修士論文

解像度非依存型動画像処理ライブラリRaVioliのCUDA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 20 年度入学　20413531 番

櫻井 寛子

平成 22 年 2 月 4 日

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

櫻井 寛子

内容梗概

近年，侵入者検知システムなどリアルタイム性が重要なシステムの開発が盛んに行われている．また，ビデオカメラなどの入力装置からリアルタイムに画像をキャプチャ可能な環境が整ってきたことや，汎用計算機の高性能化により，高度な画像処理が実行可能となってきた．そのため汎用システム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，汎用システムでは並行実行プロセスなどの外乱により，リアルタイム動画像処理に必要な CPU リソースの確保が困難である．

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 が提案されている．RaVioli では確保可能な CPU リソースの減少によりリアルタイム動画像処理が困難になった場合，解像度を変動させることで処理量を調整する．こうすることで，擬似的なリアルタイム性を保証している．しかし，抽象化のオーバーヘッドにより処理速度が低下し，また処理精度が悪化してしまうという問題点がある．

一方で GPU 向けの C 言語統合開発環境として CUDA が開発されている．CUDA は GPU 上に存在するマルチプロセッサ内の複数のコアに対して，大量のスレッドを並列に実行させることでプログラムの高速化を実現する．また GPU 内に存在する複数のコアは同時に同じ命令を実行する SIMD 型を採用している．そのため，一般的にデータ並列性を持つ動画像処理は GPU 上で高い性能が期待できる．そこで本研究では，CUDA を用いることで RaVioli の処理速度の向上を目指す．

CUDA は C 言語に似た構文であり，C 言語を使用したことのあるプログラマは容易に習得可能である．しかしプログラマは，CPU-GPU 間のデータ転送やスレッドの管理を意識してプログラムを記述する必要がある．また多くの最適化を施す必要があるため，CUDA を用いた動画像処理アプリケーションの開発はプログラマにとって負担が大きい．そこで本研究では，これらを意識せずに CUDA を使用した動画像処理プログラムが記述可能なように，RaVioli を拡張する．また従来の RaVioli で記述されたプログラムを，拡張後の RaVioli で記述されたプログラムへと変換を行うトランスレータも提案する．

拡張後の RaVioli を使用して記述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 から最大約 164 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 CUDA を用いた高速化

目次

1 はじめに **1**

2 背景 **2**

2.1 動画像処理 2

2.2 RaVioli 3

2.2.1 処理量の自動調整 3

2.2.2 動画像処理の抽象化 4

2.2.3 RaVioli の問題点 7

2.3 CUDA 8

2.3.1 プログラミングモデル 9

2.3.2 メモリモデル 10

2.3.3 CUDA の問題点 13

3 RaVioli+CUDA **15**

3.1 実行構成 16

3.2 画像処理プログラム 18

3.2.1 処理単位がウィンドウの場合の画像処理プログラム 22

3.3 動画像処理プログラム 25

3.3.1 従来の RaVioli 記法を用いた動画像処理プログラム 25

3.3.2 提案記法を用いた動画像処理プログラム 26

3.3.3 オーバーラップ 28

3.4 ライブラリの仕様 29

4 トランスレータ **34**

4.1 トランスレータの方針 34

4.2 構成要素関数から Kernel 関数への変換の基本方針 35

4.3 リダクション処理の生成を含む変換 39

4.3.1 ループをまたがる依存関係の解析手法 39

4.3.2 変換手法 40

5 評価 **45**

5.1 画像処理の速度比較 46
5.2 動画像処理の速度比較 47
5.3 トランスレータの適用範囲 49

6 関連研究 **49**
6.1 画像処理の抽象化 49
6.2 処理時間の自動調整 50
6.3 CUDA 50

7 おわりに **51**

謝辞 **52**

参考文献 **53**

1　はじめに

近年，空港や工場などの侵入者検知システムや，自動車走行中の前方車両もしくは障害物の認識による衝突回避システムなど，リアルタイム性を重要視した動画像処理システムの開発が盛んに行われている．一方で計算機の高性能化に伴ない，顔認識アルゴリズム等の処理量の多い動画像処理を汎用 PC 上で行うことが可能となってきた．そのため今後，比較的安価な汎用 PC 上においても，このようなリアルタイム動画像処理アプリケーションの必要性がますます多くなると考えられる．

しかし Linux に代表される汎用 OS 上で，動画像処理アプリケーションのリアルタイム性を保証することは困難である．その理由として，複数プロセスの並行実行による使用可能な CPU リソースの変動があげられる．またフレーム毎の処理量が異なる動画像処理も実在するため，1/30 または 1/60 秒毎に 1 フレームの処理を可能とする CPU リソースを常に確保することは困難である．そこで Linux をリアルタイム OS に拡張するプロジェクトも存在する．しかし Linux は元来，リアルタイム処理であってもカーネル実行中は割り込みができない非リアルタイム OS である．そのため，リアルタイム性を 100%保証できるわけではない．

そこで汎用 OS 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 では使用可能な CPU リソースに応じて，空間解像度（1 フレーム上の画素数）または時間解像度（フレームレート）を変動させることで疑似リアルタイム動画像処理を実現する．

このように動的に解像度を変動させる場合，処理フレームや処理画素にアクセスする際の，イテレーション幅やイテレーション回数の変動に対応したプログラムを記述する必要がある．しかしプログラマが，これらの処理量の変動を意識して動画像処理アプリケーションを開発することは困難である．そこで RaVioli では，プログラマから画像データや画像サイズ，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．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 RaVioli の問題点として，抽象化のオーバーヘッドによる処理速度の悪化と，それに伴う処理精度の低下があげられる．例えばテンプレートマッチングを RaVioli で実装した場合，C 言語で記述したプログラムの約 5 倍の処理時間が掛かってしまう．

一方で GPU 向けの C 言語統合開発環境として CUDA（Compute Unified Device Architecture）が開発されている．CUDA は GPU 上に存在する複数のマルチプロセッサ内の複数のコアに対して，大量のスレッドを並列に実行させることでプログラムの高速化を実現する．また GPU 内に存在する複数のコアは同時に同じ命令を実行する SIMD 型を採用している．そのため，一般的にデータ並列性を持つ動画像処理は GPU 上で高い性能が期待できる．

CUDA は C 言語に似た構文を持っており，C 言語を使用したことのあるプログラマは容易に習得可能である．しかしプログラマは，CPU-GPU 間のデータ転送やスレッドの管理を意識してプログラムを記述する必要がある．これらの処理は動画像処理の本質ではなく，また CUDA を用いて効率のよいプログラムを記述したい場合，多くの最適化を施す必要がある．そのため CUDA を用いた動画像処理アプリケーションの開発はプログラマにとって負担が大きい．そこで本研究では，CPU-GPU 間のデータ転送，スレッドの管理，最適化を意識せずに，CUDA を使用した動画像処理プログラムが記述可能なように，RaVioli を拡張する．また従来の RaVioli で記述された動画像処理プログラムを，本研究で拡張後の RaVioli で記述されたプログラムへと自動変換するトランスレータも提案する．

以下 2 章では本研究の背景，動画像処理ライブラリ RaVioli，および GPU 向けの C 言語統合開発環境 CUDA について概説する．3 章では CUDA に対応した RaVioli を提案し，4 章では，従来の RaVioli で記述されたプログラムから，3 章で提案する改良後の RaVioli を用いて記述されたプログラムへと変換を行うトランスレータについて述べる．次に 5 章で提案の評価とそれに対する考察を示し，6 章では提案の関連研究について説明する．最後に 7 章で本論文全体をまとめる．

2　背景

2.1　動画像処理

近年，ビデオカメラなどの入力装置からリアルタイムに画像をキャプチャ可能な環境が整ってきた．また計算機の高性能化によって，従来では不可能であった顔認証などの処理量の多い高度な画像処理を，汎用 PC 上で行うことが可能になった．そのため今後汎用計算機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．

しかし汎用システム上でリアルタイム動画像処理を行う場合，処理に必要な CPU リソースの確保が困難である．その原因として 1 フレームあたりの処理量が変動することや，使用可能 CPU リソースの変動などが挙げられる．例えば顔検出の処理では，

キャプチャした画像から肌色部分を検出し，エッジ抽出を行った後，その結果に対してハフ変換による円抽出を実行する．このとき，人物の人数によって処理量が変動すると考えられる．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 リソースの変動が起こるため，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，実時間並列画像処理アプリケーション構築環境 RPV[1] などが提案されている．しかし，これは高速ネットワークに接続された PC クラスタを利用したものであり，汎用システム上でリアルタイム動画像処理を実現するものではない．

一方で，擬似的なリアルタイム処理を保証する動画像処理ライブラリ RaVioli[2][3] 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，解像度を自動調整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．次節で RaVioli の詳細と問題点について述べる．

2.2 RaVioli

2.2.1 処理量の自動調整

一般に汎用計算機上では，他プロセスの動作により使用可能な CPU リソースが変動する．また 1 フレームあたりの演算量も変動する可能性がある．そのため 1/30 もしくは 1/60 秒毎にカメラ等から送られてくる画像に対し，リアルタイムに処理を施すことは困難である．そこで RaVioli では，使用可能な CPU リソースに応じて動画像の解像度を変動させ，処理量を調整することでこれを解決する．

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，使用可能な CPU リソースに応じてストライドを変動させることで処理量の調整を実現している．

図 1 は，空間解像度ストライド S_I と時間解像度ストライド S_T がそれぞれ 4 と 3 の場合の処理対象部分を示した図である．薄いグレーのフレームが処理対象となるフレームであり，そのフレーム中の濃いグレーの画素が処理対象となる画素である．処理対象フレームは，S_T が 3 であるため通常の 1/3 となる．さらに処理対象画素は，S_I が 4 であるため通常の 1/16 となる．このようにストライドを増加させることで処理画素数や処理フレーム数を変動させ，演算量を低減させる．

また 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．例えば厳密にリアルタイム性を保証したい場

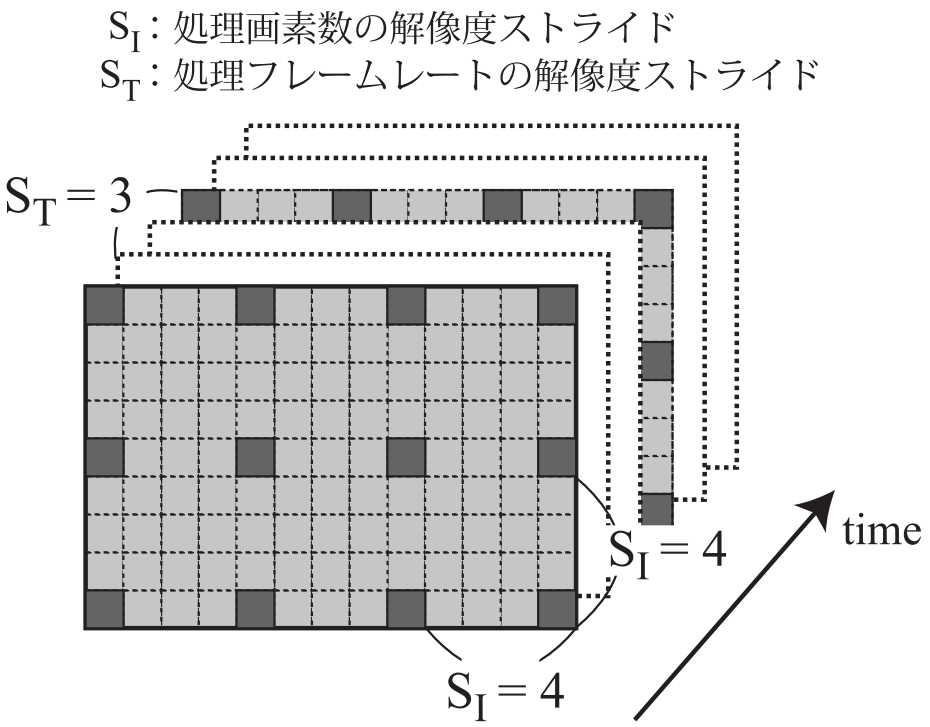

図 1: 解像度ストライドに基づいたアクセス位置の指定

合，空間解像度を低減させ，高いフレームレートを維持する必要がある．ユーザは高い優先度を時間解像度に設定することで高いフレームレートを維持することができる．また顔認証などのように画像の精度が重要なアプリケーションの場合，ユーザは高い優先度を空間解像度に設定することで，時間解像度を低減させ，空間解像度を維持したリアルタイム動画像処理プログラムを実現することができる．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．

