情報処理学会研究報告
IPSJ SIG Technical Report

Vol.2010-CG-138 No.11
2010/2/12

GPUを用いた画像の連結成分抽出に関する一考察

井村 誠孝[†1]　大城 理[†1]　千原 國宏[†2]

コンピュータビジョンを用いた画像解析の研究においては，前処理として連結領域ラベリングが利用されることが多い．本研究の目的は，連結領域ラベリング処理を，GPU を用いて並列処理するアルゴリズムを開発することである．本稿では，ラスタベース連結領域ラベリング手法を取り上げる．特にラスタセグメントの抽出に関して，画像の各ピクセルにスレッドを割り当てる並列性の高いアルゴリズムを提案し，連結領域ラベリングにおける GPU の利用可能性について検討する．

A Study on GPU-Based Connected Component Labeling

Masataka Imura,[†1] Osamu Oshiro[†1]
and Kunihiro Chihara[†2]

Connected component labeling is widely used as the preprocessing of image analysis based on computer vision theory. This research aims to develop the parallel connected component labeling algorithm suitable for GPU computing. In this paper, we deal with a raster-based connected component labeling algorithm. We propose a highly parallelized algorithm in which each thread treats one pixel and discuss applicability of GPU to connected component labeling.

1. はじめに

連結領域ラベリング（connected component labeling）は，画像内の連結した領域を構成するピクセルに共通のラベルを付与する処理である[1)]．コンピュータビジョンを用いた画像解析の研究においては，前処理として連結領域ラベリングによる関心対象物体の抽出を必要とすることが多く，アプリケーションも人物認識やヒューマンコンピュータインタフェースなど多岐にわたっている．高解像度カメラや高速度カメラの普及により，リアルタイム画像処理の分野においては入力される情報量が増大の一途をたどっており，連結領域ラベリングの高速化が希求されている．

本研究の目的は，連結領域ラベリング処理を，GPU を用いて並列処理するアルゴリズムを開発することである．GPU 上での画像処理の実装には，CPU での処理と比較して次のような利点がある．

- 近年の GPU は飛躍的な勢いで高速化しており，また今後も，プロセッサ数を増大させることが容易であるため，計算能力の拡大が見込まれること．
- GPU で処理を行うことにより，CPU が他の作業に利用できること．
- GPU 上で各種画像処理を実装する事例が増えており，GPU 上で処理を完結させることができれば，GPU-CPU 間のデータ転送を省略でき，高速化がなされること．
- 画像処理結果を提示するためのレンダリング処理との連携が容易となること．

しかしながら，連結領域ラベリング処理の手順中には，逐次的な処理として実装する方が効率的な部分もあり，CPU と GPU との協調が最適であるか，あるいは GPU 上で全ての実装を行うことが最適であるのかについては検討が必要である．本稿では，ラスタベース連結領域ラベリング手法を取り上げ，並列化の効果が見込めるラスタ単位の処理に焦点を絞り，GPU の利用可能性について検討する．

2. 関連研究

連結領域ラベリングは画像処理の基本的な処理単位の一つであり，近傍処理やラン解析によるアルゴリズムが普遍的に利用されている[1)]．本研究で取り上げるラスタベース連結領域ラベリング手法に関しても，多くの研究がなされている[2),3)]．

連結領域ラベリングの並列化は，主にグラフ理論の問題として古くから研究がなされている[4),5)]．画像を対象とした研究としては，Bader らの階層的に領域を結合していく手法[6)] などがあるが，一般的な並列計算機を前提とした議論がなされており，そのまま GPU 上に適用することは困難である．

3. 連結領域ラベリング手法

本研究では，ラスタベース連結領域ラベリング手法を GPU を利用して並列化するアルゴ

†1 大阪大学
Osaka University
†2 奈良先端科学技術大学院大学
Nara Institute of Science and Technology

