https://doi.org/10.20965/jrm.2020.p0548

**Paper:**

# Fast Euclidean Cluster Extraction Using GPUs

**Anh Nguyen***, **Abraham Monrroy Cano***, **Masato Edahiro***, **and Shinpei Kato****

*Graduate School of Informatics, Nagoya University
Furo-cho, Chikusa-ku, Nagoya 464-8601, Japan
E-mail: {anh, amonrroy, eda}@ertl.jp
**Graduate School of Information Science and Technology, The University of Tokyo
7-3-1 Hongo, Bunkyo-ku, Tokyo 113-8656, Japan
E-mail: shinpei@pf.is.s.u-tokyo.ac.jp

[Received October 21, 2019; accepted February 20, 2020]

**Clustering is the task of dividing an input dataset into groups of objects based on their similarity. This process is frequently required in many applications. However, it is computationally expensive when running on traditional CPUs due to the large number of connections and objects the system needs to inspect. In this paper, we investigate the use of NVIDIA graphics processing units and their programming platform CUDA in the acceleration of the Euclidean clustering (EC) process in autonomous driving systems. We propose GPU-accelerated algorithms for the EC problem on point cloud datasets, optimization strategies, and discuss implementation issues of each method. Our experiments show that our solution outperforms the CPU algorithm with speedup rates up to 87X on real-world datasets.**

**Keywords:** Euclidean clustering, GPGPU, point cloud, autonomous driving systems

## 1. Introduction

A point cloud is a data structure containing a large number of 3D points. These can be obtained using 3D scanners like LiDAR [a]. Point clouds are important resources for autonomous driving systems, as they contain information on the driving environment such as locations of obstacles or road signs. However, processing unorganized point clouds is computing-intensive due to the number of points. To reduce the time to process point clouds, we can use a method called clustering. This method divides a point cloud into distinct groups of points called clusters based on their similarities. In this paper, we use the Euclidean distance to determine the similarities: when the Euclidean distance between two points is less than a threshold, they belong to the same cluster. **Fig. 1** shows a result of the Euclidean clustering (EC) on a point cloud. This EC plays an important role in various object detection applications, but it is also a computing-intensive task. Its long execution time is not appropriate for current autonomous driving systems since modern 3D scanners used by those systems generate high-resolution point clouds at high frequencies. Failing to perform the clustering on them quickly may result in errors.

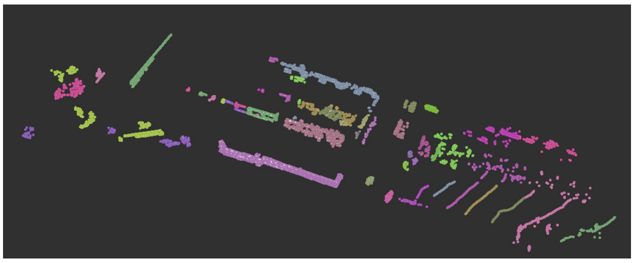


**Fig. 1.** The Euclidean clustering on the point cloud dataset.

The EC problem described above can be solved by a graphical approach. In this approach, we represent a point cloud as a graph in which every point is a vertex and edges connect two vertices with distances less than the threshold. The EC problem then becomes identifying all connected components of the graph. This is known as the connected-component labeling (CCL) problem. The CCL problem can be solved by data-parallel algorithms such as the ones proposed by Hawick et al. [1] or Komura [2].

In this paper, we study the use of NVIDIA graphics processing units (GPUs) and their programming platform named Compute Unified Device Architecture (CUDA) [b] that converts point clouds into graphs and uses them to perform a fast EC on the point cloud dataset. Our approach leverages features of NVIDIA GPUs including high memory bandwidth and fast on-chip memory to accelerate the clustering process. We also conducted experiments to evaluate the performance of our solutions on our synthetic datasets and real-world datasets.

In summary, our paper makes the following contributions.

1. We present designs and implementations GPU-accelerated algorithms for the EC problem on point cloud datasets.

2. Our algorithms' performance is compared with each other and with the algorithm implemented in the Point Cloud Library (PCL) via experiments on synthetic and real-world datasets.

3. We also analyze the impact of factors of a point cloud on the performance of our algorithms.
4. Based on experimental results, we discuss the potential and the remaining problems of the proposed algorithms.

The rest of this paper is organized as follows: Section 2 reviews previous approaches to the EC problem we are working on. Section 3 describes the CPU-GPU system used in this paper. Our approaches are explained in Section 4 while experiments are described in Section 5. Section 6 discusses lessons we learned from this work. Finally, the conclusion is presented in Section 7.

## 2. Related Work

The problem we described in Section 1 is a subproblem of the general 3D point cloud segmentation problem. It has been studied extensively over decades, resulting in the development of various segmentation techniques such as [3–6]. These techniques target specific cases of the clustering problem and attempt to extract cluster information from point clouds efficiently and precisely with minimum requirements of the input parameters. Other methods focus on the clustering process in systems with low computational demands. For example, the range image-based segmentation technique proposed by Bogoslavskyi et al. [7] or Moosmann et al. [8] creates a range image from the input 3D point cloud and segments pixels of the image. However, their applications are limited to specific cases due to assumptions about the structure of the input point cloud.

The presence of the threshold Euclidean distance $d_{th}$ allows us to use scan line [9] to solve the EC problem. This algorithm propagates the label of each point to its neighbors using a process similar to the breadth-first search (BFS). The key point of this algorithm is the assumption that the input point cloud is composed of elliptical layers. Under that assumption, the neighbor search is performed by examining points from consecutive layers, rather than the whole point cloud. Nevertheless, in general, we have no idea about the structure of the input point cloud. If the point cloud is not composed of elliptical layers, scan line will not work.

General point clouds may require clustering techniques that make *no assumption* about their inner structure. DBSCAN [10] appears to be a good choice in such cases. This algorithm also runs a BFS-like process to propagate the label of each point to its neighbors. The neighbor search is performed by an R$^*$-tree [11]. DBSCAN detects noise by examining the number of a point's neighbors. This algorithm has the computational complexity $O(n \times \log(n))$, and it has demonstrated its efficiency in discovering clusters with arbitrary shapes from large datasets. The Euclidean cluster extraction (ECE) algorithm presented by Rusu [12] is similar to DBSCAN. The major difference is that ECE, unlike DBSCAN, does not identify noise during the label propagation step. Instead, clusters that have less than a specific number of points are filtered out *after* that step. Besides, this algorithm uses a kd-tree [13] to organize points and perform the neighbor search. ECE has the computational complexity $O(n \times \log(n))$, and it is implemented in the PCL library [14], one of the most popular libraries for processing point clouds. Despite being efficient and simple, ECE executes poorly even on high-end machines. A GPU-based version of ECE is implemented in the same library by Buys and Rusu as a class named pcl::gpu::EuclideanClusterExtraction. This class uses a GPU-based octree to accelerate the neighbor search step of ECE but leaves the rest of the clustering to the CPU. The code, however, is incomplete. It contains runtime errors and its performance is worse than its CPU counterparts. Besides, in many cases, its results are incorrect compared with the ones generated by the CPU alone.

In this paper, we propose several fast GPU-based approaches for the EC problem in general point clouds. Our techniques leverage features of modern GPUs to accelerate the clustering process. Additionally, we investigate the impact of the inner structure of point clouds on the performance of our algorithms.

## 3. The Architecture of NVIDIA GPUs and CUDA

GPUs are widely used as co-processors for CPUs. An NVIDIA GPU is composed of a scalable array of streaming multiprocessors (SM), which can execute a number of GPU threads concurrently. GPU threads are organized into warps, blocks, and grids. Each warp contains 32 GPU threads on the same SM. Multiple warps form a block and multiple blocks form a grid. There are several types of GPU memory, but in this paper we focus on three major types: the global memory, the shared memory, and the registers. The global memory has the largest capacity and is accessible for all GPU threads. Though it also has the highest latency, the access to the global memory can be enhanced by the *coalesce constraint*: when threads in a warp access addresses that fall within the same aligned transaction, their accesses are grouped into one transaction and the throughput is increased. The shared memory is an on-chip memory with higher bandwidth and lower latency memory than the global memory. However, it is small and only available for threads in the same block. Finally, the register is the fastest GPU memory type, but its numbers per block are limited and cannot be shared among different threads.

