卒業論文

GPU を用いた液晶用ガラスの欠損検出画像処理の高速化(Ⅲ)

氏名：原田　健史
学籍番号：2260060085-4
指導教員：山崎 勝弘 教授
提 出 日：2011 年２月 17 日

立命館大学 理工学部 電子情報デザイン学科

内容梗概

液晶用ガラスは近年のデジタルテレビやスマートフォンの普及により採用数が激増している液晶パネルの主要部品である。液晶用ガラスはその加工工程において高温処理が必要とされるため、石英ガラスや無アルカリガラスといった比較的高価格なガラスを利用しなくてはならず、高コストになりやすいという問題を抱えており、検査時間短縮によるコストなどの低減が重要な課題となっている。本研究では GPU を用いた液晶用ガラスの欠損検出画像処理の高速化について述べる。並列化の対象として、「TDI フィルタ」、「ラプラシアンフィルタ」、「ラベリングフィルタ」の３つの画像処理を行うプログラムの並列化を行った。実行時間はＴＤＩフィルタについては約１倍、ラプラシアンフィルタについては約６．９倍、ラベリングフィルタについては２．４倍とむしろ処理速度が悪化してしまったが、そこから処理するデータ量を今回の実験環境においては 4 倍(5120x3840・18.8MB)まで増加させれば並列化の効果が発揮され処理時間が短縮されるであろうことが分かった。

目次

図目次

表目次

１．はじめに

液晶用ガラスは近年のデジタルテレビやスマートフォンの普及により採用数が激増している液晶パネルの主要部品である。液晶用ガラスはその加工工程において高温処理が必要とされるため、石英ガラスや無アルカリガラスといった比較的高価格なガラスを利用しなくてはならず、高コストになりやすいという問題を抱えている。それを解決するためには製造後の検査工程を省略するようなコストダウン方法が思いつくが、液晶用ガラスは表面に欠損が無いかを厳密に検査する必要があり、実際には検査の省略を行うのは難しい。よって検査に掛かる時間を減らしてコストを軽減するという方法を検討することになるが、近年の液晶用ガラスの面積増大により、プログラムの見直しのような細部の変更では処理速度の向上が難しい背景がある。そのため検査工程を高速化するためにはプログラムの処理方法を根本的に変更して処理を高速化する必要がある。

本研究では GPU を用いた液晶用ガラスの欠損検出画像処理の高速化を目的とする。欠損検出画像処理には欠損を発見しやすくするために TDI(Time Delayed Integration)フィルタ、ラプラシアンフィルタ、ラベリングフィルタ３つのアルゴリズムを利用する。TDI フィルタは大量の画像を読み込み、それを時間遅延積分方式によって合成することで、時間成分ごとに発生しているノイズの影響を減らすことができるフィルタである。それによって検査用カメラのノイズを軽減し、画像を鮮明にする。ラプラシアンフィルタは画像に輪郭強調処理を行い、欠損の存在を鮮明にすることで、小さな欠損であっても検出できるようにするアルゴリズムである。それによって液晶用ガラスの小さくて見逃しやすい欠損を発見しやすくする。三つ目がラベリングフィルタである。ラベリングフィルタは画像にラベリング処理を行って液晶用ガラスに存在する各欠損にラベルとして番号を割り振り、それによって確認すべき箇所の数の目安を作る。それによって液晶用ガラスに存在する欠損の数の大まかな目安を作り、確認作業を高速化する。

GPU とは PC に搭載されている画像処理・動画処理用チップである。従来 GPU は画像処理や動画処理しか行えないものであったが、近年では進歩が進み汎用的な計算を並列処理することができる特徴を持つ高速演算チップへと変化を遂げ、GPU プログラミングによって汎用的な演算に利用できるようになった。GPU プログラミングのための開発環境には NVIDIA 社の CUDA を利用する。CUDA は HPC 分野での採用例もある、NVIDIA が提供する GPU 向けの C 言語の統合開発環境である。

検証は、C 言語プログラミングによって画像処理アルゴリズムを記述したプログラムと、C 言語プログラミングと同等の記述を GPU プログラミングで行ったプログラムを作成し、その二つのプログラムの実行速度を測定して比較することで行う。

使用するアルゴリズムについての詳しい説明は２章、GPU プログラミングによる変更点や処理の高速化方法に関しては３章、比較による検討は４章にて行う。

２．　Ｃ言語による画像処理

２.１ TDI フィルタ

TDI フィルタとは、撮影する時間を少しずつずらした画像を大量に合成し、それによってノイズの影響を軽減する処理のことである。入力画像として連続撮影された画像 a と画像 b という２枚の画像があり、画像 a の座標(x,y)の画素値を(A(x),A(y))、画像 b の座標(x,y)の画素値を(B(x),B(y))、出力画像 out の座標(x,y)の画素値を OUT、１画像ごとの被写体の移動量を(α1,α2)とする。このときＴＤＩフィルタは次のような演算を行う

$$OUT = \{(A(x),A(y)) + (B(x+\alpha 1),B(y+\alpha 2))\}/2$$

この演算を任意の枚数 N で行った場合には、次のような形になる。

$$OUT = \{(A(x),A(y)) + (B(x+\alpha 1),B(y+\alpha 2)) + (C(x+\alpha 1*2),C(y+\alpha 2*2)) + \ldots + (N(x+\alpha 1*(N-1)),N(y+\alpha 2*(N-1)))\} / N$$

このような処理を画像で表すと図１のようになる。

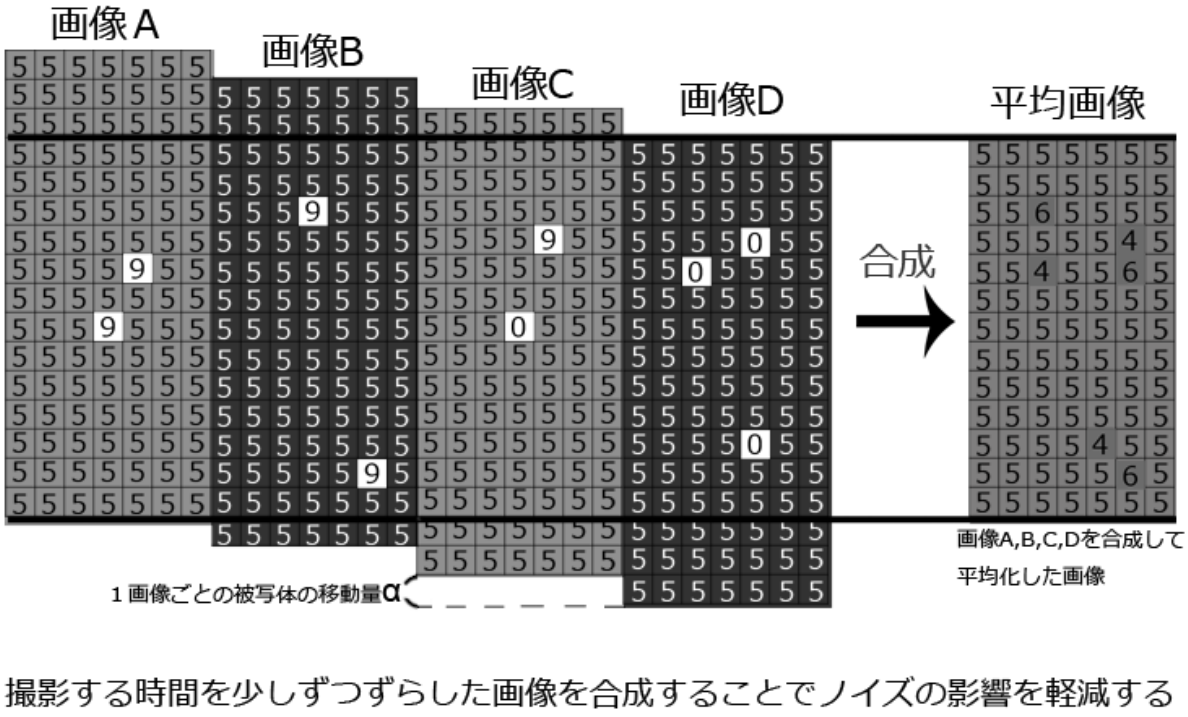


図 １：ＴＤＩフィルタ

2.2 ラプラシアンフィルタ

ラプラシアンフィルタとは画像に写っている物体の輪郭を強調する処理のことである。ラプラシアン∇^2は画像のような２次元のデータに二回微分を実行し、次のような演算を行う。

$$\nabla^2 f(x,y) = f_{xx}(x,y) + f_{yy}(x,y)$$

これを差分形式で表現すると、次の式のようになる。

$$\begin{aligned}\nabla^2 f(x,y) &= f(x-1,y) - 2f(x,y) + f(x+1,y) + f(x,y-1) - 2f(x,y) + f(x,y+1)\\ &= f(x-1,y) + f(x+1,y) + f(x,y-1) + f(x,y+1) - 4f(x,y)\end{aligned}$$

これを係数によって表現すると、

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

図 2：マスクパターン

となる。この係数をマスクパターンとして注目画素の近傍に微分を行う。

画像に対してラプラシアンフィルタを適用した際の例が図３である。

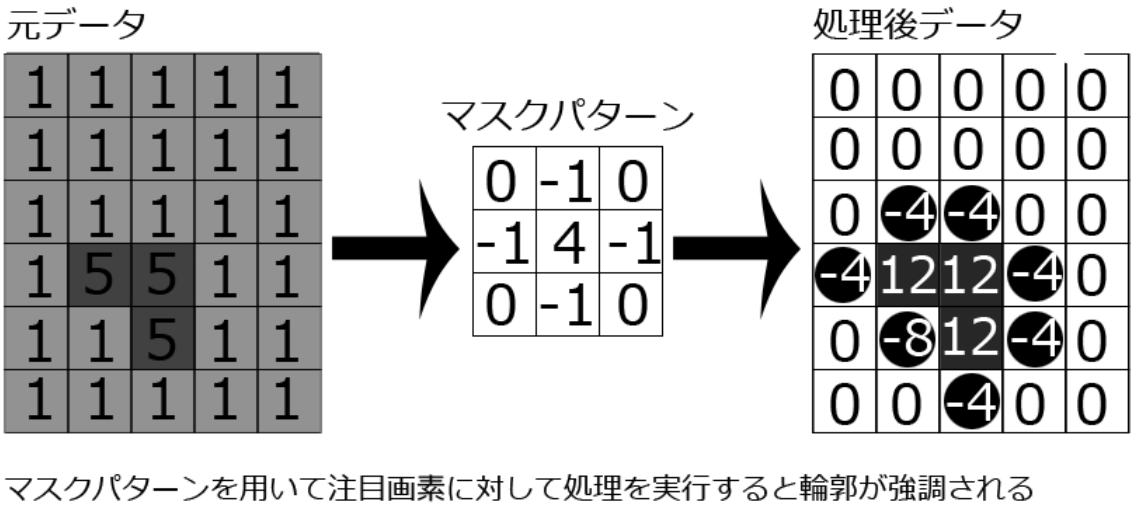


図 3：ラプラシアンフィルタ

２.３　ラベリングフィルタ

ラベリングは、対象画像において画素値が連続した画素に同じ番号を割り振る処理のことである。まずラベリングは全画素を走査し、設定した閾値に従って画素を白黒二値化する。次に画素を走査し、注目画素が未ラベルかつ画素値がラベル対象として設定された値（黒）と一致する場合に、注目画素に新規ラベル番号を設定する。続いて注目画素の隣接画素(上下左右)を走査し、隣接画素もさきほどの画素と同じ条件に一致する場合には、先ほど設定したのと同じラベル番号をその画素に設定する。その後さらにその画素の隣接画素に対しても同じ処理を繰り返す。そしてラベルが設定された画素に隣接する全ての画素の走査が終わると、ラベル番号を一つ繰り上げて新規ラベルをセットしてから次の画素を走査し、最終的に全ての画素に対して走査が終了するまでそれを繰り返す。

