Fast **A**pproach for intersecting **N**umerous Polyhedra with scalability on **G**pus: **FANG**

Jacob Jones* Daniel Shevitz* Rao Garimella*

Abstract

Polyhedron intersection is a fundamental problem in computational geometry with applications in fields such as interface reconstruction, high-order remap, and numerical methods requiring polyhedra moments. Traditional methods for polyhedra intersection based on the "Clip-and-Cap" method [17] are hard to port to GPUs for harnessing their parallel processing capabilities. This is because the computational complexity of a kernel that clips an arbitrary polyhedron with a plane can be highly variable whereas GPUs and other accelerators, prefer simple kernels with fixed complexity. In this paper, we propose a novel approach to efficiently compute polyhedra-polyhedra intersections on GPUs by decomposing all polyhedra into tetrahedra and performing exclusivley tetrahedra-tetrahedra intersections in parallel. This method, while increasing the number of intersection tests, reduces and bounds the complexity of each intersection operation making it amenable to porting to the GPU. We optimize the efficiency of the tetrahedra-tetrahedra intersection algorithm by precomputing bounding boxes to eliminate non-intersecting pairs and using static arrays to avoid dynamic memory allocation. Additionally, we address the efficient implementation of tetrahedra-plane intersections, identifying five geometric cases and precomputing resulting topologies for rapid lookup. Our approach simplifies the problem of polyhedra intersection and reduces it to many smaller computations, making it well-suited for applications requiring high-performance geometric processing. Experimental results validate the effectiveness and scalability of our method on CPUs with with the porting and testing on GPUs planned as the next steps.

1 Introduction and Motivation

Polyhedron intersection is a fundamental problem in computational geometry with numerous applications across various fields, including interface reconstruction [3], high-order remap [6, 7, 2], and numerical methods [5, 14, 1]. This problem is particularly significant in hydrodynamic simulations involving moving meshes, such as the Arbitrary Lagrangian-Eulerian (ALE) method [8]. Moving meshes can become distorted, causing elements to invert. To address this issue, many codes employ a remapping or remeshing step, transferring the solution from the distorted mesh to a new, higher-quality mesh. These ALE methods [11] sometimes require the computation of the intersection between the old and new meshes, which can be computationally expensive.

The problem of mesh-mesh intersection is addressed by breaking it down into polyhedron-polyhedron intersections. Initially, a bounding box intersection check is performed to identify all cells in the source mesh that are close to a cell in the target mesh. Usually, a geometric accelearation data structure (such a KD-tree) is used. After this preprocessing step, the problem simplifies to computing the intersection of two polyhedra at a time and computing the moments of the resulting intersection polyhedron. These moments are used to accumulate integrated values from the source cell to the target cell for conservative remap [6, 7], as well as the Moment of Fluids (MOF) method [15].

While several methods have been proposed to address this problem [16, 12, 4, 13], most do not take advantage of modern architectures, such as GPUs. Although these existing methods are robust, they do not scale well on GPUs as they are not designed to leverage the parallelism that GPUs offer. There are a couple of reasons for this: first, the size of data necessary for polyhedra-polyhedra intersection using the Clip-and-Cap method is highly variable and potentially expensive to store on the GPU. In the absence of dynamic memory management capabilities on the GPU, one must decide on the maximum data size at compile time and statically allocate it. In order to do this, the maximum number of vertices and edges must be decided at compile time. Second, the Clip-and-Cap method often involves complex conditional logic involving the current polyhedron, which leads to thread divergence on the GPU. This divergence can significantly impact performance, as threads that take different paths must be serialized, reducing the overall parallelism.

In this work, we address the challenge of effi-

*Los Alamos National Laboratory.

ciently computing polyhedra-polyhedra intersections by proposing a novel approach that leverages the parallel processing capabilities of modern GPUs. Our method decomposes all polyhedra into tetrahedra and performs numerous tetrahedra-tetrahedra intersections in parallel. Although this approach may seem counterintuitive due to the increased number of intersection tests, it allows us to fully exploit the massive parallelism offered by GPUs. This is achieved by utilizing a simpler kernel with bounded complexity. By optimizing the efficiency of the tetrahedra-tetrahedra intersection algorithm and minimizing branch divergence, our method demonstrates excellent performance on GPUs.

This note is organized as follows: In Section 2, we provide background information on existing methods for polyhedra intersection. In Section 3, we describe our methodology for polyhedra-polyhedra intersection, including the tetrahedra-tetrahedra and tetrahedra-plane intersection algorithms. In Section 4, we present experimental results demonstrating the effectiveness and scalability of our approach on modern GPU architectures. Finally, in Section 5, we conclude with a summary of our findings and future research directions.

