Efficient Position Estimation Based on GPU-Accelerated Content-based Image Retrieval

Yuta Kusamura[1], Toshiyuki Amagasa[2], Hiroyuki Kitagawa[2] and Yusuke Kozawa[3]

[1]*Graduate School of Systems and Information Engineering, University of Tsukuba, Tsukuba, Ibaraki, 305–8573, Japan*
[2]*Center for Computational Sciences, University of Tsukuba, Tsukuba, Ibaraki, 305–8573, Japan*
[3]*Artificial Intelligence Research Center, AIST, Koto-ku, Tokyo, 130–0064, Japan*
E-mail: kusamura@kde.cs.tsukuba.ac.jp; amagasa@cs.tsukuba.ac.jp; kitagawa@cs.tsukuba.ac.jp; yusuke.kozawa@aist.go.jp

Received 13 January 2018; Accepted 20 February 2018; Publication 25 April 2018

Abstracts

We propose an efficient position estimation method based on GPU-accelerated content-based image retrieval (CBIR). The idea is to use videos of *first-person vision* associated with geographical position information as the database. When a user sends a current subjective image, the system estimates the position using CBIR. Since features extracted from images are in general high-dimensional vectors, thousands of vectors are extracted even from a single image, resulting in high processing cost. On the other hand, GPUs (graphics processing unit) have contributed to accelerate various processing, while they are originally for graphics processing. Therefore, we utilize GPU to accelerate CBIR with appropriate data structures and algorithms. Moreover, our proposed method considers *spatial locality* of pedestrians in position estimation applications in order to improve accuracy.

Journal of Mobile Multimedia, Vol. 14_2, 197–230.
doi: 10.13052/jmm1550-4646.1423

We demonstrate the efficiency and accuracy of the proposed method through experiments using a video dataset.

Keywords: GPU, LSH, SIFT, Content-based Image Retrieval, Position Estimation.

1 Introduction

Geographical position is commonly used in various applications because of the popularization of mobile devices, such as smartphones. It is common to use GPS or radio-wave strength to estimate the position of a device. However, estimating positions suffers from several problems, such as high measurement errors in urban areas and unavailability in buildings or underground.

To tackle these problems, Kameda and Ohta [1] proposed a position estimation method using content-based image retrieval (CBIR) of *first-person vision* images, which are taken by cameras attached on someone's head or chest. It can be used especially for supporting visually impaired people, e.g., indoor-outdoor navigation system [2] or spot reminder system [3]. This method records the movies of first-person vision with position information in daily transportation paths. From the movies, an image database is constructed by extracting each frame with position information. Then, a pedestrian can get his/her current position by taking an image of first-person vision and issuing a query with the image as key. The system retrieves from the database the image which is the most similar to the query in terms of the contents, and returns the position of the retrieved image as the query result. To improve the accuracy and the robustness as well, it is important to take multiple videos with different conditions, e.g., weather, lighting, etc., even on a single route. Notice that, when considering a real application such as navigational support, it is necessary to estimate the position of an image as quick as possible, e.g., less than one second. Besides, we only need to retrieve for a query image the best matching image, i.e., top-1 query, and more sophisticated query such as top-k is not necessary.

In such applications, a large number of images can be extracted from videos, which need to be stored and maintained by image databases. In addition, a query processing should be executed quickly so that users can get their positions in real-time. More precisely, to perform image matching, it has become common to use image descriptors, such as SIFT [4], SURF [5], to represent an image, whereby image matching can be carried out by matching sets of such descriptors, which is called CBIR (content-based image retrieval). The problem is that, from each image, hundreds of features can be extracted, which demand high storage and computational costs. For this reason, it is a reasonable assumption that there is a server responsible for maintaining images and processing user queries, and users access the server via mobile networks.

In the meantime, parallel processing using GPU (graphics processing unit) has been gaining much attentions to accelerate large scale data processing. GPU was originally used for graphics processing, and has many processing cores which allow high parallelism. GPGPU (general-purpose computing on GPUs) is a technique to use GPUs for general-purpose computation utilizing these characteristics and has contributed to accelerate processing on many applications [6].

We propose an efficient position estimation method based on GPU-accelerated SIFT-based CBIR in this paper. The main ideas are 1) proposing appropriate data structures and algorithms for GPU computing, 2) compressing feature vectors of images, and 3) utilizing *spatial locality* of pedestrians. The process of proposed method can be divided into two phases, namely, database construction and query processing. Preprocessing is performed for efficient retrieval in the database construction phase, and the position is estimated based on the input query in the query processing phase.

We propose an efficient image matching algorithm on GPU, because appropriate algorithms and implementation are necessary to exploit the parallel processing power of GPU. One of the drawbacks of the naive matching method is low scalability due to the write conflicts among threads.

Also, our proposed method compresses the feature vectors because the data compression is important for GPU computing: The device memory of GPU is relatively small compared with main memory, while the data is necessary to be stored on the device memory. Therefore, it is important to compress the data loaded on GPU. There are various compression schemes and their characteristics are different. In particular, LSH (locality sensitive hashing) [7] is suitable for parallel computing because the calculations are simple. LSH is a hashing method which converts similar vectors into the same hash value, which can be used to accelerate similarity searching. For this reason, we utilized LSH to compress the feature vectors.

Our proposed method considers *spatial locality* of pedestrians to improve the accuracy of estimation, supposing actual applications. The position of the pedestrian is continuous because the position estimation may also be performed continuously. In other words, the position of the pedestrian can somewhat be estimated in advance, when position estimation is performed continuously. Therefore, we try to improve accuracy by reducing the candidate to only images around the query position.

In order to evaluate the performance of our proposed method, we conducted experiments over a video dataset. The experiments indicated that it is possible to estimate pedestrian's position by consistent high-speed CBIR for multiple parameters, using LSH and GPU. We also showed that accuracy of position estimation can be improved using spatial locality while keeping processing speed. Also, we discuss usefulness of our proposed method for position estimation applications.

The rest parts of this paper are organized as follows. First, we explain preliminaries in Section 2. Then, we explain our proposed method in Section 3 and Section 4. Sections 3 and 4 focus on acceleration of position estimation based on CBIR and improvement accuracy using spatial locality, respectively. We discuss the performance of proposed method through the experiments in Section 5. Then, we introduce related work in Section 6. Finally, we conclude this paper in Section 7.

2 Preliminaries

In this section, we explain preliminaries related to this research. Specifically, Section 2.1 explains the basis of content-based image retrieval, and Section 2.2 introduces the concept of GPU computing.

2.1 Position Estimation Based on Content-based Image Retrieval

Content-based image retrieval is to retrieve, for a given query image, similar images from an image database on the basis of visual contents [8]. As with Kameda and Ohta [1], we apply CBIR for position estimation applications, which can be defined as the following system.

Definition 1: As a premise, a set of images with position information, which could be potentially large, are given, and an image database is constructed from the images. Given a query image, the system retrieves from the database such an image that is the most similar to the query, and outputs its position information as the result. □

To measure the similarity between two images, many CBIR methods use feature vectors, which represent the features of an image. While various types of feature vectors such as SURF [5] and AKAZE [9] have been proposed, we utilize SIFT (scale-invariant feature transform) descriptor [4]. SIFT has an important property that it is robust against rotation, scale and brightness, which are desirable for CBIR.

2.1.1 SIFT descriptors

SIFT descriptor [4] is known to be one of the most basic image feature descriptors. One image is represented by a set of 128-dimensional feature vectors corresponding to the *keypoints* in the image. Keypoints are detected according to the difference in light levels and shades, and the SIFT descriptor with regard to the keypoint is computed based on the gradients of the image. The number of keypoints depends on the contents of images: In general, several hundreds of keypoints are detected, and the same number of feature vectors are generated.

2.1.2 Content-based image retrieval using SIFT matching

For a set of images, content-based image retrieval (CBIR) is to retrieve images that are similar to the given query image in terms of the visual contents. CBIR has been well-studied for several decades, and recent advances have been enabled by the introduction of latest image feature descriptors (as discussed above). For example, using SIFT descriptors, CBIR can be carried out as follows. The process is divided into two phases, namely, database construction phase and query processing phase.

