2004年度 卒業論文

GPUを利用した文字認識システム用の一般化Hough変換

提出日：2005年2月2日

指導：筧 捷彦教授

早稲田大学理工学部情報学科

学籍番号：1G01P116-9

吉田 光寿

目 次

第1章　はじめに

1.1　分担について

本研究は宮永直樹氏と共同で行ったものである。吉田は主に一般化 Hough 変換関連資料の調査，改良アルゴリズム (algorithm) の検討，実装，デバグ (debug) 等を担当した。宮永氏は主に GPGPU 関連資料の調査，改良アルゴリズムの考案，実装，デバグ等を担当した。

1.2　背景と目的

本研究では画像認識アルゴリズムの一つである一般化 Hough 変換を汎用計算機上の文字認識に応用するため，これを GPU 上で動作させることで処理の高速化を図ることを目的とする。

30 年ほど前から光学文字読取り (OCR：optical character reader) についての研究が行われ，種々の成果が得られている。しかし，特に汎用計算機上での文字認識においては実用化に耐える認識率および認識速度を実現できていないのが現状である。OCR 用アルゴリズムのうち主に活字認識に利用されるパターンマッチング (pattern matching) 的手法は，認識率について次の二点のような問題点を抱えている。

- ノイズ (noise) や画像の劣化に弱い。
- 文字の変形に弱い。

これらの問題に対応すべく渦巻き照合法やパータベーション (perturbation) 等の手法が採用されている [1] が，完全な解決には至っていない。

解決策の一つとして、一般化 Hough 変換 [2] を利用することが考えられる。一般化 Hough 変換はパターンマッチング的手法に属する画像認識アルゴリズムの一種であるが，ノイズや画像の劣化，画像の回転に強いという特徴を持っている。この事から，一般化 Hough 変換を活字認識に利用すれば認識率の改善に繋がる可能性があると言える。しかし，残念ながら一般化 Hough 変換やその改良に当たる Chord-Tangent 変換 [3] は多くの処理時間を必要とし，実用的とは言えない。

一般化 Hough 変換の処理時間改善策として我々が目をつけたのが GPGPU[4] という研究である。これは等価な CPU と比較して何倍もの性能を持つ GPU に，CPU で行われるような一般的な計算をさせて処理の高速化を図るという試みである。先に述べた一般化 Hough 変換も GPU 上で動作するように実装することは可能である。

文字認識ではないが，実際に GPU 上で一般化 Hough 変換を動作させた例 [5] も存在する。

本研究では参考文献 [5] での実装の模倣に留まらず，フラグメントプロセッサ (Fragment Processor) を利用した別の手法と，頂点プロセッサ (processor) とラスタライズ (rasterize) 処理を利用した手法を提案し，これらの性能を比較・考察する。

1.3 論文の構成

本論文の構成を紹介する。2 章では一般化 Hough 変換アルゴリズムについての概要を述べる。3 章では GPGPU の概念と，GPU のアーキテクチャ(architecture)，GPU 上で動作するプログラム (program) の作成方法についての概要を述べる。4 章では今回提案するいくつかの実装方法の概要を述べる。5 章ではそれらの実装を評価する実験方法と実験結果・考察を述べる。最後に 6 章で本論文のまとめを述べる。

第2章　一般化Hough変換

この章では図形検出手法であるHough変換について述べた後，Hough変換の一般化である一般化Hough変換について述べる。

2.1　Hough変換

Hough変換とは，パラメタ(parameter)で表現できる図形(例えば，直線，円，楕円，放物線)を画像中から検出するための手法である。パラメタ数が多くなるほど実装は困難となり，一般的には直線の検出に対して利用されている。

2.1.1　DudaとHartの方法

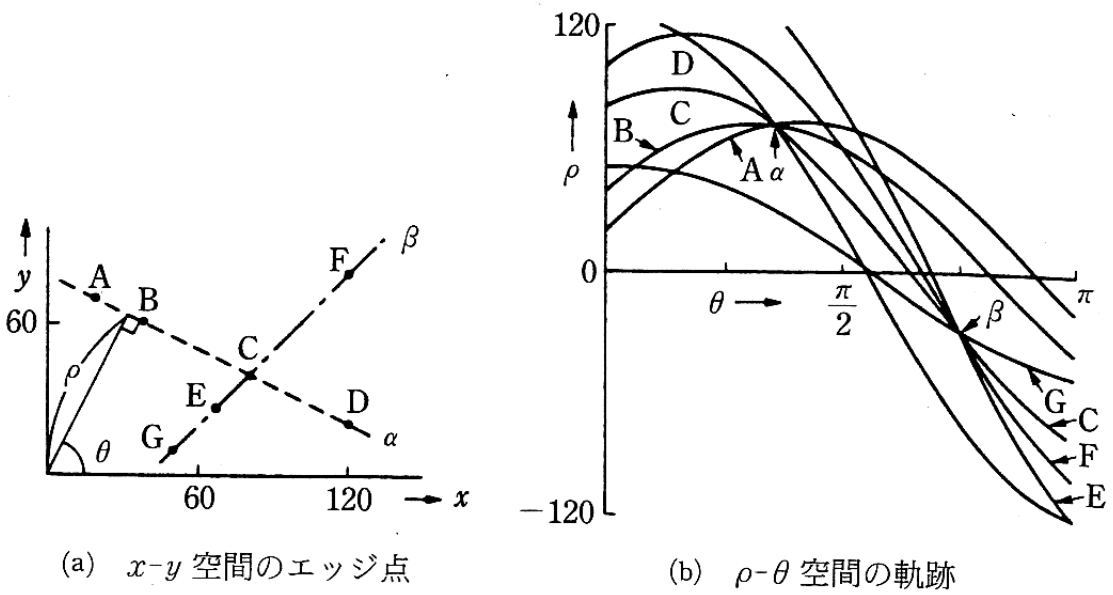
ここでは，最も広く使われているDudaとHartの方法[8]による直線検出法を紹介する。直線を式$\rho = x\cos\theta + y\sin\theta$で表現し，直線を記述するためのパラメタとして$(\rho, \theta)$を用いる。ここで，$\rho$は原点から直線へおろした垂線の長さ，$\theta$は垂線と$x$軸とのなす角である。この直線が画像上の点$(x_0, y_0)$を通るとすると，

$$\rho \quad = \quad x_0\cos\theta + y_0\sin\theta \tag{2.1}$$

の関係が成り立つ。この関数はパラメタ空間$\rho-\theta$上では正弦曲線となる。すなわち$x-y$空間の1点は$\rho-\theta$空間の1本の軌跡に対応し，逆に式(2.1)であらわされる$\rho-\theta$空間の軌跡は，$x-y$空間において点(x_0, y_0)を通るすべての直線群をあらわしていることになる。$x-y$空間上で1本の直線上の点を$\rho-\theta$空間に写した場合，これらの点からつくられる$\rho-\theta$空間上での軌跡は1点で交わることになる。

具体的な直線抽出の手法について述べる。まずエッジ(edge)あるいは線検出演算子によって，原画像から直線の要素の候補となる画素を抽出する。これらの画素の$x-y$座標を(x_i, y_i)とする。十分に細かく量子化された$\rho-\theta$空間を用意し，$\rho-\theta$空間内のすべてのθ_jについて$\rho_{ij} = x_i\cos\theta_j + y_i\sin\theta_j$を求め，$\rho-\theta$空間上の点$(\rho_{ij}, \theta_j)$に「投票」する。もし画像上に直線$\rho_0 = x\cos\theta_0 + y\sin\theta_0$が存在していれば，その直線上の画素はすべて$\rho-\theta$空間上の点$(\rho_0, \theta_0)$に対し投票を行うため，$(\rho_0, \theta_0)$にピーク(peak)が生じることになる。このピークを検出することによって直線を検出する。

図2.1はこの様子を図示したものである。$x-y$空間での点A，B，C，D，E，F，Gそれぞれに対応する$\rho-\theta$空間での軌跡を描く。すると，点A，B，C，Dに対応した軌跡の交わるピークαと，点C，E，F，Gに対応した軌跡の交わるピークβが$\rho-\theta$空間上にあらわれる。



(a) x-y 空間のエッジ点　　(b) ρ-θ 空間の軌跡

図 2.1: Hough 変換

2.2 一般化 Hough 変換

2.1 で述べた Hough 変換では，どのようにパラメタ表現すべきかが明確でない任意形状に対してはうまく適用できない。そこで提案されたのが一般化 Hough 変換である。

2.2.1 一般化 Hough 変換の考え方

ここでは三角形の検出を例として一般化 Hough 変換の基本的な考え方について述べる。まずテンプレート (template) を用意する。テンプレート上の 1 点に代表点を決め，それをテンプレートの位置とし，テンプレートの形はこの代表点を原点とする座標系で表す。

今，とある三角形 abc が存在するとする。また，abc とまったく同じ図形を表すテンプレート def が存在するとする。図 2.2 は三角形 abc とテンプレート def の関係を図示している。O_0 は代表点を表す。

この前提の下で，テンプレート def によって三角形 abc を検出する事を考える。図 2.3 はこの様子を図示したものである。まず，三角形 abc において頂点 b を観測点とする。テンプレート def の頂点 d が三角形 abc の頂点 b の位置に来るようにテンプレート def を三角形 abc に重ね合わせる。そのまま辺 df 上の各点が頂点 b を通るようにテンプレート def を平行移動させると，代表点の軌跡は O_1O_3 となる。同様に辺 de の場合は OO_1，辺 ef の場合は OO_3 となる。すなわち，三角形 abc の頂点 b がテンプレート def の一部であることを仮定すると，代表点の軌跡は OO_1O_3 となる。観測点を頂点 b だけでなく三角形 abc の辺上の任意の点とした場合，代表点の

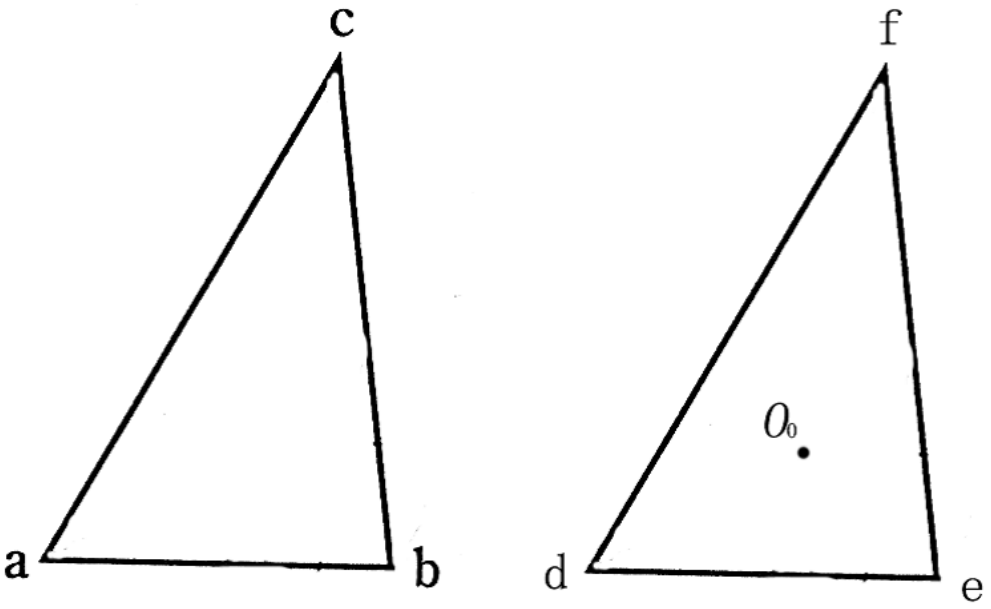


図 2.2: 三角形 abc とテンプレート def

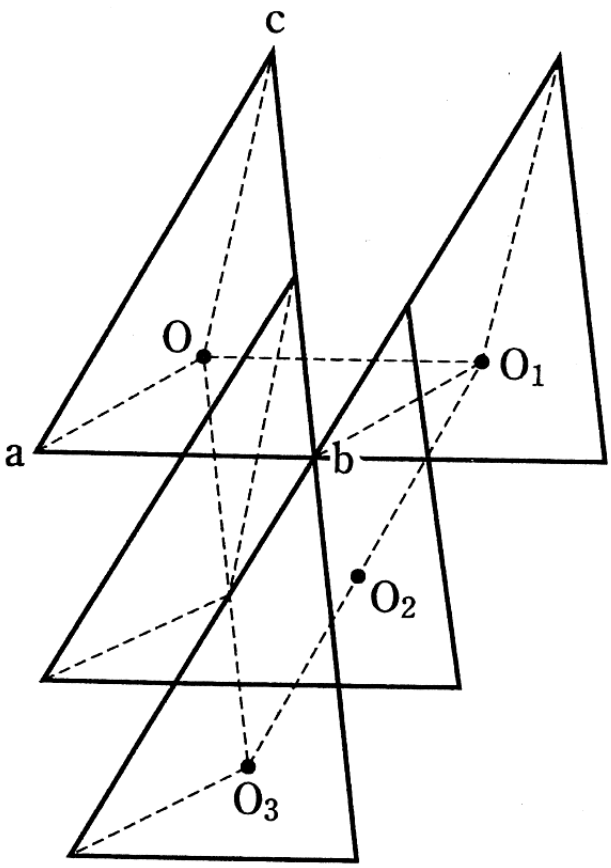


図 2.3: 一般化 Hough 変換による三角形の検出

軌跡が必ず O を通ることは容易に想像できる。三角形 abc の辺上のすべての点を観測点とし，それぞれにおける代表点の軌跡に対して投票を行えば，O がピークとなることは明らかである。このピークを検出することで三角形 abc はテンプレート def と同じ形であり，O の位置に存在する事が分かる。

これが一般化 Hough 変換の基本的な考え方であるが，この例では説明の簡単化のために座標変換のパラメタを (x, y) の 2 次元に制限している。実際には位置・回転角・拡大率の座標変換パラメタ空間上で投票と多数決を行う事で，三角形の大きさの変化や回転を許容する事も可能である。

