Parallelizing Chamfer Distance Computation for Point Cloud Similarity

Cem Koc* and Eric Liu*

Department of Electrical Engineering and Computer Sciences
University of California, Berkeley

Spring 2021

1 Introduction

Given the increasing prevalence of point cloud data from LiDAR sensors, efficient processing and manipulation of massive amounts of this data is becoming more important. We seek to tackle inefficiencies in point cloud data comparison (seeing how similar two scanned spaces, "scenes," are) with the Chamfer distance metric. We utilize both kdtree and octrees to avoid the quadratic nature of the problem and implement four different Chamfer distance algorithms including two approximation algorithms which we expect to outperform exact calculations. We implement parallelized versions of these Chamfer distance algorithms using OpenMP and CUDA to explore the potential for increased computational efficiency. We see the expected nearly optimal scaling for the naive algorithms and impressive performance for OpenMP implementations, boding well for consumer applications.

2 Related Work

Measure of a distance between two point sets or point clouds in Euclidean space is incredibly useful for loss functions for point cloud neural nets such as PointNet [1] and VoxNet [2]. Since cardinality of a single set of point cloud can be in the order of millions, billions or in some cases tens of billions [3] it is difficult to run sequential pair-to-pair comparison based distance algorithms at scale. The only work we discovered so far that attempts to parallelize the sequential computation of Chamfer distance, given in Eq.4, is the work from Pham et al. [4] which used OpenMP [5] to do shared memory parallelism on a multi-core machine and achieved a speedup factor of 1.3 on a dual-core machine and 2.6 on a quad-core machine. This work, while certainly significant for computing distances for images, falls short for 3D sets such as point clouds because the methods used are for computing 2D super pixels and are not suitable for higher dimensional datasets.

3 Methods

3.1 Data Collection

The dataset was collected using an iPhone 12 Pro equipped with a time-of-flight type of sensor which works by emitting multiple beams of light and measuring the time it takes for those beams to come back to the sensor receiver, which then can measure the relative distance of objects in the frame. This

provides a relatively accurate measurement of depth and distance of objects in a 3D space which cannot be achieved to this level of accuracy using a stereo camera technology without a LiDAR. Even though consumer LiDARs have come a long way, they are not yet as accurate as the enterprise LiDAR sensors that are available (such as Velodyne LiDAR) for orders of magnitude of the cost and are heavily used in self-driving cars in the learning and planning algorithms [3]. Their cost, size and power need often makes enterprise LiDARs infeasible to situate on low-cost millirobots [6], or for hand held use cases.

Figure 1: Tripod mounted iPhone 12 Pro with LiDAR sensor (left) which can be rotated freely during scanning. The point cloud scene of the living room (after centering and processing) visualized using CloudCompare.

We faced significant challenges working with a low-cost LiDAR sensor. LiDAR point clouds produced by this kind of sensors are extremely sensitive to perturbations, where you started collection (scanning) and collection time. Depending on the minutiae of variances in collection, the output can be sparser or denser, misaligned (between two scans of the same scene) and can have artifacts due to the surface reflections from different materials. In order to mitigate these factors we mounted the LiDAR sensor (which in this case an iPhone 12 Pro) on a tripod with the iPhone making 90 degrees with the transverse plane and with 360 degrees of rotational freedom along the sagittal axis as seen in Fig. 1. The iPhone was rotated by hand ranging from 2 minutes to 4 minutes (to increase point density) and scanned the entire room with limited view of the ceiling and the floor. For each scene (room and living room) which can be seen in Fig.2, we processed the raw dataset using a Python script to remove sensor outliers from dataset, we centered each scans of the same scene to be able to compare and removed extraneous features (such as colors and surface normals) other than x,y,z coordinates.

3.2 Point Cloud Dataset

We collected 3 sets of point clouds with varying densities (small, medium and large) each with 2 scans. The former was always named "cloud A" and latter being "cloud B". The Chamfer distance was then computed on A and B clouds of the same scene pair coming from the same dataset. In the Table 1 you can see the dataset sizes given by number of points contained within the cloud.