In the database construction phase, the system is given a set of images. Then the system extracts SIFT descriptors from each image, and stores them in a database D. This database contains the records of the form $(\boldsymbol{f}, id)$ for each feature vector of the image, where $\boldsymbol{f}$ and id are a feature vector and the ID of image from which $\boldsymbol{f}$ is extracted, respectively. Hereafter, we denote by $id(\boldsymbol{f})$ the image ID corresponding to feature vector $\boldsymbol{f}$.

In the query processing phase, the system takes as input a query image, and outputs the image in the database which is the most similar to the query. More precisely, the system extracts SIFT vectors from the query image, and tries to find, for each query vector, the most similar vector from the database. Having identified those vectors, the system probes the image IDs that are associated to each vector, and conducts a majority vote, which in turn will be output as the query result. This process is described in Algorithm 1. Note that `dist` computes the distance between two vectors and `argmax` outputs the index that has the largest population in the array. Note also that, in general image retrieval, top-k similar images are returned as the results whereas we only need top-1 result in this work for the purpose of position estimation.

2.1.3 Content-based image retrieval using SIFT matching with LSH

LSH (locality sensitive hashing) [7] is a class of hashing methods where two vectors are mapped to the same hash bucket with high probability if they are similar. It has been used for various purposes, such as data compression and feature matching [10].

LSH converts a vector into a hash value by a hash function $H()$, which is computed by k different hash functions $h_i()$ $(0 \leq i < k)$. Specifically, a d-dimensional vector $\boldsymbol{v}$ is converted into a hash value $\boldsymbol{v}'$ as follows:

$$\boldsymbol{v}' = H(\boldsymbol{v}) = (h_0(\boldsymbol{v}), h_1(\boldsymbol{v}), \ldots, h_{k-1}(\boldsymbol{v})). \tag{1}$$

Notice that the hash functions $h_i()$ $(0 \leq i < k)$ are generated probabilistically, and the method to generate them depends on the distance criterion of the vector space. There have been several hashing functions for different distance criteria, such as l_p norm and cosine similarity [11].

Algorithm 1 CBIR using SIFT descriptors.

```
Input: $Q, D, num\_images$ ( = # of images in the image database)
Output: $result\_id$
1:  $freq[num\_images]$ // init with 0
2:  for $i = 0 : Q.size - 1$ do
3:      $min\_id = -1$
4:      $min\_dist = DOUBLE\_MAX$
5:      for $j = 0 : D.size - 1$ do
6:          $dist = dist(\boldsymbol{q_i}, \boldsymbol{f_j})$
7:          if $dist < min\_dist$ then
8:              $min\_id = id(\boldsymbol{f_j})$
9:              $min\_dist = dist$
10:         end if
11:     end for
12:     $freq[min\_id] + +$
13: end for
14: $result\_id = argmax(freq)$
```

We utilize l_2 norm-based LSH [12], because the l_2 norm is considered to be the standard distance criterion for SIFT feature vectors. A hash value is calculated using three variables $\boldsymbol{a}$, W and b: $\boldsymbol{a}$ is a d-dimensional vector whose elements are selected at random from Gaussian distribution; W is a predetermined positive real number; and b is also a real number selected from the uniform distribution of half-open interval $[0, W)$. The hash function $h_i()$ is defined as follows:

$$h_i(\boldsymbol{v}) = \left\lfloor \frac{\boldsymbol{a} \cdot \boldsymbol{v} + b}{W} \right\rfloor . \tag{2}$$

Intuitively, the original vector space is divided into some randomly generated non-overlapping partitions, and the vectors belonging to the same partition are given the same hash value. W determines the width of subspace as a parameter. W is to control the size of partitions and the size of has vectors as well, resulting in different sensitivity that the hash values for two vectors coincide with each other.

A typical usecase of LSH is to substitute it for similarity search over vectors; i.e., instead of performing pair-wise vector comparison, we can just retrieve such vectors that are in the same hash bucket. To leverage LSH in SIFT matching-based CBIR (Section 2.1.2), we need to slightly modify the procedure of above mentioned CBIR as follows.

In the database construction phase, after extracting SIFT features from images, we convert the feature vectors in the tuples into hash values by applying LSH, i.e., $\boldsymbol{f}_i' = H(\boldsymbol{f_i})$ for the tuple of $(\boldsymbol{f}_i, id)$. After that, the pairs of hash value and the associated image ID are stored in database D', using a data structure like dictionary, whereby a set of image IDs are retrieved quickly for a given hash value.

In the query processing phase, similar to the database construction phase, for a given query image, the SIFT features $\boldsymbol{q} \in Q$ are extracted and are converted to a set of hash values $\boldsymbol{q'} \in Q'$ using the same LSH function. After that, for each $\boldsymbol{q'} \in Q'$, the system retrieves the associated image IDs in D', and hash values $\boldsymbol{f'}$ ($\boldsymbol{q'} = \boldsymbol{f'}$) in D', and obtains the associate image IDs $id(\boldsymbol{f'})$. Finally, a majority vote is conducted by counting the frequency of IDs, and returns the ID which appears the most as the result.

2.2 GPU Computing

GPGPU (general-purpose computing on GPUs) is a technique to use GPU (graphics processing unit) for general-purpose computation rather than graphics processing. Although GPUs are originally developed for graphics processing, they have many processing cores, thereby allowing highly parallel computation. For this reason, GPGPU has contributed to accelerate processing in many applications [6]. NVIDIA's GPUs and their parallel computing platform CUDA[a] is one of the most widely used platform, and we exploit them in this work.

One of the key issues of GPU computing is that programming style is quite different from the one for ordinary CPUs due to the difference in the processor architecture, and inappropriate implementations easily deteriorate the performance. For this reason, choosing appropriate algorithms and data structures for GPU is important to harness the power of GPUs.

GPUs have hierarchically organized processors and memory architecture. A GPU has several streaming multiprocessors (SMs), which share L2 cache, and an SM is composed of hundreds of CUDA cores, which is the smallest processor in the architecture. In Tesla K40, there are 15 SMs, and each SM comprises 192 CUDA cores, which results in 2,880 CUDA cores in total. Regarding the memory architecture, global memory is the largest on GPU, whose size is around 10 GBs. Shared memory is available in each SM that the CUDA cores within the SM can share it, and the size is 16 KB to 96 KB, depending on specific GPUs. Besides, registers are available in each CUDA core. In terms of access latency, registers are the fastest, followed by shared memory and global memory. In terms of size, global memory is the largest, followed by shared memory and registers.

The processing model of GPU is also hierarchical. It consists of *thread*, *block*, and *grid*. Threads are minimum processing units in GPUs, which are processed by CUDA cores. Multiple threads are dealt with together; the processing units are called blocks. Threads in the same block are processed by an SM, and the resource of the SM is shared. Multiple blocks are grouped into a grid, which is the maximum processing unit in GPU.

In CUDA environment, exploiting SIMD (single instruction, multiple data) is important to efficiently perform parallel computing, and there are several well-known data-parallel primitives. These primitives can be computed efficiently in parallel, because they are frequently used and, therefore, efficiently implemented as a library.[b]

[a]Parallel Programming and Computing Platform | CUDA | NVIDIA `www.nvidia.com/cuda`
[b]CUDA Toolkit Documentation `http://docs.nvidia.com/cuda/`

- **reduce**:
  Returns a scalar value $a_0 \oplus a_1 \oplus \ldots \oplus a_{n-1}$ for the input of an array $[a_0, a_1, \ldots, a_{n-1}]$ and operator $\oplus$.
- **scan**:
  Returns an array $[0, a_0, a_0 \oplus a_1, \ldots, a_0 \oplus a_1 \oplus \ldots \oplus a_{n-2}]$ for the input of an array $[a_0, a_1, \ldots, a_{n-1}]$ and operator $\oplus$.
- **sort**:
  Sorts the input array $[a_0, a_1, \ldots, a_{n-1}]$.

3 GPU-based Acceleration of CBIR using LSH

We explain in this section the detail of acceleration of CBIR. Our proposed method takes as input a query image and an image database with position information and outputs the position where the query image is supposed to be taken. To obtain the result position, we utilize CBIR. More precisely, we first perform an ordinary CBIR, and get the most similar image against the query. Then we extract the position information associated to the query result. Indeed, the most time-consuming part is the image retrieval part. For this reason, we propose an efficient algorithm of LSH-based CBIR (Section 2.1.3) using GPU.

