Comparison of SLAM methods with CUDA implementation

Course: Research project - GPU algorithms

Coordinator: Kaczmarski Krzysztof
Authors: Rogala Michał, Stasiak Szymon, Śliwakowski Mateusz
Description: This document covers three different SLAM methods with an emphasis on CUDA implementation
Code repository: https://github.com/Sliwson/cuda-slam

Input files: Point clouds as .obj files

September 13, 2020

Contents

1 Report goals

The report is devoted to a research of SLAM methods using CUDA technology. We want to give a brief introduction what SLAM is and what are its applications. After reading the document you should know three different methods of approaching the problem, their advantages and disadvantages. The cornerstone of the work is answering the question - how well these methods can be converted to GPU architecture and how much will we gain?

2 Problem statement

2.1 Theoretical background

Simultaneous localization and mapping (in short SLAM) is the computational problem of constructing or updating a map of an unknown environment while simultaneously keeping track of an agent's location within it [6]. In theory the problem can be simplified to more mathematical definition - given two point sets $P_1 = \{x_1, \ldots, x_n\}$ and $P_2 = \{y_1, \ldots, y_n\}$ find translation t and rotation R that minimizes the mean squared error:

$$MSE(R,t) = \frac{1}{N} \sum_{i=1} N(x_i - Ry_i - t)^2$$

The reality, unfortunately, is much more complicated. Sizes of point sets might differ, clouds are unordered and noise points can occur.

2.2 Applications

The main field of SLAM application is robotics. There are a lot of machines that utilizes this algorithm to keep track of the space in which they are moving. The most popular are autonomous vacuum cleaners such as presented below.

Figure 1: Autonomous vacuum cleaner

Another popular application is mapping real life environments to digital equivalents. The programs produce *Geospatial mappings* which can be later processed and analysed. Variety of different devices

is used such as hand scanners, drone scanners or even satellites. Due to rapid technological progress nowadays, even smartphone cameras can be used to perform simple environmental scan.

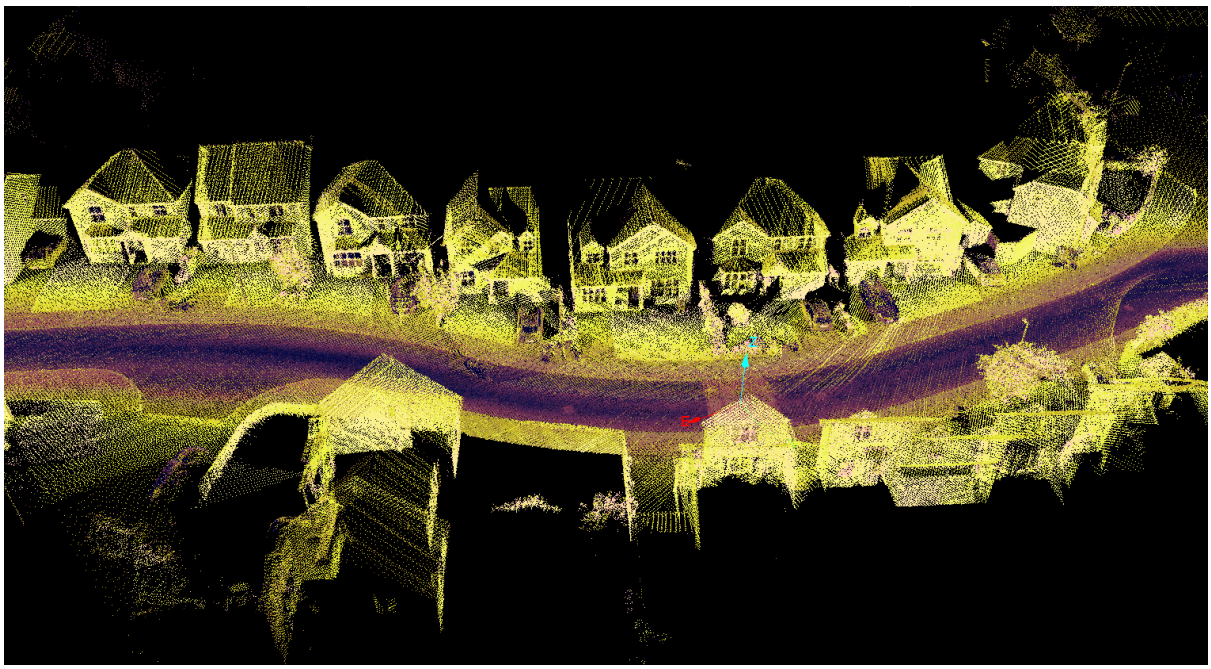


Figure 2: 3D laser scan

3 Computational method

3.1 Iterative Closest Point

The most widely described and most popular method for solving SLAM problem is Iterative Closest Point algorithm. The main idea of the algorithm is stated below [1]:

1. Determine corresponding points.
2. Calculate translation and rotation using Singular Value Decomposition (in short SVD).
3. Apply calculated rotation and translation to the point cloud.
4. Compute the mean squared error.
5. If error decreased and it is greater than given threshold repeat all the steps.

Various modifications have been proposed to this algorithm (such as sampling point sets, weighting the correspondences or rejecting outliers). However, in this project we focused on the base version of the method as it is good enough for comparison with the other algorithms.

3.2 Non-iterative Closest Point

Compelling alternative to an iterative point cloud matching algorithms is a non-iterative SVD-based solution. The idea has been described by Shinji Oomori, Takeshi Nishida, and Shuichi Kurogi of Kyushu Institute of Technology. They proclaim that this method is capable of matching clouds even "with less than 4% of the computational time of the ICP (iterative closest point) algorithm with nearly identical accuracy"[4].
The idea of this algorithm can be described in the following points:

1. Take both point sets (containing respectively N_1 and N_2 points) and find mass centres for them (C_1 and C_2).
2. Align the clouds to their mass centres and store the result points in matrices: A_1 (size $N_1 \times 3$) and A_2 (size $N_2 \times 3$) respectively.

3. Find matrices U from SVD of both matrices, respectively U_1 and U_2.
4. Rotation matrix is then defined as $R = U_1 \times U_2^T$.
5. Find the translation vector as $t = C_2 - (R \times C_1)$.

Unfortunately, for permuted clouds of different sizes this method does not always converge. That is why one more step is necessary for those conditions:

6. To increase convergence ratio, repeat those steps for multiple scans of observed object and select the transformation minimizing the mean squared error.

This step is indeed a huge improvement but it can be used with promising results only when the item is observed from multiple angles.
Implementation described in this paper covers different approach - the copies of source and target clouds are created, each copy with differently permuted points. Transformations received by using steps from 1. to 5. can differ for various permutations, so the result is chosen among them based on the minimal value of the mean squared error, similarly to what is proposed in the original idea.

3.3 Coherent Point Drift

In this method, alignment of two point sets is considered as a probability density estimation problem. One point set represents the Gaussian mixture model (GMM) centroids. The second one represents the data points. At the optimum, two point sets become aligned. Method uses Expectation Maximization (EM) algorithm to find scale, rotation matrix and translation vector.
Main idea of a coherent point drift algorithm can be presented in several steps:

E-step:

1. Compute probabilities matrix P. P describes probability of correspondence of points from both sets.

M-step:

2. Using P and point sets calculate matrix A.
3. Calculate rotation matrix using Singular Value Decomposition of matrix A in a similar way as in the ICP algorithm.
4. Calculate scale and translation vector using point sets and SVD result.

Repeat E-step and M-step until convergence.

In order not to impair readability we decided to skip mathematical formulas in this schema. Fully detailed description and more information about Coherent Point Drift can be found in article written by Andriy Myronenko and Xubo Song [3].
The base version of the algorithm has multiple vulnerabilities, especially when it comes to computation time. For this reason many improvements are known and some of them have been implemented in our solution. Firstly the size of P matrix is $N \times M$, where N is size of first point set and M is size of second point set. It requires a lot of memory to keep matrix of this size, so instead we can calculate vectors: $P1$, P^T1 and matrix PX, where 1 is vector of all ones and X is first point set. Using these vectors and matrix we can achieve the same result even faster because in calculations we had to multiply P matrix by the vector of ones and we have already done it.
Secondly the main bottleneck of the algorithm is the calculation of matrix-vector products $P1$, P^T1 and PX. This is done with $O(MN)$ complexity. The Fast Gauss Transform, introduced by Greengard and Strain [2], reduces complexity to only $O(M + N)$.

4 Program architecture

4.1 Project structure

The solution consists of three projects:

- *common* - static library project, contains all elements that are shared between CPU and GPU implementations.
- *cpu-slam* - executable project, contains implementation of slam methods for CPU and CPU entry point.
- *gpu-slam* - executable project, contains implementation of slam methods for GPU and GPU entry point.

4.2 Modules

The project is constructed using several modules:

- *Renderer* - implemented in *common* project. Capable of rendering point clouds in different colors. Can be used multiple times in one executable launch. Widely using for results visualisation.
- *CloudLoader* - implemented in *common* project. Module responsible for loading .obj point clouds.
- *Configuration* - implemented in *common* project. Module representing configuration, tightly coupled with *ConfigParser* which allows loading configuration from .json files.
- *TestRunner* - implemented in *common* project. Module used to define and run benchmarks.
- *Algorithms* - implemented in *cuda-slam* as well as in *cpu-slam*. Each algorithm is implemented in a separate file as a global function in appropriate namespace. Some operations that are common for CPU as well as GPU are implemented in *common* project.

4.3 Dependencies

- *Assimp* - for cloud loading.
- *Eigen* - for CPU matrix operations.
- *Glad* - wrapper for OpenGL.
- *GLFW* - for window system.
- *Nlohmann/json* - for configuration parsing.
- *cuBLAS*, *cuSOLVER* - for GPU matrix operations.

5 Implementation details

5.1 Common elements

All of the methods implemented in the scope of this project base to a greater or lesser extent on matrix operations, which have significant impact on computation times and quality of the results. The most important operation is a singular value decomposition (SVD). Apart from that, also large number of less complicated operations, for instance multiplication or addition, are needed. Unfortunately, there is no solid library offering all those functionalities both for CPU and GPU accelerated computations at the same time. For this reason we decided to choose three libraries allowing to receive satisfying results no matter which method is used:

- *Eigen* - widely recommended, fast and open source library for CPU matrix operations.
- *cuBLAS* and *cuSOLVER* - libraries created and natively supported by Nvidia for GPU accelerated matrix operations using CUDA technology.
- *glm* - OpenGL Mathematics library used mainly for storing common parts like point clouds, vectors, transformation matrices and basic operations on them. This library is available also for both host and device managed code what allows to reuse some parts of it for CPU and GPU computations and at the same time simplifies visualising the results.

It is worth clarifying that Eigen implements the partial support for CUDA kernels but this feature is still experimental and there are multiple known issues (for example with support for gcc or MS Visual Studio). Adding that to potential problems with performance, we decided not to use this library for GPU accelerated operations in order to maintain higher scalability and stability of the project.

5.2 Iterative Closest Point

Because of the fact, that for our research we have chosen the simplest method of finding corresponding points across the point sets, the algorithm benefits greatly from using GPU accelerated computations. Each GPU thread is associated with one point from the first cloud and finds the nearest one from the second cloud. It results in almost 100% GPU occupancy and major improvement over CPU computations. Furthermore we used cuSOLVER's singular value decomposition although it does not offer as big performance leap as correspondences step.

5.3 Non-iterative Closest Point

The non-iterative algorithm does not offer too many ways to benefit from parallel CUDA operations. For that reason the most natural path for parallelising the code is to run SVDs for multiple permutations of source and target clouds on parallel. Unfortunately, natively supported solution from cuSOLVER library - so called "batched SVD" is not an option due to limitations for decomposed matrices size (it cannot be greater than 32 x 32 which corresponds point clouds of 32 points). For that reason another solution has been developed - matrices are decomposed by separate threads in batches of configured size. Implementing this solution also brought multiple issues and attempts to obtain expected results can be broken down into several stages:

1. Running SVD from separate CPU threads - although the SVD methods from cuSOLVER are marked "thread-safe", running them in parallel as CPU threads leaded to memory access violation and program randomly crashed every few times.

2. Running SVD from one CPU thread with separate CUDA streams used for each decomposition - by using this method problems with memory access have been solved, but profiling the application showed that streams run sequentially which is not the result expected.

3. Using separate CPU threads for running decompositions using separate CUDA streams - this method combined the advantages of both approaches, giving truly parallel execution of device managed code, at the same time providing higher parallelization level due to limiting the number of concurrent operations by number of GPU streams instead of CPU threads.

