

GPGPU 向け 2 値画像の同時連結成分抽出方式の提案と実装

木網啓人^{†1} 佐藤裕幸^{†1}

概要: 連結成分抽出は画像処理において最も基本的なアルゴリズムの一つであり、古くから様々なアルゴリズム、アーキテクチャでの実装が提案されている。連結成分抽出処理は、同じ値を持つ連結画素を抽出する技術である。その主な用途は、光学画像を用いた製品の異常診断に製品やノイズ領域を抽出する用途で使われる。特に、2 値化後のノイズ除去処理では、本来製品である領域を抽出するために背景成分の連結成分を抽出する。その後、前景成分の連結成分を抽出し、診断に利用する。本稿では、穴埋め処理における 2 値画像の前背景成分を同時に抽出できる手法について、GPGPU 向けの並列アルゴリズムとその実装を提案する。NVIDIA 社製の組込み SoC である Jetson TX2 上で処理時間を計測したところ、GPU 実行時に前景、背景抽出を別々に行う従来手法に比べ提案手法の方が約 5-22% 程度の性能改善を確認した。また、従来手法の CPU 実行時に対し、提案手法の GPU 実行により約 1.15-2.4 倍の高速化を実現した。

キーワード: 連結成分抽出, ラベリング, GPGPU

Proposal and implementation of simultaneous connected-component labeling of the binary images for GPGPU

HIROTO KIZUNA^{†1} HIROYUKI SATO^{†1}

Abstract: Connected-component labeling is one of the most fundamental algorithms in image processing, and implementation with various algorithms and architectures has been proposed for a long time. The connected component extraction process is a technique for extracting connected pixels having the same value. It is mainly used for extracting products and noise areas for abnormality diagnosis of products using optical images. Especially, in the noise removal processing after binarization, in order to extract a region which is originally a product, connected component of the background is extracted. After that, the connected component of the foreground component is extracted and used for diagnosis. In this paper, we propose a parallel algorithm and its implementation for GPGPU, which can extract simultaneous fore-background components of binary image in filling a hole process. We measured the execution time of the proposed method on Jetson TX 2, an embedded SoC made by NVIDIA Corporation, and the performance of the proposed method is improved by about 5 - 22% compared with the conventional method. The execution time of the proposed method on the GPU was about 1.15-2.4 times faster than the execution time of the conventional method on the CPU.

Keywords: Connected-component labeling, GPGPU

1. はじめに

連結成分抽出 (Connected-Component Labeling) [1]は画像処理の中で最も重要なカーネルの一つであり、主に製造ラインの製品欠陥の診断や CT 画像を用いた医療画像診断、農作物の病害虫診断や収穫量推定など、画像を用いた診断に使われる。連結成分抽出は入力した 2 値画像の前景の連結成分を抽出する。これは画像に写った物体やノイズ領域の抽出、応用的な用途として、物体領域内に非物体領域が存在する穴あきを埋める処理 (以下、穴埋め処理) にも用いられる[2][3]。この穴埋め処理では、面積の小さい非物体連結成分を物体連結成分に連結することで、ノイズ除去済みの物体領域を抽出できる。

連結成分抽出処理はガウシアンフィルタ等の画像処理カーネルと比べると計算量が高く、組込み機器でのリアルタイム処理においてボトルネックになる可能性が高い。更に、一般的な連結成分抽出アルゴリズムは、単純な並列性

が無く、更に複雑で多数の分岐命令を必要としており、CPU のマルチコアのパイプライン処理や GPU 等のメニーコアプロセッサの SIMD (Single Instruction Multiple Data stream) アーキテクチャに適しているとは言いがたく、単純な並列化による性能改善が見込めない。穴埋め処理においては、前背景を順番に連結成分抽出を行う必要があり、より長い処理時間を必要とする。また、穴埋め処理は連結成分抽出を前背景それぞれ合わせて 2 回、全画素アクセスを 2 回行う必要がある。しかし、前景の連結成分抽出時であれば背景画素、背景時は前景画素に対して処理をする必要がないため、GPU の様な SIMD 演算実行時にアイドル状態に成るスレッドが多くなることが想定され、リソースのアイドル時間を高め、実行性能の劣化を招く要因に成り得る。

本研究では、前背景同時ラベリングアルゴリズムの提案と実装評価を行った。柴田らが提案した GPGPU 等のメニーコアアーキテクチャ向け伝播型連結成分抽出アルゴリズム[4][5]をベースに、著者らが提案した GPGPU 向けラベル整理アルゴリズム[6]を活用する。また、工場のラインや車

^{†1} 岩手県立大学
Iwate Prefectural University

載用途で用いられることを想定し、実装環境は NVIDIA 社が提供する Jetson Tegra X2 (以下 TX2) を用いた。これは GPU を搭載しながら小型かつ軽量の組み向け SoC である。本研究では、TX2 で並列化による処理速度や実行効率の検証を行った。

2 章ではモバイル系 GPGPU についての背景及び概要について記載し、3 章では連結成分抽出処理の先行関連研究及び、一般的な手法の概要について記載する。4 章では GPGPU 向け前背景同時連結成分抽出処理についての提案手法を説明する。5 章では検証結果を示し、6 章で本稿のまとめを記載する。

2. モバイル向け GPGPU 環境

SoC (System-on-a-Chip) はスマートフォンやタブレット PC をはじめとするモバイル端末用のプロセッサとして用いられ、その性能は著しい発展を遂げている。2017年3月、NVIDIA 社は TX2 と呼ばれる、同社製 GPU が一体となった SoC を発表した[7]。図 1 左が本体基板であり、50x87mm 四方と名刺サイズの非常に小さい基板に CUDA コアを 256 基搭載しており、GPGPU 技術に対応している。この TX2 を搭載した NVIDIA Jetson TX2 Developer Kit (図 1 右) が発