The process can be divided into two phases, database construction and query processing as in ordinary LSH-based CBIR. Notice that our main target is query processing phase rather than database construction phase, because the latter is an offline process, which can be executed in advance. For this reason, for the database construction, we concentrate on explaining the in-memory data structure for GPUs, while parallel query execution will be explained for the query processing.

The technical challenge is how to maximize parallelism among threads by avoiding read/write contentions. This is because the performance bottleneck lies on the process of aggregating results from thousands of threads. If we implement it naively, the performance degrades easily because of the contentions. To this end, we have carefully designed the data structures and the algorithm.

3.1 Database Construction

In the database construction phase, the system extracts the features from given images and compresses them to hash values using LSH. We then store the hash values using three distinct arrays, which are suitable for parallel processing on GPUs, and send them to the global memory on a GPU. Concretely, database D' is created as explained in Section 2.1.3. From D', the system generates the following three arrays, and transmits them to the GPU's global memory.

- f_val:
  An array that stores all hash values $\boldsymbol{f'}$ in D', which is sorted according to the ascending order of the values. Storing hash values f_val in ascending order enables binary search.
- id_val:
  An array that stores all image IDs associated with $\boldsymbol{f'}$ in line with the order of hash values in f_val. Notice that one hash value may be associated to two or more image IDs.
- id_ptr:
  An array that represents the correspondence between f_val and id_val. Concretely, $id\_ptr[i]$ is an offset of id_val from the initial record.

This process is described in Algorithm 2 and Figure 1 illustrates an example of the data structure. We will explain the query processing phase according to the example.

3.2 Query Processing

In the query processing phase, the system retrieves such an image from the database that matches the best with the query image in terms of the feature vectors. The proposed method firstly extracts a bag of feature vectors Q from the query image by CPU, and transmits the vectors to the GPU's global memory. The rest of the process is performed in the GPU. To the vector

Algorithm 2 Array construction based on D'.

```
Input: $D'$, where $(\boldsymbol{f}'_i, id_i) \in D'$, $id_{ij} \in id_i$
Output: $f\_val$, $id\_val$,$id\_ptr$
1:  Sort $D'$ in ascending order of $\boldsymbol{f}'_i$.
2:  $offset = 0$
3:  $id\_ptr[0] = 0$
4:  for $i = 0 : rows\_of(D') - 1$ do
5:      $f\_val[i] = \boldsymbol{f}'_i$
6:      for $j = 0 : size\_of(id_i) - 1$ do
7:          $id\_val[offset++] = id_{ij}$
8:      end for
9:      $id\_ptr[i+1] = offset$
10: end for
```

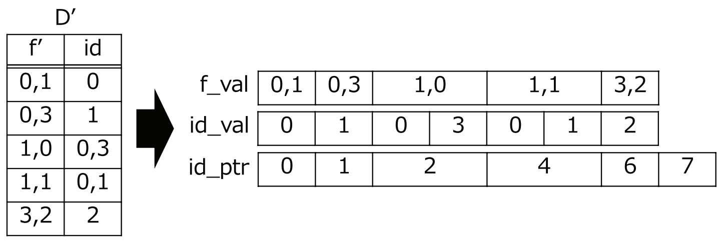


Figure 1 Array construction based on D'.

Q, our method applies LSH in parallel for yielding hash values Q'. After that, GPU matches the hash values of the query and those in the database in parallel. Finally, the query result is determined by aggregating the matching result. The detail will be explained below.

3.2.1 Compressing feature vectors

In this step, we assume that feature vectors are already extracted from the query image and transmitted to the GPU's global memory. To these vectors, we apply LSH to obtain hash values.

Each feature vector $\boldsymbol{q_i} \in Q$ is 128-dimensional vector, and hundreds to thousands of features are extracted from an image depending on the contents. Computations of compression for multiple feature vectors are independent and therefore can be processed in parallel. Regarding the conversion of each feature vector $\boldsymbol{q_i}$, predefined number of hash functions $h_j(\boldsymbol{q_i})$ (explained in Section 2.1.3) need to be applied, and the hash functions can be independently applied to the elements of a feature vector. Thus the conversion of a feature can be also processed in parallel. In summary, the conversions of each feature vector $\boldsymbol{q_i}$ are processed in block parallel, and the computations of the hash value $h_j(\boldsymbol{q_i})$ are processed in thread parallel. This algorithm is described in Algorithm 3.

Algorithm 3 Compressing feature vectors of the query.

```
Input: $Q$, where $q_{ij} \in Q$
Output: $Q'$, where $q'_{ij} \in Q'$
1: for $i = 0 : Q.size() - 1$ do in block parallel
2:     for $j = 0 : k - 1$ do in thread parallel
3:         $q'_{ij} = h_k(q_{ij})$
4:     end for
5: end for
```

3.2.2 Matching process

This step matches hash values for identifying the most similar image. The system retrieves the hash values from f_val which are exactly the same with one of the hash values extracted from the query image. It then extracts the image IDs corresponding to the retrieved hash values using id_val and id_ptr. Afterwards, the most frequently appearing image ID is identified.

If we implement it naively, one of the easiest ways is to use binary searching to find the hash value, probe the associated image IDs, and use `atomicAdd` to count the occurrences of image IDs. The `atomicAdd` function is an atomic addition operation on two numbers on global memory or shared memory provided by CUDA. It enables multiple threads to concurrently update the same data by avoiding write contentions. By using `atomicAdd`, the naive matching process becomes as follows: We create a counter on global memory or shared memory for each image ID, and, as soon as an image ID is detected, we count up the corresponding counter by one using `atomicAdd`. However, this suffers from performance degradation for three reasons: 1) the large access latency to global memory, 2) poor scale up performane due to the global locking incurred by `atomicAdd`, and 3) not enough capacity of shared memory for large scale databases.

Thus, we propose an efficient matching method using multiple arrays, without `atomicAdd`. The process is divided into two steps, namely, obtaining matched list and detecting the most frequent ID.

3.2.2.1 *Obtaining matched list*

This step generates a matched image ID list (matched list) M for the query by using Algorithm 4. The matched list M contains all image IDs that match with the hash values being processed. This can be done by copying the partial array of id_val that corresponds to the hash value. In order to efficiently parallelize this by blocks in GPU, we compute 1) the beginning position of the source, 2) the copy size, and 3) the beginning position of destination for each block in advance, thereby avoiding using expensive operations, such as `atomicAdd`. For storing the computed values, we introduce arrays F, N and T. M can be generated efficiently using these arrays. We explain the details of the generating process of M in the following.

To construct F, N and T, first, the system retrieves the hash values in f_val which are matched to one of the hash values of the query by binary searching, and gets the matched index of f_val as $index$ (line 2 in Algorithm 4). If the matching fails, the value of $index$ is -1. Second, F

Algorithm 4 Obtaining matched list.

```
Input: $Q', f\_val, id\_ptr, id\_val$
Output: $M$
1: for $i = 0 : Q'.size() - 1$ do in thread parallel
2:     $index = binary\_search(q'_i, f\_val)$
3:     if $index == -1$ then
4:         $F[i] = -1$
5:     else
6:         $F[i] = id\_ptr[index]$
7:     end if
8:     if $index == -1$ then
9:         $N[i] = 0$
10:    else
11:        $N[i] = id\_ptr[index + 1] - id\_ptr[index]$
12:    end if
13: end for
14: $T = scan(N)$
15: for $i = 0 : Q'.size() - 1$ do in block parallel
16:    for $j = 0 : N[i] - 1$ do in thread parallel
17:        $M[T[i] + j] = id\_val[F[i] + j]$
18:    end for
19: end for
```

is generated by copying the value of $id\_ptr[index]$ (line 3–7). Third, N is generated by calculating the difference between the value of $id\_ptr[index]$ and $id\_ptr[index + 1]$ (line 8–12). These computations are processed in thread parallel for each hash value of the query. Finally, T is generated by applying `scan` to N (line 14). Figure 2 illustrates an example of this process.

M can be created based on these arrays F, N and T. Concretely, the elements of M are computed as follows:

$$M[F[i] : F[i] + N[i] - 1] = id\_val[T[i] : T[i] + N[i] - 1]. \qquad (3)$$

This part is described in lines 15–19 in the algorithm, and Figure 3 shows an example.