2.2.2 アルゴリズムの概要

アルゴリズムの概要を箇条書きで示す。

1. テンプレート画像上に代表点 O を任意に決める。
2. テンプレートを構成しているエッジすべてについて，
   (a) 3 つの要素を計算する (図 2.4)。
      - ω : エッジにおける濃淡勾配の方向と水平軸の成す角の角度
      - r : エッジと代表点 O の距離
      - α : エッジと代表点 O を結ぶ線分と水平軸の成す角の角度

   (b) 各 ω を見出しとして，r，α を登録した R-Table を作成しておく (表 2.1)。
3. 4 次元パラメタ空間 vote$[U][V][\Theta\,][S]$ を用意する。ここで (U, V) は位置を，Θ は回転角を，S は拡大率を表す。
4. 検出対象画像を構成しているエッジすべてについて，
   (a) エッジ（座標 (X_i, Y_i)) における濃淡勾配の方向 ω_i を求める。
   (b) パラメタ空間中のすべての θ と s の組み合わせに対して，
      i. $\omega_i - \theta$ に対応する R-Table 中のエントリ (entry) $(r(\omega_i - \theta), \alpha(\omega_i - \theta))$ の組をすべて求める。
      ii. 4(b)i で求めたすべてのエントリについて，
         A. 次の 2 式で (u, v) を求める。

$$u = X_i + r(\omega_i - \theta)s\cos(\alpha(\omega_i - \theta) + \theta) \tag{2.2}$$
$$v = Y_i + r(\omega_i - \theta)s\sin(\alpha(\omega_i - \theta) + \theta) \tag{2.3}$$

         B. vote$[u][v][\theta][s]$ をインクリメント (increment) する。
5. 最も多く投票された (u, v, θ, s) の組を探す。

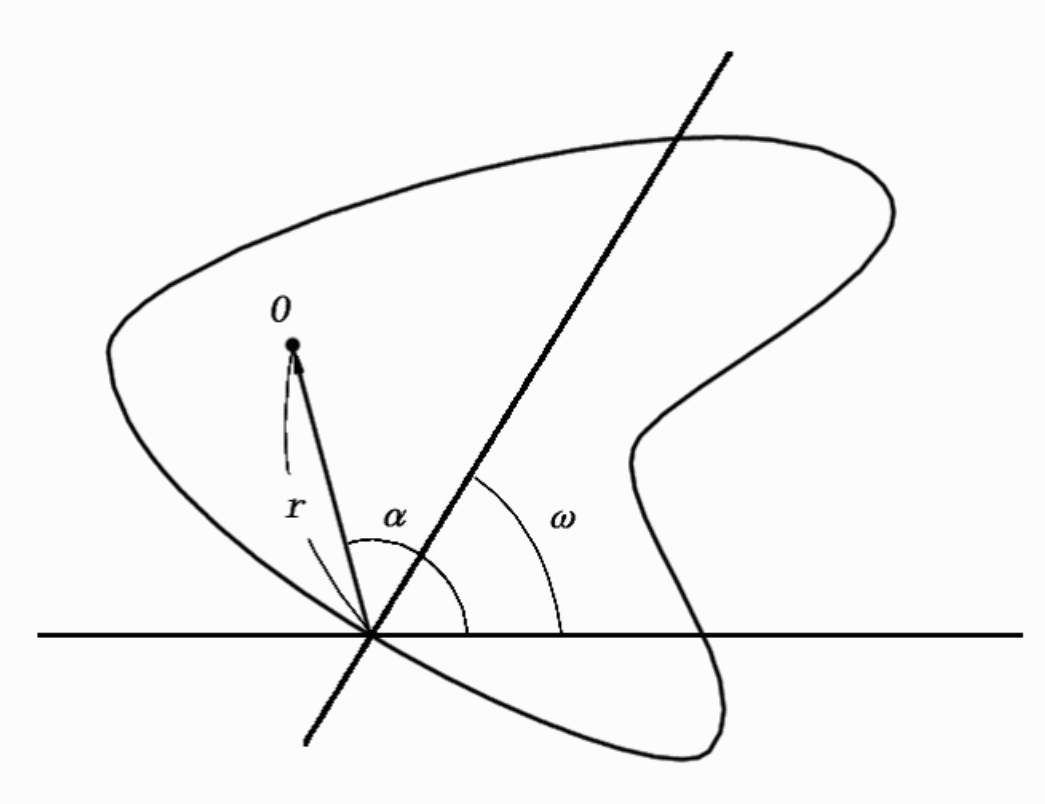


図 2.4: R-Table に格納する情報

表 2.1: R-Table

ω	(r, α)
ω_0	$(r_{00}, \alpha_{00}), (r_0, \alpha_{01})$
ω_1	$(r_{10}, \alpha_{10}), (r_{11}, \alpha_{11}), (r_{12}, \alpha_{12}), (r_{13}, \alpha_{13})$
ω_2	(r_{20}, α_{20})
$\cdots$	$\cdots$

代表点 O の選び方には任意性があるが，検出対象図形の重心を代表点とした場合に検出の精度がもっとも良くなる事が明らかにされている [9]。

エッジの検出や濃淡勾配の方向は画像に空間微分フィルタ (filter) をかける等の手法によって実現する。濃淡勾配の方向を正確に求める事は困難であり，これが検出精度低下の原因にも繋がるが，投票先を分散させるなどの改善策が提案されている [7]。

2.2.3 一般化 Hough 変換の問題点

一般化 Hough 変換には，次のような問題点がある。

- 考慮する座標変換パラメタの次元が増加するごとに処理時間が爆発的に増加する。
- 考慮する座標変換パラメタの次元が増加するごとに使用メモリ (memory) 領域が爆発的に増加する。

これらの問題点を改善するために Chord-Tangent 変換 [3] や FGHT(Fast Generalized Hough Transform)[6][7] などの手法が提案されているが，検出精度や処理時間について未だ多くの課題が残されていると言える。

第3章　GPGPU

この章では本研究の内容を理解するために必要な，GPGPU に関連する基礎知識を述べる。

3.1　GPU

GPU(Graphics Processing Unit) は，3D グラフィックス (graphics) の表示に必要な計算処理を行なう半導体チップ (chip) である。従来 3D グラフィックスアクセラレータ (graphics accelerator) と呼ばれていたチップの発展形で，3D グラフィックスアクセラレータと比べて担当する処理が多くなっている。3D グラフィックスアクセラレータはテクスチャ(texture) の張り込みなど最終的なレンダリング (rendering) 処理のみを担当していたが，GPU はレンダリングの前処理にあたる 3D 座標から 2D 座標への座標変換なども担当する。

3.2　GPGPU の概念

GPGPU(General-Purpose computation on GPUs[4]) とは，GPU 上で汎用数値計算処理を行うという試みの総称である。

アプリケーション (application) からの需要に支えられ，GPU の性能は著しい向上を続けている。そしてその進化は明らかに CPU のそれを超えるものである。多少古い情報 (2002 年 7 月) であるが，図 3.1，図 3.2，図 3.3 はそれぞれ CPU の MHz と GPU の Mpixels の推移，CPU の MHz と GPU の Mpixels の成長率の推移，CPU と GPU のトランジスタ (transistor) 数の推移を線グラフ (graph) 化したものである [10]。 表 3.1 は GPU のチップレベル (chip level) での仕様を Prescott コア (core) の Pentium4 と比較したものである [11][12][13][14]。一昨年リリース (release) された GeForceFX5900Ultra は，Pentium4 を上回るトランジスタ数を持つ。また，昨年 5 月に発表され現在市場において最高級の性能を誇る GeForce6800Ultra では，トランジスタ数が Pentium4 の 2 倍弱になるとともに理論最大性能も GeForceFX5900Ultra の約 2 倍弱に達している [11][12]。

また，近年の GPU は，浮動小数点演算を正式に内蔵するとともに，従来固定機能であったグラフィックス演算処理フロー (flow) のプログラム制御が可能になっている [15][16][17]。このような GPU の高性能化と汎用化，さらには Cg や HLSL に代表される GPU 向けプログラミング (programming) 環境の整備は，GPU 上で汎用数値計算処理を行うという新たな研究分野を確立させた。

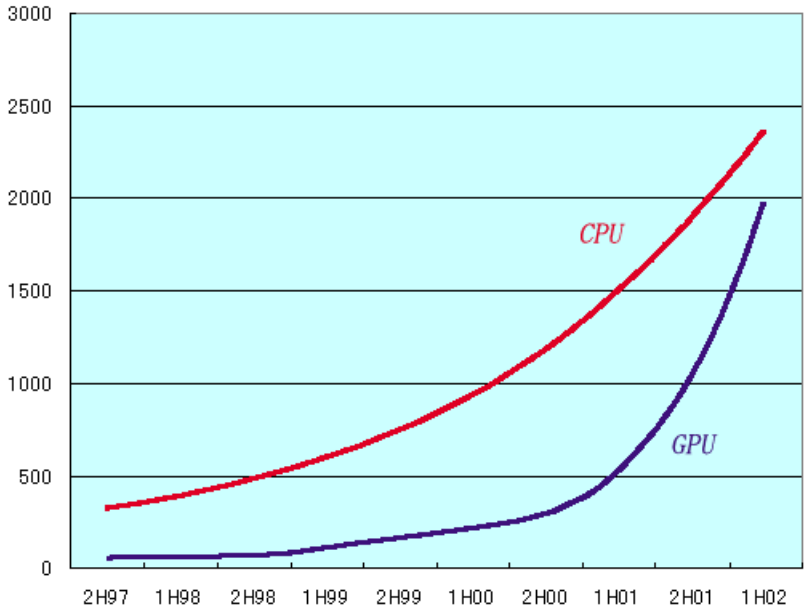


図 3.1: CPU の MHz と GPU の Mpixels の推移 [10]

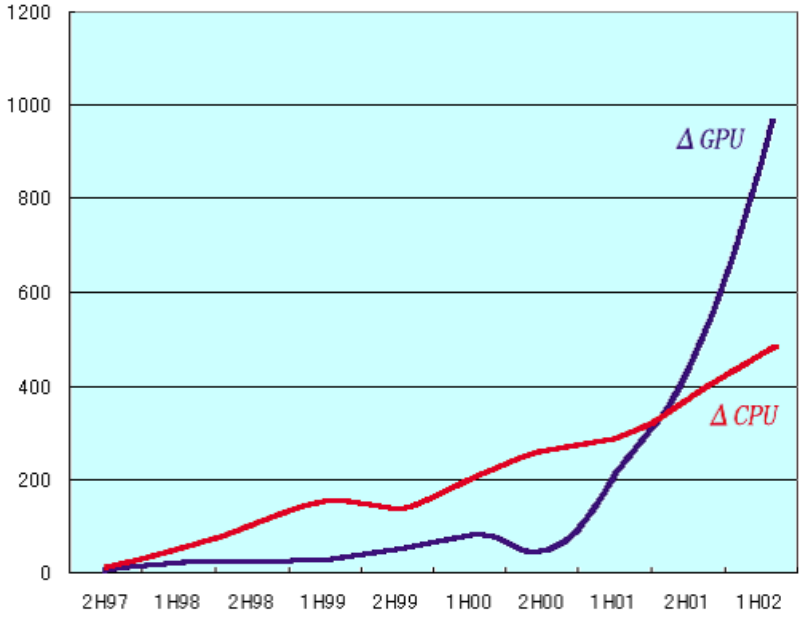


図 3.2: CPU の MHz と GPU の Mpixels の成長率の推移 [10]

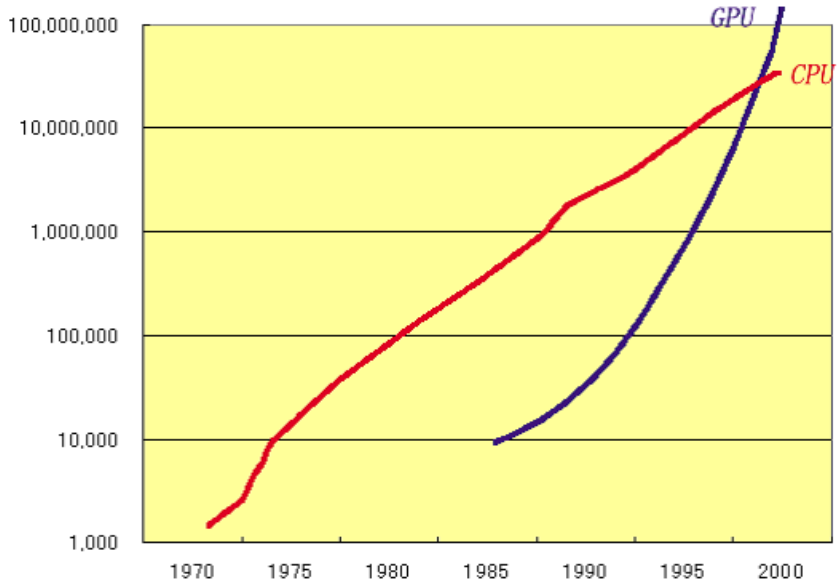


図 3.3: CPU と GPU のトランジスタ数の推移 [10]

表 3.1: チップ比較 [11][12][13][14]

	Pentium4	GeForceFX5900Ultra	GeForce6800Ultra
Core Name	Prescott	NV35	NV40
Core Clock	3.6GHz	400MHz	400MHz
Process	90nm low-k	0.13μm 銅	0.13μm 銅
Transistor	125M	135M	222M
Cache Size	1MB	非公開	非公開 (+外部)
Die	112mm^2	???mm^2	300mm^2

3.3 GPU におけるレンダリング処理の流れ

レンダリング処理の流れの概略について述べる。図 3.4 はレンダリング処理の流れを図示したものである。

1. 視点等の初期パラメタを設定した後，CPU から描画すべき物体 (三角形ポリゴン (polygon) など) のデータ (data) を GPU に送る。

2. 頂点座標の幾何変換を行ない，物体が 2 次元スクリーン (screen) 空間に投影された時の位置および形状を求める。

3. 投影物体をラスタライズ処理により，スクリーンの個々のスキャンライン (scanline) に対応するフラグメント (flagment。投影物体をスキャンラインで切った時の断片) を計算する。簡単に言えば，ピクセルデータ (pixel data) を生成する。

