GPUを用いたビデオ映像の安定化

藤澤 誠[†1] 天谷 貴大[†2] 三浦 憲二郎[†3]

この論文では，ビデオ映像に含まれる振動成分を取り除くための処理の計算をGPUを用いて行う手法を提案する．映像の安定化処理には，グローバルモーションの推定，振動補正，モザイキングの3つの処理を行うが，CPUでこれらの処理を行うと処理時間が長く，その中でもグローバルモーションの推定が処理の大半を占めている．そこで，並列処理が可能なGPUで計算処理を行うことで処理時間の短縮を図った．提案した手法は，ビデオ映像のフレーム画像をテクスチャデータとしてGPUに転送し計算を行い，計算結果をオフスクリーンバッファに描画し，ピクセルの値を読み込むことによって結果を得る．ピクセルを読み込む速度は描画速度に比べて時間がかかるため，計算結果を1つのピクセルにまとめることで，読み込み時間を短縮することに成功した．

Video Stabilization with GPU

Makoto Fujisawa,[†1] Takahiro Amaya[†2] and Kenjiro T. Miura[†3]

This paper proposes a fast computational method of video stabilization using the Graphics Processing Unit (GPU) that removes unwanted vibrations from videos. The video stabilization is composed of estimation of the global motion, removal of the undesired motion and mosaicking. When they are processed with CPU, the computational cost for the global motion estimation is very high. We improve the speed of this computation with GPU that enables parallel processing. Our method can obtain the result by forwarding the frame image of the video to GPU as texture data, and drawing the calculation result to the offscreen buffer. Although the transfer speed from GPU to CPU is very slower than the other way around, the method only has to transfer one pixel data from GPU.

1. 緒　　言

災害時において活躍するレスキューロボットは荒れた路面や地震により不安定な状況で走行する．そのため，ロボットに搭載されたカメラから送られてくる映像にはゆれが生じ，即座の状況把握が困難になり，オペレータが画面酔いを起こして操作に影響が出る可能性がある．したがって，映像のゆれによる影響を抑えるために，リアルタイムでの動画像処理を行いゆれを軽減する必要がある．

現在，デジタルカメラ向けに開発，研究されているゆれを軽減する手法として，電子式，光学式手ぶれ補正などがあげられる．しかしながら，これらはカメラに対する補正であり，そのカメラで撮影した映像だけしか補正できず，またすべてのカメラにこれらの手法を搭載できるとは限らない．そのため，どのような映像でも処理できるようにするにはPCを利用した安定化処理が望まれる．しかしながら，動画像処理はデータ量が多く，それらを処理するにはCPUでは負荷がかかりすぎリアルタイムでの処理は難しい．そこで，処理時間を短縮するために，並列処理による高速演算が可能なGPU（Graphics Processing Unit）にCPUで負荷が多くかかる計算処理を行わせる．GPUは，元々グラフィックス処理専用のプロセッサだったが，近年ではプログラマブルシェーダの搭載により，Cg言語[3]などを使用して，これまでCPUで行ってきた汎用計算がGPUでも可能になった．GPUを計算に使用している研究例として，流体などのシミュレーション[4]や画像処理[9]，形状処理[15]などがあげられる．この論文では安定化処理を，専用のハードウェアではなく汎用のハードウェアを用いて行うことで，一般のPCでも簡単に行えるようにし，低コストかつリアルタイムでの映像安定化を実現することを目的とし

†1 静岡大学大学院理工学研究科
Graduate School of Science and Engineering, Shizuoka University
†2 静岡大学大学院工学研究科
Graduate School of Engineering, Shizuoka University
†3 静岡大学創造科学技術大学院
Graduate School of Science and Technology, Shizuoka University

た，GPU を用いた映像の安定化を提案する．

映像の安定化には，Litvin らの手法[7] や Matsushita らの手法[8] があり，どちらもカメラの動き（グローバルモーション）を推定し，それを基に振動補正を行う．Litvin らは，グローバルモーションにカルマンフィルタリングを行うことによってゆれを抑えた動きを求め，フレーム画像を変形させる．さらに，変形による画像の劣化をモザイキングを行うことで補間する．しかし，モザイキングでは映像の中の物体の動き（ローカルモーション）を補間しきれない．そこで，Matsushita らはローカルモーションを推定し[2]，それを Motion inpainting によって補間することで，より良画質な映像を生成する手法を提案した．また，彼らは，階層的な運動推定[1] を行うことで，グローバルモーションの推定時間を減少させ，さらに，ガウスカーネルを用いることで映像の不必要なぶれを取り除くことで振動補正を行った．しかし，ローカルモーションの推定には多くの時間を費やしてしまうためリアルタイム処理が難しくなる．我々は，ガウスカーネルを用いて振動補正を行い，モザイキングによって補間を行う．

