卒業研究論文

解像度非依存型動画像処理ライブラリ RaVioliのCell/B.E.を用いた高速化

指導教員　　松尾 啓志 教授
津邑 公暁 准教授

名古屋工業大学 工学部 情報工学科
平成 18年度入学　18115020 番

稲葉 崇文

平成22 年 2月 8 日

解像度非依存型動画像処理ライブラリ RaVioli の Cell/B.E. を用いた高速化

稲葉 崇文

内容梗概

近年，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の開発が盛んに行われている．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となった．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 秒毎に処理を行う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 フレームあたりの処理量の変動や，他のプロセスによる使用可能な CPU リソース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 では処理対象画像の解像度を CPU 使用率や並行実行プロセスによる負荷に応じて，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，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 フレームあたりの幅および高さを隠蔽し，解像度をライブラリ内で制御している．こう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が可能となり，より直感的な動画像処理プログラミングが実現できる．

しかしながら，RaVioli は画素情報を隠蔽することによって抽象的な記述を可能としているが，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．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．そこで本研究では，RaVioli の抱える問題点を解決するため，マルチメディア処理を得意とする CPU である Cell/B.E. に着目し，より高速な画像処理を実現する手法を提案し，実装した．

サンプルプログラムを用いて評価を行った．従来の RaVioli を用いて汎用 CPU 上で画像処理を行った場合と，提案手法によって拡張した RaVioli を用いて Cell/B.E. 上で画像処理を行った場合の実行時間を比較し，後者の実行時間が前者の実行時間よりも短く高速に処理できていることを確認する．

解像度非依存型動画像処理ライブラリ RaVioli の Cell/B.E. を用いた高速化

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 . . . 4
  - 2.1.4 RaVioli の問題点 . . . 6
- 2.2 Cell/B.E. . . . 7
  - 2.2.1 Cell/B.E. について . . . 7
  - 2.2.2 Cell プログラミングの問題点 . . . 8

3 提案ライブラリ 9
- 3.1 特徴 . . . 9
- 3.2 仕様 . . . 10
  - 3.2.1 RaVioli の拡張 . . . 10
  - 3.2.2 ユーザインタフェース . . . 11

4 実装 12
- 4.1 高階メソッドの拡張 . . . 12
  - 4.1.1 PPE の処理 . . . 13
  - 4.1.2 SPE の処理 . . . 17
- 4.2 トランスレータによる変換規則 . . . 19
  - 4.2.1 トランスレータの必要性 . . . 19
  - 4.2.2 PPE プログラム . . . 19
  - 4.2.3 SPE プログラム . . . 21

5 評価 23

6 おわりに 24

謝辞 25

1　はじめに

近年，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の開発が盛んに行われている．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である．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 秒毎に処理を行う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 フレームあたりの処理量の変動や，他のプロセスによる使用可能な CPU リソース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 では処理対象画像の解像度を CPU 使用率や並行実行プロセスによる負荷に応じて，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，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 フレームあたりの幅および高さを隠蔽し，解像度をライブラリ内で制御している．こう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が可能となり，より直感的な動画像処理プログラミングが実現できる．

しかしながら，RaVioli は画素情報を隠蔽することによって抽象的な記述を可能としているが，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．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．

そこで本研究では，RaVioli の抱える問題点を解決するため，マルチメディア処理を得意とする CPU である Cell/B.E. に着目し，より高速な画像処理を実現する手法を提案する．

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 Cell/B.E. について述べ，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る．次に 5 章で提案の評価とそれに対する考察を述べる．最後に 6 章で本論文全体をまとめる．

2　研究背景

2.1　RaVioli

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，エッジ抽出を行い，その結果に対しハフ変換を行う．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使用可能な CPU リソースの変動がおこるため，リアルタイム動画像処理に必要な CPU リソースが常に確保可能だという保証はない．

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，解像度を自動調節することで処理量を減らしリアルタイム性の保証を行う．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 フレーム中の高さと幅の画素数やフレームレートを隠蔽し，空間解像度および時間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 では動画像の構成要素である画素またはフレームに対する処理を定義した関数（この関数を構成要素関数と呼ぶ）のみを記述し，それをライブラリで提供しているメソッド（このメソッドを高階メソッドと呼ぶ）に渡すことで，動画像中の全ての構成要素に対して処理を施すことが可能である．たとえば，カラー画像をグレースケール画像へ変換する処理は，**図 1** のように記述される．ユーザは対象画素をグレースケール化する関数 GrayScale を定義し，すべての構成画素に処理を適用するメソッド 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，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．さらに，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，動画像処理中の繰り返し単位が明確となりデータ並列性の抽出が容易となる．