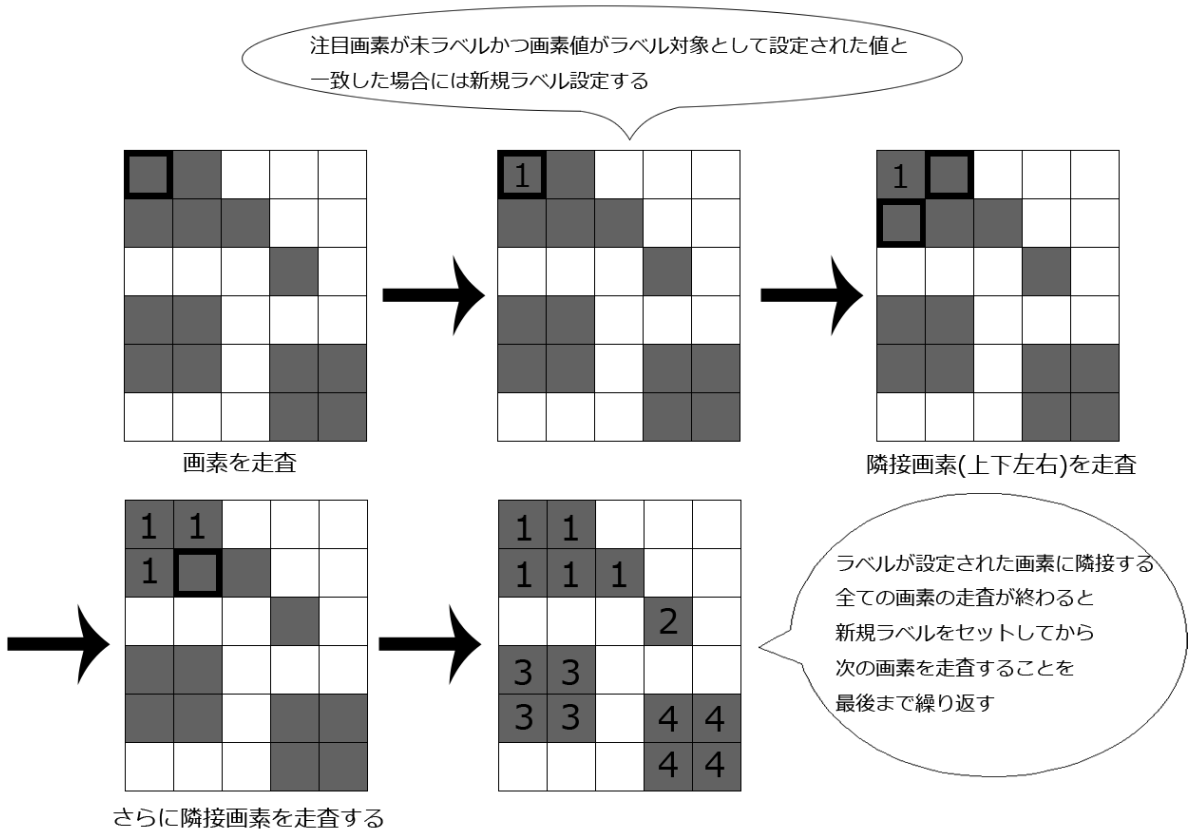


図 4：ラベリングフィルタ

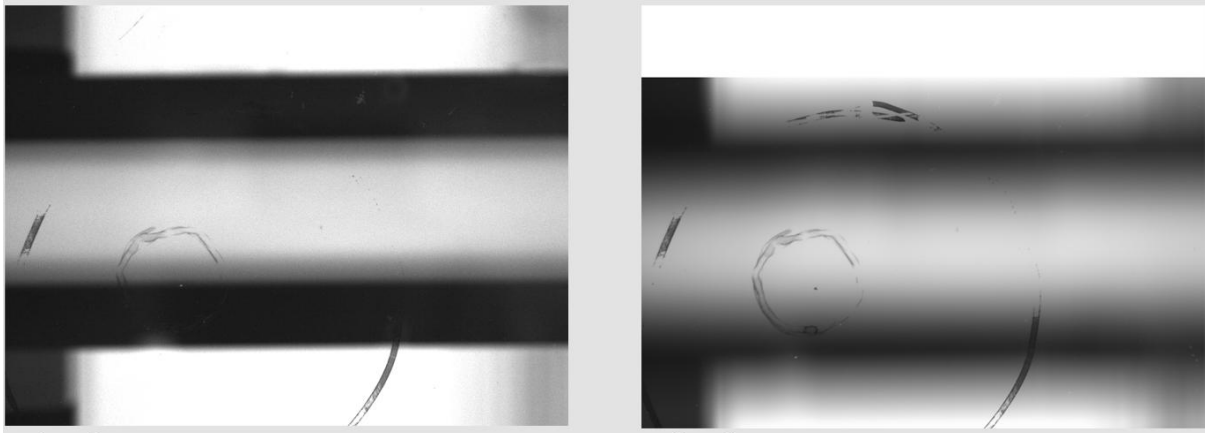
２.４　実行結果

２.４.１　実験条件

実験はＣＰＵに intel Core i7-950(3.07Ghz)、メモリにＤＤＲ3-1066Hz(8GB)を搭載したＰＣで行った。対象画像は１２８０＊９６０画素の８ビットビットマップファイルである。また、ラプラシアンフィルタ処理についてはそれに加えて対象画像を 2 倍(2560x1920・4.69MB)・4 倍(5120x3840・18.8MB)・8 倍(10240x7680・75MB)・16 倍（20480x15360・300MB）に拡大した画像での実験も行った。

２.４.２　ＴＤＩフィルタ実行結果

ＴＤＩフィルタ処理を枚数 165 枚で行った処理結果を以下に示す。実行時間は表１のとおりであり、実行時間は平均 365.721 ミリ秒であった。出力画像は入力画像に比べてノイズが軽減されており、ＴＤＩフィルタの効果が確認できる。



入力画像　　出力画像

図 5：ＴＤＩフィルタ実行結果

表 1：C 言語によるＴＤＩフィルタ実行時間(単位はミリ秒)

	一回目	二回目	三回目
TDI フィルタ	373.702	361.372	362.089

２.４.３　ラプラシアンフィルタ実行結果

ラプラシアンフィルタ処理を行った処理結果を以下に示す。フィルタ処理の効果が分かりやすくするため出力結果の各画素値に 50 のオフセットを与えて表示している。実行時間は表２のとおりであり、実行時間は平均 14.321 ミリ秒であった。出力画像は入力画像の輪郭部分を強調した画像になっており、ラプラシアンフィルタの効果が確認できる。
また、入力画像を 2 倍・4 倍・8 倍・16 倍に拡大して処理を行った場合の実行時間についても測定した。

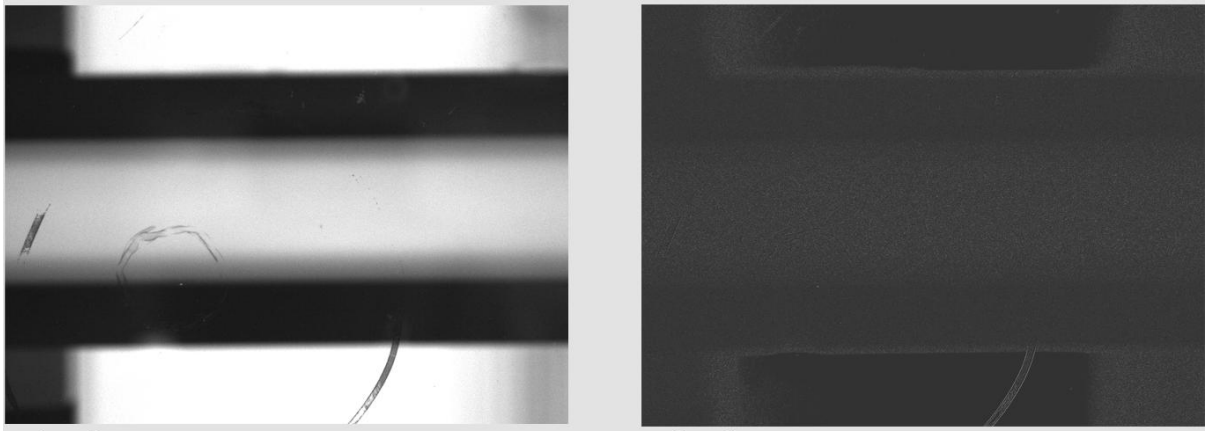


図 6：ラプラシアンフィルタ実行結果

表 2：C 言語によるラプラシアンフィルタ実行時間(単位はミリ秒)

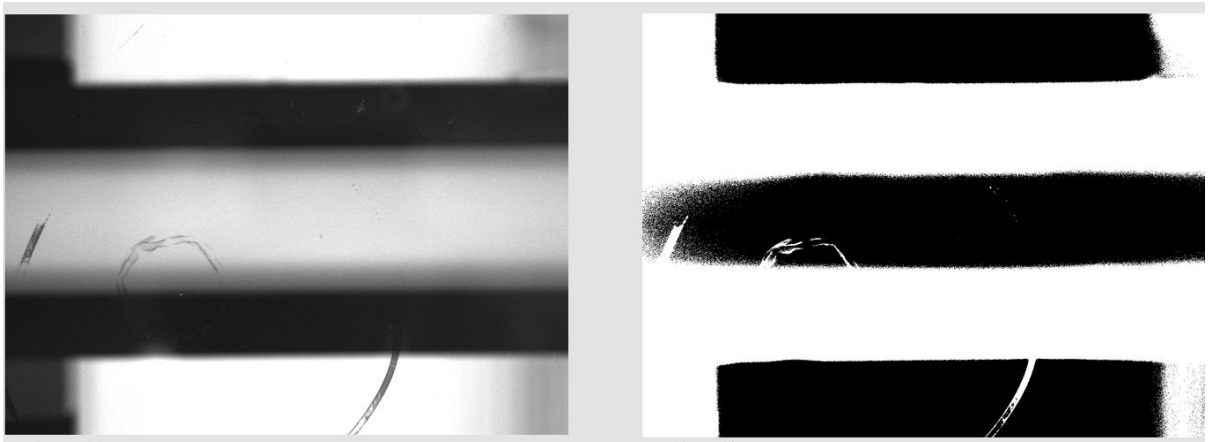
	一回目	二回目	三回目
ラプラシアン	14.379	14.193	14.392

表 3：C言語による拡大画像を用いたラプラシアンフィルタ実行時間

	2 倍(2560x1920・4.69MB)	4 倍(5120x3840・18.8MB)	8 倍(10240x7680・75MB)	16 倍(20480x15360・300MB)
ラプラシアン拡大	56.715	237.261	1274.623	6786.344

２.４.４　ラベリングフィルタ実行結果

ラベリングフィルタ処理を行った処理結果を以下に示す。白黒二値化の閾値は 180 とした。実行時間は表２のとおりであり、実行時間は平均 13375.258 ミリ秒、ラベル数は 3614 であった。出力画像とラベル数により入力画像に写っている物体の数とその場所がはっきりと分かるようになっており、ラベリングフィルタの効果が確認できる。



入力画像　　出力画像

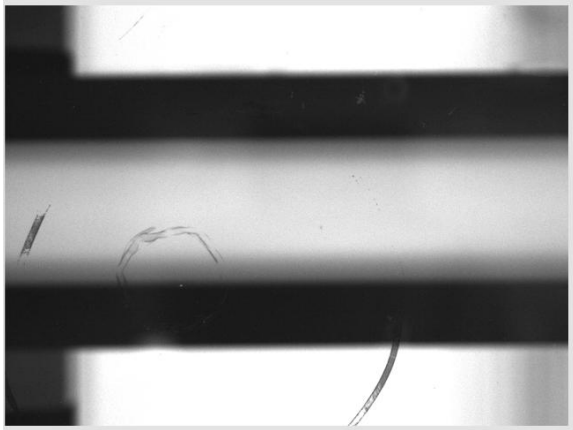
図 7：ラベリングフィルタ実行結果

表 4：ラベリングフィルタ実行時間(単位はミリ秒)

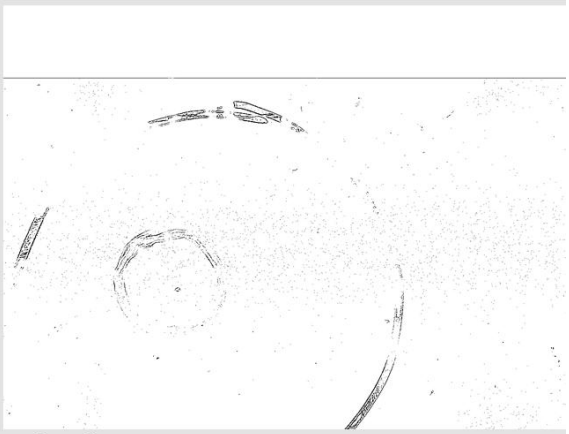
	一回目	二回目	三回目
ラベリング	13405.806	13325.166	13394.802

２.４.５　全フィルタ利用時の実行結果

ＴＤＩフィルタ、ラプラシアンフィルタ、ラベリングフィルタを続いて実行した際の処理結果を以下に示す。TDI 処理枚数は 165 枚、ラプラシアンフィルタ処理のオフセット値は０、ラベリング処理の白黒二値化の閾値は６とした。実行時間は表３とおりであり、実行時間は平均 9547.483 ミリ秒、ラベル数は 3962 であった。出力画像とラベル数により画像に存在する物体の場所と数が明確に確認できるようになっており、画像処理の効果が確認できる。



入力画像



出力画像

図 8：全フィルタ利用時の実行結果

表 5：全フィルタ実行時間(単位はミリ秒)

	一回目	二回目	三回目
全フィルタ利用	9614.839	9510.321	9516.585

３．GPU による処理の高速化

３.１ GPU のアーキテクチャ

GPU とは PC に搭載されている画像処理・動画処理用チップである。従来 GPU は画像処理や動画処理しか行えないものであったが、近年では進歩が進み汎用的な計算を並列処理することができる特徴を持つ高速演算チップへと変化を遂げ、GPU プログラミングによって汎用的な演算に利用できるようになった。本研究にて使用する NVIDIA 社の GPU「Geforce シリーズ」はストリーミングマルチプロセッサと呼ばれるプロセッサのセットを複数搭載することで構成されている。ストリーミングマルチプロセッサは、単純な演算機能しか持たないストリーミングプロセッサ、プロセッサごとのメモリであるシェアードメモリ、読み込み専用のコンスタンスメモリとテクスチャキャッシュメモリ、特別な機能を実現するためのスペシャルファンクションユニット、プロセッサセットの命令実行を行うインストラクションユニットで構成されている。

