www.cad-journal.net
Computer-Aided Design AND APPLICATIONS™

# GPU and Multi-GPU Point-Cloud Processing with Applications in Reverse Engineering

Alexander Agathos[1] and Philip Azariadis[2]

[1]University of West Attica, alexander.agathos@gmail.com
[2]University of West Attica, fazariadis@uniwa.gr

Corresponding author: Philip Azariadis, fazariadis@uniwa.gr

**Abstract.** Point-cloud processing is central to reverse-engineering workflows, but the scale of modern scan data makes neighborhood queries a dominant computational bottleneck. This keynote paper reviews the authors' grid-based GPU and multi-GPU framework for exact k-nearest-neighbor (KNN) search and its use in ICP registration, elliptic Gabriel–based Taubin smoothing, and point-cloud normal estimation and orientation. Experiments on point sets containing up to 100.8 million samples demonstrate substantial acceleration over parallel CPU implementations and established software packages, while also revealing the communication and replicated-data costs that limit scaling. The same neighborhood infrastructure supports two higher-level operations performed directly on raw point clouds: optimal point-to-point geodesic-path computation and protrusion-oriented semantic segmentation, without requiring intermediate surface triangulation.

**Keywords:** Point Clouds, GPU Computing, Multi-GPU, k-Nearest Neighbors, Reverse Engineering, Geodesic Paths, Semantic Segmentation.
**DOI:** https://doi.org/10.14733/cadaps.2027.152-174

## 1 INTRODUCTION

Reverse engineering (RE) aims to reconstruct accurate digital models of physical objects from point clouds acquired with 3D scanning devices. Modern optical and laser scanners easily produce clouds consisting of tens or even hundreds of millions of points, and every stage of the RE pipeline, e.g., the registration of partial scans, the denoising of the acquired samples, the estimation and orientation of normal vectors, the computation of characteristic curves on the cloud, and the segmentation of the cloud into meaningful regions, must process such massive datasets within a reasonable time. At the heart of most of these algorithms lies a common computational kernel: the calculation of the k-nearest neighbors (KNN) of a query point set with respect to a reference point set. When executed on the CPU, this kernel quickly becomes the main performance bottleneck of the whole pipeline.

The objective of this paper is to demonstrate that a carefully designed GPU and multi-GPU KNN methodology can serve as the computational backbone of th

e complete pipeline for point-cloud processing in reverse engineering. Graphics processing units offer massive fine-grained parallelism at low cost; however, the memory of a single device limits the size of the point clouds and the number of neighbors that can be processed. The presented methodology [2] mitigates both limitations by combining a grid-based nearest-neighbor search with partitioning of the query set across multiple GPUs while requiring only the minimum necessary inter-GPU communication.

Based on this infrastructure, the paper reviews the acceleration of three classical low-level RE tasks, namely the iterative closest point (ICP) registration of range images, the elliptic Gabriel Taubin smoothing of point clouds [5], and the estimation and orientation of normal vectors [23]. Furthermore, the paper outlines two higher-level applications developed by the authors that rely on the same neighborhood structures and operate directly on the raw point cloud: the generation of optimal point-to-point geodesic paths [3] and the protrusion-oriented semantic segmentation of point clouds [4]. These applications demonstrate that the presented methodology scales from the lowest level of the RE pipeline up to the extraction of semantic information, without the need for any intermediate surface triangulation.

The rest of the paper is organized as follows. Section 2 reviews the state of the art. Section 3 presents the multi-GPU KNN methodology. Section 4 describes the RE applications that are directly accelerated by the multi-GPU KNN, and Section 5 provides the corresponding experimental results. Section 6 outlines the computation of geodesic paths and the semantic segmentation of point clouds. Finally, Section 7 concludes the paper and discusses future work.

## 2 RELATED WORK

This section reviews KNN computation on single- and multi-GPU systems. It also discusses GPU-based methods for smoothing, ICP registration, and normal estimation. Finally, it briefly reviews the state of the art in geodesic-path computation and point-cloud semantic segmentation.

### 2.1 GPU-Based KNN

Many researchers have exploited the massive parallelism of GPUs to compute k-nearest neighbors. These algorithms fall into two main categories: brute-force methods and spatial-subdivision methods. Brute-force methods compute the distance from every query point to every reference point, whereas spatial-subdivision methods use spatial data structures to compute only the distances required to identify nearby reference points.

#### *2.1.1 Brute-Force Methods*

A representative brute-force approach was proposed by Garcia et al. [21]. First, an $N_p \times N_p$ distance matrix is computed, with each thread calculating a single matrix element. The threads then use insertion sort to identify the k-nearest neighbors. The main drawbacks of this method are its high computational cost and memory requirements. Kuang and Zhao [27] attempted to alleviate these limitations by partitioning the matrix into row blocks. Liang et al. [31] further partitioned both the rows and columns into blocks and subsequently merged the partial results using a global sorting algorithm.

#### *2.1.2 Spatial Subdivision Methods*

In this category, the space is partitioned using spatial data structures, allowing each query point to search only among nearby reference points. The principal structures used for this purpose are kd-trees [9], grid-based structures [20], and R-trees [22], all of which are constructed on the GPU.

A representative kd-tree approach was proposed by Zhou et al. [48]. They constructed the tree in a top-down, breadth-first manner and identified the nearest neighbors through range searches with progressively increasing spherical search radii during tree traversal.

Among grid-based approaches, a representative method was presented by Leite et al. [29]. The space is partitioned into a regular grid, with each voxel storing information about the reference points that fall within it. Each point is assigned a unique key that identifies its corresponding voxel.

An R-tree-based approach was proposed by You et al. [46]. R-trees and kd-trees generally incur considerable computational costs during their construction. The approach proposed in this paper is therefore based on a grid structure. Its novelty lies in the nearest-neighbor search strategy and the multiresolution scheme employed.

## 2.2 GPU-Based ICP

Two principal approaches are commonly used to register a pair of range images: the Iterative Closest Point (ICP) method and its variants, including Generalized ICP [11],[16],[41], and the Normal Distributions Transform (NDT) [12]. In ICP-based methods, the transformation aligning the source range image with the target is estimated by identifying corresponding closest-point pairs and computing the transformation that minimizes their mean squared error. This process is repeated iteratively until convergence. By contrast, NDT employs a probabilistic formulation that maximizes the likelihood of Gaussian distributions and does not explicitly compute nearest-neighbor correspondences. Consequently, NDT-based registration is not examined further in this work.

Qiu et al. [39] accelerated nearest-neighbor computation by serializing a kd-tree and transferring it to a GPU memory buffer. Nearest-neighbor queries are then performed in parallel using the stored kd-tree and GPU-based priority queues. Koide et al. [26] accelerated Generalized ICP by voxelizing the target range image and considering only source points located in voxels containing target points. The main limitation of this approach is its dependence on the selected grid resolution. Many applications require submillimeter registration accuracy, whereas voxel sizes are commonly set on the order of centimeters, which may not provide sufficient accuracy. The approach proposed in this paper supports high-precision registration because its accuracy is not determined by the resolution of the grid employed.

## 2.3 GPU-Based Laplacian Smoothing

Most research on Laplacian smoothing has focused on triangulated surfaces. A representative contribution in this domain was presented by Moench et al. [35], who used OpenGL to perform Laplacian smoothing iterations in parallel by exploiting GPU hardware shaders. Subsequently, Mei et al. [32] employed CUDA to parallelize the Laplacian iterations.

For point clouds, Agathos et al. [5] proposed a GPU-based Laplacian smoothing method that uses elliptic Gabriel neighbors [36] to define the first-order neighborhood required for each point. In their implementation, however, the k-nearest neighbors required to determine the elliptic Gabriel neighborhoods are computed on the CPU. In the present work, the proposed multi-GPU KNN method is used to compute these nearest neighbors. In addition, a multi-GPU Laplacian smoothing algorithm is introduced that minimizes communication among the GPUs by exchanging only the necessary data.

## 2.4 GPU-Based Normal Estimation

Most point-cloud normal estimation methods rely on k-nearest-neighbor information. The resulting neighborhoods are used to fit either a plane [23] or a quadratic surface [14], from which the normal at each point is derived. In this work, the method of Hoppe et al. [23] is adapted to estimate point-cloud normals using multiple GPUs. In addition, the computed normals are consistently oriented using a parallel, single-GPU implementation of the minimum spanning tree (MST) method based on Borůvka's algorithm [13].

## 2.5 Geodesic Paths on Point Clouds

Methods for computing geodesic paths can be classified into two main families. The first family comprises global methods, which compute the geodesic distance field from a source point to all other points of the domain. Representative approaches are the fast marching method of Kimmel and

Sethian [25], which propagates the distance from the source in a wavefront pattern, the heat method of Crane et al. [17], which recovers the distance field from the solutions of the heat and Poisson equations, graph-based methods relying on Dijkstra's algorithm [28], and the exact MMP algorithm of Mitchell et al. [34], which operates exclusively on the polygonal domain. The second family comprises local point-to-point methods, which iteratively improve an initial path connecting two fixed points until it converges to a geodesic. This family includes the straightest geodesics method of Polthier and Schmies [37] and the edge-flipping method of Sharp and Crane [42], which are both defined on the polygonal domain, as well as the method of Yu et al. [47], which operates on point clouds using a grid structure and moving-least-squares projections. The method outlined in Section 6.1 [3] belongs to the local family; it operates directly on the point cloud without any surface triangulation and computes the path by minimizing its geodesic curvature.