2.2.2 動画像処理の抽象化

前節では，リアルタイム性を保持するために解像度を自動的に調整するという RaVioli の仕様について述べた．しかし，これをプログラミングのレベルで解決するには，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を考慮してプログラムを記述する必要がある．これはプログラムの可読性を低下させ，デバッグの際にバグの特定を困難にさせる等の問題を引き起こす可能性がある．

ここで，画像を構成する要素である「画素」や「フレーム」に焦点をあてる．これらの構成要素は，画像や動画像を計算機上で扱うために導入された概念であり，そも

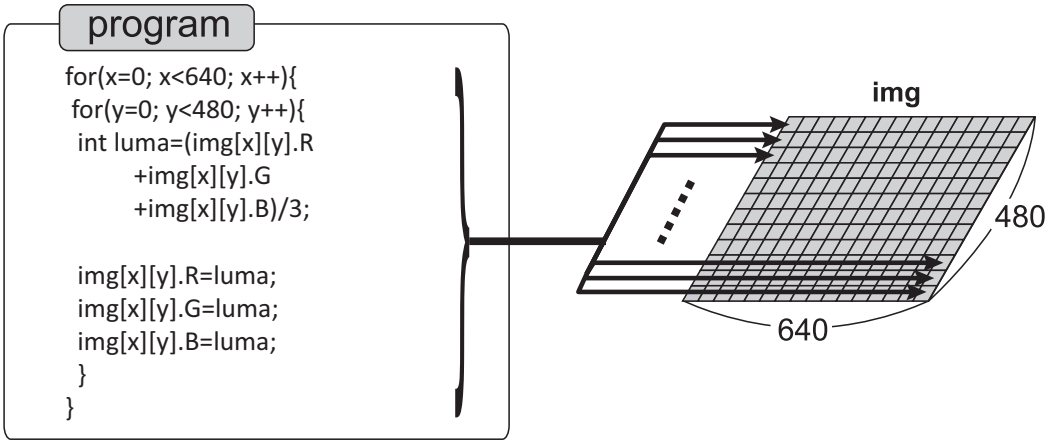

図 2: 通常の画像処理

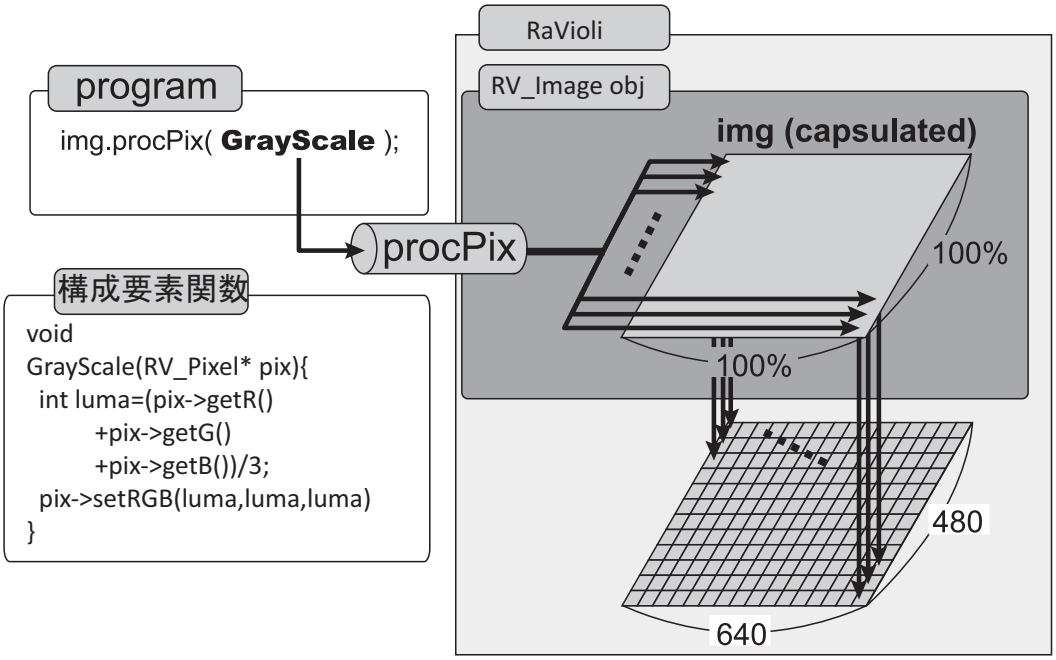

図 3: RaVioli が提案する画像処理

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ざるを得ない．またプログラムを記述する際は，for 文などのループ文を用いてこれらの全ての構成要素に対して繰り返し処理を施す必要がある．この繰り返し処理もまた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．1 フレーム中の画素配列や画像の幅・高さ，フレームレート等をプログラマから隠蔽し，RaVioli 側ですべて管理することで，プログラマ

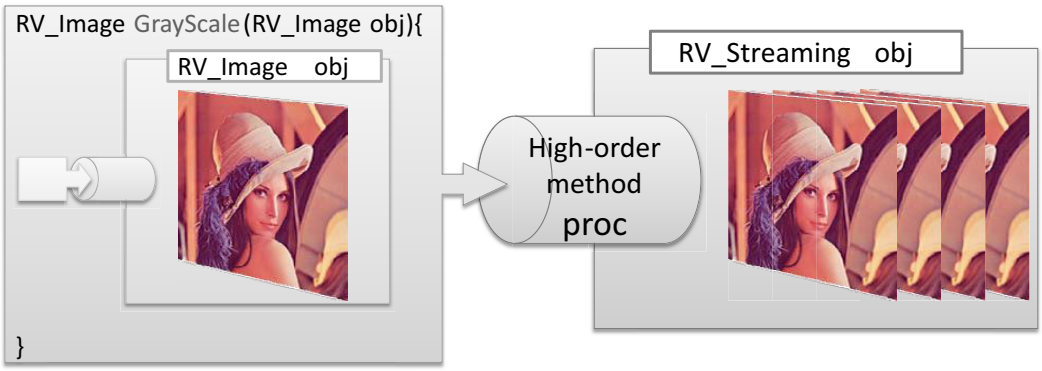

図 4: RaVioli を用いた動画像処理

は解像度を意識せずに動画像処理を記述できる．

一般に画像処理は，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である．また，テンプレートマッチング等の処理では処理単位は小さなウィンドウである．そしてこれらの処理は，**図 2** のように通常ループイテレーションを用いて記述され，構成要素に対する処理を画像に対して繰り返し適用する形で行われる．ここで空間解像度の変動の影響を受けるのは，処理対象画像の画素数に対応するイテレーション回数や，イテレーション変数のインクリメント幅である．

そこで RaVioli では，画像の幅や高さ，画素配列データを RV_Image クラスにカプセル化することによって，ループイテレーション自体の管理をライブラリ内に隠蔽する．**図 3** に，画像をグレースケール化する RaVioli プログラムの例とその処理モデルを示す．プログラマは，まず 1 画素に対する処理を記述した関数 GrayScale() を定義する．RV_Pixel は 1 画素の RGB 値を持つクラスである．RaVioli では，この 1 構成要素（1 画素，1 ウィンドウ等）に対する処理を記述した関数を構成要素関数と呼ぶ．その後，RV_Image インスタンス img が持つ高階関数 procPix() に構成要素関数 GrayScale() の関数ポインタを渡すだけで図 2 と同様の処理を実現することができる．高階関数 procPix() 内では，ループイテレーションによって GrayScale() の処理を画像中のすべての画素に適用する．このようにループイテレーションをライブラリ側で制御することのより空間解像度を隠蔽する．

また空間解像度と同じく，時間解像度の隠蔽も同様に行う．動画像中のフレームや，

表 1: 画像処理の速度比較 (ms)

プログラム名	w/o RaVioli	w/ RaVioli	処理時間の増加率 (倍)
GrayScale	0.841	8.208	9.759
EmbossFilter	1.497	118.327	79.043
TPmatching	1898.223	10453.549	5.507

CPU:Core2Quad(2.83GHz)，memory:3GB

フレームレートといった動画像に関する情報を，**図 4** に示すように RV_Streaming クラスにカプセル化している．プログラマは当該フレーム，または当該フレームおよび隣接するフレームに対する処理のみを記述し，これを RaVioli が提供する RV_Streaming インスタンスの高階関数に渡すことで，動画像中のすべてのフレームに処理を適用することが可能である．

2.2.3 RaVioli の問題点

RaVioli を使用することで，プログラマは解像度を意識せずに直感的にプログラムを記述可能になった．また使用可能な CPU リソースに応じて処理解像度を変動させ，擬似的なリアルタイム処理も実現している．しかし，抽象化のオーバーヘッドにより処理速度が低下し処理精度が悪化するという問題点がある．

表 1 に RaVioli 不使用時と使用時の処理速度の比較と処理時間の増加率を示す．使用したプログラムは上から，グレースケール化，エンボスフィルタ，テンプレートマッチングである．w/o RaVioli は RaVioli を使用せず C 言語で記述したプログラム，w/ RaVioli は RaVioli を使用して記述したプログラムを表す．また増加率は，RaVioli 使用時の処理時間が RaVioli 不使用時から何倍に増加したかを表す．ここで w/o RaVioli の処理時間は，画像の入出力の時間は含めない，画像に対する処理のみの実行時間であり，w/ RaVioli の処理時間は高階メソッド呼び出しの実行時間である．

表 1 に示すように，RaVioli を使用した場合，グレースケール化，エンボスフィルタ，テンプレートマッチングでそれぞれ約 10 倍，80 倍，5.5 倍の速度低下がみられた．そこで本研究では，CUDA を用いて RaVioli の高速化を目指す．次節では CUDA の詳細と問題点について述べる．

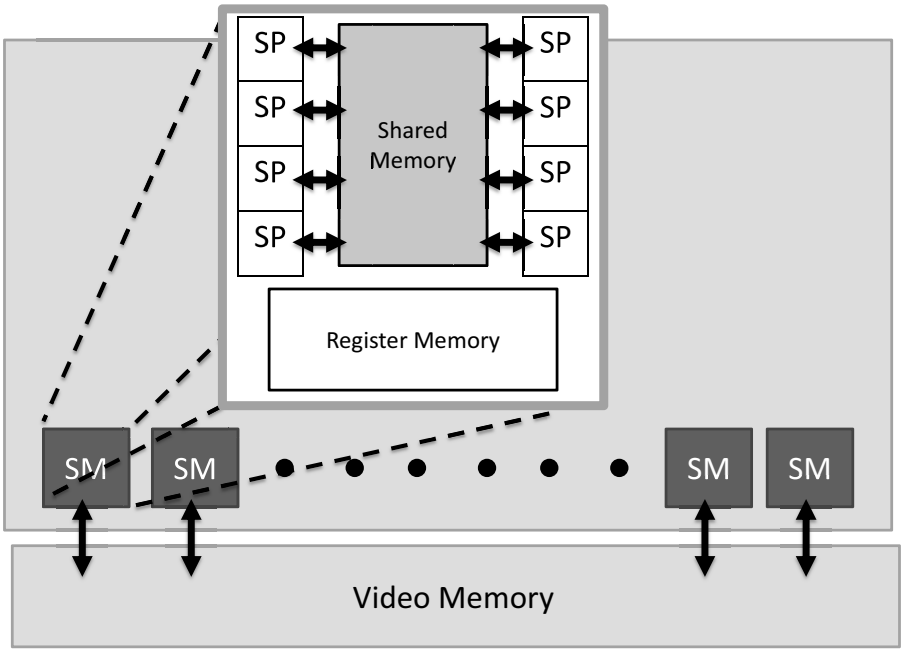

図 5: GPU のアーキテクチャ

2.3 CUDA

現在，GPU 向けの C 言語統合開発環境として CUDA [4][5] が NVIDIA 社により開発されている．GPU（Graphics Processing Unit）は画像処理に特化した専用プロセッサであり，広域なメモリバンド幅や高い演算性能を持つ．**図 5** に NVIDIA 社の GPU のアーキテクチャを示す．GPU の構造はコアの世代と型番で異なるが，そのアーキテクチャはほぼ同じである．

GPU チップの内部には多数のストリーミング・マルチプロセッサ（SM）が存在する．さらに各 SM には 8 個の演算処理ユニット（ストリーミング・プロセッサ，SP と呼ぶ）が入っている．GPU では，この SM 内の 8 個の SP に対して同じ命令を実行する SIMD 型を採用している．一般的に画像処理はデータ並列性を持っており，SIMD 型を採用している GPU 上で高い性能が期待できる．そこで本研究では GPU を使用することで RaVioli の高速化を図る．

なお CUDA は，NVIDIA の GPU 向けのプログラミング環境一式であり，プログラミングモデルおよびプログラミング言語と，そのコンパイラ，ライブラリ等の事を指す．以下の節では，CUDA のプログラミングモデルとメモリモデルについて述べる．

