Refactoring, Modernisation & Feature Addition with Emphasis on GPU Module

Point Cloud Library | Haritha Jayasinghe

Abstract

Despite being the go-to library for point cloud based computation, PCL can cause some friction to users due to its old-fashioned and sometimes inconsistent API and the lack of certain features. This proposal aims to introduce the following new features to the PCL library;

- GPU implementation of Iterative Closest Point (ICP) algorithm
- Implementation of Fast Resampling of 3D Point Clouds via Graphs

As well as to refactor and modernize the library by means of;

- Introducing better type for point indices, thereby providing support for larger point clouds
- Introducing a fluent API for algorithms
- Modernising the GPU Octree module to align with the it's CPU counterpart

Applicant information:

Name	: J.M. Haritha Anushanga Jayasinghe
Email	: haritha.16@cse.mrt.ac.lk
University	: University of Moratuwa, Sri Lanka
Major	: Computer Science & Engineering
Graduation date	: December 2020 (Expected)
GitHub profile	: https://github.com/haritha-j
LinkedIn profile	: https://www.linkedin.com/in/haritha-jayasinghe/
Address	: 519/2, Bauddhaloka Mawatha, Colombo 8, Sri Lanka
Contact	: +94 7104 15528

Deliverables

- GPU-based Iterative Closest Point (ICP) algorithm implementation
- Graph based sampling filter implementation
- Migration to new point index type
- Fluent API support for existing algorithms
- Improved GPU Octree module

Table of Contents

Abstract 1

Applicant information: 1

Deliverables 2

Problem description 3

- Type for point Indices 3
- Implementation of Fast Resampling of 3D Point Clouds via Graphs 3
- GPU Implementation of ICP algorithm 4
- GPU Octree enhancement 4
- Fluent API support for existing algorithms 5

Implementation Considerations 6

- GPU Implementation of ICP algorithm 6
- Implementation of Fast Resampling of 3D Point Clouds via Graphs 7
- GPU Octree enhancement 8
- Fluent API support for existing algorithms 9
- Type for point Indices 10

Proposed Timeline 11

About me 12

- Experience with PCL 13
- Commitment 14

References 14

Problem description

Type for point Indices

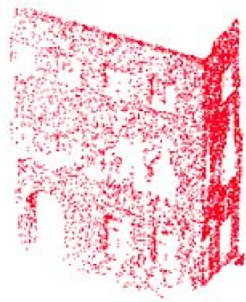
As laser scans and LIDAR becomes more popular, the need arises for handling clouds with a very large number of points. However many algorithms within the Point Cloud Library are incapable of handling point clouds containing over 2 billion points due to their indices being of the type int, which is capped at 2 billion[1]. Furthermore there currently isn't one standard type being used for indices, instead a variety of types such as int, long, unsigned_int and others are being used. Therefore there is a pressing need to switch to a standard type for indices.

Aside from this, there may be certain other areas in PCL which are also incapable of handling larger point clouds due to various reasons, and such inconsistencies also need to be addressed. This is described in more detail in the implementation section.

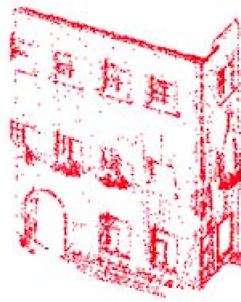
However due to the increased memory usage of types with larger capacity, the memory efficiency of the library may be significantly reduced, not to mention further complications with caching etc., which can be a serious concern considering the large variety of platforms that PCL is used on. Thus the ideal solution would be to allow the user to choose which point type to utilize at compile time, based on his intended use case and platform.

Implementation of Fast Resampling of 3D Point Clouds via Graphs

Point cloud resampling is another primary use case for users of PCL, and as such PCL currently offers a number of sampling algorithms such as randomSampling and voxel grid, and in fact one of my current pull requests is an implementation of a new farthest point sampling filter. However each of these filters have distinct advantages as well as disadvantages, from the nature of the output to time complexity, and as such it is crucial to select the correct algorithm for a particular task. The proposed filter is based on the paper Fast Resampling of 3D Point Clouds via Graphs[2] and proposes a method for resampling a point cloud based on graphs, which not only reduces the computation time, but more crucially, performs sampling in such a way that highlights the contours of the point cloud.



