DEIM Forum 2017 E7-3

# SIFT 特徴量を用いた画像検索の GPU による高速化

草村　優太†　　天笠　俊之††　　北川　博之††

† 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
E-mail: †kusamura@kde.cs.tsukuba.ac.jp, ††{amagasa,kitagawa}@cs.tsukuba.ac.jp

**あらまし**　画像検索とは，画像集合中からクエリ画像と視覚的に類似した画像を見つけ出す処理であり，様々なアプリケーションで利用されている．画像間の類似度を計算するため，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る．しかし，画像検索では扱う特徴ベクトル数が膨大であり，特徴ベクトルが高次元であるため，多大な処理コストを要する．そのため，高速な画像検索の実現は非常に重要である．本研究では，GPU (graphics processing unit) を用いた画像検索の高速手法を提案する．その際，データ量の削減と処理の高速化のために，提案手法では特徴ベクトルの圧縮を行う．その上で GPU 上での計算に適したデータ構造及びアルゴリズムを用いて画像検索を行う．さらに，画像データ及び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，画像検索，SIFT 特徴量

## 1. はじめに

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画像検索と呼ぶ [4]．画像データは普遍的に利用されるため，画像検索は様々なアプリケーションで利用されている．

画像検索を適用可能なアプリケーションとして，スマートフォン等を利用した歩行者の位置推定 [17] があげられる．このアプリケーションは利用者として視覚障害者を想定し，利用者の日常生活における移動を支援する．事前に普段よく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．検索精度を向上させるためには，同じ経路であっても異なる気象条件やライティングの条件が異なる複数の映像を格納しておくことが望ましい反面，検索時には各映像フレーム毎に位置を特定する必要があるため，大量の画像に対して高速な検索を行う必要がある．また，検索結果は最もマッチする上位 1 件のみが必要であり，Top-k 検索のような複雑な処理は不要である．

このようなアプリケーションにおいて，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．

画像の特徴を表すベクトルのことを特徴ベクトルといい，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程度のベクトルを用いて画像の特徴を表現し，画像検索はこれらを比較することにより実現可能である．しかし，画像データベースでは特徴ベクトル数が莫大となり，さらに，特徴ベクトルが高次元であるため，画像検索は多大な処理コストを要する．そのため，画像検索の高速化は非常に重要である．

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]．

そこで本稿では，SIFT 特徴量を用いた画像検索の GPU による高速化手法を提案する．提案手法では，特徴ベクトルの圧縮のために LSH (locality sensitive hashing) [11] を用いる．LSH は類似した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，類似検索の高速化等に利用されている．

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される．まず，データベース構築では，画像データベースから抽出された特徴ベクトルを GPU での処理に適したデータ構造に変換し，GPU への転送を行う．また，クエリ処理は与えられたクエリ画像を GPU 上で検索する処理である．

また，本研究では提案手法の性能を評価するために実験を行った．画像および動画データセットを用いた実験により，CPU を用いた検索手法と比較して約 3.1 倍の高速化が可能であることを示した．また，歩行者の位置推定アプリケーションへの適用を想定した精度評価を行い，62 % の正解率を実現できることを示した．

本稿の構成を以下に示す．まず，2. 節にて，本研究に必要となる前提知識について説明する．次に，3. 節にて，本研究の関連研究を紹介する．その後，4. 節にて，本研究の提案手法について説明し，その性能評価のための実験について 5. 節にて説

明する．最後に，6. 節にて，まとめと今後の展望について説明する．

## 2. 前提知識

本節では，本研究に必要となる前提知識について説明する．2.1 節にて，画像検索について説明し，2.2 節にて GPU および GPU コンピューティングについて説明する．

### 2.1 画像検索

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画像検索と呼ぶ [4]．画像検索は様々なアプリケーションで利用されているが，本研究では画像検索を利用した歩行者の推定のアプリケーションを想定し，本研究で扱う画像検索を以下のように定義する．まず，入力として画像データベースと一つのクエリ画像が与えられる．その際，クエリ画像と最も視覚的に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，近年の画像検索では局所特徴量を用いる手法が主流である．特に，SIFT (scale-invariant feature transform) 特徴量 [12] 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．そこで，本研究では SIFT 特徴量を用いた画像検索の高速化手法を提案する．

#### 2.1.1 SIFT 特徴量

SIFT 特徴量 [12] は局所特徴量の一つであり，数千程度の 128 次元ベクトルで画像の特徴を表現する．SIFT 特徴量は照明変化，拡大縮小，回転に対して強固であるという特徴を持つ．そのため，視覚的に類似した画像を見つける処理である画像検索に適している．

