Journal of Computational Physics 444 (2021) 110574

Contents lists available at ScienceDirect

Journal of Computational Physics

www.elsevier.com/locate/jcp

A hardware accelerated unstructured overset method to simulate turbulent fluid flow

Wyatt James Horne, Krishnan Mahesh *

University of Minnesota, Minneapolis 55455, USA

ARTICLE INFO

Article history:
Available online 24 July 2021

Keywords:
GPU acceleration
Unstructured overset method
Real-time raytracing
Incompressible fluid flow
Turbulent flows
DNS/LES

ABSTRACT

Hardware acceleration consists of offloading computational work to devices such as graphics processing units (GPUs) to produce overall speed-up. Algorithms and numerical methods must be constructed to suit the available hardware in order to effectively produce speed-up. In this work a numerical method is presented which can effectively use hardware acceleration to simulate incompressible turbulent fluid flow. The method is an unstructured overset method where unstructured meshes are attached to individual bodies and connected throughout the flow domain to produce a single domain solution through an overset assembly process. The unstructured overset method shown in Horne and Mahesh [1] and Horne and Mahesh [2] was found capable of scaling to $O(10^5)$ computational cores for $O(10^5)$ moving bodies in turbulent flow fields while producing accurate flow results. This highly scalable method is modified and extended to effectively utilize on-node hardware acceleration. Overset assembly algorithms which use hardware acceleration are presented based on successful accelerated algorithms in real-time ray tracing and computational geometry. Timing results for core overset assembly operations are presented showing a maximum O(100x) speedup when using hardware acceleration. A novel method for turbulent fluid flow is presented which utilizes over-decomposition of the flow domain to produce task-parallelism allowing asynchronous calculation of the different steps of the method while also providing overlap between data transfer and computation. A mixed precision solver is utilized which provides a balance between optimal performance and numerical accuracy. A cost effective and accurate artificial compressibility pressure regularization is used which has minimal memory complexity and minimizes computational cost while maintaining accuracy. A primal-dual Laplacian operator is introduced which produces accurate results on skewed meshes. Results for canonical flow cases with overset meshes are shown illustrating the method's accuracy and numerical properties. Substantial speed-up is demonstrated for the numerical method reaching upwards of 50 times as fast as the non-accelerated method for high cell loadings.

1. Introduction

Accurately and quickly simulating rigid moving bodies in turbulent fluid flows is of great interest to many industrial applications, such as rotorcraft, wind energy applications, and marine propulsors. One potential path to accelerate such

* Corresponding author.
E-mail address: kmahesh@umn.edu (K. Mahesh).

https://doi.org/10.1016/j.jcp.2021.110574
0021-9991/

simulations is the use of hardware acceleration where expensive computational tasks are offloaded to devices such as graphics processing units (GPUs) to produce substantive performance gains. Over the past decade hardware acceleration has proven to be a potent tool in fluid simulations capable of producing order-of-magnitude performance gains over pure central processing unit (CPU) calculations [3–7].

Algorithms and methods which are effective on CPU devices are not necessarily effective on a GPU. A GPU contains many computational cores and has overall higher latency when compared to a CPU. Float precision plays an important role on GPUs where peak floating-point-operations-per-second (FLOPS) can range from O(100) TFLOPS (10^9 FLOPS) when using half precision to O(1) TFLOPS when using double precision as is the case for a NVIDIA V100 GPU. Memory access is an important consideration when using a GPU. Regular, strided, memory access can produce orders of magnitude improved computational throughput relative to random memory access. Additional to these considerations, data transfer to the GPU memory is relatively slow and is a common bottleneck in computational performance. Algorithms must be sufficiently parallel, account for the heightened latency, efficiently use float precision, and overlap calculations with GPU data transfer to be performant.

The goal of this work is to produce an unstructured overset numerical method that efficiently utilizes GPUs to simulate incompressible turbulent fluid flows to produce substantive speed-up relative to pure CPU implementations. The method to be presented is based on the work found in Horne and Mahesh [1] and Horne and Mahesh [2] in which an unstructured overset method capable of simulating O(10^5) moving bodies in turbulent fluid flow on O(10^5) computational cores was presented. Strong-scaling was demonstrated in this work up to 492,000 computational cores for simulations of 100,000 moving spheres in a turbulent flow field. This highly scalable method is modified and extended in this work to use on-node hardware acceleration.

In the method, unstructured overset meshes are attached to each individual moving body. The arbitrary, overlapping meshes are connected through an overset assembly process producing a single cohesive flow solution over the domain. For arbitrary motion it is automatic and dynamic with time as the bodies move throughout the domain. For the motion of the fluid a kinetic energy preserving, finite volume method is utilized with a supercell interpolant for flow reconstructions between overlapping meshes. The 6 degrees-of-freedom (6-DOF) motion of the rigid bodies is found by directly integrating fluid and forces and torques on the surfaces of bodies using Newton's and Euler's motion laws. To effectively accelerate the method, GPU algorithms must be produced for both the overset assembly process and finite volume numerical method.

Utilizing GPUs to accelerate overset assembly is not a new idea. Chandar et al. [8] produced overset assembly algorithms capable of providing a peak 60x speed-up relative to a serial CPU algorithm for donor searching. The work used a gradient search approach which locates donor cells by marching across all potential donor cells via cell-face couplings. Potential donor cells were found by a coarse axis-aligned bounding box (AABB) overlap calculation. Cutting was conducted implicitly by finding potential donors for every cell on a given overset mesh. Cells which overlap solid boundaries according to AABB overlap and could not find any suitable donors were removed from the calculation. The work did not consider mesh motion while performing GPU accelerated assembly. Only a single GPU and single processor was considered and multiple GPUs per node with multiple nodes, as is common on modern computing hardware, was considered a future extension. Additionally, the assembly was only demonstrated for a single overset mesh and a relatively limited number of cells.

More recently, Crabill et al. [9] presented an overset assembly method for curved meshes which utilizes GPUs. A parallel direct cutting algorithm was shown which has each cell calculate a signed distance to nearby boundaries to determine which cells lie within solid bodies. Donor searches were performed using an alternating digital tree structure (ADT) [10]. In the work, ADT were constructed on CPUs and then transferred to the GPUs. Searches within the ADT were done using a stack system to avoid recursion. Rigid body motion was assumed such that the tree only had to be constructed once and then updated through the use of translation and rotation matrices. A substantive 16x speedup was demonstrated for the assembly compared to a pure CPU overset assembly calculation for a moving overset case using K80 GPUs compared to 4 computational cores on a modern Intel processor.

Numerical methods to simulate incompressible flow have been developed that can effectively utilize GPUs [3–7,11]. Artificial compressibility (AC) methods are an example that have grown in recent popularity [12–18]. These methods introduce the time derivative of pressure to the mass equation to regularize the incompressible fluid equation system. They have the advantage of not requiring the solution of an expensive Poisson equation while meeting or exceeding the time accuracy of other methods. AC methods achieve this while producing linear systems with lower condition numbers which are more amenable to lower precision or mixed precision solutions and allow practical calculation using relatively simple linear solvers [14,15].

Previous work has applied AC methods to overset calculations, e.g. Tang et al. [19]. Generally expensive dual time stepping strategies have been employed and hardware acceleration has not been considered. In this work the predictor-corrector overset method presented in Horne and Mahesh [2] is modified by the addition of the material derivative of pressure to the pressure equation. The pressure and velocity are solved at the same time level each time step removing the splitting time error that is common to predictor-corrector methods. The method retains the time order of accuracy of the previous method while also producing linear systems with lower condition numbers. A mixed precision solver is used for both velocity and pressure to effectively use GPUs while retaining adequate numerical precision. Over-decomposition is used to introduce additional concurrency over multiple GPUs where velocity and pressure iterations can be conducted simultaneously with GPU data transfer, halo data exchanges, and overset boundary reconstructions. A primal-dual Laplacian operator is used in place of the two-point-flux-approximation (TPFA) in the original work to produce more accurate flow fields on skewed meshes.

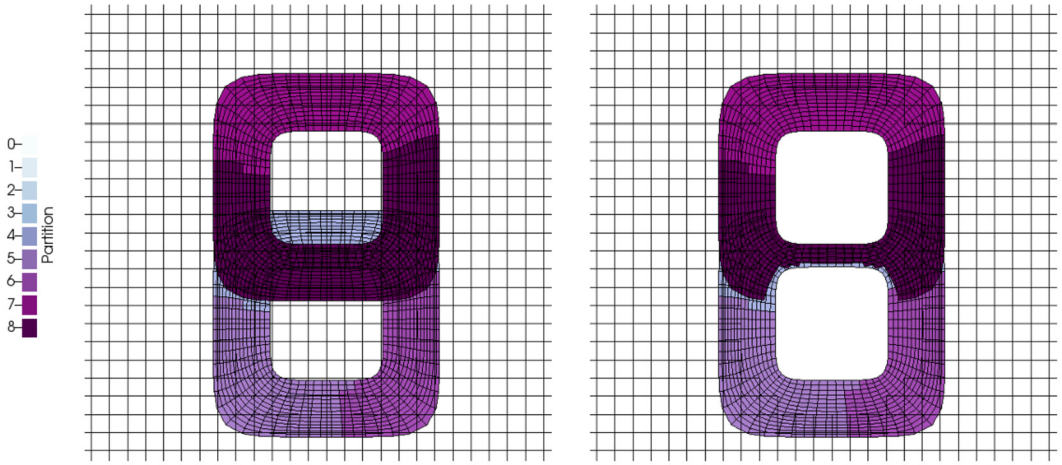

Fig. 1. Example overset meshes colored by partition numbering before and after overset assembly respectively.

GPU accelerated overset assembly is performed using modifications to modern algorithms found in realtime raytracing and collision detection. These algorithms were designed from their core to be heavily parallel and performant on GPUs. Linear bounding volume hierarchies (LBVH) are extensively used for geometry searches and to simultaneously determine coarse geometry overlap across multiple GPUs [20,21]. This binary tree structure avoids the issues of point search structures, such as ADT, by robustly finding all volumes that can potentially intersect a given object through the use of bounding volumes. It is quickly constructed on a GPU in parallel using a space-filling curve based on Morton codes. Searches of the LBVH are performed using a stack strategy that avoids recursion while benefiting from the LBVH structure. To determine fine geometric overlap, the widely successful Gilbert-Johnson-Keerthi (GJK) algorithm is used [22,23]. This algorithm uses the concept of a Minkowski distance to exactly determine if two convex, arbitrary polyhedron overlap. Since most of all geometry calculations involve convex polyhedrons in this work, this is a highly robust calculation to the limits of the float precision used.

