卒業研究論文

GPU プログラミングを支援する
ループレス画像処理記述言語の提案と実装

指導教員　　津邑 公暁 准教授
松尾 啓志 教授

名古屋工業大学 工学部 情報工学科
平成 19 年度入学　19115034 番

小野 亜実

平成 24 年 2 月 8 日

GPU プログラミングを支援する
ループレス画像処理記述言語の提案と実装

小野 亜実

内容梗概

情報技術の発達に伴い，携帯電話，デジタルカメラ，監視装置，認証装置など，多くの情報機器が開発されている．これらの情報機器の多くは静止画や動画を扱う機能を備えているため，動画像処理プログラムを開発する機会も増加している．しかし，各プラットフォームにおいて，重要視される処理精度や出力デバイスの解像度などは異なるため，動画像処理アプリケーションを異なるプラットフォームへ移植する際は，一般にプログラムを書き換えなければならない．

一方で，GPU 向けの C 言語統合開発環境として CUDA（Compute Unified Device Architecture）が開発されている．CUDA は GPU 上に存在する複数のコアに大量のスレッドを割り当て，その大量のスレッドに同じ命令を並列に実行させることで処理の高速化を実現する．そのため，一般的にデータ並列性を持つ動画像処理は，GPU 上で実行することにより高い性能が期待できる．しかし CUDA を用いた動画像処理アプリケーションの開発には，GPU や CUDA に関する深い知識が必要であるため，プログラマにとって負担が大きい．

そこで，解像度非依存型の動画像処理ライブラリ RaVioli と，RaVioli を CUDA に対応するよう改良した RaVioli/CUDA が提案されている．RaVioli は，プログラマから画像や画素を隠蔽し，解像度をライブラリ内部で制御することにより，直観的な動画像処理プログラミングを実現している．さらに，プラットフォームにあわせて動画像処理プログラムを書き換える必要もない．しかし，画像や画素のカプセル化のオーバーヘッドにより，処理速度が低下してしまうという問題がある．そこで RaVioli/CUDA は，CUDA を利用することにより RaVioli の高速化を実現している．しかし，RaVioli と RaVioli/CUDA に共通する問題として，画像を処理するために提供されている RaVioli の複数の高階メソッドを，適切に使い分けなければならないという点がある．また，RaVioli/CUDA を利用するには，CUDA に関するいくつかの処理を追加しなければならず，やはり CUDA の基本的な知識が必要となる．

そこで本論文では，より直感的に画像処理プログラムを記述できる新しい言語を提案する．さらに，提案言語で記述した画像処理プログラムを，RaVioli で記述されたプログラムと，そのプログラムをさらに RaVioli/CUDA で記述されたプログラムへ変換

するトランスレータも提案する．トランスレータによる自動変換により，プログラマに CUDA の知識がなくても，CUDA を用いた処理の並列化が可能となる．

いくつかの画像処理プログラムを用いて，提案言語，RaVioli，C++で記述したプログラムのサイズを比較した．その結果，提案言語により記述されたプログラムは，RaVioli や C++を用いたプログラムに比べて，行数，バイト数共に削減できることを確認した．また，プログラマが記述したプログラムと，トランスレータによって生成されたプログラムの実行時間を比較した結果，生成された RaVioli/CUDA プログラムの実行時間は，RaVioli プログラムの実行時間に比べて，テンプレートマッチングにおいて最大で 177.1 倍の速度向上が達成できることを確認した．

GPU プログラミングを支援する
ループレス画像処理記述言語の提案と実装

目次

1 はじめに

情報技術の発達に伴い，携帯電話，デジタルカメラ，監視装置，認証装置など，多くの情報機器が開発されている．これらの情報機器の多くは静止画や動画を扱う機能を備えているため，動画像処理プログラムを開発する機会も増加している．しかし，各プラットフォームにおいて，重要視される処理精度や出力デバイスの解像度などは異なるため，そのプラットフォームの目的にあわせたプログラムの開発が必要である．そのため，動画像処理アプリケーションを異なるプラットフォームへ移植する際は，一般にプログラムを書き換えなければならない．

一方で，GPU 向けの C 言語統合開発環境として **CUDA (Compute Unified Device Architecture)** が開発されている．CUDA は GPU 上に存在する複数のコアに大量のスレッドを割り当て，その大量のスレッドに同じ命令を並列に実行させることで処理の高速化を実現する．そのため，一般的にデータ並列性を持つ動画像処理は，GPU 上で実行することにより高い性能が期待できる．しかし CUDA を利用するためには，GPU のアーキテクチャ，および CUDA のプログラミングモデルや機能に関する深い知識が必要である．そのため，CUDA を用いた動画像処理アプリケーションの開発はプログラマにとって負担が大きい．

これら 2 つの問題を解決するため，解像度非依存型の動画像処理ライブラリ **RaVioli (Resolution-Adaptable Video and Image Operating Library)** と，RaVioli を CUDA に対応するよう改良した RaVioli/CUDA が提案されている．RaVioli は，プログラマから画像や画素を隠蔽し，解像度をライブラリ内部で制御することにより，直観的な動画像処理プログラミングを実現している．さらに，目的の処理にあわせて内部で処理量を自動調整することができるため，プラットフォームにあわせて動画像処理プログラムを書き換える必要がない．しかし，画像や画素のカプセル化のオーバーヘッドにより，処理速度が低下してしまうという問題がある．そこで RaVioli/CUDA は，CUDA を利用することにより RaVioli の高速化を実現している．CUDA による並列化に必要となるメモリ管理などもライブラリ内部で行うため，並列化を考慮することなく動画像処理を記述することができる．しかし，RaVioli と RaVioli/CUDA に共通する問題として，画像を処理するために提供されている RaVioli の複数の高階メソッドを，適切に使い分けなければならないという点がある．また，RaVioli/CUDA を利用するには，CUDA に関するいくつかの処理を追加しなければならず，やはり CUDA の基本的な知識が必要となる．

そこで本論文では，より直感的に画像処理プログラムを記述できる新しい言語を提案する．さらに，提案言語で記述した画像処理プログラムを，RaVioli で記述されたプログラムと，そのプログラムをさらに RaVioli/CUDA で記述されたプログラムへ変換するトランスレータも提案する．トランスレータによる自動変換により，プログラマに CUDA の知識がなくとも，CUDA を用いた処理の並列化が可能となる．

以下，本論文では，2 章で本研究の背景と動画像処理ライブラリ RaVioli，さらに CUDA に対応した RaVioli/CUDA について述べ，3 章でループレスな画像処理記述言語を提案する．4 章で提案言語で記述したプログラムを変換するトランスレータについて述べ，5 章で提案言語やトランスレータを評価し，最後に 6 章で本論文のまとめと今後の課題を述べる．

2　研究背景

本章では従来の動画像処理に存在する問題点を述べ，それらに対する既存研究について説明する．また，本提案の基盤となる動画像処理ライブラリ RaVioli の特徴と問題点を述べる．

2.1　関連研究

ビデオカメラなどの入力機器と計算機間のデータ転送が高速になり，リアルタイムに画像を取り込む環境が整ってきている．また，汎用 PC の性能は向上し，価格は低下しているため，高い演算能力をもつ PC を容易に入手可能になってきている．そのため，今後，汎用 OS や汎用 PC 上にリアルタイム動画像処理システムが盛んに開発されると予想される．

しかし，汎用 PC 上でリアルタイム動画像処理に必要な計算資源を常に確保することは困難である．その原因として，1 フレーム当たりの処理量が入力によって変化することや，利用可能な CPU リソース量の変動が挙げられる．汎用 OS 上では複数の他プロセスが動画像処理プロセスと並行に実行されており，動画像処理に必要な CPU リソース量を常に確保できるという保証はない．

この問題を解決するには，利用可能な CPU リソース量に応じて動画像の処理量を調整する必要がある．Imprecise Computation Model [1] は計算時間の長さに応じて処理精度を変化させるモデルである．また，このモデルに基づき，処理精度および処理時間に関する知識を利用し，適切なアルゴリズムを動的に選択するアーキテクチャも提案されている [2]．しかしこの方法では，処理精度を変化させるために，あらかじめ

複数のルーチンを定義しなければならず，プログラマの負担が大きくなる．

また，計算機を使って画像を処理する際には解像度という概念が不可欠である．従来の動画像処理では画像の大きさを考慮して，画像全体のピクセルに対して処理を記述する．しかし，画像処理プログラムを記述する人間には，解像度という概念は不自然である．そもそも人間が物体を認識する過程には解像度という概念は存在しないため，解像度を意識した画像処理プログラミングは直感的でない．

そこで，擬似的にリアルタイム性を保証する解像度非依存型動画像処理ライブラリ RaVioli[3, 4] が提案されている．RaVioli では利用可能な CPU リソース量の不足によりリアルタイム処理が困難になった場合，解像度を自動調節することで処理量を調整し，リアルタイム性を保証する．また，動的に解像度を変動させるためにプログラマから解像度を隠蔽している．これによりプログラマは解像度を意識することなく処理を記述できる．

一方で，動画像処理プログラミングを抽象化するためのライブラリはこれまでにも提案されている．その中でもよく知られた動画像処理ライブラリに VIGRA[5] や OpenCV[6, 7] がある．VIGRA は C++の STL と同様にテンプレートを用い，プログラマに抽象的な処理を提供している．また，OpenCV は多くの動画像処理アルゴリズムを C 言語の関数や C++のメソッドとして提供している．しかし，これらは単純に処理内容を抽象化してライブラリの形で提供しているものであり，プログラマは解像度を意識することなく画像に対する処理内容を記述することはできない．そのため，これらのライブラリが行う画像処理の抽象化は，RaVioli とはまったく異なっている．

また金井らは数式エディタを用いて記述できる独自の言語を提案し，画像処理プログラミングを抽象化している [8, 9]．処理単位となる画素配列の大きさを定義し，その配列の要素に対して処理を記述することでループレスな記述ができるという点は RaVioli と似ているが，この記述言語は構成画素数を明示的に指定する必要があるため，動的な構成画素数の変動には対応していない．

2.2 RaVioli

この節では，まず，RaVioli の特徴である動画像処理の抽象化と処理量の自動調整について述べる．そして，RaVioli が持つ問題点について述べる．

2.2.1 動画像処理の抽象化

RaVioli ではプログラマから動画像の解像度を隠蔽することで，抽象的なプログラミングを可能にしている．ここで解像度とは空間解像度と時間解像度の 2 つを意味して

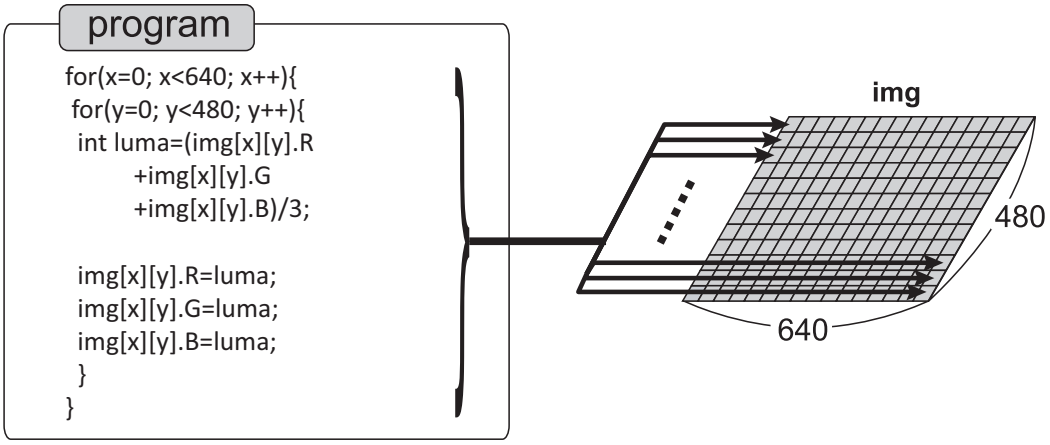

図 1: 一般的な動画像処理プログラム記述例

おり，空間解像度は 1 フレームを構成する画素数を，また時間解像度はフレームレートを意味している．この 2 つの解像度を隠蔽することで，プログラマは動画像のフレーム数や画像の幅，高さを考慮した記述を省略できる．