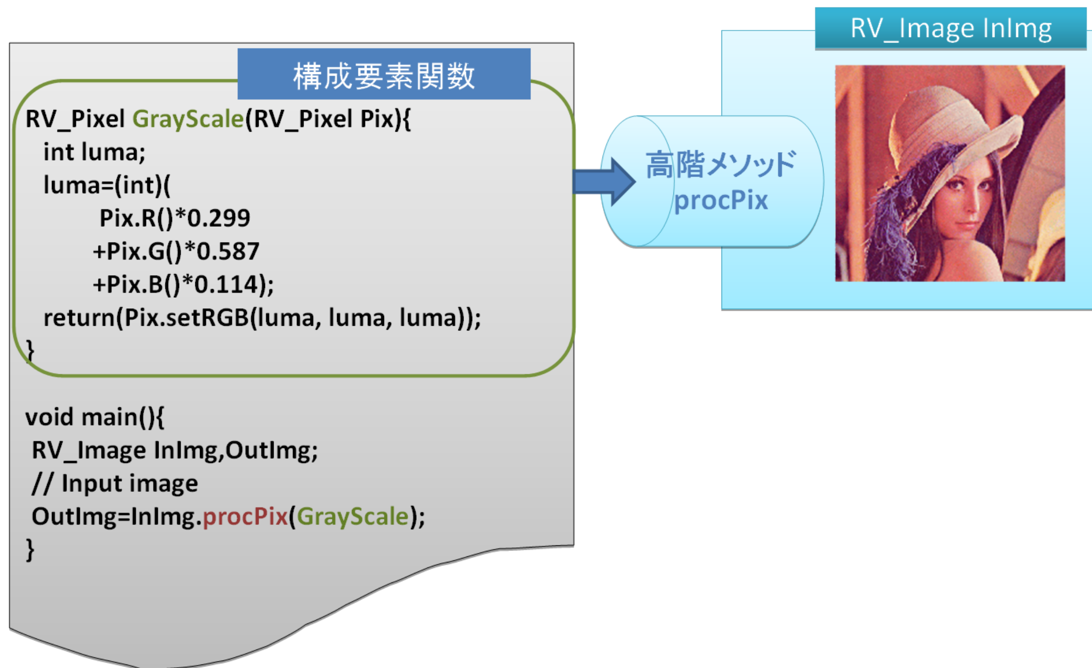

図 1: 高階メソッド呼び出し

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，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コードの違いを**図 2** に示す．図 2 はグレースケール化の処理を行うプログラムの例である．図中の左側に示しているプログラムは，RaVioli を用いていないプログラムであり，右側に示しているプログラムは，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，各画素に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いた．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する．図 2 では，画素 `InImg[x][y]` に対して処理を実行した後，画素 `InImg[x + 1][y]` に対して処理を実行することになる．この

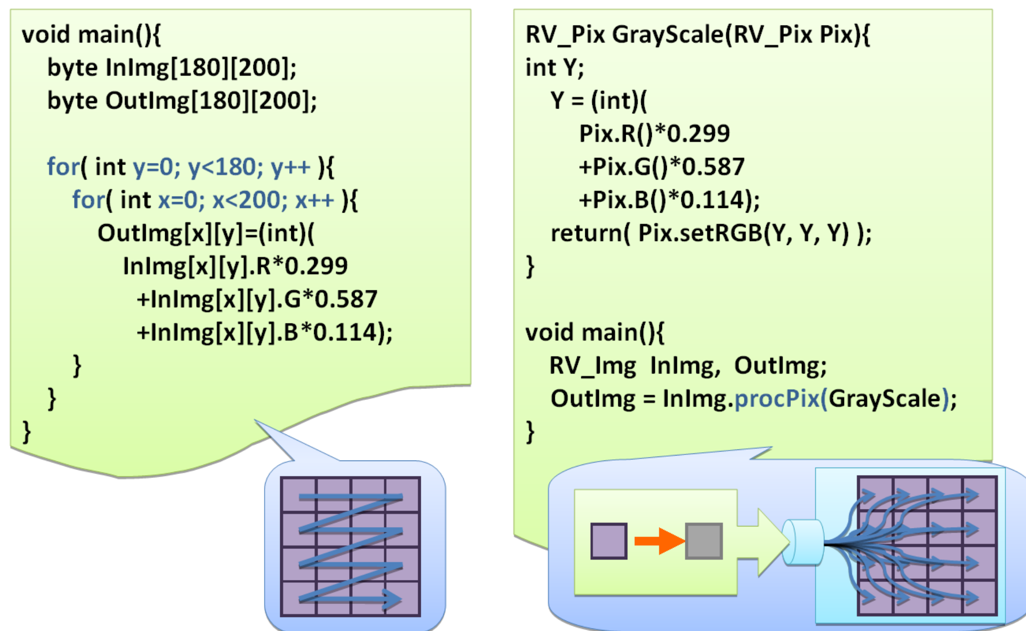

図 2: RaVioli の実行モデル

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．ループを記述することによって，本来存在しないはずの処理順依存が存在するかのように見えてしまうのである．

一方で，RaVioli を用いたプログラムでは，各画素への処理の適用はライブラリ側で行われる．RaVioli を用いたプログラミングにおいてループが存在する箇所があるとすれば，それは高階メソッド，すなわちライブラリ内である．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，各画素への処理の適用をライブラリ側で行うため，処理順序を与える過程は存在しない．これによって，各画素への処理を全順序化してしまうようなことはなくなり，また，処理順依存が存在しないことから，構成要素関数を各画素に対して並列に適用できることは明らかであるため，自動並列化が容易であると言える．

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

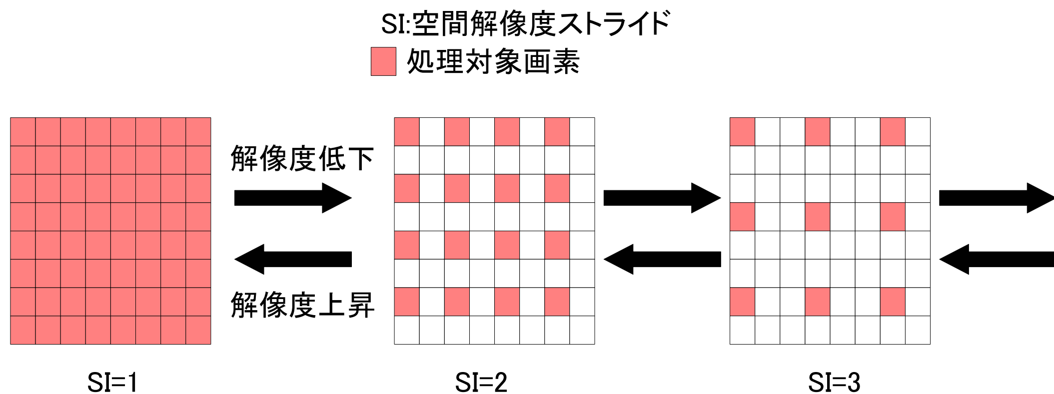

図3: 空間解像度ストライドの変動

動画像処理を行うことは困難である．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る．一方，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低減を実現している．