2 Background

Previous methods for polyhedra intersection often use variations of the Sutherland-Hodgman algorithm [17], also known as the "Clip-and-Cap" method. In this approach, one polyhedron is decomposed into planes forming convex shapes, which sequentially clip the other polyhedron. After each clipping, the resulting polyhedron is reconstructed for the next step. This method, along with extensions [16], forms the basis for many intersection codes, such as r3d [13] and IRL [4].

A key constraint is that the polyhedron transformed into planes must be convex. Non-convex polyhedra can be decomposed into convex subsets through techniques like tetrahedralization, allowing each tetrahedron-polyhedron intersection to be processed independently. However, this step can add computational overhead and complexity, especially for irregular polyhedra.

The robustness of the Clip-and-Cap method comes from the fact that nodes can only lie on one side of a plane, mitigating numerical accuracy and roundoff errors. When a node is exactly on the plane, it is consistently assigned to the positive or negative side, making coincidence manageable algorithmically. However, these methods can sometimes produce unexpected polyhedra [10], where disconnected pieces are erroneously connected by flat volumes. Such anomalies can be detected and eliminated by a post-processing step if necessary. However, they do not affect the overall moments of intersection.

As discussed before, despite its robustness, the Clip-and-Cap method has limitations in handling complex and concave polyhedra and leveraging modern computational architectures like GPUs. Addressing these limitations requires innovative approaches that efficiently handle polyhedra intersections while fully exploiting the parallel processing capabilities of modern hardware.

3 Methodology

Our approach to polyhedra-polyhedra intersection is simple but effective. Instead of decomposing only one, possibly non-convex, polyhedron into tets and intersecting those tets with the other polyhedron, we decompose both polyhedra into tets and perform only tet-tet intersections.

Suppose we have a polyhedron-polyhedron intersection where both polyhedra have the same topology (the same number of nodes,edges and faces and same connectivity). One polyhedron is decomposed into n tetrahedra, which become $4n$ planes. To find the intersection of these polyhedra, we would require $4n$ polyhedron-plane intersections. With a full decomposition of both polyhedra, the number of tetrahedron-plane intersections becomes $4n^2$. While this estimate may initially seem worse, we leverage several key optimizations to speed up the process.

Firstly, we precompute the bounding boxes of the tetrahedra and eliminate pairs whose bounding boxes do not intersect, reducing the number of intersection tests. Bounding box precomputation helps quickly discard non-intersecting pairs, focusing computational resources on potential intersections.

Secondly, we use a static array of tetrahedra coordinates instead of complex data structures with dynamic memory allocation. Static arrays improve cache coherence and reduce overhead, which is beneficial in high-performance computing environments.

Additionally, the simplicity of tetrahedra makes intersection tests straightforward with fewer special cases. By reducing the problem to tetrahedra-plane intersections, we use efficient algorithms optimized for these basic geometric primitives.

Moreover, our approach is highly scalable. As the number of polyhedra increases, the decomposition into tetrahedra and intersection tests can be distributed across multiple processors or computing nodes. This parallelization capability is crucial for handling large-scale simulations and datasets, achieving significant speedups and managing complex scenarios efficiently.

3.1 Vertex-Plane Distance Calculation

A plane $\mathbf{P}$ in n-dimensional space is defined by $\mathbf{n} \cdot \mathbf{x} = d$, where

$\mathbf{n} \in \mathbb{R}^n$ is the normal vector and d is the distance from the origin. The signed distance between a vertex $\mathbf{v}$ and the plane is computed as $\|\mathbf{v}\|_\mathbf{P} = \mathbf{n} \cdot \mathbf{x} - d$.

The sign indicates the side of the plane: $\|\mathbf{v}\|_\mathbf{P} \geq 0$ means the vertex is opposite the normal vector, while a negative distance means it is on the same side. This is crucial for determining vertex positions relative to a polyhedron in intersection tests. A vertex is clipped if the distance is negative.

For the intersection of a line segment defined by points $\mathbf{v}_1$ and $\mathbf{v}_2$ with the plane, the intersection point $\mathbf{v}_i$ is given by:

$$\mathbf{v}_i = \frac{\|\mathbf{v}_1\|_\mathbf{P} \cdot \mathbf{v}_2 - \|\mathbf{v}_2\|_\mathbf{P} \cdot \mathbf{v}_1}{\|\mathbf{v}_1\|_\mathbf{P} - \|\mathbf{v}_2\|_\mathbf{P}} \tag{3.1}$$

3.2 Tetrahedra-plane Cut To discuss the complete algorithm for tetrahedra-tetrahedra intersection, we must first address the efficient implementation of tetrahedra-plane intersection. Given that a tetrahedron is a simple shape, we simply account for all the geometries resulting from the intersection of a plane with a tetrahedron.