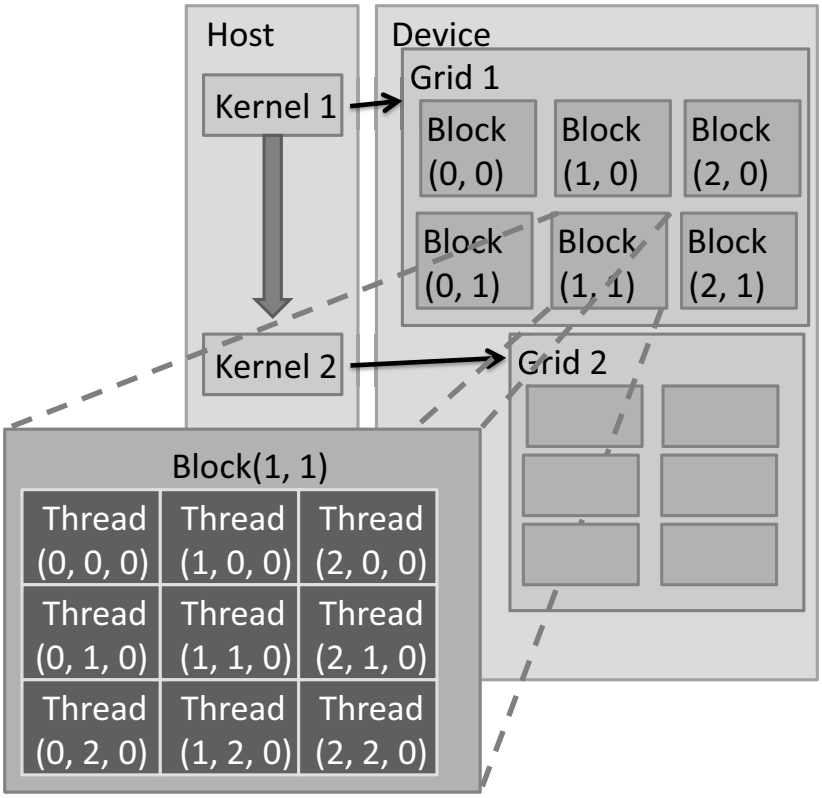

図 6: プログラミングモデル

2.3.1 プログラミングモデル

GPU は大量の thread を並列に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 に対して最大で「65535 × 65535 × 512」個の thread を使用することが可能である．ここでこの大量の thread をどのように管理するかが問題となる．CUDA ではこの thread を**図 6** のように階層的に管理する．CUDA では CPU を Host，GPU を Device と呼んでいる．また thread の集合を Block と呼ぶ．Block の中で thread は x 軸方向，y 軸方向，z 軸方向の 3 次元的に配置され管理される．また同次元，同数の thread から成る Block の集合を Grid と呼ぶ．Grid の中で Block は 2 次元的に配置され管理される．図 6 の Grid1 は 3×2 の Block の集合から成り，それぞれの Block は $3 \times 3 \times 1$ の thread の集合から成る．このように階層的かつ多次元的に thread を割り振ることで thread の総数が大きくなっても効率的に処理可能である．

Host から呼ばれ Device で実行される関数を Kernel 関数と呼ぶ．Device はひとつの Grid を実行単位として Kernel 関数を実行する．そのため Grid 内のすべてのスレッドは同じ Kernel 関数を実行することになる．Grid 内の Block 数や Block 内の thread 数は Host のプログラム中で Kernel 関数を実行する際に指定する．

```
__global__ void matAdd(float* A, float* B, float* C){
   int i = gridDim.x*blockIdx.x+threadIdx.x;
   int j = gridDim.y*blockIdx.y+threadIdx.y;
   int k = gridDim.x*blockDim.x;
   C[j*k+i] = A[j*k+i] + B[j*k+i];
}
```

図 7: 例:カーネル関数

Kernel 関数には GPU で実行される 1thread の処理を記述する．Kernel 関数の例を**図 7** に示す．これはサイズ $N \times N$ の 2 つの配列 A と B を足して配列 C へ結果を代入するプログラムである．Grid 内には $N \times N$ の thread が存在することとする．Kernel 関数は__global__という修飾子を用いて宣言される．またこの Kernel 関数を実行する各 thread は固有の thread ID を持ち，ビルトイン変数である threadIdx，blockIdx，gridDim 等を用いることで当該 thread がどのメモリアドレスにアクセスするかを指示することが可能である．例えば threadIdx.x は，当該 thread が Block 内で x 軸方向のどの位置に存在するかを示す．また gridDim.x は Grid の x 軸方向のサイズを示す．

この Kernel 関数が実際に呼ばれると，まず Device 側では $N \times N$ の thread が生成される．Grid 内の各 thread は，Kernel 関数に記述された，配列の 1 要素分の計算を行う．すべての thread は並列に Kernel 関数を実行するため，高速に処理を行うことができる．

2.3.2 メモリモデル

CUDA のメモリモデルを**図 8** に示す．CUDA では，Register Memory，Local Memory，Shared Memory，Global Memory，Texture Memory，Constant Memory が使用できる．これらはグラフィック・ボード上に存在するので,「デバイス・メモリ」に分類される．これらの詳細を**表 2** に示す．表中のアクセス欄は Device からのアクセスを表し，R/W は読み書き可能，R は読み出しのみを表す．

Global Memory

GPU のチップの外に存在するため，Register や Shared Memory と比較すると 100 倍以上もアクセスが低速である．しかしグラフィック DRAM で構成されるビデオ・メモリに置かれているため，容量はとても大きい．Global Memory はすべての Block 中のすべての thread から読み書き可能であり，Host からも CUDA の API を使用するこ

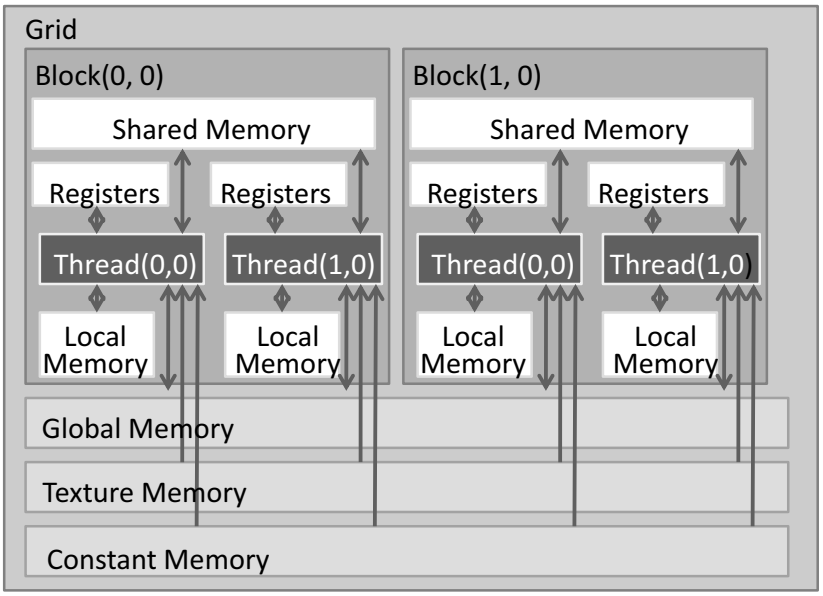

図 8: メモリモデル

表 2: デバイス・メモリ

種類	場所	キャッシュ	アクセス	スコープ
Register Memory	チップ上	-	R/W	thread
Local Memory	チップ外	されない	R/W	thread
Shared Memory	チップ上	-	R/W	Block
Global Memory	チップ外	されない	R/W	Grid と Host
Texture Memory	チップ外	される	R	Grid と Host
Constant Memory	チップ外	される	R	Grid と Host

とで読み書きが可能である．また，CUDA の API である cuMemMalloc() によってメモリが確保されてから，cuMemFree() によって解放されるまでの間であれば，Kernel 関数が実行されても Global Memory 上の領域は確保されつづける．

Texture Memory

SM ごとにテクスチャ・キャッシュが搭載されており，Texture Memory にアクセスした際のデータが一時的に保存される．そのためデータアクセスに局所性がある場合に高速に読み出しが可能である．テクスチャ・キャッシュは 2 次元空間の局所性に最適化されているため，SM 上での実行単位である 32 thread が互いに近いアドレスを参照する場合に高速にアクセス可能である．Device からは読み出しのみ可能である．Host

からはCUDAのAPIを用いることで読み書きが可能である．またGlobal Memoryと同様で，Hostから割り当てられている間であれば確保されつづける．

Constant Memory

SMごとにコンスタント・キャッシュが搭載されており，Constant Memoryにアクセスした際のデータが一時的に保存される．そのためデータアクセスに局所性がある場合に高速に読み出しが可能である．しかし高速な読み出しが可能なのは，SM上での実行単位である32 thread全体が同じアドレスを参照する場合にのみである．異なるアドレスを参照するthread数に応じて，アクセスコストは直線的に増加する．Hostと Deviceからのアクセス，およびスコープはTexture Memoryと同様である．

Register Memory

SMごとに搭載されているオンチップ・メモリである．チップ上に置かれているので高速に読み書きが可能である．Kernel関数で使用される変数の値はここに保持される．SM上で実行されるBlockの使用するRegister数が，SM中（オンチップ上）に存在するRegister数以上である場合，Kernel関数を実行できなくなる．Deviceからは読み書きが可能であり，当該thread内でのみ使用可能である．Hostからはアクセスすることができない．またthreadの実行が終了すると，Register Memoryは解放される．

Local Memory

Global Memoryと同様に，GPUのチップの外に存在する，グラフィックDRAMで構成されるビデオ・メモリに置かれているため，Registerと比べると100倍以上もアクセスが低速である．Register数が不足した際，コンパイラ・オプションで「- maxrregcount32」などと指定をすると，1 threadあたりに使用するRegister数を32に抑えることができ，足りない分のRegisterデータの退避場所として使用される．アクセスやスコープはRegister Memoryと同様である．

上記のようにアクセス速度が低速であるため，GPUを使用して高速計算を目指すには，できる限りRegister数の容量を越えないようにし，Local Memoryは使用しないのがよいとされる．なおLocal Memoryは，コンパイラで自動的に割り当てられるため，プログラマが意識して使用することはない．

Shared Memory

NVIDIAのGPUのアーキテクチャを特徴付けるメモリであり，SMごとに16,384byte搭載されている．オンチップ上に存在するため，Registerと同等に高速アクセスが可能である．Block内のすべてのthreadは，同期を取ることでShared Memoryを介してデータのやりとりをすることができる．一方Blockをまたいでの Shared Memoryを介

したデータのやりとりはできない．ここで，Host と Device からのアクセスは Register，Local Memory と同様である．また当該 Block の実行中は Shared Memory は確保されつづける．

Shared Memory は Kernel 関数中で静的に確保することや，Kernel 関数の呼出し時に動的に確保することが可能である．また，Kernel 関数の引数の値を確保する場所としても使用される．

2.3.3 CUDA の問題点

CUDA は上述のようなプログラミングモデルを実現することで，GPU 上で動作するプログラムを以前よりも容易に記述可能とした．しかし thread やメモリの管理は動画像処理の本質ではないため，動画像処理アプリケーションを開発しようとするプログラマにとって負担が大きいといえる．また効率のよいプログラムを記述しようと考えると以下のような最適化が必要になる．

実行構成の最適化

SM に含まれる 8 個の SP は 4 サイクルに渡って同じ命令を実行する．これは SP のクロック周波数が SM の 1/3 強であるため，毎サイクル新しい命令を供給することができないからである．つまり 1 つの SP は時分割で 4 つの thread の処理を担当することになる．よって最低 32 本の thread がないと，SM 中の SP の実行に空きが生じてしまうことになる．この 32 thread の単位は Warp と呼ばれる．一方，Grid 内の各 Block はそれぞれの SM 上で実行される．そのため SM 上で効率よく動作させるためには Block 内の thread 数を Warp 内の thread 数である 32 の倍数にする必要がある．

また SM 上で動作する Block は時分割で実行される．そのため，メモリからのデータ待ちやリソースの競合が原因で起こる遅延を隠蔽するためには 1 つの SM 上で 2 つ以上の Block を動作させることが理想的であるといえる．ここで SM 上で動作させることの出来る Block 数は，Kernel 関数内で使用される Shared Memory の量と Register 数に依存する．そもそも 1 つの SM で使用可能な Shared Memory の量と Register 数には限りがある．そのため Kernel 関数が使用する Register 数を増加させると起動することが可能な thread が減少してしまう．この 2 つはトレードオフの関係にあるため，プログラマは Kernel 関数内で使用する Register 数やシェアードメモリの量，実行構成を考慮してプログラムを記述する必要がある．

