Vol.2014-CG-156 No.6
2014/9/16

画像の射影変換における高速化法の比較・検討

常盤 勇太[1,a)]　金澤 靖[1,b)]

概要：
画像の射影変換は基本的な幾何学的画像変換の一つで，四角形のテクスチャマッピングやパノラマ画像のモザイキングなど様々な用途に応用されている．これらの応用においてリアルタイムなどの時間にシビアな場合では，画像の射影変換にかかる時間が重要となる．本研究では変換アルゴリズムの工夫や SIMD や GPGPU の利用といった様々な方法で画像の射影変換を実装し，実装方法による計算時間等の違いを比較した．また，その結果からどのような応用，計算機環境においてどの方法を選択すべきかについて検討を行った．

Comparison of acceleration methods for homography computation

Yuta Tokiwa[1,a)]　Yasushi Kanazawa[1,b)]

Abstract:
Homography is one of the most important and fundamental geometric transformation. So, it can be applied to quadric texture mapping, image mosaicing, camera calibration, and so on. The computation time of homography is severe for some applications and will be changed by its implementation. In this study, we compare with the comutational times of the conversion by various implementations of homography computation. The implementations include using only CPU, using SIMD operations and GPGPU with or whiout several preprocessing methods.

1. はじめに

ある平面をある位置から撮影した時の画像を元に別のある位置から撮影した画像への変換である画像の射影変換は基本的な幾何学的画像変換の一つである．画像の射影変換は，3DCG における四角形のメッシュへのテクスチャ貼り付け，カメラ画像のパノラマ合成（モザイキング）処理 [5]，プロジェクションマッピングにおけるテクスチャの変形，マルチプロジェクションにおける幾何補正 [7]，3 次元復元 [3] など様々な用途に応用されている．

これらの応用では画像の射影変換の速度が要求されることがある．例えば，リアルタイムに入力される動画の各フレームを射影変換する場合では，入力のインターバルよりも短い時間で射影変換を終わらせなければ，コマ落ちや映像の遅延を招いてしまう．オフラインで変換を行う場合でも，大量の画像を射影変換しなければならないため，画像の射影変換を高速に終わらせなければ，変換時間の増大を招いてしまう．そのため，多くの場合において，画像の射影変換の高速することは重要な課題となる．

画像の射影変換を高速化するための方法には様々なものが考えられるが，どの方法が最も有効であるかは適用する問題によって変わってくることが考えられる．そこで本研究では，考えられる様々な方法で画像の射影変換を実装し，変換にかかる時間についての実験を行った．また，その結果からどのような問題・状況においてどの方法が有効であるかの比較・検討を行った．

2. 一般的な画像の射影変換アルゴリズム

ある平面を異なる箇所から撮影した画像間の変換を射影変換と呼ぶ．この射影変換は 3×3 の正則行列 $\boldsymbol{H}$ を用いて

[1] 豊橋技術科学大学 情報・知能工学系．Department of Computer Science and Engineering, Toyohashi University of Technology, Japan.
[a)] y-tokiwa@img.cs.tut.ac.jp
[b)] kanazawa@cs.tut.ac.jp

Algorithm 1 一般的な画像の射影変換のアルゴリズム

```
for y' = I'_height/2 − 1 to −I'_height/2 do
  for x' = −I'_width/2 to I'_width/2 − 1 do
    x' ← (x'/f, y'/f, 1)^⊤
    x ← H^{-1} x'
    (x, y, z) ← znorm(x, f)
    I'(x', y') ← sampling(I, x, y)
  end for
end for
```

次のように表すことができる [2], [4].

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} \simeq \boldsymbol{H} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \simeq \begin{pmatrix} H_{11} & H_{12} & H_{13} \\ H_{21} & H_{22} & H_{23} \\ H_{31} & H_{32} & H_{33} \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \quad (1)$$

ベクトル $\boldsymbol{x} = (x, y, 1)^\top$, $\boldsymbol{x'} = (x', y', 1)^\top$ はそれぞれ変換前の画像上の点と変換後の画像上の点を表す同次座標ベクトルである．また，記号 $\simeq$ は定数倍の不定性を除いて等しいことを表す．

式 (1) は，記号 $\simeq$ を使わずに，次のように書ける．

$$x' = \frac{H_{11}x + H_{12}y + H_{13}}{H_{31}x + H_{32}y + H_{33}}, \quad y' = \frac{H_{21}x + H_{22}y + H_{23}}{H_{31}x + H_{32}y + H_{33}} \quad (2)$$