Apart from parallel and sequential variants of Non-iterative Closest Point algorithm, also three different approximation types have been implemented for this method:

1. **Full** - to avoid finding corresponding points in each algorithm run, this approximation type bases the error calculation on distance between two points with the same indices in both clouds. It offers promising results with fast calculations for insufficiently permuted clouds but otherwise the error approximation might differ from actual one and for this reason returned result might not be the best.

2. **None** - after each repetition, error is calculated by finding correspondences between subcloud of given size taken from the first cloud and the entire second cloud. This method offers higher reliability but instead increases the execution time of one iteration noticeably, especially using CPU.

3. **Hybrid** - error for each run is calculated the same way as when using full approximation but five best results are stored and the error is recalculated for those in the same way as when using none as approximation type (with finding correspondences). Then, the one with the smallest error is returned as an actual result.

5.4 Coherent Point Drift

CPU solution of a Coherent Point Drift is based on algorithm described in [3]. It also uses Matlab code provided by authors as an example of how some technical details were handled. FGT algorithm implementation is based on Matlab sample available in [5]. Parallelised solution using GPU acceleration mainly focuses on matrices multiplication using cuBLAS and utilizing thrust. FGT algorithm was not implemented on GPU due to complexity of this solution and lack of reliable external sources.

Three different approximation types have been implemented for this method:

1. **None** - Probabilities are calculated without FGT.

2. **Full** - Uses only FGT to calculate probabilities.
3. **Hybrid** - FGT is used only when error is above certain level.

6 Input data description

As we wanted to keep the input simple we decided to use .obj files as the point clouds. It is easy to find interesting models in this format which allowed us to test the algorithms on multiple distinctive samples. Furthermore, one can edit .obj files using tools (for example *Blender*) to add noise or even cut off a part of the cloud. We found that method most convenient to simulate real life conditions as 3D scanners data has to be processed using methods that are beyond the scope of our work.

Figure 3: Sample .obj file

7 Execution, configuration and user guide

7.1 Execution

Currently the only supported operating system is Windows and the Visual Studio environment as a development tool. All the requested libraries are included in the solution folder and the application should be able to run out of the box on supported environments with Cuda Toolkit 10.2 and common C++ libraries. Necessary configuration files and schemas for config files are attached as well. Each executable (for both CPU and GPU implementations) is supposed to handle 0 or 1 command line parameters with following results:

- No parameters - load config from default.json.
- One parameter (config file name) - load config from the given file.

```
C:\Projekty\Studia\6 semestr\GPU\cuda-slam\bin\cpu\Release>cpu-slam.exe not-default.json
Loading config from: not-default.json
================================
```

Figure 4: Execution example with given config file

Support for Linux systems and other development tools are some of the areas of improvement for the future. Porting, however, is not supposed to be an issue. But due to multiple dependencies between used libraries we decided not to focus on it during development phase as it is not the point of this research.

7.2 Configuration

Possible fields and values for config files are defined by *JSON Schema* and they are described in the table below.

Field name	Type	Default Value	Description
before-path*	String	–	Path to the first point set (also called "cloud before") representing object before transformation.
after-path*	String	–	Path to the second point set (also called "cloud after") representing object after transformation. This cloud is to be modified by applying scale, transformation and rotation values mentioned below.
method*	Enum	–	Selected algorithm. Possible values are: • icp - Iterative Closest Point • nicp - Non-iterative Closest Point • cpd - Coherent Point Drift
policy	Enum	parallel	Use sequential or parallel computations on CPU. Possible values are: • sequential • parallel Used only for ICP and NICP.
scale	Number	1.0	Transformation scale. Scales the second point set by given value.
translation	Array: Length 3	–	Translation vector. Translates the second point set by given value.
rotation	Array: Length 9	–	Rotation matrix in row-major order. Represents the rotation which is to be applied to the second point set.
translation-range	Number	–	Translation distance to be applied to the second point set. Translation direction is chosen randomly.
angle-range	Number	–	Translation angle in radians to be applied to the second point set. Rotation axis is chosen randomly.
max-iterations	Integer	–	Maximum number of iterations for iterative algorithms - ICP and CPD. After reaching this value, algorithm stops returning the best match found so far.
cloud-before-resize	Integer	–	Desired number of points in the first point set. If value is given, the set will be resized to this size omitting randomly selected points. Otherwise, whole cloud will be used.

cloud-after-resize	Integer	–	Desired number of points in the second point set. If value is given, the set will be resized to this size omitting randomly selected points. Otherwise, whole cloud will be used.
cloud-spread	Number	–	The maximum distance in a single dimension between the extreme points in second point set. Based on this value cloud can be stretched or shrunk.
random-seed	Integer	–	A seed for random number generator used in the program. If value is not given, the seed is random as well.
show-visualisation	Boolean	False	Specifies if graphic visualisation of clouds should be displayed.
max-distance-squared	Number	1000.0	Maximum distance to search for closest point in ICP algorithm implemented on CPU. If the squared distance between two points is larger than value of this parameter, then this pair is excluded from further analysis.
approximation-type	Enum	hybrid	Approximation variant used for CPD or Non-iterative Closest Point methods. More details about those types are covered in sections devoted to the methods.
nicp-batch-size	Number	16	Number of Non-iterative Closest Point repetitions run simultaneously. Used only on GPU as the CPU always uses hardware concurrency (number of actual threads in CPU).
nicp-iterations	Number	32	Total number of repetitions for Non-iterative Closest Point algorithm.
nicp-subcloud-size	Number	1000	Number of points taken for error computation in Non-iterative Closest Point algorithm
cpd-weight	Number	0.3	Weight of the uniform distribution that accounts for noise and outliers in Coherent Point Drift algorithm.
cpd-const-scale	Boolean	False	Coherent Point Drift parameter, its value indicate if both clouds are assumed to be in the same scale.
cpd-tolerance	Number	$1e-3$	Minimum relative error value between subsequent iterations for Coherent Point Drift algorithm, indicating if the algorithm should stop computations as if the match was found.
convergence-epsilon	Number	$1e-3$	Minimum error value for indicating if the point sets are treated as matched. Used in all methods.
noise-affected-points-before	Number	0.0	Ratio (in range $[0.0, 1.0]$) of points from the first point set which should be affected by noise.

noise-affected-points-after	Number	0.0	Ratio (in range $[0.0, 1.0]$) of points from the second point set which should be affected by noise.
noise-intensity-before	Number	0.1	Noise intensity for first point set. Noise is then applied by translating each dimension of each point by random value limited by the value of this parameter multiplied by *cloud-spread* value.
noise-intensity-after	Number	0.1	Noise intensity for second point set. Noise is then applied by translating each dimension of each point by random value limited by the value of this parameter multiplied by *cloud-spread* value.
additional-outliers-before	Integer	0	Number of randomly generated points added into the first point set.
additional-outliers-after	Integer	0	Number of randomly generated points added into the second point set.
fgt-ratio-of-far-field	Number	10.0	Ratio of far field used in Fast Gauss Transform. Applies only to a Coherent Point Drift algorithm with hybrid or full approximation.
fgt-order-of-truncation	Integer	8	Order of truncation used in Fast Gauss Transform. Applies only to a Coherent Point Drift algorithm with hybrid or full approximation.

Table 1: Configuration

** - field is required*

7.3 GUI

Both GPU and CPU programs implement the possibility to display visualisation of point clouds utilizing OpenGL. By default, the window display three cloud sizes of different colors:

- Red - the first point set (also called "cloud before")
- Green - the second point set (also called "cloud after")
- Yellow - first point cloud transformed with result transformation

The visualisation window also supports some basic user control. All the possible actions are described in the table below.

Action	Result
Escape	Close the visualisation.
Mouse movement	Rotate the camera.
WSAD	Move the camera (front, back, left, right).
Spacebar	Move the camera up.

Left Shift	Move the camera down.
Scroll	Zoom.
1	Toggle visibility of the first point set.
2	Toggle visibility of the second point set.
3	Toggle visibility of the transformed point set.
4	Toggle visibility of the (0,0,0) point.
[	Reduce the point size.
]	Increase the point size.

Table 2: Renderer controls

8 Description of the results

8.1 Execution times

8.1.1 Introduction

We did not expect CUDA implementations to bring better stability or noise tolerance but we did hope to achieve major improvements in terms of the methods execution times. All the performance tests were run on the machine with the following configuration:

- CPU - AMD Ryzen 7 2700X, 8 cores, 16 threads, 3.7 GHz
- GPU - NVIDIA GeForce RTX 2060 SUPER
- RAM - DDR4 16GB, 3333 MHz, CL16

Different point clouds were used depending on test size:

- $0 - 14904$ points - *bunny.obj*
- $14905 - 35008$ points - *bird.obj*
- $35009 - 333536$ points - *rose.obj*
- $333537 - 376401$ points - *mustang.obj*
- $376402 - 1375028$ points - *airbus.obj*

To achieve a reasonable rate of convergence we decided to test the clouds without any noise and outliers. We used transformation with $0.2\,rad$ rotation and translation with magnitude of 10 units for clouds with spread equal to 10 units. It is worth noting, that the times presented on the charts contains resource allocation and release.

8.1.2 Iterative Closest Point

For Iterative Closest Point method we run benchmarks using one-threaded CPU solution, multi-threaded CPU solution and CUDA-accelerated solution. On the charts, a single iteration time is presented because different runs result in different iterations count which makes the results hard to interpret. The results are presented using logarithmic scale as the differences between variants of the algorithm are significant.

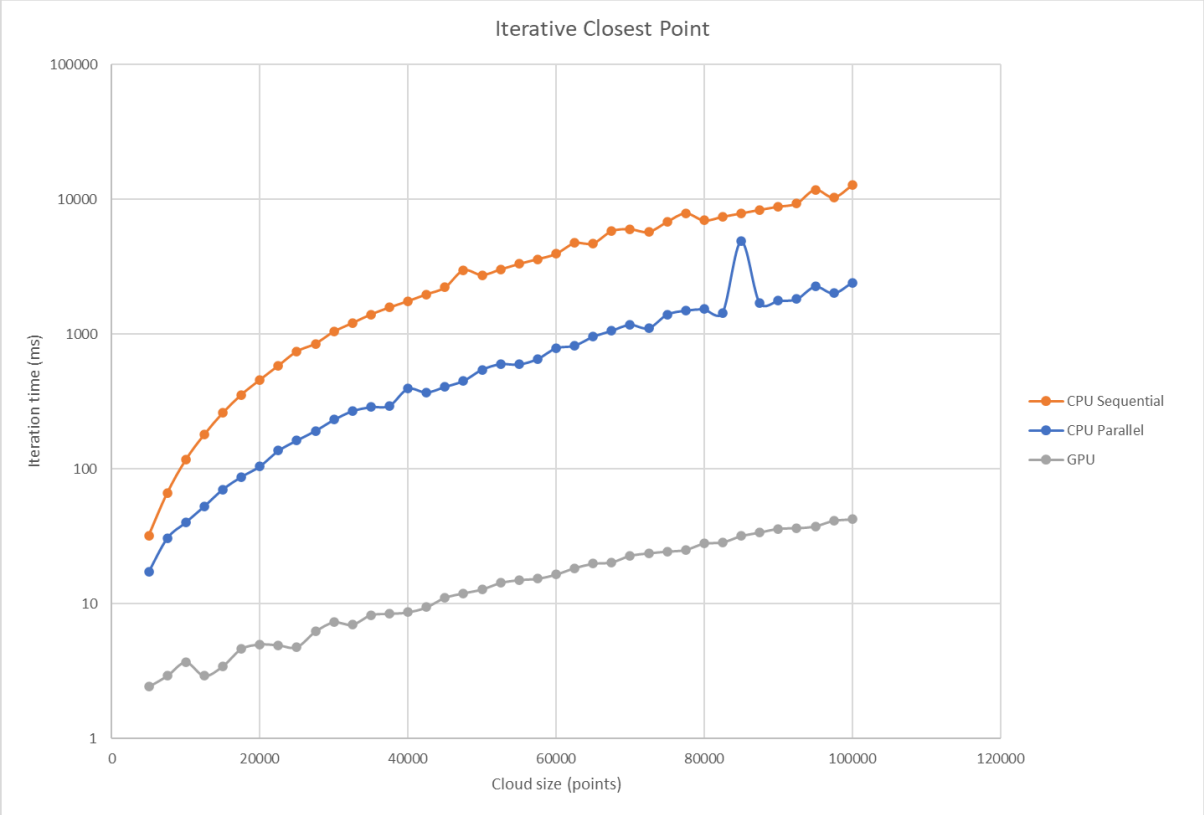


Figure 5: Results of the benchmark for CPU Parallel, CPU Sequential and GPU implementations of the ICP method

As we can see, the benchmarks prove the expectations and parallelisation of the ICP algorithm results in big performance improvements. Having iteration time lower than $100ms$ for clouds with size of 100000 points more tests could be run for the GPU implementation.