### 2.6 Semantic Segmentation of Point Clouds

Segmentation methods can be classified as geometric or semantic. Geometric methods partition the cloud into regions that are homogeneous with respect to a geometric property using, e.g., region growing [10], model fitting with RANSAC [40], or clustering techniques, without assigning any semantic meaning to the resulting regions. Semantic methods can be further classified as supervised or unsupervised. Supervised methods learn perceptual labels from annotated datasets employing classifiers or deep networks operating directly on point sets, such as PointNet [38]; they require large annotated corpora and retraining for every new part family. Unsupervised methods extract perceptually meaningful parts from the geometry alone; a representative approach is the approximate convexity analysis (WCSeg) of van Kaick et al. [45]. This category also includes the protrusion-oriented mesh segmentation of Agathos et al. [6], which has recently been extended by the authors to operate directly on raw point clouds [4]. The latter method is outlined in Section 6.2.

## 3 MULTI-GPU KNN METHODOLOGY

This section describes the methodology proposed by Agathos and Azariadis [2]. The method begins by partitioning the bounding box of the reference point set into a grid of resolution $h$. Each voxel records the reference points it contains, making it straightforward to determine the index coordinates of the voxel to which each reference point belongs.

Nearest-neighbor computation for each query point begins in the grid voxel containing that point. The search then expands to neighboring voxels until all k nearest neighbors have been identified. The index coordinates of the voxel containing a query point may be negative; nevertheless, the search continues to expand until the search region contains reference points.

The nearest neighbors are stored in an array of size k and sorted in increasing order of distance from the query point. At the beginning of the search, the array is empty; it is updated as the search region expands until the k nearest neighbors have been identified. The methodology is described in greater detail below.

### 3.1 Grid Construction

Grid construction begins by scaling the reference points to the unit cube $[0,1]^3$. The query points are scaled using the same minimum and maximum values; therefore, their coordinates may lie outside the unit cube. The number of voxels $n$ per dimension defines the resolution $h = 1/n$ of the grid. The 3D index of the voxel to which a point with coordinates $(x, y, z) \in \mathbb{R}^3$ belongs is given by $(x^c, y^c, z^c) = (\lfloor x * n \rfloor, \lfloor y * n \rfloor, \lfloor z * n \rfloor) \in \mathbb{Z}^3$.

A reference point $p$ always lies within a voxel with non-negative integer coordinates $(x^c, y^c, z^c)$. Assuming $n$ voxels along each grid dimension, the voxel containing $p$ can be uniquely identified by the key $key(p) = x^c + y^c * n + z^c * n^2$. This key is used to efficiently identify the reference points contained in each voxel. Specifically, (i) a key is computed for every reference point, (ii) the points are sorted according to their keys, and (iii) the sorted sequence is compacted by key to determine

the index range corresponding to each occupied voxel. Consequently, empty voxels are not stored or processed.

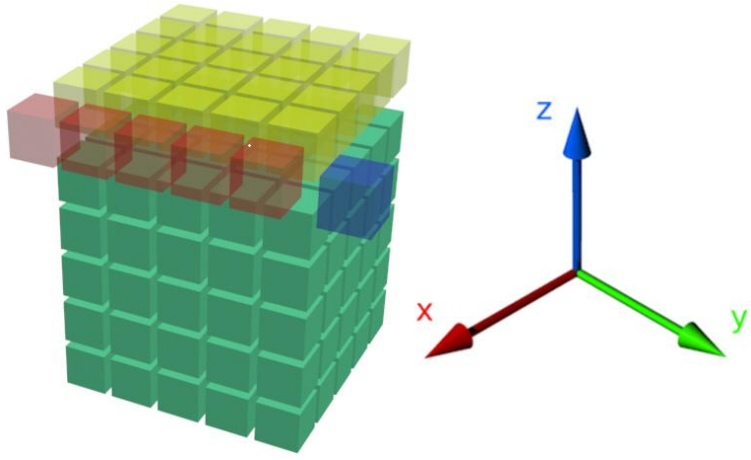



**Figure 1:** The previously searched region of the grid is shown in green. Face-, edge-, and corner-adjacent neighboring voxels are shown in yellow, red, and blue, respectively. Neighbors in all three spatial dimensions, X, Y, and Z, must be considered. Figure reproduced from [2].

### 3.2 Expansion of the Search Area to Neighboring Voxels

The nearest-neighbor search starts in the voxel containing the query point and then expands by $\pm offset, offset = 1,2,\ldots$ voxels in each dimension until it identifies the k nearest neighbors. At each offset, the face- and edge-adjacent voxels are processed first, followed by the corner-adjacent voxels (Figure 1). There are six face-adjacent neighbors, two along each dimension; twelve edge-adjacent neighbors, four for each dimension; and eight corner-adjacent neighbors. Algorithm 1 shows how the face-adjacent voxels are searched along the X dimension; Algorithm 2 shows how the edge-adjacent voxels are searched along the X dimension; and Algorithm 3 shows how the eight corner-adjacent voxels of the search region are searched. $Q^{left} = \{q_i, i = 1, \ldots, N_{left}\}$ denotes the set of remaining query points for which all k nearest neighbors have not yet been found.

The parallel search is performed on a per-voxel basis, as this is computationally less demanding than allowing each query point to expand its search independently, as in [29]. In practice, the search offset should remain small, since a large offset indicates that the query point is far from the reference points and that continuing the search at the current resolution would require substantial computational resources. Therefore, when the offset reaches 2, the voxel width $h$ is doubled (equivalently, $n$ is halved), and the search restarts at offset 1 on the coarser grid.

**Algorithm 1: Searching the face-adjacent voxel neighbors along the X dimension**

**Input:** $Q^{left}$, *offset*

```
1  xDimensionOffset = {−offset, offset}
2  for ii = 0 to 1 do
3      for jj = −offset + 1 to offset − 1 do
4          for kk = −offset + 1 to offset − 1 do
5              for i = 1 to N_left in parallel do
```

```
                    x_i^nc = x_i^c + xDimensionOffset[ii]
                    y_i^nc = y_i^c + jj
                    z_i^nc = z_i^c + kk
                    Search in voxel (x_i^nc, y_i^nc, z_i^nc) for KNN of q_i ∈ Q^left
                end for
            end for
        end for
    end for
```

**Algorithm 2: Searching the edge-adjacent voxel neighbors along the X dimension**

```
Input: Q^left, offset
yzDimensionOffset = {−offset, offset}
for ii = −offset + 1 to offset − 1 do
    for jj = 0 to 1 do
        for kk = 0 to 1 do
            for i = 1 to N_left in parallel do
                x_i^nc = x_i^c + ii
                y_i^nc = y_i^c + yzDimensionOffset[jj]
                z_i^nc = z_i^c + yzDimensionOffset[kk]
                Search in voxel (x_i^nc, y_i^nc, z_i^nc) for KNN of q_i ∈ Q^left
            end for
        end for
    end for
end for
```

**Algorithm 3: Searching the corner-adjacent voxel neighbors**

```
Input: Q^left, offset
xyzDimensionOffset = {−offset, offset}
for ii = 0 to 1 do
    for jj = 0 to 1 do
        for kk = 0 to 1 do
            for i = 1 to N_left in parallel do
                x_i^nc = x_i^c + xyzDimensionOffset[ii]
                y_i^nc = y_i^c + xyzDimensionOffset[jj]
                z_i^nc = z_i^c + xyzDimensionOffset[kk]
                Search in voxel (x_i^nc, y_i^nc, z_i^nc) for KNN of q_i ∈ Q^left
            end for
        end for
    end for
end for
```

### 3.3 k-Nearest-Neighbor Search Kernel

Algorithms 1–3 perform a k-nearest-neighbor search for each voxel under consideration (line 9). One thread is launched for each remaining query point in $Q^{left}$, and all threads run in parallel on the

GPU. The indices of the k-nearest neighbors are stored in a $k \times N_Q$ array $kNNindexes$ and the corresponding distances from the query points are stored in a $k \times N_Q$ array $kNNdistances$. The current number of neighbors found for each query point is stored in the array $kIndexes$. The search for a query point terminates when its k nearest neighbors have been identified; this condition is checked at the last voxel of the search region. The array $continue$ contains a Boolean value for each of the $Q^{left}$ points, indicating whether its search should continue after the kernel has executed.

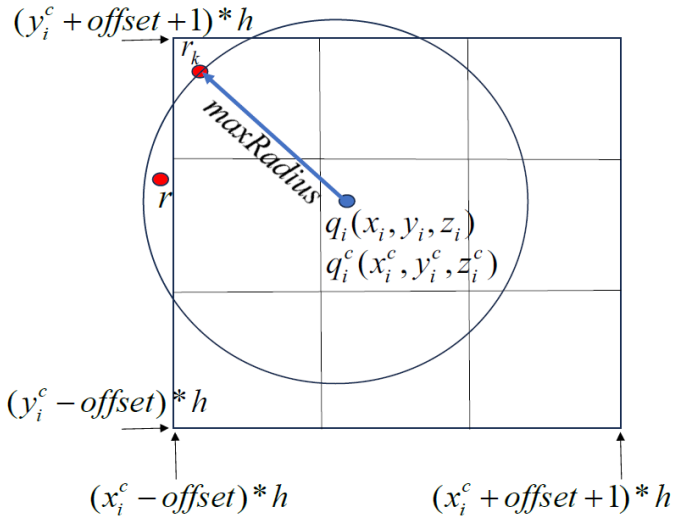



**Figure 2: Current search region** for the query point $q_i$ in voxel $q_i^c$ $(offset = 1)$. Reference point $r_k$ is the current kth-nearest candidate; however, the unexamined reference point $r$ is closer. Therefore, the search region must be expanded further. Figure reproduced from [2].

