Bulletin of Networking, Computing, Systems, and Software – www.bncss.org, ISSN 2186–5140
Volume 3, Number 1, pages 76–80, January 2014

Fast image component labeling on the GPU (Preliminary version)

Akihiko Kasagi, Koji Nakano, and Yasuaki Ito
Department of Information Engineering
Hiroshima University
Kagamiyama 1-4-1, Higashi Hiroshima, 739-8527 Japan

***Abstract*—Image component labeling is a process that assigns unique labels to the connected components in a binary image. In this paper, we propose a fast image component labeling using a GPU. The key idea of our approach is to make a connected graph from connected components in sub-images and propagate these labels on that graph. We have implemented our algorithm on the NVIDIA GeForce GTX680. The experimental results for a spiral pattern image and a hilbert curve image show that our implementation is 1.7 and 2.9 times faster than that of the existing research, respectively.**

***Index Terms*—Image component labeling, GPU, CUDA**

I. はじめに

ラベリング処理は様々なアプリケーションやシミュレーションで用いられる重要な処理である．2 次元画像に対するラベリング問題は，入力された画像の各ピクセルをノードとする格子状のグラフと見なした，各ノードを互いに連結する何れかのクラスタに分類するグラフカラーリング問題の一種である [1], [2]．この多大な計算量を必要とするラベリング問題には多くのアルゴリズムの研究が行われており，単一プロセッサによる逐次アルゴリズム [5], [6], [7] や，複数のコアを用いた並列アルゴリズム [2], [8] が存在している．

GPU(Graphics Processing Units) は多数のコアを並列に用いて処理を行うアーキテクチャであり，2006 年より NVIDIA 社が提供した CUDA(Compute Unified Device Architecture) によってプログラミングが可能となり，近年では汎用演算に GPU を用いる GPGPU(General-Purpose computing on GPU) の研究が盛んに行われている [3], [4].

CUDA を用いた 2 値画像に対するラベリングアルゴリズムでは，幾つかの手法が Hawick らによって提案されており，ポインタジャンプを用いた高速な手法 *Label Equivalence* 法が紹介されている [2]．この LE 法は木構造のグラフを複数作成した後に木を連結させていく方法を取っており，隣接するノードにラベルを伝播させる手法よりも，メモリへのアクセス回数が削減され高速に動作を行う．また文献 [8] では LE 法をメモリ消費という観点から改良しており，従来の LE 法と比べてラベリング処理に必要なメモリサイズを抑えることに成功しているが，性能評価では従来の LE 法の方が速いという結果を示している．

本稿では CUDA 上にて高速に処理を行うアルゴリズムを提案する．この手法は入力された 2 次元画像を幾つかの部分領域へと分割しラベリングを行い，各ピクセル群に割り当てられたラベルを一つのノードとし，隣接する部分領域の連結成分から新たにグラフを作成する．図 1 は新たに作成するグラフを図示したものである．このグラフ上にてラベルを伝播させることで，高速なラベル伝播を実現している．性能評価では NVIDIA 社が提供する GeForce GTX 680 に実装を行い，実行時間の測定を行った．その結果，渦巻状の 2 値画像に対しては最大 1.79 倍の高速化を達成し，ヒルベルト曲線に対して対しては最大 2.99 倍の高速化を達成した．

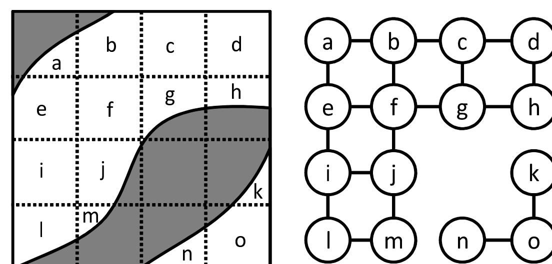

Fig. 1. 各領域の隣接関係を示したグラフ

本稿は 2 章で CUDA についてを述べ，3 章ではラベリングアルゴリズムを述べる．4 章では CUDA への実装を述べ，5 章では性能評価を示す．

II. CUDA

本章では CUDA アーキテクチャとプログラミングモデルについて述べる．CUDA は NVIDIA 社が提供する統合開発環境であり，NVIDIA 社の GPU に対してプログラミングモデルとインターフェースを提供している [3]．GPU は多数のコアからなる複数の Streaming Multiprocessor(SM) と高いメモリバンド幅を持つアーキテクチャであり，その高い演算能力を汎用演算に用いる研究が盛んに行われている．GPU は異なるアクセス速度のメモリを持つ NUMA アーキテクチャであり，CUDA のハードウェアモデルを示した図 2 では，*Global Memory* と *Shared Memory* の二種類のメモリを示している．