3.3 Chamfer Distance and Other Metric Distances

Chamfer distance Eq. 4, builds on top of its predecessor Hausdorff distance [7] which is an early distance measure that attempts to compare two point sets within Euclidean space and it is described as the

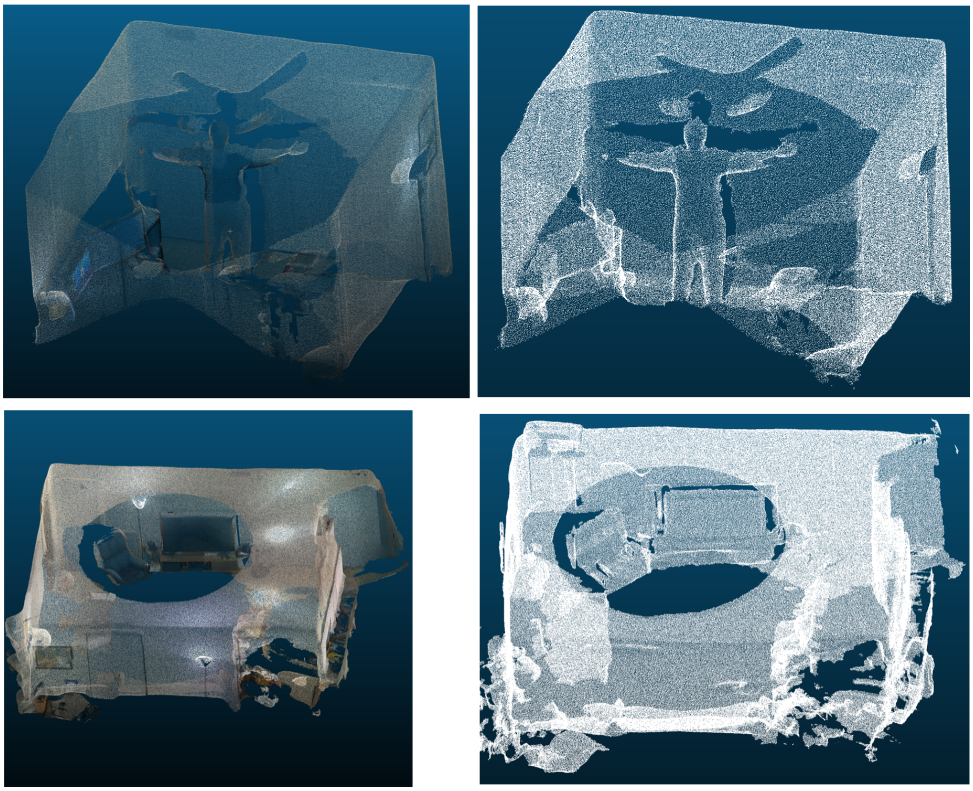

Figure 2: Point clouds of the room and living room scenes are processed to remove outliers, center it and remove color information. Each point is described in x,y,z dimensions.

Dataset Name	Scene A (points)	Scene B (points)
Small	222,924	229,453
Medium	603,292	617,699
Large	1,676,084	1,667,098

Table 1: Dataset sizes used in computing Chamfer distance. Cloud A and Cloud B are referring to two different scans of the same scene.

minimum distance from a point to a set as given in Eq. 1.

$$d(x, y) = ||x - y||_2 \tag{1}$$

$$\mathcal{D}(x, S) = \min_{y \in S} d(x, y) \tag{2}$$

$$\mathcal{D}_H(S_A, S_B) = \max\{\max_{a \in S_A} \mathcal{D}(a, S_B), \max_{b \in S_B} \mathcal{D}(b, S_A)\} \tag{3}$$

$$\mathcal{D}_{CD}(S_A, S_B) = \frac{1}{|S_A|} \sum_{a \in S_A} \min_{y \in S_B} ||a - y||_2^2 + \frac{1}{|S_B|} \sum_{b \in S_B} \min_{y \in S_A} ||b - y||_2^2 \tag{4}$$

Another significantly used distance measure worth mentioning is the Earth mover's distance ($\mathcal{D}_{EMD}$) (Eq. 5) which was first discussed by Rubner et al. [8], also known as the Wasserstein metric. This distance is different from the two aforementioned distances in such a way that it does not depend on finding pairwise point-to-set distances rather "it is based on finding a 1-1 bijection (ξ) between the two sets such that the sum of Euclidean distances between the corresponding points is minimized" [8, 9]. We did not end up choosing this as our distance metric because of one crucial requirement that in order to create a bijection the two point cloud sets must have the same cardinality which is incredibly infeasible to achieve with low-cost LiDAR sensors without sub-sampling. Therefore, we used Chamfer Distance as our metric distance to compare two point clouds.