Algorithm 4 presents the pseudocode of the k-nearest-neighbor search kernel. In line 3, the coordinates of the voxel containing the query point are determined. In line 4, these coordinates are incremented or decremented to identify the neighboring voxel corresponding to the current search $offset$, where $offset = 1,2,\ldots$. A voxel is searched only if it lies within the unit cube (line 6). Its unique key is then computed (line 7), and the index range $[Start, End]$ of the reference points contained within it is determined (lines 8–11). For each of these reference points, the arrays $kNNindexes$, $kNNdistances$ and $kIndexes$ are updated according to its distance from the query point. After the last voxel at the current offset has been processed, line 27 checks whether k neighbors have been found and whether the search for closer neighbors can be terminated. This additional check is necessary to ensure that the exact nearest neighbors are identified, as illustrated in Figure 2. Algorithm 4 can also support a range query defined by a radius threshold while returning at most k nearest points. This functionality is useful in applications such as ICP, where only nearby points need to be considered (Section 4.2). Further implementation details of the KNN method are provided in [2].

### 3.4 Multi-GPU KNN Implementation

The number of query points and the value of k are limited by the memory available on each GPU. Beyond these limits, executing the previously described search on a single GPU becomes impractical. This limitation can be mitigated by using multiple GPUs. Assuming that the number of GPUs is $N_{gpus}$, the query points can be divided into $N_{gpus}$ groups, each containing $|Q|/N_{gpus}$ consecutive query points. Thus, the total amount of memory needed to perform the search for k-nearest neighbors is reduced by $N_{gpus}$.

In the proposed implementation, each GPU stores its own copy of the grid. This replication eliminates inter-GPU communication during the KNN search and thereby increases throughput. Each GPU is coordinated by a dedicated host thread, and all host threads execute in parallel.

**Algorithm 4: k-Nearest-Neighbor Search Kernel**

**Input:** $P$, $Q^{left}$, $N_{left}$, $ii_{inc}$, $jj_{inc}$, $kk_{inc}$, $n$, $k$, *uniqueKeys*, *indexRange*, *kNNindexes*, *kNNdistances*, *kIndexes*, *continue*, *offset*, *lastCube*, *threshold*

```
1  i ← threadIdx.x + blockDim.x * blockIdx.x
2  if i < N_left then
3      Let q_i ∈ Q^left belong to cube q_i^c = (x_i^c, y_i^c, z_i^c)
4      (x_i^nc, y_i^nc, z_i^nc) ← (x_i^c + ii_inc, y_i^c + jj_inc, z_i^c + kk_inc)
5      k_current ← kIndexes[i]
6      if inUnitcube(x_i^nc, y_i^nc, z_i^nc) then
7          key^nc ← x_i^nc + y_i^nc * n + z_i^nc * n^2
8          keyIndex ← BinarySearchIn(uniqueKeys, key^nc)
9          if keyIndex ≥ 0 then
10             Start ← indexRange[keyIndex]
11             End ← indexRange[keyIndex + 1] − 1
12             for j = Start to End do
13                 distance ← ||q_i − p_j||
14                 if distance < threshold then
15                     checkKnearest(i, j, distance, kNNindexes, kNNdistances, k_current, k)
16                 end if
17             end for
18         end if
19         kIndexes[i] ← k_current
20     end if
21     if k_current == k && lastCube then
22         maxRadius ← kNNdistances[k − 1 + k * i]
23         finished ← checkIfFinished(q_i, q_i^c, offset, maxRadius, n)
24         continue[i] ← !finished
25     else if k_current < k && lastCube then
26         finished ← checkIfFinished(q_i, q_i^c, offset, threshold, n)
27         continue[i] ← !finished
28     end if
29 end if
```

## 4 POINT-CLOUD PROCESSING BASED ON THE MULTI-GPU METHODOLOGY

This section presents three indicative processing applications that directly benefit from the proposed multi-GPU KNN methodology: (i) the Iterative Closest Point (ICP) algorithm, (ii) point-cloud normal estimation and orientation, and (iii) elliptic Gabriel–based Taubin smoothing of point clouds. All three algorithms rely extensively on k-nearest-neighbor computations, which constitute their principal computational bottleneck. The results demonstrate that the proposed multi-GPU KNN methodology significantly accelerates each of these applications.

### 4.1 Multi-GPU Elliptic Gabriel Taubin Smoothing

A representative application of Taubin's iterative smoothing method to point clouds was introduced by Agathos et al. [5]. The method proceeds as follows: (i) the elliptic Gabriel neighborhood $EGN(p_i)$ [36] is computed for each point $p_i$ in the point cloud, and (ii) Taubin smoothing is applied using the resulting Gabriel neighbors as the one-ring neighborhood of $p_i$, with Gaussian weights assigned to the neighboring points. Specifically, at each iteration, the position of $p_i$ is updated as follows:

$$p_i \leftarrow p_i + [\lambda|\mu]\frac{1}{\sum_j w_{ij}} \sum_{p_j \in EGN(p_i)} w_{ij}(p_j - p_i)$$
$$w_{ij} = e^{-\|p_i - p_j\|^2} \tag{1}$$

Here $[\lambda|\mu]$ denotes $\lambda$ in the first Taubin pass and $\mu$ in the second.

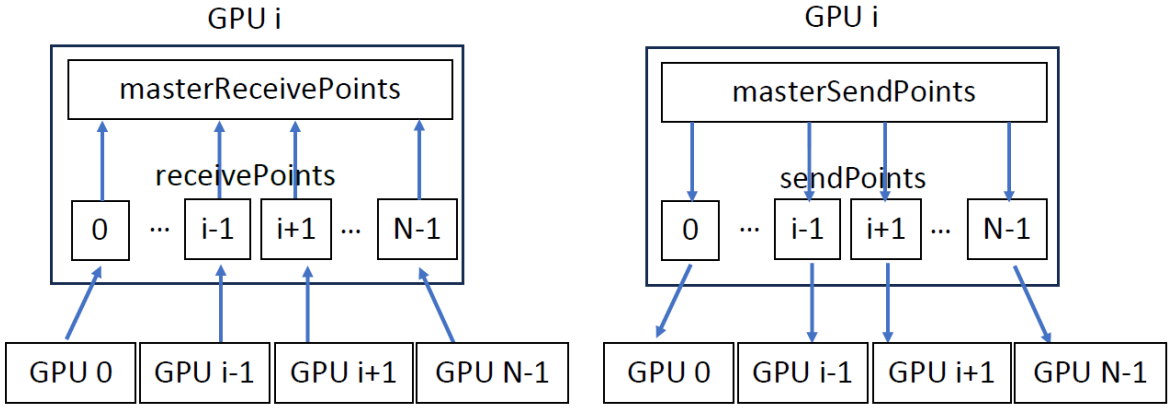



**Figure 3: Left:** $GPU_i$ receives from the other GPUs only the neighboring points required to perform Taubin smoothing. Right: $GPU_i$ sends to the other GPUs only the points required for their smoothing computations. Figure reproduced from [2].

**Algorithm 5: EllipticGabrielTaubinSmoothing Kernel**

**Input:** $P^{in}$, $P^{out}$, *kNNindexes*, *k*, *masterReceiveIndexes*, $dev_{id}$, $gpu_{ids}$, *scale*, *N*, $N_{dev\_id}$, $N_P$, *masterReceivePoints*, *receiveIndexesSumCount*

```
1  idx ← threadIdx.x + blockIdx.x * blockDim.x
2  if idx < N_dev_id then
3      partitionCount ← N_P / N
4      i ← idx + dev_id * partitionCount
5      sum_x ← 0
6      sum_y ← 0
7      sum_z ← 0
8      sum ← 0
9      (x_1, y_1, z_1) ← (P^in_x[i], P^in_y[i], P^in_z[i])
10     for n = 0 to k − 1 do
11         neigh ← knnIndexes[idx * k + n]
12         if neigh ≥ 0 then
13             dev_neigh ← dev_membership(neigh, partitionCount, N)
```

```
14                  (x2, y2, z2) ← (0, 0, 0)
15                  if dev_neigh == dev_id then
16                      (x2, y2, z2) ← (P^in_x[neigh], P^in_y[neigh], P^in_z[neigh])
17                  else
18                      GPU_IDX ← gpu_ids[dev_neigh]
19                      startFromPosition ← masterReceiveIndexes + receiveIndexesSumCount[GPU_IDX]
20                      start ← 0
21                      end ← receiveIndexesSumCount[GPU_IDX + 1] − receiveIndexesSumCount[GPU_IDX]
22                      idxL ← BinarySearch(startFromPosition, start, end)
23                      startFromPosition ← receiveIndexesSumCount[GPU_IDX]
24                      x2 ← masterReceivePoints[startFromPosition + 3 * idxL]
25                      y2 ← masterReceivePoints[startFromPosition + 3 * idxL + 1]
26                      z2 ← masterReceivePoints[startFromPosition + 3 * idxL + 2]
27                  end if
28                  distance ← ||(x2, y2, z2) − (x1, y1, z1)||^2
29                  w ← e^−distance
30                  sum_x += w* (x2 − x1)
31                  sum_y += w* (y2 − y1)
32                  sum_z += w* (z2 − z1)
33                  sum += w
34              end if
35          end for
36          P^out_x[i] ← x1 + scale* sum_x / sum
37          P^out_y[i] ← y1 + scale* sum_y / sum
38          P^out_z[i] ← z1 + scale* sum_z / sum
39      end if
```

The computation of the elliptic Gabriel neighborhood requires the k-nearest neighbors of each point; therefore, the proposed multi-GPU KNN methodology can be directly employed. Specifically, each GPU computes the elliptic Gabriel neighborhoods of its assigned points by executing the kernel described in Algorithm 1 of [5].

