Proceedings of the Canadian Society for Mechanical Engineering International Congress
32nd Annual Conference of the Computational Fluid Dynamics Society of Canada
Canadian Society of Rheology Symposium
CSME/CFDSC/CSR 2025
May 25–28, 2025, Montréal, Québec, Canada

N/A

TOWARDS A GPU-NATIVE ADAPTIVE MESH REFINEMENT SCHEME FOR THE LATTICE BOLTZMANN METHOD IN COMPLEX GEOMETRIES

Khodr Jaber[1]∗, Ebenezer E. Essel[2], Pierre E. Sullivan[1,1],

[1]Department of Mechanical and Industrial Engineering, University of Toronto, Toronto, Canada
[2]Department of Mechanical, Industrial and Aerospace Engineering, Concordia University, Montreal, Canada
∗khodr.jaber@mail.utoronto.ca

February 17, 2025

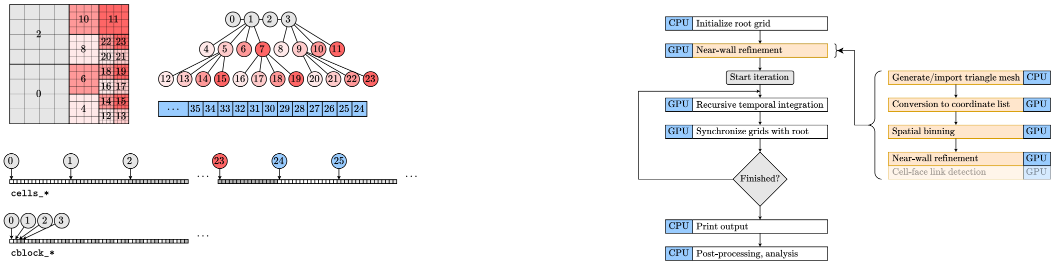
***Abstract*—**

We present a GPU-native mesh adaptation procedure that incorporates a complex geometry represented with a triangle mesh within a primary Cartesian computational grid organized as a forest of octrees. A C++/CUDA program implements the procedure for execution on a single GPU as part of a new module with the AGAL framework, which was originally developed for GPU-native adaptive mesh refinement (AMR) and fluid flow simulation with the Lattice Boltzmann Method (LBM). Traditional LBM is limited to grids with regular prismatic cells with domain boundaries aligned with the cell faces. This work is a first step towards an implementation of the LBM that can simulate flow over irregular surfaces while retaining both adaptation of the mesh and the temporal integration routines entirely on the GPU. Geometries can be inputted as a text file (which generates primitive objects such as circles and spheres) or as an STL file (which can be generated by most 3D modeling software). The procedure is divided into three steps: 1) an import step where the geometry is loaded into either an index list arrangement or directly as a face-vertex coordinates list, 2) a spatial binning step where the faces are distributed to a set of bins with user-defined density, and 3) a near-wall refinement step where the cells of the computational grid detect adjacency to the faces stored in the appropriate bin to form the links between the geometry and the boundary nodes. We validate the implementation and assess its performance in terms of total execution time and speedup relative to a serial CPU implementation using a 2D circle and a 3D Stanford bunny.

***Keywords-component*—Adaptive Mesh Refinement; General-Purpose GPU (GPGPU); Triangle Mesh; Complex Geometry; Open-Source**

I. INTRODUCTION

Adaptive mesh refinement (AMR) provides a flexible non-uniformity to the underlying grid in computational physics simulations that can be adjusted at runtime. This is especially important for computational fluid dynamics simulations based on the Lattice Boltzmann Method (LBM) — an emerging competitor to conventional numerical methods for weakly-compressible flow that is well-known for its suitability for parallel implementation with GPUs — which is traditionally limited to uniform grids due to the discretization procedure. The LBM has been applied to phase-field modeling [1], free-surface flow [2], and external flows [3]. Many grid refinement schemes have been developed in the past two decades for the LBM; we refer the reader to the works of Gendre et al. [4] and Schukmann et al. [5] for further reading. Several open-source packages implement the LBM with grid refinement such as Palabos [6], waLBerla [7], and ESPResSo [8]. An AMR scheme in computational fluid dynamics consists of the mesh adaptation routines, which refine the grid according to a user-specified criterion based on the underlying geometry or physics, and the solver routines, which describe coarse-fine grid coupling along refinement interfaces during temporal integration. GPU-acceleration of AMR has also become popular due to advancements in hardware and the increased availability of heterogeneous high-performance computing platforms offered by packages such as AMReX [9] and Daino [10]. It is common for the mesh to be managed and adapted on the CPU, and the data retained and advanced entirely on the GPU due to the challenges associated with organizing the



(a) Forest-of-octrees grid in 2D.

