社団法人 情報処理学会 研究報告
IPSJ SIG Technical Report

2007－CG－126 (13)
2007／2／20

GPUを用いたモルフォロジー演算のベクトル化

前野滝授[†] 伊野文彦[†] 萩原兼一[†]

本稿では，大きな構造要素を用いるモルフォロジー演算の高速化を目的として，GPU上で動作するベクトル化手法を提案する．提案手法は，GPUを用いる既存手法を以下の2点において改良する．(1) 元画像のベクトル化によるGPU計算時間の短縮および (2) フラグメントプログラムの分割による大きな構造要素への対応．実験の結果，既存手法と比較してGPU計算時間を4分の1程度に短縮できた．また，提案手法は 32×32 画素を超える大きな構造要素を処理でき，既存手法における構造要素のサイズに関する制限を除去できた．

Vectorizing Mathematical Morphology Using the GPU

Ryuju Maeno,[†] Fumihiko Ino[†] and Kenichi Hagihara[†]

This paper presents a GPU-accelerated vectorization method, aiming at achieving fast mathematical morphology with large structuring elements. Our method improves a prior GPU-based method in terms of the following two points: (1) time reduction of GPU computation by vectorization of the original image; and (2) support of large structuring elements by splitting the fragment program. The experimental results show that the proposed method reduces GPU computation time by 75%, as compared with the prior method. Furthermore, our method is capable of dealing with larger structuring elements of 32×32 pixels, removing the limitation of the element size.

1. はじめに

モルフォロジー演算[1]は画像フィルタの1つである．この演算は元画像および構造要素を入力として，これらに対する集合演算として定義されている．演算の種類に応じて，2値画像または濃淡画像内の領域を拡張もしくは縮退できる．例えば，医用画像に対するノイズ除去[2]や特徴抽出[3),4]などに応用されている．

モルフォロジー演算は集合演算であるため，その計算量は元画像の画素数 $W_1 \times H_1$ および構造要素の画素数 $W_2 \times H_2$ に比例する．例えば，$W_1 = H_1 = 1000$ および $W_2 = H_2 = 60$ の2値画像が与えられたとき，安易な実装は2.6 GHz駆動のPentium 4上で50秒程度の実行時間を要する．したがって，多数のスライスからなる3次元画像に対しては，さらに長い実行時間を要する．

そこで，急速に性能を向上しているGPU[5),6)] (Graphics Processing Unit) を用いた高速化手法[7),8)]が提案されている．これらの手法は，画素値計算を独立な部分問題に分割し，これらをGPU上で並列処理する．分割には2通りの手法がある．元画像分割[8)]は，Eidheimによる手法[7)]の拡張であり，元画像の画素ごとの計算を部分問題とみなす．一方，構造要素分割[8)]は，構造要素の画素ごとの計算を部分問題とみなす．

前者は後者よりも高速であるが[8)]，従来の実装ではGPUが提供するベクトル演算を用いていない．したがって，性能に関して改善の余地がある．さらに，サイズ $W_2 \times H_2$ の増大とともに，GPU上で動作するフラグメントプログラム（以下，プログラム）が長くなる．したがって，従来の実装を用いて大きな構造要素を扱う場合，GPUの持つ命令スロットの数が不足し，プログラム実行に失敗する可能性がある．

そこで，本研究では大きな構造要素に対して高速なモルフォロジー演算を実現することを目的として，元画像分割に対するベクトル化手法を提案する．提案手法は，GPU上で効率のよい計算を実現するために，条件分岐を回避できるように元画像をベクトル化する．さらに，構造要素のサイズに関する制限を除去するために，モルフォロジー演算の分配則に着目しプログラムを一定の長さ以下に分割する．

以降では，2章でモルフォロジー演算について述べ，3章でGPUを用いる既存の高速化手法を紹介する．その後，4章で提案手法を示し，5章で実験結果を示

† 大阪大学大学院情報科学研究科コンピュータサイエンス専攻
Department of Computer Science, Graduate School of Information Science and Technology, Osaka University