Sec. 2 shows details of the accelerated overset assembly. Sec. 3 gives details of the numerical method. Illustrative results are shown in each section to demonstrate different features of the various algorithms and method. Validation and peformance of the method is shown in Sec. 4. Conclusions of the work are then drawn in Sec. 5.

2. Accelerated overset assembly

Fig. 1 depicts an example overset case of two objects before and after the overset assembly procedure. A partitioned mesh is attached to each object and to a background flow domain. To perform overset assembly in parallel, overlapping mesh partitions throughout the domain must establish communication patterns to exchange geometry and flow information. Computational cells that lie outside the flow domain, such as cells within the objects or the channel walls, must be removed from the simulation. It is also generally desirable to reduce mesh overlap as much as possible while still maintaining complete coverage of the flow domain to reduce computational cost. Both of these tasks are achieved by setting a masking variable which turns on or off the flow solution throughout the flow domain through a process commonly referred to as “cutting” [24]. Cells which border masked cells or which lie on the outer edges of overset meshes must be provided boundary conditions through flow reconstruction from overlapping meshes. This requires a detailed geometrical cell partner search across all overlapping mesh partitions.

To develop a hardware accelerated overset assembly method we begin with the work shown in Horne and Mahesh [1] which outlines a series of algorithms that perform the full overset assembly process in parallel. In the work multiple unstructured mesh partitions are assigned to available computational cores. A Cartesian processor mesh is used to determine communication patterns between the cores. As depicted in Fig. 2, explicit volumes, such as spheres, object-oriented boxes (OOB) and more, are used to cut redundant cells. Fine detail cutting is done on any remaining cells that intersect solid boundaries using K-Dimensional tree (K-D tree) searches and signed distance calculations. A ‘Forest-Fire’ flood-fill algorithm is then used to cut any cells which lie deep within solid boundaries. The reader is invited to read the work for more details on any of these assembly steps.

To introduce hardware acceleration into an established code, algorithms amenable to acceleration must first be identified. In the timings reported in the work the two most expensive operations were shown to be the cutting and reconstruction partner pairing algorithms. The communication pattern operations mainly consisted of Message-Passing-Interface (MPI) related algorithms with little opportunity to introduce hardware acceleration. Due to both of these considerations the cutting and reconstruction partner pairing operations are selected to be accelerated.

At their core both of these assembly steps consist of the same operation: given an object, all overlapping objects in a given group of objects must be found. For instance, in the cutting process one must find all of the cells that overlap with

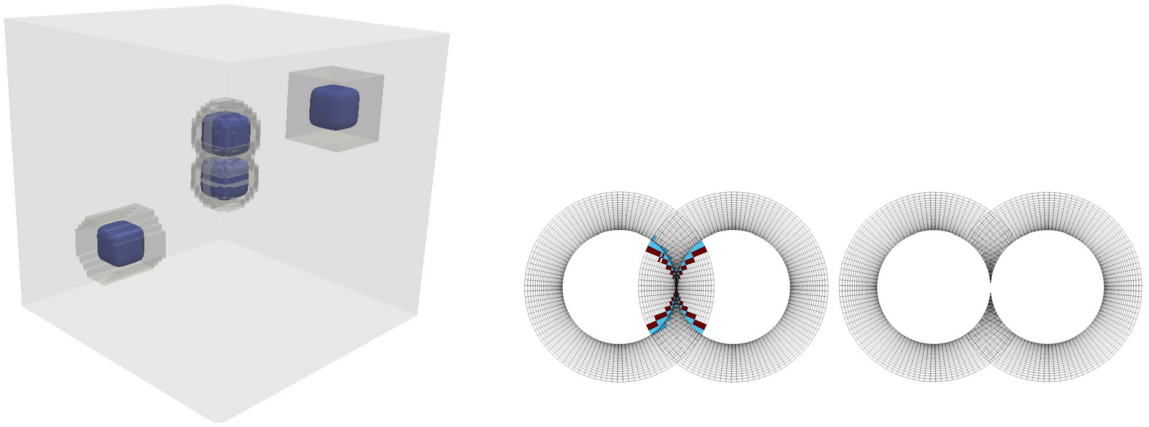

Fig. 2. Cutting examples depicting explicit cutting volumes and implicit cutting using flood-fill respectively. Explicit cutting volumes are shown as shaded volumes and include sphere, cylinder, and OOB cuts around the objects shown in dark blue (■). Fine detail cutting is then depicted for two touching spheres taken from Horne and Mahesh [1]. Cells which are found to overlap are depicted in blue (■) and flood-fill starting cells are shown in red (■). The last figure shows the final result of the fine detail cut. (For interpretation of the colors in the figure(s), the reader is referred to the web version of this article.)

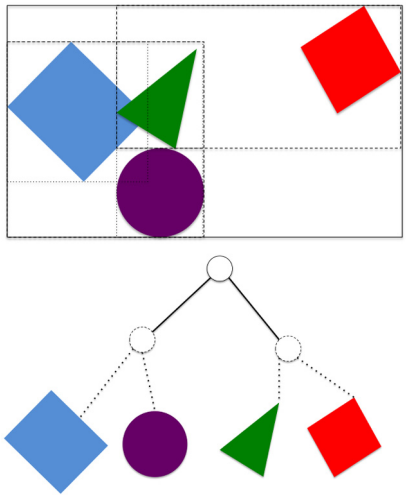

Fig. 3. BVH example for four objects. Note the correspondence between AABB line styling shown in the top figure with the nodes shown in the bottom figure. To determine overlap a calculation checks AABB overlap starting at the top of the tree and only continues down a branch if the corresponding AABB overlaps. This is done until all leaf nodes containing objects which overlap are found.

a solid boundary or a explicit volume. For cell reconstruction one must find all overlapping cells to a given reconstruction cell or face. Performant GPU accelerated algorithms to perform this operation are commonly found in real-time ray tracing and collision detection [21,23,25].

In real-time ray tracing one seeks to find all triangles along the surfaces of objects that intersect the path of a given ray. This calculation must be fast enough that the scene can be manipulated in real time. It is not computationally feasible to achieve this by performing an expensive and detailed overlap calculation for each triangle and ray as this would involve $O(N_{ray}N_{tri})$ expensive calculations where both the number of rays, N_{ray}, and the number of triangles, N_{tri}, are large. Instead algorithms extensively use binary tree structures such as bounding volume hierarchies (BVH) to yield $O(log(N))$ cheap operations for each ray with significantly less expensive detailed calculations.

BVH are binary tree structures that use simple bounding volumes of different sizes to determine the splits in the tree as shown in Fig. 3. At the top of the tree the bounding volume encompasses the entire collection of objects. At the bottom of the tree the volumes encompass individual objects commonly referred to as leaf nodes. To determine overlap tree traversal starts at the top. At each branching node a relatively cheap intersection calculation is performed with each relevant bounding volume. If intersection is found with a branch's bounding volume the algorithm will continue to traverse the branch. Detailed overlap calculations are only performed at the bottom of tree with objects attached to leaf nodes.

For real-time ray tracing BVH construction must be done on a GPU efficiently as objects move in a scene. One algorithm to achieve this was shown in Karras [21]. To construct a BVH on a GPU, a bottom up approach is used where the leaves of

```
class object {
 vec3* vertices;
}

class bvh {
 bvhNode* internal_nodes;
 bvhNode* leaf_nodes;
 object* objects;
 size_t Nobjs;
}

class bvhNode {
 size_t minId;
 size_t maxId;
 size_t objId;
 bool isLeaf;
 bvhNode* left;
 bvhNode* right;
 AABB box;
}
```

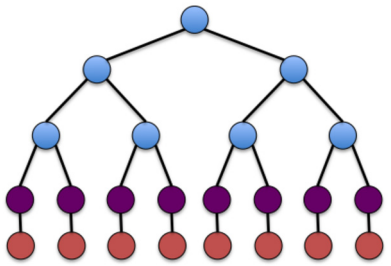

Fig. 4. BVH data structure psuedo code and corresponding structure. As shown, the main data structure consists of 3 arrays: internal BVH nodes, leaf BVH nodes, and objects.

the tree are first defined and the branches are determined after. To determine the ordering of the leaves at the bottom of the tree, a space-filling curve based on Morton codes is used. This ensures that objects that share branches in the constructed tree are close in space such that the BVH is well balanced and efficient. All of these operations are done in parallel on a GPU yielding high performance relative to a pure CPU equivalent. Additionally, tree traversal is done for many searches in parallel and optimized to minimize thread divergence and other potential performance losses.

This algorithm is utilized to construct a BVH for cutting and reconstruction partner pairing. The data structure for the BVH is depicted in Fig. 4. In each BVH there are three arrays: an array of internal BVH nodes, an array of leaf BVH nodes, and an array of objects. The object class shown in the figure contains vertices for a given object. The node class in the figure is used to store the branch splits in the tree. Each node contains pointers to a left and right node which could either be another branch or a leaf. AABB over the left and right entries are included in the node class for tree traversal. The leaves of the tree consist of an array of nodes that each point to an object. Since the tree is a binary tree, exactly $2N_{obj} - 1$ nodes are required, where N_{obj} is the number of objects in the BVH. The main task to construct the BVH is to correctly arrange the node and object arrays, and to ensure the left and right pointers at each node are defined.

To begin construction, the data structure must be allocated and relevant geometry must be transferred to GPUs. In general, memory transfer and dynamic memory allocation/deallocation is expensive on GPUs. To avoid unnecessary dynamic memory allocation, two empty tree structures are allocated at the beginning of each simulation on each GPU within a node and only expanded when necessary during a calculation. One BVH is set to contain the cells in the mesh partition, which will be largely static during the simulation, and the other is for dynamic BVH evaluations. The initial size of both is assumed to be $N_{obj} = N_{cell}/N_{gpu}$ where N_{cell} is the number of cells on a partition and N_{gpu} is the number of GPUs on the node. To utilize multiple GPUs on a node a given mesh partition is re-partitioned N_{gpu} times. Since most of the geometrical overlap calculations will be done over all cells in a mesh partition this is a reasonable estimate as to the final size of the tree.

