GPU-Based Lock-Free Mesh Reduction Using Deterministic Vertex Clustering

by

Khizra Hanif
MSc, Lahore College for Women University, 2019

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

University of Victoria

We acknowledge and respect the Lək^{w}əŋən (Songhees and X^{w}sepsam/Esquimalt) Peoples on whose territory the university stands, and the Lək^{w}əŋən and W̱SÁNEĆ Peoples whose historical relationships with the land continue to this day.

GPU-Based Lock-Free Mesh Reduction Using Deterministic Vertex Clustering

by

Khizra Hanif
MSc, Lahore College for Women University, 2019

Supervisory Committee

Dr. Sean Chester, Supervisor
Department of Computer Science

Dr. Mario Nascimento, Departmental Member
Department of Computer Science

Abstract

The growing demand for real-time rendering and processing of complex 3D models in film production, gaming, and scientific visualization has exposed the limitations of CPU-based mesh simplification techniques. While S-Weld ensures deterministic clustering, its sequential execution makes it unsuitable for large scale meshes. The multi-core P-Weld improves performance through lock-free multithreading but remains constrained by CPU core count and memory bandwidth.

To address these issues, this thesis presents a GPU-accelerated vertex clustering framework that extends the deterministic behavior of P-Weld to a fully parallel and memory-adaptive GPU architecture. The work begins with a direct CUDA port of P-Weld and introduces an On-the-fly neighbor evaluation method that performs clustering without storing explicit adjacency lists, followed by a fully GPU-resident sparse voxel-grid framework for efficient processing of large meshes. Shared-memory caching, warp-synchronous centroid updates, and sparse neighbor filtering are used to minimize redundant computations and improve parallel efficiency.

Existing GPU-based libraries, such as FRNN and cuNSearch, were evaluated but found unsuitable for large irregular meshes on consumer laptops. A custom sparse grid-based neighbor search was therefore developed to perform efficient ε-neighborhood queries entirely on the GPU within limited VRAM. The proposed pipeline was evaluated on five benchmark meshes, including Bunny, Lucy, Thai Statue, Manuscript, and the point cloud LiDAR dataset using various clustering thresholds. The GPU versions achieved identical results while providing 10-26$\times$ speedup, improving the scalability of vertex clustering for large 3D mesh simplification.

In summary, this thesis presents the first deterministic GPU extension of vertex clustering, introducing a memory-efficient sparse-grid neighbor search and a fully parallel pipeline that reproduces accurate results, advancing scalable and reproducible mesh simplification.

Table of Contents

List of Tables

List of Figures

List of Algorithms

Acknowledgments

I would like to express my deepest gratitude to my supervisor, Dr. Sean Chester, for his continuous guidance, patience, and encouragement throughout my research journey. His insightful feedback and expertise were invaluable in shaping this work and helping me grow as a researcher.

Finally, I owe my heartfelt thanks to my family for their unconditional love, prayers, and motivation, which gave me the strength to complete this thesis.

Khizra Hanif,
Victoria, Monday 4th May, 2026.

Chapter 1

Introduction

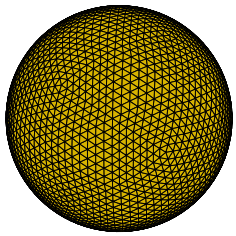
1.1 Motivation and Background

Triangular meshes are often used to show three-dimensional surfaces in computer graphics and geometry processing. As digital models get more detailed, they can contain millions or even billions of triangles. Modern tools like laser scanners, photogrammetry, and CAD software can create these detailed models with high accuracy. But dense meshes are expensive to render, send, and store. When viewed from a distance, many vertices overlap the same pixel, which does not improve the image but still consumes computing power. As shown in figure 1.1, mesh simplification reduces geometric complexity while preserving the model's main shape and appearance.

Among the various simplification approaches, including vertex decimation and edge contraction, vertex clustering has remained attractive because of its conceptual simplicity, deterministic output, and nearly linear runtime [RB93]. In vertex clustering, nearby vertices are merged into a single representative vertex according to a spatial distance threshold ε. The choice of ε controls how aggressively the mesh is simplified. Figure 1.2 shows a two-dimensional example of vertex clustering based on a fixed ε-neighborhood radius. The dashed circles denote ε-neighborhoods centered at individual vertices, defining regions of spatial proximity. Vertices located within overlapping neighborhoods are grouped into clusters and replaced by representative vertices, shown in red. This clustering mechanism forms the fundamental principle behind vertex clustering approaches for mesh simplification

The Serial S-Weld algorithm performs deterministic clustering by visiting vertices sequentially in increasing order of their indices. Once a vertex is assigned to a cluster, all of its neighbors within the ε radius are merged under that cluster. This procedure guarantees consistent results regardless of hardware or thread

(a) A sphere represented by a triangular mesh with numerous vertices and triangles (V=2,562, F=5,120)



(b) A simplified triangular mesh representation of the same sphere (V=162, F=320)

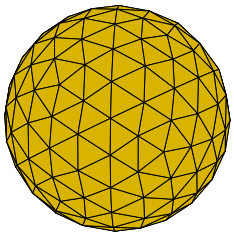


Figure 1.1: Mesh reduction lowers vertex and triangle counts while preserving the visible geometry.

scheduling but is inherently sequential and therefore slow for large meshes.

To address this, Fathollahi and Chester proposed a parallel multicore P-Weld algorithm [FC23]. P-Weld uses the FLANN KD-Tree library as an external library for efficient nearest-neighbour searches and adopts a forward clustering strategy that allows multiple vertices to be processed concurrently while preserving the same results. His work achieved significant speedups over S-Weld on multicore CPUs but remains limited by available CPU cores, particularly on user-grade CPUs.

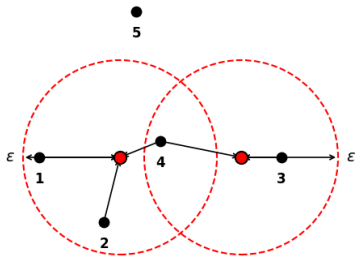


Figure 1.2: Example of vertex clustering with a fixed ε radius. Vertices within the same ε neighborhood are merged into a representative vertex.

1.2 GPU-Based Motivation

Graphics Processing Units provide thousands of lightweight threads that can efficiently execute data-parallel tasks. NVIDIA's general-purpose GPU programming framework, called CUDA allows developers to use shared memory, atomic operations, and synchronization primitives to coordinate these threads. These capabilities make GPUs excellent for spatial problems such as neighbour search and centroid propagation. However, moving a complex CPU algorithm to the GPU is not straightforward, as P-Weld depends on ordered updates, shared dependency counters, and predictable synchronization. In contrast, the GPU executes threads concurrently with minimal ordering guarantees. The challenge of this work is therefore not just performance but also maintaining deterministic, reproducible clustering behaviour on massively parallel hardware.

In this thesis, I explore GPU acceleration for strict-mode vertex clustering. My objective is to design a deterministic GPU pipeline that reproduces the exact output of the CPU-based P-Weld while leveraging GPU parallelism to overcome CPU limitations in computational speed and memory bandwidth.

1.3 Problem Definition

This research focuses on rethinking the dependency-based clustering methodology of P-Weld for massively parallel execution. The main research questions driving this work are:

1. How can the spatial ε-neighbor search be performed efficiently and deterministically within GPU memory without relying on CPU-built data structures?
2. How can the deterministic update order of P-Weld be preserved when exe-

cuted on thousands of GPU threads?
3. What data structures are needed to manage neighborhood information entirely within GPU memory?
4. To what extent can GPU-accelerated clustering outperform existing CPU implementations in terms of speed and scalability without compromising output correctness?
5. How can host–device communication be minimized to approach theoretical speedup limits?

The overall objective is to create a GPU clustering pipeline that maintains the same correctness guarantees as the CPU version while achieving higher throughput and speedup on large meshes.

1.4 Research Objectives

I set the following objectives for this research to address these challenges:

1. Develop a deterministic GPU implementation of strict-mode vertex clustering that produces results identical to the P-Weld algorithm.
2. Exploit GPU parallelism across threads, warps, and thread blocks to achieve substantial performance gains on large meshes.
3. Design a sparse voxel-based neighbor search structure that supports efficient ε-queries without CPU dependency.
4. Quantify host–device transfer overheads and analyze their impact on total runtime.
5. Validate the proposed methods on standard benchmark datasets such as Bunny, LiDAR, and Lucy.

1.5 Novelty and Technical Contributions

I introduce two GPU-based versions of strict-mode vertex clustering and several key algorithmic and structural innovations in this research. The main technical contributions are summarized as follows:

1. Two GPU-based clustering frameworks:
    - *On-the-fly GPU version:* Constructs the voxel grid on the CPU but performs neighborhood discovery and clustering directly on the GPU during each iteration.
    - *Fully GPU-resident sparse voxel-grid version:* Eliminates all CPU dependency by performing voxel grid construction, frontier propagation, and mesh reconstruction entirely within GPU memory.

2. Deterministic reformulation of P-Weld for GPUs: Extends the dependency-based logic of P-Weld into a frontier-driven, SIMT-compatible design where each thread represents a vertex and applies the ordering rule $(v > u)$ to ensure deterministic cluster propagation without cycles. Vertices whose dependencies reach zero form the active frontier and are processed in parallel. This design converts coarse CPU synchronization steps into continuous, fine-grained GPU updates, allowing asynchronous progress across warps.
3. Sparse voxel hash grid with atomic insertions: A GPU-built sparse voxel hash grid is designed using open addressing and atomic operations to allocate memory only for occupied cells. This enables large-scale neighbor search without dense memory allocation.
4. Two-pass deterministic neighbor construction: Implemented a two-stage CUDA process, first counting valid ε-neighbors and then compacting them with prefix-sum offsets to produce adjacency lists that are identical across executions.
5. Fully GPU-resident clustering pipeline: Developed CUDA kernels for centroid computation, compact ID assignment, vertex averaging, and face remapping using prefix-sum operations, completing all steps entirely on the GPU and removing PCIe transfer overhead.
6. Memory and hardware-aware optimizations: Introduced adaptive kernel configurations and shared-memory reuse strategies that allow the framework to process meshes with tens of millions of vertices on GPUs with as little as 6 GB of VRAM.

Together, these contributions form a first ever deterministic, lock-free, and memory-efficient GPU framework that achieves near-linear scalability while maintaining output equivalence with the P-Weld algorithm.

1.6 Integration Challenges with External Libraries

During development, I evaluated two state-of-the-art GPU libraries for neighborhood search. FRNN [LX21], a PyTorch-based implementation, offered fast radius queries but was tightly coupled with Python and Tensor data structures, which made it incompatible with a pure C++/CUDA environment. cuNSearch [Int23], a native C++/CUDA library, was more suitable but suffered from high memory overhead and allocation failures on large meshes due to its dense grid representation under limited GPU memory.

To overcome these issues, I develope a custom sparse GPU neighbour structure inspired by cuNSearch's logic but reimplemented to ensure deterministic key–vertex mappings, better memory utilization, and full compatibility with the strict-mode clustering kernels.

1.7 Overview of Following Chapters

In Chapter 2, the fundamental concepts related to mesh representation, geometry processing, and parallel computing are introduced. Various mesh simplification techniques are categorized, and the evolution of vertex clustering methods is reviewed. The chapter also provides an overview of GPU computing and CUDA's execution model, which lays the foundation for understanding the subsequent GPU-based algorithms developed in this work.

Chapter 3 reviews existing research on serial and parallel vertex clustering. It covers the classical S-Weld algorithm and the parallel CPU-based P-Weld, explaining how each achieves deterministic results through dependency tracking and lock-free updates. The chapter also discusses prior attempts to accelerate neighborhood queries using GPU libraries such as FRNN and cuNSearch, highlighting their limitations in deterministic clustering. This review establishes the motivation for developing a fully deterministic GPU framework.

In Chapter 4, the first GPU implementation of strict-mode vertex clustering is presented, referred to as the *On-the-fly GPU version*. In this design, the voxel grid is constructed on the CPU, while neighborhood discovery and clustering are performed directly on the GPU during each iteration. The chapter explains the kernel structure, data flow between host and device, and the mechanisms used to preserve deterministic cluster propagation. Experimental results from this version serve as a baseline for evaluating later improvements.

Chapter 5 introduces the main contribution of this thesis: a *fully GPU-resident sparse voxel-grid framework* that eliminates all CPU involvement during clustering and reconstruction. This chapter explains the design of the sparse voxel hash grid, the deterministic two-pass neighbor construction, and the lock-free frontier propagation process that operates entirely within GPU memory. It also details the implementation of post-clustering operations such as centroid averaging, compact ID assignment, and face remapping. Performance and scalability improvements over the *On-the-fly version* are analyzed and discussed in depth.

Chapter 6 presents the empirical validation of the proposed framework. The benchmarking setup, datasets, and evaluation metrics are described, followed by a detailed comparison between P-Weld, the *On-the-fly GPU version*, and the fully GPU-resident *sparse voxel-grid framework*.

Finally, Chapter 7 concludes the thesis by summarizing the main findings and outlining the limitations of the current implementation. Potential directions for future research are also discussed, including adaptive grid resolutions, multi-GPU extensions, and integration with real-time rendering pipelines for large-scale geometric datasets.

Chapter 2

Related Work

2.1 3D Mesh Simplification Techniques

Mesh simplification has long been an important topic in computer graphics, aimed at reducing geometric complexity while preserving visual quality. Early methods such as vertex decimation [SZL92] and edge contraction [GH97] remove vertices or collapse edges based on local geometric error measures. While these approaches perform well for moderately sized models, they depend heavily on mesh topology and iterative error computation, which limits their scalability to large, high-resolution datasets.

Later, Out-of-core approaches were proposed to handle very large models that exceeded system memory. Lindstrom's work on external-memory simplification [LT00] efficiently streamed partial meshes from disk while maintaining topology, marking an important milestone for scalability.

2.2 Vertex Clustering

Rossignac and Borrel [RB93] first introduced vertex clustering as a geometry-driven mesh simplification technique that merges spatially close vertices within a uniform voxel grid. The method offers predictable behaviour, linear time complexity, and intuitive control of simplification through the voxel size ε. It provides a scalable and efficient alternative to topology-dependent methods, making it effective for large-scale and real-time applications. However, as the authors note, vertex clustering does not preserve mesh topology or manifold properties, often leading to non-manifold edges or disconnected regions in aggressively simplified models.

Subsequent implementations in Open3D [ZPK18], as well as industry tools like Blender [Fou24a], and Modo [Fou24b] adopted the same voxel-based concept to simplify 3D meshes. These frameworks focused primarily on usability and inte-

gration rather than computational acceleration, leaving vertex clustering as an important but largely serial preprocessing step within many 3D geometry-processing pipelines.

2.3 Deterministic Serial Clustering (S-Weld)

S-Weld was formalized by Fathollahi and Chester [FC23] as Algorithm 1 in his thesis, the fastest publicly available implementation was introduced by Zhou, Pan, and Kobbelt [ZPK18]. In this approach, vertices are processed in ascending ID order, and each vertex adopts the smallest representative ID among its ε-neighbors. This ordered update sequence guarantees reproducible and consistent results across executions, providing a reliable reference for correctness evaluation. Despite its deterministic behavior, S-Weld remains inherently sequential, limiting its scalability and making it unsuitable for parallel hardware.

2.4 Parallel Lock-Free Algorithms: P-Weld and P-Weld-Async

Fathollahi and Chester [FC23] extended the S-Weld with P-Weld, a lock-free parallel algorithm for multicore CPUs. P-Weld represents the mesh as a spatial graph and applies atomic operations (`compare_and_swap`, `atomicAdd`) to propagate centroid updates concurrently, while preserving deterministic equivalence with the serial version. The algorithm provides guarantees of progress, safety, and determinism, and achieves near-linear speedups on multicore systems. A subsequent variant, P-Weld-Async, removes global synchronization to further improve throughput, though intermediate states may temporarily diverge from the deterministic baseline. These methods confirm that strict-mode clustering can be parallelized effectively on CPUs.

2.5 GPU-Based Mesh Simplification and Neighbor Search

GPU acceleration for mesh simplification has been explored for almost two decades. DeCoro and Tatarchuk [DT07] presented one of the earliest real-time simplification systems using vertex shaders, showing that geometric decimation could be performed completely on the graphics pipeline. Later, Mousa and Hussein [MH21] used CUDA for triangular surface reduction and reported significant speed improvements on million-triangle models. Despite these results, most

GPU-based simplification methods are heuristic or non-deterministic and cannot reproduce serial results exactly.

Researchers have also developed GPU data structures for neighborhood search and proximity queries. Hoetzlein [Hoe14] introduced a fixed-radius neighbor search method using a uniform grid, where each point is assigned to a grid cell, sorted by ID, and accessed through prefix-sum offsets while inspecting the 27 neighboring cells. Later works improved cell indexing using Morton codes or compacted per-cell lists [BR14]. These uniform-grid methods perform well for moderate ε values in dense domains but cause high memory usage when ε is small or when meshes are sparse and irregular.

Two types of spatial queries are commonly used in neighbor search. A *k-nearest neighbor (kNN) query* returns exactly the k closest points to a query point, where k is a fixed integer regardless of how far those points are. A *range query* (or ε-neighborhood query) returns all points within a fixed distance ε, with no limit on how many are returned. Since vertex clustering must find every vertex within distance ε to merge correctly, it requires range queries rather than kNN queries.

It is worth clarifying the type of spatial search used throughout this work. P-Weld is often associated with FLANN, which is described as an approximate nearest neighbor library. In practice, however, P-Weld uses nanoflann [BR14], a separate exact KD-tree library that guarantees every point within radius ε is returned no approximation is involved.

More broadly, the libraries evaluated in this work nanoflann [BR14], FRNN [LX21], and cuNSearch [Int23] all support both kNN and range query functionality, but vertex clustering only needs range queries. P-Weld uses the `SearchRadiusSmallerAndBigger` function of nanoflann [BR14], which returns all vertices within ε and separates them into lower- and higher-indexed neighbors for building the dependency graph. There is no k parameter anywhere in this process.

cuNSearch works similarly as a range query tool, but its kernels stop early once a hard limit `CUDA_MAX_NEIGHBORS` is reached. Consequently, in dense regions, some valid neighbors within ε may not be reported. This behavior can compromise correctness in vertex clustering, where missing even a single neighbor may lead to incorrect cluster formation. To address this limitation, the method proposed in this work employs a custom sparse voxel-grid that performs complete ε-range queries on the GPU, ensuring that all neighbors within ε are identified. The search is therefore exact, with no truncation and no approximation.

2.6 Exact kNN on Grids

Several GPU implementations of exact k-nearest-neighbor (kNN) search build upon the same uniform-grid spatial indexing. A common approach, for example by Leite et al. [Lei+12], expands the neighborhood ring cell-by-cell around the query until at least k candidates are found, then prunes the result to the exact k nearest points. Although this improves kNN correctness, the method retains the same dense grid memory model, making it sensitive to the same scalability limitations as fixed-radius search.

2.7 Grid-Based FRNN Libraries

Two already existing open-source libraries adopt this fixed-radius grid paradigm:

- FRNN (PyTorch) [LX21]: wraps the grid-binning and prefix-sum pipeline behind a PyTorch interface. It can reuse grids when ε and the reference set remain constant and notes that for small datasets, brute-force GPU kNN can outperform grid-based search. However, for small ε or very large bounding boxes, grid resolution increases sharply, leading to high memory usage.
- cuNSearch (C++/CUDA) [Int23]: a CUDA counterpart to CompactNSearch designed for particle-based simulation workloads. It bins points into uniform spatial grids (optionally Z-sorted) and exposes per-point neighbor iteration. While performance is strong for dense domains, memory footprint scales with grid resolution, and users typically reorder data to improve locality rather than reduce allocation size.

Both libraries follow Hoetzlein's [Hoe14] uniform-grid design, compute a 3D cell ID for each point, sort by ID, build prefix-sum cell ranges, and iterate over neighboring cells during queries. Whether stored as a dense 3D array or a sparse list of occupied cells, at least one structure scales with the total number of addressable cells, making these frameworks highly sensitive to small ε radii on large domains. This dependence on global grid resolution makes uniform-grid methods particularly inefficient for sparse or highly non-uniform spatial distributions. Other alternative hierarchical data structures such as octrees, KD-trees, and bounding-volume hierarchies (LBVH or Radix-tree variants) reduce dependence on a single global grid resolution and offer more predictable memory use for non-uniform data. Despite these advantages, uniform grids remain the dominant choice for fixed-radius GPU searches due to their regular memory access patterns and high arithmetic intensity. This explains their widespread use in frameworks like FRNN, cuNSearch, and smoothed-particle hydrodynamics (SPH) solvers.

Recent GPU clustering research has highlighted the limitations of dense neighborhood representations. He et al. [He+22] proposed scalable algorithms using

sparse storage that construct the affinity matrix directly in CSR format on the GPU, eliminating the need for an $n \times n$ dense matrix. This allows clustering on millions of points using libraries such as cuSPARSE and nvGRAPH. However, their method assumes a fixed sparsity threshold and operates on high-dimensional feature vectors rather than spatially embedded 3D data.

2.8 Summary and Research Gap

Existing mesh simplification and neighbor-search approaches either rely on CPU computation or dense GPU data structures that are not memory-efficient. While strict-mode (S-Weld) and lock-free (P-Weld) algorithms achieve determinism and correctness on CPUs, their scalability is bounded by limited cores and bandwidth. Conversely, GPU frameworks such as FRNN and cuNSearch excel at parallel neighbor search but lack memory adaptivity. To the best of my knowledge, no prior GPU-based mesh clustering framework simultaneously achieves deterministic output equivalence, memory-efficient spatial indexing, and full GPU residency. I address this gap in this thesis by proposing a GPU-accelerated, sparse voxel-based vertex clustering framework that preserves the deterministic semantics of strict-mode clustering while adapting its grid representation to the available GPU memory for meshes like Lucy with more than 14 million vertices. The approach maintains exact CPU-equivalent neighborhoods, ensures safety against overflow, and achieves scalable performance across diverse mesh datasets.

Chapter 3

Serial and Parallel Strict-Mode Vertex Clustering

3.1 Overview

In this chapter, I will start by introducing the basic geometric concepts and terms that appear throughout this thesis, as described in Section 3.2. After covering these basics, I will explain the vertex clustering algorithm, which is one of the key parts of this research.

I will describe both the serial version, called S-Weld, and the parallel version P-Weld in Section 3.5 and Section 3.6 along with their corresponding pseudocodes. The parallel version P-Weld serves as the CPU baseline for the GPU implementations that are developed in later chapters. It includes a step-by-step example to help explain the strict-mode clustering process and to show how it works from both an algorithmic and architectural perspective. This discussion will help the reader to understand the proposed GPU-based pipelines.