Chamfer distance 4 is an absolute deterministic metric distance and as with all distances defined on a Euclidean space, it is associative. It's implementation has two directions: forward and backward which can be computed independently from each other. Forward direction computes the pairwise distance from point cloud A to B and backward direction computes the distance from point cloud B to A. The chamfer distance is given as a squared distance.

$$\mathcal{D}_{EMD}(S_A, S_B) = \min_{\xi : S_A \longrightarrow S_B} \sum_{a \in S_A} ||a - \xi(a)||_2 \tag{5}$$

3.4 Serial Chamfer Distance Implementations

3.4.1 All-pairs (naive) Chamfer Distance Implementation

In implementing serial all-pairs Chamfer distance given as is in Eq. 4, we use two top level for loops: one for the forward direction and the other for the backward direction. Within each of the for loops we also have another level of inner for loop that computes the min given in the Eq. 4. For simplicity, we hereto forth refer to the origin point cloud as "cloud A" while "cloud B" refers to the neighboring point cloud in our dataset in Table. 1. In the forward direction of the algorithm, for each point in cloud A, we iterate through all of the points in cloud B and find the point in B that minimizes the squared Euclidean distance to point in A. We add this minimum distance to our running total in the forward direction. For backward direction, we do the same operation only in reverse direction and finally the Chamfer distance is calculated from the weighted averages of the two sum values. This is trivially seen to be an $O(n^2)$

algorithm and it is not suitable for computing distances over bigger datasets for instance collected by self-driving vehicles.

3.4.2 Space-Partitioned Approximate Chamfer Distance Implementations

Exploiting a unique feature of our dataset, we can make the serial $O(n^2)$ all-pairs algorithm nearly linear, and thus compute optimal, by making a few informed assumptions. First of all we begin by noticing that the clouds we are comparing are assumed to describe the same scene with similar topological properties. This means all scenes can be assumed to have a convex hull that is (at least topologically) a convex polyhedron and specifically a cuboid in 3D. If two clouds, A and B, are describing the same scene that means for a given point in A, when computing the forward Chamfer, we do not need to search for all points in B and instead can focus on nearby points in B because if these two scenes are aligned, then the point in B that minimizes the distance to our point in A must be contained in a finite ball of points defined by a fixed radius which depends on the sparsity of the point cloud. The points enclosed within this ball, centered at our point in A p_A with a fixed radius r, creates a subset of cloud B which are at most r distance away from p_A. This new constrained search space allows us to compute the minimum distanced points in the inner for loop of forward and backward Chamfer efficiently, reaching near linear speeds which are discussed in the **Results** section. Then with this assumption, the original Chamfer distance equation can be modified to include the constrained search spaces as given in Eq.6 which we refer to as "approximate Chamfer distance".

$$\mathcal{D}_{approxCD}(S_A, S_B) = \frac{1}{|S_A|} \sum_{a \in S_A} \min_{y \in K_B^{(a)}} ||a - y||_2^2 + \frac{1}{|S_B|} \sum_{b \in S_B} \min_{y \in K_A^{(b)}} ||b - y||_2^2 \tag{6}$$

With this assumption and new framing, the 3D points can be efficiently divided and searched for radius based nearest neighbors queries using two popular space partitioning data structures: KD-Tree, Octree. KD-Tree divides the space by continually splitting the data in the direction of highest variance which leads to unequal space partitions. Octree divides the space to 8 child octants at each level. In this paper, we implemented two flavors of the approximate Chamfer distance algorithm: one with KD-Tree and other with Octree and compared which one is more efficient and can be further parallelized. Note that point clouds are inherently sparse therefore Octree voxels will have extremely differing number of points contained in the cuboids when doing radius search. KD-tree is also affected by this but an order of magnitude less due to it employing a variance based division heuristic approach rather than purely geometric approach. We used the PCL library [10] and its implementations of KD-tree and Octree data structures to partition our point clouds A and B. KD-tree uses FLANN [11] underneath to do fast, approximate nearest neighbor search which we used and in Octree we used radius based nearest neighbors search and an exact nearest neighbor search.

Octree Nearest Neighbor Using an Octree, we can much more efficiently find nearest neighbors due to this space partitioning. To find the nearest neighbor, we first calculate the distance to the nearest neighbor in the same sub-octant. If no other sub-octant boundaries are within that distance, then we have found the nearest neighbor. If there are other sub-octants within that distance, then we must search all points in the overlapping sub-octants. This repeats until a nearest neighbor has been found and no overlapping sub-octants must be explored. In the case of no nearest neighbor existing within the sub-octant, the containing octant is searched instead and the remainder of the algorithm is identical but with lower granularity.