To harness the computing power of the NVIDIA GPUs, developers may use CUDA, a parallel computing platform and programming model for NVIDIA GPUs. A program written in CUDA has two parts: the host part executed by the CPU, and the kernel part handled by the GPU. The host side allocates buffers on the GPU global memory for the input and output data and launches GPU kernels to request the GPU to do specific tasks. Blocks of threads are distributed to available SMs so GPU threads in each block

**Table 1.** Notation list.

| Notation | Description |
|---|---|
| $P$ | The array of input points |
| $C$ | The array storing labels of points |
| $d_{th}$ | The threshold of the EC clustering |
| $d(p,q)$ | The distance between point $p$ and $q$ |
| $\lvert S \rvert$ | The number of elements in a set $S$ |

**Algorithm 1** Building an adjacency list using the GPU.

```
Allocate the integer array $S$ of $|P|$ entries.
Launch a GPU kernel of $n$ threads. Thread $i$ counts the number of neighbors of the point $P[i]$ and writes that number to $S[i]$.
Perform an exclusive scan on $S$.
Allocate the integer array $N$ of $S[n]$ entries.
Launch a GPU kernel of $n$ threads. Thread $i$ write the indices of neighbors of the point $P[i]$ to $N$ at locations starting at $S[i]$.
```

run concurrently to finish the task. Input/output data can be made accessible to the GPU/CPU via explicit memory copies between the host and the GPU memory, or via direct access with the unified memory. CUDA also provides atomic functions to prevent GPU threads from race conditions when they are working on the same GPU memory area.

## 4. Approaches

In this section, we explain our three approaches for the EC problem using NVIDIA GPUs and CUDA C: vertex-based, edge-based, and matrix-based. Here we assume the GPU memory is large enough to hold both the point cloud and data structures to be created. All three methods take a point cloud $P$ composed of $n$ points as an input, and output an array $C$ containing labels of every point from the input. $P$ is copied to the GPU global memory in advance. Initially, each point is given a unique label, which is equal to the index of the point in $P$, i.e., $C[i] = i$.

**Table 1** lists all frequently used notations through this section.

### 4.1. The Vertex-Based Method

In this method, we first build an undirected graph in the form of an adjacency list from the point cloud. Thereafter, we propagate the labels of each point to all of its neighbors.

#### 4.1.1. Data Structures

We use two integer arrays for representing the adjacency list: the starting locations of neighbor lists $S$ and the indices of neighbors $N$. The indices of neighbors of a vertex $i$ can be accessed by traversing from $N[S[i]]$ to $N[S[i+1]-1]$.

#### 4.1.2. Algorithms

The first stage, building the adjacency list, is described in **Algorithm 1**.

The exclusive scan [15] on $S$ produces an array containing the writing location of each GPU thread in the neighbor list $N$. It also determines the number of entries in $N$ for the host side to allocate enough GPU memory for the output of this step.

We use the vertex-based CCL algorithm [1] for the second stage. This algorithm propagates labels of vertices in the graph to their neighbors *in parallel*. To avoid conflict when multiple GPU threads try to update the label of one point, this algorithm uses the CUDA's *atomicMin* function. This function ensures that all points receive the smallest possible label. It also helps GPU threads avoid the circular label assignment, in which two neighbors A and B keep exchanging their labels back and forth.

#### 4.1.3. Implementation and Optimization

We use the exclusive scan provided by Thrust [c] for the first stage. For the second stage, the parallel vertex-based CCL algorithm uses a variable $m_d$ located in the global memory to record the state of the clustering. To prevent GPU threads from accessing $m_d$ frequently, we use two more variables $m_t$ and $m_b$. $m_t$ is a local variable defined by every thread while $m_b$ is located in the shared memory. Both of them are set to *false* prior to the loop on the adjacency list. If a thread detects changes in labels, it sets $m_t$ to *true*. After that, if a thread has $m_t = true$, it sets $m_b$ to *true*. At the end of the kernel execution, $m_d$ is only changed to *true* by the thread 0 in each block when it finds that $m_b = true$. Coalesced access is used intensively to maximize global memory bandwidth.

#### 4.1.4. Implementation Issues

First, CUDA's *atomicMin* may serialize the accesses to the label of the same vertex and slow down the clustering process. Second, accesses to arrays $N$ and $C$ underutilize the global memory bandwidth since they are non-coalesced. Third, the workload among GPU threads may not be balanced if the sizes of the vertices' adjacency lists differ. In such cases, some of the GPU threads may conclude their work earlier than others do and are wasted during the rest of the kernel. Finally, branch divergence may occur when threads in a block follow different branches when comparing the label of a vertex with the labels of its neighbors.

### 4.2. The Edge-Based Method

This method is also composed of two stages: building an edge-set from the point cloud and the clustering being run on the edge-set.

#### 4.2.1. Data Structures

The edge set $E$ is an array each element of which is a pair of indexes of points that are neighbors to each other.

We use an array of CUDA's *int*2 for $E$ in our implementation. *int*2 is a vector data structure of CUDA derived from the basic integer type. Its components are aligned and are accessible via the fields $x$ and $y$. We use it to store the indices of two vertices of each edge during the clustering.

#### 4.2.2. Algorithms

The algorithm for the first stage of this method is similar to **Algorithm 1** with the following changes.

- In steps 2 and 5, only neighbors whose indices are larger than $i$ are counted. Others are ignored.
- In step 4, an array $E$ of type *int*2 instead of type *int* is allocated in the GPU memory.
- In the final step, each GPU thread writes both the index of the point it is tasked with and the neighbor of that point to $E$.

In the second stage, we apply the edge-based CCL algorithm [1] to adjust labels of points in parallel. Again, the CUDA's *atomicMin* function is used by GPU threads to avoid circular label assignments and conflicts between GPU threads. The algorithm ends when the system records no changes in the labels anymore.

#### 4.2.3. Implementation and Optimization

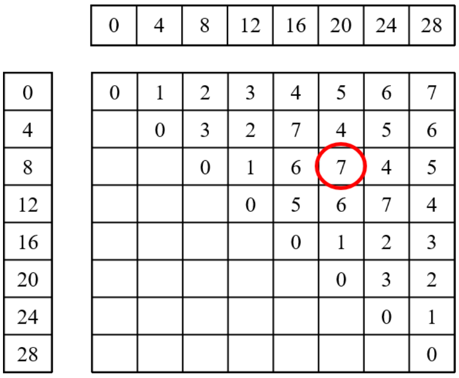
We use the same technique as the vertex-based method with the shared variable $m_b$ to avoid frequent access to the GPU global memory during the clustering stage. Besides, accesses to the point cloud $P$ in the first stage and the edge set $E$ in the second stage are coalesced again.

#### 4.2.4. Implementation Issues

First, access to the label array $C$ in the second stage is non-coalesced and may reduce the bandwidth of the global memory. Second, similar to the vertex-based method, the *atomicMin* may serialize operations of a large number of GPU threads and cause performance degradation.

### 4.3. The Matrix-Based Method

Both the previous two methods face a similar problem: the performance degradation caused by the *atomicMin* function. In the matrix-based approach, we aim to reduce the effect of atomic operations on the overall performance by isolating the clustering process among GPU thread blocks. To that end, we do not try to propagate the label of each point to its neighbors as do the two previous methods. Instead, we *merge* cluster labels as follows: if two clusters $C_0$ and $C_1$ have two points $c_0 \in C_0$ and $c_1 \in C_1$ and $d(c_0, c_1) < d_{th}$, then the labels of all points in one cluster are changed to the label of the other. We perform the clustering by repeatedly assigning GPU thread blocks to merge labels until no labels can be merged anymore. An adjacency matrix $M$ is used to track the relationship of labels during this process. $M(i, j) = 1$ means the clusters $i$ and $j$ are mergeable and 0 means they are not.



|  | 0 | 4 | 8 | 12 | 16 | 20 | 24 | 28 |
|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 4 |  | 0 | 3 | 2 | 7 | 4 | 5 | 6 |
| 8 |  |  | 0 | 1 | 6 | 7 | 4 | 5 |
| 12 |  |  |  | 0 | 5 | 6 | 7 | 4 |
| 16 |  |  |  |  | 0 | 1 | 2 | 3 |
| 20 |  |  |  |  |  | 0 | 3 | 2 |
| 24 |  |  |  |  |  |  | 0 | 1 |
| 28 |  |  |  |  |  |  |  | 0 |

**Fig. 2.** An example of matrix partition when $b = 4$ and $|R| = 32$. Matrix $32 \times 32$ $M$ is divided into $4 \times 4$ sub-matrices. Each sub-matrix describes the relationship between labels of two groups or between labels in one group in the case the sub-matrix is on the main diagonal of $M$. For instance, the $(8, 20)$ sub-matrix (the one in the cycle) describes the relationship between labels of the groups 8 and 20. These sub-matrices are grouped into eight layers. The number in each sub-matrix displays its layer index value.

