修士論文

解像度非依存型動画像処理ライブラリ RaVioliのCell/B.E.向け最適化

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 22 年度入学　22413512 番

稲葉 崇文

平成 24 年 2 月 3 日

解像度非依存型動画像処理ライブラリ RaVioli の Cell/B.E. 向け最適化

稲葉 崇文

内容梗概

侵入者検知システムや自動車の衝突回避システムなどリアルタイム性の重要なシステムが盛んに開発されている．また，汎用計算機の高性能化と価格低下により，高性能な計算機を容易に手に入れることが可能になってきた．そのため今後，汎用 PC および汎用 OS 上でリアルタイム動画像処理を行うことが多くなると予想される．しかし，汎用 OS 上で，1/30 または 1/60 秒毎に 1 フレームを処理しなければならないリアルタイム動画像処理の実現は困難である．その主な理由として，1 フレームの処理量の変動や，他のプロセスが並行実行されていることによる，利用可能な CPU リソース量の変動があげられる．

そこで，汎用システム上で擬似的なリアルタイム性を保証する動画像処理ライブラリ RaVioli が提案されている．RaVioli では CPU 使用率や並行実行プロセスによる負荷に応じて，処理対象画像の解像度を自動的に変動させる．これによって処理量を調節し，擬似的なリアルタイム動画像処理を実現している．このように動的に解像度を変動させる場合，プログラマは，1 フレームあたりの画素数やフレームレートの変動に対応したプログラムを記述しなければならない．そこで，RaVioli ではプログラマから 1 フレームあたりの幅および高さ，すなわち画素数を隠蔽し，解像度をライブラリ内で制御している． このようにすることで，人間の映像認識過程に存在しない画素およびフレームといった概念を排除することも可能となり，より直感的な動画像処理プログラミングが実現できる．

しかし，RaVioli は画素情報を隠蔽することによって動的な解像度変動と抽象的な記述を可能としている一方で，抽象化のオーバ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．リアルタイム動画像処理では処理速度が非常に重要であるため，現在の RaVioli の実用化には大きな課題が残っていると言える．

一方，今日ではマルチコア環境が広く用いられている．また，マルチコア環境の普及によって，汎用 CPU と同時に GPU を搭載し，ヘテロジニアスな環境を構築している場合や，Cell/B.E. のように，CPU 自体がヘテロジニアスマルチコアである場合など，様々な構成のアーキテクチャが登場してきている．画像処理は並列化が比較的容易であるため，マルチコア環境とは相性が良いと考えられるが，処理の並列化を考慮

したプログラミングは煩雑であるという問題がある．そのため，こうした環境におけるプログラミング支援技術が望まれている．

そこで，本研究では，RaVioli を拡張し，マルチコア環境の中でも特に特徴的なアーキテクチャである Cell/B.E. に対応させることによって，RaVioli とマルチコア環境の抱える問題を同時に解決する手法を提案するとともに，マルチコア環境でのプログラミング支援の意義を探る．

拡張後の RaVioli を利用して記述されたサンプルプログラムを用いて評価を行い，従来の RaVioli を用いて記述されたプログラムを PPE のみを用いて実行した場合と，提案手法によって拡張した RaVioli を用いて記述されたプログラムを SPE を 6 基使用して実行した場合の処理時間を比較した．その結果，PPE のみを用いて実行した場合に対し，SPE を 6 基使用して実行した場合は，最大で約 16 倍の速度向上が確認できた．

解像度非依存型動画像処理ライブラリ RaVioli の Cell/B.E. 向け最適化

目次

1 はじめに **1**

2 研究背景 **2**

2.1 RaVioli 2

2.1.1 RaVioli を用いた画像処理プログラミングモデル 2

2.1.2 RaVioli の実行モデル 3

2.1.3 リアルタイム性の保証 5

2.1.4 RaVioli の問題点 7

2.2 マルチコア環境 8

2.2.1 Cell/B.E. 8

2.2.2 マルチコア環境の問題点 10

2.2.3 関連研究 10

3 提案ライブラリ **12**

3.1 基本設計 13

3.2 仕様 15

3.2.1 RaVioli の拡張 15

3.2.2 ユーザインタフェース 16

4 ライブラリ実装 **20**

4.1 RV_Pixel クラスの拡張 21

4.2 PPE 向け高階メソッド 23

4.3 SPE 向け高階メソッド 27

5 トランスレータ **37**

5.1 トランスレータの概要 38

5.2 PPE プログラムの生成 40

5.3 SPE プログラムの生成 41

6 評価 **44**

7 おわりに **48**

謝辞 49

参考文献 49

1　はじめに

侵入者検知システムや自動車の衝突回避システムなどリアルタイム性の重要なシステムが盛んに開発されている．また，汎用計算機の高性能化と価格低下により，高性能な計算機を容易に手に入れることが可能になってきた．そのため今後，汎用 PC および汎用 OS 上でリアルタイム動画像処理を行うことが多くなると予想される．しかし，汎用 OS 上で，1/30 または 1/60 秒毎に 1 フレームを処理しなければならないリアルタイム動画像処理の実現は困難である．その主な理由として，1 フレームの処理量の変動や，他のプロセスが並行実行されていることによる，利用可能な CPU リソース量の変動があげられる．

そこで，汎用システム上で擬似的なリアルタイム性を保証する動画像処理ライブラリ **RaVioli (Resolution-Adaptable Video and Image Operating Library)** が提案されている．RaVioli では CPU 使用率や並行実行プロセスによる負荷に応じて，処理対象画像の解像度を自動的に変動させる．これによって処理量を調節し，擬似的なリアルタイム動画像処理を実現している．このように動的に解像度を変動させる場合，プログラマは，1 フレームあたりの画素数やフレームレートの変動に対応したプログラムを記述しなければならない．そこで，RaVioli ではプログラマから 1 フレームあたりの幅および高さ，すなわち画素数を隠蔽し，解像度をライブラリ内で制御している． このようにすることで，人間の映像認識過程に存在しない画素およびフレームといった概念を排除することも可能となり，より直感的な動画像処理プログラミングが実現できる．

しかし，RaVioli は画素情報を隠蔽することによって動的な解像度変動と抽象的な記述を可能としている一方で，抽象化のオーバ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．リアルタイム動画像処理では処理速度が非常に重要であるため，現在の RaVioli の実用化には大きな課題が残っていると言える．

一方，プロセッサのスカラ処理性能の向上が困難になりつつあることから，今日ではマルチコア環境が広く用いられている．また，マルチコア環境の普及によって，様々な構成のアーキテクチャが登場してきている．画像処理は並列化が比較的容易であるため，マルチコア環境とは相性が良いと考えられるが，処理の並列化を考慮したプログラミングは煩雑であるという問題がある．さらに，汎用 CPU と同時に GPU を搭載し，ヘテロジニアスな環境を構築している場合や，**Cell/B.E.（Cell Broadband Engine）**のように，CPU 自体がヘテロジニアスマルチコアである場合など，マルチ

コア環境におけるプログラミングはますます煩雑になりつつある．そのため，こうした環境におけるプログラミング支援技術が望まれている．

そこで，本研究では，RaVioli を拡張し，マルチコア環境の中でも特に特徴的なアーキテクチャである Cell/B.E. に対応させることによって，RaVioli とマルチコア環境の抱える問題を同時に解決する手法を提案するとともに，マルチコア環境でのプログラミング支援の意義を探る．

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，そしてマルチコア環境について述べる．3 章で RaVioli と Cell/B.E. との連携手法について提案し，4 章でその実装について述べ，5 章でトランスレータによって従来の RaVioli プログラムを変換する手法について述べる．6 章で提案手法の評価結果と，それに対する考察を述べ，最後に 7 章で本論文全体をまとめる．

2　研究背景

本章では，研究の対象である RaVioli の概要と，マルチコア環境の現状について述べる．

2.1　RaVioli

本節ではまず，RaVioli の特徴である動画像処理の抽象化と処理量の自動調節について，また，RaVioli が抱える問題点について述べる．

2.1.1　RaVioli を用いた画像処理プログラミングモデル

たとえば，顔検出を行うプログラムでは，背景画像とキャプチャした画像との差分をとり，エッジを抽出し，その結果に対しハフ変換を適用する．このとき背景画像とキャプチャした画像に差がない場合とある場合とで，ハフ変換の処理量が変動する．また，汎用 OS 上では複数のプロセスが並行実行されている．それらのプロセスによって利用可能な CPU リソース量が変動するため，リアルタイム動画像処理に必要な CPU リソース量を常に確保可能であることが保証できない．

そこで，擬似的なリアルタイム性を保証する動画像処理ライブラリ RaVioli[1] [2] が提案されている．RaVioli では CPU リソースの変動によりリアルタイム処理が困難になった場合，解像度を自動調節し，処理量を減らすことによってリアルタイム性を保証している．解像度を動的に変動させる場合，1 フレームあたりの画素数やフレームレートの変動に対応したプログラムの記述が必要になってくる．そこで RaVioli ではプログラマから，1 フレームの高さと幅やフレームレートを隠蔽し，解像度をライブ

ラリ内で制御している．そうすることで，プログラマは1フレームあたりの画素数やフレームレートを意識した記述を省略できる．

RaVioliでは，画像の幅や高さ，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を記述した関数のみを定義し，それを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適用することが可能である．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，カラー画像をグレースケール画像へと変換する処理は**図1**のように記述される．まず，ユーザは対象画素をグレースケール化する構成要素関数GrayScaleを定義する．ここで，GrayScaleで使用されているgetR，getG，getBは，RV_PixelインスタンスPixの持つ色情報を取得するメソッドであり，setRGBは色情報を格納するメソッドである．この構成要素関数を，すべての構成画素に処理を適用する高階メソッドであるprocPixに渡す．こうすることで，procPixはすべての構成画素にGrayScaleの処理を施す．

このように，プログラマから画像の幅や高さを隠蔽することで，本来人間の動物体認識過程に存在しない画素やフレームといった概念を意識させない，直感的なプログラミングを可能にする．また，これによって，プログラマに解像度の変動を意識させずに処理量を変動させることが可能となる．

2.1.2 RaVioliの実行モデル

前項で述べたように，RaVioliはプログラマが記述した構成要素関数を高階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適用することが可能な環境を提供している．RaVioliを用いた場合とそうでない場合のソースコードの違いを**図2**に示す．図2はグレースケール化プログラムの例であり，図中の左側に示すプログラムは，RaVioliを用いていないプログラム，右側に示すプログラムは，前項で述べたRaVioliを用いたプログラムである．RaVioliを用いるプログラマは，まず，画像の幅や高さといった情報と，画素情報を格納した配列を保持するクラスであるRV_Imageクラスをインスタンス化する．そして，画素，画素集合，部分画像など，各処理単位に対して適用したい処理を構成要素関数として定義し，RV_Imageクラスの提供している高階メソッドに渡す．高階メソッドでは，自身が保持する画素情報を格納した配列の各要素に対して，プログラマの定義した構成要素関数を適用していく．

従来の画像処理プログラムでは，各画素へ処理を適用するためにループを用いて記

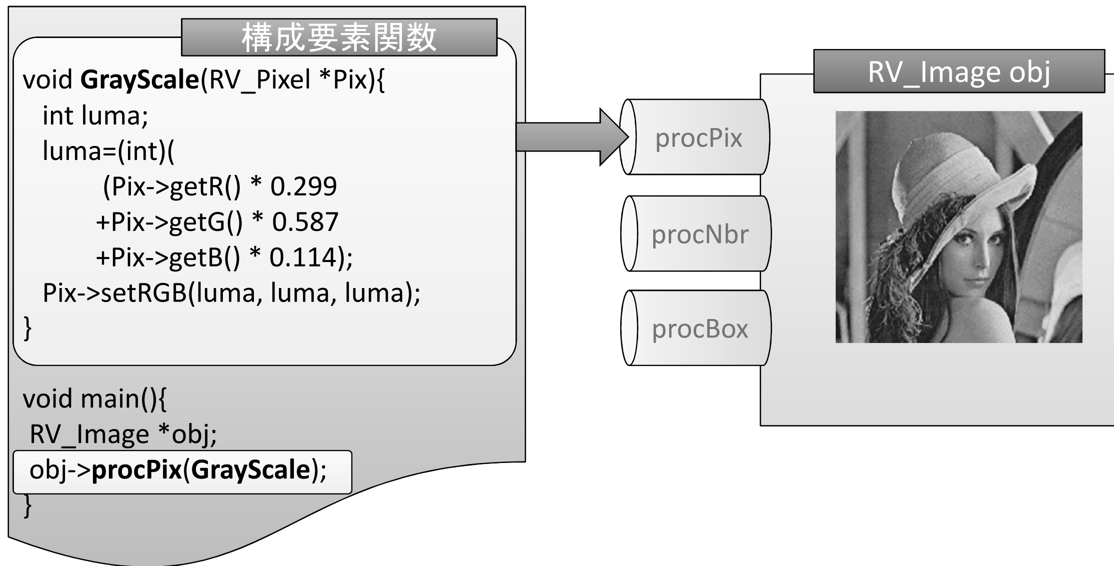

図 1: 高階メソッド呼び出し

述しなければならなかった．ループを用いた処理では，イタレータの増減によって処理を進めていくことになる．図 2 の例では，変数 x と y がそれぞれイタレータとなっており，この値を変化させていくことで、各画素へ処理を適用している．この例では，画素 `InImg[x][y]` に対して処理を適用した後，画素 `InImg[x + 1][y]` に対して処理を適用することになる．このような記述は，たとえ各画素に対する処理に処理順依存が存在しない場合であっても，イタレータの増減という処理を記述することによって，処理順序が与えられてしまう．しかし，今回の例で示しているプログラムはグレースケール化の処理であり，本来は各画素の処理順に依存するようなことはない．このように，本来並列性のある処理であっても，プログラムを記述するというプロセスを経ることによって，全順序化されてしまい，場合によっては，処理順依存の有無を検出できなくなってしまう．

一方で，RaVioli を用いたプログラムでは，各画素への処理はライブラリ側での適用される．従来の画像処理プログラムでは，ループをプログラマ自身が記述していたため，本来処理順依存が存在しないようなプログラムを記述する場合でも，処理順序を与えるように記述せざるを得なかった．しかし，RaVioli を用いた場合では，各画素への処理をライブラリ側で適用するため，プログラムを記述するプロセスにおいて処理順序を与える過程は存在しない．そのため，従来の画像処理プログラムに比べ，データ並列処理の処理単位が明確になっている．

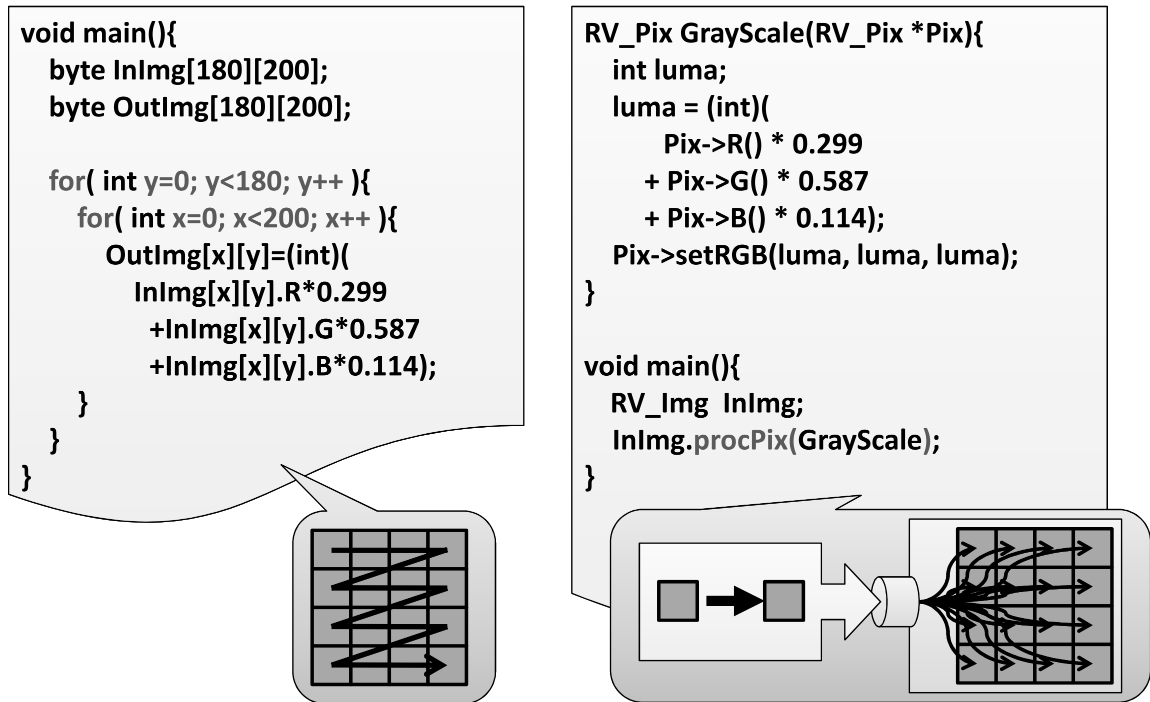

図 2: RaVioli の有無による実行モデルの比較

2.1.3　リアルタイム性の保証

一般的に，汎用 PC および汎用 OS では，1 フレームあたりの処理量の変動や，並列実行されている他のプロセスによる CPU リソースの減少などによって，リアルタイムに動画像を処理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処理量を減らす方法が考えられる．動画像における解像度には**空間解像度**および**時間解像度**の 2 種類がある．空間解像度とは 1 フレームを構成する画素数であり，時間解像度とはフレームレートである．RaVioli は各解像度を制御する解像度ストライドを持ち，CPU リソース量に応じてこれを変更することで処理量の調整を実現している．以下では，空間解像度と時間解像度のそれぞれが変動した場合について説明する．