4. 個々のフラグメントに対して，テクスチャの張り付けやフィルタ処理，シェーディング (shading) 処理を行なう。

5. 最後に，既に描画済のデータと合成し，フレームバッファ(frame buffer) に格納する。

この一連の処理の流れをグラフィックス演算処理フロー，3D グラフィックスパイプライン (graphics pipeline) などと呼ぶ。

3.4 基本アーキテクチャ

3.3 で述べたグラフィックス演算処理フローのうち，頂点データに対する処理は頂点プロセッサ又は頂点シェーダユニット (vertex shader unit) と呼ばれる演算器によって行なわれる。また，ピクセルデータに対する処理はフラグメントプロセッサ又はピクセルシェーダユニット (pixel shader unit) と呼ばれる演算器によって行われる。それぞれのプロセッサが処理の対象とするデータはまったく異なり，処理の形態も異なるため，両プロセッサは各々に特徴的な性質を持ったハードウェア (hardware) 構成となる。例えば，頂点プロセッサはは 4×4 の座標変換行列と 4 次元座標ベクトル (vector) との積の計算を高速に行なう専用ハードウェアをもつとともに分岐処理をサポートするのに対して，フラグメントプロセッサでは分岐処理はサポートせず，2 枚の画像の色情報をピクセル単位で合成するための比較的単純なハードウェアを多数備える傾向にある。

また，両プロセッサとも複数の演算パイプライン (pipeline) をもっており，入力データ列に対して SIMD (Single Instruction/Multiple Data) 的な動作を行なう。SISD (Single Instruction/Single Data) である場合に比べて，単純にパイプライン数倍の性能が期待できる。

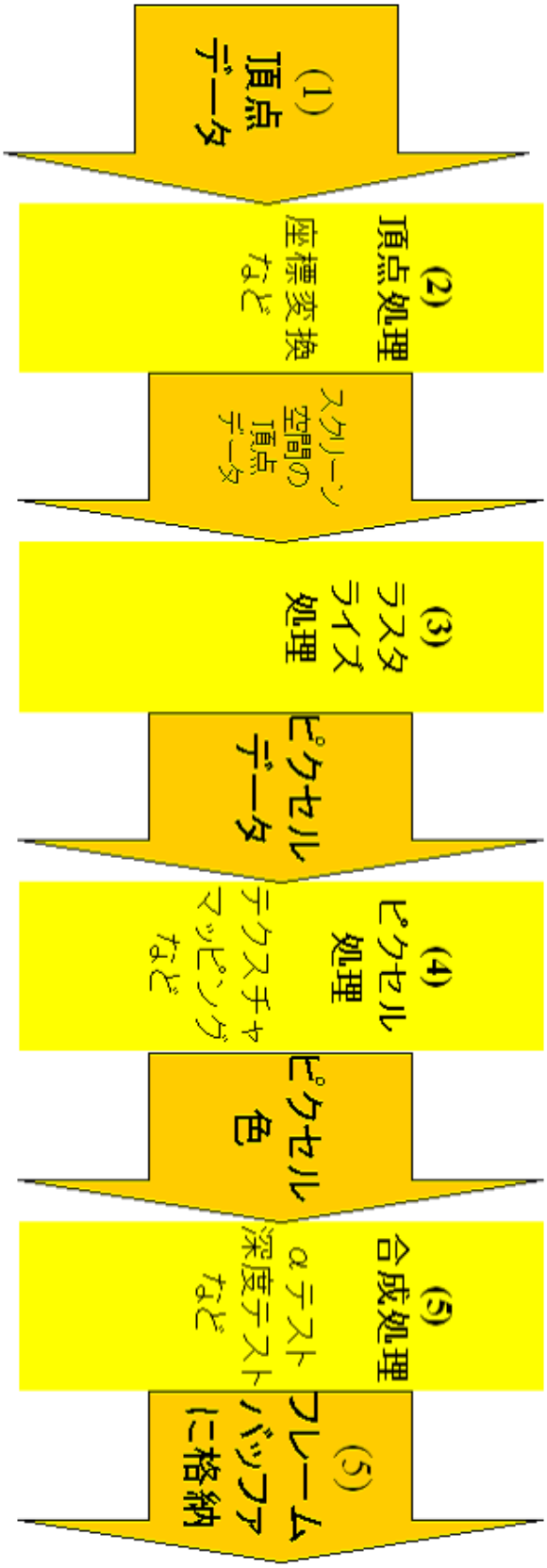


図 3.4: レンダリング処理の流れ

3.4.1 近年の GPU

近年の GPU は，高性能化とともに，汎用化の傾向にある。旧来の GPU では，3.3, 3.4 で述べたようなレンダリングパイプライン (rendering pipeline) の構成は固定であり，グラフィックス演算処理フローも固定的であった。これに対して，近年，複雑なレンダリング処理を実現するという要求に答えるため，頂点プロセッサとフラグメントプロセッサにおけるレンダリング処理の流れを動的に制御するプログラマブルパイプライン (programmable pipeline) の概念が導入された。また，より精度の高い計算を実現するため，扱えるデータ形式として 16bit あるいは 32bit の浮動小数点形式が追加された。更に，フラグメントプロセッサでも分岐処理をサポートし，SIMD 演算ユニット (unit) を積むようになった。

近年の GPU の代表例として，GeForce6800Ultra のパイプライン構成を図 3.5 に，頂点プロセッサの構成を図 3.6 に，フラグメントプロセッサの構成を 3.7 に，それぞれ掲載する [18]。 GeForce6800 Ultra は，頂点プロセッサ 6 基，フラグメントプロセッサ 16 基という構成になっている。

頂点プロセッサは，頂点テクスチャフェッチ (texture fetch) ユニットが入って従来不可能であった頂点プロセッサからのテクスチャアクセス (texture access) が可能になった事と，ベクトルユニット (vector unit) とスカラユニット (scalar unit) が並列化されている事という二つの特徴がある。また，ベクトル演算ユニットは，従来の SIMD ではなく MIMD(Multiple Instruction/Multiple Data) となっており、より柔軟性が高くなっている。

フラグメントプロセッサは，16 基のプロセッサによる並列動作が最大の特徴である。また，テクスチャ関連の処理とテクスチャ処理をしていない時のみ普通の 32bit 浮動小数点演算を行うことができるユニーク (unique) な 4waySIMD 演算器一つと，普通の 4waySIMD 演算器を備えている。勿論，分岐処理もサポートしている。

3.5 基本機能

GPU には，グラフィックス処理に特化した専用のハードウェアが多く組み込まれている。これらのハードウェアを有効活用することで，GPU を利用したプログラムはその効率を大幅に上げることが可能である。ここでは，汎用計算に応用可能な代表的な機能について紹介する。

1. 内積計算

   2 つの 4 次元ベクトル間の内積計算は，グラフィックス処理で非常に多用される機能である。例えば，物体からの反射光の強度計算や距離を求める場合の自乗和計算で使用する。

2. 座標変換

   1. を 4 つ組合せた計算で，行列 (4x4) とベクトル (要素数 4) の積である。頂点プロセッサに入力される座標データは基本的に全てが座標変換の対象となる。

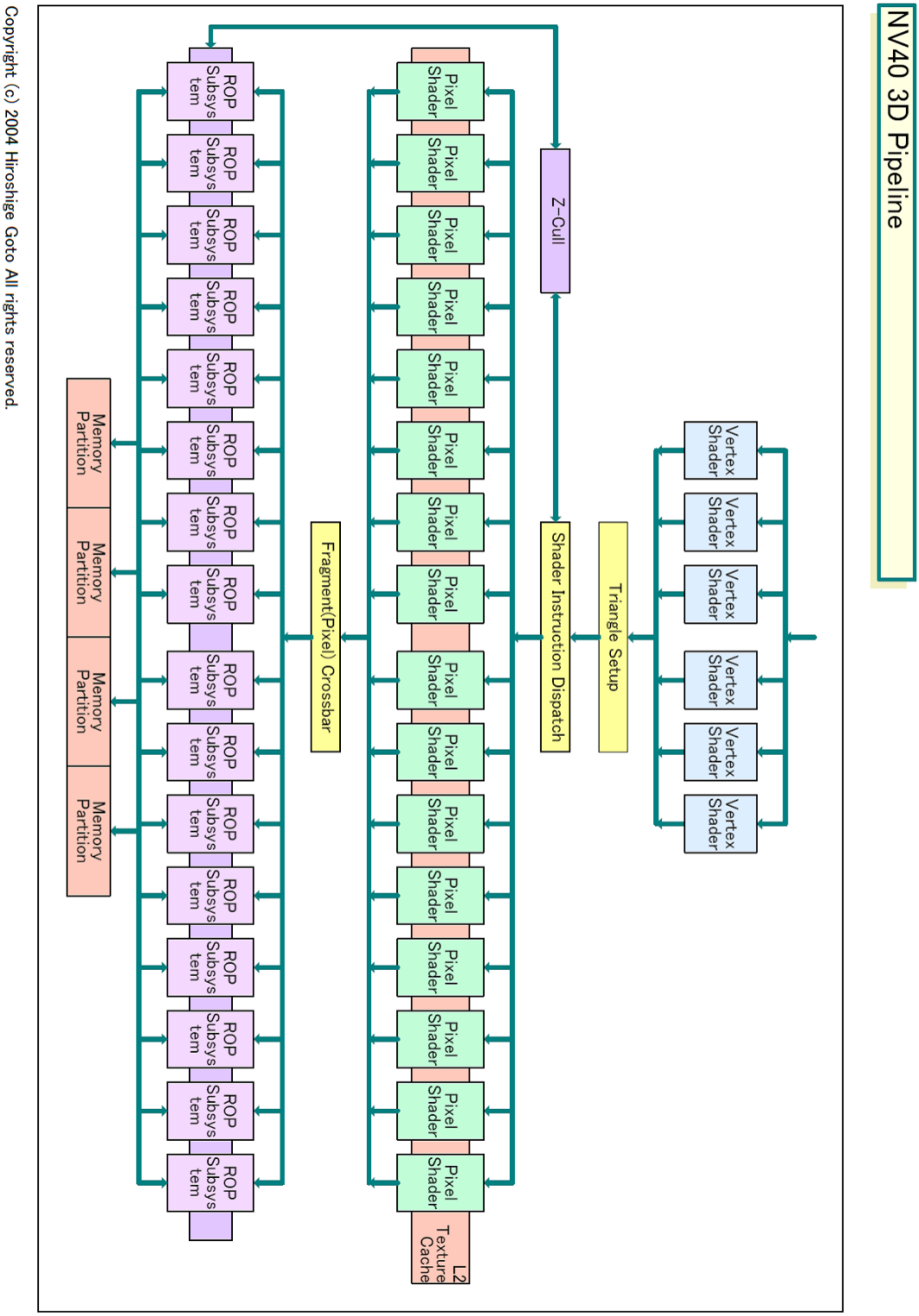


図 3.5: GeForce6800Ultra のパイプライン構成 [18]

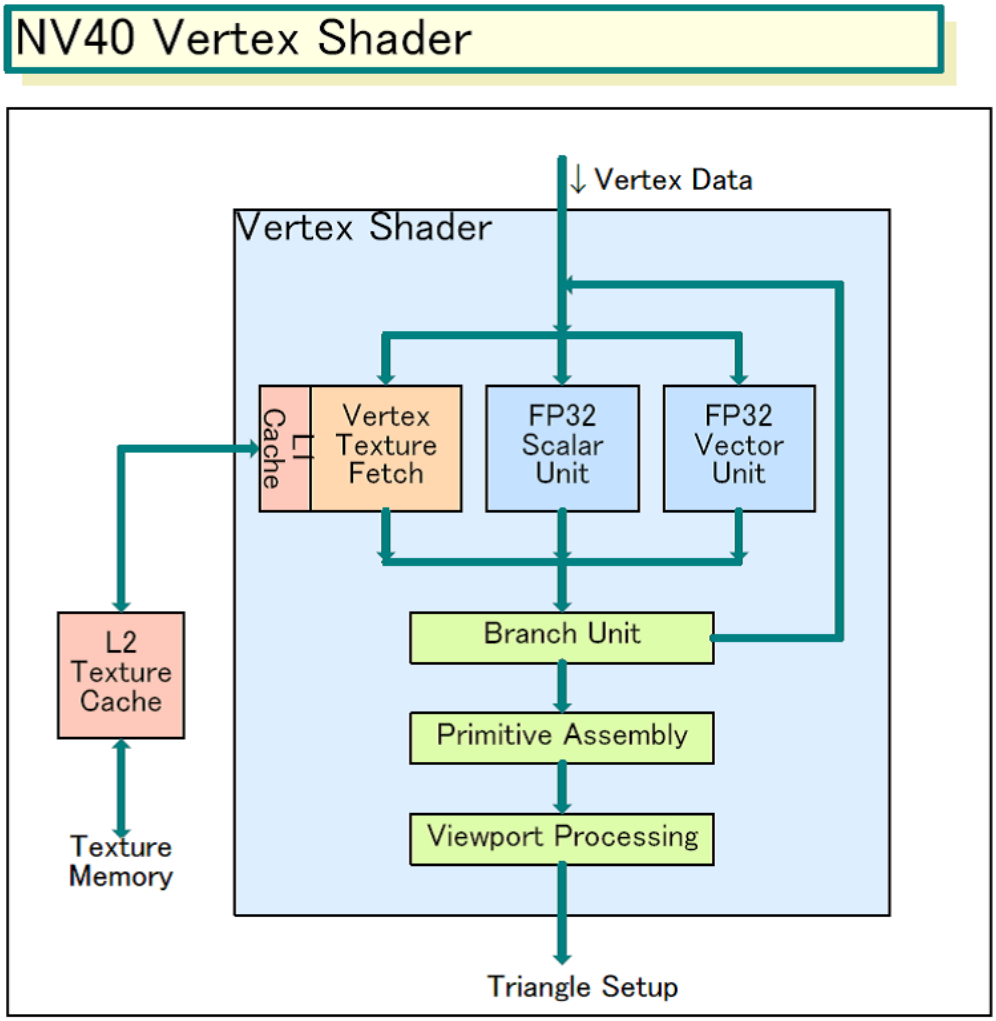


図 3.6: GeForce6800Ultra の頂点プロセッサ [18]