In terms of the process model, this is processed by assigning a block for a matched hash value and by assigning threads for distinct image IDs. Notice that no write conflict happens in this algorithm because write addresses are predetermined and different threads write different addresses. Consequently, it can be executed efficiently in parallel.

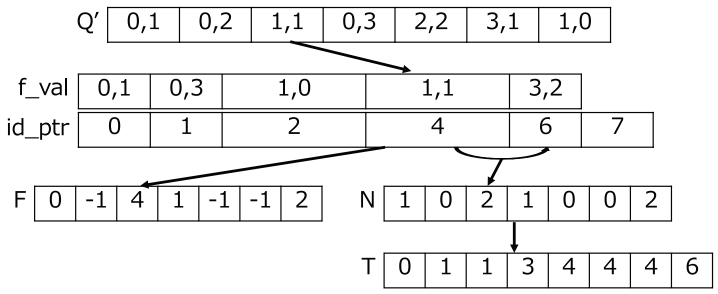


Figure 2 Generating F, N, T.

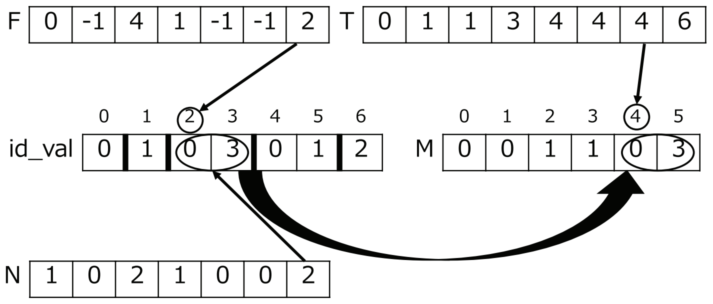


Figure 3 Copying matched IDs using F, N, T.

3.2.2.2 *Detecting the most frequent ID*

Having generated matched list M, this step finds the most frequently appeared image ID (most frequent ID). This can be simply computed by sorting the matched ID list and counting the lengths of contiguous runs of the same elements. Here, we define *borders* and *sections*. In a sorted list, a border refers to a position where the adjacent elements store different values and a section refers to a sublist between two consecutive borders.

The algorithm for detecting ID is shown in Algorithm 5 and an example is shown in Figure 4. Firstly, M is sorted, and M' refers to the sorted list (line 1). Secondly, the borders are detected by comparing adjacent elements in M' and the results are stored in an array B where a border is represented as “1” (line 2–5). Thirdly, section numbers are detected by applying `scan` primitive to B, and the results are stored in an array I (line 6). After that, for each border, the position is obtained and stored in an array O. Now, I keeps the positions where the results of image IDs should be stored (line 7–11), and O describes the offsets to end position of each section. Subsequently, the system can obtain the size for each section by comparing adjective elements in O

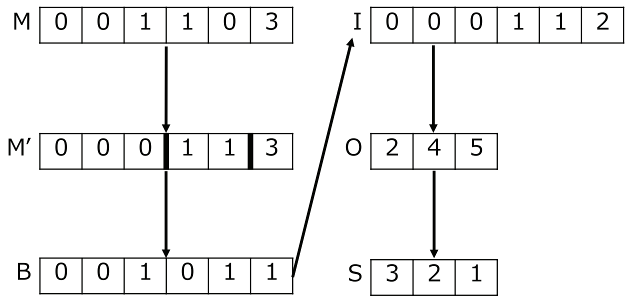


Figure 4 Detecting frequency of the image IDs using matching list.

Algorithm 5 Detecting frequency of the image IDs using matching list.

```
Input: $M$
Output: $id$
1:  $M'=sort(M)$
2:  for $i = 0 : M'.size() - 2$ do in thread parallel
3:      $B[i] = M'[i] \mathrel{!}= M'[i+1]$
4:  end for
5:  $B[M'.size() - 1] = 1$
6:  $I = scan(B)$
7:  for $i = 0 : B.size() - 1$ do in thread parallel
8:      if $B[i] == 1$ then
9:          $O[I[i]] = i$
10:     end if
11: end for
12: $S[0] = O[0] + 1$
13: for $i = 1 : O.size() - 1$ do in thread parallel
14:     $S[i] = O[i] - O[i-1]$
15: end for
16: $r = reduce\_with\_index(S)$
17: $id = M'[O[r]]$
```

(line 12–15). Since each calculation is independent with respect to image ID, these calculations can be parallelized by threads. The `sort` and `scan` primitives are available in library, so these process are efficiently executed in parallel on GPU.

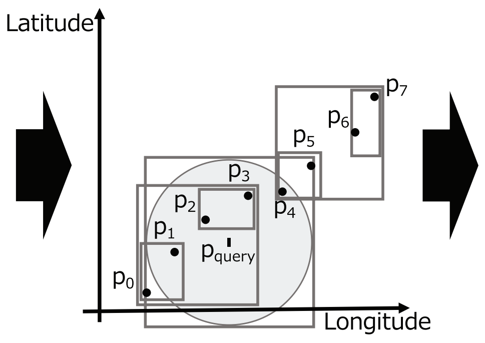
The result can be detected by finding the section whose length is the longest, followed by finding the corresponding image ID. The system obtains such a section number using `reduce_with_index` and define it as r (line 16). Although `reduce_with_index` is a function, similar to reduce, which returns the position obtained the result of standard `reduce`. Since this function is not in library for GPU, we implemented it by slightly modifying an existing reduce implementation. After that, the image ID associated to the section r can be determined by M' and O (line 17), and this image ID is the result.

4 Accuracy Improvement Considering Spatial Locality

We try to improve accuracy considering spatial locality of pedestrians, based on the method explained in Section 3. Query images are assumed to continuously be input in pedestrian's position estimation applications. The interval of query image inputs is very short because the position estimations are performed in real time. For this reason, the positions where two consecutive query images are taken are supposed to be very closed. We define this property as *spatial locality*, and we propose the method leveraging it for improving accuracy. Our method assumes that the position information is associated with a query image. The latest estimated position can be used for it because query images are assumed to continuously be input, as we explained above.

The system detects images which are taken near the position of query image using R-tree [13] as described in Figure 5 (define these images as *neighbors*), and targets neighbors as candidates of CBIR. R-tree is one of the spatial index, which supports range query searching for a given point. The R-tree construction is performed in database construction phase, and the neighbors detection and candidate reduction are performed in query processing.

Image	Position
I_0	p_0
I_1	p_1
I_2	p_2
I_3	p_3
I_4	p_4
I_5	p_5
I_6	p_6
I_7	p_7



neighbors
I_1
I_2
I_3
I_4

Figure 5 Detecting neighbors using R-tree.

4.1 Database Construction

R-tree against positions is constructed in database construction phase of proposed method. The data points of R-tree are position information corresponding to the each image in the image database, and the labels are the image ID. The longitude-latitude space of positions is set as R-tree's 2-dimensional space. While several errors occur because the earth is a sphere, it is acceptable because we suppose position estimation to be performed in short distance.

4.2 Query Processing

Given a query, the system detects the neighbors using R-tree, and retrieves the similar image using CBIR with targeting only neighbors. In order to perform query processing on GPU efficiently, we extend the method explained in Section 3 to select the candidate based on neighbors. Concretely, the image IDs which are not included in neighbors are converted into "NA" (=`INT_MAX`) at the time of generating matched list, and NA is not counted in the detecting the most frequent ID process. The algorithm of detection process can be described in Algorithm 6. The extended parts from the original algorithm are the processes of line 2, 5–7 and 10, which reduce the NA from M' and B using $last$, which describes the border position between NA and real image IDs. The lines 5–7 detect $last$, which detects the border whose right element is NA as $last$. Then the section of NA in M' and B is resized using $last$ in the process of line 10. After that, the most frequent ID can be detected in the same procedure of original algorithm. Figure 6 is an example of the detection process.