This algorithm thus has an average-case complexity of $O(\log(n))$, but when implemented efficiently, this algorithm grows nearly linearly with problem size[12]. This is a significant improvement over the naive quadratic explosion.

Octree Radius An alternative method which simplifies the neighbor search is to use a radius search. Although radius implies a spherical shape, we functionally use the voxels that are given by the octree. In order to find points within a radius, we select all the points that are within octants that are within radius from the source point. We grab points that may lie up to 1 octant away from the radius desired. In any practical implementation, there must be a cap on the total number of points to grab (we choose 1000) and these points are selected arbitrarily from all points within the highlighted octants. Thus this algorithm is approximate by nature, as the nearest neighbor may not be contained in the set even if it is contained in the radius.

NOTE: For this octree algorithm, no single radius is sufficient to cover every use case. With point clouds that can be capturing any kind of scene, there are no guarantees on the densities at any point in space. Our base guiding heuristic is that if the cloud has uniform density, close to 250 points will lie within a voxel with side length 2*radius. This yields the following simple proportion:

$$\frac{\#points}{cloud\_volume} = \frac{31.25}{x^3}$$

For example, if a point cloud is 3m by 3m by 2m and contains 250,000 points, we would have the radius be around 0.12m. We choose this target of 250 points since we expect density to be extraordinarily high in many places, with as many as 10 times the density of an averaged density.

We conducted an exhaustive search of different radii for several point clouds and saw variation from our rule of thumb of up to around 50% in either direction.

3.5 Parallelizing Chamfer Distance

With these three serial approaches, we move onto parallelization. We choose to use OpenMP and CUDA, as these are the tools most likely accessible to applications needing Chamfer distance.

3.5.1 OpenMP Implementations

All-Pairs OpenMP all-pairs Chamfer distance implementation leverages the independence of forward and backward Chamfer computation routines. We tried a few different approaches in utilizing OpenMP. In our first approach we only used a `#pragma omp parallel for schedule(dynamic) private(minsofar)` over the forward and backward for loops asking for the maximum number of threads. In this approach, we utilize the fork-join parallelism in OpenMP to launch a team of threads work-sharing the forward and backward computations of Chamfer distance. In each direction after initialization, the master thread splits up the for loop into independent chunks of computations and share the points in cloud A and cloud B (forward and backward respectively) dynamically. We opted for dynamic worksharing here after testing with the static work sharing showed slower times. We think the reason for this is the non-uniform density in point clouds thus dynamic scheduling worked better. Each thread then computes the `minsofar` value that it sees in cloud B and because this value needs to be updated in a thread-safe way, we used a `private` flag to make it thread-local. After it computes the minimum distance in cloud B the threads then hit the critical zone to update the running total of distances. To prevent numerical issues stemming from race conditions of multiple threads writing at the same time to the running sum, we utilize a `#pragma omp critical` block to update the sum with `minsofar` value each thread has computed. We do the same for backward computation. After computing the distances in forward and backward passes, the master thread computes the weighted averaging and outputs.

In our second approach using OpenMP, we tried to further parallelize our initial approach given above by utilizing min reductions in the inner for loop. For the inner for loop located in both forward and backward loops, since min operation is associative we can utilize a reduction operation in OpenMP to leverage further parallelization. We define a new pragma on top of the inner for loops `#pragma omp parallel for`

`reduction(min : minsofar)` which reduces on the `minsofar` value. Each spawned thread hits this and then spawns and other team of threads which work collectively to reduce. After the reduction we follow the same steps as above with the critical zone to update the forward distance total and backward distance total.

After testing on Cori and on AWS, we realized that the second approach is not actually more efficient than the first approach and while we are not fully certain why this is happening, we think that it is because of the over-parallelization that's being caused due to each thread launching another set of threads in each for loop. In other words if 32 threads are given to work share the outer for loop, each of these threads will attempt to launch another set of threads to reduce which slightly increases the overhead and thus the computation time.