#### 2.1.2 SIFT 特徴量を用いた画像検索

SIFT 特徴量を用いた単純な画像検索 [1] は以下のように実現することができ，その処理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．

また，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．クエリ画像が与えられると，はじめにクエリ画像から特徴量 $Q$ を抽出する．次に，クエリのそれぞれの特徴ベクトル $\boldsymbol{q} \in Q$ に対して，最も距離が近い $D$ 中のベクトルに対応する画像 ID を得る．つまり，以下の式で表される画像 ID を得る．

$$id(\boldsymbol{f}_i), s.t. \forall \boldsymbol{f}_i, \boldsymbol{f}_j \in D \wedge dist(\boldsymbol{q}, \boldsymbol{f}_i) \leqq dist(\boldsymbol{q}, \boldsymbol{f}_j)$$

ただし，$dist()$ は二つのベクトル間の距離である．この処理をクエリのそれぞれの特徴ベクトル $\boldsymbol{q}$ に対して行い，最も多く得られた画像 ID に対応する画像が検索結果となる．

#### 2.1.3 LSH 及び SIFT 特徴量を用いた画像検索

LSH (locality sensitive hashing) [11] 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の一つである．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 は特徴ベクトル圧縮は近傍検索高速化に利用される [3,8]．

LSH では，複数のハッシュ関数 $h_i()$ $(0 \leqq i < k)$ からなるハッシュ関数 $H() = (h_0(), h_1(), \ldots, h_{k-1}())$ を用いてベクトルをハッシュ値に変換する．つまり，ベクトル $\boldsymbol{v}$ は以下の式により，サイズ $k$ のハッシュ値 $\boldsymbol{v}'$ に変換される．

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

ハッシュ関数 $h_i()$ $(0 \leqq i < k)$ 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 [2]．

SIFT 特徴ベクトル間の距離は $l_2$ ノルムが用いられることが標準的であるため，本研究では $l_2$ ノルムに対する LSH [9] 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 $d$ $(d = |\boldsymbol{v}|)$ 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．

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

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の部分空間に分割され，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．そのため，LSH を近傍検索に用いた際のあいまい性が高くなる．

LSH 及び SIFT 特徴量を用いた画像検索 [16] は，2.1.2 節の手法を拡張することで実現できる．まず，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 $D'$ に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，得られた各ハッシュ値 $\boldsymbol{q}'$ 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．なお，LSH を用いているため，完全一致するハッシュ値の検索が特徴ベクトルの近傍検索の代用となる．

### 2.2 GPU コンピューティング

GPGPU (general-purpose computing on graphics processing units) は，従来グラフィック処理向けに利用される GPU (graphics processing unit) を汎用計算に用いる技術のことである．GPU には，CPU と比較して多くのコアが搭載されているため，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]．

GPU のアーキテクチャは独特であるため，不適切な実装やアルゴリズムは CPU よりも悪いパフォーマンスを示す．そのため，GPU の特性を理解した上で適切なプログラムを組む必要がある．本研究では，NVIDIA 社の GPU とその統合開発環境である CUDA (注1) を用いる．

GPU は一般的に，プロセッサとメモリにおいて階層的な構造を持つ．まず，GPU にはグローバルメモリとストリーミングマルチプロセッサが搭載されている．グローバルメモリは GPU 上で最も大きなメモリであり，ストリーミングマルチプロセッサにはシェアードメモリとレジスタが搭載されている．シェアードメモリは最大で 64 KB と比較的小さいメモリだが，同一のストリーミングマルチプロセッサ内からの高速なアクセスが可能である．レジスタはシェアードメモリよりもさらに小さく高速なメモリであり，それぞれのスカラプロセッサからのみのアクセスが可能である．

また，GPU はその処理モデルも階層的である．GPU には，スレッド，ブロック，グリッドと呼ばれる三つの処理モデルが存在する．ブロックは数百程度のスレッド集合であり，グリッドは最大で数万個のブロックからなる．スレッド，ブロック，グリッドはそれぞれスカラプロセッサ，ストリーミングマルチプロセッサ，デバイスで処理される．

本研究ではいくつかのデータ並列アルゴリズムプリミティブを用いる．データ並列アルゴリズムプリミティブは並列計算で頻出し，並列処理により高速な計算が可能である．また，GPU 向けのライブラリ (注2) も存在し，利用することで高速な処理が可能である．本研究では，以下のデータ並列アルゴリズムプリミティブを利用する．

