# Parallel Point Cloud Registration

Yujie Wei, Hanzhou Lu | CMU-15418 2016 Fall
Website: https://hanzhoulu.github.io/Parallel-Point-Cloud-Registration/

## SUMMARY

In this project, we implemented the Iterative Closest Point Algorithm (ICP) for point cloud registration using CUDA and compared it with the implementation on CPU (Pure Eigen) and on OpenMP. We also compared the performance using conventional brute force search, k-d tree search, and our compressed k-d tree search algorithm on these machines.

## INTRODUCTION

### PROBLEM

Point Cloud Registration (PCR) plays an important role in computer vision since a well-aligned point cloud model is the bedrock for many subsequent applications such as Simultaneous Localization and Mapping (SLAM) in the robotics and autonomous cars domain or Automatic Building Information Modeling in the architectural industry. Nowadays, point clouds are usually gathered by multiple cameras or laser scanners with their own coordinate systems. The objective of PCR is to search a transformation that could align a reading point cloud with a reference point cloud in a consistent coordinate system. The process of finding the transformation and the closest point involves lots of matrix operations that are usually independent with each other. Given the intermediate level of dependency and the huge size of the problem, exploiting the parallelism can be a good alternative to speed up the algorithm.
The objective of the algorithm can be formally expressed as follows:

$$\mathcal{T}_A^B = \min_{\mathcal{T}}(Error(\mathcal{T}(\mathcal{P}_A), \mathcal{Q}_B))$$

where *T* is the transformation, *P* is the reading point cloud captured in the coordinate System A, and Q is the reference point cloud captured in the coordinate system B. The error is defined as the distance between each closest point pairs in different point clouds. The generic algorithm (Besl, 1992) is shown below:

**Algorithm 1** Summary of ICP algorithm.

**Require:** ${}^{\mathbb{A}}\mathcal{P}$ ▷ `reading`
**Require:** ${}^{\mathbb{B}}\mathcal{Q}$ ▷ `reference`
**Require:** $\mathcal{T}_{init}$ ▷ initial transformation
  ${}^{\mathbb{A}}\mathcal{P}' \leftarrow \text{datafilter}({}^{\mathbb{A}}\mathcal{P})$ ▷ data filters
  ${}^{\mathbb{B}}\mathcal{Q}' \leftarrow \text{datafilter}({}^{\mathbb{B}}\mathcal{Q})$ ▷ data filters
  ${}_{i-1}{}^{i}\mathcal{T} \leftarrow \mathcal{T}_{init}$
  **repeat**
    ${}^{i}\mathcal{P}' \leftarrow {}_{i-1}{}^{i}\mathcal{T}({}^{i-1}\mathcal{P}')$ ▷ move `reading`
    $\mathcal{M}_i \leftarrow \text{match}({}^{i}\mathcal{P}', \mathcal{Q}')$ ▷ associate points
    $\mathcal{W}_i \leftarrow \text{outlier}(\mathcal{M}_i)$ ▷ filter outliers
    ${}_{i}^{i+1}\mathcal{T} \leftarrow \arg\min_{\mathcal{T}} \left(\text{error}\left(\mathcal{T}\left({}^{i}\mathcal{P}'\right), \mathcal{Q}'\right)\right)$
  **until** convergence
**Ensure:** ${}_{\mathbb{A}}^{\mathbb{B}}\hat{\mathcal{T}} = \left(\bigcirc_i {}_{i-1}{}^{i}\mathcal{T}\right) \circ \mathcal{T}_{init}$

Figure 1 ICP Algorithm

The program will first apply filters to the point clouds to remove outliers, then start searching from the initial transformation. In each iteration, the algorithm establishes the correspondence, creates point pairs from the reading and reference, conducts transformation over the reading point cloud, and find the transformation that minimizes the match error. The process will continue until the result converges or reaches the maximum number of iteration.
A more recent implementation of ICP is shown below:

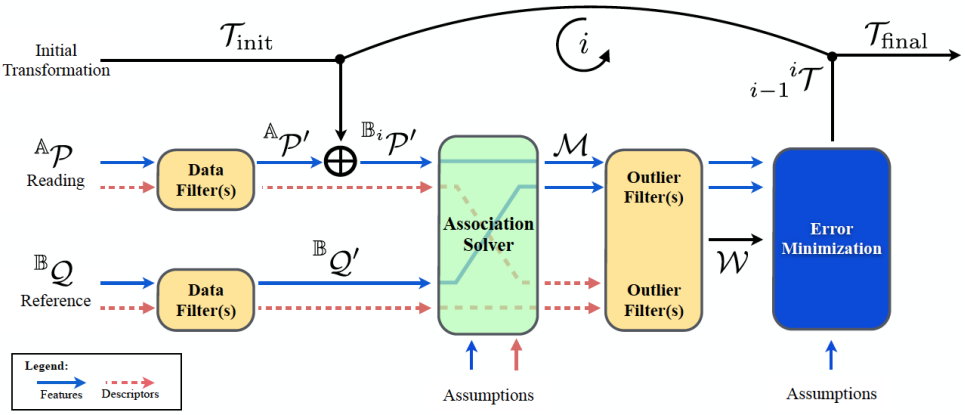



Figure 2 Generic Scheme for Registration Algorithms (Pomerleau, 2015)

## RELATED WORK

The challenges of the project come from the problem size and the dependency of the transformation series. A typical point cloud usually contains millions of points that may cause the cache performance to be quite poor. Also, notice that the next transformation has to be calculated based on the previous result that limits the level of parallelism. Nowadays, to make the algorithm more robust when dealing with occlusions and outliers, some refined versions of ICP adopt different distance metrics and objective functions that bring new challenges. Our goal is to develop a robust parallel implementation of ICP that could be extended in the future to handle unexpected changes.
Previous research concentrated on two directions: First, increase the parallelism of the searching algorithm such as splitting the point cloud data and use each thread to calculate part of the point cloud. Second, reduce the amount of computation by employing approximate algorithm such as Quaternion method (Langis, 2001) or applying advanced data structures. According to S. Rusinkiewicz and M. Levoy (2001), the k-d tree is a good candidate for ICP searching.

1. Our Approach and contribution

Based on previous research, we parallelized both the traditional algorithm and the K-d tree search algorithm. On the traditional search algorithm, we employed the shared address memory model and split the data to parallelize it over the $O(mn^2)$ complexity. We then implemented a parallel K-d tree version to reduce the computation amount to O(mnlog(n)) while making it still friendly to data-parallel model. We also added a quicksort for the K-d tree implementation to remove outliers that has the largest distance, which made the algorithm more robust. After implementing the naive K-d tree, we realized that the communication time between CPU and GPU became non-trivial. To further decrease the time when there was no more speedup as the number of threads increases, we implemented a compressed K-d tree and applied bucketsort to balance the speedup and the accuracy at the same time.

# IMPLEMENTATION

## BRUTE-FORCE SEARCH WITHOUT SORTING

We first implemented the parallel brute-force search without sorting. As this method was suitable for the data parallel model since if the data was simply stored in a long array, we could easily split the data into smaller pieces and assign them to multiple threads. We implemented the brute-force algorithm on OpenMP and CUDA. The speedup was quite good as the number of thread increases. However, the approach was not perfect due to two reasons: First, we didn't use the resources wisely. Even the speedup was good, the computation amount was actually proportional to $O(mn^2)$ where m was the number of iterations and n was the number of points in the point cloud. Second, the accuracy was not good. The approach assumed that all points in point cloud A had a corresponding point in point cloud B yet this was not true. To reduce the computation amount for each thread fundamentally, we should organize the data as a tree instead of a plain array. To improve the accuracy of the final result, we should add a outlier detection module that sorts the corresponding pair and removes those with highest distance.

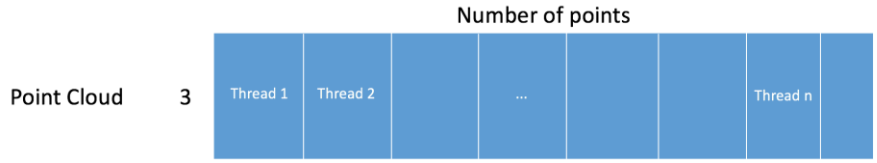



Figure 3 Brute-Force Parallelism

## K-D TREE WITH QUICKSORT

We then thought of implementing our algorithm using K-d tree. K-d tree (short for K-dimensional tree) is a space-partitioning data structure for organizing points in a k-dimensional space. Conventional K-d tree saves points' data in each node, and use these nodes to partition remaining points. Each node in a K-d tree is a point in the point cloud.