(b) Interaction of Mesh and Geometry modules.

Figure. 1: Left: Nodes in the tree correspond to blocks in the grid and identify locations in the data arrays. Right: scope of the current work. The Geometry module replaces the old near-wall refinement routine and enables grid cells to identify the proper domain boundary.

required structures on the GPU. However, GPU-native AMR has recently been achieved, where the mesh and data are both processed on the GPU with specialized algorithms and data structures [11]–[13]. In the context of the LBM, the available software packages offering dynamic adaptation of the mesh with GPU-acceleration continue to use the hybrid approach where the mesh is hosted on the CPU, to the best of our knowledge.

We recently developed our own GPU-native AMR approach tailored for Lattice Boltzmann simulations [14], however, the code was limited to square/cubic domains. This work presents a first step towards the integration of complex geometries (in the form of edge/triangle meshes) within our AMR framework for improved and more realistic fluid flow simulations with the LBM.

II. METHODOLOGY

This section describes the data structures and algorithms that represent the complex geometry and its incorporation in the forest-of-octree computational grid. We describe how the geometry, decomposed into a set of edges or triangles, is distributed to CUDA thread blocks via spatial binning to enable efficient mesh refinement and the imposition of boundary conditions.

A. Adaptive Mesh Refinement on GPUs

In an earlier work [14], we described an adaptation algorithm for a mesh organized as a forest of octrees tailored for execution on a single GPU, and a recursive time-stepping scheme for the LBM on this mesh. The algorithm was implemented in 2D and 3D in an open-source C++/CUDA code[1] called the AGAL framework. We will briefly summarize the organization of the mesh and the solver scheme before presenting the complex geometry components.

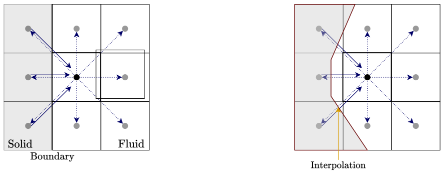


Figure. 2: Half-way bounce-back boundary conditions are second-order accurate for geometries aligned with the cell edges/faces, but interpolation or ghost methods are needed to maintain accuracy for more general surfaces.

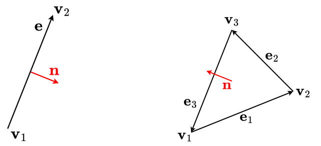
An octree is a tree data structure in which each node is split into exactly eight children. The unique node without a parent is referred to as the root node. A node's level in the tree is the number of parent-child links between it and the root node. Each node corresponds to a single block in the grid composed of 4^D cells (Figure 1a), where D is the number of dimensions. Integer indices, referred to as block IDs, uniquely determine the position of the block and cell data in the solution and metadata arrays. Blocks on a level L are denoted as the grid level L. The forest of octrees is enumerated with a set of ID sets $\{\texttt{id\_set}_L\}_{L=0}^{\text{max.}}$, which store the block IDs by grid level.

The Mesh class implements the forest of octrees structure. The root nodes of all octrees are arranged as a structured grid denoted as the root grid, which can be refined near the domain boundaries before the solver loop takes place, and/or dynamically within the loop. We performed near-wall refinement [14] in the lid-driven cavity and flow past a square cylinder benchmark tests by hard-coding the near-wall distance formula according to the known simple geometry (i.e., refining blocks that are a certain distance away from the four/six faces of the square/cubic domain or from the cylinder). Basic bounce-back boundary conditions for the LBM are accurate for this type of geometry but require interpolation or ghost-cell methods when the geometry is not specifically aligned with the cell edges/faces (Figure 2). The procedures outlined in this paper specifically address the issue of meshing (Figure 1b). Work on the updated LBM scheme is in progress.