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 $[I, a_0, a_0 \oplus a_1, \ldots, a_0 \oplus a_1 \oplus \ldots \oplus a_{n-1}]$ を返す．

sort: 配列 $[a_0, a_1, \ldots, a_{n-1}]$ を規則に基づいてソートする．

これらのデータ並列アルゴリズムプリミティブは，全てライブラリが利用可能である (注3)．

## 3. 関 連 研 究

Sivic らは BoVW (bag-of-visual-words) フレームワーク [15] を提案した．このフレームワークでは，特徴ベクトルを量子化し複数の特徴ベクトルを同一の特徴ベクトルとみなす．この量子化されたベクトルは visual words と呼ばれ，これらを用いて文書検索に基づいた方法でクエリ画像の検索を行う．

GPU を用いた画像検索の高速化に関する研究も行われている．Cevahir らの提案した手法 [6] は，2-ステップマッチングにより特徴ベクトルの検索を行う．2-ステップマッチングでは，まず階層的 k-means を用いて特徴ベクトルをクラスタリングしておく．そしてクエリ画像が与えられると，クエリ画像の特徴ベクトルに対して最も近いクラスタを検索し，さらにクラスタ内で最近傍のベクトルを検索する．これらの処理を GPU 上で行うことにより，画像検索の高速化を実現した．

特徴ベクトルを扱うアプリケーションにおいては，その空間コストや転送コストの削減が重要であり，SIFT 特徴ベクトルの圧縮方法に関する研究も多く行われている．Chandrasekhar らはハッシュ法，変換符号化，ベクトル量子化の三つに大別される圧縮手法を比較し，その性能を評価した [7]．本研究で用いる LSH はハッシュ法の一種であり，特徴ベクトルの圧縮に有効である．

Carraher らは，SIFT 特徴ベクトルの高速な近傍検索手法である LSH-NN を提案した [5]．LSH-NN では，LSH と GPU を用いることにより特徴ベクトルの近傍検索を高速に行う．しかし，LSH-NN で実現されたのは特徴ベクトルの近傍検索の高速化であり，特徴ベクトルを用いた画像検索の高速化は実現されていない．

## 4. 提 案 手 法

提案手法では，2.1.3 節で述べた LSH 及び SIFT 特徴量を用いた画像検索の高速化を実現する．その処理は，2.1.2 節及び 2.1.3 節と同様に，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．

本研究で高速化する処理は，クエリ処理の内，特徴ベクトルを抽出してから画像 ID を返すまでの処理を対象とする．特に，LSH による特徴ベクトルの圧縮とデータベースとの照合処理の高速化を図る．また，データベース構築では，データベースとの照合処理の高速化を実現するため，GPU 上での処理に適したデータベースに変換する．

### 4.1 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 2.1.3 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 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*: データベース中の全ての $\boldsymbol{f}'_i$ を要素とする配列．要素は昇順でソートして格納される．

*id_val*: $\boldsymbol{f}'_i$ 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*: $id\_ptr[i]$ が *id_val* の先頭レコードからのオフセットを示す配列．

クエリ処理で *f_val* に対する二分探索を行うため，*f_val* はデー

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

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

(注3)：Thrust :: CUDA Toolkit Documentation http://docs.nvidia.com/cuda/thrust/

**Algorithm 1**

**Input:** $D'$ ただし，$(\boldsymbol{f}'_i, id_i) \in D'$，$id_{ij} \in id_i$
**Output:** $f\_val$, $id\_val$, $id\_ptr$

```
D' を f'_i の値に基づき昇順ソート
offset = 0
id_ptr[0] = 0
for i = 0 : rowsof(D') − 1 do
    f_val[i] = f'_i
    for j = 0 : sizeof(id_i) − 1 do
        id_val[offset++] = id_ij
    id_ptr[i + 1] = offset
```

**Algorithm 2**

**Input:** $Q$ ただし，$q_{ij} \in Q$
**Output:** $Q'$ ただし，$q'_{ij} \in Q$

```
for i = 0 : Q.size() − 1 do in block parallel
    for j = 0 : k − 1 do in thread parallel
        q'_ij = h_k(q_ij)
```

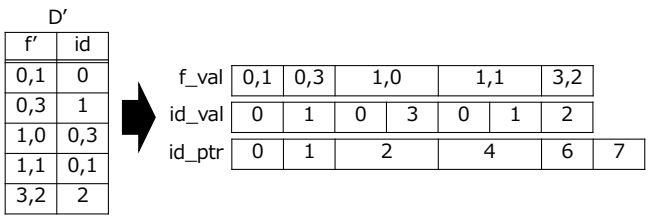


図 1　データベース $D'$ に基づく配列の作成.