(a) Uniform resampling.



(b) Contour-enhanced resampling.

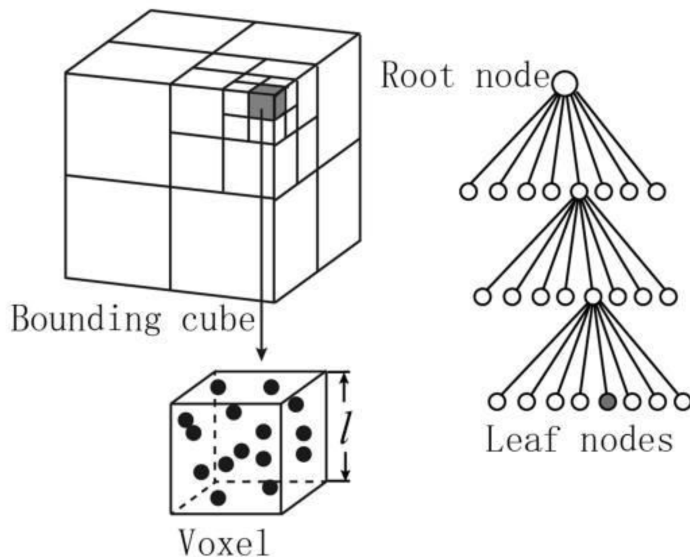
This makes it ideal for use cases such as contour detection, visualization, registration, and shape modeling[2]. The following figure from the above paper demonstrates the comparison between uniform resampling and the contour enhanced resampling provided by the proposed algorithm.

GPU Implementation of ICP algorithm

Iterative Closest Point (ICP) is the most commonly used algorithm for point cloud registration[3], which in turn is one of the primary features that are needed for when performing computations on point clouds. Unfortunately the ICP algorithm is by design an expensive operation with $O(n^2)$ time complexity with respect to the number of points in the cloud[3]. However the nature of the ICP calculation makes it ideal for parallelization as distance computations are performed for each and every point, with respect to the other points in the cloud. The current CPU implementation is accelerated by the use of kd-trees (where nearestKSearch() is utilized in the correspondence estimation step) to compute distance between points, however, the existing GPU Octree module could be used instead in a GPU implementation to provide an extremely fast implementation of the ICP algorithm.

GPU Octree enhancement

Octrees are specialized data structures with nodes that split into eight subchildren, and are a widely used structure when working with point clouds, since they can be used to efficiently perform many operations. The PCL GPU module is a crucial yet somewhat overlooked component of PCL. The following figure shows a typical octree.



In many cases the GPU algorithms can execute tasks magnitudes faster than it's CPU counterpart, which can be crucial when working with large point clouds. Unfortunately the GPU API is quite limited and often lacks much of the functionality offered in the equivalent CPU algorithms. I aim to modify some of the more vital components of the GPU module to bring it up to speed with the rest of the library, as well as to fix / improve some of the functionality.

In particular there are certain features missing in the GPU based Octree implementation such as the ability to calculate distances to points found using radiusSearch() or nearestKSearch() functions, which are some of the primary use cases of the GPU module. Furthermore RadiusSearch() provides incorrect results when search is performed over multiple radii.

In addition the current static height implemented in the octree module also limits the potential uses of the module. The use of void* pointers in "pointer to implementation" as opposed to forward declaration and other inconsistencies are currently present in the module. Therefore the Octree module is, on the whole, is in need of modification in multiple areas.

Fluent API support for existing algorithms

Switching to a fluent style API would not only make code cleaner and simpler, but allows integration with ranges, one of the major features introduced in C++20[4], which could be quite useful in future upgrades, allowing for more efficient operations on many use cases (due to lazy execution). For instance, using ranges, different functions can be easily combined using view. Consider the following example.[5]

```
std::vector vec{1, 2, 3, 4, 5, 6};
auto v = vec | std::views::reverse | std::views::drop(2);
```

Here, multiple operations are combined together, which are only executed once an element in the result is accessed. In order to move to this method of execution, it must be possible to chain PCL functions together, which is exactly what a fluent API achieves.