空間解像度の変動

空間解像度を低減させる概念モデルを**図 3** に示す．空間解像度とは 1 フレームを構成する画素数のことである．RaVioli で空間解像度を変更する場合，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**を大きくするかまたは小さくすることで実現する．たとえば，空間解像度を低減させる場合には，空間解像度ストライドを大きくし，処理対象画素の間隔を大きくする．図 3 に示すように，空間解像度ストライ

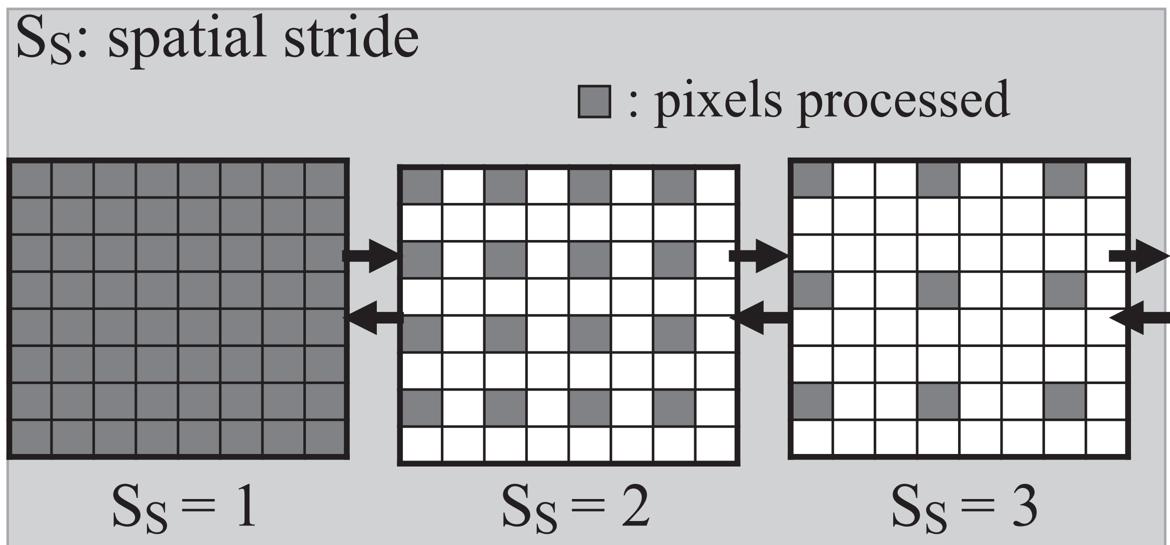

図 3: 空間解像度ストライドの変動

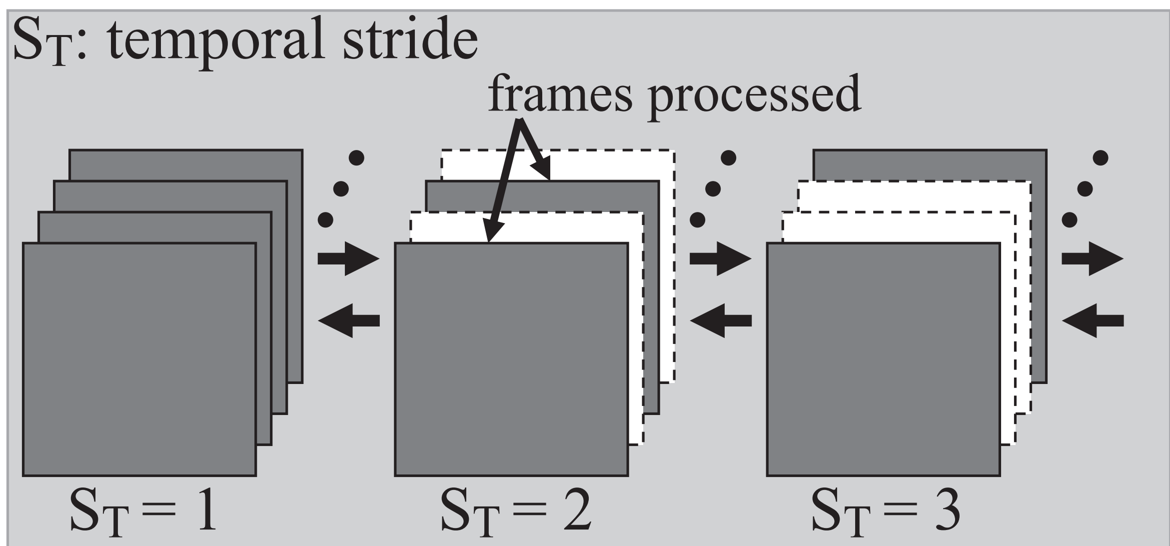

図 4: 時間解像度ストライドの変動

ド $S_S = 1$ のとき，画像中の全ての画素が処理される．処理を継続していく中で，画像中の全ての画素に対する処理が一定時間内に終わらないと RaVioli が判断すると，空間解像度を低下させ，空間解像度ストライドを $S_S = 2$ に増加させる．空間解像度ストライドが $S_S = 2$ に増加したことよって，処理対象画素は 1 つおきとなり，全体の処理画素数は $S_S = 1$ のときの 1/4 となる．同様に，さらに空間解像度が低下し $S_S = 3$ に増加した場合，処理画素数は 1/9 となる．

時間解像度の変動

時間解像度を低減させる概念モデルを**図 4** に示す．時間解像度とはフレームレートのことである．RaVioli で時間解像度を変更する場合，処理するフレームの間隔を示す**時間解像度ストライド**を大きくするかまたは小さくすることで実現する．たとえば，時間解像度を低減させる場合には，処理するフレームの間隔を大きくする．図 4 に示すように，時間解像度ストライド $\mathrm{S_T} = 1$ のとき，全てのフレームが処理される．空間解像度の変動の例と同様に，動画像中の全てのフレームに対する処理が一定時間内に終わらないと RaVioli が判断すると，時間解像度を低下させ，時間解像度ストライドを $\mathrm{S_T} = 2$ に増加させる．時間解像度ストライドが $\mathrm{S_T} = 2$ に増加したことによって，フレームを 1 つ飛ばしで処理することになり，フレームレートは $\mathrm{S_T} = 1$ のときに対して 1/2 になる．同様に，さらに時間解像度が低下し $\mathrm{S_T} = 3$ に増加した場合，フレームレートは 1/3 となる．

なお，RaVioli ではユーザが指定した優先度に応じて空間解像度および時間解像度を自動的に変動させることが可能である．たとえば，侵入者検知システムのように，高いフレームレートを維持し，厳密にリアルタイム性を保証したい場合，空間解像度を低減させ，時間解像度を維持したままリアルタイム処理をする．また，顔認証などのように厳密なリアルタイム処理が必要ではなく，より鮮明な画像が必要な処理の場合には，時間解像度を低減させ，空間解像度を維持する．このようにユーザは処理内容に応じて優先度を設定することで目的の解像度を維持したリアルタイム処理が可能である．

2.1.4 RaVioli の問題点

前項で述べたように解像度を動的に変動させる場合，1 フレームあたりの画素数やフレームレートの変動に対応したプログラムの記述が必要になる．そこで，RaVioli では，2.1.1 項で述べたようなインタフェースを提供し，解像度をライブラリ内部で制御することによって，プログラマから解像度を隠蔽している．

しかし，この手法を用いた場合，より直感的なプログラミングが可能になる反面，解像度隠蔽のためのオーバヘッドによって処理速度が低下してしまう．一方で，画像処理は比較的並列化が容易であるため，マルチコア環境で並列実行することによる高速化が考えられる．次節以降では，マルチコア環境の現状について述べる．

2.2　マルチコア環境

プロセッサのマルチコア化に伴い，動画像処理を始めとした，アプリケーションの並列化による高速化手法が盛んに研究されている．これは，ゲート遅延が支配的であった 2000 年代初頭までは，配線プロセスの微細化による高周波数化により，プロセッサの高速化を実現できたが，数年前からはゲート遅延に対する配線遅延の相対的な増大や，集積回路の微細化に伴う消費電力および発熱量の増大が問題となり，プロセッサの動作周波数の向上は困難になってきているからである．そのため，今日のプロセッサでは，マルチコア化を進め並列処理性能を向上させる事で，消費電力や発熱量の問題を解決しつつ，プロセッサ全体としての処理性能の向上を図っている．

マルチコアプロセッサの普及に伴い，さらなる高性能化やエネルギー効率の向上を目的として，様々な構造を持つプロセッサが開発されるようになった．また，GPU に画像処理以外の演算を実行させる GPGPU 技術や，Cell/B.E.[3] のように，複数種類のコアを搭載する，ヘテロジニアスマルチコアも登場し，単純なマルチコア環境ではなく，複雑な構成を持つ環境が広まりつつある．Cell/B.E. は，こうしたマルチコア環境の中でも特に複雑な構成を持ち，アプリケーション開発には高度なプログラミング技術が要求される．そのため，こうした環境でも容易にアプリケーション開発が可能な，プログラミング支援環境が望まれている．特に，Cell/B.E. のように複雑な構成を持つ環境向けのプログラミング支援環境を提供することは，それよりも単純な構成を持つマルチコア環境でも同等以上の支援が可能であることを示すことに繋がり，大きな意義がある．本節では，特に複雑な構成を持つマルチコア CPU である Cell/B.E. の現状と，それを解決するための関連研究について述べる．

2.2.1　Cell/B.E.

Cell/B.E. は，SONY，東芝，IBM の 3 社により共同開発されたプロセッサである．Cell/B.E. は，1 基の汎用プロセッサ PPE（PowerPC Processor Element）と 8 基の演算プロセッサ SPE（Synergistic Processor Element）を 1 チップ上に集約したヘテロジニアスマルチコアプロセッサである．Cell/B.E. は，シングルスレッド時の性能よりもむしろ，マルチスレッド時の性能を目指したプロセッサであり，9 基のコアをあわせた浮動小数点演算能力はピーク時で 200GFLOPS を超える．Cell/B.E. アーキテクチャの概略を**図 5** に示す．各プロセッサコアは、EIB（Element Interconnect Bus）と呼ばれる高速なバスで接続されている．EIB の転送速度は 204.8GB/秒である．また，EIB はメインメモリや外部入出力デバイスとも接続されている．

PPE は汎用プロセッサであり，そのアーキテクチャ構造は従来のプロセッサと大き

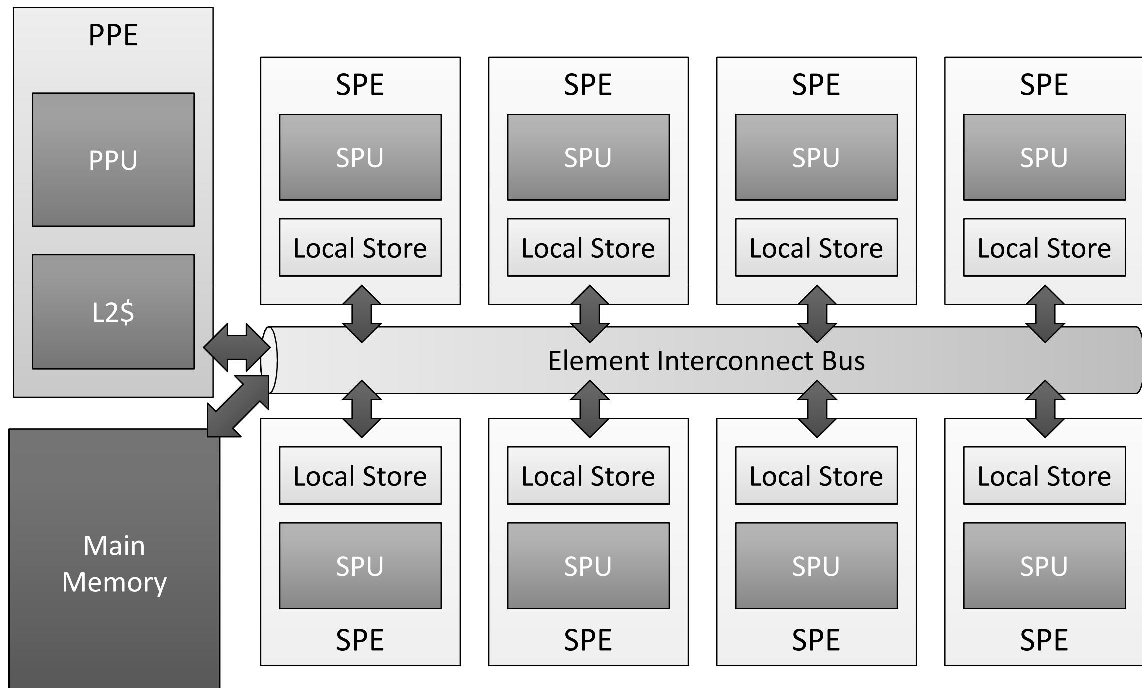

図 5: Cell/B.E. アーキテクチャ

な差異は無い．PPE は，SPE に比べスレッドの切り替えが高速であることから，OS の駆動などの処理を受け持つ．PPE のみを用いてプログラムを実行することも可能であるが，演算性能が低いため，通常は SPE へのリソースマネージメントを行う．

SPE は演算用のプロセッサであり，高い演算性能を持つ．そのため，Cell/B.E. では SPE を用いて処理を並列に実行することで，その性能を発揮することができる．SPE は，それぞれ 256KB のローカルストア（以下 LS）と呼ばれるスクラッチパッドメモリと，SPU（Synergistic Processor Unit），MFC（Memory Flow Controller）と呼ばれるユニットを持つ．SPU とは，SPE の演算処理を行う核となるユニットであり，各 SPU は直接メインメモリや他の SPE 上の LS にアクセスすることはできない．そのため，処理に必要なデータはメインメモリから LS へ転送する必要がある．データの転送は MFC によって制御される．この LS とメインメモリ間でのデータ転送に要する時間は非常に長いため，メモリレイテンシを隠蔽する方法として，ダブルバッファリングという手法がよく使用される．これは，LS 上にバッファを 2 つ用意しておき，片方のバッファに対してデータを転送している裏で，もう片方のバッファのデータに対して計算を行うという方法である．また SPE は，128bit の SIMD 命令用のパイプラインを持ち，LS へのアクセス命令用のパイプラインと合わせて，2-Way のスーパースカラパ

イプライン構造となっている．計算のレイテンシは大きいが，128本の豊富なレジスタを利用して，複数のデータに対して同時に処理を適用することで，計算のレイテンシを隠蔽することが可能である．

2.2.2 マルチコア環境の問題点

マルチコア環境におけるアプリケーション開発では，そのアプリケーションの持つ並列性や，開発環境の構成を強く意識する必要があり，プログラマの負担となってしまう．特に，前項で述べた，Cell/B.E.のような複雑な構成を持つ環境であれば，様々な制約が発生し，より高度なプログラミング技術が要求される．Cell/B.E.が抱える問題点としては，以下のようなものが考えられる．

(1) Cell/B.E.の特徴を活かした，PPEとSPEを共に利用するアプリケーションを開発するためには，性質が異なる2種類のコアで動作するプログラムを，それぞれに対して用意する必要がある．また，それらのプログラムは，協調して動作するように設計にする必要があるため，プログラマには並列分散プログラミングの技術が要求される．

(2) PPEと複数のSPEを協調動作させるようなプログラムを記述する場合，Cell/B.E.で用いられているDMA（Direct Memory Access）転送と呼ばれるデータ転送方式や，プロセッサ間で同期をとる場合のメモリシステムの機構など，アーキテクチャの詳細を理解する必要がある．特に，DMA転送では，一度に転送できるデータサイズ等に制約があり，この制約を意識したプログラミングが必要となる．

(3) SPEは，SPU SIMD命令を用いることで，高速な演算を実現している．しかし，コンパイラや開発ツールを用いて自動的に最適化を行い，プログラムの高速化に繋がる箇所を抽出することは未だ困難である．そのため，SPEの演算性能を引き出すためには，プログラマがSPEを効率的に使用する技術を学習する必要があり，Cell/B.E.を用いたプログラミングの技術障壁はかなり高い．

これらを意識することはプログラマにとって負担となるため，その負担を軽減する開発環境が必要とされている．

2.2.3 関連研究

前項で述べたように，マルチコア環境におけるプログラミングには様々な問題がある．特に，Cell/B.E.は複雑な構成を持つため，一般的なマルチコア環境と同様の手法では解決できない問題も多い．Cell/B.E.や，CPUとGPUを組み合わせたヘテロジニアス環境におけるプログラミングの煩雑さを解決するための様々な手法が研究されており，その中にCVCell[4]やCellSs[5]がある．