Geometry is passed to the allocated structures as a polyhedron soup consisting of a list of vertices and connectivity lists which outline how the vertices are connected to the various shapes. All GPU related data transfer in this work is chosen to be done asynchronously such that GPU and CPU can perform computation at the same time. After data has transferred, the connectivity lists and vertices are used to fill the allocated object array in parallel on each GPU where one GPU thread is used for each object.

To arrange the leaves of the tree, a space filling Morton curve is constructed as depicted for the example in Fig. 5. To construct the curve a Morton code must first be calculated for each object. A Morton code is calculated by interweaving the bits from the coordinates of each shape's centroid resulting in a single integer value for each object. If the objects are arranged by increasing Morton code the result is a space filling curve where objects close in the array are close in space. Morton codes are calculated in parallel on GPUs for each object. To allow lower float precision, scaling is used where each coordinate is first normalized by the size of a mesh's partition. The objects are then sorted according to their Morton codes using a Radix sort, which has been shown to be performant on GPUs [26]. Rigid body motion is assumed for the cells within each mesh partition such that the Morton curve does not need to be updated with time. This means that past the initialization of a simulation, only the object vertices and AABB need to be updated. The BVH node connections and leaf arrangements remain static with time allowing for significant time savings.

Once the BVH leaves are arranged according to the Morton curve, the internal branches of the BVH can be considered to contain a linear range over sections of the arranged objects as shown in Fig. 5. For instance the head branch contains the entire range of objects. The range is split and the two resulting ranges of objects constitute the left and right branches attached to the head branch. This process repeats until the range is 1 object constituting a leaf node in the BVH. To determine where the splits occur at each branch a binary search is conducted using the algorithms shown in Karras [21]

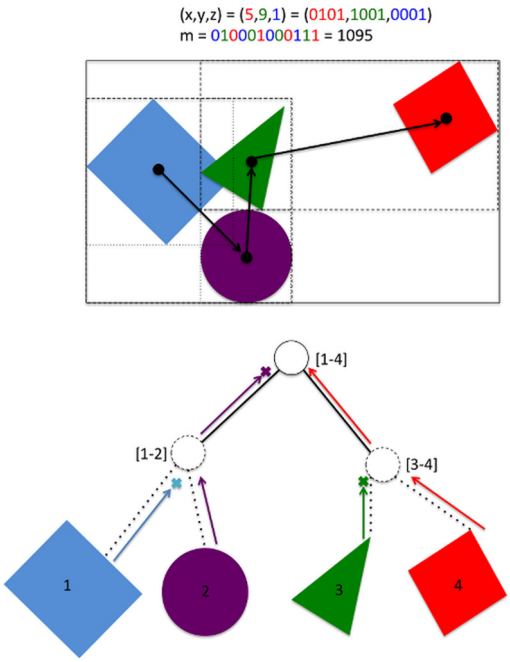

Fig. 5. Morton curve construction and connectivity algorithm for example BVH. Morton codes are calculated using bit weaving for each object's centroid, as depicted for the Morton code m using coordinate (x, y, z). Each object is sorted according to the resulting Morton code giving the curve depicted with arrays in the top figure. To build connections, threads, indicated here as arrows in the bottom figure, start at bottom and only continue up tree if given thread is second thread to reach node. Ranges of object indices are shown next to each node showing the resulting index range of the binary search algorithm.

and Garanzha et al. [27]. The search consists of finding a split such that the highest differing bit of the Morton codes of one branch will be 0 and 1 in the other branch. This corresponds to splitting the objects by an axis aligned plane in 3D. The partitioning and internal connection of the BVH is done in parallel where GPU threads are assigned to each level. To finalize building the BVH a bottom up algorithm is used starting at the leaves of the BVH. Threads move up the tree filling in the AABB and connections at each node. Atomic operations are used to ensure that only the second thread that arrives at each node fills the details of the node to avoid the possibility of a data race condition.

To utilize the tree and search for overlap, a stack searching strategy is used to avoid recursion. Each object is assigned a GPU thread for the search. Each GPU thread has an initially empty stack of pointers to BVH node objects and an empty array that will be filled with the indices of overlapping objects in the BVH. To begin the search, a pointer to the head branch of the BVH is first added to the stack. Then, each thread loops through all entries in its stack checking for AABB overlap between the object and the nodes. New stack entries are dynamically added to the end of the stack from the branches at each node. Each time a leaf node is encountered, the corresponding object index in the BVH is added to the thread's object index array. Each thread exits the search when their search stack is empty. The found overlapping object indices are placed within a combined array. This combined array indicates the locations of objects in the BVH which have AABB that overlaps each object involved in the search.

AABB overlap is not a good indicator of general object overlap for general cells within unstructured meshes. A finer detail overlap calculation is necessary to ensure that the cutting and reconstruction operations are robust and accurate. The GJK algorithm is a performant option that has successfully been used for this purpose in real-time collision detection [23,22]. The core of the algorithm is built around the concept of a Minkowski difference as depicted in Fig. 6. Given two sets of vertices from two polyhedrons, a Minkowski difference is the set of vertices formed by subtracting each member of one set of vertices by each member of the other set. A new shape can be formed as the convect hull over the new set of vertices. If the two original polyhedrons are convex and the coordinate (0,0,0) lies within the constructed shape, the two polyhedrons are guaranteed to overlap.

One could directly calculate the Minkowski difference between each object pairing in an overlap calculation but it would be an expensive $O(N_1 N_2)$ calculation where N_i is the number of vertices in the ith object. Instead the GJK algorithm consists of dynamically constructing a polytope within the Minkowski difference which attempts to encompass the coordinate (0,0,0) at each iteration as depicted in Fig. 6. The algorithm begins by selecting one point within the Minkowski difference. New points are added each iteration creating a line, then a triangle, and finally a tetrahedron. All points are selected using the support of the Minkowski difference in opposing directions to the previously selected point. If the polytope is ever found to contain (0,0,0) the algorithm returns that the objects overlap. If at any point it is determined that (0,0,0) cannot be contained by the polytope the iterations cease and the algorithm returns that the objects do not overlap.

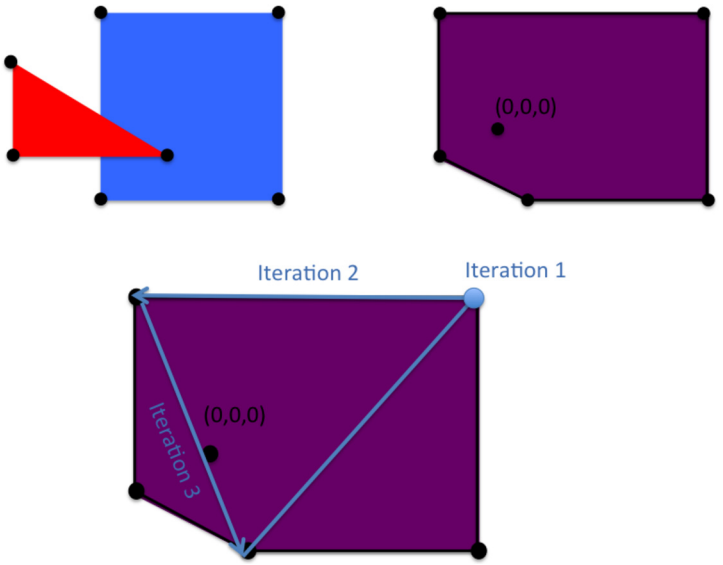

Fig. 6. Minkowski difference and GJK algorithm for two example shapes. The resulting Minkowski difference is shown to the right of the two shapes. The iterations of the GJK algorithm are then shown where first a point is created, then a line and then finally a triangle which contains the (0,0,0) coordinate.

In this work detailed overlap is only calculated using computational cells and polygonal faces of which the majority are convex such that the GJK algorithm is accurate. In its application to this work, the algorithm is found to need 3-5 iterations to determine overlap for the majority of overlap calculations such that it is highly efficient. Resulting object pairs from a BVH search are each assigned a GPU thread which performs a GJK calculation returning a boolean of object overlap. An array of booleans is returned from the algorithm for each pairing. The array of booleans and object pair indices are then returned from each GPU on a node to the CPU host during overset assembly.

To outline the accelerated overset assembly process: An arbitrary number of unstructured mesh partitions are assigned to computational cores and communication patterns are determined using the same algorithms shown in Horne and Mahesh [1]. Each partition is over-partitioned to yield N_{gpu} partitions for each original partition, where N_{gpu} is the number of GPUs on a node. The smaller partitions are assigned cyclically to GPUs on each node and BVHs are constructed or updated as described before. Cells within explicit volumes are then strategically removed, as depicted in the work, except that now the explicit volumes are passed to GPUs and one GPU thread per cell is used to determine if the cell overlaps the given volume. Fine detail cutting is then conducted on the remaining cells that exactly overlap the faces of boundaries as found using the outlined BVH and the GJK algorithms. Cells inside boundaries which neighbor overlapping cells from this calculation are then marked to initialize a flood-fill algorithm. The flood-fill calculation is performed, removing all cells deeper within boundaries. Finally reconstruction partner pairings are found using the BVH and GJK algorithms along the edges of overset meshes and cut cell regions.

3. Numerical method

The motion of the fluid in this work is modeled using the incompressible Navier-Stokes equations. Mesh movement is incorporated using an Arbitrary Lagrangian-Eulerian (ALE) formulation given as

$$\begin{aligned} \int_{\Omega} \frac{\partial u_i}{\partial t} dV + \int_{\partial\Omega} (u_i(u^n - u^n_{mesh}) dA) &= -\int_{\Omega} \frac{1}{\rho} \frac{\partial p}{\partial x_i} dV + \int_{\partial\Omega} \nu \frac{\partial u_i}{\partial n} dA, \\ \int_{\partial\Omega} u^n dA &= 0, \end{aligned} \tag{1}$$

where the equations are integrated over a volume Ω, with faces $\partial\Omega$ which have normals n. u_i is the fluid velocity in the ith Cartesian direction, u^n is the fluid velocity normal to the faces of the volume, u^n_{mesh} is the velocity of the mesh normal to the faces of the control volume, p is the pressure, t is time, and ρ is the fluid density.