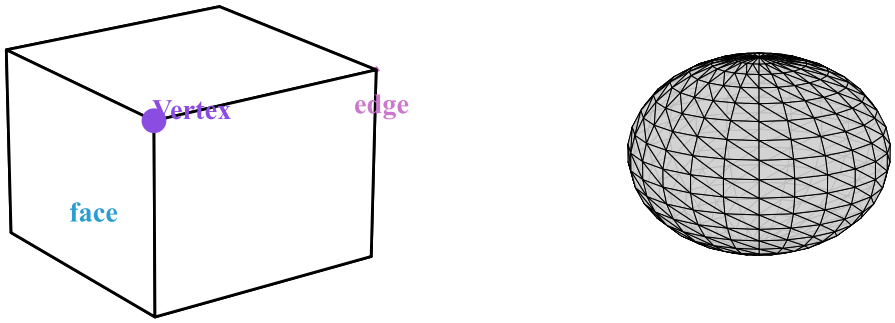
3.2 Terminology

A polyhedron, as depicted in Figure 3.1(a), is a three-dimensional shape made up of flat polygon faces and straight edges, and distinct corners, called vertices. You can think of it as a closed shape created by connecting flat polygons along their edges. Each face is a flat part of the surface, and where two faces meet, they form an edge. Polyhedra come in many different shapes, from simple ones like cubes and pyramids to more complex and non-convex shapes.

A mesh is a digital model made up of small pieces that together represent a smooth 3D surface. It uses vertices, edges, and faces to closely match the shape of an object. A mesh stores both the geometry, which gives the 3D position of each

vertex, and the connectivity, which shows how the vertices are linked to make edges and faces. The main reason for using a mesh is to create a model of real or imaginary shapes that computers can handle easily.

A triangle mesh, shown in Figure 3.1(b), is a type of mesh where every face is a triangle. These meshes are widely used in computer graphics, simulations, and geometry processing because of their topologically simple structure and their compatibility with modern graphics hardware. Triangles are inherently flat and defined by three vertices, which makes them efficient choice for computation and rendering. In a triangle mesh, adjacent faces often share the same vertices and edges. This reduces repetition and helps computers use memory more efficiently when searching.



(a) Polyhedron (b) Triangle mesh

Figure 3.1: (a) A polyhedron showing its basic elements: vertices, edges, and faces. (b) A triangle mesh representation of a 3D surface.

High-resolution triangle meshes can have millions of vertices and faces, which puts a lot of pressure on memory and processing power. While these meshes enable highly detailed visual representations, they also make rendering and geometric processing much more computationally expensive. To solve this problem, mesh reduction techniques are used to simplify models while keeping their overall shape and appearance. This kind of simplification is now an important research topic in computer graphics and geometric modelling. In this thesis, the vertex clustering is used as one such method. It works by merging nearby vertices into groups so that the resulting groups closely match the original surface.

3.3 Problem Statement

Let $M = (V, F)$ be a triangle mesh with vertex set $V \subset \mathbb{R}^3$, where each vertex $v \in V$ is associated with a position $p_v \in \mathbb{R}^3$ and a unique integer identifier $v.id$. Let $\varepsilon \geq 0$ denote a user-defined merge radius.

Definition 1 (ε-Neighborhood)**.** *For each vertex $v \in V$, the ε-neighborhood is defined as*

$$N_\varepsilon(v) = \{u \in V \mid \|p_u - p_v\| \leq \varepsilon\}.$$

The vertex clustering problem is to compute a representative mapping $R : V \to V$, where $R(v)$ denotes the representative vertex assigned to v, such that each vertex is mapped to the smallest-indexed vertex reachable via transitive ε-neighborhood relations.

This mapping is defined recursively through a sequence of vertices. For $i \geq 0$, let

$$N_\varepsilon^-(v) = \{u \in N_\varepsilon(v) \mid u.id < v.id\}.$$

The sequence is then defined as

$$R(v) = \begin{cases} 0, & \text{if } v.id = 0, \\ \arg \min\limits_{u \in \{v' \in N_\varepsilon^-(v) \mid R(v') = v'\} \cup \{v\}} u.id, & \text{otherwise.} \end{cases}$$

The mapping R induces a partition of V into disjoint clusters, where each cluster consists of vertices sharing the same representative. Each cluster is replaced by a single vertex (e.g., its centroid), and the mesh connectivity is updated accordingly.

The objective of this work is to accelerate computing the mapping R massively in parallel on the GPU.

3.4 Mesh Simplification Pipeline

The objective of mesh simplification is to reduce this complexity while still keeping the original shape and appearance intact as much as possible. The mesh simplification process usually involves five main steps:

1. calculating geometric importance values (*grading*);
2. converting the polygon model into triangles (*triangulation*);
3. grouping nearby vertices within an ε-distance (*clustering*);
4. replacing each group with a single representative point (*synthesis*); and
5. removing duplicate or invalid faces (*elimination*).

In this thesis, I treat the first two steps as preprocessing stages, similar to the approach taken by Fathollahi and Chester [FC23]. The focus is on the remaining stages, particularly the clustering and synthesis phases that follow. These steps were redesigned for efficient GPU execution.

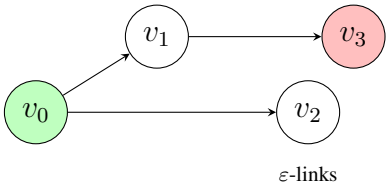


Figure 3.2: Directed ε-adjacency graph showing active and merged vertices. Green marks the active centroid, red a merged vertex, and arrows indicate directed ε-links between nearby vertices.

Figure 3.2 shows the initial ε-adjacency graph, where edges represent proximity relationships between vertices. The main purpose of clustering is to group vertices into disjoint subsets. Each group is then replaced by its centroid, producing a simplified mesh.

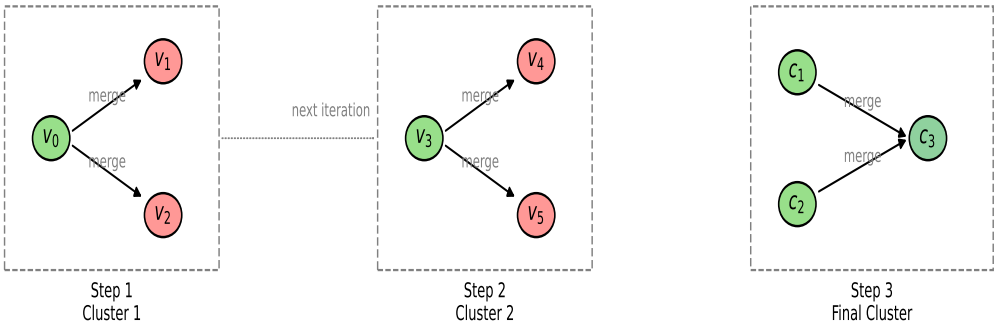


Figure 3.3: Stepwise progression of vertex clustering in an ε-adjacency graph. Step 1: v_0 merges (v_1, v_2); Step 2: v_3 merges (v_4, v_5); Step 3: centroids (c_1, c_2) merge into c_3 forming the final cluster. Green nodes show active centroids, red nodes show merged vertices, and dashed boxes indicate cluster boundaries.

Figure 3.3 demonstrates the stepwise progression of vertex merging, showing how local merges progressively form larger clusters until convergence. Green ver-

tices denote active centroids, while red vertices indicate merged points that have already been assigned to a cluster.

Building on this classical pipeline, Fathollahi and Chester [FC23] extended the serial vertex clustering to a lock-free multicore algorithm. It preserves this behaviour while enabling concurrent execution on modern central processing units using multithreading. He utilized the KDTree FLANN structure in his implementation from the Open3D library, which internally relies on the lightweight `nanoflann` library [BR14] for efficient ε-range queries. KD-tree traversal exhibits irregular memory access patterns that limit GPU efficiency. These limitations motivated me to redesign of the spatial phase in this research, resulting in a fully GPU-resident sparse voxel grid structure that performs neighborhood construction, clustering, and synthesis directly on the device. This approach eliminates costly host–device synchronization and exploits the GPU's fine-grained parallelism to make the clustering process faster. The voxel-grid representation divides space into uniform 3D cells, enabling efficient neighbor lookups through integer hashing rather than recursive tree traversal. This structure naturally aligns with GPU memory access patterns and can scale to large meshes within the constrained VRAM of standard hardware.

3.5 Serial Algorithm (S-Weld)

This section describes the S-Weld algorithm 1, which forms the deterministic baseline of the clustering process. The algorithm processes each vertex sequentially, based on its unique identifier rather than its geometric position. This approach ensures that only one vertex is active at a time. When a vertex is active, it either joins an existing cluster or starts a new one if it has not been assigned to one yet. Because the process is strictly ordered and single-threaded, the final result is fully deterministic and independent of hardware. At first, the S-Weld constructs an ε-spatial graph represented as an adjacency list for each vertex by finding all neighbours within distance ε, using a parallel KD-tree based preprocessing. Then in the clustering process, at each iteration t, the active vertex u_t scans its adjacency list and merges all unclustered neighbors v within the distance ε. Each merged vertex acquires u_t as its representative and is marked as clustered. After all vertices have been visited, cluster centroids are computed as the mean position of their members, and mesh triangles are remapped using the new vertex identifiers.

Algorithm 1 S-Weld: Serial clustering [FC23]

Input: Mesh vertices V, triangles T, distance threshold ε
Output: Reduced vertex set $\bar{V}$ and transformed triangles $\bar{T}$

for $u \in V$ **do**
    $\texttt{adjList}[u] \leftarrow \{\, v \in V \mid \|p_v - p_u\| \leq \varepsilon \,\}$
end for
for $u = 1$ **to** $|V|$ **do**
    if u is not clustered **then**
        mark u as clustered with a new cluster ID
        $\texttt{represent} \leftarrow u$
        for $v \in \texttt{adjList}[u]$ **do**
            if v is not clustered **then**
                mark v as clustered with `represent`
            end if
        end for
        compute centroid of cluster `represent` and append to $\bar{V}$
    end if
end for
for $t \in T$ **do**
    replace vertex IDs of t with their representatives
end for

3.6 Parallel Strict-Mode Algorithm (P-Weld)

This section describes the novel contribution of P-Weld in mesh simplification field by Fathollahi. As I discussed, S-Weld guarantees determinism, but its serial execution limits scalability on modern multicore systems. So to make clustering faster by accelerating the multithreading in the CPU, Fathollahi introduced the novel P-Weld algorithm, which extends S-Weld to support lock-free parallelism and keeps the clustering output identical by using a dependency-driven topological processing order.

3.6.1 Motivation and Design Reasoning

It is really difficult to parallelize serial clustering logic due to its inherent data dependencies; e.g., processing a vertex is permitted only after all higher-indexed neighboring vertices have been merged. Traditional synchronization mechanisms, such as locks and critical sections, serialize updates and thereby restrict parallelism. Therefore, these approaches diminish the potential for effective parallel processing. To cope with these issues, P-Weld uses a lock-free methodology utilizing atomic operations to guarantee correctness without introducing race conditions. Multiple threads operate independently in this approach, requiring synchronization solely through atomic primitives. Each thread processes a distinct subset of vertices and coordinates with other threads exclusively via shared atomic counters. This technique significantly reduces the necessity for complex synchronization mechanisms; its core concept is that the activation order of vertices can be modeled as a dependency graph. Within this graph, directed edges specify which vertices must complete processing before others may begin. All edges are directed from vertices with smaller to larger identifiers, ensuring the graph remains acyclic by construction. Parallel processing is achievable provided that the execution order sticks to these dependency constraints.

3.6.2 Dependency Graph Construction

As mentioned earlier, P-Weld transforms the sequential vertex traversal into a *dependency graph* defined over the spatial adjacency structure. For each vertex u, the outgoing dependencies are constructed as:

$$\texttt{adjList}[u] \leftarrow \{\, v \in V \mid \|p_u - p_v\| \le \varepsilon \ \wedge \ u < v \,\},$$

$$\texttt{depend}[u] \leftarrow \left|\{\, v \in V \mid \|p_u - p_v\| \le \varepsilon \ \wedge \ v < u \,\}\right|, \quad \texttt{centroid}[u] \leftarrow u.$$

Here, $\texttt{depend}[u]$ represents the number of lower-indexed neighbors that must be processed before u becomes active. Vertices with $\texttt{depend}[u] = 0$ form the initial frontier and are eligible for concurrent processing.

Algorithm 2 P-Weld: Parallel strict-mode vertex clustering[FC23]

Input: V: vertices ordered by grade, T: triangles, $\varepsilon \geq 0$
Output: reduced vertex set, transformed triangles

1: **for** $u \in V$ **in parallel with spatial index do**
2: `adjList`$[u] \leftarrow \{\, v \in V \mid \|p_u - p_v\| \leq \varepsilon \ \wedge \ u < v \,\}$
3: `depend`$[u] \leftarrow |\{\, v \in V \mid \|p_u - p_v\| \leq \varepsilon \ \wedge \ v < u \,\}|$ ▷ in-degree
4: `centroid`$[u] \leftarrow u$
5: **end for**
6: `shouldContinue` $\leftarrow$ `true`
7: **while** `shouldContinue` **do** ▷ i.e., at least one thread votes not to halt
8: `syncBarrier`
9: `shouldContinue` $\leftarrow$ `false` ▷ thread-local copy reduced on Line 5
10: **for** $u \in V$ **in parallel do**
11: **if** `depend`$[u] = 0$, i.e., u is now a source **then**
12: decrement `depend`$[u]$ to mark it as done
13: **for** $v \in$ `adjList`$[u]$ **do**
14: **if** `centroid`$[v] = v$, i.e., v is a centroid **then**
15: `centroid`$[v] \leftarrow \min($`centroid`$[v], u)$ **via CAS**
16: atomically decrement `depend`$[v]$
17: `shouldContinue` $\leftarrow$ `true`
18: **end if**
19: **end for**
20: **end if**
21: **end for**
22: **end while**
23: **for** $u \in V$ **in parallel do**
24: **if** u is a centroid **then**
25: Create a cluster and assign it an auto-increment ID
26: Add u to the cluster created by its `centroid`$[u]$
27: **end if**
28: **end for**
29: **for** $c \in$ clusters **in parallel do**
30: Calculate the representative and append it to `newVertices`
31: **end for**
32: **for** $t \in T$ **in parallel do**
33: replace vertex IDs of t with their new representatives' IDs
34: **end for**

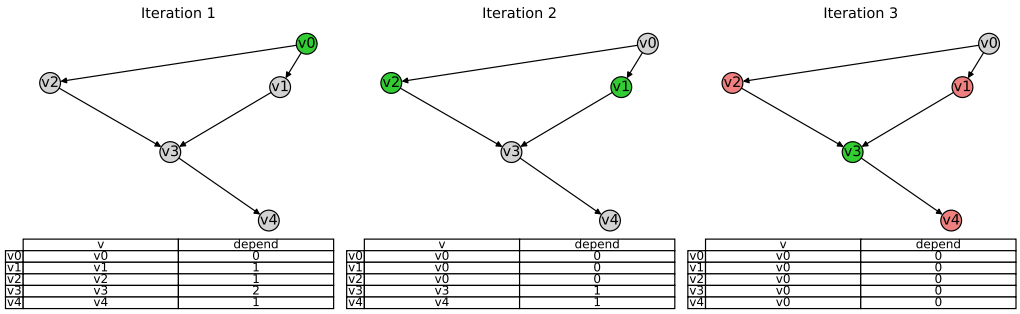


Figure 3.4: Execution trace of P-Weld on a toy 1D point set with ε-neighborhood. Each panel shows one iteration. Green vertices are active (`depend`=0 and self-centroid), and arrows denote directed edges $u \to v$ ($u < v$). Tables track the `centroid` and `depend` arrays. The process converges in three iterations, producing clustering identical to S-Weld.

Because edges are directed from lower to higher vertex identifiers, the resulting dependency graph is acyclic by construction. This guarantees a valid activation order and ensures deterministic convergence of the clustering process.

3.6.3 Parallel Activation and Atomic Updates

In each iteration of P-Weld, there are three main steps:

1. Active frontier: Threads loop through all vertices and activate any with $\texttt{depend}[u] = 0$ and $\texttt{centroid}[u] = u$. These vertices have no unresolved dependencies, so they can become new cluster centroids safely.
2. Atomic Cluster Merging: For each active vertex u, the algorithm checks its adjacency list $\texttt{adjList}[u]$. For each neighbor v:
   - If u is still its own representative ($\texttt{centroid}[u] = u$), attempt to merge v by executing

     $$\texttt{atomicCAS}(\texttt{centroid}[v], v, u),$$

     making sure the smaller index always wins.
   - Whether or not the merge works, atomically decrease $\texttt{depend}[v]$ to show that one of its lower-index neighbors has finished. When $\texttt{depend}[v]$ reaches zero, v is added to the next frontier.
3. Convergence Check: A shared Boolean flag called `changed` keeps track of whether any updates happened during the step. If no thread reports a change, the process has finished and the clustering is marked as complete.

This atomic update method keeps three important properties:

- Determinism: since smaller identifiers always dominate, the final mapping does not depend on how the operating system schedules the threads
- Lock-free Safety: atomic operations eliminate race conditions and do not block other threads
- Progress Guarantee: dependency counters strictly decrease, ensuring eventual activation of all vertices.

3.6.4 Finalization and Output

Finally, each vertex points to its cluster representative in `centroid` after convergence. Vertices with $\texttt{centroid}[u] = u$ form the set of unique clusters. The position of each cluster's representative is then found as the average of its member vertices, which gives the reduced vertex set $\bar{V}$. At the end, all triangles are remapped using these new vertex indices, and any duplicate or degenerate faces are removed to create the simplified mesh. Figure 3.4 illustrates the iteration-wise progression of P-Weld. Initially, only v_0 has $\texttt{depend} = 0$ and activates, claiming v_1 and v_2 via CAS and reducing their counters. In iteration 2, v_1 and v_2 activate concurrently, propagating updates to v_3. Finally, v_3 merges with v_4. The algorithm converges when all dependencies are resolved. Despite parallel execution, P-Weld produces bit-identical clustering to S-Weld due to strict index-based priority and atomic enforcement.

Both algorithms [FC23] maintain a monotonically decreasing mapping in `centroid` through sequential or atomic compare-and-swap updates. In P-Weld, a vertex is only active when $\texttt{depend}[u] = 0$, which means all of its lower-indexed neighbors have already been processed. Since the dependency graph is finite and has no cycles, every vertex will eventually settle on its final representative. Because atomic updates are linearizable, the final configuration does not depend on the order in which threads run, which guarantees it is equivalent to S-Weld.

Summary

This chapter recalled the serial S-Weld and parallel P-Weld algorithms, which together form the CPU foundation of GPU-based clustering framework. S-Weld enforces a total merge order that guarantees reproducibility but processes sequentially. P-Weld introduces atomic synchronization to exploit parallelism while preserving the deterministic results of S-Weld. The above theorem provides the theoretical foundation for all subsequent GPU versions. Each GPU variant developed in the following chapters inherits the deterministic semantics of P-Weld while leveraging more fine-grained parallelism on the GPU.

Chapter 4

First Proposed Solution: Baseline On-the-Fly GPU

4.1 Overview

In this chapter, I present the first deterministic GPU port version of the P-Weld algorithm. My main goal was to reproduce the exact semantics of the CPU implementation while adapting its dependency-driven logic to a massively parallel CUDA environment. This GPU version, referred to as the ”On-the-fly” version, performs both neighbourhood search and clustering entirely on the device. In this approach, I remove the need to precompute neighbour lists, performing almost all computations directly on the GPU. To maximize the use of the GPU, only minimal grid indexing is handled by the host. I maintained the deterministic merge order by using dependency counters and atomic operations in my implementation.

4.2 Motivation and Conceptual Parallels

As I observed, the P-Weld algorithm used multicore CPU parallelism using OpenMP, the thread count and synchronization overhead are still constraints for scalability. So to extend this CPU-based logic to device, the recursive KD-tree traversal is needed to be replaced with a voxel-based spatial grid that supports fully concurrent lookups and fine-grained dependency updates across thousands of threads. Furthermore, Blelloch's parallel prefix-sum design [Ble90], which decomposes inherently sequential dependencies into associative parallel steps, inspired me to break down dependency propagation and merge decisions into separate kernel passes connected through global memory atomics on GPU. Each kernel performs associative updates independently, ensuring deterministic convergence while exploiting massive parallelism.

GPU threads follow a SIMT execution model, where threads within a warp share instruction scheduling but may progress independently.This is different from CPU threads. so I had to rely on hardware-supported atomic operations for coordination. My challenge was to design deterministic device functions that maintain parent–child relationships while avoiding race conditions or divergent execution paths.

I also explored several stages using the Thrust library [NVI23] to understand how GPU primitives such as sorting, reduction, and compaction could support dependency-driven clustering before implementing the full clustering kernel. These experiments revealed that scalable GPU implementations must rely on associative updates, such as atomic additions during dependency counting and reductions during centroid averaging, instead of explicit global synchronization barriers. The deterministic accumulation pattern in `thrust::reduce_by_key` inspired me to use it for the centroid averaging and vertex remapping logic integrated into the On-the-fly implementation.

4.3 Data Structures and Memory Layout

In this version, I implement a lightweight voxel hash grid that resides entirely in device memory. The grid structure itself is constructed once on the host to assign vertices to voxel cells and then transferred to the GPU. Each vertex is assigned to a voxel cell based on its spatial position and a uniform grid resolution determined by the merge radius ε. The voxel grid serves as the core spatial index for fast neighborhood queries after transfer. All further operations dependency initialization, clustering, and centroid averaging are performed entirely on the device without further PCIe data movement.

Each voxel cell is identified by its integer grid coordinate (i_x, i_y, i_z) and converted into a unique hash key using large prime multipliers to minimize collisions and ensure reproducible lookups [Tes+03]:

$$\text{cell\_key}(i_x, i_y, i_z) = i_x \times 73856093 \oplus i_y \times 19349663 \oplus i_z \times 83492791$$

The hash function is implemented using multiplicative hashing with large prime constants combined through bitwise XOR operations. This formulation is computationally efficient on GPU hardware, as it relies solely on integer arithmetic and avoids complex control flow. The use of distinct prime multipliers helps decorrelate structured grid coordinates, thereby reducing systematic collisions when mapping voxel indices to hash keys. In particular, spatially adjacent cells that differ by only one coordinate typically produce significantly different hash values, reducing the likelihood of clustering in the hash table. The hash function maps voxel cell

indices to keys that are approximately uniformly distributed over the key space, reducing the likelihood of collisions during concurrent access.

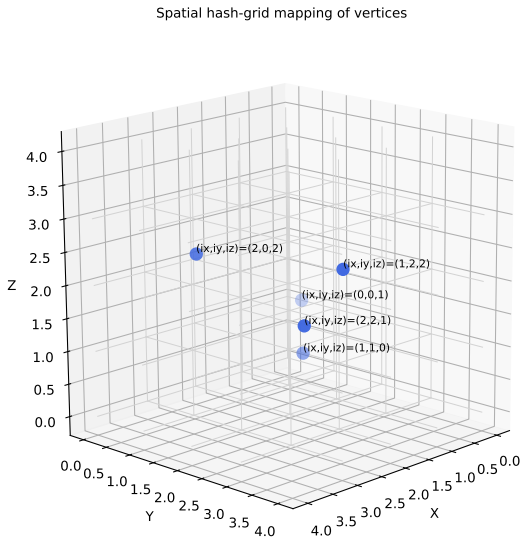


Figure 4.1: Spatial discretization of vertex positions into a uniform 3D voxel grid. Each occupied cell (i_x, i_y, i_z) is converted into a unique integer key using the hash function above.