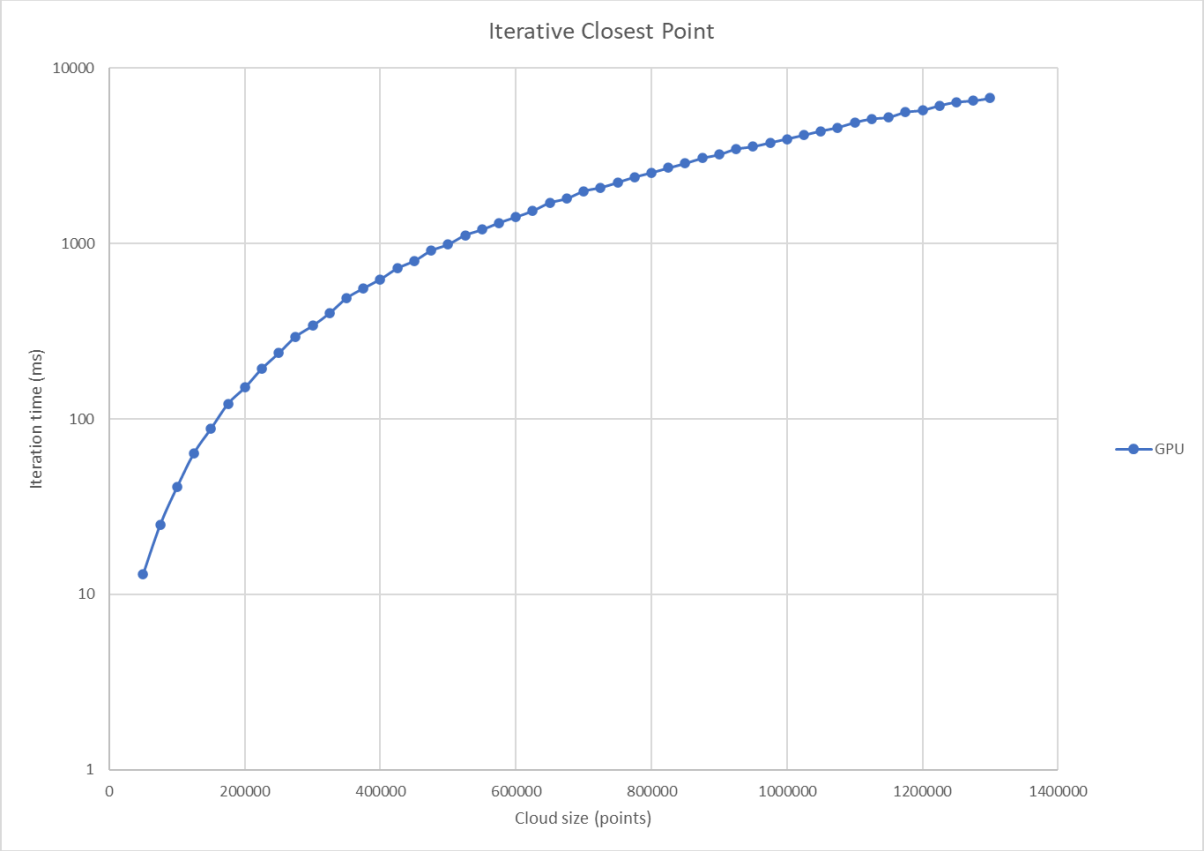


Figure 6: Results of the benchmark for GPU implementation of the ICP method

Even for clouds with size over 1000000 points, one iteration lasts less than 10 s which is a reasonable amount. The benchmarks show that the CUDA-accelerated solution is a perfect choice when utilizing the ICP algorithm.

8.1.3 Non-iterative Closest Point

Non-iterative Closest Point method is performance-wisely a truly interesting one. We expect it to be the fastest because it requires only matrix multiplication and singular matrix decomposition. To avoid an overwhelming amount of information on one chart we split comparisons to separate ones. Because only setting approximation mode to *None* results in a different iteration count in each run we present benchmarks for the whole runs, not single iterations as in ICP section. The maximum iterations parameter is set to 64, batch size to 16 and subcloud size to 1000. We found normal scale on y axis the most suitable to present the results.

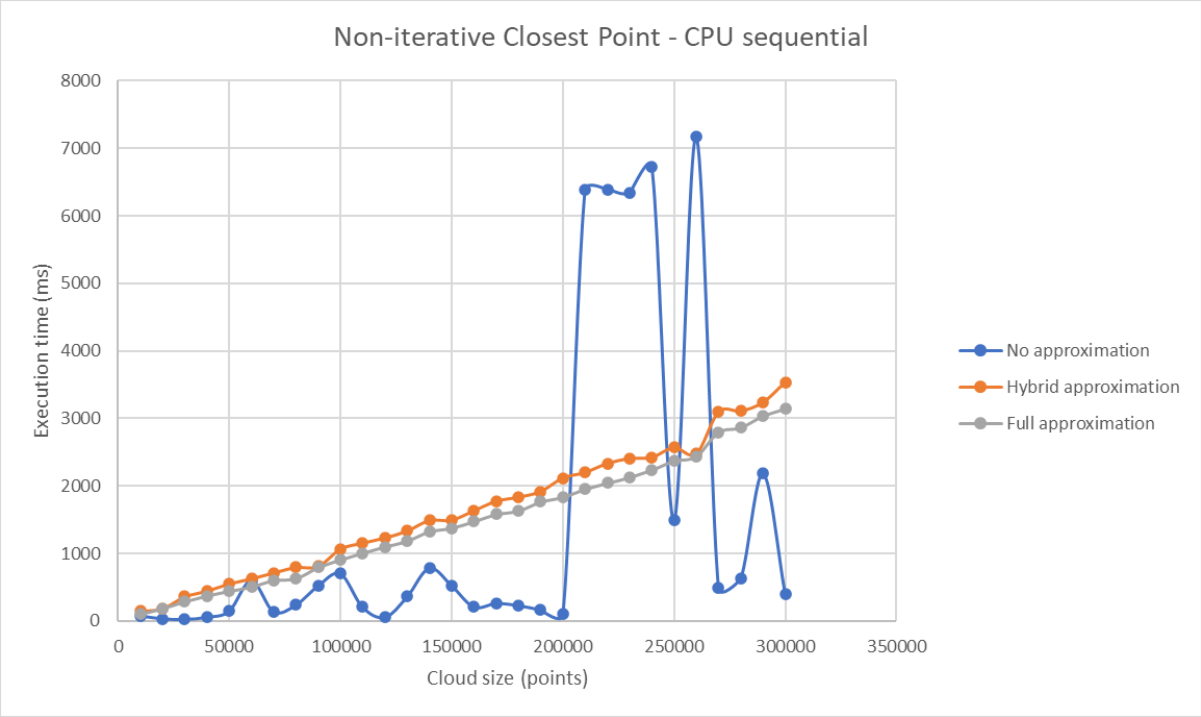


Figure 7: Results of the benchmark for sequential CPU implementation of the Non-iterative Closest Point method

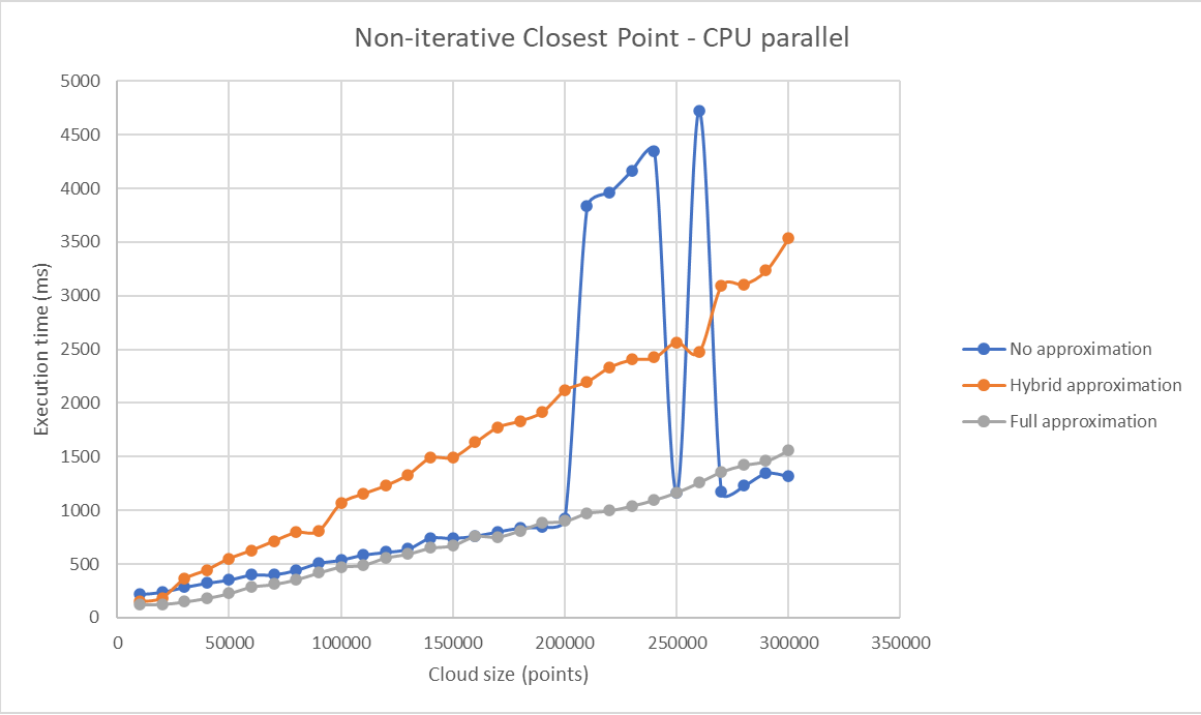


Figure 8: Results of the benchmark for parallel CPU implementation of the Non-iterative Closest Point method

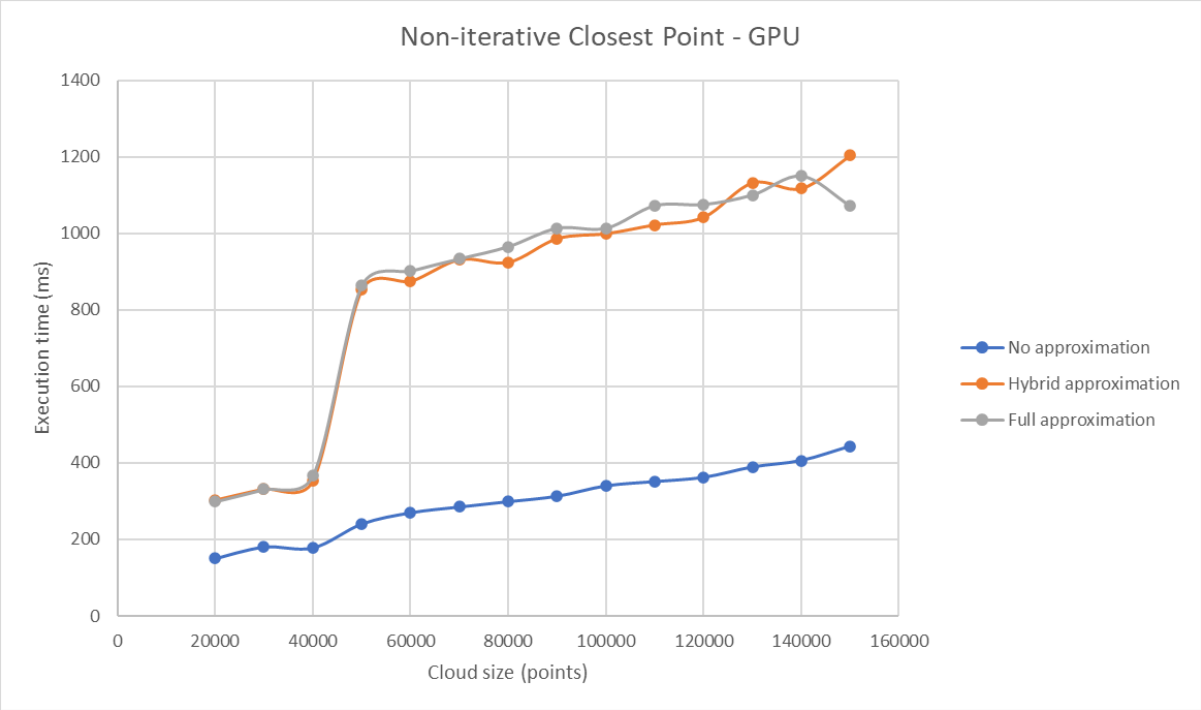


Figure 9: Results of the benchmark for GPU implementation of the Non-iterative Closest Point method

The benchmarks have clearly shown that in practical usages the version without any approximation yields the best results. We expected *Hybrid* and *Full* approximation to improve overall performance, although the tests have shown that calculating the error each step is a better choice than approximating it and choosing the result after all iterations are finished.

