DEIM Forum 2018 I2-5

# GPUを用いた類似画像検索による歩行者位置推定の高速化

草村　優太†　　天笠　俊之††　　北川　博之††　　小澤　佑介†††

† 筑波大学大学院システム情報工学研究科　〒305–8573 茨城県つくば市天王台 1–1–1
†† 筑波大学計算科学研究センター　〒305–8573 茨城県つくば市天王台 1–1–1
††† 産業技術総合研究所人工知能研究センター　〒130–0064 東京都江東区青海 2–4–7
E-mail: †kusamura@kde.cs.tsukuba.ac.jp, ††{amagasa,kitagawa}@cs.tsukuba.ac.jp, †††yusuke.kozawa@aist.go.jp

**あらまし**　類似画像検索は与えられたクエリ画像に類似した画像をデータベース中から検索する技術である．画像データベースに事前に正確な位置情報を与えておくことにより類似画像検索を歩行者位置推定に利用できる．歩行者支援においてはリアルタイムな位置推定が要求されるが，類似画像検索では画像から抽出される特徴ベクトルを用いる手法が主流であり，そのベクトルが高次元であることとベクトル数が膨大であるために，類似画像検索の処理コストは大きい．そこで，本研究では GPU を用いた類似画像検索の高速化手法を提案する．具体的には，GPU での並列化に適したデータ構造とアルゴリズムを提案し，効率的な類似画像検索を実現する．また，歩行者の空間局所性を用いて検索の精度を向上させる．さらに，動画データを用いた実験により，提案手法の性能を評価する．

**キーワード**　GPU，SIFT，LSH，類似画像検索，歩行者位置推定

## 1. 序　　論

スマートフォンをはじめとするモバイルデバイスの普及により，位置情報は様々なアプリケーションで広く用いられている．そのようなデバイスの位置情報の取得には，GPS や電波強度を用いる手法が主流である．しかし，このような方法では，ビルが林立する都市部では測定誤差が大きくなることや，地下では利用が出来ない等の問題がある．

デバイスの地理的位置情報を取得することにより，歩行者支援を行うことができ，Kameda と Ohta [11] は都心においても利用可能な歩行者支援のための類似画像検索手法を提案した．これは特に，視覚障がい者のサポートに有用であり，実アプリケーションとしては屋内外での歩行者支援 [12] や場所通知システム [15] が挙げられる．この手法は，カメラを胸や頭に装着した状態で撮影される一人称視点の画像を利用する事前に日常において頻繁に移動する経路を位置情報付きのビデオ映像で撮影しておき，各フレームとその位置情報をデータベースに格納しておく．位置推定は，歩行者が撮影した現在のカメラ映像に最も類似した画像をデータベース上から検索し，その位置情報を取得することで行う．

このようなアプリケーションにおいては，膨大なビデオ映像と位置情報が必要となるため，データベースはサーバー上で保持されることが想定される．位置推定の際には，まず歩行者が撮影したカメラ映像をクエリ画像としてサーバーに送信する．その後，サーバー上でデータベースとの照合を行い，得られた位置情報をユーザーに送信する．このように，サーバー上でクエリを処理することにより，高速な処理が可能となりリアルタイムに位置情報を取得可能となる．

類似画像検索では画像間の類似度計算に特徴ベクトルを用いる手法が主流である．特徴ベクトルは数千個程度のベクトルを用いて画像の特徴を表現する．類似画像検索はこれらのベクトルの比較により実現可能である．しかし，画像データベース全体では特徴ベクトル数が莫大となることに加え，特徴ベクトルが高次元であるため，検索には多大な処理コストを要する．

一方，近年では莫大なデータを扱う処理の高速化のために GPU (graphics processing unit) を用いた並列コンピューティングが注目されている．GPU は従来グラフィック処理に用いられるプロセッサであるが，多数のコアを搭載し高い並列処理性を持つという特徴がある．この特徴を活かして GPU を汎用計算に用いる技術のことを GPGPU (general-purpose computing on graphics processing units) と呼び，様々なアプリケーションにおいて処理の高速化に貢献している [14].

そこで，本研究では GPU を用いた類似画像検索による歩行者位置推定の高速化手法を提案する．本研究の基本的なアイディアは，1) GPU に適したデータ構造とアルゴリズムの提案，2) 特徴ベクトルの圧縮，3) 歩行者の空間局所性の利用，である．また，本研究の提案手法はデータベース構築とクエリ処理の二つの処理に大別できる．データベース構築では，効率的な検索のために事前処理可能な処理を行い，クエリ処理で実際に与えられたクエリ画像に基づき位置情報を推定する．

GPU の高い並列処理性能を活かすためには適切なアルゴリズムとプログラミングが必要であるため，GPU 上での効率的な画像マッチングアルゴリズムを提案する．特に，単純な手法では GPU の複数スレッドによる書き込みの衝突が発生し，これが性能低下の要因となる．これを防ぐような効率的なデータ構造とアルゴリズムを提案し，GPU の性能を活かした検索を行う．

また，GPU 上での処理において，データの圧縮も重要であるため，特徴ベクトルの圧縮を行う．GPU 上で処理を行うためには，デバイスメモリにデータを格納させておく必要があるが，その容量はメインメモリと比較して低容量である．そのため，GPU に載せるデータの圧縮が重要となる．特徴ベクトルの圧縮方法は複数存在するが，本研究では並列処理に適した

LSH (locality sensitive hashing) [9] を用いてデータの圧縮を行う．LSH は類似ベクトルを高確率で同一のハッシュ値に変換するハッシュ法であり，特に類似検索の高速化等に用いられる．提案手法では，データベース構築とクエリ処理において，それぞれ，画像データベースとクエリ画像の特徴ベクトル圧縮を行う．

さらに，実アプリケーションを想定し，歩行者の空間局所性を用いた検索を行う．歩行者位置推定において，位置推定は連続的に行われることが想定されるため，歩行者の位置も連続的となる．すなわち，歩行者の現在位置がある程度事前に推定可能である．そこで，推定された位置周辺の画像のみを検索候補とすることで検索精度を向上させる．

本研究では，提案手法の性能を評価するために評価実験を行なった．LSH と GPU を用いることにより，複数の条件に対して一貫して高速な類似画像検索による歩行者位置推定が可能であることを示した．また，空間局所性を用いることにより，処理速度を保ちながら位置推定精度が向上できることを示した．さらに，提案手法の有用性についての考察を行なった．