コアレッシング （coalescing）

Global Memory へのアクセスの単位は以下の 3 種類である．

- 32 バイト境界にアライメントされた 32 バイトのブロック

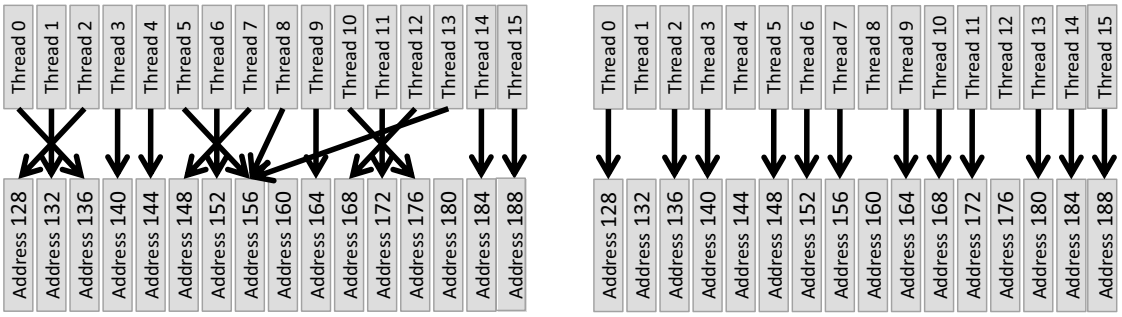

図 9: コアレッシングが可能なメモリアクセス

- 64 バイト境界にアライメントされた 64 バイトのブロック
- 128 バイト境界にアライメントされた 128 バイトのブロック

このように必ず，32 バイト，64 バイト，128 バイトという大きな単位で行われる．そのため，4 バイトのデータを読むだけのために 32 バイトが転送されることもあり，これはとても非効率的である．そこで，GPU メモリのデータ転送能力を有効に活用するコアレッシングという仕組がある．

コアレッシングとは，Warp の半分である 16 thread（以下，half warp と呼ぶ）が，連続したメモリに同時にアクセスすることで，メモリ・アクセスの効率化を実現する仕組である．コアレッシングが可能なメモリアクセスの例を**図 9** に示す．またコアレッシングの条件は以下の通りである．

- half warp が，それぞれ同一のデータサイズ（8 ビット，16 ビット，32 ビット，64 ビット）にアクセスする場合
- それぞれアクセスする先が一定サイズ（8 ビット→ 32 バイト，16 ビット→ 64 バイト，32 ビット，128 バイト，64 ビット→ 128 バイト）のセグメント内に収まる場合

さらに，コアレッシングされる際の先頭アドレスが，32 バイト，64 バイトまたは 128 バイト境界にアライメントされていなければ，アクセスの性能が下がってしまう．またコンピュート・ケイパビリティが現在のものよりも古い 1.0〜1.1 の場合は，より厳密な制約がある．ここでコンピュート・ケイパビリティとは GPU のバージョン番号のことであり，この数値によってハードウェア・レベルでの CUDA のサポート範囲が変わる．

Register，Shared Memory の使用

Register および Shared Memory へのアクセスは Global Memory へのアクセスと比較すると 100 倍以上も高速である．そのため，可能な限りこれらのメモリを使用する

必要がある．またShared Memoryを用いることで，Block内のthread間でデータを交換することが可能である．シェアード・メモリは16個のバンクによって構成されており，Global Memoryと同様16 threadずつ処理される．各バンクは，1度に1箇所へのアクセスしか対応できない．そのため，同じバンクに属する複数の箇所にアクセスが起こる場合には，順番に処理されるため，より長い時間掛かってしまう．これをバンク・コンフリクトと呼び，プログラマはこれを回避するようにプログラムを記述する必要がある．

データ転送とKernel関数の実行のオーバーラップ

ストリームという機能を使用することで，Host-Device間の通信とKernel関数を同時に実行させることが可能である．この機能によって，GPUを効率よく利用することが可能である．詳細については3.3.3節で述べる．

Warpダイバージョントの回避

SM上の8個のSPは4サイクルに渡って同じ命令を実行するSIMD型であることはすでに述べた．しかしKernel関数はSPMD型のプログラミング手法を採用しており，if文などの制御構文で異なる方向に分岐するthreadが存在する可能性がある．SM上での実行単位であるWarpにおいて，分岐先の異なるthreadが存在すると，まず分岐なしの方向のthreadだけを実行し，次に分岐ありの方向のthreadだけを実行するという処理を行う．そのため，分岐が起こった場合，両方のフローが合体するまで，分岐の両方向の命令を実行するための時間が必要となり，1サイクルに有効な仕事ができるthread数が減少する．これをWarpダイバージェントと呼ぶ．プログラマは，できるだけ条件分岐を使用しないか，Warp内の条件分岐の分岐先が同一になるようなプログラムを記述する必要がある．

このような最適化を施すには，GPUのハードの知識，およびCUDAのプログラミングモデルやCUDAの機能に関するより深い知識が必要とされる．そこで本研究では，Host-Device間の転送や実行構成の設定，CUDAプログラムの最適化を意識せずに，動画像処理プログラムの記述が可能となるようにRaVioliを拡張する．

3 RaVioli+CUDA

Host側で必要になるCUDAの管理には以下のようなものがある．

- デバイス管理
- コンテキスト管理
- メモリ管理

- コードモジュール管理
- 処理制御
- テクスチャリファレンス管理

これらを扱う API には，低レベル API であるドライバ API と，ドライバ API の上位に実装された高レベル API であるランタイム API の 2 つが存在する．この 2 つの API は排他的であり，ひとつのアプリケーション内ではどちらか一方の API しか使用できない．

ランタイム API は，暗黙的な初期化やコンテキスト管理，モジュール管理を提供することで，デバイスコードの管理をより容易にする．ドライバ API では，より多くのコードが必要になり，プログラムとデバッグが困難になる．しかしドライバ API の場合，Device 側で実行される Kernel 関数のみ CUDA に依存した記述が必要で，Host 側のコードは通常の C/C++言語で記述することが可能である．一方ランタイム API では，CUDA 用に拡張された C/C++言語で記述する必要がある．

例えばランタイム API では，テクスチャ参照変数はファイルスコープ変数であるため，同一ファイル内でしか使用することができない制限がある．一方ドライバ API では，ハンドルベースの命令 API であり，ハンドルを用いることでファイルを跨いで使用することが可能である．また，ドライバ API ではモジュールが利用可能である．これは動的にロード可能なデバイスコードとデータのパッケージである．UNIX のシェアード・オブジェクトや Windows の DLL などと似た機能を提供する．以上を踏まえ本研究では，より制限が少なくかつ使用可能な機能が多い，ドライバ API を使用して RaVioli の拡張を行うこととした．

3.1 実行構成

Grid の次元や Grid 内に含まれる Block 数，および Block の次元や Block 内に含まれる thread 数を**実行構成**と呼ぶ．この実行構成が Kernel 関数の実行時間に影響をどのくらい与えるかは，Kernel 関数のコードに依存する．そのため経験的に決定されるのが一般的によいとされている．しかしいくつかの制約が存在する．

まず 1 Block あたりの最大 thread 数（512 threads）を越える場合や，1 Block が使用する Register 数や Shared Memory がマルチプロセッサあたりのメモリ量を越える場合は，Kernel 関数の実行は失敗する．Block あたりに必要とされる Register 数の総和

は以下の式で示される．

$$Ceil(R \times Ceil(T, 32), \quad \frac{R_{max}}{32})$$

- R: Kernel 関数 (1 thread) が使用する Register 数
- R_{max}: マルチプロセッサあたりの Register 数
- T: 1 Block あたりの thread 数

$Ceil(x, y)$ は y の倍数に近い値に端数を切り上げた x の値を表す．例えば，GeForce 280GTX を使用する場合，R_{max} は 16,384 になるため，Kernel 関数が使用する Register 数を 16 とすると，Block サイズを最大数である 512 threads に設定しても，Block あたり 8,192 となり，マルチプロセッサあたりの Register 数の総数の半分であるため，1 マルチプロセッサあたりのアクティブ Block 数を 2 とすることが可能である．またこのように Block サイズを 512 threads とする場合は，Kernel 関数あたり最大 32 Register を使用することが可能であるが，データアクセスの遅延を隠蔽するためにはアクティブ Block 数を 2 以上にする必要があるため，最大 16 Register に抑えるべきである．

ここで Register は，Kernel 関数のプログラム中で変数を宣言すると，その変数の値を格納するために割り当てられる．実際に Kernel 関数が GPU 上で動作させられる際は，Register は使いまわされるため，宣言された変数の数だけ Register が使用されるわけではない．しかし相関はあるため，プログラマはある程度 Register の使用数を予想してプログラムの記述をすることが可能である．

一方 1 Block あたりの Shared Memory は静的または動的に割り当てられた Shared Memory に等しい．またシェアード・メモリは Kernel 関数の引数の値を確保する場所としても使用される．各 SM 中に 16,384byte 搭載されており，1 Block あたりの Shared Memory の使用量がこれを越える場合は，Kernel 関数の実行は失敗する．

次に 1 Grid あたりの適切な Block 数について説明する．まず最低でも Device 中に存在するマルチプロセッサ数と同等な Block 数は必要である．またマルチプロセッサあたりの Block 数が 1 つしかないと，Block 内にロード遅延を隠蔽するのに十分な thread 数がないため，thread 同期や Device メモリへのアクセスを行う間アイドル状態になってしまう．そのため，2 つ以上の Block がマルチプロセッサ上で動作することが理想的であるといえる．

上記の状況を実現するためには，Device 上に存在するマルチプロセッサ数の 2 倍以上の Block 数にするだけでなく，1 アクティブ Block あたりの Register 数や Shared Memory を調整する必要がある．

またRegisterアクセスは1命令1サイクル掛からないが，Registerのread-after-writeやバンクコンフリクトによる遅延が起こる可能性がある．この遅延を隠蔽するには少なくともマルチプロセッサあたり192のアクティブthreadがいる．

以上より，十分なRegister数やShared Memoryを確保し，データアクセスの際の遅延を隠蔽するためには，1 Blockあたりのthread数は192または256が妥当であると考えられる．またGlobal Memoryを使用する際にコアレッシングさせるために，Blockのx軸方向のサイズは16の倍数にする必要があるため，今回は1 Blockのサイズを16×16とすることとした．また，1 Gridあたりの Block数はDeviceのマルチプロセッサ数の2倍以上になるように画像サイズを考慮して決定する．

3.2 画像処理プログラム

RaVioliは，画像の構成要素である「画素」および「フレーム」をプログラマから隠蔽する新しいプログラミングパラダイムを提供している．動画像処理プログラムから空間解像度と時間解像度を示す変数をライブラリ内に隠蔽し，画素データや解像度の管理をRaVioli側で行うことで，プログラマは解像度を意識せずに動画像処理プログラムを記述可能となった．

ここでCUDAを使用して画像処理プログラムを記述する場合を考える．プログラマはDevice側のメモリやthreadの実行構成を意識してプログラムを記述する必要がある．具体的には以下の処理が必要となる．

- Device側に画像サイズ分のメモリを確保する
- 処理対象画像の画素データをHostからDeviceへ転送する
- 実行構成を設定する
- Kernel関数の引数を設定し，Kernelを呼び出す
- 全スレッドの同期をとる
- 処理結果画像をDeviceからHostへ転送する
- Device側のメモリを開放する

これらの処理は画像処理の本質ではないため，プログラマの負担になると考えられる．そこで本研究ではこの問題に対し，Device側のメモリ管理をRaVioli内で行うことによってプログラマから上記の処理を隠蔽するインターフェースを提供する（**図10**）．

RaVioliでは，構成要素関数のポインタを引数として受け取り，それをイテレーションによって画像全体へと繰り返し適用する高階メソッドを持っている．これは，画像処理プログラムから画像の幅，高さ，画素配列を隠蔽し，それらをライブラリ内で管

