

CUDA を用いた高速なモルフォロジー演算

前 野 滝 授[†] 伊 野 文 彦[†] 萩 原 兼 一[†]

本稿では CUDA (Compute Unified Device Architecture) 互換の GPU を用いたモルフォロジー演算の高速化手法を提案する。提案手法は、プロセッサおよびメモリ間のデータ転送量が少ないスライド型アルゴリズムを高速化する。この実現のために、提案手法は共有メモリを用いてデータを再利用し、さらなるデータ転送量の削減を図る。これにより複数のストリームプロセッサがデータを共有できるため、ストリームプロセッサは低速な VRAM 間のデータ転送量を削減できる。単純な CUDA 実装と比較して、提案手法はデータ転送量を最大で半減し、GPU 実行時間を 10 分の 1 に削減できている。また、CPU 実装と比較して 8 倍の高速化を実現できている。

A Fast Mathematical Morphology using CUDA

RYUJU MAENO,[†] FUMIHIKO INO[†] and KENICHI HAGIHARA[†]

This paper proposes a fast method for mathematical morphology using the compute unified device architecture (CUDA) compatible GPU. The proposed method accelerates a sliding-based algorithm, which requires less amount of data access between the processing unit and the memory device. To realize this, the method further reduces the amount by applying a data reuse technique to the data loaded on shared memory. This technique allows multiple stream processors to share the data, so that stream processors can reduce the amount of data access from slower video random access memory (VRAM). As compared with a simple CUDA implementation, our method reduces GPU execution time to 1/10 with reducing the amount into a half. It also achieves an 8x speedup over a CPU implementation.

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。この演算は原画像および構造要素を入力として、画像内の領域を拡張もしくは縮退できる。その演算の組み合わせにより、様々な処理を実現できる。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り、メモリ集中型 (memory intensive) の問題である。このため、そのデータ転送量は原画像のサイズ $W_1 \times H_1 \times D_1$ および構造要素のサイズ $W_2 \times H_2$ に比例する。例えば、 $W_1 = H_1 = 1024$, $D_1 = 64$ および $W_2 = H_2 = 32$ の多値画像に対し、安易な実装は 2.93GHz 駆動の CPU 上で 5 分程度の実行時間を要する。そこで、インタラクティブな処理を目指して様々な高速化手法が提案されている。例えば、構造要素の分割により計算量を削減する手法^{5)~8)}、計算を距離変換に置き換えることで

高速化する手法⁹⁾ および構造要素のスライドにより差分のみを計算する手法¹⁰⁾ がある。

一方、GPU (Graphics Processing Unit)¹¹⁾ を汎用計算に用いる試みである GPGPU (General-Purpose Computation on GPUs)¹²⁾ の研究が盛んである。GPU とは、グラフィックス処理を高速化するためのプロセッサである。GPU は著しい速さで性能が向上しており、最近の GPU は CPU を超える浮動小数点演算性能をもつ。例えば nVIDIA GeForce 8800 GTX は 345.6GFLOPS を達成している。

GPU でモルフォロジー演算を高速化する既存手法として、グラフィックスライブラリ OpenGL¹³⁾ を用いる手法が提案されている。例えば、原画像の各画素に対する処理を並列化する原画像分割手法¹⁴⁾ および構造要素の各画素に対する処理を並列化する構造要素分割手法¹⁴⁾ がある。これらの手法は、原画像もしくは構造要素の各画素に対する処理が独立である単純なアルゴリズムを用いている。OpenGL は、画素ごとの処理が独立であることを前提としていて、画素ごとの処理においてデータを共有できない。