図 1 Jetson TX2 モジュールと開発キット
Figure 1 Jetson TX2 Module and Developer Kit

■ 前景画素 □ 背景画素

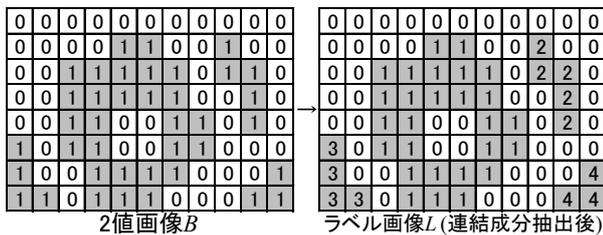


図 2 連結成分抽出例

Figure 2 Example of Connected Component Labeling

■ 前景画像 □ 背景画像

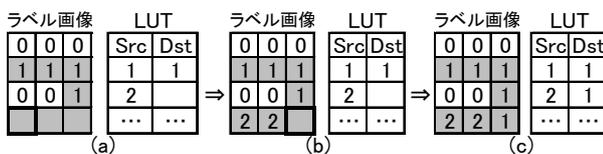


図 3 画素アクセスパターン

Figure 3 Pixel Access pattern

売された。これは、TX2 Module から LAN ポートや HDMI など標準的な IO を搭載した開発ボードであり、\$599 (日本では 9.4 万円) で販売されている。この TX2 は、小型で軽量、少消費電力でありドローン等に積載することが可能である。

3. 利用技術及び先行・関連研究

本章では、本稿での提案手法に用いる連結成分抽出技術についての概要と一般的な処理手法、GPGPU 上での高速化事例について紹介する。

3.1 連結成分抽出の概要

連結成分抽出とは、図 2 のような True (1) と False (0) の 2 値画像を入力とし、どちらかが連続している画素群に対してユニークな ID を付加する処理である。

連結成分抽出処理の共通手順について説明する。連結成分抽出処理の入力は 2 値画像を想定し、出力はラベル画像であり、これは連結成分画素にはラベル ID (1 以上の値) を背景画素には 0 (=背景ラベル ID) を要素に持つ。

この連結成分抽出処理アルゴリズムには大きく分けて、ラスタ型と伝播型の 2 つが存在する。なお本研究では上下左右の 4 近傍連結成分抽出を前提とし、以下にその説明を記載する。

3.2 ラスタ型アルゴリズム

ラスタ型は 2 値画像の左上端から右下方向に逐次走査し、注目画素にラベル ID を貼る。注目画素の上と左の 2 近傍画素に前景画素が存在すれば、その中の最小のラベル ID を貼り、前景画素が存在しなければ、新たなラベル ID を貼る。ラスタ型では前景画素に例えばコの字のようなパターンを入力すると 1 回の走査で正しいラベル付けができない。これを図 3 で説明する。(a)の左下の画素 (太枠) を注目画素とすると、ラベル 1 が貼られた画素と連結しているのにも関わらず、上と左の 2 近傍が前景画素ではないため、新たなラベル 2 が貼られる。そして、中下の画素は左画素がラベル 2 なので、2 が貼られる。そして、右下の画素は、上側の画素がラベル 1、左側の画素が 2 なので (図中の(b))、小さい方のラベル 1 が貼られ、ラベル 2 と 1 は連結していることを記憶するために、Look-Up Table (以後、LUT) へそれが記録される (図中の(c))。

このようにラスタ型では、すべての画素へのラベル付けが完了した後、LUT を用いて再度画像を走査してラベルを書き換える必要があるが、基本的に 2 回の走査で処理を完結する。そのため、次に説明する伝播型より相対的に計算量は低いが、左上端から右下方向に逐次走査するため単純な並列性はない。

ラスタ型アルゴリズムは古くから数多くの研究がなされており、概要を説明する。

鈴木らが提案した 4 回走査型連結成分抽出[8]は、ラベル付けを左上から順に探索する正方向と右下から探索する逆

方向探索を行うことで幾何学形状に依存せず、画像サイズに線形比例するアルゴリズムを提案した。Wu らが提案した 2 回走査連結成分抽出[9]は、1 回走査時に貼られた複数の暫定ラベル同士の連結情報を記録し、Union-find により大域的なラベルの連結関係を変更する。最後 2 回走査時に連結関係情報を用いラベルを書き戻すことで処理を完了する。He らが提案した連を利用した 2 回走査連結成分抽出[10]は、走査時に画素単位でラベルの同一性を評価するのではなく、ラスタごとの連結成分単位ごとに評価することでメモリアクセスを削減している。

3.3 伝播型アルゴリズム

柴田らが提案した伝播型アルゴリズム[4][5]は CPU のマルチコア及び GPU のような SIMD で動作することを前提に設計されている。伝播型連結成分抽出処理は、初期化、近傍探索、ラベル ID 更新を画素ごとに並列に行い、加えて著者らが提案した、連続ラベル ID の書き換え処理により完了する。これら 4 つの処理について説明する。

初期化処理ではラベル ID を保存するラベル画像 (以後ラベル画像) の各画素に配列インデックスを格納する。このインデックスは画素座標が(x,y)、画像幅が width とすると、 $y \cdot \text{width} + x$ により算出される。また、簡単のためにラベル画像周囲 1 ピクセルは 0 (=背景ラベル ID) を格納する。

Algorithm1 は近傍画素の前景判定と最小ラベル ID 判定を行う。伝播処理を **Algorithm2** に示す。3 から 6 行目で最小ラベル ID を探索し、8 から 11 行目でその最小ラベル ID をラベル画像の当該画素に格納し、伝播処理を完了する。