In Horne and Mahesh [2] Eqn. (1) was solved using a kinetic energy preserving, collocated, cell-centered, finite volume method with a pressure projection time stepping scheme. It was demonstrated to provide accurate results ranging from canonical flow cases to freely moving bodies in a turbulent fluid flow. The goal of this section is to modify this method to effectively use GPUs as well as provide some additional benefits to the original scheme.

3.1. Pressure formulation

Projection time stepping schemes introduce a computationally expensive pressure Poisson equation which upon discretization produces matrices that have condition numbers of $O(1/\delta x^2)$. The generally high condition numbers of these matrices greatly limits the potential use of low precision or mixed precision solvers which will perform well on GPUs. Heavy synchronization of the GPU and CPU is required to solve this system where velocity must be solved followed by pressure with synchronizations every linear solver iteration for each. This is wasteful since it would cause CPUs to consistently wait for GPUs to process data to perform halo updates and then GPUs would wait for halo transfers and so on. Ideally CPUs and GPUs would do useful work at the same time requiring a more asynchronous method.

To modify the projection method we begin by rewriting Eqn. (1) in matrix form yielding

$$\begin{bmatrix} a & g \\ d & 0 \end{bmatrix} \begin{bmatrix} u \\ p \end{bmatrix} = \begin{bmatrix} f \\ 0 \end{bmatrix} \tag{2}$$

where a is the Navier-Stokes operator, g is the pressure gradient operator, d is the velocity divergence operator, and f is any external force applied to the fluid. To efficiently solve this system using iterative methods a regularization is generally introduced by the inclusion of a pressure equation in the diagonal of the continuity equation. For the instance of projection methods the Laplacian of pressure, $\alpha \triangle p$ is introduced. In artificial compressibility methods the time derivative of pressure is introduced, $\epsilon \frac{\partial p}{\partial t}$. Different regularizations introduce different errors into the system. For the case of projection methods, a well documented artificial boundary condition on pressure is introduced, which negatively affects the accuracy of pressure [28]. For artificial compressibility methods, an artificial pressure wave speed is introduced which can produce erroneous compressible flow features inhibiting accuracy. A combination of different regularizations could potentially be used to provide a balance between introduced error and overall computational cost. This strategy is used in this work.

We start from a standard Crank-Nicolson projection scheme given as

$$\begin{aligned} \frac{u^* - u^n}{\delta t} + 1/2C^* - 1/2V^* &= -1/2C^n + 1/2V^n \\ \nabla \cdot u^* &= \delta t \triangle p^{n+1} \\ \frac{u^{n+1} - u^*}{\delta} t &= -\nabla p^{n+1} \end{aligned} \tag{3}$$

where u^* is the predicted velocity field, u^n is the velocity field at the nth time level, C is the discretized convective term, V is the discretized viscous term, ∇p is the discretized pressure gradient, and δt is the fixed time step.

Adding the first and third equation results in

$$\begin{aligned} \frac{u^{n+1} - u^n}{\delta t} + 1/2C^* - 1/2V^* &= -1/2C^n + 1/2V^n - \nabla p^{n+1} \\ \nabla \cdot u^* &= \delta t \triangle (p^{n+1}) \end{aligned} \tag{4}$$

To modify this system the material time derivative of pressure is introduced into the second equation and all predicted terms are taken at the $n+1$ time level yielding

$$\begin{aligned} \frac{u^{n+1} - u^n}{\delta t} + 1/2C^{n+1} - 1/2V^{n+1} &= -1/2C^n + 1/2V^n - \nabla p^{n+1} \\ \epsilon \frac{dp}{dt} + \nabla \cdot u^{n+1} &= \delta t \triangle (p^{n+1}) \end{aligned} \tag{5}$$

where ϵ is related to the artificial pressure wave speed yet to be defined. This removes the splitting error, but introduces a new error through the addition of the material derivative of pressure. The inclusion of the pressure Laplacian is important for a collocated finite volume method to remove odd-even decoupling in pressure and aids in the erroneous compressible waves that may be generated by this system. In general we seek to ensure that $\epsilon \frac{dp}{dt}$ is small and convergent while producing a linear system that can be efficiently solved using GPUs. This is achieved by finding a suitable definition of ϵ.

Past work has evaluated suitable definitions of ϵ [14,13]. Generally the smaller ϵ is, the more incompressible the flow field is at the cost of increased computational cost. In Shen [29] the following related formulation was analyzed

$$\begin{aligned} \frac{u^{n+1} - u^n}{\delta t} + C^{n+1} - V^{n+1} &= -\nabla p^{n+1} \\ \epsilon \frac{p^{n+1} - p^n}{\delta t} + \nabla \cdot u^{n+1} &= 0 \end{aligned} \tag{6}$$

where it was mathematically demonstrated to yield $O(\delta t)$ time accuracy for all flow quantities if $\epsilon = \delta t$. It has also been noted that this system produces a linear system with condition numbers that are $O(\delta t/\delta x^2)$, or $O(v_{scale}^{-1} \delta x^{-1})$ if δt is taken

within the Courant-Friedrichs-Lewy (CFL) condition regime where v_{scale} is a relevant CFL velocity scale [14]. These condition numbers are substantially smaller than those produced by a projection method making the system much more amenable to lower precision and mixed precision iterative solvers. Due to these findings ϵ is set to be $\epsilon = \delta t$.

Placing $\epsilon = \delta t$ into the mass equation from Eqn. (5) yields

$$O(\delta t) + \nabla \cdot u^{n+1} = \delta t \triangle (p^{n+1}) \tag{7}$$

since $p^{n+1} - p^n = O(\delta t)$. This introduces an additional O(δt) error to the pressure equation over the standard projection method with the benefit of removing the splitting error. The standard projection method shown in Eqn. (3) has been shown to have at best O($\delta t^{1/2}$) accuracy for pressure for general meshes, largely due to the application of the boundary conditions $\frac{\partial p}{\partial n} = 0$ [28]. The proposed method alleviates this issue by instead taking

$$-\frac{\partial p^{n+1}}{\partial n} = n \cdot \left(\frac{u^{n+1} - u^n}{\delta t} + C^{n+1} - V^{n+1} \right). \tag{8}$$

With this change, the removal of the splitting error, and the addition of the O(δt) term in the pressure equation, the overall accuracy is anticipated to be comparable to what is found in an incremental projection pressure scheme, like the one used in the original overset method. It also produces linear systems which can be solved on GPUs more effectively due to its lower condition number linear system and potential for asynchronous velocity and pressure iterations.

3.2. Discretization & reconstruction

In Horne and Mahesh [2] a collocated, cell-centered, finite volume discretization was used which emphasizes kinetic energy conservation. A supercell interpolant was used for reconstructions along overset boundaries and a penalty method was introduced to enforce pressure continuity across overlapping meshes. The discretization shown in the work is modified here to solve the equation system presented in the previous section on overlapping overset meshes and to provide additional accuracy for general unstructured meshes.

In the original work the convective, viscous, and pressure Laplacian terms were discretized as

$$\begin{aligned} C &= \sum_{f=1}^{n_f} \frac{u_{cv} + u_{nbr}}{2} (u_n - u_f) A_f \\ V &= \sum_{f=1}^{n_f} \nu \frac{u_{nbr} - u_{cv}}{d_f} A_f \\ \int \triangle p dV &= \sum_{f=1}^{n_f} \frac{p_{nbr} - p_{cv}}{d_f} A_f \end{aligned} \tag{9}$$

where u_{cv} is the velocity for a computational control volume (cv), u_{nbr} is a neighboring cv's velocity across a face connection, p_{cv}/p_{nbr} is the corresponding cv pressure, u_n is the convective face velocity, A_f is the face area, u_f is the mesh velocity at the face, and d_f is the normal distance between the cv and its neighbor.

The convective term was selected to be a central average for its benefits to kinetic energy. This same convective operator is used here. The viscous term and pressure Laplacian terms depicted in Eq. (9) use a two point flux approximation (TFPA) to estimate the normal gradient between cvs for the Laplacian operator. While being functionally simple the TFPA can introduce significant errors into Laplacian solutions for general skewed unstructured meshes. In order to be accurate, a computational mesh must satisfy a K-orthogonality condition where the normal vector of a face between two control volumes must closely align with a vector between the centroids of the neighboring control volumes. In this work a primal-dual discretization is used to augment the TFPA in all Laplacian operators to avert this error.

Fig. 7 depicts an example computational stencil using TFPA and the proposed primal-dual discretization. Flow values are stored in primal volumes as was done before. Dual volumes are constructed around each node where each cv attached to a given node constitutes a new node of the dual volume. To reconstruct values and gradients at faces a least squares reconstruction is performed at each node of the face using its corresponding dual. To evaluate a flux across the face, the reconstructed values and gradients at the nodes are integrated assuming a linear basis function across the face. For a 2nd order approximation this corresponds to taking an average of each reconstructed dual volume's value and gradient at the face's centroid. Note that it is not necessary to store values at the nodes. This procedure is directly converted into reconstruction weights on primal values to get reconstruction values at each face.

In Eqn. (9) the TFPA could potentially be directly replaced with a normal gradient calculated entirely using the dual reconstruction, removing the K-orthogonality requirement. Unfortunately, the resulting linear system would be non-symmetric, not diagonally dominant, and possibly non-positive definite resulting in poor convergence using traditional iterative techniques unless sophisticated, and likely expensive, preconditioners are employed. Instead the TFPA is corrected using

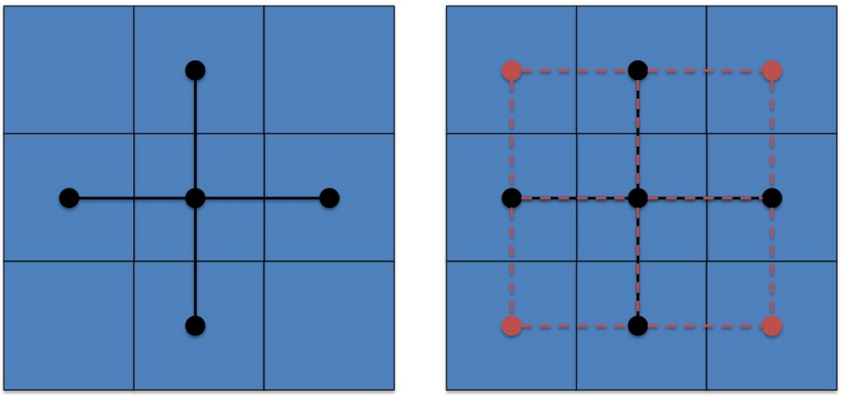

Fig. 7. Computational stencil for the original TFPA and proposed primal-dual discretization respectively. Dual volumes (- -) are constructed around each node of a given control volume to improve the accuracy of Laplacian operators.