(a) 元画像 A　　(b) 構造要素 B　　(c) 結果画像 $A \oplus B$

図 1　モルフォロジー演算の例

```
// Inputs: Original image A, structuring element B
// Output: Result image O = A ⊕ B
1: Initialize O;
2: foreach point (i, j) do begin
3:   foreach point (k, l) do begin
4:     if (A(i, j) = 1 && B(k, l) = 1) then
5:       O(i + k, j + l) := 1;
6:   end
7: end
```

図 2　ミンコフスキー和 $A \oplus B$ の Scatter 型アルゴリズム

```
// Inputs: Original image A, structuring element B
// Output: Result image O = A ⊕ B
1: Initialize O;
2: foreach point (i, j) do begin
3:   foreach point (k, l) do begin
4:     if (A(i − k, j − l) = 1 && B(k, l) = 1) then
5:       O(i, j) := 1;
6:   end
7: end
```

図 3　ミンコフスキー和 $A \oplus B$ の Gather 型アルゴリズム

す．最後に，6 章で本稿をまとめる．

2. モルフォロジー演算

モルフォロジー演算[1] は，ミンコフスキー和およびミンコフスキー差を基にしている．これらの組み合わせにより，ノイズ除去[2] や特徴抽出[3),4)] などの様々な処理を実現できる．

図 1 に示すように，集合 A（元画像）および集合 B（構造要素）が与えられたとき，A および B に属する要素 $\mathbf{a} \in A$ および $\mathbf{b} \in B$ のすべての組み合わせの和 $\mathbf{a}+\mathbf{b}$ からなる集合のことをミンコフスキー和と呼ぶ．ミンコフスキー和は次式で定義される．

$$A \oplus B = \bigcup_{\mathbf{b} \in B} A_{\mathbf{b}} \tag{1}$$

ここで，$\oplus$ はミンコフスキー和の演算子を表し，$A_{\mathbf{b}}$ は A を $\mathbf{b}$ だけ平行移動して得られる集合 $\{\mathbf{a}+\mathbf{b} : \mathbf{a} \in A\}$ を表す．

一方，ミンコフスキー差は，集合 B のすべての要素 $\mathbf{b}$ に対して $\mathbf{x}-\mathbf{b}$ が集合 A の要素となるような $\mathbf{x}$ の集合として定義されている．ミンコフスキー差は次式で定義される．

$$A \ominus B = \bigcap_{\mathbf{b} \in B} A_{\mathbf{b}} \tag{2}$$

ここで，$\ominus$ はミンコフスキー差の演算子を表す．

モルフォロジー演算は分配則が成り立つ．つまり，集合 A および B に加えて集合 C（構造要素）が与えられたとき，次式を満たす．

$$A \oplus (B \cup C) = (A \oplus B) \cup (A \oplus C) \tag{3}$$

なお，2 値画像に対するミンコフスキー差はミンコフスキー和を基に実現できる．具体的には，式 (1) における操作において，黒領域の拡張を白領域の拡張に置き換える．すなわち，構造要素の白黒を反転し，白い画素に対するミンコフスキー和を計算することでミンコフスキー差の結果画像を得られる．

図 2 に，ミンコフスキー和 $A \oplus B$ に対する単純なアルゴリズムを示す．このアルゴリズムでは，ij ループが元画像を走査し，kl ループが構造要素を走査している．以降では，白点および黒点の画素値を各々0 および 1 で表す．

