修士論文

CPU/GPUの協調動作を支援する動画像処理プログラミング環境の提案と実装

A Video Processing Programming Environment for CPU-GPU Hybrid Computing

指導教員　津邑 公暁 准教授
松尾 啓志 教授

名古屋工業大学大学院 工学研究科
修士課程 創成シミュレーション工学専攻
平成 25 年度入学　25413567 番

松永 拓也

平成 27 年 2 月 4 日

CPU/GPU の協調動作を支援する 動画像処理プログラミング環境の提案と実装

松永 拓也

内容梗概

情報技術の発達に伴い，静止画や動画像を扱える多くの情報機器が開発され，普及してきている．一方でプラットフォームに目を向けると，汎用プロセッサコアとアクセラレータコアを搭載した，ヘテロジニアスな計算機環境が一般的となりつつある．これは，低消費電力で高い性能を実現できるためである．このような背景から，ヘテロジニアスな計算機環境上で，動画像処理プログラムを開発する機会が増加している．しかしながら，このような計算機環境を対象とした高性能な動画像処理プログラムを開発するためには，その複雑なプロセッサ構成を理解した上でプログラムを記述し，さらにこれをチューニングする必要がある．そのため，プログラム開発コストの大きさが問題となっている．

この問題を解決するため，動画像処理ライブラリ RaVioli が提案されている．RaVioli は画像や動画像における画素やフレームをプログラマから隠蔽することで動画像処理プログラミングを抽象化し，プログラマの負担を軽減している．RaVioli を用いる場合，プログラマは画像の構成要素に対する処理を関数として定義し，その関数を RaVioli が提供する高階メソッドに渡す．このようにすることで，プログラマが記述した処理が画像全体に繰り返し適用される．以上のように，RaVioli を用いたプログラムでは，画像の構成要素に対する処理が関数として定義されるため，処理単位が明確となる．そのため，ライブラリ内での自動並列化の実現が比較的容易である．そこで，RaVioli では様々な環境に対応した自動並列化機能が実現されている．その中で，現在アクセラレータコアとして最も普及している GPU に対応させた RaVioli の拡張が RaVioli/CUDA である．なお，CUDA とは NVIDIA 社が提供する GPU コンピューティング向け統合開発環境である．RaVioli/CUDA では，この CUDA 特有の記述をライブラリ内に隠蔽することでプログラマの負担を軽減している．

しかし，RaVioli/CUDA には大きく 2 つの問題点が存在する．1 つ目は，RaVioli/CUDA がまだ画像処理を十分に抽象化できていない点である．RaVioli/CUDA を用いる場合，プログラマは 1 画素や近傍画素集合など処理単位毎に異なるメソッドを適切に選択する必要がある．2 つ目は，ヘテロジニアスな環境を十分に活かしきれていない点である．RaVioli/CUDA では，CPU は GPU へデータを転送して構成要素関

数を呼び出した後，GPU がその構成要素関数の処理結果を返すまでの間，遊休状態となってしまう．これでは CPU/GPU という 2 つのユニットを効率的に活用できない．

そこで本論文ではまず，RaVioli/CUDA よりも抽象度の高い言語を提案する．さらに，CPU/GPU の協調動作を支援する RaVioli/CUDA の拡張も併せて提案する．提案する言語では，処理パターンに依存しない統一的な記述方式を採用する．これにより，処理単位毎に適切なメソッドを選択しなければならない既存の RaVioli/CUDA よりも，さらに抽象度の高いプログラミングを可能にする．また RaVioli/CUDA の拡張では，画像処理および動画像処理における 2 つの並列性に着目し，2 種類の CPU/GPU 協調動作手法を提供する．この 2 種類の協調動作手法では，画像処理および動画像処理に対して CPU/GPU が協調的に動作し，どちらかのユニットが遊休状態になることを抑制する．これにより，ヘテロジニアスな計算機環境をより有効に活用できるようにする．さらに，提案言語で記述したプログラムから，拡張後の RaVioli/CUDA を用いたプログラムへと自動的に変換するトランスレータを提供する．これにより，プログラマは簡単な記述のみで高性能な動画像処理プログラムを開発可能となる．

提案するプログラミング環境を実装し，その有効性を評価した．まず，記述性を評価する 1 つの指標として C++，既存の RaVioli，提案言語を用いてそれぞれいくつかの画像処理および動画像処理プログラムを記述し，そのプログラムサイズを比較した．その結果，提案言語で記述されたプログラムは，C++や既存の RaVioli を用いたプログラムと比べて，行数，バイト数共に削減できることを確認した．さらに，2 種類の協調動作手法の有効性を確かめるために，提案手法を用いて実装したいくつかの画像および動画像処理プログラムと既存の RaVioli，RaVioli/CUDA を用いて実装したプログラムの実行速度を比較した．その結果，直線検出を施す動画像処理プログラムにおいて，提案手法で記述したプログラムは，既存の RaVioli で記述したプログラムに対して 25.80 倍，既存の RaVioli/CUDA で記述したプログラムに対して 1.45 倍の速度向上を達成できることを確認した．

CPU/GPU の協調動作を支援する
動画像処理プログラミング環境の提案と実装

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，認証装置など静止画や動画像を扱える様々な情報機器が普及してきている．一方でプラットフォームに目を向けると，汎用プロセッサコアと GPU に代表されるようなアクセラレータコアを搭載した，ヘテロジニアスな計算機環境が一般的となりつつある．これは，低消費電力で高い性能を実現できるためである．このような背景から，ヘテロジニアスな計算機環境上で，動画像処理プログラムを開発する機会が増加している．しかしながら，このような計算機環境を対象とした高性能な動画像処理プログラムを開発するためには，その複雑なプロセッサ構成を理解した上で，目的とするアプリケーションの並列性を抽出し，プログラムを記述，チューニングする必要がある．そのため，プログラム開発コストの大きさが問題となっている．

この問題を解決するために，解像度非依存型動画像処理ライブラリ **RaVioli**（**Resolution-Adaptable Video and Image Operating Library**）[1, 2] が提案されている．RaVioli は **空間解像度**（1 フレーム内の画素数）および**時間解像度**（フレームレート）をプログラマから隠蔽することで動画像処理プログラミングを抽象化し，プログラマの負担を軽減している．RaVioli を用いる場合，プログラマは画素や近傍画素集合などといった画像の構成要素に対する処理を関数として定義し，その関数を RaVioli が提供する高階メソッドに渡す．これにより，プログラマが記述した処理が画像全体に繰り返し適用される．この高階メソッドは画素や近傍画素集合，テンプレート画像など様々な処理単位毎に用意されており，プログラマは目的とするアプリケーションに応じて適切な高階メソッドを選択することで様々な画像処理を実現することができる．

以上のように RaVioli を用いる場合では，画像の構成要素に対する処理を関数として定義するため，処理単位が明確となる．そのためライブラリ内での自動並列化の実現が比較的容易である．そこで，RaVioli では様々な環境に対応した自動並列化機能が実現されてきた．その中で，現在アクセラレータコアとして最も普及している GPU に対応させた RaVioli の拡張が **RaVioli/CUDA** である．RaVioli/CUDA は，NVIDIA が提供している GPU コンピューティング向け統合開発環境の **CUDA**[3, 4] をバックエンドとして実装されている．CUDA では GPU 上に大量に存在するコアのそれぞれに対してスレッドを割り当て，この大量のスレッドを用いて同一の命令を並列に実行することで処理の高速化を実現する．しかし CUDA を用いたプログラミングでは，GPU が持つ様々なメモリに対してデータを転送する処理など，CUDA 特有の記述が必要と

なる．RaVioli/CUDA ではこれらの CUDA プログラミングに特有の記述をライブラリ内に隠蔽することでプログラマの負担を軽減している．

しかし，この RaVioli/CUDA には大きく 2 つの問題点が存在する．1 つ目は，RaVioli/CUDA がまだ動画像処理を十分に抽象化できていない点である．RaVioli/CUDA を用いる場合，プログラマは 1 画素や近傍画素集合など処理単位毎に異なるメソッドを適切に選択して使用する必要がある．また，データ転送などは隠蔽しているものの，CUDA を利用するためのいくつかの記述をプログラムに追加しなければならず，やはり CUDA の基本的な知識がある程度必要となる．2 つ目の問題は，ヘテロジニアスな環境を十分に活かしきれていない点である．RaVioli/CUDA では，CPU は GPU へデータを転送し構成要素関数を呼び出した後，GPU がその構成要素関数の処理を完了するまで遊休状態となってしまう．これでは CPU/GPU という 2 つのユニットを効率的に活用できない．

そこで本研究ではまず，RaVioli/CUDA よりも抽象度の高い言語を提案する．さらに，CPU/GPU の協調動作を支援する RaVioli/CUDA の拡張も併せて提案する．提案する言語では，処理パターンに依存しない統一的な記述方式を採用する．これにより，処理単位毎に適切なメソッドを選択しなければならない既存の RaVioli/CUDA よりも，さらに抽象度の高いプログラミングを可能にする．一方，RaVioli/CUDA の拡張では，画像処理に存在するデータ並列性および動画像処理に存在するタスク並列性に着目し，2 種類の CPU/GPU 協調動作手法を提供する．この 2 種類の協調動作手法では，画像処理および動画像処理に対して CPU/GPU が協調的に動作し，どちらかのユニットが遊休状態になることを抑制する．これにより，ヘテロジニアスな計算機環境をより有効に活用できるようにする．さらに，提案言語で記述したプログラムから，拡張後の RaVioli/CUDA を用いたプログラムへと自動的に変換するトランスレータを提供する．これにより，プログラマは簡単な記述のみで高性能な動画像処理プログラムを開発可能となる．

以下，2 章では関連研究について述べ，3 章では動画像処理ライブラリ RaVioli および RaVioli/CUDA の概要とその問題点について述べる．4 章では CPU/GPU の協調動作を支援する動画像プログラミング環境を提案する．5 章では提案言語プログラムから拡張後の RaVioli/CUDA プログラムへと自動変換するトランスレータについて述べ，6 章では 2 種類の CPU/GPU 協調動作手法の実現方法について述べる．そして 7 章で提案言語，拡張後の RaVioli/CUDA およびトランスレータの評価とそれらに対する考察を述べる．最後に 8 章で本論文全体をまとめる．

2　関連研究

静止画や動画像を扱う情報機器の普及と，低消費電力で高い性能を実現可能なヘテロジニアスな計算機環境が広く普及してきていることから，動画像処理プログラムを開発する機会が増加している．このような背景を受けて，これまでに多くの画像処理や動画像処理のためのライブラリが提案されている．例えば，VIGRA[5]，OpenIP[6]，Pandore[7]，およびOpenCV[8, 9] はよく知られた画像処理ライブラリである．VIGRA はC++のSTL（Standard Template Library）を基本とするテンプレートを用いて一般的な画像処理パターンを抽象化したライブラリである．画像の反転や回転，エッジ処理などの基本的な処理から，ガウスやガボールに代表されるフィルタ処理，画像の分析処理などを抽象化して提供している．OpenIP は教育，研究，および産業分野の画像処理やコンピュータビジョンに対する要求を満たす，C++のライブラリ一式を提供する．このライブラリ一式は画像処理のためのデータ構造や，フィルタリングなどの画像処理アルゴリズム，画像の入出力のインタフェースなども提供している．Pandore は実行可能な画像処理演算子のセットを提供している．プログラマはそれらの演算子を組み合わせることで画像処理アプリケーションを作成することが可能である．OpenCV は，現在最も普及している画像処理ライブラリであり，様々な画像処理アルゴリズムをC言語やC++，Java などのメソッドとしてプログラマに提供している．また，プログラマ自身が独自のアルゴリズムを実装するための基本的な機能も備えている．これらのライブラリは処理内容を抽象化し，関数やメソッドとして提供することで画像処理を容易に記述可能にしている．しかし，これらのうちVIGRA，OpenIP，Pandore では，処理内容が関数やメソッドがライブラリ内部に隠蔽されてしまっているため，これらを用いて汎用マルチコアプロセッサを対象とした並列プログラムを記述することは困難である．一方で，OpenCV ではTBB[10] やOpenMP[11] といった並列化ライブラリと併用することで比較的容易に画像処理プログラムを並列化させることが可能である．

並列処理の分野では，一般的に用いられる並列処理パターンをライブラリの形で抽象化して提供する並列スケルトンの考え方が古くから存在する．それぞれのスケルトンは並列化のための動作を内部に隠蔽しているため，プログラマは逐次プログラムを作成する場合とほとんど同様に並列処理プログラムを実現することができる．例えば，Intel で開発された並列計算パターン用ライブラリ TBB は，並列スケルトンの形で書かれたプログラムをマルチコア環境上で並列に動作させるライブラリであり，スレッド管理を隠蔽することで処理の並列化を抽象的に扱える機能を提供する．また並列プ

ログラミングの標準化されたモデルである OpenMP は，C 言語や C++で記述された逐次プログラム内の並列化したい部分を簡単な指示文で指定し，それをプリプロセッサが半自動的に並列処理プログラムへと変換するディレクティブ形式を採用している．しかしながら，これらのライブラリを用いても，現在市場に広く流通しているヘテロジニアスな環境を対象とした効率的な並列プログラムの実現は困難である．

さらに，ライブラリだけでなく画像処理向けのプログラミング言語も提案されている．例えば，PPL（Picture Processing Language）[12]，金井らによる画像処理言語[13, 14]，および Halide[15, 16] が挙げられる．PPL は画像処理アルゴリズムのカプセル化を基本方針として設計された言語である．PPL を用いることで画像処理アルゴリズムを実装する際に個々の画素を扱う必要がなくなり，デジタル画像処理について明るくないプログラマでも画像処理プログラムを比較的容易に記述できるようになる．なお，PPL は一般的によく用いられる画像処理アルゴリズムを提供しているが，PPL を拡張することで新しい画像処理アルゴリズムを使用することもできる．そのため，プログラマは PPL が提供する API を使用し，よりカスタマイズされた関数やメソッドを PPL に追加することも可能である．しかし，これはエンドユーザのプログラマにとって容易ではない．さらに，この言語にはプログラムの並列化に対応していないという問題点も存在する．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．この言語では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．しかしこれは，プログラマが画像処理の本質とは関係のない，配列の大きさを考慮した記述をしなければならないことを意味する．そして，この言語では汎用マルチコアプロセッサを対象とした並列化には対応しているものの，ヘテロジニアスな環境には対応していない．一方，Halide は汎用マルチコアプロセッサや GPU，ARM プロセッサなど様々なプラットフォーム上での並列化を可能にする画像処理言語である．Halide を用いたプログラムでは画像処理アルゴリズム定義部分と並列化手順定義部分を分けて記述する．これによりプログラマは画像処理アルゴリズムに変更を加えることなく，最適な並列化スケジューリングを実現できる．しかし，効率の良い並列化を実現するためにはプログラマが並列スケジューリングに精通している必要があり，必ずしもプログラマにとって利用しやすい環境ではない．

以上で述べたように，これまでに提案されている画像処理および動画像処理のためのライブラリや言語は，処理内容を抽象化することでプログラマの負担を軽減している．また，プログラムの並列化を支援することにより処理速度の向上を図るものも存在

する．しかしながらこれらを用いて，現在広く普及しているヘテロジニアスな計算機環境の性能を有効活用するプログラムを実装することは困難である．これに対し，本論文で提案する動画像処理プログラミング環境のアプローチはこれまでに説明してきた画像/動画像処理ライブラリや言語のそれとは大きく異なる．提案する環境では，後述する動画像処理ライブラリ RaVioli を改良することで，簡単な記述のみでヘテロジニアスな計算機環境を有効に活用可能なプログラムを開発可能にする．さらにこの提案環境では，プログラマから解像度という概念を隠蔽し，処理パターンに依存しない統一的な記述方式を採用する，これにより直感的な動画像処理プログラミングを実現する．また，このように解像度を隠蔽するためプログラムにおける並列性を抽出することが容易になる．これを利用し，提案する環境では自動的に並列化プログラムを生成する．このとき，単に並列プログラムを生成するだけでなく，CPU と GPU が協調動作するようなプログラムを生成可能とする．これによりプログラマは容易に，ヘテロジニアスな環境を有効に活用するプログラムを開発することが可能となる．

3　動画像処理ライブラリ RaVioli

本章では，本研究の対象となる動画像処理ライブラリ RaVioli の概要について述べる．その後，RaVioli の GPU 向け拡張である RaVioli/CUDA について説明し，その問題点について述べる．

3.1　RaVioli 概要

動画像を構成する要素である「画素」や「フレーム」という概念は，そもそも人間の脳内における視覚情報の認識過程には存在しない．しかし量子的に情報を扱う必要のある計算機上では，画像を画素の集合として，動画像をフレームの集合として扱わなければならない．またこのように量子化されているが故に，動画像処理プログラムを記述する際は for 文などのループ構造を用いて，これらの構成要素に対して繰り返し処理を施す必要があるが，この繰り返し処理も動画像処理の本質ではない．