B. Representation of the Complex Boundary Geometry

The geometry is composed of a set of edges/triangles (Figure 3) and represented in the form of a set of index lists (i.e., vertex coordinates stored in one array and indices of the edge/triangle vertices in another), or as a single coordinate list for the edges/triangles (which may allow duplicate storage of vertex coordinates). These are illustrated in Figure 4. In this paper, we will refer to both

[1] The code repository is hosted on Github: https://github.com/KhodrJ/AGAL.



(a) Edge (2D). (b) Triangle (3D).

Figure. 3: Visualization of the current edge and triangle orientations.

edges and triangles as faces without loss of generality. The routine `G_ImportFromTextFile` constructs primitive sub-meshes using user-supplied resolution and location values and appends them to vectors that store the index lists. `G_ConvertIndexListsToCoordsList` converts these index lists into a single array of triangle coordinates `geom_f_face_X` that is organized in a structure of arrays format. This is the more suitable form for the CUDA kernels to be introduced shortly. Alternatively, we provide a routine `G_ImportSTL` that reads the vertex coordinates of the faces to directly build the coordinate list. The list is then permanently transferred to the GPU, and it remains accessible by the `Mesh` object to be passed to the refinement and coarsening and solver routines.

C. *Cell-Face Identification Algorithms*

1) Identification of Cells Near the Boundary: A naive implementation of the face-detection scheme entails a traversal of the blocks in the primary computational grid, where each block checks if it is adjacent to at least one edge/triangle in the whole geometry. The implementation of e.g., interpolated boundary conditions, requires that cells are able to identify links with the appropriate faces of the domain boundary, compute the distance along a vector defined by the quadrature abscissae in discretizing the Boltzmann equation in particle-velocity space (i.e., a direction in the velocity set), and finally computing the interpolated density distribution function.

Blocks in the grid are traversed as per the so-called primary mode of access as defined in [14]. Each grid level is traversed using individual ID sets (i.e., one level at a time), and CUDA threads first read a fixed number of block IDs. Then, in a for-loop, the threads are assigned to each cell in the cell-block and its cells are processed simultaneously. The thread-block size is equal to the cell-block size to ensure that all threads participate in updating the cell-block data.

geom_ID_face

0	1	2	3	4	5	6
$v_{0,x}$	$v_{1,x}$	$v_{2,x}$	$v_{3,x}$	$v_{4,x}$	$v_{5,x}$	$v_{6,x}$
$v_{0,y}$	$v_{1,y}$	$v_{2,y}$	$v_{3,y}$	$v_{4,y}$	$v_{5,y}$	$v_{6,y}$
$v_{0,z}$	$v_{1,z}$	$v_{2,z}$	$v_{3,z}$	$v_{4,z}$	$v_{5,z}$	$v_{6,z}$

geom_f_node_X

0	1	2	3	4	5
0	1	2	3	2	0
1	3	3	4	5	2
2	2	5	5	6	6

geom_f_face_X

0	1	2	3	4	5
$v_{0,x}$	$v_{1,x}$	$v_{2,x}$	$v_{3,x}$	$v_{2,x}$	$v_{0,x}$
$v_{1,x}$	$v_{3,x}$	$v_{3,x}$	$v_{4,x}$	$v_{5,x}$	$v_{2,x}$
$v_{2,x}$	$v_{2,x}$	$v_{5,x}$	$v_{5,x}$	$v_{6,x}$	$v_{6,x}$
$v_{0,y}$	$v_{1,y}$	$v_{2,y}$	$v_{3,y}$	$v_{2,y}$	$v_{0,y}$
$v_{1,y}$	$v_{3,y}$	$v_{3,y}$	$v_{4,y}$	$v_{5,y}$	$v_{2,y}$
$v_{2,y}$	$v_{2,y}$	$v_{5,y}$	$v_{5,y}$	$v_{6,y}$	$v_{6,y}$
$v_{0,z}$	$v_{1,z}$	$v_{2,z}$	$v_{3,z}$	$v_{2,z}$	$v_{0,z}$
$v_{1,z}$	$v_{3,z}$	$v_{3,z}$	$v_{4,z}$	$v_{5,z}$	$v_{2,z}$
$v_{2,z}$	$v_{2,z}$	$v_{5,z}$	$v_{5,z}$	$v_{6,z}$	$v_{6,z}$

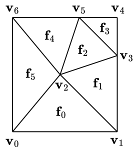


Figure. 4: Index (`geom_f_node_X`, `geom_ID_face`) and coordinate list (`geom_f_face_X`) representations of a sample triangle mesh. Yellow cells indicate the indices of the elements.