Each vertex is stored in global memory as a structure containing:

- Position vector (x, y, z);
- Dependency counter $D[v]$, representing the number of unresolved smaller neighbors;
- Parent identifier $C[v]$, initialized to v;
- Cell index (i_x, i_y, i_z) and hash key.

I choose this layout to ensure coalesced memory access during spatial neighbourhood queries and avoid irregular pointer dereferencing that made CPU adjacency lists inefficient. All data required for clustering remains in GPU global memory, which allows each thread to process vertices independently using a grid-stride loop. Because spatially close vertices are mapped to neighboring voxel cells, neighbor queries are restricted to a small and fixed set of adjacent cells. This structured access pattern improves memory coherence during traversal, even though the hash function itself does not preserve spatial ordering.

The voxel grid differs from the KD-tree based neighbor search used in the PWeld and the precomputed CSR adjacency of the later sparse-grid implementation. In this design, all spatial lookups are performed on the fly inside the clustering kernels, eliminating the need for repeated host preprocessing or intermediate PCIe

data movement after the initial grid transfer.

4.4 GPU Memory Organization and Synchronization

In this implementation, I organize the computation around three major global memory buffers indicated as dependency counters (depend), the centroid mapping array `cp_vec`, and the centroid accumulation buffers used for averaging. All these arrays are accessed concurrently by thousands of threads during clustering. I used CUDA atomic operations such as `atomicCAS`, `atomicAdd`, and `atomicSub`, which ensure safe concurrent updates to shared memory locations to maintain correctness.

Each thread executes in grid-stride loops, allowing the workload to scale with any input size. I used `cudaDeviceSynchronize()` to enforce global consistency by making the host wait for all GPU threads to finish before launching the next kernel. This per-iteration synchronization guaranteed that all dependency updates became globally visible while still maintaining fine-grained parallelism within each kernel.

To further improve performance, vertex processing is organized based on voxel cell identifiers, which are mapped to hash keys for indexing. While the hash function itself does not preserve spatial locality, vertices belonging to the same voxel share identical keys and are therefore processed together. This grouping enables efficient access to local neighborhoods and contributes to improved cache utilization during clustering. Since vertices are reordered according to their cell keys during grid construction, those belonging to the same voxel become contiguous in memory, further improving memory access coherence during neighbor traversal.

4.5 Algorithmic Framework

I preserve the core logic of the P-Weld in my On-the-fly algorithm while executing all stages in parallel on the GPU. The entire workflow is structured into three main stages, including dependency initialization, strict-mode clustering, and centroid averaging with face remapping.

Algorithm 3 GPU On-the-Fly Clustering Framework

Input: vertex set V, triangle list F, merge radius ε
Construct voxel hash grid on host:
    $(\text{cell\_keys}, \text{cell\_offsets}, \text{cell\_points}) \leftarrow$ `build_hash_grid_cpu`(V, ε)
Transfer all vertex and grid buffers to GPU global memory
Initialize dependency counters on device via
    `streaming_init_gpu_depend_kernel`
repeat
    Perform strict-mode clustering iteration using
    `streaming_iteration_hash_kernel`
    Synchronize host using `cudaDeviceSynchronize()`
until no active dependencies remain
Compute centroids via `thrust::reduce_by_key`
Remap and compact faces via `remap_faces_kernel_flags`
Output: Simplified mesh (V', F')

I present a comprehensive, device-resident pipeline for strict-mode mesh simplification (Algorithm 3), exploring the GPU parallelism to achieve real-time performance and scalability. The process begins with the construction of a voxel hash grid on the host, which spatially partitions the input vertex set V into voxel cells based on the merge radius ε. This grid, consisting of $(\text{cell\_keys}, \text{cell\_offsets}, \text{cell\_points})$, is subsequently transferred to the GPU to facilitate efficient neighborhood queries during the clustering process. Once the data resides on the GPU, the algorithm initializes dependency counters for all vertices in parallel using the `streaming_init_gpu_depend_kernel`.This kernel ensures that each vertex's dependencies are accurately recorded by querying the voxel hash grid for neighboring vertices within the merge radius ε. Atomic operations are used to guarantee thread-safe updates to the dependency counters during this initialization phase.

The core of this pipeline is an iterative strict-mode clustering loop, executed by the `streaming_iteration_hash_kernel`, which continues until no active dependencies remain. In each iteration, vertices with zero unresolved dependencies ($D[u] = 0$) are activated and processed in parallel across GPU threads. For each active vertex u, the thread retrieves all neighboring vertices v within ε and attempts to merge them into u's cluster via an atomic compare-and-swap (`atomicCAS`) operation. This ensures that the vertex with the smallest index always becomes the centroid, preserving the strict-mode semantics. Furthermore, the dependency counters are decremented atomically, and the vertices whose dependencies reach 0 are marked for activation in the next iteration. The loop ends when all vertices have been given a final cluster identifier, with a global device flag (`d_changed`) moni-

toring convergence.

After convergence, the algorithm computes cluster centroids using a parallel reduction via `thrust::reduce_by_key`, aggregating vertex positions and counts for each cluster to derive the centroid positions as the mean of all member vertices. The final stage remaps the input triangle or face list F to the simplified vertex set V' using the `remap_faces_kernel_flags` kernel, followed by a parallel compaction pass (`thrust::copy_if`) that removes degenerate or duplicate faces. The output is the simplified mesh (V', F'), preserving the geometric shape of the input while significantly reducing vertex and face count.

4.5.1 On-the-Fly Dependency Initialization

The dependency initialisation kernel (Algorithm 4) is equivalent to the CPU P-Weld's adjacency list population phase, except it uses voxel-based neighborhood lookups in GPU memory instead of KD-Tree range searches. In lines 3–5 each thread is assigned a vertex v and determines its voxel coordinates (i_x, i_y, i_z) using the uniform grid resolution derived from the merge radius ε. This cell index serves as the spatial reference point for following neighbour searches. In lines 6-9 the kernel iterates over the $3 \times 3 \times 3$ neighborhood surrounding each cell.A hash key h is computed using the spatial hashing function described earlier in Section 4 for every neighboring offset $(\Delta x, \Delta y, \Delta z)$. The vertices u stored in the corresponding hashed cell are retrieved in line 10-17 and their Euclidean distance to v is evaluated. If $u < v$ and $\|V[u] - V[v]\| < \varepsilon$, the dependency counter $D[v]$ is incremented atomically. This operation ensures that each vertex tracks how many smaller-indexed neighbors must be processed before it can participate in clustering. In P-Weld, dependency relationships are precomputed using KD-Tree spatial queries to construct adjacency lists before the clustering loop begins.On the other hand, the GPU version reconstructs these relationships on-the-fly during kernel execution using a voxel hash grid, eliminating the need for static adjacency lists. The final dependency array stores unresolved smaller neighbors of each vertex, enabling synchronization-free clustering in later phases. Despite being memory-bound due to repeated spatial queries, the kernel achieves efficient parallel scaling because dependency updates are spatially localized, minimizing contention between threads

Algorithm 4 On-the-Fly Dependency Initialization Kernel

1: **Input:** vertex array V, voxel grid (cell_keys, cell_offsets, cell_points), merge radius ε
2: **Output:** dependency array D
3: **for** each vertex v in parallel **do**
4: $D[v] \leftarrow 0$
5: Determine grid coordinates (i_x, i_y, i_z) for v
6: **for** each offset $(\Delta x, \Delta y, \Delta z)$ in the $3 \times 3 \times 3$ neighborhood **do**
7: Compute neighbor key $h = \text{hash}(i_x + \Delta x, i_y + \Delta y, i_z + \Delta z)$
8: Retrieve neighbor cell index using `hash_lookup(`h`)`
9: **if** cell is valid **then**
10: **for** each vertex u in that cell **do**
11: **if** $u < v$ and $\|V[u] - V[v]\| < \varepsilon$ **then**
12: `atomicAdd(`$D[v]$`, 1)`
13: **end if**
14: **end for**
15: **end if**
16: **end for**
17: **end for**

4.5.2 On-the-Fly Clustering Iterations

Algorithm 5 presents the iterative clustering kernel, which executes the core parallel merging loop of the On-the-fly version. This phase mirrors the dependency-driven propagation loop of the P-Weld algorithm but redesigns its synchronization model for SIMT execution using atomic primitives. In Line 3, each thread is assigned a subset of vertices in a grid-stride configuration, allowing arbitrarily large meshes to be processed efficiently regardless of GPU thread count. The kernel on lines 4–6 identifies vertices whose dependency counters have reached zero, and marks them as active frontiers ready for merging. Each such vertex is marked as processed ($D[u] = -1$) and temporarily considered a centroid candidate ($C[u] = u$). This makes sure that only vertices without dependencies are included in the current iteration.

In lines 7-14, each active vertex calculates its voxel coordinates and scans its local $3 \times 3 \times 3$ neighborhood. The kernel uses hash lookups to get vertices from nearby cells and checks if the distance between u and v is within the merge radius ε. If both conditions are met and the neighbor has not been merged yet ($C[v] = v$), the algorithm starts a deterministic merge attempt. Lines 15-22 perform this operation with an atomic compare-and-swap (`atomicCAS`), making sure that only one thread can claim ownership of a vertex at any moment.This atomic opera-

tion enforces P-Weld's strict-mode rule, where smaller identifiers control merges. Each time a merge succeeds, the dependency counter for the affected vertex is reduced using `atomicSub` on line 23-28. When $D[v]$ reaches zero, the vertex becomes active in the next iteration, forming a new frontier ready for merging. This lock-free propagation method replaces CPU-side shared synchronization, enabling thousands of threads to progress independently without conflicts. Finally, lines 29–34 record whether any thread performed a merge by setting a Boolean flag in `changed_flags`. After the kernel finishes, the host synchronizes and aggregates these flags to check convergence; if no vertex changes occur, it terminates the clustering loop. Otherwise, the process continues with another iteration. This kernel uses a fully deterministic, lock-free clustering method that matches the P-Weld algorithm's behavior. The GPU version uses atomicCAS-based ownership resolution and dependency-driven activation. This approach provides significant parallel speedups while guaranteeing the same centroid mappings, ensuring reproducible results across different runs and hardware.

Algorithm 5 On-the-Fly Strict-Mode Clustering Kernel

Input: vertex set V, dependency counters D, parent array C, merge radius ε and spatial hash-grid structure $\mathcal{H}$ for neighbor queries
Output: updated C, D, and per-thread change flags
for each thread tid in parallel **do**
  $local\_changed \leftarrow$ false
  for each vertex u assigned to thread tid in grid-stride fashion **do**
    if $D[u] = 0$ **then** ▷ vertex ready for merging
      $D[u] \leftarrow -1$; $isCentroid \leftarrow (C[u] == u)$
      Compute voxel coordinates (i_x, i_y, i_z) of u
      for $\Delta x, \Delta y, \Delta z \in \{-1, 0, 1\}$ **do** ▷ 27-cell neighborhood
        $key \leftarrow$ cell_key$(i_x + \Delta x, i_y + \Delta y, i_z + \Delta z)$
        $idx \leftarrow$ hash_lookup$(key, \mathcal{H})$
        if $idx \geq 0$ **then**
          for each vertex v in cell idx **do**
            if $u \neq v$ and $\|V[u] - V[v]\| < \varepsilon$ **then**
              if $isCentroid$ and $C[v] = v$ **then** ▷ merge smaller neighbor
                $expected \leftarrow C[v]$
                while $u < expected$ **do**
                  $old \leftarrow$ atomicCAS$(C[v], expected, u)$
                  if $old = expected$ **then break**
                  end if
                  $expected \leftarrow old$
                end while
              end if
              if atomicSub$(D[v], 1) = 1$ **then**
                $local\_changed \leftarrow$ true
              end if
            end if
          end for
        end if
      end for
    end if
  end for
  `changed_flags`$[tid] \leftarrow local\_changed$
end for

4.5.3 Centroid Averaging and Face Remapping

The last kernel stage, described in Algorithm 6, calculates the average centroids for all remaining clusters and updates the mesh connectivity. Lines 3-5 create the cluster keys and combine vertex coordinates using `thrust::reduce_by_key`. This function adds together all vertices that share the same parent ID. The sums are then divided by the number of members to get the final centroid positions V'. This replaces the P-Weld centroid averaging loop with a fully parallel reduction, which produces deterministic results because Thrust uses a fixed reduction order. Lines 6-7 handle face remapping and compaction; each triangle's vertex IDs are updated to their cluster representatives $(C[F[j].x], C[F[j].y], C[F[j].z])$. Moreover, faces where two or more vertices are the same are removed using a parallel compaction step. This ensures the output mesh remains topologically valid. This stage ensures centroid computation is consistent and repeatable across runs by using data-parallel reductions and compaction and achieves substantial performance gains.

Algorithm 6 Centroid Averaging and Face Remapping

1: **Input:** vertex positions V, parent mapping C, face list F
2: **Output:** reduced vertex set V', remapped face list F'
3: Derive cluster keys: $K[i] \leftarrow C[i]$
4: Aggregate centroids: $(K', S) \leftarrow \texttt{reduce\_by\_key}(K, V)$
5: Normalize: $V'[k] \leftarrow S[k]/|S[k]|$
6: Remap faces: $F'[j] \leftarrow (C[F[j].x], C[F[j].y], C[F[j].z])$
7: Compact: remove degenerate faces where any vertex indices coincide

4.6 Work Efficiency and Parallel Behaviour

In this section, I analyze both the computational complexity and its parallel efficiency to understand the scaling behaviour of my GPU implementation. Let n denote the total number of vertices and k represent the average number of neighbours per vertex within the merge radius ε.

Each clustering iteration performs $O(nk)$ distance checks, but the actual number of atomic updates is typically an order of magnitude smaller, since only a fraction of vertices become active per iteration. In practice, the number of active vertices decreases exponentially as merges progress, which significantly reduces work in later iterations. The total runtime therefore scales approximately as $O(nk/p)$, where p denotes the number of concurrently executing threads. Since p can reach tens of thousands on modern GPUs, the algorithm achieves strong parallel speedups, particularly for large meshes, as observed in my experiments

presented in the empirical validation chapter. The number of iterations depends on the longest dependency chain in the mesh graph, which tends to be very small compared to n due to the local nature of geometric adjacency. It is also observed that the implementation is primarily memory-bound rather than compute-bound. The profiling results confirmed that the majority of the execution time is spent on global memory accesses rather than arithmetic operations. However, the spatial sorting of vertices by hash key, along with the coalesced access pattern within each voxel neighborhood, helps alleviate this limitation. The result is a nearly linear improvement in throughput as the number of streaming multiprocessors increases. Warp-level behavior also remained stable across datasets. While the use of atomics introduces warp divergence (since only one thread succeeds in a `atomicCAS` at a time), this divergence does not accumulate significantly. The strict dependency model ensures that only ready vertices (those with $D[v] = 0$) participate in merges during any iteration, which naturally throttles unnecessary contention.

Consequently, the on-the-fly version still shows some underutilization in later iterations because both the dependency-update and clustering kernels continue to run spatial queries for vertices that have already converged. This repeated neighbourhood traversal increases global memory pressure and causes warp idling as the frontier size decreases. Thus, while overall throughput scales well with the available SMs, GPU utilization drops a bit as the process nears convergence. This finding motivated me to propose the fully GPU-resident sparse-grid method described in the next chapter. This method caches voxel adjacency, eliminating the need for repeated spatial lookups in each iteration.

4.7 Proof of correctness

One of the main design goal of my On-the-Fly GPU implementation was to ensure that, despite thread-level parallelism, the clustering process converges deterministically to the same result as the P-Weld algorithm. In this section, I provide a conceptual explanation and formal justification for this property. The proof builds upon three core concepts: (1) monotonic atomic updates, (2) finite dependency propagation, and (3) global convergence toward a stable configuration.

Conceptually, the algorithm behaves like a distributed system of local agents (vertices). Each one decides to merge only after its smaller neighbours have finished. The use of atomic operations ensures that all updates respect a consistent ordering, that smaller identifiers always dominate, and that no vertex can reverse a merge, helping avoid race conditions and ensuring a consistent global outcome.

4.7.1 Monotonic Parent Updates

Every vertex v stores a parent identifier $C[v]$, which either refers to itself (if it is a centroid) or to another vertex with a smaller index. I enforced this property in the clustering kernel using an atomic operation. Whenever two vertices u and v are within the merge radius ε, the smaller one atomically updates the larger's parent pointer:

$$C[v] \leftarrow \min(C[v], C[u])$$

This guarantees that parent identifiers always decrease or remain constant across iterations.

Lemma 4.7.1 (Monotonic Parent Update)**.** *For all vertices v, $C[v]_{t+1} \leq C[v]_t$ during every clustering iteration.*

Proof. The `atomicCAS` operation only replaces $C[v]$ if the proposed value $C[u]$ is smaller than the current one. Since vertex identifiers are globally unique and ordered, no thread can increase $C[v]$. This defines a monotonic non-increasing sequence that must eventually stabilize. □

4.7.2 Finite Dependency Resolution

Each vertex has a dependency counter $D[v]$, which tracks how many smaller neighbors must complete their merge before v can become active. These counters ensure that merges follow a dependency-based processing order, just as in P-Weld's dependency logic [FC23].

Lemma 4.7.2 (Bounded Dependency Decrease)**.** *The dependency counter $D[v]$ always decreases over time and never becomes negative.*

Proof. The algorithm performs one or more `atomicSub` operations to reduce the dependency counter of neighboring vertices. Since `atomicSub` can only decrease a value and cannot make it go below zero, and no part of the algorithm ever increases $D[v]$, every dependency counter moves in a downward direction. Starting from a finite positive value, every counter eventually reaches zero, at which point the corresponding vertex becomes active and participates in the clustering process. □

Together, these two lemmas imply that all vertices eventually stop changing both their parent identifiers and their dependency values, meaning no further updates or merges are possible.

4.7.3 Global Convergence to a Fixed Point

I define the global clustering state $S = (C, D)$, where C represents the parent (centroid) mapping and D represents the dependency counters, to demonstrate the convergence of the clustering process. In each kernel step, an update function f changes the current state S_t into a new state S_{t+1}:

$$S_{t+1} = f(S_t)$$

Since every atomic operation either lowers or keeps the values of $C[v]$ and $D[v]$, and these values are within a finite range, the sequence S_t will eventually reach a stable state. This stable clustering state is a fixed point S^*, where $S_{t+1} = S_t = S^*$. At this point, there are no more parent or dependency updates.

Lemma 4.7.3 (Finite Convergence)**.** *The On-the-Fly GPU clustering algorithm reaches a stable clustering state after a finite number of iterations.*

Proof. Each vertex carries out only a limited number of `atomicCAS` and `atomicSub` operations. Both operations result in integer sequences that do not increase and are limited to certain ranges. `atomicCAS` assigns only smaller parent identifiers, while `atomicSub` decrements counters until they reach zero. As a result, there cannot be an endless sequence of updates. Once all vertices have $D[v] = 0$, the mapping from each vertex to its centroid, $C[v]$, does not change anymore, which means the algorithm has reached a stable clustering state. □

Lemma 4.7.4 (Determinism of Clustering)**.** *Given the same input (V, F, ε), the On-the-Fly GPU clustering algorithm always produces an identical centroid mapping $C[v]$, ensuring full deterministic output across various runs.*

Proof. atomic operations behave as if executed in a consistent serial order that aligns with program semantics, meaning they are *linearizable*. Since `atomicCAS` only replaces a parent identifier with a smaller value, all updates converge toward a unique minimal parent assignment. This update rule is independent of thread scheduling, ensuring that each execution results in the same final clustering configuration. □

As observed, the centroid averaging phase is also deterministic because `thrust::reduce_by_key` operation combines vertex positions within sorted cluster keys, guaranteeing that the reduction sequence remains identical across executions.

Lemma 4.7.5 (Determinism of Centroid Averaging)**.** *For each cluster i, the computed centroid*

$$\mathbf{c}_i = \frac{1}{|M_i|} \sum_{v \in M_i} \mathbf{p}_v,$$

where $M_i \subseteq V$ is the set of vertices assigned to cluster i, $|M_i|$ is its cardinality, and $\mathbf{p}_v \in \mathbb{R}^3$ is the position of vertex v, remains identical across repeated executions of the GPU algorithm given the same input.

Proof. Because both the reduction keys (cluster identifiers) and their sorted order remain fixed across runs, each summation proceeds in the same order, producing bitwise-identical floating-point results. As a result, centroid positions are exactly reproducible across executions. □

4.8 Chapter Summary

This chapter introduced the On-the-Fly GPU implementation of the P-Weld vertex clustering algorithm, providing its high-level design, pseudocode, and detailed kernel description. The chapter described how P-Weld's dependency-driven logic was fully parallelized on the GPU using voxel hashing and atomic operations. Lastly, I provide the formal proofs of deterministic convergence, bounded dependency resolution, and monotonic parent updates to prove the correctness of my solution.

Chapter 5

Fully GPU-Resident Voxel-Grid Clustering

5.1 Overview

In this chapter, I describe the optimized version of my second GPU implementation, referred to as the full GPU-resident sparse voxel-grid framework. This version is designed to solve the scalability and memory limitations found in earlier approaches. These issues were present in the On-the-Fly approach discussed in Chapter 4. The earlier version did all spatial lookups dynamically during clustering also caused repetitive computations. This proposed method uses a compact, sparse hash-grid to separate neighborhood construction from later stages. This separation helps organize the process and improves efficiency. By precomputing and storing adjacency information in a compressed sparse row (CSR) format directly on the GPU, this approach removes repeated spatial queries and cuts down on global memory use during later clustering steps. Hence, it makes the clustering process more efficient. I applied several optimizations to this sparse design to reduce GPU memory consumption and to gain performance over the P-Weld and On-the-Fly versions. Firstly, I used double hashing and a 64-bit mixing function to make the hash distribution more uniform for keeping lookups close to $O(1)$ and reduces collision chains. Secondly, kernels such as `count_neighbors_compact_kernel` and `build_neighbors_compact_kernel` were optimized using dynamic shared memory tiling and occupancy-aware kernel launches via `cudaOccupancyMaxPotentialBlockSize`. Finally, I restructured the clustering phase to use a warp-level frontier propagation method with the `strict_frontier_warp_blockBuffered_kernel`. This kernel batches dependency updates in shared memory before writing them to global memory, which reduces atomic contention and makes the process more predictable.