これらの問題に対し RaVioli は，プログラマから解像度の概念を隠蔽するプログラミングパラダイムを提供している．ここで解像度とは空間解像度と時間解像度の 2 つであり，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する．RaVioli は，1 フレーム中の画素配列や画像の幅・高さ，フレームレートなどをプログラマから隠蔽し， RaVioli 側でこれら全てを管理している．これによりプログラマは解像度を意識せずに動画像処理を記述できる．

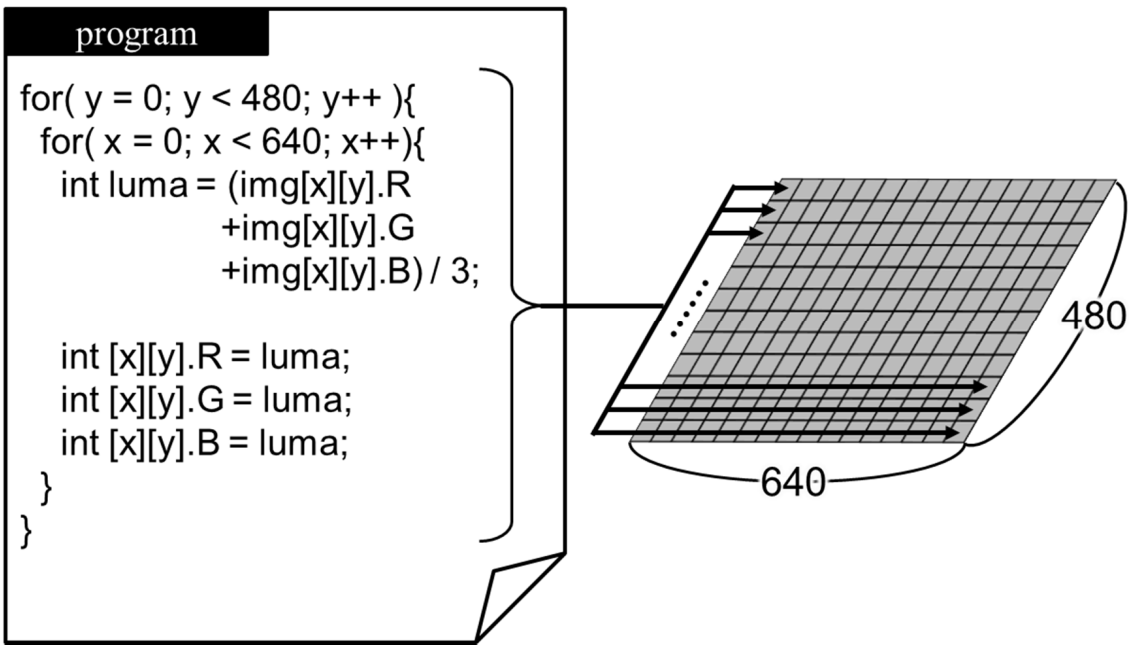

図 1: 一般的な画像処理プログラム

一般に画像処理では，画像の構成要素に対する処理を，画像全体または任意の範囲に繰り返し適用するものが多い．例えばカラー画像からモノクロ画像への変換や色の反転などの処理では処理単位は画素であり，ぼかしやエッジ強調などの近傍処理では，処理単位は画素およびその近傍画素集合である．また，テンプレートマッチングなどの画素の集合を扱う処理では処理単位は部分画像である．そしてこれらの処理は，一般的に**図 1** のようにループイテレーションを用いて記述される．例えばカラー画像をグレースケールに変換する場合，プログラマはまず，画像の幅と高さに合わせてイテレーション回数を指定したループ文を記述する．そして，各画素をグレースケール化するための処理を最も内側のループ内に記述する．これらの記述により，グレースケール化を画像中の全ての画素に繰り返し適用できる．このようにループを用いる場合，プログラマは画像の幅と高さを意識してプログラムを記述しなければならない．

一方 RaVioli では，図 1 の最も内側のループ内の記述に対応する，画像の構成要素に対する処理のみを関数として定義する．そして，その関数を RaVioli が提供しているメソッドに渡すことで，画像中の全ての画素に対して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 を用いてカラー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．RaVioli は画像情報

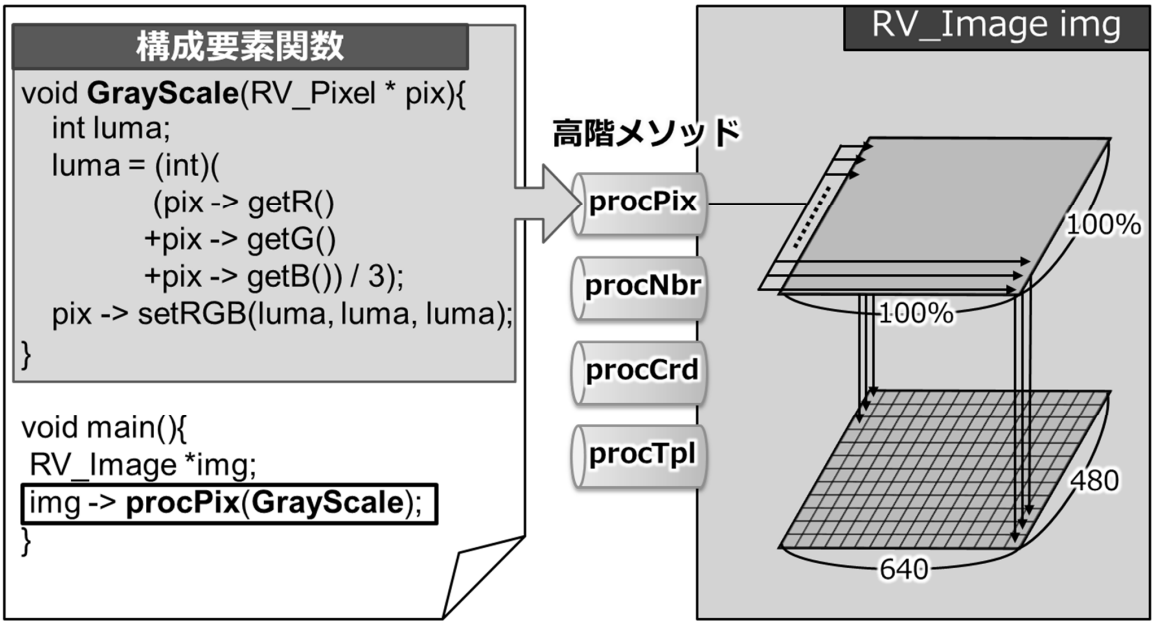

図 2: RaVioli で記述した画像処理プログラム

を RV_Image クラスにカプセル化している．このカプセル化された画像に対して処理を適用するには，RV_Image インスタンスの高階メソッドに構成要素関数を渡すのみでよい，この例では，RV_Image インスタンス img の高階メソッド procPix() に，1 画素をグレースケール化する処理が記述された構成要素関数 GrayScale() を渡している．そして，高階メソッド procPix() は img が持つ画像の全ての画素に，GrayScale() を繰り返し適用する．なお，RaVioli は処理パターンに応じて複数種類の高階メソッドを提供している．例えば図 2 で用いられている，単一画素を処理単位とする procPix() の他にも，処理対象画素および近傍画素を処理単位とする procNbr()，処理対象画素の座標を変換する procCrd()，部分画像を処理単位とする procTpl() などを提供している．プログラマはこれらの中から，自身が定義した構成要素に対する処理に適した高階メソッドを選択し，構成要素関数を渡すことで，画像全体に処理を適用することができる．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．

次に，RaVioli で記述した動画像処理プログラムとその処理の様子を**図 3** に示す．画像情報を RV_Image クラスにカプセル化しているのと同様に，動画像におけるフレームのサイズや数，フレームレートといった動画像に関する情報を，RV_Streaming クラスにカプセル化している．そして，プログラマは動画像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する．ここで，単一フレームに対する処理とは，

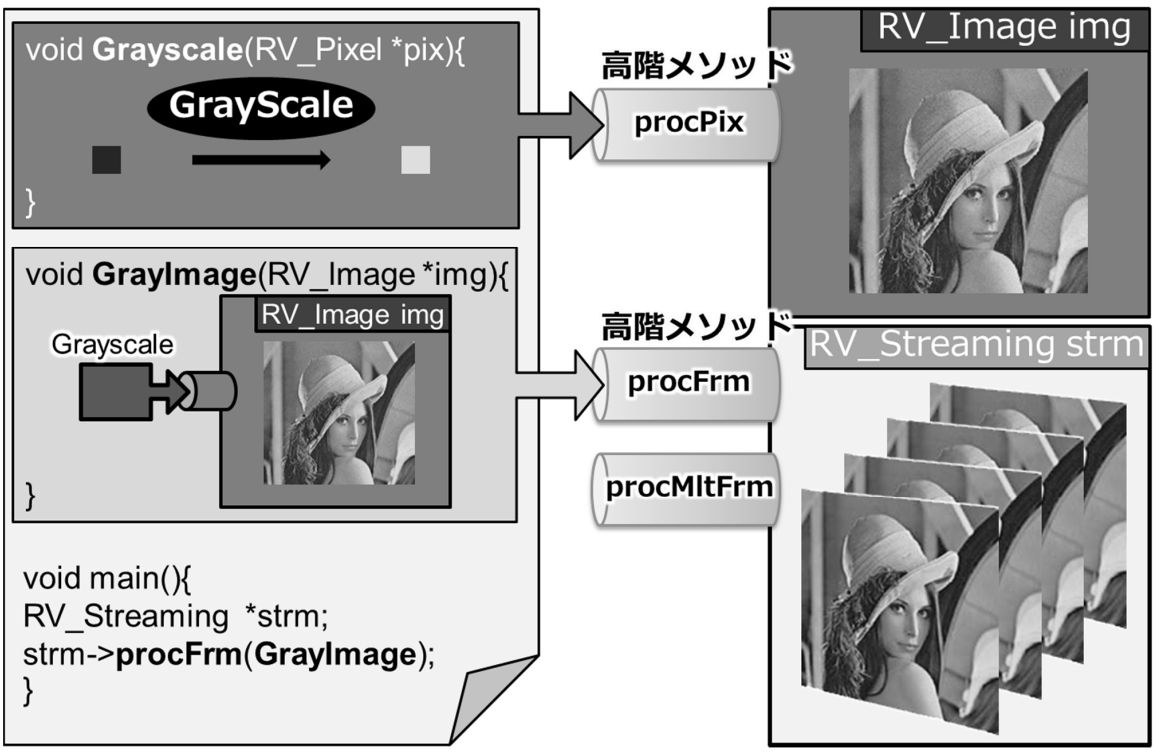

図 3: RaVioli で記述した動画像処理プログラム

先ほどの画像に対する処理そのものである．図 3 の例では，この単一フレームに対する処理は構成要素関数 GrayImage() が相当する．この GrayImage() 内では，GrayScale() などの構成要素関数を引数にとる RV_Image クラスの高階メソッド呼び出しが含まれる．そして，その関数 GrayImage() を RV_Streaming クラスの高階メソッドに渡すことで，動画像中の全てのフレーム対して処理を適用することが可能である．このような処理構造を用いて動画像処理を抽象化することで，プログラマは動画像の構成要素であるフレームの幅や高さ，フレームレートを意識することなく動画像処理プログラムを記述できる．

前述したように，RaVioli を用いて記述されたプログラムでは，画像および動画像の構成要素に対する処理が関数として定義されるため，並列処理単位が明確になる．そのため，自動並列化の実現が比較的容易である．そこで RaVioli では汎用マルチコアプロセッサや GPU，Cell/B.E.[17] など様々な環境に対応した自動並列化が実現されている．これらのうち，現在アクセラレータコアとして最も普及している GPU に対応させた RaVioli の拡張が RaVioli/CUDA である．RaVioli/CUDA は，NVIDIA 社が提供している GPU コンピューティング向け統合開発環境である CUDA をバックエンド

として実装されている．本論文では，この RaVioli/CUDA のさらなる拡張について述べる．

3.2 RaVioli/CUDA

本節ではまず，CUDA の概要を説明する．そして，RaVioli/CUDA とその問題点について述べる．

3.2.1 CUDA 概要

1 章で述べたように，低消費電力で高い演算性能を実現できることから，汎用プロセッサコアとアクセラレータコアを搭載したヘテロジニアスな環境が一般的になってきている．そのような環境の中でも特に，アクセラレータコアとして GPU を用いた環境が広く普及している．この 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．また現在，GPU コンピューティング向け統合開発環境として CUDA が NVIDIA 社により提供され，広く用いられている．

GPU は，世代や製品により多少の差異はあるが，そのアーキテクチャモデルはほぼ同じである．ここで NVIDIA 社の GPU アーキテクチャの例として，GF104/114[18] の概要を**図 4** に示す．GPU チップの内部には多数の SM（Streaming Multiprocessor）が存在し，SM 内には CUDA Core と呼ばれる演算器や，ロード・ストアユニット，複雑な命令を実行するための SFU（Special Function Unit）などのユニットが存在する．GPU は大量の CUDA Core のそれぞれに対してスレッドを割り当て，それらを並列実行する SIMT（Single Instruction, Multiple Thread）型の実行方式を採用している．一般的に，画像処理にはデータ並列性が存在するため，この SIMT 型を採用することで GPU では画像処理に対する高い演算性能を実現している．以下では，CUDA の階層的なスレッド管理やメモリ構成，プログラミングモデルについて述べる．

階層的なスレッド管理

前述したように，GPU は CUDA Core を用いて大量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 CUDA Core で最大 $(2^{31}-1)\times 65535\times 65535\times 512$ 個のスレッドを使用することが可能である．さらに，この大量のスレッドは**図 5** に示すように階層的に管理される．なお CUDA ではスレッドの集合をブロック，ブロックの集合をグリッドと呼ぶ．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 次元的に管理される．またグリッド内のブロックは，Fermi[19] よりも前のアーキテクチャでは x 軸方向，y 軸方向の 2 次元的に，Fermi 以

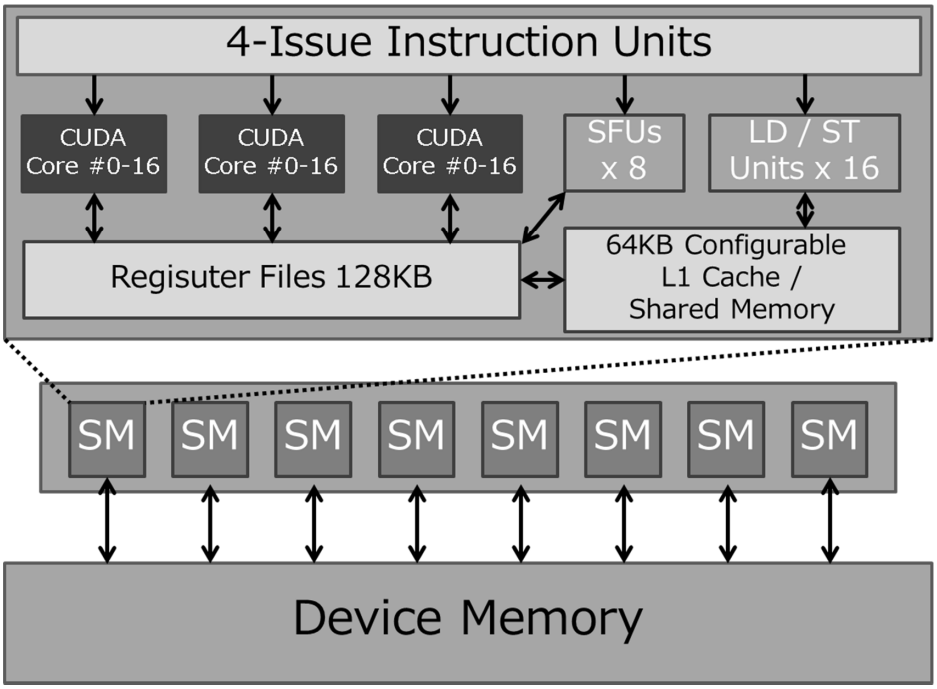

図 4: GPU のアーキテクチャ（GF104/114）

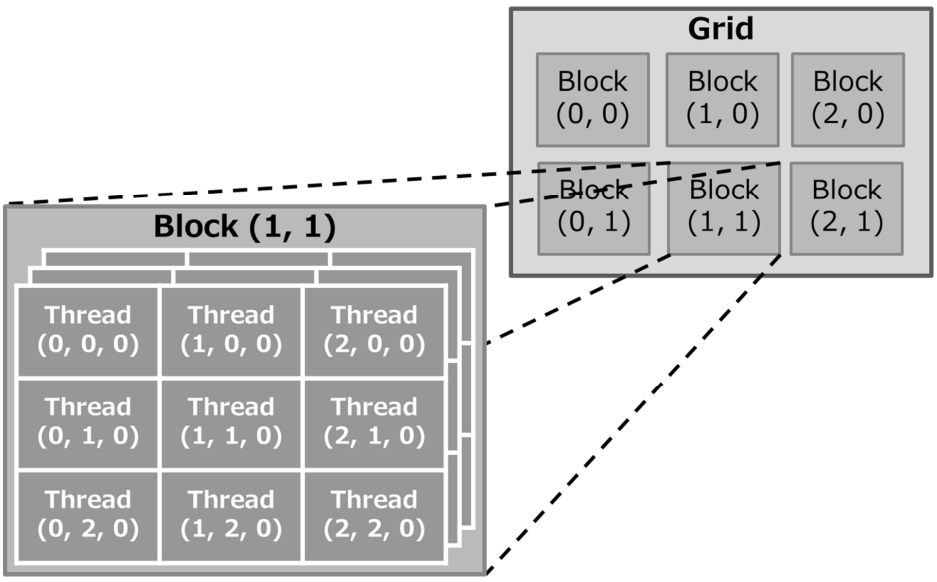

図 5: 階層的なスレッド管理（実行構成）

降のアーキテクチャではスレッドと同様に x 軸方向，y 軸方向，z 軸方向の 3 次元的に管理される．そして，このように管理されているグリッド内のブロックやスレッドをどのように構成するかを**実行構成**と呼ぶ．この実行構成は，GPU 上で実行される関数を呼び出す際に指定する必要があり，GPU の性能を引き出すためには，CUDA Core