## 2. 前提知識

本節では，本研究に関連する前提知識について説明する．2.1 節では本研究で扱う類似画像検索による歩行者位置推定について述べ，2.2 節では GPU コンピューティングについて述べる．

### 2.1 類似画像検索による歩行者位置推定

画像データベース中からクエリ画像と視覚的に類似した画像を検索する処理のことを類似画像検索 [4] と呼ぶ．Kameda と Ohta [11] は歩行者支援のための類似画像検索手法を提案した．この手法では，歩行者が撮影した現在のカメラ映像に最も類似した画像を位置情報付きの画像データベース中から検索し，その検索結果の画像に対応した位置情報を取得する．歩行者支援を目的とした類似画像検索では最もクエリ画像にマッチする上位 1 件の検索結果のみが必要であるため，本研究で扱う類似画像検索による歩行者位置推定を以下のように定義する．

**定義 1.** 入力として位置情報付きの画像データベースと一つのクエリ画像が与えられた際，クエリ画像と最も類似した画像を画像データベース中から見つけ出し，その画像に付与されている位置情報を結果として出力する．□

画像間の類似度を計算するために，近年の類似画像検索では SURF [5] や AKAZE [1] といった局所特徴量を用いる手法が主流である．特に，SIFT (scale-invariant feature transform) [13] は，最も基本的な特徴量として画像認識の分野で広く用いられている．SIFT は，画像の特徴を複数の 128 次元ベクトルで表現し，照明変化，拡大縮小，回転に対して堅固であるという特徴を持つ．そこで，本研究では SIFT 特徴量を用いた類似画像検索の高速化手法を提案し，歩行者位置推定の高速化を目指す．

#### 2.1.1 SIFT マッチングに基づく類似画像検索

SIFT 特徴量を用いた単純な画像検索 [2] は，データベース構築とクエリ処理の二つの処理に大別される．

まず，データベース構築について述べる．画像データベースが与えられると，画像データベース中の各画像から SIFT 特徴量を抽出する．次に，抽出されたそれぞれの特徴ベクトル $\boldsymbol{f}$ と，そのベクトルが抽出された画像 ID を示す値 $id$ のタプル $(\boldsymbol{f}, id)$ を，データベース $D$ に格納する．このとき，$id(\boldsymbol{f})$ を特徴ベクトル $\boldsymbol{f}$ に対応する画像 ID として定義する．

**Algorithm 1** SIFT マッチングを用いた類似画像検索．

```
Input: Q, D, num_images (= データベース中の画像数．)
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(q_i, f_j)
 7:         if dist < min_dist then
 8:             min_id = id(f_j)
 9:             min_dist = dist
10:     freq[min_id] + +
11: result_id = argmax(freq)
```

クエリ処理では $D$ に基づいて，与えられたクエリ画像と最も類似した画像を返す．具体的には，まず，クエリ画像から特徴ベクトルを抽出し，各特徴ベクトルの最近傍ベクトルをデータベース中から検索する．最近傍ベクトルが得られたら，それらのベクトルが得られた画像 ID を得る．最後に，得られた画像 ID の個数を集計し，最も多く得られた画像 ID が類似画像検索の結果となる．この処理は Algorithm 1 で示される．ただし，`dist` は二つのベクトル間の距離を計算する関数であり，`argmax` は配列中の最も大きな値を持つ番地を返す関数である．

#### 2.1.2 LSH を用いた SIFT マッチングに基づく類似画像検索

LSH (locality sensitive hashing) [9] は，ハッシュ法の一つであり，類似ベクトルを高確率で同一ハッシュ値に変換する．類似画像検索において莫大な画像を扱う場合，特徴ベクトル数も莫大となる．そのため，LSH はデータ圧縮や特徴マッチングの高速化などに利用される [6].

LSH では，複数のハッシュ関数 $h_i()$ $(0 \leqq i < k)$ を用いてハッシュ値が計算され，そのハッシュ関数は $H() = (h_0(), h_1(), \ldots, h_{k-1}())$ で表される．すなわち，ベクトル $\boldsymbol{v}$ は，ハッシュ値 $\boldsymbol{v}' = H(\boldsymbol{v}) = (h_0(\boldsymbol{v}), h_1(\boldsymbol{v}), \ldots, h_{k-1}(\boldsymbol{v}))$ に変換される．$h_i()$ は，確率的に生成され，その生成方法は圧縮するベクトル空間の距離指標によって異なる．現在では，$l_p$ ノルム，コサイン類似度，ハミング距離など，様々な距離指標に対するハッシュ関数が提案され利用されている [3].

SIFT 特徴ベクトル間の距離計算には一般的に $l_2$ ノルムを用いるため，本研究では $l_2$ ノルムに対する LSH [7] を用いる．$l_2$ ノルムに対する LSH では，三つの変数 $\boldsymbol{a}$，$b$，$W$ に基づいてハッシュ値を計算する．$\boldsymbol{a}$ は $d\ (= |\boldsymbol{v}|)$ 次元のベクトルであり，ガウス分布から選択された $d$ 個の確率変数を要素とする．$W$ は $W > 0$ となる任意の実数であり，$b$ は半閉区間 $[0, W)$ の一様分布から選択される実数である．ハッシュ関数 $h_i()$ はこれらを用いて，以下の式で表される．

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

これは，元空間が $\boldsymbol{a}$ に直行する複数の超平面により等幅に分割され，同一部分空間に属しているベクトルが同一ハッシュ値となることを意味する．この時，$W$ の値はこのハッシュ関数のパラメータとなり，同一ハッシュ値に変換される部分空間の領域

サイズを決定する．具体的には，$W$ が大きいほど領域サイズが大きくなり，同一ハッシュ値に変換されるベクトル数が多くなる．すなわち，近傍検索の曖昧性が高くなる．

LSH を用いることで，類似画像検索における SIFT 特徴ベクトルの最近傍検索をハッシュ値の一致検索で代用可能となる．LSH の SIFT マッチングを用いた類似画像検索 (2.1.1 節) への適用は，検索手順を拡張することで可能となる．