3. GPU を用いた既存の高速化手法

Eidheim[7] は，元画像分割に基づく手法を示している．この手法は，元画像をテクスチャとして保持し，元画像上の各画素に対する計算を GPU 内の FP[6]（Fragment Processor）を用いて並列処理する．この場合，図 2 のアルゴリズムでは，FP は画素 $A(i,j)$ を中心として，その周辺画素 $O(i+k, j+l)$ へ書き込むことになる（Scatter 型）．したがって，互いの書き込み先が重複してしまい，そのままでは並列化できない．そこで，この重複を除去するために，画素 $O(i,j)$ の周辺画素 $A(i-k, j-l)$ を参照して $O(i,j)$ へ書き込むようアルゴリズムを変換している（Gather 型，**図 3**）．この変換により外側 ij ループ間の依存関係を除去でき，それらの並列処理が可能となる．

```
 1: float4 dilation(
        uniform sampler2D image: TEXUNIT0,
 2:     float2 coords: TEX0): COLOR
 3: {
 4:   static const float offset_w = 1/W1;
 5:   static const float offset_h = 1/H1;
 6:   p0 = tex2D(image,coord+(0,0));
 7:   p1 = tex2D(image,coord+(offset_w,0));
 8:   p2 = tex2D(image,coord+(-offset_w,0));
 9:   p3 = tex2D(image,coord+(0,offset_h));
10:   p4 = tex2D(image,coord+(0,-offset_h));
11:   return max(p0,max(p1,max(p2,max(p3,p4))));
12: }
```

図 4 Eidheim[7] の手法が用いるフラグメントプログラムの例

なお，FP における $O(i,j)$ への書き込みの詳細は，フラグメントプログラムにより記述する（**図 4**）．このプログラムは，図 3 における内側 kl ループの実行を担当する．つまり，構造要素内の黒点それぞれに対して参照命令 tex2D および算術命令 max を記述する．図 4 の例では，互いに隣接する 5 つの黒点からなる十字型の構造要素を用い，ミンコフスキー和を計算している．このように，プログラム内で参照する画素は構造要素の形状に依存する．なお，プログラムは Cg 言語[9] により記述されている．

一方，前野ら[8] は Eidheim の手法を基に，任意の構造要素を扱える拡張を示している．この拡張は前処理として，構造要素ごとのプログラムを自動生成する．この際，プログラムの長さは構造要素が持つ黒点の数に比例する（図 4）．したがって，プログラムが GPU の許容量を超えて命令スロットを要求する場合，プログラム実行に失敗する．また，この手法はベクトル演算を用いない．

さらに，前野ら[8] は構造要素分割に基づく手法を示している．この手法は図 2 のアルゴリズムにおける内側 kl ループを並列処理する．具体的には，構造要素をテクスチャに格納し，元画像上の各画素へテクスチャを張り付けることを繰り返す．この手法はフラグメントプログラムを用いない．

4. 提案手法

本章では，提案手法におけるベクトル化，プログラム分割および処理の流れについて述べる．

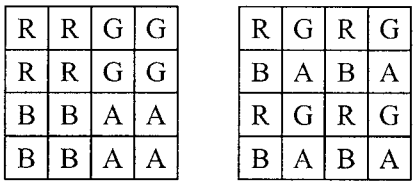
4.1 ベクトル化

よい性能を得るために，ベクトル化手法が満たすべき条件は下記の 2 点である．

R1. 計算量の多い箇所をベクトル演算すること

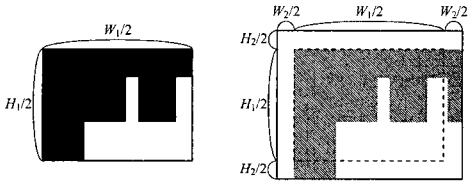
R2. 条件分岐を新たに必要としないこと

これらの条件に対し，提案するベクトル化手法の主な



(a) ブロック　(b) サイクリック

図 5 ベクトル化のための元画像の格納方法



(a) なし　(b) あり

図 6 ブロック分割におけるマージンの有無

特徴は下記の 3 点である．

C1. ベクトル化の対象が元画像である点

C2. ブロック分割に基づいたベクトル化である点

C3. ブロックの上下左右にマージンを持つ点

まず C1 に挙げた対象に関しては，提案手法の基となる手法が元画像に対する並列処理を行うため，元画像における画素間の並列性をベクトル化にそのまま利用できる．したがって，提案手法は元画像をベクトル化して高速化を図る．なお，構造要素そのものは一連の参照命令としてプログラム内に記述されているため，構造要素をデータとしてベクトル化することはできない．