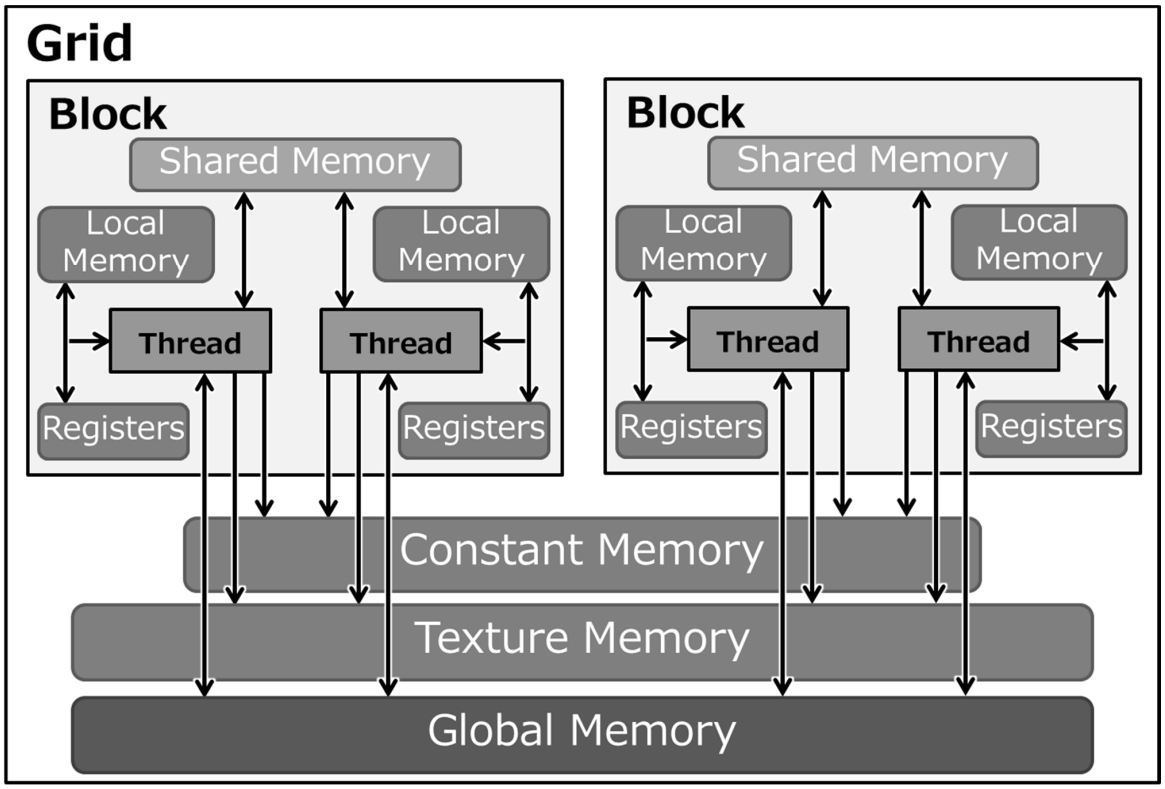

図 6: CUDA のメモリ構成

をはじめとする GPU ハードウェアの資源量や処理内容を考慮して，実行構成を適切に設定する必要がある．

メモリ構成

CUDA では，CPU 側のメモリをホスト・メモリ，GPU 側のメモリをデバイス・メモリと呼び，この 2 種類のメモリはそれぞれ独立したアドレス空間を持つ．そのため CPU-GPU 間でデータを共有するには，デバイス・メモリ上に配置するデータ用の領域確保や，その領域に対するデータ転送を，CUDA が提供する API を用いてプログラマが明示的に記述する必要がある．なお，CUDA で利用可能なデバイスメモリには**図 6** に示すように，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・メモリと複数の種類がある．

ここで，各デバイス・メモリの詳細を**表 1** に示す．表中の「アクセス」の欄はスレッドからのアクセス権を示し，R/W は読み書き可能，R は読み出しのみ可能であることを表す．また,「範囲」の欄はどの単位でその領域にアクセス可能かを示す．この欄がブロックやグリッドとなっているものはそれらの単位で共有されるが，スレッドとなっているものはスレッドごとに固有の領域であり共有されない．この表に示すよう

表 1: デバイス・メモリ

種類	場所	キャッシュ	アクセス	範囲
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	される（Fermi 以降）	R/W	グリッド
テクスチャ・メモリ	チップ外	される	R	グリッド
コンスタント・メモリ	チップ外	される	R	グリッド

に，各メモリはアクセス速度やアクセス範囲，キャッシュの有無などがそれぞれ異なるため，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．しかし，各メモリを利用するためには，それぞれ個別に用意されたアクセス用 API や変数修飾子を使うといった特有の記述が必要である．

プログラミングモデル

CUDA を用いて記述されたプログラムは，CPU 上で実行されるホスト・コードと GPU 上で実行されるデバイス・コードで構成される．このうち，ホストコードには通常の C 言語プログラムに加えて，GPU 上で実行する関数を呼び出すための処理を記述する．このとき，その関数を実行するスレッド数も併せて指定する必要がある．なお CUDA では，この GPU 上で実行する関数を**カーネル関数**と呼ぶ．一方，デバイス・コードにはカーネル関数の定義を記述する．このカーネル関数内では，関数実行時に各スレッドがどのメモリアドレスにアクセスするか指示する必要がある．なお CUDA では，各スレッドに固有の ID が割り当てられており，プログラマは CUDA のビルトイン変数を用いて，このスレッド ID を指定することができる．これを利用し，プログラマは各スレッドがどのメモリアドレスアクセスするか指示する．

3.2.2 RaVioli/CUDA 概要

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管理することにより，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・メモリの構成や，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，これらの記述は画像処理の本質ではないため，プログラマの負担が大きくなってしまう．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ことでプログラマの負担を軽減している．

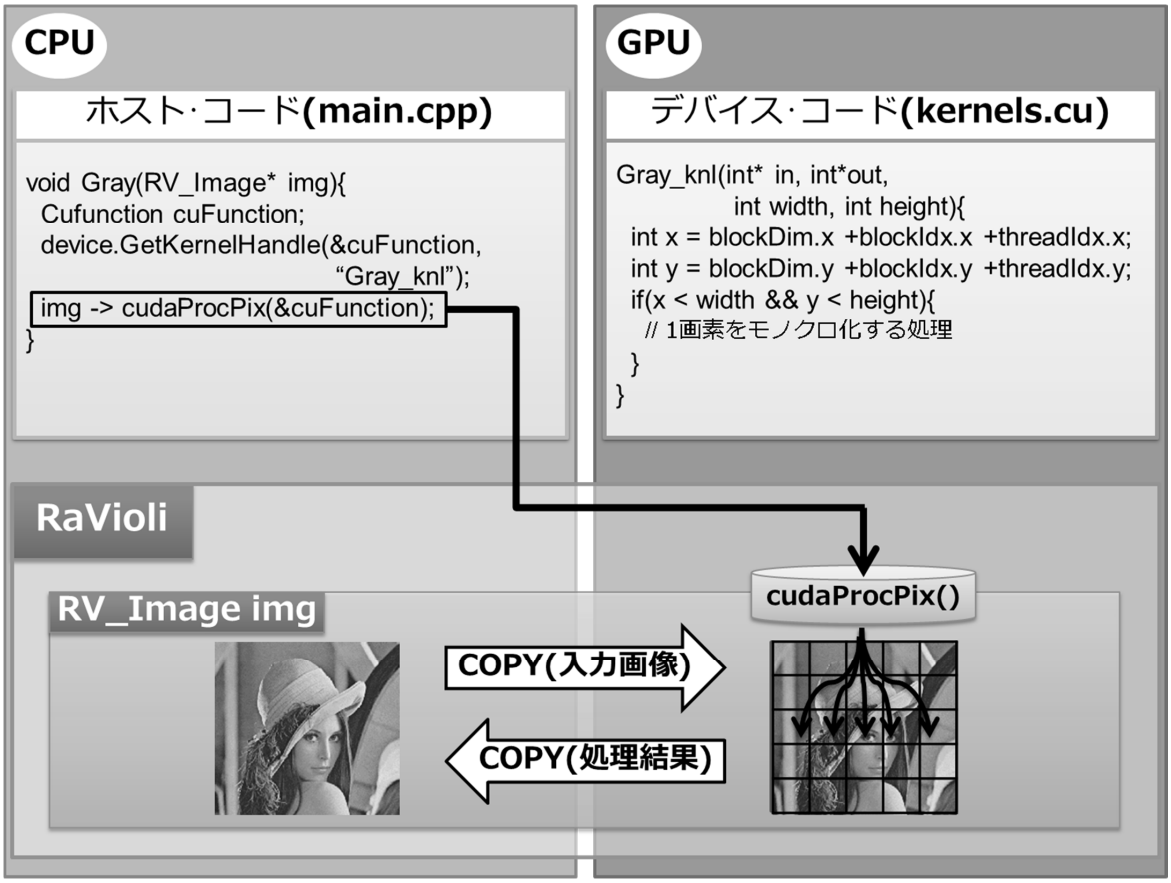

図 7: RaVioli/CUDA のインタフェース

RaVioli/CUDA のインタフェースを**図 7** に示す．プログラマはまず，デバイス・コード内で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，各スレッドが担当する画素を決定するための処理を記述する．なお RaVioli/CUDA では，実行時にホストからカーネル関数を動的にロードするために，カーネル関数をモジュールとして扱う必要がある．そこで，カーネル関数を別ファイル（*.cu）内に記述することでこれを実現する．一方ホスト・コードでは，カーネル関数を実行するための関数を定義する．この関数の引数には画像データである RV_Image インスタンスを指定する．関数のボディには，モジュールをロードするための処理とカーネル関数のハンドルを取得するための処理を記述する．そして，取得したハンドルを 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，ホストからデバイスへデータが転送される．さらに実行構成も自動的に定義され，カーネル関数が高階メソッドに応じて画素や近傍画素集合といった処理

対象に適用される．その後，画像に対する全ての処理が終わると処理結果がホストに返される．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ライブラリ内部で自動的に行う．

3.2.3 RaVioli/CUDAの問題点

RaVioli/CUDAでは，CUDAを使用するために必要となるメモリ確保やデータ転送のためのAPIの記述をライブラリ内部で隠蔽することにより，プログラマの負担を軽減している．しかし，RaVioliとRaVioli/CUDAには共通して，画像処理を十分に抽象化できていないという問題が存在する．3.1項で述べたように，RaVioliとRaVioli/CUDAは，処理パターンに応じて複数種類の高階メソッドを提供している．そのため，RaVioliやRaVioli/CUDAを使用して画像処理プログラムを記述する際には，プログラマは目的とするアプリケーションの処理内容を意識し，その処理に適した高階メソッドを選択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り，プログラマにはCUDAの基本的な知識が必要となる．

さらに，RaVioli/CUDAにはヘテロジニアスな環境を十分に活かし切れていないという問題点も存在する．3.2.2項で述べたようにRaVioli/CUDAでは，プログラム実行時に自動的にホストからデバイスへ画像データを転送し，そのデータに対してカーネル関数を適用する．つまりCPUは，GPUへ画像データを転送してからその処理結果が転送されてくるまでの間，遊休状態となってしまう．これではCPU/GPUの2つのユニットを効率的に活用できない．

4 動画像処理プログラミング環境の提案

本章ではまず，本論文で提案するプログラミング環境の概要について述べる．次に，提案する動画像処理記述言語の仕様および文法について述べる．そして，RaVioli/CUDAを拡張することで実現する2種類のCPU/GPU協調動作手法について説明する．

4.1 提案手法の概要

3.2.3項で述べたようにRaVioli/CUDAには，動画像処理を抽象化しきれていない点，そしてヘテロジニアスな計算機環境の性能を十分に活かしきれていない点という2つの問題点が存在する．これらの問題点を踏まえて，RaVioli/CUDAよりも抽象度の高い動画像処理記述言語を提案する．さらに，CPU/GPU間の協調動作を支援可

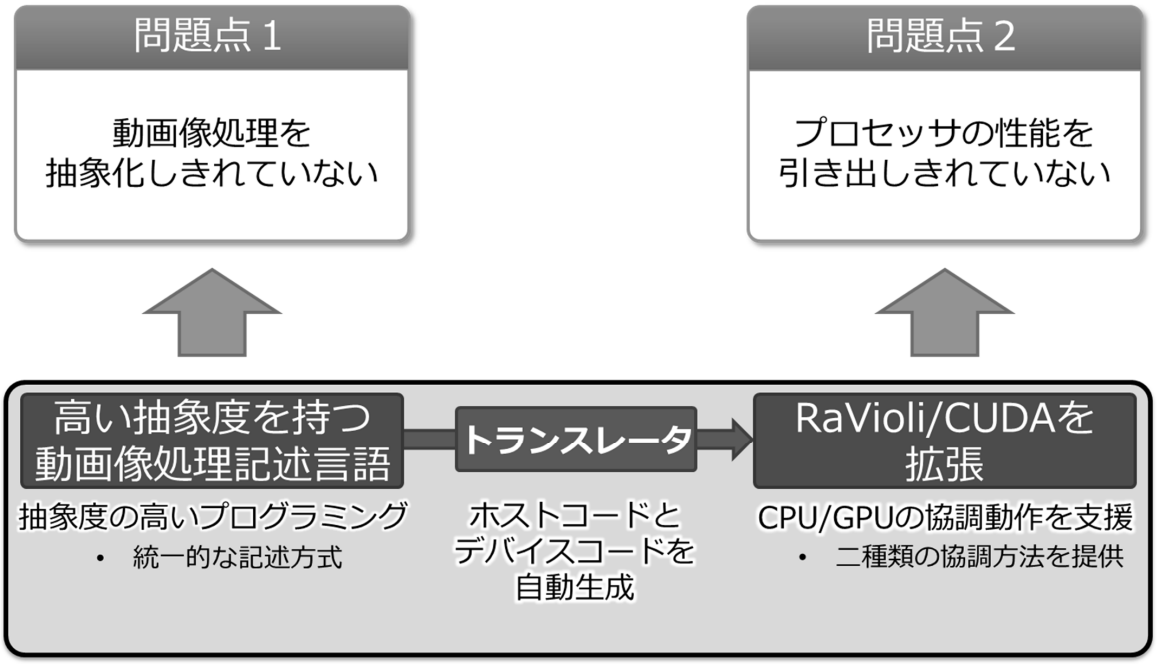

図 8: 提案手法の概念図

能とするために RaVioli/CUDA を拡張する．まず提案する言語では，津邑研究室でこれまでに提案されてきた画像処理向け言語 [20] の仕様を踏襲し，処理パターンに依存しない統一的な記述方式を採用する．この統一的な記述方式により，プログラマは RaVioli/CUDA よりも直感的で抽象度の高いプログラミングが可能となる．これに加え，画像処理および動画像処理に存在するデータ並列性およびタスク並列性に着目し，2 種類の CPU/GPU 協調動作手法を RaVioli/CUDA の拡張として実現する．この 2 種類の協調動作手法では，画像処理および動画像処理に対して CPU/GPU が協調的に動作し，どちらかのユニットが遊休状態になることを抑制する．これにより，ヘテロジニアスな計算機環境をより有効に活用できるようにする．さらに，提案言語で記述したプログラムから，拡張した RaVioli/CUDA を用いたプログラムへと自動変換するトランスレータを提供することで，プログラマが CUDA の知識を一切持たなくとも，CPU/GPU を混載したシステムの性能を引き出せるプログラミング環境を提供する．