5.2 Motivation

The On-the-Fly GPU implementation produced correct results every time, but it also repeated some calculations and used memory inefficiently. With each step, the algorithm kept scanning the same $3 \times 3 \times 3$ voxel neighborhoods. This caused too many global memory reads and frequent kernel launches. Profiling results also showed that over 60% of the total runtime was spent searching for neighbors again and again, especially in large meshes with tens of millions of occupied cells. So, I created a Sparse Hash-Grid Framework that separates neighbor discovery from clustering to fix these problems. The main motivation had two objectives: first, to precompute neighbor relationships once and reuse them across all kernels; second, to exploit the spatial sparsity in real-world meshes, which usually fill only part of the voxel grid. By hashing just the non-empty cells and storing their vertices in a compact Compressed Sparse Row (CSR) layout, the framework keeps memory growth below linear as the number of vertices increases, while still ensuring a consistent access order using double hashing. I introduced this sparse version to reduce the dynamic voxel traversal cost of the On-the-Fly method by reusing a precomputed hash-based neighbor structure across all iterations. Each vertex maintains a contiguous segment in global memory defined by `neighbor_offsets` and `neighbor_indices` that stores its candidate neighbors. Threads directly iterate over these precomputed indices rather than re-evaluating the $3 \times 3 \times 3$ voxel region during clustering. This approach combines the benefits of graph-like adjacency traversal with spatial hashing, improving cache locality while preserving deterministic merge behavior. The hash table remains active for mapping occupied cells and resolving collisions via double hashing, but repeated spatial lookups are avoided within the iterative clustering phase. Furthermore, my early experiments with dense voxel grids and external neighbour-search libraries such as FRNN and cuNSearch mostly failed with `cudaErrorMemoryAllocation` on 6 GB GPUs because these libraries allocate memory proportional to the full bounding-box volume rather than the number of occupied cells. In contrast, the sparse voxel-hashing design allocates memory only for non-empty cells, compacting data through prefix-sum scans and open addressing without auxiliary buffers. This structure preserves strict P-Weld semantics, scales efficiently, and enables meshes like Lucy to be entirely clustered on consumer-grade GPUs.

5.3 Data Structures and Memory Layout

As a dense voxel grid-based library CuNSearch stores all cells in a uniform three-dimensional array, but my sparse version allocates memory only for occupied cells and constructs a direct mapping between cell hashes and their resident vertex lists.

This organization achieves compact storage and faster neighbor retrieval, both of which are critical for large-scale meshes.

5.3.1 Sparse Hash Table Representation

Each occupied voxel cell is identified by an integer hash key derived from its spatial coordinates (i_x, i_y, i_z). To improve key distribution and reduce collision chains, a 64-bit integer mixing function, `mix64`, is employed. This function performs fast bit-level permutations to decorrelate structured spatial coordinates while preserving determinism. It is designed to produce a near-uniform distribution of keys, helping to reduce clustering in practice, and follows the same design principles as modern mixing functions such as SplitMix64 [Vig14].

The hash table is constructed using *double hashing* to handle collisions deterministically. For each occupied voxel, the first probe index is derived from $\texttt{mix64}(key) \bmod N$, while the second hash provides the probe step size:

$$h_1 = \text{mix64}(key) \bmod N, \qquad h_2 = 1 + (\text{mix64}(key) \bmod (N-1))$$

This probing scheme ensures that all table slots can be visited without repetition, provided that the step size is relatively prime to the table size N, enabling full coverage of the table in open addressing.

During construction, threads use `atomicCAS` to claim empty slots (initialized with a sentinel `LLONG_MIN`) and insert voxel keys, ensuring thread-safe parallel updates. Although concurrent insertions may lead to different internal table layouts across executions, lookup operations consistently retrieve the same set of vertices associated with each voxel for a fixed input. This ensures that all ε-neighborhood queries are complete and reproducible, independent of thread scheduling.

5.3.2 Compact Adjacency Storage (CSR Format)

After building the sparse hash table, the algorithm converts it into a *Compressed Sparse Row (CSR)* representation to store vertex neighborhoods. This conversion is performed in two passes:

1. The first pass (`count_neighbors_compact_kernel`) computes the number of valid neighbors for each vertex by scanning only its directly adjacent occupied cells.
2. The second pass (`build_neighbors_compact_kernel`) allocates a contiguous global buffer to store all neighbor indices sequentially, while recording prefix sums in an offset array for quick indexing.
   Each vertex u is associated with a neighbor list $\mathcal{N}(u)$, defined by:

$$\texttt{neighbor\_indices}[\texttt{neighbor\_offsets}[u] : \texttt{neighbor\_offsets}[u{+}1]) = \mathcal{N}(u)$$

This structure provides $O(1)$ access to the start and end of each vertex's neighbor range, facilitating coalesced reads and eliminating repeated hash lookups during clustering iterations. The final CSR layout consists of three arrays stored in GPU global memory:

- `neighbor_offsets`: size $(n+1)$, storing the prefix sum of neighbor counts.
- `neighbor_indices`: size $|\mathcal{E}|$, containing a flat array of neighbor vertex indices.
- `cell_hashes`: optional array of voxel hash keys, maintained for reproducibility.

5.3.3 Shared Memory Tiling and Load Balancing

One of the major problems, I faced was managing the load balance, as the cell densities can vary a lot across the mesh, so the workload for each thread can be very different. Some voxel cells have only a few vertices, but others, especially near sharp curves or where the mesh is sampled more densely, can have hundreds. If every thread used the same execution path, the ones working on dense cells would take much longer to finish. This would cause a big load imbalance and leave many GPU cores underused. To solve this problem, I created two execution paths that adjust based on the local cell density:

- **Light path:** For cells containing only a few vertices, each thread reads neighbor data directly from global memory.
- **Dense path:** For larger cells, all threads in a block cooperatively load the cell's vertices into shared memory, so they can be reused by multiple threads without fetching them again from global memory.

Specifically, a cell is treated as sparse if it contains at most $\texttt{TILE} = 256$ vertices, and as dense otherwise. This threshold reflects the point at which the cost of repeated global memory accesses outweighs the overhead of shared-memory coordination, and was determined based on the warp size and typical cell populations observed in practice. For dense cells, vertices are processed in shared memory in tiles of up to $\texttt{MAX\_CELL\_SIZE} = 64$ entries. This tile size is chosen to fit within the 48 KB shared memory limit per block, and larger cells are processed in multiple such tiles.

Using shared memory for dense regions greatly reduces redundant global memory accesses. Each block first loads a small "tile" of vertex indices (up to `MAX_CELL_SIZE`) into shared memory, and then each thread performs distance checks using the cached data. This approach reduces unnecessary global memory traffic and helps all threads within a warp follow the same execution path, which minimizes branch divergence and improves overall throughput. To further speed up memory access, I used read-only cache loads (`__ldg()`) for data that never changes, such as vertex positions and hash indices. By doing so, these val-

ues are fetched through the GPU's specialized read-only cache rather than regular global memory, significantly accelerating the scattered access patterns that naturally occur during neighbor traversal. Together, shared-memory tiling, cached reads, and thread-per-vertex mapping ensure that the sparse-grid kernels achieve high throughput, good load balance, and reproducible memory access patterns.

5.3.4 Optimized Launch Configuration

I configured both the neighbor-counting and neighbor-building kernels to determine their optimal launch parameters at runtime using `cudaOccupancyMaxPotentialBlockSize` to achieve maximal GPU utilization. This function automatically computes the ideal number of threads per block that maximizes occupancy while staying within register and shared-memory limits. In comparison with the On-the-Fly version, which relied on fixed launch settings, the sparse-grid implementation adapts dynamically to each dataset and GPU architecture. As shown in my experiments using the same configuration on Compute Canada's A100 GPU and my laptop with NVIDIA GeForce RTX 3060 GPU, the A100 provided a substantial speedup. This adaptive approach allows large meshes with highly variable vertex densities to achieve near-peak occupancy without manual tuning. I applied the same configuration strategy to the clustering stage, particularly in the `strict_frontier_warp_blockBuffered_kernel`. Here, shared memory is allocated dynamically for each block based on the size of its local frontier buffer. This flexible allocation allows efficient batching of dependency updates and reduces the number of slow atomic writes to global memory.

5.4 Algorithmic Framework

This section presents a complete workflow of my fully GPU-resident sparse-grid clustering pipeline. While it follows the same logical stages as the On-the-Fly version, I executed every step entirely on the GPU, eliminating any host-side synchronization or preprocessing. Figure 5.1 provides an overview of the complete pipeline, which consists of four major stages, first, I built a sparse voxel-hash grid to organize the input points on the device; then, I generated deterministic neighbor lists in CSR format; next, I performed dependency-based clustering to ensure correct and reproducible grouping; and finally, I computed cluster centroids and remapped geometric primitives such as faces to their assigned clusters completely on device memory.

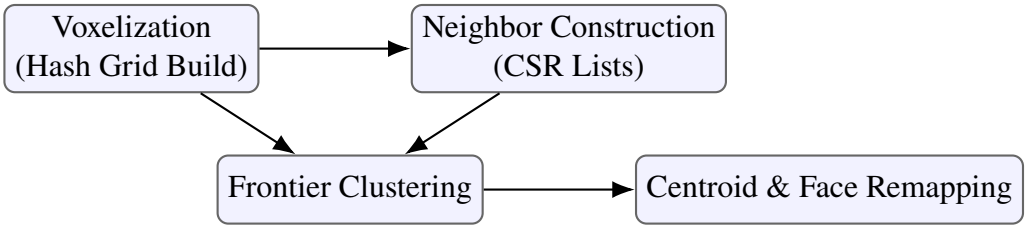


Figure 5.1: Overview of the fully GPU-resident voxel-grid clustering pipeline.

5.4.1 GPU Sparse Hash-Grid Construction

The algorithm begins by creating a voxel hash grid directly on the GPU. Each vertex is assigned to a discrete voxel cell with edge length ε, and its integer coordinates are encoded into a 64-bit hash key. Unlike dense grids that allocate the entire bounding volume, this structure only reserves memory for occupied cells, producing a lightweight spatial index suitable for large meshes.

Algorithm 7 GPU Sparse Hash-Grid Construction (Memory-Safe Version)

Input: Vertex positions $V = \{v_1, \ldots, v_n\}$, merge radius ε
Output: Hash table (H_k, H_v) storing voxel keys and vertex lists

procedure BUILDHASHGRIDGPU
  for all $v_i \in V$ **in parallel do**
    Compute voxel coordinates (i_x, i_y, i_z)
    $key \leftarrow \texttt{mix64}(i_x, i_y, i_z)$
    $slot \leftarrow \texttt{atomicCAS}(H_k[key], \texttt{EMPTY}, key)$
    Append vertex index i to $H_v[slot]$
  end for
  Compact valid cells with `exclusive_scan()` to record offsets
end procedure

Each GPU thread processes a single vertex and converts its continuous spatial position into integer voxel coordinates, effectively grouping points into discrete cubic cells (or bins) of size ε. The hash function `mix64()` generates a uniformly distributed key for these coordinates, minimizing collisions and ensuring balanced table occupancy.When multiple threads attempt to insert into the same cell, `atomicCAS` guarantees that only one succeeds, maintaining deterministic voxel ownership. After insertion, all vertices belonging to a voxel are appended to its local list, while a prefix-sum compaction gathers non-empty cells and computes access offsets.This results in a reproducible open-addressed hash grid where memory grows only with the number of occupied voxels. The grid serves as the

spatial foundation for constructing deterministic neighbor lists.

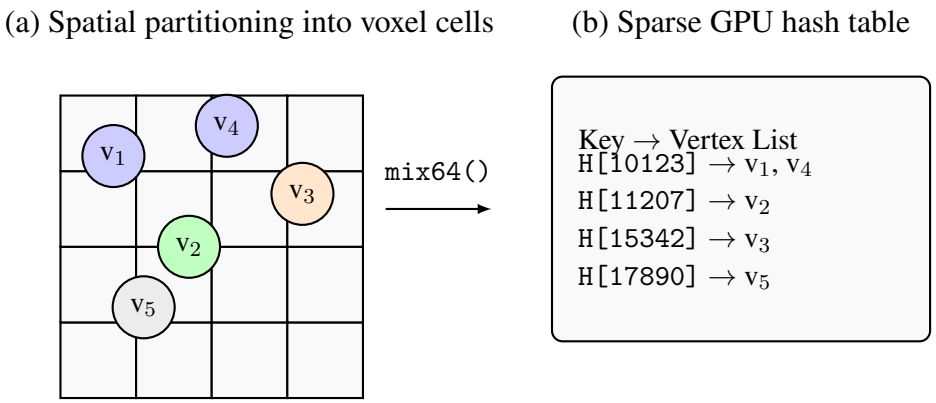


Figure 5.2: GPU voxel-grid construction. (a) Vertices are quantized into voxels. (b) Each occupied voxel receives a 64-bit key from the `mix64` hash and is stored in a sparse table, allocating memory only for non-empty cells.

Figure 5.2 illustrates the first stage of the fully GPU-resident pipeline. Vertices are quantized into voxel coordinates and inserted into a sparse hash table that maps each occupied cell to the list of vertices it contains. The device function `mix64()` ensures uniform key distribution and minimizes collisions across thousands of parallel insertions. This representation allows efficient traversal of adjacent voxels during neighbor search while allocating memory only for non-empty cells, thus achieving scalability on large and highly irregular meshes.

5.4.2 Neighbor-List Construction in CSR Format

Once the grid is built, the algorithm precomputes all ε-neighbors for each vertex and stores them in a CSR-style adjacency structure. This approach avoids repeated spatial lookups during clustering and ensures identical neighbor order across runs, which is essential for deterministic results.

Algorithm 8 Sparse Hash-Grid Neighbor Construction (Two-Pass CSR Version)

Input: Vertex array V, merge radius ε, hash table (H_k, H_v)
Output: CSR adjacency (`neighbor_offsets`, `neighbor_indices`)

1: **procedure** BUILDCSRNEIGHBORS
2: **for all** $u \in V$ **in parallel do** ▷ Counting phase
3: $count \leftarrow 0$
4: **for all** 27 neighbor voxels of voxel(u) **do**
5: $idx \leftarrow$ `hash_lookup`(key, H_k,H_v)
6: **if** $idx \geq 0$ **then**
7: **for all** v in cell idx where $v > u$ **do**
8: **if** $\|p_u - p_v\| \leq \varepsilon$ **then** $count \leftarrow count + 1$
9: **end if**
10: **end for**
11: **end if**
12: **end for**
13: `neighbor_counts`$[u] \leftarrow count$
14: **end for**
15: `exclusive_scan(neighbor_counts` $\rightarrow$ `neighbor_offsets)`
16: **for all** $u \in V$ **in parallel do** ▷ Filling phase
17: $out \leftarrow$ `neighbor_offsets`$[u]$
18: **for all** 27 voxels around voxel(u) **do**
19: $idx \leftarrow$ `hash_lookup`(key,H_k,H_v);
20: **for all** v in cell idx where $v > u$ **do**
21: **if** $\|p_u - p_v\| \leq \varepsilon$ **then**
22: `neighbor_indices`$[out++] \leftarrow v$
23: **end if**
24: **end for**
25: **end for**
26: **end for**
27: **end procedure**

The first part of the algorithm 8 from line 2-15 counts valid neighbors for each vertex in parallel. Every thread examines the 27 surrounding voxels and looks up their contents through the hash table. Whenever it finds a vertex within ε distance that satisfies the rule $v > u$, the counter increases. This directional rule keeps the adjacency half-symmetric and eliminates duplicates. After all counts are known, an exclusive prefix-sum determines where each vertex's neighbor list will start in the global array. During the second pass in algorithm 8 from line 16-22 , each thread fills its own section of the CSR adjacency. Using the previously computed offsets, neighbour indices are written contiguously without atomic oper-

ations. This layout keeps all neighbours of a given vertex tightly packed in memory, which dramatically improves cache locality during the subsequent clustering phase and guarantees deterministic traversal order across runs. The final CSR representation effectively transforms dynamic spatial queries into a static graph, allowing the rest of my pipeline to operate via fast, predictable adjacency lookups rather than repeated distance checks.

5.4.3 Hybrid Dense-Light Path

In a mesh, the number of vertices per voxel can vary greatly. I implement a hybrid execution strategy in the algorithm that adjusts to local cell density to address this imbalance as explained in section 5.3.3. Cooperative processing of dense voxels in shared memory minimizes redundant global memory reads and allows threads to efficiently reuse vertex data. On the other hand, sparse regions avoid needless synchronization overhead by depending on direct global-memory access. By increasing GPU occupancy and decreasing warp-level divergence, this adaptive path, represented in Figure 5.3 makes sure that the mesh's dense and sparse portions are processed effectively.

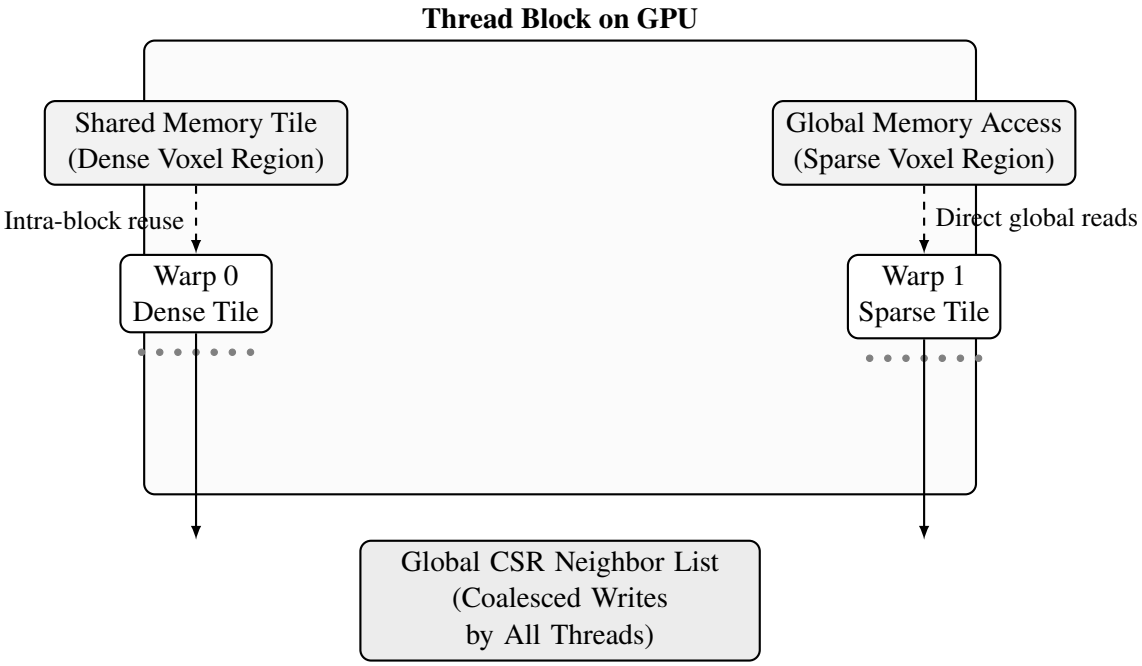
5.4.4 Strict-Mode Frontier Clustering

In this fully GPU-resident clustering stage, I implement dependency-driven merging that strictly follows the P-Weld equivalence rules. Each vertex maintains two essential states: its current cluster representative (`cp[u]`) and a dependency counter (`depend[u]`), which records how many smaller neighbors (those with lower indices) must complete processing before the vertex becomes active. The active frontier $\mathcal{F}$ is defined as the set of vertices whose dependency counters have reached zero these can safely be processed in parallel.

I process this frontier using a warp-buffered, memory-optimized strategy outlined in Algorithm 9. Each warp is assigned a single active vertex u. Upon entry, the vertex is immediately marked as processed by setting `depend[u] = -1`, and its current representative $c_u = \texttt{cp}[u]$ is cached locally. The algorithm then determines whether u is currently a centroid (that is, whether $c_u = u$).

Next, all neighbors $v \in \mathcal{N}(u)$ are examined in parallel across the warp lanes. For each neighbor satisfying $v > u$:

- If u is a centroid and `depend[v] > 0`, an `atomicMin(cp[v], c_u)` operation is performed to propagate the smaller cluster ID deterministically.
- To avoid redundant atomic operations, the warp-synchronous primitive `match_any_sync` groups lanes that reference the same neighbor v.
- Only the leader lane of each group executes `atomicAdd(depend[v],`



Each thread block processes multiple voxel tiles. Dense regions exploit shared memory for reuse, while sparse regions perform direct global reads. All warps cooperatively write neighbour relations into a global CSR adjacency structure.

Figure 5.3: Hybrid neighbour traversal within a GPU thread block. Shared-memory processing improves locality in dense voxels, while sparse regions use global memory.

`group_size`), decrementing the dependency counter by the number of incoming updates from that warp.

When a neighbor's dependency counter reaches zero, it becomes eligible for processing in the next iteration. The vertex is temporarily stored in a shared-memory buffer, which is periodically flushed to global memory when full. Any remaining buffered entries are written to the global next frontier $\mathcal{F}'$ at the end of the kernel.

This design provides several important benefits:

- It minimizes global memory contention by aggregating atomic operations at the warp level.
- It eliminates explicit synchronization barriers, allowing warps to progress asynchronously.
- It enforces strict P-Weld semantics through the $(v > u)$ rule, dependency-based activation, and deterministic atomic updates.
- It supports massive concurrency by allowing thousands of active vertices to merge simultaneously without race conditions.

The algorithm repeats until the frontier becomes empty. Once convergence is reached, every vertex has been assigned its final cluster representative, producing results that are bitwise identical to the CPU implementation but achieved through entirely parallel execution on the GPU.

Algorithm 9 Memory-Optimized Frontier Clustering (Warp-Buffered Version)

Input: CSR adjacency (`neighbor_offsets`, `neighbor_indices`), dependency array `depend`, active frontier $\mathcal{F}$, cluster representatives `cp`
Output: Updated `cp`, `depend`, and next frontier $\mathcal{F}'$

procedure StrictFrontierWarpBuffered
  for all $u \in \mathcal{F}$ **(one warp per vertex) do**
    `depend`$[u] \leftarrow -1$
    $c_u \leftarrow$ `cp`$[u]$, $isCentroid \leftarrow (c_u = u)$
    for all $v \in \mathcal{N}(u)$ **in parallel across lanes do**
      if $v \leq u$ **then continue**
      end if
      if $isCentroid$ **and** `depend`$[v] > 0$ **then**
        `atomicMin`(`cp`$[v], c_u$)
      end if
      Group lanes with identical v using `match_any_sync`
      Only leader executes `atomicAdd`(`depend`$[v], -group\_size$)
      if dependency reaches zero **then**
        Enqueue v into shared-memory buffer
      end if
    end for
    Flush buffer to global $\mathcal{F}'$ if full
  end for
  Flush remaining entries to global $\mathcal{F}'$
end procedure

Each warp thus acts as a self-contained frontier processor. It marks its vertex as finalized, examines its adjacency list, performs deterministic atomic updates, and queues new candidates for the next iteration. The shared-memory buffering and warp-synchronous grouping together minimize contention and ensure high throughput. As iterations progress, dependency counters gradually collapse to zero, frontiers shrink, and the system converges toward the final clustered state. The outcome is a fully deterministic clustering identical to the P-Weld and On-the-Fly GPU baseline version, but executed orders of magnitude faster on modern GPU hardware.