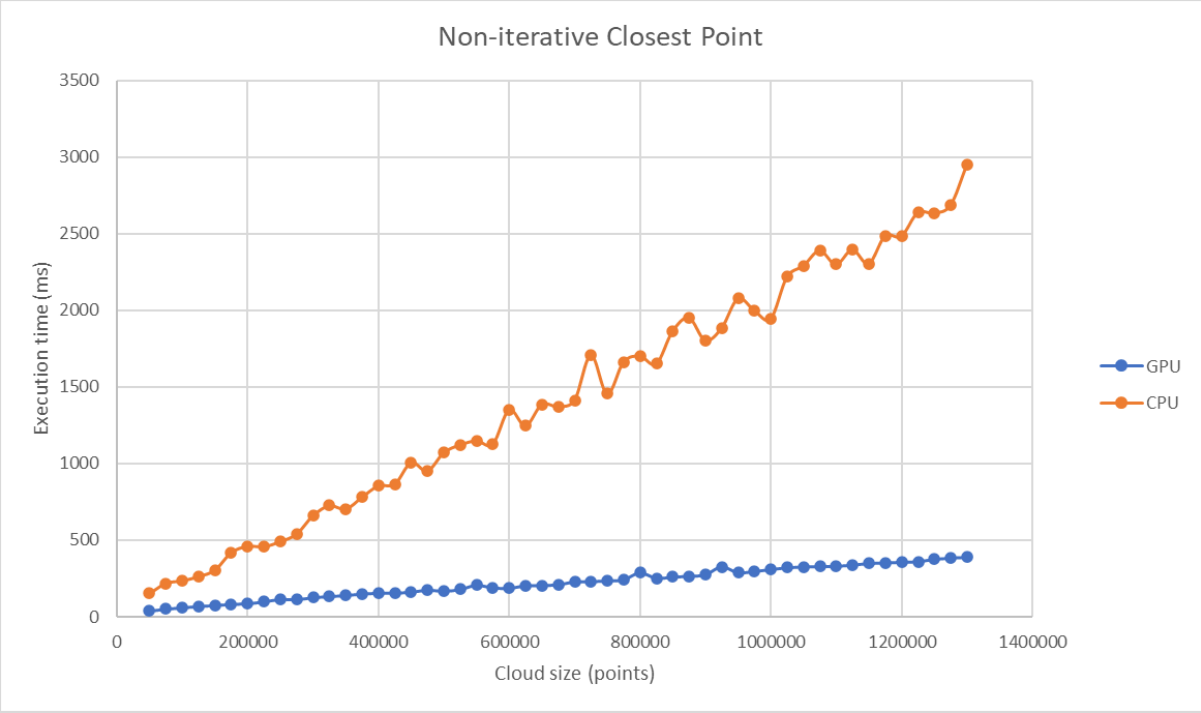


Figure 10: Results of the benchmark for CPU and GPU implementation of the Non-iterative Closest Point method

To the benchmark comparing the GPU and CPU implementations we chose the most performant variants of the algorithms. The CUDA-accelerated implementation proved to be faster than the CPU

parallel. It is worth noting that for cloud over 1000000 points it achieves execution time below 500 ms.

8.1.4 Coherent Point Drift

Coherent Point Drift algorithm was the hardest to implement and it is certainly the most complicated considering the scope of our research. Therefore we do not expect CUDA solution to bring significant improvement in this case.

For the performance tests *cpd-weight* parameter was set to 0.1 and *const-scale* to $false$. The other parameters were the same as in the other benchmarks. We present the results using logarithmic scale on y axis. Similarly to Non-iterative Closest Point benchmarks we compare three approximation types, this time approximations relying on using Fast Gauss Transform.

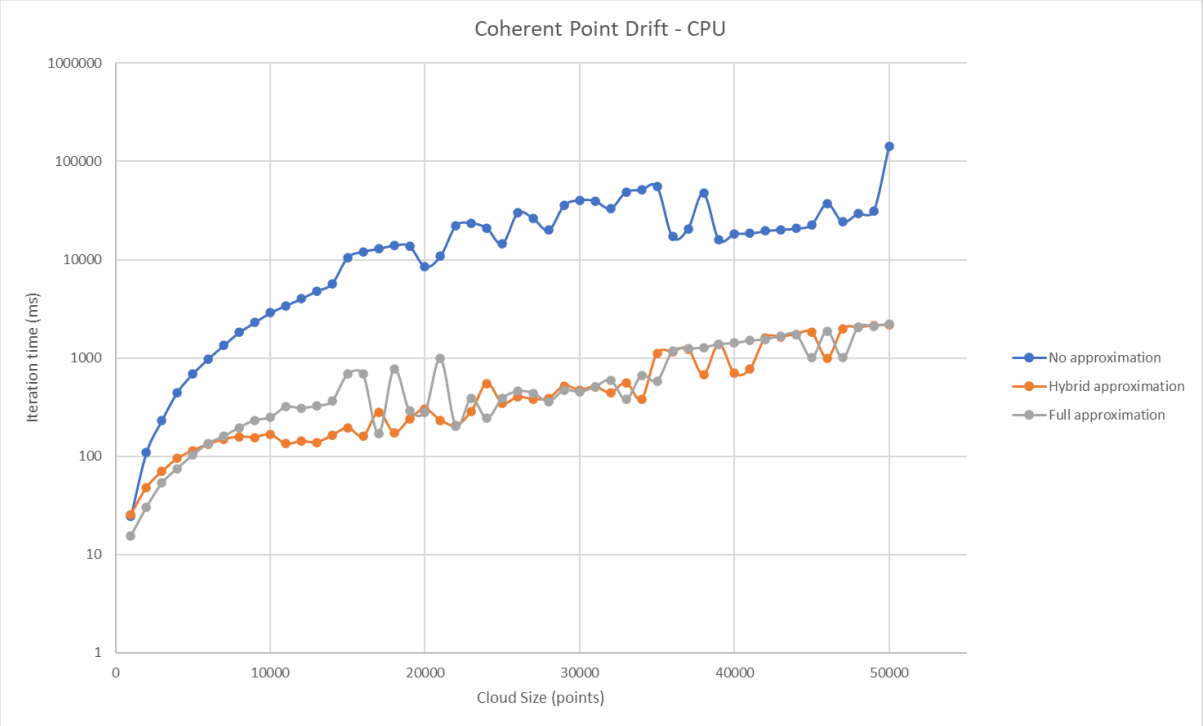


Figure 11: Results of the benchmark for CPU implementation of the Coherent Point Drift method

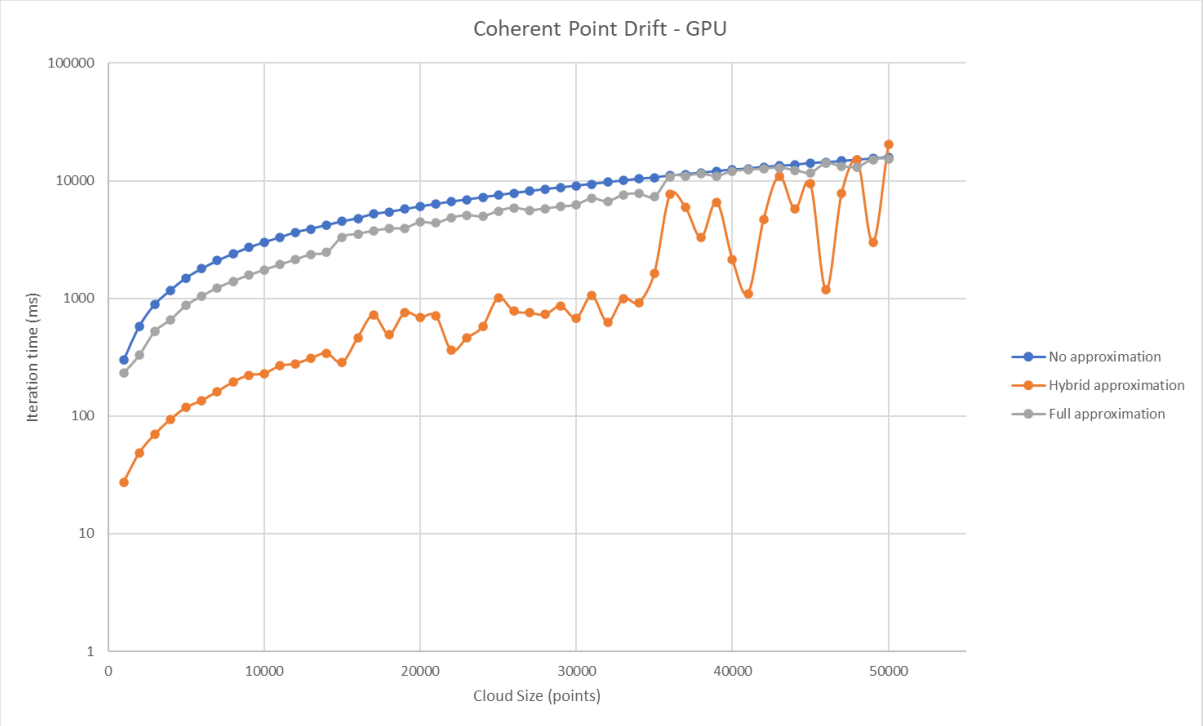


Figure 12: Results of the benchmark for GPU implementation of the Coherent Point Drift method

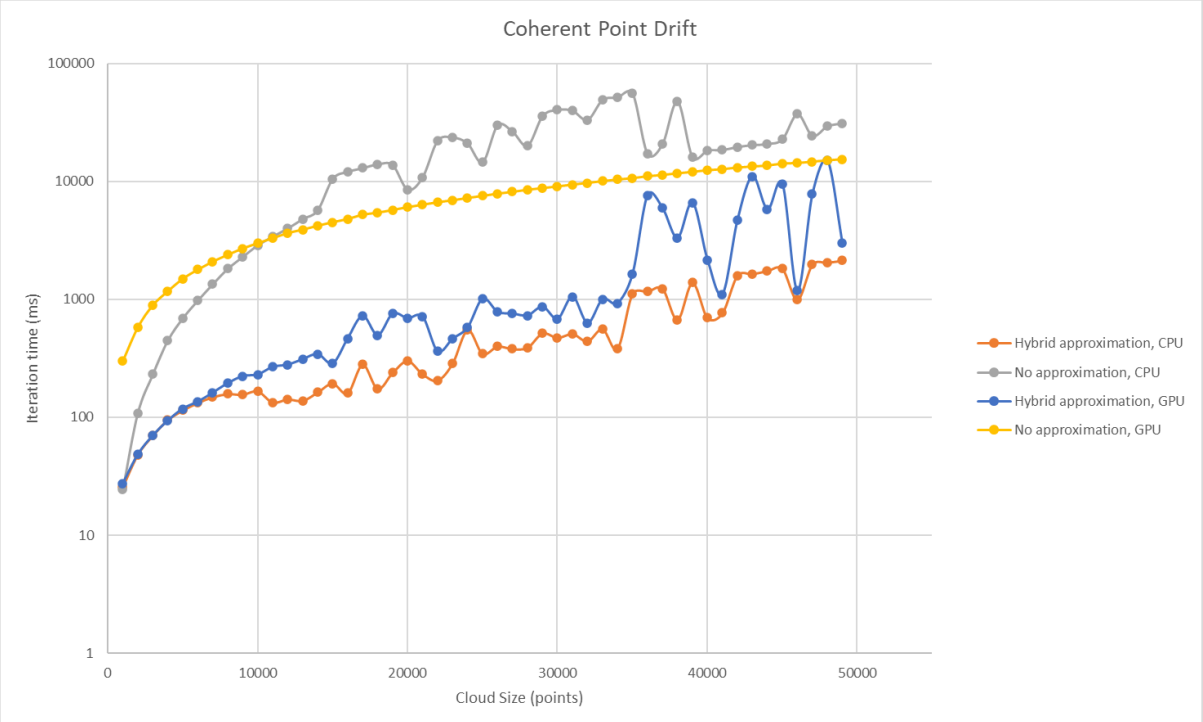


Figure 13: Results of the benchmark for CPU and GPU implementation of the Coherent Point Drift method

We can note that implementations utilising Fast Gauss Transform are considerably faster than the ones without approximation. GPU acceleration did not bring satisfying improvements, however the variant without approximation is an exception because it is fastest and gives more predictable execution times than CPU implementation.