以上で述べた提案手法を概念的に表すと**図 8** のようになる．高抽象度な言語と CPU/GPU 間の協調動作を支援可能とする RaVioli/CUDA の拡張により，RaVioli/CUDA に存在する 2 つの問題点は解消される．さらにトランスレータを提供することで，高抽象度な言語と RaVioli/CUDA の拡張により得られる利点を活かしたプログラミング環境を実現することができる．つまりこの環境を利用することにより，プログラマは

簡単な記述のみで，ヘテロジニアスな環境をより効率的に利用可能なプログラムを開発することが可能となる．

4.2 高い抽象度を持つ動画像処理記述言語

本節では，本論文で提案する動画像処理記述言語の仕様及び文法について述べる．

4.2.1 統一的な記述方式

一般的な画像処理プログラムでは 3.1 節で述べたように，画素や部分画像といった画像の構成要素・構成部分に対する処理と，その処理を画像全体に適用するためのループ文の記述が必要である．これは動画像処理プログラムにおいても同様であり，構成要素である動画像フレームに対する処理と，その処理を全フレームに適用するループ文を記述する必要がある．

一方で，既存の RaVioli や RaVioli/CUDA は，画像の画素や幅，高さといった情報，そして動画像のフレームをライブラリ内に隠蔽することでこれらを考慮することなく動画像処理を記述できるプログラミングパラダイムを提供してきた．提案言語においてもこの方針を採用し，画像や動画像の解像度およびフレームレートを意識する必要のないプログラミングパラダイムを提供する．さらに提案言語では，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を指定するだけで動画像処理プログラムを記述することを可能とする．この，処理と適用範囲を記述するという方針は既存の RaVioli や RaVioli/CUDA と似ている．しかし，RaVioli や RaVioli/CUDA を用いる場合では 3.2.3 節で述べたように，プログラマは目的とする画像処理や動画像処理の内容に応じて，それぞれ適切な高階メソッドを選択する必要があった．具体的には既存の RaVioli では以下の処理パターンに合わせて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．

これに対し本言語では，処理単位と処理範囲を指定することでループ文を省略し，記述方式を大きく変更せずとも，上記の処理パターンを統一的に記述可能とする方式を採用する．これにより，提案言語では既存の RaVioli や RaVioli/CUDA よりもさらに抽象度の高いプログラミングが可能となる．

本言語を用いたプログラムは基本的に以下のような構造をとる．

プログラムの基本構造

```
入力変数 > 関数名 > 出力変数 {
    施したい処理内容
}
```

ここで，入力変数および出力変数とは入出力を表す変数であり，“{” と “}” で囲まれた部分にはこれらの変数に対する処理を記述する．具体的には，以下に示す形式で処理内容を記述できる．

統一的な記述方式

```
処理単位 @ 処理範囲 {
    処理単位に対する処理
}
```

ここで**処理範囲**とは，処理を施したい画像あるいは動画像の範囲および対象のことである．また**処理単位**とは，処理範囲内において繰り返し処理を適用する際の単位となる構成要素のことであり，例えば 1 対 1 写像では 1 つの画素を，多対 1 写像では部分画像を意味する．プログラマはこの処理単位に対する処理を記述することによって，画像および動画像全体に処理を施すことができる．

4.2.2　処理内容に応じた言語仕様

本項では，提案言語を用いて 4.2.1 項で示した 3 つの画像処理パターンを記述する方法を説明する．さらに，提案言語を用いて動画像処理を記述する方法も説明する．

(a) 1 対 1 写像

本言語を用いて 1 対 1 写像を記述する方法を，画像に対してグレースケール化を施すプログラムである**図 9** を例に説明する．このプログラムでは，入力変数および出力変数は img1，関数名は Grayscale として定義されている．この例のように，入力変数と同一のものが出力変数として指定されている場合は，関数の処理結果を入力データに上書きすることを意味する．なお，入力データを処理結果で上書きしたくない場合

```
1 (image)img1 > Grayscale > img1{
2     (pixel)p1@img1{
3         ave = (p1.R + p1.B + p1.G) / 3;
4         p1.{R, G, B} = {ave, ave, ave};
5     }
6 }
```

図 9: 提案言語で記述したグレースケール化プログラム

表 2: 提案言語が提供する型の種類

型名	型が表す対象
pixel	単一画素
box	部分画像
image	単一画像
stream	動画像
array	配列（ただし画素配列とは異なる用途向け）

には，関数呼び出しの際に入力変数と出力変数を異なる変数とすることで対応できる．ここで，提案言語では画像処理および動画像処理プログラムの記述を容易にするために，いくつかの型を提供している．提案言語が提供する型の種類を**表 2** に示す． 画像処理のための基本的な型として単一画素を表す pixel 型や部分画像を表す box 型を提供するほか，画像である 2 次元画素配列とは異なる用途向けの配列変数のために array 型なども提供している．この array 型は，例えば複数の処理を段階的に適用し最終的な結果を得るような画像処理において，処理の中間結果を配列を用いて保持したい場合などに利用される．なお，型が宣言されていない変数はスカラ変数として扱われる．各型の変数は，変数名の直前に “(” と “)” で囲って型名を表記することで宣言できる．図 9 の例の場合，1 行目において，入力変数 img1 は単一画像を意味する image 型で宣言されている．

ではここから，入力変数および出力変数である img1 に対する処理内容が記述されている 2〜5 行目について説明する．2 行目の “(pixel)p1@img1” は，以降で pixel 型の処理単位 p1 に対する処理が定義されていること，その p1 は img1 内の任意の要素を表すこと，そして定義されている p1 に対する処理が img1 を構成する処理単位全てに適用されることを示している．すなわち 3，4 行目に記述された画素 p1 に対する処理は，

img1 内の全画素要素に適用される．まず，3 行目では，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 行目では，ave の値が p1 の R 値，G 値，B 値として設定されることで，画素 p1 がグレースケール化される．なおこの例の，“p1.{R, G, B}” のように “{” と “}” を使うことにより，複数の変数をタプル化してまとめて扱うこともできる．

提案言語ではこのような記述方式を採用することで，既存の RaVioli や RaVioli/CUDA と同様に，通常の画像処理におけるループのイテレーション回数や，画像内の画素数などについてプログラマが考慮する必要がなくなる．さらに，プログラマは処理単位と処理範囲を直感的に指定することが可能となる．また，型を提供することで RaVioli や RaVioli/CUDA よりもプログラムの記述性・可読性が向上する．

(b) 多対 1 写像

フィルタ処理などに代表される多対 1 写像では，座標の異なる複数の画素に対する参照が必要となる．そのため，複数画素を指定するための何らかの座標表現が必要となる．提案言語では，処理対象画素からの相対位置を表す**相対座標表現**を導入することでこれに対処する．相対座標表現は “<” と “>” で囲んで表記する COORD 指定子を用いる．この COORD 指定子は pixel 型変数の直前に記述することで，当該変数が示す画素からの相対的な座標を指定できる．例えば，相対座標表現を用いた場合の処理対象画素とその 8 近傍の画素との位置関係は**図 10** のようになる．“<” と “>” で囲まれたタプルの要素は，それぞれ処理対象となる画素を起点として，第一要素は x 軸方向の相対座標を表し，第二要素は y 軸方向の相対座標を表している．なお，pixel 型変数が示す画素の座標要素を参照することも可能であり，例えば画素 p1 の x 座標を参照したい場合は “p1.x” と表記する．

ここで，本言語を用いて多対 1 写像を記述する方法を，画像に対してエッジ抽出を施すプログラムである**図 11** を例に説明する．エッジ抽出は，グレースケール化された画像を使用し，注目画素とその近傍画素の画素値の差が大きい場合に，注目画素を黒色に設定し，そうでない場合には注目画素を白色に設定する処理である．その結果，エッジ部分を黒色で示した 2 値画像が得られる．では，図 11 のプログラムに沿って，提案言語を用いてエッジ抽出プログラムを記述する方法を説明していく．図 11 の 4〜6 行目では，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 に代入している．このとき，p1 を起点とした x 軸方向および y 軸方向のそれぞれの相対座標を指定し，p1 とその周囲 8 画素を用いた演算が記述されている．そして 7 行目

<-2, -2>	<-1, -2>	<0, -2>	<1, -2>	<2, -2>
<-2, -1>	<-1, -1>	<0, -1>	<1, -1>	<2, -1>
<-2, 0>	<-1, 0>	対象画素	<1, 0>	<2, 0>
<-2, 1>	<-1, 1>	<0, 1>	<1, 1>	<2, 1>
<-2, 2>	<-1, 2>	<0, 2>	<1, 2>	<2, 2>

図 10: 処理対象画素との位置関係

```
1 (image)img1 > Edge > img1{
2    threshold = 127;
3    (pixel)p1@img1{
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   + <1,1>p1.R    + <1,1>p1.R;
7       if(temp > threshold)
8         p1 = #black;
9       else
10        p1 = #white;
11   }
12 }
```

図 11: 提案言語で記述されたエッジ抽出プログラム

で，閾値を保持する変数 threshold と temp を比較することで，その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 p1 の画素値に設定する．

なお，8，10 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．通常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)

```
1 {(image)img1, (image)img_tp} > TPmatch > (image)img2{
2    min = INT_MAX;
3    (box)[img_tp]img_window@img1{
4       sum = 0;
5       ((pixel)p1, (pixel)p2)@(img_window, img_tp){
6            sum += abs(p1.R − p2.R) + abs(p1.G−p2.G) + abs(p1.B−p2.B);
7       }
8       if(sum < min){
9            min = sum;
10           {x,y} = img_window.{x,y};
11      }
12     }
13   img1 > drawBox(#red,<x,y>,[img_tp]) > img2;
14 }
```

図 12: 提案言語で記述したテンプレートマッチングプログラム

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，逆に茶色を構成する RGB の値を想起することも困難である．そこで，本言語では “#black” や “#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 または #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 CSSColorModule[21] の定義に準じている．

(c) 多対多写像

ここでは，本言語を用いて多対多写像を記述する方法を，画像に対してテンプレートマッチングを施すプログラムである**図 12** を例に説明する．テンプレートマッチングでは，入力として処理対象画像とテンプレート画像を使用する．このように入力画像が複数存在する場合は，1 行目のように入力変数を “{” と “}” で囲んで複数宣言する．3 行目の “(box)[img_tp]img_window@img1” は，処理単位が img_window，処理範囲が img1 であることを示している．ここで，img_window は部分画像を表す box 型で宣言されている．この box 型は変数名の直前で image 型の変数を “[” と “]” で囲むこと

により，その image 型の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 には，img_tp と同じ大きさが設定される．5 行目の “((pixel)p1, (pixel)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 内の同座標の画素 p2 を同時に処理することを示している．5〜7 行目では，部分画像 img_window とテンプレート画像 img_tp の差分を取ることで相違度が求められ，その相違度が変数 sum に加算されている．そして 8〜11 行目で，相違度を保持する変数 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が更新され，img_window の開始座標である左上の x，y 座標が変数 x，y にそれぞれ代入される．

13 行目は，画像 img1 に関数 drawBox() が適用されていることを表す．img1 は関数 drawBox() の入力であり，出力は img2 となる．この drawBox() 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を描画できる．この長方形の描画処理のように，画像処理において使用頻度の高いさまざまな処理を容易に記述可能とするため，本言語では様々な組込み関数を用意している．例えば drawBox() 以外にも，drawLine() という関数が用意されており，この関数を画像に適用すると，その画像内に直線を描画できる．これらの関数は以下のように使用する．

組込み関数

drawBox(#color, $< x, y >$, [img])

drawLine(#color, rho, theta)

関数 drawBox() を適用することによって描画される長方形の大きさは，image 型の変数を “[” と “]” で囲んで与えることによってその image 型の変数名が表す画像と同じ大きさになるよう指定できる．また，長方形の左上の位置を座標 $< x, y >$ で指定することにより，長方形が描画される位置を指定できる．一方，関数 drawLine() は，画像の左上を原点とし，原点からの距離を “rho”，x 軸からの角度を “theta” とする直線を描画できる．両方の関数に共通する #color は，本項の (b) で述べた COLOR 変数を用いて枠線や直線の色を指示するための引数である．

(d) 動画像処理プログラム

提案言語を用いて動画像処理を記述する場合，本項でこれまでに述べてきたような画像処理のための構成要素関数に加えて，動画像処理のための関数を定義する．本言

```
1 (image)img1 > Grayscale > img1{
2    //img1 をグレースケール化
3 }
4
5 (stream)st1 > StreamGray > st1{
6    (image)frame1@st1{
7       frame1 > Grayscale > frame1;
8    }
9 }
```

図 13: 提案言語で記述した動画像処理プログラム

語では入力変数および出力変数として，動画像を表す stream 型の変数を用いることで動画像処理のための関数を定義できる．

本言語を用いて動画像処理を記述する方法を，入力動画像に対してグレースケール化を施すプログラムである**図 13** を例に説明する．まず 1〜3 行目で定義されている関数 Grayscale は本項の (a) で述べた画像をグレースケール化する関数である．つぎに，5〜9 行目で定義されている関数 StreamGray は動画像処理のための関数であり，入力変数および出力変数に stream 型変数である st1 が指定されている．この関数内では，stream 型の変数 st1 に対する処理が定義されている．6 行目の “(image)frame1@st1” は処理範囲が st1，処理単位が image 型の frame1 であることを表しており，この frame1 は st1 内の任意の動画像フレームに対応する．7 行目の記述によって，1〜3 行目で定義された処理 Grayscale() が，st1 に含まれる全ての frame1 に適用される．この Grayscale() のような単一フレームに対する処理は 1 枚の画像に対する処理と同義である．そのため，本項の (a)〜(c) までで述べてきたような画像処理のための関数を，frame1 のような処理単位に用いられている image 型変数に適用することで，様々な処理を動画像に施すことが可能となる．このようにして，動画像処理の場合も画像処理と同一の方式でプログラムを記述することが可能である．

(e) main 関数の記述

プログラム実行時に最初に呼び出される main 関数の記述方法を，**図 14** に示す 3 つのプログラム例を用いて説明する．まず図 14(i) および (ii) は，提案言語を用いて記述した画像処理プログラムの main 関数である．本言語では，図 14(i) および (ii) の 1 行目のように関数名を “main” とすることで，main 関数を定義できる．このうち図 14(i) の main 関数内では，プログラマがプログラム実行時に引数として指定する入力画像に対

```
(image)in > main > (image)out{
    in > Grayscale > out;
}
```

(i) 画像処理（グレースケール化）

```
(image)in > main > (image)out{
    in > Binary | Edge | Hough | ReHough > out;
}
```

(ii) 画像処理（直線検出）

```
(stream)in > main > (stream)out{
    in > StreamGray > out;
}
```

(iii) 動画像処理

図 14: main 関数の記述方法

して，構成要素関数 Grayscale を適用する処理が記述されている．ではこの図 14(i) に沿って，提案言語を用いた main 関数の記述方法を説明する．まず 1 行目では，main 関数における入力変数と出力変数が，本項の (a)〜(c) でこれまでに述べてきた画像処理のための関数と同様に image 型で宣言されている．このとき，main 関数における入力変数および出力変数は，プログラム実行時の入力画像および出力画像と対応する．そして関数内では，プログラマ自身で定義した画像処理のための関数が入力変数に適用され，その結果が出力変数に渡されている．図 14(i) では，関数 Grayscale が入力画像 in に適用され，結果として画像 out が出力される．

また，一般的に画像処理および動画像処理には，複数の処理を段階的に適用し，最終的な結果を得るものが多く存在する．例えば，画像中の直線の検出は 2 値化，エッジ抽出，ハフ変換，逆ハフ変換の 4 つの段階的処理から構成される．このような場合，プログラムの記述性や可読性を考慮すると，各処理毎に関数を定義できるのが望ましい．そこで，本言語では複数の関数の定義および定義した関数の逐次的な適用を可能にしている．このような複数の関数適用は，図 14(ii) の 2 行目のように処理を適用したい順に左から関数を記述し，UNIX シェルのパイプのように各関数を “|” で接続する形で記述することで実現できる．この例では，Binary，Edge，Hough，ReHough の 4 つの関数がこの順に入力画像 in に適用される．なお，“|” で接続された関数の出力は，次の関数の入力として渡される．ここでは Binary の出力が Edge の入力として渡され，同様に Edge の出力が Hough の入力として，Hough の出力が ReHough の入力として渡される．