射影変換は，一般的にはアルゴリズム 1 に示す手順で行われる．ここで座標系は，上方向を x 軸，画像の右方向を y 軸とする右手系とする．

アルゴリズム 1 中の $\mathrm{znorm}(\boldsymbol{x}, f)$ はベクトル $\boldsymbol{x}$ の z 要素が仮の焦点距離 f になるようにベクトル $\boldsymbol{x}$ をスカラー倍する操作である．また，$\mathrm{sampling}(I, x, y)$ は画像 I の実数座標 (x, y) の画素値をサブピクセルサンプリングする操作を表す．

3. 高速な画像の射影変換アルゴリズム

変換後の座標 $\boldsymbol{x'}$ は，ラスタ走査の開始点である変換後の画像 I' の左上端上の点から $(-x'_d, y'_d)$ だけ移動した点であるとすると，それらは以下のように定義できる．

$$\boldsymbol{x}'_o = \begin{pmatrix} x'_t \\ y'_l \\ 1 \end{pmatrix}, \quad (3)$$

$$\boldsymbol{x}'_s = \begin{pmatrix} x'_t - x'_d \\ y'_l \\ 1 \end{pmatrix} = \boldsymbol{x}'_o - \begin{pmatrix} x'_d \\ 0 \\ 0 \end{pmatrix}, \quad (4)$$

$$\boldsymbol{x}'_d = \begin{pmatrix} x'_t - x'_d \\ y'_l + y'_d \\ 1 \end{pmatrix} = \boldsymbol{x}'_o - \begin{pmatrix} x'_d \\ 0 \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ y'_d \\ 0 \end{pmatrix} \quad (5)$$

ここで，ベクトル $\boldsymbol{x}'_o$ は変換後の画像 I' の左上隅上の点，ベクトル $\boldsymbol{x}'_s$ は点 $\boldsymbol{x}'_d$ を含むスキャンラインの開始点，ベクトル $\boldsymbol{x}'_d$ は点 $\boldsymbol{x}'_o$ から $(-x'_d, y'_d)$ だけ移動した点とする．

さらに，これらの点に対応する変換前の画像 I 上の点との関係は以下のように書ける．

$$\boldsymbol{x}_o = \boldsymbol{H}^{-1}\boldsymbol{x}'_o \quad (6)$$

$$\boldsymbol{x}_s = \boldsymbol{H}^{-1}\boldsymbol{x}'_s \quad (7)$$

$$\boldsymbol{x}_d = \boldsymbol{H}^{-1}\boldsymbol{x}'_d \quad (8)$$

ベクトル $\boldsymbol{h}_1^{-1}$ と $\boldsymbol{h}_2^{-1}$ は，行列 $\boldsymbol{H}^{-1}$ の 1 列目および 2 列目の列ベクトルとすると，式 (4) より，式 (7) はベクトル ${\boldsymbol{h}^{-1}}_1$ を用いて次のように書き直せる．

$$\boldsymbol{x}_s = \boldsymbol{H}^{-1}\boldsymbol{x}'_o + \boldsymbol{H}^{-1}\begin{pmatrix} -x'_d \\ 0 \\ 0 \end{pmatrix} = \boldsymbol{x}_o - \boldsymbol{h}_1^{-1}x'_d \quad (9)$$

同様に，式 (5) より，式 (8) はベクトル $\boldsymbol{h}_2^{-1}$ を用いて次のように書ける．

$$\boldsymbol{x}_d = \boldsymbol{H}^{-1}\boldsymbol{x}'_o - \boldsymbol{H}^{-1}\begin{pmatrix} x'_d \\ 0 \\ 0 \end{pmatrix} + \boldsymbol{H}^{-1}\begin{pmatrix} 0 \\ y'_d \\ 0 \end{pmatrix} = \boldsymbol{H}^{-1}\boldsymbol{x}'_s + \boldsymbol{h}_2^{-1}y'_d \quad (10)$$

これらの式 (9) と式 (10) は次のことを意味している．まず，変換後の画像 I' 上で 1 ピクセル下に移動すると（スキャンラインを移動すると）変換前の画像 I 上では $-\boldsymbol{h}_1^{-1}$ だけ移動する．また，変換後の画像 I' 上で 1 ピクセル右に移動すると（スキャンライン内で 1 ピクセル進むと）変換前の画像 I 上では $\boldsymbol{h}_2^{-1}$ だけ動く．