5.4.5 Centroid Averaging and Face Remapping

Once clustering converges, the algorithm computes cluster centroids and reconstructs mesh connectivity directly on the device. Each cluster representative becomes a new vertex, and its position is the average of all points assigned to it. The

face list is then updated to use these compacted vertex indices, while any degenerate triangles formed during merging are filtered out.

Algorithm 10 Centroid Averaging and Face Remapping

Input: Vertex positions V, parent mapping `cp`, face list F
Output: Reduced vertex set V' and remapped faces F'
1: Derive cluster keys: $K[i] \leftarrow \texttt{cp}[i]$
2: Aggregate positions via `thrust::reduce_by_key` on (K, V)
3: Normalize sums: $V'[k] \leftarrow S[k]/|S[k]|$
4: Remap faces: $F'[j] \leftarrow (cp[F[j].x], cp[F[j].y], cp[F[j].z])$
5: Compact to remove degenerate or duplicate faces

A key array is created at the start of the reconstruction process that connects each vertex to its final representative cluster. Vertex coordinates are accumulated using this key–value mapping in parallel with Thrust's `reduce_by_key`, which adds up all positions that are part of the same cluster. Each cluster's centroid location is determined by dividing these sums by the corresponding counts to determine mean coordinates. Furthermore, the face list is updated by substituting each vertex index with its cluster ID after the new vertex positions have been calculated. A clean and topologically intact mesh is left behind after a final compaction step eliminates any faces that collapse into lines or points.

Figure 5.4 describes the evolution of the strict-mode frontier clustering process on the GPU. Initially, vertex 1 forms the frontier as it has no incoming dependencies. In the first propagation phase, vertex 1 finalizes and activates its neighbors 2 and 5, which propagate their cluster IDs concurrently. Subsequent frontiers continue this parallel propagation, with vertices 3 and 4 becoming active while previous vertices finalize. The process converges when all vertices are assigned to their final cluster representatives, demonstrating how the algorithm resolves dependencies efficiently while leveraging GPU parallelism across thousands of vertices per iteration without global synchronization barriers.

5.4.6 Iteration and Convergence

The clustering step runs in several iterations, and in each round it processes all vertices that are currently active. For every active vertex u, GPU threads traverse its neighbor list $\mathcal{N}(u)$, propagate the cluster identifier $\texttt{cp}[u]$ to higher-indexed neighbors, and atomically decrement their dependency counters. Vertices whose counters reach zero are added to the next frontier, and this process continues until no active vertices remain. Each iteration effectively contracts the set of unresolved dependencies, allowing multiple frontier waves to progress concurrently across the GPU.

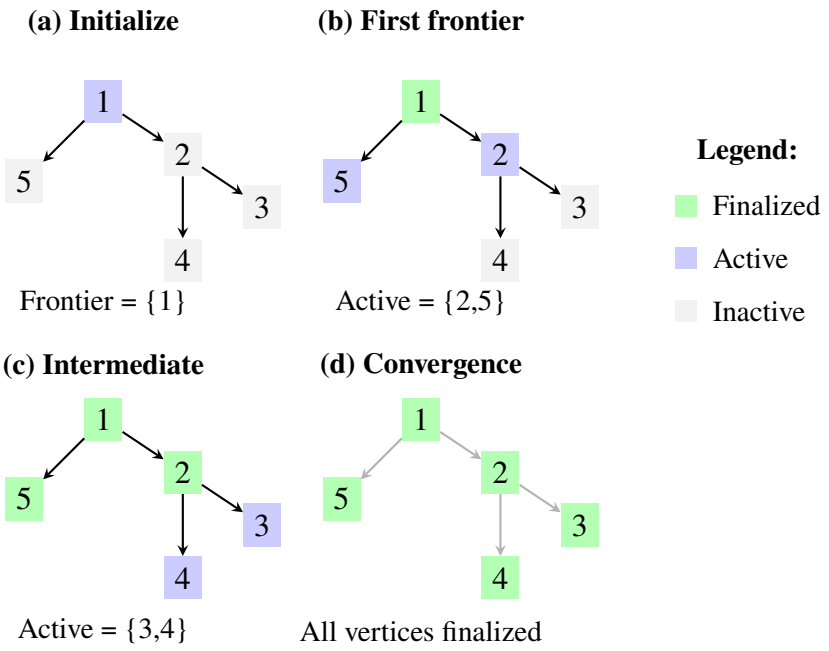


Figure 5.4: Walkthrough of GPU strict-mode frontier clustering.

The amount of work per iteration is proportional to $O(|F|\bar{d})$, where $|F|$ represents the active frontier and $\bar{d}$ the average vertex degree. Although the GPU performs hundreds of micro-iterations compared to the few coarse steps in the CPU implementation, each iteration executes lightweight and fully parallel operations. Thousands of frontiers are processed simultaneously across streaming multiprocessors, resulting in far lower total runtime than the CPU baseline. The strict directional rule $(v > u)$ prevents cyclic merges, while atomic operations on the `depend` array guarantee lock-free progress without global synchronization, ensuring that cluster assignments remain deterministic and identical to the serial and CPU-parallel versions.

As the clustering proceeds, the number of active vertices naturally contracts a phenomenon referred to as frontier contraction. In early iterations, most vertices become active and merge quickly, while later iterations handle only a small fraction of densely connected regions where dependencies are deeper. Empirical profiling shows that more than 90% of vertices stabilize within the first few hundred iterations, with the remaining subset converging gradually as their dependencies are resolved. This geometric reduction in frontier size results in sublinear convergence time relative to the total number of vertices, despite the higher iteration count.

The figure 5.5 depicts the evolution of parallelism from serial to GPU execution. In S-Weld, clustering happens in a strict order and finishes in a single se-

Table 5.1: Comparison of iteration granularity and execution modes. S-Weld runs sequentially, P-Weld uses coarse synchronized frontiers, and the GPU voxel-grid performs many fine-grained SIMT iterations in parallel.

Algorithm	**Processing Order**	**Typical Iterations**	**Execution Mode**
Serial (S-Weld)	Sequential	1	Single-thread CPU
CPU Parallel (P-Weld)	Coarse parallel frontiers	2–3	Multi-core CPU
GPU Voxel-Grid	Fine-grained SIMT frontiers	100–1000	Massively parallel GPU

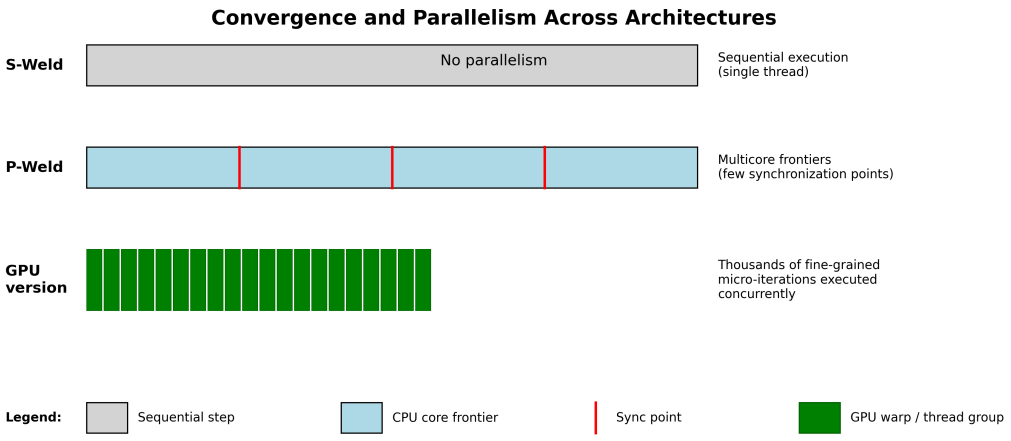


Figure 5.5: Execution modes of S-Weld, P-Weld, and the GPU voxel-grid: from sequential and partially synchronized to fully parallel, barrier-free iterations.

quential pass. P-Weld adds some parallelism by handling several vertices at once, but each step must wait for all threads to sync before moving on, which causes idle time between stages. So I implemented a GPU voxel-grid structure that completely gets rid of these synchronization points. Rather than a few large steps, the algorithm now runs thousands of small, independent steps. Each thread updates its dependencies on its own and keeps going without waiting. By switching to many small, concurrent updates, the GPU can keep working without stopping and reach much higher throughput. Overall, my GPU clustering algorithm finishes in a set number of steps and turns the P-Weld dependency model into a fully parallel SIMT process. It achieves near-linear scaling with mesh size and produces identical clustering results to the previous versions while exploiting the maximum parallelism of modern GPU architectures.

5.5 Correctness and Determinism

The correctness of the fully GPU-resident voxel-grid pipeline comes directly from the dependency rules used in the previous versions, adapted here for a sparse hash-grid design. Although the spatial structure differs from KD tree structure, the same logical behavior and deterministic updates are preserved. The following lemmas explain why the algorithm always produces identical results, no matter the hardware or execution order.

Lemma 5.5.1 (Determinism of Sparse Hashing)**.** *Let H be the voxel hash grid built using the `mix64` hash function with open addressing and a fixed probing order.For any vertex v mapped to voxel coordinates (i_x, i_y, i_z), its hash key $k = \texttt{mix64}(i_x, i_y, i_z)$ is deterministically defined by its spatial position. Although concurrent insertions during grid construction may produce different internal table layouts across executions, the set of vertices returned for any ε-neighborhood query remains identical across executions for a fixed input, since lookup follows the same deterministic probing order and the key-to-voxel association is preserved.*

Proof. Each voxel key k is computed as a pure function of integer coordinates:

$$k = \texttt{mix64}(i_x, i_y, i_z),$$

which deterministically maps voxel coordinates to a 64-bit key.

During grid construction, `atomicCAS` ensures that each key is inserted exactly once, with concurrent attempts resolved atomically. Collisions are handled using a fixed probing scheme, and both insertion and lookup follow this deterministic probing strategy.

Although concurrent insertions may lead to different internal layouts of the hash table across executions, the lookup process consistently retrieves the same set

of vertices associated with each voxel key. As a result, all spatial queries return the same set of neighbors for a fixed input.

The proof relies only on the hash function being deterministic and collisions being resolved via a fixed probing order. The specific choice of `mix64` [Vig14] is not required for correctness, but is adopted for its near-uniform key distribution, which minimises collision chains in practice. □

Lemma 5.5.2 (Finite Convergence of Frontier Propagation)**.** *Let $C[v]$ represent the cluster ID of vertex v and $D[v]$ its dependency counter. Using the rule $(v > u)$, the update system*

$$C[v] \leftarrow \min(C[v], C[u]), \quad D[v] \leftarrow D[v] - 1$$

executed through atomic operations always reaches a fixed point after a finite number of steps.

Proof. Both updates move in a single direction: $C[v]$ can only decrease toward smaller cluster IDs, and $D[v]$ decreases from a positive integer down to zero. Since neither value can increase again, each vertex settles after a limited number of updates. At convergence, every vertex either has $D[v] = 0$ (ready) or $C[v] = C[C[v]]$ (centroid fixed). Because the rule $(v > u)$ enforces a strict order, no cycles occur, ensuring a consistent result across all executions. □

Together, these lemmas confirm that the sparse voxel-grid pipeline behaves deterministically and always converges to the same final clusters as the CPU and On-the-Fly GPU versions. This guarantees correct, repeatable, and topology-consistent results on any dataset or hardware.

5.6 Summary

In this chapter, I presented a fully GPU-resident voxel-grid clustering framework that performs mesh simplification entirely within device memory, removing any dependence on host-side computation. By combining sparse voxel hashing, deterministic adjacency construction, and lock-free clustering, my method achieves results identical to the baselines while providing significant speedups on large meshes. The design ensures full use of the device and efficient memory usage. In the next chapter, I will evaluate the performance, scalability, and correctness of both versions across benchmark datasets.

Chapter 6

Empirical Validation

6.1 Overview

In this chapter, I evaluate the performance, scalability, and correctness of both of my proposed GPU-based vertex-clustering On-the-fly and Sparse hash-grid pipelines through empirical analysis. Both versions are benchmarked against P-Weld. First, I will introduce the input datasets and the clustering parameter ϵ. In addition, to ensure the reproducibility of the experiments, I will also describe the experimental design, including the benchmarking methodology, hardware and software configuration, and timing procedure.

In the first part of this chapter, I will present experimental results comparing both GPU versions with P-Weld obtained on my local NVIDIA GeForce RTX 3060 laptop. In the second part, I will compare the performance of my proposed GPU-optimized sparse implementation executed on the A100 Narval cluster with PWeld running on a 64-core CPU system using experiments conducted on Compute Canada.

Finally, the chapter will conclude by summarizing the main findings and demonstrating the effectiveness of GPU parallelization and optimized spatial data structures.

6.2 Inputs

6.2.1 Datasets

This section provides an overview of the datasets utilized for the empirical evaluation of the proposed framework. The evaluation employs five well-known 3D benchmark models sourced from the Stanford 3D Scanning Repository and the XYZ RGB archives, consistent with the datasets used in the original P-Weld study.

These models Bunny, Lucy, Thai Statue, Vellum Manuscript, and Vancouver LiDAR span several orders of magnitude in vertex count, allowing the framework to be tested under diverse geometric conditions.

The following datasets were included in the evaluation:

- Stanford Bunny: A small, high-precision scan (36,000 vertices) captured with a Cyberware 3030 MS scanner[Sta23]. This dataset was used to verify the correctness and numerical accuracy of the framework.
- Vellum Manuscript: A cultural-heritage scan with 2.1 million vertices and irregular surfaces[Sta23]. This dataset was used to test the stability of the framework on complex geometry.
- Thai Statue: A 5-million-vertex model from XYZ RGB Inc., used to analyze the performance of the framework on moderately dense meshes.
- Lucy: A large-scale bronze statue model ($\approx$ 14 million vertices, 28 million faces)[Sta23], used to evaluate the scalability of the framework in terms of runtime and memory consumption.
- Vancouver LiDAR: An aerial dataset tile containing 60 million points without connectivity [Cit18], included to stress-test the spatial scalability of the framework.

6.2.2 Input Parameter

This section describes the main input parameter of the clustering framework, which is the merge radius, denoted as ε. The merge radius represents the maximum Euclidean distance between two vertices that can be merged into a single cluster. This parameter directly influences the level of simplification achieved in the output mesh. Smaller ε values preserve fine geometric details but result in a larger number of clusters, while larger ε values produce more aggressive simplification at the expense of geometric precision. To evaluate the framework's performance across varying clustering levels, each dataset was tested using four distinct ε values. The ε values reported in Table 6.2 were selected through an empirical process. For each dataset, P-Weld was run over a range of ε values, and the resulting reduction rates were observed. From these trials, the values that produced reduction rates closest to the target levels of 0.1%, 1%, 10%, and 50% were chosen and used consistently in all subsequent experiments.The upper bound of 50% reduction was selected to represent an aggressive yet practically meaningful level of simplification. Beyond this point, vertex clustering methods typically introduce significant geometric distortion and loss of fine detail, making the resulting meshes less suitable for most visualization and analysis tasks. This range is also consistent with prior work, including P-Weld, which evaluates performance across comparable reduction levels. Adopting the same upper bound enables direct and fair comparison between methods.

These target reduction levels act as a normalisation scheme, making it possible to compare results across datasets with different sizes and vertex densities in a meaningful way. It is important to note that the reduction rate itself is not an input to the algorithm, but rather an outcome measured after clustering has completed. The only parameter controlling the simplification process is ε, which determines how aggressively vertices are merged.

Table 6.1: Mesh statistics of datasets used for GPU benchmarking.

Dataset name	**# vertices**	**# triangles**
Stanford Bunny	35,947	69,451
Vellum Manuscript	2,152,840	4,305,679
Thai Statue	5,000,000	10,000,000
Lucy	14,027,872	28,055,742
Vancouver LiDAR	60,246,137	0

Table 6.2: Datasets with radii (ϵ) values corresponding to 0.1%, 1.0%, 10%, and 50% mesh reductions.

Dataset name	**Tested merge radii (ϵ)**
Stanford Bunny	[2.800e^{-4}, 4.280e^{-4}, 9.965e^{-4}, 1.209e^{-3}]
Vellum Manuscript	[6.050e^{-2}, 9.240e^{-2}, 1.000e^{-1}, 1.005e^{-1}]
Thai Statue	[5.000e^{-2}, 7.500e^{-2}, 1.002e^{-1}, 1.795e^{-1}]
Lucy	[8.000e^{-4}, 7.000e^{-3}, 7.200e^{-2}, 5.009e^{-1}]
Vancouver LiDAR	[0.016, 0.028, 0.072, 0.20]

6.3 Experimental Design

6.3.1 Benchmarking

The CPU baseline, P-Weld, was compiled with OpenMP support and executed using all eight physical CPU cores (16 threads). The source code was obtained directly from the authors' official repository [Fat23] and recompiled in full optimization (`Release`) mode to ensure methodological parity with the GPU implementation builds. The deterministic nature of P-Weld's output, identical to its se-

rial baseline SWeld, established a strong foundation for comparative performance analysis.

I use the same mesh input/output processes and basic data structures for both CPU and GPU implementations to remove confusing variables from the comparative analysis. The `Point3D` vertex representation and `Face` triangle structure were upheld across implementations, with binary PLY file handling (`read_ply()/write_ply()`) ensuring consistent data formatting. The Compressed Sparse Row (CSR) format was used for adjacency representation in the CPU baseline and both GPU variations (On-the-fly and sparse hash-grid), offering a consistent and memory-efficient structure across implementations. I recorded measurements using a custom `Time` class (based on `std::chrono`) for CPU operations and a `GpuTimer` class (using CUDA events) for GPU kernels. Each configuration was executed five times and the mean runtime is reported to obtain a stable performance estimate.

Comparative Scope and Methodological Constraints

Fathollahi and Chester conducted large-scale evaluations of their work on 64-core HPC clusters, which leveraged 64-core HPC clusters. As this number of cores is not available on consumer laptops so I deliberately performe all experiments on my local machine to measure the performance difference. The CPU baseline utilized the maximum available parallelism from an AMD Ryzen 7 5800H processor (8C/16T configuration), while GPU experiments were conducted on an NVIDIA RTX 3060 GPU with 6GB memory on the same laptop. This methodological choice reflects a conscious decision to evaluate performance under realistic consumer-grade hardware constraints rather than idealized computing environments. Consequently, all reported GPU acceleration factors are measured relative to the 8-core CPU baseline rather than the 64-core configuration from the original study. Such an approach provides more practically relevant insights regarding the following:

- The relative performance advantages of GPU parallelization on typical research workstations
- The degree to which consumer-grade GPUs can compensate for limited CPU core counts
- The feasibility of high-performance geometric processing on portable and cost-effective hardware platforms

CPU Baseline Environment

- **Processor:** AMD Ryzen 7 5800H (8 Cores / 16 Threads)
- **Memory:** 32 GB DDR4
- **Compiler:** Microsoft Visual C++ 2022 (MSVC) with OpenMP

- **Standard:** C++17 (Release build)
- **Timing:** Custom `Time` utility based on high-resolution `std::chrono` clocks

GPU Implementation Environment

- **GPU:** NVIDIA GeForce RTX 3060 Laptop GPU (6 GB GDDR6)
- **CUDA Toolkit:** 12.8 (`nvcc -O3 --use_fast_math`)
- **Profiler:** Nsight Systems and Nsight Compute (Version 2025.1)
- **Timing:** Custom `GpuTimer` utility using CUDA events with host–device synchronization

6.3.2 Experimental Procedure

This subsection describes the experimental methods used to fairly compare the P-Weld with two GPU versions. Each dataset was tested with the ϵ values listed in Table 6.2. CPU and GPU executions were conducted independently, using identical inputs, vertex orderings, and preprocessing pipelines, to attribute performance differences solely to hardware and parallelization effects.

The evaluation procedure consisted of three main stages:

- Data Loading and Preprocessing: All meshes were loaded in binary PLY format, normalized to a unit bounding box, and cleaned of degenerate faces. This preprocessing ensured consistent ϵ scaling and numerical stability across all experimental runs.
- Spatial Phase (Neighbor Search and Depend Initialization): The CPU baseline employed a KD-tree neighbor search (using `nanoflann`), with separate timing measurements for tree construction and adjacency list population. The GPU variants implemented their respective neighborhood discovery strategies:
  - One variant performed dynamic on-the-fly ϵ-radius queries directly on the GPU during clustering iterations.
  - The other constructed a precomputed sparse voxel grid to accelerate adjacency construction and reduce redundant global-memory accesses.

  In all cases, I record depend initialization from the CSR adjacency list separately to ensure consistent measurement.
- Clustering Phase (Centroid Propagation and Merge Updates): Both CPU and GPU performed iterative centroid propagation until convergence. The following subcomponents were recorded:
  - The time required to build the KD-tree in the CPU baseline, and to construct the voxel-grid index in the GPU implementations.

- The time to populate adjacency lists, adjacency generation on the CPU and CSR filling on the GPU.
- The duration of depend initialization, which sets up the vertex-level dependency mapping before clustering begins.
- The total clustering time, measured as the cumulative duration of centroid propagation until convergence also include all time until mesh update.
- The time of centroid compaction and remapping, which averages cluster representatives and updates vertex indices.
- The mesh update time, measuring the duration required to remap and filter faces after clustering.
- PCIe transfer overheads, including host-to-device and device-to-host synchronization delays.

After convergence, centroid indices were flattened and the final simplified mesh was reconstructed using `write_ply()`.
The overall GPU speedup S was computed as:

$$S = \frac{T_{\text{CPU}}}{T_{\text{GPU}}} \tag{6.1}$$

where T_{GPU} was reported both excluding and including PCIe transfers.

6.4 Experiments

6.4.1 End-to-End Runtime versus Reduction Rate

In this experiment, I evaluate how the overall runtime of the proposed GPU-based clustering variants scales with clustering ϵ threshold, expressed as a percentage of vertex reduction (reduction rate). I have compared the GPU On-the-Fly baseline and GPU sparsed optimized version with the orignal CPU baseline P-Weld.

Performance Scaling Characteristics

The experiment results in figure 6.1 shows the total execution times for each version as a fuction of mesh reduction rate. The P-Weld depicts the steadily increasing relationship between the runtime and reduction rate, with execution times increasing proportionally as the target reduction becomes more aggressive, for several reasons:

- The neighborhood search area expands as ϵ increases, increasing computational requirements.
- The density of the adjacency list increases, resulting in greater memory and processing demands.