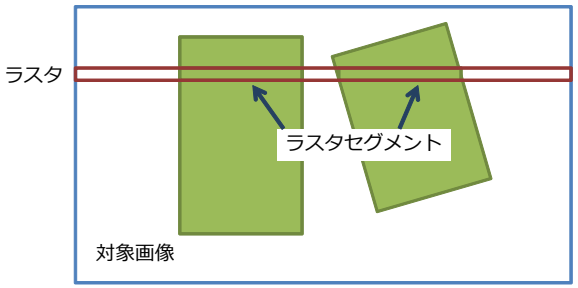

図 1 ラスタおよびラスタセグメント
Fig. 1 Raster and raster segment

リズムを考案する．本節ではラスタベース連結領域ラベリングの手順を紹介する．

3.1 入力画像

処理対象となる入力画像は，前処理により二値化されているものとする．連結領域ラベリングの対象となるピクセルの画素値を 1，背景ピクセルの画素値を 0 とする．以下では入力画像を $I(i,j)$ とし，入力画像の各画素は 0, 1 のいずれかの値を保持しているとして処理手順の解説を行う．

多値への拡張は，連結領域の判定を画素値の一致により行う条件のもとであれば容易である．

3.2 ラスタおよびラスタセグメント

本稿では y 座標が一定であるピクセルの連なり（水平方向の直線）をラスタと呼ぶ．ラスタ $R_j(i)$ は以下のように定義される．

$$R_j(i) = I(i,j) \tag{1}$$

また，ラスタ内の同一画素値の連なりをラスタセグメントと呼ぶ．各ラスタセグメントの位置は，開始（左端）ピクセルの x 座標，終了（右端）ピクセルの x 座標，y 座標の三つの数字で特定できる．

図 1 にラスタおよびラスタセグメントの例を示す．

3.3 処理手順

処理は以下の手順で行う．

(1) ラスタセグメントの抽出
ラスタごとに，画素値が 1 である画素が連続している区間を探索し，ラスタセグメントとして取り出す．
各ラスタセグメントは，以下の情報を保持している．
- 開始 x 座標: x^s
- 終了 x 座標: x^e
- y 座標: y

抽出されたラスタセグメントは，ラスタごとにリストとして保持される．

(2) ラスタセグメントの連結情報の構築
(a) 連結領域番号が未確定のラスタセグメント S を一つ選び出し，既に抽出された連結領域に付与された番号と重複しない連結領域番号 N を与える．
(b) ラスタセグメント S が属しているラスタに隣接している上下のラスタに関して，手順 1 で構築されたリストを探索し，S と隣接している全てのラスタセグメントを取り出す．取り出されたラスタセグメントには S と同じ連結領域番号 N を与え，ラスタ毎のリストから消去し，連結領域のリストに保存する．
(c) 取り出されたラスタセグメントに対して，同様の操作を，取り出すラスタセグメントがなくなるまで続ける．
二つのラスタセグメントが連結しているかは，以下の条件で判断する．
- 4 近傍の場合: $x_1^s \le x_2^e$ かつ $x_2^s \le x_1^e$
- 8 近傍の場合: $x_1^s \le x_2^e + 1$ かつ $x_2^s \le x_1^e + 1$

実装に際しては，探索中のラスタセグメントを格納するためにスタック構造を用いる．

(3) 領域サイズ順の識別番号の付与
各連結領域に属している ラスタセグメントの長さを合計して各連結領域の面積を算出する．面積の降順にソートし，連結領域番号を 1 から順に再付与する．

(4) 出力画像への書き込み
各連結領域ごとに，その連結領域を構成しているラスタセグメントが占めている画素に対して，領域番号を書き込む．

3.4 画像へのアクセス回数

本手法では，ラスタセグメントの抽出の際に入力画像の各ピクセル値を一度ずつ読み出す処理を行うが，これが入力画像へのアクセスの全てである．連結領域ラベリングにおいては，各画素が対象画素であるかそうでないかの判別を行うために，各画素の画素値は最低でも一度は読み込まれなければならないため，本手法の入力画像へのアクセス量は連結領域ラベリング手法として最小である．

また，出力画像へのアクセスについては，連結領域の領域番号が最終的に決定した後で，

各画素への書き込みを行うため，やはり一度で済んでいる．結果の出力処理は必須であるから，出力に関しても必要最低限のアクセス量を達成している．入力画像を上書きしてと出力する場合には背景画素への書き込みが必要なくなるため，更にアクセス回数を削減できる．