つまり，一般的な方法では 1 ピクセル進むごとに対応する座標を計算するために行列とベクトルの積を計算しなおしていたが，最初に $\boldsymbol{x}_o$ さえ計算してしまえば後は 1 ピクセル進む事にベクトルの加算を行い，1 スキャンライン進むごとにベクトルの減算を行うだけで良いということである．このことに基づいた射影変換のアルゴリズムをアルゴリズム 2 に示す．

4. ピクセル書き込み領域の制限による高速化

画像を縮小するような射影変換を行う場合は画素値の計算時に画像 I の領域外を参照してしまう可能性がある．そのため，画素値の計算時に (x, y) が画像の範囲外であれば画像 $I'(x', y')$ にピクセルを書き込まないか，あるいは特定の輝度値で塗りつぶすようにする必要がある．

この範囲外へのピクセル書き込みを検出するためには 1 ピクセルごとに if 文により範囲外かどうか判別をしなければならないが，条件分岐は低速化を招きやすい．また，条

Algorithm 2 高速な画像の射影変換のアルゴリズム

$\boldsymbol{x}'_o \leftarrow ((I_{height}/2-1)/f, (-I_{width}/2)/f, 1)^\top$
$\boldsymbol{x}_o \leftarrow \boldsymbol{H}^{-1}\boldsymbol{x}'_o$
$\boldsymbol{x}_s \leftarrow \boldsymbol{x}_o$
$y' \leftarrow I'_{height}/2-1$
while $-I'_{height}/2 \le y'$ **do**
  $\boldsymbol{x}_d \leftarrow \boldsymbol{x}_s$
  $x' \leftarrow -I'_{width}/2$
  while $x' < I'_{width}/2$ **do**
    $(x, y, z) \leftarrow \mathrm{znorm}(\boldsymbol{x}_d, f)$
    $I'(x', y') \leftarrow \mathrm{sampling}(I, x, y)$
    $\boldsymbol{x}_d \leftarrow \boldsymbol{x}_d + \boldsymbol{h}_2^{-1}$
    $x' \leftarrow x' + 1$
  end while
  $\boldsymbol{x}_s \leftarrow \boldsymbol{x}_s - \boldsymbol{h}_1^{-1}$
  $y' \leftarrow y' - 1$
end while

件分岐でピクセルの書き込みをやめた場合，その条件分岐のための座標計算は結果的にはする必要のなかった無駄な計算ということになる．

ところで，変換後の画像 I' 上のピクセル書き込みを行う領域は射影変換行列 $\boldsymbol{H}$ ごとに決まっており，さらに今回想定している応用では凸四角形から凸四角形への斜影変換である限り塗りつぶし領域は必ず凸四角形になる．このことから，塗りつぶし領域の上端と下端の x 座標を記憶し，さらに各スキャンラインごとに塗りつぶし領域の左端と右端の y 座標を記憶しておくことで，塗りつぶし領域外での計算を全てカットし大幅な時間短縮が見込める．

なお，拡大するような画像の射影変換を行う場合は，画像 I' 全体がピクセル書き込みとなるため，領域外の計算カットによる恩恵を得ることはできないが，変換時に変換後画像の各ピクセルを走査するループ内から条件分岐を排除できるため若干の高速化が見込める．

5. 変換テーブルによる高速化

画像の I' と画像 I の各ピクセルの対応関係は射影変換行列 $\boldsymbol{H}$ により一意に決まる．そこで，画像 I' の各ピクセルが画像 I のどのピクセルと対応するかを予め計算しテーブルとして保持しておき，実際の変換処理の時はこのテーブルを参照してピクセルサンプリングを行うことを考える．

このテーブル化により，ピクセルサンプリングを最近傍法 (Nearest Neighbor) で行うとすれば，対応する座標の計算を行わずにメモリ IO とアドレス計算のための整数演算だけで画像の射影変換をおこなえる．対応する座標を計算するための浮動小数点演算をすべてカットできるため大幅な時間短縮が見込める．また，事前計算のコストを無視できると考えるならば，浮動小数点演算の精度を維持したまま変換速度だけを高速化することができる．