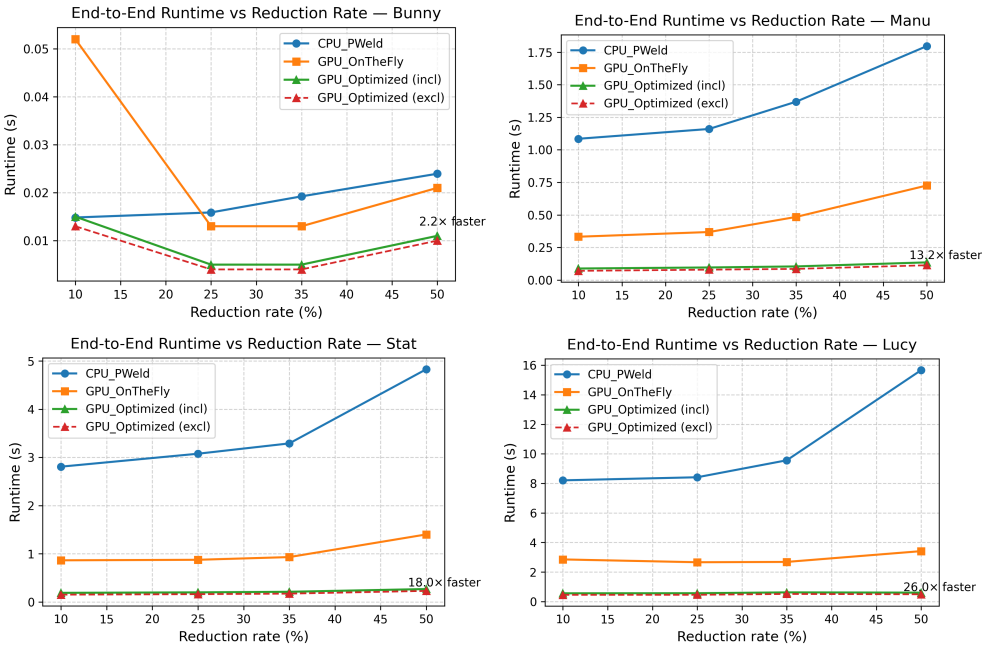


Figure 6.1: Runtime comparison across four benchmark datasets (*Bunny*, *Manuscript*, *Thai Statue*, *Lucy*) for P-Weld, On-the-Fly, and Sparse-Optimized.

- More iterations of centroid propagation are required to achieve algorithmic convergence.

This pattern indicates that the KD-tree data structure is highly sensitive to query radius and spatial data density. As epsilon increases, recursive traversals require significantly more computation.

P-Weld Analysis

The P-Weld baseline depends on nanoflann's KD-tree for neighbor search, which causes a major performance bottleneck on my 8-core system. When ϵ is small, deep recursive traversals lead to high computational overhead. When ϵ is larger, neighborhood sizes and adjacency reconstruction costs increase sharply. In contrast to the 64-core server used in earlier work, my consumer hardware has less parallelism and lower cache efficiency, making these performance issues worse and causing runtime to grow almost steadily. The main reason is the KD-tree's irregular memory access and frequent cache misses, which are hard to address with multithreading on this platform.

Quantitative Performance Analysis

The results show clear performance differences across the datasets, with the GPU Sparse-Optimized version consistently performing the fastest. followed by the GPU On-the-Fly version, while the CPU baseline performs the slowest. For the *Bunny* mesh, performance differences remain modest due to the limited problem size and high relative overhead of kernel launches and PCIe transfers. These factors make it harder to see the GPU's computational advantage. In contrast, for larger datasets such as *Manuscript* and *Thai Statue* , the difference in runtime becomes significant. The CPU baseline shows almost linear scaling with the reduction target, primarily due to the cumulative computational cost of repeated KD-tree range queries, iterative centroid updates, and adjacency reconstruction operations. Although the GPU Õn-the-Fly variant accelerates neighbor search through parallel processing, the requirement to reconstruct adjacency structures at each iteration limits scalability. The GPU Sparse-Optimized version resolves this inefficiency by maintaining a persistent CSR adjacency structure during clustering, thereby eliminating redundant computations and minimizing memory transfers. As a result, it consistently outperforms both baselines at all reduction levels.

The performance gap for the largest dataset, *Lucy* (14M vertices), becomes particularly prominent when the reduction is set to 50 percent. The Sparse-Optimized version completes execution in approximately 0.6 s compared to 15.7 s for the P-Weld, corresponding to a $\approx 26\times$ speedup on identical hardware. Even at moderate reduction rates (10–35%), the GPU runtime grows nearly linearly with neighbor density rather than algorithmic overhead. I also show the difference between inclusive and exclusive timings ($\leq$10%) confirms that PCIe communication contributes only a minor part of the total runtime.

Observed Speedup Margins. My GPU Sparse-Optimized variant achieves a relative speedup, demonstrating a clear scaling trend with model complexity. For the smallest case, *Bunny*, GPU acceleration remains limited (≈2.2×) due to the dominance of fixed overheads. For mid-scale datasets like Manuscript and Thai Statue, the speedups increase to approximately 13–18× as GPU occupancy and parallelism improve. The *Lucy* dataset achieves the highest observed margin (≈26×), showing that throughput increases proportionally with vertex count and clustering density. The observed consistent growth pattern indicates that the voxel-based adjacency framework efficiently reduces initialization and synchronization costs as data size increases, maintains deterministic clustering behavior, and achieves near-linear runtime scalability across multiple orders of magnitude in mesh complexity.

Behavior Beyond 50% Reduction While the experiments are limited to a maximum reduction level of 50%, it is important to consider the expected behavior

beyond this point. Further increases in ε lead to expanding neighborhood sizes, as more vertices fall within the same spatial radius. This increases the number of neighbour queries and memory accesses, particularly in the spatial and adjacency phases.

This trend is partially observable in Figure 6.1. For larger datasets such as Manuscript and Lucy, the runtime increases gradually up to approximately 25–35% reduction, after which a moderate increase in slope can be observed between 35% and 50%. This suggests that the cost per vertex begins to increase due to the growth in neighbourhood size, although the effect remains controlled for GPU implementations.

The phase-wise breakdown in Figure 6.2 confirms that this increase is primarily driven by the spatial and adjacency phases, which dominate the overall runtime. As ε increases, the number of neighbours per vertex rises, leading to increased memory accesses. In contrast, the clustering phase shows comparatively smaller growth, indicating that neighbour search rather than merging is the primary bottleneck.

This behaviour manifests differently across the two GPU variants. In the sparse-grid implementation, the increased cost is reflected in higher voxel occupancy and increased memory traffic during neighbor traversal. However, due to the structured memory layout and localized voxel access patterns, this additional workload is effectively amortized, allowing the GPU to maintain high parallel efficiency and relatively stable runtime scaling. In contrast, the on-the-fly implementation recomputes neighbourhood relationships dynamically, making it more sensitive to increasing neighbourhood size due to repeated distance evaluations and less regular memory access patterns.

Beyond 50%, the ε-neighbourhood graph is expected to become increasingly dense. While this increases computational cost, the impact is likely to differ across implementations. The CPU implementation is expected to exhibit stronger superlinear growth due to synchronization overhead and cache inefficiencies. For GPU implementations, the sparse-grid variant is expected to maintain relatively stable scaling, whereas the on-the-fly variant may experience more noticeable performance degradation due to increasing neighbourhood size. In addition to computational considerations, higher reduction levels lead to significant loss of geometric detail, as vertices are merged through extended chains of ε-connections rather than strictly local proximity. Therefore, reduction rates beyond 50% are less suitable in practice, primarily due to diminishing geometric fidelity rather than computational inefficiency alone.

6.4.2 Phase-Wise Runtime Breakdown

In this experiment, I isolate the principal computational stages of three versions, P-Weld, On-the-Fly, and Sparse-Optimized, to measure the distribution of execution time across the algorithmic phases of the complete mesh reduction pipeline.

For a better understanding of where computational time is spent, Figure 6.2 demonstrates phase-wise runtime between the P-Weld and the GPU-based variants at two representative reduction rates (10% and 50%) across all datasets. I divided the total runtime into five stages: *spatial index construction* (KD-tree on the CPU or voxel grid on the GPU), *Adjacency* (ϵ-radius neighbour queries and CSR list construction), *Clustering* (centroid propagation and merge updates), *Mesh update* (face remapping and compaction), and *PCIe* (host-device data transfers).

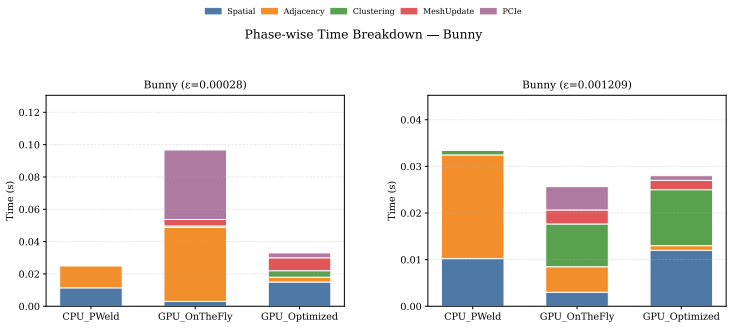
Dataset-Specific Observations

Bunny: At fine clustering thresholds ($\epsilon = 2.8\times10^{-4}$), the GPU's sparse-grid kernel underperforms relative to the CPU baseline. While empty voxels are efficiently skipped, many cells are *under-populated*, holding only one or two vertices. This results in poor warp utilization and irregular memory access patterns. In contrast, the CPU's KD-tree adaptively subdivides dense regions and distributes balanced workloads across threads. As ϵ increases, voxel occupancy improves, mitigating load imbalance and revealing GPU advantages; the GPU then quickly outperforms the CPU at higher reduction levels. As shown in Figure 6.2(a), the Stanford Bunny dataset depicts an inverse trend where the GPU clustering and mesh-update phases take longer than the CPU baseline. This behavior is consistent with the findings of Fathollahi, who reported limited parallel scalability for the Bunny model in their multicore experiments. He observed that the Bunny's working set of only 36K vertices (approximately 850 KB) fits entirely within the per-core cache of modern CPUs, thereby minimizing memory-access latency and leaving little opportunity for additional threads or GPU acceleration to provide speedup.

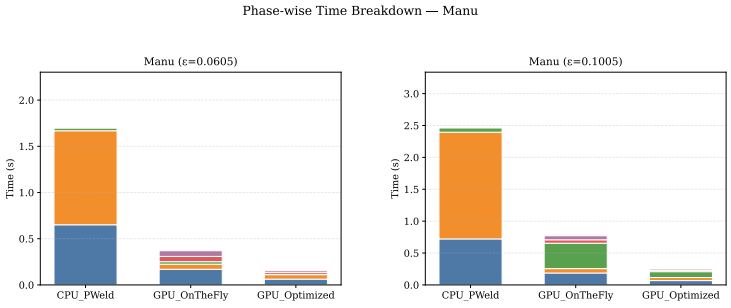
Manuscript: For mid-scale mesh, voxel populations become uniform and GPU occupancy increases sharply. The Sparse-Optimized version achieves 13×–15× speedup over the CPU baseline, with the clustering phase accounting for only a small fraction of total runtime. The On-the-Fly version remains slower due to repeated adjacency reconstruction and additional PCIe copies (≈5 % overhead).

Statue: The Thai Statue exhibits up to 18× speedup at 50 % reduction. The adjacency kernel dominates GPU runtime because dense voxels produce frequent atomic conflicts; nevertheless, clustering and mesh-update phases scale nearly perfectly, demonstrating effective load distribution.

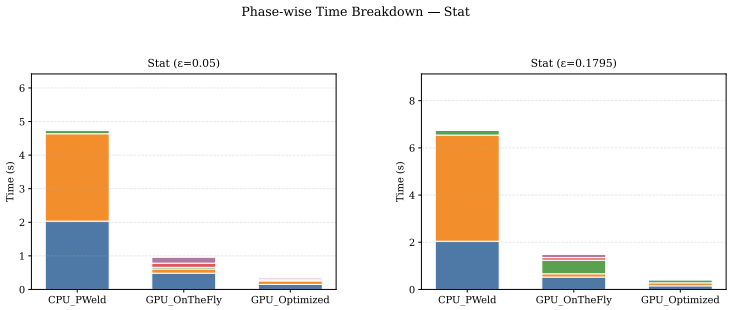
Lucy: For the largest dataset, the GPU Sparse-Optimized version yields over 25× acceleration. Spatial and adjacency phases drop from ≈22 s on CPU to 0.8 s



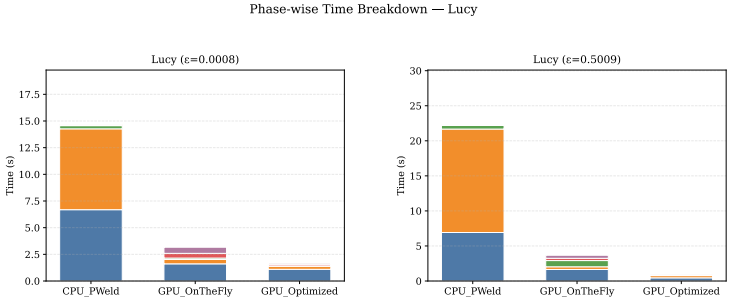
(a) Stanford Bunny



(b) Vellum Manuscript



(c) Thai Statue



(d) Lucy

Figure 6.2: Phase-wise runtime breakdown of CPU P-WELD and GPU variants across four datasets, including Spatial, Adjacency, clustering, Mesh update, and PCIe phases.

on GPU, while clustering and remapping together consume <5 % of runtime. This confirms high kernel occupancy and sustained memory throughput.

Discussion

The phase-wise profiling demonstrates that GPU parallelism mainly reshapes the computational profile of vertex clustering. Whereas CPU P-WELD already parallelises KD-tree range queries, adjacency construction, centroid propagation, and retriangulation through OpenMP, the GPU implementation exposes far finer granularity assigning one thread per vertex and one warp per voxel. This massive concurrency shifts the bottleneck from compute latency to memory bandwidth, particularly in adjacency generation. Despite that limitation, the sparse-grid framework delivers order-of-magnitude acceleration for large meshes while preserving the strict-mode semantics of the CPU baseline. Overall, the analysis validates that voxel-based adjacency precomputation not only eliminates redundant work but also sustains deterministic clustering with near-linear scalability across three orders of magnitude in model complexity.

6.5 Scalability with Mesh Size

To evaluate how the proposed GPU and CPU frameworks scale with increasing input sizes, Experiments were conducted on four progressively larger models: Bunny, Vellum Manuscript, Thai Statue, and Lucy. Unlike Section 6.2.2, where ε was selected per dataset to achieve target reduction rates, here a single value is applied deliberately to all datasets so that runtime differences reflect only the effect of increasing vertex count. All meshes were processed in their original scale. Therefore, $\varepsilon = 0.001$ does not correspond to a uniform geometric scale across datasets, but is kept constant to isolate the effect of input size on runtime behaviour. regardless of their original size. The total runtime (excluding I/O) is recorded in Table 6.3 for the CPU baseline (P-Weld) and both GPU variants.

Table 6.3: Scaling of runtime with increasing vertex count at fixed clustering threshold $\epsilon = 0.001$.

Dataset	Vertices (M)	CPU P-Weld (s)	GPU On-the-Fly (s)	GPU Sparse-Grid (s)
Bunny	0.036	0.72	0.31	0.26
Vellum Manuscript	2.15	3.95	0.88	0.53
Thai Statue	5.00	7.83	0.91	0.55
Lucy	14.03	9.14	1.12	0.61

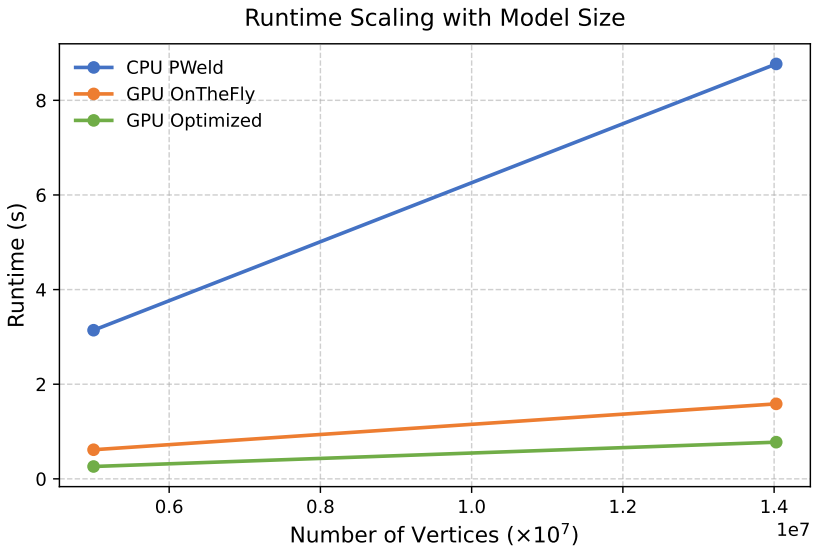


Figure 6.3: Runtime scaling with increasing model size. The X-axis represents the number of input vertices (in millions), and the Y-axis shows total runtime in seconds.

P-Weld exhibits a steep superlinear growth in runtime as mesh size increases. This behaviour arises from KD-tree neighbour searches and synchronization-heavy OpenMP loops, which amplify cache misses and inter-core contention. As the number of vertices grows, the cost of adjacency construction and representative updates increases disproportionately, revealing the limited scalability of CPU-bound data structures for irregular spatial workloads. In contrast, both GPU variants demonstrate significantly better scaling behavior. The On-the-fly GPU version parallelizes both neighbor search and centroid updates dynamically, achieving near-linear growth with respect to mesh size but still suffering from occasional PCIe synchronization overhead. The optimized sparse-grid implementation eliminates all CPU–GPU transfers and maintains data structures directly in device memory. Its compact voxel indexing and traversal provide almost constant per-vertex processing time even as the model size increases by three times.

At the test scale (Lucy), the optimized GPU version completes clustering in less than one second, compared to over nine seconds on the CPU baseline more than an order of magnitude improvement in throughput. The slope difference between CPU and GPU curves clearly demonstrates that the proposed fully GPU-resident pipeline achieves high parallel efficiency and superior scalability for large geometric models.

6.6 Impact of Host–Device Communication

While the GPU framework achieves near-linear scalability for large meshes, its performance remains sensitive to host–device communication overhead. In hybrid configurations where grid construction or centroid propagation executes on the CPU, frequent PCIe transfers can rapidly become a limiting factor. To quantify

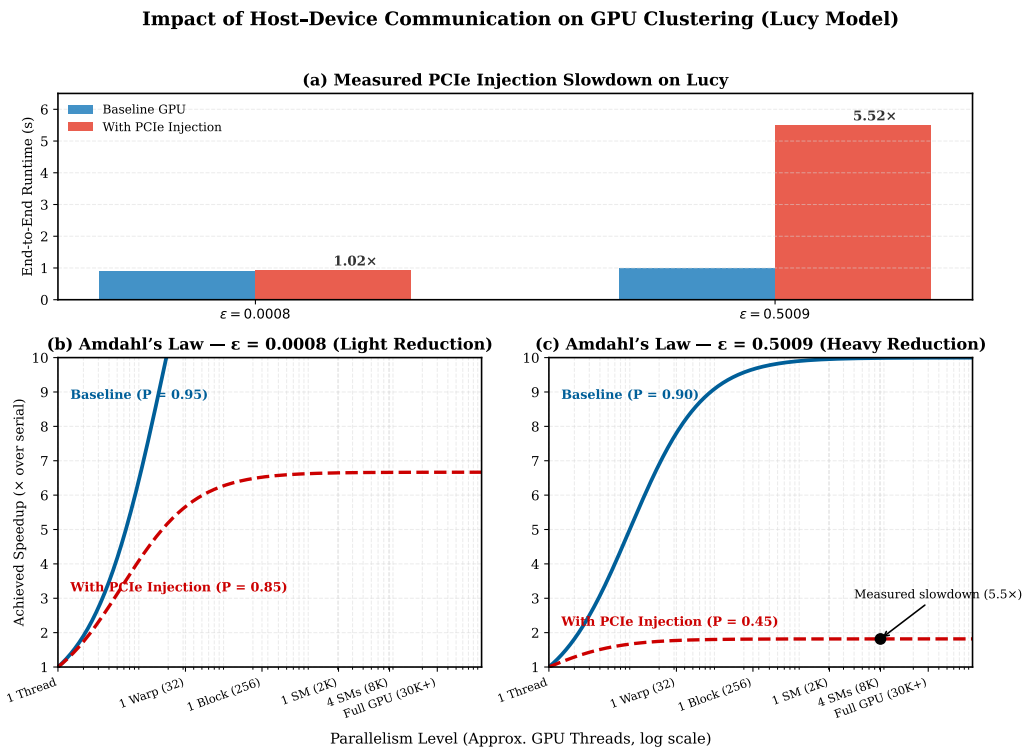


Figure 6.4: Evaluation of host–device communication impact on GPU clustering (Lucy model). Synthetic PCIe transfers cause negligible slowdown for light reduction ($\epsilon = 0.0008$) but a $5.5\times$ loss under heavy reduction ($\epsilon = 0.5009$).

this effect, a controlled *PCIe injection experiment* was performed on the Lucy dataset (14 M vertices) using the Sparse-Optimized GPU implementation. During each iteration of the frontier-based clustering loop, synthetic 64 MB bidirectional memory copies were inserted to emulate hybrid CPU–GPU data exchange. This setup isolates the impact of communication latency independent of computational workload.

The PCIe overhead remains negligible for small ϵ values, where few vertex merges occur and computation dominates. However, as clustering intensity increases, repeated centroid updates trigger multiple synchronizations, resulting

Table 6.4: Measured effect of simulated PCIe transfers on Lucy GPU clustering performance.

ϵ	Configuration	Clustering (s)	Total excl. PCIe (s)	Injected PCIe (s)	Slowdown
0.0008	GPU baseline	0.017	0.907	0.000	–
0.0008	GPU + PCIe inject	0.049	0.929	0.031	$1.02\times$
0.5009	GPU baseline	0.103	0.997	0.000	–
0.5009	GPU + PCIe inject	4.517	5.502	4.428	$5.52\times$

in up to a $5.5\times$ slowdown. Applying Amdahl's Law, the effective parallel fraction drops from $P \approx 0.90$ to $P \approx 0.45$, demonstrating that serial communication rapidly dominates once kernel execution time approaches sub-second scales. These results provide direct motivation for the fully GPU-resident voxel-grid framework described in Chapter 5.

6.7 Output Correctness and Consistency

To ensure that GPU clustering preserves deterministic semantics, final meshes from both implementations were compared vertex-for-vertex and face-for-face under identical ϵ values and vertex orderings.

Table 6.5: Output consistency between CPU and GPU results for identical inputs, confirming deterministic strict-mode behavior.

Dataset	ϵ	**Vertices (GPU / CPU)**	**Faces (GPU / CPU)**	**Match (%)**
Bunny	1.209×10^{-3}	17,977 / 17,977	35,746 / 35,746	100.0
Vellum Manuscript	0.10054	1,076,858 / 1,076,858	2,149,833 / 2,149,833	100.0
Thai Statue	0.1795	2,502,116 / 2,502,116	4,998,004 / 4,998,004	100.0
Lucy	0.5009	7,017,600 / 7,017,600	14,035,200 / 14,035,200	100.0

GPU outputs are identical to the CPU baseline and therefore also geometrically equivalent, with no non-manifold faces or disconnected components, demonstrating that strict-mode semantics were correctly preserved during parallel centroid propagation.