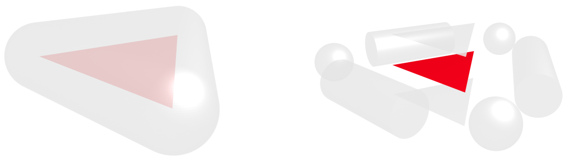
For near-wall refinement and cell-face link identification, each cell loops over the known number of faces and, for each face, the vertex coordinates are loaded from `geom_f_face_X`. A face detection algorithm commences whereby the cell checks if its center is a specified distance away from the face. If so, the cell's block is marked for refinement. The cell can potentially store the faces that is identifies in a separate list for future use in the solver routines as well. The refinement and coarsening procedure (`M_RefineAndCoarsenBlocks`, detailed in [14]) then subdivides the marked blocks.

2) Face Detection: A point is considered adjacent to a face if the distance between the two is less than the user-specified near-wall distance $d_{\text{spec.}}$. For a triangle, this requires that the distances from the point to both the planar region and the individual edge segments are less than $d_{\text{spec.}}$. The point may be collinear to the plane (e.g., at a corner) but near one of the edge segments, in which case it is also considered adjacent. This is equivalent to checking that the point is in the region defined by three circles centered on the vertices, three cylinders oriented along the edges, and a triangular prism oriented along the plane (Figure 5, Algorithm 1). The radius of the sphere/cylinder and height of the prism are both equal to $d_{\text{spec.}}$. In 2D, this degenerates to a check on two circles centered on the vertices and a rectangular region aligned length-wise with the edge. We use the point-line distance formula [15] when testing against the cylinders, and the half-space test when testing against the triangular prism.

3) Spatial Binning Algorithm: The naive implementation is highly inefficient: accessing all faces from each cell incurs a large global memory access cost, and this is unnecessary since only relatively few cells in the whole domain are in the vicinity of the boundary. We employ a spatial binning

Algorithm 1 Check if point is in range of a triangle

```
procedure CheckNearTriangle($\mathbf{x}_p, R$)
    for all $k \in \{1, 2, 3\}$ do
        if $\left(\|\mathbf{x}_p - \mathbf{v}_k\|^2 \leq R\right)$ then                ▷ Sphere
            return true
        end if
        $d^2 \leftarrow \dfrac{\|(\mathbf{v}_{k+1} - \mathbf{v}_k) \times (\mathbf{v}_k - \mathbf{x}_p)\|^2}{\|\mathbf{v}_{k+1} - \mathbf{v}_k\|^2}$
        $d_{e,1} \leftarrow -(\mathbf{v}_k - \mathbf{x}_p) \cdot \mathbf{e}_k$
        $d_{e,2} \leftarrow (\mathbf{v}_{k+1} - \mathbf{x}_p) \cdot \mathbf{e}_k$
        if $\left(d^2 \leq R \text{ and } d_{e,1}, d_{e,2} \geq 0\right)$ then                ▷ Cylinders
            return true
        end if
    end for
    $C_1 \leftarrow \bigwedge_{k=1}^{3} \left(-(\mathbf{v}_k - \mathbf{x}_p) \cdot (\mathbf{e}_k \times \mathbf{n}) \geq 0\right)$
    $C_2 \leftarrow (-(\mathbf{v}_1 - R\mathbf{n} - \mathbf{x}_p) \cdot \mathbf{n}) \wedge ((\mathbf{v}_1 + R\mathbf{n} - \mathbf{x}_p) \cdot \mathbf{n})$
    if $C_1$ and $C_2$ then                ▷ Prism
        return true
    end if
end procedure
```



(a) Near-wall region. (b) Subdomains to be checked.

Figure. 5: Visualization of the near-wall region around a single triangle oriented in 3D.

approach to reduce the search-loop size (i.e., the set of all faces that need to be searched). The initial computational grid, which is rectangular/prismatic in shape, can be divided into a set of regularly-sized bins. Faces are assigned to bins if they intersect with them or are enclosed entirely by them. While it is possible to check for intersection by using line-plane tests (point-line tests in 2D), we choose to discretize the faces with a number of nodes that (Figure 7) scale with the size of the face, and to check if at least one point lies in the bin. We use three arrays to represent the binning: `binned_face_ids`, which stores the face IDs of each bin in order by group, `binned_face_ids_n`, the number of faces in each bin, and `binned_face_ids_N`, the indices in `binned_face_ids` where the first face of each bin is located (Figure 6).