Algorithm2 の 3 から 6 行目にある { up(), left(), down(), right() } は引数 p の相対的な近傍インデックスを返す関数であり、それぞれの対応を図 4 (a) に示す。また、9 から 10 行目にある atomic はアトミック演算を示す。atomic 演算とは、複数のスレッドがあるグローバル変数等の共有変数に対して同時に演算する際に、必ずある単一のスレッドが演

算を終了するまで、他のスレッドがその変数に対して読み書きを禁止するものである。atomic 演算の中に min や max などの関数があるが、これも単一のスレッドが大小判定をもとに変数へ値を格納する。これを擬似的に記述したものが 9 から 10 行目にある。atomic 演算の範囲内で < (代入) することでアトミック演算の結果を取得する。

Algorithm2 の処理を並列に実行する場合、近傍の最小ラベル ID が最初の 4 近傍参照時より注目画素の更新後の方が小さい可能性がある。よって、探索した最小ラベル ID を再度ラベル画像で連想参照探索することで、より遠くのラベル ID を得られ、伝播速度が高まる。この伝播処理を複数回実行することで各連結成分の画素群が同一のユニークなラベル ID に置き換わり、連結成分抽出を完了する。この伝播処理では、最終的なラベル画像は図 4 (b) のように各連結成分の全画素の中で最小のラベル ID (=1 次元配列インデックス) がユニークなラベル ID として伝搬される。よって伝播後のラベル ID の最大値は入力画像の画素数に成り得る。

No.	Algorithm1 search_neighbor(t, n)
1	return (n ≠ 0 and t > n)? n : t;

No.	Algorithm2 propagate(p)
1	o ← L[p];
2	if o ≠ 0
3	g ← search_neighbor(L[p], L[up(p)]);
4	g ← search_neighbor(L[p], L[left(p)]);
5	g ← search_neighbor(L[p], L[down(p)]);
6	g ← search_neighbor(L[p], L[right(p)]);
7	g ← L[L[L[g]]];
8	if g ≠ 0
9	atomic{ L[o] ← min(L[o], g); }
10	atomic{ L[p] ← min(L[p], g); }
11	end if
12	end if

本研究の応用アプリケーション実装では、連結成分の画素数 (=面積) 算出に用いる。この面積算出処理は、いわゆるヒストグラム計算のような処理であり、ラベル ID をキーに記録用配列へインクリメントを行うことが考えられる。連結成分抽出後は図 4 (b) のようにラベル ID が 1 から連続していないが、Key-Value ストアのような辞書型のデータ構造を使えば効率的に面積算出処理を実装できる。また、辞書型データ構造を持たない処理系で実装する場合、ラベル ID をキーに配列要素をインクリメントすることで面積を算出する。しかしこの実装では、非連続なラベル ID をキーに配列へアクセスするので、アクセス局所性を満たせないため、キャッシュヒット率が著しく低下し、更に未使用なメモリ領域が多く、メモリ効率も低い。また、このヒストグラム算出を GPU 上で並列に実行する場合、並列実行数と画素数を乗算した数の共有メモリ領域が必要であり実用的でない。

そこで、連続でないラベル ID を 1 オリジンの連続した

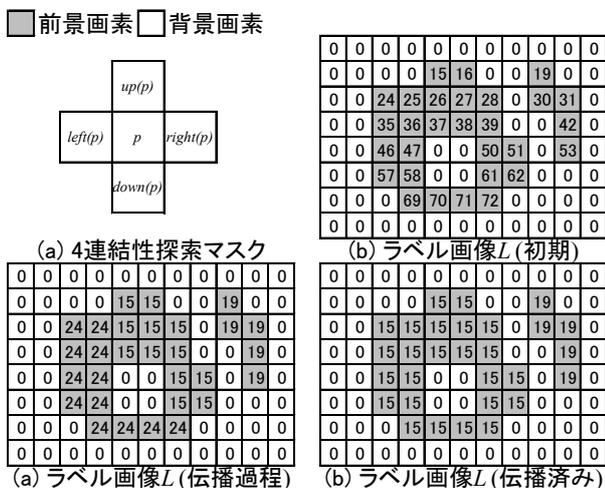


図 4 伝播型連結成分抽出例

Figure 4 Propagation Type Connected-Component Labeling

ラベル ID へ書き換えることが考えられる。これを **Algorithm3** に示す。新たに出てくる T は各成分の非連続ラベルと連続ラベルの対応を示すものであり、いわゆる LUT (Look Up Table) のようなものである。T は画素数分の要素を持つ 1 次元配列であり、インデックスは非連続ラベルで要素値には連続ラベル ID が格納される。変数 i は最後に取得されたラベル ID を保持しており、この変数に対してインクリメントすることで新たな連続 ID を取得できる。この処理には単純な並列性がないため、逐次的に更新することになり、GPU の数千コアの内 1 コアしか使うことができないため、処理時間が長く、実行効率も極めて低く、GPGPU に適さない。

No.	Algorithm3 <i>rewrite_label_in_sequential(p)</i>
1	<i>if</i> $L[p] \neq 0$
2	$l \leftarrow T[L[p]]$;
3	<i>if</i> $l < 0$
4	$u \leftarrow ++i$;
5	$T[L[p]] \leftarrow u$;
6	<i>end if</i>
7	<i>end for</i>

そこで著者らは画素単位の並列性を持った連続ラベルへの書き換えアルゴリズム[6]を提案した。そのアルゴリズムを **Algorithm4** に示す。画素ごとにスレッドを生成した場合のアルゴリズムの概要は、まず、各成分の暫定ラベルを連続したユニークなラベルに書き換える代表スレッドを選