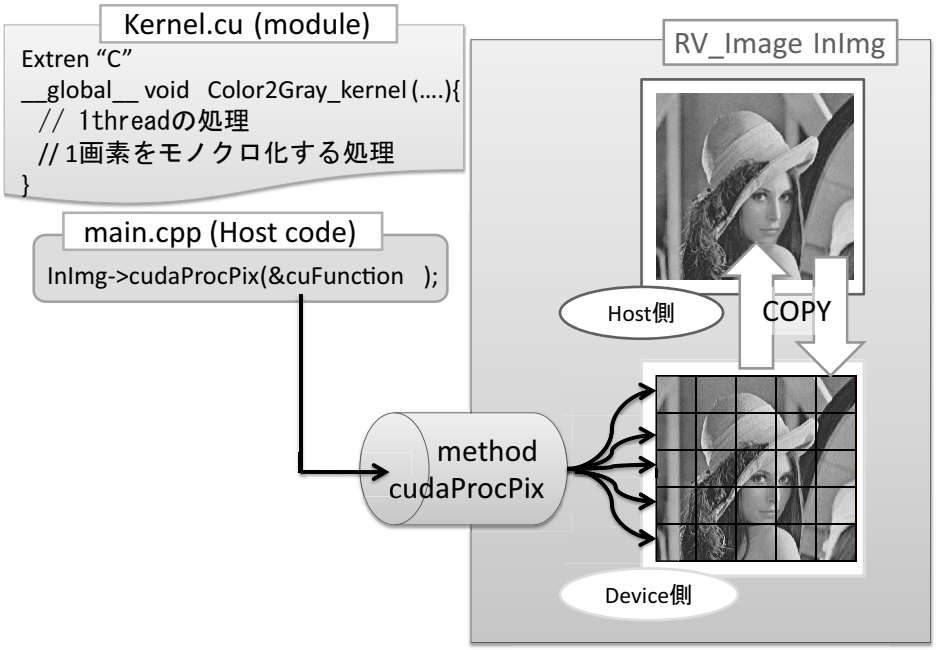

図 10: cudaProcPix:RaVioli+CUDA のインターフェース

理することによって実現している．

画像の幅，高さや画素配列をライブラリ内に隠蔽することで，Device 側のメモリ確保や Host-Device 間のデータ転送，実行構成の設定もライブラリ内で管理することが可能である．プログラマは図 10 に示すように，まず画像を RV_Image インスタンスとして定義する．RV_Image インスタンスは画像の幅，高さ，および画素配列データなど画像に関する情報を持つ．その後，あらかじめ定義した Kernel 関数のハンドルを RV_Image インスタンスの高階メソッド cudaProcPix() の引数に渡すだけで，CUDA を使用した画像処理が可能である．ここで，Kernel 関数はモジュールとして扱うために別ファイル（*.cu）内に記述するものとする．*.cu ファイルに記述された Device コードは，nvcc を使用してコンパイルすることでアセンブリ形式（ptx コード）またはバイナリ形式（cubin コード）へと変換される．この変換後のコードはモジュールとして扱うことが可能である．Host 側からはこのモジュールをロードし，また使用したい Kernel 関数のハンドルを取得することで，モジュール内の Kernel 関数を使用することができる．

Kernel 関数は**図 11** のように定義する必要がある．引数は左から処理対象画素配列 idata，出力画像を格納するための配列 odata，画像の幅 width，高さ height である．これらの値は cudaProcPix() 内で設定される．プログラマはこれらの変数を使用してプ

```
1   /* kernel.cu （モジュール） */
2   extern "C" __global__ void
3   Color2Gray(int* idata,int* odata,int width, int height){
4      int x=blockDim.x*blockIdx.x+threadIdx.x;
5      int y=blockDim.y*blockIdx.y+threadIdx.y;
6      int rgb;
7      if(x<width && y<height){
8        rgb=idata[y*wid+x];
9        int ave=(getR(rgb)+getG(rgb)+getB(rgb))/3;
10       setRGB(&odata[y*wid+x],ave,ave,ave);
11     }
12  }
```

図 11: プログラマが定義するカーネル関数の枠組

ログラムを定義する．またコアレッシングアクセスを考慮して，隣接する thread が隣接する画素を処理するように記述する必要がある．

ここで図 11 に示すように，プログラマは画像の幅，高さを意識したプログラムの記述が必要になる．3.1 で述べたように，RaVioli 側では実行構成を設定する際，処理対象の画像サイズを考慮した 2 次元に配置された thread の構成にする．現在のところ，Block は 16×16 の 2 次元に設定し，Grid は $(width/16 + (0! = width\%16)) \times (height/16 + (0! = height\%16))$ と設定している．そのため thread の実行構成が画像サイズよりも大きくなる場合が考えられるので，図 11 のように条件式によって画像の範囲外にアクセスしないようにしなければならない．

画像の幅と高さを意識したプログラミングを回避する手段として，従来の RaVioli と同様な考えで**図 12** のような記述方式を考えた．プログラマは，1 画素に対する Kernel 関数 UserKernel() を記述し，その Kernel 関数のハンドルを RaVioli が持つ高階メソッド cudaProcPix() の引数へと渡す．cudaProcPix() 内では，RaVioli が持つモジュール ProcKernel() の引数として UserKernel() のハンドルを渡し，Grid 内の全 thread に UserKernel() を実行させる．そうすることで，プログラマは構成要素に対する処理を記述して高階メソッドの引数に渡すだけで，画像全体に処理を施すことが可能である．

しかし CUDA を使用する際には，Host から呼ばれ Device で実行される Kernel 関数

```
1   /* User Module*/
2   __device__ void
3   UserKernel(int* rgb){
4     int ave=(getR(rgb)+getG(rgb)+getB(rgb))/3;
5     setRGB(rgb,ave,ave,ave);
6   }
7   /* User Main*/
8   int main(){
9     RV_Image img;
10    // UserKernel のハンドル UserK を取得
11    img->cudaProcPix(&UserK);
12  }
13  /* RaVioli Module （cudaProc() 内で呼ばれる）*/
14  _extern "C" __global__ void
15  ProcKernel(CUfunction* UserKernel,int* idata,int* odata
16             ,int width, int height){
17     int x=blockDim.x*blockIdx.x+threadIdx.x;
18     int y=blockDim.y*blockIdx.y+threadIdx.y;
19     int rgb;
20     if(x<width && y<height){
21       rgb=idata[y*width+x];
22       // UserKernel の引数に rgb のアドレスを渡して呼び出す
23       odata[y*width+x]=rgb;
24     }
25  }
```

図 12: 実現不可能な記述方式

である__global__関数の関数ポインタは使用できるが，Device から呼ばれ Device で実行される関数である__device__関数の関数ポインタは使用できないという制約がある．ここで UserKernel() は__device__関数である．よって図 12 のような仕様は実現不可能である．そのためプログラマは図 11 のような画像の幅や高さを意識したプログラミング

が必要不可欠となる．

このようにプログラマは，Host-Device間のデータ転送や実行構成を意識せずにCUDAを使用した画像処理の記述が可能であるが，一方で画像の幅や高さ，さらにコアレッシングやRegister等の管理を意識してKernel関数を定義する必要がある．そこで本研究では，従来のRaVioliで記述されたプログラムからRaVioli+CUDAを使用したプログラムへと変換するトランスレータも提案する．

3.2.1 処理単位がウィンドウの場合の画像処理プログラム

処理単位が画素（および近傍画素）の場合は，前節のとおりに実装することで，CUDAを使用した画像処理プログラムを実現することが可能である．しかし処理単位は画素の他に，ウィンドウの場合が存在する．処理単位がウィンドウの場合の画像処理プログラムの具体的な記述例として，テンプレートマッチングを挙げる．簡略化したテンプレートマッチングのプログラムを**図13**に示す．テンプレートマッチングは，処理対象画像からテンプレート画像と最も類似した箇所を探索する処理を行う．順次テンプレート画像をずらして，処理対象画像中の部分画像との類似度を求めることでこの処理を実現する．ここで類似度はテンプレート画像中の画素と，部分画像中の画素の両画素値の絶対差の総和であり，値が小さいほど類似度が高い．

プログラマはまず1windowに対する処理を記述したKernel関数TPmatching_kernel()を定義する．このKernel関数内では，まず処理対象画像中の(x, y)から始まるwindowとテンプレート画像の類似度を求める．ここで，Grid中の各threadは(x, y)から始まるwindowとテンプレート画像の類似度をそれぞれ持っていることになる．そのため次の操作として，Grid内の各threadが持つ類似度の中で最小の値を見つけ，またそのときの座標を求める必要がある．

Grid内の各threadが持つ類似度の最小値を求めるには，リダクション処理が必要になる．リダクション処理とは，並列数分用意した一時的な格納領域に対してthreadローカルな処理結果を格納し，その処理結果を最後に統合することで，全体の最終的な結果を求める処理である．

CUDAではShared Memoryを使用することで，Block内のthread間でデータのやり取りをすることが可能である．そのため，**図14**のように木構造的に効率的にリダクション処理を行うことが可能である．ここで，本研究ではthreadの実行構成としてBlockのサイズを16×16としている．そこで図14のレベル1の部分では，256要素の中での最小値を求めることが効率的であると考えた．そのためGrid中のBlock数を256（x軸方向に16Block，y軸方向に16Block）に設定することとした．

```
1  /* kernels.cu */
2  texture<int, 2, cudaReadModeElementType> texTP;
3  extern "C"__global__ void
4  reduction_kernel(/*...*/){
5    // リダクション処理
6  }
7  extern "C"__global__ void
8  TPmatching_kernel(/*...*/){
9    int x=blockDim.x*blockIdx.x+threadIdx.x;
10   int y=blockDim.y*blockIdy.x+threadIdx.y;
11   // 1thread の処理
12   // (x,y) から始まる 1window に対する処理
13 }
14 /* program.cpp */
15 RV_Cuda device;
16 int main(int argc, char* argv[]){
17   RV_Image* image; RV_Image* TPimage;
18   CUarray d_TPimage;
19   // 処理対象画像を image に読み込む
20   // テンプレート画像を TPimage に読み込む
21   device.RaCudaInit();      // Device 側の初期化処理
22   // Texture 参照変数 texTP のハンドル cuTexTPref の取得
23   TPimage->TexRefSetImage(&d_TPimage,&cuTexTPref);
24   // TPmatching_kernel のハンドル cuFunction の取得
25   // reduction_kernel のハンドル cuFunction2 の取得
26   cuParamSetTexRef(cuFunction, CU_PARAM_TR_DEFAULT, cuTexTPref);
27   int4 result=image->cudaProcBox(&cuFunction,TPimage->Width,
28                                  TPimage->Height, &cuFunction2);
29   cuArrayDestroy(d_TPimage);
30   device.RaCudaExit();    // Dvice 側の終了処理
31   return 0;
32 }
```

図 13: テンプレートマッチング

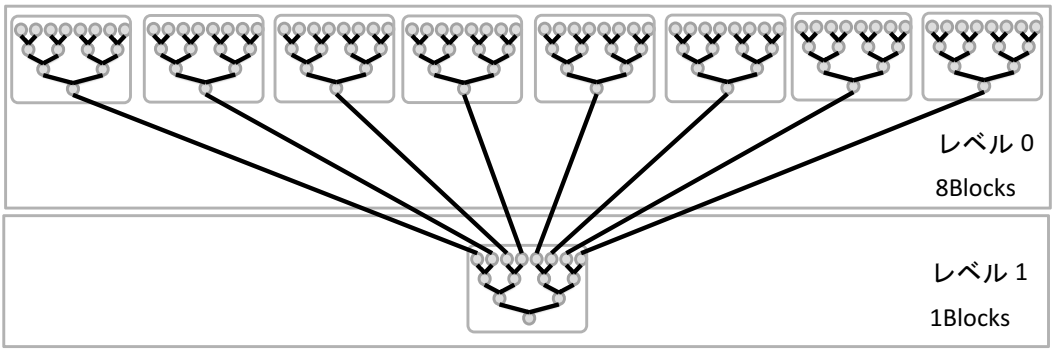

図 14: カーネル関数を複数回実行して最終的な結果を求める

よって Grid 中の thread のサイズは 256×256 になる．そのため TPmatching_kernel() 内では，画像の処理範囲のサイズが 256×256 以上である場合，$x+ = 256$，$y+ = 256$ のように当該画素から 256 画素間隔でサイクリック的に処理を施す必要がある．またサイクリック的に処理を施す中で thread ローカルな最小値とそのときの座標を求める必要がある．TPmatching_kernel() で求めた thread ローカルな処理結果を，前段落で述べたように Shared Memory を用いて木構造的に最終的な結果を求める．その処理を行う Kernel 関数もまたプログラマによって記述される必要がある（以下の説明のために，この Kernel 関数の名前を reduction_kernel() （図 13 中 3〜6 行目）と呼ぶこととする）．

これらのリダクション処理を意識した Kernel 関数の記述はプログラマの負担になる．そこで本研究では，従来の RaVioli で記述されたプログラムを解析することで，リダクション処理が必要な箇所を見つけ，自動的にこれらのリダクション処理を施したプログラムを生成するトランスレータも提案する．トランスレータの詳細は 4 章で述べる．また TPmatching_kernel() および reduction_kernel() の詳細も 4 章に譲ることにする．