To isolate the merging process among GPU blocks, we divide the label list into small non-overlapped groups of $b$ consecutive labels. Each group is identified by the index $g$ of the initial label of the group in the label list. The relationship between labels in two groups $g_0$ and $g_1$ is determined by a $b \times b$ sub-matrix starting from row $g_0$ and column $g_1$ of the matrix $M$. Sub-matrices are further grouped into non-overlapped layers such that sub-matrices in one layer share no common label. Layers are then evaluated one by one by the GPU. **Fig. 2** is an example of this partition method on a $32 \times 32$ adjacency matrix.

#### 4.3.1. Data Structures

We use the following data structures in the GPU memory in addition to the arrays $P$ and $C$.

- $R$: the array of remaining labels.
- $M$: the adjacency matrix of labels in $R$. $M(i, j) = 1$ if labels $R[i]$ and $R[j]$ are mergeable and 0 otherwise.
- $L$: the array containing locations of labels in $R$. The index of a label $c$ in $R$ is $L[c]$.

#### 4.3.2. Algorithms

**Algorithm 2** provides an overview of the matrix-based EC.

Initially, the number of labels is $|P|$. The matrix $M$ created on those labels may exceed the capacity of the GPU global memory. To reduce the size of the matrix $M$, we first merge contiguous points in $P$. **Algorithm 3** explains this step in detail.

The *status* array in the GPU shared memory is for tracking the status of all labels in each iteration. It is reset to

**Algorithm 2** The matrix-based EC on the GPU.

```
1: Run procedure INITIAL_EC.
2: Run procedure BUILD_MATRIX.
3: Define $m_d := true$.
4: While $m_d = true$, set $m_d := false$ and perform steps 5–9.
5: while $m_d = false$, perform steps 6–9.
6: Evaluate a layer of the adjacency matrix $M$.
7: If $m_d = true$, run procedure UPDATE and go back to step 4.
8: If $m_d = false$ and the current layer is not the final one, move
   to the next layer and go back to step 7.
9: if $m_d = false$ and the current layer is the final one, copy the
   result back to the host memory and conclude.
```

**Algorithm 3** The procedure *INITIAL_EC*. The local index of a thread in a GPU block and the index of a block are denoted by *tid* and *bid*, respectively. In our implementation with CUDA, we use a 1D grid of 1D blocks, so $tid = threadIdx.x$ and $bid = blockIdx.x$.

```
1: Launch a GPU kernel of $n/b$ blocks of $b$ threads each. Each
   thread performs steps 2–12.
2: Get the coordinates of the point $p := P[tid + bid \times b]$.
3: Define integer arrays $sp$, $labels$, $status$ in the shared memory.
   Each array has $b$ elements.
4: Pre-load points to the shared memory $sp[tid] := p$.
5: Initialize local labels $labels[tid] := tid$.
6: For $j$ from 0 to $b-2$, perform steps 7–11.
7: Zero-out $status$ by $status[tid] := 0$.
8: Get the coordinates of the current point $q := sp[j]$.
9: Get the current labels of points $cc := labels[tid]$ and $rc :=
   labels[j]$.
10: If $tid > j$ and $d(p,q) < d_{th}$ and $rc \neq cc$, set $status[cc] := 1$.
11: If $status[cc] = 1$, set $labels[tid] := rc$.
12: Update new label of the point by $C[i] := bid \times b + labels[tid]$.
```

**Algorithm 4** The procedure *BUILD_MATRIX*.

```
1: Allocate an integer array $L$ of $n+1$ elements. Zero out $L$.
2: Launch a GPU kernel of $n$ threads. Thread $i$ marks $L[C[i]] :=
   1$.
3: Perform an exclusive scan on $L$.
4: Allocate $R$ as an integer array of $L[n]$ elements.
5: Launch a GPU kernel of $L[n]$ threads. Thread $i$ sets $R[i] := i$.
6: Launch a GPU kernel of $n$ threads. Thread $i$ updates $C[i] :=
   R[C[L[i]]]$.
7: Allocate an integer array $M$ of size $|R| \times |R|$ in the GPU mem-
   ory. Zero-out $M$.
8: Launch a GPU kernel of $n$ threads. Thread $i$ searches for all
   neighbors $P[j]$ of the point $P[i]$ and set $M[C[i] + C[j] \times |R|] :=
   1$.
```

**Algorithm 5** Merging column labels to row labels of a sub-matrix $(g_0, g_1)$ by a GPU block of $b$ threads. The local index of thread in the block is denoted by $i$. In our implementation with CUDA C, we use 1D block so $i = threadIdx.x$.

```
1: Threads in the block define an integer array $status$ of $b$ ele-
   ments in the shared memory.
2: Define local variables $cl := i$ and $c := g_1 + i$.
3: Define local variable $m_t$ and shared variable $m_b$. All of them
   are initially set to $false$.
4: Loop $b-1$ times. In the $j$-th loop, thread $i$ performs steps 5–
   8.
5: Define $r := g_0 + j$.
6: Zero-out $status[i] := 0$ and $status[i+b] := 0$.
7: If $M[r \times |R| + c] = 1$, set $status[cl] := 1$ and $m_t := true$.
8: If $status[cl] = 1$, change $cl := i + b$.
9: If $m_t = true$, perform steps 10–12.
10: Update the new label $R[c] := r + cl - b$.
11: Set $m_b := true$.
12: If $i = 0$ and $m_b = true$, set $m_d := true$.
```

*unchanged* at the beginning of each iteration. The function *__syncthreads*() must be called after every read/write operation on arrays. This ensures that labels and the status of labels are always correctly updated by all threads at the end of each iteration. The condition $tid > j$ is to avoid circular label changes.

The procedure *BUILD_MATRIX* creates the remaining label list $R$, updates the label list $C$, and builds the matrix $M$. It is described in **Algorithm 4**.

After obtaining the remaining label list $R$ and the matrix $M$, the system starts iterating over the layers of $M$ and merges column labels with row labels of all sub-matrices in each layer. Merging labels in layer 0 is similar to the initial EC in which the input column labels and the row labels are the same. However, instead of computing the pair-wise distances of points, threads access sub-matrices on the main diagonal of $M$ to obtain information about the relationship between labels. On other layers, the process is slightly different because the column labels and row labels of each sub-matrix are not the same. **Algorithm 5** explains how threads in a GPU block evaluate a sub-matrix in such layers. Again, the function *__syncthreads*() is called after every read/write operation of threads to make sure labels are updated correctly. **Fig. 3** shows an example of this algorithm.

If labels are merged, the procedure *UPDATE* rebuilds arrays $R$, $L$, $C$, and $M$. In this procedure, duplicated labels are removed for the GPU blocks to merge the remaining unique labels without conflicts. This procedure is explained in **Algorithm 6**. We repeat the entire process until no further change in the remaining label list $R$ is made.

#### 4.3.3. Implementation and Optimization

We use the parallel exclusive scan and sort provided by Thrust again. The shared memory and registers are used temporarily to store points' coordinates and intermediate labels while labels are being merged. By doing so, GPU threads do not have to access the global memory when computing pairwise distances or when changing labels of points in a group. Besides, accesses to adjacency sub-matrices by threads in a block as well as to point coordinates on $P$ are coalesced to maximize the GPU global memory bandwidth.

#### 4.3.4. Implementation Issues

This method requires significant GPU memory for holding the adjacency matrix $M$. If points in the cloud are far from each other, most of the matrix's entries are 0 and a large amount of GPU global memory is wasted. Besides, with a large graph, the GPU memory may not be sufficient to contain the entire matrix.