データベース構築では，特徴ベクトルと画像 ID のタプル $(\boldsymbol{f}_i, id)$ を得た後に各特徴ベクトルに LSH を適用する．すなわち，特徴ベクトル $\boldsymbol{f}_i$ と LSH のハッシュ関数 $H()$ に対して，ハッシュ値 $\boldsymbol{f}'_i = H(\boldsymbol{f_i})$ を得る．その後，データベース $D'$ としてハッシュ値をキーとしそれに対応する画像 ID の多重集合を値とする連想配列を格納する．このとき，異なる特徴ベクトルから同一のハッシュ値が得られた場合は，それらに対応するすべての画像 ID を保持する多重集合が連想配列の値となる．

クエリ処理では，クエリ画像から特徴量 $Q$ が抽出されるとデータベース構築と同様に各特徴ベクトル $\boldsymbol{q} \in Q$ に LSH を適用して $\boldsymbol{q'} \in Q'$ とする．ここで，得られた $\boldsymbol{q'}$ に対して，$D'$ のキーから $\boldsymbol{q'} = \boldsymbol{f'}$ となるような $\boldsymbol{f'}$ を検索し，対応する画像 ID を取得する．この処理で最頻出の画像 ID が検索結果となる．

### 2.2 GPU コンピューティング

GPU は従来グラフィック処理向けに利用されるプロセッサである．GPU を汎用計算に用いる技術のことを，GPGPU と呼ぶ．GPU には，CPU と比較して多くのコアが搭載されており，高い並列処理性を持つという特徴がある．そのため，GPGPU は科学技術計算やデータベースクエリ処理など様々なアプリケーションにおいて処理の高速化に貢献している [14]．特に，NVIDIA 社の GPU 及びその統合開発環境である CUDA(注1) は GPU コンピューティングにおいて広く用いられており，本研究でもこれらを用いる．

GPU アーキテクチャが独特であるため，GPU プログラミングは CPU と比べて大きく異なる．一般に，GPU は並列処理性能に優れているため CPU よりも高速な処理が可能であることが多いが，不適切な実装では GPU の性能を活かしきれずに CPU よりも低速になることがある．すなわち，高速化には適切なデータ構造の設計とアルゴリズムが必要である．

GPU は階層的なプロセッサ構成を持つ．一つの GPU には複数のストリーミングマルチプロセッサと呼ばれる演算部が搭載され，共有メモリと複数の CUDA コアを有する．CUDA コアは GPU で最小の演算装置であり，それぞれがレジスタファイルを有する．

プログラミングモデルも階層的な構造であり，スレッド，ブロック，グリッドの処理単位で構成される．スレッドは GPU での最小の処理単位であり，CUDA コアにより処理される．複数のスレッドは同時に扱われ，その処理単位をブロックと呼ぶ．ブロックはストリーミングマルチプロセッサ単位で処理され，その共有メモリなどの計算資源を共有する．さらに複数のブロックはグリッドと呼ばれる最大の処理単位でまとめられる．

並列処理において頻出する処理はデータ並列プリミティブと呼ばれ，GPU プログラミングにおいて非常に重要である．本研究では，以下に示す三つのデータ並列プリミティブを利用する．これらのデータ並列プリミティブは Thrust ライブラリ (注2) として実装されており，高速な並列処理が可能である．

**reduce**: 配列 $[a_0, a_1, \ldots, a_{n-1}]$ と二項演算子 $\oplus$ に対して，スカラ値 $a_0 \oplus a_1 \oplus \ldots \oplus a_{n-1}$ を返す．

**scan**: 配列 $[a_0, a_1, \ldots, a_{n-1}]$ と二項演算子 $\oplus$，単位元 $I$ に対して，配列 $[I, a_0, a_0 \oplus a_1, \ldots, a_0 \oplus a_1 \oplus \ldots \oplus a_{n-2}]$ を返す．

**sort**: 配列 $[a_0, a_1, \ldots, a_{n-2}]$ を規則に基づき並べ替える．

## 3. 提案手法

提案手法は，入力としてクエリ画像と位置情報つきの画像データベースが与えられた時に，クエリ画像が撮影されたと推定される場所の位置情報を出力する．位置推定は，類似画像検索を用いてクエリ画像に類似する画像を 1 件取得し，その画像に紐付けられた位置情報を参照することにより行う．この処理においては，類似画像検索の処理コストが特に大きい．そのため，2.1.2 節で述べた LSH を用いた SIFT マッチングに基づく類似画像検索の GPU 上での効率的なアルゴリズムを提案する．

提案手法は 2.1.2 節の手法と同様にデータベース構築とクエリ処理の二つの処理に分けられる．ここで，データベース構築は事前処理が可能であるため，クエリ処理の高速化を目指す．

本手法では歩行者の空間局所性を利用する．歩行者位置推定においては，クエリ画像は連続的に入力されることが想定される．そのため，連続する二つのクエリを考えた際に，それらのクエリ画像が撮影された地点は非常に近くなる．本研究では，これを歩行者の空間局所性と定義し，この条件を前提とした手法を提案する．空間局所性を利用するために，クエリ画像には撮影された地点付近の位置情報が付与されていることを想定する．前述のように，クエリ画像は連続的に入力されるため，直前の位置推定結果をこの位置情報に用いることが可能である．

本研究では，図 1 のように位置情報に対する R 木 [8] を用いてクエリ周辺で撮影された画像 (以下，近傍画像とする) を検出しておき，それに該当しない画像を検索結果から除外する．R 木は空間インデックスのひとつであり，与えられた点に対する高速な範囲検索をサポートする．R 木はデータベース構築において構築され，R 木を用いたクエリ周辺の画像の検出と検索結果の除外処理はクエリ処理において行われる．

以下では，データベース構築について 3.1 節で説明し，クエリ処理について 3.2 節で説明する．

### 3.1 データベース構築

データベース構築では，画像データベースを GPU に適したデータ構造に変換した上で GPU に転送する．その処理では，まず 2.1.2 節と同様にデータベース $D'$ を作成する．その後，$D'$ を元に以下の配列を作成し GPU に転送する．これは，配列が

(注1)：Parallel Programming and Computing Platform | CUDA | NVIDIA www.nvidia.com/cuda

(注2)：Thrust :: CUDA Toolkit Documentation http://docs.nvidia.com/cuda/thrust/

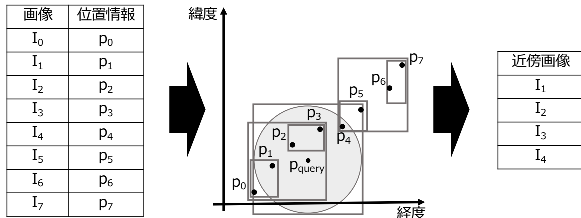


図 1: R 木を用いた近傍画像の取得.