次に，動画像処理プログラム記述する場合の main 関数の記述方法を図 14(iii) に示

す．この main 関数内では図 13 の関数 StreamGray を，プログラム実行時に渡される入力動画像に適用している．提案言語を用いて記述した動画像処理の場合は，本項の (d) で述べた動画像処理のための構成要素関数と同様に，入力変数と出力変数を stream 型の変数とする．その他，構成要素関数の呼び出し方法などは画像処理のための main 関数と同様に記述する．このようにして，本言語は複数の処理から構成される動画像処理の記述を可能にし，様々な動画像処理に対応する．

4.3 2 種類の CPU/GPU 協調動作手法

CPU と GPU を混載したシステムの性能を引き出すためには，各ユニットがそれぞれ遊休状態になることなく処理を継続することが重要である．そこで，CPU と GPU が協調動作するように RaVioli/CUDA を拡張することでこれを実現する．提案手法では画像処理，動画像処理に存在する 2 つの並列性に着目し，2 種類の協調動作手法を提供する．

4.3.1 空間的協調

3.1 節で述べたように一般的な画像処理では，1 画素や近傍画素集合などに対する処理がループ文を用いた繰り返し処理により画像全体に適用される．つまり，同一の処理が異なるデータに対して繰り返し適用される．従って，この繰り返し処理にはデータ並列性が存在する．このような場合，入力画像をいくつかの領域に分割し各部分領域を並列に処理することで高速な処理を実現できる．そこで提案手法では，**図 15** のように入力画像を 2 つに領域分割し，各部分領域に対する処理を CPU/GPU にそれぞれ割り当てるという操作を自動的に行う枠組みを提供する．そして，各ユニットは自身に割り当てられた部分領域をそれぞれ並列に処理する．このようにすることで画像に対して CPU と GPU が協調的に処理を行える．その結果，CPU，GPU どちらかのユニットが遊休状態になることを抑制でき，ヘテロジニアスな環境をより有効に活用できると考えられる．本論文ではこのような協調処理を**空間的協調**と呼ぶ．

なお，CPU と GPU は全く異なる構成をもつプロセッサであるため，これらに適した処理はそれぞれ異なっている．また，CPU も GPU も種類や世代ごとにその性能は様々である．そのため，対象とするプログラムの処理内容や，それを実行するプロセッサの性能に応じて空間的協調を利用する際の最適な分割比は異なると考えられる．そこで提案手法では，CPU/GPU へ割り当てる部分領域の大きさの比を容易に指定可能な記述方式を提供する．これにより，プログラマは画像処理プログラムに対して大きな変更を施すこと無く，最も高速化が実現できる分割比を模索することができる．な

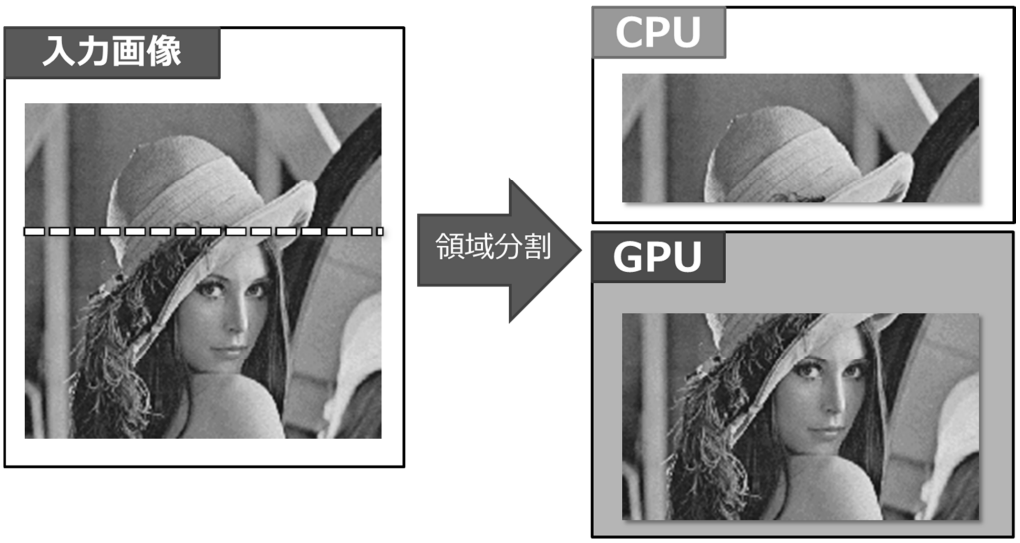

図 15: 空間的協調の分割方法

お，提案言語における空間的協調を利用するための記述方法の詳細については 5 章で述べる．

4.3.2 時間的協調

動画像処理は複数のステージから構成されることが多く，これらのステージ間にはタスク並列性が存在する [22]．例えば，2 値化，エッジ抽出の 2 つのステージから構成される動画像処理プログラムを実行する場合を考える．このようなプログラムを逐次的に実行した場合，その処理は**図 16** のように進んでいく．なお，この図における各ステージの左上の数字は処理対象のフレーム番号を表している．このような処理を，複数のユニットを搭載するプロセッサで実行する場合，各ユニットに対してステージを割り当て，パイプライン的に実行することで高速な処理を実現可能である．例えば，図 16 のような処理を，2 つのユニットを搭載したシステム上でパイプライン的に実行する際には，2 値化とエッジ抽出の処理をそれぞれ別のユニットに割り当て，実行することで**図 17** のようにスループットを向上させることができる．

そこで提案手法では CPU，GPU それぞれのユニットに対して各ステージを割り当て，動画像処理をパイプライン的に実行するプログラムを容易に記述できる環境を提供する．このようにすることで動画像処理に対して CPU と GPU が時間的に協調して動作し，どちらかのユニットが遊休状態になる時間を短縮できる．その結果，ヘテロジニアスな環境をより効率的に利用可能になると考えられる．本論文ではこれを**時間的協調**と呼ぶ．なお一般的に，動画像処理における各ステージの処理量はステージ毎

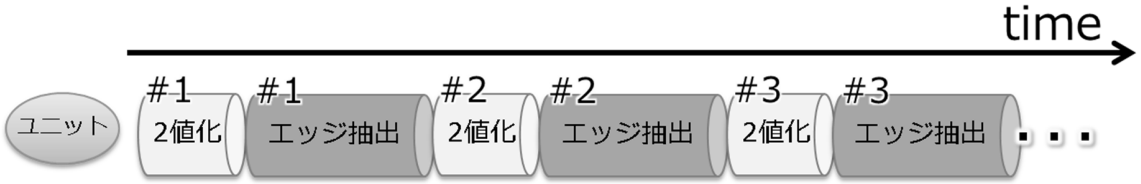

図 16: 動画像処理を実行する様子

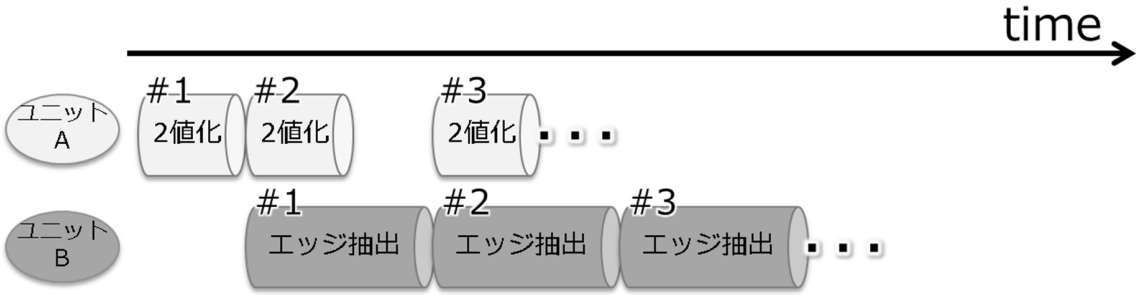

図 17: パイプライン実行による高速化

に大きく異なる．そのため，CPU と GPU のように大きく性能が異なるユニットを用いる場合，各ユニットに対してその性能に見合ったパイプラインステージを割り当てることが可能となる．これは，それぞれ同程度の性能を持っている CPU の複数コアに対してパイプラインステージを割り当てる場合と比較して，各ユニットの処理性能をより有効に活用できる可能性がある．

例えば，図 16 で示した 2 値化とエッジ抽出の 2 つのステージで構成される動画像処理に対して時間的協調を適用する場合を考える．2 値化は 1 対 1 写像，エッジ抽出は多対 1 写像であるため，どちらのステージにも高い並列性が存在する．そのためどちらも GPU に割り当てることで，その実行に要する時間は CPU に割り当てるよりも短くなると考えられる．ここで，これら 2 つのステージにおける処理量を比べると，2 値化の方が処理内容が単純であるため処理量が小さい．そこで，並列処理性能の高い GPU に対して処理量が大きいエッジ抽出を，CPU に対して処理量の小さい 2 値化をそれぞれ割り当てパイプライン的に実行することを考える．この場合の動画像処理は**図 18** のように進み，各ステージの実行に要する時間の差は，処理量の差よりも縮小する．一方で，CPU の複数コアに対してステージを割り当てる場合は，各ステージ間の処理量の差がそのまま処理時間の差に直結する．このような理由から，CPU/GPU に対してパイプラインステージを割り当てる場合では，各ユニットに対してその性能に見合ったステージを割り当てることができるため，CPU の複数コアに対してステージを割り

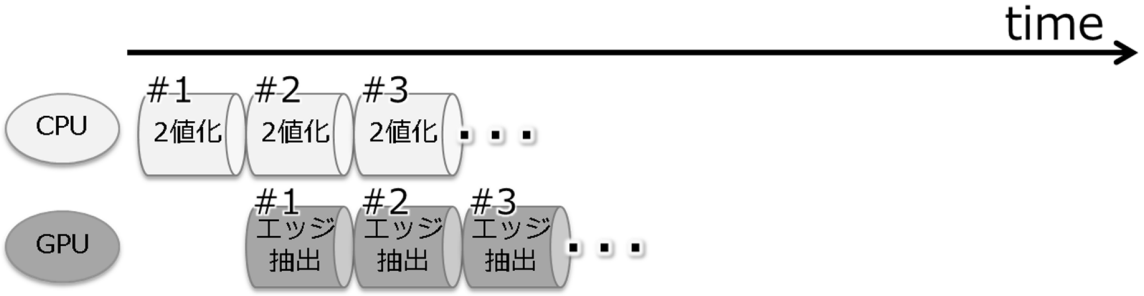

図 18: 時間的協調利用時の実行の様子

当てる場合と比較して，CPU/GPU どちらかのユニットが遊休状態となる時間的割合を小さくすることができる．

5　トランスレータの実装

4.2 節で提案した動画像処理記述言語で記述したプログラムから，4.3 節で述べた CPU/GPU 協調動作を実現するように拡張した RaVioli/CUDA を用いたプログラムを生成するトランスレータを実装した．これにより，プログラマは簡単な記述のみでヘテロジニアスな環境を有効活用できるプログラムを開発することが可能となる．

なお，トランスレータの実装には bison/flex を使用した．bison[23] は構文解析器を生成するパーサジェネレータの一種であり，flex[24] は字句解析器生成ツールの一種である．bison を用いて構文解析部およびコード生成部を，flex を用いて字句解析部を実装した．そして，これらを組み合わせることでプログラムの解析および変換の機能を持つトランスレータを実現した．

本章では，このトランスレータを用いてどのように提案言語プログラムから拡張した RaVioli/CUDA プログラムを生成するのか説明する．また，提案言語における空間的協調および時間的協調を利用する際の記述方法についても述べる．

5.1　空間的協調を利用する画像処理プログラムとその変換方法

空間的協調を利用する画像処理プログラムの例として，提案言語を用いて記述したグレースケール化プログラムを**図 19** に示す．提案言語を用いたプログラムは 4.2 節で述べたように，構成要素関数と main 関数で構成される．ここで，提案言語を用いて空間的協調を利用する場合は 11 行目のように，main 関数内で画像処理のための構成要素関数を呼び出す際，協調時に GPU へ割り当てる画像の割合を “[” と “]” で囲んで指

```
1 /* 構成要素関数 */
2 (image)img1 > Gray > img1{
3     (pixel)p1@img1{
4         ave = (p1.R + p1.B + p1.G) / 3;
5         p1.{R, G, B} = {ave, ave, ave};
6     }
7 }
8
9 /* main 関数 */
10 (image)in > main > in{
11     in > Gray[80] > in;
12 }
```

図 19: グレースケール化プログラム（空間的協調利用）

定する．なおこの割合は 0〜100 までの値で指定可能である．この例では “80” と指定されているため，80%の領域を GPU へ，残りの 20%の領域を CPU へ割り当てて並列に実行する．

では，トランスレータが提案言語プログラムから空間的協調を利用する RaVioli/CUDA プログラムへ変換する流れについて説明する．まずトランスレータは，提案言語プログラム内の画像処理のための構成要素関数を解析し，ホスト・コードに CPU 用の構成要素関数を，デバイス・コードに GPU 用の構成要素関数を生成する．次に，これら 2 つの構成要素関数を空間的に協調して実行させるための空間的協調用関数をホスト・コードに生成する．その後，トランスレータは提案言語プログラム内の変換前の main 関数を解析し，ホスト・コードに変換後の main 関数を生成する．ここでは，画像の読み出し，および書き込みのためのコードと空間的協調用関数を呼び出すためのコードを生成する．

具体的にどのようにコードが生成されるか，図 19 からトランスレータが生成したホスト・コードを**図 20** に，デバイス・コードを**図 21** にそれぞれ示し，これらの図に沿って説明していく． まず，トランスレータは画像処理のための構成要素関数 Gray（図 19，2〜7 行目）を解析し，ホスト・コードである Gray.cpp に CPU 用の構成要素関数である GrayCPU()（図 20，2〜5 行目）を，デバイス・コードである kernels.cu に GPU 用の構成要素関数である GrayGPU() （図 21）を生成する．

ホスト・コードへの GrayCPU() の生成では，まず関数宣言子を指定する（図 20，2 行

```
1 /* 構成要素関数 (CPU) */
2 void GrayCPU(RV_Pixel* p1){
3     int ave = (getR(p1)+getG(p1)+getB(p1)) / 3;
4     p1->setRGB(ave,ave,ave);
5 }
6
7 /* 空間的協調用関数 */
8 void GrayCPUGPU(RV_Image* img, int gpupart){
9     CUfunction cufunc;
10    device.GetKernelHandle(&cufunc, "GrayGPU");
11    //空間的協調用高階メソッド
12    img->procCPUGPU(grayCPU, &cufunc, gpupart);
13 }
14
15 /* main 関数 */
16 void main(int argc, char* argv[]){
17    RV_IOHandler io;
18    RV_Image* in;
19    io.Input(argv[1], in);
20    GrayCPUGPU(in, 80);
21    io.Output(argv[2], in);
22 }
```

図 20: 生成されるホスト・コード（Gray.cpp）

```
1 /* 構成要素関数 (GPU) */
2 extern "C" __global__
3 void GrayGPU(int* in_img, int* out_img, int width, int height){
4     int x = blockDim.x * blockId.x + threadId.x;
5     int y = blockDim.y * blockId.y + threadId.y;
6     if(x<width && y<height){
7         int p1 = in_img[y*width+x];
8         int ave = (getR(p1)+getG(p1)+getB(p1)) / 3;
9         setRGB(&out_img[y*width+x], ave, ave, ave);
10    }
11 }
```

図 21: 生成されるデバイス・コード（kernels.cu）

目）．このとき，関数の引数は構成要素関数の処理単位および処理範囲に応じてトランスレータが自動的に決定する．例えば図 19 の構成要素関数 Gray のように，“p1@img1” と記述されている場合は，処理単位が pixel 型，処理範囲が image 型の 1 対 1 写像であることを意味する．そのため引数の型には，画像の 1 画素を扱う RV_Pixel を指定する．次に，関数本体の処理を生成する（図 20，3〜4 行目）．ここで，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．また，4 行目の setRGB() は RGB それぞれの画素値を引数として受け取り，対象画素にその値を設定する関数である．これらの関数は RaVioli/CUDA によって提供されている．