しかし，事前計算に画像の射影変換 1 回分のコストがかかる他，変換後画像サイズと比例するサイズの変換テーブルを事前計算データとして保持しておかなければならないため，事前計算の時間的・空間的コストは高くなりやすい．

さらなる高速化としてピクセル書き込み領域の制限による高速化を組み合わせることで，変換の高速化と変換テーブルのサイズの縮小が可能である．高速化とサイズ縮小の効果は書き込み領域の制限による高速化と同様，ピクセル書き込み領域の狭さと比例する．

6. SIMD による高速化

対応する座標の計算はほとんどをベクトルの演算で行うため，複数の変数に同一の種類の演算を行うことが多くある．そのため，拡張命令を用いてデータだけが違う複数の処理を 1 命令で行うことにより高速化が見込める．このように 1 命令で複数のデータを処理する命令のことを SIMD (Single Instruction Multiple Data) 命令と言い，複数回のスカラ演算を 1 つのベクトル演算にまとめることをベクタライズと言う．

今回は SIMD 拡張命令の一種である SSE (Streaming SIMD Extensions) と SSE2 を使用する．これは 128bit (16Byte) のレジスタにデータを一気に読み込むことができるため，単精度浮動小数点演算の場合は 4 つの浮動小数点を 1 命令で処理できる．今回のケースでは単精度浮動小数点で表現される 3 次元ベクトルを取り扱うため SIMD レジスタ 1 本で 1 つのベクトルを表現できることになる．また，レジスタを任意の個数で切ることで任意サイズの整数に対する演算も可能である．

SSE, SSE2 は本来 CPU のインストラクションであるが，コンパイラに intrinsics として命令を使用するためのインターフェイスが用意されている．本研究ではプログラムを C++で記述し g++でコンパイルしているため，SSE, SSE2 の呼び出しは gcc の intrinscs から行う．

ベクタライズの他にも SSE, SSE2 で実装されている特殊な関数を用いた高速化を図る．まず，逆数の近似値を求める関数を用いて除算を逆数の積に置き換えることで除算を排除し高速化を図る．また，整数値の丸めに専用の関数を用いることで高速化を図る．これらの，特殊な関数は値のレジスタへのストア/ロードの仕方によりスカラ演算として呼び出すことができる．

7. GPGPU による実装

ここまでの方法では全て処理を CPU 上で行ってきたが，そうではなく GPU 上で行うという方法がある．このような汎用の目的のために GPU を利用することを GPGPU (General-purpose computing on graphics processing unit) といい，並列性の高い問題に対して極めて高い効果を発揮する．GPGPU のための開発環境としては CUDA[1] や OpenCL[6] といったものが存在するが，本研究では CUDA を採用する．

表 1　ハードウェア/ソフトウェア環境

ラベル	値
CPU	Intel Core i7-3960X @ 3.30GHz
マザーボード	ASRock Fatal1ty X79 Champion
メモリ	8GBx4(Quad channels)
GPU	GeForce GTX 660(GK106, Kepler)
グラフィックスメモリ	2GB
OS	Debian 7 Wheezy 64bit
C++コンパイラ	g++ 4.7.2(Debian 4.7.2-5)
オプション	-O3 -march=native -DNDEBUG
依存ライブラリ	eigen-3.1.0-1
	OpenCV-2.4.9
	CUDA-6.0
CUDA コンパイラ	nvcc V6.0.1
オプション	-arch=sm_30 -O -DNDEBUG

表 2　手法のラベルとその説明

ラベル	説明
eigen	対応座標の計算に Eigen3 を使用
opencv	変換に OpenCV の関数を使用
standard	従来のアルゴリズムをそのまま実装
fast	高速なアルゴリズムを実装
fast-sse	fast をベクタライズ
fast-bound	fast に領域の制限を追加して実装
fast-bound-sse	fast-bound をベクタライズ
table	変換テーブルを用いて実装
table-bound	table に領域の制限を追加して実装
opencv-gpu	変換に OpenCV の GPU 機能を使用
npp	変換に NPP[*1]を使用
cuda-standard	CUDA を用いて実装
cuda-fast	CUDA で高速なアルゴリズムを実装
cuda-table	CUDA で変換テーブルを用いて実装