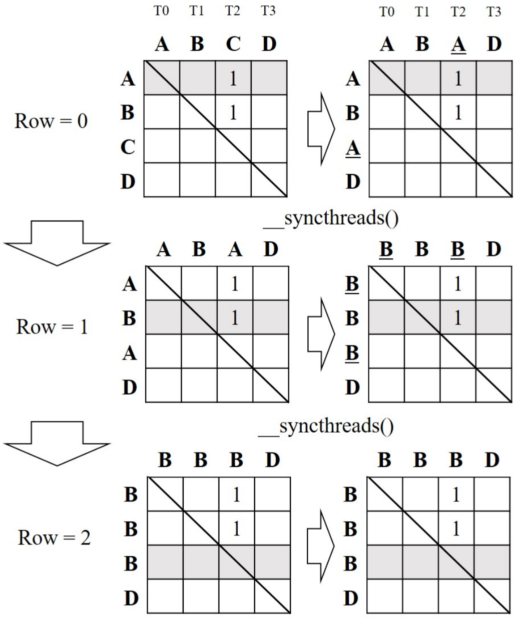


**Fig. 3.** Clustering on a 4 × 4 sub-matrix using a GPU block of 4 threads. Thread $T_i$ handles the label of column $i$. After passing one row, a *__syncthreads*() is called to make sure all threads in the block finish changing labels before moving to the next row.

**Algorithm 6** The procedure *UPDATE* rebuilds $R$, $C$, $L$, and $M$ on the GPU. The global index of threads is denoted by $i$. In our implementation with CUDA C, $i = threadIdx.x + blockIdx.x \times blockDim.x$.

```
1: Launch a GPU kernel of $n$ threads. Thread $i$ updates the label of point $i$-th by performing $C[i] := R[L[C[i]]]$.
2: Zero-out $L$.
3: Launch a GPU kernel of $|R|$ threads. Thread $i$ sets $L[R[i]] := 1$.
4: Perform an exclusive scan on $L$.
5: Allocate the new remaining list $R'$ of $L[n]$ integer elements.
6: Launch a GPU kernel of $|R'|$ threads. Thread $i$ sets $R'[i] := i$.
7: Launch a GPU kernel of $n$ threads. Thread $i$ updates label $C[i] := L[C[i]]$.
8: Allocate the new adjacency matrix $M'$ of $|R'| \times |R'|$ integer entries. Zero-out this matrix.
9: Launch a GPU kernel of $|R|$ threads. Thread $i$ performs steps 10–13.
10: Get the column index $c := i$ and the new column index of the new column label $nc := L[R[c]]$.
11: For $r$ from 0 to $c-1$, perform steps 12–13.
12:     Get the new row index of the new row label $nr := L[R[r]]$.
13:     If $M[r \times |R| + c] = 1$, set $M'[nr \times |R'| + nc] := 1$ and $M'[nc \times |R'| + nr] := 1$.
14: Replace $R$ by $R'$ and $M$ by $M'$ and free old buffers.
```

### 4.4. Range Search and Exhaustive Search

A key step in all three methods is searching for the neighbors of every point when building graph structures. This can be done by exhaustive searches, in which each GPU thread computes the Euclidean distance between one point and all other points, and compares them with $d_{th}$. This approach can be optimized by using the GPU's shared memory to store the coordinates of points handled by threads in the same block. However, the number of comparisons in that case is still $n^2$; thus, it decreases the overall performance on a large point cloud.

**Algorithm 7** Building a grid of voxel on the point cloud $P$ using the GPU.

```
1: By parallel reduction, determine the boundaries of the box that contains the entire point cloud.
2: Determine the number of required voxels.
3: For each point in $P$ in parallel, compute the index of the voxel that the point belongs to. The results are stored in an array $vid$.
4: Allocate an array $pid$ of $|P|$ elements containing the index of points in $P$.
5: Use Thrust's parallel sort to sort the key $vid$ and value $pid$.
6: Allocate an integer array of $|P|+1$ elements named $mark$.
7: For $i$ from 0 to $|P|-1$ in parallel, if $vid[i] < vid[i+1]$, set $mark[i] := 1$.
8: Exclusive scan $mark$. The value $k$ of the last element $mark[|P|]$ is the number of non-empty voxels.
9: Allocate two integer arrays: $V$ of $k$ elements and $S$ of $k+1$ elements.
10: For $i$ from 0 to $|P|-1$ in parallel, if $vid[i] < vid[i+1]$, set $V[mark[i]] := vid[i]$ and $S[mark[i]+1] := i+1$.
```

A better option for the neighbor search in large point clouds is using range search on a voxel grid. In this approach, we first use a grid of $d_{th} \times d_{th} \times d_{th}$ cubes to organize the point cloud. **Algorithm 7** explains a simple method for building the grid.

This algorithm produces three arrays: $pid$ – the indexes of points sorted by voxel indexes, $V$ – the indexes of non-empty voxels, and $S$ – the starting location of indexes of points that belong to each voxel. We can obtain indexes of points in a voxel $i$ by iterating from $pid[S[i]]$ to $pid[S[i+1]-1]$. To look for neighbors of a point $p$, we inspect points in voxels included by the sphere that takes $p$ as the origin and $d_{th}$ as the radius. Points that belong to other voxels may be excluded, thereby reducing significantly the number of comparisons.

## 5. Evaluation

### 5.1. System Configuration and Datasets

We now evaluate the performance of the PCL's CPU-based method, the exhaustive search (ES) GPU-based methods, and the range search (RS) GPU-based methods. As mentioned in Section 2, the GPU-based ECE method implemented in the PCL library is incomplete. It caused runtime errors when processing our synthetic datasets, and produces results different from other methods. Hence, we did not use that method in our experiments. The specifications of the PC used in our experiments are as follows: CPU Intel Core i7-4790 3.6 GHz × 8, 16 GB RAM, GPU NVIDIA GTX Titan X Maxwell with max clock rate 1.08 GHz and 12 GB GDDR5 memory, CUDA Driver / Runtime Version 9.1, and Ubuntu 16.04 LTS OS. Evaluations are performed on our synthetic dataset and real-world dataset assembled during ex-

**Table 2.** Values of factors for generating synthetic point clouds.

| Experiment | Size | Cluster number | Point degree | Point distance |
|---|---|---|---|---|
| Size variation | 4096 to 262144 | 128 | 32 | 4 |
| Cluster number variation | 262144 | 16 to 8192 | 32 | 4 |
| Point degree variation | 262144 | 128 | 1 to 2048 | 4 |
| Point distance variation | 65536 | 1024 | 32 | 1 to 1024 |

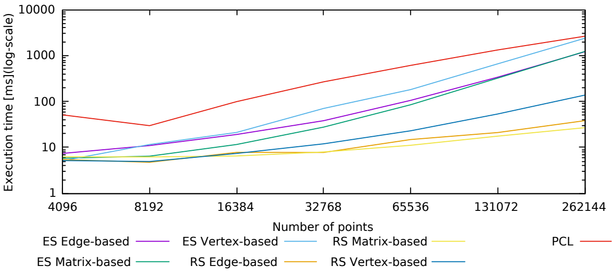


**Fig. 4.** Execution time of clustering methods in the size-variation test (ES: exhaustive search, RS: range search).

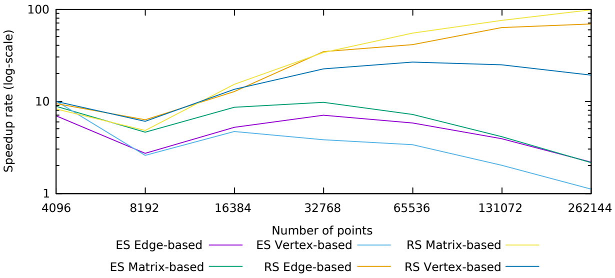


**Fig. 5.** Speedup rate of the GPU methods over the PCL library's method in the size-variation test (ES: exhaustive search, RS: range search).

periments of the Autoware project [d]. We generate our synthetic point clouds using four factors as follows.

- *Size*: the number of points in the point cloud.
- *Cluster number*: the number of clusters in the point-cloud. This must not exceed the size of the point cloud.
- *Average point degree*: the average number of neighbors of a point. The product of the point degree and the cluster number must not exceed the size of the point cloud.
- *Point distance*: this value is to adjust the distribution of clusters' members in the input point cloud. A point cloud with point distance $D$ has points at indices $i, i+D, i+2\times D$ that belong to the $i$-th cluster. Due to this arrangement, the point distance must not exceed the cluster number.