The implementation is characterized by bin density B, the number of bins to consider along each Cartesian axis (for a total of $N_{\text{bins}} = B^D$, and bin fraction B_f, the number of bins to update at a time in the CUDA kernel. The bin fraction determines the amount of memory to allocate in GPU memory. A larger value means that the workload is divided a greater number of times so that fewer bins are updated at a time (which requires a smaller allocation). The bin length is obtained by normalizing the domain size by the bin density. Since faces may be duplicated when crossing bins, a larger amount of memory is allocated to ensure that there is enough space at the beginning. We chose 10x the number of faces, though experiments should that the overlap is relatively minimal. A bin indicator array is also defined for the faces in a structure of arrays format such that each face can indicate occupancy in up to B_n= N_{bins}/B_f bins.

Looping over B_f, an amount B_n are updated with the `Cu_FillBins` kernel. The face coordinates list is traversed with CUDA threads assigned to each face, and each face loops over B_n bins. If a face is detected as occupying a bin, it is assigned by modifying the appropriate indicator with the face ID. After the kernel execution is completed, a stream compaction is performed where the face IDs in the indicator array are copied into `binned_face_ids` to ensure that they are contiguous in memory. We use Thrust's [16] `count_if` and `copy_if` to count the number of IDs to be copied and then to perform the copy, respectively.

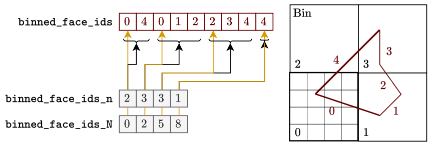
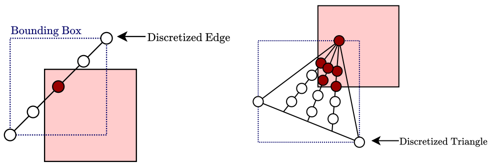


Figure. 6: Illustration of the output of the binning procedure.



(a) Discretized edge. (b) Discretized triangle.

Figure. 7: Discretization of edges and triangles during spatial binning. Discretization of a triangle begins with one edge, followed by the segments formed from the discrete points on the edge and the vertex opposite to it.

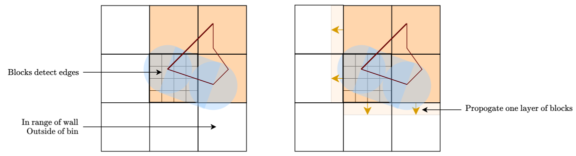


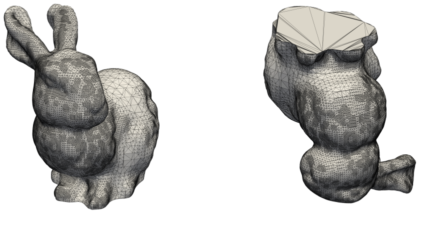
Figure. 8: Illustration of refinement mark propagation for binning with a large near-wall refinement distance criterion.

When the number of bins is increased, cells within a large specified near-wall distance may fail to detect faces. To meet the requirement, we introduce a propagation step (Figure 8) where blocks marked for refinement check their neighboring blocks and mark them for refinement as well until a total distance equal to $d_{\text{spec.}}$ has been covered (i.e., a total of $N_{\text{prop.}} = 1 + \lfloor d_{\text{spec.}}/\Delta x_b \rfloor$ are performed, where Δx_b is the cell-block length). The propagation is done in two steps to avoid a race condition: the refinement IDs of the nearby blocks are first switched to an intermediate value, and then the intermediate values are replaced with a mark for refinement in a second traversal. The first step follows the so-called secondary mode of memory access as defined in [14].