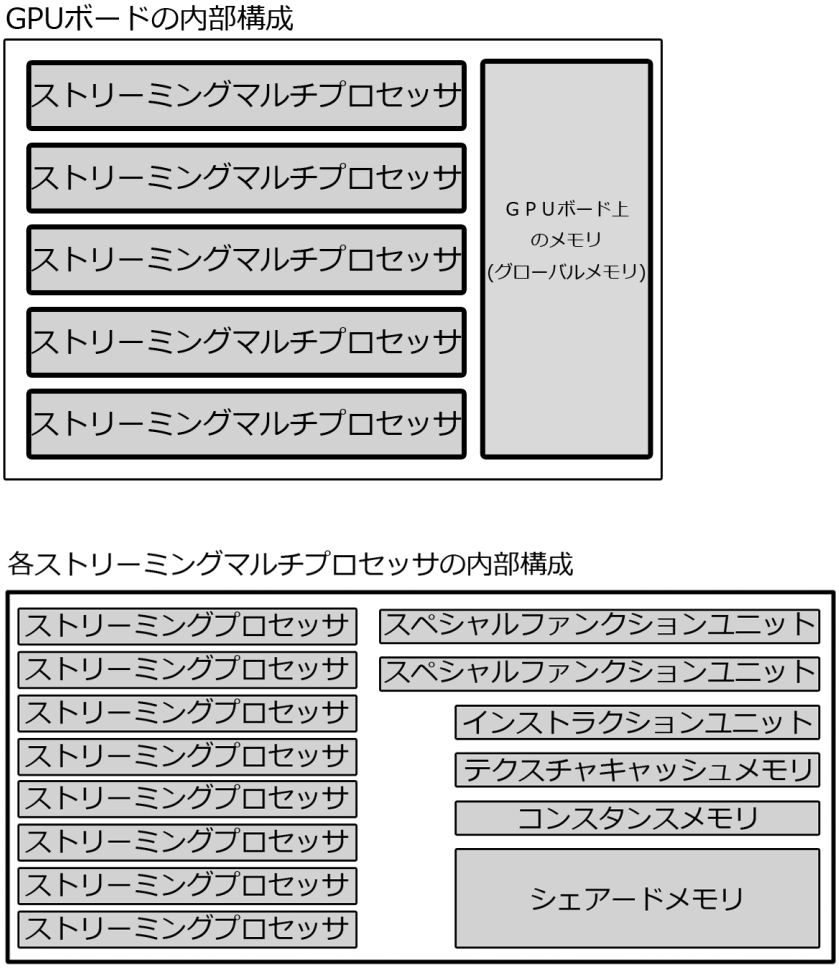


図 9：ＧＰＵボードの内部構成

ＧＰＵプログラミングではこの内部リソースをグリッド・ブロック・スレッドという３単位で分割して扱うことで並列処理を行う。グリッドは GPU のリソース全てに及ぶ単位である。グリッドは内部リソースをブロック単位で分割して動作する。開発環境 CUDA の現在のバージョンではグリッドは GPU 一つにつき一つしか定義出来ないが、将来のバージョンでは複数の GPU を一つのグリッドとして扱えるようになることが予告されている。ブロックはグリッドを分割した各範囲を指す単位である。ブロックはその内部リソースをスレッド単位に分割して動作する。スレッドはブロックを分割した各範囲を指す単位である。スレッドは各スレッドでプログラムされた処理を並列的に実行する。生成できる最大スレッド数は１グリッド当たり 65535 個である。なお GPU 上のリソースはグリッド・ブロック・スレッドの３単位に分割されて利用されるが、GPU ボード上に配置されている各メモリについてはその影響を受けず、各メモリは常にその種類に応じた特徴を持ち続ける。そのためメモリアクセス効率を良くしてプログラムの処理速度を高速化させるには GPU 上に複数存在するメモリのタイプに合わせたメモリアクセスを意識してプログラミングを行う必要がある。

３.２ GPU プログラミング

ＧＰＵプログラミングの開発環境 CUDA は、C 言語を利用して GPU プログラミングを行うものである。そのため、Ｃ言語プログラムの一部をＧＰＵプログラミング用に変更することでＧＰＵ利用プログラムへと変更することができ、利用が比較的容易である。Ｃ言語プログラムをＧＰＵプログラミングに変更する際の主な変更は、for 文によるデータへの連続アクセスを行う部分に加えられる。たとえばＣ言語プログラミングで for 文によるデータへの連続アクセスが 1000 万回繰り返されているとすると、その部分をＧＰＵプログラミングによって 1000 個の処理に分割すれば処理回数 1 万回の処理を並列的に 1000 個処理することになり、理論上処理時間は 1/1000 に短縮される。

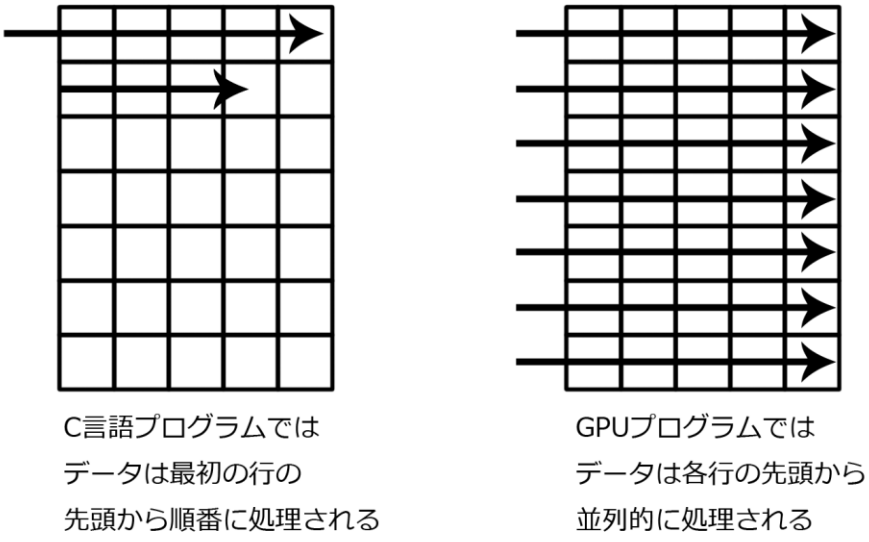


図 10：ＧＰＵによる並列処理

ただし、実際のプログラム実行速度は理論上の速度を下回る。これには主に二つの原因がある。一つ目はＧＰＵ利用そのもののオーバーヘッドによる処理時間の増加である。ＧＰＵを利用したプログラムはＧＰＵでの処理の開始時にＰＣからＧＰＵにデータを転送し、処理が終わると今度はＧＰＵからＰＣへとデータを転送する。そのためＧＰＵプログラムはＣ言語プログラムと比べると、データ転送に掛かる時間の分だけ確実に処理時間が長くなる。二つ目は並列処理のメモリアクセス効率の問題である。ＧＰＵはスレッド単位に分割したタスクを並列処理するが、各スレッドは対象とするデータや範囲がそれぞれで違う。そのためもし大量のスレッド中に一つだけ処理が遅れるような分岐を繰り返して処理するスレッドが存在した場合には、全てのスレッドは最も処理の遅いスレッドが処理を完了するまで次の処理を待たなくてはならない。そのため処理方法次第では並列処理を行っているにも関わらず処理速度が遅くなる。ＧＰＵプログラミングではこれらのことを意識した上でプログラム記述を行わなければ処理速度が低下する。

３.３ アルゴリズムの並列化

本研究ではＧＰＵプログラミングによってラプラシアンフィルタアルゴリズムとラベリングフィルタアルゴリズムを並列化した。ＴＤＩフィルタアルゴリズムについては並列処理によって高速化できる範囲が狭く、ＧＰＵ利用によるオーバーヘッドを超える処理速度短縮を期待できなかったため、ＧＰＵプログラミングによる並列化を行わなかった。

３.３.１　ラプラシアンフィルタの並列化

ラプラシアンフィルタの高速化はラプラシアンフィルタ処理を CUDA による並列処理に置き換えることで行った。変更した範囲は以下の図の通りである。処理は全体を１グリッドとして扱い、それを３２ブロックに分割して行った。スレッド数は画像の縦解像度の数値と同じであり、1280x960 の画像であれば 960 並列である。

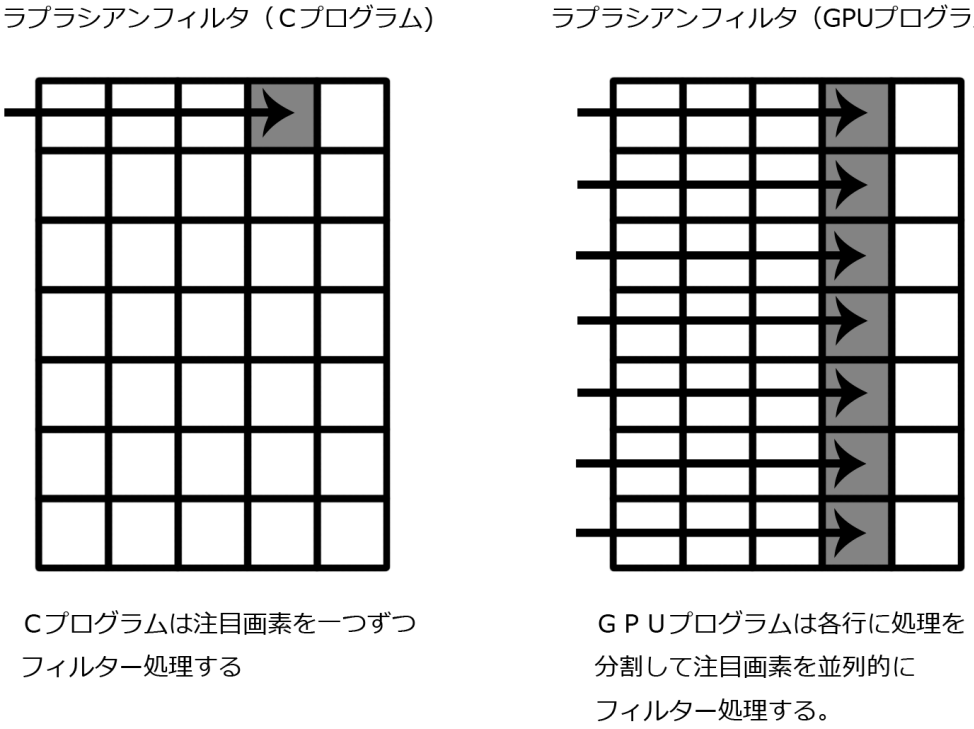


図 11：ラプラシアンフィルタの高速化

３.３.２　ラベリング処理の並列化

ラベリング処理の高速化はラベリング処理の一部を CUDA による並列処理に置き換えることで行った。変更した範囲は以下の図の通りである。ラベリング処理は一部を CPU で行っているため、CUDA による処理を行う度に GPU-ホストＰＣ間のデータ転送が発生する。処理は全体を１グリッドとして扱い、それを３２ブロックに分割して行った。スレッド数は画像の縦解像度の数値と同じであり、1280x960 の画像であれば 960 並列である。

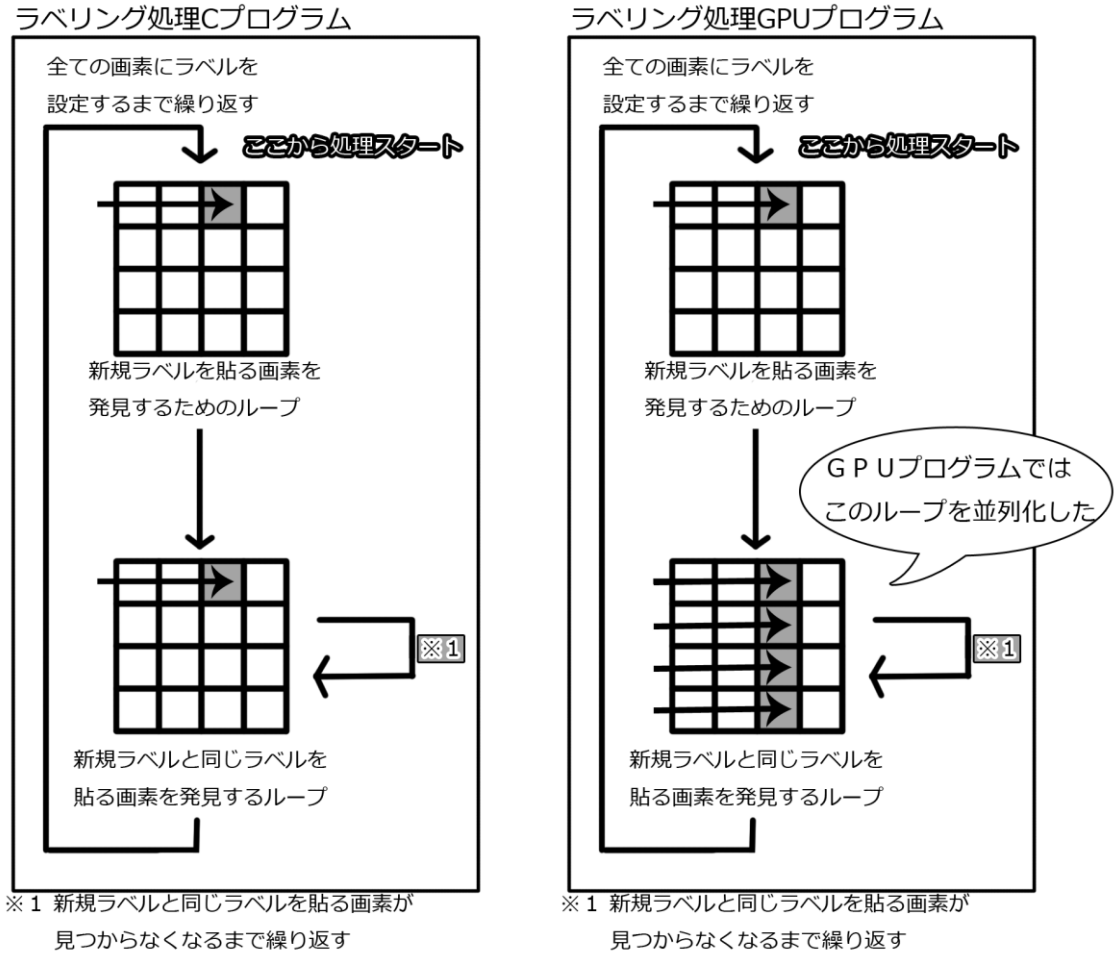


図 12：ラベリングの高速化

３.３ 実行結果

３.３.１　実験条件

実験はＣＰＵに intel Core i7-950(3.07Ghz)、メモリにＤＤＲ3-1066Hz(8GB)、ＧＰＵボードに ENGTX480 を搭載したＰＣで行った。GPU ボードである ENGTX480 のスペックは