■ 前景画素 □ 背景画素

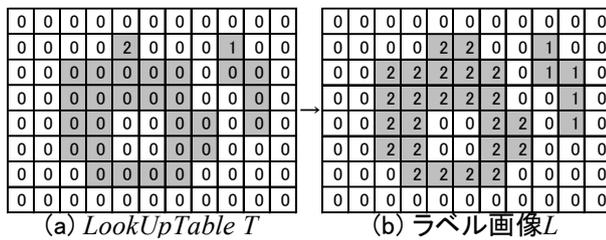


図 5 ラベル書き換え用 LUT 及び連続ラベル画像
Figure 5 LUT for Rewriting to Continual Label ID and Continual Label Image

□ 前景画素 ■ 背景画素

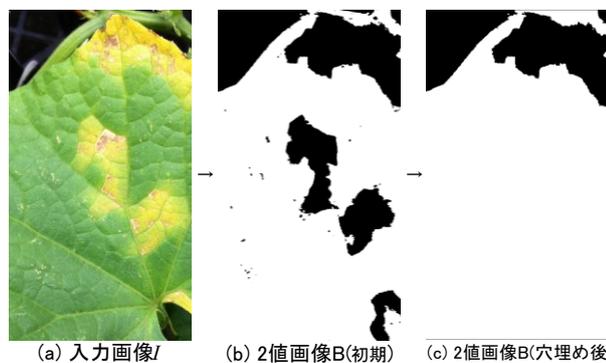


図 6 2 値画像での穴埋め処理
Figure 6 Filling Holes with Binary Image

出する。その代表スレッドは連続ラベル ID を取得し、LUT へ格納する。最後この LUT を使いラベル画像を連続ラベルに書き換える。

代表スレッドの選出方法を説明する。T の初期値は 0、連続ラベル ID は 1 以上の値になる。よってある成分を担当するスレッド群が暫定ラベル ID を用いて LUT へアクセスし、要素値が 0 であれば、まだ連続ラベル ID を取得していないことがわかる。次に、2 行目にあるようにアトミック演算で LUT の要素値から 1 を減算した結果が -1 であれば、そのスレッドは初めてアトミック演算したことがわかる。したがって、演算結果が -1 であったスレッドを代表スレッドとする。また、1 行目で LUT を参照した結果が 0 であったすべてのスレッドは当該要素値から 1 を減算しているため、6 行目のように代表スレッド以外も 1 を加算する。次にユニークな連続ラベル ID の取得方法を説明する。代表スレッドは、変数 i に対しアトミック演算によりインクリメントすることで連続ラベル ID を取得でき、それを T に格納することで LUT を更新する。この結果を図 5 (a) に示す。最後にラベル画像を更新した LUT を用いて書き換えることで、ラベル画像は連続ラベル ID に書き換えられる。最終的な連続ラベル画像を図 5 (b) に示す。

Algorithm5 は連番ラベル画像の生成に至るまでの処理フローである。伝搬処理は対して複数回実行することで伝搬を完了する。また、少ない伝搬回数で最小ラベルを伝搬させるために、全画素に対する伝搬処理を終えるごとにグローバル同期を取る。

No.	Algorithm4 <i>set_uniqid_to_lut(p)</i>
1	<i>atomic</i> { $v = T[L[p]] - 1$ };
2	<i>if</i> $v = -1$
3	<i>atomic</i> { $u = ++i$ };
4	<i>atomic</i> { $T[L[p]] \leftarrow T[L[p]] + u + 1$ };
5	<i>else</i>
6	<i>atomic</i> { $T[L[p]] \leftarrow T[L[p]] + 1$ };
7	<i>end if</i>

No.	Algorithm5 <i>execute_CCL()</i>
1	<i>preparation</i> (p);
2	<i>syncthread</i> s();
3	<i>for</i> idx <i>in</i> <i>range</i> (MAX_NUM_LOOPS):
4	<i>propagate</i> (p);
5	<i>syncthread</i> s();
6	
7	<i>lut_update</i> (p);
8	<i>syncthread</i> s();
9	<i>label_rewrite</i> (p);

3.4 背連結成分を用いた穴埋め処理

本節では画像処理による葉の病気診断を題材に穴埋め処理を図 6 で説明する。(a) は入力画像、(b) は 2 値画像で、葉の色情報をもとに (a) を閾値処理した結果である。穴埋め処理は、入力画像から閾値処理で抽出出来なかった葉領域を補完するために用いられる。閾値は葉の正常な緑色の特徴を使い実験的に設定しているが、(a) の葉領域には病気領域も存在しており、色に変色している。そのため (b) のよう

に病気領域を葉領域として抽出することが出来ない。(b)を見ると、葉領域の内側にある病気領域は穴のような背景領域として判定されていることがわかる。そこで、病変領域の穴のような背景領域を前景化する穴埋め処理を用い、それを抽出する。その結果が(c)である。この穴埋め処理の具体的な処理手順を以下に示す。

Step1. 背景領域の連結成分抽出

背景領域の連結成分抽出により穴領域を抽出する。

Step2. 背景成分の面積ヒストグラム生成

各背景成分の画素数(=面積)のヒストグラムを算出する。GPGPU 向けヒストグラム算出アルゴリズムには[11]を用いる。この並列アルゴリズムは、入力となるラベル画像を分割し、少領域ごとにヒストグラムを算出し、最後1つのヒストグラムに統合する。この統合処理は、分割数分の各ヒストグラムビンの値を集計加算することで算出する。この集計処理の並列アルゴリズムにリダクション[12]があり、2つの要素を1つの組み合わせとし、それぞれの組を1スレッドが計算することで、総数 N 個の要素の計算を logN 回の演算を行えば集計できる。これに加え、Mark らが提案した、アトミック演算を用いたリダクション手法を用いる。これは新たな GPU アーキテクチャに最適化されており、より高速な集計処理を実現している。