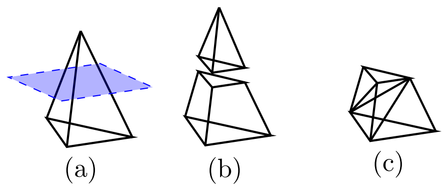

Figure 1: (a) A tetrahedra being cut by a plane. (b) The resulting two new shapes from the cut. (c) If the prism shape is chosen it is split into three tetrahedra which are put back into the algorithm.

There are five geometric cases for the intersection of a tetrahedron and a plane, each corresponding to the number of vertices of the tetrahedron that are clipped by the plane:

Number of Vertices Clipped	Resulting Shape
0	The tetrahedron itself
1	Triangular prism
2	Triangular prism
3	Tetrahedron
4	No resulting polyhedron

Table 1: Resulting shapes based on the number of vertices clipped from tetrahedra.

Since the only two possible shapes from this clipping are a tetrahedron or a triangular prism, we can ensure that only tetrahedra remain after the clip by splitting the triangular prism into three tetrahedra. This case is illustrated in Figure 1. The process of splitting a triangular prism into tetrahedra involves identifying the vertices and edges of the prism and then constructing new tetrahedra that fill the volume of the prism without overlapping. This process occurs once and then the splits are recorded in topology tables.

Given that a tetrahedron has four vertices, each with two states (clipped or unclipped), there are only $2^4 = 16$ possible ways a tetrahedron can be clipped by a plane. This allows us to precompute the resulting topologies and store them in lookup tables. These lookup tables are then used to quickly determine the resulting polyhedra for any given clipping scenario, significantly speeding up the intersection computation. We index into this tables by using the bits set by clipped vertices to construct an integer between 0 and 15. For example, assume we have a tetrahedron with a plane that clips its first and third vertices. This would result in the set of bits 1010 which once converted from binary to an integer would be 10. We then use 10 as the index which gives us the corresponding topological infromation from out table.

The precomputation involves analyzing each of the 16 possible clipping configurations and determining the resulting set of tetrahedra for each case. This analysis is done offline and the results stored in a compact and efficient data structure.

The use of lookup tables is particularly advantageous in high-performance computing environments, where the overhead of dynamic computation can be a bottleneck. By leveraging precomputed results, we achieve constant-time lookups for each clipping scenario, thereby reducing the overall computational complexity. This approach also enhances the robustness of the algorithm, as it eliminates the need for complex conditional logic and ensures that all possible cases are handled correctly.

In addition to the computational benefits, the use of precomputed lookup tables also simplifies the implementation of the algorithm. By encapsulating the clipping logic in a set of precomputed results, we reduce the complexity of the code and make it more maintainable. This modular approach also facilitates debugging and testing, as each component of the algorithm can be verified independently.

3.3 Overall Tetrahedron-Tetrahedron Intersection Algorithm With the tetrahedra-plane intersection algorithm established, we now discuss the overall tetrahedra-tetrahedra intersection algorithm. First, we check if the bounding boxes of the two tetrahedra intersect. If they do, we proceed with the intersection algorithm.

We add one tetrahedron to a list to store intermediate and final results. We then loop over the planes of the other tetrahedron, performing a tetrahedra-plane intersection with each tetrahedron in the list. If the intersection is non-empty, the resulting tetrahedra are added to a new list. This process is repeated for all planes of the second tetrahedron.

Since we can add at most three tetrahedra to the list for each tetrahedra in the list, the maximum size required for the list of all tetrahedra is $\sum_{i=0}^{i=4} 3^i = 121$. This is a manageable size for a static array and is parallelized on a GPU. The use of a static array allows for good cache performance and reduced memory overhead when compared to dynamic memory structures.

3.4 Moment Calculation In many simulations, the moments of the intersection of two polyhedra are essential. These moments are utilized in remap methods [6, 7] and other numerical techniques that require the volume or centroid of the intersection. Typically, the moments of the intersection are computed by summing the moments of the tetrahedra that are created through the facetization of the polyhedra [9]. Since our algorithm provides the intersecting polyhedra as a list of tetrahedra we also directly use these methods as well.