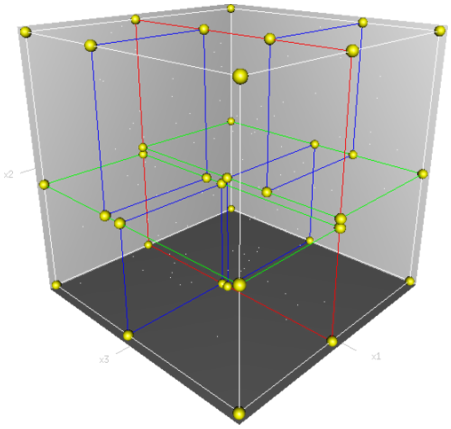


Figure 4 Conventional K-d tree structure

The process of the K-d tree version was shown below. The host (CPU) first parses the point cloud and initialized the environment. Then the device (CPU) will construct the K-d tree based on the point cloud position and send the K-d tree to device. Device will then conduct the search, sort, and filer operation, then send the data back to the host. The host will take charge of SVD

decomposition and transformation. Then the whole program will move to the next iteration until the MSE converges to a certain level.

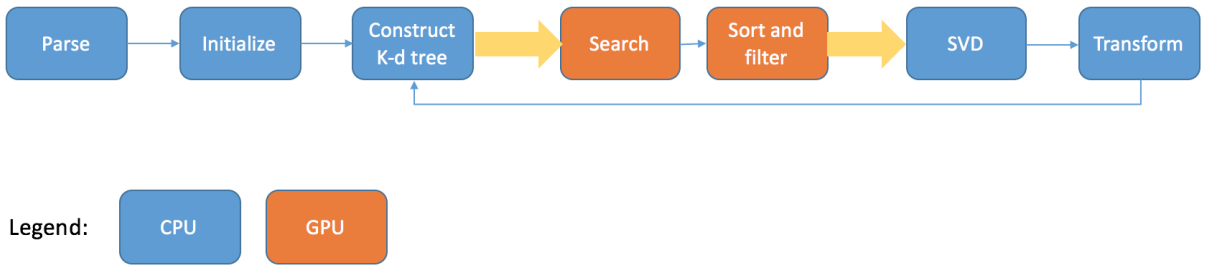



Figure 5 Program flow

After adding the outlier removal module, the program can converge much faster and converge to a better result as Figure 6 shows.