RaVioli を使用しない一般的な画像処理では，**図 1** のように，各画素を変換する処理は最も内側のループ内に記述され，この処理が画像中の全ての画素に繰り返し適用される．このように，量子化された動画像データを扱うためにループがよく用いられるが，ループを用いる画像処理では，プログラマは画像の幅と高さを意識した記述をしなければならない．

一方，RaVioli では，画像の幅，高さや画素配列を RV_Image クラスに，画像を構成する画素の色情報などを RV_Pixel クラスにそれぞれカプセル化している．そして，画像の構成要素である画素や部分画像，また動画の構成要素である単一フレームに対する処理のみを関数として定義し，その関数を RaVioli が提供しているメソッドに渡すことで，動画像中の全ての構成要素に処理を施すことが可能である．RaVioli ではこの構成要素に対する処理を記述した関数を**構成要素関数**と呼び，その構成要素関数を引数にとるメソッドを**高階メソッド**と呼ぶ．ここで，RaVioli を用いて画像をグレースケールに変換する処理の様子を**図 2** に示す．1 画素をグレースケール化する処理を記述した関数 GrayScale() を定義し，この関数を RV_Image インスタンスの高階メソッド procPix() に渡す．高階メソッドの procPix() が，RV_Image インスタンスが持つ画像の全ての画素に GrayScale() を繰り返し適用することで，図 1 と同様の処理が実現できる．このような処理構造を用いることで，プログラマは解像度や繰り返し処理を意識することなく画像処理プログラムを記述できる．

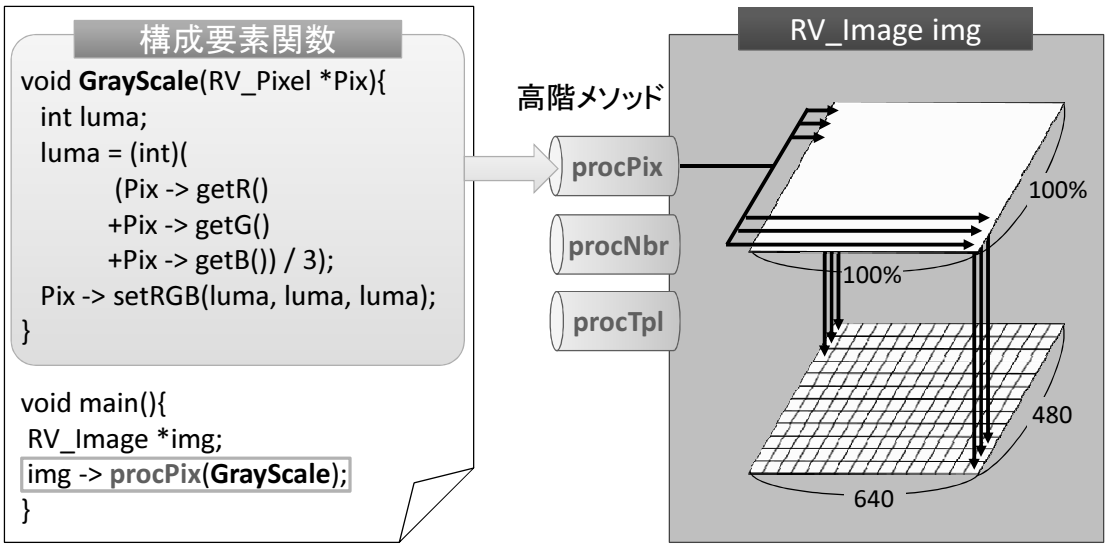

図 2: RaVioli 使用時プログラム記述例

2.2.2 処理量の自動調整

2.1 節で述べたように，汎用 OS 上で動画像処理に必要な CPU リソースを常に確保することは困難である．そこで，これを解決する方法として，動画像の解像度を低減させて処理量を減らすことが考えられる．RaVioli は解像度をプログラマから隠蔽したことにより，ライブラリ内部で負荷に応じて処理解像度を動的に変動させることが可能になった．

RaVioli は空間解像度と時間解像度を制御するために，1 フレーム上で処理する画素の間隔を示す**空間解像度ストライド** (S_S) と，処理対象のフレームの間隔を示す**時間解像度ストライド** (S_T) を持っている．これらのストライドを増減させることにより空間解像度および時間解像度を変動させている．

ここで，空間解像度を変動させるときの処理方法を**図 3** に示す．空間解像度ストライド $S_S = 1$ のとき，画像中の全ての画素が処理される．空間解像度ストライドを増加させ $S_S = 2$ となると，処理対象画素は 1 つおきとなり，空間解像度が低減する．このとき，全体の処理画素数は $S_S = 1$ のときの $1/4$ となる．さらに空間解像度ストライドを増加させ $S_S = 3$ とすると，処理画素数は $1/9$ となる．

一方，時間解像度を変動させるときの処理方法を**図 4** に示す．時間解像度ストライド $S_T = 1$ のとき，入力フレーム全てを処理する．時間解像度ストライドを増加させ $S_T = 2$ となると，処理対象フレームは 1 つおきとなり，時間解像度が低減する．この

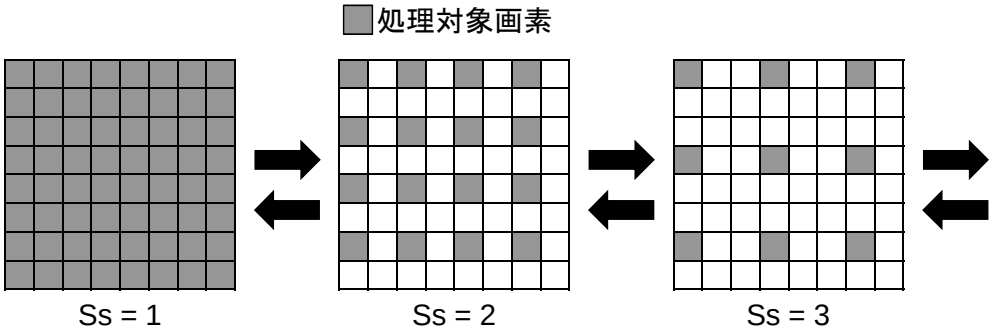

図 3: 空間解像度ストライド (S_S) の変更

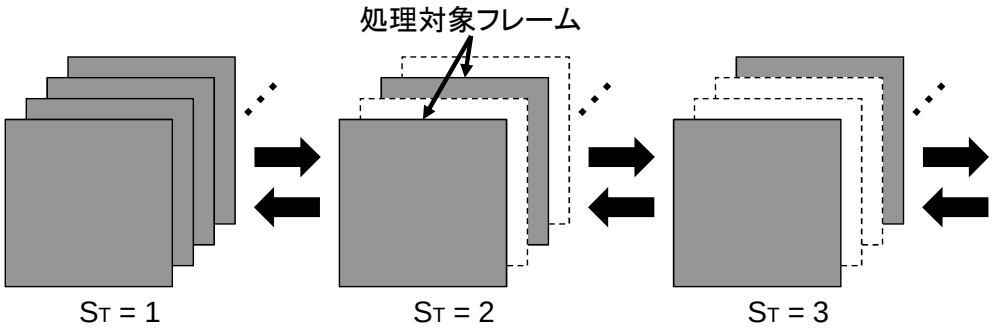

図 4: 時間解像度ストライド (S_T) の変更

とき，全体の処理フレーム数は $S_T = 1$ のときの 1/2 となる．さらに時間解像度ストライドを増加させ $S_T = 3$ とすると，処理フレーム数は 1/3 となる．

また，プログラマは空間解像度および時間解像度に対する優先度を指定することができ，RaVioli は指定された優先度の比に応じて解像度を維持する．これにより，プログラマは処理内容に応じて優先度を設定するだけで目的のプラットフォームに適したアプリケーションの作成が可能となる．例えば，時間分解能の重要な処理では，時間解像度が優先されるように設定することで，RaVioli は空間解像度を優先的に低減させる．一方，顔認証などの空間分解能の重要な処理では，空間解像度が優先されるように設定することで，時間解像度が優先的に低減され，精細さを確保しつつリアルタイム性を実現することができる．

2.2.3 問題点

2.2.1 項で述べたように，RaVioli はプログラマから動画像の解像度を隠蔽することで，より抽象的なプログラミングを可能にしている．また，利用可能な CPU リソース量にあわせて処理解像度を変動させ，擬似的なリアルタイム処理も実現している．しかし，画像や画素のカプセル化のオーバーヘッドにより処理速度が低下してしまうと

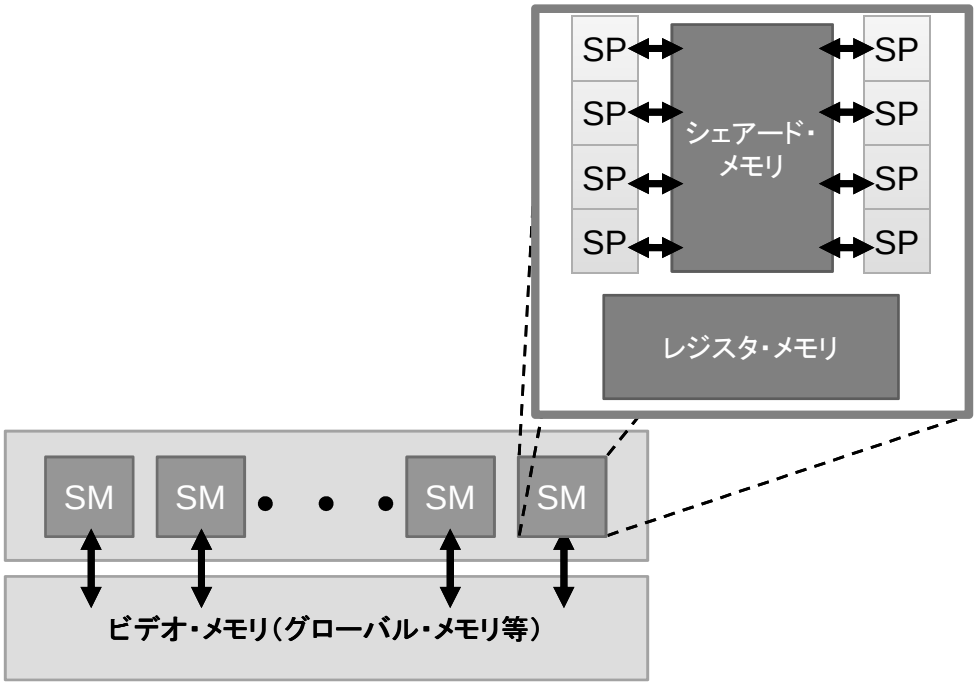

図 5: GPU のアーキテクチャ

いう問題がある．そこで，RaVioli を高速化するために，CUDA に対応するよう改良した RaVioli/CUDA がある．次節ではこれについて説明する．

2.3 RaVioli の CUDA への対応

この節では，CUDA について触れ，RaVioli/CUDA とその問題点について述べる．

2.3.1 CUDA

現在，GPU 向けの C 言語統合開発環境として CUDA[10, 11] が NVIDIA 社により開発されている．GPU (Graphics Processing Unit) は画像処理に特化した専用プロセッサであり，広域なメモリバンド幅や高い演算性能を持つ．