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，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．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

空間解像度が低下した場合の例を**図3**に示す．空間解像度とは1 フレームにおける画素数である．RaVioli で空間解像度の変更を行う場合，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空間解像度の変更を行っている．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の場合，空間解像度ストライド $SI = 1$ のとき，画像中の全ての画素を処理する．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，

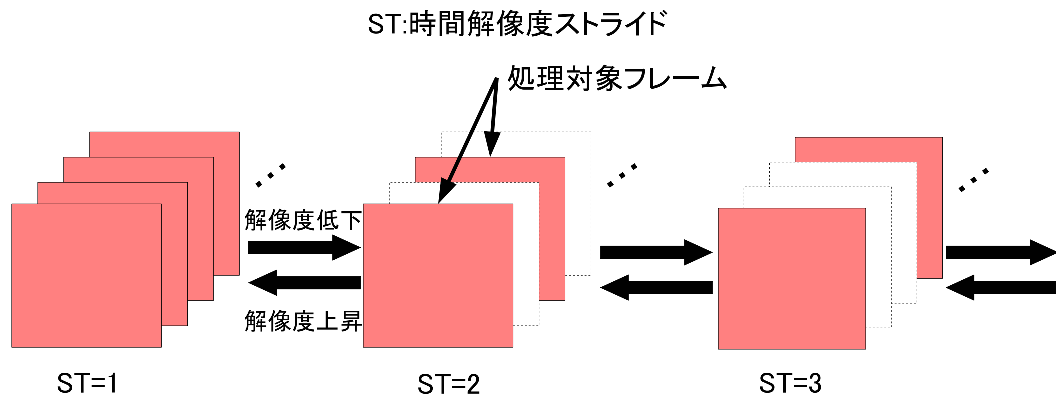

図 4: 時間解像度ストライドの変動

空間解像度の低下が発生し，空間解像度ストライドが $SI = 2$ に増加する．空間解像度ストライドが $SI = 2$ 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 $SI = 1$ のときの $1/4$ となる．同様に，さらに空間解像度が低下し $SI = 3$ 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

時間解像度が低下した場合の例を**図 4** に示す．時間解像度とはフレームレートである．RaVioli で時間解像度の変更を行う場合，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時間解像度の変更を行っている．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 の場合，時間解像度ストライド $ST = 1$ のとき，全てのフレームを処理する．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，時間解像度の低下が発生し，時間解像度ストライドが $ST = 2$ に増加する．時間解像度ストライドが $ST = 2$ 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 $ST = 1$ 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 $ST = 3$ に増加した場合，フレームレートは 1/3 となる．

2.1.4　RaVioli の問題点

2.1.3 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ってくる．そこでこの問題に対する RaVioli の解決策として 2.1.1 項で述べたような手法が実装されている．RaVioli を用いた画像処理プログラムの例を**図 5** に示す．

```
void main{
 // Input image
 int luma;
 for(int y=0;y<180;y++){
   for(int x=0;x<200;x++){
     luma = (int)(
            InImg[x][y].R*0.299
           +InImg[x][y].G*0.587
           +InImg[x][y].B*0.114);

     OutImg[x][y].R = luma;
     OutImg[x][y].G = luma;
     OutImg[x][y].B = luma;
   }
 }
}
```

```
RV_Pixel GrayScale(RV_Pixel Pix){
  int luma;
  luma=(int)(
        Pix.R()*0.299
       +Pix.G()*0.587
       +Pix.B()*0.114);
  return(Pix.setRGB(luma, luma, luma));
}

void main(){
 RV_Image InImg,OutImg;
 // Input image
 OutImg=InImg.procPix(GrayScale);
}
```

図 5: RaVioli を用いて書き換えた画像処理

図 5 で左側に示されているプログラムは RaVioli を用いていない画像処理プログラムの例であり，右側に示されているプログラムは RaVioli を用いた画像処理プログラムの例である．図 5 における書き換え前の画像処理プログラムでは，画像の幅と高さを条件式に用いたループを記述することで，画像中の各画素に対して処理を適用している．一方で，RaVioli を用いて記述した画像処理プログラムでは，プログラム中にループが現れず，画像の幅や高さを考慮したプログラムを記述する必要はない．各画素への処理はライブラリ側で行われるため，プログラマは各画素に対して適用したい処理のみを記述すれば良い．

このように，RaVioli はプログラマから解像度を隠蔽し，より直感的なプログラミングが可能な環境を提供している．しかし，この手法を用いた場合，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，マルチメディア処理に適した CPU に Cell/B.E. がある．次節以降では，Cell/B.E. について触れる．

2.2 Cell/B.E.

2.2.1 Cell/B.E. について

Cell/B.E. [3] は，SONY，東芝，IBM の 3 社により共同開発された，高い処理性能を目指したマルチコア SIMD プロセッサである．Cell/B.E. は， 1 つの汎用プロセッ