グラフィックエンジン NVIDIA GeForce GTX480
バス規格 PCI Express 2.0
ビデオメモリ GDDR5 1536MB
エンジンクロック 700 MHz
シェーダークロック 1401 MHz
メモリクロック 3696 MHz (924 MHz DDR5)
RAMDAC 400MHz
メモリインターフェイス幅 384-bit

である。
対象画像は１２８０＊９６０画素の８ビットビットマップファイルである。
また、ラプラシアンフィルタ処理についてはそれに加えて対象画像を 2 倍(2560x1920・4.69MB)・4 倍(5120x3840・18.8MB)・8 倍(10240x7680・75MB)・16 倍 (20480x15360・300MB)に拡大した画像での実験も行った。

３.３.２ ＴＤＩフィルタ実行結果

ＴＤＩフィルタ処理を枚数 165 枚で行った処理結果を以下に示す。実行時間は表１のとおりであり、実行時間は平均 381.687 ミリ秒であった。出力画像によりＧＰＵプログラムがＣプログラムと同じ演算結果を出力していることが確認できた。

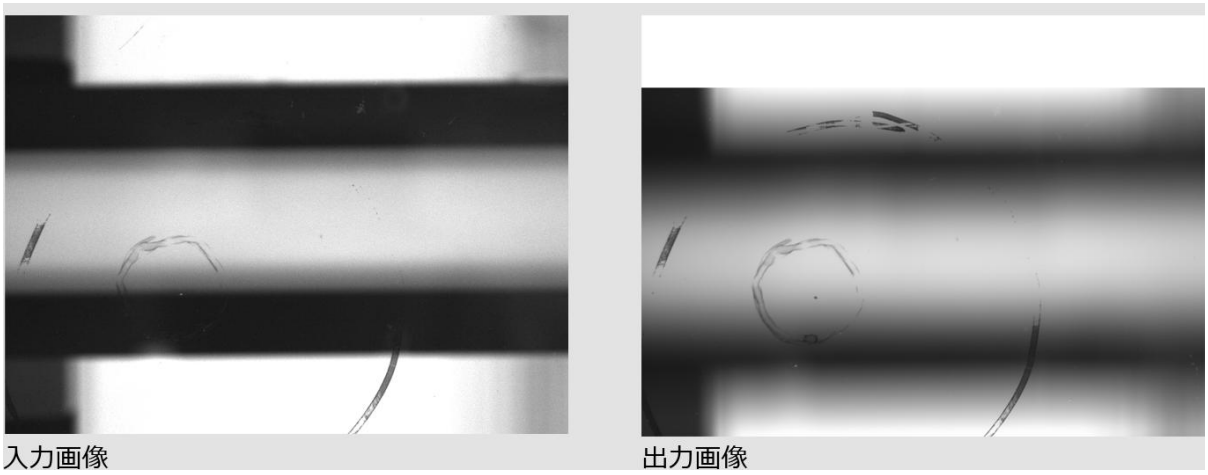


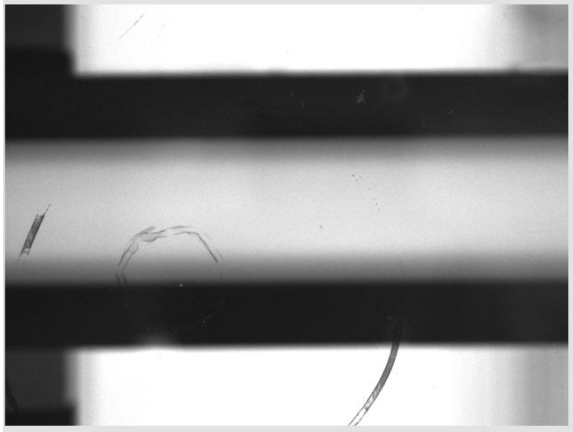
図 13：ＧＰＵプログラムでのＴＤＩフィルタ実行結果

表 6：ＧＰＵプログラムでのＴＤＩフィルタ実行時間(単位はミリ秒)

	一回目	二回目	三回目
TDI 処理	406.794	367.386	370.883

３.３.３ ラプラシアンフィルタ実行結果

ラプラシアンフィルタ処理を行った処理結果を以下に示す。フィルタ処理の効果が分かりやすくするため出力結果の各画素値に 50 のオフセットを与えて表示している。実行時間は表２のとおりであり、実行時間は平均 95.693 ミリ秒であった。出力画像によりＧＰＵプログラムがＣプログラムと同じ演算結果を出力していることが確認できた。
また、入力画像を 2 倍・4 倍・8 倍・16 倍に拡大して処理を行った場合の実行時間についても測定した。


入力画像

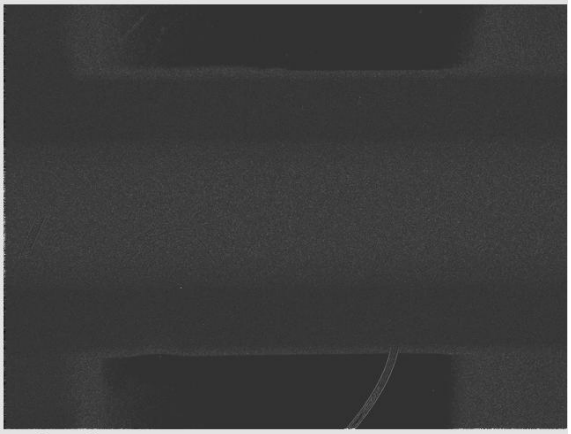

出力画像

図 14：ＧＰＵプログラムでのラプラシアンフィルタ実行結果

表 7：ＧＰＵプログラムでのラプラシアンフィルタ実行時間

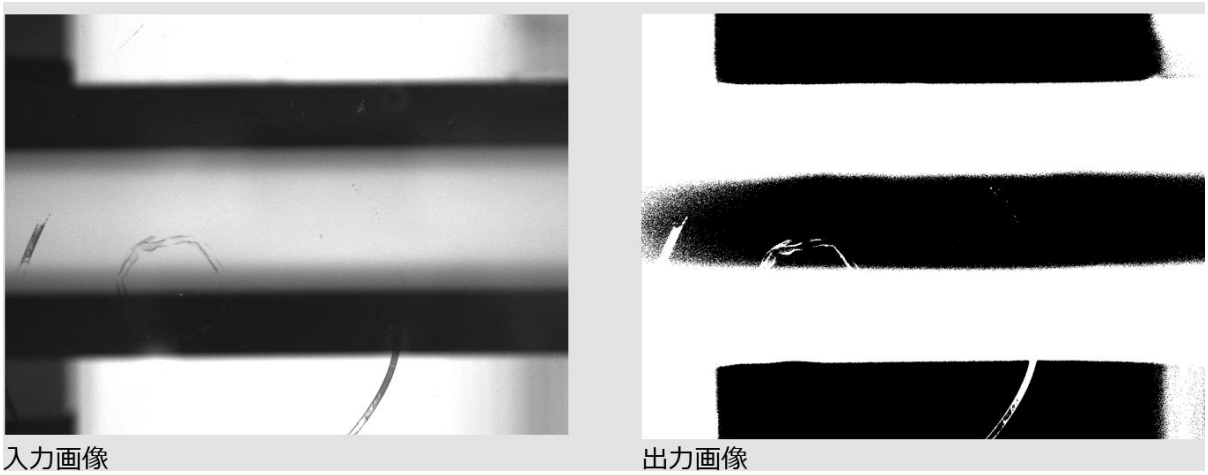
	一回目	二回目	三回目
ラプラシアン	130.778	75.082	81.22

表 8：ＧＰＵプログラムでの拡大画像を用いたラプラシアンフィルタ実行時間

	2 倍(2560x1920・4.69MB)	4 倍(5120x3840・18.8MB)	8 倍(10240x7680・75MB)	16 倍(20480x15360・300MB)
ラプラシアン拡大	127.399	202.802	996.731	4557.385

３.３.４　ラベリングフィルタ実行結果

ラベリングフィルタ処理を行った処理結果を以下に示す。白黒二値化の閾値は 180 とした。実行時間は表２のとおりであり、実行時間は平均 32431.477 ミリ秒、ラベル数は 3614 であった。出力画像とラベル数によりＧＰＵプログラムがＣプログラムと同じ演算結果を出力していることが確認できた。



入力画像　　出力画像

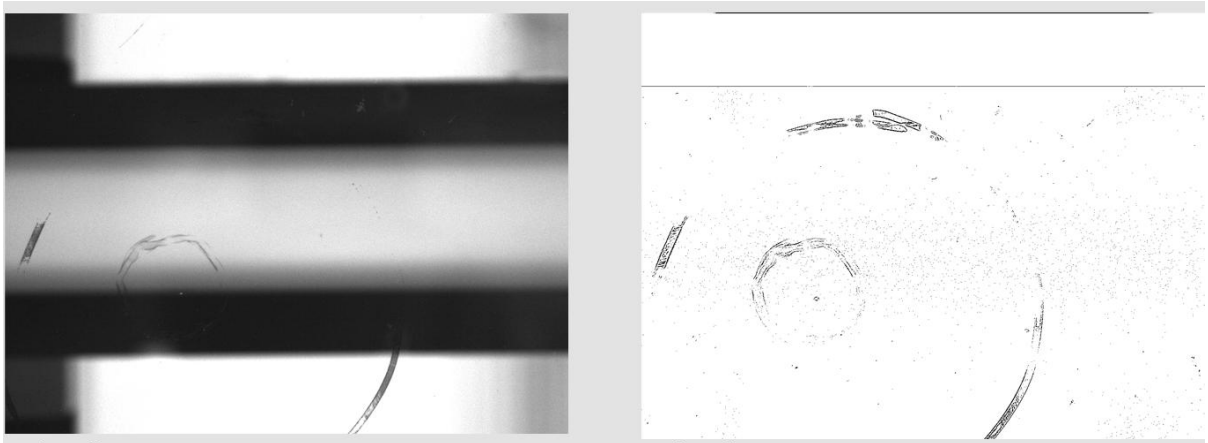
図 15：ＧＰＵプログラムでのラベリングフィルタ実行結果

表 9：ＧＰＵプログラムでのラベリングフィルタ実行時間

	一回目	二回目	三回目
ラベリング	33065.152	32599.281	31630.988

３.３.５　全フィルタ利用時の実行結果

ＴＤＩフィルタ、ラプラシアンフィルタ、ラベリングフィルタを続いて実行した際の処理結果を以下に示す。TDI 処理枚数は 165 枚、ラプラシアンフィルタ処理のオフセット値は０、ラベリング処理の白黒二値化の閾値は６とした。実行時間は表３とおりであり、実行時間は平均 30188.25 ミリ秒、ラベル数は 3962 であった。出力画像とラベル数によりＧＰＵプログラムがＣプログラムと同じ演算結果を出力していることが確認できた。



入力画像　　出力画像

図１６：ＧＰＵプログラムでの全フィルタ利用時の実行結果

表 10：ＧＰＵプログラムでの全フィルタ実行時間

	一回目	二回目	三回目
全てのフィルタを適用	30214.06	30216.937	30133.753

４．考察

GPU プログラムの実行時間は TDI を除いて全てのアルゴリズムで増加している。これはGPUによる並列処理を行わなかったTDIの処理時間がほぼ1倍になっていることからも分かるように、明らかにGPU利用が速度低下の原因である。

表 11：画像処理実行時間比較

	TDI	ラプラシアン	ラベリング	全てのフィルタ
GPU/C 実行時間	約 1 倍	約 6.9 倍	約 2.4 倍	約 3.4 倍

しかし、拡大画像を使ってデータ処理量を増やしてラプラシアン処理を実行した場合においては、画像のサイズが 4 倍(5120x3840・18.8MB)を超えた辺りからＧＰＵプログラムの実行時間はＣプログラムよりも短くなっている。なおラプラシアンフィルタとラベリング処理は同様の方法でＧＰＵを利用するようにプログラムを記述していることから、処理するデータ量を変更すればラベリング処理においても同じような速度の推移が起きると考えられる。

