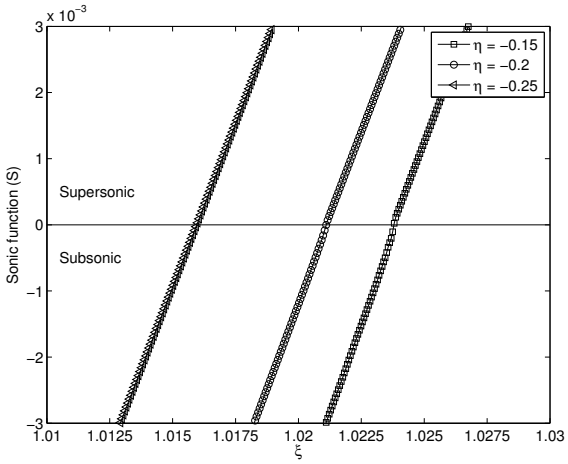
We have outlined a GPU-parallelized h-adaptive implementation of the DG method for hyperbolic conservation laws on unstructured meshes. The highlights of this implementation are memory management and the use of a coloring algorithm to eliminate race conditions. Our memory management techniques allow for quickly resizing the data arrays resulting in the smallest number of necessary memory transfers. This is done by using a modified stream-compaction operation. The coloring algorithm prevents race conditions in the evaluation of an integral over cell edges. It is also used in the smoothing subroutines to ensure proper nonconformity between elements. In fact, the smoothing module is the most expensive part of the AMR algorithm. This procedure can easily be done recursively on CPUs, but it is difficult to implement efficiently on GPUs. This is because the smoothing on one element may trigger smoothing of its neighbors. Using coloring yields a lightweight implementation relative to an element-wise operation.



(A) Isolines of sonic function. The red isoline corresponds to the sonic line, i.e., where $S = 0$.



(B) Cross sections of S on $\eta = 0, -0.05, -0.1$.



(C) Cross sections of S on $\eta = -0.15, -0.2, -0.25$.

FIGURE 32. Sonic function (S) and its cross sections for the shock diffraction problem in Example 5.2.2.

We have presented a number of computed examples in gas dynamics demonstrating the performance of the AMR algorithm. In particular, we computed two problems that are intractable without AMR due to the extremely high resolution required to resolve the solution features. We present numerical evidence that further supports Guderley's solution to the von Neumann triple point paradox. We also include numerical

experiments that suggest shock disappearance occurs on the sonic line in the self-similar diffraction of a shock around a thin film. To our knowledge, these are the first results to the shock diffraction problem on the full Euler equations.

7. Acknowledgment

This work was supported in part by the Natural Sciences and Engineering Research Council of Canada grant 341373-07 and an Alexander Graham Bell PGS-D grant. We gratefully acknowledge the support of the NVIDIA Corporation with the donation of hardware used for this research.

References

[1] A. Karakus, T. Warburton, M. Aksel, and C. Sert, "A GPU-accelerated adaptive discontinuous Galerkin method for level set equation," *International Journal of Computational Fluid Dynamics*, vol. 30, no. 1, pp. 56–68, 2016.

[2] J. Chan, Z. Wang, A. Modave, J.-F. Remacle, and T. Warburton, "GPU-accelerated discontinuous Galerkin methods on hybrid meshes," *Journal of Computational Physics*, vol. 318, pp. 142–168, 2016.

[3] M. Fuhry, A. Giuliani, and L. Krivodonova, "Discontinuous Galerkin methods on graphics processing units for nonlinear hyperbolic conservation laws," *International Journal for Numerical Methods in Fluids*, vol. 76, no. 12, pp. 982–1003, 2014.

[4] R. Gandham, D. Medina, and T. Warburton, "GPU accelerated discontinuous Galerkin methods for shallow water equations," *Communications in Computational Physics*, vol. 18, no. 1, pp. 37–64, 2015.

[5] J. Chan and T. Warburton, "GPU-accelerated Bernstein–Bézier discontinuous Galerkin methods for wave problems," *SIAM Journal on Scientific Computing*, vol. 39, no. 2, pp. A628–A654, 2017.