The values of the four factors above in each experiment are listed in **Table 2**. The execution time is measured in milliseconds. It includes both the CPU and the GPU processing times and the time spent on transferring data between memory areas. The PCL's CPU-based method is used as a baseline method to evaluate other methods. Hence, their speedup rates are defined as the ratio of the CPU-based method's execution time to that of the GPU-based methods.

## 5.2. Experiment Results

### 5.2.1. Size of the Point Cloud

First, we study the effect of the size of the point cloud on clustering methods. The results of this experiment are shown in **Figs. 4** and **5**.

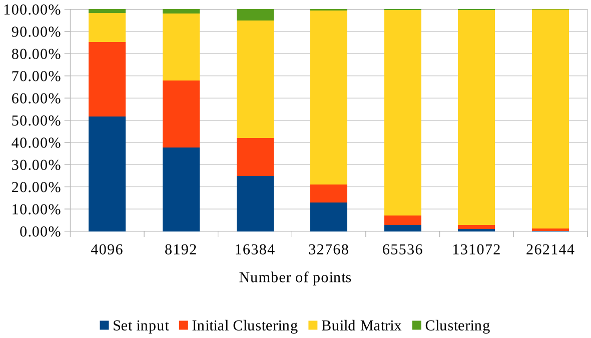


**Fig. 6.** Breakdown of execution time of the ES matrix-based method.

In all cases, the GPU-based methods' execution times are less than those of the PCL method. However, the difference between ES methods and the PCL's method is less significant when the point cloud becomes larger. We can observe a considerable decrease in the speedup rates of those methods in **Fig. 5** as the cloud has more than 32768 points. This is explained by the computation cost of the exhaustive search. Due to the large number of pair-wise distances examined, this cost increases quickly as the number of points increases. For instance, we can observe this change of the GPU ES matrix-based method from **Fig. 6**. In the largest point cloud, the build matrix stage takes about 99.9% of the total execution time of this method. RS methods use range search instead of exhaustive search, so the speedup rates of the RS edge and matrix-based methods keep increasing even in later cases. An exception is the RS vertex-based method. Its speedup rate also decreases at some points. This is due to the huge global memory access in the clustering phase

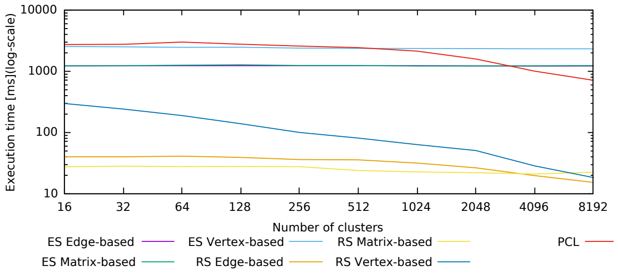


**Fig. 7.** The execution time of clustering methods as the cluster number changes (ES: exhaustive search, RS: range search).

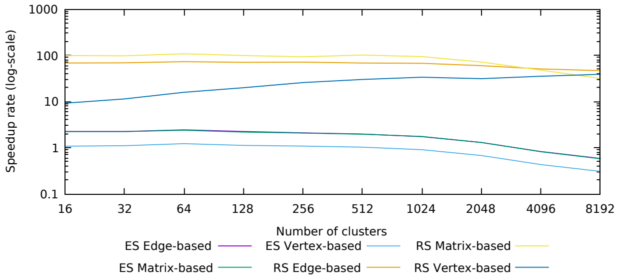


**Fig. 8.** The speedup rate of clustering methods as the cluster number changes (ES: exhaustive search, RS: range search).

of this method: when examining an edge $e$, GPU threads have to inspect vertices of $e$ twice, while it is only once for the two other RS methods.

### 5.2.2. Number of Clusters

In this experiment, we inspect how the execution time of clustering methods changes when the number of clusters changes. The results are shown in **Figs. 7** and **8**.

When the number of clusters increases while the size of the point cloud remains unchanged, the number of members in each cluster decreases. In such cases, all clustering methods need less time for building data structures and adjusting labels. Therefore, all of their execution times tend to decrease. However, the changes when using the ES methods are minimal. This is because about 90% of their execution times are for building graph structures, and the cost of that step remains similar as the size of the point cloud is fixed. In the two final cases, their speedup rates over the PCL method are less than 1, which means they are even slower than the PCL method.

### 5.2.3. Point Degree

**Figure 9** shows the execution times of the clustering methods when the point degree changes from 1 to 2048.

The execution time of the PCL's clustering increases rapidly as the point degree increases. This is because the cost of searching neighbors increases as the point degree increases. The ES methods' results remain almost unchanged due to the same reason to the previous experiment.

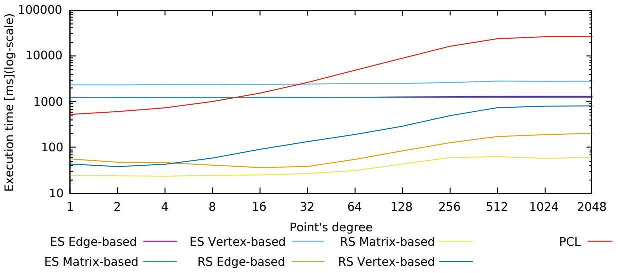
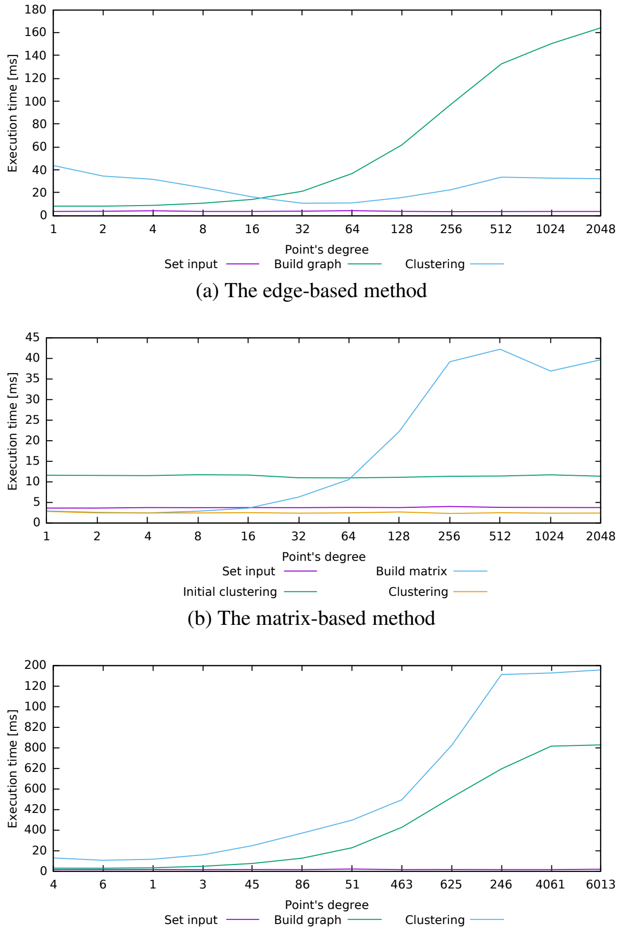


**Fig. 9.** The execution time of clustering methods as the point degree changes (ES: exhaustive search, RS: range search).



(a) The edge-based method

(b) The matrix-based method

(c) The vertex-based method

**Fig. 10.** The execution time of each stage in RS GPU-based methods.

**Figure 10** details how the execution time of each stage in RS methods change. In the building step of all three methods, a larger point degree increases the number of candidate points and the number of indices written to the output data structures. These result in longer execution times during the building stages of all RS methods. Conversely, the time differences in the clustering stage of the three methods are significant. In the edge-based method, because both the number of clusters and the size of the point cloud are fixed, each point can reach its final label in fewer iterations. However, that does not mean the clus-

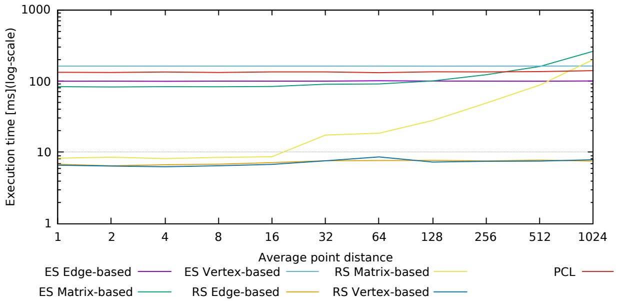


**Fig. 11.** The execution time of clustering methods as the average point distance changes (ES: exhaustive search, RS: range search).