KD-tree Approximate Nearest Neighbors We follow the all-pairs OpenMP implementation with a few minor changes. Firstly, before forward or backward computations, the master thread builds two KD-trees. The first KD-tree is built using the points of cloud A and the second KD-tree is built using the points in cloud B. Here we also initialize the two vectors first of which will hold the nearest neighbor search results (points) and the second will hold their corresponding distances used when computing nearest neighbors. In both forward and backward computations we workshare using a team of threads on the outer for loops with `#pragma omp parallel for schedule(dynamic) private(minsofar)` and each spawned child thread, when computing the minimum distance value, utilizes the KD-tree to conduct an approximate nearest neighbor search. The search returns a pre-set number of nearby points (controlled by one of the input arguments) and we then compute the minimum within that constrained set of points. In order to avoid race conditions we declare not only the `minsofar` but the two additional vectors that house the returned points and their corresponding distances as `private`. For all of our KD-tree based OpenMP code executions, we used a 100 points to return in nearest neighbor searches which we found to increase the accuracy of the returned distance result (cross checked with the all-pairs distance result) at the slight expense of computation time.

Octree Radius Nearest Neighbors Similar to the KD-tree based implementation, the Octree based implementation is as follows. Firstly, in the master thread, we initialize the global variables such as the forward and backward distance total (which are both set to zero as before) and we also read the two point clouds and construct two separate Octrees over the points in cloud A and cloud B respectively (similar to the KD-tree approach). We then initialize the two vectors which are going to be used to house the radius based nearest neighbor search results (points) and their corresponding squared distances. Different from previous implementations we initialize an additional variable `approxdist` that will house the approximate minimum distance after the radius based search is conducted in the inner for loop (which is situated as before inside the forward and backward chamfer routines). This variable is also used in case the radius search returns empty result lists which happens often due to the inherent sparsity of point clouds. If the radius search returns empty, we approximately pick a nearby point defined by the Octree API. If the radius search returns points we then pick the minimum distance contained in that list. The rest is the same as KD-tree implementation. Similarly to KD-tree, we declare the two vectors and the `approxdist` as `private` in order to avoid race conditions. In the Octree implementation we found that setting following parameters worked the best so far in terms of striking a balance between computation performance and computation accuracy. When building the Octree we set the "resolution" to `octree_a(0.01f)` and when doing the radius search we do: `0.1f` for search radius value.

3.5.2 CUDA (GPU) Implementations

All-Pairs Our GPU parallel all-pairs Chamfer distance implementation is fairly simple. The implementation consists of 3 sections: data transfer to the GPU, the compute kernel, and a reduction. The only data needed for either direction of the Chamfer distance calculations are the raw point arrays for

each cloud. Although this cost does increase linearly with cloud size, it is most apparent for small point cloud sizes, as the quadratic compute cost quickly dwarfs the transfer cost. In addition to holding the data, an additional `double` array is CUDA allocated to hold minimum distances for each point. This makes our peak GPU memory consumption `#total_points * sizeof(float) * 3 + sizeof(double) * larger_cloud.size`. This data upload overhead is discussed in the Results section.

The direction agnostic compute kernel functions similarly to HW 2.3, where each GPU thread is "assigned" one point in cloud A `point_id = threadIdx.x + blockIdx.x * blockDim.x` and iterates over all points in cloud B to find the minimum distance. Once the minimum distance is found, it stores the minimum distance in the double array at `point_id`, which was passed by pointer to the function, for reduction in the future. Using this array as opposed to a single variable avoids needing to protect distance updates with a critical section.

Once the minimum distance array is filled, the final step is an accumulation over the minimum distance array to find the sum of minimum all-pairs distances for one direction. This entire process is repeated with swapped parameters for the backward direction before combining and returning the overall Chamfer distance.

This one-point-per-thread approach performs exceptionally well due to the compute power of the v100. With 5120 threads per SM and 80 SMs, we can execute over 400k threads simultaneously to find our minimum, greatly improving the parallelism.

Octree Radius Nearest Neighbors The CUDA octree algorithm has one key difference from our OpenMP implementation. We opt not to conduct any nearest neighbor search on points with no neighbors within the radius. In our tests, less than 0.1% of points have no neighbors with a suitable radius. Because of this, we assume that points that have no neighbors are outliers and should be discarded. We expected a drastic decrease in accuracy but due to the small proportion of points being discarded we see accuracy staying ¿95% when the proper radius is chosen.

Another consideration is that adding a diverging branch into the compute kernel could cause unnecessary slowdowns on the GPU due to an additional masking being used on top of the one needed due to varying numbers of returned neighbors. We see little reason to have 0.1% of threads essentially blocking the remaining 99.9% of threads for such a minimal accuracy improvement for a method that is fundamentally approximate.