**Algorithm 3**

**Input:** $Q', f\_val, id\_ptr, id\_val$
**Output:** $M$

```
for i = 0 : Q'.size() − 1 do in thread parallel
    index = binarySearch(q'_i, f_val)
    if index == −1 then
        P[i] = −1
    else
        P[i] = id_ptr[index]
    if index == −1 then
        N[i] = 0
    else
        N[i] = id_ptr[index + 1] − id_ptr[index]
F = scan(N)
for i = 0 : Q'.size() − 1 do in block parallel
    for j = 0 : N[i] − 1 do in thread parallel
        M[F[i] + j] = id_val[P[i] + j]
```

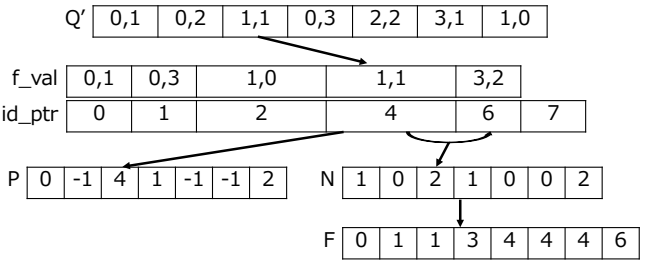


図 2　配列 P，N，F の作成.

タを昇順にソートする．また，ハッシュ値に対応する画像 ID の集合を二つの配列 $id\_val$ および $id\_ptr$ で表現する．これらの配列の作成処理は，Algorithm 1 で表される．また，図 1 は配列作成処理の例である．

### 4.2　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する．その処理では，まずクエリ画像から特徴量 $Q$ を抽出し，GPU 上のグローバルメモリに転送する．そして，GPU 上で LSH が適用され，ハッシュ値に変換される．LSH による圧縮後，ハッシュ値とデータベースとの照合を行う．最後に，照合の結果が検索結果となる．

クエリ処理のうち，GPU 上で行われる処理は，特徴ベクトルの圧縮とデータベースとの照合の二つに大別される．以下でそれぞれの処理について説明する．

#### 4.2.1　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 LSH による変換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，2.1.3 節で述べたハッシュ値 $h_j(\boldsymbol{q_i})$ の計算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．この変換処理は Algorithm 2 で表される．これにより，GPU の並列処理性能を活かすことができ，高速な処理が可能である．

#### 4.2.2　データベースとの照合

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 を用いることで，GPU 上で特徴ベクトル毎のスレッド並列で処理が可能である．atomicAdd とは，排他制御を持つ加算関数であり，同一アドレスへの書き込みの衝突を回避できる．これを用いて出現頻度集計用の配列要素をインクリメントすることにより，最頻出の画像 ID を検出する．しかし，atomicAdd による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．シェアードメモリ上のアドレスに対して atomicAdd の書き込みを行う事により，比較的高速な処理が可能であるが，扱う画像の数が多い場合，メモリ容量が不足する．

そこで，本研究では複数の配列を用いて照合処理を高速に行う方法を提案する．本研究で提案する照合処理は，一致リストの取得と最頻出 ID の検出の二つの処理に分けられる．一致リストの取得は，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

一致リストの取得では，Algorithm 3 により一致リスト $M$ を

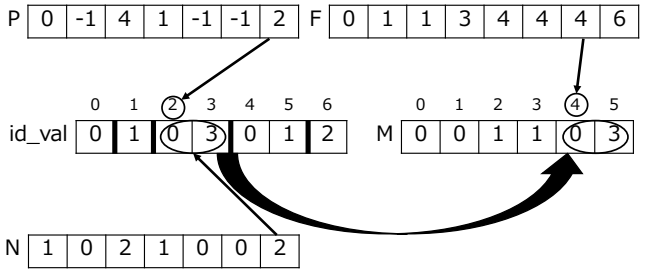


図 3　配列 P，N，F を用いた配列 M の作成

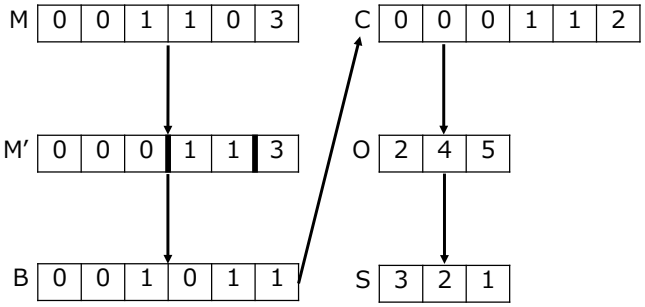


図 4　配列 M に対する，配列 M'，B，C，O，S の作成

取得する．一致リストは適切な $id\_val$ の値を配列 $M$ にコピーすることによって取得可能である．GPU のブロック並列で $M$ を取得するために，各ブロックがコピーする $id\_val$ の値の先頭インデックスとコピーサイズ及び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 $P, N, F$ とする．さらにこれらの配列を用いて $M$ を効率的に作成する．以下で，配列 $P, N, F$ の作成と，$P, N, F$ を用いた $M$ の作成の詳細を説明する．