While internal class structures would need to be modified, fluent API support can be added without breaking the current user API.

Implementation Considerations

GPU Implementation of ICP algorithm

The idea behind the Iterative Closest point algorithm is to calculate a transformation that best aligns one point cloud with another, by minimising the distance between the two clouds, and most commonly the sum of square differences between the coordinates of matched pairs is used as the distance metric.[6] The purpose of this task is to add a new ICP class to the PCL GPU module, where the distance computations are performed parallely.

The simplest implementation of such an algorithm would seek to parallelize a traditional distance computation[7]. However this would involve comparing each point in the target cloud with each point in the source cloud, resulting in an $O(n^2)$ time complexity[3]. Fortunately the GPU octree module in PCL which provides a nearestKSearchBatch() function for k = 1[8], which can be utilized to identify the nearest point parallely to compute the distance metric. Unfortunately this method currently does not return the distance to the point. Either an additional computation must be performed to calculate distance (which can be parallelized) or the octree module itself must be modified to return distances to the points as well as point indices. This is further discussed under the “GPU Octree enhancement” heading below.

One drawback with this method is the GPU memory usage of uploading both the octree of the target point cloud as well as the points of the source point clouds to the VRAM. Fortunately it is possible to upload the points of the source cloud in batches which significantly reduces the memory overhead. This allows comparison of full point clouds with limited VRAM constraints, without the drawbacks of downsampling (Typically downsampling reduces the likelihood of finding ‘the best match’ for a given point). However benchmarking is required to ascertain if the increase in accuracy can justify the time penalty of repeatedly loading the entire cloud. It is crucial to build the octree from the target cloud since the source cloud will be transformed in each iteration, and we do not wish to rebuild the octree in each iteration.

Once a transformation is identified, the application of the transformation may also be accelerated by utilizing CUDA programming, since it is essentially an application of matrix multiplication over the point set.

Additional performance enhancements may be gained by performing octree based subsampling to reduce point densities, implementing a caching policy etc.[6], however the practical viability of such methods must be determined by experimentation. Furthermore additional enhancements such as the removal of outliers may also be required to prevent issues due to local minima in the distance metric[6]. A new test class similar to that of the CPU ICP implementation must also be developed, where a point cloud is manually created, which allows checking the locations of the points of the resultant cloud using an assertion such as EXPECT_NEAR.

Implementation of Fast Resampling of 3D Point Clouds via Graphs

The proposed resampling method would be implemented as a new class in the fiters module, which would most probably be built on top of the proposed samplingFilter base class. A graph must be constructed from the point cloud, by encoding local geometry information of the cloud using an adjacency matrix[9]. In simple terms the edgeweight between two points represents a distance similar to euclidean distance between the two points, given that the distance between them is below a certain threshold[2]. Due to the large memory usage of this approach, sparse matrices must be utilized for point representation. (Fortunately, due to the fact that an edge is recorded only if the distance between the two points is below a certain threshold, the space complexity becomes On(n) as opposed to $O(n^2)$, and thus, sparse matrices would be ideal for this representation). In terms of computational complexity, octree based methods such as radiusSearch() can be utilized to identify distances to all points below the threshold in O(n) time as well.

Next a graph filter must be constructed to extract a sample of points. A graph filter is essentially an algorithm that applies a function to an input graph to produce an output graph. Similar to image processing, where a high pass filter can be utilized to extract edges and contours in a 2D image, a similar high pass filter can be applied to a graph to identify features such as contours in a 3D point cloud[2]. Such a filter can be utilized to identify points that ‘break the trend’ of the surrounding points, which indicates a contour. Application of the actual filter is simply a matrix multiplication operation on the adjacency matrix described above, similar to how a high pass filter is applied to a 2D image’s pixels.