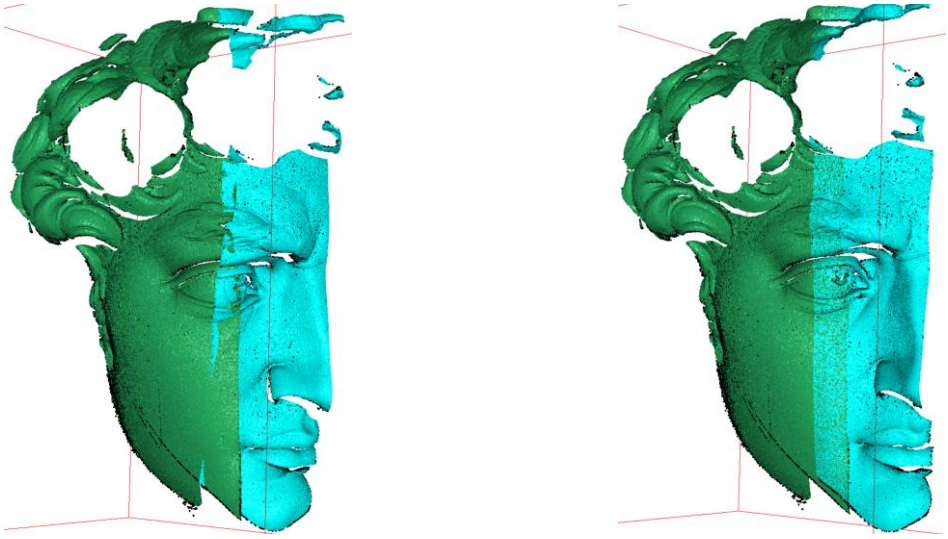
The application of Taubin's smoothing algorithm (Equation 1) may require neighboring points whose coordinates are updated on different GPUs, thereby necessitating inter-GPU communication. Each GPU, denoted $GPU_i$, must receive from the other GPUs precisely the points required for its local computations and, conversely, send them the points required for theirs. This exchange is illustrated in Figure 3. Agathos and Azariadis [2] presented efficient methods for constructing the *MasterReceivePoints* and *MasterSendPoints* arrays. Algorithm 5 presents the smoothing kernel executed by each GPU. As shown in lines 17–27, when a neighboring point is not processed locally, its updated coordinates are retrieved from *MasterReceivePoints* using the appropriate indexing scheme. Further implementation details are provided in [2].

### 4.2 Multi-GPU ICP

The ICP algorithm registers a pair of range images by aligning the reference range image $\boldsymbol{R}$ with the target range image $\boldsymbol{T}$. The registration is performed by finding the quaternion $q_R$ and the translation vector $q_T$ such that the following mean squared error (MSE) is minimized:

$$f(q) = \frac{1}{N_R}\sum_{i=1}^{N_R}\|t_i - R(q_R)r_i - q_T\|^2 \tag{2}$$

where $t_i$ are the closest points of the target image to the $r_i$ points of the reference image, $i = 1, \ldots, N_R$, as illustrated in Figure 4. The minimization is repeated after transforming the reference points with the current rigid transformation until the error converges to a local minimum. The algorithm can converge to the global minimum if the two range images are roughly aligned and sufficiently close. Since each fixed-correspondence rigid update has a closed-form solution, the computational bottleneck is the computation of the nearest neighbors. Our experiments demonstrate that using the proposed multi-GPU KNN methodology leads to a significant speedup of the ICP process.



(a) (b)

**Figure 4:** (a) Target range image (green) before alignment with the source range image (light blue); (b) alignment of the two images after ICP.

### 4.3 Multi-GPU Normal-Vector Estimation and Consistent Orientation

Hoppe et al. [23] proposed a method for estimating the normals of a point cloud. The method first computes the k-nearest neighbors of each point $p_i$ and then constructs the following $3 \times 3$ covariance matrix:

$$\begin{aligned} \mathbf{C}(p_i) &= \sum_{y \in Nhd(p_i)} (y - cm_i)(y - cm_i)^T \\ cm_i &= \frac{1}{k}\sum_{y \in Nhd(p_i)} y \end{aligned} \tag{3}$$

where $Nhd(p_i)$ is the k-nearest neighborhood of $p_i$. The eigenvector corresponding to the smallest eigenvalue is used as the estimated normal at $p_i$.Using the Gabriel neighborhood makes the result less sensitive to the choice of k.

Normal estimation is performed independently on each GPU for its assigned subset of points. Algorithm 6 presents the kernel executed by each GPU. The kernel receives the k-nearest neighbors

of the assigned points, constructs the covariance matrix defined in Equation 3, and computes its eigenvectors using Eberly's Geometric Tools [19]. Further implementation details are provided in [2].

**Algorithm 6: ComputeNormals Kernel**

**Input:** *kNNindexes, k, P, nP, $N_{dev\_id}$, $N_P$, N, dev_id*

```
1  i ← threadIdx.x + blockDim.x * blockIdx.x
2  if i < N_dev_id then
3      partitionCount ← N_P / N
4      idx ← i + dev_id * partitionCount
5      (sum_x, sum_y, sum_z) ← (P^x[idx], P^y[idx], P^z[idx])
6      counter ← 1
7      for j = 0 to k − 1 do
8          n ← kNNindexes[i * k + j]
9          if n ≥ 0 then
10             (sum_x, sum_y, sum_z) += (P^x[n], P^y[n], P^z[n])
11             counter++
12         end if
13     end for
14     (c_x, c_y, c_z) ← (sum_x, sum_y, sum_z) / counter
15     (dx, dy, dz) ← (P^x[idx], P^y[idx], P^z[idx]) − (c_x, c_y, c_z)
16     (a_00, a_01, a_02, a_11, a_12, a_22) ←(dx*dx, dx*dy, dx*dz, dy*dy, dy*dz, dz*dz)
17     for j = 0 to k − 1 do
18         n ← kNNindexes[i * k + j]
19         if n ≥ 0 then
20             (dx, dy, dz) ← (P^x[n], P^y[n], P^z[n]) − (c_x, c_y, c_z)
21             (a_00, a_01, a_02, a_11, a_12, a_22) +=
                 (dx*dx, dx*dy, dx*dz, dy*dy, dy*dz, dz*dz)
22         end if
23     end for
24     eigenvalues: vector containing the three eigenvalues
25     eigenvectors: 3×3 matrix containing the eigenvectors
26     symmetricEigensolver3x3(a_00, a_01, a_02, a_11, a_12, a_22, eigenvalues, eigenvectors)
27     nP[idx] ← eigenvectors[0]
28 end if
```

The normal vectors estimated using this procedure are not guaranteed to have a consistent orientation; in particular, they may not point outward from the latent surface represented by the point cloud. Hoppe et al. [23] addressed this problem by constructing a minimum spanning tree (MST) of a graph whose nodes correspond to the point-cloud samples and whose edges are defined by the k-nearest-neighbor connectivity. The weight assigned to an edge $e_{ij}$ is $w_{ij} = 1 - |n_i^T * n_j|$.

The MST is then traversed from a selected root, and the orientation of each child normal is made consistent with that of its parent. For an edge $e_{pc}$, where $n_p$ and $n_c$ denote the normals of the parent and child nodes, respectively, $n_c$ remains unchanged if $n_p^T * n_c > 0$; otherwise, its direction is reversed. The graph connectivity may alternatively be defined using Gabriel neighbors rather than k-nearest neighbors.

In Agathos and Azariadis [2], the MST is constructed on a single GPU rather than across multiple GPUs. Specifically, their implementation employs the parallel Borůvka algorithm [2], after which the MST is traversed using the method described in Chapter 13 of [24] to consistently orient the normals. Developing a parallel multi-GPU MST construction method based on Borůvka's algorithm [13] remains future work.

## 5 EXPERIMENTAL EVALUATION

This section presents the experiments conducted to evaluate the proposed multi-GPU KNN algorithm and its application to the processing tasks described above. The evaluation employs multiple GPUs and considers several performance and accuracy metrics, including speedup, communication time, and root mean square error (RMSE). For each experiment, the relevant hardware specifications of the computational platform, including the number and type of GPUs and the number of CPU hardware threads, are explicitly reported.