そこで最近、GPU を汎用計算に用いるための環境

[†] 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

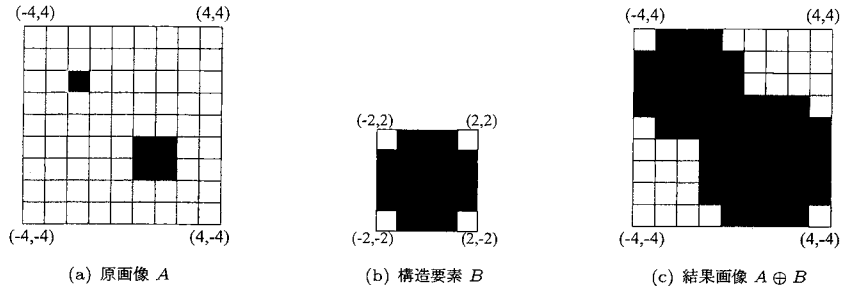


図 1 モルフォロジー演算の例

CUDA (Compute Unified Device Architecture)¹⁵⁾ が注目を集めている。CUDA は GPU を用いて多数のスレッドを並列処理する。また、CUDA はスレッド間でデータを共有できる共有メモリを持つ。共有メモリはレジスタ並に高速にアクセス可能である。したがって、メモリ集中型の問題に対し、共有メモリを使用することで高速化できる可能性がある。

本研究ではモルフォロジー演算の高速化を目的として、CUDA を用いた高速化手法を提案する。提案手法は、モルフォロジー演算がメモリ集中型の問題であることに着目し、データ転送量の削減により演算を高速化する。提案手法はスライド型アルゴリズム¹⁰⁾を用いる。スライド型アルゴリズムは、データ転送量を削減することで高速化を図るアルゴリズムである。このアルゴリズムは共有メモリを用いたデータ再利用に適しており、提案手法によりさらにデータ転送量を削減できる。

以降では、2 章でモルフォロジー演算およびスライド型アルゴリズムについて説明し、3 章で提案手法について述べる。その後、4 章で評価実験の結果を示し、最後に 5 章で本稿をまとめる。

2. モルフォロジー演算

本章では、モルフォロジー演算の定義を示し、既存の高速化手法としてスライド型手法を説明する。

2.1 定義

モルフォロジー演算¹⁾は、ミンコフスキー和、ミンコフスキー差およびそれらの組合せによる演算である。図 1 に示すように、集合 A (原画像) 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 $a \in A$ および $b \in B$ のすべての組み合わせの和 $a+b$ 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。

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

図 2 ミンコフスキー和 $A \oplus B$ のアルゴリズム

$$A \oplus B = \bigcup_{b \in B} A_b \quad (1)$$

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

一方、ミンコフスキー差は、集合 B のすべての要素 b に対して $x - b$ が集合 A の要素となるような 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。

$$A \ominus B = \bigcap_{b \in B} A_b \quad (2)$$

ここで、 $\ominus$ はミンコフスキー差の演算子を表す。

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 ループが原画像を走査し、 kl ループが構造要素を走査している。 $A(i, j)$ 、 $B(k, l)$ 、および $O(i, j)$ は各点の画素値である。ミンコフスキー差を求める場合は、このアルゴリズムの $\min$ 演算を $\max$ 演算に置き換えればよい。

2.2 スライド型手法

スライド型手法¹⁰⁾は、結果画像を求める際に計算結果を再利用する。具体的には、互いに隣接する画素 $O(i, j)$ および $O(i+1, j)$ において、それらの計算のために必要な画素が重複していることに着目し、計算結果を再利用する。図 3 に $O(i, j)$ および $O(i+1, j)$ を求めるときに重複する参照範囲を示す。

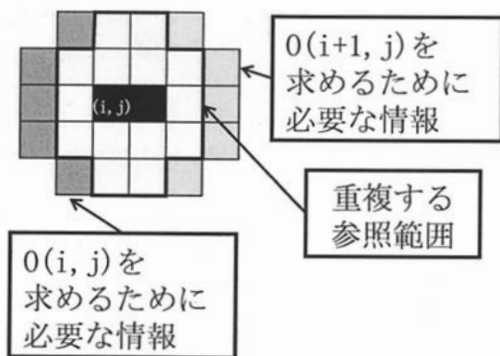


図3 隣接画素を求める際に重複する参照範囲

ここで、参照範囲とは結果画像の画素 $O(i, j)$ に対して、結果画像を求めるために参照する原画像の領域 $\{A(i-k, j-l) \mid (k, l) \in B\}$ である。この領域は原画像の画素 $A(i, j)$ に構造要素の原点を重ねた時に構造要素の範囲に含まれる原画像の領域である。