8.1.5 Methods comparison

To sum up the section describing execution times, we gathered all the methods benchmarks on one chart. The parameters were the same as in all the other benchmarks, however this time an execution time is presented for all the methods. Because the differences between all the variants are substantial we used a logarithmic scale on y axis.

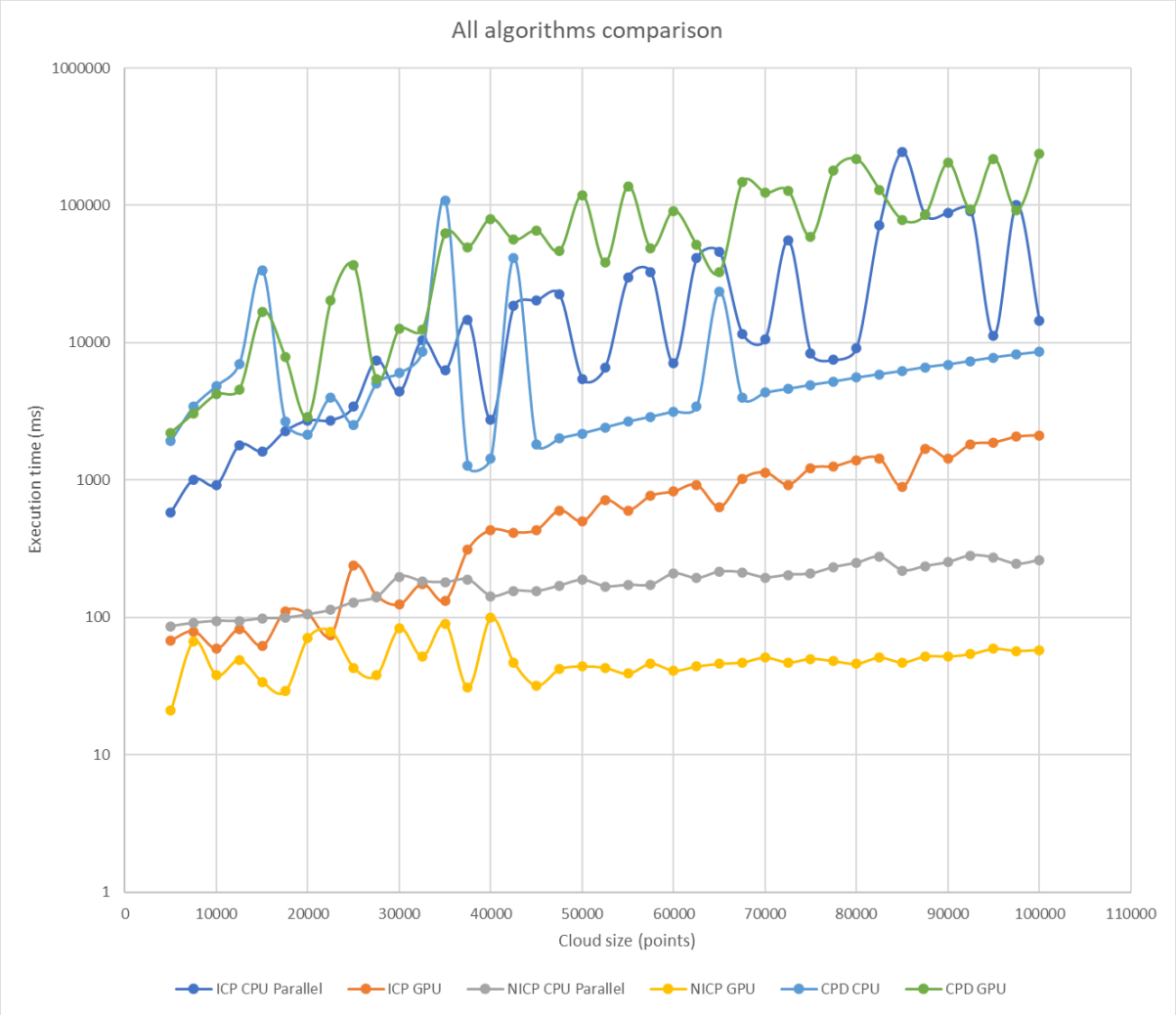


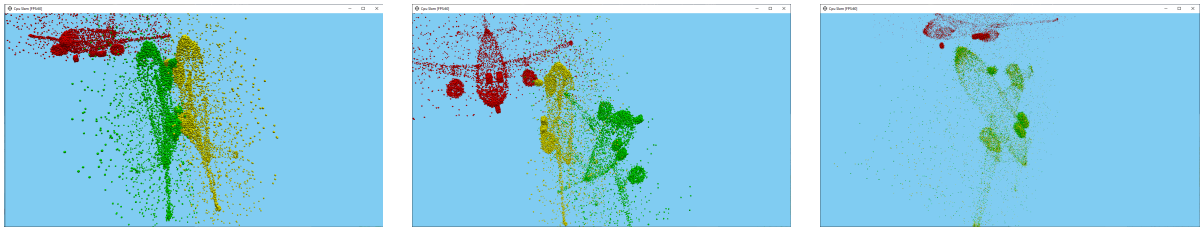
Figure 14: Comparison of all methods execution times

As we can see the execution times differ vastly depending on method and execution policy. We are satisfied with the results especially for Iterative Closest Point and Non-iterative Closest Point methods because we managed to significantly improve the performance by utilizing CUDA library. Therefore, these two methods would be the best choice if time is the most important factor while utilizing SLAM. Coherent Point Drift was expected to be the slowest of all the methods due to its sophistication. However its CPU implementation using Fast Gauss Transform occurred to be faster than ICP CPU variant and it is usable even for clouds with considerable sizes.

8.2 Noise and outliers tolerance

We tested our methods using the clouds and conditions that may be closer to reality. That is, we used clouds that are missing points, even large part of them, and we corrupted clouds by noise and outliers. For this purpose we have used CPU versions of algorithms. For ICP and NICP *policy* was set to *parallel* and *approximation-type* to *None*. For CPD *approximation-type* was set to *Hybrid*, mainly to achieve lower computation times. Many tests were conducted and it turned out that the results are highly dependent on configuration. Therefore, in this section you can find the most interesting results and tests

that have shown the most consistent characteristics. All the results are available in spreadsheet kept on Github: https://github.com/Sliwson/cuda-slam/tree/master/doc/noise. All the test configurations are to be found in a subdirectory *configs*.
After testing, we came to several conclusions. Firstly, the ICP algorithm gave the worst results for almost all the test sets. The main reason could be cloud sizes used in tests, which occurred to be often too big for ICP capabilities. Algorithm had also trouble with noise. Even low noise level such as *noise-affected-points-before* equal to 0.1 and *noise-intensity-before* equal to 0.05 may cause convergence failure. Secondly, increasing *max-iterations* can help achieving more exact results.
CPD algorithm mostly achieved correct results. Main key to CPD convergence is to select parameters value correctly. The most important one is *cpd-weight*. Difference of 0.3 can lead to significantly different result. Second CPD parameter is *cpd-const-scale*. Setting it to *true* may help to achieve better results when we know that the scale did not change after transformation. Especially it helps when clouds contain a lot of noise and outliers. Also, decreasing *cpd-tolerance* and *converge-epsilon* may result in increasing iterations count and lead to more exact results.
All the algorithms had tendency to match areas where the points density was the highest. An example of this behaviour may be observed using *config21.json*. It uses two "airbus" clouds with decreased sizes and added noise and outliers. Result achieved by algorithms are presented below and vary among each other.

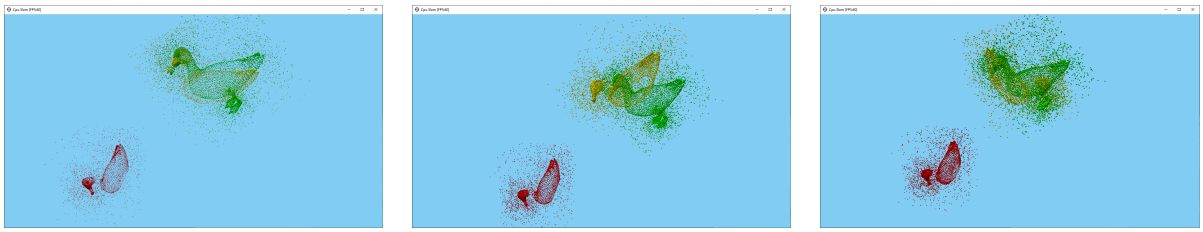


(a) CPD (b) ICP (c) NICP

Figure 15: Tests with *config21.json*

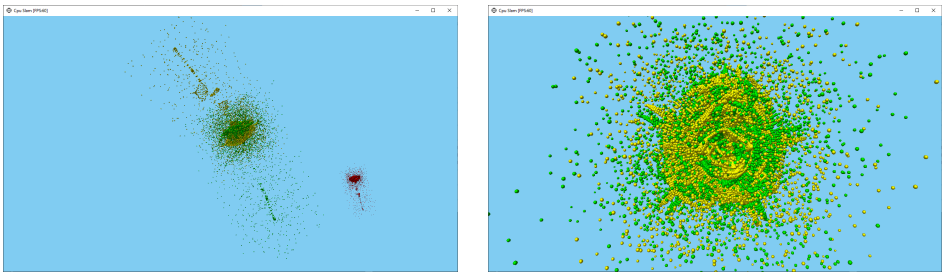
The ICP did not manage to converge in this example but the result achieved by CPD is interesting. Algorithm connected wheels and engines of both the airplanes. Feasibly, that is because those are areas with the highest point density in the entire cloud. NICP returned the exact transformation.

Similar situation was met using *config11.json* and *config18.json*. Unfortunately in tests using the latter one, ICP returned identity matrix. The results are presented below.



(a) CPD (b) ICP (c) NICP

Figure 16: Tests with *config11.json*

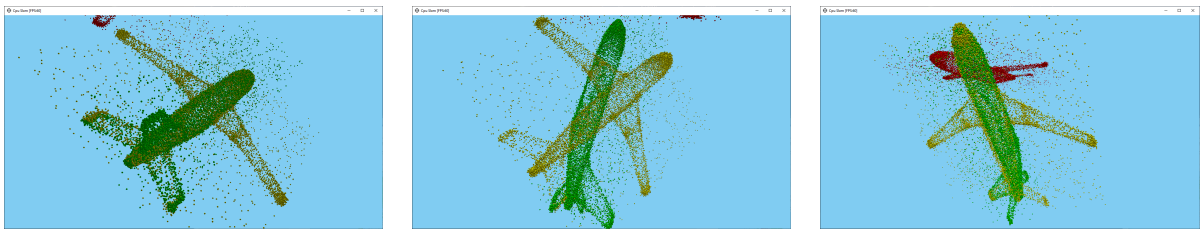


(a) CPD (b) NICP

Figure 17: Tests with *config18.json*

In contrary to *config18.json* and *config21.json*, using *config11.json* CPD found almost exact solution while this time NICP focused on matching areas with the highest density.

Another example of CPD accuracy in tough conditions can be observed when using *config26.json*. We used two "airbus" clouds where first had no wheels, engines and a vertical stabilizer. The second one had wings and wheels cut. Also we have added noise and outliers to both clouds. Results are presented below.