GPU 上での処理に適したデータ構造であるためである.

$f_val$: $D'$ 中の全ての $\boldsymbol{f}'_i$ を要素とする配列.クエリ処理でこの配列に対する二分探索を行うため,要素は昇順でソートして格納しておく.

$id_val$: $D'$ 中の全ての画像 ID を $f_val$ に対応する順番で格納した配列.

$id_ptr$: $f_val$ と $id_val$ の対応関係を示す配列.具体的には,$id_ptr[i]$ が $id_val$ の先頭レコードからのオフセットを示す.

さらに,空間局所性を利用するために,位置情報に対する R 木を構築しておく.R 木のデータ点として,画像データベース中の全ての画像の位置情報を格納しておき,その画像 ID を各データ点のラベルとする.これにより,クエリ処理において,クエリに対する近傍画像を高速に検出することが可能となる.

### 3.2 クエリ処理

クエリ処理では,データベース中からクエリ画像に最も類似した画像を検索し,その画像に付与された位置情報を出力する.ただし,その際にはクエリに対する近傍画像を検出し,該当する画像を検索結果の候補とした類似画像検索により類似の画像を検索する.その処理では,まずクエリ画像から特徴ベクトル $\boldsymbol{q}$ $(\in Q)$ を CPU により抽出し,GPU 上のグローバルメモリに転送する.全ての $\boldsymbol{q}$ に対して,GPU 上で並列に LSH を適用し,ハッシュ値 $\boldsymbol{q}'$ に圧縮される.圧縮後,GPU 上でハッシュ値とデータベースとのマッチングを並列に行う.ここで,照合の結果が類似画像検索の検索結果となり,それに付与された位置情報を結果として出力する.本研究では,特徴ベクトルの圧縮とマッチングを GPU 上で効率的に行うアルゴリズムを提案する.それぞれの詳細について以下に示す.

#### 3.2.1 特徴ベクトルの圧縮

特徴ベクトルの圧縮では,GPU 上のグローバルメモリに転送されたクエリの各特徴ベクトルをハッシュ値へ変換する.一つのクエリ画像からは数千程度の 128 次元特徴ベクトル $Q$ が抽出されるが,それぞれのベクトルの圧縮は独立に処理が可能である.また,一つの特徴ベクトル $\boldsymbol{q_i} \in Q$ の変換に着目すると,2.1.2 節で述べたハッシュ値 $h_j(\boldsymbol{q_i})$ の計算が必要であるが,それらもそれぞれ独立に処理可能である.そこで,各特徴ベクトル $\boldsymbol{q_i}$ の変換毎にブロック並列,さらに各ハッシュ値 $h_j(\boldsymbol{q_i})$ の計算毎にスレッド並列で行う.これにより,GPU の並列処理性能を活かすことができるため,高速な処理が可能である.

#### 3.2.2 マッチング

データベースとの照合では,クエリから得られたハッシュ値と同一のハッシュ値を $f_val$ から検索し,一致したハッシュ値に対応する画像 ID を $id_val$ と $id_ptr$ を元に取得する.この処理で,最も多く得られた画像 ID を検索結果とする.

この処理は,二分探索と `atomicAdd` を用いた特徴ベクトル毎のスレッド並列での処理がナイーブな実装である.二分探索は $id_val$ からのハッシュ値の検索に用い,`atomicAdd` は検索されたハッシュ値に対応する画像 ID の出現頻度集計に用いる.`atomicAdd` とは,同一アドレスへの書き込みの衝突を回避可能な,排他制御を持つ加算関数である.そのため,二つ以上のスレッドからの同一アドレスへの書き込みを行う際に用いられる.出現頻度の集計は,GPU のグローバルメモリや共有メモリに暫定の出現頻度カウンタ配列を確保した上で,検出された画像 ID に対して,逐次配列要素をインクリメントすることにより実現できる.しかし,`atomicAdd` によるグローバルメモリへの書き込みは異なるスレッドによる同一アドレスへのアクセスが制限されるため,低速な処理の原因となる.また,共有メモリに対して `atomicAdd` を行う事により,比較的高速な処理が可能であるが,扱う画像数が多い場合,メモリが不足する.

そこで,本研究ではこの照合処理を `atomicAdd` を用いずに高速に行う方法を提案する.照合処理は,一致リストの取得と最頻出 ID の検出に分けられる.一致リストの取得とは,クエリから得られたハッシュ値に一致するハッシュ値をデータベースから検出し,それに対応する全ての画像 ID のリスト (以下,一致リスト) を得る処理である.また,最頻出 ID の検出は,一致リストの中で最頻出の画像 ID (以下,最頻出 ID) を検出する処理である.

a) 一致リストの取得

一致リストは,適切な $id_val$ の値の配列 $M$ へのコピーによって取得可能である.その処理を Algorithm 2 により効率的に行う.GPU のブロック並列で $M$ を取得するために,1) 各ブロックがコピーする $id_val$ の値の先頭インデックス,2) コピーサイズ,3) 書き込み先の先頭インデックス,を事前に計算しておく.以降は,これらを格納した配列をそれぞれ $F, N, T$ とする.これらを用いて $M$ を作成する.これにより,`atomicAdd` を使わずに効率的な一致画像 ID の検出が可能となる.以下で,$F, N, T$ の作成と,$F, N, T$ を用いた $M$ の作成の詳細を説明する.

$F, N, T$ の作成では,各クエリハッシュ値に対して $f_val$ の中から一致するハッシュ値を二分探索で検索し,マッチした要素のインデックスを $index$ とする (2 行目).このとき,二分探索がマッチしなかった場合,$index$ は $-1$ とする.コピー元の値の先頭インデックスを示す配列 $F$ は,$id_ptr[index]$ の値に基づいて計算される (3〜6 行目).コピーサイズを示す配列 $N$ は,$id_ptr[index]$ と $id_ptr[index+1]$ の差により計算される (7〜10 行目).これらの処理は,各クエリのハッシュ値に対してスレッド並列で処理を行う.$T$ は,$N$ に対して scan プリミティブを適用することにより作成される (11 行目).また,図 2 は,これらの配列の作成例を示している.