Our GPU-parallel octree radius Chamfer distance algorithm has two key steps per direction in addition to the standard upload and accumulation steps. The first is to conduct the radius search to locate up to 1024 neighbors of each point, and the second is to find the minimum distance from each of these neighbors.

We utilize PCL's out of the box GPU-based octree (`pcl::gpu::Octree`) to simplify implementation. Since their octree requires a device-based input cloud, cloud b is unfortunately duplicated in device memory, once as the device-based cloud and once as the octree backing. We also upload cloud a to a device-based input cloud, which is used to query in the forward direction and build the octree in the reverse direction. Finally, to use PCL's octree radiusSearch, we allocate an array to hold the indices of the neighbor points, which is 1024 * the size of the larger cloud. Thus, our GPU peak memory consumption is `3 * sizeof(float) * (#total_points + larger_cloud.size) + 1024 * sizeof(int) * larger_cloud.size`, significantly more than the all-pairs implementation. Fortunately, V100s have an immense amount of GPU memory and can thus handle point clouds well into the millions of points, which is much larger than we expect from our use case to begin with.

As with the non-GPU implementation, PCL's radiusSearch takes all points within the radius using the spatial partitioning from the backing octree and selects the first 1024 points it encounters.

The kernel function operates similarly to the all-pairs kernel but only checks the corresponding neighbor indices in cloud b that are reported to be within radius of the point of interest in cloud a. Thus, each thread has to iterate over a maximum 1024 cloud b points, making the compute kernel a constant

asymptotic cost.

Octree Nearest Neighbors For the octree exact nearest neighbor algorithm, we utilize PCL's built in batchKNNSearch, which is only implemented at the moment for k = 1. Once an octree is built on cloud b, the octree is able to take in the entirely of cloud a as a sequence of queries and fill an array with the squared distances to the nearest neighbor. This API enables us to avoid a handwritten compute kernel and simply conduct an accumulation over the results array to generate a sum for either direction. Since our upload, building, and accumulation steps are standard and already discussed in depth, we dove into batchKNNSearch to understand the implementation.

At the core, PCL's octree-based 1-NN search algorithm is standard as previously described in section 3.3.2. PCL implements the GPU-specific algorithm by splitting queries into batches and assigning each batch to a warp. A compute kernel then handles the actual search process, filling in the NN index and computed distance. PCL's implementation is highly optimized and we doubted that we could make any tuning improvements so we used the implementation as-is.

4 Results

For serial and OpenMP implementation, we utilized Cori's Haswell nodes. In addition to this, we tested our OpenMP implementations on a AWS C5.18xlarge node. This duplicate effort was born our of a desire to see how the results compared between different kinds of providers. Unfortunately, for the GPU implementation, we were unable to get Cori to properly build the many necessary PCL GPU libraries. In order to still get some results, we grabbed an AWS P3 node. Architecture details are listed in the below table.

Node Type	Cori Haswell	AWS C5.18xlarge	AWS P3.2xlarge
Processor	Intel Xeon Processor E5-2698 v3	Gen2 Intel Xeon Scalable Processors	Intel Xeon E5-2686 v4
Cores	32	72	8
Clock Speed	2.3GHz	3.4GHz	2.3GHz
GPU	N/A	N/A	NVIDA Tesla V100

Table 2: Node Comparison

4.1 Strong Scaling in OpenMP

We performed strong scaling experiments on the medium dataset with 1, 2, 4, 8, 16, 332, and 64 cores with no hyperthreading. For the naive algorithm, we see perfect scaling efficiency up to 32 threads (1600s vs 50s) but only a very minor speedup at 64 threads. We think this is due to the critical section for accumulating the sum distances. As expected, the runtime of the kdtree and octree-based implementations are several orders of magnitude faster, but we see a drop-off in performance beyond 16 cores. We hypothesize that there is a decent amount of thrashing occurring with access to the built data structure in addition to the critical section contraints.