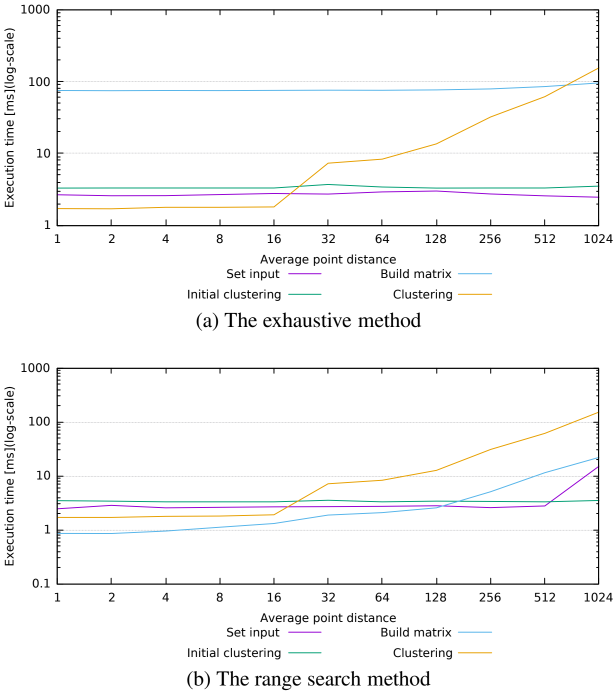
tering stage will be faster. The reasons behind that are more edges are processed and more time may be spent on each iteration due to the effect of *atomicMin*. In our experiment, this stage has the lowest execution time at the 32 and 64 point degrees. A similar occurrence is found when using the vertex-based method. Nonetheless, with this method points are examined by more GPU threads. Hence, its execution time at the clustering stage increases faster than other methods. Threads in different GPU blocks in the matrix-based method do not intervene in others' operations. Besides, global memory accesses in this method are accelerated by shared memory and coalesced constraint. For those reasons, its clustering stage does not change during the experiment.

### 5.2.4. Average Point Distance

We now keep the size of point cloud at only 65536 points and the cluster number at 1024. This allows us to experiment on a wider range of point distance values. A larger point cloud may cause an out-of-GPU-memory error as the point distance increases, and a small point distance cannot highlight the changes that occur in the execution times of matrix-based clustering methods. The execution times of the methods in this experiment are shown in **Fig. 11**.

All methods, except the matrix-based methods, have their results almost unchanged. **Fig. 12** shows the execution time of each stage in the matrix-based methods. While the set input and the initial clustering steps remain approximately the same between cases, the building matrix stage becomes slightly slower when the point distance is larger than 64 and the clustering stage increases rapidly from point distance 32 to 1024.

The increase of the step building matrix can be explained by the number of remaining clusters after the initial clustering step. After that step, all points that belong to the same cluster in a group are given the same label. The more labels remain, the larger the matrix becomes and the more time is spent on initializing it. As for the clustering step, its goal is to reduce the number of labels until they cannot be further reduced. If members in a cluster remain close to each other in the cluster list, a large number of labels can be removed with each iteration. In such cases, the clustering will finish after several iterations. Otherwise, the number of iterations increases, and each iteration lasts longer on the larger data structure. Those result in the longer clustering step. For instance, our RS matrix-based method needs only two iterations with 4.5 ms to finish the first point cloud, but requires 10 iterations and 156 ms for the last case. Besides, if too many labels remain after the initial step, the system may fail to allocate enough GPU memory for the matrix. Other methods do not have this problem because no matter how the points are distributed, their input data structures for the clustering step are still the same.



(a) The exhaustive method

(b) The range search method

**Fig. 12.** The execution time of stages in matrix-based methods in the point distance experiment.

### 5.2.5. Real-World Data

We now perform clustering methods on streaming data from real-world datasets. Those datasets are collected during the experiments of the project Autoware. LiDAR scanners were set on top of vehicles that move on specific routes. While moving, the scanners scan the environment around the vehicle and continuously produces point clouds in real time. Those point clouds are published as ROS [16] messages, and recorded into *bag* files. We then use our simulation application of project Autoware to play back those bag files and publish point clouds at the same rate as when they are collected by the LiDAR scanners. As our object detection module receives a point cloud, it invokes a callback function to perform the clustering and publishes the results. Hence, the point clouds are continuously copied from the host memory and the GPU memory, while the results are moved in the opposite direction. We run clustering methods on three datasets collected by the ZMP RoboCar (PHV) from Nagoya University in Nagoya City, Japan. Those datasets can be found at the Auto-

**Table 3.** Datasets used in the real-world experiment.

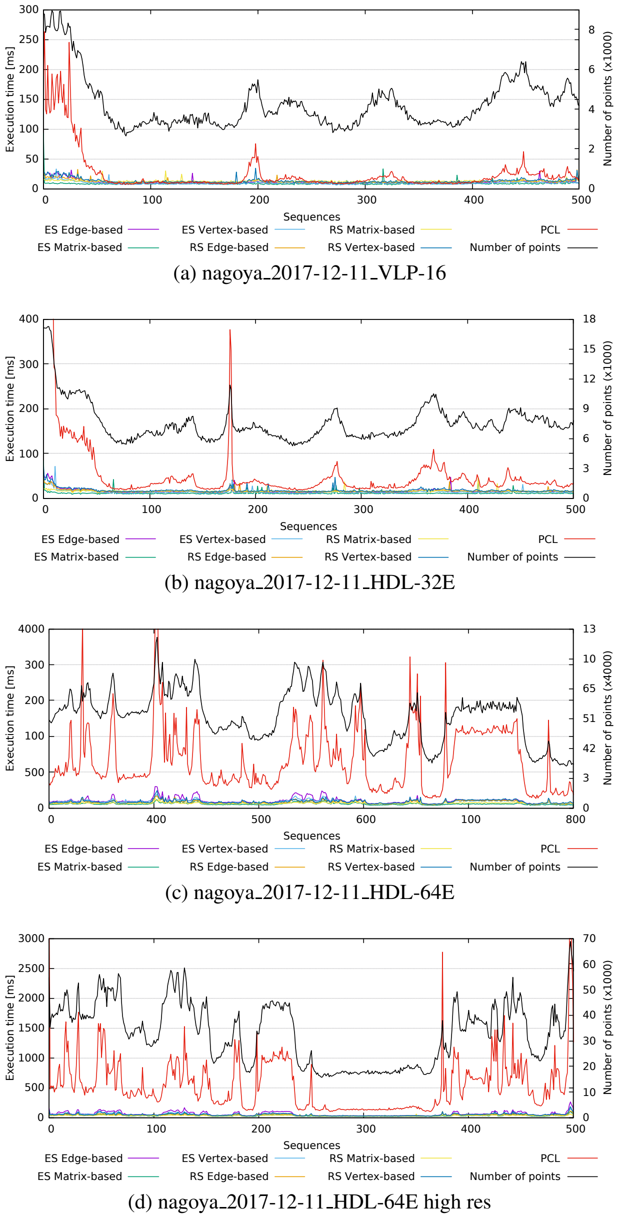
| Dataset | LiDAR type | Duration [s] | Average point number |
|---|---|---|---|
| nagoya_2017-12-11_VLP-16 | Velodyne VLP16 | 1116 | 3798 |
| nagoya_2017-12-11_HDL-32E | Velodyne HDL32E | 1131 | 6085 |
| nagoya_2017-12-11_HDL-64E | Velodyne HDL64E | 1331 | 21785 |
| nagoya_2017-12-11_HDL-64E high res | Velodyne HDL64E | 1331 | 34254 |

ware's rosbag store [e]. Their information is described in **Table 3**.

First, we evaluate the performance of clustering methods. The results are summarized in **Figs. 13** and **14**.