コンピュータビジョン分野においても，GPU を使い画像処理を高速化するための研究がさかんに行われている．GPU でグラフィック計算以外のより一般的な処理を行う際には，CPU とのデータの受け渡し方法が問題となる．一般的には CPU から GPU へはテクスチャを，その逆にはピクセルバッファを用いる．より詳しいことは文献 5) を参考にしてほしい．文献 5) は画像処理を GPU で行うためのフレームワークを示した．テクスチャやピクセルバッファは平面画像データであり，画像処理で扱う静画像と親和性が高い．また，動画像も各フレームごとでは静画像として扱えるため，動画像処理を GPU で高速化する研究も数多くなされている．Strzodka ら[12] は動画像に対するローカルモーション推定とその視覚化を GPU によってリアルタイムに実行した．Sinha ら[11] は動画像の特徴追跡を GPU で処理することで KLT 特徴追跡アルゴリズム[14] を CPU 処理に比べて 15 倍以上高速化し，リアルタイムでの実行を可能とした．この特徴追跡を用いてグローバルモーション推定を行うことも可能である[10] が，映像中の動物体の動きに全体の動きが作用され，安定した推定が難しいと考え，我々は Litvin らの安定化手法[7] を GPU で高速化する．動画像の隣接したフレーム間の動きをアフィン変換と仮定し，その差分値の合計をエラー値として最小化手法で最適解を求める．このとき，CPU 処理における計算時間のボトルネックとなっているエラー値計算部分を GPU で並列に計算するために提案されたベクトル要素の合計値計算手法[6] などを用いて高速に計算する．

この論文の構成は以下である．2 章で安定化のアルゴリズム，グローバルモーションの推定，振動補正，モザイキングについて説明し，3 章ではグローバルモーションの推定における計算を GPU で行う手法を示す．そして，4 章で本手法の結果と CPU との計算速度の比較結果を示し，5 章において結言を述べる．

2. 映像の安定化

不必要な振動を含む映像の安定化は，カメラの動きであるグローバルモーションの推定，グローバルモーションからの撮影者が意図しない動きの検出とその除去，および，振動補正によって生じる未定義領域の補間，の 3 つで構成される．

2.1 グローバルモーションの推定

映像の安定化を行うにはグローバルモーションを知る必要がある．グローバルモーションはカメラ自身の動きであり，映像内のオブジェクトの動きはローカルモーションと呼ばれる．除去したい動きはカメラ自身が振動することで，画面全体が揺れることによって起こり，その動きはアフィン変換で表すことができる．

隣接するフレーム I^n と I^{n+1} の間のアフィン変換を，

$$\begin{aligned}\boldsymbol{x}_{n+1} &= \begin{pmatrix} a_1 & a_2 \\ a_3 & a_4 \end{pmatrix} \begin{pmatrix} x_n \\ y_n \end{pmatrix} + \begin{pmatrix} b_1 \\ b_2 \end{pmatrix} \\ &= \boldsymbol{A}_n^{n+1}\boldsymbol{x}_n + \boldsymbol{b}_n^{n+1}\end{aligned} \tag{1}$$

と表す．ここで，$\boldsymbol{x}_n = (x_n, y_n)^T$ は n フレームでのピクセル座標である．$(\boldsymbol{A}_n^{n+1}, \boldsymbol{b}_n^{n+1})$ の推定は以下のエラー関数の最小化問題となる．

$$\begin{aligned}&E(n, n+1) \\ &= \sum_{\mathbf{x}\in\chi} \varphi(I^n(\boldsymbol{x}_n) - I^{n+1}(\boldsymbol{A}_n^{n+1}\boldsymbol{x}_n + \boldsymbol{b}_n^{n+1}))\end{aligned} \tag{2}$$

ここで，χ は画面平面上すべての座標値の集合，$I(\boldsymbol{x})$ はピクセル $\boldsymbol{x}$ の輝度値であり，$\varphi(x) = \sqrt{x^2 + \beta}$ である．本研究では $\beta = 0.01$ とした，式 (2) のエラー関数は，アフィン変換した座標 (x', y') における $n+1$ フレームの画像と座標 (x, y) における n フレームの画像の各ピクセル輝度値の差を合計する．輝度値の差の合計をエラー値とし，その最小化によりアフィン変換 $(\boldsymbol{A}, \boldsymbol{b})$ を算出する．ただし，場面が急激に変わるフレーム間や画面内を動的な物体が多く占める場合（ローカルモーションが大きい場合）は，対応点がとれずにグローバルモーション推定に失敗する可能性があ

ることに注意しなければならない．また，ピクセル座標値のアフィン変換 $\boldsymbol{A}_n^{n+1}\boldsymbol{x}_n + \boldsymbol{b}_n^{n+1}$ によって，輝度値が定義されていない領域（未定義域）を参照する場合，そのピクセルはエラー値の計算から除外する．エラー値 E は最終的に有効であったピクセル数 χ_e で補正する $\hat{E} = (\chi/\chi_e)E$．ただし，$\alpha = \chi_e/\chi$ が小さい場合（たとえば 1/3）に正しい結果が出ない可能性がある（実際のカメラの動きが小さくても最小化手法の反復の初期において $\boldsymbol{A}$ や $\boldsymbol{b}$ の値が大きくなることがある）．本研究では $\alpha < 1/3$ 以下では未定義域のピクセルの輝度値を 0 としてエラー値を計算し，エラー値が意図的に大きくなるように実装した．もちろん，実際のカメラの動きがあまりに高速であった場合，これは不正確な結果を生むことに注意する必要がある．