次に C2 の分割に対する単純な案としては，元画像全体を均等なブロックに 4 等分するブロック分割（**図 5**(a)）および近隣 2×2 画素ごとに 4 等分するサイクリック分割（図 5(b)）が考えられる．いずれの分割においても，等分後の各スカラ画素を RGBA 成分からなるベクトル画素に格納すればよい．したがって，ベクトル化後のテクスチャサイズは双方ともに $W_1/2 \times H_1/2$ である．

ただし，サイクリック分割はモルフォロジー演算に対して条件 R2 を満たせない．したがって，提案手法はブロック分割を用いる．例えば，十字型の構造要素を与えられたとして，座標 (i,j) におけるベクトル画素の R 成分 $\mathbf{A}_r(i,j)$ を計算する際に必要な画素は，自身に加えて $\mathbf{A}_g(i,j)$，$\mathbf{A}_b(i,j)$，$\mathbf{A}_g(i-1,j)$ および

表 1　スカラ画素からベクトル画素への対応付け

ベクトル化後の画素値	元の画素値	条件
$\mathbf{A}_r(i,j)$	0*	$i < W_2/2$ もしくは $j < H_2/2$
	$A(i - W_2/2, j - H_2/2)$	上記以外
$\mathbf{A}_g(i,j)$	0*	$i \geq W_1/2 + W_2/2$ もしくは $j < H_2/2$
	$A(i + W_1/2 - W_2/2, j - H_2/2)$	上記以外
$\mathbf{A}_b(i,j)$	0*	$i < W_2/2$ もしくは $j \geq H_1/2 + H_2/2$
	$A(i - W_2/2, j + H_1/2 - H_2/2)$	上記以外
$\mathbf{A}_a(i,j)$	0*	$i \geq W_1/2 + W_2/2$ もしくは $j \geq H_1/2 + H_2/2$
	$A(i + W_1/2 - W_2/2, j + H_1/2 - H_2/2)$	上記以外

*: ミンコフスキー差の場合は 1 に初期化

$\mathbf{A}_b(i,j-1)$ である（図 5(b)）．一方，同じ座標の G 成分に対しては，自身に加えて $\mathbf{A}_r(i+1,j)$，$\mathbf{A}_a(i,j)$，$\mathbf{A}_r(i,j)$ および $\mathbf{A}_a(i,j-1)$ を必要とする．このように，サイクリック分割では画素内の成分に応じて，参照すべき画素の座標および成分が異なるために，ベクトル演算を適用できない．一方，ブロック分割は互いに離れたスカラ画素をベクトル化するため，ブロックが構造要素に対して大きい場合（$W_1 \geq W_2$ かつ $H_1 \geq H_2$），ベクトル演算を適用できる可能性がある．

最後に C3 は，ブロック間の境界領域に対して条件 R2 を満たすために必要である．3 章で述べた通り，元画像分割では計算対象とする画素の周辺を参照する．したがって，**図 6**(a) に示すようなマージンを持たないブロック分割を用いる場合，境界内の画素および境界外の画素が異なる成分に格納されていることが原因で条件分岐が必要である．そこで，この条件分岐を回避するために，提案手法では境界上の画素において参照しうる周辺画素のすべてをマージンとして同じ成分に格納する．この際，構造要素の大きさ $W_2 \times H_2$ を超えて周辺画素を参照することはないため，マージンの大きさは構造要素の半分とする（図 6(b)）．したがって，提案手法におけるベクトル化後のテクスチャサイズは $(W_1/2 + W_2) \times (H_1/2 + H_2)$ となる．

表 1 に，提案手法におけるスカラ画素からベクトル画素への対応付けを示す．マージン領域の画素値のうち，元のスカラテクスチャに存在しないものは，計算結果を変えないよう演算の種類に応じて初期化する．例えば，ミンコフスキー和であれば 0，ミンコフスキー差であれば 1 に初期化すればよい．なお，マージン領域は画素値を参照するために設けられていて，その領域へ計算結果を書き込むことはない．したがって，マージンの有無に関わらず計算量は同一である．