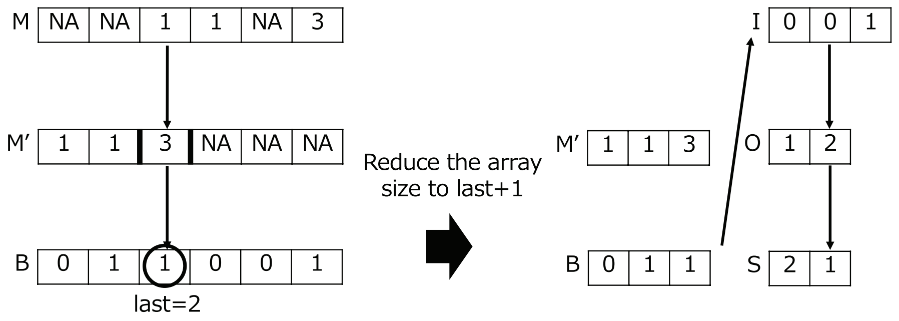


Figure 6 An example of detecting most frequent ID excluding NA.

Algorithm 6 Detecting most frequent ID excluding NA.

```
Input: $M$
Output: $id$
1:  $M'$=$sort(M)$
2:  $last = M'.size() - 1$
3:  for $i = 0 : M'.size() - 2$ do in thread parallel
4:      $B[i] = M'[i] \; != M'[i+1]$
5:      if $B[i] == 1$ && $M'[i+1] ==$ NA then
6:          $last = i$
7:      end if
8:  end for
9:  $B[M'.size() - 1] = 1$
10: Reduce the size of $M'$ and $B$ to $last + 1$
11: $I = scan(B)$
12: for $i = 0 : B.size() - 1$ do in thread parallel
13:     if $B[i] == 1$ then
14:         $O[I[i]] = i$
15:     end if
16: end for
17: $S[0] = O[0] + 1$
18: for $i = 1 : O.size() - 1$ do in thread parallel
19:     $S[i] = O[i] - O[i-1]$
20: end for
21: $r = reduce\_with\_index(S)$
22: $id = M'[O[r]]$
```

5 Experiments

We conducted experiments in order to verify the performance of proposed method.

5.1 Dataset

In the experiments, we used an original video dataset with GPS information. This dataset consists of two videos, which are taken in the almost same time. They ware taken by walking in the University of Tsukuba with putting a video camera on the front of chest. We used OLYMPUS STYLUS TG-Tracker[c] as the video camera, which can record GPS information every two seconds.

The two videos ware taken on the almost same route and the time, for the reason that the position estimation applications are performed in such

[c]STYLUS TG-Tracker | T (Tough) Series | Olympus `http://asia.olympus-imaging.com/product/compact/tgtracker/index.html`

situations. One consists of 26,401 frames, and the other consists of 25,980 frames. The experiments used the former for a database and the latter for a query set.

There are the frames without GPS information because the camera can record the information only every two seconds. For this reason, we complemented GPS information by linear complement for every frame. It should be noticed that the positions are described as longitude and latitude because of GPS information. In experiments, we calculated the distance between two points in order to verify the performance. To calculate the distance, we used Hubeny's formula,[d] which can calculate the distance with low-error, if the distance is relatively short.

5.2 Experimental Environment

We used a PC running CentOS release 6.7 (final) having two Intel Xeon Processor E5-2687W v2 CPUs and 128 GB main memory. The CPUs have eight cores and the clock frequency is 3.4 GHz. Also the machine has NVIDIA Tesla K40 GPU, which has 2,880 cores and 12 GB device memory. The clock frequency of the GPU cores is 745 MHz.

The programs were written in C++ and CUDA, compiled by nvcc 8.0 with optimization option -O3. We used OpenMP[e] and OpenCV[f] for parallelization on CPU and extraction of SIFT feature vectors, respectively.

5.3 Comparative Methods

To evaluate the performance of our proposed method, we implemented six comparative methods including our proposed method as follows:

- **Raw-SIFT-based method (RS)**:
  The CBIR method described in Section 2.1.2. This program is run in parallel (32 threads) on CPU.
- **LSH-based method @ CPU (LC)**:
  The CBIR method described in Section 2.1.3. The program is run in both of serial and parallel (32 threads) on CPU.
- **LSH-based method @ GPU with atomicAdd (LG with Atomic)**:
  The CBIR method described in Section 3, which is a simple method on

[d] Kashmir / Calculation Formula `http://www.kashmir3d.com/kash/manual-e/std_siki.htm`

[e] OpenMP `www.openmp.org/`

[f] OpenCV | OpenCV `opencv.org`

GPU using `atmicAdd`. The program is run in parallel on GPU and the hash function is l_2 norm-based.

- **LSH-based method @ GPU (LG)**:
  The CBIR method described in Section 3, which avoids conflicts of threads.
- **LSH-based method @ CPU with spatial locality (LC with SL)**:
  The CBIR method extended for spatial locality from LC.
- **LSH-based method @ GPU with spatial locality (LG with SL)**:
  The CBIR method extended for spatial locality from LG.

5.4 Experimental Methodology

We conducted four experiments and the experimental methodologies are shown in this section.

5.4.1 Experiment 1

In order to evaluate processing time, we measured the average time of retrieving multiple queries from the database. The comparative methods are all methods except spatial locality-aware methods: RS, LC, LG with Atomic and LG. The queries are 100 images randomly selected from the query set.

5.4.2 Experiment 2

We evaluate the accuracy of estimation through experiment 2, while it is supposed that the accuracy of LSH-based methods decreases because of the data compression. The evaluation is performed using position information considering the real position estimation applications. We define *success* or failure of an estimation according to the distance between the position of a query image and the estimated position. If the distance is less than 15 m, the estimation is regarded success; otherwise, it is failure. We examine the success rates (i.e., the fraction of successful estimations) as the accuracy measure. The comparative methods are LG and RS, which are LSH-based method and not, respectively. Note that, the results of LC, LG with Atomic and LG are all same.

5.4.3 Experiment 3

In order to evaluate the performance of spatial locality-aware methods, we examined the results against continuous queries. The evaluation was based on Kameda and Ohta's [1] evaluation. The comparative methods are LG and LG with SL. To consider spatial locality, the position information of latest query result is given with the next query. We compared the path distance—distance between certain position and the start position of the route—of both query position and estimated position.

We need to give the search radii of R-tree used for detecting neighbors as a parameter. Multiple search radius are tested in order to verify the performance against parameters. Also, we examined the performance against non-spatial locality-aware method in same settings. Note that, the parameters of LSH are $k = 64$ and $W = 900$.

5.4.4 Experiment 4

We investigated the processing time of spatial locality-aware methods because the algorithms are extended from the original methods to apply spatial locality. The comparative methods are LC_Parallel, LG, LC_Parallel with SL and LG with SL. We set the parameters of LSH $k = 64$ and $W = 900$ and the search radius 0.001. The other evaluation settings and methodologies are same as experiment 1.

5.5 Results

We show the experimental results in this section.

5.5.1 Experiment 1

Figures 7–9 describe the result of experiment 1. Each figure describes the result for each parameter k, whose horizontal axis is parameter W and vertical axis is processing speed in log scale. In order to analysis the result, we examine the average number of matched IDs (same as the length of matched list) and describe it in the Table 1.

We will analyze the result through the figures and the table. LG is the fastest method in the case of $k = 16$ and $k = 32$. In particular, the speed up ratio for $k = 16$ and $W = 900$ is largest: The speed of LG is about 95 times faster than LC_Serial and about 5 times faster than LC_Parallel. Moreover, the processing time of LG is 10–20 ms in almost all situations, while the time of LC_Serial or LC_Parallel are several seconds or several

Table 1 The average number of matched IDs.

	W = 300	W = 600	W = 900
k = 16	83,808.3	1,826,044.3	17,906,974.3
k = 32	3,649.8	77,689.2	600,641.2
k = 64	36.7	1,696.4	9,233.2

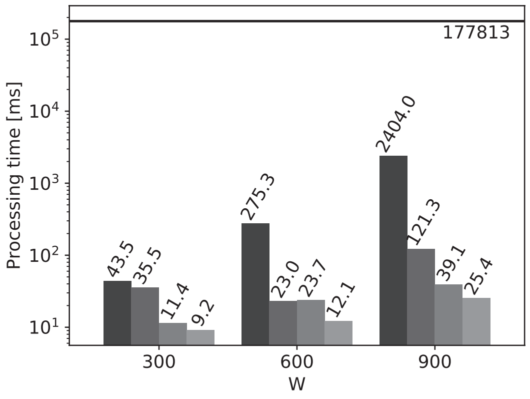


Figure 7 The processing time for $k = 16$.