### 5.1 Multi-GPU KNN

To evaluate the proposed multi-GPU KNN method, a reference set comprising 30 million randomly generated three-dimensional points is constructed within the cube $[-1,1]^3$. Three query sets, each containing 25 million randomly generated points, are also constructed. The first query set lies within $[-1,1]^3$, the second within $[-1.05,1.05]^3$, and the third within $[-0.95,0.95]^3$. These datasets are hereafter referred to as Query1, Query2, and Query3, respectively. For every query point, the k=40 nearest neighbors in the reference set are identified.

The following computational platforms are used to measure query performance:

- An Intel Core i5-11600 processor with 6 cores and 12 hardware threads.
- An AMD EPYC 7402P processor with 24 cores and 48 hardware threads.
- A server equipped with four NVIDIA RTX 3090 GPUs.
- A server equipped with four NVIDIA RTX 4090 GPUs.

These platforms are hereafter referred to as Intel, AMD, RTX-3090, and RTX-4090, respectively. On the Intel and AMD platforms, the k-nearest-neighbor searches are performed using the CGAL kd-tree implementation [15], with all available hardware threads employed to process the queries. On the two homogeneous multi-GPU platforms, the searches are performed using the KNN algorithm proposed in this work.

Table 1 demonstrates the substantial execution-time advantage of the proposed KNN methodology over the CPU-based methods. For Query2, however, the execution time approximately doubles because many query points lie farther from the reference set, requiring a considerable expansion of the search region and, consequently, additional computation. Moreover, the execution time does not decrease linearly as the number of GPUs increases. This sublinear scaling is expected because, for a dataset of 30 million reference points, the overhead associated with constructing the grid structure becomes a dominant component of the total execution time.

| *Query/ Processor* | *Intel* | *AMD* | *RTX-3090* | | | | *RTX-4090* | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | *1 GPU* | *2 GPUs* | *3 GPUs* | *4 GPUs* | *1 GPU* | *2 GPUs* | *3 GPUs* | *4 GPUs* |
| Query 1 | 47.24 | 29.13 | 7.41 | 4.62 | 3.79 | 3.42 | 2.68 | 1.92 | 1.72 | 1.74 |
| Query 2 | 44.31 | 28.79 | 15.01 | 8.65 | 6.92 | 6.12 | 4.26 | 2.84 | 2.54 | 2.36 |
| Query 3 | 44.48 | 29.65 | 7.35 | 4.63 | 3.75 | 3.35 | 2.69 | 1.93 | 1.74 | 1.69 |

**Table 1:** Execution times (s) for the query sets on the CPU and GPU platforms listed above.

### 5.2 Multi-GPU smoothing

This section evaluates the proposed smoothing algorithm using the registered scans of Michelangelo's *Saint Matthew* statue [30], shown in Figure 5. The resulting point cloud comprises 100,827,698 points, providing a representative large-scale dataset for evaluating the smoothing method on up to eight NVIDIA RTX 4090 GPUs.

**Figure 5:** Reconstructions of the Saint Matthew model before (left) and after (right) smoothing. Figure reproduced from [2].

Table 2 summarizes the execution times of the different stages of the multi-GPU smoothing algorithm. Although communication time increases with the number of GPUs, it accounts for no more than 12% of the total execution time when eight GPUs are employed. Moreover, the smoothing kernel scales nearly linearly at low GPU counts and remains sublinear beyond four GPUs; end-to-end scaling is limited by the KNN stage.

| *Number of GPUs* | *KNN time* | *Smoothing time* | *Communication time* | *% of smoothing time* |
|---|---|---|---|---|
| 2 | 13488 | 27518 | 400 | 99% |
| 3 | 10941 | 18582 | 476 | 97% |
| 4 | 8848 | 14481 | 500 | 97% |
| 5 | 8639 | 12216 | 866 | 93% |
| 6 | 8597 | 10677 | 920 | 91% |
| 7 | 8532 | 9768 | 1049 | 89% |
| 8 | 8805 | 8956 | 1100 | 88% |

**Table 2:** Execution times (ms) for smoothing the Saint Matthew model, which consists of 100,827,698 points, over 500 iterations with a Gabriel ellipsoid aspect ratio of $\alpha = 0.25$.

### 5.3 Multi-GPU ICP

To evaluate the integration of the proposed multi-GPU KNN algorithm into the ICP framework, the methodology was incorporated into the Scanalyze software [43] and compared with Scanalyze's native kd-tree-based KNN implementation. The two range images used for registration were obtained from scans of Michelangelo's *Saint Matthew* statue [30]. The first range image comprises 14,351,437 points, whereas the second contains 8,796,817 points. Scanalyze performs bidirectional

correspondence searches, with each range image serving alternately as the query and reference set, thereby doubling the number of correspondence searches. The native Scanalyze KNN implementation was executed on a server equipped with an AMD EPYC 7402 processor and 192 GB of memory.

Table 3 reports the execution times for eight ICP iterations. The results demonstrate that the proposed multi-GPU methodology provides a substantial speedup over the kd-tree-based implementation. The execution time of the first iteration also includes the GPU warm-up overhead.

| *Iteration* | *Scanalyze KNN method* | *1 RTX 4090* | | *2 RTX 4090* | |
|---|---|---|---|---|---|
| | | *KNN* | *Total* | *KNN* | *Total* |
| 1 | 122916 | 1983 | 5633 | 1393 | 5159 |
| 2 | 98509 | 1309 | 1943 | 907 | 1530 |
| 3 | 65696 | 602 | 1264 | 496 | 1154 |
| 4 | 60820 | 596 | 1276 | 475 | 1123 |
| 5 | 60271 | 589 | 1250 | 469 | 1108 |
| 6 | 60175 | 592 | 1258 | 470 | 1111 |
| 7 | 60120 | 590 | 1261 | 480 | 1122 |
| 8 | 60025 | 587 | 1248 | 479 | 1134 |

**Table 3:** Per-iteration execution times (ms) for eight ICP iterations using Scanalyze's KNN method and the proposed multi-GPU KNN method.

Figure 6 compares the convergence of the ICP algorithm when nearest-neighbor correspondences are computed using Scanalyze's native kd-tree-based KNN implementation and the proposed multi-GPU KNN method. The results indicate that ICP converges slightly faster with the proposed multi-GPU approach.

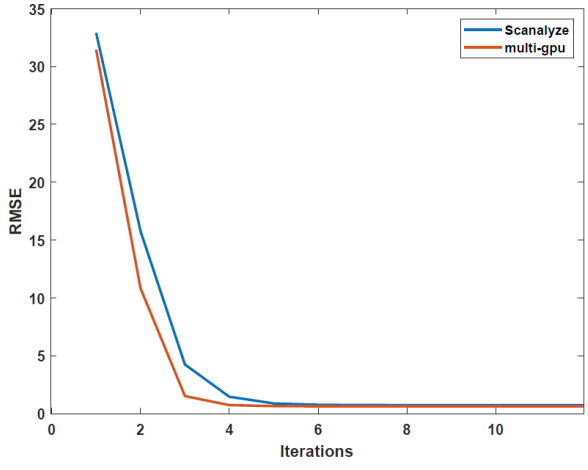



**Figure 6:** Convergence of the ICP algorithm using Scanalyze's KNN algorithm and the proposed multi-GPU KNN algorithm. The minimized objective is the root mean square error (RMSE). Figure reproduced from [2].

### 5.4 Multi-GPU Normal Estimation and Orientation

This section compares the execution times of the proposed multi-GPU normal estimation and orientation method with those obtained using established software packages, namely CGAL [15],

CloudCompare [18], MeshLab [33], and Geomagic [1]. In all implementations, normals are estimated through local plane fitting and consistently oriented using a minimum spanning tree (MST), following the same general methodology as the proposed approach. CloudCompare, MeshLab, and Geomagic were executed on an Intel Core i5-11600 processor with 6 cores and 12 hardware threads, whereas CGAL was executed on an AMD EPYC 7742 processor with 64 cores and 128 hardware threads. The proposed multi-GPU method was executed on a server equipped with four NVIDIA RTX 4090 GPUs. It should be noted that the current MST implementation uses only one GPU. Extending the MST computation to multiple GPUs is left for future work, as discussed in Section 4.3.

The datasets used for the comparison were Gargoyle (Figure 7(a)), Lucy (Figure 7(b)), Venom (Figure 7(c)), and the previously presented *Saint Matthew* model, resampled to 30 million points so that it could be processed by a single GPU. The proposed method used k=40 neighbors. MeshLab used 40 neighbors for normal estimation and 12 neighbors for MST construction, with the same parameters adopted in CloudCompare. In Geomagic, the parameters were selected automatically by the software.

Table 4 reports the execution times for normal estimation and orientation for the aforementioned models. For the proposed method, the timings are divided into normal-estimation and MST-construction times. The results show that the proposed multi-GPU implementation outperforms all the other implementations, achieving speedups of at least 24× in every case (approximately 24×–61× versus the fastest listed baseline).