$P, N, F$ の作成では，2 行目の処理で各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，マッチした配列のインデックスを $index$ とする．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．次に，3〜6 行目の処理と 7〜10 行目の処理で，配列 $P$ と $N$ を作成する．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．最後に，11 行目の処理で $N$ に対して scan プリミティブを適用し，配列 $F$ を作成する．これらの配列はそれぞれ，$P$ はハッシュ値にマッチする $id\_ptr$ の値，$N$ はハッシュ値にマッチする ID の個数，$F$ は $F[i]$ が $i$ 番目以前のハッシュ値にマッチする ID の個数を表す．また，図 2 は，これらの配列の作成例を示している．

配列 $M$ は，配列 $P, N, F$ をもとに，12 行目の処理によって作成される．これは，$id\_val$ の $P[i]$ 番目から $N[i]$ 個の値を $M$ の $F[i]$ 番目から $N[i]$ 個の要素としてコピーすることを表している．このコピーを各 $i$ 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，配列 $F$ 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．図 3 は，このコピーの例である．

b）　最頻出 ID の検出

最頻出 ID の検出では，一致リストの取得によって得られた

**Algorithm 4**

**Input:** $M$
**Output:** $id$
1: $M'$=$sort(M)$
2: $B[M'.size()] = 1$
3: **for** $i = 0 : M'.size() - 1$ **do in thread parallel**
4: $\quad B[i] = M'[i] \,!= M'[i+1]$
5: $C = scan(B)$
6: **for** $i = 0 : B.size() - 1$ **do in thread parallel**
7: $\quad$ **if** $B[i] == 1$ **then**
8: $\quad\quad O[C[i]] = i$
9: $S[0] = O[0] + 1$
10: **for** $i = 1 : O.size() - 1$ **do in thread parallel**
11: $\quad S[i] = O[i] - O[i-1]$
12: $r = reduction(S)$
13: $id = M'[O[r]]$

$M$ の中で最頻出の画像 ID を検出する．その検出は，$M$ をソートした配列 $M'$ を用いて行われる．この時，$M'$ の値が変化するインデックスを境界とし，境界間の部分配列を区間とした時，サイズが最大となる区間の要素が最頻出の画像 ID である．本手法では，Algorithm 4 によりこの検出処理を高速に行う．

まず，1 行目では，$M$ をソートし，$M'$ とする．次に，2〜4 行目の処理により，$M'$ の境界位置を表す配列 $B$ を作成する．その後，5 行目の処理で，配列 $B$ に scan プリミティブを適用し，配列 $C$ を作成する．これは，$M'$ のそれぞれの位置の区間番号を表している．さらに，6〜9 行目の処理で，各区間の終点までのオフセットを表す配列 $O$ を作成する．最後に，10〜11 行目の処理で，各区間のサイズを表す配列 $S$ を作成する．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．そのため，GPU 上での高速な処理が可能である．図 4 は，これらの配列の作成例である．

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 が検索結果となる．そこで，サイズが最大となる区間番号を検出するために，reduction 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ような配列インデックスを検出する関数は存在しないため，単純な reduction 処理を拡張した関数を実装した．この関数を利用して，12 行目の処理により区間のサイズが最大となる区間番号 $r$ を検出する．このとき，13 行目で表される処理により，検索結果が求められる．

## 5. 実　　験

本節では，評価実験について説明する．提案手法は既存手法と比較して高速な検索を実現するが，LSH を用いてデータを圧縮しているため，精度が低下することが予想される．そこで，提案手法の検索速度に加えて検索精度を検証する．

### 5.1 データセット

本実験では，公開データセットと独自データセットを含む三つのデータセットを用いた．公開データセットとして，ukbench

と First-Person Social Interactions Dataset を用いた．また，独自データセットは，ビデオカメラを用いて作成した．以下にそれぞれ詳細を示す．

ukbench [13] (注4) は，10,200 枚の画像からなる画像データセットであり，2,550 個の被写体をそれぞれ異なる角度から 4 枚ずつ撮影された写真で構成されている．それぞれの画像の解像度は 640×480 であり，約 15 百万本の SIFT 特徴ベクトルが抽出された．