III. Results and Discussion

We record the execution times of the implemented routines for two test geometries to assess the performance of the implementation: a 2D circle and a 3D Stanford bunny [17], [18][2] (Figure 9). The Stanford bunny is known for its use as a benchmark in the field of computer graphics; however, we employ it here to demonstrate the robustness of the current methodology in the presence of both concave and convex surfaces, as well as surfaces spanning a wide range of sizes with respect to edge length and area. Serial CPU implementations of the face-detection scheme complement the CUDA kernels, which we use to obtain an estimate of the speedup provided. We verify that the number of blocks marked

[2] The Stanford bunny is a well-known benchmark problem in the field of computer graphics. The original model [17] is in PLY format; we used a binary STL model obtained from Wikimedia Commons [18] and transformed it to ASCII STL with the Blender software.



(a) The Stanford bunny. (b) Upside-down view.

Figure. 9: STL model of the Stanford bunny with 112,402 faces.

for refinement with the CPU and GPU codes are identical given the same input to validate the former. The code is compiled in single-precision and the tests are executed on a single NVIDIA GeForce 970M with 3GB of VRAM.

Two individual speedup tests are performed for the circle and the bunny. The circle is instantiated with 12,800 edges in the middle of a computational grid with resolution 256^2 and size $[0, 1]^2$, and the near-wall distance criterion is set equal to $d_{\text{spec.}} = 0.1$. The grid is then refined twice, for a total of three levels in the grid hierarchy. The bunny possesses 112,402 faces and is placed inside a grid with resolution 64^3 and size $[0, 1]^3$ with $d_{\text{spec.}} = 0.05$. Two successive refinements are also used. We consider bin densities $B \in \{1, 2, 8, 16\}$, with $B = 1$ indicating the use of the naive scheme. Figures 12a and 12b illustrate the difference in order of magnitude in the execution times for the face-detection portion of the refinement.

To test the efficiency of the binning algorithm, we use the same setup for the 3D CPU-GPU comparison, and we vary the bin density $B =\in \{1, 2, 4, 6, 7, 8, 9, 10, 12, 14, 16\}$. The variation in execution times for bin setup, face-detection, and the total time is plotted against B in Figure 11. We also test the effects of B on the final number of blocks marked for refinement, and the effects of B_f on the execution time of the binning routine (Figure 10). The near-wall refinement results for the Stanford bunny are displayed in Figure 13.

The GPU code is consistently two orders of magnitude faster than the CPU counterpart for face-detection operations on both grid levels. As B increases, the execution times decrease for both refinements on both devices, reaching $\mathcal{O}(1)$ ms at $B = 16$ on the GPU and $\mathcal{O}(100) - \mathcal{O}(1000)$ on the CPU. The speedup is at a maximum for the naive algorithm, at around 450x in 3D and 160x in 2D. The speedup is less pronounced in 2D, likely due to the relatively smaller workloads. When the execution time for setting up the bins is accounted for, we find that the total time needed for the near-wall refinement is at a minimum when $B = 8$ with a value of approximately 1 second. More time is spent checking adjacency to faces for smaller B, while for larger values the bin setup time quickly overtakes the nearly negligible face-detection time. The total times for $B = 2$ and $B = 16$ are about the same, equal to $\sim 3.5 - 4$ s. We found that B_f did not impact the performance of the bin setup procedure, however, larger values were required when B was increased in order to not run out of memory.

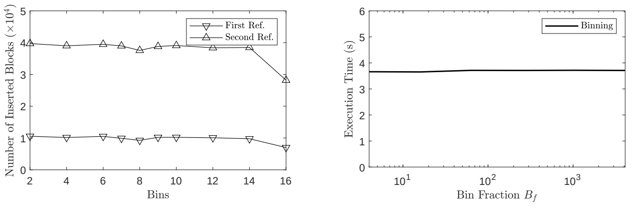


Figure. 10: Left: Inserted blocks vs. bin density. Right: Binning execution time vs. bin fraction.

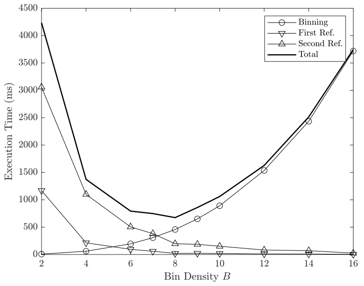
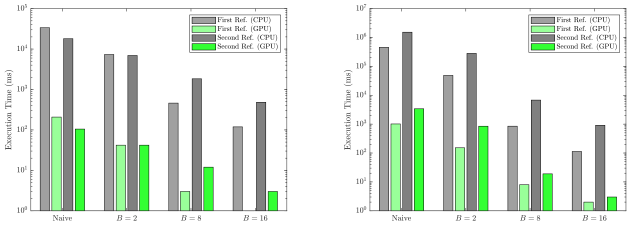


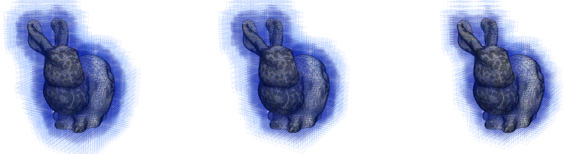
Figure. 11: Execution times of Test 2, plotted against bin density.