一般的なモルフォロジー演算では、原画像の各画素について参照範囲内のすべての画素値を参照し、最小値または最大値を算出する計算を繰り返す。一方、スライド型手法では $O(i, j)$ を求める際、重複部分の計算結果を記憶し、 $O(i+1, j)$ を求める計算に再利用する。このため、原画像に対し横軸方向に構造要素をスライドさせることを考える。構造要素をスライドさせる場合、スライド前後において参照範囲内に存在する画素については計算結果を再利用できる。具体的には、スライドにより参照範囲に加わる領域のみを新たに計算すればよい。さらに発展させれば、計算結果（部分解）を再利用できる回数ごとに分類し、これらを更新していけばよい。

スライド型手法では、原画像における各画素が参照範囲内に入ってから参照範囲外に外れるまでの構造要素のスライド回数を寿命と呼ぶ。図4に参照範囲における画素の寿命および部分解を格納する配列 S の関係を示す。参照範囲内の画素は寿命ごとの部分集合へと分類し、その部分解（最小値）を寿命ごとに配列 S に保存し再利用する。

スライド型手法では構造要素の形状を表すためのデータ構造としてスタートポイントを用いる。表1に、図4に示す構造要素のスタートポイントを示す。スタートポイントは、参照範囲のうち構造要素をスライドするとき新たな画素が加わる座標およびそれらの寿命の組合せで表す。その座標は構造要素における相対座標で表す。スタートポイントにおける寿命はスタートポイントの座標に新たに加わる画素の最初の寿

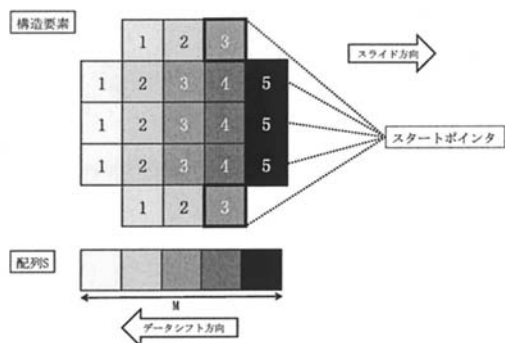


図4 参照範囲内の画素の寿命と部分解を格納する配列 S の関係

表1 図4に示す構造要素のスタートポイント

座標	寿命
(1,2)	3
(2,1)	5
(2,0)	5
(2,-1)	5
(1,-2)	3

命を表す。ゆえに、寿命の最大値 M は構造要素の横幅に等しい。また、配列 S の長さは M である。

以下にスライド型手法において結果画像の1行を求めるアルゴリズムを示す。この処理を原画像の各行に対して適用することで結果画像を求める。また、それぞれの行における処理は独立である。ここではミンコフスキー和を例に説明する。

- (1) 構造要素を参照してスタートポイントを設定する。
- (2) 原画像における対象行の左端画素に構造要素の原点を重ね、参照範囲内の画素をそれぞれ寿命 l ごとに $b_1, b_2, \dots, b_M$ に分類する。ただし、 b_l は寿命 l の画素集合である。
- (3) $1 \leq l \leq M$ を満たす全ての l について b_l の部分解（最小値）を求め、配列 S の l 番目の要素を $S[l] = \min_{b \in b_l} b$ とする。
- (4) 配列 S の部分解の最小値 $R = \min_{1 \leq l \leq M} S[l]$ を求め、現在の参照範囲における解とする。
- (5) 構造要素を1つスライドするとともに配列 S の全ての要素を左シフトする。
- (6) 原画像におけるスタートポイント位置の画素を参照し、それぞれの画素を寿命 l ごとに b_l に分類する。
- (7) $1 \leq l \leq M-1$ を満たす全ての l について、 $S[l]$ および画素集合 b_l における最小値 $S_{new} = \min_{b \in b_l} b$ を比較し、 $S[l]$ をそれらの

最小値 $\min(S[l], S_{new})$ に更新する。

- (8) 更新した配列 S の内容から現在の参照範囲における解 R を求める。
- (9) 構造要素が原画像の右端に来るまで (5) ~ (8) を繰り返す。

(5) および (7) の配列 S の更新について補足する。(5) における配列 S の左シフトは参照範囲内の各画素の持つ寿命の減少を表す。構造要素が1画素ほどスライドすると、参照範囲内の各画素の寿命は1つ減少する。つまり、寿命が l である画素集合は、寿命が $l-1$ である画素集合になる。したがって、 S に格納していた部分解は左シフトする必要がある。このとき、シフト前の $S[1]$ に格納していた部分解は破棄する。この破棄する部分解は構造要素のスライドによって参照範囲から外れる画素集合の部分解である。また、(7) における S の更新では、もとの S の内容および新たに参照範囲に加わる画素集合を比較して最小値を求める。