$M$ は,$F, N, T$ をもとに,以下の式により計算可能である.

$$M[T[i]:T[i]+N[i]-1] = id_val[F[i]:F[i]+N[i]-1]$$

これは,$id_val$ の $F[i]$ 番目から $N[i]$ 個の値を $M$ の $T[i]$ 番目

**Algorithm 2** 一致リストの取得.

```
Input: Q', f_val, id_ptr, id_val, neighbors = 近傍画像 ID 集合
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:     if index == -1 then
 8:         N[i] = 0
 9:     else
10:         N[i] = id_ptr[index + 1] - id_ptr[index]
11: T = scan(N)
12: for i = 0 : Q'.size() - 1 do in block parallel
13:     for j = 0 : N[i] - 1 do in thread parallel
14:         if neighbors.in(id_val[F[i] + j]) then
15:             M[T[i] + j] = id_val[F[i] + j]
16:         else
17:             M[T[i] + j] = NA
```

**Algorithm 3** 最頻出画像 ID の検出.

```
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: B[M'.size() - 1] = 1
 8: M' と B のサイズを last + 1 に縮小
 9: I = scan(B)
10: for i = 0 : B.size() - 1 do in thread parallel
11:     if B[i] == 1 then
12:         O[I[i]] = i
13: S[0] = O[0] + 1
14: for i = 1 : O.size() - 1 do in thread parallel
15:     S[i] = O[i] - O[i - 1]
16: r = reduce_with_index(S)
17: id = M'[O[r]]
```

から $N[i]$ 個の要素としてコピーすることを表している．ただし，空間局所性を利用するために，この時の画像 ID の値が近傍画像に含まれていない場合は，その値を “NA” (=INT_MAX) として置き換える．これを各 $i$ に対してブロック並列，コピーする各値に対してスレッド並列で行う．この時，$F$ によって書き込み先の先頭インデックスとサイズが事前に判明しているため書き込みの衝突が起こらず，高速な並列処理が可能である．この処理は 12〜17 行目に示され，図 3 はこの処理の例である．

b） 最頻出 ID の検出

最頻出 ID の検出では，一致リスト $M$ の中で最頻出の画像 ID を検出する．これは，$M$ をソートした上で，同一の要素が最も連続する箇所を検出し，その長さを得ることにより可能である．説明のため，以下で，**境界**と**区間**を定義する．まず，隣接する要素が異なる箇所を境界として定義する．また，二つの境界間の部分配列のことを区間と定義し，同一の区間に属する要素は全て同一の値を持つ．

最頻出 ID は Algorithm 3 によって検出される．例を図 4 に示す. まず，$M$ をソートし，$M'$ とする (1 行目)．次に，$M'$ の境界位置を “1” として表す配列 $B$ を作成する (4 行目，7 行目)．この時，境界位置の右側の要素が NA となる境界位置を検出し，$last$ とする (2 行目，5〜6 行目)．$last$ を用いて $M'$ と $B$ のうち，NA に該当する箇所を削減する (8 行目)．その後，$B$ に scan プリミティブを適用し，各要素が属する区間番号配列を示す配列 $I$ を作成する (9 行目)．さらに，$B$ に基づき境界位置を検出し，各境界位置に対してそのインデックスを配列 $O$ に格納する (10〜12 行目)．この際，$O$ への書き込み位置は $I$ を参照することにより特定可能であり，$O$ は各区間の終端要素のインデックスを示す．最後に，$O$ の隣接要素の差を計算することにより，各区間のサイズを表す配列 $S$ が作成される (13〜15 行目)．これらの処理は，それぞれの配列作成の各要素の計算が独立に行えるため，スレッド並列での処理が可能である．加えて，sort や scan といった GPU 向けのライブラリが存在するプリミティブを利用しているため，GPU 上での高速な処理が可能である．

これまでの処理により，各区間のサイズを検出できたが，ここで，サイズが最も大きくなる区間に属している画像 ID が最頻出画像 ID となる．そこで，そのような区間番号を検出するために，reduce プリミティブを適用する．しかし，GPU 向けのライブラリには値が最大値をもつ配列インデックスを検出するプリミティブは存在しないため，単純な reduce 処理 (注3) を拡張した関数 `reduce_with_index` を実装した．この関数を利用して，区間のサイズが最大となる区間番号 $r$ を検出する (16 行目)．このとき，区間 $r$ に属している画像 ID は $M'$ と $O$ により計算可能であり (17 行目)，これが最頻出の画像 ID となる．

## 4. 評価実験

本研究では，提案手法の性能を速度及び精度の観点から評価するために評価実験を行った.

### 4.1 データセット

本研究では，位置情報として GPS 情報が付与された独自の動画データセットを用いた．このデータセットは二つの動画により構成されており，それぞれの動画は同じ日のほぼ同時刻に撮影されている．動画の撮影はビデオカメラを胸部前方に装着した状態で，筑波大学内のペデストリアンデッキを経路として歩行しながら行なった．撮影には，OLYMPUS STYLUS TG-Tracker を使用した．このカメラには GPS センサが搭載されており，2 秒毎に位置情報が取得され，該当するフレームに位置情報が付与される.

歩行者位置推定の実アプリケーションを想定し，二つの動画はほぼ同一経路，同時刻に撮影されている．一つ目の動画は 26,401 フレーム，二つ目の動画は 25,980 フレームからなる動画である．実験においては，前者の動画を画像データベースとして用い，後者の動画をクエリセットとして用いる．その経路は，筑波大学内のペデストリアンデッキに沿った経路である.

TG-Tracker では 2 秒毎の位置情報のみが取得可能なため，ビデオカメラによって直接位置情報が付与されないフレームが存在する．そのため，本研究では線形補間によって位置情報を補完し，動画中の全てのフレームに位置情報を付与した.

### 4.2 実験環境

実験には GPU と二つの CPU を搭載しているマシンを用いた.

(注3)：Optimizing Parallel Reduction in CUDA http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/reduction/doc/reduction.pdf

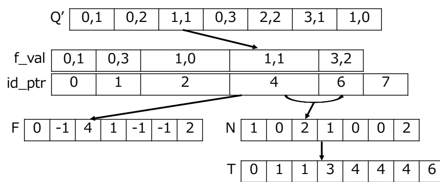


図 2: 配列 F，N，T の作成.