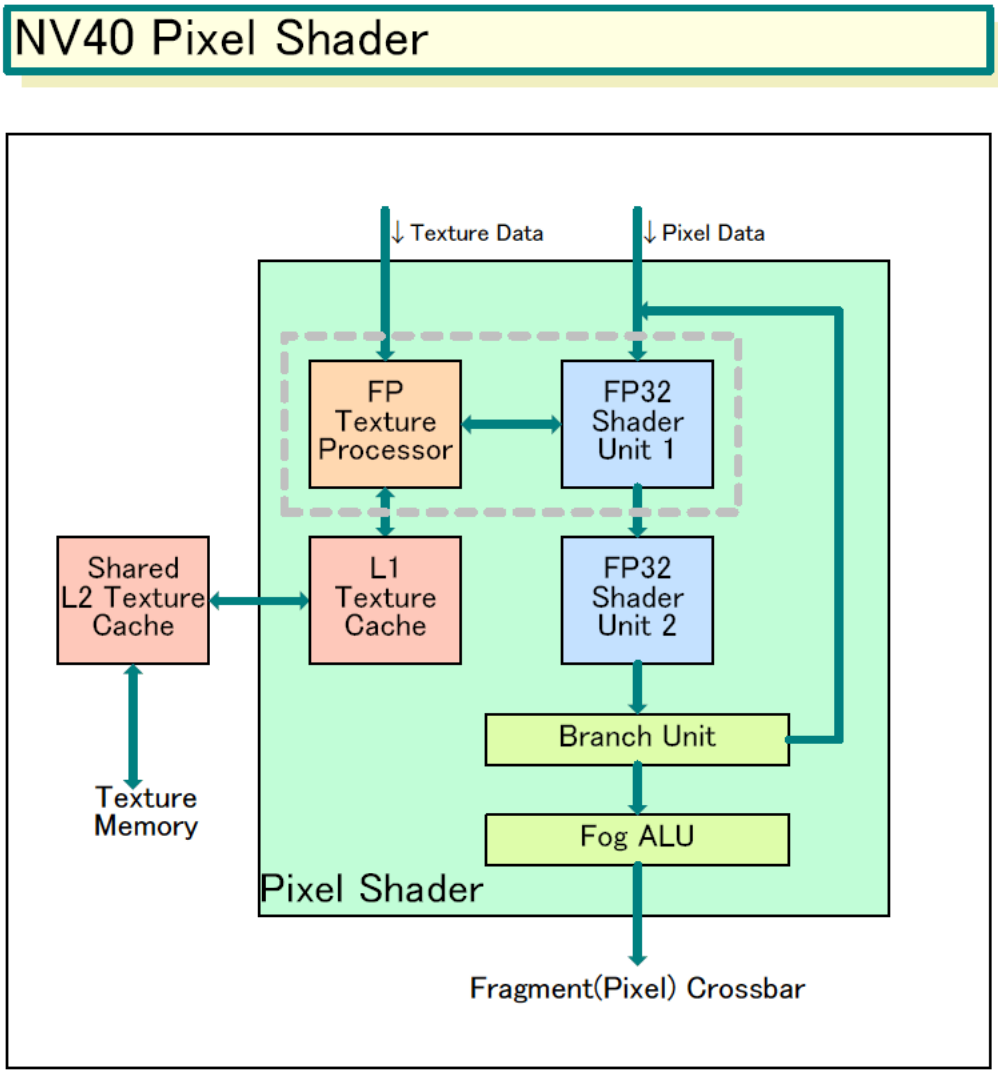


図 3.7: GeForce6800Ultra のフラグメントプロセッサ [18]

3. 2 乗根の逆数

   ベクトルの正規化等で必要な 2 乗根の逆数を計算する回路が組み込まれている。1. と組合せることで距離や距離の逆数を容易に計算することが可能である。

4. テクスチャマッピング (texture mapping)

   ビデオメモリ (videomemory) 内に格納されたテクスチャを，幾何変換後の物体に張り付けて模様を表現する機能である。テクスチャとして 2 次元配列を与え，同じ大きさの長方形ポリゴンにマッピング (mapping) すれば，2 つ以上の 2 次元配列に対する要素毎の演算が可能となる。

5. マスク (mask) 付き演算

   フラグメント処理においては，個々のピクセルの Z, Stencil, α を比較することで，当該ピクセルに対する処理を無効化することが可能である。フラグメント処理パイプラインにおける分岐がサポートされていない頃は，予めマスクパターン (mask pattern) が分かる場合にはこの機能を使うことで処理を行うか否かを分岐していた。現在はフラグメント処理における分岐がサポートされているが，その動作は遅いため，こちらの機能が利用される事が多い。

3.6 実装上の制約

これまでに述べてきたように，汎用計算処理に GPU を用いる事によって大きな恩恵を得る事が可能である。しかし，GPU であるが故の実装上の制約もまた存在する。ここでは，その中でも特に重要なものについて述べる。

1. ReadBack 性能

   CPU から GPU へのデータ転送は，転送路のバス (bus) 規格が AGP8X の場合 2GB/s に近い転送速度が得られる。しかし，その逆方向の GPU から CPU へのデータ転送では数 100MB/s 程度の転送速度しか得られない。また，アーキテクチャ実装の都合により，GPU のパイプラインを一旦フラッシュ(flush) した後でなければ GPU から CPU へのデータ転送が行なえないという制約がある。GPU から CPU へのデータ転送が頻繁に行なわれると，AGP バスのオーバヘッド (overhead) と GPU のパイプラインフラッシュ(pipeline flush) の為，GPU の性能を十分に発揮することができない。

   NV40 では GPU から CPU へのデータ転送速度が 600MB〜1GB/s 程度にまで改善されている。

2. コードサイズ (code size) の制限

   頂点プロセッサおよびフラグメントプロセッサのプログラムは実行前に GPU のビデオメモリに格納しておく必要がある。この際，各々のプロセッサに対して個々にプログラムサイズ (program size) の上限が定められている。NV35 の場合，頂点処理プログラムの命令数の上限は 256，フラグメント処理プログラ

ムに関しては1024である。プログラムのコードサイズに上限が存在するため，大きなプログラムを実行するためには複数回のレンダリング処理に分割して実行する必要がある。1回のレンダリングが終了する毎にCPUにパラメタ渡し等が必要になると1. で述べた制約の影響も受けてしまう。

NV40ではこの問題は大幅に改善され，頂点処理プログラムの命令数の上限は512，フラグメント処理プログラムに関しては16384となった。

3. ビデオメモリへのアクセス (access)

   頂点プロセッサおよびフラグメントプロセッサは，テクスチャアクセスの時のみビデオメモリへアクセスすることが可能である。その他の，値の読み書き等の操作をする事は不可能である。一時記憶や値の読み書きはGPUのレジスタ (register) を利用して行われる。C言語におけるmalloc()のように動的に記憶領域を確保するような操作は不可能であるが，これは汎用プログラムの実装において制約となりうる。対応策としては，処理を複数回のレンダリング処理に分割して，動的確保が必要な部分はCPU側で実装し，その大きさのテクスチャを生成する事が考えられる。しかし，やはり1回のレンダリングが終了する毎にCPUにパラメタ渡し等が必要になると1. で述べた制約の影響を受けてしまう。

4. ビデオメモリのアーキテクチャ

   ビデオメモリは，共有メモリアーキテクチャ(shared memory architecture) を採用しており，ディスプレイ (display) 出力用のフレームバッファ領域や，テクスチャデータ (texture data) 保存用のテクスチャ領域など，さまざまな用途で共用される。このアーキテクチャがGPU上の汎用計算における演算性能の足枷になっているという報告がある [19]。

GPU上での汎用計算処理に性能を求める場合，これらの制約について理解し，できるだけ制約の影響を受けないようなプログラムを記述することが大切である。

3.7 GPUプログラミングの基礎概念

ここでは頂点処理プログラムとフラグメント処理プログラムそれぞれが担当する処理について述べる。また，それぞれのプログラムが扱う入力と出力について述べる。

3.7.1 頂点処理プログラム

頂点処理プログラムには，一つの3次元頂点データを受け取り，その頂点を2次元スクリーン空間に投影した後の座標や色を返すような処理を記述する。ところが，CPUがGPUに与えるのは一つの頂点データではなく頂点データの列である。実際に頂点プロセッサが座標変換処理を行う時は，受け取った頂点データ列が保持する頂点データ一つ一つに対して，頂点処理プログラムに記述された処理を適用する。

また，頂点プロセッサはGPU上に複数用意されており，それらが並列に動作することでSPMD(Single Program/Multiple Data)的に頂点データを処理していく。

頂点処理プログラムの入力は一つの3次元頂点を表す色や座標，法線ベクトルなどのデータである。CPUからは頂点データの列を与える。これとは別に，頂点処理に利用する定数をCPUから与える事もできる。これを均一入力という。最新のGPU(NV40以降)では，頂点プロセッサはビデオメモリに格納されたテクスチャを参照することが可能になった。あらかじめCPUでテクスチャを作成しておけば，これもCPUからの入力として利用することができる。

頂点処理プログラムの出力は一つの2次元頂点を表すデータである。このデータはグラフィックス演算処理フローの流れに沿って2次元スクリーンにラスタライズされる。

3.7.2 フラグメント処理プログラム

フラグメント処理プログラムには，一つのピクセルデータを受け取り，そのピクセルの色をどのように変更するかを記述する。ところがフラグメント処理直前のラスタライズ処理で生成され，フラグメントプロセッサに与えられるピクセルデータは2次元配列である。実際にフラグメントプロセッサがフラグメント処理を行う時は，受け取ったピクセルデータの2次元配列が保持するピクセルデータ一つ一つに対して，フラグメント処理プログラムに記述された処理を適用する。また，頂点プロセッサと同様フラグメントプロセッサはGPU上に複数用意されており，それらが並列に動作することでSPMD的にピクセルデータを処理していく。

フラグメント処理プログラムの入力は一つのピクセルを表す座標や色のデータである。これとは別に，フラグメント処理に利用する定数をCPUから与える事もできる。これを均一入力という。また，フラグメントプロセッサはビデオメモリに格納されたテクスチャを参照することができる。あらかじめCPUでテクスチャを作成しておけば，これもCPUからの入力として利用することが可能である。

フラグメント処理プログラムの出力はピクセルの色と深度を表すデータである。このデータは深度テスト(test)やαテスト，ステンシルテスト(stencil test)といった処理を済ませて描画済みの画像データと合成された後，通常フレームバッファに格納される。ただし，レンダリングターゲットテクスチャ(rendering target texture)を作成する事で，このデータの格納先をフレームバッファからテクスチャへと変更する事が可能である。

3.8 GPU上での数値計算

ここではGPU上で数値計算をする際に利用される基本的な手法について述べる。

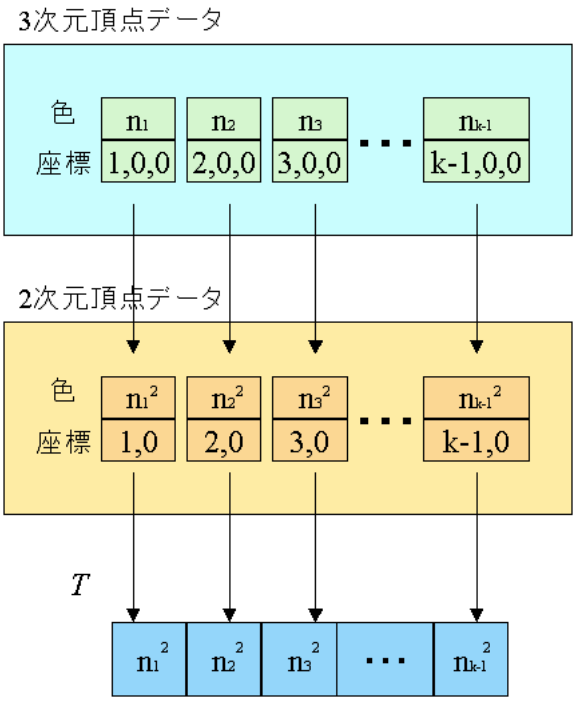


図 3.8: 頂点プロセッサによる汎用数値計算の例

3.8.1 頂点プロセッサを利用

汎用数値計算を目的とする場合，頂点処理プログラムの入力となる頂点データに計算対象のデータを詰め込み，これを処理させる。

計算結果は出力する 2 次元頂点の色データに格納する。出力した頂点は通常のグラフィックス演算処理フローの流れに沿ってラスタライズされてしまうため，頂点をラスタライズした際 1 つのピクセルに 1 つの結果を格納できるように頂点の座標を設定しておく。レンダリングターゲットテクスチャを用意しておくことで，本来ならばグラフィックス演算処理フローの最後でフレームバッファに格納されてしまう計算結果をテクスチャに格納し，CPU に送る。

図 3.8 は頂点プロセッサを利用した汎用数値計算の簡単な例を示している。ここでは与えられる数値 $n_i (i = 0, 1, 2, \cdots, k-1)$ に対して n_i^2 を返す計算を行う。入力される i 番目の頂点の色データに数値 n_i を，座標データに出力先ピクセルの座標 $(i, 0)$ を格納する。i 番目の頂点データを処理する頂点プロセッサは n_i^2 を計算し，これを出力する頂点の色データに格納する。出力する頂点の座標は，あらかじめ指定された出力先ピクセルの座標を格納する。頂点がラスタライズされ，これを $1 \times k$ の大きさを持つテクスチャ T にレンダリングする。その結果 $T(i, 0)$ に n_i^2 が格納され，CPU にこれを送ることで計算が完了する。