図 5 に NVIDIA 社の GPU のアーキテクチャを示す．GPU の世代や製品により多少の違いはあるが，そのアーキテクチャのモデルはほぼ同じである．GPU チップの内部には多数のストリーミング・マルチプロセッサ (SM) が存在する．さらに各 SM 内には演算処理ユニットであるストリーミング・プロセッサ (SP) が 8 個存在する．この SM 内の 8 個の SP は，SIMD 型で同じ命令を実行する．一般的に画像処理はデータ並列性を持っており，SIMD 型を採用している GPU 上で高い性能が期待できる．以下では，CUDA の階層的なスレッド管理やメモリモデル，プログラミングモデルについて述べる．

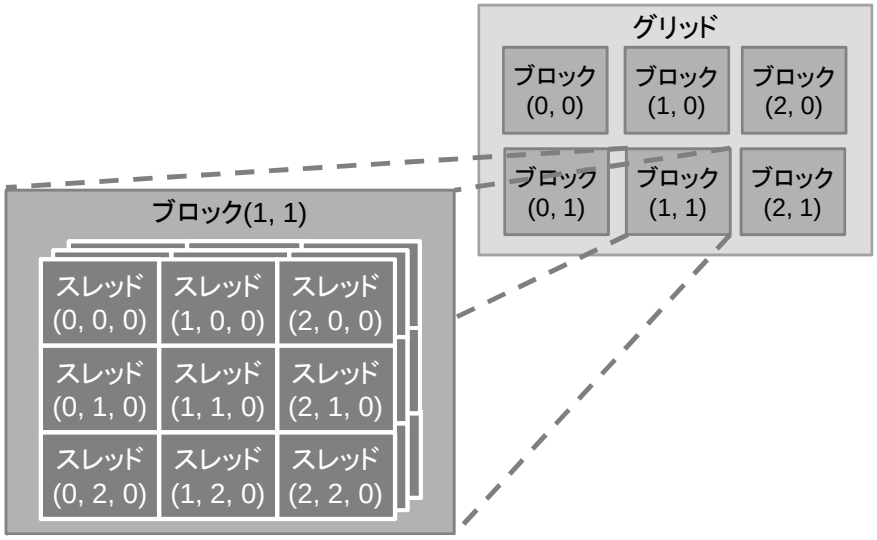

図 6: 階層的なスレッド管理 (実行構成)

階層的なスレッド管理

GPU は，SP を用いて複数のスレッドを並列実行することで高い演算性能を実現する．CUDA の仕様では，GPU の全ての SP で最大 65535×65535×512 個のスレッドを使用することが可能である．この大量のスレッドは**図 6** のように階層的に管理される．スレッドの集合をブロックと呼び，ブロック内でスレッドは x 軸方向，y 軸方向，z 軸方向の 3 次元的に配置されている．また同数のスレッドから成るブロックの集合をグリッドと呼び，グリッド内でブロックは 2 次元的に配置されている．このようなグリッドの次元やグリッド内のブロック数，およびブロックの次元やブロック内のスレッド数を**実行構成**と呼ぶ.

メモリモデル

CUDA のメモリモデルを**図 7** に示す．CUDA では，レジスタ・メモリ，ローカル・メモリ，シェアード・メモリ，グローバル・メモリ，テクスチャ・メモリ，コンスタント・メモリが使用できる．これらのメモリは全てグラフィック・ボード上に存在し，まとめてデバイス・メモリと呼ばれる．これらの詳細を**表 1** に示す．表中のアクセス欄はデバイスからのアクセスを表し，R/W は読み書き可能，R は読み出しのみ可能を表す．各メモリの物理的な場所，アクセスやキャッシュの有無などは異なるため，処理の特性に応じて適切なメモリを使用することにより，処理速度の向上が期待できる．各メモリを利用するためには，それぞれに用意されたアクセス関数や変数修飾子を使わなければならない.

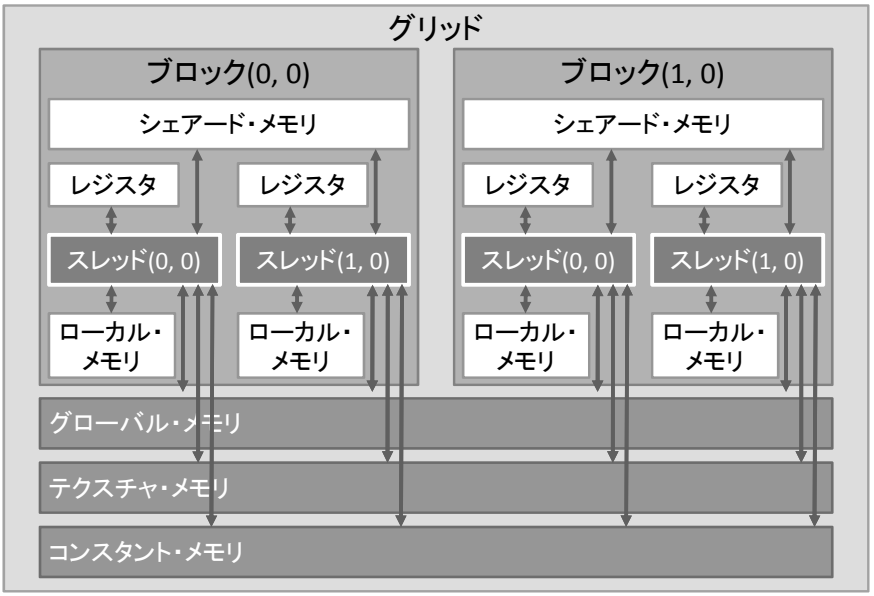

図 7: CUDA のメモリモデル

表 1: デバイス・メモリ

種類	場所	キャッシュ	アクセス	スコープ
レジスタ・メモリ	チップ上	-	R/W	スレッド
ローカル・メモリ	チップ外	されない	R/W	スレッド
シェアード・メモリ	チップ上	-	R/W	ブロック
グローバル・メモリ	チップ外	されない	R/W	グリッドとホスト
テクスチャ・メモリ	チップ外	される	R	グリッドとホスト
コンスタント・メモリ	チップ外	される	R	グリッドとホスト

プログラミングモデル

CUDA のプログラムは，GPU 上で実行されるデバイス・コードと，CPU 上で実行されるホスト・コードで構成される．デバイス・コードにはカーネル関数を記述する．カーネル関数とは，GPU で実行される処理を記述した関数のことであり，グリッド内の全てのスレッドは同じカーネル関数を実行する．一方，ホスト・コードには通常の C 言語プログラムに加えて，GPU のカーネル関数を呼び出すカーネル関数コールなどを記述する．さらに，カーネル関数コールと共に，カーネル関数を実行するスレッド数を指定できる．また，各スレッドは固有の ID を持ち，その ID を表す CUDA のビルトイン変数を用いることで，当該スレッドがどのメモリアドレスにアクセスするかを指示できる．

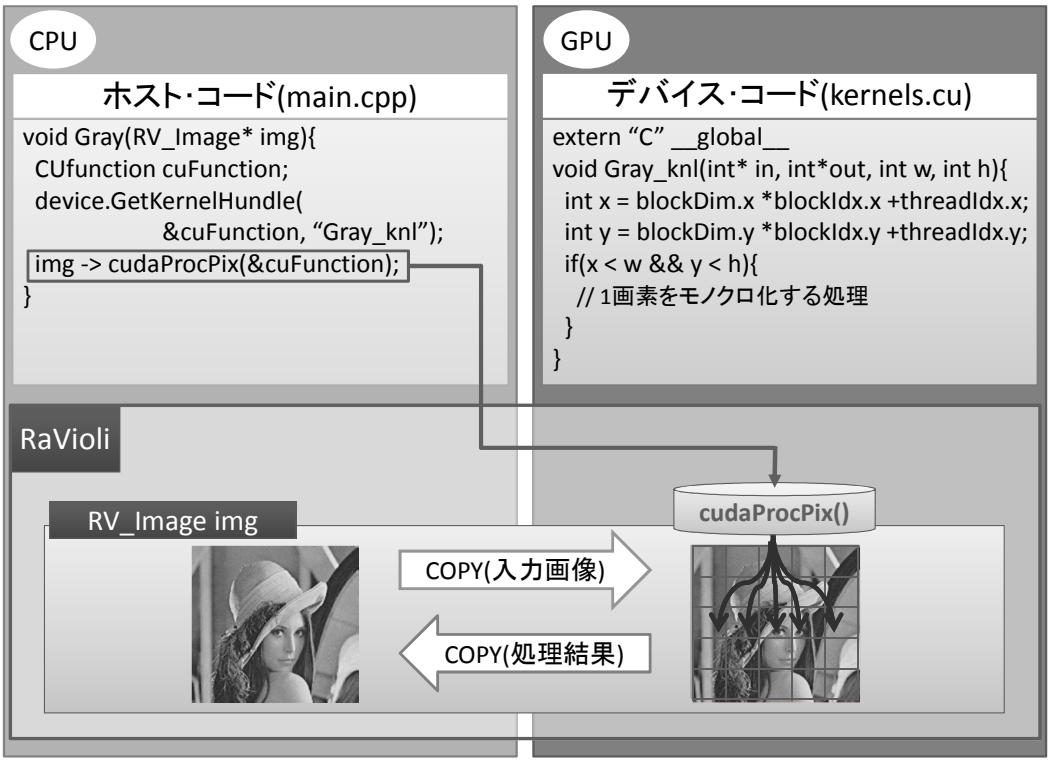

図 8: RaVioli/CUDA のインターフェイス

2.3.2 RaVioli/CUDA

RaVioli は画像情報をプログラマから隠蔽し，解像度をライブラリ内部で制御することにより，抽象的な動画像処理プログラミングを実現している．一方，CUDA を使用して画像処理プログラムを記述する場合，プログラマはデバイス側のメモリや，スレッドの実行構成などを意識してプログラムを記述する必要がある．しかし，これらの処理は画像処理の本質ではないため，プログラマの負担になると考えられる．そこで，RaVioli/CUDA は，メモリ確保や実行構成の設定などもプログラマから隠蔽する．

RaVioli/CUDA のインターフェイスを**図 8** に示す．まずデバイス・コードでは，構成要素関数をカーネル関数として定義する．カーネル関数のボディには，構成要素に対する処理の他に，CUDA のビルトイン変数を用いて，当該スレッドが担当する画素を決める処理を記述する．このカーネル関数は，モジュールとして扱うために別ファイル (*.cu) 内に記述する．これにより，実行時にホスト側から動的にロードすることが可能になる．*.cu ファイルに記述されたデバイス・コードは，CUDA が提供する nvcc コンパイラを使用してアセンブリ形式 (ptx コード) またはバイナリ形式 (cubin コード) へと変換される．一方，ホスト・コードでは，まず画像を RV_Image インスタンスと

して受け取る．次に，モジュールをロードし，さらにカーネル関数のハンドルを取得して，RV_Image インスタンスの高階メソッド cudaProcPix() の引数に渡す．これにより，自動的にデバイス・メモリに領域が確保され，ホスト側からデバイス側へデータが転送される．さらに実行構成も定義され，カーネル関数が処理対象画素に適用される．そして，全画素の処理が終わると処理結果がホスト側に返される．このようにして，RaVioli/CUDA は，メモリ確保や実行構成の設定などを全て自動的に行う．

2.3.3 問題点

RaVioli/CUDA は CUDA を用いて並列に画像処理することにより，RaVioli におけるカプセル化のオーバヘッドによる処理速度の低下を抑制している．また，CUDA を使用するために必要となるメモリ確保や実行構成の設定などをライブラリ内部で管理することにより，プログラマの負担を減らしている．