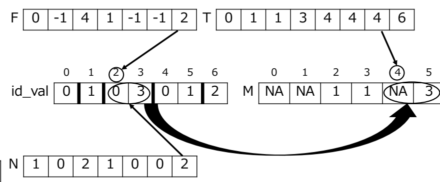


図 3: F，N，T に基づく一致リストの生成.

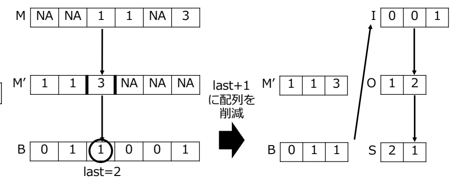


図 4: 最頻出画像 ID の検出.

CPU は Intel Xeon Processor E5-2687W v2 (8 コア，3.4GHz) であり，128GB のメインメモリを搭載している．GPU は NVIDIA Tesla K40 (2,880 コア，745MHz) を搭載しており，そのデバイスメモリは 12GB である．OS は CentOS 6.7 (final) である．

全てのプログラムは C++及び CUDA を用いて記述した．コンパイラは nvcc 8.0 を用いて，最適化オプションとして-O3 を用いた．また，CPU での並列化のために OpenMP(注4) を用い，画像からの SIFT 特徴ベクトルのために OpenCV(注5) を用いた．

### 4.3 比較手法

性能評価のために，以下の 6 つの比較手法を実装した．

**Raw-SIFT-based method (RS)**: 非圧縮の SIFT 特徴ベクトルを用いた CPU 上で動作するナイーブな類似画像検索手法．このプログラムは CPU 32 スレッドで実行した．

**LSH-based method @ CPU (LC)**: LSH により圧縮された SIFT 特徴ベクトルを用いた CPU 上で動作する類似画像検索手法．このプログラムは CPU 1 スレッド (LC_Serial) と 32 スレッド (LC_Parallel) の両方で実行した．

**LSH-based method @ GPU with atomicAdd (LG with Atomic)**: LSH により圧縮された SIFT 特徴ベクトルを用いた GPU 上で動作する類似画像検索手法．ただし，マッチング処理において `atmicAdd` 関数を用いて画像 ID の出現頻度を集計する．

**LSH-based method @ GPU (LG)**: LSH により圧縮された SIFT 特徴ベクトルを用いた GPU 上で動作する類似画像検索手法．マッチング処理において `atmicAdd` 関数を使用しないアルゴリズムによりスレッドの衝突を回避する．

**LSH-based method @ CPU with spatial-locality (LC with SL)**: LC をベースとして，歩行者の空間局所性を考慮することにより拡張された類似画像検索手法．ただし，非近傍画像の除外は画像 ID を全て集計した後に行う．

**LSH-based method @ GPU with spatial-locality (LG with SL)**: LG をベースとして，歩行者の空間局所性を考慮することにより拡張された類似画像検索手法.

### 4.4 実験方法

以下に示す実験方法で，4 つの実験を行なった．

#### 4.4.1 実 験 1

検索速度を検証するために，データベース中から複数のクエリを検索し，その平均時間を計測した．検索手法は，空間局所性を用いない全ての手法，すなわち，RS，LC，LG with Atomic，LG のそれぞれを用いた．クエリセット中からランダムに選択した 100 件をクエリとして検索を行なった．

#### 4.4.2 実 験 2

LSH を用いることでデータ量が削減され，検索精度が低下することが予想されるため，その検索精度を検証する．その検証は，実アプリケーションを想定して位置情報を用いて行う．クエリ画像に付与された位置情報と推定された位置情報の差を計算し，その差が 15m 以内であった場合を検索成功として，その検索成功率を求めた．比較手法としては，LSH を用いない手法である RS と，LSH を用いた手法である LG を用いた．なお，LC，LG with Atomic，LG，の検索結果は全て同一である．

#### 4.4.3 実 験 3

空間局所性を用いたことによる検索精度の変化を検証するために，クエリを連続的に入力した際の位置推定結果を検証した．その検証は Kameda と Ohta [11] の実験手法を参考に行なった．比較手法として，LG と LG with SL を用いてその結果を比較した．直前の検索結果の位置情報をクエリと同時に入力として与えた．また，正解の位置情報は予めクエリセットに付与されている位置情報とした．この時，推定位置と正解位置のそれぞれについて，スタート地点からの距離を求め，その差を検証した．

空間局所性を利用する手法では，与えられた位置情報から近傍画像を求める際にその検索半径を与える必要がある．検索半径による検索結果の変化を検証するために，複数の検索半径に対して実験を行なった．この時，LSH のパラメータについては，$k = 64$ と $W = 900$ を用いた．また，空間局所性を用いない手法についても同様の実験を行い，その結果を比較した．

#### 4.4.4 実 験 4

空間局所性の考慮において R 木を用いていることやアルゴリズムを改良していることにより，検索速度に影響が出ると考えられるため，検索速度についても評価した．比較した手法は，LC_Parallel，LG，LC_Parallel with SL，LG with SL の 4 つである．LSH のパラメータは $k = 64$ と $W = 900$ に固定とし，それ以外の実験方法については実験 1 と同様に行なった．空間局所性を用いた手法において，近傍画像の検索半径は 0.001 とした．

### 4.5 実験結果

実験結果について，それぞれ説明する．

#### 4.5.1 実 験 1

図 5〜7 に実験 1 の結果を示す．各図は $k$ の値毎のグラフであり，横軸が $W$，縦軸が処理時間を対数軸で示している．図 8 はその凡例である．また，分析のために，表 1 に各条件におけるマッチした画像 ID の平均個数 (= 一致リスト長) を示した．

(注4)：OpenMP www.openmp.org/

(注5)：OpenCV | OpenCV opencv.org

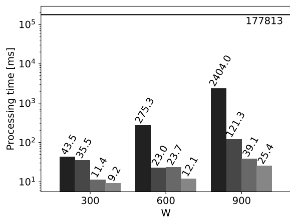


図 5: $k = 16$ に対する処理時間.