上記のように定義した TPmatching_kernel() および reduction_kernel() のハンドルと，ウィンドウの幅，高さである TPimage->Width，TPimage->Height （テンプレート画像の幅，高さ）を cudaProcBox() の引数に渡す．そうすることで，テンプレート画像の大きさのウィンドウを処理対象とする処理を，画像全体に施すことが可能である．cudaProcBox() はリダクション処理後の結果を返り値として返す．ここで「int4」は CUDA に存在するベクトル型の変数であり，4 つの int 型の値を保持することができる．今回の処理では，類似度の最小値と，そのときの x 座標，y 座標が返り値として返される．リダクション処理が必要ない場合は cudaProcBox() の 4 つめの引数はなくてよい．その際の返り値は void である．

一方，テンプレート画像は図 13 の 22，23 行目のように Texture Memory に割り当てる．Texture Memory に割り当てる際は，一端 Global Memory に割り当てる必要があるため，Texture 参照変数のハンドルの他に Global Memory を扱う変数 d_TPimage も渡す．TexRefSetImage メソッド内では，d_TPimage が指す Global Memory 上にテンプレート画像を転送し，Texture 参照変数 texTP にバインドすることでテンプレート画像を Texture Memory に割り当てている．また，26 行目で TPmatching_kernel() のパラメータとして設定することで，TPmatching_kernel() 内でテンプレート画像が扱えるようになる．また処理の後には，29 行目で d_TPimage の指す Global Memory の解放を行っている．

処理対象がウィンドウの場合は，上記のように 1 ウィンドウに対する処理を記述した Kernel 関数を定義して，そのハンドルを cudaProcBox() に渡せばよい．また処理対象画像中の 1 ウィンドウと比較したい画像がある場合は，cudaProcBox() を呼び出す前に，比較対象の画像を Texture Memory に割り付ける必要がある．その後，そのハンドルを呼び出す Kernel 関数のパラメータへあらかじめ設定しておくことで，Kernel 関数内で使用可能にする．このように，処理対象が画素以外の場合も記述することが可能である．

3.3 動画像処理プログラム

3.3.1 従来の RaVioli 記法を用いた動画像処理プログラム

従来の RaVioli を使用して，動画像中のフレームに対してエッジ抽出を行うプログラムを記述する場合，**図 15** のようになる．プログラマはまず 1 枚のフレームに対してエッジ抽出を行う関数 UserProg() を定義する．その関数ポインタを RV_Streaming インスタンスの高階メソッドである proc() に渡すことで，動画像中のすべてのフレームにたいして処理を施すことができる．UserProg() 内ではグレースケール化を行った後，画像の 2 値化を行い，閾値を用いてエッジ抽出を実行している．一方この記述方式を拡張後の RaVioli で同様に使用する場合は，プログラマは**図 16** のように記述する．プログラマはグレースケール化，2 値化，エッジ抽出を行う Kernel 関数を定義し，そのハンドルを RV_Image インスタンスの高階メソッドに順次渡すことで，1 枚のフレームに対する処理を定義する．さらにその UserProg() の関数ポインタを RV_Streaming インスタンスの高階メソッド cudaProc() に渡すことで，CUDA を使用した図 15 と同様の処理が実現できる．しかし拡張後の RV_Image インスタンスの高階メソッド cudaProcPix() 等の中では，メモリの確保や解放および Host-Device 間のデータ転送を行っている．こ

```
void UserProg(RV_Image* img)
{
  img->ProcPix(GrayScale);
  img->ProcPix(Binarize);
  img->ProcNbr(EdgeDetect);
}
int main(){
  RV_Streaming stream;
  stream.proc(UserProg);
}
```

図 15: 従来の RaVioli での動画像処理

```
void UserProg(RV_Image* img)
{
  img->cudaProcPix(GrayScaleK); //(1)
  img->cudaProcPix(BinarizeK);  //(2)
  img->cudaProcPix(EdgeDetectK);//(3)
}
int main(){
  RV_Streaming stream;
  stream.cudaProc(UserProg);
}
```

図 16: 拡張後の RaVioli での動画像処理 1

こで (2) で呼ばれる Kernel 関数の実行では (1) の処理結果画像を使用し，(3) で呼ばれる Kernel 関数の実行では (2) の処理結果画像を使用するはずである．そのため，(1) で実行される処理結果画像の Device から Host への転送処理や Device メモリの開放，(2) で実行される Device メモリの確保や処理対象画像の Host から Device への転送などといった処理は冗長であり，処理速度を低下させる原因となる．そこで本研究では，これらの冗長なメモリ確保や開放，Host-Device 間のデータ転送による処理速度の低下を防ぐ記述方式を提案する．

3.3.2 提案記法を用いた動画像処理プログラム

RaVioli+CUDA 提案記法を用いた動画像処理プログラムとその記法の概念図を**図 17** と**図 18** に示す．まずプログラマは，1 枚のフレームに対する 1 つの処理を記述した関数を複数定義する．この例では，2 値化とエッジ抽出を施す処理が記述された関数を定義している．引数として処理フレームの他に RV_Data インスタンスを受け取っている．RV_Data インスタンスは Device メモリ上の処理フレームへのポインタ等，Device 側のメモリ管理に必要な情報を持つ．ここで，この関数内で使用する高階メソッド cudaProcPix() は画像処理の際に使用する高階メソッドとは異なり，引数として Kernel 関数のハンドルの他に RV_Data インスタンスを受け取る．このメソッド内では実行構成の設定，Kernel 関数の引数の設定および Kernel 関数の呼び出しのみ行われる．この Kernel 関数が実行する処理は，RV_Data インスタンスが持つポインタが指す Device メモリ上のフレームに対して行われる．次に RV_StageVector インスタン

```
RV_CudaDevice device;
RV_Streaming stream;
RV_StageVector stageV;
void Binarize(RV_Image* image, RV_Data* data){
   ...
   image->cudaProcPix(&cuFunction, data);
}
void EdgeDetect(RV_Image* image, RV_Data* data){
   ...
   image->cudaProcPix(&cuFunction, data);
}
int main(int argc, char* argv[]){
   device.RaCudaInit();
   ....
   stageV.push(Binarize);
   stageV.push(EdgeDetect);
   stream.run(&stageV);
   ....
   device.RaCudaExit();
}
```

図 17: 提案記法を用いた動画像処理プログラム

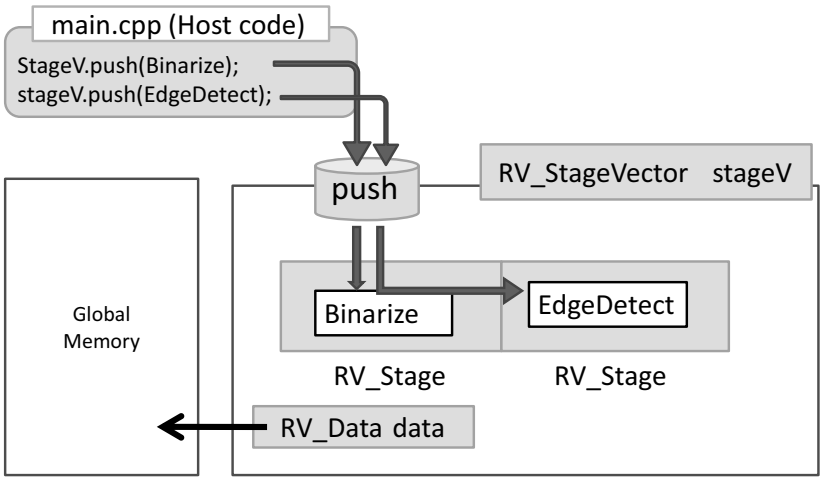

図 18: RaVioli+CUDA を用いた動画像処理記法の概念図

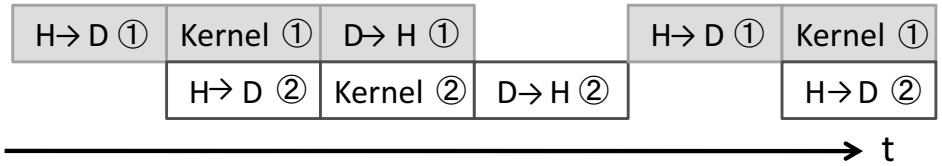

図 19: オーバーラップの概念図

スがもつメソッド push() の引数として，これらの関数のポインタを順に渡す．push() が実行されると，図 18 の概念図に示すように受け取った順に処理ステージが作成される．RV_StageVector インスタンスは，これらの処理ステージの他に，先ほどのべた RV_Data インスタンスを持つ．このように RV_StageVector インスタンスは動画像に対する処理に関連した情報を持つ．

これらの処理ステージがセットされた RV_StageVector インスタンスのオブジェクトを RV_Streaming インスタンスのメソッド run() の引数へと渡す．run() 内ではまず RV_Data インスタンスが持つ Device メモリ管理の情報と RV_Streaming がもつフレームの情報を基に，Global Memory の確保を行う．その後，確保された領域に処理対象画像データを転送し，転送されたデータに対して順に処理ステージを実行する．最後に，最終的な結果画像を Host 側へと書き戻す．このような処理記法を実現することで，冗長なメモリ確保や開放，Host-Device 間のデータ転送を防ぐことが可能になる．

3.3.3 オーバーラップ

CUDA ではストリームという機能を使用することで，データ転送と同時に Kernel 関数の実行を行うことが可能である．本研究では，この機能を使用することで動画像処理の効率化を行う．

Host-Dvice 間の転送と Kernel 関数を同時に実行するには，これらの間に処理順依存がないことが条件である．依存関係がある場合は，実行結果が正しくないものになってしまう．そこで CUDA では，タスク間の処理順を明示的に指定する機構としてストリームというものを提供している．

Host-Device 間の転送や Kernel 関数の実行には特定のストリームを指定することが出来る．転送や Kernel 関数の呼出を行うと，対応するストリームにその処理が登録される．ストリームに属する処理は，必ず登録された順番に 1 つずつ実行される．つまり，同じストリームに登録された処理は，処理順依存があると仮定される．このストリームは複数作成することが可能であり，異なるストリームに登録された処理の間には処理順依存がないと仮定され，片方が Kernel 関数の実行で，もう一方が Host-Device

間の転送であれば，同時実行される．

そこで本研究では，動画像中の連続する 2 枚のフレームの Host-Device 間の転送と Kernel 関数の実行を**図 19** のようにオーバーラップさせることで，処理の高速化を行った．①はストリーム番号 1（以下ストリーム 1）を，②はストリーム番号 2（以下ストリーム 2）を表している．まず，ストリーム 1 がフレームを Host から Device へと転送する．その後転送後のフレームに対して Kernel 関数を実行する．ストリーム 1 に登録されている処理と，ストリーム 2 に登録されている処理には依存関係がないはずなので，ストリーム 1 における Kernel 関数の実行と，ストリーム 2 が行う次のフレームの Host から Device への転送には依存関係がない．そのためこの 2 つの処理はオーバーラップさせることが可能である．次にストリーム 1 は処理結果画像を Device から Host 側へと転送する．このとき同様に，依存関係のないストリーム 2 の Kernel 関数の実行を同時に行う．最後に，ストリーム 2 の処理結果画像を Device から Host 側へと転送する．ここでその後の処理に，ストリーム 1 において次のフレームの Device 側への転送があるが，データ転送同士は同時に実行することが出来ないため，ストリーム 2 が行う処理結果画像の Host 側への転送処理とはオーバーラップさせることができない．

上記の処理を 2 枚のフレーム毎に繰り返すことで，処理の高速化を行った．なおオーバーラップを実現するためには，非同期に動作する異なるストリーム間で行われる処理中に，Host 側によって以下の操作が行われないことが条件である．

- Host 側のページロック・メモリの確保
- デバイスメモリの確保
- デバイスメモリへのデータセット
- Device-Device 間のメモリコピー
- ストリーム 0 での CUDA オペレーションの実行

RaVioli+CUDA を使用して動画像処理を記述することで上記の条件を考慮することなく，2 フレームの Host-Device 間の転送と Kernel 関数の実行のオーバーラップを実現することが可能である．

3.4 ライブラリの仕様

RaVioli では，画素数やフレームレートといった構成要素を隠蔽するために，それぞれの要素配列をカプセル化する．またカプセル化されたインスタンスに対して処理を適用するためのインターフェースとして高階メソッドを提供する．高階メソッドは，構成要素を処理単位とした関数を引数として受け取り，その関数を要素配列の全体に

施すメタ関数である．

この機能を実現するにあたりRaVioliでは，1画素の情報をもつRV_Pixel，1枚の画像情報を持つRV_Image，および動画像全体の情報をもつRV_Streaming等のクラスが実装されている．

以降本節では，従来RaVioliが備える上記のクラスに対し本研究が施した拡張と，今回の提案で追加実装したRV_CudaDeviceクラス，RV_StageVectorクラスおよびいくつかのメソッドについて述べる．