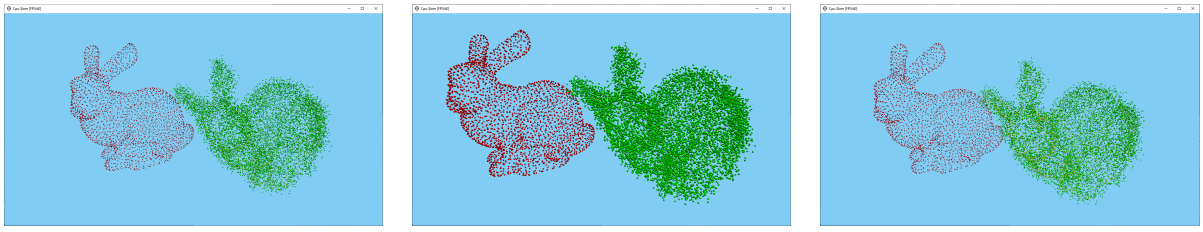


(a) CPD (b) ICP (c) NICP

Figure 18: Tests with *config26.json*

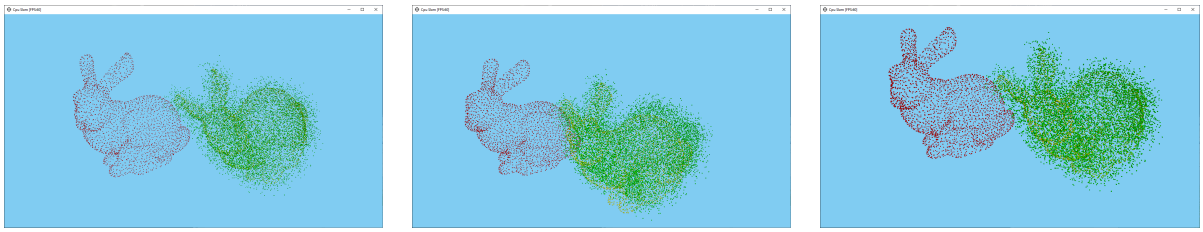
CPD found the exact solution. As mentioned above, ICP has troubles with incomplete clouds and it remains unchanged with this configuration. NICP was close to finding exact solution; however, it lacks of 90° rotation.

Finally, we prepared a test that focuses only on a situation where one cloud contains noise. We increased the *noise-intensity* by small values and compared the results. The best method occurred to be NICP, CPD was the second and ICP was the third one. For those tests, the configurations *config28.json* to *config38.json* were used. Results are presented below.



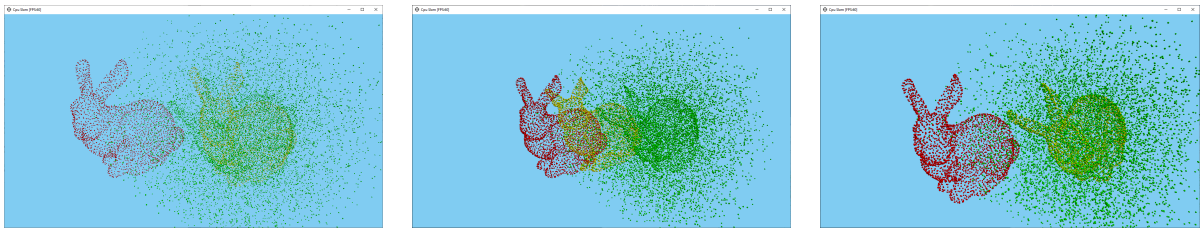
(a) CPD (b) ICP (c) NICP

Figure 19: Tests with *config29.json*



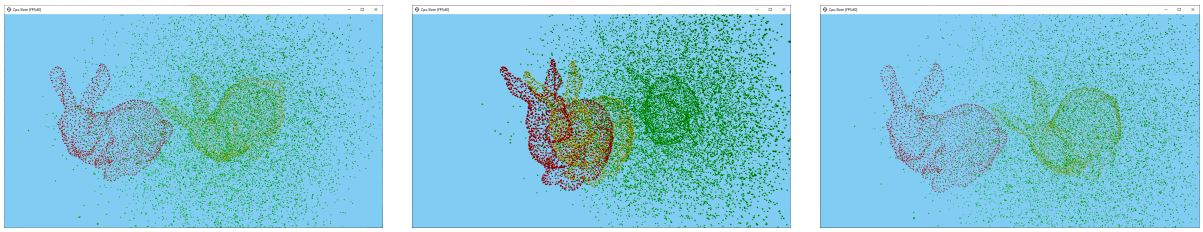
(a) CPD (b) ICP (c) NICP

Figure 20: Tests with *config30.json*



(a) CPD (b) ICP (c) NICP

Figure 21: Tests with *config34.json*



(a) CPD (b) ICP (c) NICP

Figure 22: Tests with *config38.json*

NICP managed to find the exact solution in all the test cases. Results obtained by CPD were slightly worse and sometimes correction of *cpd-weight* parameter was needed. ICP results started to differ even when *noise-intensity* was still very small, thus its usage when conditions are not perfect is very limited.

To sum up, the best algorithms for clouds with noises and outliers are NICP and CPD. In this situation, it is also worth mentioning that NICP was significantly faster than CPD in all the tests covered in this section.

8.3 Convergence ranges

8.3.1 Introduction

Another topic for a discussion when it comes to methods' usability is a quality of their results. This section covers convergence rates for all the algorithms, depending on cloud sizes, rotation and translation. Another parameters are same for all the tests, including cloud spread equal to 10. For all the methods, both GPU and CPU implementations, tests have covered three different translation ranges: 10, 20 and 30 units and three different rotation ranges: 0.2, 0.4 and 0.6 radians, which are equal to 11.5°, 22.9°and 34.4°respectively.

Based on the performance results from previous sections, cloud sizes have been chosen separately for each method and they differ significantly from each other, but remain the same between GPU and CPU implementation of a specific method. It is also worth noting that for methods with multiple approximation types, only one type is chosen for the tests and the decision is described in the section focusing on a specific method.

8.3.2 Iterative Closest Point

In case of an ICP, convergence rates are deeply correlated with a maximum number of iterations the algorithm runs. And this turns out to be the main drawback of this method. In order to compare methods most fairly, we had to limit the iterations in a way that keeps computation times close to the other methods. At the same time, we needed to keep similar parameters for both CPU and GPU implementation and for this reason iterations have been limited by 100. Cloud sizes have been chosen from range $[20000, 100000]$ which positions ICP in the middle of the road between CPD and Non-iterative Closest Point.

Unfortunately, it is noticeable from the chart below that for large clouds, convergence rate falls dramatically. It is caused mainly by limiting the iterations mentioned above. In most of the test cases, the error in consecutive iterations tended to fall down, but the dynamics of changes did not allow the algorithm to converge in acceptable time.

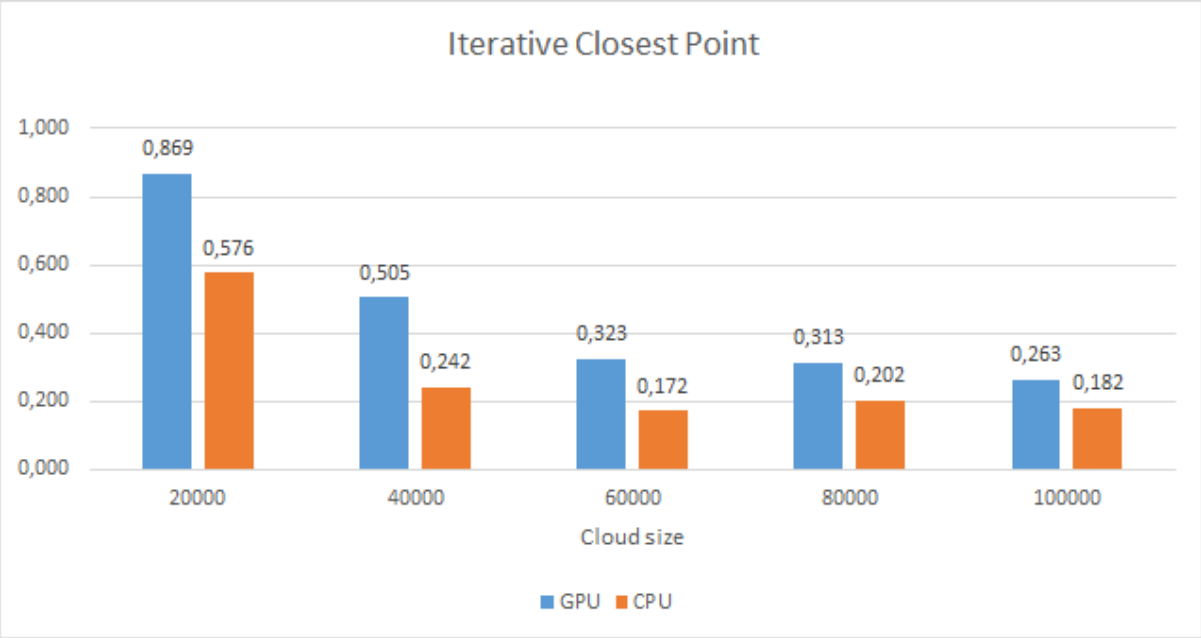


Figure 23: Convergance rates for ICP depending on cloud size

The situation is slightly different when it comes to comparing ICP efficiency based on translation between first and second point cloud. In this case, the differences are not so grand and only large translation cause the visible decrease of convergence rate. What is worth mentioning are the differences between GPU and CPU implementation. GPU turns out to be even more than 5 times better when it comes to matching clouds with large translation. This may indicate lower accuracy of math libraries used on CPU, especially for decomposition of matrices with larger values.

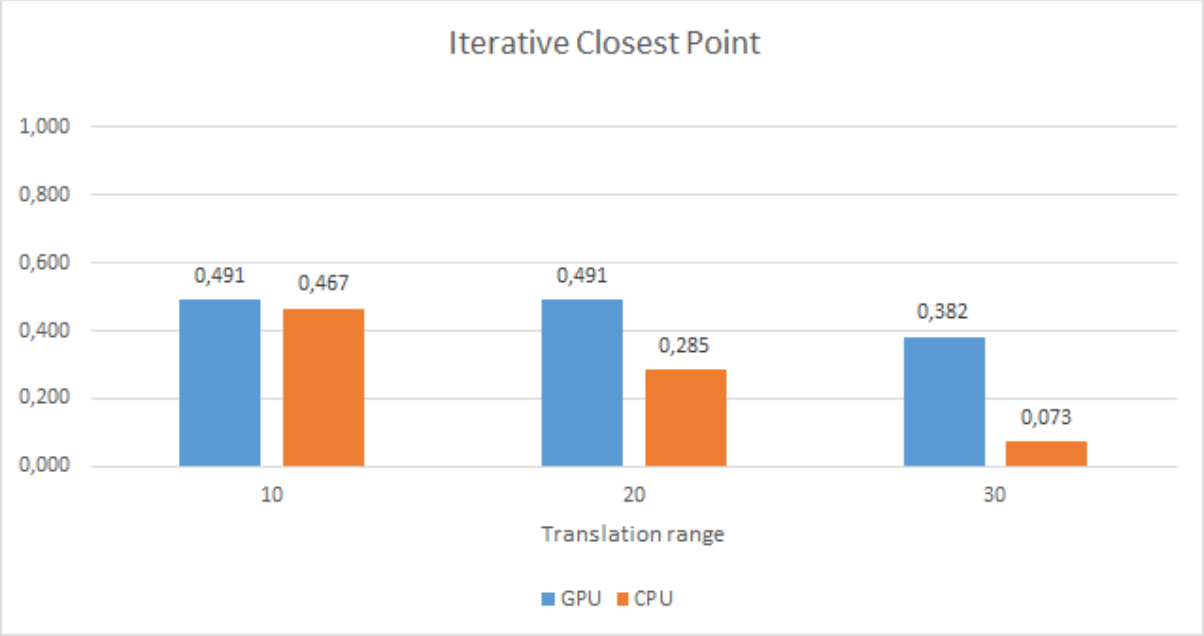


Figure 24: Convergance rates for ICP depending on translation

The differences between GPU and CPU computations are not that evident if we compare the rotated clouds. In this situation, the CPU implementation tend to maintain similar convergence rate, despite the rotation value. The overall GPU dynamics in this case is the same as for the tests based on translation.