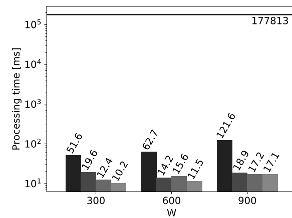


図 6: $k = 32$ に対する処理時間.

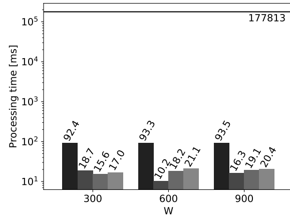


図 7: $k = 64$ に対する処理時間.

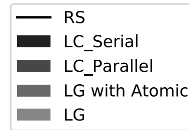


図 8: 図 5〜7 に対する凡例.

表 1: マッチした画像 ID 数.

| | 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 |

表 2: 検索成功率 [%].

| | W=300 | W=600 | W=900 |
|---|---|---|---|
| k=16 | 64 | 54 | 14 |
| k=32 | 46 | 66 | 59 |
| k=64 | 17 | 55 | 74 |

これらの図と表より，実験結果を分析する．$k = 16$(図 5) と $k = 32$(図 6) のときの処理時間を見てみると，LG が最も高速であることがわかる．特に $k = 16$ と $W = 900$ の時の速度向上が最も著しく，LC_Serial と比較して約 95 倍，LC_Parallel と比較して約 5 倍高速であることが分かる．さらに，LC_Serial や LC_Parallel が数秒や数百ミリ秒を要しているような条件であっても，LG の処理時間は概ね 10〜20 ミリ秒程度であり，一貫して高速であることがわかる．一方で，$k = 64$ の時の処理時間を見てみると，いくつかの条件においては LC_Parallel よりも LG が低速である．これは，$k = 64$ におけるマッチした画像 ID の平均個数が比較的小さいために，並列処理可能なデータが少なく，GPU の性能を活かしきれなかったと考えられる．

#### 4.5.2 実 験 2

表 2 に実験 2 の結果を示す．この表は，各セルの数値が検索成功率を表している．表より，パラメータの設定は検索成功率に大きく影響を与え，今回の条件の中では $k = 64$ と $W = 900$ の時に検索成功率が最大となることが分かった．さらに，左上のセルから右下のセルまでを対角線上に見ると，これらの 3 つの条件は全て比較的検索成功率が高いことがわかる．パラメータである $k$ と $W$ は共に検索時の曖昧性に影響しており，$k$ の値は小さくなる程曖昧性が高くなり，一方で $W$ の値は大きくなる程曖昧性が高くなる．すなわち，高精度な類似検索には適切なパラメータの設定が必要である．なお，RS の検索成功率は 95% であり，LSH により検索成功率は低下している．

#### 4.5.3 実 験 3

図 9 が空間局所性を用いない手法に対する実験結果であり，図 10〜13 が空間局所性を用いた手法に対する実験結果である．それぞれの図の横軸はクエリ位置とスタート地点の距離を表し，縦軸は推定位置とスタート地点の距離を表している．また，図中の各点が各クエリ画像に対する結果に対応している．正確な

表 3: 各条件における推定位置の平均絶対誤差 [m].

| 空間局所性非適用 | 検索半径 | | | |
|---|---|---|---|---|
| | 0.0005 | 0.001 | 0.0015 | 0.002 |
| 33.3 | 46.2 | 11.9 | 12.4 | 51.1 |

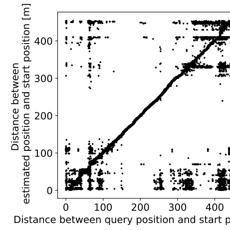


図 9: 空間局所性を用いない手法に対する実験結果.

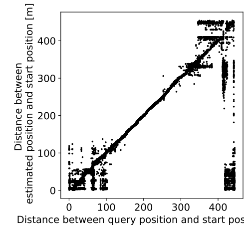


図 10: 検索半径 = 0.0005 に対する実験結果.

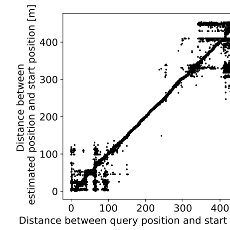


図 11: 検索半径 = 0.001 に対する実験結果.

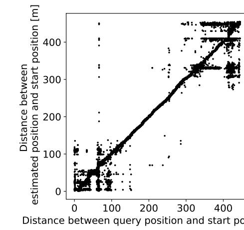


図 12: 検索半径 = 0.0015 に対する実験結果.

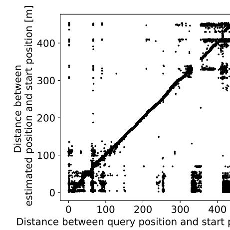


図 13: 検索半径 = 0.002 に対する実験結果.

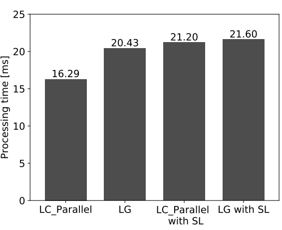


図 14: 空間局所性を用いた手法の処理速度.

検索がなされている時，二つの距離の差は 0 になる．すなわち，直感的に言えば，直線 $y = x$ のグラフに視覚的に類似するほど検索精度が高いと言える．さらに，このグラフを評価するために，各条件における推定位置の平均絶対誤差を求め，それを表 3 に示した．

図と表により，検索精度を検証する．空間局所性を用いた手法同士を比較すると，検索半径 = 0.001 の時の結果が最も精度が高いと言える．さらに，空間局所性非適用の手法と，検索半径 = 0.001 の結果を比較すると，これも検索半径 = 0.001 の結果の方が精度が良いと言える．すなわち，今回の実験から，空間局所性を利用して適切な検索半径を設定することにより，空間局所性を用いない手法よりも精度が向上すると言える．

#### 4.5.4 実 験 4

図 14 に結果を示す．図の横軸は検索手法を示し，縦軸は処理時間を表している．空間局所性を用いることにより，処理速度は低下することが予想されるが，LG と LG with SL を比較てもそれほど差がないことが分かる．これは，クエリ処理において一致リストのサイズ削減が行われることにより処理すべきデータ量が削減されることによるものだと考えられる．一方で，LC_Parallel と LC_Parallel with SL を比較すると，空間局所性を