この論文では，映像の動きを平行移動と回転移動のみと仮定してアフィン変換を

$$\mathbf{x}_{n+1} = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} b_1 \\ b_2 \end{pmatrix} \quad (3)$$

とし，求めるパラメータを (θ, b_1, b_2) の 3 つに減らすことで，最小値探索における計算速度を向上させる．最小値の探索には関数値のみで実装できる Powell 法と勾配値（導関数）を用いて探索する準ニュートン法（BGFS 法）を使用する[13]．BGFS 法を使用するときの導関数は，

$$\begin{aligned} \frac{\partial E}{\partial \theta} &= -\sum_{\mathbf{x}\in\chi} \sqrt{\frac{\Delta I^2}{\Delta I^2 + \beta}} \left(\frac{\partial I^{n+1}}{\partial x_{n+1}} \frac{\partial x_{n+1}}{\partial \theta} + \frac{\partial I^{n+1}}{\partial y_{n+1}} \frac{\partial y_{n+1}}{\partial \theta} \right) x, \\ \frac{\partial E}{\partial b_1} &= -\sum_{\mathbf{x}\in\chi} \sqrt{\frac{\Delta I^2}{\Delta I^2 + \beta}} \frac{\partial I^{n+1}}{\partial x_{n+1}}, \\ \frac{\partial E}{\partial b_2} &= -\sum_{\mathbf{x}\in\chi} \sqrt{\frac{\Delta I^2}{\Delta I^2 + \beta}} \frac{\partial I^{n+1}}{\partial y_{n+1}}. \end{aligned} \quad (4)$$

ここで，

$$\begin{aligned} \Delta I &= I^n(\mathbf{x}_n) - I^{n+1}(\mathbf{x}_{n+1}), \\ x_{n+1} &= x\cos\theta - y\sin\theta + b_1, \\ y_{n+1} &= x\sin\theta + y\cos\theta + b_2, \\ \frac{\partial x_{n+1}}{\partial \theta} &= -x\sin\theta - y\cos\theta, \\ \frac{\partial y_{n+1}}{\partial \theta} &= x\cos\theta - y\sin\theta, \end{aligned} \quad (5)$$

である．

2.2 振動補正

推定したグローバルモーションをもとに，Matsushita らの方法[8] を用いて振動を補正する．補正するフレームの前後 k フレームを利用して補正変換 S_n を

$$\boldsymbol{S}_n = \sum_{m=n-k}^{n+k} \boldsymbol{T}_n^m \star G(k) \quad (6)$$

によって求める．ここで，$\boldsymbol{T}_n^m$ はフレーム n から m までのアフィン変換，G はガウスカーネル，そして $\star$ は畳み込み演算子である．得られたアフィン行列を用いて振動補正を行う．

$$\bar{\boldsymbol{x}}_n = \boldsymbol{S}_n\boldsymbol{x}_n = \bar{\boldsymbol{A}}_n\boldsymbol{x}_n + \bar{\boldsymbol{b}}_n \quad (7)$$

図 1 に振動補正を行う前後の X 軸方向と Y 軸方向のカメラの動きの変位量を示す．破線のグラフが補正前の X 軸方向，Y 軸方向のカメラの動きを示しており，撮影者の意図的なカメラ動作による周期の大きな動きと，意図しない振動による細かな周期の動きが混在している．実線のグラフは補正後のカメラの動きである．撮影者の意図的なカメラ動作を残したまま全体の動きが滑らかになっている．

図 2 にこの補正後のカメラの動きによって振動を除去したフレーム画像を示す．図の左が元のフレーム画像，右が補正後のフレーム画像である．元のフレーム画像を変形させたため，補正後のフレーム画像はピクセルの未定義領域（黒く塗りつぶした領域）が存在する．

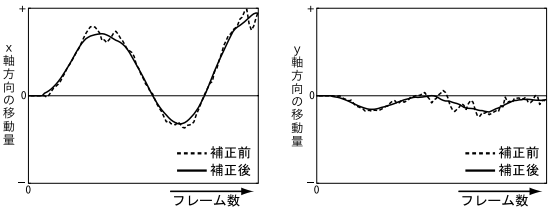


図 1 カメラの移動量の推移
Fig. 1 Translations along X and Y direction of the camera.

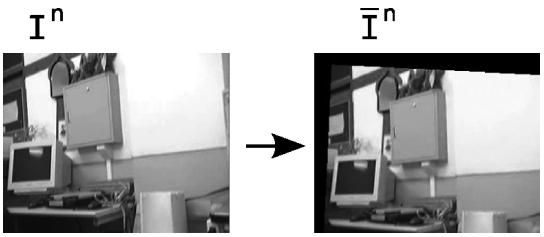


図 2 振動補正後のフレーム画像．黒い部分は未定義領域である
Fig. 2 Left: original image, right: stabilized image. The black area represents the missing image area.