以上のように，本アルゴリズムはメモリへのアクセス回数が最小であり，演算と比べてメモリアクセスが低速な今日の計算機に適していると言える．

4. GPU を用いたラスタセグメント抽出処理の並列化

3.3 節で解説した処理手順において，GPU による並列化の恩恵が大きいと見込まれる処理は，手順 1 のラスタセグメントの抽出および手順 4 の出力画像への書き込みである．特に，手順 1 は処理対象が矩形であるため，スレッドに対して処理を均等に割り振ることが容易である．本節では，手順 1 の並列化手法を二つ提案する．

4.1 並列化手法 1: ラスタ単位の並列化

3.3 節で解説した手順 1 はラスタ単位で独立した処理である．よって，各ラスタに 1 スレッドを割り当てることにより，並列化が実現できる．一般的な画像の場合，数百から数千のスレッドが並列に実行されることになる．

4.2 並列化手法 2: ピクセル単位の並列化

並列化において，実行ユニット数に対してスレッド数がある程度以上あれば，メモリアクセスのレイテンシが隠蔽され，大きな速度向上が見込まれる．以下では，各ピクセルに 1 スレッドを割り当てる並列化手法について検討する．

各ラスタセグメントの端のピクセルについて考える．ラスタセグメントの左端ピクセルは，自らのピクセル値が 1 で，左隣のピクセル値が 0 である．同様に，右端は，自らのピクセル値が 1 で，右隣のピクセル値が 0 である．ラスタセグメントを構成するそれ以外のピクセルは，自らのピクセル値および左右画素のピクセル値ともに 1 である．

ここで，ラスタ i の隣接するピクセル値間の差 $d_i(x) = R_i(x) - R_i(x-1)$ を考える（**図 2** c）と，ピクセルがラスタセグメントの左端・右端である条件は，それぞれ以下のようになる．

$$\text{左端} : d_i(x) = 1 \tag{2}$$

$$\text{右端} : d_i(x-1) = -1 \tag{3}$$

よって，各ピクセルに対して，左端であるフラグ，右端であるフラグを，$d_i(x)$ に基づいて立てることができる（図 2 d および e）．

ここで，これらのフラグの prefix sum を考える．prefix sum とは，$x_1, x_2, \ldots, x_n$ に対して，

$$y_i = x_1 + x_2 + \cdots + x_i \tag{4}$$

により計算される y_i のことである．並列アルゴリズムにおける基本的な演算の一つであり，n 個の prefix sum が $O(\log n)$ で計算できることが知られている．図 2 f および g に示すように，各フラグの prefix sum を計算することで，左端・右端である各ピクセルが，自分が左から数えて何番目のラスタセグメントの端であるを知ることができる．

フラグが立っているピクセルが，自らの x 座標を所定の順番のラスタセグメントの左端あるいは右端座標として格納することにより，ラスタセグメントの始点・終点座標を得ることができる．

このアルゴリズムは各ピクセルに対してスレッドを割り当てているため，並列度が非常に高い．将来的により多くのスレッドが同時に実行できる実行環境が現れた場合，特に有用であると思われる．一方，並列化により実行速度が低下する要因として，1 ラスタ=1 スレッドの場合（左端から順次処理を行った場合）にはピクセルごとのメモリアクセスは読み出し 1 回のみであるが，本手法を実装するとメモリアクセス回数が増える点が挙げられる．

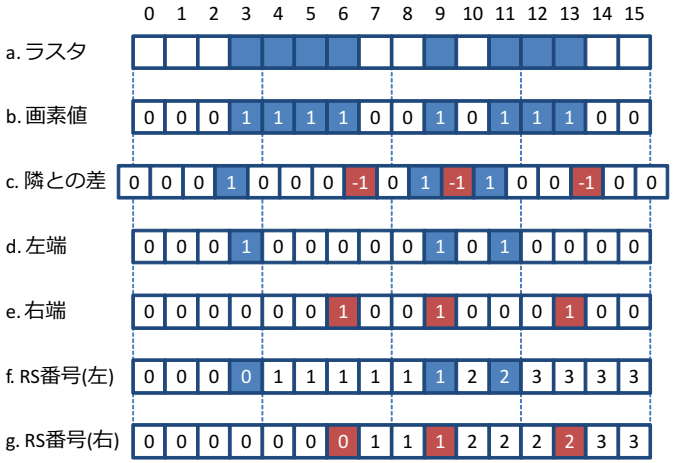

図 2 ピクセル単位の並列化におけるラスタセグメント抽出の流れ
Fig. 2 Process of parallelized extraction of raster segment