Global Memory は大容量 (1.5-6GB) のオフチップメモリで，メモリアクセスの速度は遅い．GPU 上の全てのコアは Global Memory にアクセスする事が可能で，各 MS は Global Memory を介して情報を共有する事が可能である．Shared Memory は小容量 (16-48KB) のオンチップメモリで，メモリのアクセス速度は速い．SM 上のコアは同 SM 上の Shared Memory にアクセスする事はできるが，異なる SM 上の Shared Memory にアクセスすることはできない．また各コアは，これらのメモリとは別に高速なローカルメモリ

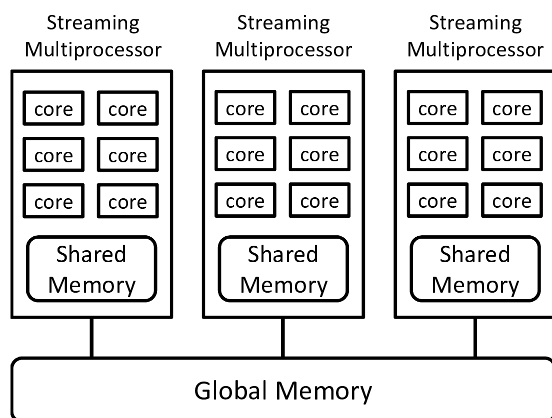

Fig. 2. CUDA ハードウェアモデル

を持っており，このメモリは他のコアがアクセスすることはできない．

CUDA のプログラミングモデルでは，並列に動作するスレッドを GPU 上の各コアに割り当てて演算を行う．また，各スレッドはブロックと呼ばれる単位で束ねられており，GPU 上ではブロック単位で SM に割り当てられる．これらのスレッドやブロックは，CUDA によって拡張されたカーネル関数を実行する際に生成することができる．本研究の提案手法では 2 値画像への割り当てを変化させつつ，処理を行っていく．

III. ラベリングアルゴリズム

本章では 2 値画像に対するラベリングアルゴリズムを提案する．本稿で取り扱う 2 値画像のラベリング問題では，周囲 4 近傍に対して連結する白ピクセル群に，ラベルとなる固有の数値を与えるアルゴリズムを取り扱う．

A. *Label Equivalence* 法

LE 法は，Suzuki らの逐次アルゴリズム [5] に似たアルゴリズムであり，Hawick らによって CUDA による並列アルゴリズムが提案された [2]．LE 法は 3 つのフェイズ”scanning”, ”analysis”, ”labeling”からなっており，scanning フェイズでは，自分の属するラベルのノード群に隣接するもっとも低い値のノードへのポインタを取得する．周囲に低い値のノードが存在しない場合は，ポインタを自身にする．次の analysis フェイズでは，scanning フェイズで取得したポインタを更新しなくなるまでジャンプさせ，根ノードに当たるラベルを指し示す．最後の labeling フェイズでは，根ノードのラベルを取得し，自身のラベルへと更新する．この 3 つのフェイズを 1 ラウンドとし，更新処理が行われなくなるまで繰り返していく．図 3 は LE 法の 1 ラウンド目の各フェイズでの様子を示したものである．

B. 提案手法のアルゴリズム

本章では GPU 上にて高速に処理を行う並列アルゴリズムを提案する．2 値画像に対するラベリング処理を高速化させる上で重要となる部分は，連結しているピクセルのラベルをどの様にして高速に伝播させるかが鍵となる．提案するアルゴリズムのアイデアとしては，まず入力された 2 値画像を格子状に分割し，各部分領域に対してラベリングを行う．