2.3 モザイキング

振動補正したフレームに発生したピクセルの未定義領域は，Litvin らのモザイキングを用いた手法[7] で補間する．周囲のフレーム $\bar{I}^{n+m}$ を補間の対象となるフ

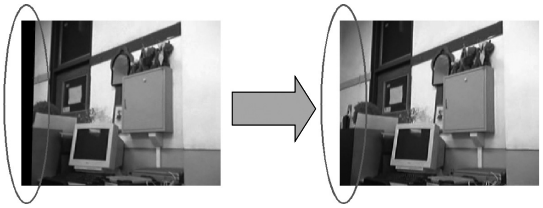


図 3 モザイキング結果
Fig. 3 Result of mosaicking.

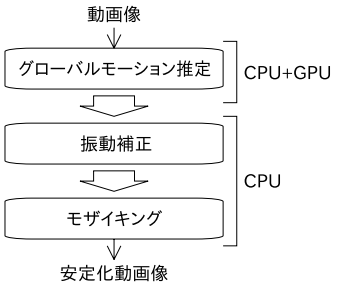


図 4 計算の流れ
Fig. 4 Flowchart of the computation.

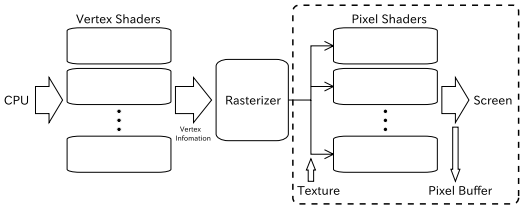


図 5 GPU の構造
Fig. 5 Architecture of the GPU.

レーム $\bar{I}^n$ の位置に変形（$\bar{I}^{n+m} \rightarrow \bar{\bar{I}}^{n+m}$）させ，以下の式でモザイキングを実行する．

$$\bar{I}^n = \frac{1}{\sum E_{\mathrm{inv}}(n,m)} \sum_{m=-M, m\neq 0}^{M} E_{\mathrm{inv}}(n,m)\bar{\bar{I}}^{n+m} \tag{8}$$

ここで $E_{\mathrm{inv}}(n,m)$ はフレーム n と $n+m$ の間のエラー値 $E(n,m)$ の逆数である．E_{inv} を重みとして周囲 m フレームの重み付き平均をとることで，よりフレーム画像との近似度が高い画像を優先的に混ぜ合わせる．

式 (8) を用いて補正をかけることによって未定義領域のピクセルを補間することができる．**図 3** にモザイキングを実際に行った画像を示す．丸で囲った未定義領域が補間された．

3. GPU での実装

2 章で提示した安定化処理を CPU 上で行うと非常に時間がかかる．4 章の結果の表 1 に示した安定化処理各ステップの CPU での計算時間から，グローバルモーション推定の処理時間が長く，この部分をさらに詳しく調べた結果，その 99% 以上を占めるのはエラー値とその勾配値の計算であった．CPU では輝度値の差を 1 ピクセルずつ計算しているためであり，画像の大きさに比例して計算時間は増大する．これを GPU で並列に計算することで高速化する．

図 4 に全体の計算の流れを示す．まず，動画像データをメインメモリから GPU に転送し，GPU がエラー関数（式 (2)）およびその勾配値（式 (4)）を計算することで，グローバルモーション推定を行う．グローバルモーション推定では，最小化計算の反復を CPU が行い，各反復において GPU を呼び出してエラー値を受け取る．グローバルモーション推定結果から，CPU が振動補正とモザイキングを行い，動画として出力する．このとき，振動補正には前後数フレームが必要なので結果の動画は数フレーム遅れて表示される．

この章では，3.1 節で画像データの GPU への転送を，3.2 節で GPU でのエラー値の計算手法について述べる．

3.1 GPU へのデータ転送

GPU の基本的な構造を**図 5** に示す．我々は図中点線の枠で示したピクセルシェーダでエラー値計算を行う．ピクセルシェーダは複数存在し（NVIDIA GeForce8800 で最大 128 個），描画画面上の各ピクセルごとに並列に処理することができるため，各ピクセルを画像の各画素に割り当てることで高速な並列計算が可能である．しかし，GPU は汎用計算用に作られていないため，プログラミング言語で一般的に用いられている配列などのデータ構造に格納したデータを直接渡すことはできず，動画像の各フレーム画像をどのように GPU に転送するのかが問題となる．

我々はテクスチャを用いて画像を GPU に転送する．図 5 のようにピクセルシェーダは Rasterizer からのデータとテクスチャデータを受け取ることができる．テクスチャデータはコンピュータのメインメモリ上から GPU のテクスチャメモリに転送して用い，多くのデータを一気にユーザが制御できる．実際の転送は画像と同じ大きさのビューポートにテクスチャを貼り付けた同サイズの矩形ポリゴンを描画することで各ピクセルシェーダから参照できる．元の画像データは RGB 各輝度値が 8 ビットの符号なし整数で表されているが，エラー値や導関数の計算において線形補間などにより浮動小数点数が必要となる．これは，OpenGL の

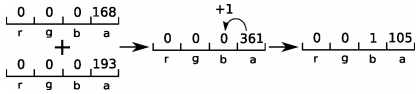


図 6 RGBA による浮動小数点数の表現
Fig. 6 Floating-point numbers represantation by RGBA.