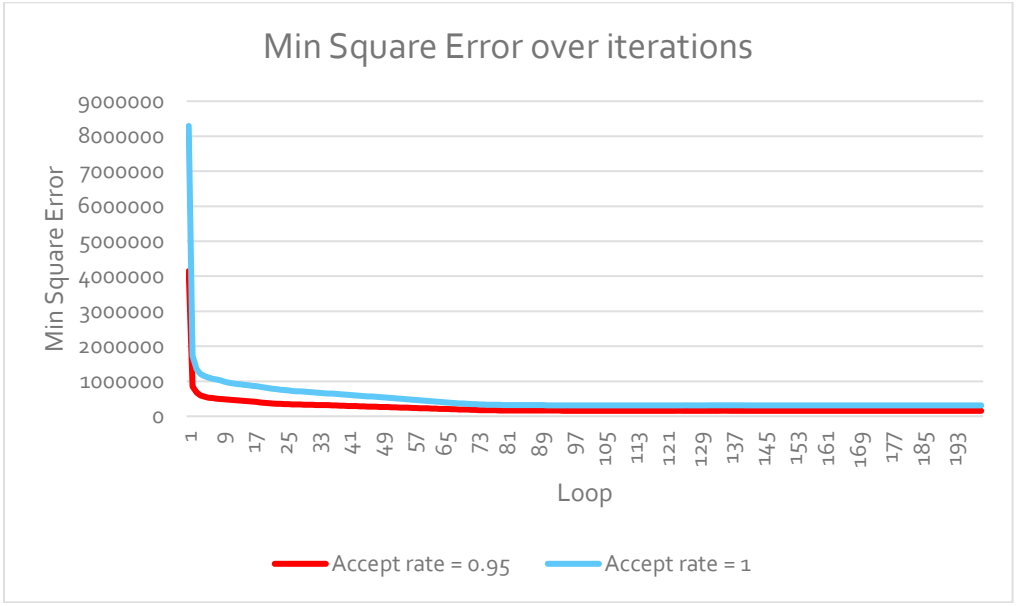



Figure 6 MSE with/without outlier removal

The difficulty here was that since the data was no longer organized as a simple array, the whole K-d tree must present when the device doing the search and sorting operations. Also since the tree is based on the point's position, the point cloud data must be transferred back and forth between the CPU and GPU. After profiling the code, we found that when the number of thread is small, the computation part takes above 98% of the time. However, as the number of threads increases, the communication time becomes significant and takes about 75% or more when the total number of threads is larger than 32 in CUDA. Another interesting observation is that the sorting actually takes a significant amount of time as well that could be a possible optimization as well. Also, we realized that searching in K-d tree was not balanced since the time it took

depended on how close these points were (Figure 7). The imbalanced work decomposition prevented the program from reaching the ideal speedup.

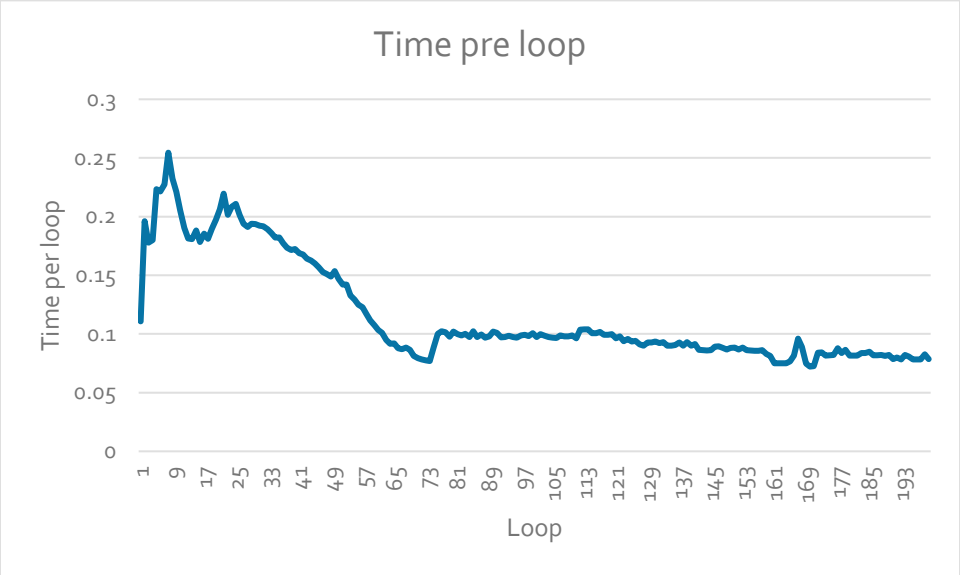



Figure 7 Time per loop as # of iteration increases

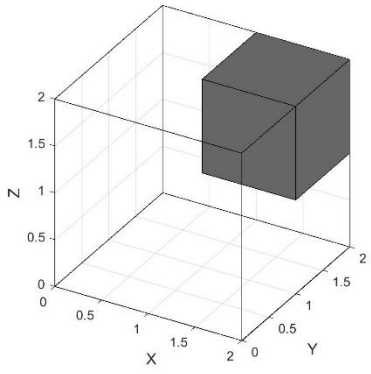
Different from brute force search, time consumed per loop in k-d tree implementation is different. As we registration process going, two point clouds are closer to each other. That means k-d tree searching will become faster. The more iterations we do, the better performance of each loop will be.

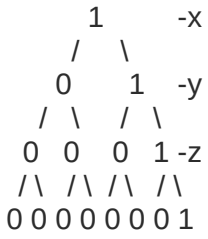
## COMPRESSED K-D TREE WITH BUCKETSORT

Realizing where the problem is, we began to find a way to reduce the amount of computation. Considering the problem that traditional k-d tree need to save coordinates of each point, one point will take (3 * sizeof(float)) + (2 * sizeof(void *)) = 40 bytes. With the conventional structure, we need 3 floats saving the x, y, z coordinates and 2 pointers pointing to its child. That will take a lot of storage space and will be time-consuming to copy it from CPU to GPU and vise versa. Therefore, we considered compress the K-d tree to reduce the communication overhead.