しかし，RaVioli と RaVioli/CUDA に共通して，処理内容に応じて適切な高階メソッドを選択しなくてはならないという問題がある．RaVioli と RaVioli/CUDA は，以下の処理パターンにあわせてそれぞれいくつかの高階メソッドを提供している．

- 画素の 1 対 1 写像
  1 つの画素を参照してその画素を処理する画像処理．グレースケール化や 2 値化，肌検出などに使われる．
- 画素の多対 1 写像
  処理対象画素とその近傍画素を参照して中心の画素を処理する画像処理．ぼかしやエッジ抽出などに使われる．
- 画素の多対多写像
  対象とする画素の集合 (部分画像) を処理し結果を得る画像処理．テンプレートマッチングなどに使われる．

RaVioli や RaVioli/CUDA を使用して画像処理プログラムを記述する際，上記のような処理パターンと，その処理に適した高階メソッドを意識してプログラムを記述しなければならない．

また，RaVioli/CUDA では，カーネル関数に対するハンドルの取得処理などをホスト・コードに，各スレッドが担当する画素を決める処理をデバイス・コードにそれぞれ分けて記述する必要がある．これらの処理はプログラマが記述しなくてはならず，プログラマには CUDA の基本的な知識が必要となる．

そこで本論文では，これらの問題を解決するために，より直感的に画像処理を記述できる言語を提案する．さらに，提案する言語で記述されたプログラムを，RaVioli で

記述したプログラムと RaVioli/CUDA で記述したプログラムへ変換するトランスレータも提案する．これにより，プログラマは RaVioli および RaVioli/CUDA が持つ種々の高階メソッドを使い分ける必要がなくなる．また CUDA の知識を一切持たなくとも，GPU を利用した画像処理が実現可能となる．

3　画像処理記述言語の提案

本章では，本論文で提案する言語の仕様および文法について述べる．

3.1　統一的な記述方式

一般的な画像処理プログラムでは 2.2.1 項で述べたように，画素や部分画像といった画像の構成要素に対する処理と，その処理を全構成要素に適用するためのループの記述が必要である．そこで，画像の幅，高さや解像度などを考慮することなく画像処理を記述できる，より直感的な画像処理記述言語を提案する．この言語では，構成要素に対する処理と，その処理が繰り返し適用される範囲とを指定するだけで画像処理プログラムを記述できる．処理と範囲を記述するというこの方針は RaVioli と似ているが，RaVioli では 2.3.3 項で述べたように，ユーザは目的の画像処理がどの高階メソッドに当てはまるかを考え，使い分ける必要があった．

そこで提案言語では，以下に示した処理単位と処理範囲を指定することでループを省略し，目的の画像処理のパターンによらない統一的な記述方式を採用する．

統一的な記述方式

```
処理単位 @ 処理範囲 {
    構成要素に対する処理
}
```

ここで処理範囲とは，処理を施したい画像全体のことである．また処理単位とは，1 対 1 写像では 1 つの画素のことであり，多対 1 写像では部分画像のことである．ユーザはこの処理単位に対する処理を記述することによって，画像全体に処理を施すことができる．また，様々な処理パターンに対して統一的な記述が可能になる．

3.2　言語仕様

本節では，2.3.3 項で示した 3 つの画像処理パターンを提案言語で記述し，そのプログラムを例に提案言語の仕様について説明する．

```
1 Func Gray(img_a){
2   p1@img_a{
3     ave = (p1.R + p1.G + p1.B) / 3;
4     p1.{R, G, B} = {ave, ave, ave};
5   }
6 }(img_a)
```

図 9: 提案言語で記述したグレースケール化プログラム

3.2.1 1 対 1 写像

本言語による 1 対 1 写像の記述を，**図 9** に示すグレースケール化プログラムを例に説明する．プログラムの基本構造は，関数名の定義，引数の宣言，施したい処理の記述，返り値である．関数を定義する際には，関数名の前に Func と記述する仕様とする．図 9 のプログラムでは，関数名は Gray，引数は img_a，返り値は処理を施された img_a となっている．また，変数の型を宣言する必要はなく，指定しない限り基本的に局所変数として扱われる．

本言語では，PIXEL 変数と IMAGE 変数という特殊な変数を用意している．これらの変数はそれぞれ，2.2.1 項で述べた RaVioli の RV_Pixel インスタンスと RV_Image インスタンスに相当し，以下のように記述される．

PIXEL 変数

pn (n は整数)

IMAGE 変数

img_a(a は英字 1 文字から始まり，
それ以降は英数字およびアンダースコアから成る文字列)

図 9 では，処理単位は “p1”，処理範囲は “img_a” となっている．p1 は PIXEL 変数であり，img_a は，画像の幅や高さなどの情報や，画像内の全ての画素を扱うための IMAGE 変数である．2 行目の “p1 @ img_a” は処理範囲を img_a とし，処理単位の画素 p1 は img_a 内の任意の要素を示している．そして，関数定義内に記述された処理単位の画素 p1 に対する処理は，img_a 内の全画素要素に適用される．

図 9 内の処理単位に対する処理では，まず 3 行目で，p1 の R 値，G 値，B 値の平均値が計算され，その値が変数 ave に代入される．p1 の RGB の値を個々に扱う場合は，“p1.R” のように “.” の後ろに色情報の記号を記述する．そして 4 行目で，ave を p1 の色とし

```
1  Func EdgeDetect(img_a){
2    threshold = 127;
3    p1@img_a{
4      temp = <-1, -1>p1.R + <0, -1>p1.R    + <1,- 1>p1.R
5           + <-1, 0>p1.R  - 8 * <0, 0>p1.R + <1, 0>p1.R
6           + <-1, 1>p1.R  + <0, 1>p1.R     + <1, 1>p1.R;
7      if(temp > threshold) p1 = #black;
8      else p1 = #white;
9    }
10 }(img_a)
```

図 10: 提案言語で記述したエッジ抽出プログラム

<-1, -1>	<0, -1>	<1, -1>
<-1, 0>	対象画素	<1, 0>
<-1, 1>	<0, 1>	<1, 1>

図 11: 相対座標

て設定することで p1 をグレースケール化することができる．ここで，“p1.{R, G, B}” のように “{” と “}” を使うことにより，複数変数をタプル化してまとめて扱うことができる．

このようにして，プログラマは処理単位と処理範囲を容易に指定することが可能になる．さらに通常の画像処理におけるループのイタレーション回数や，画像内の画素数などを考慮する必要がなくなる．

3.2.2 多対 1 写像

本言語による多対 1 写像の記述を，**図 10** に示すエッジ抽出プログラムを例に説明する．図 10 中の 4 行目から 6 行目にある “<” と “>” で囲まれた表記は，相対座標を指定できる COORD 指定子であり，PIXEL 変数の前に COORD 指定子を記述することで，PIXEL 変数からの相対的な位置を指定できる．COORD 指定子を用いた相対座標の表記を**図 11** に示す．中心が処理対象となる画素であり，“<” と “>” で囲まれたタプルの第一要素は x 方向の相対座標を表し，第二要素は y 方向の相対座標を表している．図 10 の 4 行目から 6 行目において，この相対座標を用いることで p1 とその周囲

8 画素の R 値を用いた演算が記述されている．

ここでエッジ抽出とは，グレースケール化された画像を用い，注目画素とその近傍画像の画素値の差が大きい場合に，その画素をエッジと判定する処理である．図 10 では，まず，先ほど述べたように 4 行目から 6 行目で，注目画素 p1 とその 8 近傍の画素の画素値の差を計算し，その値を変数 temp に代入する．そして 7 行目で，閾値を保持する変数 threshold と temp を比べることで，差が大きいかどうかを判断する．差が大きい場合，p1 をエッジであるとみなし黒色を，そうでない場合は白色を設定する．

7 行目と 8 行目で使用されている “#” で始まる文字列は色を記述する表現である．計算機で色を表現する際は，RGB カラーモデルという表記法を用いるのが一般的であるが，これはユーザにとって直感的ではない．例えば (R, G, B) の値が (D2, 69, 1E) であるとき，これが茶色を表すとは直感的に理解出来ない．また，逆に茶色を構成している RGB の値を直感的に理解することも困難である．そこで，本言語では “#black” と “#white” のように色を表す COLOR 変数を提供する．COLOR 変数は以下のように記述する．

COLOR 変数

```
#white or #FFF
```

“#” の後ろに，色の名称を英語，または 3 桁の 16 進数で記述する．例えば，#white, #ABC，#123 という COLOR 変数があったとする．これらはそれぞれ 16 進表記で (ff, ff, ff)，(aa, bb, cc)，(11, 22, 33) の RGB 値を持つ色として扱われる．なお色名は CSS Color Module[12] の定義に準じている．

3.2.3 多対多写像

本言語による多対多写像の記述を，**図 12** に示すテンプレートマッチングプログラムを例に説明する．図 12 の 1 行目から 3 行目のように，GLOBAL ブロック内で宣言された変数は例外的に大域変数として扱われる．7 行目の “[img_tp]img_window@img_a” は，処理単位が img_window，処理範囲が img_a であることを示している．部分画像を処理単位とする場合は，処理単位の前で IMAGE 変数を “[” と “]” で囲むことにより，その IMAGE 変数の大きさを処理単位の大きさとして指定することができる．つまりこの例では，処理単位である img_window の大きさには，img_tp の大きさが指定される．8 行目の “(p1, p2)@(img_window, img_tp)” では，img_window と img_tp の 2 つの画像を同時に扱っており，img_window 内のある画素 p1 と，img_tp 内の同位置の画素 p2 を同時に処理することを示している．8 行目から 10 行目では，部分画像 img_window

```
1  GLOBAL{
2    img_tp;
3  }
4
5  Func TPmatching(img_a){
6    min = INT_MAX;
7    [img_tp]img_window@img_a{
8      (p1, p2)@(img_window, img_tp){
9        sum += abs(p1 - p2);
10     }
11     if(sum < min){
12       min = sum;
13       <x, y> = img_window.{x, y};
14     }
15     sum = 0;
16   }
17   img_a > writeBox(#red, <x, y>, [img_tp]) > img_b;
18 }(img_b)
```

図 12: テンプレートマッチング

とテンプレート画像 img_tp の差分を取ることで類似度を求め，変数 sum に加算している．そして 11 行目から 14 行目で，類似度を保持する変数 sum が最小値を保持する変数 min よりも小さいかどうかが判定される．もし sum が min よりも小さい場合は，sum を min に代入することで最小値を更新し，img_window 内の現在処理している画素 p1 の x 座標，y 座標を変数 x，y にそれぞれ代入する．

17 行目は，画像 img_a に関数 writeBox が適用されていることを表す．img_a は関数 writeBox() の入力であり，出力は img_b となる．この “writeBox” という関数は提案言語が提供する組込み関数であり，この関数を適用することにより，画像の任意の位置に長方形の枠線を描画できる．また，“writeLine” という関数も用意されており，この関数を画像に適用すると，その画像に直線を描画できる．これらの関数は以下のように記述する．

組込み関数

writeBox(#colorname, $< x, y >$, [img_a])

writeLine(#colorname, rho, theta)

```
1 Func main(file_in, file_tp, file_out){
2   img_in <= file_in;
3   img_tp <= file_tp;
4   img_in > TPmatching > img_out;
5   img_out => file_out;
6 }
```

図 13: テンプレートマッチングの main 関数

関数 writeBox() を適用することによって描画される枠線の大きさは，IMAGE 変数を “[” と “]” で囲むことによってその IMAGE 変数が表す画像と同じ大きさになるよう指定できる．また，枠線の左上の位置を座標 $< x, y >$ で指定することにより，枠線が描画される位置を決めることができる．一方，関数 writeLine() は，画像の左上を原点とし，原点からの距離を “rho”，原点からの角度を “theta” とする直線を描画できる．両方の関数に共通する colorname は，枠線や直線の色を表す．色は 3.2.2 項で説明した COLOR 変数を用いて指定できる．