NV_texture_rectangle 拡張による 32 ビット浮動小数点テクスチャを用いることで解決した[5]．

NV_texture_rectangle 拡張は環境によっては対応していないことがあるが，各 8 ビットカラーを**図 6** のようにつなげることで未対応の環境でも浮動小数点数を扱える．また，導関数の計算のように，値のとりうる範囲が広い場合や，符号付の計算の場合にはテクスチャを複数枚使用して計算を行うことで対応可能である．

GPU で計算した結果を再度 CPU に転送するために，ピクセルバッファ（pixel buffer，以下 pbuffer）を用いる．pbuffer はオフスクリーンレンダリングのためのバッファであり，pbuffer を有効にした後，画面描画を行うとフレームバッファではなく pbuffer 上に描画される．pbuffer に格納された画像データは CPU にリードバックでき，テクスチャとして GPU から読み込むことも可能である．さらに，NV_texture_rectangle 拡張により浮動小数点データを扱える．

3.2 エラー値の計算

GPU は画面ピクセルごとの計算をピクセルシェーダで並列に実行することが可能であり，これを適切に利用することで大幅な高速化が可能である．そのために式 (2) のエラー値の計算を以下の 2 ステップに分割する：(a) 2 画像間の輝度値の差を求めてそれを格納した画像（差分画像 $\varphi(\Delta I)$）を生成し，(b) 差分画像の各画素の輝度値 $\varphi(\Delta I(\boldsymbol{x}))$ を文献 6) の手法で合計する．(a) の差分画像を生成する際には，式 (2) の $I^{n+1}(\boldsymbol{A}_n^{n+1}\boldsymbol{x}_n + \boldsymbol{b}_n^{n+1})$ のアフィン変換も GPU 側で行う．しかし，3.1 節で述べたように GPU へ画像はテクスチャとして転送されるため，GPU 側からはテクスチャ座標を用いて参照する必要がある．よって，テクスチャ座標 $(u, v)^T$ を画像のピクセル座標 $\boldsymbol{x} = (x, y)^T$ に変換する．

$$\begin{cases} x = u \times w_I, \\ y = v \times h_I. \end{cases} \tag{9}$$

ここで w_I，h_I はそれぞれフレーム画像の x 方向ピクセル数，y 方向ピクセル数である．画像のアフィン変換では変換後のピクセル位置が元の画像のピクセル位置に必ずしも一致するとは限らないため，線形補間などで周囲のピクセル値を補間した値を用いる．差分画像は 3.1 節で述べた pbuffer に格納する．

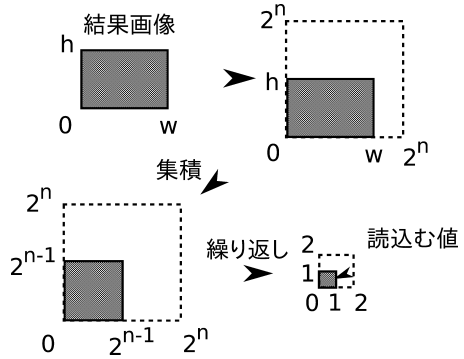


図 7 ピクセルの集積
Fig. 7 Pixels of the image are cumulated in a pixel.

pbuffer に格納された差分画像から (b) の合計値計算を行う．GPU に搭載された多数のピクセルシェーダによる並列計算を効率的に行うためには，合計値計算を単なる反復による逐次計算ではなく，個々に独立した計算ユニットを複数同時に実行する方法が必要である．我々は，画像の縦・横のサイズを 1/2 にしていき，各画素の値を元の画像の 4 ピクセルの合計値とする手法[6] を用いる．

まず，計算結果の画像を $w_I <= 2^m$ かつ $h_I <= 2^m$ となる $2^m \times 2^m$（m は最小の整数）を描画範囲とした pbuffer に出力する．この際，pbuffer の背景色は後の計算に影響がないように黒（RGBA $= (0.0, 0.0, 0.0, 0.0)$）にしておく．そして，それを 1/2 サイズの pbuffer に貼り付ける．描画する画像のピクセル座標 (x, y) での輝度値を以下で更新する．

$$\begin{aligned} \Delta I^*(x, y) = {} & \Delta I(2x, 2y) + \Delta I(2x + 1, 2y) \\ & + \Delta I(2x, 2y + 1) \\ & + \Delta I(2x + 1, 2y + 1) \end{aligned} \tag{10}$$

これにより $2^{m-1} \times 2^{m-1}$ の大きさの新たな画像が生成される．さらにこの画像を別の pbuffer に貼り付けて同じ作業を m 回繰り返すことで，差分値を合計した値を持つピクセルで構成される 1×1 画像が取得できる（**図 7**）．