Vol.2010-CG-138 No.11
2010/2/12

表 1 実装環境の諸元
Table 1 Properties of computing environment

CPU	Intel Core i7-920（2.66GHz）
RAM	3GB
コア数（スレッド数）	4（8）
GPU	NVIDIA GeForce GTX 285
RAM	1GB
プロセッサ数	240
OS	Microsoft Windows Vista Business SP2（32bit）
開発環境	Microsoft Visual Studio 2008
開発言語	C++
グラフィクスライブラリ	OpenGL 3.2
GPGPU ライブラリ	NVIDIA CUDA 3.0β および CUDPP 1.1

表 2 対象画像
Table 2 Input images

項目	VGA	HD
横ピクセル数	640	1920
縦ピクセル数	480	1080
ラスタセグメント数	8988	37906
領域数	1731	5084

5. 検証実験

本節では，4.2 節で提案した並列度の高いピクセル単位の並列化手法の実効性を検証するために，提案アルゴリズムを GPU 上に実装し，ラスタセグメント抽出の速度を CPU による実装と比較する．

5.1 実装環境

実装環境を表 1 に示す．GPU 上の計算処理の記述には，NVIDIA CUDA を利用した．prefix sum の計算には CUDA を利用した基本的な並列アルゴリズムライブラリである CUDPP を使用した．

5.2 実験手順

表 2 に示す 2 種類のサイズが異なる画像を入力として，ラスタセグメント抽出処理の速度を計測した．

CPU における連結領域ラベリングには，文献 7) にある実装を使用した．ただし，CPU 側でも複数のスレッドによる抽出処理を実行できるように，スレッドによる並列化処理を実装した．実装環境の CPU では 8 スレッド同時実行可能であるため，スレッド数が 1, 2, 4, 8, 16 の 5 通りの場合について検証した．

また，CUDA による GPU 上の実装においては，共有メモリによるデータ共有が可能なスレッド群をブロックとして扱うが，1 ブロックあたりのスレッド数に上限がある．本実験では，1 ブロックあたりのスレッド数が 96, 128, 192, 256, 384 の場合について検証した．

所要時間の計測には，CPU 版は Microsoft Visual Studio が提供する timeGetTime() 関数を，GPU 版は CUDA Utility Library として NVIDIA から提供されているタイマ関数群を用いた．

(a) (b)

図 3 連結領域ラベリングの実行例 (a) 入力画像 (b) 実行結果
Fig. 3 Result of connected component labeling (a) input (b) result

5.3 結果

図 3 に入力と出力の例を示す．入力画像は白が処理対象画素，黒が背景画素である．出力は色相の異なる領域が各連結領域を示している．

それぞれの場合において，ラスタセグメント抽出処理に要した時間を**表 3** および**図 4** に示す．所要時間は 100 回の計測結果の平均である．また，GPU での実行の際に，画像の転送に要した平均時間は，画像サイズが VGA の場合 1.89msec，HD の場合 2.60msec であった．

5.4 考察

GPU によるラスタセグメント抽出処理は，対象となる画像が大きい方が CPU と比較して高速に実行できており，並列性が高くなるほど GPU による処理が適していることが推測される．今後，一般に扱われる画像のサイズがハイビジョンから 4K 画像へと一層向上して

表 3 ラスタセグメント抽出に要した時間
Table 3 Elapsed time for extraction of raster segments

CPU			GPU		
スレッド数	所要時間（msec）		スレッド数	所要時間（msec）	
	VGA	HD		VGA	HD
1	1.83	8.12	96	1.68	3.36
2	1.15	4.61	128	1.61	3.27
4	0.91	2.74	192	1.67	3.35
8	1.16	2.56	256	1.62	3.30
16	1.87	2.94	384	1.68	3.46