3.2.4 main 関数の記述

図 12 のテンプレートマッチングプログラムを入力画像に適用する main 関数を**図 13** に示す．プログラム中の file_in，file_tp や file_out のように，“file_” で始まる変数は入出力ファイル用の変数であり，入力ファイルである画像データは “<=” 演算子を用いて IMAGE 変数に読み出される．図 13 では 2 行目と 3 行目で，file_in と file_tp がそれぞれ img_in と img_tp に読み出されている．そして 4 行目の “>” 演算子は，“img_A> 関数 >img_B” と記述することにより，画像 A を関数の入力として与え，関数の出力を画像 B という変数として受け取ることを表すものである．4 行目では，img_in に関数 TPmatching() が適用され，img_out という変数で返り値を受け取っている．最後に 5 行目のように，“=>” 演算子を用いることで出力 img_out が出力ファイルに書き込まれる．

4 トランスレータの実装

3 章で提案した言語で記述されたプログラムを変換するためにトランスレータを実装する．本章では，そのトランスレータによる変換について述べる．

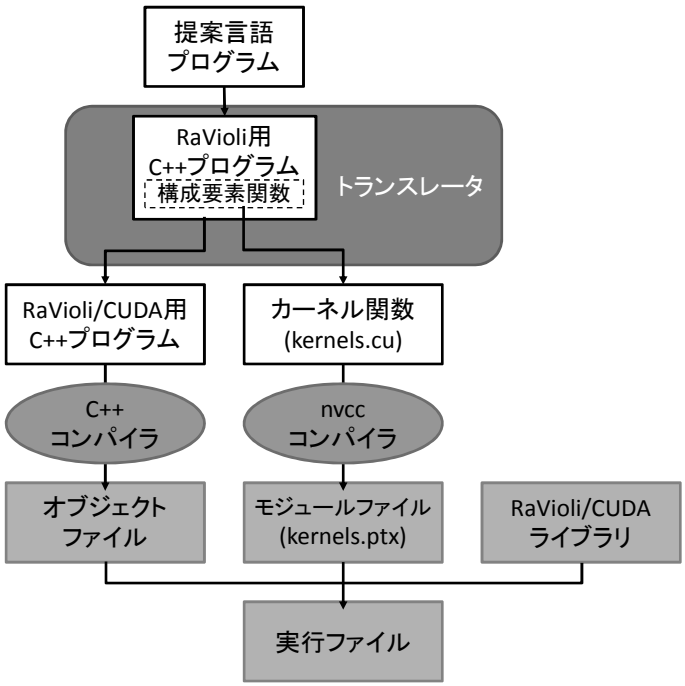

図 14: トランスレータを含むコンパイルフロー

4.1 トランスレータの概要

実装するトランスレータによる変換行程を含むコンパイルフロー全体を**図 14** に示す．トランスレータはまず，提案言語プログラムを RaVioli 用の C++プログラムへ変換する．さらに，生成された RaVioli プログラムは，RaVioli/CUDA 用の C++プログラムとカーネル関数へ変換される．

4.2 RaVioli プログラムへの変換

提案言語で記述されたグレースケール化プログラム (図 9) から変換された RaVioli プログラムを，**図 15** に示す．IMAGE 変数や PIXEL 変数などの特別な変数を除くすべての変数は，トランスレータによって変数の先頭に “_R_関数名” がつけられ，大域変数として宣言される．“処理単位@処理範囲 {}” で囲まれた処理 (図 9，2 行目から 5 行目) は画像全体に繰り返し適用される処理であり，RaVioli の構成要素関数に相当する．そのため，この範囲の処理は図 15 内の 2 行目から 5 行目のように 1 つの関数として定義される．そして，7 行目から 10 行目のように，この構成要素関数を引数として受け取る高階メソッドの呼び出しを含む関数も定義される．変数の初期化のような一度だけ実行すればよい処理は，高階メソッドを呼び出す前に実行するために，この関数内に記述される．

```
1  int _R_Gray_ave;
2  void Gray_loop (RV_Pixel* p1){
3    _R_Gray_ave = (p1->getR() + p1->getG() + p1->getB()) / 3;
4    p1 -> setRGB(_R_Gray_ave, _R_Gray_ave, _R_Gray_ave);
5  }
6
7  void Gray (RV_Image* img_a){
8    _R_Gray_ave = 0;
9    img_a -> procPix(Gray_loop);
10 }
11
12 int main(int argc, char* argv[]){
13   RV_Image* img_in;
14   file.readBMP(argv[1],img_in);
15   Gray(img_in);
16   file.writeBMP(argv[2],img_in);
17 }
```

図 15: 生成された RaVioli のグレースケール化プログラム

変換後のプログラムで使用される高階メソッドは，トランスレータによって自動的に選択される．これは，処理単位と処理範囲にそれぞれ何が指定されているかによって判断可能であるためである．例えば，図 9 では “p1@img_a” と記述されており，処理単位は PIXEL 変数，処理範囲は IMAGE 変数となっている．この場合，高階メソッドとして procPix() が選択される．高階メソッド procPix() が引数として構成要素関数 Gray_loop() を受け取ることにより，関数 Gray_loop() 内の処理は画像 img_a 全体に適用される．

なお，3 行目の getR()，getG()，getB() は，それぞれ，引数として画素値を受け取り R 値，G 値，B 値を返す関数である．そして，4 行目の setRGB() は，RGB 値を受け取り画素にその値を設定する関数である．

4.3 RaVioli/CUDA プログラムへの変換

実装するトランスレータは，RaVioli で記述されたプログラムを CPU で実行されるホスト・コードと，GPU で実行されるデバイス・コードの 2 つへ変換することも可能とする．RaVioli/CUDA では 2.3.1 項で挙げた CUDA が持つ各種メモリの使い分けや，

```
/* main.cpp */
void Gray(RV_Image* img_a){
  CUfunction cuFunction;
  device.GetKernelHundle(&cuFunction, "Gray_knl" );
  img_a -> cudaProcPix(&cuFunction);
}

int main(int argc, char* argv[]){
  RV_Image img_in;
  device.RaCudaInit(); /* initialize device */
  Gray(img_in);
  device.RaCudaExit(); /* finalize device */
}
```

図 16: 生成された RaVioli/CUDA のホスト・コード

```
/* kernels.cu */
extern "C" __global__
void Gray_knl(int* idata, int* odata, int width, int height){
  int x = blockDim.x * blockIdx.x + threadIdx.x;
  int y = blockDim.y * blockIdx.y + threadIdx.y;
  int p1;
  if(x < width && y < height){
    p1 = idata[y * width + x];
    int ave = (p1.getR() + p1.getG() + p1.getB()) / 3;
    setRGB(&odata[y * width + x], ave, ave, ave);
  }
}
```

図 17: 生成された RaVioli/CUDA のデバイス・コード

リダクション処理が必要となる場合があるため，これらの処理を自動的に追加する機能も実装する．

4.3.1 基本変換

図 15 の RaVioli プログラムは，トランスレータにより**図 16** に示すホスト・コードと**図 17** に示すデバイス・コードへとさらに変換される． 図 16 のホスト・コードでは，まずドライバの初期化とコンテキストの生成などが必要となる．また全ての処理の後には

コンテキストの削除なども必要となる．そこで処理開始の直前 (10 行目) と直後 (12 行目) に，それぞれ RaCudaInit() と RaCudaExit() の呼び出しを挿入する．RaCudaInit() と RaCudaExit() は RaVioli/CUDA が提供する関数であり，それぞれデバイスの初期化と終了処理を行う．また図 15 の構成要素関数 Gray_loop() は図 17 のカーネル関数 Gray_knl() へと変換される．RaVioli/CUDA は，このカーネル関数のハンドルを引数として受け取り，カーネル関数内の処理を自動的に対象画像全体に施す高階メソッドを持つ．そこでトランスレータは，構成要素関数を引数として受け取る高階メソッドの呼び出し (図 15，9 行目) を，カーネルハンドルの取得 (図 16，4 行目) と，それを引数に受け取る高階メソッドの呼び出し (図 16，5 行目) へと変換する．

一方，図 17 のデバイス・コードには，2 行目と 3 行目に示すカーネル関数の型指定子と関数宣言子が記述される．型指定子の前の “__global__” は関数修飾子であり，これは，関数がどこから呼ばれどこで実行されるかによって決まる．図 17 のように，ホスト側から呼ばれデバイス側で実行される場合は，“__global__” をつける必要がある．一方，カーネル関数がデバイス側から呼ばれ同じデバイス側で実行される場合は，“__device__” をつける．また，関数宣言子中の引数は高階メソッドの種類によって決まる．図 16 では高階メソッド cudaProcPix() が使われているため，この場合の引数は処理対象画像配列 idata，処理結果を格納する配列 odata，画像の幅 width，画像の高さ height となる．

さらに，4 行目から 8 行目は高階メソッド名に応じて生成される．4 行目と 5 行目では，各スレッドの ID などを用いて，そのスレッドが画像中のどの画素を処理するかを決定している．また，カーネル関数が画像の範囲外にアクセスしないように 7 行目に条件式が挿入される．

以上のようにして RaVioli で記述されたプログラムを RaVioli/CUDA に対応したプログラムへ変換することが可能である．さらに，このような基本的な変換に加え，CUDA が持つ各種メモリの使い分けやリダクション処理を考慮した変換も可能である．次項以降ではこれらの機能について述べる．

4.3.2 CUDA メモリの使い分け

2.3.1 項で述べたように，CUDA ではいくつかのメモリを使用することができる．ここで，近傍処理に用いられる procNeighbor() や，部分画像を処理するために用いられる procBox() などの高階メソッドを使う処理の場合，1 枚の画像内の注目画素とその近傍の複数画素の値が用いられるため，メモリ上の近いアドレスが参照される点に注目する．各種メモリの中でも，テクスチャ・メモリには，SM ごとにテクスチャ・キャッシュが搭載されており，テクスチャ・メモリにアクセスした際のデータは一時的にキャッ

```
1  void EdgeDetect_loop(RV_Pixel* p, RV_Pixel* p1, RV_Pixel* p2, RV_Pixel* p3,
2                                    RV_Pixel* p4, RV_Pixel* p5, RV_Pixel* p6,
3                                    RV_Pixel* p7, RV_Pixel* p8, RV_Pixel* p9){
4    int temp = p1.getR() + p2.getR()     + p3.getR()
5             + p4.getR() - 8 * p5.getR() + p6.getR()
6             + p7.getR() + p8.getR()     + p9.getR();
7    // 処理対象画素 p に適切な値を設定
8  }
9
10 void EdgeDetect(RV_Image* img_a){
11   img_a -> procNeighbor(EdgeDetect_loop);
12 }
13 // main 文
```

図 18: RaVioli のエッジ抽出プログラム (部分)

シュされる．そのためデータアクセスに局所性がある場合，高速に読み出しが可能である．したがって，高階メソッドの procNeighbor() や procBox() が使用される処理の場合，テクスチャ・メモリを使うことで処理の高速化が期待できる．そこで，これらの高階メソッドが使用される場合，テクスチャ・メモリを自動的に使用できるようにするための処理を，トランスレータによって追加するようにした．

トランスレータの変換によりテクスチャ・メモリが使用可能な例として，RaVioli のエッジ抽出プログラムを**図 18** に示す．エッジ抽出では，処理対象画素と，その近傍画素の値を用いるため，高階メソッドは procNeighbor() を使用する．また，構成要素関数 EdgeDetect_loop() の引数は，それらの画素を受け取るために複数となっている．トランスレータがこのような記述を検出した場合，テクスチャ・メモリを使用するために必要な処理を自動的に追加する．