RV_Pixelクラス

画像を構成する画素の情報を保持するクラスであり，RGBの値をまとめて32ビットのメンバ変数として持つ．また1画素のRGBの値を取得するメソッドやRGBの値をセットするメソッド等を持つ．

RV_Imageクラス

画像の実体を表すクラスであり，メンバ変数として画素データ配列や画像の幅，高さ等の画像に対する情報を持つ．CUDAに対応した拡張後のRaVioliでは，静止画像に対して処理を施す際には，画像中のすべての画素データがRV_Pixelインスタンスの配列として一時的に保持される．動画像処理の際には，Host-Device間の転送とKernel関数の同時実行のために，非同期にデータを転送させる必要がある．そのため，画素データ配列はHostのページロック・メモリ上に格納する必要がある．そこで，RV_Pixelインスタンスの配列の他に，動画像処理用にint型の配列をページロック・メモリ上に確保し，そこに画像中のすべての画素データを一時的に保持させることにした．

また，プログラマが定義したKernel関数のハンドルを受け取り，自動でHost-Device間のデータ転送，実行構成の設定を行い，画像に対してKernel関数の処理を施す高階メソッドをいくつか追加実装した．**表3**に今回追加実装した高階メソッドの種類と使用例を示す．またそれぞれの詳細について以下で述べる．ここで(I)は静止画像に対して，(M)は動画像に対して用いる高階メソッドである．

void cudaProcPix(CUfunction* cuFunc)...(I)

void cudaProcPix(CUfunctioni* cuFunc, RV_Data* data)...(M)

閾値判定による2値化や，近傍画素を使用する畳み込み演算などに利用する．プログラマは1画素および近傍画素を処理対象とするKernel関数を定義する．Kernel関数は以下のような引数の構成で定義する必要がある．

$\_\_global\_\_\ void\ kernel(int*\ idata, int*\ odata, int\ width, int\ height)$...(I)(M)

idataは処理対象画像のポインタ，odataは出力画像のポインタ，width，heightは画像

メソッド名	処理単位	使用例
cudaProcPix	1 画素，近傍	2 値化，畳み込み積分
cudaProcBox	1 ウィンドウ	テンプレートマッチング
cudaCompImg	1 画素，別画像の 1 画素	差分検出

表 3: 追加実装した高階メソッドの種類

の幅，高さである．cudaProcPix() は引数としてプログラマが定義した Kernel 関数のハンドルを受け取る．メソッド内では処理対象画像サイズ分のメモリを Device 側に確保し，画素データを Device 側のメモリ idata へとコピーする．その後，実行構成や Kernel 関数の引数を設定し，idata に対して Kernel 関数を実行する．最後に処理結果を odata へと格納し，再びそのデータを Device 側から Host 側へと書き戻す．

cudaProcBox(CUfunction* cuFunc, RV_Length width,
RV_Length height)...(I)
cudaProcBox(CUfunction* cuFunc, RV_Length width,
RV_Length height, CUfunction* reduction)...(I)'
cudaProcBox(CUfunction* cuFunc, RV_Data* data,
RV_Length width, RV_Length height)...(M)
cudaProcBox(CUfunction* cuFunc, RV_Data* data,
RV_Length width, RV_Length height,CUfunction* reduction)...(M)'

テンプレートマッチング等に利用する．プログラマは 1 ウィンドウを処理対象とする Kernel 関数を定義する．Kernel 関数は以下のような引数の構成で定義する必要がある．

$$\_\_global\_\_void\ kernel(int*\ idata, int\ width, int\ height, int\ widthBox, int\ heightBox)...(I)(M)$$

$$\_\_global\_\_void\ kernel(int*\ idata, int4*\ reductionData, int\ width, int\ height, int\ widthBox, int\ heightBox)...(I)'(M)'$$

(I)(M) はそれぞれリダクション処理が必要ない場合の画像処理用，動画像処理用の Kernel 関数である．また，(I)'(M)' はそれぞれリダクション処理が必要な場合の画像処理用，動画像処理用の Kernel 関数である．idata は処理対象画像のポインタ，width,

height は処理対象画像の幅，高さ，widthBox，heightBox はウィンドウの幅，高さ，また reductionData は 1thread のローカルな結果を保存するための配列のポインタを表す．

cudaProcBox() は引数としてプログラマが定義した Kernel 関数へのポインタ，処理対象のウィンドウの幅，高さを受け取る．メソッド内では，cudaProcPix() と同様に Host-Device 間の転送，実行構成の設定等を行う．またリダクション処理が必要な場合は，リダクション用に定義した Kernel 関数のハンドルを渡す必要がある．

cudaProcCompImg(CUfunction* cuFunc, RV_Image* image)...(I)

2 枚の画像間の差分検出などに利用する．プログラマは，比較元と比較対象の画像それぞれに対応する 2 つの画素を処理対象とする Kernel 関数を定義する．Kernel 関数は以下のように定義する．

$$\_\_global\_\_ void\ kernel(int*\ idata, int*compdata, int*odata, \\ int\ width, int\ height,)...(I)$$

idata は処理対象画像のポインタ，compdata は比較対象画像のポインタ，odata は出力画像のポインタ，width，height は処理対象画像の幅，高さを表す．

cudaProcImgComp() は引数として，上記の Kernel 関数のハンドル，比較対象である画像 image を受け取る．メソッド内では，処理対象画像および比較対象画像の Host-Device 間の転送，実行構成の設定，Kernel 関数の呼び出し等を行う．

RV_Data クラス

Device メモリ上の処理対象画像，出力画像へのポインタおよび，リダクション処理の際に必要となる thread ローカルな結果が格納される Device メモリ上の配列のポインタ等の Device 側のメモリ管理の情報を持つ．

RV_StageVector クラス

RV_StageVector クラスの概念図は図 18 で既に示した．RV_StageVector クラスは，処理ステージが格納される配列，および Device メモリの管理情報を持つ RV_Data インスタンス，またオーバーラップの際に必要なストリームの情報等，動画像の処理に関する情報を持つ．1 つの処理ステージは RV_Stage インスタンスとして定義される．RV_Stage インスタンスは図 18 に示す通り，現在のところ，フレームに対する処理が記述された関数へのポインタ（図 18 の例では Binarize と EdgeDetect）のみを持つ．

void push(void (*UserProgram)(RV_Image*, RV_Data*))

push メソッドは関数ポインタ UserProgram を引数に取るメンバ変数である．入力フレームに対し UserProgram 処理を適用するステージを作成し，図 18 に示すように配列へと格納する．

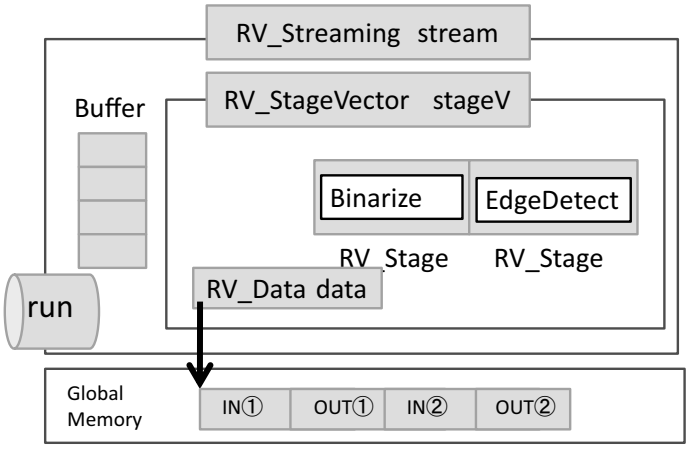

図 20: run メソッドの概念図

RV_Streaming クラス

動画像を処理するクラスであり，カメラから画像をキャプチャするメソッドと，数枚のフレームを配列で格納するリングバッファ，および動画像に対して処理を施すメソッドを持つ．取り込んだフレームは一時的にリングバッファに保存する．その後，取り込んだ複数枚のフレームから，処理フレームレートの解像度ストライドに応じた間隔でフレームのデータを Device 側のメモリへ転送する．またその際に，現段階での処理画素数の解像度ストライドを RV_Image インスタンスに格納する．

void start()

このメソッド内では，始めに，フレームに対して処理を実行するメインスレッドとは別に，カメラから画像をキャプチャする専用のスレッドを生成する．その後フレームを一時的に格納するリングバッファの領域として，Host 側にフレーム複数枚分のページロック・メモリを確保する．ページロック・メモリは Host-Device 間のデータ転送を，高速かつ非同期に行うことが可能な領域である．その後，ビデオから取り込んだフレームをこのリングバッファへと順次格納する．

void run(RV_StageVector* stageV)

run メソッドは動画像処理を開始するメソッドである．run メソッドが呼ばれた際の動作を説明するために，**図 20** を用いる．このメソッドが呼ばれた場合，まず RV_Data インスタンスの持つ Device メモリ上の処理対象画像と出力画像へのポインタを基に，これらの画素配列を格納する領域をそれぞれ Global Memory 上に確保する．なおこの際，オーバーラップにおいて 2 つのストリームが非同期に Host-Device 間の転送や Kernel 関数の実行を行うために，図 20 に示すように 2 倍の量のメモリを確保する．Global

Memory の確保の後，Buffer に格納されているフレームを 1 枚取り出し，その画素配列を Host から Device へと転送する．その後，引数として受け取った RV_StageVector インスタンスの持つフレームに対する一連の処理を，順次 Global Memory 上の処理対象画像へと適用する．最後に処理フレームを Device から Host へと書き戻す．連続する 2 枚のフレームに対する Host-Device 間の転送と処理の実行は前章で述べたように自動的にオーバーラップされる．2 フレームの処理が終了した後，優先度ストライドの値を元に各解像度ストライドを変更する．

RV_CudaDevice クラス

デバイス，コンテキスト，およびモジュールのハンドルを持つ．またデバイスの初期化，モジュール内のテクスチャ参照変数やカーネル関数のハンドルを取得するメソッドを持つ．

void RaCudaInit()

デバイスの初期化，コンテキストの生成，モジュールのロードを行う．

void RaCudaExit()

コンテキストの破棄，モジュールのアンロードを行う．

void GetKernelHundle(CUfunction* cuFunc, char* kernel)

モジュール内で定義された Kernel 関数のハンドルを取得する．

[**void GetTexrefHundle(CUtexref* cuTexref, char* kernel)**

モジュール内で定義されたテクスチャ参照変数のハンドルを取得する．

4 トランスレータ

4.1 トランスレータの方針

従来の RaVioli で記述されたプログラムを，RaVioli+CUDA で記述されたプログラムへと変換するトランスレータを提案する．トランスレータの概要を**図 21** に示す．プログラマは従来の RaVioli で記述したプログラム program.cpp を本研究で提案するトランスレータに渡すことで，Host 側で実行されるプログラム program.cpp と Device 側で実行されるプログラム kernels.cu へと変換することができる．これらの 2 つのプログラムは RaVioli が提供する makefile によって，図 21 のように実行ファイルへとコンパイルできる．

従来の RaVioli で記述されたプログラム**図 22** は，おおよそ**図 23** のように変換される．ドライバ API を使用するにあたり，まず Driver の初期化とコンテキストの生成などが必要となる．またすべての処理の後にはコンテキストの削除などが必要となる．そ

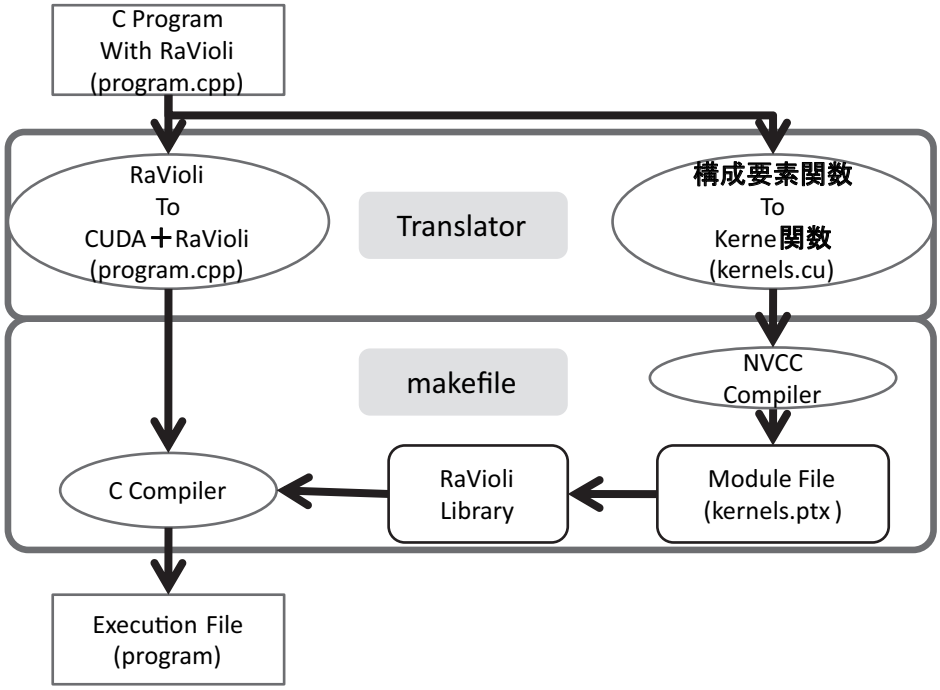

図 21: トランスレータの概要

こですべての処理の前 (図 23，13 行目) と後 (図 23，18 行目) にそれぞれ RaCudaInit() と RaCudaExit() を挿入する．また構成要素関数は Kernel 関数へと変換される．その際 kernels.cu というファイルを作成し，そのファイル内に kernel 関数を書き出す．RaVioli+CUDA は，この Kernel 関数のハンドルを引数として受け取り，この処理を自動的に対象画像全体に施すメソッドを持つ．そこでトランスレータは，構成要素関数のポインタを引数に受け取る高階関数を，Kernel ハンドルの取得とそれを引数に受け取る高階メソッドへと変換する．以下の節では，具体的な変換例を用いて，トランスレータによる構成要素関数から Kernel 関数への変換について述べる．