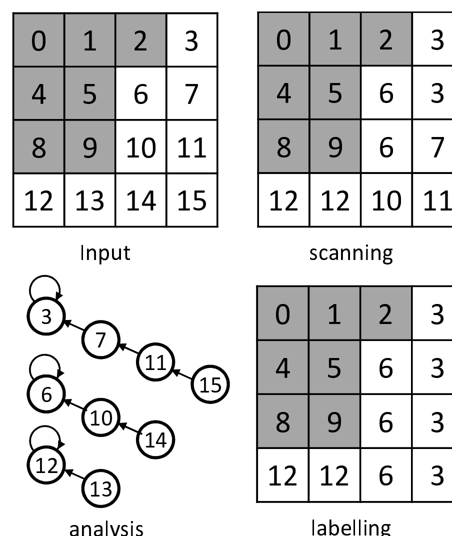

Fig. 3. LE 法での 1 ラウンド目の処理

次に各部分領域の白ピクセル群に割り当てられたラベルを 1 つのノード v_i と見なす．ここで部分領域中のノード v_i が隣接する部分領域の異なるノード v_j に面しているならば，異なるノード間の隣接関係を $e(v_i, v_j)$ のエッジと見なす．以上の操作から，入力された 2 値画像から $V = \{v_0, v_1, v_2, ...\}$ からなるノードと $E = \{e_0, e_1, e_2, ...\}$ となるエッジを持った無向グラフ $G = \{V, E\}$ を作成する．提案するアルゴリズムでは，作成した無向グラフ G 上でラベルの伝播を行う事で，部分領域単位でのラベルの伝播を行い高速化させる．このアルゴリズムを並列に処理する際は，次の 4Step によって行われる．また図 4 は各 Step での処理の様子を示したものである．

- Step1: 入力された 2 値画像を部分領域に分割し，各部分領域に対して並列にラベリングを行い，各白ピクセル群に固有のラベルを割り当てる．
- Step2: 部分領域間の境界線上にスレッドを割り当て，エッジリストを作成する．
- Step3: 作成したエッジリストを元にラベルの伝播を行う．
- Step4: 各ピクセルはそれぞれのノードが所持するラベルを参照し，自身のラベルへ更新する．

Step1 では $P = N \times N$ であるピクセルの画像を $M \times M$ の部分領域に分割し，各部分領域を並列にラベリング処理を行う．この $M \times M$ の部分領域へのラベリングには LE 法を用いて処理を行う．Step2 では $M \times M$ の部分領域の境界に当たるピクセルに対してスレッドをそれぞれ割り当て，エッジリストを作成する．図 5 は境界のエッジを取得する例であり，各スレッドは隣接するピクセルが黒ピクセルかどうかを判定し，どちらも白ピクセルであれば一方を v_i，もう一方を v_j とし，エッジ $e(v_i, v_j)$ とする．

Step3 では $2M \times 2M$ の領域を 1 つのブロックとして Step2 で作成したエッジにスレッドを割り当て，ラベルの値が小さい方へと更新を行う．ここで再び Step2 へと戻り，今度は $2M \times 2M$ を 1 つの領域として，再び境界に当たるピクセルからエッジリストを新たに作成し，Step3 では $4M \times 4M$ の領域を 1 つのブロックとしてエッジにスレッドを割り当