変換後の RaVioli/CUDA プログラムを**図 19** に示す．なお，このテクスチャ・メモリを使用した変換後プログラム全体は付録 A.1.1 に掲載する．テクスチャ・メモリに画像データを転送する処理は，高階メソッド cudaProcNeighbor() 内で自動的に実行されるため，変換後のプログラムにそれらの処理を追加する必要はない．そのため，12 行目のようにテクスチャ変数 tex の宣言を変換時にプログラムに追加するだけで，tex を通じて画像データを参照できるようになる．画像データを参照するには “tex2D()” という CUDA の組込み関数を使用し，画像データ tex における処理対象画素のアドレスを指定する．

```
/* main.cpp */
void EdgeDetect(RV_Image* img_a){
  CUfunction cuFunction;
  CUtexref cuTexref;
  device.GetKernelHundle(&cuFunction, "EdgeDetect_knl");
  device.GetTexrefHundle(&cuTexref, "tex");
  img_a -> cudaProcNeighbor(&cuFunction, &cuTexref);
}
// main 文

/* kernels.cu */
texture<int, 2, cudaReadModuleElementType> tex;
extern "C" __global__
void EdgeDetect_knl(int* odata, int width, int height){
  int x = blockDim.x * blockIdx.x + threadIdx.x;
  int y = blockDim.y * blockIdx.y + threadIdx.y;
  if(x < width && y < height){
    int temp = getR(tex2D(tex, x-1, y-1)) + getR(tex2D(tex, x, y-1)) +
               ..... + getR(tex2D(tex, x, y+1)) + getR(tex2D(tex, x+1, y+1));
    setRGB(&odata[y * width + x], temp, temp, temp);
  }
}
```

図 19: テクスチャ・メモリを使用したプログラム (部分)

4.3.3 リダクション処理の追加

RaVioli/CUDA プログラムへの変換により，カーネル関数は 1 スレッドが 1 画素に対して処理を施すように定義される．そして，各スレッドがそれぞれ 1 画素に対する処理を並列に実行することで処理の高速化が実現できる．しかし，画素間に依存関係が存在する場合，各スレッドが任意の 1 画素を処理する際に，他の画素の処理結果が必要になるため，各スレッドの処理結果の正当性が保証されない．そこで，画素間に依存関係が存在する場合は，リダクション処理を自動的に生成・追加することにより正しい結果を保証する．リダクション処理とは，並列数分用意した一時的な格納領域に対して各スレッドの局所的な処理結果を格納し，それらの処理結果を最後に統合することで全体の最終的な結果を求める処理である．

複数の処理単位にまたがる依存関係は，以下の場合に検出される．

```
1  int sum;
2  void TP_loop1(RV_Pixel* p1, RV_Pixel p2){
3    sum += abs(p1->getR() - p2.getR()) + ......;
4  }
5
6  void TP_loop0(RV_Image* img_window){
7    img_window -> procImgComp(TP_loop1, img_tp);
8    if(sum < min)
9      min = sum;
10   sum = 0;
11 }
12
13 void TP(RV_Image* img_a){
14   img_a -> procBox(TP_loop0, img_tp->Width, img_tp->Height)
15 }
16 // main 文
```

図 20: RaVioli のテンプレートマッチングプログラム (一部省略)

- 複数の構成要素関数内で，同じ大域変数に対して書き込んでいる
- if 文の条件式で他の変数と比較されている大域変数に対して書き込んでいる
- 構成要素関数内で書き込みされた大域変数が，別の構成要素関数内の if 文の条件式で他の変数と比較されている

これらのうちいずれか 1 つでもあてはまった場合は依存関係があると判断し，トランスレータが自動的にリダクション処理を追加する．依存関係が検出される例を**図 20** に示す．図 20 は RaVioli のテンプレートマッチングプログラムの一部である．構成要素関数 TP_loop0() は部分画像 img_window を処理対象とし，関数 TP() 内の高階メソッド procBox() によって画像に適用される．さらに，関数 TP_loop0 内で，構成要素関数 TP_loop1 を引数として受け取る高階メソッド procImgComp() を呼び出す．関数 TP_loop1 は画素を処理単位とするため，部分画像 img_window 内の各画素を処理することができる．このようにして，テンプレートマッチングでは構成要素関数が 2 つ必要となる．なお，procImgComp() とは 2 枚の画像を比較するための高階メソッドであり，各画像における同じ位置にある画素の値を参照することができる．

1 行目で宣言されている大域変数 sum に注目すると，まず TP_loop1() 内で sum に対して書き込みが行われ，さらに，TP_loop0() 内における if 文の条件式で参照されて

```
1  /* kernels.cu */
2  __device__ int TP_loop1_knl(int* idata){
3    // sum の値を計算
4  }
5
6  extern "C" __global__
7  void TP_loop0_knl(int* idata, int3* local_data, int width, int height){
8    int sum = 0;
9    if(x < width && y < height)
10     sum = TP_loop1_knl(idata);
11   else
12     sum = INT_MAX;
13   // 各スレッドの処理結果用配列 local_data に sum,x,y を格納
14 }
15
16 extern "C" __global__
17 void reduction_knl(int3* data){
18   // 各スレッドの処理結果をグローバル・メモリからシェアード・メモリへコピー
19   // シェアード・メモリ上でリダクションを計算
20   // 最終的な結果を配列へ格納
21 }
```

図 21: リダクション処理を追加したプログラム (一部省略)

いる．よって，この変数 sum は，リダクション処理を追加すべき変数であると判定される．リダクション処理が追加された変換後のデバイス・コードは，**図 21** のように各構成要素関数を変換したカーネル関数と，リダクション処理を含む関数から構成される．図 20 の変換前プログラムの 8 行目から 10 行目は，図 21 の 9 行目から 13 行目と，16 行目から 21 行目のリダクション処理を含む関数 reduction_knl() に変換される．図 20 の 8 行目から 10 行目は，変数 sum と変数 min を比較し最小値を求める処理である．一方，図 21 では，まず 9 行目から 12 行目で，カーネル関数が画像の範囲外へアクセスしていない場合は，関数 TP_loop1_knl を呼び出し sum の値を求める．そうでない場合は sum の値が最小値とならないように INT_MAX を代入しておく．そして 13 行目で各スレッドの処理結果を，一時的な配列に格納する．さらに，関数 reduction_knl() 内で，この各スレッドの処理結果が格納されている配列をまとめ，その中の最小値を求める処理をする．この関数は，まず，グローバル・メモリ上の各スレッドの処理結

表 2: プログラムサイズの比較

プログラム		C++	RaVioli	提案言語
グレースケール化	行数	38	22	12
	バイト数	910	486	244
エンボスフィルタ	行数	153	26	15
	バイト数	4074	867	444
テンプレート	行数	114	50	21
マッチング	バイト数	3946	1175	430
直線検出	行数	139	79	50
	バイト数	3518	2163	1084

果をシェアード・メモリにコピーする．そして，全スレッドの同期とリダクション処理によって，ブロック内の最小値とそのときの座標を求める．最後に，最終的な結果を結果用の配列に格納する．このような変換により，画素間に依存関係が存在する場合でも，処理の並列実行が可能になる．なお，このリダクション処理が追加された変換後プログラム全体は付録 A.1.2 に掲載する．

5 評価

3 章と 4 章で述べた提案言語とトランスレータを，グレースケール化，エンボスフィルタ，テンプレートマッチング，直線検出のプログラムを用いて評価した．トランスレータの実装には，flex と bison を用いた．

5.1 プログラムサイズの比較

提案言語，RaVioli，C++で記述した各プログラムのサイズを，行数とバイト数で比較した結果を**表 2** に記す．なお，評価対象の各プログラムは付録 A.2 に掲載する．

4 つの評価対象プログラムにおいて，提案言語プログラムはC++プログラム，RaVioli プログラムに比べ，行数，バイト数共に削減することができた．行数では，エンボスフィルタにおいて最大で 90%，バイト数では，エンボスフィルタとテンプレートマッチングにおいて最大で 89%の削減率となった．

5.2 可読性，記述性の比較

提案言語で記述したプログラムの可読性を，RaVioli，C++で記述したプログラムの可読性と比較した．提案言語，RaVioli，C++で記述した二値化プログラムをそれ

```
Func Binary(img_a){
  p1@img_a{
    if(p1.V < 85) p1 = #black;
    else p1 = #white;
  }
}(img_a)
img_in > Binary > img_out;
```

図 22: 提案言語で記述した二値化プログラム

```
void Binary_Pix(RV_Pixel* p){
  if((p -> getV()) < 85) p -> setRGB(0, 0, 0);
  else p -> setRGB(255, 255, 255);
}
Image -> procPix(Binary_Pix);
```

図 23: RaVioli で記述した二値化プログラム

```
void Binary(Img* img){
  int i, j, v, tmp;
  for(i = 0; i < img -> height; i++){
    for(j = 0; j < img -> width; j++){
      v = getV(img -> D[i][j]);
      if(v < 85) tmp = 255;
      else tmp = 0;
      img -> D[i][j].r = tmp;
      img -> D[i][j].g = tmp;
      img -> D[i][j].b = tmp;
    }
  }
}
Binary(&Image);
```

図 24: C++で記述した二値化プログラム

ぞれ**図 22**，**図 23**，**図 24** に記す.

提案言語と RaVioli では，どちらも解像度や画像の構成要素数などの画像情報をプログラマから隠蔽しており，プログラマはそれらを意識することなく直感的にプログラム

表 3: 評価環境

OS	Fedora15
CPU	Core2Quad
Frequency	2.83GHz
Memory	3GB
GPU	GeForce GTX280
Number of multiprocessors	30
Number of cores (SP)	240 (30 × 8)
Compute capability	1.3
Compiler	gcc 4.6.1
Compile options	-O3

を記述することが可能となっている．次に，提案言語と RaVioli を比較すると，RaVioli ではユーザ定義関数を全ての処理対象に適用するために，2.3.3 項で述べた処理パターンに応じた高階メソッドの呼び出しが必要になる．図 23 では，二値化が 1 対 1 処理であることを踏まえて，プログラマが 5 行目のように procPix() を選択しなければならない．これに対し提案言語では，3.1 節で述べたように，統一的な記述方式を用いることにより様々な処理パターンを同じ形式で記述できる．図 22 では 2 行目のように記述するだけよく，プログラマが高階メソッドの使い分けを考慮する必要がなくなっている．

5.3 実行時間

次に，トランスレータによって生成されたプログラムの実行時間を比較した．**表 3** に評価環境を示す．グレースケール化，エンボスフィルタでは 512×512 の画像，テンプレートマッチングでは 395×372 の処理対象画像，70×72 のテンプレート画像，直線検出では 450×540 の画像を使用した．結果を**表 4** に示す．“C++” は C++で記述したプログラム，“RaVioli” は RaVioli を用いて記述した RaVioli プログラム，“自動変換 RaVioli” はトランスレータによって生成された RaVioli プログラム，“CUDA のみ” は CUDA で記述されたプログラム，“RaVioli/CUDA” はトランスレータによって生成された RaVioli/CUDA プログラムを表す．

“RaVioli” と “自動変換 RaVioli” の実行時間にほとんど差はなく，トランスレータが無駄な命令を出力することなく，適切に変換できていることがわかる．一方，“RaVioli/CUDA” はグレースケール化，エンボスフィルタと直線検出において，“CUDA のみ” より処理速度が低下している．これは，RaVioli におけるカプセル化のオーバヘッ

表 4: 実行時間の比較 (ms)