CVCell は，コンピュータビジョン向けのライブラリ OpenCV を，Cell/B.E. 向けに最適化したものである．OpenCV は画像処理，構造解析，モーション解析，物体追跡，パターン認識など，多数のコンピュータビジョン向け処理の API を提供している．CVCell は，それらの API の内，主要なものを Cell/B.E. 向けに最適化している．CVCell で行われている Cell/B.E. 向けの最適化は，主に 3 種類に分けられる．1 つ目は，SPE を用いた並列処理である．OpenCV で提供されている処理は多くの場合並列性を持っている．そのため，複数の SPE を用いて並列に処理することで高速化を図っている．2 つ目は，SIMD 演算を用いて処理することである．Cell/B.E. は SIMD 演算を用いる事で高い性能を発揮できるプロセッサである．そして，OpenCV で提供されている処理は高いデータ並列性を持つため SIMD 演算との相性が良い．そのため，処理に SIMD 演算を用いることにより高速化が期待できる．3 つ目は，ダブルバッファリングである．Cell/B.E. では SPE で処理をするためには必要なデータを LS に転送する必要があるが，その転送にはオーバヘッドがかかる．CVCell ではその転送オーバヘッドをダブルバッファリングにより隠蔽することで高速化を図っている．以上の最適化の結果，OpenCV の API を用いた場合に比べ，CVCell の API を用いた場合は，数倍から数十倍の高速化に成功している．プログラマは，CVCell が提供する API を利用することで，SPE を使用するための処理をプログラマ自身が記述することなく，複数の SPE を用いて並列に実行するプログラムを記述することができる．しかし，CVCell ではライブラリによって提供されている処理しか行うことができない．さらに，OpenCV で提供されている API の全てが CVCell で提供されているわけではない．そのため，CVCell を用いて開発できるプログラムは限られている．特定の処理を高速化したい場合には優れた性能を発揮するが，プログラマの用途に合わせたアプリケーションの開発は困難である．

これに対し，任意の並列処理を記述できる開発環境として CellSs がある．CellSs は，逐次プログラムを，複数の SPE を使用して並列実行されるプログラムへと変換する．これにより，様々な処理を Cell/B.E. 向けに最適化することができる．また，プログラマは逐次プログラムを記述するだけで良いため，PPE プログラムと SPE プログラムへ処理を書き分ける必要は無い．CellSs が行う変換のフロー図を**図 6** に示す．CellSs はプリプロセッサによって，逐次プログラムを PPE プログラムと SPE プログラムへ変換し，これらをコンパイルしたものと CellSs のランタイムライブラリをリンクさせることで，Cell/B.E. 向けの実行ファイルを生成する．CellSs において，SPE で実行される処理の単位は，タスクと呼ばれる．タスクは関数として記述され，プログラマが pragma を用いて明示的に指定する．CellSs はタスク間のデータの依存を動的に分析

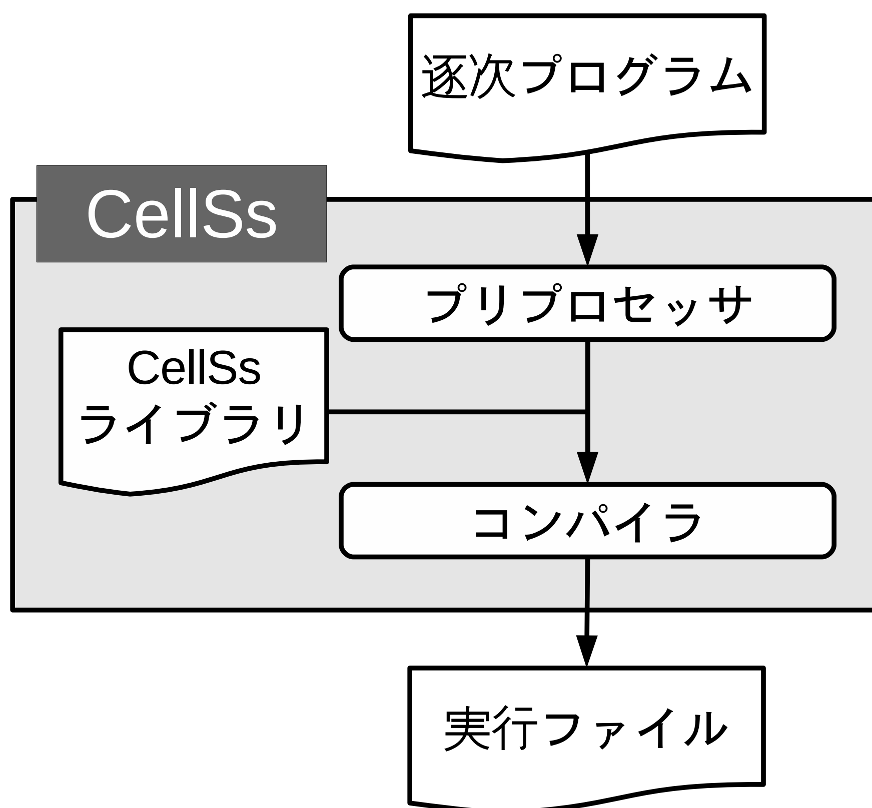

図 6: CellSs の変換フロー

し，プログラムの結果が一意に決定するように，タスクをスケジューリングする．一方，タスクの実行に必要なデータの転送は CellSs が請け負うため，プログラマが明示的にデータの転送を記述する必要は無い．このように，CellSs を用いることでプログラマは Cell/B.E. のアーキテクチャ構成を意識すること無く，プログラマ自身が記述した処理を SPE を用いて並列に実行することが可能になる．しかし，CellSs にはプログラマがプログラムの並列性を意識しなければならないことや，リダクション演算を用いて競合を自動的に回避する機能がないなどの問題がある．

3　提案ライブラリ

RaVioli は，2.1.2 項で述べたように，データ並列処理の処理単位が明確な実行モデルであるため，並列化が容易である．また，RaVioli は高速化が必要であるという問題を抱えているが，これはマルチコア環境で処理を並列実行することによって解決可能である．そのため，両者を組み合わせることにより，RaVioli の抱えている問題を解決するとともに，マルチコア環境おける並列処理プログラミングを支援することができ

る．本章では，提案するライブラリの特徴と仕様について述べる．

3.1 基本設計

2.1.4 項で述べたように，RaVioli は，抽象化のオーバヘッドにより実行速度が低下するという問題を抱えている．一方，Cell/B.E. 向け並列プログラムを記述する際には，Cell/B.E. を強く意識したプログラミングが必要であり，プログラマの学習コストが大きいことや，プログラミングそのものが煩雑であるという問題がある．そこで，本研究では，Cell/B.E. プログラミング特有の処理を全て RaVioli 内部で自動的に行うことで，動画像処理において両者の抱える問題を解決する．提案ライブラリを用いた場合，プログラマは従来の RaVioli を用いた動画像処理プログラムを記述するだけで良い．SPE プログラムの起動や DMA 転送の制御といった，Cell/B.E. を使用するにあたって必要な処理は，全て RaVioli の持つ高階メソッドが担当し，プログラマの負担となる可能性のある処理を隠蔽する．これによって，プログラマに Cell/B.E. プログラミングに必要な知識を要求することなく，高速な画像処理が可能な環境を提供することが可能になる．

従来の RaVioli を用いて，Cell/B.E. を活用するために必要な処理をプログラマ自身が記述した場合，プログラムは**図 7** のようになる．この例では，`main()` 関数（samplePPE.cpp 内）で SPE プログラムの起動と終了の処理を制御し（2-7 行目），構成要素関数である `GrayScale()`（sampleSPE.cpp 内）で DMA 転送を制御している（2,6 行目）．`proc()` は RaVioli が提供する高階メソッドであり，ravioli.so に示す処理を実行する．このように，プログラマ自身が SPE プログラムの起動や DMA 転送の制御のための処理を記述しなければならず，画像処理とは直接関係のない部分での負担が非常に大きくなることは明らかである．また，構成要素関数で DMA 転送を制御する場合，一度の DMA 転送で取得可能なデータ量が少なくなってしまうという問題がある．図 7 に示す例では，構成要素関数 `GrayScale()` には 1 画素に対する処理を記述しているため，構成要素関数が呼び出される毎に，1 画素分のデータを DMA 転送によって取得することになる．そのため，画素数と同じ回数の DMA 転送が必要になる．しかし，DMA 転送はメインメモリと SPE の持つ LS とのデータ転送であり，通信のオーバヘッドを抑えるために，転送回数は可能な限り少なく抑えるべきである．このように，従来の RaVioli を用いて記述された画像処理プログラムは，Cell/B.E. を用いて高速化することが困難である．

一方，提案ライブラリでは**図 8** に示すインタフェースを提供する．この例では，SPE

samplePPE.cpp
```
1 void main() {
2  prog=spe_image_open("sampleSPE.elf");
3  spe=spe_context_create(0,NULL);
4  spe_program_load(spe,prog);
5  spe_context_run(spe,InImg);
6  spe_context_destroy(spe);
7  sp_image_close(prog);
8 }
```

sampleSPE.cpp
```
1 void GrayScale(RV_Pixel *p) {
2  spu_mfcdma64(p, GET); //DMA_GET
3  RGB=p->getRGB();
4  /* グレースケール化 */
5  p->setRGB(RGB);
6  spu_mfcdma64(p, PUT); //DMA_PUT
7 }
8
9  int main() {
10   RV_Image InImg;
11   InImg.proc(GrayScale);
12 }
```

ravioli.so
```
1 void proc(UserProg *up) {
2  for (i = 0 to width) {
3   for (j = 0 to height) {
4    up(&pixel[i][j]);
5   }
6  }
7 }
```

図 7: 従来の RaVioli を用いた場合の Cell/B.E. 向けプログラム例

プログラムの起動や DMA 転送といった，Cell/B.E. 特有の処理は全てライブラリが提供する高階メソッドである `procPPE()` および `procSPE()` 内部で行われる．SPE プログラムの起動といった処理は `procPPE()` が，DMA 転送の制御といった処理は `procSPE()` が担当する．特に，先ほど問題となった DMA 転送は，高階メソッド内部で制御することによって，複数画素を同時に取得することが可能となり，効率良くデータを転送できるようになっている．これによって，プログラマは従来の RaVioli を用いたプログラムと同等の処理を記述するだけで，自動的に Cell/B.E. の性能を引き出すことが

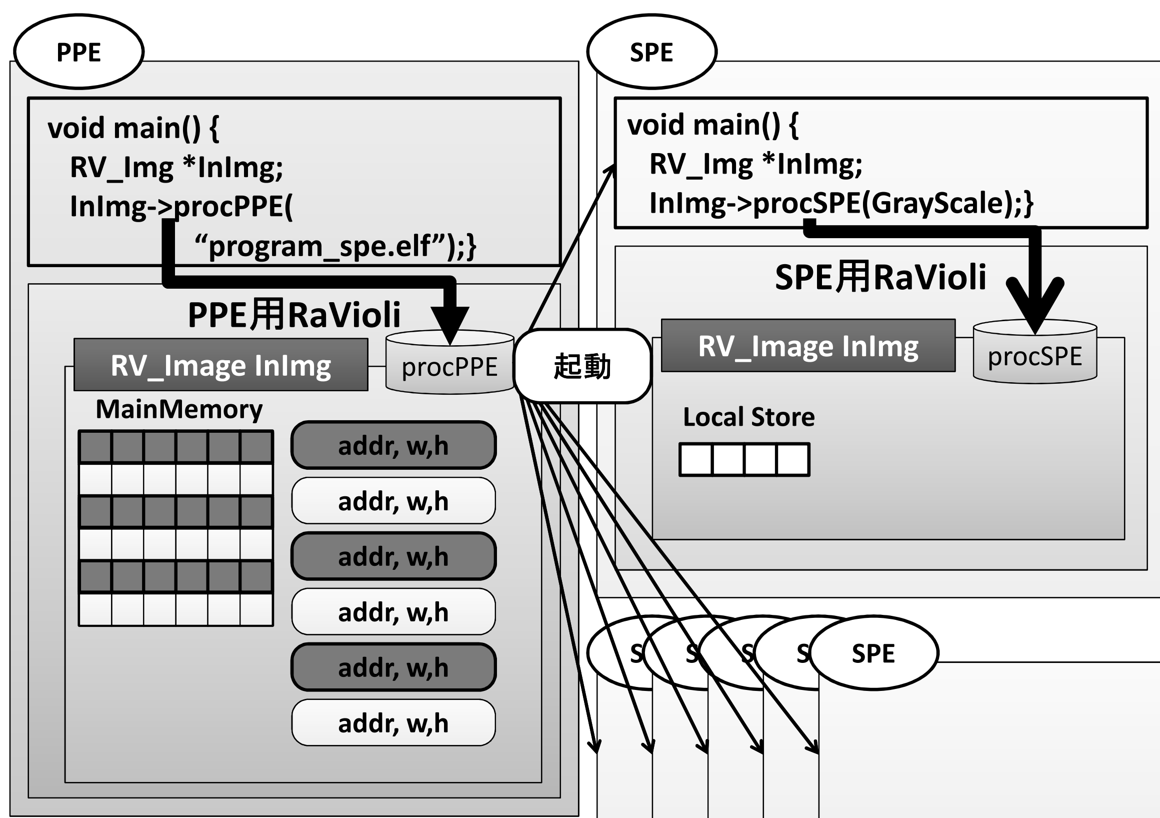

図 8: 提案ライブラリのインタフェース

可能となる．

3.2 仕様

3.2.1 RaVioli の拡張

2.2.1 項で述べたように，Cell/B.E. は PPE と SPE の 2 種類のコアを持ち，それぞれのコアは得意とする処理が異なる．PPE は，オペレーティング・システムのような，頻繁にスレッド切り替えが発生するような処理を得意とする制御系のコアである．SPE は，マルチメディア処理を得意とする演算系のコアである．この 2 種類のコアに対し，適切な処理を割り当てることで，より高速な画像処理が可能となる．Cell/B.E. を用いたプログラミングでは，PPE 向けプログラムには初期化処理等を，SPE プログラムには演算を担当させるように記述することが一般的である．そこで，本研究では，PPE に SPE の初期化，画像の読み出し，書き込みといった，プログラムの初期化処理や終了処理を担当させ，SPE に画像に対する処理を担当させる．プログラム全体の処理の流れは次のようになる．

1. 初期化
   PPE プログラムが実行され，高階メソッドが呼び出されると，ハードディスクやカメラなど，用途に応じたデバイスから画像が読み出される．画像の読み出しが完了すると，画像処理に必要な分だけの SPE プログラムを起動し，読み出した画像を分割して，各 SPE プログラムに割り当てる．SPE プログラムの起動処理を終えた PPE プログラムは，SPE プログラムが全ての処理を終えるまで待機する．
2. 画像処理
   PPE プログラムから起動された各 SPE プログラムもまた，高階メソッドを呼び出す．SPE プログラムが呼び出す高階メソッドの内部では，DMA 転送を用いた画像データの受け渡しと，プログラマの記述した構成要素関数の適用を行う．
3. プログラムの終了
   全ての SPE プログラムが割り当てられた画像データに対する処理を終えると，PPE プログラムは処理結果をファイルに書き込むなどの終了処理を行い，画像処理が完了する．

拡張した RaVioli を用いた画像処理の基本的な流れはこのようになる．

以上の処理を実現するため，本研究では，従来の RaVioli の持つ高階メソッドを，PPE プログラム向けのものと SPE プログラム向けのものの 2 種類に拡張する．これは，前述のように，従来の RaVioli を用いたプログラムの処理を，PPE プログラムが担当する部分と SPE プログラムが担当する部分に分割したことによって，それぞれのプログラム中で呼び出される高階メソッドの処理内容が異なってしまうからである．また，SPE プログラム向けの高階メソッドは，SIMD 演算によって複数画素を同時に処理することを想定するため，プログラマは SIMD 演算を用いた構成要素関数を記述しなければならない．しかし，この仕様を意識したプログラミングはプログラマの負担となってしまうため，次項で述べるユーザインタフェースを提供する．

3.2.2 ユーザインタフェース

本研究で拡張する RaVioli は，PPE プログラム向けの高階メソッドを持つ PPE 向け RaVioli と，SPE プログラム向けの高階メソッドを持つ SPE 向け RaVioli の 2 種類で構成する．3.1 節では，従来の RaVioli を用いたプログラムと同等の処理を記述するだけで Cell/B.E. の性能を引き出すことが可能であると述べた．しかし，それぞれの高階メソッドは，それぞれのコアが担当する処理に必要な引数を受け取る必要があるため，プログラムの書き換えが必要である．また，SPE を活用した Cell/B.E. 向けプログラムを作成する場合は，単純に高階メソッドの呼び出し部分を書き換えるだけで