Step3. 小さい背景成分の前景化

各背景成分面積で小さい成分を閾値により抽出し、前景化すると共に2値画像の穴領域を書き換える。

4. 提案手法

本章では、GPGPU 向け前背景同時連結成分抽出処理と穴埋めアルゴリズム処理を提案する。それぞれの手法を説明する。

4.1 前背景同時連結成分抽出

大枠は3.3で述べたアルゴリズムと同様である。よって、大きく異なるアルゴリズムのみ抜粋して説明する。

初期化処理では、前背景同時に連結成分抽出するため、ラベル画像周辺1pxを除くすべての画素に対して1次元配列インデックスを格納する。前背景同時伝搬処理における伝播処理をAlgorithm6-7に示す。相違点は大きく2つある。1つ目に、Algorithm2では当該画素が背景であれば周辺画素の連結判定を行わなかった。本処理では前景と背景のどちらも連結成分抽出を行うため、すべての当該画素において周辺画素との連結判定を行う。2つ目は、当該画素と周辺画素に対応する2値画像の値が同一かどうか判定する。この判定には高速化のためにXORを用いる。これは入力値が同一だと必ず0になる性質を利用している。

最後に、連続ラベルIDへの書き換え処理をAlgorithm8に示す。これも前背景両方向同時に書き換えるので、全ラベル画像の画素がLUTのIDの有無についてチェックする。もしLUTの当該ラベルが初期値であれば、各成分の代表ス

レッドに選ばれたスレッドは新規ラベルIDを取得し、格納する。

3.3節では最後に、更新したLUTを用い、ラベル画像のIDを書き換えている。本穴埋め処理では前背景ごとに連番ラベルに書き換えても、穴埋め処理で連結成分を統合することで、再度連番ラベルIDへ書き換える必要があり、低速かつ無駄な計算である。よって、穴埋め処理に用いる前背景同時連結成分抽出では、LUTを連番のラベルIDへ更新するとともに、ラベルの更新は、穴埋め処理のすべての連結成分を統合後に書き換えることとした。また、前背景の判定は2値画像により判定可能なので、LUTに対して格納する連続ラベルIDは前背景に関係なくユニークなIDに書き換えている。

No.	Algorithm6 search_neighbor2(bt, t, bn, n)
1	return (bt^bn = 0 and t > n) ? n : t;

No.	Algorithm7 propagate2(p)
1	o <- L[p];
2	g <- search_neighbor2(B[p], L[p], B[up(p)], L[up(p)]);
3	g <- search_neighbor2(B[p], L[p], B[left(p)], L[left(p)]);
4	g <- search_neighbor2(B[p], L[p], B[down(p)], L[down(p)]);
5	g <- search_neighbor2(B[p], L[p], B[right(p)], L[right(p)]);
6	g <- L[L[L[g]]];
7	if g ≠ 0
8	atomic{ L[o] <- min(L[o], g); }
9	atomic{ L[p] <- min(L[p], g); }
10	end if

No.	Algorithm8 update_lut2(p)
1	if T[L[p]] = 0
2	atomic{ v = -T[L[p]]; }
3	if v = -1
4	atomic{ u = ++i; }
5	atomic{ T[L[p]] <- T[L[p]] + u; }
6	else
7	atomic{ T[L[p]] <- T[L[p]] + 1; }
8	end if
9	end if

4.2 前背景同時連結成分を用いた穴埋め処理

本節では、4.1節で提案した前背景同時連結成分抽出結果を用い、3.4節の穴埋め処理実装について提案する。説明のために、前背景のラベルの関係を簡略化した図7を用いて説明する。(a)は4.1節の前背景同時連結成分抽出処理を行ったラベル画像であり、これを穴埋め処理の初期値とする。(a)から(d)にあるLを接頭辞とするインデックスは各成分のラベルIDである。また、わかりやすさと説明の簡略化のためにラベルIDは連番とする。なお、この穴埋め処理は、すでに前背景の連結成分抽出済みのラベル画像に対して行う。穴埋め対象成分は背景成分であるが、穴埋めを行う近傍成分は必ず前景成分となることで前景成分のラベルIDを統合する必要がある。以降に、アルゴリズムとそれらの必要性について示す。

Step1. 前背景成分の面積算出及び小さい背景成分抽出

■ 前景画素 □ 背景画素 フォーマット: 現ラベル(前ラベル)

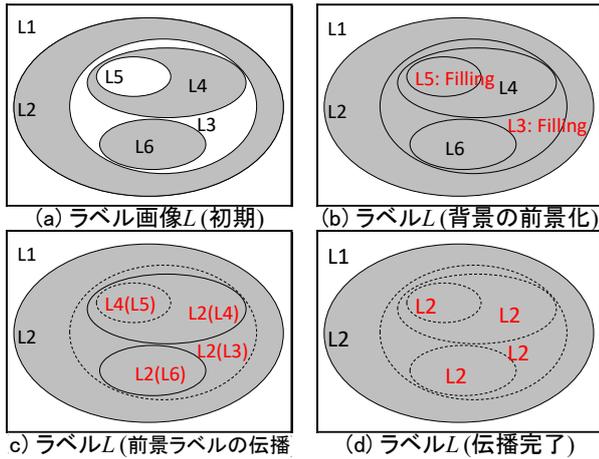


図 7 小さい背景成分の前景化処理

Figure 7 Foregroundization of Small Background

Components

各前背景成分の画素数のヒストグラムを算出する. 3.4 節と異なり, 前背景成分両方のヒストグラムを算出している. これは, 4.1 節の最後, 新たに前背景で共通した連続ラベル ID を対応させるためである. ラベル画像にあるラベル ID にこの LUT の連続ラベル ID テーブルを用い, 連続ラベルをキーに, ヒストグラム算出する. この際, 背景のみのヒストグラムを計算することも可能だが, 2 値画像を参照した上で, 条件式を用い背景ラベル判定をする必要がある. 条件式は演算器の多段パイプラインでストールを引き起こし, SIMD 演算では複数のデータに対して同一の処理を行えないため, 速度性能の劣化の要因に成り得る. そのため, 前背景ヒストグラムを算出する.