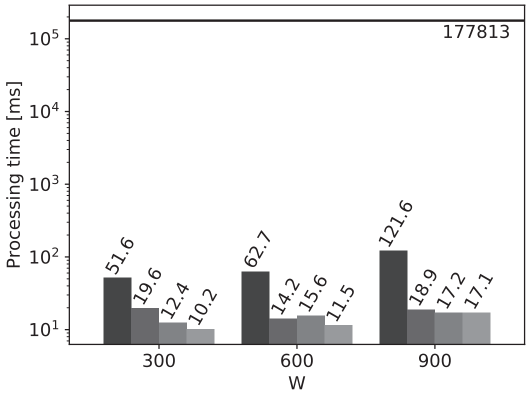


Figure 8 The processing time for $k = 32$.

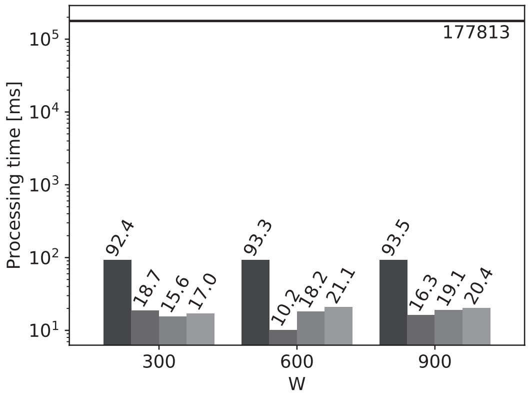


Figure 9 The processing time for $k = 64$.

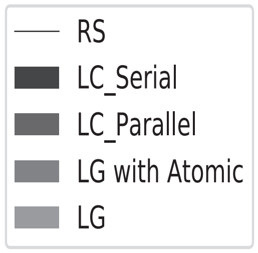


Figure 10 The legend of Figures 7–9.

hundreds milliseconds. On the other hand, LG is slower than LC_Parallel in some conditions in $k = 64$. The reason why LG is slower is that the number of matched IDs is relatively small in such situation. If the number of matched IDs is small, the performance of GPU is decreased because there are less data which can be processed in parallel.

5.5.2 Experiment 2

Figure 11 describes the result of experiment 2 as a heat map. This figure indicates the success rate in each cell, whose horizontal axis is parameter W and the vertical axis is parameter k. The color density expresses the degree of success rate: The color is darker if the success rate is higher.

From the figure, we can observe that the parameters affect the success rate. In particular, the success rate is highest in case of $k = 64$ and $W = 900$. The rates on cells on the diagonal line from upper left to lower right are relatively high. We can see that from this, an appropriate parameter setting is necessary because the parameters affects ambiguity of LSH. Note that, the success rate of RS is 95%.

5.5.3 Experiment 3

Figure 12 is the result against non-spatial locality-aware method and Figures 13–16 are the results of spatial locality-aware method. The horizontal axis describes the path distance of query position, and the vertical axis describes the path distance of estimated position. The points in the figures correspond to queries. If the estimation is performed correctly, the distance between two path distances is 0. Therefore, speaking intuitively, the estimation accuracy is high if the graph is similar to $y = x$. Also, the mean absolute errors of path distance at estimated position are described in Table 2.

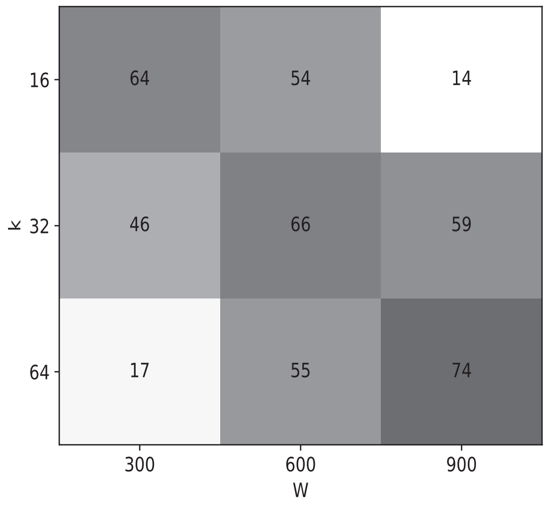


Figure 11 The success rate for each parameters [%].

Table 2 Mean absolute errors of path distance at estimated position [m].

Without spatial locality	Search radius of neighbors			
	0.0005	0.001	0.0015	0.002
33.3	46.2	11.9	12.4	51.1

We will analyze the result. Comparing the methods which are applied spatial locality, the accuracy is highest in case of the search radius is 0.001. Comparing the result with the search radius of 0.001 and without spatial locality, it can be indicated that the former is higher. Therefore, spatial locality can improve the accuracy with setting appropriate search radius.

5.5.4 Experiment 4

Figure 17 is the result of experiment 4. The horizontal axis is search method and vertical axis is processing time. In order to apply spatial locality, it can be supposed that the processing speed become slow. However, the processing time of LG and LG with SL are almost same. The reason is that the data size is decreased due to size reduction of arrays in query processing. On the other hand, comparing LC_Parallel and LC_Parallel with SL, the processing time of LC_Parallel with SL is 1.3 times longer than LC_Parallel. It can be said that the processing cost increases by applying spatial locality.

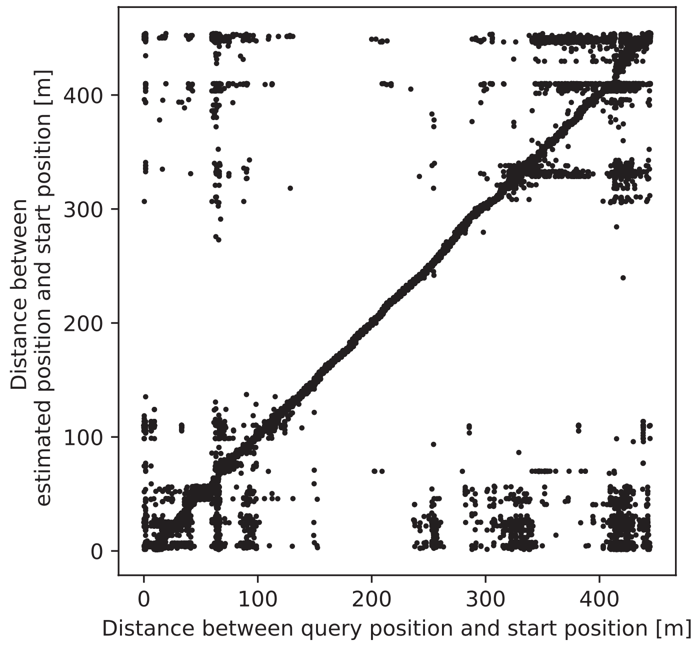


Figure 12 The path distance without spatial locality.

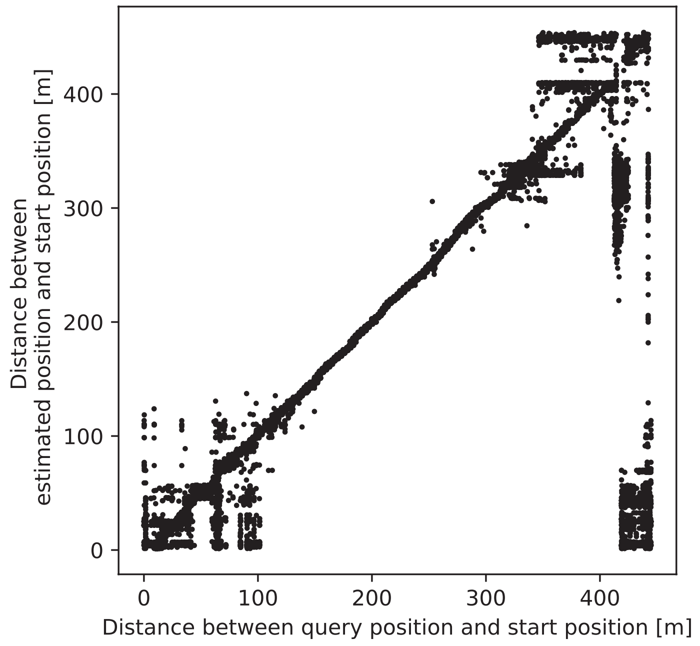


Figure 13 The path distance in the case of the search radius is $= 0.0005$.