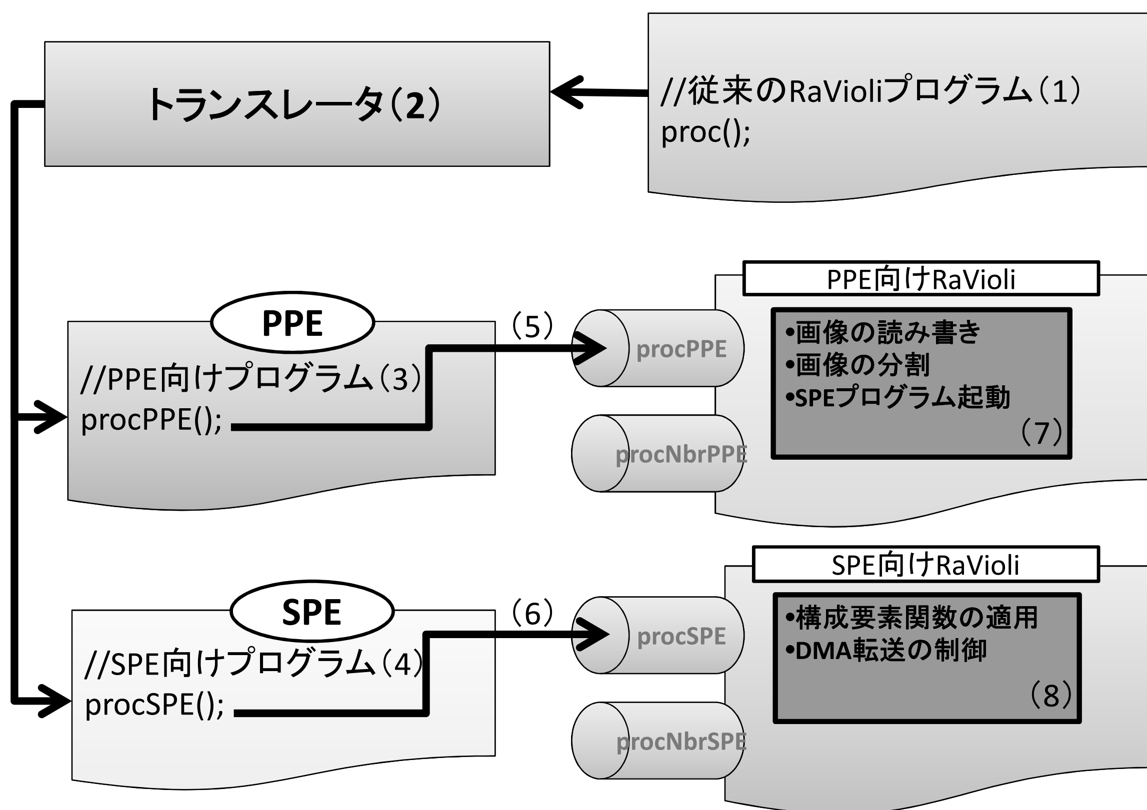

図 9: 提案ライブラリの概要

はなく，PPE プログラムと SPE プログラムの 2 種類のプログラムを記述し，それぞれのコア向けのコンパイラでコンパイルする必要がある．また，従来の RaVioli を用いて記述されたプログラムは汎用 CPU 向けのソースコードであるため，そのままでは SPE を活用することができない．従来の RaVioli プログラムをそのまま利用してしまうと，全ての処理が PPE で実行されてしまう．PPE は汎用 CPU 向けのプログラムを実行することはできるが，本来は制御系のコアであり，演算速度が遅いため，画像処理のように演算が大部分を占めるような処理は実行するべきではない．

しかし，プログラマがこのような仕様を考慮し，従来のプログラムを書き換えることは負担となってしまうため，トランスレータを用いてプログラムを変換する．トランスレータによって，従来の RaVioli を用いて記述されたプログラムから，PPE 向けプログラムと SPE プログラムの 2 種類のプログラムを生成し，従来のプログラムを書き換える負担を軽減する．ここでは，トランスレータを用いてプログラムを変換し，実行ファイルを生成するまでの過程を説明する．なお，トランスレータの動作については 5 章で詳しく述べる．

GrayScalePPE.cpp
```
int main() {
  RV_Image *img;
  readBMP(img);
  img->procPPE("GrayScaleSPE.elf");
}
```

GrayScaleSPE.cpp
```
void GrayScale(RV_Pixel *p) {
  int RGB;
  RGB = p->getRGB();
  /* グレースケール化 */
  p->setRGB(RGB);
}

int main() {
  RV_Image img;
  img.procSPE(GrayScale);
}
```

図 10: 提案ライブラリを用いた記述例

本研究で提案するライブラリの概要を**図 9** に示す．まず，提案ライブラリを利用するプログラマは，従来の RaVioli を用いた画像処理プログラムを記述する（1）．そして，このプログラムをトランスレータによって変換し（2），PPE プログラムと SPE プログラムを得る（3,4）．トランスレータによって生成された各プログラムでは，Cell/B.E. 向けプログラム特有の処理を行うための記述を含んだ高階メソッドを呼び出す（5,6）．呼び出された高階メソッド内部では，各種初期化や DMA 転送の制御を行っているが（7,8），プログラマがこれらの処理を意識する必要はなく，ライブラリ内部で自動的に適切な処理が実行される．

本研究で提案するライブラリを用いて記述した画像処理プログラムの例を**図 10** に示す．PPE プログラムでは，高階メソッド呼び出しによって SPE プログラムを起動する．SPE プログラムを起動するためには，そのプログラム名が必要であり，図 10 の例では，GrayScaleSPE.cpp をコンパイルして生成された実行ファイルである GrayScaleSPE.elf を高階メソッドの引数として与えている．これによって，高階メソッド内部で，図 7 の samplePPE.cpp に示したような，SPE プログラムの起動と終了を制御する処理が自動的に実行される．また，各 SPE に割り当てる処理量が均等になるように調整することで，特定の SPE がボトルネックになってしまうことを防ぐ．

一方，SPE プログラムは構成要素関数を実行する．PPE プログラムから起動された

```
1   void GrayScale(RV_Pixel *p) {
2     vector int RGB;
3
4     for (i = 0; i < 4; i++) {
5       RGB[i] = p[i].getRGB();
6     }
7
8     /* SIMD 演算によるグレースケール化 */
9
10    for (i = 0; i < 4; i++) {
11      p[i].setRGB(RGB[i]);
12    }
13  }
```

図 11: 従来の RV_Pixel クラスを Cell/B.E. 向けプログラムで扱う際の問題点

SPE プログラムは，従来の RaVioli と同様，構成要素関数を引数とした高階メソッドを呼び出す．SPE 向けの高階メソッドは，内部で DMA 転送を制御するため，プログラマ自身がデータ転送を意識したプログラムを記述する必要はない．こうして生成された，PPE 向け，SPE 向けの各プログラムを，各プログラム向けのコンパイラによってコンパイルすることで，実行ファイルを得ることができる．

以上の拡張とインタフェースの提供によって，プログラマは，Cell/B.E. プログラミング特有の記述を意識することなく，画像処理を記述することが可能になる．一方で，前項で述べたように，本研究で拡張した SPE プログラム向けの高階メソッドは，構成要素関数中で SIMD 演算を用いることによって複数画素を処理することを想定して実装するが，Cell/B.E. で SIMD 演算を扱うためには，各オペランドは vector 型変数として宣言しなければならない．しかし，従来の RaVioli では，そのようなことは想定されていないため，構成要素関数に SIMD 演算を用いた処理を記述する際に，不都合が生じてしまう．従来の RV_Pixel クラスを Cell/B.E. 向けプログラムで扱う際に発生する問題を，**図 11** に示すプログラムを例に説明する．ここで，図 11 は従来の RaVioli を用いて記述されたグレースケール化プログラム中の演算部分を，SIMD 演算を用いることによって 4 画素同時に処理するように書き換えたものである．

```
1   void GrayScale(RV_Pixel *p) {
2     vector int RGB;
3
4     RGB = p->getRGBs();
5     /* SIMD 演算によるグレースケール化 */
6     p->setRGBs(RGB);
7   }
```

図 12: 拡張後した RV_Pixel クラスのメソッド利用例

前述のように，従来の RaVioli では，RV_Pixel インスタンスは 1 画素分の情報しか保持していない．そのため，複数の画素を同時に処理する際には，複数の RV_Pixel インスタンスの持つ画素情報を，一つの vector 型変数へと格納し直さなければならず（4-6 行目），処理結果を格納する際も，vector 型変数の各要素に対して格納処理をしなければならない（10-12 行目）．また，図 11 に示す環境では，一つの SIMD 命令で 4 画素処理しているが，一つの SIMD 命令で 8 画素処理できる環境にプログラムを移植する場合，プログラムの書き換えが必要になり，移植性が低くなってしまうだけでなく，SIMD 演算の処理単位を意識したプログラミングが必要になってしまう．

以上のような理由から，本研究で Cell/B.E. 向けに拡張した RaVioli では，RV_Image クラスの持つデータ構造を変更し，あわせて RV_Pixel クラスの内部動作も変更することで，複数画素を扱えるように拡張した．拡張後の RV_Pixel クラスのメソッドを利用して記述されたプログラム例を**図 12** に示す．拡張後の RV_Pixel クラスのメソッドである `getRGBs()` は，内部で複数画素を操作し，vector 型変数として返している．これによって，プログラマが複数画素を扱うことを意識した記述をする負担を軽減している．`setRGBs()` メソッドについても同様であり，変数 `RGB` に格納された，複数の画素に対する処理結果を同時に格納することができる．また，図 10 の GrayScaleSPE.cpp と図 12 に示したプログラムを比較すると，インタフェースは従来の RaVioli と同様であることがわかる．そのため，プログラマは，複数画素を一つの vector 変数にまとめることや，一つの SIMD 命令で処理できるデータ量といった，Cell/B.E. 向けに施された最適化について意識することなく，各種メソッドを利用することができる．

4　ライブラリ実装

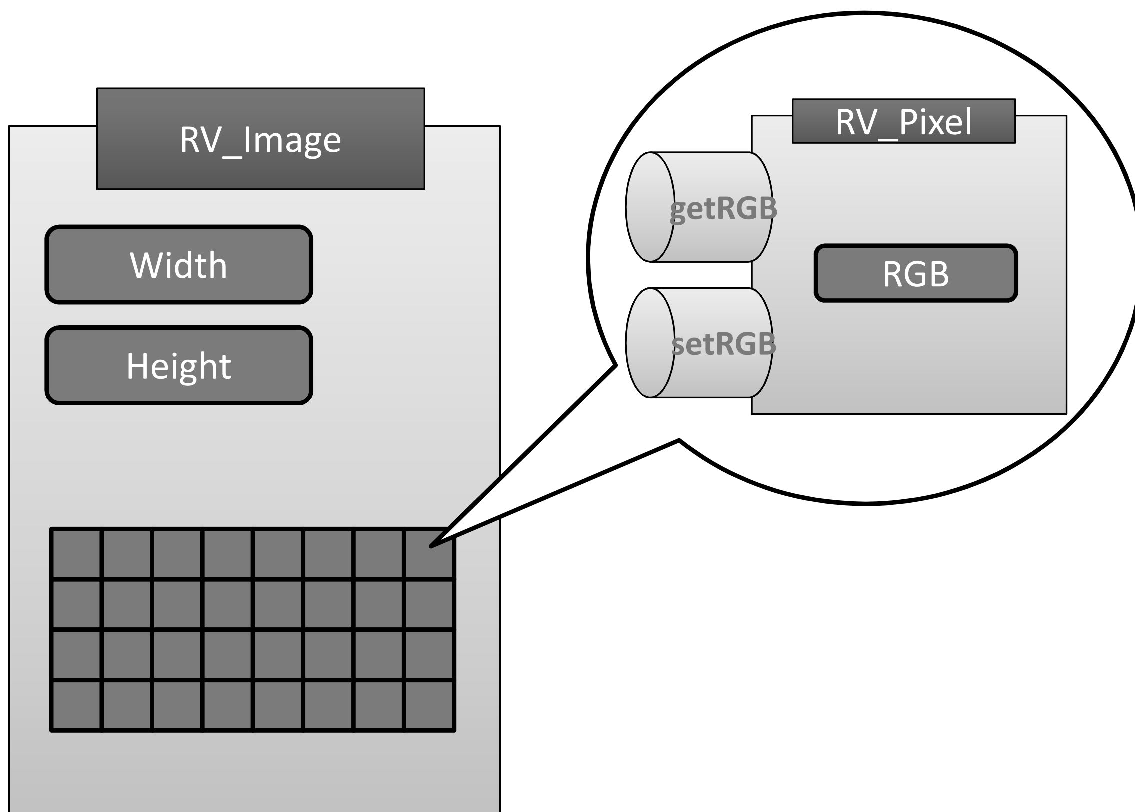

図 13: 従来の RV_Image クラスと RV_Pixel クラス

3 章では，従来の RaVioli を用いた画像処理を，PPE と SPE の特徴を考慮して各コアに分担させ，Cell/B.E. 上で効率的に画像を処理する手法を提案した．さらに，従来の RV_Image クラスおよび RV_Pixel クラスのデータ構造や動作は，SIMD 演算を扱う際に問題があることを述べた．提案ライブラリでは，従来の RaVioli の処理のうち，用途に応じたデバイスからの画像の読み出しや，処理結果のファイルへの書き込みなどの，プログラムの初期化や終了に関わる箇所は PPE プログラムの担当とする．一方，画像の構成画素に処理を適用する箇所は SPE プログラムの担当とする．本章では，処理の分担を実現するための実装と，拡張した RV_Image クラスと RV_Pixel クラスのデータ構造およびその動作について述べる．

4.1 RV_Pixel クラスの拡張

3.2.2 項で述べたように，従来の RaVioli では，構成要素関数は 1 画素，または 1 画素とその近傍画素集合といった処理単位に対する処理を記述する関数であるため，構成要素関数内部で複数画素を処理することを想定していない．そのため，RV_Pixel インス

タンスは 1 画素分の情報しか保持していない．従来の RV_Image クラスと RV_Pixel クラスのデータ構造を**図 13** に示す．このように，従来の RV_Image クラスは，RV_Pixel インスタンスの配列を保持している．そして，プログラマは，RV_Pixel クラスの提供する getRGB() メソッドや setRGB() メソッドを通じて画素データを読み書きし，処理を施す．これら従来の RV_Pixel クラスが提供するメソッドを Cell/B.E. 向けプログラムで利用する時，二つの問題が発生する．一つめは，各 RV_Pixel インスタンスが自分自身の保持するデータ以外にはアクセスできないことである．そのため，SIMD 演算によって複数画素を同時に処理したい場合，複数の RV_Pixel インスタンスから画素データを取得し，一つの vector 変数として格納し直さなければならない．二つめは，一つの SIMD 命令で処理できる画素数は環境に依存することである．4 画素を同時に処理できる環境で記述されたプログラムは，8 画素を同時に処理できる環境に移植する際，大幅な書き換えが必要となる．これは，従来の RaVioli が図 13 に示すようなデータ構造を持つために，ライブラリ側で一つの SIMD 命令で処理できるデータ量を隠蔽することができず，図 11 に示したように，構成要素関数に直接記述しなければならないことから発生する問題である．

そこで，構成要素関数で SIMD 演算を扱いやすい環境を提供するため，本研究では RV_Image クラスおよび RV_Pixel クラスのデータ構造を変更し，さらに，各種メソッドの内部動作を変更する．拡張後の RV_Image クラスと RV_Pixel クラスのデータ構造を**図 14** に示す．拡張後の RV_Image クラスは，RV_Pixel インスタンスの配列ではなく，画素配列を保持する．そして，この画素配列を操作するために，拡張後の RV_Pixel インスタンスを一つだけ保持する．一方，拡張後の RV_Pixel クラスは図 14 の右側に示すような構成になっており，RV_Image クラスの持つ画素配列へのポインタと，その画素配列を操作するためのメソッドを持っている．各種メソッドは，環境に合わせて複数画素を vector 型変数にまとめて操作するように拡張されている．例えば，一つの SIMD 命令で 4 画素を処理できる環境であれば，getRGBs メソッドによって取得できるデータは 4 画素分となる．また，SIMD 演算を使用する際には，オペランドの型が一致している必要がある．例えば，浮動小数点演算を実行したい場合は，オペランドの型が共に vector float 型でなければならない．そこで，画素を取得する段階で，その型を指定できるようにするため，拡張後の RV_Pixel クラスには，複数の get メソッドを実装した．メソッド名には命名規則があり，取得したい色と型が容易に判断できるようになっている．例として，符号付き整数型として赤色の情報の取得したい場合には，getRsi() メソッドを，浮動小数点数型として緑色の情報を取得したい場合には，