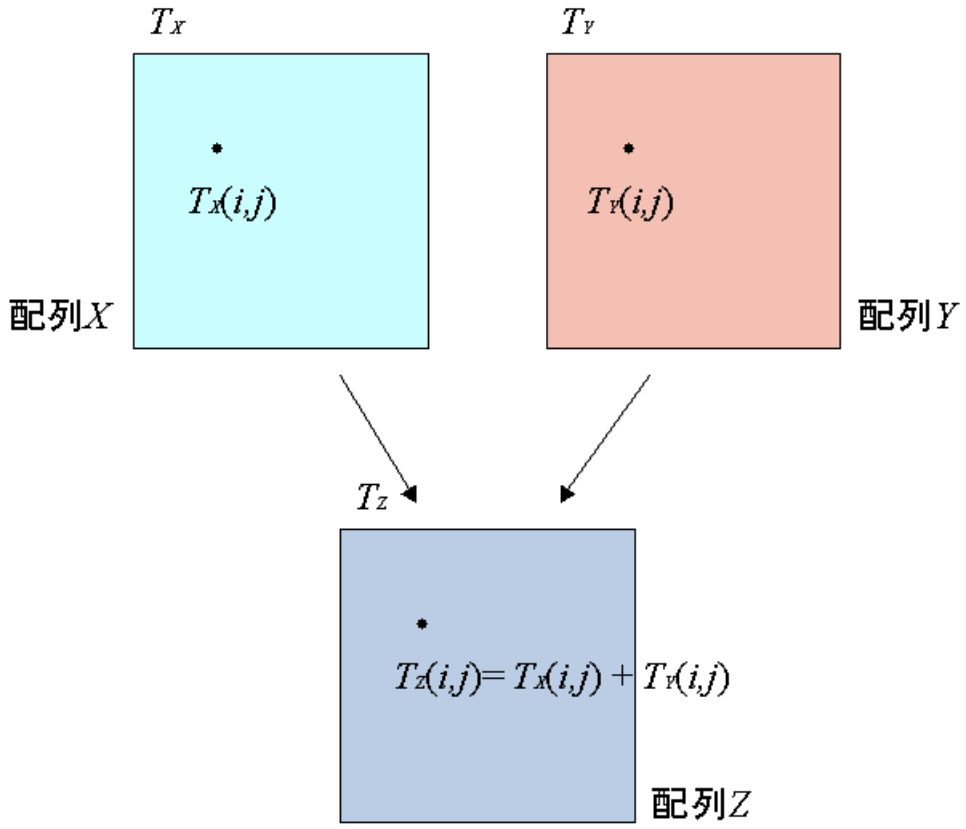


図 3.9: フラグメントプロセッサによる汎用数値計算の例

3.8.2 フラグメントプロセッサを利用する方法

フラグメントプロセッサが浮動小数点演算をサポートする以前は，頂点プロセッサを利用した汎用計算が試みられていた [19] が，フラグメントプロセッサが浮動小数点演算をサポートすると GPGPU の対象はテクスチャマッピングベース (texture mapping base) の汎用計算に移ってきた。近年の GPU はフラグメントプロセッサ数が頂点プロセッサ数より多くなる傾向にあり，GPU での数値演算はフラグメントプロセッサを使ったものが主流となりつつある。

汎用数値計算を目的とする場合，入力としては主にテクスチャを利用する。フラグメントプロセッサは複数のテクスチャを扱う事が可能である。これは複数の二次元配列を入力として扱う事と等価である。

計算結果はフラグメント処理プログラムの出力であるピクセルデータに格納する。頂点プロセッサを利用する方法と同様レンダリングターゲットテクスチャを用意しておき，本来ならばグラフィックス演算処理フローの最後でフレームバッファに格納されてしまう計算結果をテクスチャに格納し，CPU に送る。

図 3.9 はフラグメントプロセッサを利用した汎用数値計算の簡単な例を示している。$N \times N$ の 2 次元配列 X と Y に対して各要素毎の演算を行なって $Z = X + Y$ を計算する例で，フラグメントプロセッサのパイプライン性能を最も発揮できる演算パターン (pattern) である。入力に用いる 2 つのテクスチャを T_X, T_Y とする。$T_X(i,j)$

の色データに要素 X_{ij} を格納し，$T_Y(i,j)$ の色データに要素 Yij を格納する (ただし $i = 0, 1, 2, \cdots, N$，$j = 0, 1, 2, \cdots, N$)。座標 (i,j) に対応するピクセルを処理するフラグメントプロセッサは，まず $T_X(i,j)$ と $T_Y(i,j)$ にアクセスして値を取得した後，$Z_{ij} = X_{ij} + Y_{ij}$ を計算し，これを出力するピクセルの色データに格納する。$N \times N$ の大きさを持つテクスチャT_Z をレンダリングターゲットテクスチャとして用意し，結果をこれにレンダリングする。その結果 $T_Z(i,j)$ に Z_{ij} が格納され，CPU にこれを送ることで計算が完了する。

3.9 基本的な数値演算における GPU の性能

Thompson らは頂点プロセッサを用いた数値計算法を紹介している [19]。スカラ (scalar) 値配列を 4 次元座標ベクトルにパッキング (packing) して並列度を 4 倍にするとともに，複数の頂点プロセッサを SPMD 並列動作させることで，要素数が非常に大きく演算強度が高い演算に対しては CPU の 10 倍近い演算性能を出している。

後述するプログラミング言語 Brook を用いて GPU 性能を測定したデモ (demo) では，Ge Force5900Ultra を用いた浮動小数点乗算 ($A[i] = A[i] \times A[i]$) で 20GFLOPS の演算性能を記録している。

3.10 プログラミング環境

GPU プログラミングはアセンブラ (assembler) または専用の高級言語によって行う。高級言語には NVIDIA の Cg(C for graphics)[20][21] や Microsoft の HLSL(High level shading language)[22] などがある。Stanford 大学の Brook[23] や Waterloo 大学の Sh[24] 等の言語も提案・開発されている。

Cg は，高いプログラマビリティ (programability) と様々な機能をもち始めた GPU に対するプログラム開発効率を向上する目的で GPU メーカ (maker)NVIDIA が開発したプログラミング言語で，GPU の細部に渡る C-like なプログラミング環境を提供する。Microsoft が開発した HLSL もまた Cg 同様 C-Like なプログラミング環境である。Brook は，ANSI C に対してストリーム (stream) 処理のための言語拡張を行なったデータ並列言語の一種である。Merrimac[25] 等のストリームプロセッサ (stream processor) 向けに開発されたものが汎用 GPU 向けに移植され公開されている。Sh は GPU のハードウェア制約を隠蔽した仮想マシン (machine) を構築し，その仮想マシンに対する API を C++のクラスライブラリ (class library) として提供する。

Cg や HLSL はプログラマ (programmer) が GPU のハードウェアやその制約を意識してプログラムすることを前提にしているのに対して，Brook や Sh はユーザ (user) に GPU のハードウェアを隠蔽する。そのため，プログラムのどの部分が実際に GPU で実効されるのかが分りにくいという欠点がある。

第4章　実装

この章では，本研究で提案する3つの手法について述べる。まず実装環境について述べ，次に3つの手法に共通する事項を述べた後に，3つの手法それぞれの実装の概要を述べる。

4.1　実装環境

OSはWindowsXP Professionalを使用。CPUによる演算とデバイス(device)制御部分はC++言語を用い，Microsoft DirectX 9.0 SDK Update (December 2004)を利用して実装した。プログラマブルパイプラインの実装にはHLSLを用いた。

4.2　共通事項

すべての手法について共通する事項を述べる。

4.2.1　空間微分フィルタ

エッジの検出は空間微分フィルタを用いて行う。空間微分フィルタは1次微分フィルタと2次微分フィルタに分類できる。

1次微分フィルタはエッジの特定方向における濃淡の変化を検出する。横軸方向，縦軸方向について適用すれば，エッジにおける濃淡勾配を計算することができる。2次微分フィルタはエッジの全方向における濃淡の変化の変化を検出するが，濃淡勾配を検出することができない。

フィルタの実装は実に単純である。注目している画素とその周囲の画素の値に特定の係数をかけて足し合わせた値を注目画素に上書きする。これを画像上のすべての画素について行う。

1次微分フィルタの例としてSobelフィルタを，2次微分フィルタの例としてLaplacianフィルタを，図4.1に示す。それぞれ，中央の数字が注目画素にかける係数を表し，周囲の数字は注目画素の周囲の画素にかける係数を表す。

SobelフィルタやLaplacianフィルタはGPU上でもフラグメントプロセッサを利用する事で非常に容易に実装することが可能である。

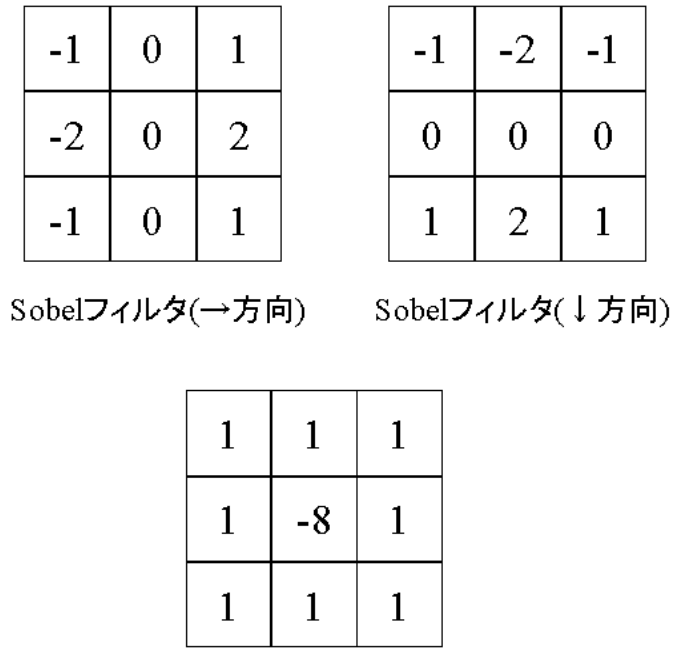


図 4.1: Sobel フィルタと Laplacian フィルタ

4.2.2 適合率の算出

入力画像とテンプレート画像の適合率は，それぞれのエッジを構成する画素数のうち，多いほうで投票数のピークを割ったものとする。この適合率が最も高いテンプレート画像が，入力画像と同じ文字を表しているとみなす。

4.2.3 拡大縮小について

本研究においては，拡大・縮小率について考慮しない。これは，OCR においてはパターンマッチングが行われる前の段階ですでに文字の大きさが特定され，パターンマッチングの段階での入力画像は一文字ごとに切り分けられた画像となるからである。

4.2.4 テンプレートについて

テンプレートはあらかじめテンプレート画像からエッジの情報 (代表点へのベクトル，エッジにおける濃淡勾配) だけを取り出して独自形式のファイル (file) として保存しておく。こうすることで，認識のたびに毎回テンプレートからエッジを検出するよりもいくらか高速化を図ることができる。

4.3 手法1

ここで紹介する手法は参考文献 [5] で紹介されている手法である。

α と r(2.2.2 参照) を算出するのは，通常の一般化 Hough 変換と同様である。テンプレートのエッジを構成する N 個の画素におけるそれぞれの α と r の値を α_i と $r_i(i = 1, 2, \cdots, N)$ とする。

入力画像に対し空間微分フィルタをかけてエッジを抽出する。エッジ画像と同じ大きさの 2 次元配列 E を用意し，エッジに対応する要素には 1 を，エッジでない部分に対応する要素には最寄のエッジまでの距離 l をパラメタとする関数 $g(l)$ によって算出される値を格納しておく。ただし，$g(l)$ は 1 未満の値を取り，単調減少であることが条件である。これはテンプレートと入力画像の形が若干ずれているだけでも敏感に反応して適合率が極端に低下してしまう事を防ぐための手法である。

次に，3 次元パラメタ投票空間 vote$[U][V][\Theta]$ を用意する。すべての vote$[u][v][\theta]$ について $\Sigma_{i=1}^{N} E[u - r_i \cos(\alpha_i + \theta)][v - r_i \sin(\alpha_i + \theta)]$ を求める。最後に vote のピークを求め，適合率を求める。

4.3.1 GPU 上での実装の概要

1. テンプレートのエッジ情報 α_i と $r_i(i = 1, 2, \cdots, N)$ を $1 \times N$ の大きさを持ったテクスチャ A に詰め込む。

2. 入力画像に Laplacian フィルタをかけ，更に 2 値化する。

3. 2 値化されたエッジ画像においてエッジでない画素に限定してガウスフィルタ (gauss filter) をかけ，ぼかす (これが単調減少関数 $g(l)$ に相当する)。この画像をテクスチャ E とする。

4. 回転角を何 °刻みで検出するかを定める。この角度を $\Delta\theta$ とする。
   $TH = [360/\Delta\theta]$ とする。

5. 入力画像と同じ大きさで TH 枚のテクスチャ $RT_j(j = 0, 1, \cdots, TH - 1)$ を用意する。

6. すべての RT について

   (a) RT_j をレンダリングターゲット (rendering target) に指定する。

   (b) 座標 (u, v) のピクセルを扱うフラグメントプロセッサは A に格納された α と r をすべて読み取り，$\Sigma_{i=1}^{N} E[u - r_i \cos(\alpha_i + j\Delta\theta)][v - r_i \sin(\alpha_i + j\Delta\theta)]$ を計算し，出力する。この出力は $RT_j(u, v)$ にレンダリングされる。

   (c) RT_j は CPU に送らず，レンダリングターゲットから外してビデオメモリ上に保持しておく。

7. 入力画像と同じ大きさを持つテクスチャ AT を用意する。AT をレンダリングターゲットに指定する。

8. 座標 (u, v) のピクセルを扱うフラグメントプロセッサはすべての j について $RT_j(u, v)$ の中で最大の値を持つ $RT_{jmax}(u, v)$ を探し，その値を出力する。この出力は $AT(u, v)$ にレンダリングされる。