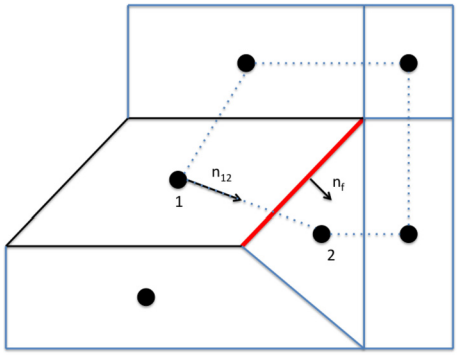

Fig. 8. Example unstructured control volume pairing depicting correction vectors.

$$\frac{\partial p}{\partial n} = \frac{p_2 - p_1}{d_f} + \frac{\partial p_{pd}}{\partial x} \cdot (n_f - n_{12}) \tag{10}$$

where $p_{1,2}$ are flow values in two neighboring control volumes, n_f is the face normal between the control volumes, $\frac{\partial p_{pd}}{\partial n}$ is the dual reconstructed normal gradient, and $n_{1,2}$ is a normal vector between the two control volumes as depicted for an example in Fig. 8. Using this correction the TFPA is exactly recovered for meshes without any skew. As more and more skew is introduced the primal-dual reconstruction is increasingly produced.

The pressure gradient was calculated in the original method using a least squares reduction procedure to minimize $\sum_f (\nabla p \cdot n_f - \frac{\partial p}{\partial n})^2$. This has been shown in previous work to produce good kinetic energy preservation properties [30]. To maintain this same property, the same least squares calculation is used in this work by substituting the normal gradient given by Eqn. (10) directly into the least squares reduction.

To reconstruct flow values between overset meshes the supercell interpolant introduced in the original method is used here. Flow reconstructions are chosen to be directly injected into the linear equations for all flow values. The original method used a penalty method to enforce pressure continuity between overset meshes. This was shown to produce accurate pressure fields while producing a linear system that was symmetric and positive definite which is much more readily solvable using traditional iterative solvers than injecting the reconstruction into the pressure equation.

While the linear system itself is straightforward to solve, the construction of the penalty weights in the linear system is non-trivial requiring each control volume in a given overset reconstruction stencil to receive reconstruction weights from every other control volume in the stencil. Additionally, care must be taken in the construction of meshes and in the selection of the penalty weight to ensure that the divergence of the velocity field is well behaved near overset boundaries. The introduction of $\epsilon \frac{dp}{dt}$ into the continuity equation alleviates this issue by producing a simpler linear system when using injected overset reconstructions that can be solved using iterative solvers without any modification. Since no penalty is introduced into the pressure equation, no additional errors are introduced to the velocity divergence near overset boundaries.

3.3. GPU implementation

Memory bandwidth is a common performance bottleneck when offloading computational work to GPUs. Generally it is desirable to maximize the arithmetic intensity defined as the number of FLOPs per Bytes of memory access. To maximize

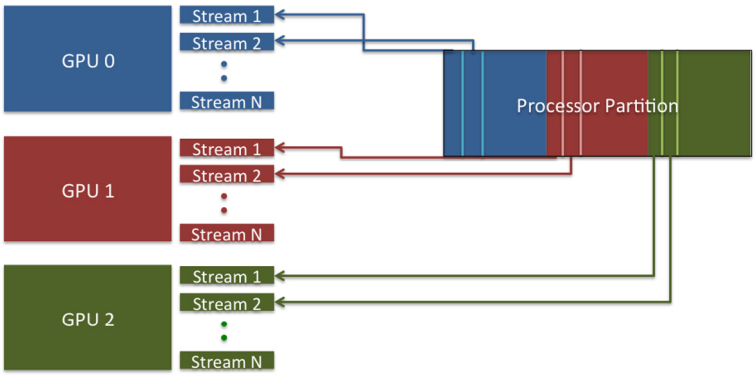

Fig. 9. Over-decomposition partitions and GPU assignment for example processor partition.

this quantity, memory transfer between the CPU and GPU and GPU memory access should be minimized. To achieve the latter, memory re-use and cache memory should be extensively used. When memory transfers are required, such as is the case for a distributed calculation over multiple nodes, they should be overlapped with useful work to provide additional concurrency.

Based on these considerations an over-decomposition strategy is used where each processor's partition is partitioned once more as depicted in Fig. 9. The number of new partitions is allowed to match or exceed the number of GPUs on a node. The work of each new partition is assigned to a work queue that works asynchronously to all other partitions. Each work queue is assigned to available GPUs on a node hierarchically where the partition is first decomposed for each GPU node and then again if further concurrency is desired. The CPU performs overset reconstructions and halo updates asynchronously to the various work queues. This strategy greatly enhances concurrency by allowing many fine-grained data transfers and calculations to occur simultaneously such that neither the host or device wait for large data transfers while performing calculations. The only synchronization point within a given iteration is a single reduction of the L2 norm of the residual to ensure convergence to a specified tolerance.

The decomposed partition data, including geometry and flow solution, is allocated on and transferred to each GPU at the beginning of a simulation. To minimize memory transfer only halo information for the partition is transferred between the CPU and GPU during a calculation. The full transfer of the flow solution to the main memory is only done for post-processing. This was found in timing experiments to be critical towards minimizing data transfer to yield high performance.

To further improve performance a mixed precision linear solver is used to solve the linear equation system resultant from Eqn. (5). Here the iterative refinement algorithm is used as shown in Algorithm 1 where the inner red-black SOR iteration is conducted at single precision with a double precision update outer loop. This strategy has been found in previous work to produce a double precision result while the majority of the computational cost is taken as single precision [31]. Cache memory on the GPUs are used to form the linear system and perform the linear solve enhancing memory bandwidth throughput.

Algorithm 1 Iterative Refinement.

while $||r|| > tol$ **do**
  $r = Ax - b$ (Double precision)
  for $j = 1, 2, ..., n_{iter}$ **do**
    Perform cycle of Red-Black SOR to solve $Ae = r$ (Single precision)
  $x = x + e$ (Double precision)
  Calculate $||r||$ (Double precision)

4. Validation & performance

4.1. Overset assembly

Overset assembly is performed on randomly arranged cubical particles in a triply periodic box, as depicted in Fig. 10, to evaluate the overall acceleration from GPUs. The overall cost of 100 different configurations is evaluated and compared between the pure CPU assembly of Horne and Mahesh [1] and the current work. A cell loading of approximately 40,000 computational cells per processor is selected. 4 nodes each containing 4 NVIDIA V100 GPUs and 1 Intel Xeon gold 6126 processor with 24 cores are used. One MPI rank per core is used for both the pure CPU timings and the current method on GPUs. As shown in the figure, an overall speed-up of around 15x is found for overset assembly using the current work on GPUs. Notably the most expensive operations, cutting and interpolation partner searching, are heavily accelerated. This is expected due to the use of targeted, highly efficient computational kernels for these two operations on the GPUs.

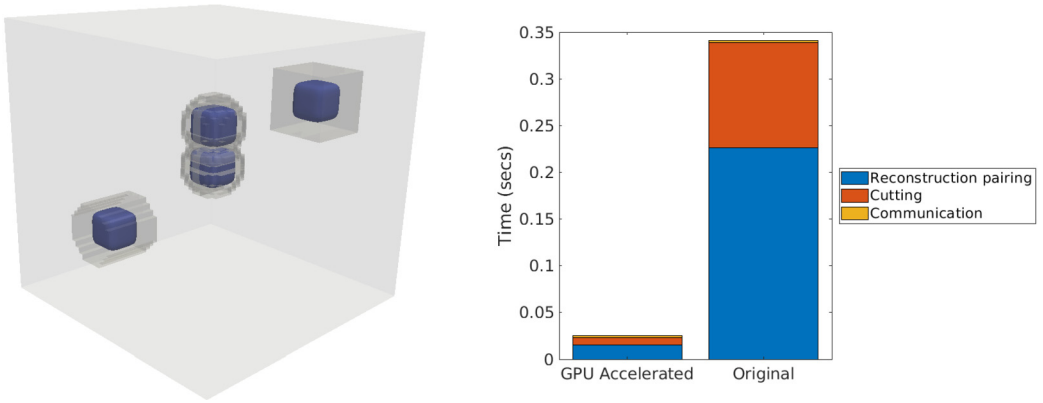

Fig. 10. Randomly arranged cuboid particles in a triply periodic box. In addition to the location of the particles, the type of volume cut and mesh overlap are randomly varied.

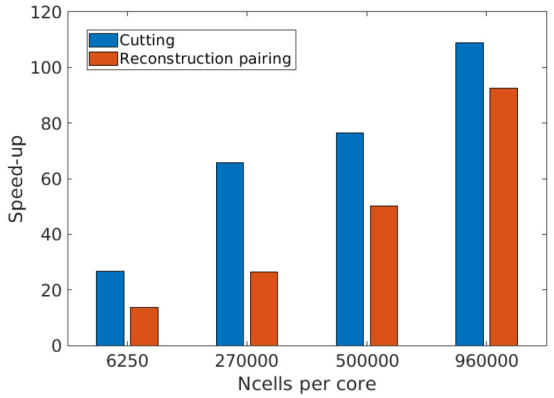

Fig. 11. Overset assembly operation speed-up relative to the original overset method of randomly arranged cuboid particles in a triply periodic box with increasing cell loading.