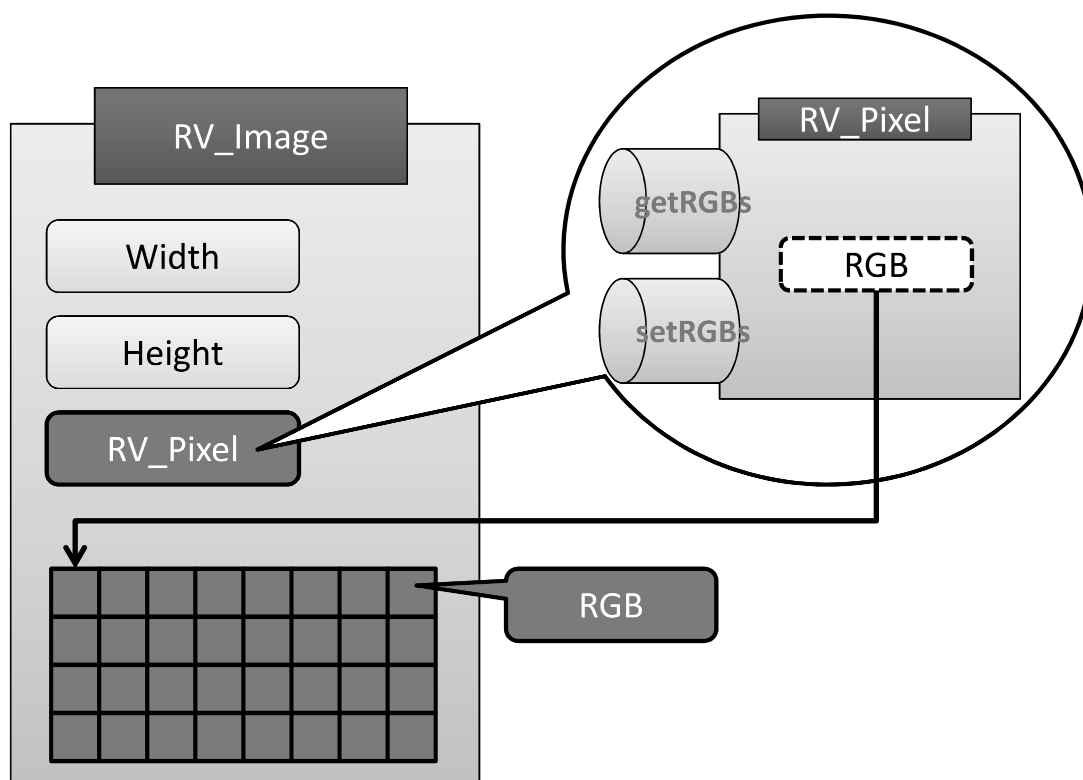

図 14: 拡張後の RV_Image クラスと RV_Pixel クラス

getGf() メソッドを利用する．このように，型を指定した画素情報の取得が可能である．

以上の拡張は，ライブラリ内部の動作を変更することによって実現した．そのため，プログラマには従来の RaVioli と同様のインタフェースを提供することが可能となり，プログラミングの負担を軽減しつつ，SIMD 演算による高速な画像処理が可能な環境を提供することができる．

4.2 PPE 向け高階メソッド

PPE プログラムから呼び出される高階メソッドは，従来の RaVioli を用いたプログラムのうち，プログラムの初期化，画像の読み出し，書き込み部分を担当する．PPE プログラムの実行を開始し，高階メソッドを呼び出すと，内部で SPE プログラムを起動するための初期化処理が行われる．一般的な Cell/B.E. 向けプログラムにおいて，SPE プログラムを起動してから終了するまでの流れを**図 15** に示す．ここでは，spe_main.elf という名前の SPE プログラムを実行するものとする．まず，`spe_image_open()` 関数を用いて，ELF 実行ファイルに格納された SPE プログラムのイメージをオープンする（8

```
1   int main(){
2     spe_context_ptr_t spe;
3     spe_program_handle_t *prog;
4     unsigned int entry;
5     spe_stop_info_t stop_info;
6
7     //SPE プログラムイメージのオープン
8     prog = spe_image_open("spe_main.elf");
9     //SPE コンテキストの生成
10    spe = spe_context_create(0, NULL);
11    //SPE プログラムのロード
12    spe_program_load(spe, prog);
13    //SPE プログラムの実行
14    entry = SPE_DEFAULT_ENTRY;
15    spe_context_run(spe, &entry, 0,
16                    NULL, NULL, &stop_info);
17    //SPE コンテキストの破棄
18    spe_context_destroy(spe);
19    //SPE プログラムイメージのクローズ
20    spe_image_close(prog);
21
22    return (0);
23  }
```

図 15: 一般的な Cell/B.E. 向けプログラムにおける SPE プログラムの実行

行目）．そして spe_context_create() 関数を用いて SPE コンテキストを生成し（10 行目），spe_program_load() 関数を用いて，オープンされた SPE プログラムを LS へロードする（12 行目）．SPE コンテキストにロードされたプログラムは，spe_context_run() 関数により実行される（14，15 行目）．処理を終え，アプリケーションにとって不要になった SPE コンテキストは，spe_context_destroy() 関数により破棄する（17 行目）．最後に spe_image_close() 関数を用いて，オープンされた SPE プログラム・イ

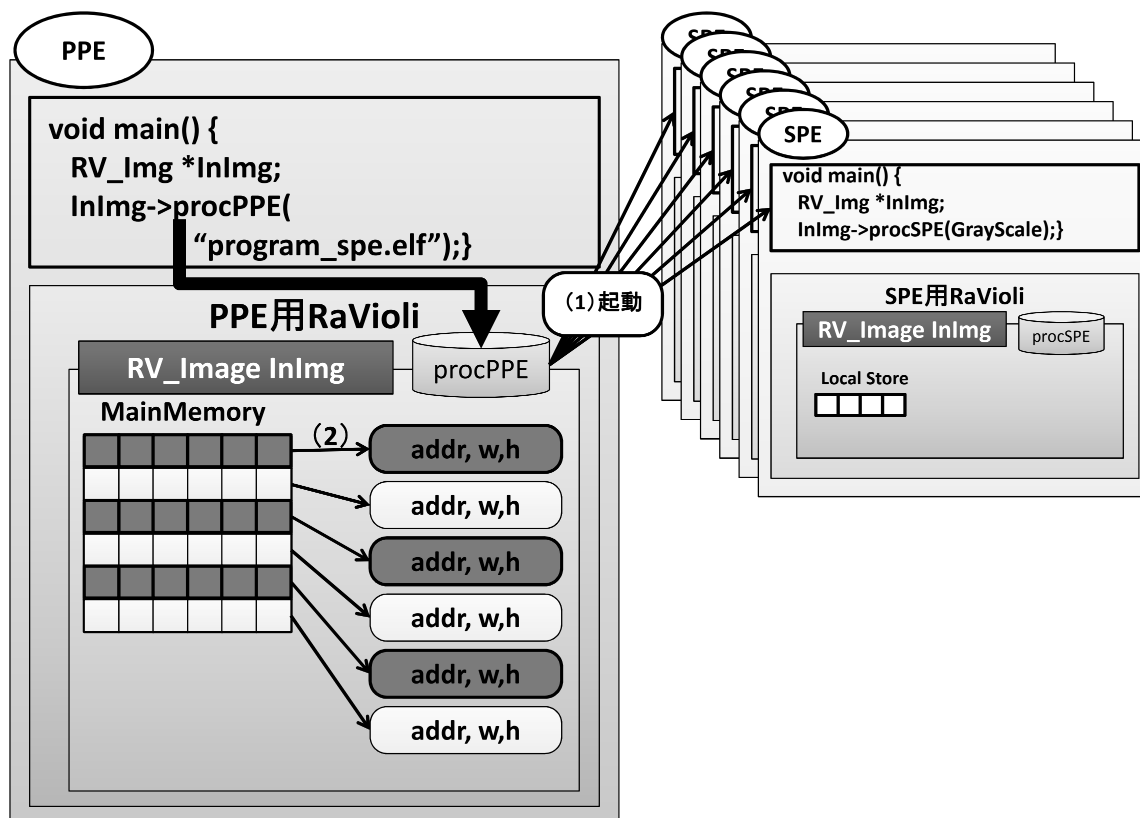

図 16: PPE 向け高階メソッドの動作

メージをクローズする（19 行目）．ここまでが，PPE プログラムから SPE プログラムを実行する基本的なプログラミング手法である．この処理を毎回プログラマが記述する負担を軽減するため，PPE プログラム向けに提供している提案ライブラリの高階メソッドでは，この初期化処理を自動的に行う．高階メソッドが 6 つの SPE プログラムを起動する様子を**図 16** に示す．図 16 に示すように，SPE プログラムを起動しているのは PPE プログラム向けの高階メソッドであり（1），プログラマが初期化処理を記述する必要はない．

また，SPE は複数使用されることが一般的であるが，SPE プログラムを実行するための `spe_context_run()` 関数は，SPE が処理を終了するまで制御を返さない仕様になっている．そこで通常，PPE プログラムではスレッドを生成し，各スレッドが SPE プログラムを起動し，最後に同期を取るようなプログラミング手法が取られる．このスレッドの生成と同期に関しても，高階メソッド内部で自動的に制御するため，プログラマは SPE プログラムの起動に関して特別な記述をする必要はない．

さらに，PPE プログラムで使用する提案ライブラリにおける高階メソッドでは，画

像の分割処理も行う．すでに述べたように，通常 SPE は複数使用されるため，画像を分割して各 SPE に割り当てることになる．図 16 の例では，6 つの SPE プログラムを起動するため，画像の分割数は，起動する SPE プログラムの数に一致する 6 分割となる．そこでまず，読み出した画像の幅と高さの情報から画素配列の要素数を計算する．そして，各 SPE に対して均等な処理量になるように，割り当てる画素数を計算する．ここで問題となるのが，DMA 転送の制約である．一度の DMA 転送で転送されるデータは，16 バイトでアライメントされている必要がある．そこで，先程計算された，各 SPE に割り当てる画素数を，そのサイズが 16 の倍数バイトになるように再計算する．画素数の計算を終えたところで，各 SPE が処理を担当する最初の要素のアドレスを設定する．画素配列のある要素のアドレスを $pixel[n]$ とし，ある SPE が担当するデータサイズを s とすると，一つ目の SPE には $pixel[0]$ を設定し，二つ目の SPE には $pixel[s]$ を設定することになる（2）．

ここで，一部の SPE については割り当てる画素数について注意が必要である．画像を N 分割するとき，元の画像のサイズを S バイト，S/N の画像のサイズを s バイトとすると，N 分割された画像の内，$N-1$ 枚の画像は全て s バイトに分割され，そのどれもが前述の計算によって 16 バイトでアライメントされている．しかし，N 枚目の画像については，$S-(N-1)s$ バイトの大きさに分割されており，この画像を担当する SPE のみ処理量が異なってしまう可能性があるだけでなく，16 の倍数バイトの大きさになっているとは限らないという問題がある．処理量の違いはそれほど大きな問題とならないが，DMA 転送で転送されるデータサイズが 16 の倍数バイトになっていないことは致命的であり，プログラムが正しく動作しなくなってしまう．そこで，N 枚目の画像を正しく転送するために，N 番目の SPE が処理することになっている $S-(N-1)s$ バイトのデータに余分なデータを付加し，16 の倍数バイトになるように調整する必要がある．この調整の様子を**図 17**，**図 18** に示す．図 17 に示すように，画像を読み出す際，画像本来のデータ量である S バイトの領域を確保するのではなく，調整用の領域を含めた $S+15$ バイトの領域を確保する．これは，図 18 に示すように，$S-(N-1)s$ バイトのデータに余分なデータ β を付加し，16 の倍数バイトのデータとして転送するためである．この余分なデータは，最大でも 15 バイトあれば十分である．以上のようにして，領域の確保と画像の分割を行う．この方法では，処理量が異なってしまう問題を解決できていないが，その差は高々 N バイトであるため，性能に大きな影響を与えることはないと考えられる．通常，SPE を用いた画像処理には，以上のような点に注意を払う必要があるが，これらは全て高階メソッド内部で自動的に処理される．そ

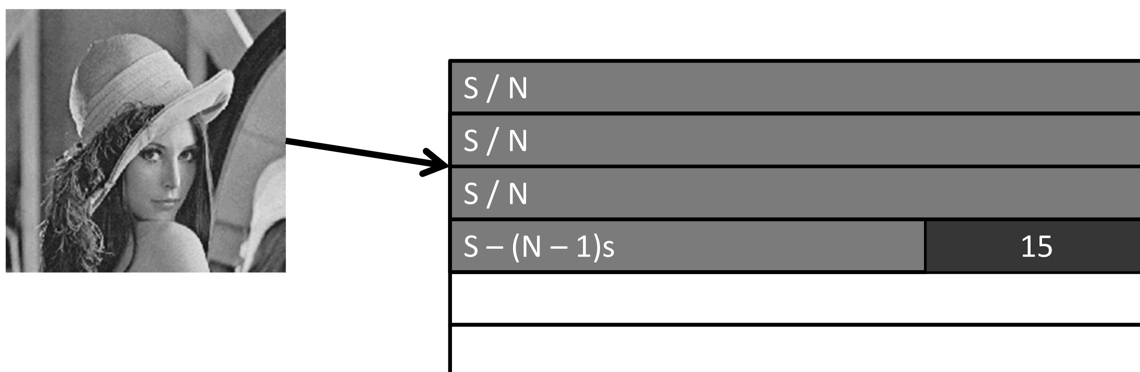

図 17: N 基目の SPE のための領域確保

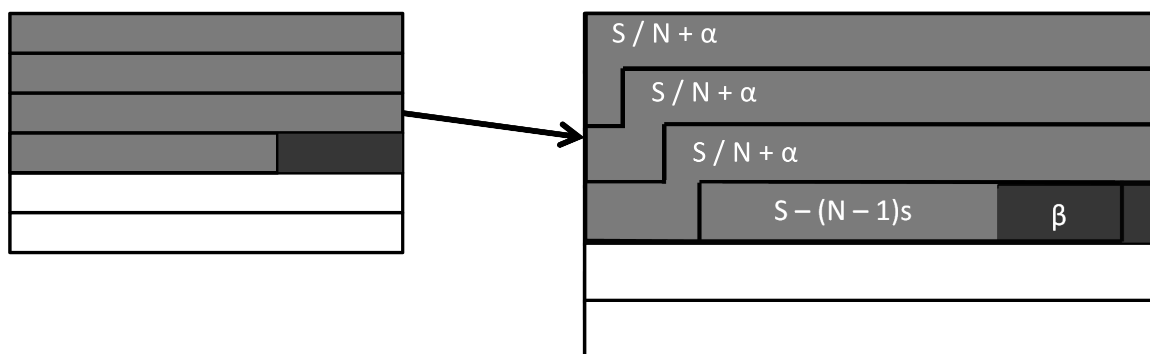

図 18: N 基目の SPE のための領域調整

のため，やはりここでもプログラマは特別な記述をする必要はない．

以上の機能は，従来の RaVioli の持つ高階メソッド内部に，SPE プログラムの実行に必要な処理と，画像データ分割のための計算処理を記述し，PPE プログラム向けの高階メソッドとして実装することで実現する．図 16 で示したように，PPE プログラムでは PPE プログラム向け高階メソッドを呼び出すのみであり，SPE プログラムの起動や画像の分割といった処理は全て隠蔽されている．

4.3　SPE 向け高階メソッド

SPE プログラムから呼び出される高階メソッドは，従来の RaVioli を用いたプログラムのうち，動画像に対して構成要素関数を適用する部分を担当する．SPE プログラム向け高階メソッドの動作を**図 19** に示す．PPE プログラムによって起動された SPE プログラムは，PPE プログラムから画像情報を受け取るための準備をする．具体的には， 2.1.2 項で説明した，画像を管理する RV_Image クラスをインスタンス化し，SPE

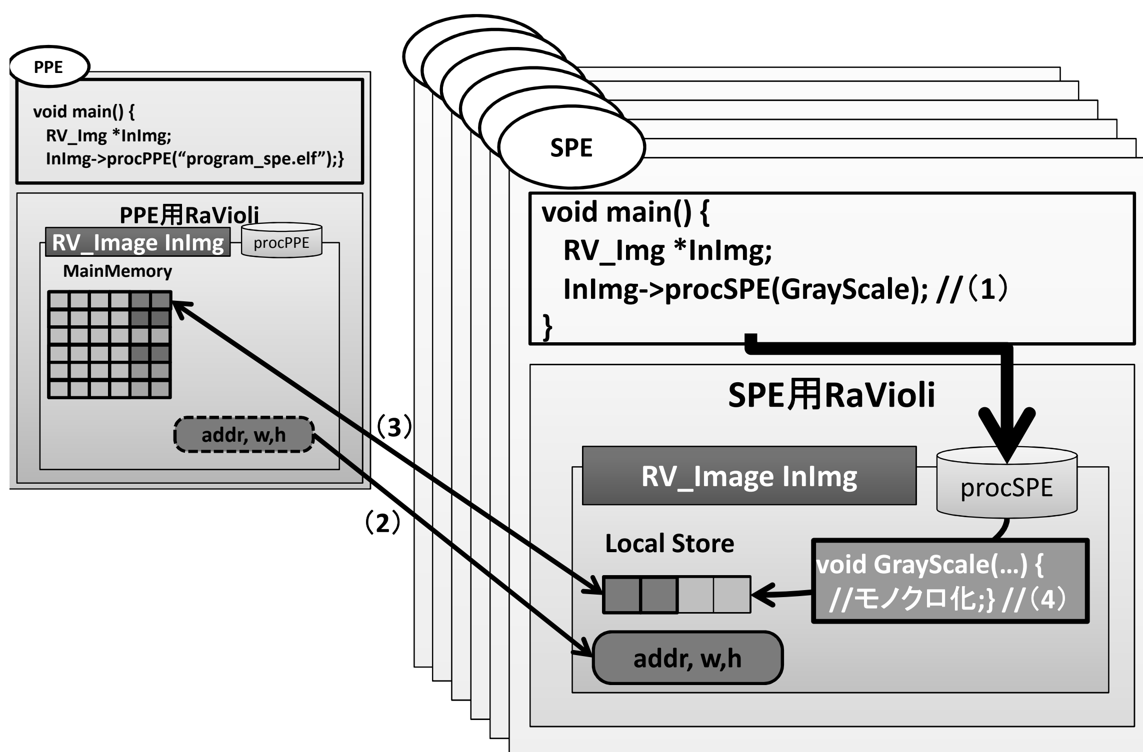

図 19: SPE 向け高階メソッドの動作

プログラム向けに提供された高階メソッドを呼び出す（1）．SPE 向け高階メソッドでは，まず，メインメモリから画像データを取得するための DMA 転送が行われる．この DMA 転送によって，画像のサイズなどの情報と，画素情報が格納されているメモリアドレスを取得する（2）．画素情報の格納アドレスは，前項で述べたように，PPE プログラム向けの高階メソッドにおいて適切に計算されており，各 SPE プログラムは，指定されたアドレスから DMA 転送によってデータを取得し始めれば良い．一度の DMA 転送では最大 16KB のデータしか転送することができないため，全画素を一度の DMA 転送で取得できるということはまずありえない．そのため，全画素に処理を適用するためには DMA 転送を繰り返し行うように制御する必要がある．すなわち，画像データの取得，取得したデータに対する処理のための構成要素関数呼び出し，処理後の画像データの書き戻しという処理を，割り当てられたサイズ分だけ処理し終えるまで繰り返す（3,4）．また，DMA 転送は Cell/B.E. を用いた処理の中でも時間のかかる処理であるため，ダブルバッファリングを用いて転送に必要な時間を隠蔽する．