In the first two datasets, there is not much difference between clustering methods' performance in sequences with small numbers of points. In some cases, the PCL's method is even faster than others are. However, in later datasets when the number of points in each sequence increase, it cannot compete with other GPU-based clustering methods. Differently from experiments on synthetic datasets, the ES matrix-based method is the fastest method in the first and the second datasets with the average speedup rate 2.2X and 3.5X respectively. It is only surpassed by the RS matrix-based method in the last dataset, when the size of input point clouds becomes too large that make the building data step become significantly longer. Matrix-based methods have better performances than edge and vertex-based methods. This can be explained by the properties of the LiDAR scanner. It rotates at high speed and continuously projects beams that reflect from objects to create points. Hence, points that belong to the same object tend to stay close together in the point cloud. As a result, the number of labels is quickly reduced after each iteration of the matrix-based method and the clustering process ends quickly.

To evaluate the precision of our GPU-based method in each dataset, for each point cloud processed, we compared the number of extracted clusters as well as the number of points in each cluster between the GPU-based methods and the baseline PCL CPU-based method. The precision of a GPU method is defined as the number of matched point clouds over the total number of point clouds processed. In all four datasets, the result of our measurements are similar: the clusters extracted from each point cloud by the GPU methods are always the same as the ones extracted by the PCL CPU method.



(a) nagoya_2017-12-11_VLP-16

(b) nagoya_2017-12-11_HDL-32E

(c) nagoya_2017-12-11_HDL-64E

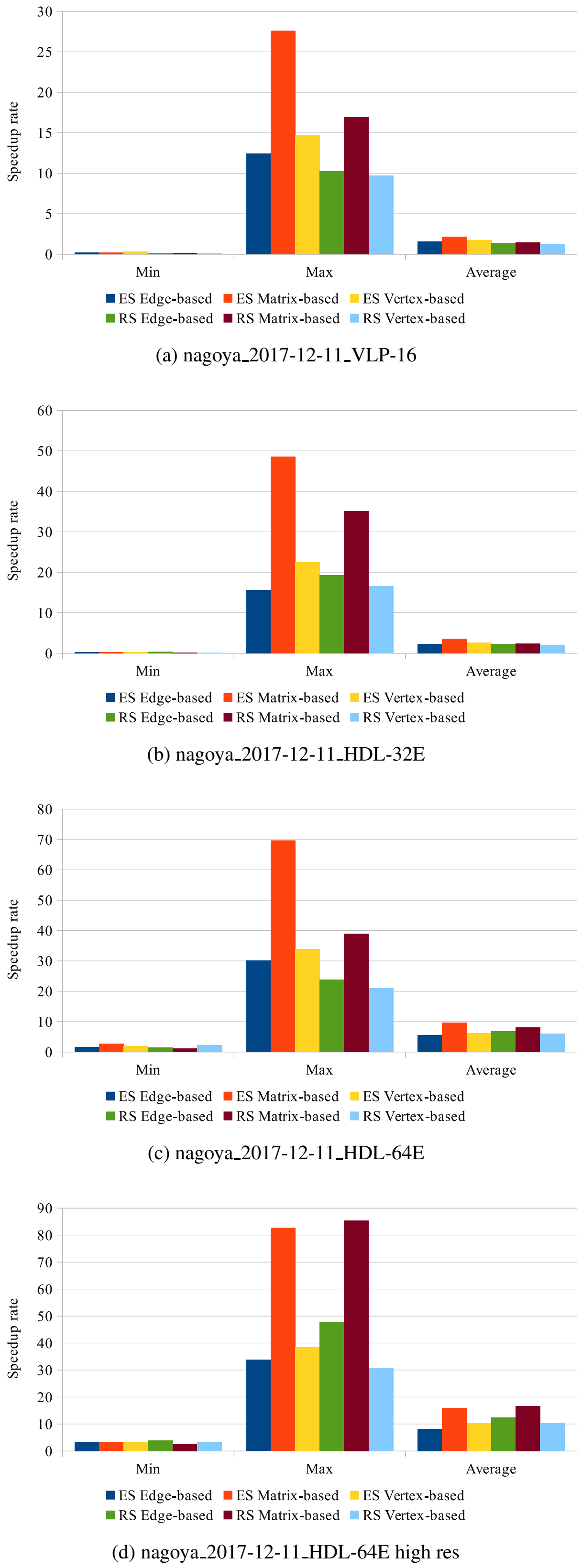
(d) nagoya_2017-12-11_HDL-64E high res

**Fig. 13.** The execution time of the clustering methods on real-world dataset (ES: exhaustive search, RS: range search).

## 6. Discussion

In this section, we discuss factors that affect the performance of our EC methods, their limitations, and opportunities.

To date, our GPU-based clustering methods have proved their efficiency by delivering significant speedup rates over the PCL's EC while maintaining precision. The edge and vertex-based methods use CUDA's *atomicMin* to propagate labels, but that function could serialize the operations of GPU threads. Hence, these methods' performance decreases significantly in point clouds which have high point degrees and a large number of members in each cluster. The matrix-based methods are more stable in such



(a) nagoya_2017-12-11_VLP-16

(b) nagoya_2017-12-11_HDL-32E

(c) nagoya_2017-12-11_HDL-64E

(d) nagoya_2017-12-11_HDL-64E high res

**Fig. 14.** The speedup rate of the GPU-based clustering methods over the PCL's on real-world dataset (ES: exhaustive search, RS: range search).

cases. They isolate work of GPU blocks, and leverage shared memory and synchronization among threads in a block to adjust labels. Nonetheless, their performance depends heavily on the distribution of points in the input. The further the average point distance is, the slower these methods become. Meanwhile, the edge and vertex-based methods are not affected by that factor.

The performance of clustering methods also depends on how graph data structures are built. This phase in ES methods is affected mostly by the size of the input point cloud as it decides the number of comparisons and global memory accesses. That factor has less effect on the same phase of the RS methods since these methods reduce the range of the neighbor search to a specific area around each point. Instead, the point degree alters the cost of this phase in RS methods. However, the difference is not always significant, especially in small point clouds with less than eight thousand points. For those reasons, estimating the computational complexity of GPU-based methods is challenging. The results obtained from the experiment on real-world datasets of Section 5.2 suggest an average computational complexity of $O(k \times \log(n))$ for RS methods, in which $k$ is much smaller than $n$. This may be applied to general point clouds collected from the real-world by LiDARs. As for ES methods, the experiment on size variation in Section 5.2 found that, on large point clouds, most of the ES methods' execution time is consumed for building graph structures. The complexity of this stage is $O(n^2/p)$, in which $p$ is the number of GPU cores, since they examines $n^2$ pair-wise distances in parallel. Consequently, the average complexity of the ES methods should be $O(n^2/p)$ as well. Further investigations are required to validate those estimations.

The major limitation of the GPU-based methods is the memory cost for graph data structures. Among clustering methods, the matrix-based methods often require the largest memory space for the adjacency matrix. Besides, currently, our GPU-based methods do not support clouds that are larger than the GPU memory capacity. Therefore, it is necessary to investigate extending those methods to larger clouds to completely solve the fast clustering problem on point cloud datasets.

For the reasons above, the ES methods appear to be good choices for small clouds in the range of four thousand to eight thousand points. For larger clouds, RS methods are better options since building graph data structures in those methods are much faster than in the ES methods. In general, the edge-based and vertex-based methods are good options for point clouds that have low point degrees and small numbers of members per cluster. The edge-based methods require the smallest memory space, so they can fit on systems with limited GPU memory. The matrix-based methods are preferred for clouds with high point degrees, large-size clusters, and small average point distances, such as the one produced by the LiDAR scanner in our final experiment. However, their memory space requirement may become a burden for the GPU global memory. With clouds having less than four thousand points, PCL's ECE is the most efficient clustering method among the clustering methods. It is the only solution for the point clouds that exceed the GPU global memory capacity.

## 7. Conclusion

In this paper, we have presented our GPU-based clustering methods. They can provide a speedup rate up to 97X over the CPU-based clustering method implemented in the PCL on synthetic datasets, and up to 87X on a real-world dataset. We also discussed how the inner structure of a point cloud affects the performance of our algorithms. Despite having higher performance, their memory requirement prevents them from being applied in large point clouds. Finding how to enable these algorithms to run on point clouds that exceed the GPU memory is necessary to bring those algorithms to real-life systems.

**References:**

[1] K. A. Hawick, A. Leist, and D. P. Playne, "Parallel graph component labelling with GPUs and CUDA," J. Parallel Computing, Vol.36, Issue 12, pp. 655-678, 2010.

[2] Y. Komura, "GPU-based cluster-labeling algorithm without the use of conventional iteration: Application to the Swendsen-Wang multi-cluster spin flip algorithm," Compu. Phys. Comm., Vol.194, pp. 54-58, 2015.