First-Person Social Interactions Dataset [10] (注5)(以下，FPSID) は，114 個の動画からなるデータセットであり，この動画の一つである AlinDay1-001 の各フレームを画像集合として用いた．これは，フレーム数 31,277 枚，解像度 1280×720 の動画である．これを，解像度 640×360 にリサイズして用いて，約 43 百万本の SIFT 特徴ベクトルが抽出された．

ビデオカメラを装着し筑波大学構内を歩きながら撮影した位置情報付きの独自データセット (以下，UTWD) も利用した．UTWD は OLYMPUS STYLUS TG-Tracker を胸部前方に固定した状態で筑波大学構内を歩くことにより撮影した．本実験では，平砂学生宿舎共用棟前 (36.097688, 140.103352 付近) から 2C 棟 (36.111261, 140.100812 付近) までの経路 (以下，UTWD-long) と，大学会館前 (36.105397, 140.102480 付近) から中央図書館前 (36.109306, 140.101669 付近) まで経路 (以下，UTWD-short) を撮影し，これらの各フレームを画像集合として利用した．なお，UTWD-short の経路は UTWD-long の経路中に含まれる．UTWD-long はフレーム数 81,936 枚，解像度 1280×720 の動画である．これを，解像度 640×360 にリサイズして用いて，約 96 百万本の SIFT 特徴ベクトルが抽出された．一方，UTWD-short はフレーム数 25,980 枚，解像度 1280×720 の動画である．また，UTWD では歩行者の位置推定に利用するために動画の各フレームに TG-Tracker で得られる GPS 情報を付加した．ただし，TG-Tracker では 2 秒毎の GPS 情報のみが得られるため，直接位置情報を得ることができない時刻については線形補間を用いておおよその位置情報を求めた．なお，このデータセットは一般の歩行者の映像が含まれているため非公開である．

### 5.2 比較手法

実験の比較手法として，提案手法を含めて三つの画像検索プログラムを実装した．三つの手法は以下の通りである．

**SIFT-CPU**: 2.1.2 節で述べた検索手法の並列実行プログラム．

**LSH-CPU**: 2.1.3 節で述べた検索手法の並列実行プログラム．

**LSH-GPU**: 4. 節で述べた提案手法のプログラム．

プログラムは C++及び CUDA で記述し，nvcc 7.0 を用いてコンパイルを行った．また，最適化オプションとして -O3 を用いた．CPU での並列化には OpenMP(注6) を用い，SIFT 特徴ベクトルの抽出には OpenCV(注7) を用いた．

プログラムは，CentOS release 6.7 を搭載しているマシン上にて実行した．このマシンは，Intel Xeon Processor E5-2687W v2 と，NVIDIA Tesla K40 を搭載している．メインメモリは 128 GB，デバイスメモリは 12 GB である．

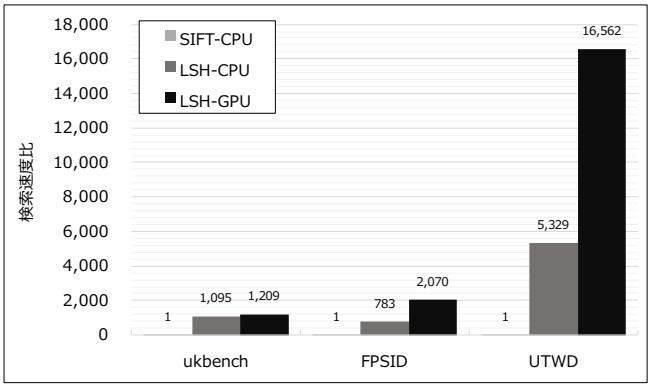


図 5 SIFT-CPU に対する各検索手法の検索速度比．

### 5.3 実験方法

前述のデータセットとプログラムを用いて，以下の二つの実験を行った．

a) 実 験 1

各プログラムにおける検索時間を比較するために，ukbench と FPSID 及び UTWD-long を画像データベースとして，複数のクエリ画像に対して各プログラムで検索を行い平均検索時間を計測した．ukbench と FPSID では画像データベース中に含まれる画像をクエリ画像とし，UTWD-long に対しては UTWD-short に含まれる画像各 100 件をクエリ画像とした．ただし，クエリ処理の内，特徴ベクトルを抽出してから画像 ID を返すまでの処理を検索時間として計測した．

b) 実 験 2

LSH による精度変化を検証するために，非圧縮の SIFT 特徴ベクトルを用いたプログラムと LSH を用いたプログラムにおける検索結果の位置情報に基づく正解率を比較した．このとき，LSH を用いたプログラムについてはパラメータ $W$ とハッシュ値サイズ $k$ をそれぞれ変化させて正解率を計算した．画像データベースとして UTWD-long を用い，クエリ画像として UTWD-short 中の画像 100 件を用いた．