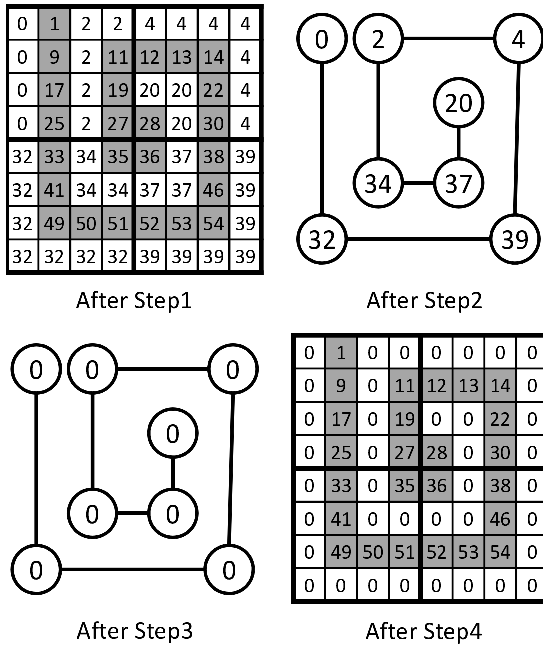

Fig. 4.　提案手法の各ステップ後の状況

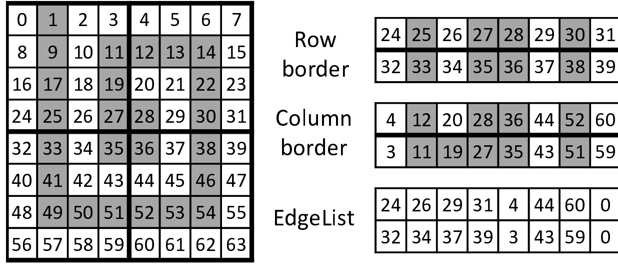

Fig. 5.　エッジリストの作成

て更新を行う．このような Step2 と Step3 を $\log\frac{N}{M}$ 回繰り返して処理を行う．図 6 は各段階毎の処理の割り当てを示した例である．最後の Step4 では更新されたラベルリストをルックアップテーブルとして各ピクセルのラベルの更新を行う．

IV. GPU 実装

本章では提案するアルゴリズムを CUDA を用いて GPU へと実装を行う．提案するアルゴリズムの Step1 では，まず各部分領域 $M \times M$ に対して LE 法を用いてラベリングを行う．部分領域 $M \times M$ は小さな領域である為，このラベリング処理は Shared Memory 上で高速に実行する事ができる．また初期値となる白ピクセルのラベルの値はローカルなスレッドの ID を割り当て，黒ピクセルには INTMAX の値を割り当て LE 法を行う．Step1 の後半では図 7 で示す通りに，各部分領域に対して固有のラベルを割り当てていくカーネル *LabelAllocate* を実行する．CUDA でのカーネルコードを **Algorithm 1** に示す．

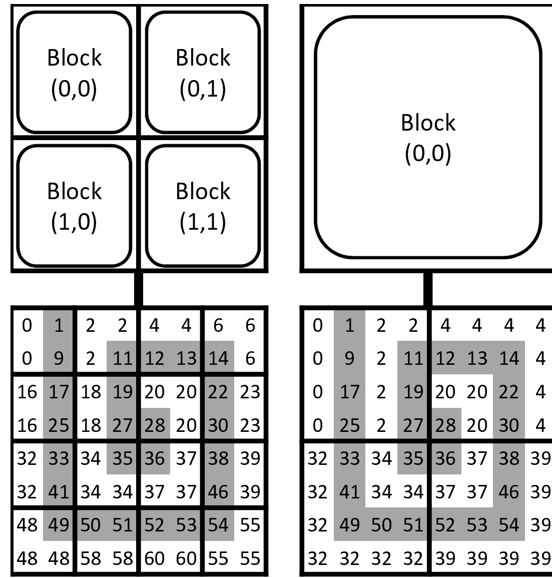

Fig. 6.　伝播処理におけるブロックの割り当て

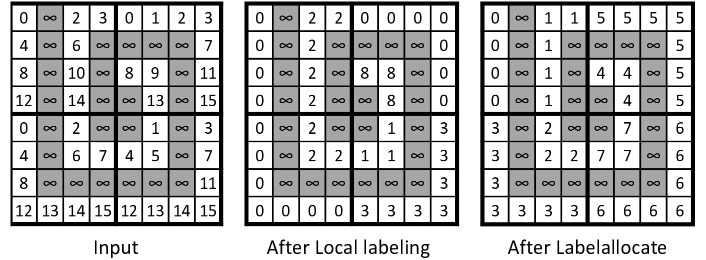

Fig. 7.　固有のラベルを振り直し

Algorithm 1. Kernel_ LabelAllocate

function Kernel_ LabelAllocate($P, L, Nodes$)
　declare integer $id, thid, ref$ in local memory
　declare integer num_s in shared memory
　declare boolean $S[threadSize]$ in shared memory
　$id \leftarrow$ threadID & blockID from CUDA runtime
　$thid \leftarrow$ threadID from CUDA runtime
　if $thid = 0$ **then**
　　$num_s \leftarrow 0$
　$S[P[id]] \leftarrow true$
　if $S[thid] = true$ **then**
　　$res \leftarrow$ atomicAdd(num_s,1)
　if $thid = 0$ **then**
　　declare integer $temp$ in local memory
　　$num_s \leftarrow$ atomicAdd($Nodes, num_s$)
　end if
　call syncthreads()
　if $S[thid] = true$ **then**
　　$res \leftarrow num_s$
　　$L[thid] \leftarrow res$
　end if
　call syncthreads()
　$P[id] \leftarrow L[P[id]]$
　return