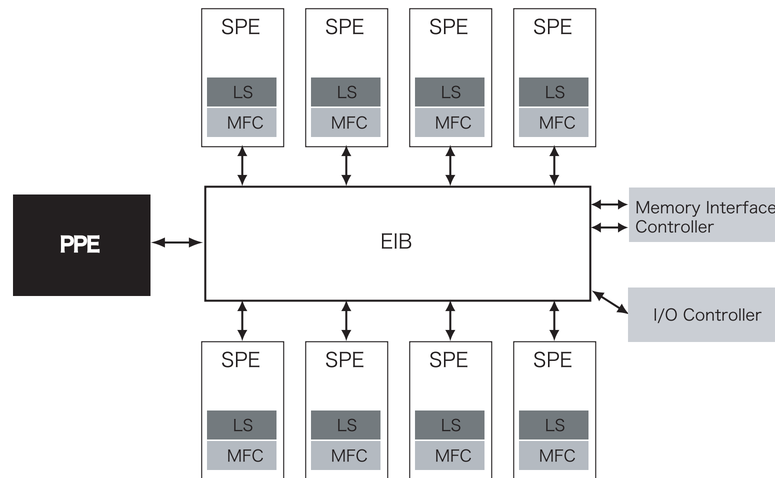

図 6: Cell/B.E. アーキテクチャ

サ PPE (PowerPC Processor Element) と 8 つの演算プロセッサ 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．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 つのコアをあわせた浮動小数点演算能力は最大時で 200GFLOPS を超える．Cell/B.E. アーキテクチャの概略を**図 6** 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．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を持つ．メインメモリへのアクセスは LS を介してのみ行う．また，SPU（Synergistic Processor Unit）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ず，MFC（Memory Flow Controller）と呼ばれるユニットを利用する必要がある．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大きいため，メモリレイテンシを隠蔽する手法として，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メインメモリのデータを転送している裏で，もう片方のバッファに対して計算を行うという方法である．

2.2.2　Cell プログラミングの問題点

Cell/B.E. は，その性能を最大限に発揮させることができれば，高度な画像処理を高速に実行可能になる．しかしながら、Cell/B.E. 向けプログラムを記述する際には様々な制約があり，以下に示すような点に注意を払う必要がある．

(1)　Cell/B.E. の特徴を活かしたプログラムの開発にはまず，Cell/B.E. に搭載され

ている，性質が異なる 2 種類のコア (PPU とSPU) で動作するプログラム (PPE プログラムと SPE プログラム) をそれぞれのプロセッサに対して用意する必要がある．そして，それぞれのプログラムが協調するような設計にする必要があるため，並列分散プログラミングの技術が必要となる．

(2) 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，アーキテクチャの詳細を理解する必要がある．これはDMA（Direct Memory Access） 転送と呼ばれる Cell/B.E. で用いられるデータ転送方式の理解や，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を理解する必要があるためである．特に，DMA 転送では，一度に転送できるデータサイズ等に制限があり，この制限を意識したプログラミングが必要となる．

(3) SPE は特に，SPU SIMD 命令等の組み込み関数を用いることで，同じような計算を単純に繰り返すようなマルチメディア系の処理を得意としている．しかしながら，コンパイラなどによる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ようなプログラムの箇所を抽出することはまだ困難である．そのため，SPE の演算性能を引き出すようなプログラミング手法を学習する必要があり，プログラマにとって，Cell/B.E. を用いたプログラミングの技術的障壁はかなり高いと言える．

以上のように，Cell/B.E. の性能を引き出し高速なマルチメディア処理を記述するためには，動画像処理の本質とは異なる部分でプログラマの負担となることがわかる．そこで本研究では，従来の RaVioli を用いたプログラミング手法を理解するだけで Cell/B.E. の性能を引き出し，高速な画像処理プログラミングを支援する手法を提案する．

3 提案ライブラリ

3.1 特徴

2 章で述べたように，RaVioli と Cell/B.E. の使用には問題がある．RaVioli は，抽象化のオーバー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，実行速度の解決手段として Cell/B.E. を用いようとすれば，複雑なプログラミング手法を身に付ける必要がある．

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持つ問題を解決する．

本研究で提案する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

動画像処理を記述するだけで良い．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．

従来の RaVioli のみを用いて，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()` 関数で SPE プログラムを起動し，構成要素関数である `Color2Gray()` で DMA 転送を行っている．`proc()` は RaVioli が提供する高階メソッドである．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 転送を行う都合上，一度の DMA 転送で取得しようとするデータは一画素分のみである．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るため，これが頻繁に行われることは好ましくない．このように，従来の RaVioli を用いて Cell/B.E. を活用することは困難である．

一方で，提案ライブラリを用いたプログラムでは，SPE 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 `proc()` 内部で行われる．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 種類のコアを持つ．また，それぞれのコアには得意とする処理がある．PPE は，オペレーティング・システムのような頻繁なスレッド切り替えなどを得意とする大型の制御系コアである．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を可能とする．

本研究では，PPE に SPE の初期化と画像の読み出しを担当させ，SPE に画像に対する処理を担当させる．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の読み出しが行われる．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．各 SPE プログラムもまた，高階メソッド呼び出しを行う．SPE プログラムが呼び

```
main(){
   prog = spe_image_open("program_name");
   spe = spe_context_create(0, NULL);
   spe_program_load(spe, prog);
   spe_context_run(spe, InImg);
   spe_context_destroy(spe);
   spe_image_close(prog);
}
```

```
proc() {
   InImg, OutImg;
   for () {
      OutImg = c2g(InImg);
   }
}
```

```
Color2Gray() {
   spu_mfcdma64(pixel, GET);
   RGB = pixel->getRGB();
   /*グレースケール化*/
   pixel->setRGB(RGB);
   spu_mfcdma64(pixel, PUT);
}
```

図 7: 提案ライブラリ未使用の例

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やり取りと，プログラマの記述した構成要素関数の適用を行う．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 プログラムは終了し，画像処理が完了する．

Cell/B.E. を用いた基本的な画像処理の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 プログラムのためのものと SPE プログラムのためのものの 2 種類に拡張した．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いるためである．しかし，この仕様を意識したプログラミングを行うことは，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