一方，デバイス・コードへの GrayGPU() の生成では，トランスレータはまず，図 21 の 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を生成する．なお，型指定子の前の “__global__” は関数指定子であり，これは，関数がどこで呼ばれ，どこで実行されるかによって使い分けられる．図 21 のように，ホストから呼ばれデバイスで実行される場合は，“__global__” を指定する必要がある．一方，カーネル関数がデバイスから呼ばれ，同じデバイスで実行される場合は，“__device__” を指定する．また，関数宣言子の引数は構成要素関数の処理単位および処理範囲によって決まる．前述したように図 19 の構成要素関数 Gray では，処理単位が pixel 型，処理範囲が image 型の 1 対 1 写像であるため，この場合の引数は処理対象画像を格納している配列 in_img，処理結果を格納する配列 out_img，画像の幅情報を保持している width，画像の高さ情報を保持している height の 4 つの変数となる．次に，各スレッドに割り当てられている ID などを用いて，各スレッドが画像中のどの画素を処理するかを指定するためのコードを生成する．さらに，カーネル関数が画像の範囲外にアクセスしないようするためのコードを生成する．図 21 の例では 4 行目〜6 行目がこれに相当し，これらも処理単位および処理範囲に応じて使い分けられる．その後トランスレータは，関数本体の処理を生成する（7〜9 行目）．この例では処理単位が画素であるため，処理対象画素の画素値を読み出すためのコードを生成する（7 行目）．そして，その画素に対してグレーケール化を適用するためのコードを生成する（8，9 行目）．以上のようにして，トランスレータは CPU 用および GPU 用の構成要素関数を生成する．

次にトランスレータは，これら 2 つの構成要素関数を空間的に協調して実行するための関数 GrayCPUGPU() を Gray.cpp に生成する（図 20，8〜13 行目）．この空間的協調用関数では，まず GPU 用の構成要素関数のカーネルハンドルを取得する（図 20，9〜10 行目）．そして，CPU 用の構成要素関数と取得したカーネルハンドルおよび GPU

へ割り当てる画像の割合の 3 つを引数として空間的協調用高階メソッドを呼び出す（図 20，12 行目）．この高階メソッドは処理パターン毎に提供されており，構成要素関数の解析結果に応じてトランスレータが自動的に指定する．この例では，1 対 1 写像の際に利用される procCPUGPU() を空間的協調用高階メソッドとして指定する．なお，この空間的協調用関数の詳しい動作については 6.1 節で説明する．

その後，トランスレータは変換前の main 関数（図 19，10〜12 行目）を解析し，これに対応する main 関数を Gray.cpp に生成する（図 20，16〜21 行目）．トランスレータは提案言語プログラムの入力および出力変数から，画像の読み出しと書き込みのためのコードを生成する（図 20，17〜19 行目，21 行目）．そして構成要素関数の呼び出し（図 19，11 行目）から，入力変数 in と協調時に GPU へ割り当てる画像の割合を引数とした GrayCPUGPU() の呼び出しのためのコードを生成する（図 20，20 行目）．以上のようにして，トランスレータは提案言語で記述された画像処理プログラムを空間的協調を利用する RaVioli/CUDA プログラムへと変換する．

なお，CUDA では 3.2.1 項で述べたように特徴の異なるいくつかのメモリを使用することができる．そのため，処理内容に応じてこれらのメモリを使い分けることで，より効率の良いプログラムを生成することが可能である．ここで，多対 1 写像や多対多写像における画像データへのアクセスパターンに注目する．処理単位が近傍画素集合や部分画像の場合，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．このようなメモリアクセスに対しては，専用のキャッシュを保持しているテクスチャメモリを用いることで高速化が期待できる．そこで，多対 1 写像や多対多写像が用いられる場合は，自動的にテクスチャメモリに対してデータを配置するための処理をトランスレータに追加した．このように，トランスレータはメモリへのアクセスパターンも考慮することで GPU の性能をより引き出すことが可能である．

5.2　時間的協調を利用する動画像処理プログラムとその変換方法

時間的協調を利用する動画像処理プログラムの例として，提案言語を用いて記述したエッジ検出プログラムを**図 22** に示す．提案言語を用いて時間的協調を利用する場合，main 関数内で動画像処理のための構成要素関数を呼び出す際に，18 行目のように協調時にパイプライン化する際のステージの境界位置を “[” と “]” で囲んだ数値で指定する．この例では，動画像処理のための構成要素関数 DetectEdge() は Binarize と Edge の 2 つのステージから構成される．そのためパイプライン化する際の境界位置は

```
1  /* 構成要素関数(画像)*/
2  (image)img1 > Binarize > img1{
3      // 2 値化
4  }
5  (image)img1 > Edge > img1{
6      //エッジ抽出
7  }
8
9  /* 構成要素関数(動画像) */
10 (stream)strm1> DetectEdge > strm1{
11     (image)img1@strm1{
12         img1 > Binarize | Edge > img1;
13     }
14 }
15
16 /* main 関数 */
17 (stream)strm > main > strm{
18     strm > DetectEdge[1] > strm;
19     //strm > DetectEdge[AUTO] > strm;
20 }
```

図 22: エッジ検出プログラム（時間的協調利用）

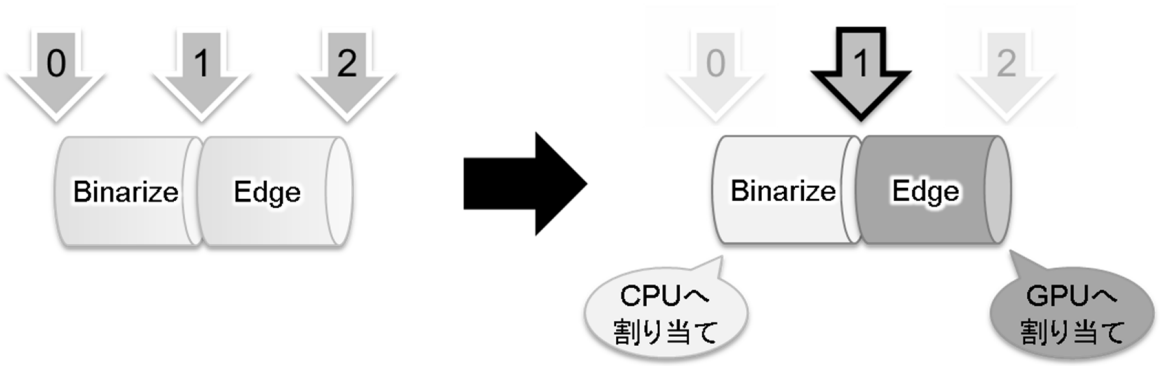

図 23: パイプライン化する際のステージ割り当て方法

図 23 に示すように 3 箇所存在する．このうち，この例では “1” が指定されているため，Binarize ステージを CPU へ，Edge ステージを GPU へ割り当て，動画像をパイプライン的に処理する．また提案する環境では，19 行目のように “AUTO” と指定することで実行時に自動的に適切なステージ割り当てを算出し，その割り当てでパイプラ

イン実行することもできる．この詳細については 6.2 節で説明する．

ここで図 22 と，このプログラムからトランスレータが生成したホスト・コードである**図 24** に沿って，時間的協調を利用する RaVioli/CUDA プログラムをトランスレータが生成する流れを説明する．まず，トランスレータは画像処理プログラムの場合と同様に画像処理のための構成要素関数を解析し，ホスト・コードに CPU 用の構成要素関数と空間的協調用関数を，デバイス・コードに GPU 用の構成要素関数を生成する．この例では，図 22 の 2〜7 行目を解析し，ホスト・コードである DetectEdge.cpp にそれぞれの構成要素関数に対応する CPU 用構成要素関数と空間的協調用関数を生成する（図 24，1〜15 行目）．そして，GPU 用の構成要素関数をデバイス・コードである kernels.cu に生成する．

次にトランスレータは動画像処理のための構成要素関数（図 22，10〜14 行目）を解析し，この関数内で画像処理のための構成要素関数の呼び出し（図 22，12 行目）を検出すると，各関数を 1 つのパイプラインステージとして扱うためのステージ用関数をホスト・コードに生成する（図 24，18〜25 行目）．この例では Binarize と Edge が呼び出されているため，BinarizeStage() と EdgeStage() が生成される．ここで，各ステージで入出力に使用されるデータは，画像データ用の配列のみではなく，中間データ用の配列やスカラ変数など様々な種類が想定される．そこで，ステージ間で異なる入出力データを使用する場合でも受け渡しインタフェースを統一化するために，汎用的なデータを格納するためのクラスである RV_Data のインスタンスをステージ用関数の入力および出力に使用する．なお，この RV_Data インスタンスは，RV_Image インスタンスへのポインタやデバイスメモリへのポインタなどをメンバとして保持している．そのため各ステージ内では，RV_Data インスタンスのメンバを参照することで，異なる入出力データを使用する場合でも必要なデータにアクセスできる．トランスレータは，この RV_Data インスタンスを関数の引数として受け取り（図 24，18 行目，22 行目），戻り値として RV_Data インスタンスのポインタを返すようにそれぞれのステージ用関数を生成する（20 行目，24 行目）．また関数内では，各ステージに対応する空間的協調用関数を呼び出すためのコードを生成する（19 行目，23 行目）．

その後，トランスレータは変換前の main 関数を解析し（図 22，17〜20 行目），動画像処理のための構成要素関数の呼び出しを検出すると，その関数がもつステージを管理するための RV_StageVector インスタンスを生成する．そして，この RV_StageVector インスタンスが持つメソッド push() へステージ用関数を順に渡していくことで，各ステージが RV_StageVector インスタンスに登録される．この例では RV_StageVector イ

```
1  /* 構成要素関数 (CPU) */
2  void BinarizeCPU(RV_Pixel* p1){
3      // 2 値化
4  }
5  void EdgeCPU(RV_Pixel* p1, RV_Pixel n_w, ...){
6      //エッジ抽出
7  }
8
9  /* 空間的協調用関数 */
10 void BinarizeCPUGPU(RV_image* img, int gpupart){
11     //空間的協調高階メソッド（1 対 1 写像)呼び出し
12 }
13 void EdgeCPUGPU(RV_image* img, int gpupart){
14     //空間的協調高階メソッド（多対 1 写像)呼び出し
15 }
16
17 /* ステージ用関数 */
18 RV_Data BinarizeStage(RV_Data* data, int gpupart){
19     binarizeCPUGPU(data->image,gpupart);
20     return data;
21 }
22 RV_Data EdgeStage(RV_Data* data, int gpupart){
23     EdgeCPUGPU(data->image,gpupart);
24     return data;
25 }
26
27 /* main 関数 */
28 void main(int argc, char* argv[]){
29     RV_Streaming strm;
30     RV_StageVector DetectEdge;
31     //パイプラインへステージを登録
32     DetectEdge.push(BinarizeStage);
33     DetectEdge.push(EdgeStage);
34     //時間的協調を利用する動画像処理の実行
35     strm.runPipeline(DetectEdge,1);
36 }
```

図 24: 生成されるホスト・コード（DetectEdge.cpp）

ンスタンスとしてDetectEdgeを生成し（図24，30行目），BinarizeStageとEdgeStageを引数としてpush()を実行する（図24，32〜33行目）．そして，RV_Streamingインスタンスの持つ時間的協調用高階メソッドrunPipeline()を呼び出すためのコードを生成する（35行目）．このrunPipeline()は，各ステージを登録したRV_StageVectorインスタンスとパイプライン化する際のステージの境界位置を示す値とを引数にとる．なお，このrunPipeline()の詳しい動作については6.2節で説明する．以上のようにして，トランスレータは提案言語で記述された画像処理プログラムを時間的協調を利用するRaVioli/CUDAプログラムへと変換する．

6 CPU/GPU協調用関数の実装

4.3節で述べた，空間的協調と時間的協調の2種類のCPU/GPU協調動作を実現するためにRaVioli/CUDAを拡張した．本章ではこれらを実現するために実装した高階メソッドについてそれぞれ説明する．

6.1 空間的協調用高階メソッド

4.3.1項で述べた空間的協調を実現するために，RaVioli/CUDAを拡張し，空間的協調用の高階メソッドを実装した．なお，この高階メソッドは処理パターンに応じて複数作成した．ここで，追加実装した高階メソッドのうち，1対1写像の際に用いるprocCPUGPU()の処理の流れを**図25**を用いて説明する．この関数ではまず，呼び出し時に与えられた引数に応じてCPUが入力画像を2つに領域分割する(i)．そしてCPUは，2つの部分画像のうち，GPUに割り当てる部分画像をGPUへ転送し，GPU用の構成要素関数を呼び出す(ii)．なお，これらの処理はCUDAが提供している非同期APIを用いて行われるため，GPU用の構成要素関数の実行と並行してCPUは処理を進めることができる．そこで，CPUはスレッドを生成し，自身が担当する部分画像を並列に処理する(iii)．なお，ここで生成するスレッドの数はプログラム実行時に引数を与えることでプログラマが指定できる．各スレッドの処理が完了すると，各スレッドの処理結果を統合し，生成したスレッドを破棄する(iv)．その後，GPUの処理結果を受け取り(v)，CPUとGPUそれぞれの処理結果を統合する(vi)．

では，以上の処理をどのように実現するのか，**図26**に示すprocCPUGPU()の擬似コードに沿って説明する． 5.1節でも述べたように，空間的協調用関数はCPU用の構成要素関数（CPUfunc），GPU用の構成要素関数（GPUfunc），そして空間的協調を利用する際にGPUへ割り当てる画像の割合（gpupart）を引数にとる（1〜3行目）．

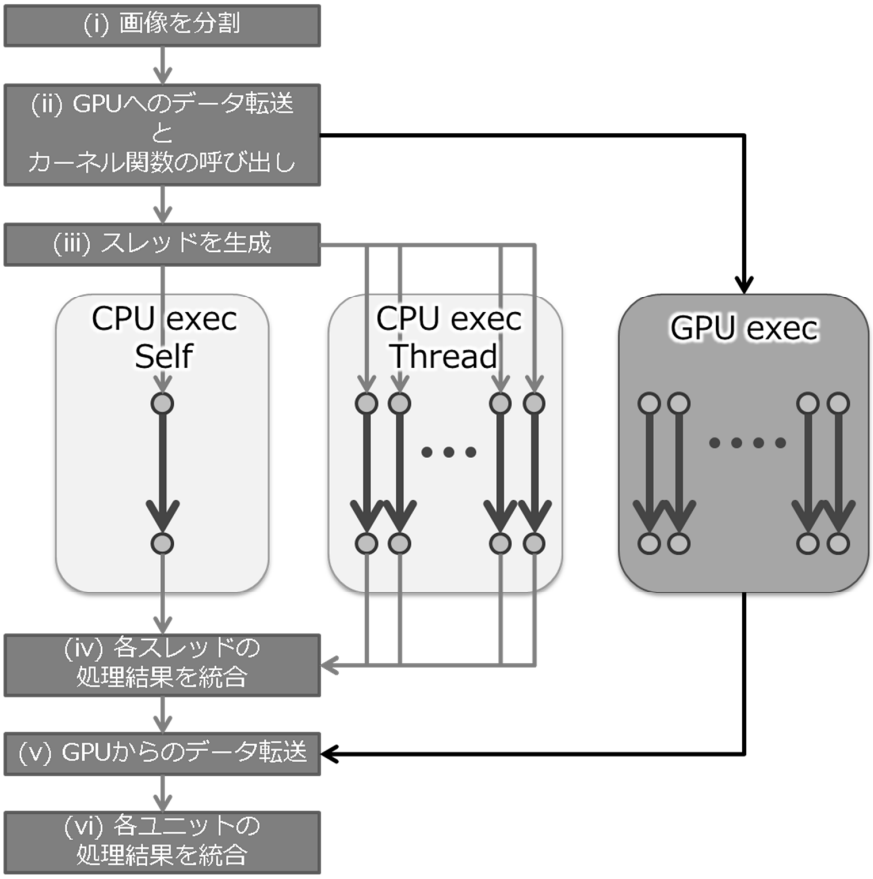

図 25: 空間的協調用高階メソッド（procCPUGPU）の処理の流れ

関数内ではまず，gpupart が 0 と等しいか，すなわち画像を分割し GPU に処理を割り当てるかどうかを調べ（4 行目），GPU に処理を割り当てる場合には，入力画像を gpupart の値に従って分割する（6 行目）．そして上述したように，CUDA が提供している非同期 API を利用して，GPU が処理を担当する部分画像を GPU へ転送し（8 行目），GPU 用の構成要素関数を非同期的に呼び出す（10 行目）．次に GPU 用の構成要素関数の実行と並行して CPU はスレッドを生成し，各スレッドはそれぞれが担当する範囲に対して CPU 用の構成要素関数を適用する（14 行目，16 行目）．各スレッドの処理が完了すると，各スレッドの処理結果を統合し，生成したスレッドを破棄する（18 行目）．その後，GPU の処理結果を CPU へ転送し（22 行目），CPU と GPU それぞれの処理結果を 1 つの画像にまとめる（24 行目）．