The acceleration of the overset assembly operations at differing cell loadings for the case is shown in Fig. 11. Substantial improvements in the timings of the two core overset assembly operations are found producing speedup relative to the pure CPU assembly ranging from O(10) to O(100). It is shown that generally higher speedups are obtained at higher cell loadings. This is an expected result since higher cell loadings more fully utilize the GPU resources. This suggests that assembly of large cases with fewer nodes may be feasible through the use of hardware acceleration allowing for substantially less resource use.

It should be noted that the speed-up and timings will be case dependent. The case presented here was selected to capture a breadth of cutting techniques and mesh configurations to reveal a bulk measure of acceleration. Additionally, the comparison between the GPU and pure CPU results is not direct in that the CPU version uses substantially different algorithms. It is anticipated that a non-trivial portion of the speed-up is due to the new algorithms presented here rather than just through the use of GPUs. To perform a completely direct comparison a fully threaded CPU form of the presented overset algorithms would have to be constructed. This would entail substantial changes to the pure CPU version which is outside the scope of this work.

4.2. Numerical method

4.2.1. Taylor-Green vortex

A simulation of an empty overset patch in a Taylor-Green vortex is used to assess the numerical character of the current method. This case has been used in past work to assess the numerical accuracy and kinetic energy properties of methods [30,2]. An overlapping overset patch is placed in a triply periodic domain rotated along all three Euler x-y-z angle axes at 25 degrees as shown in Fig. 12.

The exact solution of the case is taken as

$$u = -cos(2\pi x)sin(2\pi y)f, v = sin(2\pi x)cos(2\pi y)f, w = 0,$$
$$p = 0.25(cos(4\pi x) + cos(4\pi y))f^2, f = e^{-8\pi^2 \nu t}, \tag{11}$$

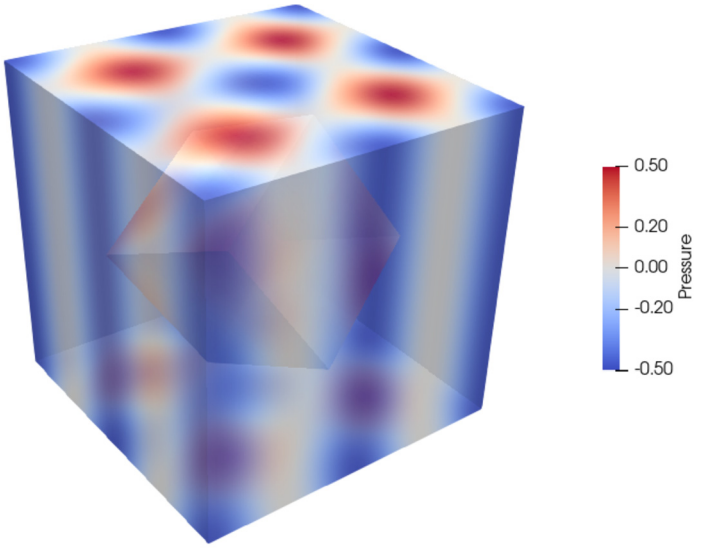

Fig. 12. Taylor-Green vortex domain and mesh arrangement. The length of the domain is selected to be $L_{x,y,z} = 1.0$. The empty overset patch is chosen to be half the size of the domain. The mesh overlap is fixed at approximately 30% overlap.

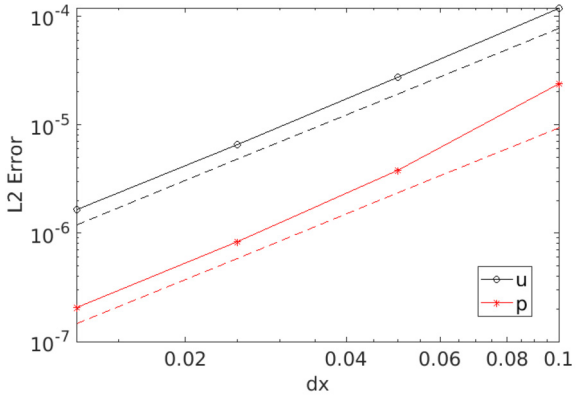

Fig. 13. L2 error of velocity, u, and pressure, p, throughout both meshes averaged from $t = 0.0 - 0.5$ for the Taylor-Green problem. Solid lines with symbols indicate simulation results, dashed lines indicate 2^{nd} order accuracy.

where u, v, and w are the Cartesian velocity components, p is the pressure, and f is the viscous time dependency of the result. To evaluate the order of accuracy of the method, the exact solution is compared to results from the method for varying both meshes over a fixed time interval taken as $t = 0.5$. The CFL is kept constant over all simulations to limit the influence of temporal accuracy on the result. Second order accuracy is found for all evaluated flow quantities as depicted in Fig. 13. Additionally, both the velocity and pressure are found to be several orders of magnitude more accurate than a previous result of the same case when using a predictor-corrector method [2].

Limiting numerical diffusion is key to accurately simulating turbulent flows. If the numerical diffusion is comparable to the viscosity, or modeled eddy viscosity as would be the case of large-eddy simulation, the energy cascade can be heavily attenuated resulting in incorrect flow statistics. To evaluate the numerical diffusion and the energy properties of the method the total kinetic energy in the flow domain is evaluated over long time intervals and compared to the exact solution derived from Eq. (11). A uniform mesh resolution of $dx = 0.0125$ is selected for both meshes. The meshes are arranged the same as shown in Fig. 12. As discussed in Mahesh et al. [30], the method is only expected to preserve the kinetic energy in the absence of temporal errors. To limit temporal error, the time step is adjusted until the solution has converged within 1% for each Reynolds number through numerical experiments before each simulation.

The resulting kinetic energy is shown in Fig. 14. As can be readily seen, the method is found to be capable of accurately capturing the evolution of kinetic energy for all Reynolds numbers evaluated. This includes $Re = 1e9$ where there is nearly no viscous diffusion in the result. This is critical result in that it suggests the method will be capable of accurately simulating high Re number flows without introducing erroneous diffusion. It additionally shows that any additional errors produced at overset boundaries or within the method are bounded such that such flows are feasible for long integration times.

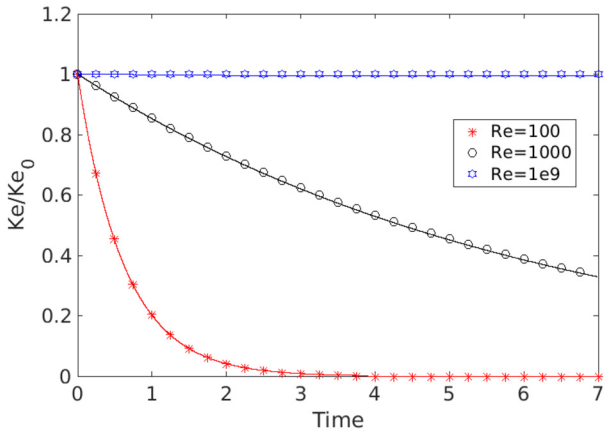

Fig. 14. Total kinetic energy throughout domain normalized by the initial kinetic energy Ke_0. Symbols are from the analytic solution given by Eq. (11), lines are the results from the simulations.

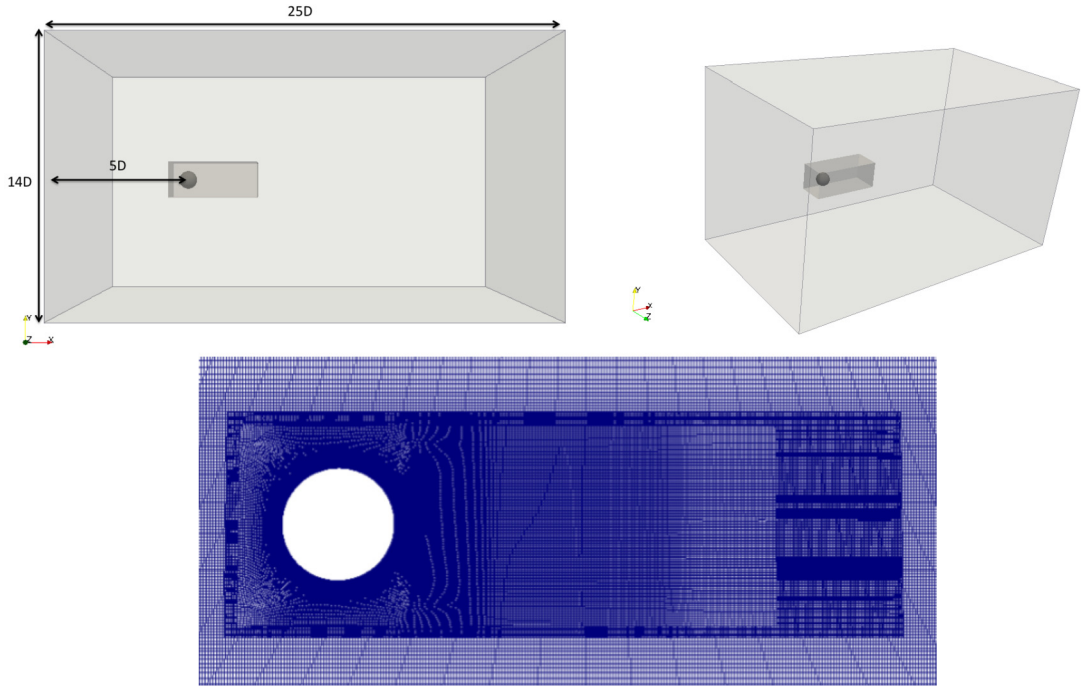

Fig. 15. Re=3700 flow over a sphere domain and meshes. All units non-dimensionalized by the sphere diameter D.

4.2.2. Re=3700 flow over a sphere

Flow over a sphere at Re=3700 is simulated to further evaluate to evaluate the speedup and the accuracy of the method for a 3-dimensional turbulent case. The domain and meshes are depicted in Fig. 15. The domain size is selected to match previous work of the case ([32]). Hexahedral unstructured meshes are used for both the overset sphere mesh and the background mesh.

First the performance of the method is evaluated for increasing cell loadings with differing numbers of GPUs. 1 node containing 4 NVIDIA V100 GPUs and 1 Intel Xeon gold 6126 processor with 24 cores is used. Both meshes are uniformly refined. Timings over 100 time steps are taken using 1 MPI rank per core and a differing number of GPUs. Each result is compared to the timings of a pure MPI, non-threaded CPU result using the same method and the original method from Horne and Mahesh [2]. The results are depicted in Fig. 16. The speedup is found to range from 8x-56x compared to the original method depending on the number of GPUs available on a node and the cell loadings. The timings are found to scale with increasing number of GPUs within a node with the best scaling occurring at the higher cell loadings. It is generally found that the GPUs provide the most benefit at the higher cell loadings.