本研究で拡張した RaVioli には，PPE プログラム向けの高階メソッドを持つものと，SPE プログラム向けの高階メソッドを持つものの 2 種類が存在する．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ように変更されているため，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

けプログラムを生成する場合は，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．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が，従来の RaVioli を用い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 Cell/B.E. 向けプログラムとしてコンパイル，実行することはできないのである．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変換を行う．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．

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．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，PPE プログラムと SPE プログラムを得る．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．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，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行われる．**図 8** に，従来の RaVioli の高階メソッド呼び出しを，**図 9** に提案ライブラリの高階メソッド呼び出しの例を示す．この例では，従来の RaVioli を用いたプログラムの 4 行目において，構成要素関数 GrayScale を引数としていた高階メソッド呼び出しを，SPE プログラム名である GrayScale.elf を引数とした高階メソッド呼び出しに変換している．このように，提案ライブラリとトランスレータを用いることで，従来のプログラムに対してプログラマ自身が書き換えを施すことなく，提案ライブラリを用いたプログラムを生成することができる．これは PPE プログラムを生成する単純な例であり，トランスレータの動作については 4.2 節で詳しく述べることとする．

4　実装

4.1　高階メソッドの拡張

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. 上で効率的な画像処理を行う手法を提案した．提案ライブラ

GrayScale.cpp
```
1   int main() {
2     RV_Image img;
3     readBMP(img);
4     img.proc(Color2Gray);
5   }
```

図 8: 従来の高階メソッド呼び出しの例

GrayScale_ppe.cpp
```
1   int main() {
2     RV_Image img;
3     readBMP(img);
4     img.proc("./Color2Gray.elf");
5   }
```

図 9: 提案する高階メソッド呼び出しの例

リにおいて，従来の RaVioli の処理のうち，用途に応じたデバイスから画像を読み出し，高階メソッドを呼び出していた部分は PPE プログラムの担当範囲となり，画像の構成画素に処理を適用するという高階メソッドが行っていた処理はSPE プログラムの担当範囲となる．本節では，こうした処理の分担を実現するための実装について述べる．

4.1.1 PPE の処理

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 プログラム向け高階メソッドの動作を**図 10** に示す．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などが行われる．一般的な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 11** 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（7 行目）する．そして `spe_context_create()` 関数を用いて SPE コンテキストを生成（8 行目）し，`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（9 行目）する．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（11，12 行目）される．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（13 行目）する．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・イメージをクローズ（14 行目）する．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

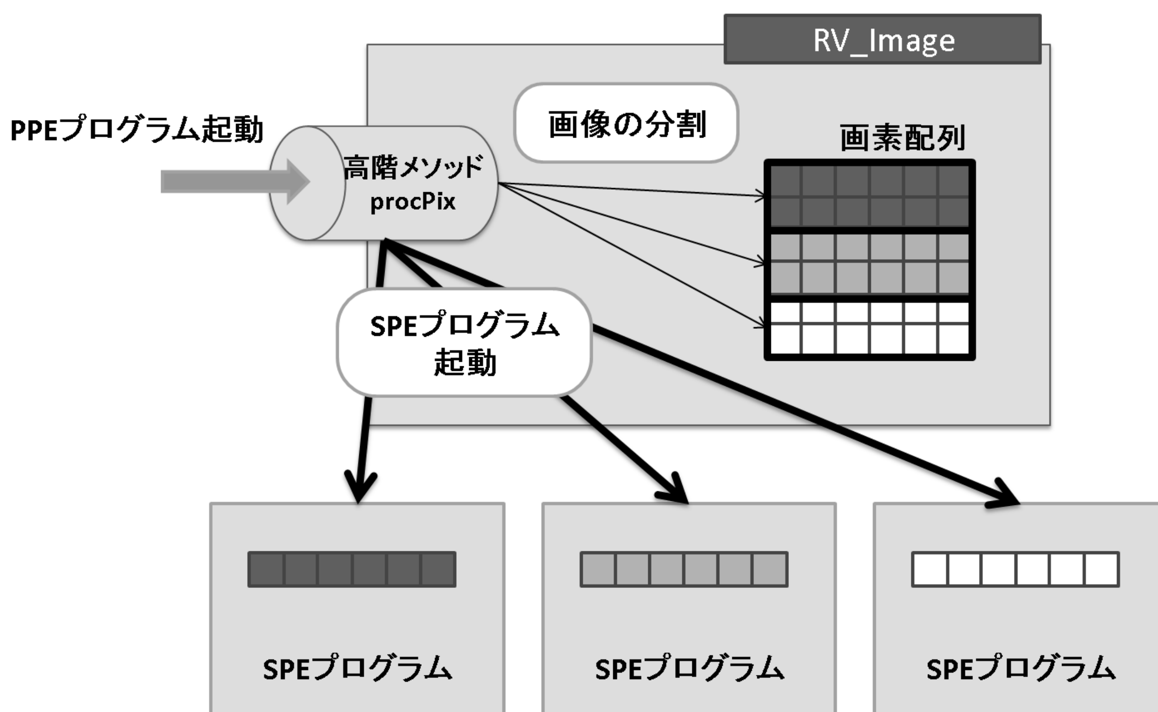

図 10: PPE 向け高階メソッドの動作

動的に行う．図 10 は，高階メソッドが 3 つの SPE プログラムを起動している様子を示している．図のように，SPE プログラムを起動しているのは PPE プログラム向けライブラリの高階メソッドであり，プログラマが初期化処理を行う必要はない．

また，SPE は複数使用されることが普通であるが，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 10 の例では，3 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 3 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，割り当てるサイズを計算する．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

```
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     prog = spe_image_open("spe_main.elf");
8     spe = spe_context_create(0, NULL);
9     spe_program_load(spe, prog);
10    entry = SPE_DEFAULT_ENTRY;
11    spe_context_run(spe, &entry, 0,
12     NULL, NULL, &stop_info);
13    spe_context_destroy(spe);
14    spe_image_close(prog);
15
16    return (0);
17  }
```