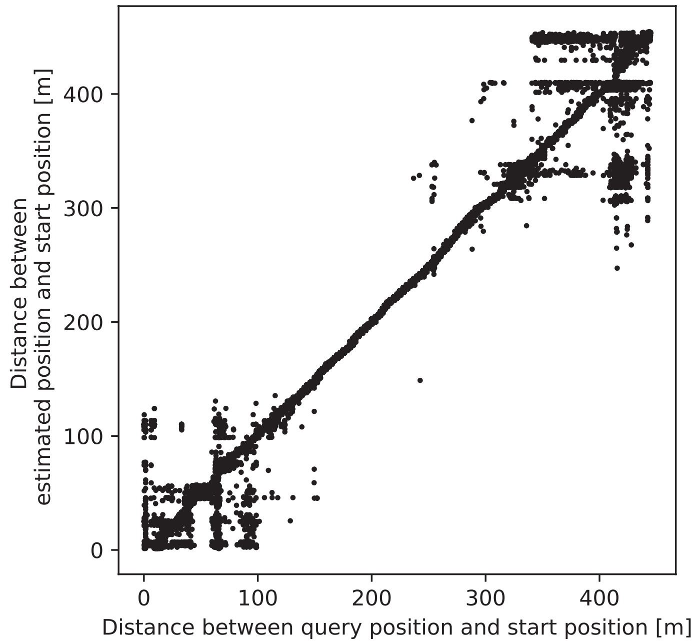


Figure 14 The path distance in the case of the search radius is $= 0.001$.

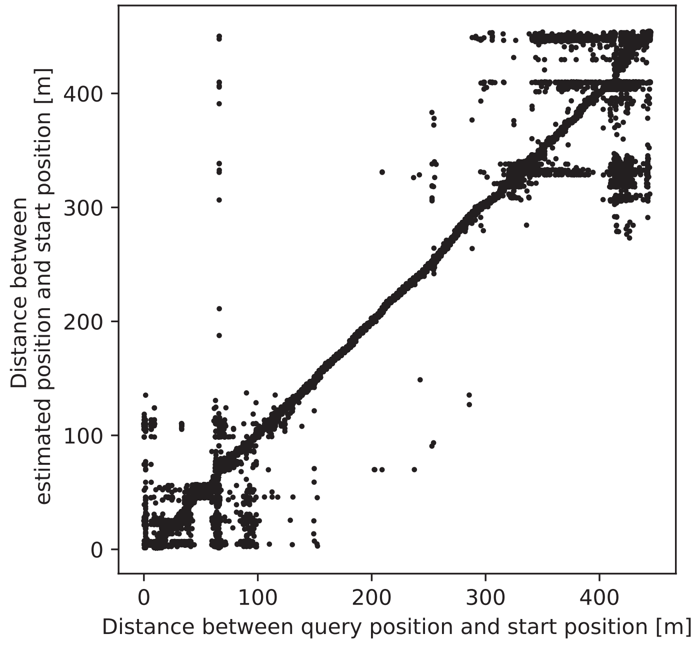


Figure 15 The path distance in the case of the search radius is $= 0.0015$.

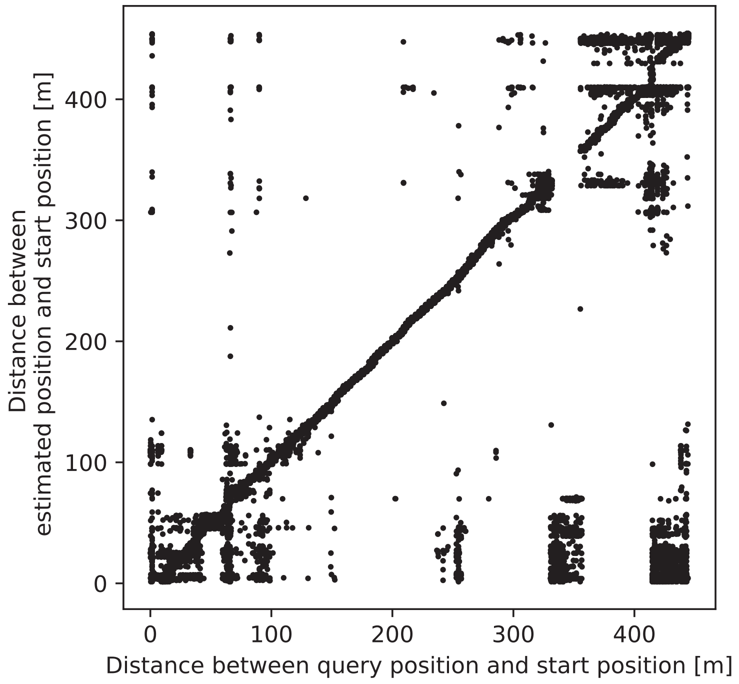


Figure 16 The path distance in the case of the search radius is $= 0.002$.

5.6 Discussion

We have shown that efficient position estimation based on high-speed CBIR can be realized by using GPU and LSH. The estimation can be performed on GPU in about 10–20 milliseconds, while the estimation costs several hundreds of milliseconds. It is demanded that the estimation can be performed in several tens of milliseconds because the queries are continuously input in real time position estimation. Therefore, the retrieval using GPU and LSH is useful from the point of view of processing time.

We proposed the extended method applied spatial locality of pedestrians because the accuracy of LSH-based methods is lower than a naive method. Through experiment 3, we demonstrate that spatial locality improves the accuracy of estimation. Moreover, spatial locality hardly affects the processing time on GPU because the cost of applying spatial locality is small due to size reduction of arrays in query processing.

For these reasons, our proposed method is useful for position estimation against raw-SIFT based method or CPU implementations. Raw-SIFT based

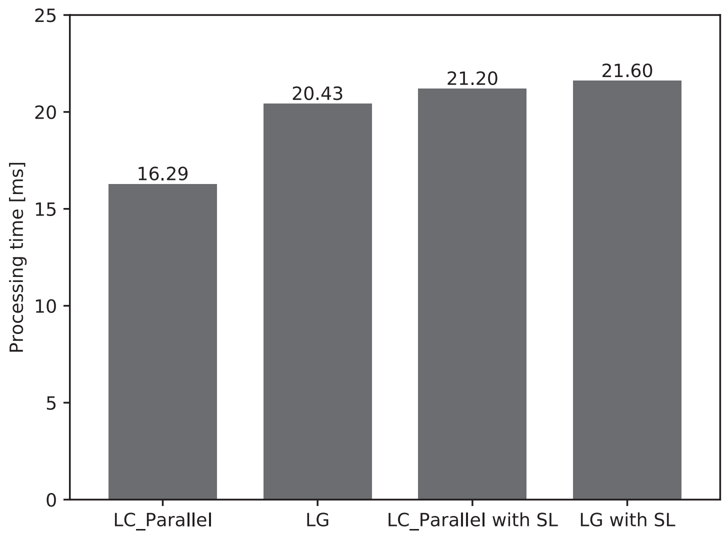


Figure 17 The processing time for each method including spatial locality-aware method.

method is not able to perform in real time because the processing cost of raw-SIFT based method is large. On the other hand, our proposed method is faster than the other methods in most conditions. As for accuracy, LSH decreases it due to data reduction, but this issue can be overcome by exploiting the spatial locality. Therefore, our proposed method can estimate the pedestrian's position with higher accuracy in real time.

6 Related Work

In this section, we review related research work. Since our research deals with the problem of position estimation in pedestrians using CBIR, we explain the ones that deal with position estimation in Section 6.1, followed by the ones for accelerating CBIR in Section 6.2.

6.1 Position Estimation Using CBIR

Kameda and Ohta [1] proposed a position estimation method using CBIR of first-person vision mainly for visually impaired people, and our work is inspired by this work. Although estimated positions can be obtained by retrieving a query image from video images with geographical locations, the accuracy might be affected by several reasons. In order to improve the accuracy, they proposed seven verification criteria for examining image matching; candidate images are verified by seven criteria and the ones that satisfy all of them are regarded as the result. However, this method is too computationally expensive and cannot be executed in the time of video frame speed.