4.2　プログラム分割

プログラム分割は，式 (3) に示した分配則に基づく．すなわち，大きな構造要素 B が与えられたとして，それを $B = C \cup D$ を満たす小さな構造要素 C および

```
// Inputs: Original image A, structuring element B
// Inputs: Maximum point M for a single program
// Output: Result image O = A ⊕ B
 1: n := 1; c := 1; プログラム P_n の初期化;
 2: foreach (k,l) do begin
 3:    if B(k,l) = 1 then begin
 4:       B(k,l) に対する命令を P_n に追加出力;
 5:       c := c + 1;
 6:       if c > M then begin // プログラム分割
 7:          n := n + 1; P_n の初期化;
 8:          c := 1;
 9:       end
10:    end
11: end
12: CPU 上で A をベクトルテクスチャA に変換; // 表 1
13: foreach P_n do begin
14:    cgGLBindProgram(P_n);
15:    A の張り付け;
16:    出力結果をスカラに変換し途中結果 O_n として格納;
17: end
18: O := ∪_n O_n;// CPU 上での最終結果の計算
```

図 7　ミンコフスキー和 $A \oplus B$ に対する提案手法

D に分割できれば，次式を得る．

$$A \oplus B = (A \oplus C) \cup (A \oplus D) \tag{4}$$

式 (4) より，A および B に対するミンコフスキー和は，分割後に得られるミンコフスキー和の和集合と等しい．したがって，分割の前後において計算結果は同一である．

この性質を基に，提案手法ではプログラム実行に必要な命令スロット数が GPU の許容量を超える場合，プログラムを複数個に分割し，各々を許容内に収める．ここで，プログラムの長さは構造要素が持つ黒点の数に比例するため（3 章），許容内か否かは黒点の数で判断できる．最後に，各プログラムが出力した計算結果の和を計算し，最終結果を得る．構造要素のサイズに応じて分割数を増やすことにより，提案手法は構造要素のサイズに関する制限を除去できる．

4.3　処理の流れ

図 7 に，ミンコフスキー和に対する提案手法の概要を示す．元画像分割に対する差分としては，下記の 4

点が挙げられる．

- 構造要素を走査しながらプログラムを分割生成する処理（5〜9 行目）．構造要素 B における黒点の各々に対し，文献 8) に記載されている手法により命令をプログラムに追加出力する．この際，黒点の数を数えておき，許容できる数 M を超えるようであれば，命令の出力先を次のプログラムに変更する．
- 元画像のベクトル化（12 行目）．表 1 の対応表に基づいてベクトルテクスチャ**A** に画素を格納する．なお，逆方向の対応，すなわち計算結果を取り出すための処理も必要である（16 行目）．
- プログラムの繰り返し実行（13〜17 行目）．一連の生成済プログラムを順に変更しながら，**A** の張り付けを繰り返し，途中結果 O_n を得る．
- 最終結果の算出（18 行目）．分配則に基づいて O_n の和集合を計算する．

5. 評価実験

提案手法を評価するために，元画像分割および構造要素分割の実行時間を比較し，ベクトル化およびプログラム分割による効果を考察する．

実験環境として用いた PC は，CPU として 3 GHz 駆動の Pentium D および GPU として nVIDIA GeForce 7800 GTX（VRAM 容量 256MB）を備える．OS には WindowsXP を用いた．実験では，図 1 に示す元画像 A および構造要素 B を用いてミンコフスキー和 $A \oplus B$ を計算した．元画像のサイズは $W_1 \times H_1 = 800 \times 600$ である．一方，構造要素のサイズは $W_2 \times H_2 = 4 \times 4$ から 64×64 まで変化させた．なお，各々の手法は C++言語，OpenGL[10] および Cg[9] を用いて実装した．