図 11: SPE プログラムの実行

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，先程計算された割り当てサイズを，16 の倍数バイトになるように再計算する．サイズ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．

ここで，一部の SPE については割り当てサイズ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 G バイト，$1/N$ の画像サイズを g バイトとすると，分割された中でも $N-1$ 枚の画像は全て g 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，残りの 1 枚については，$G-(N-1)g$ 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

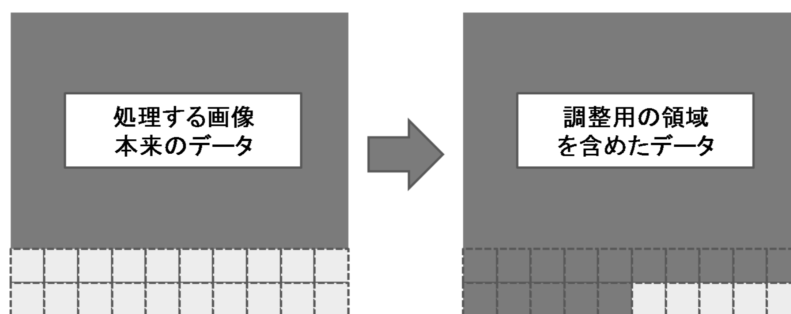

図 12: 領域確保

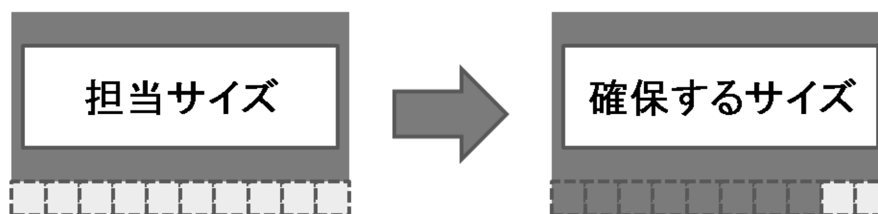

図 13: 領域調整

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，$G-(N-1)g$ バイトのデータを 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 12**，**図 13** に示す．図 12 に示すように，画像を読み出す際，左側のように G バイトの領域を確保するのではなく，右側のように調整用の領域を含めた $G+15$ バイトの領域を確保する．これは，図 13 に示すように，$G-(N-1)g$ バイトのデータに余分なデータを付加し，16 の倍数バイトのデータを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ため，領域の確保はこのような流れになる．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$16N$ バイト程度であるため，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 を用いて画像処理を行うには，以上のような点に注意を払う必要があるが，これらの処理は全て高階メソッド内部で自動的に行われる．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した．図 10 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の呼び出しを行うのみであり，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．

4.1.2 SPE の処理

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．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行う．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 プログラム向けに提供された高階メソッドを呼び出す．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．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データを取得し始めれば良い．画像データの転送には DMA 転送を用いる．一度の DMA 転送では 16KB のデータしかやり取りすることができないため，全画素を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．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かかるオーバヘッドの削減を行う．各 SPE は LS にバッファを二つ持ち，一方のバッファに対して転送処理を行っている間に，もう一方のバッファにあるデータを処理する．

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 14** に示す．この例では，最上段の部分の画像を担当している 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像を必要とする．同様に，中段の部分の画像を担当している SPE2 は，SPE1 が処理を担当する最上段の部分の画像と，SPE3 が処理を担当する最下段の部分の画像を必要とする．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15** のようにして，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15の最上段の部分の画像を担当しているSPE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左側の図のように取得する．DMA 転送の制約により，16の倍数バイトに揃っていない右側の図の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15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．

ここでは近傍処理を例として挙げたが，自身の担当範囲を越えた領域の取得は，テンプレートマッチングのような処理でも必要となる．このように，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に記述することで，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．プログラマから構成要素関数を受け取った高階メソッドの内部では，自動的にDMA 転送が制御され，画像処理が行われる．**図16**，**図17** にDMA 転送挿入前後の，高階メソッド内部の構成要素関数呼び出し部分を示す．プログラマが記述すべき処理である構成要素関数の定義は，これらの図には現れてはおらず，プログラマの負担となる処理が提案ライブラリ内部に隠蔽されたことがわかる．また，プログラマが構成要素関数にDMA 転送を記述した場合，各画素に対してDMA 転送が行われることになり，膨大なオーバヘッドが発生してしまうが，ライブラリ側でDMA 転送を制御することによって，こうした問題も解決している．