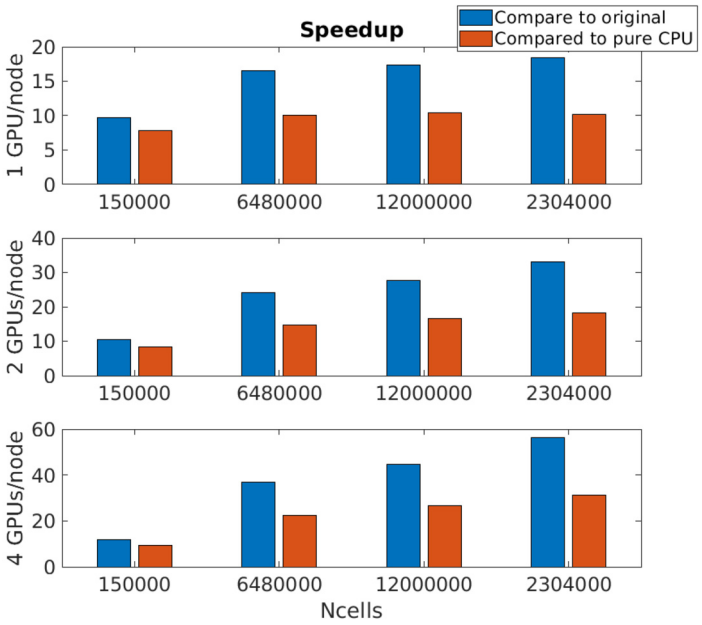

Fig. 16. Turbulent flow over sphere speedup relative to the original overset method and a pure, non-threaded, CPU version of the new method.

The speedup values reported here are anticipated to be case and hardware dependent. A significantly more challenging case could potentially alter the convergence of the method resulting in less speedup. Faster, more modern processors will produce more throughput resulting in less speedup when compared to the same NVIDIA V100 GPUs. Additionally, while a pure CPU version of the new method is shown, it is non-threaded and as such uses different algorithms for the steps of the method. This means the comparison is not exactly direct. The change to the method itself is already seen in the result to produce roughly a 2x speedup over the original method. It is likely that a change in the algorithms and further optimization of the pure CPU version would further improve the speedup relative to the original method such that the overall GPU speedup would lessen.

A simulation is performed of the case using a $N_{cv} = 12,000,000$ cell loading to evaluate ability of the method to produce accurate turbulent flow fields. The mesh spacing is chosen to be $dx = 0.015D$, where D is the sphere diameter and the near wake spacing is chosen to be $D = 0.014D$. The resolution at the edge of the overset patch is selected to closely match that of the background mesh minimizing potential reconstruction errors. The time step is selected to be $dt = 5e-4$. Flow statistics are collected after $t/(D/U_{inf}) = 175$ non-dimensional time units and are collected until $t/(D/U_{inf}) = 300$, where u_{inf} is the inflow velocity.

An instantaneous flow field is shown in Fig. 17. The vortical structures of the result qualitatively match previous work on the case with notably no vortex distortion due to the presence of the overset reconstruction boundaries. Flow averages are additionally shown. Good agreement is found between the average streamwise velocity behind the sphere and previous simulation results. The coefficient of pressure, C_p is found to agree around the surface of the sphere to an experimental result. The drag force and wake cavity length are also in good agreement with previous experimental studies as outlined in the table.

	C_d	L_w	St
Simulation	0.392	2.2D	0.22
Rodriguez et al., 2011	0.394	2.2D	0.215
Experiments	0.39, Schlichting and Gersten [33]	–	0.225, Kim and Durbin [34]

Fig. 18 depicts power spectra calculated from streamwise velocity data collected from a probe located $5D$ behind and $1.5D$ upwards from the center of the sphere. The shedding frequency is found to produce a Strouhal number of $St = 0.22$ in good agreement with previous results illustrating the method's time accuracy. The slope of the spectra is found to roughly follow a $-5/3$ slope, as has been found for probes in this section of the wake by other work [32].

The method overall is found to produce accurate results for this case, achieving a 1%-5% match to experiments and previous studies for all quantities reported. This is despite the presence of overset reconstruction boundaries and the introduction of the material derivative of pressure into the continuity equation. It was able to do this while being significantly faster than the method presented in [2] which would have taken roughly 35x as much time using the same number of nodes.

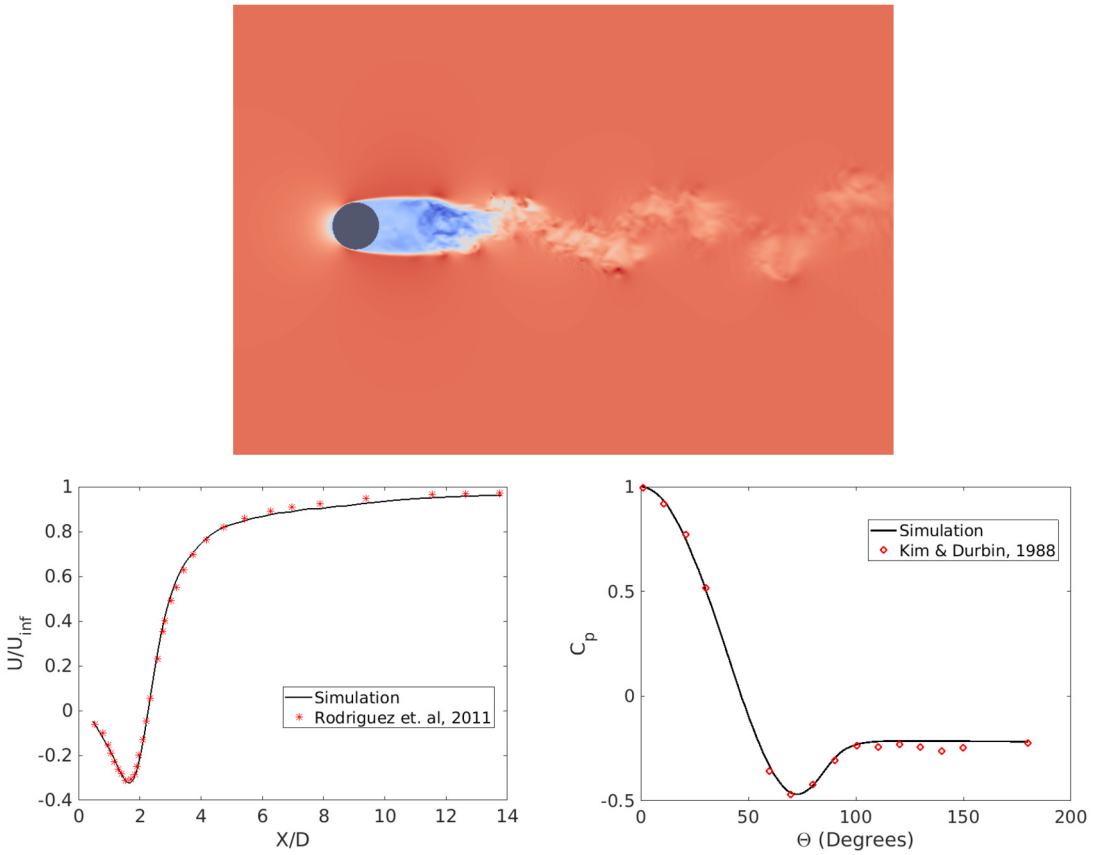

Fig. 17. Instantaneous streamwise velocity contours is first shown for Re=3700 flow over a sphere. The average streamwise velocity from the back of the sphere compared to a previous simulation result into the wake is shown. Finally the variation of the coefficient of pressure C_p around the sphere with angle θ is shown compared to experimental data.

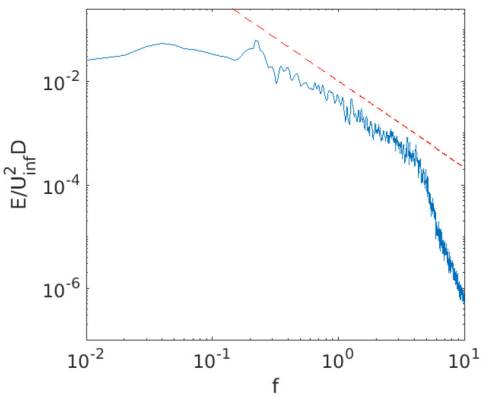

Fig. 18. Power spectra, E, of the streamwise velocity from a probe located at $x = 5D$, $y = 1.5D$ behind sphere calculated using the Lomb-Scargle power spectral density estimate. The solid line indicates the simulation result. The dashed line indicates a -5/3 slope.

4.2.3. Freely falling sphere

A freely falling sphere in a quiescent fluid is simulated to investigate the acceleration of the method for a dynamically moving case. Such cases have been used in previous work to assess the accuracy and effectiveness of methods to simulate bodies moving with full 6-DOF [2,35]. Trajectories of substantially different character are found depending on the Galileo number, Ga, and the density ratio of the sphere to the fluid, $m*$ [36,37]. The flow domain is depicted in Fig. 19. The domain dimensions are selected to match those used successfully to simulate the case in previous studies [2]. The Galileo number is chosen to be $Ga = 291$, and density ratio, $m* = 7.8$.

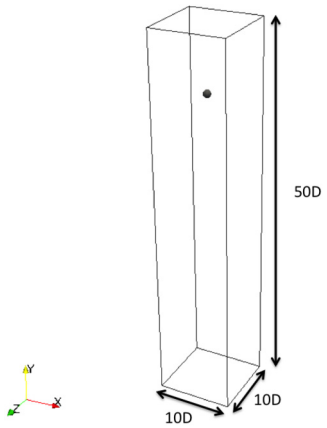

Fig. 19. Flow domain for sphere freely falling in quiescent fluid.