5.1 実行時間の評価

図 8 に，手法ごとの実行時間を示す．ベクトル化前の元画像分割と比較して，ベクトル化後の提案手法は $W_2 = H_2 = 4$ のときにおよそ 70 ミリ秒から 27 ミリ秒，$W_2 = H_2 = 32$ のときにおよそ 1130 ミリ秒から 1004 ミリ秒に実行時間を短縮できている．また，提案手法は $W_2 = H_2 = 32$ 以下のときに構造要素分割よりも高速である．しかし，構造要素分割が要する実行時間は W_2 および H_2 に関わらずほぼ一定であるため，$W_2 = H_2 = 64$ を超える場合，構造要素分割の方が高速であった．

表 2 に，各手法における実行時間の内訳を示す．ここで，表中の準備時間は cgGLBindProgram および glBindTexture に要する時間を表し，GPU 実行時間

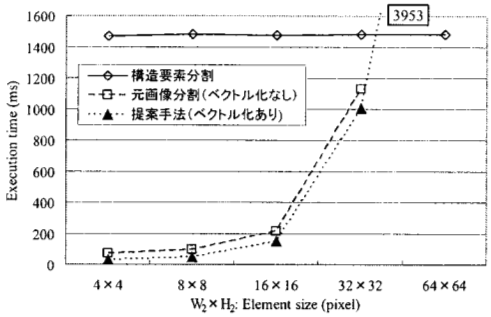


図 8 各手法の実行時間（ミリ秒）

は glBegin から glEnd までの実行時間を表す．また，CPU および GPU 間の転送時間はそれぞれ glTexImage および glReadPixels に要する時間を表す．なお，大きな構造要素に対するベクトル化の性能向上を示すため，元画像分割のスカラ実装にプログラム分割のみを追加した場合（部分実装）の内訳も示す．

提案手法は，ベクトル化により元画像分割の GPU 実行時間を 1/4 程度に短縮できている．この性能向上は妥当な結果である．なぜなら，提案手法は元画像における 4 スカラ画素を 4 成分からなる 1 ベクトル画素として一度にベクトル演算しているためである．実際に，ベクトル化の前後において，元画像上の 1 画素あたりに要する実行時間は変わっていない．例えば，$W_2 = H_2 = 16$ および $W_2 = H_2 = 64$ のとき，それぞれ 0.04 マイクロ秒および 0.6 マイクロ秒である．

5.2 オーバヘッドの評価

まず，提案手法におけるベクトル化のオーバヘッドを評価する．ベクトル化のために必要な処理は，元画像をベクトルテクスチャに格納する処理および途中の計算結果を取り出す処理である．これらに要する時間は，それぞれ 1.3 ミリ秒および 1.5 ミリ秒と短い（表 2）．したがって，ベクトル化のオーバヘッドは実行時間と比較して十分に小さい．さらに，これらはベクトル演算により帳消しされているため，ベクトル化はよい性能を得るために有用である．

次に，プログラム分割のオーバヘッドを評価する．表 2 をみると，$W_2 = H_2 = 64$ のとき，部分実装および提案手法はプログラムのバインドに長い時間を要している．この理由は，バインドに要する時間がプログラムの命令数に比例するためである．例えば，$W_2 = H_2 = 64$ のとき，これらの手法では 3073 個のアセンブリ命令を含むプログラムを 5 回ほどバインドしている．このように，プログラムの命令数が構造要素における黒点数に比例するため，準備時間はおおま

表 2　実行時間の内訳（ミリ秒）

内訳	$W_2 = H_2 = 16$				$W_2 = H_2 = 64$			
	構造要素分割	元画像分割	部分実装*	提案手法	構造要素分割	元画像分割	部分実装*	提案手法
元画像のベクトル化	—	—	—	1.3	—	—	—	1.3
プログラム生成	—	4.5	4.5	4.8	—	—	39.1	41.2
CPU→GPU 転送	2.5	2.4	2.5	0.8	2.4	—	2.4	1.0
GPU→CPU 転送	2.5	12.9	13.1	4.1	2.8	—	12.7	4.6
GPU 準備時間	0.2	131.3	131.5	124.8	0.4	—	3479.1	3492.6
GPU 実行時間	1444.9	16.3	16.7	4.2	1450.3	—	259.5	72.3
計算結果の取り出し	—	—	—	1.5	—	—	—	1.5
最終結果の計算	—	—	—	—	—	—	4.1	4.2
総実行時間	1475.6	215.4	217.8	148.1	1482.3	—	3952.7	3701.4