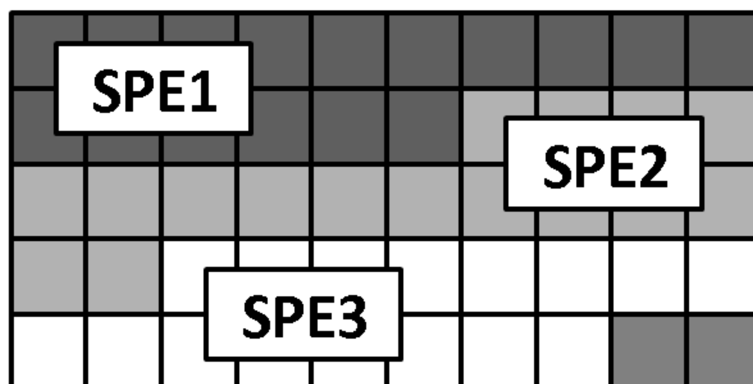

図 14: 分割画像の境界

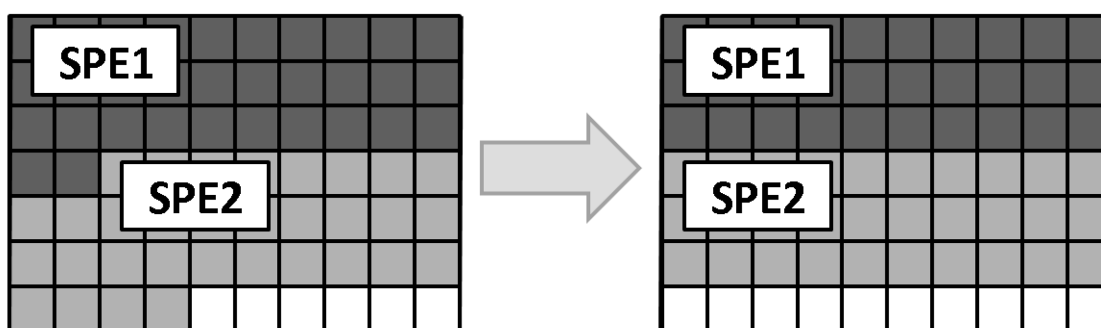

図 15: 境界の調整

4.2　トランスレータによる変換規則

4.2.1　トランスレータの必要性

従来の RaVioli を用いたプログラムを提案ライブラリを用いた Cell/B.E. 向けプログラムに書き換える負担の軽減のため，本研究ではトランスレータの実装についても検討する．まず，トランスレータの必要性の確認のため，従来の RaVioli を用いて記述されたプログラムを**図 18** に示し，提案ライブラリを用いた画像処理プログラムの例を**図 19**，**図 20** に示す．図 18 のプログラムを図 19 と図 20 のプログラムのように変換しなければ，提案ライブラリは使用できない．これはプログラマの負担となるため，トランスレータの実装は有用であると考えられる．以下では，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する．

4.2.2　PPE プログラム

従来の RaVioli を用いたプログラムから，構成要素関数以外の部分を切り出し，PPE プログラムとする．構成要素関数以外の部分とは，`main()` 関数及び，`main()` 関数が

```
void proc(UserProgram) {
  for (i = 0 to width) {
    for (j = 0 to height) {
      UserProgram(image[i][j]);
    }
  }
}
```

図 16: DMA 転送挿入前の高階メソッド

```
void proc(UserProgram) {
  while (image_size) {
    spu_mfcdma64(image, GET);
    for (i = 0 to size) {
      UserProgram(image[i]);
    }
    spu_mfcdma64(image, PUT);
  }
}
```

図 17: DMA 転送挿入後の高階メソッド

直接呼び出す関数が該当すると考えられる．図 18 の例では，`print_before()` 関数，`print_after()` 関数，`main()` 関数が該当する．高階メソッドの引数として与えられている `GrayScale()` 関数と，`GrayScale()` 関数から呼び出されている `Color2Gray()` 関数は，構成要素関数であると考えられるため，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，今回の例では文字列を表示しているのみであるが，`print_before()` 関数や `print_after()` 関数は，実際の画像処理プログラムでは初期化等に関わっている関数が該当すると考えられる．以上のような判断基準によって，従来の RaVioli を用いた画像処理プログラムから，PPE プログラムを切り出すことができる．

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，4.1.1 項で述べたように，SPE プログラムを起動する必要がある．SPE プログラムを起動するためには，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生成された PPE プログラムが，図 19 に示すプログラムである．

構成要素関数は SPE プログラムの一部であるために不要となり，高階メソッド呼び

GrayScale.cpp

```
void print_before() {
  puts("proc before");
}
int Color2Gray(int RGB) {
  return(/*グレースケール化*/);
}
void GrayScale(RV_Pixel *p) {
    int RGB = p->getRGB();
    RGB = Color2Gray(RGB);
    p->setRGB(RGB);
}
void print_after() {
  puts("proc after");
}
int main(int argc, char *argv[]) {
  RV_Image img;
  readBMP(img);
  print_before();
  img.proc(GrayScale);
  print_after();
}
```

図18: 従来の RaVioli を用いたプログラム

出しではプログラムファイル名を引数とする呼び出しに変換された．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．

4.2.3 SPE プログラム

従来の RaVioli を用いたプログラムから，構成要素関数部分を切り出し，SPE プログラムとする．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，SPE プログラムの一部として記述する必要があるためである．構成要素関数は，`proc()` 等

GrayScale_ppe.cpp

```
void print_before() {
  puts("proc before");
}
void print_after() {
  puts("proc after");
}
int main(int argc, char *argv[]) {
  RV_Image img;
  readBMP(img);
  print_before();
  img.proc("./GrayScale_spe.elf");
  print_after();
}
```

図 19: 提案ライブラリを用いた PPE プログラム

の高階メソッドの引数として与えられている関数であり，また，その関数から呼び出されている関数を含む．図 18 の例では，`Color2Gray()` 関数と `GrayScale()` 関数が該当する．前者が構成要素関数から呼び出されている関数であり，後者が構成要素関数である．残りの関数は全て，PPE プログラムが担当する処理に関わる関数であると考えられ，また 4.2.2 節で述べた判断基準によって切り出されたプログラムに一致する．よって，以上の判断基準にしたがってプログラムの切り出しを行えば良いと考えられる．

SPE プログラムとして切り出したプログラムは，このままでは実行できない．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．この `main()` 関数は，PPE プログラムから起動されると， 4.1.2 項で述べた処理を行う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にもまた，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では，メインメモリに格納されたデータの取得のため，PPE プログラムから格納先アドレスや，格納されているデータのサイズなどを受け取る必要がある．これは，PPE プログラムで呼び出される `spe_context_run()`