CUDA では非常に多数の単純な処理を並列かつ高速かつ効率的に実行することにより高いパフォーマンスを発揮する．そのため，細粒度並列性を持つ問題は CUDA による高速化の恩恵が非常に大きい．画像の射影変換はピクセル単位で並列に計算が可能であり細粒度並列性を持っているため，CUDA による並列処理の恩恵を受けた大幅な変換時間の短縮が見込める．

しかしながら，GPU 上で画像の射影変換を行うためには一度メインメモリから GPU 上のメモリに画像データを転送し，処理が終わった後に GPU 上のメモリからメインメモリに結果を転送する必要がある．そのため，この画像データのアップロード/ダウンロードがボトルネックとなり，CUDA による並列処理の恩恵を相殺してしまう可能性がある．

CUDA 上で画像の射影変換を実装する上でハードウェア的に用意されている特殊関数を使用することでパフォーマンスの改善を行った．1 つは FMA (Fused Multiply Add) 命令による浮動小数点演算コストの削減である．FMA は $a = b \times c + d$ のような積和演算を 1 命令で実行するものである．また，浮動小数点から整数への丸めを行う関数と，逆数の近似値を求める関数を用いた．

さらなる高速化として，高速な画像の射影変換と組み合わた方法と，変換テーブルによる高速化と組み合わせた方法をを実装した．まず，高速な画像の射影変換と組み合わせた方法では 1 スレッドにつき 1 ピクセルの割り当てではなく数ピクセルをまとめて割り当て，まとめて割り当てられた数ピクセルで高速な画像の射影変換を行うことでパフォーマンスの改善を図った．後者の変換テーブルによる高速化と組み合わせた方法ではホスト上で生成した変換マップを GPU 上にアップロードしこの変換マップを元に画像の射影変換を行うというものである．

8. 実験

ここまでに説明した手法に加え，既存のライブラリを用いた画像の射影変換を実装し，それら手法について比較実験を行った．

8.1 実験環境および比較手法

実験を行ったハードウェア/ソフトウェア環境を表 1 に，実装した手法のラベル名とその説明を表 2 に示す．

手法 eigen, opencv, opencv-gpu, npp は比較の上で参考とするために変換処理の一部もしくは変換処理そのものに既存のライブラリを用いている．また，standard から table-bound までの方法の中で SSE がついていない方法では，浮動小数点の整数への丸めに SSE を用いているが，スカラ演算として SSE を用いベクタライズは行っていない．なお全ての手法について，変換範囲外は塗りつぶしをせず，ピクセルサンプリングは NearestNeighbor で行い，浮動小数点演算の精度は単精度である．

8.2 変換時間の測定

画像の射影変換 1000 回にかかる時間を測定した．面積が半分になるように画像を菱型に縮小する射影変換行列を用いた場合の測定結果を表 3 に，画像を拡大するような射影変換行列を用いた場合の測定結果を表 4 に示す．なお，画像はサイズ 1280×960 の 3 チャンネルである．

8.3 事前計算コストの測定

事前計算が必要な方法の事前計算コストについて測定した．結果をまとめたグラフを図 1 に示す．図 1 の横軸の変換回数 0 は初期化にかかった時間を意味し，1 以降は n 回目の変換までにかかった総時間を表している．なお，画像

[*1] Nvidia Performance Primitives

表 3　方法ごとの初期化時間と変換時間 (縮小)

	time[sec]	
ラベル	事前計算	変換
eigen	0	12.738
opencv	0	17.041
standard	0	9.784
fast	0	7.167
fast-sse	0	6.270
fast-bound	0.002572	3.541
fast-bound-sse	0.001671	3.156
table	0.010355	2.339
table-bound	0.008645	1.943
opencv-gpu	0	3.335
npp	0	3.278
cuda-standard	0	3.267
cuda-fast	0	3.410
cuda-table	0.009235	3.254

はサイズ 1280×960 の 3 チャンネルで，射影変換行列には画像拡大するものを用いた．

9. 考察

実験結果を元に実装した方法の比較，検討を行った．

9.1 変換時間の測定

まず，表 3 表 4 全体で最も変換時間が短いのは縮小時の table-bound である．これは，画像全体に書き込みを行わなければならない拡大の場合では 3.860[sec] で，書き込み面積が半分になっている縮小の場合では 1.943[sec] となっていることから，書き込み領域の制限により書き込み領域のサイズと変換時間が比例したためであると考える．また，table の変換時間は表 3 内で 2 番目に短いため，元々 table の変換時間が短いためでもあると考える．