(a) 2D. (b) 3D.

Figure. 12: Distribution of execution times for the near-wall refinement routine on the CPU and GPU.

The binning procedure is a relatively expensive step as B increases. This may be attributed to the face discretization procedure that is used to assign faces to the bins. The discretization parameter is a quick way of assigning faces to bins, but experiments show that it can impact the total computational cost if the number is too large. There are other ways of determining if a face crosses a bin (e.g., checking that the edge(s) of the face intersect with the faces of the bin, or if the face is totally enclosed by the bin). We have also only



(a) $B = 2$. (b) $B = 8$. (c) $B = 16$.

Figure. 13: Cell-blocks near the bunny after successive near-wall refinement in the execution time vs. bin density test. As bin density increases, more propagation is needed to satisfy the near-wall refinement distance criterion.

considered static refinement of the grid prior to simulation. It is likely that the additional cost of bin setup with a greater B offsets the cost of increasing the search-loop size in the case of a moving geometry (such as with immersed boundaries).

The naive implementation of near-wall refinement makes it easier to classify a node as a solid, boundary, or fluid node since all faces in the domain are available. Ray-casting can be used to determine the node type, and the near-wall distance can be enforced exactly. The binning procedure makes the process much more local, which means different approaches to node classification are needed (such as checking the position of the node relative to the triangle orientation). This, along with the possible optimizations to the binning procedure, will be explored in future work.

Conclusion

We present an algorithm for incorporating a complex geometry in the form of an edge/triangle mesh within an adaptive Cartesian grid that is organized hierarchically as a forest of octrees. The C++/CUDA implementation of the algorithm extends the AGAL framework for GPU-native adaptive mesh refinement in the form of a new `Geometry` module as a first step toward simulating weakly-compressible fluid flow with the Lattice Boltzmann Method. We define a metric for indicating adjacency of grid cells to the wall, and describe a set of CUDA kernels that enables the cells to identify the indices of the faces (which is a necessary step for establishing links with the boundary domain for application of boundary conditions) and to mark blocks in a specified vicinity of the wall for refinement. We present a naive approach where all cells in the grid check their relative position to all faces in the geometry, and a more efficient approach based on spatial binning which greatly reduces the search-loop size.

A comparison of the execution times of the bin setup and face-detection routines assesses the performance of the implementation. The algorithms are implemented both on the CPU and the GPU to determine the relative speedup provided by the latter. A 2D circle and 3D Stanford bunny serve as test geometries. The former is imported by text file with uniform edge sizes, while the latter possesses a large, fixed number of triangles of varying size that make it a suitable candidate for validation of the binning algorithm. The GPU code executes two orders of magnitude faster than the CPU code, even when the naive face-detection approach is used. It takes $\mathcal{O}(1) - \mathcal{O}(100)$ ms to execute the face-detection procedure on the GPU with spatial binning (with larger numbers of bins corresponding to shorter times), and $\mathcal{O}(1000)$ ms with the naive approach. Bin setup generally becomes more expensive when the number of bins increases. Total time was minimized when 8^3 bins were used, producing a total of ~ 1000 ms to set up the bins.

We are currently preparing a Lattice Boltzmann solver that is capable of utilizing the data structures presented in this paper to efficiently simulate weakly-compressible fluid flow in complex geometries. Flows past a circle and a sphere will serve as benchmark tests for validation.

Acknowledgment

The authors gratefully acknowledge support from the Natural Sciences and Engineering Research Council of Canada, Canadian Microelectronics Corporation and the Digital Research Alliance of Canada.

References

[1] S. Sakane, T. Aoki and T. Takaki, ”Phase-field lattice Boltzmann simulation of three-dimensional settling dendrite with natural convection during nonisothermal solidification of binary alloy,” IOP Conference Series. Materials Science and Engineering, vol. 1281, (1), pp. 12053, 2023.