いくであろうことを考えると，GPU による高速処理を追求していくことに一定の価値を認めることができる．

CPU から GPU への画像の転送時間を加味すると，GPU での処理時間は並列化された CPU に劣ることになるが，CPU で画像処理を行っても最終的な結果として得られた画像を提示する際に GPU 側へ転送しなければならないことを考えると，転送によるオーバーヘッドはアプリケーション全体としては不利な点とは言えない．

6. おわりに

本稿では，画像処理において欠かすことのできない連結領域ラベリング処理を，GPU 上で実現することを目的として，最も並列化の効果が見込めるラスタセグメント抽出処理に関してピクセル単位の並列アルゴリズムを提案した．CPU と GPU による実装をそれぞれ行い，実行時間を比較した結果，GPU 上での連結領域ラベリング処理が将来において実用的なものになるであろうことが示唆された．今後は，ラスタセグメントの抽出だけでなく，処理の残りの部分についても GPU 上で実現するためのアルゴリズムの開発を実施していく．

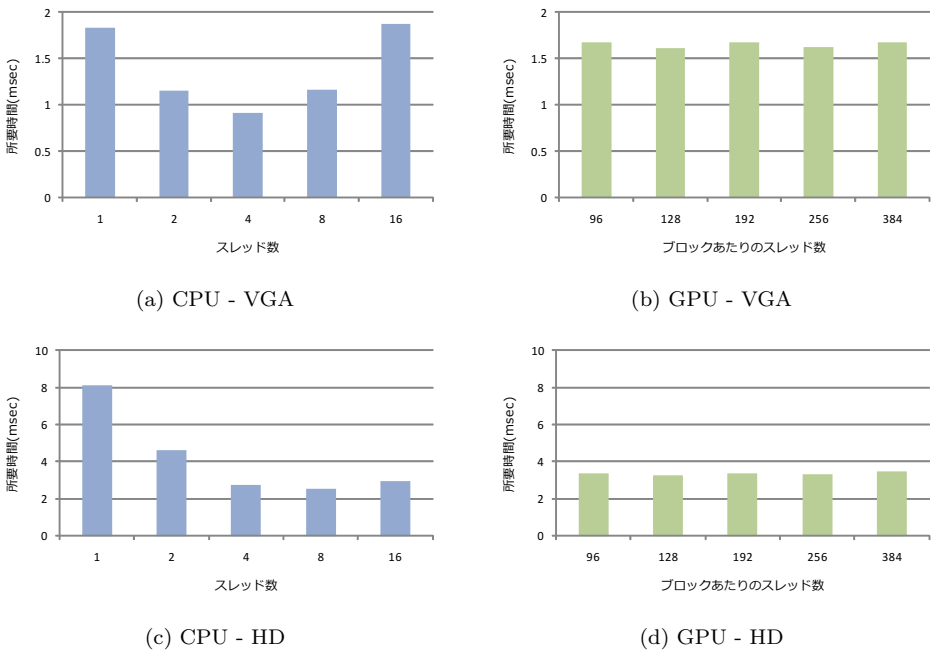

図 4 所要時間の比較
Fig. 4 Comparison of elapsed time.

参 考 文 献

1) 田村秀行（編）：コンピュータ画像処理，オーム社 (2002).
2) Shapiro, L.G. and Stockman, G.C.: *Computer Vision*, Prentice Hall (2001).
3) Suzuki, K., Horiba, I. and Sugie, N.: Linear-time connected-component labeling based on sequential local operations, *Comput. Vis. Image Underst.*, Vol.89, No.1, pp.1–23 (2003).
4) Quinn, M.J. and Deo, N.: Parallel graph algorithms, *ACM Comput. Surv.*, Vol.16, No.3, pp.319–348 (1984).
5) Hirschberg, D.S.: Parallel algorithms for the transitive closure and the connected component problems, *STOC '76: Proceedings of the eighth annual ACM symposium on Theory of computing*, pp.55–57 (1976).
6) Bader, D.A. and JáJá, J.: Parallel algorithms for image histogramming and connected components with an experimental study, *PPOPP '95: Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pp. 123–133 (1995).
7) 井村誠孝：ラベリングクラスについて.
http://oshiro.bpe.es.osaka-u.ac.jp/people/staff/imura/products/labeling/labeling.pdf.