Also, in practice only the first few moments of intersection are needed. Therefore, we opt to calculate the moments of tetrahedra directly using formulas derived from their coordinates. The moment of a tetrahedron T is defined as the integral

$$\int\int\int_T x^i y^j z^k \, dV. \tag{3.2}$$

Where $i + j + k \leq n$, with n being the chosen moment order.

Let the coordinates of a tetrahedron T be given by (x_1, y_1, z_1), (x_2, y_2, z_2), (x_3, y_3, z_3), and (x_4, y_4, z_4). To calculate the analytic moments over the tetrahedron, we use a coordinate transformation from a reference tetrahedron with coordinates $T_R = (0, 0, 0)$, $(1, 0, 0)$, $(0, 1, 0)$, $(0, 0, 1)$. The transformation is given by

$$\begin{aligned} x &= x_1 + (x_2 - x_1)\xi + (x_3 - x_1)\eta + (x_4 - x_1)\zeta, \\ y &= y_1 + (y_2 - y_1)\xi + (y_3 - y_1)\eta + (y_4 - y_1)\zeta, \\ z &= z_1 + (z_2 - z_1)\xi + (z_3 - z_1)\eta + (z_4 - z_1)\zeta. \end{aligned} \tag{3.3}$$

Where ξ, η, and ζ are the barycentric coordinates of the reference tetrahedron. The Jacobian of this transformation is given by

$$\mathbf{J} = \begin{bmatrix} \frac{\partial x}{\partial \xi} & \frac{\partial x}{\partial \eta} & \frac{\partial x}{\partial \zeta} \\ \frac{\partial y}{\partial \xi} & \frac{\partial y}{\partial \eta} & \frac{\partial y}{\partial \zeta} \\ \frac{\partial z}{\partial \xi} & \frac{\partial z}{\partial \eta} & \frac{\partial z}{\partial \zeta} \end{bmatrix} = \begin{bmatrix} x_2 - x_1 & x_3 - x_1 & x_4 - x_1 \\ y_2 - y_1 & y_3 - y_1 & y_4 - y_1 \\ z_2 - z_1 & z_3 - z_1 & z_4 - z_1 \end{bmatrix}. \tag{3.4}$$

From this Jacobian, we know the volume V of the tetrahedron is given by $V = \frac{1}{6}|\mathbf{J}|$. The moments of the tetrahedron are calculated by integrating the coordinates over the tetrahedron. Using the transformation from reference coordinates, the moment of a tetrahedron is given by

$$\int_0^1 \int_0^{1-\xi} \int_0^{1-\xi-\eta} x^i y^j z^k \, d\zeta \, d\eta \, d\xi \, |\mathbf{J}|. \tag{3.5}$$

We found that for the first several moments, using these analytic expressions is faster than the more general recursive such as the one given by Koehl [9].

4 Experimental Results

We implemented a basic prototype of our algorithm for tets only in C++ using Kokkos [18], a library for performance portability across multi-core CPUs and GPUs. The tetrahedra vertices are stored in a flattened array. We evaluated our algorithm on an AMD EPYC 7713 64-Core Processor with 128 threads, comparing it to the r3d library [13]. In the test, one tetrahedron was the unit tetrahedron, and the other was randomly generated on the unit sphere with a volume of at least 0.001. We executed 'r3d_init_tet', 'r3d_clip', and 'r3d_reduce' in parallel using OpenMP through Kokkos. The test was run 10 times, and the average execution time was recorded. The results are presented in Table 2.

The results in Table 2 show that our algorithm outperforms the r3d library as the number of tetrahedra increases. While our initialization time ('init') is slightly higher, the 'clip+reduce' phase is much faster, leading to a lower total execution time ('total'). This efficiency gain becomes more pronounced with larger tetrahedra counts, demonstrating better scalability. Overall, our approach is more efficient and scalable.

5 Conclusions

In this work, we presented an algorithm which presented a new algorithm for polyhedra-polyhedra intersection that leverages the parallel processing capabilities of modern GPUs. Our method decomposes all polyhedra into tetrahedra and performs numerous tetrahedra-tetrahedra intersections in parallel. We believe this algorithm will be amenable to the GPU architecture in the future and will be able to handle complex polyhedra intersections efficiently.