この手法の加減演算回数は，$2^m \times 2^m$ の大きさの画像では $(3/2)2^{2m} - 1$ であり，逐次反復して求める手法では 2^{2m} であるので約 1.5 倍と多くなる．しかし，上記のとおり複数のピクセルシェーダで式 (10) を同時並列に計算できるため，実際の演算では処理時間が非常に少なくすむ．そして，画像の解像度が大きくなっても面積に比例した処理時間とはならない．

最終的な合計値を CPU に転送する．CPU への転送には OpenGL の glReadPixel を用いる．glRead-

Pixel はフレームバッファの値をメインメモリに転送する関数であるが，pbuffer を有効にしていればその値がメインメモリに転送される．しかし，glReadPixel は CPU から GPU にデータを送る場合と比べて時間がかかる．本来の GPU の目的は画面への描画であり，本来 CPU にデータを返すことがなく，その帯域が狭いためである．本研究では，CPU に返すべき計算結果は 1 ピクセル分のみであるので，CPU に返すデータ量が極力少なくすみ，これによる処理時間への影響はほとんどない．

4. 結　　果

CPU と GPU によるビデオ安定化を実装した結果を示す．GPU 実装には NVIDIA の Cg 言語と OpenGL を用いた．使用した映像は，サイズが 320×240 で，フレームレートは 30 fps（frame per second），総フレーム数 187 の動画である．また，計算に用いた PC 環境は CPU：Pentium D 3.4 GHz，メモリ：2,048 Mbyte，GPU：GeForce 8800 GTX である．

サイズが 320×240 の動画のグローバルモーション推定結果を**図 8** に示す．図 8 では CPU と GPU のグローバルモーション推定（BGFS 法）によって得られた隣接フレーム間の x 軸方向変位とその差の絶対値をプロットした．CPU と GPU にある程度の差が存在する．これは，GPU 内部の浮動小数点数が 16 ビットで処理されている可能性があること，glReadPixel で GPU から CPU にデータを転送する際，[0, 255.0] で表現された色が [0, 1.0] に変換されることで丸め誤差が発生したことなどが原因と考えられる．しかし，差の大きいところでも 0.1 ピクセル以下であり，安定化した結果の動画を人の目で見てもその違いをほとんど判別できないレベルである．

図 9 に BGFS 法による各フレームでのグローバルモーション推定時間を示す．GPU では 0.2 [s] 前後で安定して推定できている．Powell 法，BGFS 法による CPU と GPU のグローバルモーション推定，CPU による振動補正・モザイキングの平均処理時間の比較を**表 1** に示す．エラー値とその勾配値を用いる BGFS 法の方が，エラー値のみを用いる Powell 法よりも収束が速く高速に推定できる．我々のビデオ安定化手法では，グローバルモーション推定が処理時間のほとんどを占めており，このグローバルモーション推定を GPU で処理することで，Powell 法で約 10 倍，BGFS 法で約 15 倍の高速化に成功した．

次に，画像の解像度を変えたときのグローバルモーション推定時間の変化を調べた．ハイビジョンカメラ

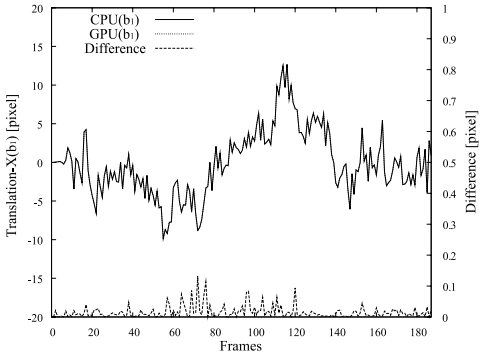


図 8 CPU と GPU による x 軸方向のグローバルモーション（式 (1) の b_1）推定結果（左軸）とその差の絶対値（右軸）をプロットした

Fig. 8 Translation along the X direction estimated by CPU and GPU, and their difference.

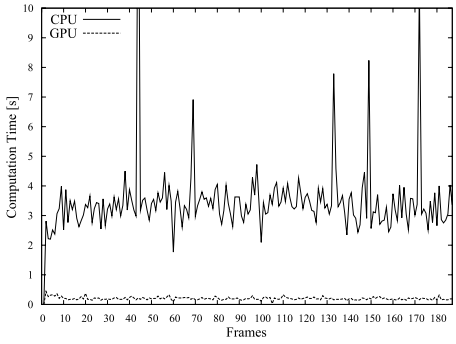


図 9 CPU と GPU によるグローバルモーション推定時間

Fig. 9 Global motion estimation times by CPU and GPU.