次に各背景成分面積が閾値以下であれば当該成分の統合フラグに TRUE を格納する. (b)では, 穴埋めする背景成分に”Filling”と記載しており, {L3, L5}である.

Step2. 小さい背景成分の前景化

次に各画素の前背景情報を記録する 2 値画像を書き換える. 各ラベルの統合判定結果と 2 値画像は同じ型で格納されている. したがって, 2 値画像と各ラベルの統合判定結果とのビット演算により, 前背景が記録されている 2 値画像へ前景画素を書き加えることが可能である.

Step3. 前背景ラベルの統合探索

Step2 において(b)の L3 と L5 ラベルは背景から前景化したことによりラベル L2-L6 はすべて同一の連結成分となる. しかし, 各成分はそれぞれ異なるラベル ID を持っており, これら同一成分の非同ーラベルを同ーラベル ID に統合する必要がある. ここでは同一成分の非同ーラベルの統合手法を提案する. アルゴリズムを Algorithm9-10 に示す. 基本的に 3.3 節の伝播アルゴリズムを応用しており, 並列に実行しても正しい結果に成るよう設計されている.

Algorithm10 で 4 近傍分の近傍前景成分探索を行い,

Algorithm9 はその探索アルゴリズムである. 当該成分が前景であり, 尚且, 当該ラベル ID より近傍ラベル ID の方が小さければ, 小さい近傍ラベル ID を当該ラベル ID とする. ラベル ID の参照と書き換えには LUT を用いる. このアルゴリズムでは前景ラベル ID を伝搬させる. (c)に伝播過程を(d)に伝播を完了した統合結果 LUT 概要図を示す. また(c)のラベル ID を伝搬した成分には, 伝播前と伝播後のラベル ID を記載している.

No.	Algorithm9 Search Minimum Foreground: smf(t, bn, n)
1	return bn and T[t] > T[n]? n : t;

No.	Algorithm10 integrate background to foreground(p)
1	if B[p]
2	g <- smf(L[p], B[up(p)], L[up(p)]);
3	g <- smf(L[p], B[left(p)], L[left(p)]);
4	g <- smf(L[p], B[down(p)], L[down(p)]);
5	g <- smf(L[p], B[right(p)], L[right(p)]);
6	if T[L[p]] ≠ T[g]
7	atomic{ T[L[p]] <- min(T[L[p]], T[g]); }
8	end if
9	end if

Step4. 新たな連続したラベル ID 対応情報の生成

Step3 で伝播処理により LUT の前景成分のラベル ID を統合した. この LUT には前景成分ラベル ID の同一性が記録されているが, 統廃合された連結成分のラベル ID は非連続な番号に成っている. これは, Step2 の背景の前景化により必ず近傍の前景成分に連結し, その連結により元々の前景成分も統合されるため, 必ず穴埋め処理後の成分数は前背景連結成分抽出時の総ラベル数より少なくなる. 更に各連結成分を統廃合すると, 各連結成分の中で最も小さいラベル ID へ統合される. この各連結成分に最小のラベル ID が各成分で連続して含まれることは保証されない.

そのため, ここで新たに統合後の前景成分のラベル ID を連続したラベル ID へ書き換えるための LUT を生成する. このアルゴリズムは 3.3 とほぼ同一であり, Algorithm11 に示す. T は連結成分抽出によって得られたラベル ID と連続ラベル ID の同一性が記録されており, ここで新たに出てきた T'は, 統廃合した元連続ラベル ID と前景成分への新たな連続ラベル ID の対応を記録する. なお, 3.3 での連続ラベル ID の取得と違い, 本処理では前背景連結成分抽出時の総ラベル数分のラベル更新チェックをすれば十分である.

No.	Algorithm11 set_new_uniqid_to_lu(l)
1	if T'[T[l]] = 0
2	atomic{ v = -T'[T[l]]; }
3	if v = -1
4	atomic{ u = ++i; }
5	atomic{ T'[T[l]] <- T'[T[l]] + u; }
6	else
7	atomic{ T'[T[l]] <- T'[T[l]] + 1; }
8	end if
9	end if

Step5. 統合した各前景成分への連続したラベル ID の付加

Step4 で生成した T を用い、ラベル画像を新たな連続ラベル ID で書き換える。

これらの手順を踏むことで小さい背景成分と前景成分を統合し、連続したラベル ID に書き換わったラベル画像を得られ、穴埋め処理を完了する。また、3.4 と同様のラベル画像を得られる。

5. 検証評価

評価では、従来手法と提案手法による前背景同時連結成分抽出による穴埋め処理の処理時間の高速化率と計算効率について検証した。GPGPU 実装の検証には NVIDIA 社の Jetson TX2 を利用するため、同社が提供する開発環境である CUDA (Compute Unified Device Architecture) で実装した。