An additional test class must also be developed for testing the functionality of the new downsampling algorithm. As with other filters such as the farthestSamplingFilter that I’m currently implementing, a custom point cloud can be created with point clusters that represent contours in the cloud. Then, it can be asserted whether the results of the filter retain a significant amount of points (depending on the sampling size) that belong to the contours.

GPU Octree enhancement

As mentioned above there are a few shortcomings in the current GPU Octree module.

- **Lack of the ability to identify the distance to the points detected using either the radiusSearch(), nearestKSearchBatch() or approxNearestSearch() methods[8]**

Upon initial inspection, it appears that this might be an implementation decision taken due to the way an octree is currently represented in GPU memory, using morton codes[10][11], which may include an additional calculation for distance computation. However, given that many use cases rely on this distance (for instance in the ICP algorithm mentioned above, or when performing change detection across two point clouds). In one of my personal workloads, I've been forced to perform radiusSearch() with multiple thresholds in order to get a rough estimate of point to point distance. Thus, I believe it would be beneficial to have the ability to measure point distances, and in fact the CPU implementation already offers this functionality.

The distance computation should be somewhat straightforward, since the morton code for a given point encodes its position relative to the centroid of the voxel[10], which allows us to calculate it's distance with the query point by simple addition.

- **radiusSearch() on multiple radii thresholds is broken and provides extremely inconsistent results**

This issue is documented in #3583 [12]. This is due to a flaw in the function used to broadcast the square of the radius in the CUDA radiusSearch() implementation, which results in the addition of points to the results at random. This function needs to be modified to correctly calculate distance based on the correct square value of the radius threshold.

- **GPU octree module does not allow for selection of voxel size**

However, allowing the user to select voxel size allows the user to select the optimum octree size to suit each use case. For instance an octree with smaller nodes would be better optimized to perform kNearestSearch() for a densely packed point cloud, while larger voxel sizes may be more suited for less dense clouds [13]. Therefore I also intend to investigate the possibility of allowing the user to select the number of octree levels or the voxel size of the leaves, which would probably involve modifying morton source code, where currently the number of levels is set to 10. The existing tests must be modified to test the additional functionality.

Fluent API support for existing algorithms

The implementation of a fluent API is in itself a rather simple task, and it can be achieved by modifying the return values of methods within classes. In most cases it might also be necessary to introduce an additional terminating method. The first step would be to conduct a review of the existing classes to identify the classes where having a fluent API would make sense to a user.

For instance, the use of classes in the filter module could be simplified using a fluent interface, where each of the filter specific parameters as well as the input and output destinations that must be set prior to filtering could be chained together. Finally, a terminating method may also be added[14].

One major concern is the fact that subclasses would be required to override all fluent API based methods from their superclass[15][16]. Consider the following simplified example, where parent class methods are overridden.

```
class FilterIndices {
    public FilterIndices setNegative() { ... }
}
class CropBox : public FilterIndices{
    public CropBox setNegative() { FilterIndices::setNegative(); return this;
} // The setNegative() function needs to be overridden to return a CropBox
object.
    public CropBox setRotation(const Eigen::Vector3f &rotation) { ... }
}
CropBox crop = new CropBox().setNegative().setRotation(rotation); // this
only works because setNegative() was overridden.
```

A different approach would be to simply implement a fluent wrapper around each class[15]. This approach has the advantage of not having to modify existing classes and being able to provide ‘fluent-style’ names for fluent methods, but would result in boilerplate code. As an example consider the following snippet from a fluent wrapper for the CropBox class, which provides a fluent API for setting rotation.

```
class FluentCropBox : private CropBox {
 public:
     CropBox() : CropBox() {}
     FleuntCropBox &withRotation() {
         setRotation (const Eigen::Vector3f &rotation);
         return *this;
     } ....
```

Since there is a large number of classes to be modified across a multitude of modules, it would make sense to roll out the fluent interface module-wise. Along with each module, the relevant test methods should also be modified to ensure correct functionality of the new API.