[6] M. de la Asunción and M. Castro, "Simulation of tsunamis generated by landslides using adaptive mesh refinement on GPU," *Journal of Computational Physics*, vol. 345, pp. 91–110, 2017.

[7] D. S. Abdi, L. C. Wilcox, T. C. Warburton, and F. X. Giraldo, "A GPU-accelerated continuous and discontinuous Galerkin non-hydrostatic atmospheric model," *The International Journal of High Performance Computing Applications*, p. 1094342017694427, 2017.

[8] A. Giuliani and L. Krivodonova, "Face coloring in unstructured CFD codes," *Parallel Computing*, vol. 63, pp. 17–37, 2017.

[9] C. Pain, A. Umpleby, C. De Oliveira, and A. Goddard, "Tetrahedral mesh optimisation and adaptivity for steady-state and transient finite element calculations," *Computer Methods in Applied Mechanics and Engineering*, vol. 190, no. 29-30, pp. 3771–3796, 2001.

[10] S. M. Schnepp and T. Weiland, "Efficient large scale electromagnetic simulations using dynamically adapted meshes with the discontinuous Galerkin method," *Journal of Computational and Applied Mathematics*, vol. 236, no. 18, pp. 4909–4924, 2012.

[11] C. Eskilsson, "An hp-adaptive discontinuous Galerkin method for shallow water flows," *International Journal for Numerical Methods in Fluids*, vol. 67, no. 11, pp. 1605–1623, 2011.

[12] P. MacNeice, K. M. Olson, C. Mobarry, R. De Fainchtein, and C. Packer, "PARAMESH: A parallel adaptive mesh refinement community toolkit," *Computer physics communications*, vol. 126, no. 3, pp. 330–354, 2000.

[13] M. Adams, "CHOMBO software package for AMR applications–design document, Lawrence Berkeley National Laboratory technical report lbnl-6616e," 2011.

[14] W. Bangerth, R. Hartmann, and G. Kanschat, "deal.II – a general purpose object oriented finite element library," *ACM Trans. Math. Softw.*, vol. 33, no. 4, pp. 24/1–24/27, 2007.

[15] K. T. Mandli, A. J. Ahmadia, M. Berger, D. Calhoun, D. L. George, Y. Hadjimichael, D. I. Ketcheson, G. I. Lemoine, and R. J. LeVeque, "Clawpack: building an open source ecosystem for solving hyperbolic PDEs," *PeerJ Computer Science*, vol. 2, p. e68, 2016.

[16] M. J. Berger and J. Oliger, "Adaptive mesh refinement for hyperbolic partial differential equations," *Journal of Computational Physics*, vol. 53, no. 3, pp. 484 – 512, 1984.

[17] J. Z. X. Zheng, *Block-based adaptive mesh refinement finite-volume scheme for hybrid multi-block meshes*. PhD thesis, 2012.

[18] L. Ivan, H. D. Sterck, S. A. Northrup, and C. Groth, "Multi-dimensional finite-volume scheme for hyperbolic conservation laws on three-dimensional solution-adaptive cubed-sphere grids," *Journal of Computational Physics*, vol. 255, no. 0, pp. 205 – 227, 2013.

[19] C. Burstedde, L. C. Wilcox, and O. Ghattas, "p4est: Scalable algorithms for parallel adaptive mesh refinement on forests of octrees," *SIAM Journal on Scientific Computing*, vol. 33, no. 3, pp. 1103–1133, 2011.

[20] K. D. Devine and J. E. Flaherty, "Parallel adaptive hp-refinement techniques for conservation laws," *Applied Numerical Mathematics*, vol. 20, no. 4, pp. 367–386, 1996.

[21] T. Richter, *Parallel multigrid method for adaptive finite elements with application to 3D flow problems*. PhD thesis, University of Heidelberg, 2005.

[22] O. Zanotti and M. Dumbser, "A high order special relativistic hydrodynamic and magnetohydrodynamic code with space-time adaptive mesh refinement," *Computer Physics Communications*, vol. 188, pp. 110–127, 2015.

[23] G. Rokos, G. Gorman, and P. H. Kelly, "Accelerating anisotropic mesh adaptivity on nVIDIAs CUDA using texture interpolation," in *European Conference on Parallel Processing*, pp. 387–398, Springer, 2011.