表 12：拡大画像を使ったラプラシアンフィルタ処理時間比較

	2 倍(2560x1920・4.69MB)	4 倍(5120x3840・18.8MB)	8 倍(10240x7680・75MB)	16 倍(20480x15360・300MB)
GPU/C 実行時間	約 2.2 倍	約 0.85 倍	約 0.78 倍	約 0.67 倍

これらのことから、ＧＰＵプログラムでは処理するデータ量が少ないと処理速度はむしろ遅くなり、逆にデータ量が多くなるにつれ並列化の効果が発揮され処理が高速になる性質を持つと考えられる。

よって、本研究では画像処理をＧＰＵプログラムで行った場合の実行時間はむしろ増加してＧＰＵ利用の効果が出なかったが、対象となるデータを 4 倍(5120x3840・18.8MB)程度まで増加させて同じ画像処理を行えば、ＧＰＵによる処理の並列化の効果が発揮され処理速度が高速化されるであろうことが分かった。

５．おわりに

本研究では、GPU を用いた液晶用ガラスの欠損検出画像処理の高速化を検証することを目的とし、GPU による１２８０＊９６０画素の８ビットビットマップファイルへの「TDI フィルタ」、「ラプラシアンフィルタ」、「ラベリング処理」の並列処理を行った。実行時間は「ＴＤＩフィルタ」では約１倍、「ラプラシアンフィルタ」では約６．９倍、「ラベリング処理」では約２．４倍とむしろ悪化してしまったが、処理するデータ量を今回の実験環境においては４倍(5120x3840・18.8MB)まで増加させれば並列化の効果が発揮され処理時間が短縮されるであろうことが分かった。

謝辞

本研究の機会を与えてくださり，貴重な助言，ご指導を頂きました山崎勝弘教授に深く感謝いたします。また、本研究に関して様々な相談に乗って頂き、貴重な助言を頂いた研究室の皆様に深く感謝いたします。

参考文献

[1]岡田賢治 他：CUDA 高速 GPU プログラミング入門，秀和システム,2010
[2] Steve Oualline 他：C 実践プログラミング 第 3 版，オライリー・ジャパン,1998
[3] 大山 佳：OpenMP を用いた画像処理プログラムの並列化，
立命館大学理工学部電子情報デザイン学科卒業論文,2010
[4] Wikipedia :Windows bitmap(http://ja.wikipedia.org/wiki/Windows_bitmap)
[5] 近藤正芳のウェブページ ：ｂｍｐファイルフォーマット
（http://www.kk.iij4u.or.jp/~kondo/bmp/）
[6] INE wiki：瀧沢和也/C 言語/バイナリファイルの操作(ビットマップファイル)
(http://www.ine.sie.dendai.ac.jp/wiki/index.php?%C2%ED%C2%F4%CF%C2%CC%E9%2FC%B8%C0%B8%EC%2F%A5%D0%A5%A4%A5%CA%A5%EA%A5%D5%A5%A1%A5%A4%A5%EB%A4%CE%C1%E0%BA%EE(%A5%D3%A5%C3%A5%C8%A5%DE%A5%C3%A5%D7%A5%D5%A5%A1%A5%A4%A5%EB))

付録ａ　Ｃプログラムソースコード

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <cutil_inline.h>

//TDI の枚数
#define TDI_num 165

//ファイル読み込み時のオフセット　通常は０
#define OFFSET 0

//ラプラシアンフィルタの結果に対するオフセット値　通常は０
#define R_OFFSET 0

int getFileSize(FILE *fp);

void RAP(unsigned int fileSize,int bmpWidth,int bmpHeight,unsigned char *buffer_copy);

void LAB(unsigned int fileSize,int bmpWidth,int bmpHeight,unsigned char *buffer_copy);


int main(void){
	FILE *readFile;		//読み込み用ファイル
	FILE *writeFile;	//書き込み用ファイル

	unsigned char *buffer_original = NULL;	//データの格納用
	unsigned char *tmpBuffer_original = 0;	//データの操作用

	unsigned char *buffer_copy = NULL;	//データの格納用
	unsigned char *tmpBuffer_copy = 0;	//データの操作用

	int *buffer_data = NULL;	//データの格納用
	int *tmpBuffer_data = 0;	//データの操作用

	unsigned int fileSize = 0;		//ファイルサイズ格納用
	char filename[100];
	int bmpWidth = 0;			//ビットマップの横幅格納用
	int bmpHeight = 0;			//ビットマップの縦幅格納用

	int tmp_name;				//TDI 処理で使用

	int i = 0, j = 0, k = 0;

	unsigned int cudaTimer = 0;
	cutilCheckError(cutCreateTimer(&cudaTimer));
	cutilCheckError(cutStartTimer(cudaTimer));


	////////////////////////////////////
	////////////////////////////////////
	//////////TDI 処理ここから//////////
	////////////////////////////////////
	////////////////////////////////////

	///////////////
	//一回目の処理//
	///////////////
	tmp_name = OFFSET;
	//ファイルネームを生成
	if(tmp_name < 10){
		sprintf(filename,"data00%d.bmp",tmp_name);
	}
	else if(tmp_name < 100){
		sprintf(filename,"data0%d.bmp",tmp_name);
```

```
}
else{
	sprintf(filename,"data%d.bmp",tmp_name);
}

if ((readFile= fopen(filename, "rb")) == NULL) {
	printf("ERR:ReadFile¥n");
	exit(1);
}
printf("filename = %s OK¥n",filename);

//fileSize にファイルサイズを格納
fileSize = getFileSize(readFile);
printf("filesize OK¥n");


//ファイルから読み込んだデータ格納用配列
if ((buffer_copy = (unsigned char *)malloc(sizeof(unsigned char) * fileSize)) == NULL) {
	printf("ERR:Memory¥n");
	exit(1);
}

//ファイルからバッファにコピー
fread(buffer_copy, sizeof(unsigned char), fileSize, readFile);
fclose(readFile);

//データから画像の幅と高さを得る
memcpy(&bmpWidth, buffer_copy + (sizeof(unsigned char) * 18), sizeof(unsigned char) * 4);
memcpy(&bmpHeight, buffer_copy + (sizeof(unsigned char) * 22), sizeof(unsigned char) * 4);


//ファイルから読み込んだデータの計算用配列
if ((buffer_data = (int *)malloc(sizeof(int) * fileSize)) == NULL) {
	printf("ERR:Memory¥n");
	exit(1);
}

//空の data に初期値を与えておく
tmpBuffer_data =buffer_data + (54 + 1024);
tmpBuffer_copy =buffer_copy + (54 + 1024);

for (i = 0; i < bmpHeight; i++) {
	for (j = 0; j < bmpWidth; j++) {
		*tmpBuffer_data = *tmpBuffer_copy;
		tmpBuffer_data++;
		tmpBuffer_copy++;
	}
}

//////////////////////////
//一回目の処理ここまで//
//////////////////////////


////////////////////
//２回目以降の処理//
////////////////////

//二回目以降の処理なので i=1 にしておく
i = 1;
while(i < TDI_num){
	i++;
	tmp_name = i + OFFSET;

	//ファイルネームを生成
	if(tmp_name < 10){
		sprintf(filename,"data00%d.bmp",tmp_name);
	}
	else if(tmp_name < 100){
		sprintf(filename,"data0%d.bmp",tmp_name);
```

```
    }
    else{
        sprintf(filename,"data%d.bmp",tmp_name);
    }

    if ((readFile= fopen(filename, "rb")) == NULL) {
        printf("ERR:ReadFile¥n");
        exit(1);
    }
    printf("filename = %s OK¥n",filename);


    //メモリ確保
    //ファイルから読み込んだデータ格納用配列
    if ((buffer_original = (unsigned char *)malloc(sizeof(unsigned char) * fileSize)) == NULL) {
        printf("ERR:Memory¥n");
        fclose(readFile);
        exit(1);
    }

    //ファイルからバッファにコピー
    fread(buffer_original, sizeof(unsigned char), fileSize, readFile);
    fclose(readFile);

    tmpBuffer_original =buffer_original;
    tmpBuffer_copy =buffer_copy;

    //データから画像の幅と高さを得る

    //ヘッダとパレットを読み飛ばすため buffer に 54+1024 足したアドレス
    //を tmpBuffer に格納する
    tmpBuffer_original = buffer_original + (54 + 1024) + (bmpWidth * i);
    tmpBuffer_data =buffer_data + (54 + 1024);

    for (j = 0; j < bmpHeight - i; j++) {
        for (k = 0; k < bmpWidth; k++) {
            *tmpBuffer_data = *tmpBuffer_data + *tmpBuffer_original;
            tmpBuffer_original++;
            tmpBuffer_data++;
        }
    }

free(buffer_original);
}
//////////////////////
//二回目の処理ここまで//
//////////////////////


//////////////////////
//ここからまとめの処理//
//////////////////////

//合算したデータ値を TDI_num の値で割って TDI 処理を行う
if(TDI_num == 1){
    tmpBuffer_data = buffer_data + (54 + 1024);

    for (i = 0; i < bmpHeight; i++){
        for (j = 0; j < bmpWidth; j++) {
            *tmpBuffer_data = (*tmpBuffer_data / TDI_num);
            tmpBuffer_data++;
        }
    }
}

else{
    tmpBuffer_data = buffer_data + (54 + 1024);

    for (i = 0; i < bmpHeight - TDI_num; i++){
        for (j = 0; j < bmpWidth; j++) {
```

```
                    *tmpBuffer_data = (*tmpBuffer_data / TDI_num);
                    tmpBuffer_data++;
                }
            }

            //データが無い部分に黒挿入
            for (i = 0; i < (bmpWidth*TDI_num); i++){
                *tmpBuffer_data = 255;
                tmpBuffer_data++;
            }
        }

        //original に読み込んだ内容を copy にコピーしておく
        tmpBuffer_data =buffer_data + (54 + 1024);
        tmpBuffer_copy =buffer_copy + (54 + 1024);
        for (i = 0; i < (bmpHeight * bmpWidth); i++) {

            if(*tmpBuffer_data > 255){
                *tmpBuffer_data = 255;
            }

            if(*tmpBuffer_data < 0){
                *tmpBuffer_data = 0;
            }

            *tmpBuffer_copy = *tmpBuffer_data;
            tmpBuffer_data++;
            tmpBuffer_copy++;
        }

        RAP(fileSize,bmpWidth,bmpHeight,buffer_copy);
        LAB(fileSize,bmpWidth,bmpHeight,buffer_copy);

        if ((writeFile = fopen("a.bmp", "wb")) == NULL) {
            printf("ERR:WriteFile¥n");
            fclose(readFile);
            exit(1);
        }

        fwrite(buffer_copy, sizeof(unsigned char), fileSize, writeFile);
        fclose(writeFile);

        free(buffer_data);
        free(buffer_copy);


        cutilCheckError(cutStopTimer(cudaTimer));
        printf("It takes %f milliseconds¥n",cutGetTimerValue(cudaTimer));
        cutilCheckError(cutDeleteTimer(cudaTimer));

        return 0;
}

int getFileSize(FILE *fp){
        fpos_t tmp_pos = 0;
        fpos_t fileSize = 0;

        tmp_pos = ftell(fp);
        fseek(fp, 0L, SEEK_SET);
        fseek(fp, 0L, SEEK_END);
        fgetpos(fp,&fileSize);

        fseek(fp, tmp_pos, SEEK_SET);

        return (int)fileSize;
}



void RAP(unsigned int fileSize,int bmpWidth,int bmpHeight,unsigned char *buffer_copy){
        unsigned char *tmpBuffer_copy = 0;
```

```
	unsigned char *buffer_copy2 = NULL;
	unsigned char *tmpBuffer_copy2 = 0;
	int tmp_int;//計算用(ラプラシアン)
	int flag = 0; //フラグ用(ラプラシアン・ラベリング)
	int i = 0, j = 0;		/*ループ用*/

	//ラベリング用のメモリ確保
	if ((buffer_copy2 = (unsigned char *)malloc(sizeof(unsigned char) * fileSize)) == NULL) {
		printf("ERR:Memory¥n");
		exit(1);
	}