5.1 検証環境

実行環境は、前述の TX2 上で CPU 及び GPU 上で検証した。表 1 に検証環境を示す。3.2 節と 3.3 節で述べた 2 方式の連結成分抽出処理を CPU 上で実装したところ、伝播型よりラスタ型の方が高速だった。そのため CPU 上での穴埋め実装には、ラスタ型連結成分抽出を採用した。

表 1 実験環境

Table 1 Experiment Environment

Jetson TX2	
CPU	Cortex-A57@2GHz 4C
# GFLOPS	16.0
GPU	Pascal@1.3GHz 256C
# GFLOPS	665.6
OS	Ubuntu 16.04 LST

5.2 前景成分ラベルの伝播とその検証

本稿での提案手法では穴埋め処理を前景成分の統合によって実現する。この統合手法は 4.2 で述べたとおり連結成分内での最小ラベル ID を伝播させることで完了させる。しかし、この処理は図 8 のような前背景成分が入れ子になった画像において、最外または最内入れ子のラベル ID が最小である場合、入れ子の前景成分数を N とすると、すべての連結前景成分を統合させるために N-1 回伝播する必要がある。逐次実行であれば画像の左上から右下へのラスタスキャンになるため、必ず 1 回の走査で伝播を完了する。

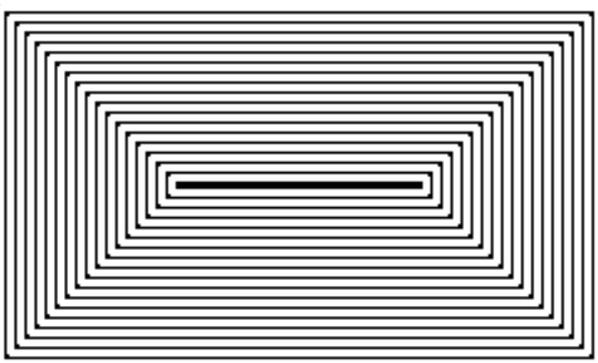


図 8 入れ子画像

Figure 8 A Nested Image

並列実行においては、最悪のケースとして、全画素に対して同時にアクセスし、同時に伝播処理を完了した場合、1 回走査で 1 近傍分のラベル ID しか伝搬できないため、統合を完了するために N-1 回画像を走査する必要がある。しかし、本稿で利用する TX2 の GPU 演算器数は検証で利用する最小画像サイズの全画素数に対して十分に少ない。よって、各画素を走査するタイミングがずれることで、小さなラベル ID がより早く伝搬する可能性が高い。

本検証では、葉画像を 5 種類と前背景成分が入れ子に存在する人工画像 (図 8) を 2 種類用意し、それぞれ 5 種類の画像サイズに変換したものをデータセットとした。このデータセットに対して穴埋め処理を 100 回実施した。施行毎に生成したラベル画像の全連結前景成分が正常に連結できていることを目視で確認した。

5.3 穴埋め処理の処理時間と高速化率

提案手法を TX2 上で実行した際の処理時間を表 2 に示す。処理時間計測に用いる検証画像には、前節で説明した葉診断向け画像 5 枚と作為的に生成した入れ子画像 2 枚を用いた。表 2 の Image 列は画像サイズを示しており、また、ラベルを付加してある。以降の表において画像サイズを表す際はこのラベルを用いる。処理時間は各画像に対して穴埋め処理を 100 回実行した平均である。単位はすべてマイクロ秒である。カーネルラベルについては、従来手法の {K1, K2, K3} はそれぞれ、背景連結成分抽出、2 値画像の穴埋め、前景連結成分抽出に対応しており、提案手法の {K1, K2} はそれぞれ、前背景同時連結成分抽出、穴埋め処理に対応している。また、紙面の都合上、葉画像の処理時間は全 5 種類の平均を載せている。各画像の画素数は前後で 4 倍ずつ大きくなっており、概ね画像サイズに対して処理時間が比例していることがわかる。また、提案手法による高速化率を表 3 に示す。実践的なデータセットである葉画像においては約 17-21% 程度、人工画像においても、Circle は約 15-22%、Rectangle は約 5-12% 程度処理時間を改善することが出来た。

従来手法及び提案手法の各画像の処理時間における各カーネルの占有率を表 4 と表 5 に示す。葉画像と人工画像との間には約 10-20% 前後の処理時間差が存在する。従来手法の場合、その差は連結成分抽出時間によるものである。葉画像の K1 及び K3 の入力となる 2 値画像は前景成分の面積が大きいため、連結成分抽出時間が増大したものと考えられる。また人工画像の Rectangle でも、K3 の入力 2 値画像の前景成分の面積が大きいため、葉画像と同様に連結成分抽出時間が増大したものと考えられる。

表 5 にある提案手法の K2 穴埋め処理時間が全体の処理時間に占める割合は従来手法のものとは比べて格段に大きくなっており、実行時間も従来手法と比べ約 2.4-3.0 倍程度増大している。これは従来手法と比べ提案手法では、穴埋め処理において前景の統合処理など計算量の大きな処理によ

るものである。

表 2 CPU と GPU での穴埋め処理時間 (us)

Table 2 Fill Holes Processing Time on a CPU and a GPU (us)

	Image	Leaf Avg.	Circle	Rectangle
Conventional Method on CPU	A) 480x270	3,633.6	3,351.7	3,414.8
	B) 960x540	9,321.5	8,361.5	6,909.0
	C) 1920x1080	28,038.7	27,148.4	23,304.0
	D) 3840x2160	99,456.2	95,316.7	84,855.2
	E) 7680x4320	380,125.3	393,415.9	330,016.9
Conventional Method on GPU	A) 480x270	1,962.5	1,854.3	1,749.1
	B) 960x540	6,117.5	5,559.4	5,006.7
	C) 1920x1080	22,367.4	19,548.3	17,540.9
	D) 3840x2160	91,618.8	79,173.4	70,551.9
	E) 7680x4320	363,128.6	318,005.2	283,795.9
Proposed Method on GPU	A) 480x270	1,671.7	1,526.1	1,565.1
	B) 960x540	5,062.3	4,648.8	4,554.5
	C) 1920x1080	18,945.3	17,010.4	16,677.4
	D) 3840x2160	76,946.2	67,715.1	66,350.2
	E) 7680x4320	300,001.1	262,055.2	256,156.8