プログラム	C++	RaVioli	自動変換 RaVioli	CUDA のみ	RaVioli/ CUDA
グレースケール化	0.90	4.92	3.76	1.22	1.47
エンボスフィルタ	4.89	10.31	9.85	1.34	1.48
テンプレートマッチング	2429.61	8457.76	8173.30	54.78	47.75
直線検出	34.91	66.33	67.35	24.27	26.69

ドによるものだと考えられる．また，テンプレートマッチングでは，RaVioli のオーバヘッドがあるにも関わらず，“RaVioli/CUDA” は “CUDA のみ” よりも速くなった．これは，“CUDA のみ” と “RaVioli/CUDA” が異なる API を用いて実装されているからだと考えられる．CUDA にはランタイム API とドライバ API という 2 つの API が用意されている．ランタイム API は暗黙的な初期化やモジュール管理などを提供することでデバイス・コードの管理をより容易にしており，ユーザにとっても記述が容易である．一方，ドライバ API はより多くの複雑なコードが必要になるが，処理を詳細に指定することができ，モジュールを直接扱うことができるなどランタイム API では記述できない処理も記述可能である．“CUDA のみ” はプログラマが実際に記述したプログラムであり，記述が容易なランタイム API で実装されている．一方，“RaVioli/CUDA” はトランスレータの変換によってドライバ API を用いて実装されている．そのため，“RaVioli/CUDA” は “CUDA のみ” よりもは速くなったと考えられる．このように，“CUDA のみ” と比べた “RaVioli/CUDA” の処理速度は，僅かな低下もしくは向上であり，大きな差はなかった．これにより，RaVioli/CUDA の使用は実用的であることを示せた．

さらに，“RaVioli/CUDA” は全てのプログラムにおいて “RaVioli” に比べて処理速度の向上を達成できた．グレースケール化，エンボスフィルタ，テンプレートマッチング，直線検出のそれぞれにおいて，約 3.3 倍，約 6.9 倍，約 177.1 倍，約 2.5 倍の処理速度の向上が確認できた．特にテンプレートマッチングにおいては，C++に比べても約 50.9 倍の速度向上が達成できた．

6 おわりに

本論文では，より直感的に画像処理プログラムを記述できる新しい言語を提案した．一般的な画像処理プログラムでは，構成要素に対する処理を全構成要素に適用するた

めのループの記述が必要である．これに対し提案言語では，構成要素に対する処理と，その処理が繰り返し適用される範囲とを指定するだけで画像処理プログラムを記述できる．また，様々な処理パターンに対して統一的な記述が可能になる．

さらに，この提案言語で記述したプログラムを，RaVioli で記述したプログラムと，RaVioli/CUDA で記述したプログラムへ変換するトランスレータも実装した．トランスレータによる自動変換により，プログラマに CUDA に関する知識がなくとも，GPU を利用した画像処理が可能となった．

いくつかの画像処理プログラムを用いて，提案言語，RaVioli，C++で記述したプログラムのサイズを比較した．その結果，提案言語により記述されたプログラムは，RaVioli や C++を用いたプログラムに比べて行数，バイト数共に削減できることを確認した．また，プログラマが記述したプログラムと，トランスレータによって生成されたプログラムの実行時間を比較した結果，2 つのプログラムの実行時間にほとんど差がなかったため，トランスレータはプログラムを適切に変換できたことを確認した．さらに，生成された RaVioli/CUDA プログラムの実行時間は，RaVioli プログラムの実行時間に比べて，テンプレートマッチングにおいて最大で 177.1 倍の速度向上が達成できることを確認した．

今後の課題として，より多くの画像処理プログラムを記述できるようにすることが挙げられる．RaVioli では多くの処理を記述できるが，提案言語ではまだその全てを記述することができない．また，現状では動画像処理プログラムには対応していないため，今後はその記述も対象として言語を拡張していく．

さらに，現在 RaVioli は CUDA 以外にも Cell/B.E. や TBB などを用いた並列化に対応するように改良が進められている．そこで，これらに対応したプログラムへの変換も可能にし，更なる処理速度の向上を目指すことも今後の課題である．

謝辞

本研究のために，多大な御尽力を頂き，御指導を賜わった名古屋工業大学の松尾啓志教授，津邑公暁准教授，齋藤彰一准教授，松井俊浩准教授に深く感謝致します．また，本研究の際に多くの助言，協力をして頂いた松尾・津邑研究室，齋藤研究室および松井研究室の方々に深く感謝致します．特に，研究に関して貴重な意見を下さった近藤勝彦氏，稲葉崇文氏，大平真司氏に感謝します．

著者発表論文

論文

1. Ami ONO, Katsuhiko KONDO, Takafumi INABA, Tomoaki TSUMURA, Hiroshi MATSUO: "A GPU-supported High-Level Programming Language for Image Processing", Proc. of The 7th Int'l Conf. on Signal-Image Technology and Internet-Based Systems (SITIS2011), pp.245-252(Nov. 2011)
2. Katsuhiko KONDO, Ami ONO, Takafumi INABA, Tomoaki TSUMURA, Hiroshi MATSUO: "Tiling with Different Spatial Resolutions for Pseudo Real-Time Video Processing Library RaVioli", Proc. of The 7th Int'l Conf. on Signal-Image Technology and Internet-Based Systems (SITIS2011), pp.253-260(Nov. 2011)

参考文献

[1] Liu, J., Shih, W.-K., Lin, K.-J., Bettati, R. and Chung, J.-Y.: Imprecise Computations, *Proceedings of the IEEE*, Vol. 82, pp. 83–94 (1994).

[2] Yoshimoto, H., Date, N., Arita, D. and Taniguchi, R.: Confidence-Driven Architecture for Real-time Vision Processing and Its Application to Efficient Vision-based Human Motion Sensing, *Proc. of the 17th Int'l. Conf. on Pattern Recognition (ICPR'04)*, Vol. 1, pp. 736–740 (2004).

[3] 岡田慎太郎, 桜井寛子, 津邑公暁, 松尾啓志: 解像度非依存型動画像処理ライブラリ RaVioli の提案と実装, 情報処理学会論文誌コンピュータビジョンとイメージメディア（CVIM）, Vol. 2, No. 1, pp. 63–74 (2009).

[4] Sakurai, H., Ohno, M., Tsumura, T. and Matsuo, H.: RaVioli: a Parallel Video Processing Library with Auto Resolution Adjustability, *Proc. IADIS Int'l. Conf. Applied Computing 2009*, Vol. 1, pp. 321–329 (2009).

[5] Köthe, U.: Generic Programming for Computer Vision: The VIGRA Computer Vision Library, http://hci.iwr.uni-heidelberg.de/vigra/ (2011).

[6] Bradski, G. and Kaehler, A.: *Learning OpenCV: Computer Vision With the OpenCV Library*, O'Reilly & Associates Inc (2008).

[7] Intel Corp.: *Open Source Computer Vision Library* (2001).

[8] 金井達徳, 瀬川淳一, 武田奈穂美: 組み込みプロセッサのメモリアーキテクチャに依存しない画像処理プログラムの記述と実行方式, 情報処理学会論文誌：コンピュー

ティングシステム, Vol. 48, No. SIG 13(ACS 19), pp. 287–301 (2007).

[9] Segawa, J. and Kanai, T.: The Array Processing Language and the Parallel Execution Method for Multicore Platforms, *The First International Symposium on Information and Computer Elements* (2007).

[10] NVIDIA Corp.: *NVIDIA CUDA Programming Guide*, 2.0 edition (2008).

[11] 青木尊之, 額田彰: はじめての CUDA プログラミング, 工学社 (2009).

[12] Çelik, T., Lilley, C. and Baron, L. D.: CSS Color Module Level 3, Technical report, W3C (2011).

付録

A.1 トランスレータによって生成された RaVioli/CUDA プログラム

A.1.1 テクスチャ・メモリを使用したプログラム (エンボスフィルタ)

```
/* main.cpp */
#include "ravioli.h"
RV_FileHandler file;
RV_Cuda device;

RV_Image* img_in;
RV_Image* img_out;

void Gray(RV_Image* img_a){
  CUfunction cuFunction;
  device.GetKernelHundle(&cuFunction,"Gray_loop0_kernel");
  img_a->cudaProc(&cuFunction);
}

void Emboss(RV_Image* img_a){
  CUfunction cuFunction;
  CUtexref cuTexref;
  device.GetKernelHundle(&cuFunction,"Emboss_loop0_kernel");
  device.GetTexrefHundle(&cuTexref, "tex");
  img_a->cudaProcNeighbor(&cuFunction,&cuTexref);
}

int main(int argc, char* argv[]){
  file.readBMP(argv[1],img_in);
  device.RaCudaInit();

  Gray(img_in);
  Emboss(img_in);

  device.RaCudaExit();
  file.writeBMP("output.bmp",img_in);
  return 0;
}

/* kernels.cu */
#include "kernels_lib.cu"
texture<int, 2, cudaReadModeElementType> tex;

extern "C"
__global__ void
```

```
Gray_loop0_kernel(int* idata, int* odata, int width, int height){
  int R_Gray_temp = 0;

  int x = blockDim.x * blockIdx.x + threadIdx.x;
  int y = blockDim.y * blockIdx.y + threadIdx.y;
  int rgb;

  if(x < width && y < height){
    rgb = idata[y * width + x];
    R_Gray_temp = (getR(rgb) + getB(rgb) + getG(rgb)) / 3;
    setRGB(&odata[y * width + x], R_Gray_temp, R_Gray_temp, R_Gray_temp);
  }
}

extern "C"
__global__ void
Emboss_loop0_kernel(int* odata, int width, int height){
  int R_Emboss_temp = 0;

  int x = blockDim.x * blockIdx.x + threadIdx.x;
  int y = blockDim.y * blockIdx.y + threadIdx.y;

  if(x < width && y < height){
    R_Emboss_temp =
    -1*getR(tex2D(tex,x-1,y-1))-1*getR(tex2D(tex,x,y-1))+0*getR(tex2D(tex,x+1,y-1))
    -1*getR(tex2D(tex,x-1,y))+0*getR(tex2D(tex,x,y))+1*getR(tex2D(tex,x+1,y))
    +0*getR(tex2D(tex,x-1,y+1))+1*getR(tex2D(tex,x,y+1))+1*getR(tex2D(tex,x+1,y+1));
    if(R_Emboss_temp < 0){
      R_Emboss_temp = (-1)*R_Emboss_temp;
    }
    R_Emboss_temp = R_Emboss_temp + 128;
    if(R_Emboss_temp > 255){
      R_Emboss_temp = 255;
    }
    else if(R_Emboss_temp < 0){
      R_Emboss_temp = 0;
    }
    setRGB(&odata[y * width + x], R_Emboss_temp, R_Emboss_temp, R_Emboss_temp);
  }
}
```