### CONSTRUCTION

Then we come up with a compressed k-d tree structure which saves us up to 90% memory space. In a compressed k-d tree, we use the concept of heap (priority queue) which is a complete tree. Firstly, we make all the points in a point cloud to be the leaf node of its k-d tree and round them to 2n(n is the smallest integer which make 2n > numOfPoints). Secondly, we do logical or operation between each pair of adjacent nodes. These 2n-1 nodes form the last but one layer of the tree. We do this iteratively until we have only 1 node left which is the root.




This process may be confusing, let's take a real k-d tree as an example:

Figure on the left shows a simple k-d tree representing a point (1, 1, 1) in a 2 by 2 by 2 cube space. We have 8 points: (0, 0, 0), (0, 0, 1), (0, 1, 0), (0, 1, 1), (1, 0, 0), (1, 0, 1), (1, 1, 0), (1, 1, 1) in this space and only (1, 1, 1) is occupied. So, we convert these 8 points into 0, 0, 0, 0, 0, 0, 0, 1. These are the 8 leaf nodes of this k-d tree. Then we do logical or operation between each two nodes and build their parent node. By such analogy, we can build a full tree. The first root 1 means that there is at least one point in the whole space. It's right child 1 means that in the space $x > 0$, there is at least one point. The third 1 means there is at least one point between (1, 1, 0) and (1, 1, 1). The leaf node 1 means (1, 1, 1) is occupied by a point.

Each time we move from one layer to its child layer, we do a simple dichotomy which divide the space by half. The division pattern will be like x-, y-, z-, x-, y-, z-…… At last we will divide the space into little cube which is 1*1*1. And these cubes are leaf nodes of a k-d tree.

Figure 8 Compressed k-d tree example

## EFFICIENCY

Like heap, we can easily find parent node and child nodes of a specific node by maintain an array:

- Parent node index = Child node index / 2;
- Left or Right child node index = Parent node index * 2 + (1 or 2);
- Furthermore, we can easily calculate point's coordinate by its index in this array.
- For example, a leaf node with index 90 is in an 8 layers compressed k-d tree. And we can get its coordinate in the space in the method below. (This space partition is in x-, y-, z-, x-, y-, z-, x- pattern.)
- Index of first leaf node = 26 – 1 = 63;
- 90 – 63 = 27 = 0x0011011;
- CoordinateX = (0 0 1 1 0 1 1) 0x011 = 3;
- CoordinateY = (0 0 1 1 0 1 1) 0x00 = 0;
- CoordinateZ = (0 0 1 1 0 1 1) 0x11 = 3;
- Coordinate of this node is (3, 0, 3).

## COMPRESSION

The compression ratio of k-d tree is related to many factors. Density of the points, granularity of cubes, and size of the space will influence compression ratio together. The table below shows how compression ratio changes with the scale of k-d tree.

| Scale | #Points | Size of traditional k-d tree(bits) | X axis length | Y axis length | Z axis length | Size of compressed k-d tree(bits) | Compression Ratio |
|---|---|---|---|---|---|---|---|
| 1200 | 27441 | 8781120 | 256 | 256 | 128 | 16777216 | 0.523396 |
| 1000 | 21551 | 6896320 | 256 | 128 | 128 | 8388608 | 0.822105 |
| 800 | 15485 | 4955200 | 128 | 128 | 128 | 4194304 | 1.181412 |
| 600 | 9750 | 3120000 | 128 | 128 | 64 | 2097152 | 1.487732 |
| 400 | 4823 | 1543360 | 64 | 64 | 64 | 524288 | 2.943726 |
| 200 | 1354 | 433280 | 64 | 32 | 32 | 131072 | 3.305664 |
| 100 | 389 | 124480 | 32 | 16 | 16 | 16384 | 7.597656 |