9. 縦，横ともに AT の半分の大きさを持つテクスチャ BT を用意する。BT をレンダリングターゲットに指定する。

10. 座標 (u, v) のピクセルを扱うフラグメントプロセッサは $AT(u/2, v/2), AT(u/2+1, v/2), AT(u/2, v/2+1), AT(u/2+1, v/2+1)$ の中で最も大きな値を選び，出力する。この出力は $BT(u, v)$ にレンダリングされる。

11. AT に施された 9，10 と同様の操作を BT についても行い，更にその結果として出力される縦，横ともに BT の半分の大きさを持つテクスチャについても同様の操作を繰り返す。こうして出力されるテクスチャの大きさが 1×1 になるまで同様の操作を繰り返す。

12. 最終的に残った大きさ 1×1 のテクスチャを CPU に送る。この値が投票数のピークである。

ここで説明したアルゴリズムでは投票数のピークしか求めていないが，同時に投票数のピークが存在した投票空間のパラメタを求めるように改変することは容易である。

図 4.2 は手法 1 を実装したアプリケーションを実行した際の画面写真である。画面上段左と中段左に入力画像とテンプレート画像，上段中央と中段中央にそれぞれのエッジが表示されている。画面上段右には RT_{TH-1} における投票結果が，画面中段右には AT の内容がグラフィカル (graphical) に表示されている。画面下段は 9，10，11 の操作が進行していく様子を表示している。

4.3.2 考えうる利点と問題点

エッジにおける濃淡勾配を求める必要がないため，濃淡勾配検出の質による認識精度低下を気にする必要がないことが利点と言える。実装も非常に容易である。

問題点は，総当たり的な手法であるため，回転角を考慮するだけでも処理のオーダー (order) が膨大になってしまう点である。

4.4 手法 2

テンプレートのエッジを構成する N 個の画素 T_i における α と r と ω の値をそれぞれ α_i，r_i，$\omega_{1i} (i = 1, 2, \cdots, N)$ とする。また，入力画像のエッジを構成する M 個の画素 S_j における X 座標と Y 座標と濃淡勾配をそれぞれ X_j，Y_j，$\omega_{2j} (j = 1, 2, \cdots, M)$

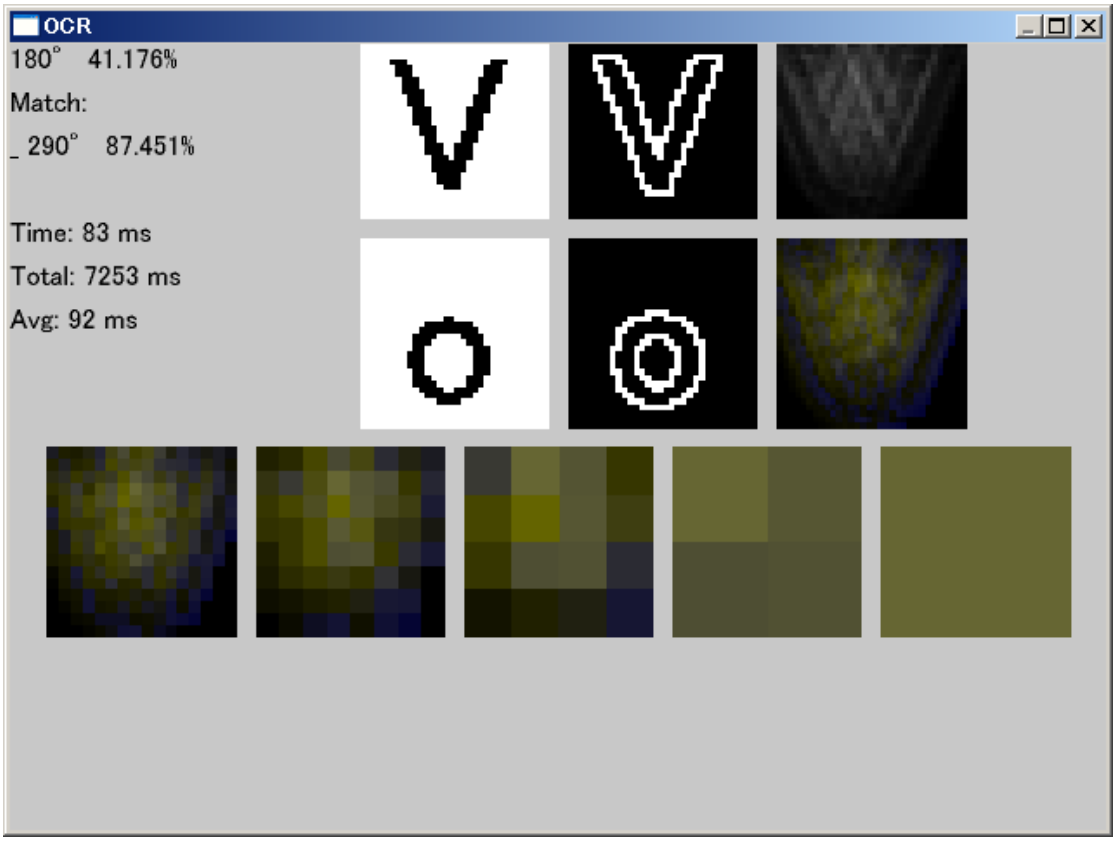


図 4.2: 手法 1 実装アプリケーション実行時

とする。また，3次元パラメタ投票空間 vote$[U][V][\Theta]$ を用意しておく。すべての T_i と S_j の組み合わせにおいて

$$RX_{ij} = X_j + sr_i \cos(\alpha_i + \omega_{2j} - \omega_{1i}) \quad (4.1)$$

$$RY_{ij} = Y_j + sr_i \sin(\alpha_i + \omega_{2j} - \omega_{1i}) \quad (4.2)$$

を求め，vote$[RX_{ij}][RY_{ij}][\omega_{2j} - \omega_{1i}]$ に投票する。実際には，投票先は投票空間中の一点でなく一定の範囲に投票する。これはテンプレートと入力画像の形が若干ずれているだけでも敏感に反応して適合率が極端に低下してしまう事を防ぐための手法である。最後に vote のピークを求め，適合率を求める。この手法は Chord-Tangent 変換の簡略化である。拡大縮小率を考慮することができないが，エッジとエッジを結ぶ線分の作成等の処理を省略し，高速化を図っている。逆に拡大縮小も考慮したい場合は，Chord-Tangent 変換を実装すれば良い。

手法 2 では RX_{ij} と RY_{ij} の計算にフラグメントプロセッサを利用する。レンダリングターゲットテクスチャとして $N \times M$ の大きさを持つテクスチャVT を用意し，$VT(i,j)$ に RX_{ij} と RY_{ij}，そして $\omega_{2j} - \omega_{1i}$ の値を格納する。VT を CPU に送り，CPU は VT のすべての要素を走査し，VT に格納された投票先に投票していく。ピークの調査も CPU で行う。

4.4.1 GPU 上での実装の概要

1. テンプレートのエッジ情報 α_i と r_i と $\omega_{1i}(i = 1, 2, \cdots, N)$ を $1 \times N$ の大きさを持ったテクスチャ A に詰め込む。

2. 入力画像のエッジを Sobel フィルタをかけることで検出する。

3. エッジ情報 X_j と Y_j と $\omega_{2j}(j = 1, 2, \cdots, M)$ を $1 \times M$ の大きさを持ったテクスチャ B に詰め込む。

4. $M \times N$ の大きさを持ったテクスチャVT を用意する。VT をレンダリングターゲットに指定する。

5. 座標 (u, v) のピクセルを扱うフラグメントプロセッサは A から α_u と r_u と ω_{1u} を読み取り，B から X_v と Y_v と ω_{2v} を読み取る。そして式 (4.1) と式 (4.2) より RX_{uv}と RY_{uv} を計算し，$\omega_{2v} - \omega_{1u}$ と合わせて出力する。この出力は $VT(u, v)$ にレンダリングされる。

6. VT を CPU に送る。

7. CPU は投票空間 vote$[U][V][\Theta]$ を用意する。

8. CPU は VT のすべての要素を走査し，VT に格納された投票先に投票していく。

9. CPU は投票数のピーク値を求める。

4.4.2　考えうる利点と問題点

手法 1 よりも明らかに処理オーダーが少ない点が利点として挙げられる。

問題点は，アルゴリズムの性質上，濃淡勾配の検出精度によって認識率が左右されるという点である。また，投票先の計算はフラグメントプロセッサで行うものの，実際の投票もピークの調査も CPU で行うため，若干のオーバヘッドが生じる点である。

4.5　手法 3

基本的な考え方は手法 2 と同様である。違うのはフラグメントプロセッサではなく頂点プロセッサを利用する点である。

手法 3 では RX_{ij} と RY_{ij} の計算に頂点プロセッサを利用する。投票はラスタライズ処理のオプション (option) として指定できる加算合成を利用する。ピークの調査は手法 1 と同様の手段を用い，GPU 上で行う。

4.5.1　GPU 上での実装の概要

1. テンプレートのエッジ情報 α_i と r_i と $\omega_{1i}(i = 1, 2, \cdots, N)$ を N の大きさを持った配列 A に保持しておく。
2. 入力画像のエッジを Sobel フィルタをかけることで検出する。
3. エッジ情報 X_j と Y_j と $\omega_{2j}(j = 1, 2, \cdots, M)$ を頂点データの列に詰め込む。
4. 回転角を何 °刻みで検出するかを定める。この角度を $\Delta\theta$ とする。
   $TH = [360/\Delta\theta]$ とする。
5. 十分に大きいテクスチャ RT を用意する。RT は入力画像と同じ大きさのブロック (block)TH 個に分ける事が可能であるくらいの大きさを持つとする。この TH 個のブロックを仮に $BRT_k(k = 0, 1, \cdots, TH - 1))$ と呼ぶ。
6. RT をレンダリングターゲットとする。
7. A のすべての要素について
   (a) $A[i]$ を均一入力として GPU に送る。
   (b) j 番目の頂点データを扱う頂点プロセッサは式 (4.1) と式 (4.2) より RX_{ij} と RY_{ij} を計算する。ここでブロック $BRT_{[(omega_{2j}-\omega_{1i})/\Delta\theta]}$ 内で最も小さい X 座標，Y 座標をそれぞれ X_{bmin}，Y_{bmin} と置く。出力する頂点の X 座標と Y 座標は，$X_{bmin} + RX_{ij}$ と $Y_{bmin} + RY_{ij}$ に設定する。出力する頂点の色は 1 を格納しておく。ただし $X_{bmin} + RX_{ij}$ と $Y_{bmin} + RY_{ij}$ の値が $BRT_{[(omega_{2j}-\omega_{1i})/\Delta\theta]}$ の範囲に収まらない場合は 0 を格納する。

(c) 結果として出力した頂点データがグラフィックス演算フローの処理に従ってラスタライズされ，更に RT に描画される。RT がすでに描画処理を受けている場合，ラスタライズで生成されたピクセルデータはすでに RT に描画されている画像と合成される。ラスタライズ・合成の方法を加算合成として指定しておくことで，$A[i]$ と各頂点データに格納された入力画像のエッジ情報から導かれる投票結果が RT に積み重なっていく。

8. 縦，横ともに RT の半分の大きさを持つテクスチャBT を用意する。BT をレンダリングターゲットに指定する。

9. 座標 (u, v) のピクセルを扱うフラグメントプロセッサは $AT(u/2, v/2), AT(u/2+1, v/2), AT(u/2, v/2+1), AT(u/2+1, v/2+1)$ の中で最も大きな値を選び，出力する。この出力は $BT(u, v)$ にレンダリングされる。

10. AT に施された 8，9 と同様の操作を BT についても行い，更にその結果として出力される縦，横ともに BT の半分の大きさを持つテクスチャについても同様の操作を繰り返す。こうして出力されるテクスチャの大きさが 1×1 になるまで同様の操作を繰り返す。

11. 最終的に残った大きさ 1×1 のテクスチャを CPU に送る。この値が投票数のピークである。

ここで説明したアルゴリズムでは投票数のピークしか求めていないが，同時に投票数のピークが存在した投票空間のパラメタを求めるように改変することは容易である。

図 4.3 は手法 3 を実装したアプリケーションを実行した際の画面写真である。画面上段左と中段左に入力画像とテンプレート画像，上段中央と中段中央にそれぞれのエッジが表示されている。画面下段は 8，9，10 の操作が進行していく様子を表示している。

4.5.2 考えうる利点と問題点

手法 1 よりも明らかに処理オーダーが少ない点が利点として挙げられる。また，CPU で投票，ピーク調査を行う必要がない。

問題点は手法 2 と同様，アルゴリズムの性質上，濃淡勾配の検出精度によって認識率が左右されるという点である。また，前章で述べた通り近年の GPU はフラグメントプロセッサ数が頂点プロセッサ数より多くなる傾向にあり，頂点プロセッサを利用してもフラグメントプロセッサを利用するほどの性能向上が見込めない。