(a) (b) (c)

**Figure 7:** The 3D models used in the evaluation: (a) Gargoyle, (b) Lucy, and (c) Venom. The normals of all models were correctly oriented by the proposed algorithm.

| *Model* | *Points* | *CGAL* | *Geomagic* | *MeshLab* | *CloudCompare* | *4 RTX 4090 GPUs* | | |
|---|---|---|---|---|---|---|---|---|
| | | | | | | *Normals* | *MST* | *Total* |
| Gargoyle | 8,252,588 | 70217 | 73000 | 65509 | 124470 | 7 | 1441 | 1994 |
| Venom | 9,653,850 | 98980 | 87000 | 110497 | 178240 | 8 | 1327 | 1697 |
| Lucy | 14,027,872 | 79111 | 116000 | 111589 | 312340 | 8 | 2552 | 3256 |
| Saint Matthew | 30,000,000 | 501892 | 385000 | 303260 | 5307000 | 13 | 3439 | 5009 |

**Table 4:** Execution times (ms) for normal estimation and orientation on the respective models. The final column (Total) also includes the KNN computation time.

## 6 GEODESIC PATHS AND SEMANTIC SEGMENTATION ON POINT CLOUDS

In this section, two higher-level applications of the presented point-cloud processing framework are outlined: the generation of optimal point-to-point geodesic paths [3] and the protrusion-oriented semantic segmentation of point clouds [4]. Both applications rely on the neighborhood infrastructure presented in the previous sections—namely, (multi-GPU) KNN computation, elliptic Gabriel neighborhoods [36], normal-vector estimation, and Taubin smoothing [44]—and operate directly on raw point clouds without intermediate surface triangulation. The applications are presented concisely; detailed descriptions and evaluations are available in [3],[4].

### 6.1 Optimal Point-to-Point Geodesic Paths on Point Clouds

Geodesic paths are important in many fields, including composite structures, architectural design, medical reconstruction, toolpath generation, robotics, and reverse engineering. As shown in Section 6.2, geodesic distances also form the basis of protrusion-oriented semantic segmentation of point clouds. Given a point cloud $S = \{s_i\}$ and two points $p_0, p_{m-1} \in S$, the aim is to compute the locally shortest path connecting them or, equivalently, the path whose geodesic curvature vanishes. Let the path be represented by an ordered sequence of points $P = \{p_0, p_1, \dots, p_{m-1}\}$ sampled from the cloud. The discrete geodesic curvature is expressed as a linear operator acting on the vector $\mathbf{p}$, which stacks the coordinates of the path points:

$$k_g = (\mathbf{K} - \mathbf{N}\mathbf{N}^T\mathbf{K})\mathbf{p} = \mathbf{M}\mathbf{p} \tag{4}$$

where $\mathbf{K}$ (3m×m) is the second-difference (curvature) operator, $\mathbf{N}$ (3m×m) is the block-diagonal matrix containing the surface normals of the path points, and $\mathbf{M} = \mathbf{K} - \mathbf{N}\mathbf{N}^T\mathbf{K}$ extracts the tangential component of the curvature. The optimal path is the minimizer of the quadratic functional:

$$\min_{\mathbf{p}} Q(\mathbf{p}) = \frac{1}{2}\|k_g\|^2 = \frac{1}{2}\mathbf{p}^T\mathbf{M}^T\mathbf{M}\mathbf{p} \tag{5}$$

where the interior path points are constrained to lie on the point cloud; this constraint constitutes the main challenge of the problem, since no explicit surface is available. The initial path is obtained using Dijkstra's algorithm on the neighborhood graph of the cloud, with a cost function blending the Euclidean distance with the discrete geodesic curvature.

The minimization is performed by repeating four steps until the length of the path stabilizes. The two endpoints remain fixed while Newton updates are applied to the interior path points.

First, a Newton step advances the current solution toward the minimizer of $Q(\mathbf{p})$:

$$\begin{gathered} \mathbf{p} \leftarrow \mathbf{p} - \mathbf{H}^{-1}\nabla Q(\mathbf{p}) \\ \nabla Q(\mathbf{p}) = \mathbf{M}^T\mathbf{M}\mathbf{p}, \mathbf{H} = \mathbf{M}^T\mathbf{M} \end{gathered} \tag{6}$$

Since the Hessian $\mathbf{H}$ is constant, the Newton iteration exhibits quadratic convergence toward a path of vanishing geodesic curvature. Second, the surface normal of each path point $p_i$ is re-estimated as the weighted average of the normals of its elliptic Gabriel neighbors [36], which are extracted from the k-nearest neighbors computed using the methodology of Section 3:

$$\begin{gathered} n_{p_i} = \frac{1}{\sum_j w_{ij}} \sum_{s_j \in EGN(p_i)} w_{ij} n_{s_j} \\ w_{ij} = e^{-\|p_i - s_j\|} \end{gathered} \tag{7}$$

Third, since the Newton step moves the path points away from the point set, each point is projected back onto the cloud along its normal direction:

$$p_i \leftarrow p_i + t n_{p_i} \tag{8}$$

where the step $t$ is computed by minimizing a weighted sum of squared distances from the projected point to the points in its neighborhood [7],[8]. Finally, the projected path is regularized with one pass of Taubin smoothing [44], analogously to Equation (1):

$$p_i \leftarrow (1 - [\lambda|\mu])p_i + [\lambda|\mu]\frac{p_{i-1} + p_{i+1}}{2} \tag{9}$$

The smoothed path becomes the input to the next iteration. Figure 8 illustrates geodesic paths computed with the proposed algorithm on point clouds sampling a sphere, a torus, and a free-form surface. The method matches the geodesic-distance accuracy of the exact polygon-based algorithms [34],[42], while the mean geodesic-curvature norm of the resulting paths is up to fifty times smaller; that is, the computed paths are considerably smoother. Moreover, the algorithm is parallelizable, and its parallel implementation was measured to be three to thirty times faster than the exact polygon-based solvers while remaining robust to noise, holes, and sampling irregularities. A detailed presentation, an extensive evaluation, and an application of the method to reconstructing free-form surfaces from networks of geodesic curves are provided in [3].

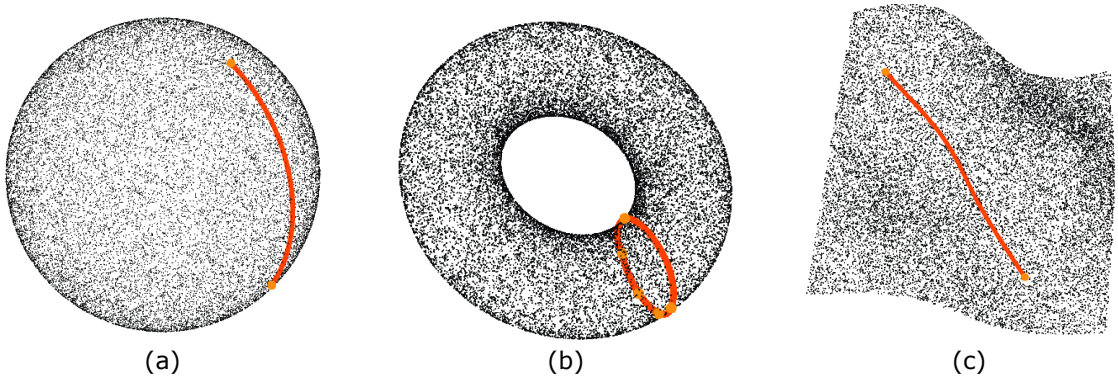



**Figure 8:** Point-to-point geodesic paths (orange) computed directly on point clouds using the algorithm of Section 6.1: (a) sphere, (b) torus, and (c) free-form surface.

### 6.2 Protrusion-oriented Semantic Segmentation of Point Clouds

Semantic segmentation aims to assign each point in a cloud the label of a perceptually meaningful part, e.g., the head, legs, or tail of an articulated object, rather than merely partitioning the cloud into homogeneous geometric regions. The method of Agathos and Azariadis [4] extends protrusion-oriented mesh segmentation [6] to raw point clouds. It is completely unsupervised, requires neither annotated data nor normal vectors, and performs all geodesic computations using Dijkstra's algorithm on the elliptic Gabriel graph [36] induced by the k-nearest neighbors described in Section 3. The method consists of five steps.

(i) The protrusion function $f_{prot}$ is evaluated on the whole cloud: for every point, it sums the geodesic distances to all other points of the cloud, yielding low values on the core of the object and

high values on the tips of the protrusions. Since the exact evaluation is quadratic in the number of points, the cloud is partitioned into non-overlapping geodesic regions $R_i$ with representative points $p_i$ and the function is approximated as:

$$f_{prot}(p) = \sum_{p' \in S} g(p, p') \approx \sum_{i} g(p, p_i) \cdot area(R_i) \tag{10}$$

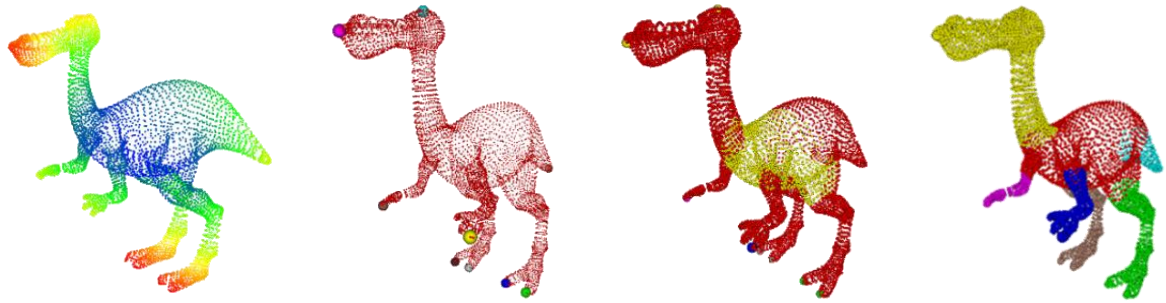
where $area(R_i)$ is computed by finding the mean radius of the one-ring neighborhood of each point contained in the patch and taking the sum of the disk areas defined by these radii.