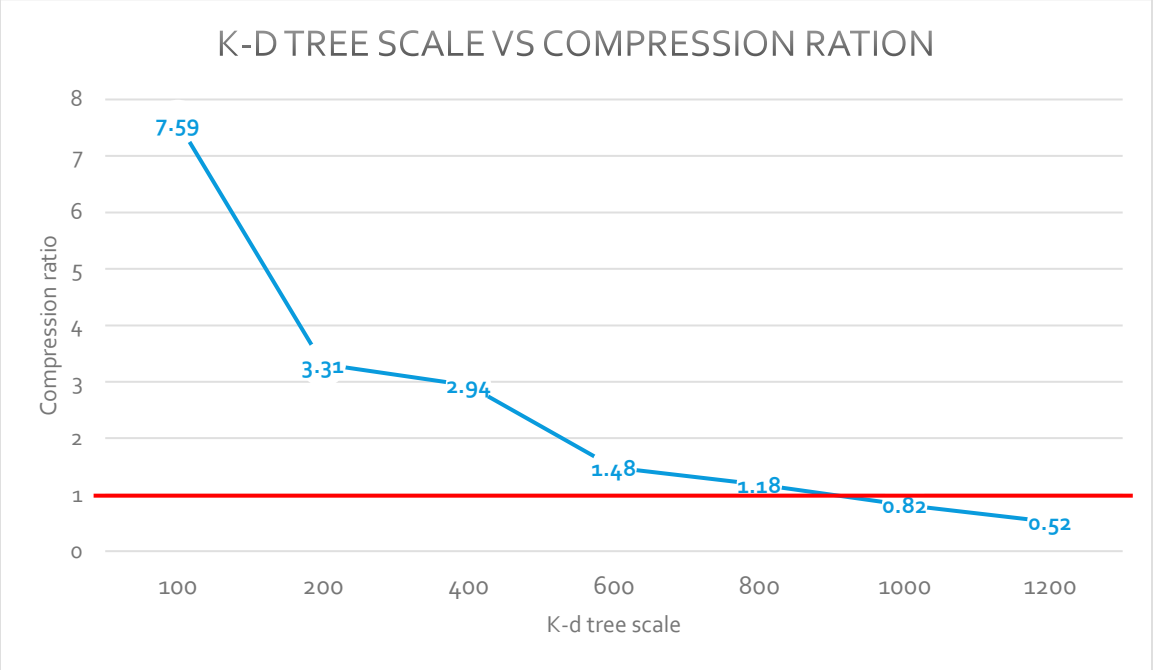



Figure 9 Compressed k-d tree VS traditional k-d tree

From the chart, we can tell that the compression ratio is quite well when the scale of k-d tree is small. A compression ratio of 7.60 means we can save 87% memory space by using compressed k-d tree. However, the ratio decreases as scale grows bigger. When it comes to 1000 level scale, we cannot get more space form compressed k-d tree. But we can still use compressed k-d tree if we increase the density of points. Since the scale is fixed, the k-d tree will not grow. Meanwhile, it can represent more points. The compression ratio will increase.

## BUCKET SORT

We also changed the way how the outlier removal module works. In real word problem, different point clouds always have points which cannot be matched. Some points in point cloud A do not exist in point cloud B. In order to rule out whose points which cannot be matched, we need to

sort all matches found by k-d tree by their Euclidean distance. We will set a parameter called acceptRate to control the number of pairs we will accept from those sorted pairs.

Then we find that doing merge sort or quick sort will require a large amount of message passing and data exchange. And the range of Euclidean distance of each two points is fixed in each loop ([0, MaxDistance]). So, we decide to use bucket sort as an approximate sort among those pairs of points.
In each thread, we maintain an array which keep track of its local counter of each bucket. After we finish searching, we sum this array of each thread and see how many buckets will be accepted.
For example, we are running a 2 threads program. Each thread calculates 40 points. AcceptRate = 0.9.

Thread 1:

| Count | 1 | 2 | 1 | 1 | 2 | 3 | 1 | 4 | 1 | 4 | 2 | 1 | 3 | 1 | 1 | 3 | 2 | 1 | 2 | 4 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | >=19 |

Thread 2:

| Count | 1 | 1 | 1 | 2 | 1 | 2 | 1 | 5 | 1 | 8 | 4 | 1 | 2 | 2 | 2 | 2 | 2 | 1 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | >=19 |

Sum:

| Count | 2 | 3 | 2 | 3 | 3 | 5 | 2 | 9 | 2 | 12 | 6 | 2 | 5 | 3 | 3 | 5 | 4 | 2 | 3 | 4 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | >=19 |

Number of accept pairs = 20 * 2 * 0.9 = 36;


| Count | 2 | 3 | 2 | 3 | 3 | 5 | 2 | 9 | 2 | 12 | 6 | 2 | 5 | 3 | 3 | 5 | 4 | 2 | 3 | 4 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | >=19 |

Then we can choose those pairs whose distance is less than 18.