4.2 OpenMP vs CUDA Performance

One surprising result is how well OpenMP performed in comparison to the CUDA implementation. Although CUDA outperformed OpenMP in all but one instance, The parallelism yielded by OpenMP

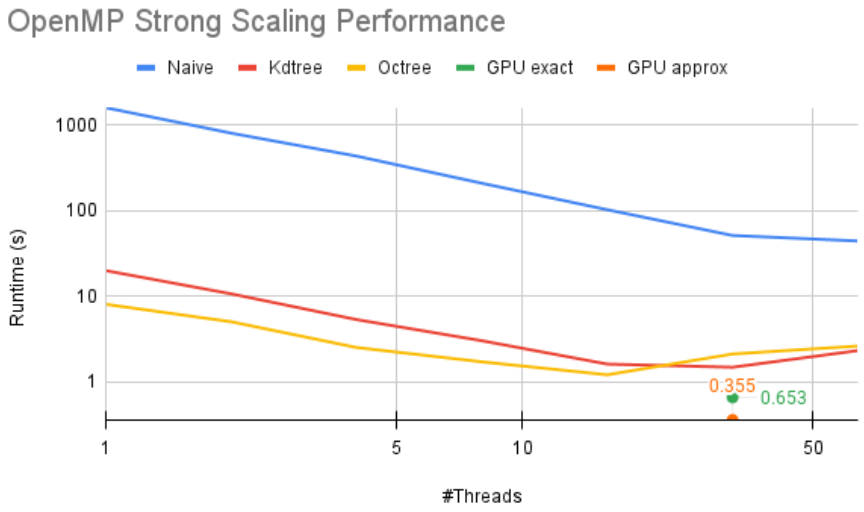

Figure 3: Strong scaling results with increasing core numbers in OpenMP using the medium dataset. CUDA performance on the same dataset given as well.

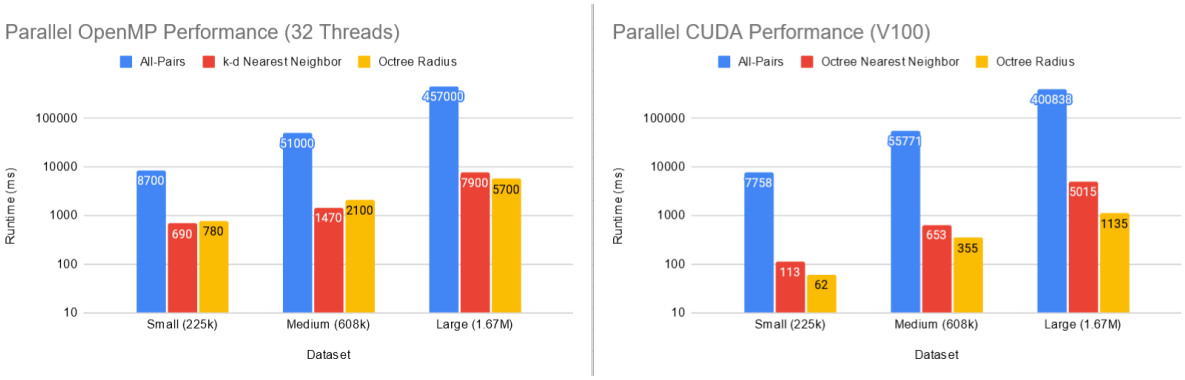

Figure 4: Algorithm performances with varying problem sizes. OpenMP given on the left and CUDA given on the right.

was nearly as powerful as the CUDA implementation. This is extremely good to see, as most consumer devices will not have a V100 but rather have OpenMP capabilities.

4.3 Cori vs AWS Comparison

Overall, the runtimes between Cori and AWS are quite comparable. This is somewhat surprising due to the higher clock rate of the AWS Cascade Lake processors but perhaps would become apparent at large dataset sizes. We do note that AWS tended to have some warmup efficiency quirks that Cori did not have, where the first run in a series of runs would be significantly slower. We believe this is mostly due to the slower data transfer from Amazon EBS to EC2.