	//ラベル配列をクリア
	tmpBuffer_copy = buffer_copy;
	tmpBuffer_copy2 = buffer_copy2;
	for (i = 0; i <((54 + 1024) + (bmpHeight*bmpWidth)); i++) {
			*tmpBuffer_copy2 = *tmpBuffer_copy;
			tmpBuffer_copy++;
			tmpBuffer_copy2++;
	}

	tmpBuffer_copy2 = buffer_copy2 + (54 + 1024);
	tmpBuffer_copy = buffer_copy + (54 + 1024);
	//画像の処理
	for (i = 0; i < bmpHeight; i++) {
		for (j = 0; j < bmpWidth; j++) {

			if(i == 0 && j == 0){
				//縦の 1 列目の一番左の bit のための処理
				tmp_int = 2 * (*tmpBuffer_copy2) - *(tmpBuffer_copy2+1) - *(tmpBuffer_copy2+bmpWidth);
				flag = 1;
			}

			if(i == 0 && j == (bmpWidth -1)){
				//縦の 1 列目の一番右の bit のための処理
				tmp_int = 2 * (*tmpBuffer_copy2) - *(tmpBuffer_copy2-1) - *(tmpBuffer_copy2+bmpWidth);
				flag = 1;
			}

			if(i == (bmpHeight - 1) && j == 0){
				//縦の最終列の一番左の bit のための処理
				tmp_int = 2 * (*tmpBuffer_copy2) - *(tmpBuffer_copy2+1) - *(tmpBuffer_copy2-bmpWidth);
				flag = 1;
			}

			if(i == (bmpHeight - 1) && j == (bmpWidth -1)){
				//縦の最終列の一番右 bit のための処理（n は任意）
				tmp_int = 2 * (*tmpBuffer_copy2) - *(tmpBuffer_copy2-1) - *(tmpBuffer_copy2-bmpWidth);
				flag = 1;
			}

			if(i == 0 && flag == 0){
				//縦の 1 列目の一番右と一番左以外の bit のための処理
				tmp_int = 3 * (*tmpBuffer_copy2) - *(tmpBuffer_copy2-1) - *(tmpBuffer_copy2+1) -
*(tmpBuffer_copy2+bmpWidth);
				flag = 1;
			}

			if(j == 0 && flag == 0){
				//縦の n 列目の一番左 bit のための処理（n は任意）
				tmp_int = 3 * (*tmpBuffer_copy2) - *(tmpBuffer_copy2+1) - *(tmpBuffer_copy2-bmpWidth) -
*(tmpBuffer_copy2+bmpWidth);
				flag = 1;
			}

			if(j == (bmpWidth - 1) && flag == 0){
				//縦の n 列目の一番右 bit のための処理（n は任意）
				tmp_int = 3 * (*tmpBuffer_copy2) - *(tmpBuffer_copy2-1) - *(tmpBuffer_copy2-bmpWidth) -
*(tmpBuffer_copy2+bmpWidth);
```

```
                flag = 1;
            }

            if(i == (bmpHeight - 1) && flag == 0){
                //縦の最終列の一番右と一番左以外のための処理（n は任意）
                tmp_int = 3 * (*tmpBuffer_copy2) - *(tmpBuffer_copy2-1) - *(tmpBuffer_copy2+1) -
*(tmpBuffer_copy2-bmpWidth);
                flag = 1;
            }

            if(flag == 0){
                //特別な処理を必要としない bit の処理
                tmp_int = 4 * (*tmpBuffer_copy2) - *(tmpBuffer_copy2-1) - *(tmpBuffer_copy2+1) -
*(tmpBuffer_copy2-bmpWidth) - *(tmpBuffer_copy2+bmpWidth);
            }


            if(tmp_int > 255){
                tmp_int = 255;
            }

            if(tmp_int < 0){
                tmp_int = 0;
            }
            *tmpBuffer_copy = tmp_int + R_OFFSET;
            tmpBuffer_copy++;
            tmpBuffer_copy2++;
            flag = 0;
        }
    }
    printf("ラプラシアン OK");
    free(buffer_copy2);
    return;
}




void LAB(unsigned int fileSize,int bmpWidth,int bmpHeight,unsigned char *buffer_copy){

    unsigned char *tmpBuffer_copy = 0;   //データの操作用

    int *buffer_label = NULL;    //データの格納用
    int *tmpBuffer_label = 0;    //データの操作用

    int flag = 0; //フラグ用(ラプラシアン・ラベリング)

    int label_num;//ラベリング処理で使用
    int bit_discover;//ラベリング処理で使用
    int bit_enlarge;//ラベリング処理で使用

    int i = 0, j = 0;

    //２値化処理
    tmpBuffer_copy =buffer_copy + (54 + 1024);

    for (i = 0; i < bmpHeight; i++) {
        for (j = 0; j < bmpWidth; j++) {
            if(*tmpBuffer_copy <= 10){
                *tmpBuffer_copy = 255;
            }
            else{
                *tmpBuffer_copy = 0;
            }
            tmpBuffer_copy++;
        }
    }
```

```
//ラベリング用のメモリ確保
if ((buffer_label = (int *)malloc(sizeof(int) * fileSize)) == NULL) {
	printf("ERR:Memory¥n");
	exit(1);
}

//ラベル配列をゼロクリア
tmpBuffer_label = buffer_label + (54 + 1024);
for (i = 0; i < bmpHeight; i++) {
	for (j = 0; j < bmpWidth; j++) {
		*tmpBuffer_label = 0;
		tmpBuffer_label++;
	}
}

label_num = 0;
bit_discover = 1;
while(bit_discover == 1){
	bit_discover = 0;
	tmpBuffer_label = buffer_label + (54 + 1024);
	tmpBuffer_copy = buffer_copy + (54 + 1024);

	//新規ビット発見用ループ
	for (i = 0; i < bmpHeight; i++) {
		for (j = 0; j < bmpWidth; j++) {
			if(*tmpBuffer_copy == 0 && *tmpBuffer_label == 0){
				label_num++;
				*tmpBuffer_label = label_num;
				printf("label = %d¥n",label_num);
				bit_discover = 1;
			}
			if(bit_discover == 1){
				break;
			}
			tmpBuffer_label++;
			tmpBuffer_copy++;
		}
		if(bit_discover == 1){
			break;
		}

	}

	//発見したビットを拡張するループ
	if(bit_discover == 1){

		bit_enlarge = 1;
		while(bit_enlarge == 1){
			bit_enlarge = 0;

			tmpBuffer_label = buffer_label + (54 + 1024);
			tmpBuffer_copy = buffer_copy + (54 + 1024);

			for (i = 0; i < bmpHeight; i++) {
				for (j = 0; j < bmpWidth; j++) {
					if(*tmpBuffer_copy == 0 && *tmpBuffer_label == label_num){
						flag = 0;

						if(i == 0 && j == bmpWidth - 1){
							flag = 1;
							if(*(tmpBuffer_copy - 1) == 0 && *(tmpBuffer_label - 1) == 0){
								*(tmpBuffer_label - 1) = label_num;
								bit_enlarge = 1;
							}
							if(*(tmpBuffer_copy + bmpWidth) == 0 && *(tmpBuffer_label + bmpWidth) == 0){
								*(tmpBuffer_label + bmpWidth) = label_num;
								bit_enlarge = 1;
							}
						}
```

```
if(i == 0 && j == 0){
	flag = 1;
	if(*(tmpBuffer_copy + 1) == 0 && *(tmpBuffer_label + 1) == 0){
		*(tmpBuffer_label + 1) = label_num;
		bit_enlarge = 1;
	}
	if(*(tmpBuffer_copy + bmpWidth) == 0 && *(tmpBuffer_label + bmpWidth) == 0){
		*(tmpBuffer_label + bmpWidth) = label_num;
		bit_enlarge = 1;
	}
}

if(i == (bmpHeight - 1) && j == (bmpWidth -1)){
	flag = 1;
	if(*(tmpBuffer_copy - 1) == 0 && *(tmpBuffer_label - 1) == 0){
		*(tmpBuffer_label - 1) = label_num;
		bit_enlarge = 1;
	}
	if(*(tmpBuffer_copy - bmpWidth) == 0 && *(tmpBuffer_label - bmpWidth) == 0){
		*(tmpBuffer_label - bmpWidth) = label_num;
		bit_enlarge = 1;
	}
}

if(i == (bmpHeight - 1) && j == 0){
	flag = 1;
	if(*(tmpBuffer_copy + 1) == 0 && *(tmpBuffer_label + 1) == 0){
		*(tmpBuffer_label + 1) = label_num;
		bit_enlarge = 1;
	}
	if(*(tmpBuffer_copy - bmpWidth) == 0 && *(tmpBuffer_label - bmpWidth) == 0){
		*(tmpBuffer_label - bmpWidth) = label_num;
		bit_enlarge = 1;
	}
}

if(i == (bmpHeight - 1) && flag == 0){
	flag = 1;
	if(*(tmpBuffer_copy - 1) == 0 && *(tmpBuffer_label - 1) == 0){
		*(tmpBuffer_label - 1) = label_num;
		bit_enlarge = 1;
	}
	if(*(tmpBuffer_copy + 1) == 0 && *(tmpBuffer_label + 1) == 0){
		*(tmpBuffer_label + 1) = label_num;
		bit_enlarge = 1;
	}
	if(*(tmpBuffer_copy - bmpWidth) == 0 && *(tmpBuffer_label - bmpWidth) == 0){
		*(tmpBuffer_label - bmpWidth) = label_num;
		bit_enlarge = 1;
	}
}

if(j == (bmpWidth - 1) && flag == 0){
	flag = 1;
	if(*(tmpBuffer_copy - 1) == 0 && *(tmpBuffer_label - 1) == 0){
		*(tmpBuffer_label - 1) = label_num;
		bit_enlarge = 1;
	}
	if(*(tmpBuffer_copy - bmpWidth) == 0 && *(tmpBuffer_label - bmpWidth) == 0){
		*(tmpBuffer_label - bmpWidth) = label_num;
		bit_enlarge = 1;
	}
	if(*(tmpBuffer_copy + bmpWidth) == 0 && *(tmpBuffer_label + bmpWidth) == 0){
		*(tmpBuffer_label + bmpWidth) = label_num;
		bit_enlarge = 1;
	}
}

if(j == 0 && flag == 0){
	flag = 1;
	if(*(tmpBuffer_copy + 1) == 0 && *(tmpBuffer_label + 1) == 0){
```

```
                                        *(tmpBuffer_label + 1) = label_num;
                                        bit_enlarge = 1;
                                    }
                                    if(*(tmpBuffer_copy - bmpWidth) == 0 && *(tmpBuffer_label - bmpWidth) == 0){
                                        *(tmpBuffer_label - bmpWidth) = label_num;
                                        bit_enlarge = 1;
                                    }
                                    if(*(tmpBuffer_copy + bmpWidth) == 0 && *(tmpBuffer_label + bmpWidth) == 0){
                                        *(tmpBuffer_label + bmpWidth) = label_num;
                                        bit_enlarge = 1;
                                    }
                                }

                                if(i == 0 && flag == 0){
                                    flag = 1;
                                    if(*(tmpBuffer_copy - 1) == 0 && *(tmpBuffer_label - 1) == 0){
                                        *(tmpBuffer_label - 1) = label_num;
                                        bit_enlarge = 1;
                                    }
                                    if(*(tmpBuffer_copy + 1) == 0 && *(tmpBuffer_label + 1) == 0){
                                        *(tmpBuffer_label + 1) = label_num;
                                        bit_enlarge = 1;
                                    }
                                    if(*(tmpBuffer_copy + bmpWidth) == 0 && *(tmpBuffer_label + bmpWidth) == 0){
                                        *(tmpBuffer_label + bmpWidth) = label_num;
                                        bit_enlarge = 1;
                                    }
                                }

                                if(flag == 0){
                                    if(*(tmpBuffer_copy - 1) == 0 && *(tmpBuffer_label - 1) == 0){
                                        *(tmpBuffer_label - 1) = label_num;
                                        bit_enlarge = 1;
                                    }
                                    if(*(tmpBuffer_copy + 1) == 0 && *(tmpBuffer_label + 1) == 0){
                                        *(tmpBuffer_label + 1) = label_num;
                                        bit_enlarge = 1;
                                    }
                                    if(*(tmpBuffer_copy - bmpWidth) == 0 && *(tmpBuffer_label - bmpWidth) == 0){
                                        *(tmpBuffer_label - bmpWidth) = label_num;
                                        bit_enlarge = 1;
                                    }
                                    if(*(tmpBuffer_copy + bmpWidth) == 0 && *(tmpBuffer_label + bmpWidth) == 0){
                                        *(tmpBuffer_label + bmpWidth) = label_num;
                                        bit_enlarge = 1;
                                    }
                                }

                            }
                            tmpBuffer_label++;
                            tmpBuffer_copy++;
                        }
                    }
                }
            }
        }
    }
    printf("label_num =%d¥n",label_num);

    free(buffer_label);
    return;