GrayScale_spe.cpp

```
int Color2Gray(int rgb) {
  return (/*グレースケール化*/);
}
void GrayScale(RV_Pixel *p) {
    int RGB = p->getRGB();
    RGB = Color2Gray(RGB);
    p->setRGB(RGB);
}
int main(unsigned long long spe,
         unsigned long long argp,
         unsigned long long envp) {
    RV_Image img;
    img.proc(GrayScale, argp);
}
```

図 20: 提案ライブラリを用いた SPE プログラム

関数の引数を通じて SPE プログラムの main() 関数が持つ引数の一つとして与えられるため，ここで受け取った値をそのまま高階メソッドに受け渡す．また，従来の RaVioli と同様，構成要素関数へのポインタが必要である．以上の変換を施した結果生成された SPE プログラムが，図 20 に示すプログラムである．

main() 関数が SPE プログラム向けに変換され，全体としては，構成要素関数の定義を含む SPE プログラムへと変換された．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．

5　評価

表 1 に示す評価環境で評価を行った．評価のために用いる画像として，2.5MB 程度のものと 5MB 程度のものを用意した．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を用いた．ただし，グレースケール化プログラムは，画像処理としては非常に処理が軽いため，グレースケール化の処理を各画素に対して 1000 回ずつ行うことで，処理が重い画像処理を擬似的に再現することとした．

表1: 評価環境

	評価環境1	評価環境2
CPU	Athlon 3500+	Cell/B.E.
動作周波数	2.2GHz	3.2GHz(PPE)
OS	CentOS 5.4	Fedora 10

以上の条件で行った評価結果を**図21**に示す．グラフの左端のものが，表1の評価環境1に示す汎用CPU上で実行した場合の実行結果である．以降はCell/B.E.を用いた実行結果であり，左から2番目のグラフがPPEのみを用いて実行した場合の結果，続いて，SPEを1，2，…，6基使用した場合の実行結果となっている．提案ライブラリを使用しているのは，SPEを用いて実行したもののみであり，汎用CPUを用いたものとPPEのみを用いたものでは，従来のRaVioliを使用している．

グラフから，SPEを5基以上使用することによって，従来のRaVioliを上回る実行速度が得られたことが確認できる．また，複数のSPEプログラムを起動するためにはスレッドを生成する必要があり，プログラム終了時には同期を取る必要があるため，使用するSPEを増やすほど，SPEプログラムを起動するオーバヘッドも大きくなると考えられるが，SPEを6基使用した場合が最も高速であることから，そのオーバヘッドを上回る性能を得られていることもわかる．しかし，汎用CPU上で従来のRaVioliを使用した場合の実行結果と，SPEを6基使用した場合の実行結果の差は僅かであり，提案するライブラリには課題が残っていると言える．

一方で，今回使用したサンプルプログラムでは，構成要素関数内部に大きなループが存在している点も考慮すべきであると考えられる．SPEは，その役割故に，分岐予測等の複雑な回路を持たない.すなわち，回数の多い，大きなループが存在していること自体が，SPEプログラムにとって不利である可能性がある．また，現時点では，SPEプログラムで行われる演算がSIMD演算ではなくスカラ演算であるため，SPEが本来の性能を発揮しているとは言い難い. SIMD演算への対応は今後の課題である．

6　おわりに

本論文ではまず，動的に使用可能なリソースが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の保証を行う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,マルチメディア処理に向いて

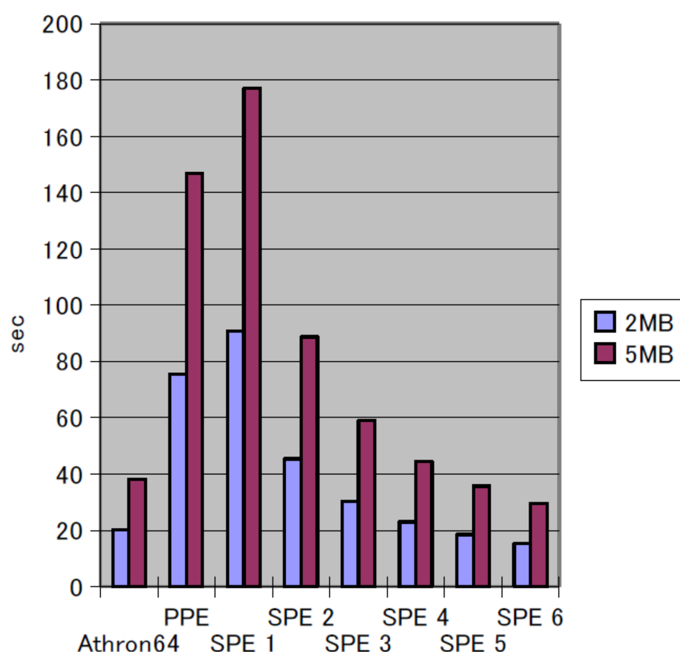

図 21: 評価

いる CPU として 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について提案し，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について考察した．今後の課題として，提案ライブラリの SIMD 命令への対応が挙げられる．Cell/B.E. を用いた画像処理で性能を最大限に発揮するためには，SIMD 演算への対応は必須であると言える．また，現段階では実装案のみとなっているトランスレータの実装もまた課題の一つである．既存のプログラムが Cell/B.E. 上で動作可能となれば，より高度な画像処理を，より高い解像度で実現可能になる．

謝辞

本研究のために，多大な御尽力を頂き，御指導を賜わった名古屋工業大学の松尾啓志教授，津邑公暁准教授，齋藤彰一准教授，松井俊浩助教に深く感謝致します．また，本研究の際に多くの助言，協力をして頂いた松尾・津邑研究室およびの齋藤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桜井寛子氏，大

野将臣氏，近藤勝彦氏，今井満寿巳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).