グレースケール化プログラムのような，現在処理を適用しようとしている画素以外の画素の情報を必要としないプログラムでは，この流れで処理をすれば良い．しかし

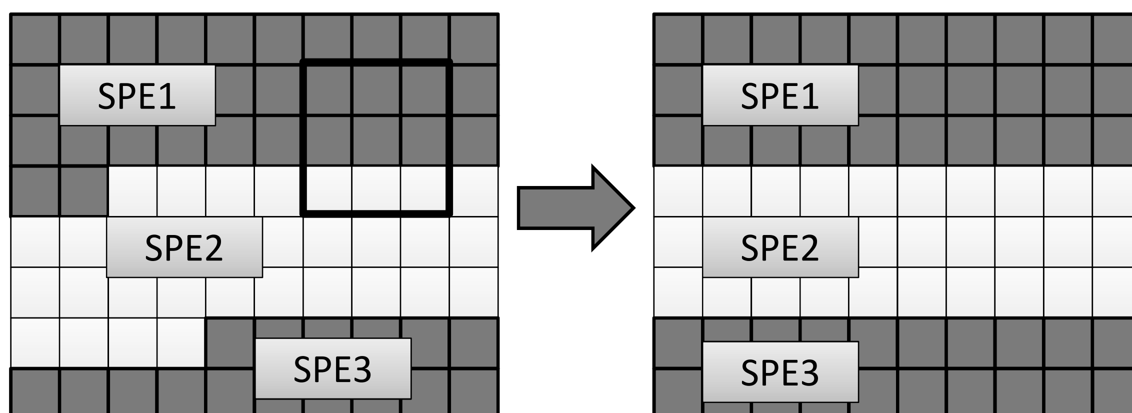

図 20: 境界の調整

ながら，現在処理を適用しようとしている画素だけでなく，例えば周囲の画素の情報を必要とするプログラムでは，やや複雑になる．ここでは，例として近傍処理を考える．近傍処理では，現在処理を適用しようとしている画素の周囲にある最大 8 画素の情報を同時に用いて処理を行う．複数の SPE を用いる場合，画像データは分割されているため，必ず別の SPE が処理を担当している部分との境界が存在する．境界がどのようになるのかを表現した例を**図 20** の左側に示す．この例では，最上段の部分の画素を担当している SPE1 は，自身の担当範囲を処理するために，SPE2 が処理を担当している中段の部分の画素を必要とする．図中の太枠は，近傍処理に必要な 9 画素が，複数の SPE の担当範囲にまたがっている様子を示している．同様に，中段の部分の画素を担当している SPE2 は，SPE1 が処理を担当する最上段の部分の画素と，SPE3 が処理を担当する最下段の部分の画素を必要とする．このように，自身の担当範囲外の画像データも取得する必要があることがわかる．そこで，図 20 の右側に示すようにして，境界部分に当たる画像の転送を容易にするための調整を行う．ここでは，図 20 の最上段の部分の画素を担当している SPE（この例では SPE1）を例にして説明する．まず，DMA 転送によって，自身が担当する部分の画像を図 20 の左側に示すように取得する．DMA 転送の制約により，16 の倍数バイトに揃っていない図 20 の右側ような転送をいきなり行うことは不可能なためである．次に，画像の横幅の倍数と，今 DMA 転送を用いて取得した画像のサイズ（通常は最大の 16KB）を比較する．画像の横幅を w，DMA 転送で取得した画像サイズを s，定数を n としたとき，$nw \leq s$ を満たす最大の n を求める．図 20 の例では，n は 3 である．これで，最上段の領域を担当する SPE の処理画素数は $3w$ であると計算された．しかし，近傍処理を行うためには，あ

```
int sum = 0;
RV_Image *tp_img;
void Compare(RV_Pixel *p1, RV_Pixel p) {
  sum += p1 - p2;
}
void CountTP(RV_DoppelImage *dimg,
             RV_Coord cs, RV_Coord ce) {
  dimg->procImgComp(Compare, tp_img);
  /* 最小値を比較して sum の値を更新 */
}
int main() {
  img->procBox(countTP, tp_img->W, tp_img->H);
}
```

図 21: テンプレートマッチングプログラム

と w だけ中段の部分の画像データを取得する必要がある．そこで，2 回目の DMA 転送で必要なだけのデータ（ここでは w を 16 の倍数バイトに調整した分の画素）を取得する．ここで初めて必要なデータが全て揃い，自身の画像処理を行うことができる．ただし，2 回目の DMA 転送を行っている間も，中段の部分の画像を使わない処理は実行することができるため，転送処理とオーバラップさせることにより，DMA 転送によるオーバヘッドを隠蔽することができる．

さらに，テンプレートマッチングのような，画像の一部にのみ処理を適用するような場合，より複雑な DMA 転送制御が必要となる．テンプレートマッチングプログラムの例を**図 21** に示し，テンプレートマッチングにおける DMA 転送のための領域分割について，**図 22** を用いて説明する．

図 21 に示すプログラムでは，構成要素関数である `CountTP()` から高階メソッドである `procImgComp()` が呼び出されている．図 22 には，左側に元画像，右側にテンプレート画像を示している．また，左側の画像中で，DMA 転送によって取得できる領域の大きさを破線，テンプレート画像に一致するサイズの領域を太枠で示している．ここで，一度の DMA 転送によって取得できる画像データのサイズは，テンプレート画像のサイズの約 2/3 であり，画像の幅と等しいとする．そのため，この時プログラマは，

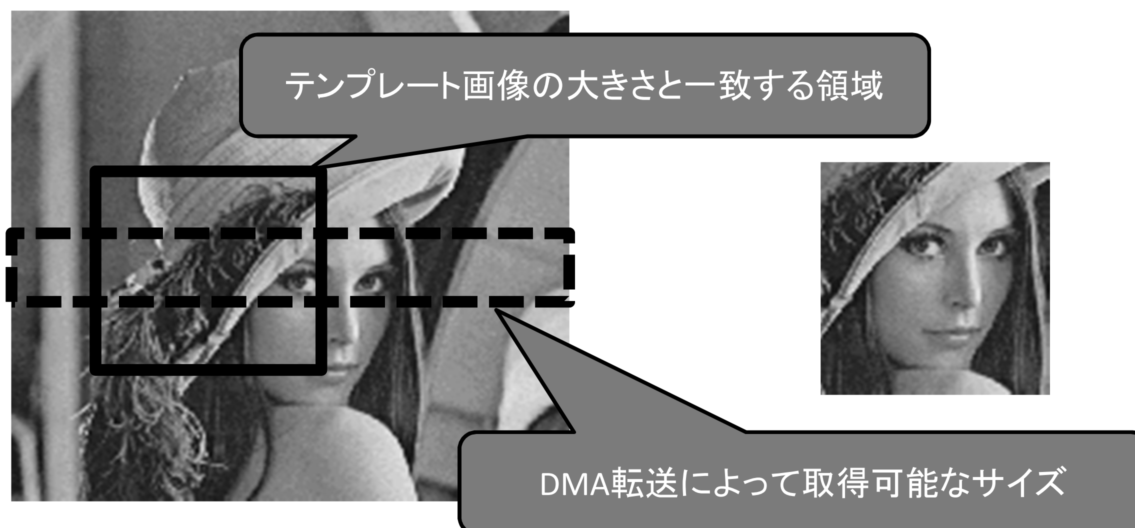

図 22: テンプレートマッチングにおける領域分割

二度の DMA 転送で処理対象の部分画像を取得することを期待する．しかし，DMA 転送は，指定したアドレスから最大 16KB の連続したデータを取得することしかできないため，図 22 中の太枠で囲まれた領域を二度の DMA 転送だけで取得することはできない．そのため，Cell/B.E. 向けに拡張した高階メソッド `procBox()` は，この太枠で囲まれた領域のデータを取得するために，3 回の DMA 転送を実行する．そのため，本来は一度の DMA 転送で最大 16KB までのデータを取得できるが，この時の DMA 転送で転送されるデータは，半分の約 8KB となる．転送効率は落ちてしまうが，図 21 に示したテンプレートマッチング処理のように，構成要素関数内部で高階メソッド呼び出しをするような記述がある場合，このように DMA 転送を制御し，プログラムに不具合がでないように配慮する必要がある．もちろん，その他の高階メソッドと同様，プログラマからはこうした処理は隠蔽されており，DMA 転送を意識したプログラミングは必要ない．

以上で述べたように，SPE プログラムでは DMA 転送の制御が非常に複雑であり，最もプログラマの負担となる処理であると考えられる．しかし，提案ライブラリでは，こうした転送処理を全て SPE プログラム向けの高階メソッド内部で処理することで，プログラマの負担を軽減することが可能である．プログラマは，構成要素関数のみを記述すれば良く，SPE プログラム向けの高階メソッドに構成要素関数を渡すという，従来と同じ手法で画像処理を記述可能である．**図 23** に，本研究による拡張後の，高階メソッド内部の構成要素関数呼び出し部分を示す．拡張後の高階メソッド内部では，構成要素

```
1   void proc(UserProgram) {
2     unsigned long long ea_src, ea_dst;
3     ea_src = ea_dst = this->pixels;
4     int ptr = 0;
5
6     initiate_dma_get_input(buf[ptr], ea_src, GET);
7     while (remain) {
8       wait_dma_completion();
9       initiate_dam_get_input(buf[ptr ^ 1], ea_src, GET);
10      for (int i = 0 to width) {
11        UserProgram(&buf[ptr][i]);
12      }
13      initiate_dma_put_result(buf[ptr], ea_dst, PUT);
14      put ^= 1;
15      --remain;
16    }
17  }
```

図 23: DMA 転送挿入後の高階メソッド

関数を呼び出すだけでなく，DMA 転送の制御が行われる．また，ダブルバッファリング手法によってDMA 転送のオーバヘッドを隠蔽する．まず，メインメモリのアドレスを変数ea_srcとea_dstに代入し，バッファを切り替えるための変数ptrを初期化する（2-4 行目）．そして，ea_srcのアドレスを用いて，DMA 転送によってデータを取得する（6 行目）．DMA 転送は，SPE で実行される演算とは非同期に行われるため，データ転送が完了したか否かを確認する必要がある．そのため，wait_dma_completion()関数を呼び出すことで，データ転送が完了するまで待機する（8 行目）．データの受け取りが完了すると同時に，次のデータをもう一つのバッファに対して転送する（9 行目）．そして，データ転送の完了を待つ間に，構成要素関数を呼び出し，画像処理を適用する（10-12 行目）．処理が完了したところで，DMA 転送によって結果をメインメモリへと返し（13 行目），処理対象のバッファを切り替える（14 行目）．以上の処理を，自身の担当範囲に対して実行する（8,15 行目）．

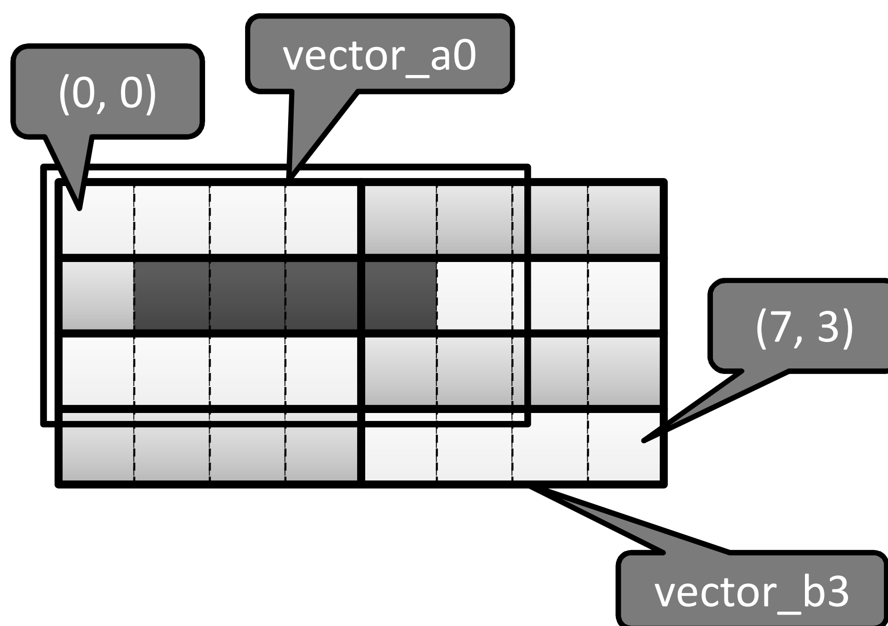

図 24: 近傍画素の扱い

このように，複雑な DMA 転送の制御をプログラマから隠蔽し，従来の RaVioli を用いた場合と同様，高階メソッドの引数に構成要素関数を与えるだけで画像処理が可能な環境を提供する．また，この拡張によって，プログラマが構成要素関数に DMA 転送を記述する必要がなくなるため， 3.1 節で述べたような，構成要素関数を適用する毎に DMA 転送が行われることによる膨大なオーバヘッドが発生しなくなり，転送効率に関する問題も解決できる．

さらに，本研究で拡張した SPE 向けの高階メソッドは， 3.2.1 項で述べたように，SIMD 演算によって複数画素を同時に処理することを想定して実装している．この時，近傍処理を始めとする，注目している画素以外の画素を処理に用いるような処理の場合，同時に扱う画素に注意しなければならない．SIMD 演算によって 4 画素同時に処理できる環境で近傍処理を行う場合を例に，この詳細について述べる．

近傍処理では，注目している画素に加え，その周囲にある 8 画素の情報が必要である．そのため，4 画素を同時に処理する場合，各画素の周囲にある 32 画素が必要となる．この例を**図 24** に示す．

図 24 中の最も色の濃い部分が注目画素であり，その周囲の太枠で囲まれた部分が今回の処理に関わる近傍画素集合である．また，以降の説明のため，各画素は座標で表現し，左上を (0, 0) 右下を (7, 3) とする．さらに，vector 変数にも id を割り振る．x 座標が

0 以上 3 以下になる画素を含む vector 変数は a，x 座標が 4 以上 7 以下になる画素を含む変数は b であるとし，そこに y 座標の値を加えて表現する．例として，$(0,0)$，$(1,0)$，$(2,0)$，$(3,0)$ を要素に持つ vector 変数は vector_a0 と表現し，$(4,3)$，$(5,3)$，$(6,3)$，$(7,3)$ を要素に持つ vector 変数は vector_b3 と表現する．この時，vector_a0 は，各注目画素の左上に位置する画素を集めたものである．同様に，vector_a1 は左の，vector_a2 は左下の画素の集合である．以上の 3 箇所については，特に複雑な処理は必要なく，構成要素関数の引数に与えれば良い．