}
```

付録ｂ　ＧＰＵプログラムソースコード

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include      <cutil_inline.h>

//TDI の枚数
#define TDI_num 1

//ファイル読み込み時のオフセット　通常は０
#define OFFSET 0

//ラプラシアンフィルタの結果に対するオフセット値　通常は０
#define R_OFFSET 50

__host__
static int getFileSize(FILE *fp);

__host__
static void GRAP(unsigned int fileSize,int bmpWidth,int bmpHeight,unsigned char *buffer_copy);

__global__
static void RAPkernel(int bmpWidth,int bmpHeight,unsigned char *tmpBuffer_copy2,unsigned char *tmpBuffer_copy3);

__host__
static void LAB(unsigned int fileSize,int bmpWidth,int bmpHeight,unsigned char *buffer_copy);

__host__
static void GLAB(unsigned int fileSize,int bmpWidth,int bmpHeight,unsigned char *buffer_copy,int *buffer_label,int
label_num);

__global__
static void LABkernel(int bmpWidth,int bmpHeight,unsigned char *tmpBuffer_copy2,int *tmpBuffer_copy3,int
label_num,int *buffer_copy4);
```

```
int main(void){
	FILE *readFile;		//読み込み用ファイル
	FILE *writeFile;	//書き込み用ファイル

	unsigned char *buffer_original = NULL;
	unsigned char *tmpBuffer_original = 0;

	unsigned char *buffer_copy = NULL;
	unsigned char *tmpBuffer_copy = 0;

	int *buffer_data = NULL;
	int *tmpBuffer_data = 0;

	unsigned int fileSize = 0;
	char filename[100];
	int bmpWidth = 0;
	int bmpHeight = 0;

	int tmp_name;//TDI 処理で使用

	int i = 0, j = 0, k = 0;			/*ループ用*/

	unsigned int cudaTimer = 0;
	cutilCheckError(cutCreateTimer(&cudaTimer));
	cutilCheckError(cutStartTimer(cudaTimer));


	////////////////////////////////////////
	////////////////////////////////////////
	//////////TDI 処理ここから//////////
	////////////////////////////////////////
	////////////////////////////////////////

	///////////////
	//一回目の処理//
```

```
////////////////
tmp_name = OFFSET;
//ファイルネームを生成
if(tmp_name < 10){
	sprintf(filename,"data00%d.bmp",tmp_name);
}
else if(tmp_name < 100){
	sprintf(filename,"data0%d.bmp",tmp_name);
}
else{
	sprintf(filename,"data%d.bmp",tmp_name);
}

if ((readFile= fopen(filename, "rb")) == NULL) {
	printf("ERR:ReadFile¥n");
	exit(1);
}
printf("filename = %s OK¥n",filename);

//fileSize にファイルサイズを格納
fileSize = getFileSize(readFile);
printf("filesize OK¥n");



//ファイルから読み込んだデータ格納用配列
if ((buffer_copy = (unsigned char *)malloc(sizeof(unsigned char) * fileSize)) == NULL) {
	printf("ERR:Memory¥n");
	exit(1);
}

//ファイルからバッファにコピー
fread(buffer_copy, sizeof(unsigned char), fileSize, readFile);
fclose(readFile);

//データから画像の幅と高さを得る
memcpy(&bmpWidth, buffer_copy + (sizeof(unsigned char) * 18), sizeof(unsigned char) * 4);
```

```
memcpy(&bmpHeight, buffer_copy + (sizeof(unsigned char) * 22), sizeof(unsigned char) * 4);


//ファイルから読み込んだデータの計算用配列
if ((buffer_data = (int *)malloc(sizeof(int) * fileSize)) == NULL) {
	printf("ERR:Memory¥n");
	exit(1);
}

//空の data に初期値を与えておく
tmpBuffer_data =buffer_data + (54 + 1024);
tmpBuffer_copy =buffer_copy + (54 + 1024);

for (i = 0; i < bmpHeight; i++) {
	for (j = 0; j < bmpWidth; j++) {
		*tmpBuffer_data = *tmpBuffer_copy;
		tmpBuffer_data++;
		tmpBuffer_copy++;
	}
}

//////////////////////////
//一回目の処理ここまで//
//////////////////////////

////////////////////
//２回目以降の処理//
////////////////////

//二回目以降の処理なので i=1 にしておく
i = 1;
while(i < TDI_num){
	i++;
	tmp_name = i + OFFSET;

	//ファイルネームを生成
```

```
if(tmp_name < 10){
	sprintf(filename,"data00%d.bmp",tmp_name);
}
else if(tmp_name < 100){
	sprintf(filename,"data0%d.bmp",tmp_name);
}
else{
	sprintf(filename,"data%d.bmp",tmp_name);
}

if ((readFile= fopen(filename, "rb")) == NULL) {
	printf("ERR:ReadFile¥n");
	exit(1);
}
printf("filename = %s OK¥n",filename);



//メモリ確保
//ファイルから読み込んだデータ格納用配列
if ((buffer_original = (unsigned char *)malloc(sizeof(unsigned char) * fileSize)) == NULL) {
	printf("ERR:Memory¥n");
	fclose(readFile);
	exit(1);
}

//ファイルからバッファにコピー
fread(buffer_original, sizeof(unsigned char), fileSize, readFile);
fclose(readFile);

tmpBuffer_original =buffer_original;
tmpBuffer_copy =buffer_copy;

//データから画像の幅と高さを得る

//ヘッダとパレットを読み飛ばすため buffer に 54+1024 足したアドレス
//を tmpBuffer に格納する
```

```
        tmpBuffer_original = buffer_original + (54 + 1024) + (bmpWidth * i);
        tmpBuffer_data =buffer_data + (54 + 1024);

        for (j = 0; j < bmpHeight - i; j++) {
            for (k = 0; k < bmpWidth; k++) {
                *tmpBuffer_data = *tmpBuffer_data + *tmpBuffer_original;
                tmpBuffer_original++;
                tmpBuffer_data++;
            }
        }

free(buffer_original);
}
////////////////////////
//二回目の処理ここまで//
////////////////////////

////////////////////////
//ここからまとめの処理//
////////////////////////

//合算したデータ値を TDI_num の値で割って TDI 処理を行う
if(TDI_num == 1){
        tmpBuffer_data = buffer_data + (54 + 1024);

        for (i = 0; i < bmpHeight; i++){
            for (j = 0; j < bmpWidth; j++) {
                *tmpBuffer_data = (*tmpBuffer_data / TDI_num);
                tmpBuffer_data++;
            }
        }
}

else{
        tmpBuffer_data = buffer_data + (54 + 1024);
```

```
        for (i = 0; i < bmpHeight - TDI_num; i++){
            for (j = 0; j < bmpWidth; j++) {
                *tmpBuffer_data = (*tmpBuffer_data / TDI_num);
                tmpBuffer_data++;
            }
        }

        //データが無い部分に黒挿入
        for (i = 0; i < (bmpWidth*TDI_num); i++){
            *tmpBuffer_data = 255;
            tmpBuffer_data++;
        }
    }

    //original に読み込んだ内容を copy にコピーしておく
    tmpBuffer_data =buffer_data + (54 + 1024);
    tmpBuffer_copy =buffer_copy + (54 + 1024);
    for (i = 0; i < (bmpHeight * bmpWidth); i++) {

        if(*tmpBuffer_data > 255){
            *tmpBuffer_data = 255;
        }

        if(*tmpBuffer_data < 0){
            *tmpBuffer_data = 0;
        }

        *tmpBuffer_copy = *tmpBuffer_data;
        tmpBuffer_data++;
        tmpBuffer_copy++;
    }

    GRAP(fileSize,bmpWidth,bmpHeight,buffer_copy);

    //LAB(fileSize,bmpWidth,bmpHeight,buffer_copy);
```

```
	//バイナリ書き込みモードでオープン
	if ((writeFile = fopen("a.bmp", "wb")) == NULL) {
		printf("ERR:WriteFile¥n");
		fclose(readFile);
		exit(1);
	}

	//バッファからファイルに書き込み
	fwrite(buffer_copy, sizeof(unsigned char), fileSize, writeFile);
	fclose(writeFile);

	//メモリの解放とファイルのクローズ
	free(buffer_data);
	free(buffer_copy);

	cutilCheckError(cutStopTimer(cudaTimer));
	printf("It takes %f milliseconds¥n",cutGetTimerValue(cudaTimer));
	cutilCheckError(cutDeleteTimer(cudaTimer));
	cudaThreadExit();
	return 0;
}

__host__
static int getFileSize(FILE *fp){
	fpos_t tmp_pos = 0;
	fpos_t fileSize = 0;

	tmp_pos = ftell(fp);
	fseek(fp, 0L, SEEK_SET);
	fseek(fp, 0L, SEEK_END);
	fgetpos(fp,&fileSize);

	fseek(fp, tmp_pos, SEEK_SET);

	return (int)fileSize;
}
```

```
__host__
static void GRAP(unsigned int fileSize,int bmpWidth,int bmpHeight,unsigned char *buffer_copy){
	int i;
	unsigned char *buffer_copy2 = NULL; /*データの格納用*/
	unsigned char *tmpBuffer_copy2 = 0; /*データの操作用*/
	unsigned char *buffer_copy3 = NULL; /*データの格納用*/
	unsigned char *tmpBuffer_copy3 = 0; /*データの操作用*/

	cutilSafeCall(cudaMalloc((void**)&buffer_copy2, sizeof(unsigned char) * fileSize));
	cutilSafeCall(cudaMemcpy(buffer_copy2, buffer_copy, sizeof(unsigned char) * fileSize,
cudaMemcpyHostToDevice));
	cutilSafeCall(cudaMalloc((void**)&buffer_copy3, sizeof(unsigned char) * fileSize));
	cutilSafeCall(cudaMemcpy(buffer_copy3, buffer_copy, sizeof(unsigned char) * fileSize,
cudaMemcpyHostToDevice));

	dim3	threads(32);
	dim3	grid((bmpHeight/32) + 1);

	tmpBuffer_copy2 = buffer_copy2 + (54 + 1024);
	tmpBuffer_copy3 = buffer_copy3 + (54 + 1024);

	RAPkernel<<<threads, grid>>>(bmpWidth,bmpHeight,tmpBuffer_copy2,tmpBuffer_copy3);

	cutilSafeCall(cudaMemcpy(buffer_copy, buffer_copy3, sizeof(unsigned char) * fileSize,
cudaMemcpyDeviceToHost));
	cutilSafeCall(cudaFree(buffer_copy2));
	cutilSafeCall(cudaFree(buffer_copy3));

	printf("ラプラシアン OK¥n");
	return;
}

__global__
static void RAPkernel(int bmpWidth,int bmpHeight,unsigned char *tmpBuffer_copy2,unsigned char *tmpBuffer_copy3){
	int j;
```

```
    int flag = 0;
    int tmp_int;
    unsigned char *tmpBuffer_copy4 = 0;
    unsigned char *tmpBuffer_copy5 = 0;
    int Yg;
    tmpBuffer_copy4 = tmpBuffer_copy2 + (blockIdx.x * blockDim.x + threadIdx.x) * bmpWidth;
    tmpBuffer_copy5 = tmpBuffer_copy3 + (blockIdx.x * blockDim.x + threadIdx.x) * bmpWidth;
    Yg = blockIdx.x * blockDim.x + threadIdx.x;

if(Yg < bmpHeight){
    // もし inHeight 以上の Y 軸をスキャンしようとした場合、終了させる
        // X 軸を 0 から inWidth - 1 までスキャンする
        for (j = 0;j < bmpWidth; j++) {
            if(Yg == 0 && j == 0){
                //縦の 1 列目の一番左の bit のための処理
                tmp_int = 2 * (*tmpBuffer_copy4) - *(tmpBuffer_copy4+1) - *(tmpBuffer_copy4+bmpWidth);
                flag = 1;
            }

            if(Yg == 0 && j == (bmpWidth -1)){
                //縦の 1 列目の一番右の bit のための処理
                tmp_int = 2 * (*tmpBuffer_copy4) - *(tmpBuffer_copy4-1) - *(tmpBuffer_copy4+bmpWidth);
                flag = 1;
            }

            if(Yg == (bmpHeight - 1) && j == 0){
                //縦の最終列の一番左の bit のための処理
                tmp_int = 2 * (*tmpBuffer_copy4) - *(tmpBuffer_copy4+1) - *(tmpBuffer_copy4-bmpWidth);
                flag = 1;
            }

            if(Yg == (bmpHeight - 1) && j == (bmpWidth -1)){
                //縦の最終列の一番右 bit のための処理（n は任意）
                tmp_int = 2 * (*tmpBuffer_copy4) - *(tmpBuffer_copy4-1) - *(tmpBuffer_copy4-bmpWidth);
                flag = 1;
            }
```

```
            if(Yg == 0 && flag == 0){
                //縦の 1 列目の一番右と一番左以外の bit のための処理
                tmp_int = 3 * (*tmpBuffer_copy4) - *(tmpBuffer_copy4-1) - *(tmpBuffer_copy4+1) -
*(tmpBuffer_copy4+bmpWidth);
                flag = 1;
            }

            if(j == 0 && flag == 0){
                //縦の n 列目の一番左 bit のための処理（n は任意）
                tmp_int = 3 * (*tmpBuffer_copy4) - *(tmpBuffer_copy4+1) - *(tmpBuffer_copy4-bmpWidth) -
*(tmpBuffer_copy4+bmpWidth);
                flag = 1;
            }

            if(j == (bmpWidth - 1) && flag == 0){
                //縦の n 列目の一番右 bit のための処理（n は任意）
                tmp_int = 3 * (*tmpBuffer_copy4) - *(tmpBuffer_copy4-1) - *(tmpBuffer_copy4-bmpWidth) -
*(tmpBuffer_copy4+bmpWidth);
                flag = 1;
            }

            if(Yg == (bmpHeight - 1) && flag == 0){
                //縦の最終列の一番右と一番左以外のための処理（n は任意）
                tmp_int = 3 * (*tmpBuffer_copy4) - *(tmpBuffer_copy4-1) - *(tmpBuffer_copy4+1) -
*(tmpBuffer_copy4-bmpWidth);
                flag = 1;
            }

            if(flag == 0){
                //特別な処理を必要としない bit の処理
                tmp_int = 4 * (*tmpBuffer_copy4) - *(tmpBuffer_copy4-1) - *(tmpBuffer_copy4+1) -
*(tmpBuffer_copy4-bmpWidth) - *(tmpBuffer_copy4+bmpWidth);
            }

            if(tmp_int > 255){
```

```