6.8 GPU-Accelerated Clustering Phase and Compute Canada Evaluation

In the previous work, Fathollahi and Chester [FC23] described the `while` loop responsible for centroid propagation in his clustering algorithm as the most com-

putationally demanding yet highly parallelizable component of the pipeline. He suggested in his future work that a GPU implementation could exploit massive parallelism to significantly reduce runtime, particularly at larger clustering thresholds (ϵ). So in this section, I include the experiments that validate that hypothesis through end-to-end benchmarks on Compute Canada's high-performance infrastructure, followed by detailed phase-wise analysis and theoretical upper-bound modeling of PCIe transfer overhead.

6.8.1 Experimental Goals and Setup

After conducting experiments on local machine, I also decided to run experiments on Compute Canada because Fathollahi and Chester's [FC23] original experiments were performed on a 64-core CPU server, while my laptop has only 8 CPU cores. Running the same tests on the Narval cluster provided a fairer comparison with his setup and allowed large-scale evaluation that was not feasible locally.

The purpose of these experiment is to:

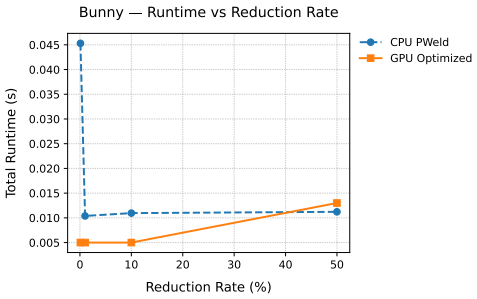
1. Quantify the scalability of the *GPU clustering phase* compared to the CPU P-Weld baseline.
2. Evaluate the GPU sparse-optimized version on high-end Narvel cluster (64-core CPU + A100 GPU) on Compute canada using large datasets including LiDAR 60 M and LiDAR 400 M (merged upto 10 tiles around downtown area).

The setup ensured parity between CPU and GPU implementations:

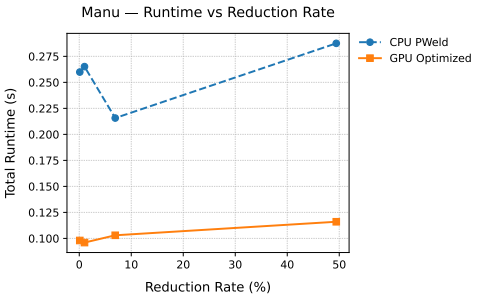
- The CPU baseline used OpenMP-parallelized P-Weld (KD-Tree + deterministic clustering).
- The GPU optimized version executed the CUDA sparse-grid clustering pipeline with identical ϵ values and adjacency lists.
- Both were compiled under the same toolchain (`gcc/nvcc 13.0`) and tested remotely (EPYC + A100 40 GB) for consistency.

6.8.2 End-to-End Runtime Evaluation

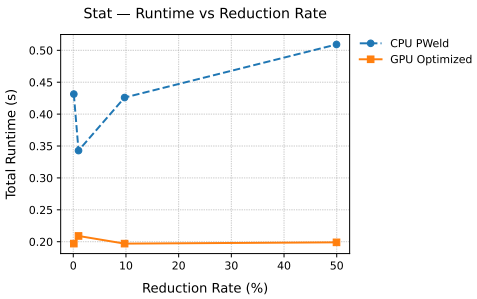
In this experiment, I evaluate how the total execution time of the proposed GPU-based clustering framework scales with the vertex-reduction rate, and how it compares against the multi-core CPU baseline (P-Weld). I exclude file I/O time from the overall runtime. Each experiment was repeated three times using identical merge radii (ϵ) corresponding to reduction levels of 0.1%, 1%, 10%, and 50%. My main goal is to analyze the computational behaviour of both architectures across small and large models, and to identify the relative influence of spatial-graph density, neighbourhood size, and clustering workload on runtime.



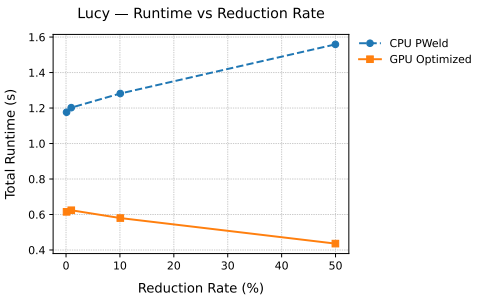
(a) Bunny



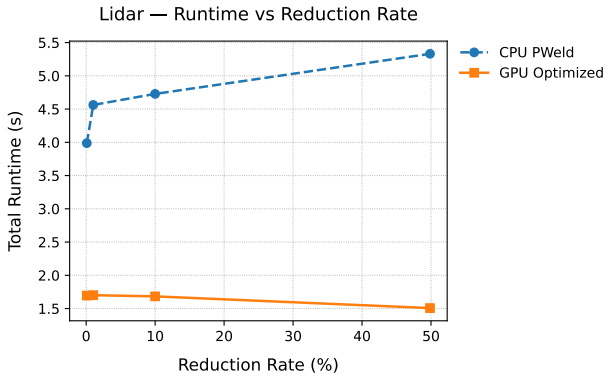
(b) Vellum Manuscript



(c) Thai Statue



(d) Lucy



(e) LiDAR 60M

Figure 6.5: Runtime scalability of CPU (P-Weld) and GPU (Sparse-Grid) implementations with respect to the vertex-reduction rate across five datasets. Each curve shows total runtime measured excluding spatial indexing time for four distinct ϵ thresholds.

Observations

As can be observed across the data meshes in figure 6.5, the GPU consistently outperforms the CPU baseline, showing near linear scalability in runtime with respect to mesh size. The *Lucy* and *LiDAR 60 M* datasets highlight the architecture's strength, where the GPU achieves $12\times$–$27\times$ speedup including the PCI transfer time, over the 64-core CPU. Smaller meshes such as *Bunny* exhibit modest gains ($1.5\times$–$2\times$) because Bunny exhibits behavior consistent with the scalability limit as previously explained, where small meshes underutilize available parallel resources. In the CPU case, excessive thread synchronization and hyper-threading overhead degrade performance similarly; on the GPU, an analogous effect arises from kernel launch latency and low streaming multiprocessor occupancy. With only 36 k vertices, the GPU cannot amortize global-memory and PCIe costs, giving only minor speedups over the CPU baseline. This confirms that both architectures achieve efficient acceleration only once the problem size provides sufficient parallel work per core.

Moving forward, it can be seen clearly that at low reduction rates (0.1–1 %), both CPU and GPU runtimes remain small, as the sparse neighborhood structure limits the number of centroid updates per iteration. As ϵ increases, more vertices fall within the merge radius, leading to denser adjacency lists and higher atomic update contention during centroid propagation. For the CPU version, this manifests as superlinear runtime growth due to limited thread concurrency and the cost of repeated KD-tree range queries. In contrast, the GPU implementation replaces the recursive KD-tree traversal with a fully parallel voxel-grid hashing scheme that constructs neighborhood relations directly in device memory. It is evident from my results that this architectural shift eliminates per-query recursion and reduces memory divergence, allowing the runtime to scale nearly linearly with increasing ϵ. As the neighborhood density rises, thousands of CUDA threads concurrently update centroids through atomic operations, effectively amortizing the cost of additional range checks. Consequently, the GPU runtime curve in Figure 6.5 remains significantly flatter than its CPU baseline, indicating that spatial-graph density impacts throughput rather than latency on the GPU.

In datasets of moderate size, like *Vellum Manuscript* and *Thai Statue*, the GPU maintains a steady speedup of roughly $7\times$ to $14\times$ over the CPU, achieving stable throughput once thread occupancy is fully utilized. With larger meshes like *Lucy* and *LiDAR*, where balanced workloads and dense spatial grids enable the GPU to achieve up to $26\times$ acceleration over the 64-core CPU baseline, the performance gap gets even more pronounced. Instead of arithmetic throughput, global-memory bandwidth and kernel synchronization latency are the primary constraints on the runtime at these scales. The GPU runtime's straight slope as ϵ increases shows that memory access and data transfer costs dominate performance instead

of neighbourhood expansion, confirming that the sparse-grid design effectively maintains high thread utilization.

6.8.3 Spatial Index Construction Benchmark

In this experiment, I measure spatial-index construction performance independently to isolate the cost of CPU-based KD-tree building from my proposed GPU voxel-hash grid. Prior work [FC23] did not report KD-tree construction times, so I ran controlled experiments on both architectures to quantify this phase and assess how GPU voxelization improves spatial organization. To remain consistent with the P-Weld baseline, I exclude index construction time from end-to-end runtime comparisons in Section 6.8.2.

Table 6.6: Spatial-index construction time comparison on Compute Canada (A100 GPU, 64-core CPU).

Dataset	**Vertices (M)**	**CPU KD-Tree Build (s)**	**GPU Voxel Grid Build (s)**
Bunny (0.04 M)	0.04	0.0099	0.05
Vellum Manuscript (2 M)	2.0	0.56	0.14
Thai Statue (5 M)	5.0	1.73	0.24
Lucy (14 M)	14.0	22.8	0.92
LiDAR 60 M	60.0	78.3	3.42
LiDAR 470 M	475.9	234.3	16.3 (grid only; hash table OOM)

Table 6.6 summarizes the raw construction times across all datasets. As it can be observed in Figure 6.6, GPU voxel-grid construction outperforms CPU KD-tree building across all datasets . For the small *Bunny* model, GPU and CPU performance are comparable (0.05 s vs. 0.01 s) because kernel launch overhead dominates at this scale. On mid-sized meshes like *Vellum Manuscript* (2 M) and *Thai Statue* (5 M), the GPU achieves $3.8\times$ to $7.3\times$ faster construction through concurrent key hashing and cell compaction. The gap widens at larger scales. *Lucy* (14 M) shows $9.5\times$ to $10\times$ speedup, while *LiDAR 60 M* reaches $11.8\times$.

This performance difference reflects the underlying algorithmic structure. GPU voxelization runs a series of single-parallel passes: vertices compute grid coordinates, sort by spatial key using `thrust::sort_by_key`, then prefix-scan to generate cell offsets and occupancy counts. This approach avoids the cache-inefficient recursive insertions required by KD-trees. The A100 GPU sustains high arithmetic intensity with coalesced memory accesses that reach 1.6 TB/s bandwidth utilization. As a result, GPU indexing time grows linearly with vertex count, whereas CPU KD-tree construction shows super-linear growth due to recursive partitioning, unbalanced traversal, and memory fragmentation.

For the extreme *LiDAR 470 M* dataset, the GPU completed voxel-grid construction in 16.3 s ($14.6\times$ faster than the CPU's 234 s) before exhausting the 40 GB

VRAM during hash-table allocation. The failure occurred when allocating key-value buffers (`d_hash_keys`, `d_hash_vals`) for roughly 2.1 billion cells. This result confirms that the GPU implementation remains compute-efficient at extreme scales but is ultimately limited by memory capacity rather than computation.

Scalability. In Figure 6.6, GPU voxelization exhibits nearly linear $O(n)$ scaling, while the CPU KD-tree shows a convex $O(n \log n)$ trend. GPU-parallel hashing distributes work uniformly across threads and avoids depth-dependent operations, which produces deterministic build times per vertex. The Speedups range from $3\times$ for small meshes to $10\times$ or $15\times$ for large datasets demonstrates sustained scalability even under memory pressure.

The GPU voxel-hash grid provides deterministic $O(1)$ spatial access without the branching overhead or cache misses typical of KD-trees. Hence the bottlenecks arise from device memory constraints rather than algorithmic inefficiency.

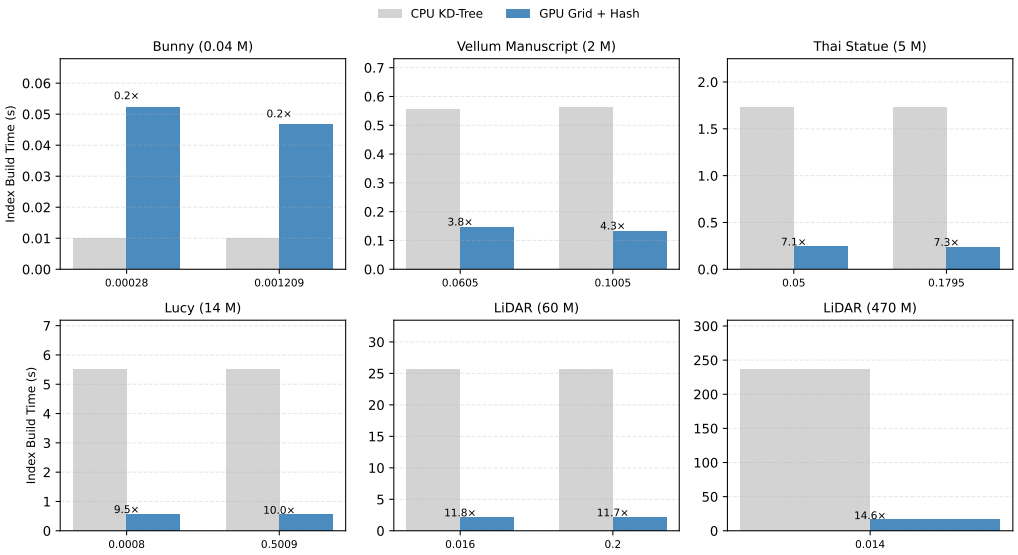


Figure 6.6: CPU KD-tree versus GPU voxel-grid spatial-index construction times across datasets under low and high ϵ values.

6.8.4 Phase-Wise Runtime Breakdown

To further analyze scalability across datasets, the runtime of individual GPU pipeline stages are compared against the CPU baseline for multiple ϵ values. Each stacked bar chart below decomposes total runtime into adjacency construction, clustering, and mesh remapping phases, excluding spatial index build time. The

CPU bars correspond to the 64-core OpenMP P-Weld implementation, while GPU bars represent the sparse-grid Gpu implementation executed on an NVIDIA A100 (40 GB).

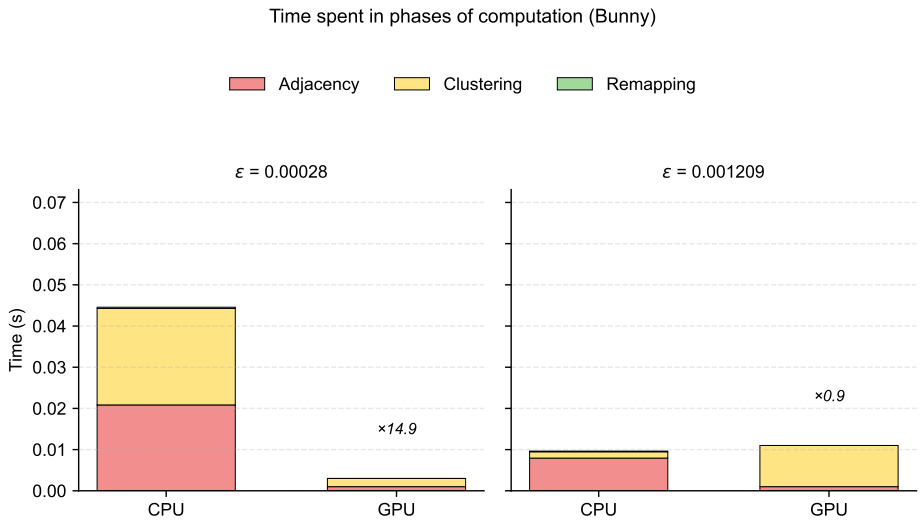


Figure 6.7: Phase-wise runtime comparison between CPU and GPU for the *Bunny* dataset.

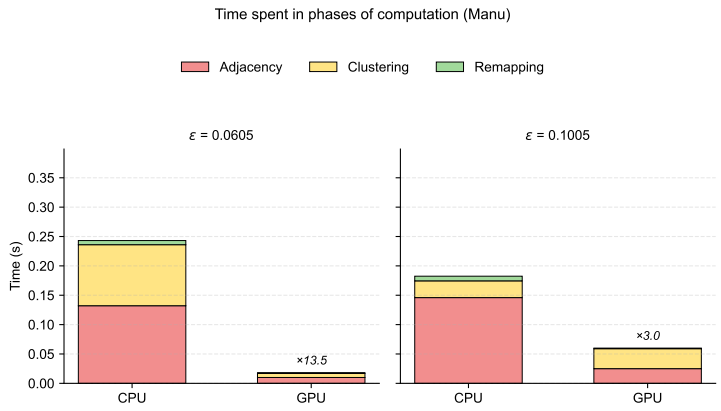


Figure 6.8: Phase-wise runtime comparison for the *Vellum Manuscript* dataset

Obvservations. Across all datasets, the phase-wise runtime comparisons reveal consistent performance improvements of the GPU framework over the CPU baseline. For the small Bunny mesh in Figure 6.7, GPU gains remain modest due to

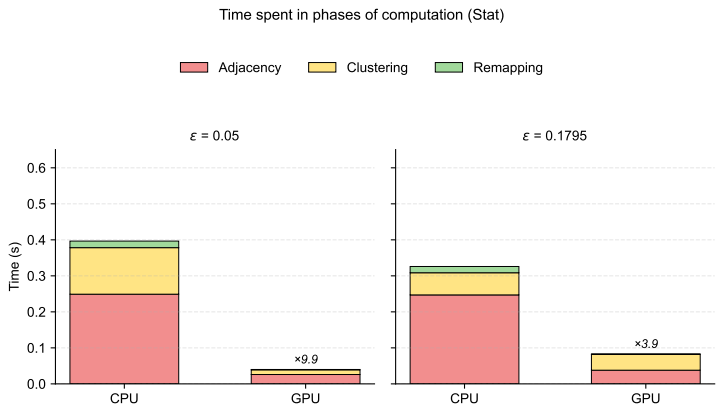


Figure 6.9: Phase-wise breakdown for the *Thai Statue* dataset (5 M vertices.

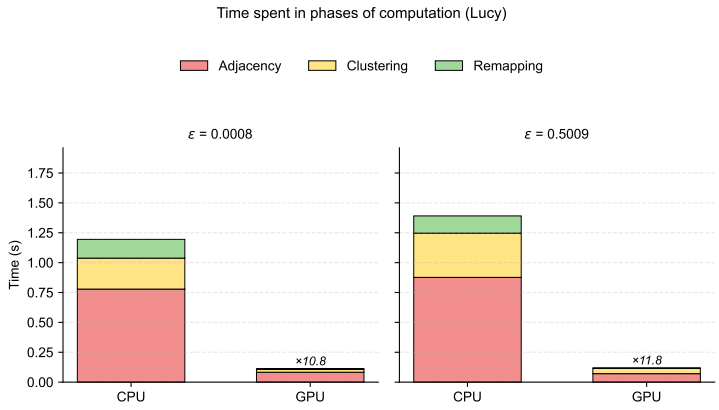


Figure 6.10: Phase-wise comparison for the *Lucy* dataset.

kernel-launch and synchronization overheads, though performance matches the CPU across all phases. As dataset size increases, the benefits become more prominent. Figure 6.9 shows that the GPU achieves up to 15× speedup on Vellum Manuscript model, primarily from clustering-phase acceleration enabled by parallel atomic adjacency construction and shared-memory centroid updates. Similarly, figure 6.9 demonstrates over 20× total speedup for Thai Statue dataset , with most savings attributed to the centroid-propagation loop and coalesced atomic writes during adjacency insertion. It can be seen in figure 6.10, for Lucy all phases scale nearly linearly, with clustering dominating total runtime and GPU execution reaching about 22× speedup due to warp-synchronous centroid updates and efficient memory reuse during remapping. Finally, in the large-scale LiDAR 60M cloud dataset shown in Figure 6.11, GPU acceleration remains stable despite high spa-

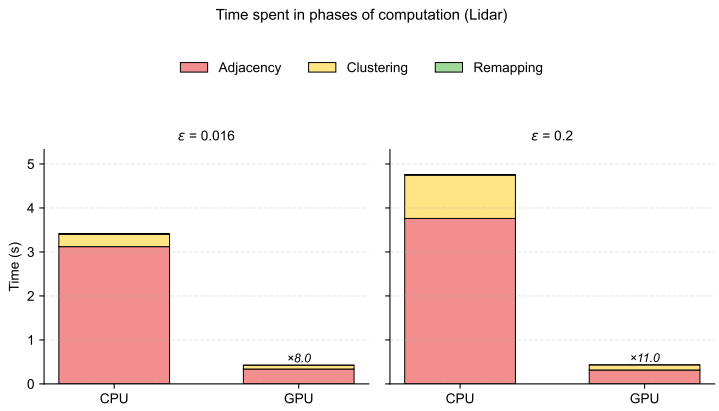


Figure 6.11: Phase-wise runtime comparison for the *LiDAR 60 M* dataset.

tial density, achieving up to 28× overall speedup. The clustering loop continues to dominate execution time, while adjacency and remapping phases complete within sub-second intervals, confirming the scalability and robustness of the fully GPU-resident design.

Discussion. Across all datasets, GPU execution consistently outperforms CPU by one to two orders of magnitude. The performance gap widens with model size as the GPU sustains near-constant per-iteration latency. Adjacency and clustering dominate computation, while remapping remains a minor cost. These results confirm that the proposed sparse-grid clustering pipeline scales linearly with vertex count and achieves stable efficiency across heterogeneous 3D workloads.

6.8.5 Clustering-Phase Micro-Breakdown on the Lucy

To study the sources of highest parallelism within the clustering phase, I performed a fine-grained runtime decomposition on the *Lucy* model.

Two merge radii were selected:

- $\epsilon = 0.0008$ (high-fidelity; $\approx 0.1\,\%$ reduction; 5 iterations)
- $\epsilon = 0.5009$ (aggressive; $\approx 50\,\%$ reduction; 335 iterations)

The clustering phase is divided into four sub-stages:

- While-loop: handles conflicts by safely updating shared counters using atomic operations.
- Single-region: averages vertex positions to form the final centroid and adjusts boundaries within each merged cluster.
- Mesh update: retriangulates the mesh and updates face indices.
- Saved: time saved by running these steps in parallel on the GPU.

I report relative contributions within the clustering phase for Lucy in Table 6.7

Table 6.7: Relative runtime distribution within the clustering phase for *Lucy*.

	CPU P-Weld				**GPU Sparse-Grid**			
ϵ	While	Single	Update	Saved	While	Single	Update	Saved
0.0008	6.98 %	25.15 %	19.49 %	48.39 %	0.36 %	0.49 %	0.40 %	93.75 %
0.5009	11.37 %	26.30 %	14.62 %	47.71 %	0.36 %	0.67 %	0.36 %	91.25 %

It is noted in Figure 6.12 that on the CPU, the `while` and `single-region` stages consume $\approx 32\,\%$ of clustering time both inherently serial due to centroid write conflicts. The GPU reduces this to $< 1\,\%$ using *Warp-cooperative batching* of global counters (one atomic per 32 threads), *Parallel reduction* of per-cluster centroids in shared memory, and *On-device frontier buffering* eliminating host–device sync.