[2] S. Watanabe and T. Aoki, ”Large-scale flow simulations using lattice Boltzmann method with AMR following free-surface on multiple GPUs,” Computer Physics Communications, vol. 264, pp. 107871, 2021.

[3] A. Suss, I. Mary, T. Le Garrec and S. Marie, ”A hybrid lattice Boltzmann - Navier-Stokes method for unsteady aerodynamic and aeroacoustic computations,” Journal of Computational Physics, vol. 485, pp. 112098-35, 2023.

[4] F. Gendre et al, ”Grid refinement for aeroacoustics in the lattice Boltzmann method: A directional splitting approach,” Physical Review. E, vol. 96, (2-1), pp. 023311-023311, 2017.

[5] A. Schukmann, A. Schneider, V. Haas and M. Böhle, ”Analysis of Hierarchical Grid Refinement Techniques for the Lattice Boltzmann Method by Numerical Experiments,” Fluids (Basel), vol. 8, (3), pp. 103, 2023.

[6] J. Latt, O. Malaspinas, D. Kontaxakis, A. Parmigiani, D. Lagrava, F. Brogi et al., ”Palabos: Parallel Lattice Boltzmann Solver,” Computers & Mathematics with Applications (1987), vol. 81, (1), pp. 334-350, 2021.

[7] M. Bauer, S. Eibl, C. Godenschwager, F. Kohl, M. Kuron, C. Rettinger et al., ”waLBerla: A block-structured high-performance framework for multiphysics simulations,” Computers & Mathematics with Applications, vol. 81, (1), pp. 478-501, 2021.

[8] M. Lahnert, C. Burstedde, C. Holm, M. Mehl, G. Rempfer and F. Weik, ”Towards lattice-boltzmann on dynamically adaptive grids — minimally-invasive grid exchange in espresso,” VII European Congress on Computational Methods in Applied Sciences and Engineering. DOI: 10.7712/100016.1982.4659.

[9] W. Zhang, A. Myers, K. Gott, A. Almgren and J. Bell, ”AMReX: a framework for block-structured adaptive mesh refinement,” Journal of Open Source Software, vol. 4, (37), pp. 1370, 2019.

[10] M. Wahib, N. Maruyama and T. Aoki, ”Daino: A high-level framework for parallel and efficient AMR on GPUs,” in 2016, . DOI: 10.1109/SC.2016.52.

[11] A. Giuliani and L. Krivodonova, ”Adaptive mesh refinement on graphics processing units for applications in gas dynamics,” Journal of Computational Physics, vol. 381, pp. 67-90, 2019.

[12] L. Wang, F. Witherden and A. Jameson, ”An efficient GPU-based h-adaptation framework via linear trees for the flux reconstruction method,” Journal of Computational Physics, vol. 502, pp. 112823, 2024.

[13] P. V. Pavlukhin and I. S. Menshov, ”GPU-native Dynamic Octree-based Grid Adaptation to Moving Bodies,” Lobachevskii Journal of Mathematics, vol. 45, (1), pp. 308-318, 2024.

[14] K. Jaber, E.E. Essel and P.E. Sullivan, GPU-Native Adaptive Mesh Refinement with Application to Lattice Boltzmann Simulations, Computer Physics Communications, 109543, doi: https://doi.org/10.1016/j.cpc.2025.109543. (in press)

[15] E. W. Weisstein, ”Point-Line Distance–3-Dimensional,” MathWorld–A Wolfram Web Resource. [Online]. Available: https://mathworld.wolfram.com/Point-LineDistance3-Dimensional.html

[16] NVIDIA Corporation, ”NVIDIA CUDA Core Compute Libraries,” NVIDIA Developer Documentation, 2024. [Online]. Available: https://docs.nvidia.com/cuda/cuda-toolkit/index.html

[17] G. Turk and M. Levoy, ”The Stanford Bunny,” Stanford Computer Graphics Laboratory, 1994. [Online]. Available: http://graphics.stanford.edu/data/3Dscanrep/

[18] ”Stanford Bunny,” Wikimedia Commons, [Online]. Available: https://commons.wikimedia.org/wiki/File:Stanford_Bunny.stl
(Originally sourced from the Stanford University Computer Graphics Laboratory)