3. 提案手法

本章では、まず CUDA におけるスライド型手法の並列化を述べ、次に共有メモリを用いたデータ転送量削減について述べる。

3.1 並列化

提案手法は原画像における列ごとの計算を並列処理する。2.2 節で述べたようにスライド型手法の処理は原画像の行ごとの独立な処理に分割できる。ただし、これは構造要素がスライドする方向が右方向の場合である。提案手法では構造要素のスライド方向を下方向とする。このため、提案手法では原画像の列ごとの処理が独立である。

図5に提案手法におけるスレッドブロック（以降では、1つのブロックと呼ぶ）およびスレッドの割り当てを示す。提案手法では1つのスレッドに原画像における1列分の処理を割り当てる。これは1列分の処理が並列化できる最小の単位だからである。それぞれの列における処理は各スレッドで逐次実行する。各ブロックの x 軸方向のサイズはブロックあたりのスレッド数と等しいため、グリッドの x 軸方向のサイズは $gx = W_1 / \text{block_size}$ である。 W_1 および block_size はそれぞれ原画像の横幅およびブロックあたりのスレッド数である。

3.2 データ転送量の削減

提案手法では、共有メモリを用いた原画像データの再利用により、グローバルメモリおよびプロセッサ間のデータ転送量を削減する。これにより実行時間を短

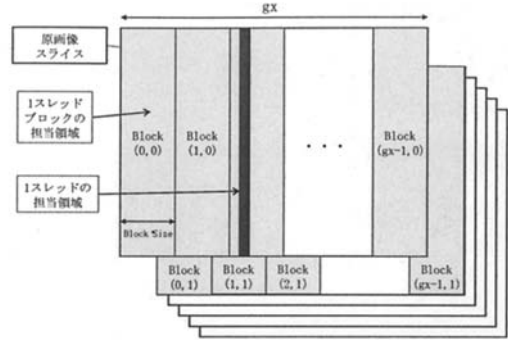


図5 ブロックおよびスレッドの割り当て

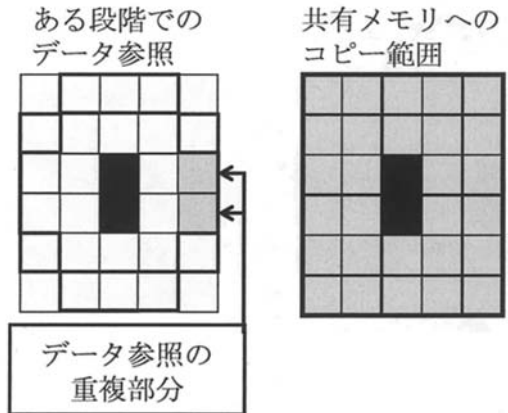


図6 データ参照の重複部分および共有メモリへコピーする範囲

縮できる。これは、共有メモリの転送遅延がグローバルメモリよりも $1/100$ 程度で短いからである。

CUDAを用いる場合、原画像データはCPU側からアクセス可能なグローバルメモリに保持する。安易な実装では、スレッドごとに必要なデータをグローバルメモリからロードするため、データ転送量はスレッド数に比例する。この場合、同一ブロック内の各スレッドがグローバルメモリからロードするデータには重複する部分がある。図6に各スレッドのデータ参照における重複部分を示す。提案手法では、重複部分を含めた原画像データのうち各スレッドが参照しうる部分を、それぞれのスレッドが属するブロックの共有メモリへ一度コピーする。その後、ブロック内の各スレッドは計算に必要な原画像データを共有メモリからロードする。ゆえに、提案手法では共有メモリへのコピーのためにグローバルメモリから1度だけデータロードすればよい。

提案手法では構造要素をスライドさせることに共有

メモリの内容を更新し、不要な部分を破棄する。不要な部分とは、ブロック内のどのスレッドも二度と参照しない部分である。このように、不要になった部分を順次破棄することで共有メモリ使用量を抑えることができ、並列処理可能なブロック数が増加し CUDA の実行効率が向上する。これは CUDA において、共有メモリ使用量の合計が共有メモリ容量を超えない範囲で複数のブロックを平行処理できるためである。