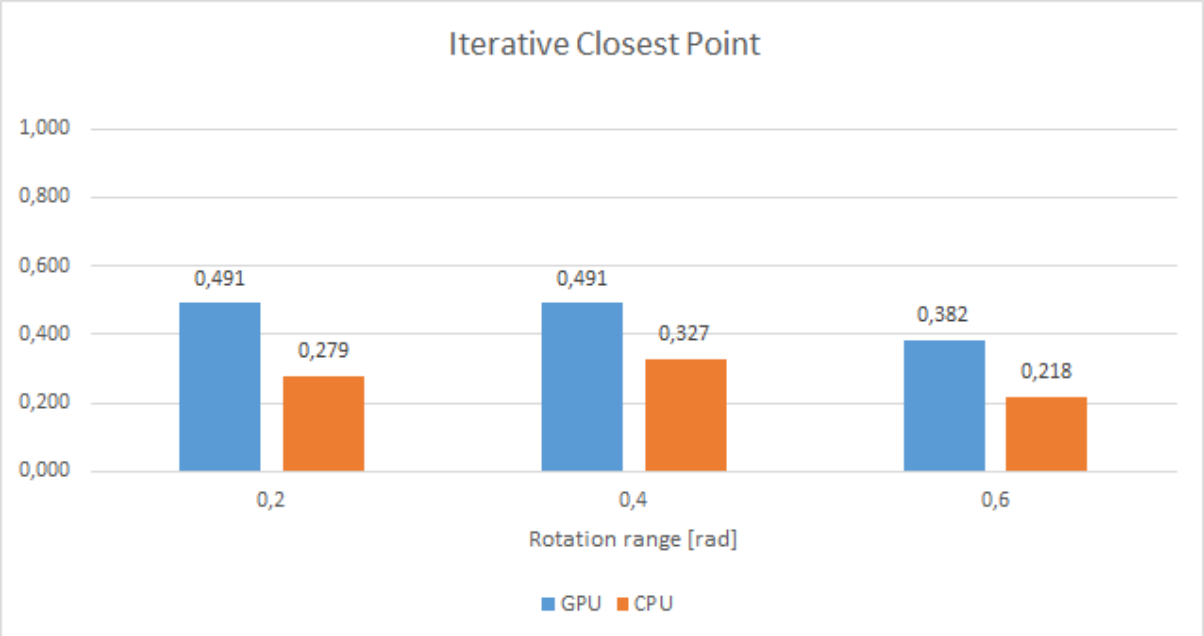


Figure 25: Convergance rates for ICP depending on rotation

Summarizing the convergence tests for an ICP algorithm, the method does not show great differences between convergence rate based on translation and rotation, but the rates are in all the cases rather low. The method's advantage is the possibility of significantly improving the results by increasing the number of allowed iterations. In most of the test cases, it was this limitation, not the maximum error what caused the computations to stop.

8.3.3 Non-iterative Closest Point

Non-iterative closest point method is the first one for which different approximation types have been implemented. However, the execution times proved that only one type is worth using in general cases. For this reason, the "None" type is used in all convergence tests, as it not only gives the most accurate results, but it also offers execution time unsurpassed by other methods.

Performance tests for proposed implementation showed as well that substantial cloud sizes are not an obstacle for a Non-iterative Closest Point algorithm and because of that, the sizes used in the convergence tests vary between 250000 and 1250000. Nevertheless, execution times are still lower than for the other algorithms.

Not only are the cloud sizes and execution times surpassing other methods, but also the convergence rates show similar tendency. The overall convergence rate for non-iterative method was equal to 1.0 for GPU implementation and 0.99 for CPU. The impact of specific parameters on the results is displayed on charts below.

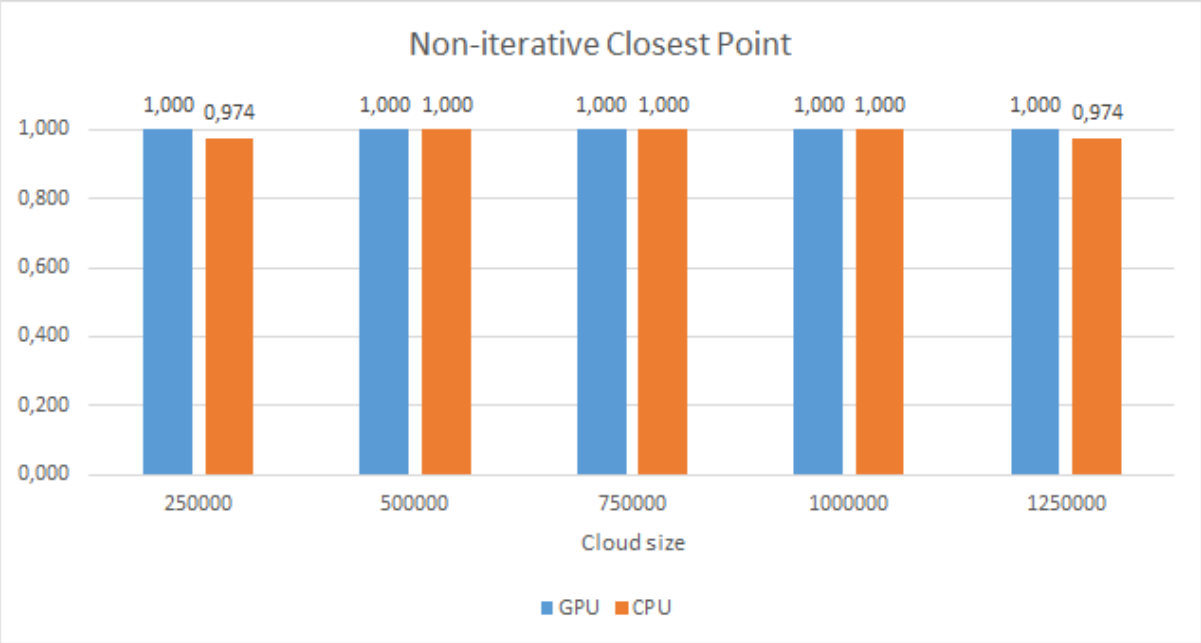


Figure 26: Convergance rates for a Non-iterative Closest Point matching depending on cloud size

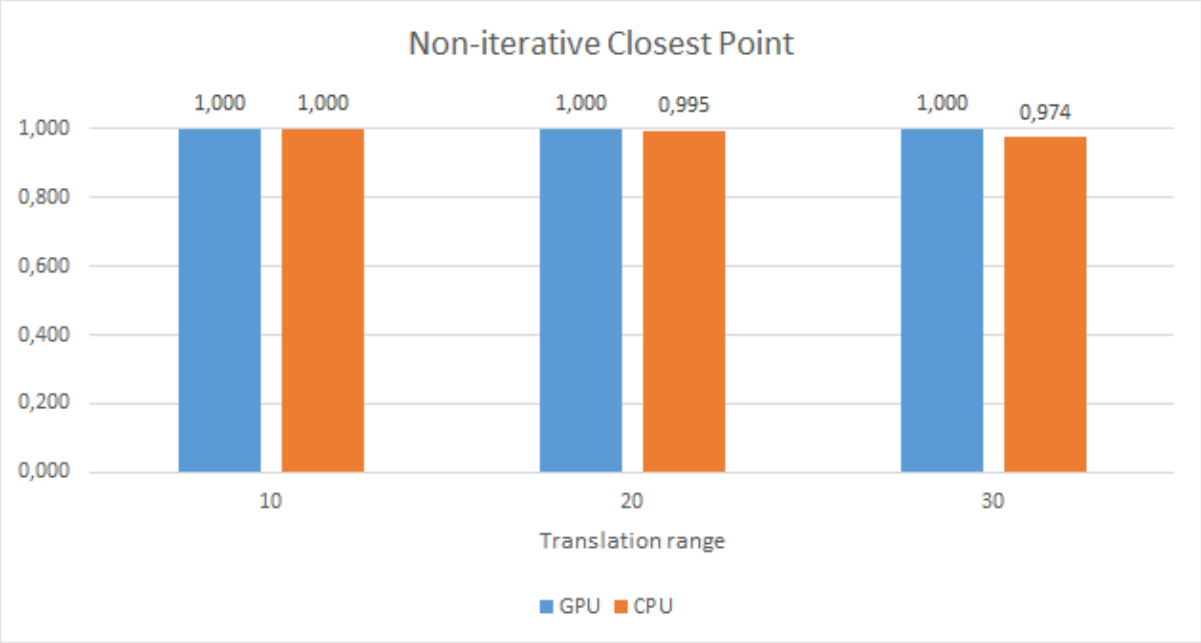


Figure 27: Convergance rates for a Non-iterative Closest Point matching depending on translation

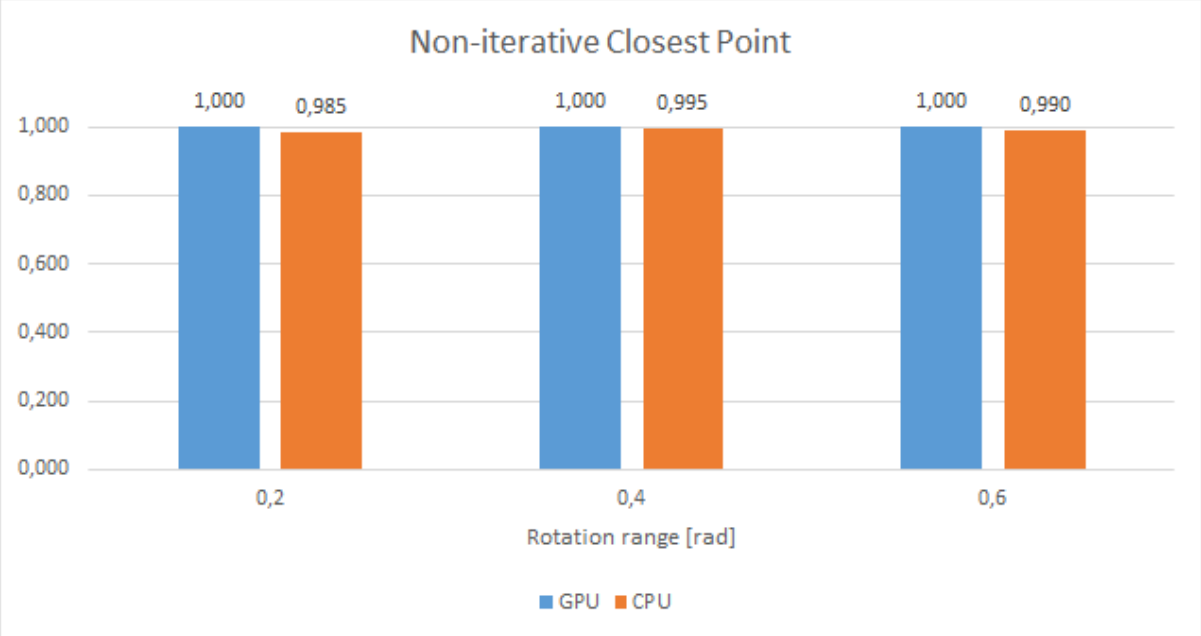


Figure 28: Convergance rates for a Non-iterative Closest Point matching depending on rotation

The slim disparity between the two variants of a implemented method are most likely caused by precision of CPU and GPU matrix operations which strengthens the hypothesis put forward while analysing the ICP algorithm. However, in this case the differences are not so grand which can be caused by reduced number of matrix operation and therefore less error propagation.

8.3.4 Coherent Point Drift

Coherent Point Drift is another method parametrized by approximation type it uses. In this case, we decided to pick the hybrid variant, guided mainly by performance results and the fact that improvements done by implementing Fast Gauss Transform have no impact on convergence rate. All above make hybrid approximation the most competitive version out of all three implemented.

However, the run times of CPD algorithm are still far higher than for the other methods and for that reason, cloud sizes chosen are significantly lower being in range $[4000, 20000]$, still offering the longest computation time. On the other hand, convergence rates are much more competitive comparing to the results of ICP for computations in comparable time.

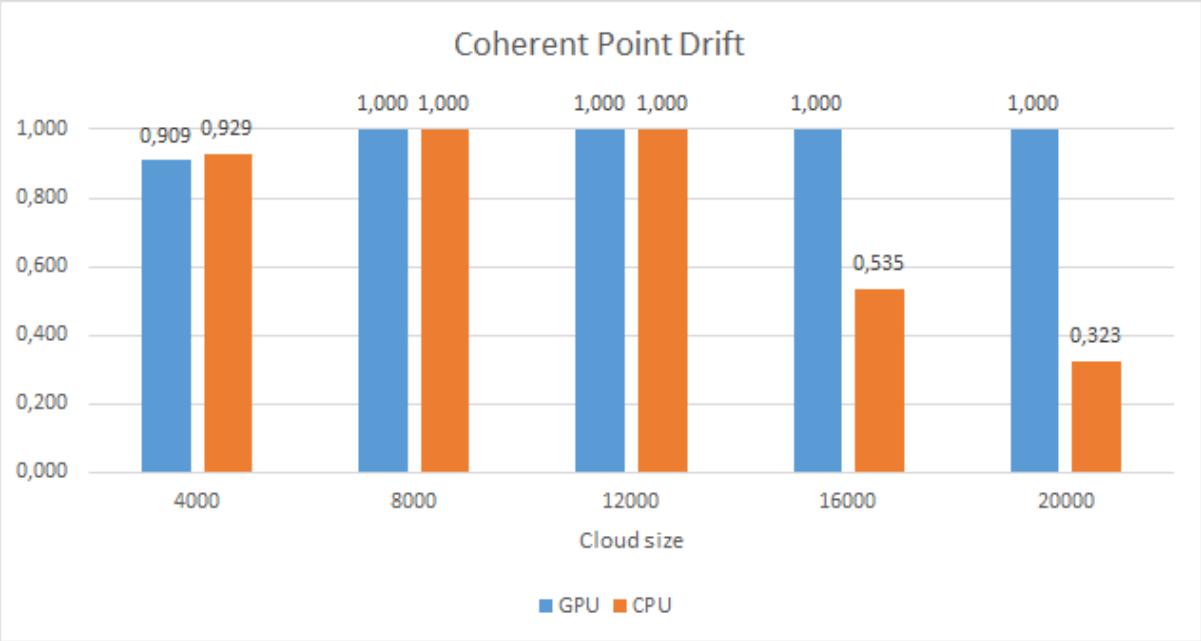


Figure 29: Convergence rates for a Coherent Point Drift depending on cloud size

Nevertheless, the convergence rates of CPU algorithm reveal a trend towards a drastic decline for larger clouds. It is noticeable that for objects of size above 16000 points, rate is even lower than 0.5 when the GPU implementation keeps the results perfectly matched. This seems to be the case also for increasing translation between clouds. This time the differences are not so grand though.