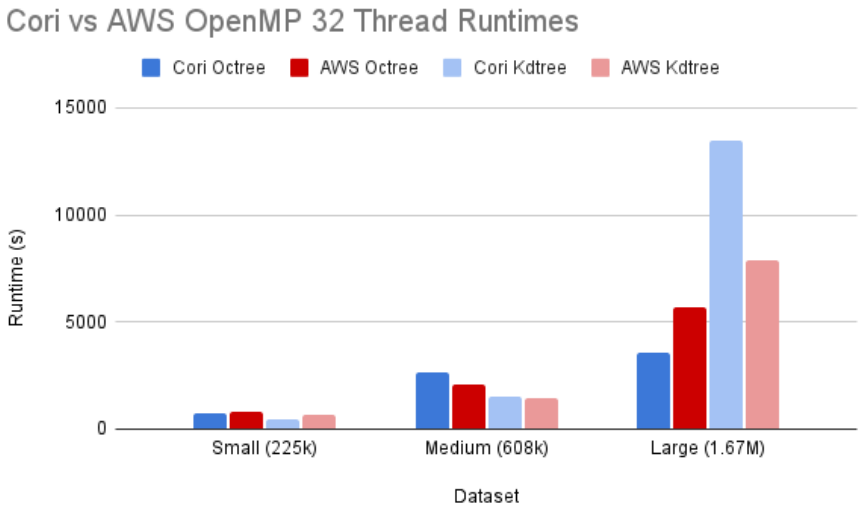

Figure 5: OpenMP Algorithm performances on Cori (Blue) and AWS (Red)

Dataset	Octree	Kdtree
Small	0.06	0.09
Medium	0.16	0.27
Large	0.4	0.8

Table 3: Data structure construction overhead time in seconds

4.4 Algorithm Overheads

We also calculated the overheads for Octree and Kdtree construction on OpenMP. This overhead is negligible, with the construction time being less than 5% of the total runtime, making the overall speedup well worth the upfront cost.

5 Conclusion

Our main goal was to assess the feasibility of parallelizing Chamfer distance calculations. To do this, we first tackled the quadratic nature of the distance calculations by building kdtree and octree representations of our target search clouds. This brought the complexity down to nearly linear in the number of points of the cloud. Once these algorithms were settled we tested OpenMP and CUDA implementations of these improved algorithms, showing that both the naive and optimized algorithms can scale efficiently in both paradigms. We conclude that both approaches are feasible and should be considered for any application that utilizes Chamfer distance calculations.

6 Future Work

Overall, we'd like to understand how to pick a good radius for our approximate radius search. As we mentioned, there was a decent amount of deviation from our rule of thumb and would like to explore efficient ways to perhaps test out different radii at low costs.

We'd also like to optimize OpenMP implementations to avoid the memory contention which is potentially slowing down implementations with many threads.

References

[1] C. R. Qi, H. Su, K. Mo, and L. J. Guibas, "Pointnet: Deep learning on point sets for 3d classification and segmentation," 2017.

[2] D. Maturana and S. Scherer, "Voxnet: A 3d convolutional neural network for real-time object recognition," in *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2015, pp. 922–928.

[3] S. Chen, B. Liu, C. Feng, C. Vallespi-Gonzalez, and C. Wellington, "3d point cloud processing and learning for autonomous driving," 2020.

[4] T. Q. Pham, "Parallel implementation of geodesic distance transform with application in superpixel segmentation," in *2013 International Conference on Digital Image Computing: Techniques and Applications (DICTA)*, 2013, pp. 1–8.

[5] OpenMP Architecture Review Board, "OpenMP application program interface version 3.0," May 2008. [Online]. Available: http://www.openmp.org/mp-documents/spec30.pdf

[6] C. Koc, C. Koc, B. Su, C. Casarez, and R. Fearing, "Body lift and drag for a legged millirobot in compliant beam environment," 2019.

[7] Wikipedia contributors, "Hausdorff distance — Wikipedia, the free encyclopedia," 2004, [Online; accessed 18-March-2021]. [Online]. Available: https://en.wikipedia.org/wiki/Hausdorff_distance

[8] Y. Rubner, C. Tomasi, and L. J. Guibas, "A metric for distributions with applications to image databases," in *Sixth International Conference on Computer Vision (IEEE Cat. No.98CH36271)*, 1998, pp. 59–66.

[9] D. Urbach, Y. Ben-Shabat, and M. Lindenbaum, "Dpdist : Comparing point clouds using deep point cloud distance," 2020.

[10] R. B. Rusu and S. Cousins, "3D is here: Point Cloud Library (PCL)," in *IEEE International Conference on Robotics and Automation (ICRA)*, Shanghai, China, May 9-13 2011.

[11] D. A. Suju and H. Jose, "Flann: Fast approximate nearest neighbour search algorithm for elucidating human-wildlife conflicts in forest areas," in *2017 Fourth International Conference on Signal Processing, Communication and Networking (ICSCN)*, 2017, pp. 1–6.

[12] I. S. Drost, B.H., "Almost constant-time 3d nearest-neighbor lookup using implicit octrees." 2017. [Online]. Available: https://doi.org/10.1007/s00138-017-0889-4