検索結果はクエリ画像が撮影された地点と検索結果の画像の地点が 15m 以内であれば正解とした．これは位置情報として GPS を用いていることによる誤差を考慮し，実際のアプリケーションにおける許容誤差より大きく設定している．このとき，GPS 情報から距離への変換には Hubeny の公式 [18] を用いた．Hubeny の公式は緯度経度情報から距離を計算可能な式であり，短距離であれば高精度な距離を取得可能である．

### 5.4 実験結果

本節では，実験 1 と実験 2 の結果および考察を順に示す．

a) 実 験 1

まず，実験 1 の結果を図 5 に示す．図 5 では，検索時間の逆

(注4)：Recognition Benchmark Images www.vis.uky.edu/~stewe/ukbench/

(注5)：First-Person Social Interactions Dataset http://ai.stanford.edu/~alireza/Disney/

(注6)：OpenMP www.openmp.org/

(注7)：OpenCV | OpenCV opencv.org

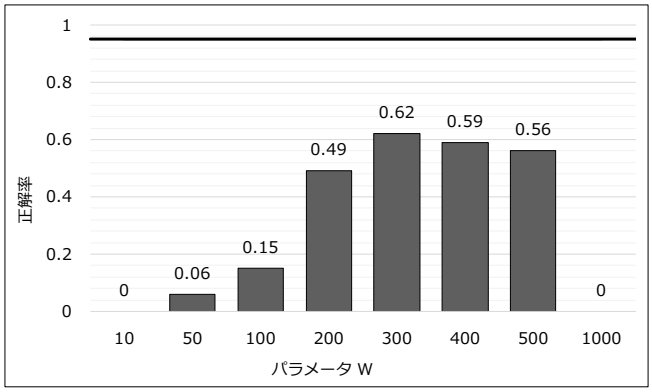


図 6　異なる $W$ における正解率.

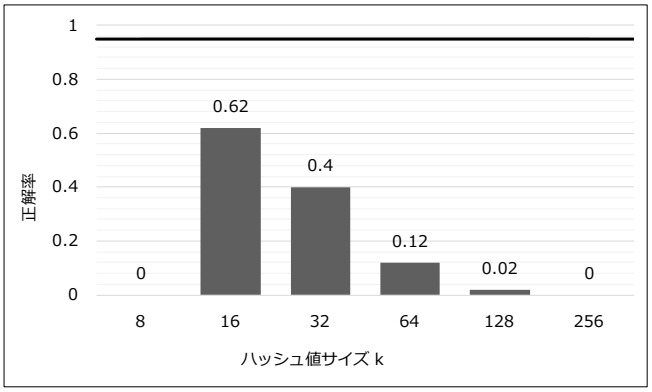


図 7　異なる $k$ における正解率.

数を検索速度とし，各データセットにおける SIFT-CPU の検索速度を 1 としたときの各検索手法の検索速度比を示している．横軸がデータセットと検索手法であり，縦軸に検索速度比を示している．ただし，ハッシュ値サイズは $k = 16$，パラメータは $W = 300$ に固定とした．UTWD データセットの結果に着目すると，SIFT-CPU と比較して約 1.7 万倍，LSH-CPU と比較して約 3.1 倍と高速である．一方，ukbench データセットと FPSID データセットの結果に着目すると，UTWD データセットよりも速度が向上していないことが分かる．この原因として，ukbench と FPSID のクエリ画像あたりの特徴ベクトル数が少ないことが考えられる．クエリ画像あたりの特徴ベクトル数は UTWD では約 3,940 に対して，ukbench と FPSID ではそれぞれ約 1,601 と約 1,441 である．そのため，並列度が上がらずに性能向上が小さくなったと考えられる．

b）実　験　2

図 6 にパラメータ $W$ を変化させた際の正解率を示す．この時，ハッシュ値サイズは $k = 16$ に設定した．図 6 では，横軸を $W$ とし，縦軸に各 $W$ における正解率を示している．また，非圧縮 SIFT を用いたときの正解率 (= 0.95) を水平線で表している．これによると，$W$ の値は検索結果の正解率に大きく影響を与えることが分かる．特に，今回の結果の中では，$W = 300$ とした時の正解率が最も高く，62 % の正解率を実現している．