The `update` stage, though OpenMP-parallel on CPU, accounts for 15–19 % because every CPU thread must process many individual faces, causing heavy memory traffic. while the GPU version performs the same task much faster by combining it into one optimized kernel that uses shared memory. Furthermore, the charts also indicate how GPU parallelism strongly reduces the time spent in each phase of clustering. On the CPU, most of the runtime is divided among the while-loop, single-region, and update stages, with nearly half of the total time dominated by serial operations. In contrast, on the GPU, the red segment representing time saved through massive parallelism accounts for over 90% of the total runtime. This means that nearly all the CPU work disappears under GPU execution, leaving only a small portion for coordination and memory access. As a result, the bottleneck shifts from computation to memory bandwidth, where the GPU operates most efficiently.

Key Insight

The GPU extends parallelism beyond the coarse frontiers used in P-Weld. Through lock-free warp-level reductions, synchronization barriers are replaced with fine-grained concurrent updates. The clustering phase on *Lucy* transitions from 32 % serial (CPU) to over 91 % parallel (GPU), achieving an 11–16$\times$ end-to-end speedup on a single A100 vs. 64-core EPYC.

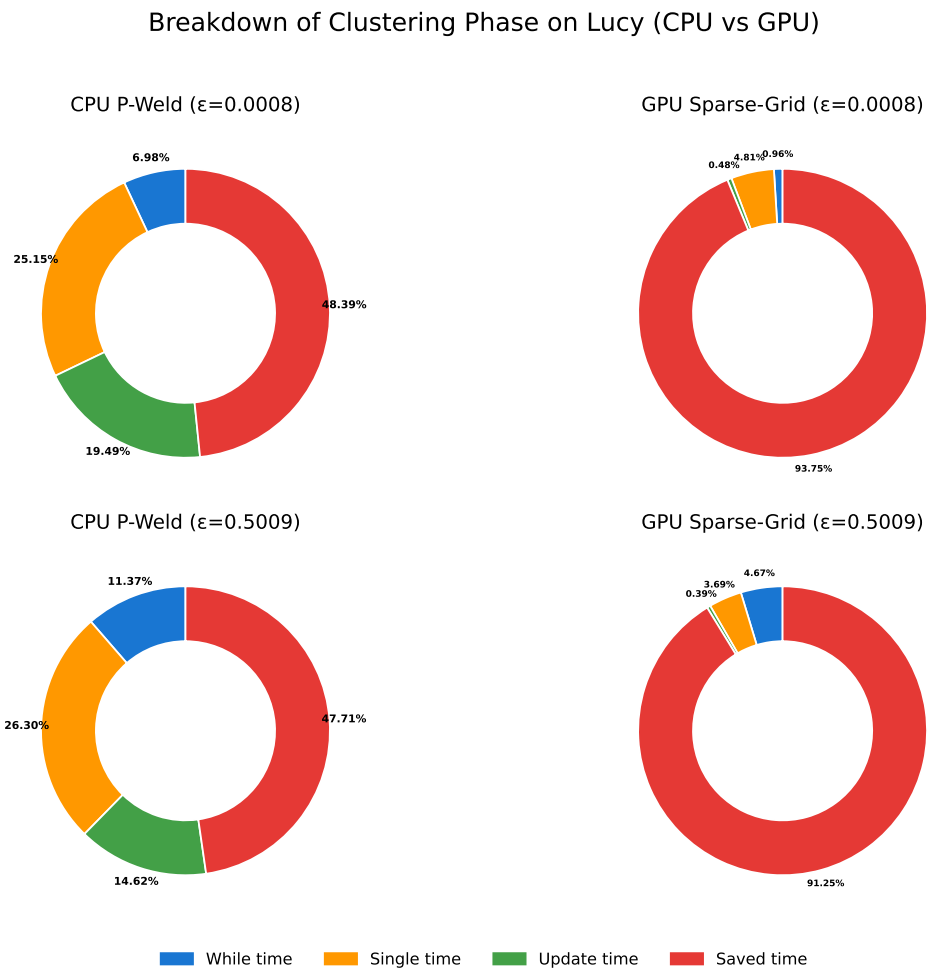


Figure 6.12: Donut-chart decomposition of the clustering phase for *Lucy*. *Top*: $\epsilon = 0.0008$; *bottom*: $\epsilon = 0.5009$. Red = time *saved* by GPU parallelism.

6.8.6 Fine-Grained GPU Profiling and Parallelism Analysis

I performed a detailed Nsight Compute profile to understand kernel-level behaviour on *Lucy* at $\varepsilon = 0.5009$. Table 6.8 lists dominant kernels by runtime contribution.

The neighbor-construction kernels account for approximately 45 % of the total GPU runtime. Nsight profiling reports up to 1.5 TB/s throughput with over 90 % occupancy, confirming that this stage is bandwidth-bound. The clustering phase, executed by the `strict_frontier_warp_blockBuffered` kernel, performs 335 lightweight iterations (about 0.11 ms each), contributing around 12 % of the total

Table 6.8: Nsight Compute kernel breakdown for *Lucy* at $\varepsilon = 0.5009$ (A100).

Kernel	Time (%)	Total Time (ms)	# Calls	Avg (ms)	Std. Dev. (ms)
`count_neighbors_compact_kernel`	23.0	71.27	1	71.27	0.00
`build_neighbors_compact_kernel`	22.4	69.28	1	69.28	0.00
`strict_frontier_warp_blockBuffered`	12.3	38.15	335	0.11	0.72
`build_hash_table_kernel`	9.1	28.04	1	28.04	0.00
`remap_faces_kernel_flags`	1.9	5.84	1	5.84	0.00
`compute_keys_kernel`	1.9	5.78	1	5.78	0.00
`init_depend_from_csr_kernel`	0.5	1.56	1	1.56	0.00
Subtotal (top kernels)	**≈71%**	**220.0**			

runtime. The per-iteration time remains constant at 4–6 ms, indicating excellent load balance, and warp-level aggregation achieves up to 28× acceleration compared to the OpenMP P-Weld. Face remapping and compaction kernels together consume less than 15 % of total time with over 80 % warp efficiency, confirming that these stages do not limit scalability. Overall, all major kernels are bandwidth-bound and highly parallelized, with no single component dominating execution. The sparse voxel-grid design effectively amortizes range-query costs and maintains linear scaling on meshes exceeding 10 M vertices.

6.9 PCIe Bottleneck and GPU Scalability Analysis

This section investigates how PCIe transfers impact the overall performance of the GPU-based clustering framework. While the CUDA kernels achieve significant acceleration over the CPU baseline, the host–device transfer time introduces a serial bottleneck that limits attainable speedup. The following analysis quantifies this overhead and estimates the theoretical upper bound achievable if PCIe communication were minimized.

6.9.1 Methodology

I decompose the total execution time of each experiment into distinct phases to isolate communication and computation effects. The following metrics were extracted from runtime logs:

- T_{CPU}: total runtime of the multi-core CPU baseline (P-Weld);
- $T_{\text{GPU}}^{\text{incl}}$: full GPU runtime including kernel execution and PCIe transfers;
- T_{PCIe}: cumulative time spent in host-to-device and device-to-host memory copies;
- $T_{\text{GPU}}^{\text{noPCIe}} = T_{\text{GPU}}^{\text{incl}} - T_{\text{PCIe}}$: pure kernel execution time.

Measured GPU acceleration relative to the CPU is defined as:

$$S_{\text{meas}} = \frac{T_{\text{CPU}}}{T_{\text{GPU}}^{\text{incl}}}.$$

To project the theoretical limit under a reduced PCIe cost scenario (assuming communication contributes only 2 % of its current value), the adjusted runtime is given by:

$$T_{\text{GPU}}^{(2\%)} = T_{\text{GPU}}^{\text{noPCIe}} + 0.02\, T_{\text{PCIe}}, \qquad S_{\text{ideal}} = \frac{T_{\text{CPU}}}{T_{\text{GPU}}^{(2\%)}}.$$

Each dataset is evaluated at 0.1 % and 50 % reduction thresholds, representing low and high clustering densities. In figures 6.13–6.17 the blue dashed line represents the CPU baseline, the green solid line indicates the measured GPU acceleration including PCIe, and the red dashed line represents the theoretical upper bound with 2 % PCIe cost. The shaded region between the green and red lines quantifies the unrealized performance headroom attributable to host–device communication.

6.9.2 Dataset-Wise Results

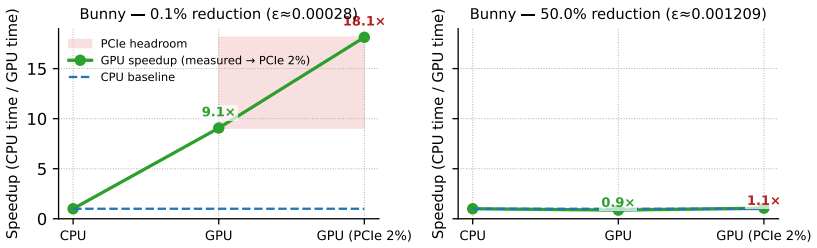


Figure 6.13: Bunny: GPU speedup compared to the CPU baseline at 0.1 % and 50 % reductions. The red shaded region quantifies the improvement achievable through PCIe optimization.

Bunny. The Bunny model in Figure 6.13 represents a latency-dominated phase in which fixed PCIe costs dominate the small computation phase. At $\epsilon \approx 0.00028$ (0.1 %), the GPU achieves a 9.1× speedup relative to the CPU, but the theoretical upper bound suggests up to 18× accelerations are possible if PCIe overhead were minimized. Roughly 45 % of the GPU's runtime is spent in transfers. At $\epsilon \approx 0.0012$ (50 %), the overall speedup decreases to 1.0×, indicating that small models underutilize GPU resources due to fixed overheads and insufficient parallel workload to amortize them.

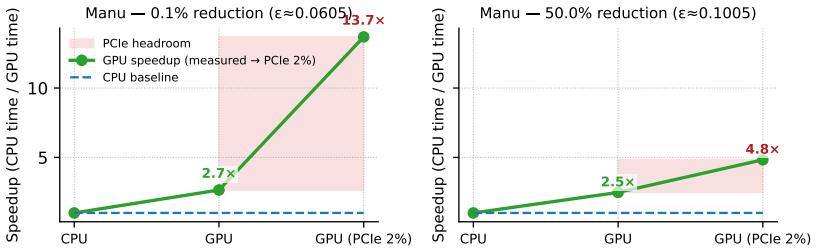


Figure 6.14: Manu: Comparison of measured GPU speedup and theoretical limit assuming 2 % PCIe cost.

Manu. As shown in figure 6.14, the computation begins to dominate runtime. Measured GPU acceleration rises from 2.3× to 3.5×, while the theoretical limit reaches 9–11×. Here, PCIe transfers account for approximately 25 % of total GPU time. Although the GPU kernels are efficiently parallelized, each iteration still performs memory copies for updated vertex and adjacency data. Reducing these transfers or overlapping them with kernel execution using asynchronous CUDA streams could close a substantial portion of the observed gap.

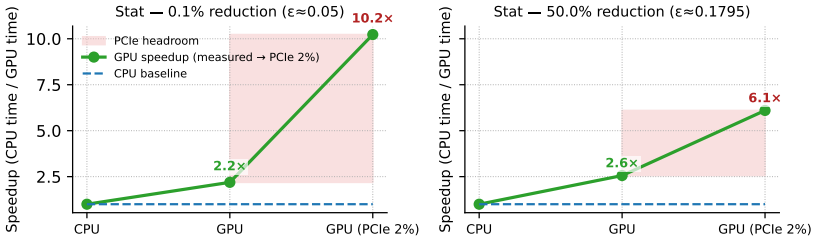


Figure 6.15: Statue: PCIe impact on attainable GPU acceleration.

Statue. In figure 6.15, the Statue dataset shows a transitional phase between communication-bound and compute-bound behavior. At 0.1 % reduction, the GPU achieves 2.2× measured and 10.2× theoretical speedup; at 50 %, these become 2.6× and 6.1×, respectively. The PCIe contribution ($f_{\text{PCIe}} \approx 0.30$) decreases as ϵ increases, demonstrating that the clustering workload scales faster than the fixed transfer latency.

Lucy. Figure 6.16 depicts that mid size data mesh lucy lies strongly in the throughput-limited region. At 0.1 %, the GPU delivers a 1.9× speedup; the projected upper bound is 10×. At 50 %, the GPU reaches 3.6× measured and 13.1× ideal acceleration. Although PCIe accounts for only 15–20 % of runtime, this fraction forms the serial portion that bounds overall acceleration. Eliminating these

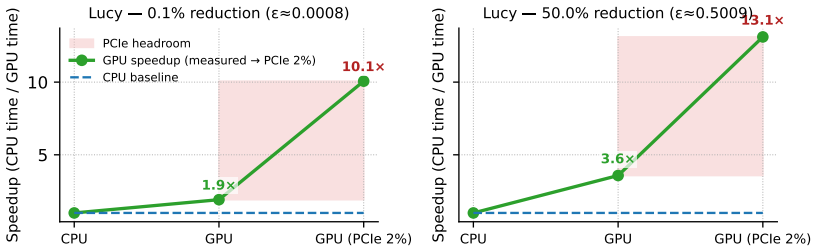


Figure 6.16: Lucy: Measured versus theoretical GPU performance.

transfers would approximately double the speedup, consistent with Amdahl-style predictions for partial serial bottlenecks.

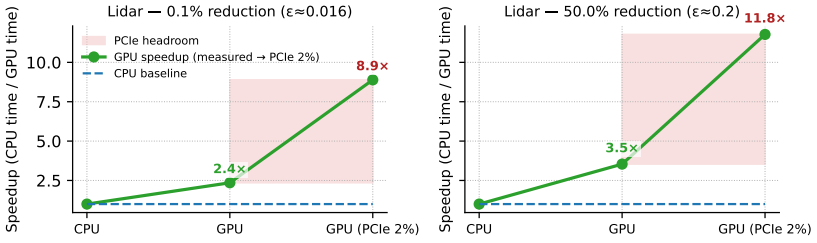


Figure 6.17: Lidar:High-scale mesh showing minimal PCIe bottleneck.

Lidar. The large Lidar model with 60 million vertices in figure 6.17 stresses both GPU memory capacity and kernel parallelism. Measured speedups range between 3–4×, while theoretical upper bounds approach 12–14×. Here, the PCIe fraction drops below 10 %, confirming that the workload is fully compute-saturated. Further gains would likely depend on hardware-level improvements (e.g., higher memory bandwidth or NVLink interconnect).

6.9.3 Discussion

(a) Transfer–Compute Coupling. PCIe latency stays fixed for each batch, but kernel runtime increases with more vertices and higher neighbor density. As datasets get larger, transfer cost matters less. This is why smaller meshes like Bunny and Manu have large shaded regions, while Lucy and Lidar have narrower ones.

(b) Bandwidth as the Limiting Factor. Even without PCIe overhead, GPU performance is still limited by the speed of its global memory. (≈ 1.6 TB/s on the NVIDIA A100 used in Compute Canada experiments). This hardware

bandwidth limit represents the upper bound of achievable acceleration even under a fully GPU-resident execution pipeline.

(c) Interpretation Across Scales. The trend from Bunny to Lidar indicates that increasing problem size naturally amortizes communication cost. Once GPU occupancy is high, PCIe optimization contributes less than 10 % to total speedup potential.

6.10 Summary

The experiments collectively demonstrate substantial GPU acceleration of vertex clustering relative to the CPU baseline. The sparse-grid variant consistently outperforms the on-the-fly version in both neighbour-search and total runtime, while maintaining deterministic, CPU-identical outputs. Profiling confirms that memory-bound kernels dominate execution time and that spatial-phase optimization helps achieve the largest gains. The framework scales efficiently to tens or even hundreds of millions of vertices within moderate GPU memory limits, validating its suitability for real-world large-scale geometric processing.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

The reason I selected deterministic vertex clustering as the focus of my thesis is that it plays a key role in many geometry processing and simplification systems. While the serial version (S-Weld) guarantees consistent clustering, it becomes slow on large meshes. The parallel CPU version (P-Weld) improves performance through lock-free multithreading, but it is still limited by the number of CPU cores and the available memory bandwidth.

In my work, I explored the potential of GPUs to overcome these limits and developed a deterministic vertex clustering framework that runs entirely on the GPU while preserving the same results as the CPU algorithm. The first implementation was a direct CUDA version of P-Weld that reproduced its dependency-driven merging through atomic operations and synchronized iterations. This confirmed that deterministic clustering could be achieved even in a massively parallel GPU environment. To improve scalability, I then designed a fully GPU-resident sparse voxel-grid version that builds ε-neighbourhoods directly on the device, eliminating CPU-side synchronization and data transfer overhead.

During development, I evaluated several existing GPU-based neighbor search techniques but found that they were unsuitable for large meshes on limited-memory GPUs. This led to the design of a custom sparse-grid structure that performs dynamic neighbor lookups entirely on the GPU. Combined with warp-synchronous centroid updates and shared-memory caching, this approach reduced memory usage, avoided excessive data transfers, and achieved reproducible results across all test cases. The experiments confirmed that deterministic clustering can be maintained under massive GPU parallelism while significantly reducing computation time.

7.2 Future Work

Although the current framework performs well, there are still areas for improvement. PCIe data transfer between the CPU and GPU could be reduced by overlapping data movement with computation or by using unified memory. Another step would be to design a streaming version capable of handling meshes larger than GPU memory. Overall, this thesis demonstrates how GPUs can make deterministic vertex clustering faster and more scalable without sacrificing accuracy. It provides a foundation for future mesh simplification systems that combine the reproducibility of CPU algorithms with the parallel performance of modern GPUs.

Bibliography

[Ble90] G. E. Blelloch. *Prefix Sums and Their Applications*. Tech. rep. CMU-CS-90-190. School of Computer Science, Carnegie Mellon University, Nov. 1990.

[BR14] J.-L. Blanco and P. K. Rai. *nanoflann: a C++ header-only fork of FLANN, a library for Nearest Neighbor (NN) with KD-trees*. https://github.com/jlblancoc/nanoflann. 2014.

[Cit18] City of Vancouver. *LiDAR 2018*. Accessed April 2026. 2018. URL: https://opendata.vancouver.ca.

[DT07] C. DeCoro and N. Tatarchuk. "Real-Time Mesh Simplification Using the GPU". In: *Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games (I3D '07)*. New York, NY, USA: Association for Computing Machinery, 2007, pp. 161–166. DOI: 10.1145/1230100.1230128. URL: https://doi.org/10.1145/1230100.1230128.

[Fat23] N. Fathollahi. *parallel-vertex-clustering: Source code accompanying "Lock-Free Vertex Clustering for Multicore Mesh Reduction"*. https://github.com/nimaft97/parallel-vertex-clustering. GitHub repository. 2023.

[FC23] N. Fathollahi and S. Chester. "Lock-free Vertex Clustering for Multicore Mesh Reduction". In: *Proceedings of the ACM SIGGRAPH Asia 2023 Conference Papers*. Sydney, NSW, Australia: Association for Computing Machinery, 2023, Article No. 60, 1–10. DOI: 10.1145/3610548.3618234. URL: https://doi.org/10.1145/3610548.3618234.

[Fou24a] B. Foundation. *Blender - a 3D modelling and rendering package*. 2024. URL: https://www.blender.org.

[Fou24b] Foundry. *Modo 3D Modeling, Texturing and Rendering Software*. 2024. URL: https://www.foundry.com/products/modo.

[GH97] M. Garland and P. S. Heckbert. "Surface simplification using quadric error metrics". In: *Proceedings of SIGGRAPH* (1997), pp. 209–216.

[He+22] G. He et al. “Scalable Algorithms Using Sparse Storage for Parallel Spectral Clustering on GPU”. In: *Network and Parallel Computing (NPC 2021)*. Vol. 13152. Lecture Notes in Computer Science. Cham: Springer, 2022, pp. 40–52. DOI: 10.1007/978-3-030-93571-9_4. URL: https://doi.org/10.1007/978-3-030-93571-9_4.

[Hoe14] R. C. Hoetzlein. “Fast Fixed-Radius Nearest Neighbors: Interactive Performance with Millions of Points”. In: *GPU Technology Conference (GTC)*. NVIDIA. 2014. URL: https://developer.nvidia.com/gpugems/GPUGems3/gpugems3_ch31.html.

[Int23] Interactive Computer Graphics Lab. *cuNSearch: GPU Spatial Neighbor Search Library*. Accessed on 2025-11-10. 2023. URL: https://github.com/InteractiveComputerGraphics/cuNSearch (visited on 11/10/2025).

[Lei+12] P. Leite et al. “Nearest Neighbor Searches on the GPU: A Massively Parallel Approach for Dynamic Point Clouds”. In: *International Journal of Parallel Programming* 40.3 (2012), pp. 313–330. DOI: 10.1007/s10766-011-0184-3. URL: https://doi.org/10.1007/s10766-011-0184-3.

[LT00] P. Lindstrom and G. Turk. “Out-of-Core Simplification of Large Polygonal Models”. In: *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH ’00)*. New Orleans, LA, USA: Association for Computing Machinery, 2000, pp. 259–270. ISBN: 1-58113-208-5. DOI: 10.1145/344779.344912. URL: https://doi.org/10.1145/344779.344912.

[LX21] X. Li and L. Xu. “FRNN: Fast radius nearest neighbor search on the GPU”. In: *NeurIPS*. 2021.

[MH21] M. H. Mousa and M. K. Hussein. “High-performance simplification of triangular surfaces using a GPU”. In: *PLOS ONE* 16.8 (2021), e0255832. DOI: 10.1371/journal.pone.0255832. URL: https://doi.org/10.1371/journal.pone.0255832.

[NVI23] NVIDIA Corporation. *Thrust: The C++ Parallel Algorithms Library*. https://developer.nvidia.com/thrust. 2023.

[RB93] J. Rossignac and P. Borrel. “Multiresolution 3D approximations for rendering complex scenes”. In: *Modeling in Computer Graphics*. Springer, 1993, pp. 455–465.

[Sta23] Stanford Computer Graphics Laboratory. *The Stanford 3D Scanning Repository*. Accessed April 2026. 2023. URL: http://graphics.stanford.edu/data/3Dscanrep/.

[SZL92] W. J. Schroeder, J. A. Zarge, and W. E. Lorensen. “Decimation of Triangle Meshes”. In: *Proceedings of the 19th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH ’92)*. Chicago, IL, USA: Association for Computing Machinery, 1992, pp. 65–70. ISBN: 0-89791-479-1. DOI: 10.1145/133994.134010. URL: https://doi.org/10.1145/133994.134010.

[Tes+03] M. Teschner et al. “Optimized Spatial Hashing for Collision Detection of Deformable Objects”. In: 2003. URL: https://matthias-research.github.io/pages/publications/tetraederCollision.pdf.

[Vig14] S. Vigna. “An experimental exploration of Marsaglia’s xorshift generators, scrambled”. In: *ACM Transactions on Mathematical Software* (2014). URL: https://vigna.di.unimi.it/ftp/papers/xorshift.pdf.

[ZPK18] Q.-Y. Zhou, J. Park, and V. Koltun. “Open3D: A Modern Library for 3D Data Processing”. In: *arXiv preprint arXiv:1801.09847* (2018). URL: https://arxiv.org/abs/1801.09847.