# of tet-tet intersections	r3d			Our Algorithm		
	init	clip+reduce	total	init	clip+reduce	total
100000	0.00253203	0.00703764	0.00956967	0.00397144	0.00693702	0.01090846
200000	0.00604507	0.00906832	0.01511339	0.00596718	0.0114017	0.01736888
400000	**0.0101806**	**0.0213422**	**0.0315228**	**0.0107987**	**0.017279**	**0.0280777**
800000	0.0199746	0.0480378	0.0680124	0.0212801	0.0295645	0.0508446
1600000	0.0505566	0.095476	0.1460326	0.0408311	0.0547552	0.0955863
3200000	0.0918407	0.181224	0.2730647	0.0788684	0.104732	0.1836004
6400000	0.193545	0.369134	0.562679	0.156132	0.20269	0.358822

Table 2: Timing results for different numbers of tet-tet intersections. The times are in seconds. The row in bold represents the point at which our algorithm becomes faster than r3d.

References

[1] H. T. Ahn and M. Shashkov, *Adaptive moment-of-fluid method*, Journal of Computational Physics, 228 (2009), pp. 2792–2821.

[2] R. W. Anderson, V. A. Dobrev, T. V. Kolev, and R. N. Rieben, *Monotonicity in high-order curvilinear finite element arbitrary lagrangian–eulerian remap*, International Journal for Numerical Methods in Fluids, 77 (2015), pp. 249–273.

[3] D. J. Benson, *Volume of fluid interface reconstruction methods for multi-material problems*, Appl. Mech. Rev., 55 (2002), pp. 151–165.

[4] R. Chiodi and O. Desjardins, *General, robust, and efficient polyhedron intersection in the interface reconstruction library*, Journal of Computational Physics, 449 (2022), p. 110787.

[5] V. Dyadechko and M. Shashkov, *Moment-of-fluid interface reconstruction*, Los Alamos Report LA-UR-05-7571, 49 (2005).

[6] J. Grandy, *Conservative remapping and region overlays by intersecting arbitrary polyhedra*, Journal of Computational Physics, 148 (1999), pp. 433–466.

[7] A. M. Herring, C. R. Ferenbaugh, C. M. Malone, D. W. Shevitz, E. Kikinzon, G. A. Dilts, H. N. Rakotoarivelo, J. Velechovsky, K. Lipnikov, N. Ray, et al., *Portage: A modular data remap library for multiphysics applications on advanced architectures*, Journal of Open Research Software, 9 (2021).

[8] C. W. Hirt, A. A. Amsden, and J. Cook, *An arbitrary lagrangian-eulerian computing method for all flow speeds*, Journal of computational physics, 14 (1974), pp. 227–253.

[9] P. Koehl, *Fast recursive computation of 3d geometric moments from surface meshes*, IEEE transactions on pattern analysis and machine intelligence, 34 (2012), pp. 2158–2163.

[10] J. López and J. Hernández, *On polytope intersection by half-spaces and hyperplanes for unsplit geometric volume of fluid methods on arbitrary grids*, Computer Physics Communications, 300 (2024), p. 109167.

[11] R. Loubère, P.-H. Maire, and M. Shashkov, *Reale: a reconnection arbitrary-lagrangian–eulerian method in cylindrical geometry*, Computers & fluids, 46 (2011), pp. 59–69.

[12] D. E. Muller and F. P. Preparata, *Finding the intersection of two convex polyhedra*, Theoretical Computer Science, 7 (1978), pp. 217–236.

[13] D. Powell, *r3d: Software for fast, robust geometric operations in 3d and 2d*, Report of Los Alamos national laboratory, LA-UR-15-26964. Report and software are available at https://github. com/devonmpowell/r3d, (2015).

[14] M. Shashkov, *Closure models for multimaterial cells in arbitrary lagrangian–eulerian hydrocodes*, International Journal for Numerical Methods in Fluids, 56 (2008), pp. 1497–1504.

[15] M. Shashkov and E. Kikinzon, *Moments-based interface reconstruction, remap and advection*, Journal of Computational Physics, 479 (2023), p. 111998.

[16] K. Sugihara, *A robust and consistent algorithm for intersecting convex polyhedra*, in Computer Graphics Forum, vol. 13, Wiley Online Library, 1994, pp. 45–54.

[17] I. E. Sutherland and G. W. Hodgman, *Reentrant polygon clipping*, Communications of the ACM, 17 (1974), pp. 32–42.

[18] C. R. Trott, D. Lebrun-Grandié, D. Arndt, J. Ciesko, V. Dang, N. Ellingwood, R. Gayatri, E. Harvey, D. S. Hollman, D. Ibanez, N. Liber, J. Madsen, J. Miles, D. Poliakoff, A. Powell, S. Rajamanickam, M. Simberg, D. Sunderland, B. Turcksin, and J. Wilke, *Kokkos 3: Programming model extensions for the exascale era*, IEEE Transactions on Parallel and Distributed Systems, 33 (2022), pp. 805–817.