                tmp_int = 255;
            }

            if(tmp_int < 0){
                tmp_int = 0;
            }
            *tmpBuffer_copy5 = tmp_int + R_OFFSET;
            tmpBuffer_copy4++;//進める
            tmpBuffer_copy5++;//進める 2
            flag = 0;
        }
    }
    __syncthreads();
}


__host__
static void LAB(unsigned int fileSize,int bmpWidth,int bmpHeight,unsigned char *buffer_copy){
    unsigned char *tmpBuffer_copy = 0;
    int *buffer_label = NULL;
    int *tmpBuffer_label = 0;
    int label_num;//ラベリング処理で使用
    int bit_discover;//ラベリング処理で使用
    int i = 0, j = 0;

    //２値化処理
    tmpBuffer_copy =buffer_copy + (54 + 1024);

    for (i = 0; i < bmpHeight; i++) {
        for (j = 0; j < bmpWidth; j++) {
            if(*tmpBuffer_copy <= 6){
                *tmpBuffer_copy = 255;
            }
            else{
                *tmpBuffer_copy = 0;
            }
```

```
            tmpBuffer_copy++;
        }
    }

    //ラベリング用のメモリ確保
    if ((buffer_label = (int *)malloc(sizeof(int) * fileSize)) == NULL) {
        printf("ERR:Memory¥n");
        exit(1);
    }

    //ラベル配列をゼロクリア
    tmpBuffer_label = buffer_label + (54 + 1024);
    for (i = 0; i < bmpHeight; i++) {
        for (j = 0; j < bmpWidth; j++) {
            *tmpBuffer_label = 0;
            tmpBuffer_label++;
        }
    }

    label_num = 0;
    bit_discover = 1;
    while(bit_discover == 1){
        bit_discover = 0;
        tmpBuffer_label = buffer_label + (54 + 1024);
        tmpBuffer_copy = buffer_copy + (54 + 1024);

        //新規ビット発見用ループ
        for (i = 0; i < bmpHeight; i++) {
            for (j = 0; j < bmpWidth; j++) {
                if(*tmpBuffer_copy == 0 && *tmpBuffer_label == 0){
                    label_num++;
                    *tmpBuffer_label = label_num;
                    printf("label = %d¥n",label_num);
                    bit_discover = 1;
                }
                if(bit_discover == 1){
```

```
                    break;
                }
                tmpBuffer_label++;
                tmpBuffer_copy++;
            }
            if(bit_discover == 1){
                break;
            }
        }

        //発見したビットを拡張するループ
        if(bit_discover == 1){
            tmpBuffer_copy = buffer_copy + (54 + 1024);
            tmpBuffer_label = buffer_label + (54 + 1024);
            GLAB(fileSize,bmpWidth,bmpHeight,tmpBuffer_copy,tmpBuffer_label,label_num);
        }
    }
    free(buffer_label);
    return;
}


__host__
static void GLAB(unsigned int fileSize,int bmpWidth,int bmpHeight,unsigned char *buffer_copy,int *buffer_label,int label_num){
    unsigned char *buffer_copy2 = NULL;
    int *buffer_copy3 = NULL;
    int *buffer_flag = NULL;
    int *buffer_flag_zero = NULL;
    int *tmp_buffer_flag_zero = NULL;
    int bit_enlarge = 1;
    int i;

    cutilSafeCall(cudaMalloc((void**)&buffer_copy2, sizeof(unsigned char) * fileSize));
    cutilSafeCall(cudaMemcpy(buffer_copy2, buffer_copy, sizeof(unsigned char) * fileSize, cudaMemcpyHostToDevice));
```

```
    cutilSafeCall(cudaMalloc((void**)&buffer_copy3, sizeof(int) * fileSize));
    cutilSafeCall(cudaMemcpy(buffer_copy3, buffer_label, sizeof(int) * (fileSize - 1078), cudaMemcpyHostToDevice));
    cutilSafeCall(cudaMalloc((void**)&buffer_flag, sizeof(int) * ((bmpHeight/256) + 1) * 256));

    dim3    threads(256);
    dim3    grid((bmpHeight/256) + 1);

    if ((buffer_flag_zero = (int *)malloc(sizeof(int) * ((bmpHeight/256) + 1) * 256)) == NULL) {
        printf("ERR:Memory¥n");
        exit(1);
    }

    while(bit_enlarge != 0){
        bit_enlarge = 0;
        tmp_buffer_flag_zero = buffer_flag_zero;
        for (i = 0;i < ((bmpHeight/256) + 1) * 256; i++){
            *tmp_buffer_flag_zero = 0;
            //printf("*tmp_buffer_flag_zero = %d¥n",*tmp_buffer_flag_zero);
            tmp_buffer_flag_zero++;
        }
        //printf("ラベリング OK¥n");
        cutilSafeCall(cudaMemcpy(buffer_flag, buffer_flag_zero, sizeof(int) * ((bmpHeight/256)+1) * 256,
cudaMemcpyHostToDevice));
        LABkernel<<<threads, grid>>>(bmpWidth,bmpHeight,buffer_copy2,buffer_copy3,label_num,buffer_flag);
        cutilSafeCall(cudaMemcpy(buffer_flag_zero, buffer_flag, sizeof(int) * ((bmpHeight/256)+1) * 256,
cudaMemcpyDeviceToHost));

        tmp_buffer_flag_zero = buffer_flag_zero;
        for (i = 0; i < ((bmpHeight/256)+1)*256;i++){
            //printf("*tmp_buffer_flag_zero = %d¥n",*tmp_buffer_flag_zero);
            if(*tmp_buffer_flag_zero != 0){
                bit_enlarge = 1;
            }
            tmp_buffer_flag_zero++;
        }
```

```
    }
    cutilSafeCall(cudaMemcpy(buffer_copy, buffer_copy2, sizeof(unsigned char) * fileSize,
cudaMemcpyDeviceToHost));
    cutilSafeCall(cudaMemcpy(buffer_label, buffer_copy3, sizeof(int) * (fileSize - 1078), cudaMemcpyDeviceToHost));
    cutilSafeCall(cudaFree(buffer_copy2));
    cutilSafeCall(cudaFree(buffer_copy3));
    return;
}

__global__
static void LABkernel(int bmpWidth,int bmpHeight,unsigned char *buffer_data,int *buffer_label,int label_num,int
*buffer_flag){
    unsigned char *tmpBuffer_data = 0;   /*データの操作用*/
    int *tmpBuffer_label = 0;    /*データの操作用*/
    int *tmpBuffer_flag = 0;     /*データの操作用*/
    int flag = 0; //フラグ用(ラプラシアン・ラベリング)
    int j = 0;                /*ループ用*/
    int Yg;

    tmpBuffer_data = buffer_data + (blockIdx.x * blockDim.x + threadIdx.x) * bmpWidth;
    tmpBuffer_label = buffer_label + (blockIdx.x * blockDim.x + threadIdx.x) * bmpWidth;
    Yg = blockIdx.x * blockDim.x + threadIdx.x;
    tmpBuffer_flag = buffer_flag + (blockIdx.x * blockDim.x + threadIdx.x);

                    for (j = 0; j < bmpWidth; j++) {
                        if(*tmpBuffer_data == 0 && *tmpBuffer_label == label_num){
                            flag = 0;

                            //画素が０、ラベルが label_num と一致する画素を見つけた→ラベル拡張する場合の処
理
                            if(Yg == 0 && j == bmpWidth - 1){
                                flag = 1;
                                if(*(tmpBuffer_data - 1) == 0 && *(tmpBuffer_label - 1) == 0){
                                    *(tmpBuffer_label - 1) = label_num;
                                    *tmpBuffer_flag = 1;
                                }
```

```
        if(*(tmpBuffer_data + bmpWidth) == 0 && *(tmpBuffer_label + bmpWidth) == 0){
            *(tmpBuffer_label + bmpWidth) = label_num;
            *tmpBuffer_flag = 1;
        }
    }

    if(Yg == 0 && j == 0){
        flag = 1;
        if(*(tmpBuffer_data + 1) == 0 && *(tmpBuffer_label + 1) == 0){
            *(tmpBuffer_label + 1) = label_num;
            *tmpBuffer_flag = 1;
        }
        if(*(tmpBuffer_data + bmpWidth) == 0 && *(tmpBuffer_label + bmpWidth) == 0){
            *(tmpBuffer_label + bmpWidth) = label_num;
            *tmpBuffer_flag = 1;
        }
    }

    if(Yg == (bmpHeight - 1) && j == (bmpWidth -1)){
        flag = 1;
        if(*(tmpBuffer_data - 1) == 0 && *(tmpBuffer_label - 1) == 0){
            *(tmpBuffer_label - 1) = label_num;
            *tmpBuffer_flag = 1;
        }
        if(*(tmpBuffer_data - bmpWidth) == 0 && *(tmpBuffer_label - bmpWidth) == 0){
            *(tmpBuffer_label - bmpWidth) = label_num;
            *tmpBuffer_flag = 1;
        }
    }

    if(Yg == (bmpHeight - 1) && j == 0){
        flag = 1;
        if(*(tmpBuffer_data + 1) == 0 && *(tmpBuffer_label + 1) == 0){
            *(tmpBuffer_label + 1) = label_num;
            *tmpBuffer_flag = 1;
        }
```

```
        if(*(tmpBuffer_data - bmpWidth) == 0 && *(tmpBuffer_label - bmpWidth) == 0){
            *(tmpBuffer_label - bmpWidth) = label_num;
            *tmpBuffer_flag = 1;
        }
    }

    if(Yg == (bmpHeight - 1) && flag == 0){
        flag = 1;
        if(*(tmpBuffer_data - 1) == 0 && *(tmpBuffer_label - 1) == 0){
            *(tmpBuffer_label - 1) = label_num;
            *tmpBuffer_flag = 1;
        }
        if(*(tmpBuffer_data + 1) == 0 && *(tmpBuffer_label + 1) == 0){
            *(tmpBuffer_label + 1) = label_num;
            *tmpBuffer_flag = 1;
        }
        if(*(tmpBuffer_data - bmpWidth) == 0 && *(tmpBuffer_label - bmpWidth) == 0){
            *(tmpBuffer_label - bmpWidth) = label_num;
            *tmpBuffer_flag = 1;
        }
    }

    if(j == (bmpWidth - 1) && flag == 0){
        flag = 1;
        if(*(tmpBuffer_data - 1) == 0 && *(tmpBuffer_label - 1) == 0){
            *(tmpBuffer_label - 1) = label_num;
            *tmpBuffer_flag = 1;
        }
        if(*(tmpBuffer_data - bmpWidth) == 0 && *(tmpBuffer_label - bmpWidth) == 0){
            *(tmpBuffer_label - bmpWidth) = label_num;
            *tmpBuffer_flag = 1;
        }
        if(*(tmpBuffer_data + bmpWidth) == 0 && *(tmpBuffer_label + bmpWidth) == 0){
            *(tmpBuffer_label + bmpWidth) = label_num;
            *tmpBuffer_flag = 1;
        }
```

```
}

if(j == 0 && flag == 0){
	flag = 1;
	if(*(tmpBuffer_data + 1) == 0 && *(tmpBuffer_label + 1) == 0){
		*(tmpBuffer_label + 1) = label_num;
		*tmpBuffer_flag = 1;
	}
	if(*(tmpBuffer_data - bmpWidth) == 0 && *(tmpBuffer_label - bmpWidth) == 0){
		*(tmpBuffer_label - bmpWidth) = label_num;
		*tmpBuffer_flag = 1;
	}
	if(*(tmpBuffer_data + bmpWidth) == 0 && *(tmpBuffer_label + bmpWidth) == 0){
		*(tmpBuffer_label + bmpWidth) = label_num;
		*tmpBuffer_flag = 1;
	}
}

if(Yg == 0 && flag == 0){
	flag = 1;
	if(*(tmpBuffer_data - 1) == 0 && *(tmpBuffer_label - 1) == 0){
		*(tmpBuffer_label - 1) = label_num;
		*tmpBuffer_flag = 1;
	}
	if(*(tmpBuffer_data + 1) == 0 && *(tmpBuffer_label + 1) == 0){
		*(tmpBuffer_label + 1) = label_num;
		*tmpBuffer_flag = 1;
	}
	if(*(tmpBuffer_data + bmpWidth) == 0 && *(tmpBuffer_label + bmpWidth) == 0){
		*(tmpBuffer_label + bmpWidth) = label_num;
		*tmpBuffer_flag = 1;
	}
}

if(flag == 0){
	if(*(tmpBuffer_data - 1) == 0 && *(tmpBuffer_label - 1) == 0){
```

```
                    *(tmpBuffer_label - 1) = label_num;
                    *tmpBuffer_flag = 1;
                }
                if(*(tmpBuffer_data + 1) == 0 && *(tmpBuffer_label + 1) == 0){
                    *(tmpBuffer_label + 1) = label_num;
                    *tmpBuffer_flag = 1;
                }
                if(*(tmpBuffer_data - bmpWidth) == 0 && *(tmpBuffer_label - bmpWidth) == 0){
                    *(tmpBuffer_label - bmpWidth) = label_num;
                    *tmpBuffer_flag = 1;
                }
                if(*(tmpBuffer_data + bmpWidth) == 0 && *(tmpBuffer_label + bmpWidth) == 0){
                    *(tmpBuffer_label + bmpWidth) = label_num;
                    *tmpBuffer_flag = 1;
                }
            }

        }
        tmpBuffer_label++;
        tmpBuffer_data++;
    }
    __syncthreads();

}
```