表 3 GPU での従来/提案手法の処理時間高速化率 (倍)

Table 3 Acceleration Rate of the Processing Time in Conventional and Proposed Method on a GPU (times)

Image	Leaf Avg.	Circle	Rectangle
A)	1.174	1.215	1.118
B)	1.208	1.196	1.099
C)	1.180	1.149	1.052
D)	1.191	1.169	1.063
E)	1.211	1.214	1.108

表 4 従来手法の各カーネル処理時間の占有比率(%)

Table 4 Occupancy Ratio of Each Kernel Processing Time in the Conventional Method (%)

Image	Leaf Avg.			Circle			Rectangle		
	K1	K2	K3	K1	K2	K3	K1	K2	K3
A)	33	8	42	46	10	43	32	11	57
B)	33	6	45	45	8	47	29	9	62
C)	31	6	47	45	8	47	26	9	65
D)	30	6	48	45	8	47	25	9	66
E)	30	6	48	45	8	47	25	9	66

表 5 提案手法の各カーネル処理時間の占有比率

Table 5 Occupancy Ratio of Each Kernel Processing Time in the Proposed Method

Image	Leaf Avg.		Circle		Rectangle	
	K1	K2	K1	K2	K1	K2
A)	60	23	73	27	70	30
B)	63	20	76	24	71	29
C)	65	18	78	22	72	28
D)	65	18	78	22	72	28
E)	65	18	78	22	71	29

6. まとめ

本稿では、GPGPU 向け前背景同時連結成分抽出及び穴埋め処理の並列アルゴリズム及び最適化手法を提案した。

前者では GPGPU 等の SIMD 演算におけるコアのアイドル時間を削減し、後者では伝播処理を活用した前景成分の統合を行う並列アルゴリズムを用いることで処理を最適化し、全体的な処理時間を短縮した。これにより、TX2 上において、従来手法の GPU 実行時と比べて処理時間を約 1.05-1.22 倍高速化した。これは、従来手法の CPU 実行と比較すると、約 1.15-2.4 倍の高速化となる。また、提案手法を FHD 葉画像の処理時間は約 18.9 ms あり、穴埋め処理単体でのスループットは約 52.8fps を実現した。これらの結果から、穴埋め処理を用いた組み込み機器向け画像処理のリアルタイム化に対して有用な手法を提案したと言える。

参考文献

- [1] He, L., Gao, Q., Zhao, X., et al.: The connected-component labeling problem: A review of state-of-the-art algorithms, *Pattern Recognition* 70: 25-43 (2017).
- [2] 小川秀夫, 酒井大輔: 画像処理によるキュウリの葉の病気診断, *Bulletin of Aichi Univ. of Education*, 58(Natural Sciences), pp.13-19 (2009).
- [3] Rakhmadi, A., Rahim, M.S.M., Bade, A., et al.: Loop back connected component labeling algorithm and its implementation in detecting face, *International Journal of Computer, Electrical, Automation, Control and Information Engineering*, Vol.4, No.4, pp.635-640 (2010).
- [4] Shibata, N., Yamamoto, S.: GPGPU-Assisted Subpixel Tracking Method for Fiducial Markers, *Journal of Information Processing*, Vol.22, No.1, 19-28 (2014).
- [5] 柴田直樹, 山本真也: 連結成分抽出のための並列アルゴリズムの AVX2 命令セットを利用した CPU 向け実装, *マルチメディア通信と分散処理ワークショップ論文集(DPSWS2013)*, pp.300-307 (2013).
- [6] 木網啓人, 佐藤裕幸: 入退室管理システムにおける人物特定精度向上に向けた顔検出の高速化, *電子情報通信学会技術研究報告*, Vol.117, No.484, pp.229-234 (2018).
- [7] NVIDIA: NVIDIA JETSON The embedded platform for autonomous everything (online), available from <<https://www.nvidia.com/en-us/autonomous-machines/embedded-systems-dev-kits-modules/>> (accessed 2018-06-06).
- [8] Suzuki, K., Horiba, H., and Sugie, N.: Linear-time connected-component labeling based on sequential local operations, *Computer Vision and Image Understanding*, Vol.89, No.1, pp.1-23 (2003).
- [9] Wu, K., Otoo, E. and Suzuki, K.: Optimizing two-pass connected-component labeling algorithms, *Pattern Analysis and Applications*, Vol.12, No.2, pp.117-135 (2009).
- [10] He, L., Chao Y., and Suzuki, K.: A run-based two-scan labeling algorithm, *IEEE transactions on image processing*, Vol.17, No.5, pp.749-756 (2008).
- [11] Sakharnykh, N.: GPU Pro Tip: Fast Histograms Using Shared Atomics on Maxwell, *NVIDIA Developer Blog* (online), available from <<https://devblogs.nvidia.com/gpu-pro-tip-fast-histograms-using-shared-atomics-maxwell/>> (accessed 2018-06-08).
- [12] Luitjens, J.: Faster Parallel Reductions on Kepler, *NVIDIA Parallel for all* (online), available from <<https://devblogs.nvidia.com/parallelforall/faster-parallel-reductions-kepler/>> (accessed 2018-06-06).