一方，その他の画素については，そのまま扱うことができない．各注目画素の上側に位置する画素を例に説明する．各注目画素の上側に位置する画素の座標はそれぞれ $(1,0)$，$(2,0)$，$(3,0)$，$(4,0)$ となる．ここで，$(1,0)$，$(2,0)$，$(3,0)$ の 3 画素は，vector_a0 の一部である．しかし，$(4,0)$ の画素は，vector_b0 の一部である．このように，vector 変数をまたがって使用される画素を処理に用いる場合は，各変数から必要な画素を取り出すための並べ替え処理が必要となる．この並べ替えの処理は，spu_shuffle() 関数を利用する．

spu_shuffle() 関数は，3 つの引数を取り，第一引数と第二引数に指定した vector 変数を，第三引数に指定したパタンに従って並べ替え，その結果を返す関数である．spu_shuffle() 関数の使用例を**図 25** に示す．変数 `a` と変数 `b` には，int 型の要素が 4 つずつ格納されている．変数 `c` は，変数 `a` と変数 `b` の各要素をどのように取り出すのかを指定するための値が格納されており，並べ替えの対象の型（図 25 の例では vector int 型）に関わらず，vector unsigned char 型である．`c` には，16 個の char 型の値を格納し（4-9 行目），`c` の 1 番目の要素は `d` の 1 バイト目の要素を，`c` の 2 番目の要素は `d` の 2 バイト目の要素を，`c` の 16 番目の要素は `d` の 16 バイト目の要素を指定する．図 25 の 5 行目に注目すると，`c` の 1 番目から 4 番目の要素にはそれぞれ `0x08`，`0x09`，`0x0a`，`0x0b` が代入されている．そのため，`d` の 1 バイト目から 4 バイト目の要素はそれぞれ，`a` の 9 バイト目，`a` の 10 バイト目，`a` の 11 バイト目，`a` の 12 バイト目の各要素が代入される．同様に図 25 の 6 行目に注目すると，`c` の 5 番目から 8 番目の要素にはそれぞれ `0x14`，`0x15`，`0x16`，`0x17` が代入されているため，`d` の 5 バイト目から 8 バイト目の要素はそれぞれ，`b` の 5 バイト目，`b` の 6 バイト目，`b` の 7 バイト目，`b` の 8 バイト目の各要素が代入される．このようにして，`c` に 0x00 から 0x0f までの値を代入した場合には，`spu_shuffle()` 関数の第一引数に指定した変数の要素（図 25 の例では変数 `a`）が，0x10 から 0x1f までの値を代入した場合には第二引数に指定した変数の要素（図 25 の例では変数 `b`）が取り出され，最終的に変数 `d` に格納される値は 11 行目のようになる．

```
1   void main() {
2     vector signed int a = (vector int){0, 1, 2, 3};
3     vector signed int b = (vector int){4, 5, 6, 7};
4     vector unsigned char c = (vector unsigned char) {
5       0x08, 0x09, 0x0a, 0x0b,
6       0x14, 0x15, 0x16, 0x17,
7       0x10, 0x11, 0x12, 0x13,
8       0x00, 0x01, 0x02, 0x03,
9     };
10
11    //dの中身は{2, 5, 4, 0}となる
12    vector signed int d = spu_shuffle(a, b, c);
13  }
```

図 25: spu_shuffle() 関数の使用例

SPE 向けに拡張した高階メソッドでは，この spu_shuffle() 関数を用いて，画素を並べ替える．先述の通り，左上，左，左下に位置する画素は並べ替える必要はないため，上下と右端の 5 箇所に対して，この処理を施す．たとえば，下に位置する画素は，vector_a2 のうち $(1, 2)$ から $(3, 2)$ の要素と，vector_b2 の要素である $(4, 2)$ で表される画素から構成する．同様に，右上に位置する画素は，vector_a0 のうち $(2, 0)$ から $(3, 0)$ の要素と，vector_b0 のうち $(4, 0)$ から $(5, 0)$ の要素から構成する．このようにして生成した全 36 画素を構成要素関数の引数に与え，近傍処理を実行する．

以上の並べ替え処理は，DMA 転送によって取得した画像データを構成要素関数に与える直前に行う．そのため，プログラマが記述しなければならない範囲には，並べ替え処理は一切現れない．このようにすることで，Cell/B.E. プログラミング特有の処理の一つである並べ替え処理に関する知識を身に付けることなく，SIMD 演算を用いた近傍処理が可能となる．また，今回は近傍処理を例に挙げたが，テンプレートマッチングのような部分画像を処理する場合も，同様の並べ替え処理が必要となる．部分画像処理の場合は，並べ替えが発生するタイミングは注目している部分画像に対する処理が終わり，次の部分画像に対する処理に移行する時点でフラグを立てる．このフラグに応じて，並べ替えが必要な位置の画素を処理に利用しているかどうかを判定し，

実際に並べ替え処理を実行するか否かを決める．このように，部分画像を処理する場合は，近傍処理とはやや違う対応が必要である．しかし，RaVioli を用いた画像処理プログラミングでは，処理のパタンに応じて高階メソッドを使い分けることになる．そのため，このように，その処理に適した並べ替え処理を施すことができる．

以上の拡張は，従来の RaVioli を用いた場合のプログラミングモデルを踏襲したものである．RaVioli を用いた画像処理は，2.1.1 項で述べたように，構成要素関数を定義し，それを RaVioli の提供する高階メソッドに渡すことで画像全体に処理を適用する，という流れになる．そのため，ライブラリ内部では，構成要素関数が何度も呼び出されることになる．しかしながら，Cell/B.E. という環境で画像処理を実行する場合，このプログラミングモデルは適していない．これは，Cell/B.E. はコアを単純化させ高クロック化を実現するために，分岐予測機構のような複雑な回路を搭載していないからである．そのため，分岐命令が頻繁に実行されるようなプログラムを苦手としており，また，分岐予測ミスが発生した場合，20 サイクル程度のペナルティが発生してしまう．そこで，プログラム中の関数呼び出し回数を可能な限り削減するため，高階メソッドをさらに拡張する．この拡張によって，従来の RaVioli よりも大きな処理単位を持つ構成要素関数を定義可能にし，関数呼び出し回数の削減を狙う．

処理単位を拡大するためには，画像の幅や高さといった情報と，従来の RaVioli プログラムではアクセスすることのなかった領域にアクセスする手段が必要である．これは，次のようにして実現する．まず，構成要素関数の引数に，画像の幅を追加する．これによって，従来は 1 画素に対する処理を記述していたのに対し，拡張後は画像 1 行に対する処理を記述することが可能となる．次に，RV_Pixel クラスに，参照する画素をずらすためのメソッドを実装する．図 14 に示したように，本研究で拡張した RV_Pixel クラスは，メンバとして，画素情報の実体ではなくポインタを保持している．そのため，メソッドを通じてこのポインタを操作することによって，画像 1 行分の画素にアクセス可能になる．

拡張後の高階メソッドを用いてグレースケール化プログラムを記述する例を**図 26** に示す．従来の RaVioli を用いてグレースケール化プログラムを記述する場合，構成要素関数には 1 画素をモノクロ化するための処理を記述していたが，拡張後の高階メソッドでは，このように 1 行単位の処理を記述可能とする．高階メソッド `procLine()` は，画素 RV_Pixel インスタンスに加え，画像 1 行の幅 `length` を引数とする構成要素関数を受け取る．一方，構成要素関数 `GrayScaleLine()` には，画像 1 行をモノクロ化するための処理を記述する．RV_Pixel の提供する `next()` メソッドは，先述した，参照し

```
void GrayScaleLine(RV_Pixel *p,
                   int length) {
  for (int i = 0; i < length; i++) {
    vector int RGB = p->getRGBs();
    /* SIMD 演算によるグレースケール化 */
    p->setRGBs(RGB);
    p->next();
  }
}

int main(unsigned long long spe,
         unsigned long long argp,
         unsigned long long envp) {
  RV_Image img;
  img.procLine(GrayScaleLine, argp);
}
```

図 26: 拡張後の高階メソッド使用例

ている画素をずらすためのメソッドである．現在参照している画素 p に対する処理を終えた後，このメソッドを使用することによって，次に処理すべき画素を参照対象とする．これによって，構成要素関数内部で，従来よりも広い範囲を処理することが可能となり，関数呼び出し回数を削減することによって，RaVioli を利用する際に発生するオーバヘッドの一部を削減することができる．

しかし，この高階メソッドを利用してグレースケール化プログラムを記述する例の場合，1 行分の処理を記述する必要があるため，従来の RaVioli を用いる場合に比べ，抽象度が低くなってしまう．そのため，RaVioli を利用するプログラマは，抽象度または処理速度のいずれかを諦めなければならなくなってしまうという問題がある．この問題を解決する手法は，今後の課題である．

5 トランスレータ

3 章および 4 章で述べた拡張によって，プログラマの負担となる可能性のある処理

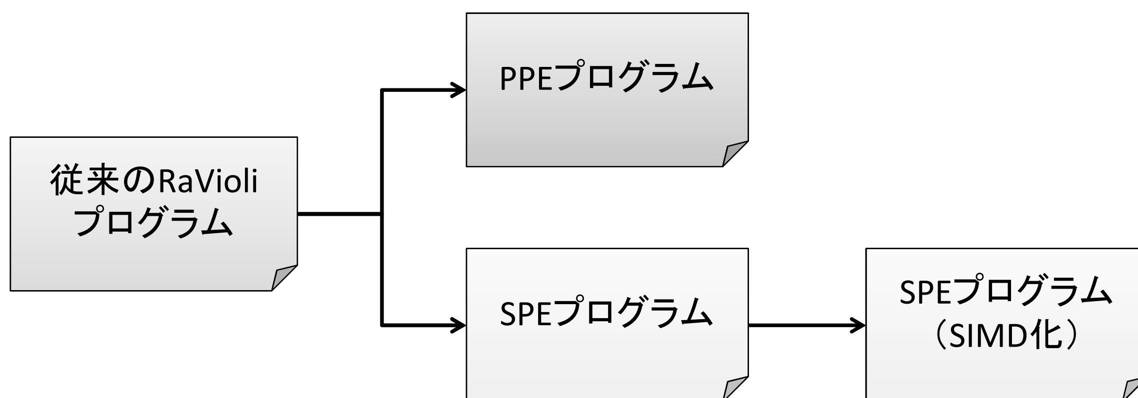

図 27: トランスレータを用いたプログラム変換の流れ

をライブラリ内部に隠蔽することが可能となった．しかし，高階メソッドの提供するインタフェースを変更したことによって，従来の RaVioli を用いて記述されたプログラムをそのまま流用することはできず，プログラムの書き換えが必要になってしまう．また，その書き換えの際，PPE プログラムと SPE プログラムの役割を考慮する必要があるなど，高階メソッドを提供するだけでは解決しきれない問題が残っている．そこで，本研究では，トランスレータを提供することによって，この問題を解決する．従来の RaVioli を用いて記述されたプログラムを，トランスレータによって変換することで，各コアの役割を考慮した記述や，SIMD 演算を意識した記述をすることなく，提案ライブラリを利用可能にする．本章では，従来の RaVioli を用いて記述された画像処理プログラムを，トランスレータによって PPE プログラムと SPE プログラムに変換するための変換規則について説明し，さらに SPE プログラムを最適化して，Cell/B.E. の性能を発揮させるための変換規則について説明する．

5.1　トランスレータの概要

本研究で提案するトランスレータは，従来の RaVioli を用いて記述されたプログラムを 2 ステップで変換する．変換全体の流れを**図 27** に示す．まず，従来の RaVioli を用いて記述されたプログラムは，PPE プログラムと SPE プログラムに変換される．この時点で，提案ライブラリを用いて実行可能となるが，このままでは，SPE プログラムとして切り出された構成要素関数内部の演算がスカラ演算であるため，Cell/B.E. が本来持つ性能を十分に発揮できない．そのため，次のステップで SPE プログラムをさらに変換し，構成要素関数内部の演算を SIMD 演算へと変換する．これによって，プログラマは自身でプログラムを書き換えることなく，従来の RaVioli を用いて記述さ

GrayScale.cpp

```
1   void GrayScale(RV_Pixel *p) {
2     int r, g, b, y;
3     p->getRGB(r, g, b);
4     y = (int)(r * 0.299
5             + g * 0.587
6             + b * 0.114);
7     p->setRGB(y, y, y);
8   }
9
10  void print_time(timeval t1, timeval t2) {
11    int sec, usec;
12    sec  = tv2.tv_sec - tv1.tv_sec;
13    usec = tv2.tv_usec - tv1.tv_usec;
14    printf("time  :%d.%06d\n", sec, usec);
15  }
16  int main(int argc, char *argv[]) {
17    RV_Image *img;
18    timeval tv1, tv2;
19    readBMP(img);
20    gettimeofday(&tv1, NULL);
21    img->proc(GrayScale);
22    gettimeofday(&tv2, NULL);
23    print_time(tv1, tv2);
24  }
```

図 28: 従来の RaVioli プログラム

れたプログラムから，本研究で拡張した RaVioli を用いて記述されたプログラムを得ることができる．

GrayScalePPE.cpp

```
1   void print_time(timeval tv1, timeval tv2) {
2     int sec, usec;
3     sec  = tv2.tv_sec - tv1.tv_sec;
4     usec = tv2.tv_usec - tv1.tv_usec;
5     printf("time  :%d.%06d\n", sec, usec);
6   }
7
8   int main(int argc, char *argv[]) {
9     RV_Image *img;
10    timeval tv1, tv2;
11    readBMP(img);
12    gettimeofday(&tv1, NULL);
13    img->proc("GrayScaleSPE.elf");
14    gettimeofday(&tv2, NULL);
15    print_time(tv1, tv2);
16  }
```

図 29: 提案ライブラリを用いた PPE プログラム

5.2 PPE プログラムの生成

変換前のプログラムである**図 28** と，変換後のプログラムである**図 29** を例として，変換手順を説明する．PPE プログラムは，従来の RaVioli を用いて記述されたプログラムから，`main()` 関数及び，`main()` 関数が直接呼び出す関数を切り出して生成する．図 28 の例では，`print_time()` 関数，`main()` 関数が該当する（10-15 行目，16-24 行目）．高階メソッドの引数として与えられている `GrayScale()` 関数は，構成要素関数であると判定できるため，PPE プログラムとして切り出される対象からは除外される．

PPE プログラムとして切り出される `main()` 関数では，画像の読み出しと高階メソッド呼び出しが行われている．この画像の読み出しと高階メソッド呼び出しとは，3 章で述べた PPE プログラムが担当する処理そのものである．また，`print_time()` 関数は，`gettimeofday()` 関数によって取得した値を引数に取り，その値を用いてプログラムの実行時間を表示する関数として定義されている．このように，`main()` 関数内部ま

たはmain() 関数が直接呼び出す関数内部で初期化やI/O 処理等を実行している部分もPPE プログラムへの変換対象である．以上の箇所を，従来のRaVioli を用いた画像処理プログラムから，PPE プログラムとして切り出す．

次に，PPE プログラム向けの高階メソッドに与えられる引数について説明する．PPE プログラムは， 4.2 節で述べたように，SPE プログラムを起動する必要があり，そのためには，起動したいSPE プログラムのプログラムファイル名が必要である．そこで，PPE プログラムで呼び出される高階メソッドには，従来のRaVioli の高階メソッドの引数であった構成要素関数へのポインタではなく，SPE プログラムのプログラムファイル名を引数として与えるように変換する．以上の変換を施した結果，最終的に生成されるPPE プログラムが，図29 に示すプログラムである．こうして得られたプログラムをPPE プログラムとしてコンパイルすることで，実行ファイルが得られる．

5.3 SPE プログラムの生成

変換前のプログラムである図28 と，変換後のプログラムである**図30** を例として，変換手順を説明する．SPE プログラムは，従来のRaVioli を用いて記述されたプログラムから，構成要素関数を切り出して生成する．これは，構成要素関数は，各画素に適用する処理が定義された関数であり， 2.2.1 項や 3.2.1 項で述べたように，各画素に処理を適用するのはSPE の役割であると考えられるからである．構成要素関数は，proc() 等の高階メソッドの引数として与えられる関数であり，図28 の例では，GrayScale() 関数が該当する．

構成要素関数のみを切り出して生成されたプログラムは，このままでは実行できない．これは，SPE プログラムもまたmain() 関数から開始されるため，従来のRaVioli には存在しなかった，SPE プログラムのためのmain() 関数を生成する必要があるためである．そこで，図30 に示すように，SPE プログラムにはmain() 関数を挿入する（10-15 行目）．PPE プログラムから起動されたSPE プログラムは，このmain() 関数から，プログラムの実行を開始する．そして，4.3 節で述べたように，DMA 転送を制御し，構成要素関数を適用するような高階メソッドを呼び出し，画像処理を実行する．

SPE プログラム向けの高階メソッドを呼び出す際もまた，引数に注意しなければならない．SPE プログラム向けの高階メソッドは，メインメモリに格納されたデータを取得するために，画像データが格納されているアドレスやそのデータのサイズなどを，PPE プログラムから受け取る必要がある．これは，PPE 向け高階メソッド内で呼び出されるspe_context_run() 関数の引数を通じて，SPE プログラムのmain() 関数が持

GrayScaleSPE.cpp

```
1   void GrayScale(RV_Pixel *p) {
2     int r, g, b, y;
3     p->getRGB(r, g, b);
4     y = (int)(r * 0.299
5             + g * 0.587
6             + b * 0.114);
7     p->setRGB(y, y, y);
8   }
9
10  int main(unsigned long long spe,
11           unsigned long long argp,
12           unsigned long long envp) {
13      RV_Image img;
14      img.proc(GrayScale, argp);
15  }
```

図 30: 提案ライブラリを用いた SPE プログラム

つ引数の一つである argp として受け取り，これをそのまま高階メソッドの引数として与える．また，SPE 向け高階メソッドを呼び出す際には，従来の RaVioli と同様，構成要素関数へのポインタを引数として与える必要がある．そのため，図 30 に示すように，SPE 向け高階メソッドは，引数を二つ受け取る（14 行目）．