図 7 にハッシュ値サイズ $k$ を変化させた際の正解率を示す．この時，パラメータは $W = 300$ に固定して計測した．図 7 では 6 と同様に，横軸を $k$ とし，縦軸に各 $k$ における正解率を示している．$W$ と同様に，$k$ も検索結果の正解率に影響することが分かる．

## 6. ま と め

本研究では，SIFT 特徴量を用いた画像検索の GPU による高速化手法を提案した．その際，LSH を用いて特徴ベクトルを圧縮することにより，空間コストと時間コストの削減を実現した．また，GPU 上での処理に適したデータ構造およびアルゴリズムにより，高速な処理を実現した．

実験により，CPU による検索手法と比較して約 3.1 倍程度の高速化が可能であることを示した．また，実アプリケーションでの適用を想定した精度評価を行い，62 % の正解率が実現可能であることを示した．

今後の展望としては，クエリ処理のさらなる高速化のために特徴ベクトルの高速な抽出方法を検討する必要がある．また，精度向上のために，LSH における適切なパラメータやハッシュ値サイズの検討も必要である．

## 謝　辞

本研究は JSPS 科研費 JP26280037 の助成を受けたものです．

## 文　献

[1] 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.

[2] 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.

[3] Adrien Auclair, Laurent D Cohen, and Nicole Vincent. How to use sift vectors to analyze an image with database templates. In *Adaptive Multimedia Retrieval: Retrieval, User, and Semantics*, pp. 224–236. Springer, 2007.

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

[5] Lee Carraher, Philip A. Wilsey, and Fred S. Annexstein. A GPGPU algorithm for c-approximate r-nearest neighbor search in high dimensions. In *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum, Cambridge, MA, USA, May 20-24, 2013*, pp. 2079–2088, 2013.

[6] Ali Cevahir and Junji Torii. Gpu-enabled high performance online visual search with high accuracy. In *2012 IEEE International Symposium on Multimedia, ISM 2012, Irvine, CA, USA, December 10-12, 2012*, pp. 413–420, 2012.

[7] Vijay Chandrasekhar, Mina Makar, Gabriel Takacs, David Chen, Sam S Tsai, Ngai-Man Cheung, Radek Grzeszczuk, Yuriy Reznik, and Bernd Girod. Survey of sift compression schemes. In *Proc. Int. Workshop Mobile Multimedia Processing*, pp. 35–40. Citeseer, 2010.

[8] Jian Cheng, Cong Leng, Jiaxiang Wu, Hainan Cui, and Hanqing Lu. Fast and accurate image matching with cascade hashing for 3d reconstruction. In *2014 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2014, Columbus, OH, USA, June 23-28, 2014*, pp. 1–8, 2014.

[9] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the 20th ACM Sym-*

*posium on Computational Geometry, Brooklyn, New York, USA, June 8-11, 2004*, pp. 253–262, 2004.

[10] Alireza Fathi, Jessica K. Hodgins, and James M. Rehg. Social interactions: A first-person perspective. In *2012 IEEE Conference on Computer Vision and Pattern Recognition, Providence, RI, USA, June 16-21, 2012*, pp. 1226–1233, 2012.

[11] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *Proceedings of the Thirtieth Annual ACM Symposium on the Theory of Computing, Dallas, Texas, USA, May 23-26, 1998*, pp. 604–613, 1998.

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

[13] David Nistér and Henrik Stewénius. Scalable recognition with a vocabulary tree. In *2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR 2006), 17-22 June 2006, New York, NY, USA*, pp. 2161–2168, 2006.

[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] Josef Sivic and Andrew Zisserman. Video google: A text retrieval approach to object matching in videos. In *9th IEEE International Conference on Computer Vision (ICCV 2003), 14-17 October 2003, Nice, France*, pp. 1470–1477, 2003.

[16] W. Weihong and W. Song. A scalable content-based image retrieval scheme using locality-sensitive hashing. In *2009 International Conference on Computational Intelligence and Natural Computing*, Vol. 1, pp. 151–154, June 2009.

[17] 亀田能成, 大田友一. 街中での歩行者カメラによるオンライン位置推定のための検討. 画像の認識・理解シンポジウム MIRU, pp. 364–369, 2010.

[18] 日本測量協会創立 30 周年記念「現代測量学」出版委員会. 現代測量学, 第 4 巻. 日本測量協会, 1988.