用いた手法の方が約 1.3 倍の処理時間がかかっており，空間局所性を用いることにより処理コストが増大している．

### 4.6 考察

LSH と GPU を利用することにより高速な類似画像検索による歩行者位置推定が実現可能であることを示した．CPU を利用して検索する場合は，条件によっては数百ミリ秒程度の時間を要することがあるが，GPU を用いた検索では一貫して 10〜20 ミリ秒程度での検索が可能であった．リアルタイムな位置推定を行う場合，クエリ画像も連続して入力されることが想定されるため，数十ミリ秒程度で検索可能であることが要求される．すなわち，LSH と GPU による検索はその処理速度において非常に有用であると言える．

しかし，その精度はナイーブな類似画像検索を利用する場合と比較して低下するため，歩行者の空間局所性を利用した手法を提案した．実験 3 を通して，歩行者の空間局所性を用いることにより，検索精度が向上することが分かった．また，GPU による検索においては空間局所性を用いることによる処理コストも小さく，処理速度への影響が小さいということが分かった．

以上より，非圧縮の SIFT を用いた手法や CPU を用いた手法と比較して，本研究の提案手法は歩行者位置推定において有用であると言える．非圧縮の SIFT を用いたナイーブな類似画像検索による歩行者位置推定では，その処理コストが大きく，リアルタイム性に乏しい．また，LSH と CPU を用いた類似画像検索手法と比較して，提案手法は一貫して高速である．LSH を用いることによりその検索精度は低下するが，空間局所性を用いることにより精度を向上可能である．すなわち，提案手法では，ナイーブな手法と比較して精度が落ちてしまうものの，リアルタイムな歩行者位置推定が可能である．

## 5. 関連研究

本研究は Kameda と Ohta [11] の歩行者支援のための類似画像検索手法をベースとしている．この研究は，主に視覚障がい者の外出支援を目的としており，類似画像検索を用いてユーザーの位置を推定する．位置情報は事前に撮影された動画中のフレームからクエリ画像を類似検索し，検索結果のフレームに紐付けられた位置情報を参照することにより得られる．精度向上のために，Kameda らは検索結果を検証するための 7 つの指標を提案し，それらを全て満たす画像のみを検索結果としている．実際の位置推定アプリケーションではクエリ画像も連続して入力されることが想定されるが，この手法は計算コストが大きく，動画のフレーム間隔での検索は不可能である．

また，Kamasaka ら [10] は Kameda らの手法の拡張として，条件の異なる複数の動画に対する歩行者位置推定手法を提案した．Kameda らの手法は複数の動画をデータベースとして用いることにより，その検索精度の向上が可能である．しかし，それらの動画中の各フレームには正確な位置情報が付与されている必要があり，全てのフレームに位置情報を付与しておくことはコストがかかる．そこで Kamasaka らの手法では，検索の前処理において位置情報が付与された動画 (Main video) と付与されていない動画 (Sub video) の二つの動画を統合することによりそのコストを削減する．その統合は Sub video の各フレームをクエリとして Main video 中から類似フレームを検索することにより行われ，これにより位置推定精度が向上する．

## 6. 結論

本研究では，類似画像検索を用いた歩行者位置推定の高速化手法を提案した．具体的には，LSH を用いた GPU 上での効率的な類似画像検索アルゴリズムを提案し，さらに歩行者の空間局所性を用いて位置推定精度を向上させた．

また，評価実験を通して提案手法の有用性について検証した．LSH と GPU を用いることにより，複数の条件に対して一貫して高速な類似画像検索による歩行者位置推定が可能であることを示した．また，空間局所性を用いることにより，処理速度を保ちながら位置推定精度が向上できることを示した．

今後の課題としては，天候やライティング条件の異なる動画や，大規模動画データベースに対する性能検証が挙げられる．

## 文献


[1] Pablo Fernández Alcantarilla, Jesús Nuevo, and Adrien Bartoli. Fast explicit diffusion for accelerated features in nonlinear scale spaces. In *BMVC 2013*, pp. 1–11, 2013.

[2] Jurandy Almeida, Ricardo da S Torres, and Siome Goldenstein. Sift applied to cbir. *Revista de Sistemas de Informacao da FSMA n*, Vol. 4, pp. 41–48, 2009.

[3] Alexandr Andoni and Piotr Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Commun. ACM*, Vol. 51, No. 1, pp. 117–122, 2008.

[4] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval: The Concepts and Technology Behind Search*, Vol. 2. Addison Wesley, 2011.

[5] Herbert Bay, Tinne Tuytelaars, and Luc J. Van Gool. SURF: speeded up robust features. In *ECCV 2006*, pp. 404–417, 2006.

[6] Jian Cheng, Cong Leng, Jiaxiang Wu, Hainan Cui, and Hanqing Lu. Fast and accurate image matching with cascade hashing for 3d reconstruction. In *CVPR 2014*, pp. 1–8, 2014.

[7] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *SCG 2004*, pp. 253–262, 2004.

[8] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD 1984*, pp. 47–57, New York, NY, USA, 1984.

[9] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *STOC 1998*, pp. 604–613, 1998.

[10] Kazuho Kamasaka, Itaru Kitahara, and Yoshinari Kameda. Image based location estimation for walking out of visual impaired person. In *AAATE 2017*, pp. 709–716, 2017.

[11] Yoshinari Kameda and Yuichi Ohta. Image retrieval of first-person vision for pedestrian navigation in urban area. In *ICPR 2010*, pp. 364–367, 2010.

[12] Takeshi Kurata, Masakatsu Kourogi, Tomoya Ishikawa, Yoshinari Kameda, Kyota Aoki, and Jun Ishikawa. Indoor-outdoor navigation system for visually-impaired pedestrians: Preliminary evaluation of position measurement and obstacle display. In *ISWC 2011*, pp. 123–124, 2011.

[13] David G. Lowe. Object recognition from local scale-invariant features. In *ICCV 1999*, pp. 1150–1157, 1999.

[14] John D Owens, Mike Houston, David Luebke, Simon Green, John E Stone, and James C Phillips. Gpu computing. *Proceedings of the IEEE*, Vol. 96, No. 5, pp. 879–899, 2008.

[15] Hotaka Takizawa, Kazunori Orita, Mayumi Aoyagi, Nobuo Ezaki, and Shinji Mizuno. A spot reminder system for the visually impaired based on a smartphone camera. *Sensors*, Vol. 17, No. 2, p. 291, 2017.