[3] B. Douillard et al., "On the segmentation of 3D LIDAR point clouds," Proc. of 2011 IEEE Int. Conf. on Robotics and Automation, Shanghai, pp. 2798-2805, 2011.

[4] M. Liu, "Efficient segmentation and plane modeling of point-cloud for structured environment by normal clustering and tensor voting," Proc. of 2014 IEEE Int. Conf. on Robotics and Biomimetics (ROBIO 2014), Bali, pp. 1805-1810, 2014.

[5] A. Golovinskiy and T. Funkhouser, "Min-cut based segmentation of point clouds," Proc. of 2009 IEEE 12th Int. Conf. on Computer Vision Workshops (ICCV Workshops), Kyoto, pp. 39-46, 2009.

[6] T. Rabbani, F. A. van den Heuvel, and G. Vosselman, "Segmentation of point clouds using smoothness constraint," Proc. of Int. Archives of Photogrammetry, Remote Sensing and Spatial Information Sciences, p. 36, 2006.

[7] I. Bogoslavskyi and C. Stachniss, "Fast range image-based segmentation of sparse 3D laser scans for online operation," Proc. of 2016 IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS), pp. 163-169, 2016.

[8] F. Moosmann, O. Pink, and C. Stiller, "Segmentation of 3D lidar data in non-flat urban environments using a local convexity criterion," Proc. of 2009 IEEE Intelligent Vehicles Symposium, Xi'an, pp. 215-220, 2009.

[9] D. Zermas, I. Izzat, and N. Papanikolopoulos, "Fast segmentation of 3D point clouds: A paradigm on LiDAR data for autonomous vehicle applications," 2017 IEEE Int. Conf. on Robotics and Automation (ICRA), Singapore, pp. 5067-5073, 2017.

[10] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu, "A density-based algorithm for discovering clusters a density-based algorithm for discovering clusters in large spatial databases with noise," Proc. of the 2nd Int. Conf. on Knowledge Discovery and Data Mining (KDD'96), pp. 226-231, 1996.

[11] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger, "The R$^*$-tree: an efficient and robust access method for points and rectangles," Proc. of the 1990 ACM SIGMOD Int. Conf. on Management of data (SIGMOD '90), pp. 322-331, 1990.

[12] R. B. Rusu, "Semantic 3d object maps for everyday manipulation in human living environments," Ph.D. thesis, Computer Science Department, Technische Universität München, 2009.

[13] J. L. Bentley, "Multidimensional binary search trees used for associative searching," Commun. ACM, Vol.18, No.9, pp. 509-517, 1975.

[14] R. B. Rusu and S. Cousins, "3D is here: Point Cloud Library (PCL)," Proc. of 2011 IEEE Int. Conf. on Robotics and Automation, Shanghai, pp. 1-4, 2011.

[15] M. Harris, S. Sengupta, and J. D. Owens, "Parallel prefix sum (scan) with CUDA," H. Nguyen (Ed.), "GPU Gems 3," NVIDIA Corporation, 2007.

[16] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Ng, "ROS: an open-source Robot Operating System," ICRA Workshop on Open Source Software, Vol.3, pp. 1-5, 2009.

**Supporting Online Materials:**

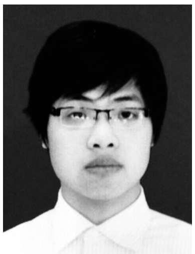
[a] Velodyne LiDAR. https://velodynelidar.com/ [Accessed October 20, 2019]

[b] NVIDIA CUDA (Compute Unified Device Architecture). https:///developer.nvidia.com//cuda-zone [Accessed October 20, 2019]

[c] Thrust Library. https:///developer.nvidia.com//thrust [Accessed October 20, 2019]

[d] Autoware.AI. https:///www.autoware.ai [Accessed October 20, 2019]

[e] Autoware Data. https:///data.iter4.jp [Accessed October 20, 2019]



**Name:**
Anh Nguyen

**Affiliation:**
Graduate School of Informatics, Nagoya University

**Address:**
711 National Innovation Complex (NIC), Furo-cho, Chikusa-ku, Nagoya 464-8601, Japan

**Brief Biographical History:**

2014 Received Bachelor degree from College of Information and Engineering, Ritsumeikan University

2014-2015 Software Engineer, FPT Software

2017 Received Master degree from Graduate School of Informatics, Nagoya University

2017- Ph.D. Student, Nagoya University

**Name:**
Abraham Monrroy Cano

**Affiliation:**
Graduate School of Informatics, Nagoya University

**Address:**
711 National Innovation Complex (NIC), Furo-cho, Chikusa-ku, Nagoya 464-8601, Japan

**Brief Biographical History:**

2008 Received B.E. from School of Engineering, National Autonomous University of Mexico

2016 Received M.S. from Graduate School of Informatics, Nagoya University

2016- Ph.D. Student, Nagoya University

**Main Works:**

- A. M. Cano, E. Takeuchi, S. Kato, and M. Edahiro, "An Open Multi-Sensor Fusion Toolbox for Autonomous Vehicles," IEICE Trans. on Fundamentals of Electronics, Communications and Computer Sciences, Vol.E103.A, pp. 252-264. 10.1587/transfun.2019TSP0005, 2020.
- K. Sakivama, S. Kato, Y. Ishikawa, A. Hori, and A. Monrroy, "Deep Learning on Large-Scale Multicore Clusters," Proc. of 2018 30th Int. Symp. on Computer Architecture and High Performance Computing (SBAC-PAD), Lyon, France, pp. 314-321, 2018.
- M. Hirabayashi, A. Sujiwo, A. Monrroy, S. Kato, and M. Edahiro, "Traffic light recognition using high-definition map features," Robotics and Autonomous Systems, Vol.111, pp. 62-72, 10.1016/j.robot.2018.10.004, 2018.
- K. Minemura, H. Liau, A. Monrroy, and S. Kato, "LMNet: Real-time Multiclass Object Detection on CPU using 3D LiDAR," Proc. of 2018 3rd Asia-Pacific Conf. on Intelligent Robot Systems (ACIRS), pp. 28-34, 2018.

**Name:**
Masato Edahiro

**Affiliation:**
Professor, Graduate School of Informatics, Nagoya University

**Address:**
C3-1 (631), Furo-cho, Chikusa-ku, Nagoya 464-8603, Japan

**Brief Biographical History:**
1985- NEC Corporation
2011- Nagoya University

**Main Works:**
- "A clustering-based optimization algorithm in zero-skew routings," Proc. of Design Automation Conf., pp. 612-616, 1993.

**Membership in Academic Societies:**
- The Institute of Electrical and Electronics Engineers (IEEE)
- The Operations Research Society of Japan (ORSJ)
- Information Processing Society of Japan (IPSJ)
- The Institute of Electronics, Information and Communication Engineers (IEICE)

**Name:**
Shinpei Kato

**Affiliation:**
Associate Professor, Graduate School of Information Science and Technology, The University of Tokyo
Founder & CTO, Tier IV, Inc.

**Address:**
7-3-1 Hongo, Bunkyo-ku, Tokyo 113-8656, Japan
13F Sumitomofudosan-Hongo Building, 3-22-5 Hongo, Bunkyo-ku,
Tokyo 113-0033, Japan

**Brief Biographical History:**
2008 Received Ph.D. from Keio University
2009-2012 Postdoctoral Scholar, Keio University, The University of Tokyo, Carnegie Mellon University, and University of California, Santa Cruz
2012-2016 Associate Professor, Graduate School of Information Science, Nagoya University
2017- Associate Professor, Graduate School of Information Science and Technology, The University of Tokyo

**Main Works:**
- Y. Suzuki, Y. Fujii, T. Azumi, N. Nishio, and S. Kato, "Real-Time GPU Resource Management with Loadable Kernel Modules," IEEE Trans. on Parallel and Distributed Systems, Vol.28, No.6, pp. 1715-1727, 2017.
- M. Yabuta, A. Nguyen, S. Kato, M. Edahiro, and H. Kawashima, "Relational Joins on GPUs: A Closer Look," IEEE Trans. on Parallel and Distributed Systems, Vol.28, No.9, pp. 2663-2673, 2017.

**Membership in Academic Societies:**
- The Autoware Foundation, Board of Directors