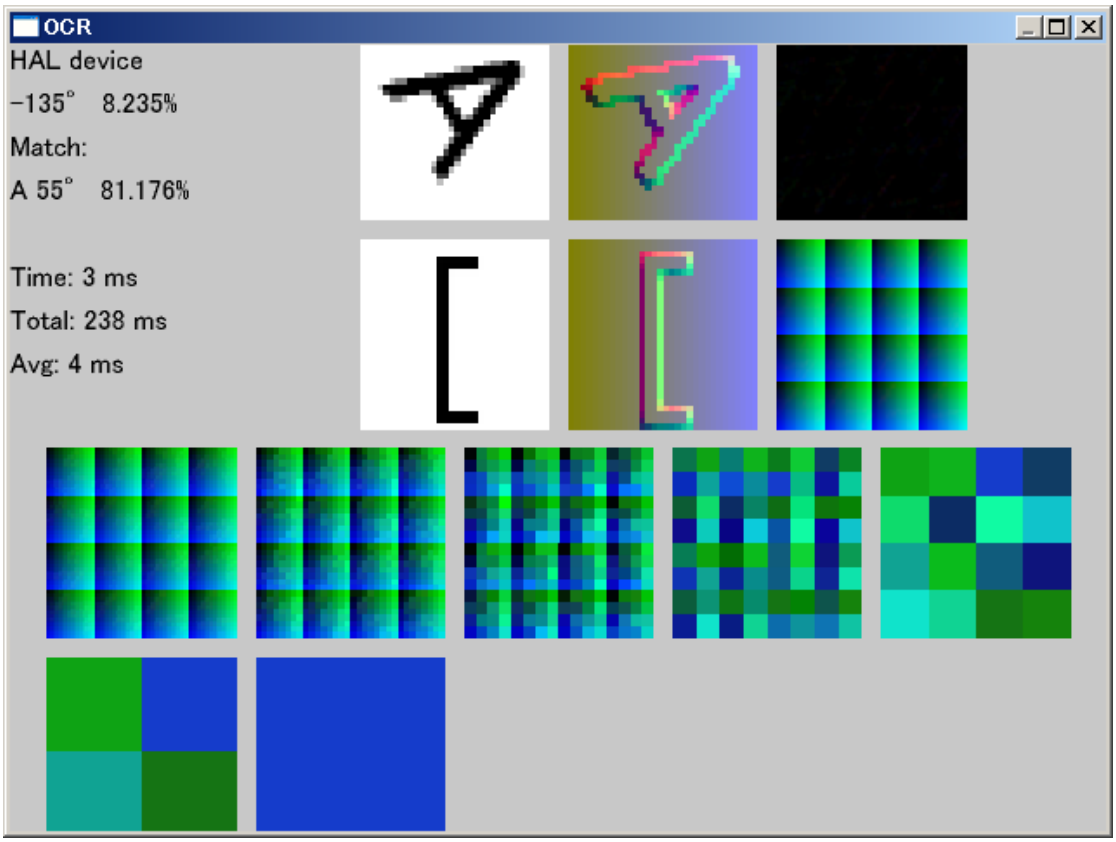


図 4.3: 手法 3 実装アプリケーション実行時

第5章　実験と評価

この章では，本研究にて提案した手法の性能を評価するための実験方法とその結果について述べる。その後，実験結果から提案手法の評価考察を行う。

5.1　実験環境

実験を行った環境を次に示す。

表 5.1: 実験環境

CPU(frequency)	Pentium4 (2.6GHz)
VGA(chipset)	6800/Albatron (GeForce 6800)
Memory Size	1GB
OS	Windows XP Professional

5.2　CPU での実装

比較対象とするため，手法 2 と同様のアルゴリズムを CPU 上で動く形に実装した。情報の格納にテクスチャを使わず通常の配列を用い，すべての計算を CPU 上で行っている他は，ほぼ手法 2 と同じ手順を踏んでいる。実装の詳細はここでは割愛する。

5.3　実験方法

実験方法の概要を述べる。

1. 入力画像を作成する。
    - ＭＳ Ｐゴシック，ＭＳ Ｐ明朝，HG 正楷書体-PRO の 3 種のフォント (font) を利用する。
    - 3 種のフォントそれぞれについて半角英数記号 (94 文字)，全角ひらがなカタカナ (169 文字)，漢字第 16 区 (94 文字) の入力画像を作成する。

図 5.1: 入力画像の例

- 入力画像は一つの文字につき 3 つずつ作成する。
- フォントサイズ (font size) は 32 とし，入力画像は 32 ピクセル*32 ピクセルの大きさとする。
- 画像はすべて，入力となる文字を任意の角度回転させたものとする。この操作により文字の一部が欠損する可能性がある。

入力画像の例を図 5.1 に示す。

2. テンプレートを作成する。

   - ＭＳ Ｐゴシック, ＭＳ Ｐ明朝, HG 正楷書体-PRO の 3 種のフォント (font) を利用する。
   - 3 種のフォントそれぞれについて半角英数記号，全角ひらがなカタカナ, 漢字第 16 区のテンプレートを作成する。
   - フォントサイズは 32 とする。

3. 実装手法 1，実装手法 2，実装手法 3，CPU による実装のそれぞれについて正解率と処理時間を計測する。

   - 半角英数字記号の入力画像は半角英数記号のテンプレートで，全角ひらがなカタカナの入力画像は全角ひらがなカタカナのテンプレートで，漢字第 16 区の入力画像は漢字第 16 区のテンプレートで認識を行う。認識を文字の種類ごとに分けるのは，それぞれの手法がどのような入力画像に良く，又は悪く働くのかを見る為である。
   - テンプレートのフォントと入力画像のフォントとのすべての (9 通りの) 組み合わせに対して認識を行う。
   - 正解率は 1 位正解率 (認識候補の 1 位に挙がったものが正解である率) とする。

5.4 結果

表中の項目に関して述べる。カテゴリ (category) は認識した文字が半角英数記号，全角ひらがなカタカナ，漢字第 16 区のいずれのカテゴリに属すかを表す。テンプレートは認識に使用したテンプレートのフォントを表す。入力は認識した入力画像のフォントを表す。正解率はそのカテゴリ，テンプレート，入力における 1 位正解率を表す。総処理時間はそのカテゴリ，テンプレート，入力におけるすべての文字を認識処理するのにかかった時間を示す。平均処理時間はそのカテゴリ，テンプレート，入力における 1 文字あたりの平均処理時間を示す。

次に実験の結果を示す。

5.4.1 CPU での実装

表 5.2: 結果–CPU での実装

カテゴリ	半角英数記号		
テンプレート	ＭＳ Ｐゴシック		
入力	ＭＳ Ｐゴシック	ＭＳ Ｐ明朝	HG 正楷書体-PRO
正解率 (%)	71.986	41.489	35.106
処理時間 (ms)	251797.323	268390.967	284139.612
平均処理時間 (ms)	892.898	951.741	1007.587

カテゴリ	半角英数記号		
テンプレート	ＭＳ Ｐ明朝		
入力	ＭＳ Ｐゴシック	ＭＳ Ｐ明朝	HG 正楷書体-PRO
正解率 (%)	44.326	65.248	45.745
処理時間 (ms)	270002.423	288291.499	305628.934
平均処理時間 (ms)	957.455	1022.310	1083.791

カテゴリ	半角英数記号		
テンプレート	HG 正楷書体-PRO		
入力	ＭＳ Ｐゴシック	ＭＳ Ｐ明朝	HG 正楷書体-PRO
正解率 (%)	31.560	44.681	70.567
処理時間 (ms)	292376.015	312807.105	332447.762
平均処理時間 (ms)	1036.794	1109.245	1178.893

カテゴリ	全角ひらがなカタカナ		
テンプレート	ＭＳ Ｐゴシック		
入力	ＭＳ Ｐゴシック	ＭＳ Ｐ明朝	HG 正楷書体-PRO
正解率 (%)	75.148	35.108	6.509
総処理時間 (ms)	1708903.730	1761232.046	1482243.981
平均処理時間 (ms)	3370.619	3473.830	5256.184

カテゴリ	全角ひらがなカタカナ		
テンプレート	ＭＳ Ｐ明朝		
入力	ＭＳ Ｐゴシック	ＭＳ Ｐ明朝	HG 正楷書体-PRO
正解率 (%)	29.586	83.432	18.935
総処理時間 (ms)	1801651.916	1857862.993	1565729.810
平均処理時間 (ms)	3553.554	3664.424	3088.224

カテゴリ	全角ひらがなカタカナ		
テンプレート	HG 正楷書体-PRO		
入力	ＭＳ Ｐゴシック	ＭＳ Ｐ明朝	HG 正楷書体-PRO
正解率 (%)	7.692	24.655	76.529
総処理時間 (ms)	1501442.829	1545972.527	1312403.361
平均処理時間 (ms)	2961.426	3049.255	2588.567

カテゴリ	漢字第 16 区		
テンプレート	ＭＳ Ｐゴシック		
入力	ＭＳ Ｐゴシック	ＭＳ Ｐ明朝	HG 正楷書体-PRO
正解率 (%)	98.936	83.688	17.021
処理時間 (ms)	1083787.637	1282265.578	1100732.773
平均処理時間 (ms)	3843.219	4547.041	3903.308

カテゴリ	漢字第 16 区		
テンプレート	ＭＳ Ｐ明朝		
入力	ＭＳ Ｐゴシック	ＭＳ Ｐ明朝	HG 正楷書体-PRO
正解率 (%)	91.489	99.291	38.652
処理時間 (ms)	1308375.214	1551327.140	1328217.465
平均処理時間 (ms)	4639.628	5501.160	4709.991

カテゴリ	漢字第 16 区		
テンプレート	HG 正楷書体-PRO		
入力	ＭＳ Ｐゴシック	ＭＳ Ｐ明朝	HG 正楷書体-PRO
正解率 (%)	17.376	31.915	100.000
処理時間 (ms)	1140922.026	1353063.983	1158034.485
平均処理時間 (ms)	4045.823	4798.099	4106.505

5.4.2 手法1

表 5.3: 結果–手法1

カテゴリ	半角英数記号		
テンプレート	ＭＳ Ｐゴシック		
入力	ＭＳ Ｐゴシック	ＭＳ Ｐ明朝	HG 正楷書体-PRO
正解率 (%)	78.723	47.872	39.716
処理時間 (ms)	1880114.945	1879917.451	1880598.734
平均処理時間 (ms)	6667.074	6666.374	6668.790

カテゴリ	半角英数記号		
テンプレート	ＭＳ Ｐ明朝		
入力	ＭＳ Ｐゴシック	ＭＳ Ｐ明朝	HG 正楷書体-PRO
正解率 (%)	50.355	85.816	64.184
処理時間 (ms)	2052364.716	2061637.734	2051129.625
平均処理時間 (ms)	7277.889	7310.772	7273.509

カテゴリ	半角英数記号		
テンプレート	HG 正楷書体-PRO		
入力	ＭＳ Ｐゴシック	ＭＳ Ｐ明朝	HG 正楷書体-PRO
正解率 (%)	26.241	43.262	85.106
処理時間 (ms)	2267266.041	2270030.048	2267425.804
平均処理時間 (ms)	8039.951	8049.752	8040.517

カテゴリ	全角ひらがなカタカナ		
テンプレート	ＭＳ Ｐゴシック		
入力	ＭＳ Ｐゴシック	ＭＳ Ｐ明朝	HG 正楷書体-PRO
正解率 (%)	81.854	20.316	1.972
総処理時間 (ms)	10225004.344	10217051.130	10217596.167
平均処理時間 (ms)	20167.661	20151.975	20153.050

カテゴリ	全角ひらがなカタカナ		
テンプレート	ＭＳ Ｐ明朝		
入力	ＭＳ Ｐゴシック	ＭＳ Ｐ明朝	HG 正楷書体-PRO
正解率 (%)	35.897	76.726	13.609
総処理時間 (ms)	10543973.832	10533934.100	10532509.536
平均処理時間 (ms)	20796.793	20776.990	20774.181

カテゴリ	全角ひらがなカタカナ		
テンプレート	HG 正楷書体-PRO		
入力	ＭＳ Ｐゴシック	ＭＳ Ｐ明朝	HG 正楷書体-PRO
正解率 (%)	25.641	57.791	82.643
総処理時間 (ms)	8625888.944	8628807.841	8626389.176
平均処理時間 (ms)	17013.588	17019.345	17014.574

カテゴリ	漢字第 16 区		
テンプレート	ＭＳ Ｐゴシック		
入力	ＭＳ Ｐゴシック	ＭＳ Ｐ明朝	HG 正楷書体-PRO
正解率 (%)	96.809	47.872	6.383
処理時間 (ms)	5055405.977	5055463.476	5054365.464
平均処理時間 (ms)	17926.972	17927.175	17923.282

カテゴリ	漢字第 16 区		
テンプレート	ＭＳ Ｐ明朝		
入力	ＭＳ Ｐゴシック	ＭＳ Ｐ明朝	HG 正楷書体-PRO
正解率 (%)	48.936	92.908	13.121
処理時間 (ms)	5730290.911	5733640.504	5746801.037
平均処理時間 (ms)	20320.181	20332.059	20378.727

カテゴリ	漢字第 16 区		
テンプレート	HG 正楷書体-PRO		
入力	ＭＳ Ｐゴシック	ＭＳ Ｐ明朝	HG 正楷書体-PRO
正解率 (%)	9.929	18.440	98.227
処理時間 (ms)	5031267.763	5030395.983	5032679.159
平均処理時間 (ms)	17841.375	17838.284	17846.380

5.4.3 手法 2

表 5.4: 結果–手法 2

カテゴリ	半角英数記号		
テンプレート	ＭＳ Ｐゴシック		
入力	ＭＳ Ｐゴシック	ＭＳ Ｐ明朝	HG 正楷書体-PRO
正解率 (%)	62.411	42.908	34.397
処理時間 (ms)	416772.812	420012.765	422867.054
平均処理時間 (ms)	1477.918	1489.407	1499.529

カテゴリ	半角英数記号		
テンプレート	ＭＳ Ｐ明朝		
入力	ＭＳ Ｐゴシック	ＭＳ Ｐ明朝	HG 正楷書体-PRO
正解率 (%)	29.787	57.092	41.489
処理時間 (ms)	419858.579	425371.132	426592.521
平均処理時間 (ms)	1488.860	1508.408	1512.739

カテゴリ	半角英数記号		
テンプレート	HG 正楷書体-PRO		
入力	ＭＳ Ｐゴシック	ＭＳ Ｐ明朝	HG 正楷書体-PRO
正解率 (%)	19.149	31.560	60.284
処理時間 (ms)	423917.568	428102.734	431540.583
平均処理時間 (ms)	1503.254	1518.095	1530.286

カテゴリ	全角ひらがなカタカナ		
テンプレート	ＭＳ Ｐゴシック		
入力	ＭＳ Ｐゴシック	ＭＳ Ｐ明朝	HG 正楷書体-PRO
正解率 (%)	61.933	30.572	2.170
総処理時間 (ms)	1514845.653	1520778.233	1470419.681
平均処理時間 (ms)	2987.861	2999.563	2900.236

カテゴリ	全角ひらがなカタカナ		
テンプレート	ＭＳ Ｐ明朝		
入力	ＭＳ Ｐゴシック	ＭＳ Ｐ明朝	HG 正楷書体-PRO
正解率 (%)	15.187	64.103	9.073
総処理時間 (ms)	1533616.065	1540590.892	1485711.214
平均処理時間 (ms)	3024.884	3038.641	2930.397