こうして得られたプログラムは，構成要素関数内部の演算がスカラ演算であるため，Cell/B.E. の性能を発揮させるためには，さらに SIMD 演算を使用したプログラムに変換する必要がある．図 30 に示すプログラムをさらに変換し，構成要素関数内部の演算を SIMD 演算に変換したプログラムの例を**図 31** に示す．

図 30 に示すプログラムを図 31 に示すプログラムに変換するためには，2 つのステップを経る必要がある．まず，最初のステップでは，図 30 で使用される変数や定数を解析し，記憶する．この解析によって得た情報を基に，変数の宣言部分や定数の生成部分を変換する．図 30 の例では，r, g, b, y が変数として宣言されており（2 行目），0.299, 0.587, 0.114 が定数として演算に使用されていることが検出できる（4-6 行目）．ま

GrayScaleSPE_SIMD.cpp

```
1   void GrayScale(RV_Pixel *p) {
2     vector float vf_r, vf_g, vf_b;
3     vector float vf_cr, vf_cg, vf_cb;
4     vector unsigned int vui_y;
5
6     vf_cr = spu_splats(0.299f);
7     vf_cg = spu_splats(0.587f);
8     vf_cb = spu_splats(0.114f);
9
10    vf_r = p->getRf();
11    vf_g = p->getGf();
12    vf_b = p->getBf();
13
14    vf_r = spu_mul(vf_r, vf_cr);
15    vf_g = spu_madd(vf_g, vf_cg, vf_r);
16    vf_b = spu_madd(vf_b, vf_cb, vf_g);
17    vui_y = spu_convtu(vf_b, 0);
18
19    p->setRGB(vui_y, vui_y, vui_y);
20
21  }
```

図 31: 最終的に生成される SPE プログラム

た，浮動小数点定数を検出したことにより，vector float 型の変数を宣言しなければならないこともわかる．

次のステップでは，先ほど取得した情報を基に，SPE プログラムを生成する．まず，検出した変数に対応する宣言部分を生成する．今回の例では，検出した 4 つの変数に対応する，`vf_r`, `vf_g`, `vf_b`, `vui_y` が宣言されている（2,4 行目）．これらの変数の型は，先ほどの解析で取得した情報に従って決定される．浮動小数点演算のオペランドとなっていた `r`, `g`, `b` は vector float 型として，演算結果を格納していた `y` は vector unsigned

型として宣言される．図 30 に示す変換前のプログラムでは，これらの変数は int 型として宣言されていたが，SIMD 命令を実行する際，オペランドに指定した変数同士の型が異なるとエラーとなってしまうため，このように型を統一する必要がある．今回は，浮動小数点演算を実行しなければならないため，vector float 型としている．次に，定数部分を生成する．定数を格納するための変数 `vf_cr`, `vf_cg`, `vf_cb` を宣言し（3 行目），各変数に値を代入する（6-8 行目）．ここで呼び出されている `spu_splats()` 関数は，引数に指定された定数を 4 個の符号付き 32 ビットデータに展開したベクタデータを作成する関数である．さらに，`getRGB()` メソッドは，必要とされる型に従って変換される．今回の例では，浮動小数点演算を実行するため，値を vector float 型で取得するためのメソッドである `getRf()`, `getGf()`, `getBf()` へと変換される（10-12 行目）．演算部分は，演算子とオペランドを解析し，演算子に対応する SIMD 演算とオペランドへと変換される．`spu_madd(a, b, c)` は，$a \times b$ を実行した後，その演算結果に c を加算する関数である．また，今回は演算結果を vector unsigned 型として得るため，`spu_convtu()` 関数を挿入し，vector float 型から vector unsigned 型へと変換する．この変換が必要か否かは，図 30 の計算結果をキャストしている箇所（4 行目）を解析することによって検出できる．最後に，`setRGB()` メソッドを変換し，最終的な演算結果である `vui_y` を格納するプログラムが生成される．

こうして得られたプログラムを SPE プログラムとしてコンパイルすることで，実行ファイルが得られる．構成要素関数内部の演算を SIMD 演算へと変換したことにより，プログラマに負担を掛けることなく，Cell/B.E. の性能を発揮させることが可能となる．

6　評価

提案手法を実装し，**表 1** に示す評価環境で評価した．静止画像処理の評価には，サンプルプログラムとしてグレースケール化プログラム，エッジ検出プログラム，直線検出プログラムを用いた．また，グレースケール化プログラムには 1680×1050 ピクセルの画像を，残りの 2 つのプログラムには 1280×1024 ピクセルの画像を用いた．一方，動画像処理の評価にはグレースケール化プログラムを用いて，384×320 ピクセルの画像を 30 枚の画像を読み出し，全ての画像に処理を適応し終えるまでの時間を計測するという方法で評価した．また，この時，30 枚の画像はあらかじめメインメモリにロードされているものとし，ファイルを操作する時間等は計測しないものとした．

以上の条件で静止画像処理に対して評価した結果を**図 32** に示す．縦軸は実行時間を表しており，単位はミリ秒である．3 本あるグラフの内，左端のものが，PPE のみを

表 1: 評価環境

プラットフォーム	PLAYSTATION3
CPU	Cell/B.E.
動作周波数	3.2GHz(PPE)
使用 SPE 数	6 基
OS	Fedora 10
コンパイラ	ppu-g++
	spu-g++
最適化オプション	-O3

用いて実行した場合の実行結果である．すなわち，従来の RaVioli プログラムを，そのまま Cell/B.E. 上で実行した場合に得られる実行結果である．中央のグラフは，従来の RaVioli プログラムをトランスレータを用いて変換し，Cell/B.E. 対応 RaVioli を用いて実行した場合の実行結果である．ただし，構成要素関数を Cell/B.E. 向けに最適化せず，全てスカラ演算を用いて実行した場合の実行結果となっている．右端のグラフは，中央のグラフで用いたプログラムをトランスレータによってさらに変換し，構成要素関数内部の演算を SIMD 演算に変換した場合の実行結果である．なお，直線検出プログラムについては，後述する理由のためにトランスレータによって変換することができない箇所があるため，従来の RaVioli を用いて記述されたプログラムを，手動で書き換えたものを使用した．そのため，スカラ演算を用いて実行した場合の実行結果を計測していない．また，直線検出プログラムの実行結果に示す内訳は，下からグレースケール化とエッジ検出，ハフ変換，逆ハフ変換のそれぞれの処理に要した時間となっている．

グラフから，全てのサンプルプログラムにおいて，従来の RaVioli を上回る実行速度が得られたことが確認できた．従来の RaVioli を用いたプログラム（左端）と，トランスレータによって変換したプログラム（中央）を比較すると，グレースケール化プログラム，エッジ検出プログラムの両方で高速化が確認できた．前者は約 3 倍，後者は約 5 倍の高速化が達成できた．これは，SPE が PPE と比べて演算性能が遙かに高いことと，複数の SPE を起動し，処理を並列実行していることから得られる結果である．SPE を 6 基使用しているにも関わらず，6 倍程度の高速化が達成できない原因としては，SPE を起動するためのオーバヘッドが挙げられる．特に，グレースケール

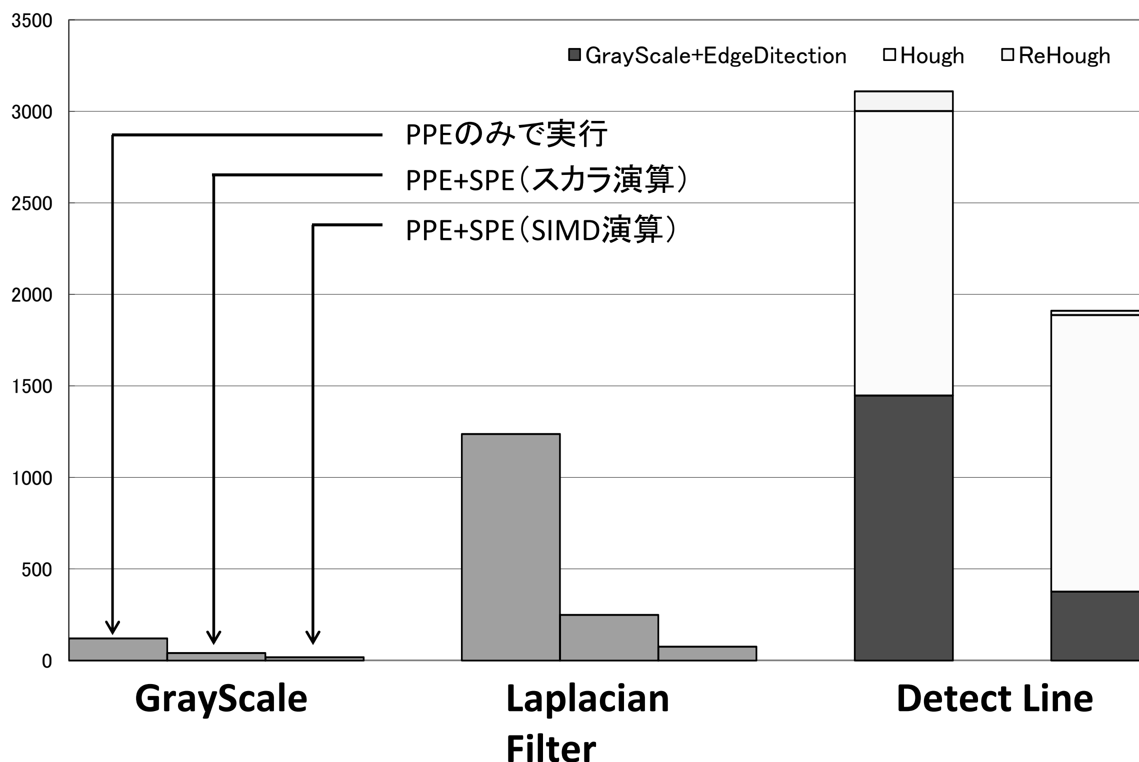

図 32: 静止画像処理の評価結果

化の処理は非常に軽量であるため，画像処理全体の実行時間に対して，SPE を起動するための処理が占める割合が大きくなったと考えられる．このことは，グレースケール化よりも重い処理であるエッジ検出の処理の実行時間が，従来の RaVioli を用いた場合の約 5 倍になっていることからも確認できる．SPE の起動に必要な時間は，起動する SPE の数が同じであれば，どのような処理をする場合であってもほぼ一定であるが，画像処理に必要な時間は，その処理内容によって違いがあるからである．

さらに，中央のグラフと，SPE プログラムをさらに最適化した右端のグラフを比較すると，より高速化していることがわかる．グレースケール化プログラムでは約 2 倍，エッジ検出プログラムでは約 3 倍の高速化が達成できた．直線検出プログラムでは，従来の RaVioli を用いた場合と比較して，約 1.6 倍の高速化が達成できた．SPE は SIMD 演算に特化したコアであるため，SPE プログラム内の演算を SIMD 演算へと変換したことによって，より高い演算性能を発揮できたと考えられる．SIMD 演算を使用しているにも関わらず，スカラ演算を使用した場合に対して 4 倍程度の高速化が達成できない原因もまた，先ほどと同様に SPE の起動オーバヘッドであると考えられる．以上の結果から，SIMD 演算を使用することの重要性が確認できるとともに，プログラマ

表 2: 動画像処理の評価結果

	コピー時間（PPE）	処理時間（SPE）	コピー時間合計（PPE）	処理時間合計（SPE）	全体の実行時間（全体）
1 回	5.5	0.3	165	9	190
100 回	5.5	11	165	330	383

に負担を掛けることなく，高速な画像処理が可能となったことがわかる．

しかしながら，直線検出プログラムにおいては，その他二つのプログラムと比較して，大きな性能向上を得られなかった．これは，次のような理由が考えられる．直線検出プログラムでは，グレースケール化，エッジ検出，ハフ変換，逆ハフ変換の順に処理を適用している．この時，ハフ変換および逆ハフ変換では，配列にランダムアクセスする必要があり，この処理を SIMD 演算で処理することはできない．また，SPE は SIMD 演算に特化したコアであり，配列にシーケンシャルアクセスしない処理とは相性が良くない．さらに，一度の DMA 転送によって取得できるデータは最大 16KB であり，アクセスしたい全ての要素が取得できるとは限らないだけでなく，LS は 256KB と小容量であるため，処理に必要な全ての要素が LS に載り切らない可能性もある．以上のような理由により，今回使用した直線検出プログラムでは，グレースケール化とエッジ検出の部分を SPE で，ハフ変換と逆ハフ変換の部分を PPE で実行している．グラフ中で内訳として示しているように，直線検出が高速化した部分のほとんどは，SPE 側で処理しているグレースケール化とエッジ検出の部分であり，従来の RaVioli プログラムを用いた場合と全く同じ動作をするハフ変換に関しては，提案手法の恩恵が得られない． このように，配列にランダムアクセスするような処理を含むプログラムへの対応は，今後の課題である．

続いて，動画像処理について評価した結果を**表 2** に示す．表に示すコピー時間とは，メインメモリ上にロードされている画像を，作業用バッファにコピーするのに要する時間であり，この処理は PPE のみが実行する．処理時間とは，1 枚の画像に処理を適用するのに要する時間であり，この処理は SPE のみが実行する．コピー時間合計及び処理時間合計とは，それぞれ前述の処理を 30 枚の画像全てに適用するのに要する時間である．全体の実行時間は，以上の処理を含む，動画像処理全体に要する時間であり，SPE プログラムの起動等の初期化処理に要する時間を含んでいる．以上の実行時間について，構成要素関数内部でグレースケール化の処理を 1 回だけ実行したものと，100

回実行することにより，擬似的に重い処理を実現したものの 2 通りを評価した．

まず，グレースケール化の処理を 1 回しか適用していない場合については，その処理時間のほとんどを，PPE プログラムのコピー処理に費やしていることがわかる．すなわち，SPE プログラムの処理は PPE プログラムの処理に完全にオーバラップしており，処理時間が隠蔽されていることがわかる．このように，処理内容が非常に軽量な場合，PPE プログラムによる画像読み出し処理がボトルネックとなってしまう．そのため，作業用バッファを複数用意し，あらかじめ画像をロードしておくなどの対策が必要であるとわかる．

一方，グレースケール化の処理を 100 回適用し，擬似的に重い処理を実現した場合については，その処理時間のほとんどを SPE プログラムの処理に費やしていることがわかる．すなわち，先ほどとは逆に，PPE プログラムの処理が SPE プログラムの処理に完全にオーバラップしており，画像のコピー時間が隠蔽されていることがわかる．このように，処理内容が重い場合，処理量の調整が必要である．従来の RaVioli は，リアルタイム処理が困難になったと判断すると，自動的に処理量を調整する機構を備えているため，この問題を解決できるが，本研究で拡張した RaVioli はこの機構を備えておらず，処理量調整については今後の課題である．

7　おわりに

本論文ではまず，利用可能なリソース量が動的に変動するような環境において，動的に解像度を変動させることで処理量を減らし，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 について述べた．また，マルチコア環境について触れ，その中でも特に特徴的なアーキテクチャを持つ Cell/B.E. について述べた．そして従来の RaVioli の問題点を指摘し，さらに従来の RaVioli を用いて Cell/B.E. 向けのプログラムを記述することが困難であることを述べ，画像処理における両者の問題を解決するためのライブラリを提案し，実現した．提案したライブラリによって，プログラマは Cell/B.E. を用いた画像処理を，RaVioli を用いて記述することが可能となった．また，提案ライブラリを用いたプログラムを得るためのトランスレータも提案した．トランスレータによって従来の RaVioli を用いて記述されたプログラムを変換することで，プログラマの負担を軽減しつつ，高速な画像処理が可能な環境を提供した．これにより，従来の RaVioli を用いて記述されたプログラムを一切書き換えることなく，Cell/B.E. の持つ性能を引き出すことが可能となった．

今後の課題として，ハフ変換など，ランダムアクセスが発生する処理の最適化が考

えられる．6 章で述べたように，今回評価に用いた直線検出プログラムでは，ハフ変換と逆ハフ変換の処理を PPE で実行してしまっており，Cell/B.E. の性能を十分に発揮できていない．そのため，こうした処理をより Cell/B.E. 向けに最適化することにより，さらに多くの画像処理プログラムを高速に処理することが可能となる．

また，画像処理に特化した言語との連携が考えられる．より直感的に画像処理を記述し，トランスレータによって本研究で提案したライブラリを使用可能になれば，プログラマの負担を大きく軽減させることができる．同時に， 5 章で述べたトランスレータも拡張し， 4.3 節で述べた，SPE 向けにさらに拡張した高階メソッドを使用したプログラムへの変換に対応することで，より高速な画像処理が可能な環境を提供できるようになる．さらに，Cell/B.E. だけでなく，GPU を搭載している環境や，ホモジニアスマルチコアな環境も存在する中で，特定の環境を意識したプログラミングが必要なくなり，より汎用的なライブラリを提供することができるようになると考えられる．

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，今井満寿巳氏，小野亜実氏，祖父江宏祐氏に深く感謝致します．

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

[4] Fixstars Corporation: OpenCV on the Cell, http://cell.fixstars.com/opencv/index.php/OpenCV_on_the_Cell.

[5] Bellens, P., M.Perez, J., M.Badia, R. and Labarta, J.: CellSs: a Programing Model

for the Cell BE Architecture (2006).