(ii) The salient points are detected as the local maxima of the protrusion function above a threshold; they reside on the protrusions of the object.

(iii) Since a single protrusion typically produces several local maxima, salient points separated by less than the mean pairwise geodesic distance are grouped together and the point with the highest protrusion value is kept as the representative of each protrusion.

(iv) The core of the object is recovered by linking every pair of representatives with minimum-cost paths and by growing a region, in ascending protrusion order, until it covers a prescribed fraction of these paths; the core separates the protrusions from each other.

(v) Each protrusion is finally cut off at its boundary with the core: successive geodesic regions are grown from its representative and an abrupt jump in the region area indicates the transition to the main body, while a minimum graph cut on the elliptic Gabriel graph, with edge capacities driven by the Gaussian curvature estimated via jet fitting [15], refines the boundary so that it snaps onto deep concavities, in accordance with human perception. Figure 9 illustrates the main steps of the method on the Dodo point cloud.



(a) (b) (c) (d)

**Figure 9:** The main steps of the protrusion-oriented semantic segmentation on the Dodo point cloud: (a) protrusion function (blue: low values, red: high values), (b) salient points, (c) core of the object, (d) final semantic parts. Figure reproduced from [4].

Figure 10 compares the point-based method with its mesh-based counterpart [6] on representative models. The method recovers the same semantic parts as the mesh-based algorithm, although it operates on a completely different domain, and it remains robust when the points undergo random displacements of up to 1% from their original positions, a setting in which convexity-based approaches, e.g., WCSeg [45], fail.

Table 5 reports the execution times of the two methods. Because no triangulation is required, the point-based method is competitive with the mesh-based pipeline and is faster for most models. The method has also been extended to the hierarchical segmentation of complex geometries; for a detailed presentation and evaluation, the reader is referred to [4].

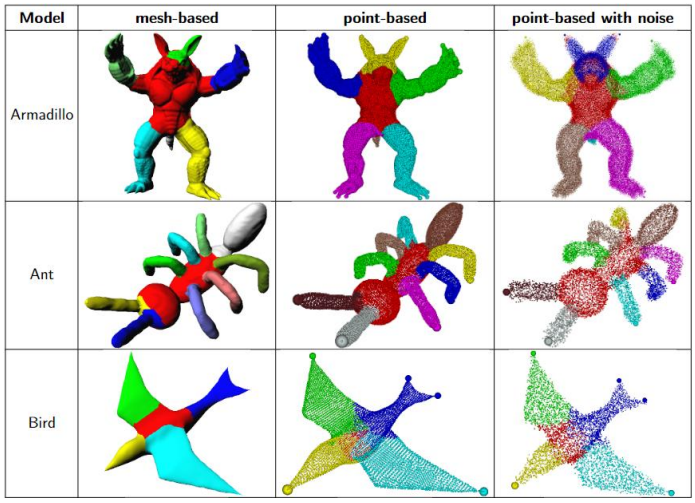



**Figure 10:** Comparison between the mesh-based segmentation (left column) [6], the presented point-based segmentation (middle column), and the point-based segmentation with 1% random displacement of the points from their initial positions (right column).

| *Model* | *Points* | *Mesh-based method [6]* | *Point-based method [4]* |
|---|---|---|---|
| Armadillo | 172,974 | 13.31 | 8.22 |
| Hand | 43,442 | 2.93 | 2.46 |
| Donkey | 10,436 | 0.83 | 0.44 |
| Fox | 4,712 | 0.56 | 0.27 |
| Human | 2,639 | 0.35 | 0.18 |

**Table 5:** Execution times (s) for mesh-based [6] and point-based [4] protrusion-oriented segmentation on an Intel Core i5-11600K CPU using OpenMP.

## 7 CONCLUSIONS

This paper presented the authors' work on GPU and multi-GPU point-cloud processing, followed by representative higher-level applications in geodesic-curve computation and point-cloud segmentation. It accompanies the authors' keynote presentation at CAD'26. The grid-based multi-GPU methodology described for k-nearest-neighbor computation partitions the query set among the available devices and allows each GPU to maintain its own copy of the grid, eliminating inter-GPU communication during the search. The methodology was applied to three fundamental tasks in the reverse-engineering pipeline: ICP registration of range images, elliptic Gabriel Taubin smoothing, and normal-vector estimation and orientation. The experimental evaluation demonstrated significant speedups: KNN computation outperformed well-established parallel CPU implementations by more than an order of magnitude; the smoothing kernel continued to accelerate with up to eight GPUs, although end-to-end scaling was limited by the KNN stage; and normal-vector estimation and orientation achieved speedups greater than 24× relative to established software packages.

Two higher-level applications built on the same neighborhood infrastructure were outlined: an algorithm for generating optimal point-to-point geodesic paths that converges quadratically to paths of vanishing geodesic curvature, matches the accuracy of exact polyhedral algorithms, and produces considerably smoother results; and protrusion-oriented semantic segmentation, which extracts perceptually meaningful parts from raw point clouds without training data. Together, these applications extend the framework to higher-level tasks without requiring intermediate surface

triangulation. Future work includes constructing the minimum spanning tree across multiple GPUs using Borůvka's algorithm [13] to consistently orient normal vectors and computing optimal geodesic paths on large, complex point-cloud scenes.

*Alexander Agathos*, https://orcid.org/0000-0002-0402-0821
*Philip Azariadis*, https://orcid.org/0000-0002-8687-5777

## REFERENCES

[1] 3D Systems: Geomagic Design X, 3D Systems, Rock Hill, SC, 2023, https://support.3dsystems.com.
[2] Agathos, A.; Azariadis, P.: Multi-GPU 3D k-Nearest Neighbors Computation with Application to ICP, Point Cloud Smoothing and Normals Computation, Parallel Computing, 121, 2024, 103093. https://doi.org/10.1016/j.parco.2024.103093.
[3] Agathos, A.; Azariadis, P.: Optimal Point-to-Point Geodesic Path Generation on Point Clouds, Computer-Aided Design, 162, 2023, 103552. https://doi.org/10.1016/j.cad.2023.103552.
[4] Agathos, A.; Azariadis, P.: Protrusion-Oriented Point Cloud Semantic Segmentation, Computer-Aided Design and Applications, 21(2), 2024, 328-349. https://doi.org/10.14733/cadaps.2024.328-349.
[5] Agathos, A.; Azariadis, P. N.; Kyratzi, S.: Elliptic Gabriel Taubin Smoothing of Point Clouds, Computers & Graphics, 106, 2022, 20-32. https://doi.org/10.1016/j.cag.2022.05.009.
[6] Agathos, A.; Pratikakis, I.; Perantonis, Stavros; Sapidis, Nickolas S.: Protrusion-Oriented 3D Mesh Segmentation, The Visual Computer, 26(1), 2010, 63-81. https://doi.org/10.1007/s00371-009-0383-8.
[7] Azariadis, P. N.: Parameterization of Clouds of Unorganized Points Using Dynamic Base Surfaces, Computer-Aided Design, 36(7), 2004, 607-623. https://doi.org/10.1016/S00104485(03)00138-6.
[8] Azariadis, P. N.; Sapidis, Nickolas S.: Drawing Curves onto a Cloud of Points for Point-Based Modeling, Computer-Aided Design, 37(1), 2005, 109-122. https://doi.org/10.1016/j.cad.2004.05.004.
[9] Bentley, J. L.: Multidimensional Binary Search Trees Used for Associative Searching, Communications of the ACM, 18(9), 1975, 509-517. https://doi.org/10.1145/361002.361007.
[10] Besl, P. J.; Jain, R. C.: Segmentation through Variable-Order Surface Fitting, IEEE Transactions on Pattern Analysis and Machine Intelligence, 10(2), 1988, 167-192. https://doi.org/10.1109/34.3881.
[11] Besl, P. J.; McKay, N. D.: A Method for Registration of 3-D Shapes, IEEE Transactions on Pattern Analysis and Machine Intelligence, 14(2), 1992, 239-256. https://doi.org/10.1109/34.121791.
[12] Biber, P.; Strasser, W.: The Normal Distributions Transform: A New Approach to Laser Scan Matching, Proceedings of the 2003 IEEE/RSJ International Conference on Intelligent Robots and Systems, Las Vegas, NV, USA, IEEE, 2003, 2743-2748. https://doi.org/10.1109/IROS.2003.1249285.
[13] Borůvka, O.: O Jistém Problému Minimálním, Práce Moravské Přírodovědecké Společnosti, 3(3), 1926, 37–58.
[14] Cazals, F.; Pouget, M.: Estimating Differential Quantities Using Polynomial Fitting of Osculating Jets, Computer Aided Geometric Design, 22(2), 2005, 121-146. https://doi.org/10.1016/j.cagd.2004.09.004.
[15] CGAL Project: CGAL, CGAL Project, 2023, https://www.cgal.org/.
[16] Chen, Y.; Medioni, G. G.: Object Modeling by Registration of Multiple Range Images, Proceedings of the 1991 IEEE International Conference on Robotics and Automation, Sacramento, CA, USA, IEEE Computer Society, 1991, 2724-2729. https://doi.org/10.1109/ROBOT.1991.132043.