このように，空間的協調用高階メソッドでは，CPU と GPU が並行動作するため高速な処理を実現可能である．なお上述したように，この高階メソッドは 4.2.1 項で述べた各処理パターンに応じて複数実装したが，どのメソッドも概ね同様の流れで処理を

```
1  void RV_Image::procCPUGPU(void (*CPUfunc)(RV_Pixel*,int),
2                            CUfunction* GPUfunc,
3                            int gpupart){
4      if(gpupart!=0){//GPUへ処理を割り当てる場合
5          //入力画像を分割
6          devideImage();
7          //GPUへ割り当てるデータをデバイスへ転送
8          cudaMemcpyHtoDAsync();
9          //GPU用構成要素関数の呼び出し
10         cudaKernelCallAsync(GPUfunc);
11     }
12     if(gpupart!=100){//CPUへ処理を割り当てる場合
13         //スレッドを生成しCPUfuncを実行
14         createThread(CPUfunc);
15         //自身も画像を処理する
16         CPUfunc();
17         //各スレッドの処理結果統合とスレッド破棄
18         joinThread();
19     }
20     if(gpupart!=0){//GPUへ処理を割り当てる場合
21         //デバイスからホストへデータ転送
22         cudaMemcpyDtoHAsync();
23         //処理結果を統合
24         unifyImage();
25     }
26 }
```

図 26: 空間的協調用高階メソッド procCPUGPU() の擬似コード

実行する．

6.2 時間的協調用高階メソッド

また，時間的協調を実現するために RaVioli/CUDA を拡張し，時間的協調用の高階メソッド runPipeline() を実装した．このメソッド内では CUDA が提供するストリームを用いることで CPU と GPU を用いたパイプライン処理を実現している．ストリームとは GPU 上の処理を管理するキューであり，CUDA ではデータ転送や関数の呼び出し時に特定のストリームを指定し，この指定したストリームに対して処理を発行す

```
1 void main(int argc, char* argv[]){
2     RV_Streaming strm;
3     RV_StageVector DetectLine;
4     /*パイプラインへステージを登録*/
5     DetectLine.push(BinarizeStage);   //2 値化
6     DetectLine.push(EdgeStage);       //エッジ抽出
7     DetectLine.push(HoughStage);      //ハフ変換
8     DetectLine.push(ReHoughStage); //逆ハフ変換
9     /*時間的協調を利用する動画像処理の実行*/
10    strm.runPipeline(DetectLine, 2);
11    //strm.runPipeline(DetectLine, AUTO);
12 }
```

図 27: 直線検出プログラムの main 関数

ることができる．ここで，同一のストリームに発行された処理は，必ず発行された順に逐次的に実行される．これに対し，異なるストリームに発行された処理は並行に実行可能である．なお，CUDA5 からストリームの機能が拡張され，ストリームに対して CPU 処理を発行することが可能となった．時間的協調用関数ではこのストリームを利用し，各フレームに対する処理を，2 つのストリームに対して交互に発行する．そして，2 つストリームで CPU の処理と GPU の処理を並行実行させながら動画像処理を進めていく．これによりタスク並列性が存在する動画像処理の高速化を実現する．

ここで，提案言語で記述した直線検出を施す動画像処理プログラムからトランスレータが生成した RaVioli/CUDA プログラムの，main 関数の例を**図 27** に示す．なお，この直線検出プログラムは 2 値化，エッジ抽出，ハフ変換，逆ハフ変換の 4 つのステージで構成される．そして，このプログラム例の 10 行目のようにして runPipeline() が呼び出された際の動作の流れを，**図 28** 示す runPipeline() の擬似コードに沿って説明する．

5.2 節で述べたように runPipeline() は，動画像処理の各ステージをあらかじめ登録した RV_StageVector インスタンス（stageV）と，パイプライン化する際のステージの境界（pipepoint）を引数として呼び出される．runPipeline() の関数内ではまず，与えられた値（pipepoint）に応じて，stageV に登録されたパイプラインステージを CPUStage と GPUStage へ登録する（図 28，4 行目）．なお，CPUStage は CPU へ割り当てるステージを管理し，GPUStage は GPU へ割り当てるステージを管理する．この例では

```
1  void RV_Streaming::runPipeline(RV_StageVector* stageV, char pipepoint){
2      int loop;
3      /*パイプラインステージ割り当て*/
4      initPipeline(stageV, pipepoint);
5      /*stream 生成*/
6      CUstream stream[2];
7      for(int i=0; i<2; i++)
8          cudaStreamCreate(&stream[i]);
9      /*pipeline 実行開始*/
10     kernelCallAsync(stageV->CPUStage(frame[0]), stream[0]);
11     while(loop < FRAME_NUM-2){
12         kernelCallAsync(stageV->GPUStage(frame[loop], stream[0]));
13         kernelCallAsync(stageV->CPUStage(frame[++loop], stream[1]));
14         cudaStreamSync(stream[0]);//stream[0] の処理の完了を待つ
15         kernelCallAsync(stageV->GPUStage(frame[loop], stream[1]));
16         kernelCallAsync(stageV->CPUStage(frame[++loop], stream[0]));
17         cudaStreamSync(stream[1]);//stream[1] の処理の完了を待つ
18     }
19     kernelCallAsync(stageV->GPUStage(frame[loop], stream[0]));
20     if(FRAME_NUM%2 == 0){//総 FRAME 数が偶数の場合
21         kernelCallAsync(stageV->CPUStage(frame[++loop], stream[1]));
22         kernelCallAsync(stageV->GPUStage(frame[loop], stream[1]));
23     }
24     /*stream 破棄*/
25     for(int i=0; i<2; i++)
26         cudaStreamDestroy(stream[i]);
27 }
```

図 28: 時間的協調用高階メソッド runPipeline() の擬似コード

pipepoint に “2” が格納されているため（図 27，10 行目），BinarizeStage と EdgeStage が CPUStage に，HoughStage と ReHoughStage が GPUStage に登録される．その後，パイプライン処理のために用いる 2 つのストリームを生成し（図 28，6〜8 行目），各ストリームの処理をパイプライン的に実行する（図 28，10〜23 行目）．runPipeline() では奇数番目のフレームを stream[0] に，偶数番目のフレームを stream[1] に対して発行することで，CPU と GPU が時間的に協調して動作することを実現している．なお，この例では，BinarizeStage と EdgeStage が CPUStage へ，HoughStage と ReHoughStage

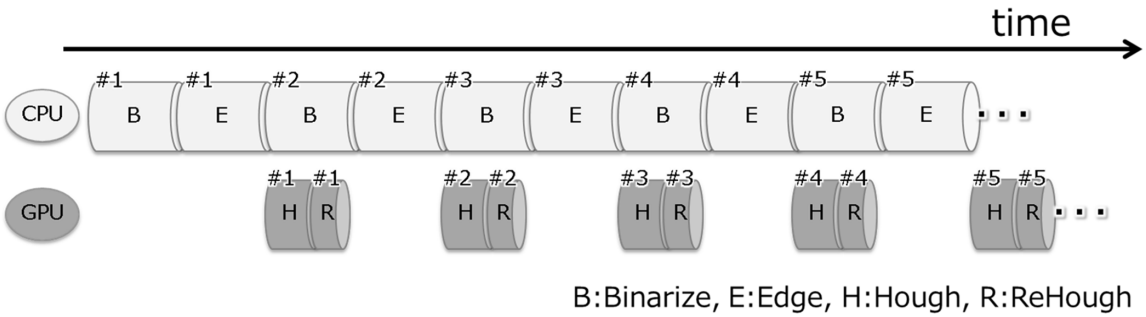

図 29: “2” が指定された場合のパイプライン実行の様子

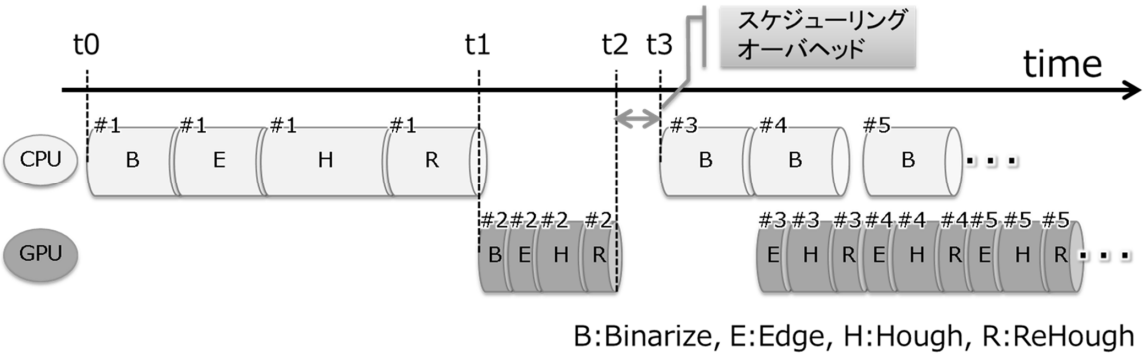

図 30: “AUTO” が指定された場合のパイプライン実行の様子

が GPUStage へ登録されているため，**図 29** のように動画像処理が実行される．この図では，ハフ変換と逆ハフ変換が GPU 上で実行されている間に，次のフレームに対する 2 値化が CPU 上で実行されている．このようにすることで時間的協調では，動画像処理の高速化が実現される．最後に，全てのフレームに対して処理が完了するとストリームを破棄する（図 28，25〜26 行目）．

また，提案手法では 5.2 節で述べたように，パイプライン化する際のステージの境界位置を指定する際に図 27 の 11 行目のように “AUTO” と指定することで，自動的に適切なステージの割り当てを算出し，設定することができる．ではここで，“AUTO” が指定されて runPipeline() が呼び出された場合の処理の様子について**図 30** を用いて説明する．“AUTO” を引数として呼び出された runPipeline() は，まず 1 フレーム目に対して，RV_StageVector インスタンスに登録された処理を全て CPU で実行し（t0〜t1），このときの各ステージの実行に要した時間を計測する．次に 2 フレーム目に対して，RV_StageVector インスタンスに登録された処理を全て GPU で実行し（t1〜t2），同様に各ステージの実行に要した時間を計測する．このようにすることで，各

ステージを CPU と GPU それぞれのユニットで実行した際にどの程度の時間を要するか知ることができる．これらの結果を用いて，もっとも高速化が見込める CPU/GPU のステージ割り当てを算出し，CPU と GPU へ各ステージを割り当てる（t2〜t3）．この例では BinarizeStage を CPU へ，EdgeStage，HoughStage，ReHoughStage を GPU へ割り当てている．その後，3 フレーム目以降に対してパイプライン実行を行う（t3 以降）．このようにすることで，プログラマは各ステージの処理量を意識すること無く，自動的に効率の良いパイプライン処理を実現できる．

7　評価

提案手法の有効性を確認するために，4〜6 章で述べた提案言語，RaVioli/CUDA の拡張およびトランスレータに対していくつかのサンプルプログラムを用いて評価を行った．本章ではまず，提案言語プログラムにおける記述性および可読性について述べる．次に，RaVioli/CUDA の拡張によって実現される 2 種類の CPU/GPU 協調動作手法について評価するため，トランスレータによって生成された RaVioli/CUDA プログラムの実行時間を比較し，その結果を考察する．

7.1　プログラムサイズの評価

まず，C++，RaVioli，提案言語で記述した画像処理および動画像処理プログラムのサイズを，行数とバイト数で比較した．評価対象プログラムには画像処理プログラムとして，グレースケール化，エンボスフィルタ，エッジ抽出，テンプレートマッチングを使用した．また動画像処理プログラムとして，直線検出のプログラムを使用した．評価結果を**表 3** に記す．

5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 プログラムと比べ，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 88%，バイト数ではエッジ抽出において最大で 89%の削減率となった．なお，この行数やバイト数の比較による評価はあくまで記述性を評価する 1 つの指標であり，記述性が高いことの証明にはならないが，この結果から少なくとも提案言語を用いる場合，プログラムは簡潔になることが確認できた．

7.2　記述性，可読性の評価

提案言語で記述したプログラムの可読性を，C++，RaVioli で記述したプログラムの可読性と比較した．C++，RaVioli，提案言語で記述した 2 値化プログラムをそれぞ

表 3: プログラムサイズの比較

プログラム		C++	RaVioli	提案言語
画像処理プログラム				
グレースケール化	行数	30	26	10
	バイト数	711	492	185
エンボスフィルタ	行数	153	36	19
	バイト数	3,562	831	419
エッジ抽出	行数	174	48	22
	バイト数	4,410	1,069	491
テンプレート	行数	106	60	18
マッチング	バイト数	3,736	1,180	423
動画像処理プログラム				
直線検出	行数	261	85	56
	バイト数	6,924	2,087	1,218

れ**図 31**，**図 32**，**図 33** に記す．

C++で記述したプログラムでは図 31 の 3，4 行目のように動画像処理の本質ではないループ文を記述する必要がある．それに対し，RaVioli および提案言語で記述したプログラムでは，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を記述することが可能となっている．次に，RaVioli と提案言語を比較すると，RaVioli ではプログラマは構成要素関数を定義した後， 4.2.1 項で述べた処理パターンに応じて適切な高階メソッドを選択する必要がある．図 32 では，2 値化が 1 対 1 写像の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，4.2.1 項で述べたように，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．実際に図 33 では 2 行目のように処理単位と処理範囲を記述するだけでよく，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．

7.3 実行時間の評価

次に，RaVioli/CUDA の拡張によって実現される 2 種類の CPU/GPU 協調動作手法について評価するため，トランスレータによって生成された，RaVioli/CUDA プログラムの実行時間を比較した．本節ではまず，評価に用いた環境について述べる．次に，

```
1  void Binary(Img* img){
2      int i, j, v, tmp;
3      for(i = 0; i < img -> height; i++){
4          for(j = 0; j < img -> width; j++){
5              v = getV(img -> D[i][j]);
6              if(v < 85) tmp = 255;
7              else tmp = 0;
8              img -> D[i][j].r = tmp;
9              img -> D[i][j].g = tmp;
10             img -> D[i][j].b = tmp;
11         }
12     }
13 }
14 Binary(&Image);
```

図 31: C++で記述した 2 値化プログラム

```
1  void Binary_Pix(RV_Pixel* p1){
2      if(getV(p1) < 85) p1->setRGB(0, 0, 0);
3      else p1->setRGB(255, 255, 255);
4  }
5  Image -> procPix(Binary_Pix);
```

図 32: RaVioli で記述した 2 値化プログラム

```
1  img1 > Binary > img1{
2      p1@img1{
3          if(p1.V < 85) p1 = #black;
4          else p1 = #white;
5      }
6  }
7  img_in > Binary > img_out;
```

図 33: 提案言語で記述した 2 値化プログラム

空間的協調を利用する画像処理プログラムの実行時間を評価し，その結果を考察する．その後，時間的協調を利用する動画像処理プログラムの実行時間を評価し，その結果を考察する．

表 4: 評価環境

OS	CentOS 6.4
CPU	Intel Core i7-4770
Clocks	3.3 GHz
Memory	16 GB
GPU	GTX TITAN
Generation	Kepler2
Core Clocks	837 MHz
Memory Clocks	6,008 MHz
SMs	14
CUDA version	5.5.0
Compiler	gcc 4.4.7
Compile options	-O3

7.3.1 評価環境

評価環境を**表 4** に示す．なお，本評価で用いる GPU として，NVIDIA 社が提供している GeForce のうち，Kepler2 アーキテクチャを採用している GTX TITAN を使用した．また，CPU は 8 コア構成である Core i7-4770 を使用し，CPU 上のマルチスレッド処理における並列数は 8 として評価した．

7.3.2 空間的協調の評価

まず，空間的協調を利用する画像処理プログラムの実行時間を評価した．評価には，グレースケール化，エッジ抽出，ハフ変換，テンプレートマッチングの 4 つのプログラムを使用した．入力画像には一般的なスマートフォンで撮影可能な解像度である XGA（1024×768）を使用した．なお，テンプレートマッチングの際に用いられるテンプレート画像には 120×120 の画像を使用した．

評価結果を**図 34** に示す．図中では，各プログラムの評価結果が 10 本のバーで表されている．これらのバーは左から順に，GPU へ割り当てる画像の割合を 0〜100%まで 10%刻みで変化させた場合の実行に要した時間をそれぞれ表している．つまり，一番左のバーは CPU のみ，一番右のバーは GPU のみで実行した場合の実行時間を表している．そして各実行時間は CPU のみで実行した場合を 1 として正規化している．なお，各プログラムにおいて最も実行時間が短いバーを黒色で表している．