A.1.2 リダクション処理を追加したプログラム (テンプレートマッチング)

```
/* main.cpp */
#include "ravioli.h"
RV_FileHandler file;
```

```
RV_Cuda device;

RV_Image* img_in;
RV_Image* img_tp;
RV_Image* img_out_tmp;
RV_Image* R_tp_img_tp;

CUtexref cuTexTPref;
CUarray d_TPimage;

void tp(RV_Image* img_a){
  CUfunction cuFunction;
  CUfunction cuFunction2;
  int3 result;
  device.GetKernelHundle(&cuFunction,"tp_loop0_kernel");
  device.GetKernelHundle(&cuFunction2, "reduction_kernel");
  cuParamSetTexRef(cuFunction, CU_PARAM_TR_DEFAULT, cuTexTPref);
  result = img_a->cudaProcBox_global(&cuFunction,
                  img_tp->Width,img_tp->Height,&cuFunction2);
}

int main(int argc, char* argv[]){
  file.readBMP(argv[1],img_in);
  file.readBMP(argv[2],img_tp);
  device.RaCudaInit();

  device.GetTexrefHundle(&cuTexTPref,"texTP");
  img_tp->TexRefSetImage(&d_TPimage,&cuTexTPref);
  tp(img_in);
  cutilDrvSafeCall(cuArrayDestroy(d_TPimage));

  device.RaCudaExit();
  return 0;
}

/* kernels.cu */
#include "kernels_lib.cu"
texture<int, 2, cudaReadModeElementType> texTP;

__device__ int
tp_loop1_kernel(int* idata, int w, int h, int wTP, int hTP, int x, int y){
  int R_tp_sum=0;
  int i,j;

  for(j=0; j<hTP; j++){
```

```
    for(i=0;i<wTP;i++){
      int rgb=idata[(y+j)*w + (x+i)];
      int rgbTP=tex2D(texTP, i, j);
      R_tp_sum += abs((getR(rgb))-(getR(rgbTP)))
                + abs((getG(rgb))-(getG(rgbTP)))
                + abs((getB(rgb))-(getB(rgbTP)));
    }
  }
  return R_tp_sum;
}

extern "C"
__global__ void
tp_loop0_kernel(int* idata, int3* data4reduction,
                int3* data4reduction2, int w, int h, int wTP, int hTP){
  int R_tp_min=INT_MAX;
  int x=blockDim.x*blockIdx.x+threadIdx.x;
  int y=blockDim.y*blockIdx.y+threadIdx.y;
  int wwtp= gridDim.x*blockDim.x;

  if(x<w-wTP && y<h-hTP){
    int R_tp_sum=tp_loop1_kernel(idata, w, h, wTP, hTP, x, y);
    if(R_tp_sum < R_tp_min){
      R_tp_min=R_tp_sum;
      data4reduction[y*wwtp+x].x=x;
      data4reduction[y*wwtp+x].y=y;
      data4reduction2[y*wwtp+x].z = data4reduction[y*wwtp+x].z=R_tp_sum;
    }
  }
}

extern "C"
__global__ void
reduction_kernel(int3* data4reduction, int3* g_odata){
  __shared__ int sdatax[256];
  __shared__ int sdatay[256];
  __shared__ int sdataz[256];

  unsigned int tid=threadIdx.x;
  unsigned int i=blockIdx.x*blockDim.x+threadIdx.x;
  sdatax[tid]=data4reduction[i].x;
  sdatay[tid]=data4reduction[i].y;
  sdataz[tid]=data4reduction[i].z;
  __syncthreads();
```

```
  for(unsigned int s=blockDim.x/2;s>0;s>>=1){
    if(tid<s){
      if(sdataz[tid]>sdataz[tid+s]){
sdatax[tid]=sdatax[tid+s];
sdatay[tid]=sdatay[tid+s];
sdataz[tid]=sdataz[tid+s];
      }
    }
    __syncthreads();
  }
  if(tid==0){
    g_odata[blockIdx.x].x=sdatax[0];
    g_odata[blockIdx.x].y=sdatay[0];
    g_odata[blockIdx.x].z=sdataz[0];
  }
}
```

A.2 評価に用いたプログラム (直線検出)

A.2.1 提案言語

```
GLOBAL{
  weight[360.0][2*DIAGONAL];
}

Func binary(img_a){
  threshold = 85;
  p1@img_a{
    if(p1.V < threshold) p1 = #black;
    else p1 = #white;
  }
}(img_a)

Func edge(img_a){
  threshold = 127;
  p1@img_a{
    temp = <-1,-1>p1 + <0,-1>p1 + <1,-1>p1
      + <-1,0>p1 - 8*<0,0>p1 + <1,0>p1
      + <-1,1>p1 + <0,1>p1 + <1,1>p1;
    if(temp > threshold) p1 = #black;
    else p1 = #white;
  }
}(img_a)

Func hough(img_a){
  threshold = 50;
```

```
  p1@img_a{
    if(p1 == #black){
      theta@(0..359.0){
rho = p1.x * cos(theta/180*PI) + p1.y * sin(theta/180*PI);
weight[theta][rho]++;
      }
    }
  }
}(img_a, weight)

Func re_hough(img_a, weight){
  threshold = 100;
  theta@(0..359){
    rho@(-DIAGONAL..DIAGONAL){
      if(weight[theta][rho] > threshold){
img_a > writeLine(#orange, rho, theta) > img_b;

      }
    }
  }
}(img_b)

Func main(file_in,file_out){
  img_in <= file_in;

  img_in > binary | edge | hough | re_hough > img_out;

  img_out => file_out;
}
```

A.2.2 RaVioli

```
#include "ravioli.h"
#define MAX_THETA 360.0
#define PI 3.141592653589
#define threshold1 85
#define threshold2 100

RV_Image* input_image;
RV_Image* output_image;
RV_Array<int>* RH_THdata;

RV_Length XMAX,YMAX;
RV_Color color(255,0,255);
RV_Length DIAG;
RV_Length zero;
```

```
void grayscale(RV_Pixel* p){
  if((p->getV())<threshold1){
    p->setRGB(0, 0, 0);
  }
  else{
    p->setRGB(255, 255, 255);
  }
}

void edgeDetect(RV_Pixel* target,RV_Pixel NW,RV_Pixel N,RV_Pixel NE,
                RV_Pixel W,RV_Pixel C,RV_Pixel E,
                RV_Pixel SW,RV_Pixel S,RV_Pixel SE){

  int tmp;
  tmp=NW.getR()+N.getR()+NE.getR()
     +W.getR()-8*C.getR()+E.getR()
     +SW.getR()+S.getR()+SE.getR();
  if(tmp>127){
    target->setRGB(0,0,0);
  }
  else{
    target->setRGB(255,255,255);}
}

void Hough(RV_Pixel* p,RV_Coord coord){
  if(p->getR() == 0) {
    RV_Length rho;
    double theta;
    RV_Length tmpx=coord.getXLength();
    RV_Length tmpy=coord.getYLength();
    for (theta = 0; theta < MAX_THETA; theta++) {
      rho=tmpx*cos(theta*2.0*PI/MAX_THETA)
         +tmpy*sin(theta*2.0*PI/MAX_THETA);
      rho=rho+DIAG;
      if(rho>zero){
if(rho<DIAG){
  RH_THdata->setData((int)theta,rho,
                             RH_THdata->getData((int)theta,rho)+1);
}
      }
    }
  }
}
```

```
void reHough(int* data,RV_Coord coord){
  if (*data >= threshold2){
    RV_Length theta=coord.getXLength();
    RV_Length rho=coord.getYLength();
    rho=rho-DIAG;
    output_image->writeline(color,rho,theta);
  }
}

int main(int argc,char* argv[]){
  int grain;
  RV_FileHandler file;
  zero.Zero();
  file.readBMP(argv[1],input_image);
  output_image=new RV_Image(*input_image);

  input_image->proc(grayscale);
  input_image->procNeighbor(edgeDetect);

  XMAX=input_image->Width;
  YMAX=input_image->Height;
  DIAG=XMAX*XMAX+YMAX*YMAX;
  DIAG=DIAG.Sqrt();

  RH_THdata = new RV_Array<int>((int)MAX_THETA,2*DIAG);

  input_image->procCoord(Hough);//procCooord
  RH_THdata->procCoord(reHough);//procCoord RV_Array

  if(argc>=3){
    file.writeBMP(argv[2],output_image);
  }
  else{
    file.writeBMP("output.bmp",output_image);}

  return 0;
}
```

A.2.3 C++

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <sys/time.h>
typedef unsigned char byte;
```

```
#define MAX_THETA 360
#define PI 3.141592653589
#define threshold 100

int main(int argc,char* argv[]){
  int i,j;
  imageFile input_image;
  imageFile output_image;
  readBMP(argv[1],input_image);
  readBMP(argv[1],output_image);

  int width=input_image.width;
  int height=input_image.height;

  for(j=0;j<height;j++){
      for(i=0;i<width;i++){
int max,min,R,G,B,d;
R=input_image.R[j*width+i];
G=input_image.G[j*width+i];
B=input_image.B[j*width+i];
max=(int)((R> G) ? (R> B?R:B) : (G> B?G:B));
min=(int)((R< G) ? (R< B?R:B) : (G< B?G:B));

if(max<128)
  input_image.R[j*width+i]=
    input_image.G[j*width+i]=
    input_image.B[j*width+i]=0;
else
  input_image.R[j*width+i]=
    input_image.G[j*width+i]=
    input_image.B[j*width+i]=255;
      }
  }


  byte* newR=new byte[input_image.width*input_image.height];
  byte* newG=new byte[input_image.width*input_image.height];
  byte* newB=new byte[input_image.width*input_image.height];
  int tmp;

  for(j=0;j<height;j++){
    newR[j*width]=newG[j*width]=newB[j*width]
      =newR[j*width-1]=newG[j*width-1]=newB[j*width-1]=255;
  }
  for(i=0;i<width;i++){
```

```
    newR[i]=newG[i]=newB[i]
      =newR[(height-1)*width+i]
      =newG[(height-1)*width+i]
      =newB[(height-1)*width+i]=255;
  }

  for(j=1;j<height-1;j++){
      for(i=1;i<width-1;i++){
tmp=
  (int)input_image.R[(j-1)*width+(i-1)]
  +input_image.R[(j-1)*width+i]
  +input_image.R[(j-1)*width+(i+1)]
  +input_image.R[j*width+(i-1)]
  +input_image.R[j*width+(i+1)]
  +input_image.R[(j+1)*width+(i-1)]
  +input_image.R[(j+1)*width+i]
  +input_image.R[(j+1)*width+(i+1)]
     -input_image.R[j*width+i]*8;

if(tmp>128)
  newR[j*width+i]=
    newG[j*width+i]=
    newB[j*width+i]=0;
else
  newR[j*width+i]=
    newG[j*width+i]=
    newB[j*width+i]=255;
      }
  }

  input_image.R=newR;
  input_image.G=newG;
  input_image.B=newB;

  int DIAG,theta;
  DIAG=(int)sqrt(width*width+height*height);
  int* RH_THdata = new int[MAX_THETA*DIAG];
  int rho;
  for(j=0;j<height;j++){
      for(i=0;i<width;i++){
if(input_image.R[j*width+i]==0){
  for (theta = 0; theta < MAX_THETA; theta++) {
    rho=(int)(i*cos((double)theta*2.0*PI/MAX_THETA)
      +j*sin((double)theta*2.0*PI/MAX_THETA));
    rho=rho+DIAG/2;
```

```
    if(rho>0)
      RH_THdata[rho*MAX_THETA+theta]++;
  }
}
      }
  }

  int nx,ny;
  double _theta,a,b;
  for(j=0;j<DIAG;j++){
    for(i=0;i<MAX_THETA;i++){
      if(RH_THdata[j*MAX_THETA+i]>=threshold){
rho=j-DIAG/2;
_theta=(double)i/180*3.1415;
a=cos(_theta);
b=sin(_theta);
a=-a/b;
b=rho/b;
for (nx = 0; nx < width; nx++) {
  ny = (int)rint((a * nx) + b);
  if (0 <= ny && ny < height){
    output_image.R[width*ny+nx]=100;
    output_image.G[width*ny+nx]=200;
    output_image.B[width*ny+nx]=100;
  }
}
for (ny = 0; ny < height; ny++) {
  nx = (int)rint((ny - b) / a);
  if (0 <= nx && nx < width){
    output_image.R[width*ny+nx]=100;
    output_image.G[width*ny+nx]=200;
    output_image.B[width*ny+nx]=100;
  }
}
      }
    }
  }
  writeBMP("output.bmp",output_image);
}
```