[24] Y. Xia, J. Lou, H. Luo, J. Edwards, and F. Mueller, "OpenACC acceleration of an unstructured CFD solver based on a reconstructed discontinuous Galerkin method for compressible flows," *International Journal for Numerical Methods in Fluids*, vol. 78, no. 3, pp. 123–139, 2015.

[25] M. J. Berger and R. J. LeVeque, "Adaptive mesh refinement using wave-propagation algorithms for hyperbolic systems," *SIAM Journal on Numerical Analysis*, vol. 35, no. 6, pp. 2298–2316, 1998.

[26] K. Schaal, A. Bauer, P. Chandrashekar, R. Pakmor, C. Klingenberg, and V. Springel, "Astrophysical hydrodynamics with a high-order discontinuous Galerkin scheme and adaptive mesh refinement," *Monthly Notices of the Royal Astronomical Society*, vol. 453, no. 4, pp. 4278–4300, 2015.

[27] W. Bangerth and R. Rannacher, *Adaptive finite element methods for differential equations*. Birkhäuser, 2013.

[28] K. G. Guderley, *Considerations on the structure of mixed subsonic-supersonic flow patterns*. Headquarters Air Materiel Command, 1947.

[29] A. M. Tesdall, R. Sanders, and B. L. Keyfitz, "Self-similar solutions for the triple point paradox in gasdynamics," *SIAM Journal on Applied Mathematics*, vol. 68, no. 5, pp. 1360–1377, 2008.

[30] E. I. Vasilev and A. N. Kraiko, "Numerical simulation of weak shock diffraction over a wedge under the von Neumann paradox conditions," *Zhurnal Vychislitel'noi Matematiki i Matematicheskoi Fiziki*, vol. 39, no. 8, pp. 1393–1404, 1999.

[31] B. W. Skews and J. T. Ashworth, "The physical nature of weak shock wave reflection," *Journal of Fluid Mechanics*, vol. 542, pp. 105–114, 2005.

[32] A. M. Tesdall, "High resolution solutions for the supersonic formation of shocks in transonic flow," *Journal of Hyperbolic Differential Equations*, vol. 8, no. 03, pp. 485–506, 2011.

[33] A. M. Tesdall and J. K. Hunter, "Self-similar solutions for the diffraction of weak shocks," *Journal of Computational Science*, vol. 4, no. 1-2, pp. 92–100, 2013.

[34] M. Dubiner, "Spectral methods on triangles and other domains," *Journal of Scientific Computing*, vol. 6, pp. 345–390, Dec 1991.

[35] D. Merrill, "CUB." https://nvlabs.github.io/cub/index.html. version 1.5.1.

[36] A. Giuliani and L. Krivodonova, "Analysis of slope limiters on unstructured triangular meshes," *Journal of Computational Physics*, vol. 374, pp. 1 – 26, 2018.

[37] P. E. Vincent, P. Castonguay, and A. Jameson, "Insights from von Neumann analysis of high-order flux reconstruction schemes," *Journal of Computational Physics*, vol. 230, no. 22, pp. 8134–8154, 2011.

[38] N. Chalmers and L. Krivodonova, "A robust CFL condition for the discontinuous Galerkin method on triangular meshes," *Draft available at* `http://www.math.uwaterloo.ca/~lgk/A_Characteristic_Based_CFL_Number.pdf`.

[39] V. Springel, "E pur si muove: Galilean-invariant cosmological hydrodynamical simulations on a moving mesh," *Monthly Notices of the Royal Astronomical Society*, vol. 401, no. 2, pp. 791–851, 2010.

[40] P. Woodward and P. Colella, "The numerical simulation of two-dimensional fluid flow with strong shocks," *Journal of Computational Physics*, vol. 54, no. 1, pp. 115–173, 1984.

[41] A. Zakharian, M. Brio, J. Hunter, and G. Webb, "The von Neumann paradox in weak shock reflection," *Journal of Fluid Mechanics*, vol. 422, pp. 193–205, 2000.

[42] A. M. Tesdall, R. Sanders, and N. Popivanov, "Further results on Guderley Mach reflection and the triple point paradox," *Journal of Scientific Computing*, vol. 64, no. 3, pp. 721–744, 2015.