Because we use shared memory model in our implementation, the sum array is saved in shared memory space and can be accessed by each thread. The number of buckets will influence the amount of data needed to be passed.

## EXPERIMENT SETUP

### ENVIRONMENT AND LIBRARY

The CPU part of the program was tested on Latedays cluster equipped with two Xeon E5-2630 v3 processors. The GPU part of the program was tested on a NVidia GTX 1070 graphic card.
Put the following libraries in the picp/library path:

- Eigen 3.3.1
- Boost 1.62.0

Install the following libraries in your computer:

- GCC 5.0 or above
- OpenMP
- CUDA 7.5
- Point Cloud Library 1.7.2

## POINT CLOUD FILE FORMAT

The point cloud format should follow the PLY format to be correctly parsed by the code.

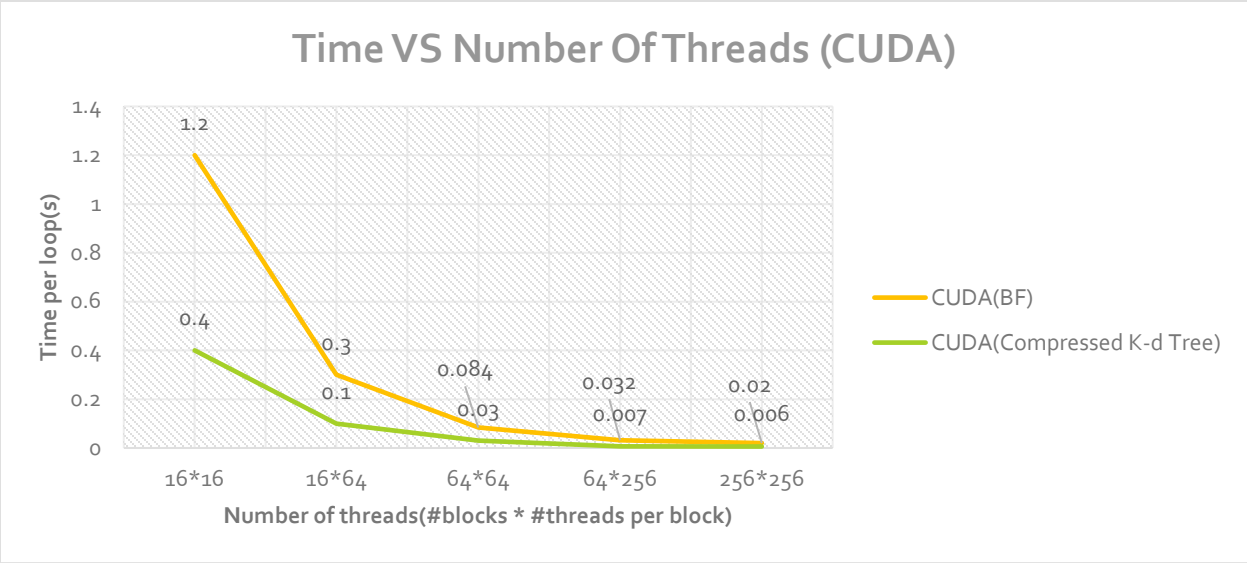
## COMPILE & SETUP

Enter the picp root library and use the following command to compile.

- make: Create a parallel CPU program called “icp”
- make gpu: Create a parallel GPU program called “icpgpu”