Type for point Indices

The first consideration would be to decide which options would be offered to the user during compilation, to utilize as the type for indices, as well as to identify the default approach to take. The simplest method would be to provide a binary choice between large and small indices in cMake, with a 'PCL_LARGE_INDICES' option. By default, it makes sense to use the smaller indices, as clouds with points above 2 billion are currently more of a niche use case rather than the norm. Additional options for smaller indices can also be offered, which may perhaps be required for smooth operation in resource constrained real time environments. Whichever option selected by the user would be assigned as 'pcl::index_t'.

An additional option may be offered to switch to the use of unsigned indices, which would double the capacity of each type, and allow the current 32bit indices to support 4 billion points[1].

Once the above factors are decided, the current 'std::vector<int>' objects must be replaced with 'std::vector<pcl::index_t>' and current uses of size_t must be switched to the new 'index_t'. There is widespread use of <int> vectors as indices across most modules of PCL, which must all be updated. All occurrences of the use of other types such as 'int' would need to be updated across the public API, since many functions currently require indices in the form of 'int' which would unfortunately cause significant breakage in the user API.

Finally it would be wise to inspect popular existing libraries which may have other inherent limitations that may break intended functionality when using extremely large point clouds. For instance the random sampling module currently uses the older 'rand' random number generator (This is an issue which is currently being addressed). Apart from the other well documented drawbacks of the rand module, the seed used in the generator is limited by the limits of RAND_MAX[17], which is capped at around 2 billion for most systems, and therefore, it would no longer be random for clouds with over 2 billion points. There may be other modules with such restrictions which would need to be modified to allow proper support for larger indices.

Proposed Timeline

Week	Duration		Task
1	June 1	Junee 7	● Background research and class design for fast resampling algorithm implementation ● Research on GPU Octree and morton code representations
2	June 8	June 14	● Implementation of distance measurements for octree based search methods
3	June 15	June 21	● Creation of a graph representation of points for resampling ● Providing support for user defined octree levels
4	June 22	June 28	● Providing a high pass filter for point resampling ● Apply fixes to the GPU radiusSearch() method
5	June 29	July 5	● Evaluation of classes that are suitable for a fluent style API ● Evaluation 01
6	July 6	July 12	● Implementation of tests for the high pass filter ● Class design for GPU ICP registration class
7	July 13	July 19	● Application of fluent style API for identified classes
8	July 20	July 26	● Distance metric computations for ICP registration ● CMake modifications for supporting large point indices
9	July 27	August 2	● Implementing tests for fluent style API ● Evaluation 02
10	August 3	August 9	● Implement changes to indices types across the library ● Evaluate additional changes required for

			supporting large indices and make necessary changes
11	August 10	August 16	● Implement iterative computation of distance metric and application of required transformations
12	August 17	August 23	● Provide additional enhancements to ICP algorithm for increased performance and for avoiding local minima. ● Implement tests for GPU ICP algorithm
13	August 24	August 30	● Finalize documentation and tests
	August 31		Conclusion

About me

I'm a 4th year undergraduate studying Computer Science & Engineering at the University of Moratuwa, Sri Lanka. I'm passionate about making an impact through creative problem solving, and I took up programming since it's a great tool in achieving that goal. My spare time is mostly spent on reading, cycling and some gaming.

I have experience working with c++, python, java, javascript and golang languages in a variety of areas from deep learning to video decoding. Furthermore I'm currently studying concurrent programming at my university, including CUDA programming.

Last year I completed a 6 month internship as a student researcher at the University of Sydney, where I worked on two primary projects. One of these was regarding differentiating similar items using pointclouds to identify privacy leakage in mixed reality devices, where I worked extensively with point clouds. This project involved both point cloud manipulation using PCL as well as using custom neural network architectures based on pointNet.

Experience with PCL

I have been working extensively with PCL and point clouds for around 2 years on a few projects. Aside from the research project mentioned above, these also include;

- Identification of components of Railway tracks from aerial laser scans
- Comparison of scans of buildings and design files to identify imperfections that arise during construction

While working on the above projects, I've become quite familiar with PCL, and in particular modules such as octree/kdtree, registration, filters, visualization, sample consensus and GPU.