次に，表 3 表 4 の両方で変換時間が短いのは cuda-standard である．縮小のケースにおいて fast-bound-sse, table, table-bound, cuda-table よりも長い変換時間になってしまっているが，それ以外では最も短い変換時間である．このことから，前提条件を置くことによる高速化を除けば，GPU による実装が最も速いことがわかる．

次に，表 3 表 4 中の GPU を使用しない方法で最も変換時間が短いのは table か table-bound である．縮小のケースでは書き込み領域の制限による差が大きくでているが，拡大のケースでは書き込み領域の制限によるさがほとんど出ていない．これは，書き込み領域の制限による恩恵を全く受けられず，領域を制限するためのオーバーヘッドの影響が出たためであると考える．

次に，表 3 表 4 中の事前計算を必要としない方法の中で最も変換時間が短いのは cuda-standard である．これは，GPU による並列計算の恩恵を大きく受けられたためであると考える．また，opencv-gpu や npp と比べて若干変換時間が短いのは FMA などの特殊関数を用いて最適化を図ったためであると考える．なお，cuda-standard は表 4 内で最も変換時間が短い方法でもある．

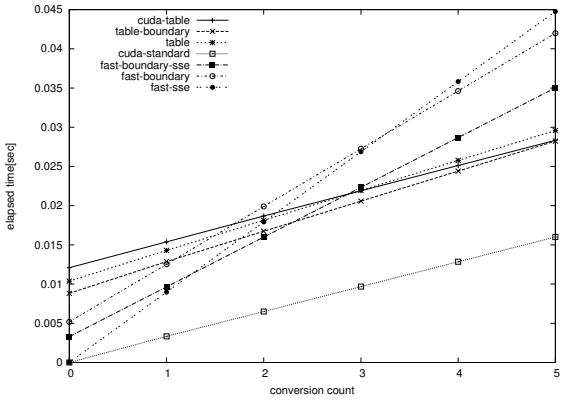

図 1　初期化コストと変換コスト

表 3 表 4 中の GPU を使用せず事前計算を必要としない方法のなかで最も変換時間が短いのは fast-sse である．standard と比較して 80%から 60%程度まで変換時間を削減出来ている．これは，アルゴリズムの改善とベクタライズにより，浮動小数点演算の数が削減されているためであると考える．

これらのことから，仮定を置けない状況で画像の射影変換にかかる時間が短いのは GPU を用いた方法であることがわかる．ただし，事前計算が可能で，かつ書き込み領域が狭くなるという前提がおけるという限られた状況においては書き込み領域を考慮した変換テーブルを用いる方法のほうが変換にかかる時間が短いということがわかる．

表 3 表 4 の両方の事前計算コストについて注目すると，変換テーブルを用いた方法は領域の制限を考慮した方法よりも事前コストが高い．これは，変換テーブルを用いた方法では変換前の画像 I と変換後の画像 I' の対応関係を全て計算しなければならないが，領域の制限を考慮した方法では書き込み領域の境界さえ分かった時点で計算を打ち切るため計算コストが小さくなりやすいためであると考える．

表 3 と表 4 から fast と cuda-fast の高速なアルゴリズムの実装による効果について注目すると，standard と fast では変換時間が短縮されているが，cuda-standard と cuda-fast ではむしろ変換時間が増大している．このことから，高速なアルゴリズムの実装は CPU 上では効果があるが，GPU 上では効果がないことがわかる．これは，高速なアルゴリズムがあるピクセルに対応する座標を計算するために 1 ピクセル前あるいは 1 スキャンライン前の計算に使用した値を必要とする逐次的なアルゴリズムであるため，時間的に並列に命令を実行する GPU での実行に向いていないためであると考える．

表 4　方法ごとの初期化時間と変換時間 (拡大)

ラベル	時間 [sec] 事前計算	変換
eigen	0	14.727
opencv	0	17.737
standard	0	11.160
fast	0	9.041
fast-sse	0	8.917
fast-bound	0.005093	7.329
fast-bound-sse	0.003310	6.688
table	0.010983	3.821
table-bound	0.009103	3.860
opencv-gpu	0	3.021
npp	0	2.959
cuda-standard	0	2.929
cuda-fast	0	2.954
cuda-table	0.013363	2.962