で撮影した動画（サイズ 1,920×1,080）の中心部分を 1,000×1,000 にトリミングし，それを元に 100×100 まで画像幅を 100 ピクセルずつ縮小させた 10 個の映像で解析した．フレームレートは 30 fps（frame per second），総フレーム数 300 の動画である．**図 10** に画像の大きさによる計算時間の変化のグラフを示す．横軸は画像の横幅ピクセル数，縦軸はグローバルモーションの平均計算時間である．CPU の計算では画像の総ピクセル数に比例した計算時間がかかった．たとえば，400×400 の画像では平均計算時間は約 5.94 秒，総ピクセル数がその 4 倍である 800×800 の画像では約 24.4 秒であった．これは式 (2) の計算を 1 ピクセルずつ逐次計算し合計しているためである．一方，GPU では 400 × 400 の画像の平均計算時間は約 0.158 秒，800 × 800 の画像の平均計算時間は約 0.191 秒となった．GPU による計算では並列計算により総ピクセル数に比例した計算時間とはならず，ピクセル数が増えても高速に計算可能である．GPU における処理時間の変化を見るために y 軸のスケールを変えたグラフ

表 1 1 フレームあたりの平均計算時間の比較
Table 1 Comparison of computational times.

	Powell 法（CPU）	Powell 法（CPU+GPU）	BGFS 法（CPU）	BGFS 法（CPU+GPU）
モーション推定	27.78	2.68	3.48	0.20
振動補正	5.03×10^{-6}			
モザイキング	0.03			
合計	27.81	2.71	3.51	0.23

(sec/frame)

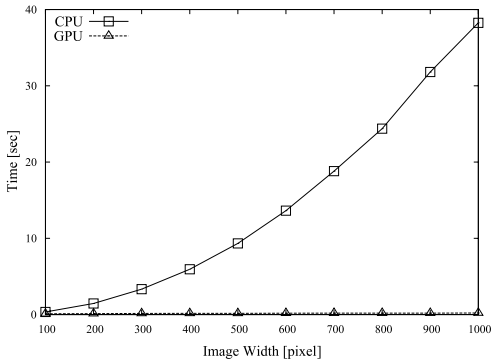


図 10 画像の解像度によるグローバルモーション平均推定時間の変化
Fig. 10 Averages of global motion estimation times with respect to resolution of images.

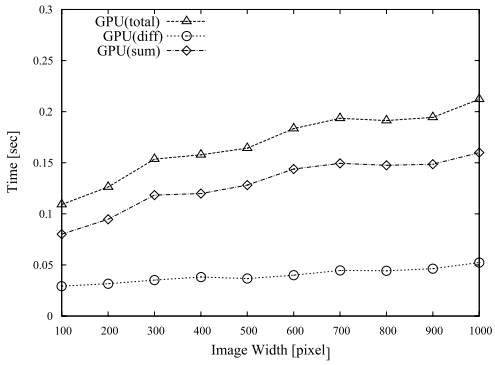


図 11 画像の解像度によるグローバルモーション平均推定時間の変化（GPU のみ）
Fig. 11 Averages of global motion estimation times with respect to resolution of images (GPU only).

を**図 11** に示す．図 11 では 3.2 の (a) 差分画像生成（diff）と (b) 差分画像の合計（sum）に要した時間も計測しそれぞれプロットした．(a) の差分画像生成は解像度の変化に対する処理時間の増加が少なく，(b) の合計値計算が合計の処理時間の増減に大きく影響している．特に，画像幅が 100 から 200，200 から 300，500 から 600 の間で他と比べて処理時間の変化が大きい．これらは，合計値計算の際の最初の pbuffer 確保量がそれぞれ，128 から 256，256 から 512，512 から 1,024 に増えるところであり，それによる合計値計算処理の大幅な増大が原因である．

安定化によるフレーム画像の修正結果を**図 12** に示す．図 12 の左の列は補正前のフレーム画像，中心の列は振動補正を行った後のフレーム画像，右の列は補正後のフレーム画像にモザイキングを行った結果画像である．モザイキングによる未定義領域の補間がなされている．しかし，本実験で用いた動画は動物体を含まないものを用いており，動物体が画像境界付近に存在していたとき，モザイキングによる画像補間では元の画像とのずれが生じる．この解決法としては，モーションインペインティングを用いた補間[8] などがある．

5. 結　　言

本研究では，リアルタイム処理が今後必要になってくるであろう PC を用いた映像安定化手法の CPU 処理における計算時間のボトルネックとなっているエラー値計算部分を GPU で並列に計算することでその高速化に成功した．しかし，現状では BGFS 法でも 5 fps 程度であり，リアルタイムは実現できていない．ただ，今後の GPU の進化やアルゴリズムの改良によって十分リアルタイムが実現できる数値である．また，GPU の計算では画像の解像度に依存しない計算速度により，今後普及が進むと思われるハイビジョンカメラの解像度でも現在のカメラとほとんど変わらない速度で安定化が可能である．

今後のさらなる高速化のための展望としては，

- 2 枚のグラフィックカード（GPU）を並列に実行することでほぼ 2 倍に性能を向上させる技術（ex. NVIDIA SLI（スケーラブルリンクインタフェース）），同様に 4 枚を用いたもの（ex. QuadSLI）の利用，
- グローバルモーション推定処理のための階層化アルゴリズム[1] の実装，
- GPU 上での振動補正やモザイキングの実装，

があげられる．また，シフト演算などを用いて 2 つの 16 ビット浮動小数点数で 32 ビット浮動小数点数精度を実現することでより CPU に近い精度を実現すること，さらには，動物体へ対応した未定義領域補間手法の GPU での実装も今後の課題である．

図 12 安定化結果．左の列が補正前のフレーム，中央の列が補正後のフレーム，右の列が補正後のフレームにモザイキングを行ったフレーム

Fig. 12 Result of the video stabilization. Left: before the correction, center: after the correction, right: with mosaicking.