*: 元画像分割のスカラ実装にプログラム分割を加えたもの

かに W_2 および H_2 に比例する．

プログラム分割のオーバヘッドは小さくないが，提案手法は元画像分割が扱えない大きな構造要素を扱える．具体的には，黒点数が 1364 個を超える構造要素を扱える．

以上をまとめると，提案手法はベクトル化により 4×4 程度の小さな構造要素に対しておよそ 2 倍の高速化を達成できた．また，プログラム分割により 1364 個を超える黒点を持つ大きな構造要素を扱える．しかし，構造要素の増大とともに，テクスチャをバインドするための時間が長くなり，32×32 画素を超えると，構造要素分割よりも低速であることが分かった．

6. まとめと今後の課題

大きな構造要素を用いるモルフォロジー演算の高速化を目的として，元画像分割に基づくベクトル化手法を提案した．提案手法は，GPU が提供するベクトル演算を用い，高速化を図る．この際，条件分岐を回避できるように元画像をベクトル化する．さらに，大きな構造要素を扱うために，モルフォロジー演算の分配則に着目してフラグメントプログラムを分割する．

実験の結果，ベクトル化により既存の元画像分割と比較して GPU 実行時間を 1/4 程度に短縮した．また，ベクトル化は転送量を削減でき，4×4 画素からなる小さな構造要素に対し，構造要素分割に比べて 2 倍の高速化を実現した．一方，構造要素のサイズに関しては，提案手法は 32×32 画素を超える大きな構造要素を処理でき，サイズに関する制限を除去できた．しかし，構造要素分割の性能を上回ることはできなかった．

今後の課題としては，準備時間の短縮や高速化のための他のデータ構造の考案などが挙げられる．

謝辞　本研究の一部は，科学研究費補助金基盤研究 (B) (2) (18300009) および特定領域研究 (17032007) の補助による．

参 考 文 献

1) 小畑秀文：モルフォロジー，コロナ社，東京 (1996).
2) 野村行弘，斗澤秀亮，呂　建明，関屋大雄，谷萩隆嗣：モフォロジー処理を用いたスプクトルサブトラクションにおけるミュージカルノイズ除去，電子情報通信学会論文誌，Vol.J89-D, No.5, pp.991–1000 (2006).
3) 顧　力栩，金子豊久：3 次元モルフォロジによる腹部臓器領域の抽出法，電子情報通信学会論文誌，Vol.J82-D-II, No.9, pp.1411–1419 (1999).
4) 古川　章，南久松眞子，早田　勇：Morphological フィルタを用いたメタフェーズファインダの製作，電子情報通信学会技術研究報告，MI2005-105, pp.85–89 (2005).
5) Pharr, M. and Fernando, R.(eds.): *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*, Addison-Wesley, Reading, MA (2005).
6) Montrym, J. and Moreton, H.: The GeForce 6800, *IEEE Micro*, Vol.25, No.2, pp.41–51 (2005).
7) Eidheim, O. C.: Mathematical Morphology (2005). `http://www.idi.ntnu.no/emner/tdt16/lectures/lecture3.pdf`.
8) 前野滝授，伊野文彦，萩原兼一：モルフォロジー演算の高速化のための GPU 実装の比較，第 10 回 Visual Computing／グラフィクスと CAD 合同シンポジウム予稿集，pp.239–244 (2006).
9) Mark, W.R., Glanville, R.S., Akeley, K. and Kilgard, M.J.: Cg: A system for programming graphics hardware in a C-like language, *ACM Trans. Graphics*, Vol.22, No.3, pp.896–897 (2003).
10) Shreiner, D., Woo, M., Neider, J. and Davis, T.: *OpenGL Programming Guide*, Addison-Wesley, Reading, MA, fifth edition (2005).