Step2 では部分領域間の境界となるピクセルにスレッドを割り当て，エッジリストを作成するカーネル *MakeEdge* を実行する．CUDA でのカーネルコードを **Algorithm 2** に示す．

Algorithm 2. Kernel_ MakeEdge

function Kernel_ MakeEdge(P, L, E_i)
  declare integer id, ref in local memory
  declare integer num_s in shared memory
  declare integer v_1, v_2 in local memory
  $id \leftarrow$ threadID & blockID from CUDA runtime
  if $thid = 0$ **then**
    $num_s \leftarrow 0$
  end if
  $v_1 \leftarrow$ vertex from P
      where the edge between the subregions has
  $v_2 \leftarrow$ vertex from P
      where the edge between the subregions has
  if $v_1 ! =$ INTMAX **and** $v_2 ! =$ INTMAX **then**
    $ref \leftarrow$ atomicAdd(num_s,1)
    $E_i[ref] \leftarrow L[v_1]$
    $E_i[ref + offset] \leftarrow L[v_2]$
  end if
  return

カーネル関数である *LabelAllocate* と *MakeEdge* で Global Memory に作成されるラベルリストとエッジリストは，それぞれ atomicAdd 関数を用いて 1 つのエッジリストに詰めて格納している．これは後のアクセス時に Coalescing を用いて Global Memory へのアクセスを高速化させている．

Step3 では *MakeEdge* で作成したエッジリストに対してスレッドをそれぞれ割り当て，カーネル *LabelPropagate* にてラベルリストの更新を行う．このカーネルでは，各ブロックの全スレッドが更新処理が行われなくまでリストの更新を行う．CUDA でのカーネルコードを **Algorithm 3** に示す．

Algorithm 3. Kernel_ LabelPropagate

function Kernel_ LabelPropagate(L, E_i)
  declare integer $id, thid$ in local memory
  declare integer v_1, v_2 in local memory
  declare boolean m_s in shared memory
  $id \leftarrow$ threadID & blockID from CUDA runtime
  $thid \leftarrow$ threadID from CUDA runtime
  $v_1 \leftarrow E_i[id]$
  $v_2 \leftarrow E_i[id + offset]$
  if $v_1 = v_2$ **then return**
  repeat
    if $L[v_1] > L[v_2]$ **then**
      $L[v_1] \leftarrow L[v_2]$
      $m_s \leftarrow true$
    end if
    if $L[v_1] < L[v_2]$ **then**
      $L[v_2] \leftarrow L[v_1]$
      $m_s \leftarrow true$
    end if
    call syncthreads()
  until $m_s = false$
  return

最後の Step4 ではラベルの伝播が終わったラベルリストをルックアップテーブルとして各ピクセルへ更新していく．CUDA でのカーネルコードを **Algorithm 4** に示す．

Algorithm 4. Kernel_ LookUp

function Kernel_Lookup(P, L)
  declare integer id in local memory
  $id \leftarrow$ threadID & blockID from CUDA runtime
  $P[id] \leftarrow L[P[id]]$
  return

以上のカーネルコードを用いてラベリングを GPU 上で行う．またカーネルを呼び出すホストコードについては **Algorithm 5** に示す．

Algorithm 5. HostCode

function Kernel_ Host(P, m)
  declare integer $L[N]$
  declare integer $E_0[N], E_1[N], ..., E_m[N]$
  declare integer i, j
  call Local_Label_Equivalence(P)
  call Labelallocate(P, L)
  $i \leftarrow 0$
  repeat
    call MakeEdge(P, L, E_i)
    $j \leftarrow i$
    repeat
      call LabelPropagate(L, E_j)
      $j \leftarrow j - 1$
    until $j < 0$
    $i \leftarrow i + 1$
  until $i > m$
  call LookUp(P, L)
  return