参 考 文 献

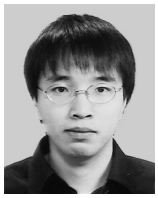
1) Bergen, J.R., Anandan, P., Hanna, K.J. and Hingorani, R.: Hierarchical Model-Based Motion Estimation, *ECCV'92 : Proc. 2nd European Conference on Computer Vision*, pp.237–252 (1992).

2) Bouguet, J.: Pyramidal Implementation of the Lucas Kanade Feature Tracker: Description of the Algorithm (2000). OpenCV Document, Intel, Microprocessor Research Labs.

3) Fernando, R. and Kilgard, M.J.: *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*, Addison-Wesley Pub. (2003).

4) Harris, M.: *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics*, Chapter 38: Fast Fluid Dynamics Simulation on the GPU, pp.637–665, Addison-Wesley Pub. (2004).
5) Jargstorff, F.: *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics*, Chapter 27: A Framework for Image Processing, pp.445–467, Addison-Wesley Pub. (2004).
6) Krüger, J. and Westermann, R.: Linear algebra operators for GPU implementation of numerical algorithms, *ACM Trans. Graphics* (*ACM SIGGRAPH 2003*), Vol.22, No.3, pp.908–916 (2003).
7) Litvin, A., Konrad, J. and Karl, W.C.: Probabilistic video stabilization using Kalman filtering and mosaicking, *IS&T/SPIE Symposium on Electronic Imaging, Image and Video Communications*, pp.663–674 (2003).
8) Matsushita, Y., Ofek, E., Ge, W., Tang, X. and Shum, H.-Y.: Full-Frame Video Stabilization with Motion Inpainting, *IEEE Trans. Pattern Analysis and Machine Intelligence*, Vol.28, No.7, pp.1150–1163 (2006).
9) Mitchell, J.L., Ansari, M.Y. and Hart, E.: *ShaderX2: Shader Programming Tips and Tricks with DirectX 9*, chapter Advanced Image Processing with DirectX9 Pixel Shaders, Wordware Publishing (2004).
10) Ready, J.M. and Taylor, C.N.: GPU Acceleration of Real-time Feature Based Algorithms, *IEEE Workshop on Motion and Video Computing* (*WMVC'07*), pp.8–9 (2007).
11) Sinha, S.N., Frahm, J.-M., Pollefeys, M. and Genc, Y.: GPU-Based Video Feature Tracking and Matching, Technical report, Technical Report 06-012, Department of Computer Science, UNC Chapel Hill (2006).
12) Strzodka, R. and Garbe, C.: Real-Time Motion Estimation and Visualization on Graphics Cards, *Proc. IEEE Visualization 2004*, pp.545–552 (2004).
13) Teukolsky, S.A., Vetterling, W.T. and Flannery, B.P.: *Numerical Recipes in C++: The Art of Scientific Computing*, Cambridge University Press (2002).
14) Tomasi, C. and Kanade, T.: Detection and Tracking of Point Features, Technical Report, Carnegie Mellon University Technical Report CMU-CS-91-132 (1991).
15) 金井　崇，安井悠介：GPU による細分割曲面の意匠形状評価，グラフィックスと CAD/Visual Computing 合同シンポジウム 2004 予稿集，pp.85–90 (2004).

(平成 19 年 5 月 24 日受付)
(平成 19 年 11 月 6 日採録)



藤澤　誠（正会員）

昭和 55 年生．平成 15 年静岡大学工学部機械工学科卒業．平成 17 年静岡大学大学院理工学研究科修士課程修了．静岡大学大学院理工学研究科博士課程在学中．日本学術振興会特別研究員 DC．物理シミュレーション等の研究に従事．ACM 会員．

天谷　貴大

昭和 59 年生．平成 19 年静岡大学工学部機械工学科卒業．静岡大学大学院工学研究科修士課程在学中．GPU を用いた画像処理等の研究に従事．

三浦憲二郎（正会員）

昭和 34 年生．昭和 57 年東京大学工学部精密機械工学科卒業．昭和 59 年東京大学大学院修士課程修了．同年キャノン（株）入社．機械系 CAD/CAM システムの開発に従事．平成 3 年コーネル大学大学院機械工学科博士課程修了（Ph.D.）．平成 5 年会津大学コンピュータ理工学部コンピュータソフトウェア学科助教授．平成 9 年静岡大学工学部機械工学科助教授．平成 16 年同教授．平成 18 年同大学創造科学技術大学院教授．形状処理工学，CAD/CAM，物理シミュレーション等に興味を持つ．ACM，IEEE 各会員．