提案手法は 32×32 画素より大きい構造要素を扱えない。その場合、共有メモリ使用量が共有メモリ容量の 16KB を超えるためである。ブロックあたりの共有メモリ使用量は構造要素サイズに比例する。

4. 評価実験

本章では提案手法の性能を評価するために、単純な CUDA 実装、OpenGL 実装および CPU 実装の実行時間を比較し評価する。OpenGL 実装はグラフィックスライブラリ OpenGL¹³⁾ を用いた GPU 実装である。

表 2 に実験に用いた各手法の特徴を示す。GPU を用いない比較対象として CPU 版のスライド型¹⁰⁾、距離変換型⁹⁾ および 1 次元分解型⁸⁾ を用いた。また、OpenGL を用いた比較対象には構造要素分割手法¹⁴⁾ および原画像分割手法¹⁴⁾ を用いた。そして、CUDA を用いた比較対象として単純型を用いた。

実験環境として用いた PC は、CPU として 2.93GHz 駆動の Core2 Extreme X6800 および GPU として nVIDIA GeForce 8800 GTX (VRAM 容量 768MB) を備える。OS には Windows XP を用いた。また、GPU のドライバは NVIDIA 社の ForceWare 169.09 を用いた。原画像にはマルチスライス CT 画像を用いた。原画像サイズは 512×512 画素のスライス 300 枚である。一方、構造要素として 3×3 画素から 9×9 画素の円を用いた。

4.1 実行時間の評価

提案手法および比較対象手法の実行時間を比較した。図 7 に提案手法および比較対象の実行時間を示す。また、図 8 に 9×9 画素の構造要素を適用した場合の実行時間の内訳を示す。なお、ダウンロード時間は主記憶からグローバルメモリへのデータ転送時間を表し、リードバック時間はグローバルメモリから主記憶へのデータ転送時間である。

提案手法は実験に用いた各手法の中で最も高速であった。提案手法は CPU スライド型に比べ 20 倍程度高速である。さらに、図 8 によると CPU および GPU 間のデータ転送時間を除いた計算時間については、提案手法は 1 次元分解型より 15 倍高速である。また、

表 2 各手法の特徴

手法	実装	多値	構造要素の制限
スライド型 ¹⁰⁾	CPU	○	制限なし
距離変換型 ⁹⁾		×	特定の形状のみ
1次元分解型 ⁸⁾		○	制限なし
構造要素分割型 ¹⁴⁾	OpenGL	○	制限なし
原画像分割型 ¹⁴⁾		○	制限なし
単純型	CUDA	○	制限なし
提案手法		○	32×32 画素まで *

*: nVIDIA GeForce 8800 GTX における値

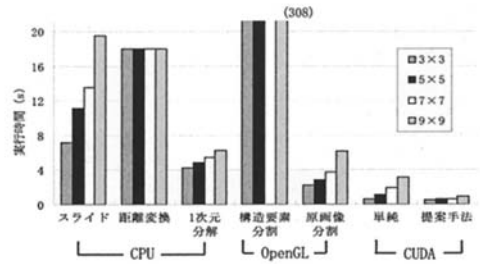


図 7 実行時間 (単位: ms)

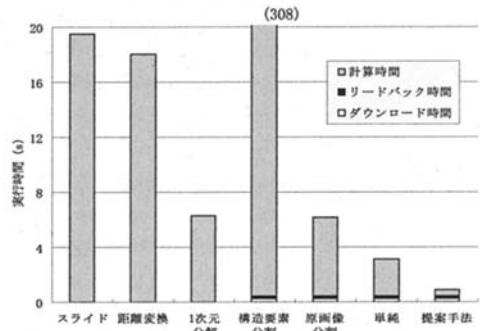


図 8 9×9 の構造要素に対する実行時間の内訳 (単位: ms)

構造要素が 3×3 画素のとき提案手法は CUDA 単純型と同程度の実行時間である。それ以上の構造要素サイズの場合は提案手法の方が高速で、 9×9 画素の場合 CUDA 単純型に比べ 4 倍程度高速化できている。

提案手法による高速化により、性能ボトルネックが計算時間から主記憶およびグローバルメモリ間のデータ転送時間に移っている。今回の実験では、原画像分割型におけるデータ転送時間が全体実行時間のうち 10%程度であるのに対し、提案手法では全体実行時間の 70%程度がデータ転送時間であった。