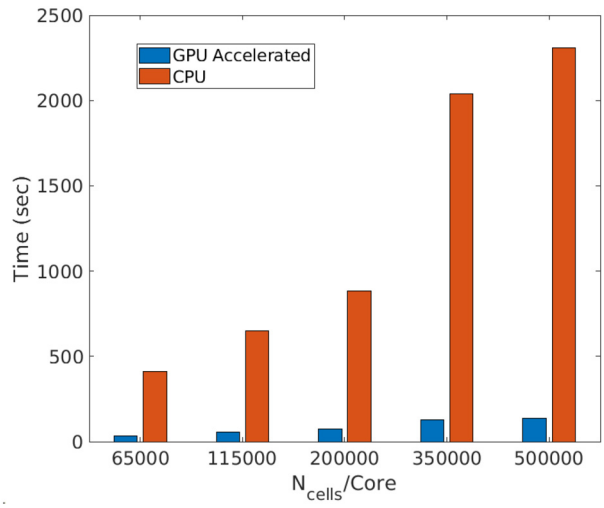

Fig. 20. Timings for 100 time steps of a sphere freely dropping in quiescent fluid at different cell loadings per core.

Timings are assessed over 100 time steps for different mesh refinement levels comparing the original overset method to the accelerated solver at differing processor cell loadings. Two computational nodes are used with 2 V100 GPUs per node, each with an Intel Xeon gold 6126 24 core processor. One MPI rank is used per available core. The resulting timings are shown in Fig. 20. The speedup is found to range from 12x to 20x over the cell loadings evaluated. This is despite the dynamic memory requirements due to the changing mesh connectivity. The speedup is found to be improved at lower cell loadings compared to the timing results found for the static flow over a sphere case. This is likely due to bottlenecks in the original method regarding dynamic matrix assembly, dynamic overset assembly, and mesh movement, all of which have been accelerated in the current method.

5. Conclusions

The unstructured overset method presented in this work was created from its core to effectively utilize GPU hardware acceleration. Performant algorithms from real-time ray-tracing and collision detection such as LBVH and the GJK algorithm are modified and extensively used to assemble arbitrarily overlapping overset meshes into a single cohesive result. Speedups of core kernels up to O(100x) are found relative to a previous benchmark overset method illustrating the effectiveness of hardware acceleration for assembly.

A novel over-decomposition strategy is used to enhance concurrency during the fluid flow solve. Halo updates, overset reconstructions, velocity iterations, and pressure iterations are able to be performed asynchronously across an arbitrary number of GPUs within a node. A novel numerical method was introduced to solve the motion of the fluid that produces low condition number linear systems while maintaining desired accuracy. This method enabled the use of an effective mixed-precision linear solver that maximizes memory bandwidth using GPU cache memory. Overall, this strategy is found to be able to produce substantive speedup, ranging from 15x to 50x, relative to a pure MPI implementation using modern

heterogeneous computing resources for high cell loadings. This is achieved while also having good kinetic energy properties and producing time accurate flow results for a turbulent flow case.

Overall when compared to the original pure CPU overset method larger simulations of turbulent flow cases are clearly feasible. This is shown in the overall high speedups found using the method over all cases presented in this work. Additionally, less resources can be used for current simulations where substantially less computational nodes can match the performance of current pure CPU calculations.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

Computing resources from the Minnesota Supercomputing Institute (MSI) are gratefully acknowledged for all results shown in this work. This work was supported by the United States Office of Naval Research under Grants N00014-18-1-2356 and N00014-20-1- 2717 monitored by Drs. Ki-Han Kim and Peter Chang respectively.

References

[1] W.J. Horne, K. Mahesh, A massively-parallel, unstructured overset method for mesh connectivity, J. Comput. Phys. 376 (2019) 585–596.
[2] W.J. Horne, K. Mahesh, A massively-parallel, unstructured overset method to simulate moving bodies in turbulent flows, J. Comput. Phys. 397 (2019) 108790.
[3] W. Xian, A. Takayuki, Multi-gpu performance of incompressible flow computation by lattice Boltzmann method on gpu cluster, Parallel Comput. 37 (2011) 521–535.
[4] D. Rossinelli, M. Bergdorf, G.-H. Cottet, P. Koumoutsakos, Gpu accelerated simulations of bluff body flows using vortex particle methods, J. Comput. Phys. 229 (2010) 3316–33333.
[5] A. Corrigan, F. Camelli, R. Lohner, J. Wallin, Running unstructured grid-based cfd solvers on modern graphics hardware, Int. J. Numer. Methods Fluids 66 (2011) 221–229.
[6] V. Asouti, X. Trompoukis, I. Kampolis, K. Giannakoglou, Unsteady cfd computations using vertex-centered finite volumes for unstructured grids on graphics processing units, Int. J. Numer. Methods Fluids 67 (2011) 232–246.
[7] A. Khajeh-Saeed, J.B. Perot, Direct numerical simulation of turbulence using gpu accelerated supercomputers, J. Comput. Phys. 235 (2013) 241–257.
[8] D. Chandar, J. Sitaraman, D. Mavriplis, A gpu-based incompressible Navier–Stokes solver on moving overset grids, Int. J. Comput. Fluid Dyn. 27 (2013) 268–282.
[9] J. Crabill, F. Witherden, A. Jameson, A parallel direct cut algorithm for high-order overset methods with application to a spinning golf ball, Comput. Fluid Dyn. J. 374 (2018).
[10] J. Bonet, J. Peraire, An alternating digital tree (adt) algorithm for 3d geometric searching and intersection problems, Int. J. Numer. Methods Eng. 31 (1991) 1–17.
[11] F. Witherden, A. Farrington, P. Vincent, Pyfr: an open source framework for solving advection-diffusion type problems on streaming architectures using the flux reconstruction approach, Comput. Phys. Commun. 185 (2014) 3028–3040.
[12] A. Chorin, A numerical method for solving incompressible viscous flow problems, J. Comput. Phys. 2 (1967) 12–26.
[13] T. Ohwada, P. Asinari, Artificial compressibility method revisited: asymptotic numerical method for incompressible Navier–Stokes equations, J. Comput. Phys. 229 (2010) 1698–1723.
[14] J. Guermond, P. Minev, High-order time stepping for the incompressible Navier–Stokes equations, SIAM J. Sci. Comput. 37 (2015) A2656–A2681.
[15] J. Guermond, P. Minev, High-order time stepping for the Navier–Stokes equations with minimal computational complexity, J. Comput. Appl. Math. 310 (2017) 92–103.
[16] N. Loppi, F. Witherden, A. Jameson, P. Vincent, A high-order cross-platform incompressible Navier–Stokes solver via artificial compressibility with application to a turbulent jet, Comput. Phys. Commun. 233 (2018) 193–205.
[17] J. Clausen, Entropically damped form of artificial compressibility for explicit simulation of incompressible flow, Phys. Rev. E 87 (2013) 013309.
[18] D. Dupuy, A. Toutant, F. Bataille, Analysis of artificial pressure equations in numerical simulations of a turbulent channel flow, J. Comput. Phys. (2020) 109407.
[19] H. Tang, S. Jones, F. Sotiropoulos, An overset-grid method for 3d unsteady incompressible flows, J. Comput. Phys. 191 (2003) 567–600.
[20] J. Pantaleoni, D. Luebke, Hlbvh: hierarchical lbvh construction for real-time ray tracing of dynamic geometry, in: Proceedings of the Conference on High Performance Graphics, 2010, pp. 87–95.
[21] T. Karras, Maximizing parallelism in the construction of bvhs, octrees, and k-d trees, in: Proceedings of the Fourth ACM SIGGRAPH/Eurographics Conference on High-Performance Graphics, 2012, pp. 33–37.
[22] E. Gilbert, D. Johnson, S. Keerthi, A fast procedure for computing the distance between complex objects in three-dimensional space, IEEE J. Robot. Autom. 4 (1988) 193–203.
[23] C. Ericson, Real-Time Collision Detection, CRC Press, 2004.
[24] R. Meakin, Composite overset structured grids, in: Handbook of Grid Generation, CRC Press, New York, 1998.
[25] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, D. Manocha, Fast Bvh Construction on Gpus, Computer Graphics Forum, vol. 28, Wiley Online Library, 2009, pp. 375–384.
[26] E. Stehle, H. Jacobsen, A memory bandwidth-efficient hybrid radix sort on gpus, in: Proceedings of the 2017 ACM International Conference on Management of Data, 2017, pp. 417–432.
[27] K. Garanzha, J. Pantaleoni, D. McAllister, Simpler and faster hlbvh with work queues, in: Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics, 2011, pp. 59–64.
[28] J. Guermond, P. Minev, J. Shen, An overview of projection methods for incompressible flows, Comput. Methods Appl. Mech. Eng. 195 (2006) 6011–6045.
[29] J. Shen, On error estimates of the penalty method for unsteady Navier–Stokes equations, SIAM J. Numer. Anal. 32 (1995) 386–403.
[30] K. Mahesh, G. Constantinescu, P. Moin, A numerical method for large-eddy simulation in complex geometries, J. Comput. Phys. 197 (2004) 215–240.

[31] M. Baboulin, A. Buttari, J. Dongarra, J. Kurzak, J. Langou, J. Langou, P. Luszczek, S. Tomov, Accelerating scientific computations with mixed precision algorithms, Comput. Phys. Commun. 180 (2009) 2526–2533.
[32] I. Rodriguez, R. Borell, O. Lehmkuhl, C.D. Perez Segarra, A. Olivia, Direct numerical simulation of the flow over a sphere at re = 3700, J. Fluid Mech. 679 (2011) 263–287.
[33] H. Schlichting, K. Gersten, Boundary-Layer Theory, Springer, 2016.
[34] H. Kim, P. Durbin, Observations of the frequencies in a sphere wake and of drag increase by acoustic excitation, Phys. Fluids 31 (1988) 3260–3265.
[35] M. Uhlmann, An immersed boundary method with direct forcing for the simulation of particulate flows, J. Comput. Phys. 209 (2005) 448–476.
[36] M. Horowitz, C.H.K. Williamson, The effect of Reynolds number on the dynamics and wakes of freely rising and falling spheres, J. Fluid Mech. 651 (2010) 251–294.
[37] M. Jenny, J. Dusek, G. Bouchet, Instabilities and transition of a sphere falling or ascending freely in a Newtonian fluid, J. Fluid Mech. 508 (2004) 201–239.