Kamasaka et al. [14] proposed an estimation method, which is an extended version of Kameda and Ohta's method, for multiple videos that are recorded in different conditions. According to Kameda and Ohta's method, the accuracy can be improved by multiple videos as a database. However, the accurate position is necessary for every frame in the videos, and the processing cost is large. The method proposed by Kamasaka et al. integrates two videos, namely, main video and sub video in preprocessing, and reduce the processing cost. The main video has position information and the other has not. The integration can be performed by retrieving images in the sub video from frames in the main video, and the position information is attached to the query image according to the result. This can increase the accuracy with low cost.

6.2 Accelerating CBIR

One of the most famous approaches for CBIR is BoVW (bag-of-visual-words) framework proposed by Sivic et al. [15]. In this framework feature vectors are quantized into clusters, and the vectors in the same cluster are dealt as the same vector. The quantized vectors are called *visual words*, whereby CBIR is enabled by matching them between query image and images being queried. Specifically, standard techniques for information retrieval (IR), such as inverted index and TF-IDF, are used.

To accelerate this method and related ones as well, there have been several methods that exploit GPUs. Cevahir et al. [16] proposed a 2-step matching method on GPU to extract feature vectors efficiently. In the first step, hierarchical k-means is applied to feature vectors which in turn are divided in some clusters. In the query processing, the system retrieves the nearest cluster to the feature vectors extracted from the query. Then the nearest feature vectors in the cluster are retrieved, thereby reducing the search space. This process is executed in parallel using multiple threads.

Data compression of SIFT is important for accelerating due to the fact that volume significantly increases when dealing with many images. Chandrasekhar et al. [17] surveyed SIFT compression schemes. They compared several schemes by experiments. The schemes are divided into three types, *hashing*, *transform coding*, and *vector quantization*. LSH is one of the hashing methods introduced as a popular hashing technique for high dimensional descriptors without training.

7 Conclusion

In this paper, we have proposed an efficient position estimation method based on content-based image retrieval over video images. The main ideas are 1) proposing appropriate data structures and algorithms for GPU computing, 2) compressing feature vectors, and 3) utilizing spatial locality of pedestrians. In order to exploit the parallel processing power of GPU, we proposed appropriate data structure and algorithms. Also, the feature vectors are compressed by using LSH because the compression is important due to the capacity of GPU. Moreover, we extended the implementation on GPU by applying spatial locality of pedestrians.

In order to evaluate the performance of our proposed method, we conducted experiments. The experiments indicated that it is possible to estimate pedestrian's position by consistent high-speed CBIR for multiple parameter settings, using LSH and GPU. We also showed that the accuracy of position estimation can be improved using spatial locality while keeping processing speed. Then, we discussed usefulness of our proposed method for position estimation application.

We introduce two problems as future work. One is the performance evaluation on the multiple videos, which are different in conditions such as lighting or weather. In position estimation applications, it is assumed that multiple videos are used, which are different in conditions, in order to improve the accuracy, while we examined the performance on the only one video in this paper. Therefore the evaluation is important in the point of view of practicality. The other is improvement of data structure and compression. If multiple videos are used, it may suffer from lack of capacity of device memory. To address this problem, it is necessary to reduce the data size or reading the data only when needed.

References

[1] Kameda, Y., and Ohta, Y. (2010). Image retrieval of first-person vision for pedestrian navigation in urban area. In *20th International Conference on Pattern Recognition (ICPR)*, pp. 364–367.

[2] Kurata, T., Kourogi, M., Ishikawa, T., Kameda, Y., Aoki, K., and Ishikawa, J. (2011). Indoor-outdoor navigation system for visually-impaired pedestrians: Preliminary evaluation of position measurement and obstacle display. In *15th Annual International Symposium on Wearable Computers (ISWC)*, pp. 123–124.

[3] Takizawa, H., Orita, K., Aoyagi, M., Ezaki, N., and Mizuno, S. (2017). A Spot Reminder System for the Visually Impaired Based on a Smartphone Camera. *Sensors*, 17(2), 291.

[4] Lowe, D. G. (1999). Object recognition from local scale-invariant features. In *Proceedings of the 7th International Conference on Computer Vision (ICCV 1999)*, pp. 1150–1157.

[5] Bay, H., Tuytelaars, T., and Van Gool, L. (2006). Surf: Speeded up robust features. In *European conference on computer vision*, (pp. 404–417). Springer, Berlin, Heidelberg.

[6] Owens, J. D., Houston, M., Luebke, D., Green, S., Stone, J. E., and Phillips, J. C. (2008). GPU computing. In *Proceedings of the IEEE*, 96(5), 879–899.

[7] Indyk, P., and Motwani, R. (1998). Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*, pp. 604–613.

[8] Baeza-Yates, R., and Ribeiro-Neto, B. (2011). *Modern Information Retrieval: The Concepts and Technology Behind Search*, volume 2. Addison Wesley: Boston.

[9] Alcantarilla, P. F., Nuevo, J., and Bartoli, A. (2013). Fast explicit diffusion for accelerated features in nonlinear scale spaces. In *Proceedings of the British Machine Vision Conference (BMVC 2013)*, pp. 1–11.

[10] Cheng, J., Leng, C., Wu, J., Cui, H., and Lu, H. (2014). Fast and accurate image matching with cascade hashing for 3d reconstruction. In *2014 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 1–8.

[11] Andoni, A., and Indyk, P. (2008). Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Commun. ACM*, 51(1), 117–122.

[12] Datar, M., Immorlica, N., Indyk, P., and Mirrokni, V. S. (2004). Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the Twentieth Annual Symposium on Computational Geometry*, pp. 253–262.

[13] Guttman, A. (1984). R-trees: A dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, *SIGMOD '84*, pp. 47–57, New York, NY, USA.

[14] Kamasaka, K. and Kitahara, I., and Kameda, Y. (2017). Image based location estimation for walking out of visual impaired person. In *Proceedings of the 14th European Conference on the Advancement of Assistive Technology, AAATE Conf. 2017, Sheffield, UK, September 12–15*, pp. 709–716.

[15] Sivic, J., and Zisserman, A. (2003). Video google: A text retrieval approach to object matching in videos. In *Proceedings of the 9th IEEE International Conference on Computer Vision (ICCV 2003)*, pp. 1470–1477.

[16] Cevahir, A., and Torii, J. (2012). GPU-enabled high performance online visual search with high accuracy. In *2012 IEEE International Symposium on Multimedia (ISM)*, pp. 413–420.

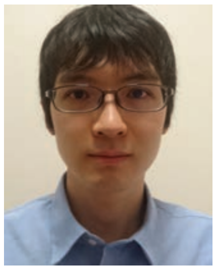
[17] Chandrasekhar, V., et al. (2010). Survey of SIFT compression schemes. In *Proceedings of the International Workshop Mobile Multimedia Processing*, pp. 35–40.

Biographies

Yuta Kusamura received B.Eng. and M.Eng. from University of Tsukuba in 2016 and 2018, respectively. His research interest covers accelerating content-based image retrieval using GPU.

Toshiyuki Amagasa received B.E., M.E., and Ph.D degrees from the Department of Computer Science, Gunma University in 1994, 1996, and 1999, respectively. Currently, he is a professor at the Center for Computational Sciences, University of Tsukuba. His research interests cover database systems, Web mining, and database application in scientific applications. He is a senior member of IEICE and IEEE, and a member of IPSJ, DBSJ, and ACM.

Hiroyuki Kitagawa received the B.Sc. degree in physics and the M.Sc. and Dr.Sc. degrees in computer science, all from the University of Tokyo. He is currently a full professor at Center for Computational Sciences, University of Tsukuba. His research interests include databases, data integration, data mining, stream processing, information retrieval, and scientific databases. He served as President of DBSJ and Chairperson of ACM SIGMOD Japan Chapter. He is a Fellow of IPSJ and IEICE, and a member of ACM, IEEE, and JSSST.



Yusuke Kozawa is a postdoctoral researcher at Artificial Intelligence Research Center, National Institute of Advanced Industrial Science and Technology (AIST), Japan. He received B.Inf.Sc., M.Eng., and Ph.D. degrees from University of Tsukuba, Japan, in 2011, 2013, and 2016, respectively. His research interests include databases, data mining, and parallel computing, especially GPU computing and its application to data analysis.