4.2 データ転送量の評価

グローバルメモリおよびプロセッサ間のデータ転送量を評価する。表 3 に原画像分割型、CUDA 単純型および提案手法のデータ転送量を示す。

表 3 原画像分割手法および CUDA を用いた各実装のデータ転送量 (単位: GB)

構造要素	3 × 3	5 × 5	7 × 7	9 × 9
原画像分割型	1.8	6.4	11.1	20.5
CUDA 単純型	1.8	6.4	11.1	20.5
提案手法	3.9	6.3	7.4	11.0

3 × 3 画素より大きい構造要素の場合, スライド型の方がデータ転送量が少ない. CUDA 単純型および CUDA スライド型を比較すると, それぞれ実行時間がデータ転送量に比例していることから, データ転送量削減は実行時間短縮に効果的であると言える.

5. おわりに

本稿では CUDA を用いた高速なモルフォロジー演算手法を提案した. 提案手法はスライド型アルゴリズムを用い, CUDA で並列計算する. その際, VRAM の原画像データを高速にアクセスできる共有メモリにコピーしプロセッサ間で再利用する. これにより, VRAM およびプロセッサ間のデータ転送量を削減する.

実験の結果, 提案手法は CPU 実装および OpenGL 実装に比べ最も高速であった. その実行時間は 512 × 512 × 300 画素の原画像に対して 9 × 9 画素の構造要素を適用した場合で 870 ミリ秒程度であった. これは, CPU での既存手法に比べ 8 倍高速であった. また, 単純なアルゴリズムに比べ, 提案手法はデータ転送量を最大で半減できた.

今後の課題として, 共有メモリ使用量の削減や VRAM 上に格納している部分解の共有メモリへの移動が挙げられる.

謝辞 本研究の一部は, 科学研究費補助金基盤研究 (B) (2) (18300009), 若手研究 (B) (19700061) および大阪大学グローバル COE プログラム「予測医学基盤」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: モルフォロジー処理を用いた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 力棚, 金子豊久: 3次元モルフォロジー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) Pitas, I. and Venetsanopoulos, A. N.: Morphological Shape Decomposition, *IEEE Trans. Pattern Analysis and Machine Intelligence*, Vol.12, No.1, pp.38-45 (1990).
- 6) Xu, J.: Decomposition of Convex Polygonal Morphological Structuring Elements into Neighborhood Subsets, *IEEE Trans. Pattern Analysis and Machine Intelligence*, Vol. 13, No.2, pp.153-162 (1991).
- 7) Park, H. and Chin, R. T.: Decomposition of Arbitrarily Shaped Morphological Structuring Elements, *IEEE Trans. Pattern Analysis and Machine Intelligence*, Vol. 17, No. 1, pp. 2-15 (1995).
- 8) 横井茂樹, 鳥脇純一郎, 福村晃夫: 画像処理のための 2 次元フィルタリングの 1 次元分解について, 電子情報通信学会論文誌, Vol.J61-D, No.7, pp.512-513 (1978).
- 9) 櫻井敦史, 平田富夫: 効率の良いモルフォロジー演算が可能なフィルタ形状について, 情報処理学会論文誌, Vol.41, No.12, pp.3344-3351 (2000).
- 10) 萩原義裕, 小畑秀文: 高速モルフォロジーフィルタリングアルゴリズム, 電子情報通信学会論文誌, Vol.J85-D-II, No.8, pp.1341-1350 (2002).
- 11) Pharr, M. and Fernando, R.(eds.): *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*, Addison-Wesley, Reading, MA (2005).
- 12) GPGPU: General-Purpose Computation Using Graphics Hardware (2007). <http://www.gpgpu.org/>.
- 13) Shreiner, D., Woo, M., Neider, J. and Davis, T.: *OpenGL Programming Guide*, Addison-Wesley, Reading, MA, fifth edition (2005).
- 14) 前野滝授, 伊野文彦, 萩原兼一: GPU を用いたモルフォロジー演算のベクトル化, 情報処理学会研究報告, 2007-CG-126, pp.73-78 (2007).
- 15) nVIDIA Corporation: CUDA Programming Guide Version 1.1 (2007). <http://developer.nvidia.com/cuda/>.