# EXPERIMENT EVALUATION

## ACCELERATION RESULT



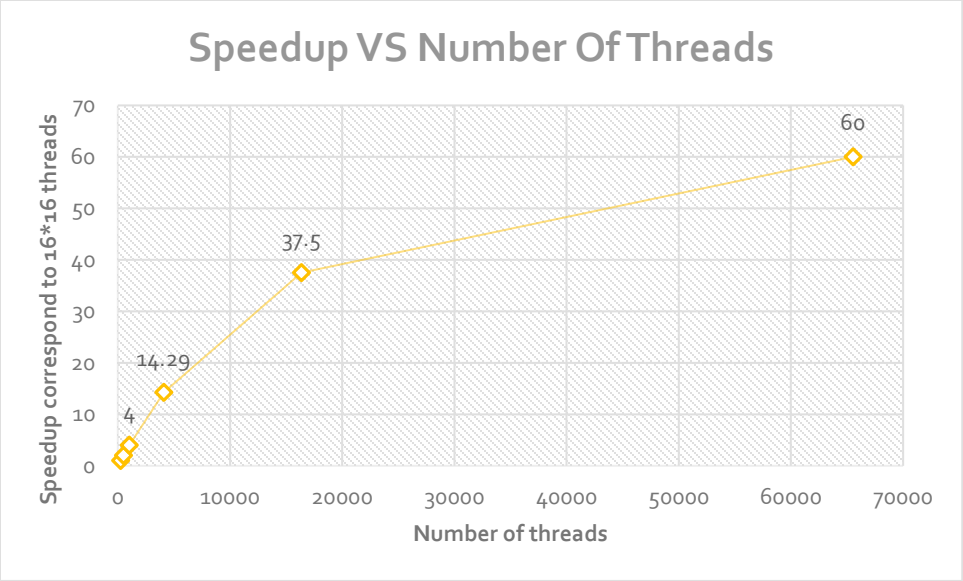



Figure 10 Speedup on CUDA

From the figure above we can tell that brute force solution is far slower than k-d tree solution. At first, speed up is scale with the number of threads. However, when number of threads grows larger, speed up increasing rate drops because of calculation spent on message passing and other unparalleled operations.

## REGISTRATION RESULT

### POINT CLOUDS FROM SAME DATA SET

The first registration result we get done based on two point clouds which are from the same data set. We moved and rotated one of the point cloud from its original place and tried to register it back to the other one.

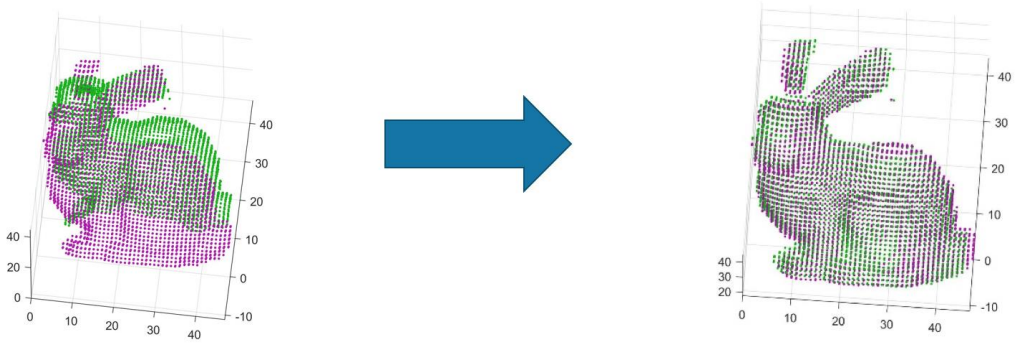


Figure 11 Before registration VS after registration

From figure above we can see that before registration, there are two point clouds (green and red). After run point cloud registration algorithm, we have these two point clouds matched with each other.

### POINT CLOUDS FROM DIFFERENT DATA SET

The second result is done with two point clouds which has different data source. One point cloud contains points scanned at the front of the object. The other one contains points scanned at the top of the object.


Figure 12 Real object and registration result

From the figures above we can see that green dots are from one point cloud which scanned from the top, red dots are from the other point cloud which scanned from the front.

## SURPRISES AND LESSONS LEARNED

At first, we thought traditional k-d tree version would be much better than the brute force search. However, after we run both, they have similar performances. This phenomenon really surprised us and made us upset. Then we dug into these two implementations. We found that k-d tree runs a little bit slower in small scale problem. But when the number of points increases, it has much better performance. Maybe O(nlogn) algorithm is slower than $O(n^2)$ one when n is small, but when n becomes larger O(nlogn) algorithm will be the best choice.
Secondly, we have found that finding a good solution or implementation is a long process. We must try a lot of methods and analyze its benefits and drawbacks so that we can move forward to a better solution.

## REFERENCE

Langis, C., Greenspan, M., & Godin, G. (2001). The parallel iterative closest point algorithm. In 3-D Digital Imaging and Modeling, 2001. Proceedings. Third International Conference on (pp. 195-202). IEEE.
Besl, P. J., & McKay, N. D. (1992, April). Method for registration of 3-D shapes. In Robotics-DL tentative (pp. 586-606). International Society for Optics and Photonics.
Kubelka V, Oswald L, Pomerleau F, et al. Robust data fusion of multimodal sensory information for mobile robots[J]. Journal of Field Robotics, 2015, 32(4): 447-473.