評価結果から，入力画像が同じであるにも関わらず，プログラムによって最適な分割比が異なることが確認できた．これは，CPU と GPU は異なるアーキテクチャであ

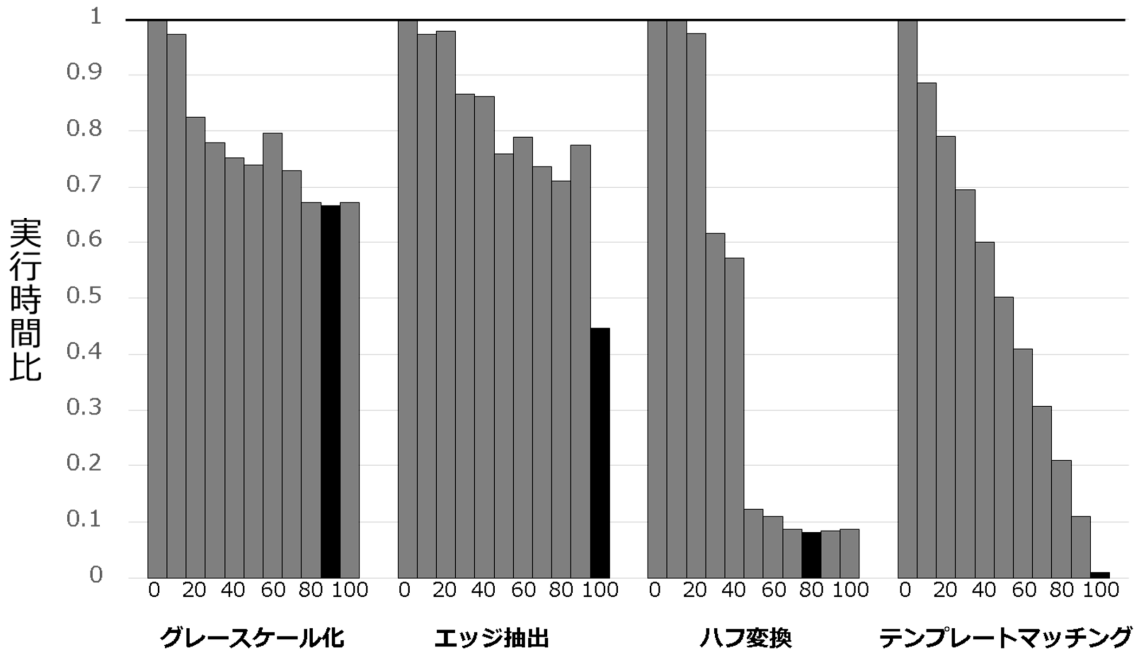

図 34: 空間的協調を用いた各画像処理プログラムの実行時間

り，そのアーキテクチャの特徴を活かせる処理がそれぞれ異なることと，各プログラムでは処理の特徴が異なっていることが原因であると考えられる．ここで，得られた評価結果に関して使用したプログラムごとにそれぞれ考察する．まず，グレースケール化では画像の 90%の領域を GPU へ，残りの 10%の領域を CPU へ割り当てた場合の実行時間が最も短くなった．このプログラムは 1 対 1 写像であるため並列性が高く，このようなプログラムは，一般的に CPU で処理するよりも GPU で処理する方が高速化を期待できる．しかし，グレースケール化は処理内容が非常に単純であるため，CPU と GPU で処理時間の差が大きくなく，むしろ CPU-GPU 間のデータ転送コストが相対的に大きくなってしまう．そのため，ある程度の画像を CPU に割り当てることで CPU-GPU 間のデータ転送時間を抑制することができ，性能が向上したと考えられる．次に，エッジ抽出とテンプレートマッチングは画像の全領域を GPU へ割り当てた場合の実行時間が最も短くなった．このうち，エッジ抽出は多対 1 写像であるためグレースケール化と同様に，並列性が高く GPU で処理させる方が有効である．加えて，提案手法では多対 1 写像のようなメモリ上の近いアドレスが参照される処理の場合に，アクセスが高速なテクスチャメモリに対してデータを配置する．そのため，CPU-GPU 間でデータアクセス速度の差が大きくなり，GPU のみで実行した際に最も高速に処理できたと考えられる．また，テンプレートマッチングは多対多写像であり，テンプレー

ト画像の各画素を繰り返し比較するというプログラムであるため，メモリアクセス回数がエッジ抽出よりもさらに多い．このためテクスチャメモリへのデータ配置の効果がより顕在化し，エッジ抽出よりも GPU で実行した際の性能向上幅が大きくなったと考えられる．最後に，ハフ変換は画像の 80%の領域を GPU へ，残りの 20%の領域を CPU へ割り当てた場合の実行時間が最も短くなった．このハフ変換は基本的に 1 対 1 写像であるが，その処理中には各画素の画素値に応じて異なる動作をするための条件分岐が含まれている．ここで，GPU は 3.2.1 項で述べたように SIMT 型の実行方式を採用している．この方式では，各スレッドが全分岐先を実行するため処理時間が増大してしまう **Branch Divergence** という現象が発生する [3]．そのため GPU は，CPU よりも条件分岐を持つプログラムの実行に時間がかかってしまう．このような理由から，グレースケール化よりも多くの領域を CPU に割り当てた際に高速化したと考えられる．

なお，本評価では XGA 解像度の画像を使用したが，より解像度が高い画像，すなわち画素データ量が大きい画像の場合では，GPU へのデータ転送がボトルネックとなり得る．そのような場合，CPU に対しても処理を割り当てる空間的協調はより効果を発揮すると考えられる．

以上で述べたように，空間的協調を用いる場合ではプログラムの処理内容によって最適な分割比が異なる．提案手法では，簡単な記述のみで分割比を調整できるためプログラマは容易に効率的な分割比を模索できる．ただ理想的にはプログラマが指定せずとも，自動的に適切な分割比が設定されることが望ましい．そのため今後は，様々な処理内容や入力画像サイズを用いて評価を行い，その特徴を考察することで自動的に分割比を設定することを可能にし，プログラマの負担をより軽減できる環境を実現していく予定である．

7.3.3 時間的協調の評価

次に，時間的協調を利用する動画像処理プログラムの実行時間を評価した．評価には，2 値化，エッジ抽出，ハフ変換，逆ハフ変換の 4 つのステージで構成される直線検出プログラムを使用した．入力には，解像度が XGA で，長さが 1 分間の動画像を使用した．なお，この動画像のフレームレートは 30fps であり，総フレーム数は 1,800 枚である．評価結果を**図 35** に示す．グラフは上から順に，既存の RaVioli で記述し CPU のみを用いて実行したプログラム，既存の RaVioli/CUDA で記述し GPU のみを用いて実行したプログラム，提案言語で記述し CPU と GPU を時間的に協調させたプログラムの実行時間をそれぞれ表している．なお，提案手法によってプログラマが簡単な記

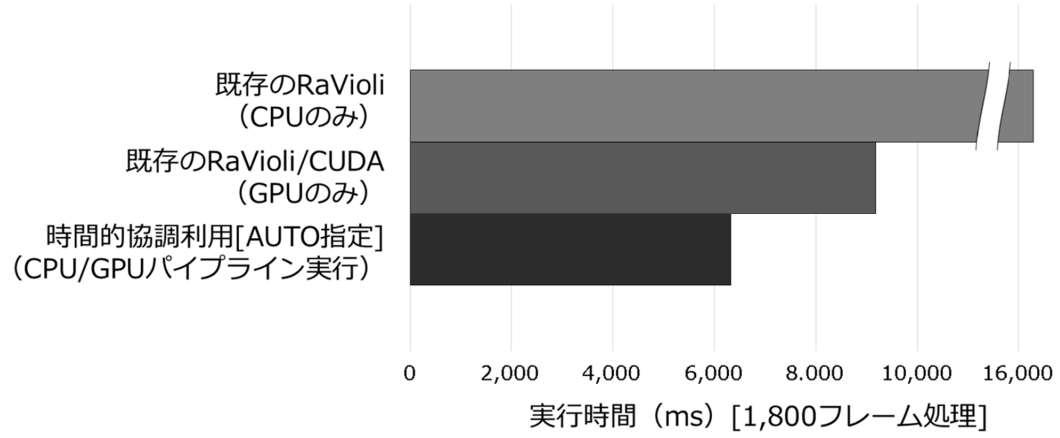

図 35: 動画像処理プログラムの実行時間

表 5: 既存の RaVioli，RaVioli/CUDA と提案手法の速度比較

プログラム	速度向上比
RaVioli（CPU のみ）	25.80 倍
RaVioli/CUDA（GPU のみ）	1.45 倍

述のみで効率的なプログラムを開発できることを確認するために，提案言語における時間的協調を用いるプログラムではパイプライン化する際の境界位置指定に “AUTO” を指定した.

評価結果から，提案言語で記述した時間的協調を用いたプログラムでは，既存の RaVioli，RaVioli/CUDA で記述したプログラムと比較して大きく高速化していることが確認できる．ここで，既存の RaVioli，RaVioli/CUDA で記述したプログラムの実行時間に対する，提案言語で記述したプログラムの実行時間の速度向上比は**表 5** のようになり，提案手法で生成されたプログラムでは RaVioli プログラムと比べて 25.80 倍，RaVioli/CUDA プログラムと比べて 1.45 倍の速度向上を達成した．以上の結果から時間的協調では，プログラマは動画像処理をパイプライン化する際の境界位置指定に “AUTO” と指定することで，動画像処理における各ステージの処理量を意識すること無く，CPU と GPU が協調動作する効率的なパイプライン実行を実現できることが確認できた.

8 おわりに

本論文ではまず，解像度をプログラマから隠蔽することで動画像処理プログラミングを抽象化する動画像処理ライブラリ RaVioli，およびその GPU 向け拡張である RaVioli/CUDA について述べた．そして，RaVioli および，RaVioli/CUDA の問題点として動画像処理を十分に抽象化できていないこと，さらに，CPU と GPU を搭載したヘテロジニアスな計算機環境を十分に活かしきれていないことを示した．

これらの問題を解決するために，RaVioli/CUDA よりも高抽象度な動画像処理記述言語と，CPU/GPU 間の協調動作を支援可能とするための RaVioli/CUDA の拡張を提案した．提案言語では，処理パターンに依存しない統一的な記述方式を採用することで，RaVioli/CUDA よりも抽象度の高いプログラミングを可能にした．一方 RaVioli/CUDA の拡張では，画像処理に存在するデータ並列性および，動画像処理に存在するタスク並列性に着目し，空間的協調と時間的協調の 2 種類の CPU/GPU 協調動作手法を実装した．これにより，CPU と GPU が遊休状態になる時間を短縮でき，ヘテロジニアスな計算機環境をより有効に活用することを可能にした．さらに，提案言語プログラムから，拡張した RaVioli/CUDA プログラムへと自動的に変換するトランスレータを実装した．これにより，簡単な記述のみでヘテロジニアスな計算機環境を活かしたプログラムを開発できるプログラミング環境を実現した．

最後に，提案手法の有効性を示すため，いくつかのサンプルプログラムを用いて提案言語，拡張した RaVioli/CUDA，およびトランスレータを評価した．C++，既存の RaVioli，提案言語のそれぞれで記述したプログラムのサイズを比較した結果，提案言語で記述されたプログラムは，C++や既存の RaVioli を用いたプログラムと比べて，行数，バイト数共に削減できることを確認した．また，空間的協調の有効性を確かめるために，空間的協調を利用した画像処理プログラムにおいて，CPU/GPU へ割り当てる画像の分割比を変化させて評価した．その結果，プログラムの特徴に応じて最適な分割比が異なることを確認し，空間的協調を利用することによって，CPU あるいは GPU のみで実行した場合よりも処理が高速化することを確認した．さらに，時間的協調の有効性を確かめるために，既存の RaVioli と既存の RaVioli/CUDA を用いた動画像処理プログラムと，時間的協調における CPU/GPU への自動ステージ割り当て機能を用いた動画像処理プログラムの実行時間を比較した．その結果，提案手法で記述した動画像処理プログラムは，既存の RaVioli で記述したプログラムと比較して 25.80 倍，既存の RaVioli/CUDA で記述したプログラムと比較して 1.45 倍の速度向上を達成

できることを確認した.

今後の課題として，空間的協調を利用する際のCPU/GPU間の処理量比の自動設定が挙げられる．空間的協調では目的とする画像処理の処理内容に応じて最適な分割比が異なることを確認できた．これはCPUとGPUは異なるアーキテクチャであり，そのアーキテクチャの特徴を活かせる処理がそれぞれ異なるためである．そこで処理パターンに応じて適切な分割比を自動的に設定する機能をプログラマに提供することで，よりプログラマの負担を減らすことが可能であると考えられる．また，今回は単一のGPUで評価したが，複数GPUを搭載した環境に対応させることで提案手法の汎用性をより向上させることも検討していく.

他にも，より多くの動画像処理プログラムを記述できるようにすることが今後の課題として挙げられる．そのため，まず現時点で記述できない複雑な動画像処理について調査し，その動画像処理を記述できない原因を分析する．そして，その分析に基づいて，対象の処理を提案言語で直感的に記述できるよう言語を拡張していきたい.

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津邑公暁准教授，松尾啓志教授，齋藤彰一准教授，松井俊浩准教授，梶岡慎輔助教，川島龍太助教に深く感謝致します．また，本研究の際に多くの助言，協力をして頂いた津邑研究室，松尾研究室，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大平真司氏，澤田晃平氏，内山寛章氏，井手上慶氏，柴田裕貴氏，橋本高志良氏，竹嶌良氏，および水野雅大氏に感謝致します.

著者発表論文

論文

1. Masarhiro Mizuno, Takuya MATSUNAGA, Tomoaki TSUMURA, Hiroshi MATSUO: “Auto-Parallelization for a Video Processing Library with Content-Aware Resolution” , Proc. 2nd Int’l Symp. on Computing and Networking (CANDAR’14) , Shizuoka, Japan, pp.185-191 (Dec. 2014)
2. Takuya MATSUNAGA, Shinji OHIRA, Tomoaki TSUMURA, Hiroshi MATSUO: “Content-Aware Precision Control on a Real-Time Video Processing Library”, Proc. 2013 High Performance Computing & Simulation Conf. (HPCS2013), Helsinki, Finland, pp.453-460 (Jul. 2013)

報文

1. 水野 雅大, 松永 拓也, 津邑 公暁, 松尾 啓志: “領域別解像度調整及び自動並列化機能を備える動画像処理ライブラリの実現”, 信学研報 (SWoPP2014), Vol.114, No.155, pp.197-202 (Jul. 2014)
2. 松永 拓也, 大平 真司, 津邑 公暁, 松尾 啓志: “リアルタイム動画像処理ライブラリにおける入力の重要度を考慮した処理量調整手法”, 信学研報 (SWoPP2013), Vol.113, No.169, pp.67-72 (Jul. 2013)

参考文献

[1] 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).

[2] 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).

[3] NVIDIA Corp.: *NVIDIA CUDA Programming Guide*, 6.5 edition (2014).

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

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

[6] Kovács, G., Iván, J. I., Árpád Pányik and Fazekas, A.: The openIP Open Source Image Processing Library, *Proc. Int'l Conf. on Multimedia (MM'10)*, ACM, pp. 1489–1492 (2010).

[7] Regis Clouard: Pandore: A library of image processing operators (Version 6.4). [Software]. Greyc Laboratory, http://www.greyc.ensicaen.fr/˜regis/Pandore (2011).

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

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

[10] Reinders, J.: *Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism*, O'Reilly (2007).

[11] Dagum, L. and Menon, R.: OpenMP: an Industry Standard API for Shared-Memory Programming, *IEEE Computational Science and Engineering*, Vol. 5

(1998).

[12] Wang, S., Dong, Z., Chen, J. X. and Ledley, R. S.: PPL: A whole-image processing language, *Comput. Lang. Syst. Struct.*, Vol. 34, pp. 18–24 (2008).

[13] 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).

[14] 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).

[15] Ragan-Kelley, J., Adams, A., Paris, S., Leboy, M., Amarasinghe, S. and Durand, F.: Decoupling Algorithms from Schedules for Easy Optimization of Image Processing Pipelines, *ACM Transactions on Graphics (TOG) - SIGGRAPH 2012 Conference Proceedings*, ACM (2012).

[16] Ragan-Kelley, J., Barnes, C., Adams, A., Paris, S., Durand, F. and Amarasinghe., S. P.: Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines, *PLDI '13 Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, ACM, pp. 519–530 (2013).

[17] Sony Computer Entertainment: *Cell Broadband Engine Architecture*, 1.01 edition (2006).

[18] NVIDIA Corp.: GeForce GTX 460M, http://www.geforce.com/hardware/notebook-gpus/geforce-gtx-460m.

[19] NVIDIA Corp.: GeForce GTX 420M, http://www.geforce.com/hardware/notebook-gpus/geforce-gtx-420m.

[20] Ono, A., Kondo, K., Inaba, T., Tsumura, T. and Matsuo, H.: A GPU-supported High-Level Programming Language for Image Processing, *Proc. 7th Int'l Conf. on Signal-Image Technology and Internet-Based Systems (SITIS2011)*, pp. 245–252 (2011).

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

[22] 美濃導彦: 並列画像処理, 株式会社 コロナ社 (1999).

[23] The GNU Project: Bison - GNU parser generator, http://www.gnu.org/software/bison/.

[24] The Flex Project: Flex, the fast lexical analyzer, http://flex.sourceforge.net/.