9.2　事前計算コストの測定

図 1 の cuda-standard について着目すると，全ての変換回数において他の全ての方法よりも短い時間で変換している．

変換回数 1 について着目すると事前計算を必要としない fast-sse と cuda-standard の経過時間は事前計算を必要とする他の方法よりも短い．このような結果になった理由としては，fast-sse と cuda-standard には事前計算時間が存在しないためであると考える．

変換回数 2 については cuda-standard とそれ以外の方法に分かれている．cuda-standard 以外の方法の間にはそれほど大きな差は見られない．

変換回数 3 以降では cuda-standard とテーブルを用いた方法とアルゴリズムに工夫をした方法の 3 つのグループに分けられる．テーブルを用いた方法は 3 回目以降の変換で事前計算を必要としない fast-sse よりも経過時間が短くなる．fast-bound-sse は 4 回目以降の変換でテーブルを用いた方法よりも経過時間が長くなる．

これらのことから，同一の射影変換行列を用いて画像の射影変換を行う場合，数十回程度の変換なら GPU での計算が最も速いことがわかる．GPU を使用できない場合の 1 回だけの変換なら事前計算を必要としない fast-sse が最も速いことがわかる．GPU を使用せず同一の射影変換行列で 3 回以上の複数回画像の射影変換を行う場合は table か table-bound が最も速いことがわかる．

9.3　全体の考察

まず，使用する射影変換行列についての仮定を置かない場合は GPU を用いた画像の射影変換が最も適している．GPU を用いた場合は入力画像を与えてから出力画像が戻ってくるまでの時間が他の全ての方法と比べて非常に短いため，時刻ごとに変化する射影変換行列を用いて短い応答時間で画像の射影変換を行わなければならないような状況に適している．

しかしながら，変換後画像 I' 上の書き込み領域が比較的狭くなる傾向にあるという仮定を置くことが出来，同一の射影変換行列を用いて何度も画像の射影変換を繰り返すという限定された状況においては領域の制限を考慮した変換テーブルを用いた方法がもっとも有効である．これは，例えばプロジェクションマッピングなどの用途において動画の全てのフレームを変換したいという時に有効である．

何らかの理由で GPU を使用することができず，かつ同一の射影変換行列を用いて何度も画像の射影変換を繰り返すという場合はテーブルを用いた方法が最も適している．これは例えば，動画に何らかの変換を施すときに，すでに GPU で何らかの計算を行っているために使用することができないというな状況のことである．

もし，GPU を使用できず，かつ 1 回の画像の射影変換ごとに射影変換行列が変化するような場合では高速なアルゴリズムを SIMD で実装した方法が最も適している． 1 回の変換ごとに射影変換行列が変化するような場合とは，何らかの反復的な手法の中で射影変換行列の推定と画像の射影変換を交互に繰り返すような状況のことである．

10.　まとめ

画像の射影変換について様々な方法を実装し，実際に変換にかかる時間や事前計算にかかる時間を測定し，それらを比較しどの方法がどのような状況に適しているか検討した．

結果，GPU を用いて変換を行う方法は状況によらず高速であることがわかった．GPU を使用しない場合は事前計算の有無により適した方法が変わる．事前計算有りの場合ではテーブルを用いた方法が，事前計算無しの場合では高速なアルゴリズムを SIMD で実装した方法が適していることがわかった．

参考文献

[1] http://www.nvidia.co.jp/object/cuda-jp.html
[2] R. Hartley and A. Zisserman, “Multiple View Geometry in Computer Vision,” 2nd Edition, Cambridge University Press, 2004.
[3] 船本 将平, 金澤 靖, “複数の射影変換行列を用いた単眼移動カメラによるシーンの 3 次元復元,” 情処研報: CVIM, vol.2009-CVIM-166, no.15, pp.97–104, March 2009.
[4] K. Kanatani, “Statistical Optimization for Geometric Computation: Theory and Practive,” Elsevier Science, 1996.
[5] 金澤 靖, 金谷 健一, “段階的マッチングによる画像モザイク生成,” 信学論 D-II, Vol. J86-D-II, No.6 (2003), pp. 816–824.
[6] http://jp.khronos.org/opencl/
[7] 安田 朋広, “マルチプロジェクションシステムにおける映像の幾何補正と投影方法に関する研究,” 豊橋技術科学大学修士論文, 2010.