V. 性能評価

本章では提案するアルゴリズムと LE 法を GPU に実装し，実行時間を計測し性能評価を行う．提案手法は $M \times M$ の部分領域のサイズを $M = 32$ として実行を行った．実験に用いた GPU は GeForce GTX 680 を使用し，性能評価には図 8 に示す渦巻きとヒルベルト曲線の 2 値画像を，それぞれ $N = 256, 512, 1024, 2048$ のサイズで入力し実行時間を計測した．表 I と表 II は実行時間を比較したものである．

表 I の渦巻きの画像に対する実行時間からは，画像サイズが大きくなるほど提案手法が高速であり，サイズが小さいほど LE 法が高速であることが分かる．これは入力画像

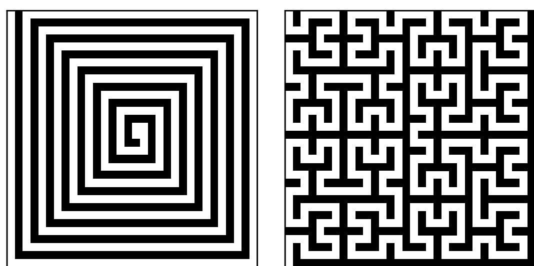

Fig. 8.　32×32 の渦巻きとヒルベルト曲線

TABLE I
渦巻き画像のラベリング実行時間 [MS]

Size N	256	512	1024	2048
LE	0.236	0.606	2.037	7.587
提案手法	0.296	0.708	2.081	4.550

TABLE II
ヒルベルト曲線のラベリング実行時間 [MS]

Size N	256	512	1024	2048
LE	0.501	1.427	5.198	21.560
提案手法	0.183	0.514	1.815	7.209

のサイズが小さい場合，Global Memory へのアクセスを多様する LE 法は L2 キャッシュの影響が強くなる．しかし入力画像のサイズが大きくなるにつれて，キャッシュに入らなくなるため処理時間の増加が顕著になる．また入力画像のサイズが小さい際、提案手法はラベル伝播の際に割り当てられるブロックやスレッドの数が少なく，GPU のオキュパンシーが低い状態で稼働しているなどの原因が考えられる．ヒルベルト曲線に対するラベリングでは，LE 法はラウンド回数が増えてしまう為，何れのサイズでも処理時間が大きくなっている．一方，提案手法は生成されるグラフが渦巻き画像よりも小さいものが生成されるため，ラベルの伝播が高速に行われている．しかし，ローカルなラベリングに LE 法を用いている為，サイズが大きくなるにつれてヒルベルト曲線の実行時間は渦巻き画像よりも遅い結果となった．

VI. まとめ

本研究では GPU を用いて 2 値画像に対するラベリングを高速に処理を行うアルゴリズムを提案し，GPU への実装を行い性能評価を行った．性能評価では先行研究である LE 法との実行時間の比較を行い，提案アルゴリズムは大きなサイズの画像に対して高速に処理を行える事を示した．その結果，渦巻状の 2 値画像に対しては最大 1.79 倍の高速化を達成し，ヒルベルト曲線に対して対しては最大 2.99 倍の高速化を達成した．

REFERENCES

[1] J. Bai, *Fast Dynamic Programming for Labeling Problems with Ordering Constraints*,Computer Vision and Pattern Recognition, 2012 IEEE Conference.

[2] K. Hawick, A. Leist, D. Playne, *Parallel graph component labeling with GPU and CUDA*, Parallel Computing 36, 2010.

[3] NVIDIA Corporation, “NVIDIA CUDA C programming guide version 5.5,” 2013.

[4] NVIDIA Corporation, “NVIDIA CUDA C best practice guide version 5.0,” 2012.

[5] K. Suzuki, I. Horiba, N. Sugie, *Linear-time connected-component labeling based on sequential local operation*, Computer Vision and Image Understanding 89, 2003.

[6] J. Hoshen, R. Kopelman, *Percolation and cluster distribution. i. cluster multiple labeling technique and critical concentration algorithm*, Physical Review B 14, 1976.

[7] K. Wu, E. Otoo, K. Suzuki, *Optimizing two-pass connected-component labeling algorithms*, Pattern Analysis & Applications, 2009.

[8] O. Kalentev, A. Rai, S. Kemnitz, R. Schneider, *Connected component labeling on a 2D grid using CUDA*, Journal of Parallel and Distributed Computing, 2010.