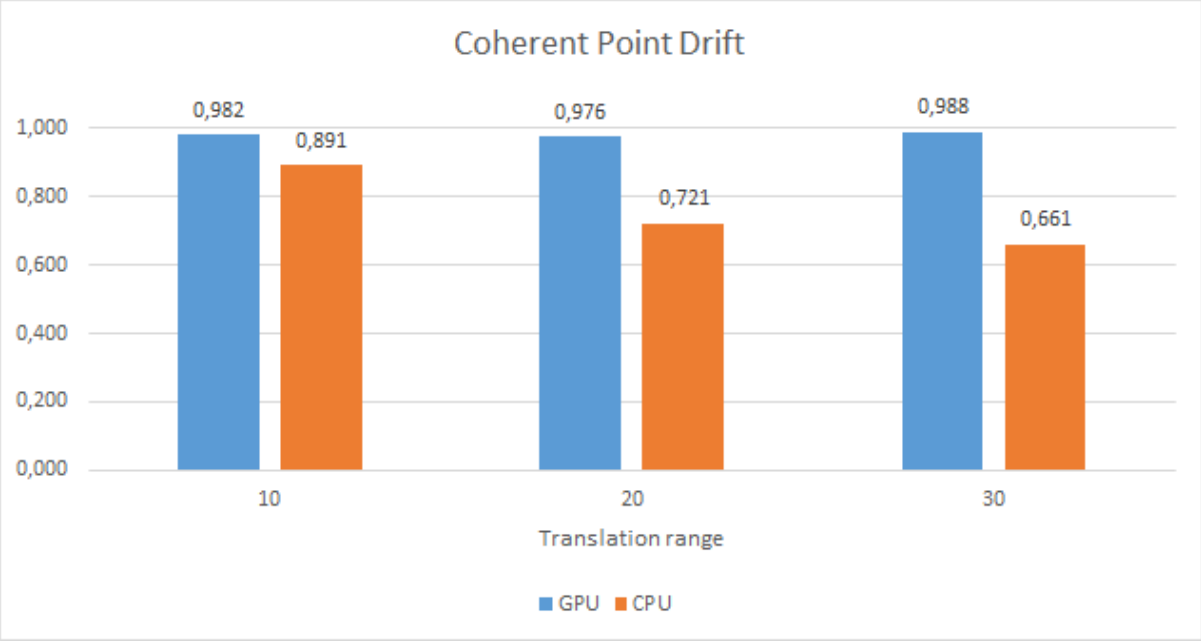


Figure 30: Convergence rates for a Coherent Point Drift depending on translation

Modifying a rotation of target cloud, the phenomenon described above seems to completely disappear. What is more, larger rotation results in higher convergence rate of tested algorithm but differences

between them are significantly lower.

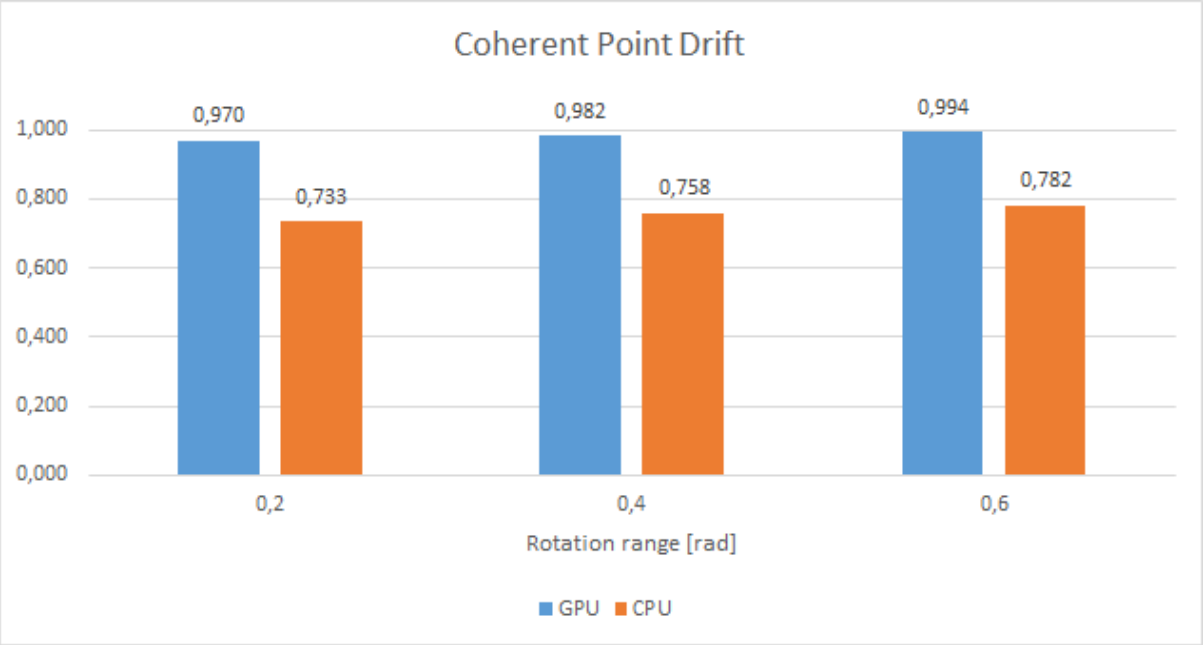


Figure 31: Convergence rates for a Coherent Point Drift depending on rotation

Summarizing the convergence of implemented CPD algorithms, we strongly encourage to use GPU version rather than CPU. Even tough computation times can be greater, convergence rate is then kept on similar and very high level above 0.9. This method is also the only one that can match scaled clouds which makes it more universal.

Once again, CPU implementation gives far worse results than the GPU one. For this method that might be caused by using operations on large matrices not only for SVD but also for a multiplication.

8.3.5 Methods comparison

To sum up the convergence tests, the average rate for all tests have been calculated for each method, with division to GPU and CPU implementation. Based on that, it is easier to visualise the results that were already mentioned in all the sections.

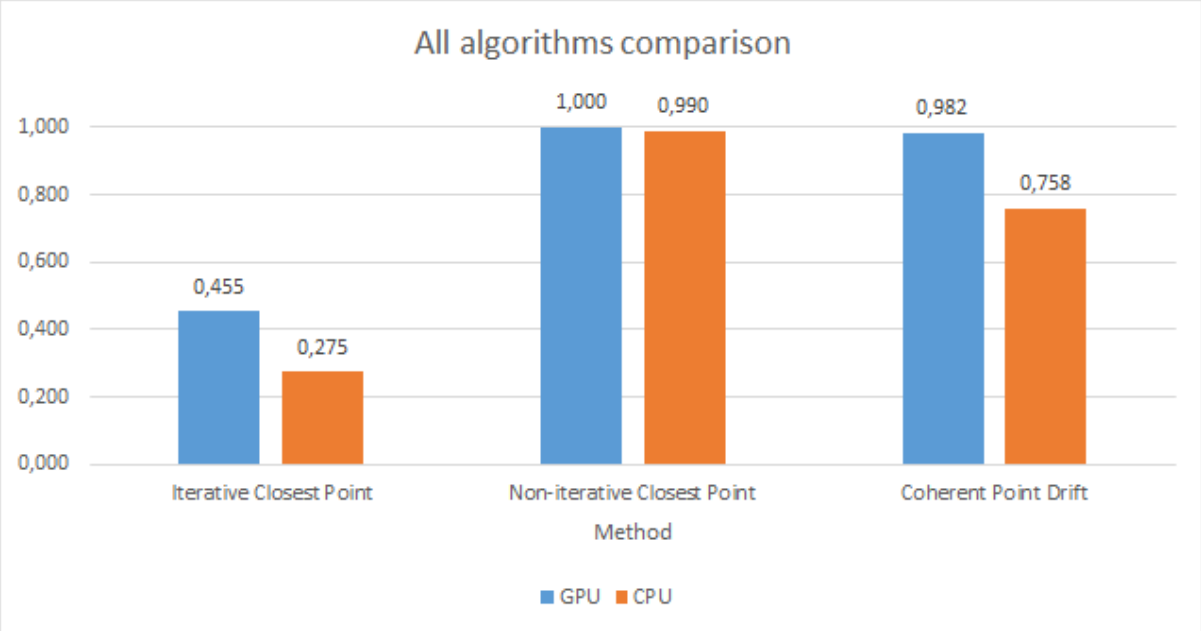


Figure 32: Comparison of all algorithms convergence rates

And once again it is clear that CPD and Non-iterative closest point methods are a lot more accurate when it comes to finding transformation between two point sets.

Another conclusion is that CPU implementation of complicated matrix operations is significantly inferior to the one accelerated by GPU. It might be the case only for the two libraries that were used in this project, however the CPU library we used is most widely recommended.

9 Future works

9.1 Iterative Closest Point

We implemented a solid ground base for development of and ICP algorithm. There are a lot of ways it can be improved but the most important would be enhancing performance of correspondences step as it is crucial for the whole method execution time.

9.2 Coherent Point Drift

We implemented whole rigid part of the algorithm on CPU, including FGT. Unfortunately FGT was not implemented on GPU as mentioned in *5. Implementation details.* This is main area of future improvement for GPU solution. Coherent Point Drift also has affine and non-rigid versions that are worth mentioning. There were not implemented because it reaches beyond the scope of our project. Future works can mainly focus on improving GPU solution and moving to non-rigid transformations.

9.3 Non-iterative Closest Point

This method is for sure the most narrowly documented among all three described in this report. On the other hand, the idea does not leave much space for improvements. The biggest obstacle to using it on greater scale is its undetermined convergence ratio. Although the tests conducted showed that running the algorithm for a reasonable number of pseudo-random permutations gives convergence ratio of almost a 100%.

Other area of improvements is related closely to implementation issue. Different libraries and methods of computing SVD offers different accuracy. This can be seen for example by comparing results of CPU computations using Eigen library (recommended as one of the top when it comes to finding matrix decomposition) and GPU's cuSOLVER, where for the second one, error is smaller in majority of the test cases. There are also multiple other, both closed and open source implementations for both CPU and GPU which might offer better results when it comes to convergence or computation time, especially for some edge cases.

9.4 Common improvements

The key to performant SLAM for big clouds is sampling. We have not covered it in our work as it is a broad topic and mostly dependent of the specific application of the algorithm. Although, there is no contraindication to use sampled clouds as the input of our program so everybody can make their own experiments in that field.

References

[1] Wolfram Burgard et al. *Iterative Closest Point Algorithm*. http://ais.informatik.uni-freiburg.de/teaching/ss12/robotics/slides/17-icp.pdf.

[2] Leslie Greengard and John Strain. "The Fast Gauss Transform". In: *SIAM J. Sci. Stat. Comput.* 12.1 (Jan. 1991), pp. 79–94. ISSN: 0196-5204.

[3] Andriy Myronenko and Xubo Song. "Point Set Registration: Coherent Point Drift". In: *IEEE transactions on pattern analysis and machine intelligence* 32 (Dec. 2010), pp. 2262–75. DOI: 10.1109/TPAMI.2010.46.

[4] Shinji Oomori, Takeshi Nishida, and Shuichi Kurogi. "Point cloud matching using singular value decomposition". In: *Artificial Life and Robotics* 21 (June 2016), pp. 149–154. DOI: 10.1007/s10015-016-0265-x.

[5] Sebastien Paris. *Fast Gaussian Transform mex implementation*. 2020. URL: https://www.mathworks.com/matlabcentral/fileexchange/17438-fast-gaussian-transform-mex-implementation.

[6] *Simultaneous localization and mapping*. https://en.wikipedia.org/wiki/Simultaneous_localization_and_mapping.