カテゴリ	全角ひらがなカタカナ		
テンプレート	HG 正楷書体-PRO		
入力	ＭＳ Ｐゴシック	ＭＳ Ｐ明朝	HG 正楷書体-PRO
正解率 (%)	6.114	30.966	60.947
総処理時間 (ms)	1478816.393	1483545.652	1439734.009
平均処理時間 (ms)	2916.798	2926.126	2839.712

カテゴリ	漢字第 16 区		
テンプレート	ＭＳ Ｐゴシック		
入力	ＭＳ Ｐゴシック	ＭＳ Ｐ明朝	HG 正楷書体-PRO
正解率 (%)	96.454	82.979	14.539
処理時間 (ms)	586305.155	617747.009	581183.197
平均処理時間 (ms)	2079.096	2190.592	2060.933

カテゴリ	漢字第 16 区		
テンプレート	ＭＳ Ｐ明朝		
入力	ＭＳ Ｐゴシック	ＭＳ Ｐ明朝	HG 正楷書体-PRO
正解率 (%)	83.333	99.645	26.596
処理時間 (ms)	629450.073	670799.388	625084.203
平均処理時間 (ms)	2232.092	2378.721	2216.611

カテゴリ	漢字第 16 区		
テンプレート	HG 正楷書体-PRO		
入力	ＭＳ Ｐゴシック	ＭＳ Ｐ明朝	HG 正楷書体-PRO
正解率 (%)	11.348	33.333	99.645
処理時間 (ms)	598231.844	632899.483	593480.846
平均処理時間 (ms)	2121.390	2244.324	2104.542

5.4.4 手法 3

表 5.5: 結果–手法 3

カテゴリ	半角英数記号		
テンプレート	ＭＳ Ｐゴシック		
入力	ＭＳ Ｐゴシック	ＭＳ Ｐ明朝	HG 正楷書体-PRO
正解率 (%)	68.440	41.844	32.624
処理時間 (ms)	33188.755	33875.435	35036.235
平均処理時間 (ms)	117.691	120.126	124.242

カテゴリ	半角英数記号		
テンプレート	ＭＳ Ｐ明朝		
入力	ＭＳ Ｐゴシック	ＭＳ Ｐ明朝	HG 正楷書体-PRO
正解率 (%)	38.298	70.922	51.773
処理時間 (ms)	34164.237	35781.661	37135.040
平均処理時間 (ms)	121.150	126.885	131.685

カテゴリ	半角英数記号		
テンプレート	HG 正楷書体-PRO		
入力	ＭＳ Ｐゴシック	ＭＳ Ｐ明朝	HG 正楷書体-PRO
正解率 (%)	33.333	51.418	83.333
処理時間 (ms)	35717.398	37092.745	39034.295
平均処理時間 (ms)	126.657	131.535	138.419

カテゴリ	全角ひらがなカタカナ		
テンプレート	ＭＳ Ｐゴシック		
入力	ＭＳ Ｐゴシック	ＭＳ Ｐ明朝	HG 正楷書体-PRO
正解率 (%)	80.079	39.842	8.284
総処理時間 (ms)	171441.797	175168.696	154936.425
平均処理時間 (ms)	338.150	345.500	305.595

カテゴリ	全角ひらがなカタカナ		
テンプレート	ＭＳ Ｐ明朝		
入力	ＭＳ Ｐゴシック	ＭＳ Ｐ明朝	HG 正楷書体-PRO
正解率 (%)	30.178	78.501	25.641
総処理時間 (ms)	178652.925	181545.875	162125.603
平均処理時間 (ms)	352.373	358.079	319.774

カテゴリ	全角ひらがなカタカナ		
テンプレート	HG 正楷書体-PRO		
入力	ＭＳ Ｐゴシック	ＭＳ Ｐ明朝	HG 正楷書体-PRO
正解率 (%)	6.903	28.994	73.767
総処理時間 (ms)	156128.684	158443.518	143389.177
平均処理時間 (ms)	307.946	312.512	282.819

カテゴリ	漢字第 16 区		
テンプレート	ＭＳ Ｐゴシック		
入力	ＭＳ Ｐゴシック	ＭＳ Ｐ明朝	HG 正楷書体-PRO
正解率 (%)	99.291	84.752	18.440
処理時間 (ms)	97814.106	110496.978	96736.784
平均処理時間 (ms)	346.859	391.833	343.038

カテゴリ	漢字第 16 区		
テンプレート	ＭＳ Ｐ明朝		
入力	ＭＳ Ｐゴシック	ＭＳ Ｐ明朝	HG 正楷書体-PRO
正解率 (%)	97.518	99.645	47.518
処理時間 (ms)	115381.402	131404.722	113735.877
平均処理時間 (ms)	409.154	465.974	403.319

カテゴリ	漢字第 16 区		
テンプレート	HG 正楷書体-PRO		
入力	ＭＳ Ｐゴシック	ＭＳ Ｐ明朝	HG 正楷書体-PRO
正解率 (%)	14.539	37.234	100.000
処理時間 (ms)	102675.760	116018.596	100505.339
平均処理時間 (ms)	364.098	411.413	356.402

5.5 考察

5.5.1 正解率について

正解率はおおむねどの手法についても同様の結果となった。

文字の回転を考慮に入れたため，回転すると別の文字とまったく同じものになってしまう文字（<，>，u，n，など）や非常に単純な形の文字，似たような文字の組を多く含む英数記号，かな文字の正解率が低くなってしまった。変形に対する許容範囲を導入したことで，その傾向は更に強くなったと考えられる。漢字第 16 区の文字については，かな文字や英数記号と同じ条件（任意の回転，文字の一部欠如）であるにも関わらず，正解率はほぼ 100%であった。これは，漢字第 16 区の文字は複雑で特徴的なエッジが多く存在するからであると考えられる。

テンプレートと入力画像でフォントが違っている場合は，著しく正解率が低下してしまった。これは，本研究ではすべての手法において 1 ピクセルから 2 ピクセル程度の変形に対する許容範囲を儲けたが，その程度の許容範囲ではフォントの違いによる文字形の変化に対応できなかった為であると考えられる。また，手法 1 は他の手法と比べてこの許容範囲が若干狭くなってしまった為に，他の手法の結果に比べてテンプレートと入力画像でフォントが違っている場合の正解率が低くなっている。

5.5.2 処理時間について

手法 1 はその総当り的手法が仇となり，並列化の恩恵は十分に受けることが出来たものの，CPU による実装よりも大幅に処理時間が増加してしまった。

手法 2 は手法 1 に比べれば大幅に処理時間を削減することができた。テンプレート，入力画像それぞれのエッジを構成する画素数が少ない英数字の認識においては，CPU による実装よりも多少処理時間が多くなる傾向にある。しかし，エッジを構成する画素数の増加による処理時間の増加は，CPU による実装におけるそれよりも小さい。最終的にエッジを構成する画素数の多い漢字の認識においては，CPU よりも優れた性能を記録した。これは実装の特性上，エッジを構成する画素数が少数の場合は並列化の恩恵よりも GPU から CPU への ReadBack 性能による制限の方が大きくなるからであると考えられる。また，投票と投票数のピーク検出を CPU にて行っているため，この部分の処理については CPU による実装よりも優れた性能を望む事はできない。

手法 3 は手法 2 に比べ更に大幅に処理時間を削減することができた。また，手法 2 と同様エッジを構成する画素数の増加による処理時間の増加は，CPU による実装におけるそれよりも小さい。エッジを構成する画素数が少ない英数字の認識においては CPU による実装の 8 倍弱の性能を示し，更にエッジを構成する画素数の多い漢字の認識においては 10 倍以上もの性能を記録した。この手法では GPU から CPU への ReadBack が少ない事，投票・ピーク検出においても並列化の恩恵を最大限に生かしている事が高速化に結びついていると考えられる。

第6章　おわりに

6.1　成果

本研究では画像認識アルゴリズムの一つである一般化 Hough 変換を汎用計算機上の文字認識に応用するため，これを GPU 上で動作させることで処理の高速化を図り，そのための 3 種類の手法を提案した。そのうち手法 3 においては大幅な処理時間削減に成功し，CPU による実装の 10 倍以上もの性能を示した。

6.2　改善点

6.2.1　回転角について

本研究では拡大率を考慮しなかった。これは OCR においてはパターンマッチングが行われる前の段階ですでに文字の大きさが特定され，パターンマッチングの段階での入力画像は一文字ごとに切り分けられた画像となるからである。

同様に，実際は回転角もほとんど考慮する必要はない。実際に本研究で提案した手法を OCR に応用するにあたっては，文字の回転は微小であると仮定すれば良いと考えられる。本研究では文字の自由な回転を考慮した為に英数記号や，かな文字の正解率が低くなってしまった。文字の回転角が制限されていると仮定できれば，提案手法は更なる高速化，高正解率が期待できる。

6.2.2　認識処理そのものの削減

本研究では GPU 上での一般化 Hough 変換の性能評価のため，一つの入力画像に対してすべてのテンプレートとの一致率を計算した。GPU 上で動作させ高速化されたとはいえ，一般化 Hough 変換がコスト (cost) のかかる処理であることに変わりはない。力任せにすべてのテンプレートとの一致率を計算するのは得策ではない。

現在実用されている OCR ではパターンマッチングを行う前に平均濃度比較等の手法によって，ある程度パターンマッチング処理を減らすための工夫が為されている。実際に本研究で提案した手法を OCR に応用する場合も，同様の工夫が必要である。

謝辞

お忙しい中ご指導下さった筧捷彦教授，長慎也さん，そして共に研究を行った宮永直樹君に，この場を借りて多大なる感謝の意を表すとともに，厚く御礼を申し上げます。

参考文献

[1] OCR イメージスキャナ用語集
http://it.jeita.or.jp/document/ocr_scanner/

[2] D.H. Ballard, "Generalizing the Hough transform to detect arbitrary shapes," Pattern Recognit., vol.13, no.2, pp.111-122, 1981.

[3] T.E. Dufresne, A.P. Dhawan, "Chord tangent transformation for object recognition," Pattern Recognit., vol.28, no.9, pp.1321-1331, 1995.

[4] GPGPU
http://www.gpgpu.org/

[5] Ivo Ihrke, Marcus Magnor, "A Graphics Hardware Implementation of the Generalized Hough Transform for fast Object Recognition, Scale, and 3D Pose Detection," International Conference on Image Analysis and Processing (ICIAP 2003), pp.188-193, 2003.

[6] 木村彰男, 渡辺孝志, "高速一般化ハフ変換　－相似変換不変な任意図形検出法－," 電子情報通信学会論文誌, Vol.J81-DII, No.4 pp.726-734, April 1998.

[7] 木村彰男, 渡辺孝志, "図形検出力を向上させた高速一般化ハフ変換," 電子情報通信学会論文誌, Vol.J83-DII, No.5 pp.1256-1265, May 2000.

[8] R.O. Duda and P.E. Hart, "Use of the Hough transformation to detect lines and curves in pictures," Communications of ACM, vol. 15, pp. 11-15, 1972.

[9] 佐藤誠, 石瀬裕之, 小川英光, "一般化 Hough 変換の最適性について," 第 18 回画像工学コンファレンス論文集, pp.347-350, 1987.

[10] "【レポート】映画のクオリティをリアルタイムで、NV30 の CineFX アーキテクチャー (1)," MYCOM PC WEB
http://pcweb.mycom.co.jp/news/2002/07/29/16.html

[11] デスクトップ用 CPU - intel, ATCOMPARTS
http://www.geocities.jp/atcomparts/cpudesktop2.html

[12] ビデオチップ - NVIDIA , ATCOMPARTS
http://www.geocities.jp/atcomparts/videonvidia1.html

[13] 後藤弘茂, "NVIDIA GeForce 6800(NV40) のウイークポイント," 後藤弘茂の Weekly 海外ニュース, PC Watch
http://pc.watch.impress.co.jp/docs/2004/0421/kaigai085.htm

[14] 後藤弘茂, "IDF で明らかになった Intel の将来 CPU 計画〜すべての CPU はマルチコアへ," 後藤弘茂の Weekly 海外ニュース, PC Watch
http://pc.watch.impress.co.jp/docs/2003/0919/kaigai025.htm

[15] Ertl, T., Weiskopf,D., Kraus, M., Engel,K., Weiler, M., Hopf, M., Rottger, S., and Rezk-Salama C., "Programmable Graphics Hardware for Interactive Visualization," Eurographics2002 Tutorial Note(T4) (2002).

[16] NVIDIA Corporation, NVIDIA GEFORCE FX 5900 PRODUCT OVERVIEW (2003).

[17] NVIDIA Corporation, NVIDIA CineFX Technical Breif, TB-00626-001_v01, (2003).

[18] 後藤弘茂, "NVIDIA が 16 パイプラインの強力 GPU「GeForce 6800(NV40)」を発表," 後藤弘茂の Weekly 海外ニュース, PC Watch
http://pc.watch.impress.co.jp/docs/2004/0415/kaigai083.htm

[19] Thompson, C., Hahn, S. and Oskin, M. , Using Modern Graphics Architectures for General-Purpose Computing : A Framework and Analysis, Proc. of 35th International Symposium on Microarchitecture (MICRO-35) pp. 306–320 (2002).

[20] NVIDIA Corporation, Cg Toolkit User's Manual Release 1.2 (2004).

[21] NVIDIA Corporation, NVIDIA OpenGL Extension Specifications (2003).

[22] Craig Peeper, Jason Mitchell, "DirectX 9 High Level Shading Language 入門," Microsoft.com Japan
http://www.microsoft.com/japan/msdn/directx/techart/shaderx2_introductionto.asp

[23] BrookGPU
http://graphics.stanford.edu/projects/brookgpu/

[24] Sh: A high-level metaprogramming language for modern GPUs
http://libsh.sourceforge.net/

[25] W.J. Dally, et al., "Merimac:Supercomputing with Streams," SC2003, November 2003