Through my experience with the above projects I have gained a deep appreciation for all the functionality provided by PCL, as well as become familiar with some of its drawbacks and caveats. Therefore I was delighted to learn that PCL was participating in GSoC this year, and was very eager to participate and contribute to the library. In fact, many of the enhancements that this project aims to build were based on pain points I experienced during the above projects.

In terms of contributions, I have linked a couple of pull requests I have submitted to PCL below.

- Add farthest point sampling filter
  https://github.com/PointCloudLibrary/pcl/pull/3723
  This PR implements a new filter to perform farthest point sampling in $O(n^2)$ time, utilizing euclidean distance.

- Add point size check to savePCDFileBinary method
  https://github.com/PointCloudLibrary/pcl/pull/3775
  This PR provides a fix to a bug in the file writer where the output file can become corrupt due to the height*width value being different to the number of points in the point cloud.

I have also linked below an example contribution for another open source project - Centrifuge, which is a blockchain based platform written in golang that powers decentralized financial transactions. My contribution was focused on implementing test cases to assess functionality for edge cases.
https://github.com/centrifuge/go-centrifuge/commit/a6d396c1399690d718af83e251f0b290b41c25a7

Commitment

As per my current schedule, I would be having my vacation during the GSoC time period except for about 2 weeks (from the beginning of June to the middle of August). During this time I can commit 6+ hours of work per day. Unfortunately the current schedule may quite possibly change because of disruptions to the academic schedule due to the Covid-19 pandemic. We're currently carrying on work online, and as of now, it seems like the entire schedule would be delayed by two weeks.

References

1. Summary of C/C++ integer rules, Nayuki - https://www.nayuki.io/page/summary-of-c-cpp-integer-rules
2. Fast Resampling of 3D Point Clouds via Graphs, Siheng Cheng et al. - https://arxiv.org/pdf/1702.06397.pdf
3. Notes on Iterative Closest Point Algorithm, Jana Procházková and Dalibor Martišek - https://www.researchgate.net/publication/324500004_Notes_on_Iterative_Closest_Point_Algorithm
4. Standard Ranges, Eric Niebler - http://ericniebler.com/2018/12/05/standard-ranges/
5. A beginner's guide to C++ Ranges and Views, Hannes Hauswedell - https://hannes.hauswedell.net/post/2019/11/30/range_intro/
6. Iterative Closest Point (ICP) Algorithm, Yaraslav Halchenko - http://www.onerussian.com/classes/cis780/icp-slides.pdf
7. GPU-Accelerated Nearest Neighbor Search for 3D Registration, Deyuan Qiu et al. - https://www.researchgate.net/publication/221410064_GPU-Accelerated_Nearest_Neighbor_Search_for_3D_Registration
8. GPU octree documentaiton - http://docs.pointclouds.org/trunk/classpcl_1_1gpu_1_1_octree.html
9. point cloud processing using linear algebra and graph theory, Tim Volodine - http://www.cs.kuleuven.be/publicaties/doctoraten/tw/TW2007_05.pdf

10. Thinking Parallel, Part III: Tree Construction on the GPU, Tero Karras - https://devblogs.nvidia.com/thinking-parallel-part-iii-tree-construction-gpu/

11. Advanced Octrees 2: node representations, David Geier - https://geidav.wordpress.com/2014/08/18/advanced-octrees-2-node-representations/

12. Inaccurate results with gpu octree search example - https://github.com/PointCloudLibrary/pcl/issues/3583

13. Efficient Processing of Large 3D Point Clouds, Jan Elseberg, Dorit Borrmann, Andreas Nuchter - https://robotik.informatik.uni-wuerzburg.de/telematics/download/icat2011_1.pdf

14. Method Chaining, Fluent Interfaces, and the Finishing Problem, Dave Glick - https://daveaglick.com/posts/method-chaining-fluent-interfaces-and-the-finishing-problem

15. Fluent Interface - https://en.wikipedia.org/wiki/Fluent_interface

16. Fluent Interfaces are Evil, Marco Pivetta - https://ocramius.github.io/blog/fluent-interfaces-are-evil/

17. USING THE C OR C++ rand() FUNCTION, David Deley - http://daviddeley.com/random/crandom.html