[17] Crane, K.; Weischedel, C.; Wardetzky, M.: Geodesics in Heat: A New Approach to Computing Distance Based on Heat Flow, ACM Transactions on Graphics, 32(5), 2013, 152:1-152:11. https://doi.org/10.1145/2516971.2516977.
[18] DanielGM: CloudCompare, CloudCompare Project, 2023, https://www.danielgm.net/cc/.
[19] Eberly, D.: A Robust Eigensolver for 3×3 Symmetric Matrices, Geometric Tools, 2023, https://www.geometrictools.com/.
[20] Elfangary, L.; Ahmed, M.; Mostafa, S. B. Ali: Review of k-Nearest Neighbor Search Methods Based on Grid Indexing Technique, Proceedings of the 2009 International Conference on Information & Knowledge Engineering (IKE 2009), Las Vegas, NV, USA, CSREA Press, 2009, 608-614.
[21] Garcia, V.; Debreuve, E.; Barlaud, M.: Fast k-Nearest Neighbor Search Using GPU, Proceedings of the 2008 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops, Anchorage, AK, USA, IEEE, 2008, 1-6. https://doi.org/10.1109/CVPRW.2008.4563100.
[22]
[23] Guttman, A.: R-Trees: A Dynamic Index Structure for Spatial Searching, Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data, Boston, MA, USA, ACM, 1984, 47-57. https://doi.org/10.1145/971697.602266.
[24] Hoppe, H.; DeRose, T.; Duchamp, T.; McDonald, J. A.; Stuetzle, W.: Surface Reconstruction from Unorganized Points, Proceedings of the 19th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH 1992), Chicago, IL, USA, ACM, 1992, 71-78. https://doi.org/10.1145/133994.134011.
[25] Hwu, W. W.; Kirk, D. B.; El Hajj, I.: Programming Massively Parallel Processors: A Hands-on Approach, 4th ed., Morgan Kaufmann, Cambridge, MA, 2022.
[26] Kimmel, R.; Sethian, J. A.: Computing Geodesic Paths on Manifolds, Proceedings of the National Academy of Sciences, 95(15), 1998, 8431-8435. https://doi.org/10.1073/pnas.95.15.8431.
[27] Koide, K.; Yokozuka, M.; Oishi, S.; Banno, A.: Voxelized GICP for Fast and Accurate 3D Point Cloud Registration, Proceedings of the 2021 IEEE International Conference on Robotics and Automation, Xi'an, China, IEEE, 2021, 11054-11059. https://doi.org/10.1109/ICRA48506.2021.9560835.
[28] Kuang, Q.; Zhao, L.: A Practical GPU-Based kNN Algorithm, Proceedings of the International Symposium on Computer Science and Computational Technology (ISCSCT 2009), Huangshan, China, Academy Publisher, 2009, 151-155.
[29] Lanthier, M.; Maheshwari, A.; Sack, J.-R.: Approximating Weighted Shortest Paths on Polyhedral Surfaces, Algorithmica, 30(4), 2001, 527-562. https://doi.org/10.1007/s00453-001-0027-5.
[30] Leite, P. J. S.; Teixeira, J. M. X. N.; Farias, T. S. M. C.; Reis, B.; Teichrieb, V.; Kelner, J.: Nearest Neighbor Searches on the GPU - A Massively Parallel Approach for Dynamic Point Clouds, International Journal of Parallel Programming, 40(3), 2012, 313-330. https://doi.org/10.1007/s10766-011-0184-3.
[31] Levoy, M.; Pulli, K.; Curless, B.; Rusinkiewicz, S.; Koller, D.; Pereira, L.; Ginzton, M.; Anderson, S. E.; Davis, J.; Ginsberg, J.; Shade, J.; Fulk, D.: The Digital Michelangelo Project: 3D Scanning of Large Statues, Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH 2000), New Orleans, LA, USA, ACM, 2000, 131-144. https://doi.org/10.1145/344779.344849.
[32] Liang, S.; Wang, C.; Liu, Y.; Jian, L.: CUKNN: A Parallel Implementation of k-Nearest Neighbor on CUDA-Enabled GPU, Proceedings of the 2009 IEEE Youth Conference on Information, Computing and Telecommunication, Beijing, China, IEEE, 2009, 415-418. https://doi.org/10.1109/YCICT.2009.5382329.
[33] Mei, G.; Tipper, J. C.; Xu, N.: A Generic Paradigm for Accelerating Laplacian-Based Mesh Smoothing on the GPU, Arabian Journal for Science and Engineering, 39(11), 2014, 7907-7921. https://doi.org/10.1007/s13369-014-1406-y.

[34] MeshLab: MeshLab, Visual Computing Laboratory, ISTI-CNR, Pisa, 2023, https://www.meshlab.net/.
[35] Mitchell, J. S. B.; Mount, D. M.; Papadimitriou, Christos H.: The Discrete Geodesic Problem, SIAM Journal on Computing, 16(4), 1987, 647-668. https://doi.org/10.1137/0216045.
[36] Mönch, T.; Kubisch, C.; Lawonn, K.; Westermann, R.; Preim, B.: Visually Guided Mesh Smoothing for Medical Applications, Proceedings of the Eurographics Workshop on Visual Computing for Biology and Medicine, Norrköping, Sweden, The Eurographics Association, 2012, 91-98. https://doi.org/10.2312/VCBM/VCBM12/091-098.
[37] Park, J. C.; Shin, H.; Choi, B. K.: Elliptic Gabriel Graph for Finding Neighbors in a Point Set and Its Application to Normal Vector Estimation, Computer-Aided Design, 38(6), 2006, 619-626. https://doi.org/10.1016/j.cad.2006.02.008.
[38] Polthier, K.; Schmies, M.: Straightest Geodesics on Polyhedral Surfaces, in Mathematical Visualization, Springer, Berlin, Heidelberg, 1998, 135-150. https://doi.org/10.1007/978-3-662-03567-2_11.
[39] Qi, C. R.; Su, Hao; M., Kaichun; Guibas, L. J.: PointNet: Deep Learning on Point Sets for 3D Classification and Segmentation, Proceedings of the 2017 IEEE Conference on Computer Vision and Pattern Recognition, Honolulu, HI, USA, IEEE, 2017, 652-660. https://doi.org/10.1109/CVPR.2017.16.
[40] Qiu, D.; May, S.; Nüchter, A.: GPU-Accelerated Nearest Neighbor Search for 3D Registration, in Computer Vision Systems, Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, 2009, 194-203. https://doi.org/10.1007/978-3-642-04667-4_20.
[41] Schnabel, R.; Wahl, R.; Klein, R.: Efficient RANSAC for Point-Cloud Shape Detection, Computer Graphics Forum, 26(2), 2007, 214-226. https://doi.org/10.1111/j.14678659.2007.01016.x.
[42] Segal, A.; Haehnel, D.; Thrun, S.: Generalized-ICP, Robotics: Science and Systems V, Seattle, WA, USA, Robotics: Science and Systems Foundation, 2009. https://doi.org/10.15607/RSS.2009.V.021.
[43] Sharp, Nicholas; Crane, Keenan: You Can Find Geodesic Paths in Triangle Meshes by Just Flipping Edges, ACM Transactions on Graphics, 39(6), 2020, 249:1-249:15. https://doi.org/10.1145/3414685.3417839.
[44] Stanford Computer Graphics Laboratory: Scanalyze Software, Stanford University, Stanford, CA, 2023, http://graphics.stanford.edu/software/scanalyze/.
[45] Taubin, G.: Curve and Surface Smoothing without Shrinkage, Proceedings of the IEEE International Conference on Computer Vision, Cambridge, MA, USA, IEEE Computer Society, 1995, 852-857. https://doi.org/10.1109/ICCV.1995.466848.
[46] van Kaick, O.; Fish, N.; Kleiman, Y.; Asafi, S.; Cohen-Or, D.: Shape Segmentation by Approximate Convexity Analysis, ACM Transactions on Graphics, 34(1), 2014, 4:1-4:11. https://doi.org/10.1145/2611811.
[47] You, S.; Zhang, J.; Gruenwald, L.: Parallel Spatial Query Processing on GPUs Using R-Trees, Proceedings of the 2nd ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data, Orlando, FL, USA, ACM, 2013, 23-31. https://doi.org/10.1145/2534921.2534949.
[48] Yu, H.; Zhang, J. J.; Jiao, Z.: Geodesics on Point Clouds, Mathematical Problems in Engineering, 2014, 2014, 860136. https://doi.org/10.1155/2014/860136.
[49] Zhou, K.; Hou, Q.; Wang, R.; Guo, B.: Real-Time KD-Tree Construction on Graphics Hardware, ACM Transactions on Graphics, 27(5), 2008, Article 126, 1-11. https://doi.org/10.1145/1409060.1409079.