4.2　構成要素関数から Kernel 関数への変換の基本方針

構成要素関数から Kernel 関数への基本的な変換について，グレースケール化の変換例を用いて説明する．**図 24** と**図 25** に，それぞれ変換前の構成要素関数と変換後の Kernel 関数を示す．この処理では，1 画素の RGB 値の平均を取ることでカラー画像をモノクロ画像へと変換している．

まずトランスレータは，モジュールファイルを作成するために kernel.cu を生成する．そして変換後の Kernel 関数を順次このファイルへと書き出す．変換後の関数の型指定子と関数宣言子は 2，3 行目のようになる．Host 側から呼ばれ Device 側で実行される

従来の RaVioli で記述された画像処理プログラム

```
1   RV_FileHandler file;
2   void UserProgram(RV_Pixel* p1){
3     // 1 画素に対する処理
4   }
5   int main(int argc, char* argv[]){
6      RV_Image* image;
7      file.readBMP(argv[1], image);
8      image->proc(UserProgram);
9      file.writeBMP("output.bmp",image);
10  }
```

図 22: トランスレータの基本方針（変換前）

トランスレータで変換後の画像処理プログラム

```
1   /* kernel.ptx （モジュール） */
2   extern "C" __global__ void
3   UserProgram_kernel(int* idata,int* odata,int wid, int hei){
4     // 1thread の処理
5     // 1 画素に対する処理
6   }
7   /* main.cpp */
8   RV_FileHandler file;
9   RV_CudaDevice device;
10  int main(int argc, char* argv[]){
11     RV_Image* image;
12     file.readBMP(argv[1],image);
13     device.RaCudaInit();
14     CUfunction cuFunction;
15     device.GetKernelHundle(&cuFunction,"UserProgram_kernel");
16     image->cudaProc(&cuFunction);
17     file.writeBMP("output.bmp",image);
18     device.RaCudaExit();
19  }
```

図 23: トランスレータの基本方針（変換後）

```
1   void Color2Gray(RV_Pixel* p1){
2      int ave=((p1->getR())+(p1->getG())+(p1->getB()))/3;
3      p1->setRGB(ave,ave,ave);
4   }
```

図 24: 従来の RaVioli で記述された画像処理プログラム

```
1   /* kernel.cu （モジュール） */
2   extern "C" __global__ void
3   Color2Gray(int* idata,int* odata,int width, int height){
4      int x=blockDim.x*blockIdx.x+threadIdx.x;
5      int y=blockDim.y*blockIdx.y+threadIdx.y;
6      int rgb;
7      if(x<width && y<height){
8        rgb=idata[y*wid+x];
9        int ave=(getR(rgb)+getG(rgb)+getB(rgb))/3;
10       setRGB(&odata[y*wid+x],ave,ave,ave);
11     }
12  }
```

図 25: トランスレータで変換後の画像処理プログラム

Kernel 関数の型指定子の前には”__global__”指定子をつける必要がある．Kernel 関数の種類には他に Device 側から呼ばれ Device で実行される関数が存在し，この場合には型指定子の前に”__device__”をつける．関数宣言子中の引数は，変換前の構成要素関数を渡される高階メソッドの種類によって決まる．グレースケール化の場合は高階メソッド cudaProcPix() の引数として Color2Gray を指定する．この場合は，3 行目のように引数を与える．左から処理対象画像配列 idata，処理結果を格納する配列 odata，画像の幅 wei，高さ hei である．

変換後のコード図 25 の 4 行目〜8 行目，11 行目は高階メソッド名に応じて生成される．Kernel 関数は 1thread の処理が記述された関数である．ここではコアレッシングアクセスが可能なように 1thread が 1 画素を処理し，かつそれぞれの隣り合う thread は隣り合う画素を処理するよう変換されている．このようにすることで，隣り合う 16thread の Global Memory へのアクセスを並列に実行することが可能になる．また当該 thread が担当する画素値は変数 rgb へと格納し，その後の処理では変数 rgb を使用することとした．これは当該画素へとアクセスする際に，Global Memory 上に存在する idata へ毎回アクセスするよりも，当該画素の値を Register 変数 rgb へと一端格納し，そこへ毎回アクセスするほうがより高速であると考えられるからである．さらに，Grid 内の thread サイズが処理対象画像のサイズ以上の場合に，Kernel 関数が画像の範囲外にアクセスしないように，図 25 の 7 行目と 11 行目に条件式が挿入される．

次に変換前コード図 24 の 2，3 行目の変換について述べる．それぞれのコードは変換後のコード図 25 の 9，10 行目へと変換される．getR()，getG()，getB()，setRGB() は，あらかじめ本研究で用意した Kernel 関数である．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 setRGB() は RGB 値を受け取り画素配列の 1 要素にセットする関数である．ここでは 12 行目で求めた値 ave を，処理結果画像を格納する配列 odata の 1 つの RGB 値としてセットしている．

本研究で提案するトランスレータは，上述のように，従来の RaVioli で記述された構成要素関数を，CUDA の Device コードである Kernel 関数へと変換することが可能である．

4.3 リダクション処理の生成を含む変換

4.3.1 ループをまたがる依存関係の解析手法

Kernel 関数を定義する際，1thread が 1 画素に対して処理を施すようにしてきた．実際にこの Kernel 関数が呼び出される場合には，設定された Grid サイズ分の thread がそれぞれ 1 画素に対する処理を並列に実行することで，処理の高速化が実現できる．ここで，C 言語で画像処理を記述する際には，1 画素に対する処理はループイテレーションで画像全体に繰り返し適用される．このときループ間にまたがる依存関係が存在すると仮定する．注目画素に対して処理を施したい場合に，前回の処理結果が必要となるため，並列化をした場合には処理結果の正当性を保証することができない．そのため，ループにまたがる依存関係が存在する場合には，並列化をすることは適切ではない．しかし，ループイテレーション内の演算に結合律と交換律が共に成り立つ場合，リダクション処理を行うことで並列化した場合にも正しい結果を得ることが可能となる．リダクション処理とは，並列数分用意した一時的な格納領域に対して thread ローカルな処理結果を格納し，その処理結果を最後に統合することで，全体の最終的な結果を求める処理である．

イテレーション間にまたがる依存関係は，RaVioli 既存のマルチコア CPU 上における自動並列化機構 [3] と同様な手法で発見することが可能である．RaVioli はループイテレーションをライブラリ内で管理しており，プログラマに構成要素に対する処理の順序を規定させない．そこで既存手法では，現段階において以下の場合にループ間に依存関係があるとしている．

(A) 大域変数に対して読み出しはなく書き込みのみである
(B) 読み書きのある大域変数に対して+もしくは-を使用し，なおかつ * もしくは / を使用している
(C) if 文の条件式で使われている大域変数に対して，比較した変数と異なる値が代入されている
(D) 四則演算を施した大域変数を画素へ書き込んでいる
(E) ライブラリレベルで定義されている関数の引数に大域変数を使用している（RaVioli の関数は除く）

これらの 5 つのうちどれかひとつでも検出されれば逐次と並列処理の結果に一貫性がないと判断される．一方，同じ大域変数に対して読み出しと書き込みの両方が行われている場合には，その大域変数が reduction 演算を使用することで並列化可能であると判断する．

```
1   RV_Image* imageTP;
2   RV_Coord start;
3   RV_Coord end;
4   int sad;
5   void SAD(RV_Pixel* p1,RV_Pixel p2){
6     int abs=p1->absDiff(p2);
7     sad+=abs;
8   }
9   void TPmatching(RV_DoppelImage* imageSmall,
10                  RV_Coord startNow,RV_Coord endNow){
11    sad=0;
12    int min=INT_MAX;
13    imageSmall->procImgComp(SAD, imageTP);
14    if(min>sad){
15      min=sad;
16      start=startNow;
17      end=endNow;
18    }
19  }
```

図 26: 変換前:テンプレートマッチング

4.3.2 変換手法

リダクション処理の生成を含む変換について，テンプレートマッチングの変換例を用いて説明する．**図 26** に変換前の構成要素関数，**図 27** および**図 28** に変換後の Kernel 関数を示す．テンプレートマッチングは，処理対象画像からテンプレート画像と最も類似した箇所を探索する処理を施す．順次テンプレート画像をずらして，処理対象画像中の部分画像との類似度を求めることでこの処理を実現する．ここで類似度はテンプレート画像中の画素と，部分画像中の画素の両画素の差分の総和であり，値が小さいほど類似度が高い．

従来の RaVioli を使用してテンプレートマッチングのプログラムを記述する場合，図 26 のように 2 つの構成要素関数を定義する．TPmatching は 1 ウィンドウ画像 im-

```
1  /* kernel.cu（モジュール）*/
2  texture<int, 2, cudaReadModeElementType> texTP;
3  __device__ int
4  SAD(int* idata, int wid, int hei, int widBox, int heiBox,
5      int x, int y){ int sad=0;
6    int rgb,rgb2;
7    for(int j=0; j<heiBox; j++){
8      for(int i=0; i<widBox; i++){
9        rgb=idata[(y+j)*w+(x+i)]; rgb2=tex2D(texTP,i,j);
10       int abs=absDiff(rgb,rgb2);
11       sad+=abs;
12     }  }
13   return sad;
14 }
15 /* main.cpp */
16 extern "C"
17 __global__ void
18 TPmatching_kernel(int* idata, int4* data4reduction,
19               int wid, int hei, int widBox, int heiBox){
20   int x=blockDim.x*blockIdx.x+threadIdx.x;
21   int y=blockDim.y*blockIdx.y+threadIdx.y;
22   int incX=gridDim.x*blockDim.x; int incY=gridDim.y*blockDim.y;
23   int sad;  int min=INT_MAX;
24   for(int j=y; j<(hei-heiTP); j+=incY){
25     for(int i=x; i<(wid-widBox); i+=incX){
26       sad=SAD(idata, wid, hei, widBox, heiTP, i, j);
27       if(sad<min){
28         data4reduction[y*256+x].z=sad;
29         data4reduction[y*256+x].x=i;
30         data4reduction[y*256+x].y=j;
31       }    }  }
32 }
```

図 27: 変換後:テンプレートマッチング

```
1  extern "C"
2  __global__ void
3  reduction_kernel(int4* data4reduction, int4* g_odata){
4    __shared__ int sdatax[256];
5    __shared__ int sdatay[256];
6    __shared__ int sdataz[256];
7    //Global Memory から Shared Memory へ
8    unsigned int tid=threadIdx.x;
9    unsigned int i=blockIdx.x*blockDim.x+threadIdx.x;
10   sdatax[tid]=data4reduction[i].x;
11   sdatay[tid]=data4reduction[i].y;
12   sdataz[tid]=data4reduction[i].z;
13   __syncthreads();
14   //Shared Memory 上でリダクションを計算
15   for(unsigned int s=blockDim.x/2;s>0;s>>=1){
16     if(tid<s){
17       if(sdataz[tid]>sdataz[tid+s]){
18         sdatax[tid]=sdatax[tid+s];
19         sdatay[tid]=sdatay[tid+s];
20         sdataz[tid]=sdataz[tid+s];
21       }
22     }
23     __syncthreads();
24   }
25   if(tid==0){
26     g_odata[blockIdx.x].x=sdatax[0];
27     g_odata[blockIdx.x].y=sdatay[0];
28     g_odata[blockIdx.x].z=sdataz[0];
29   }
30 }
```

図 28: 変換後 2:テンプレートマッチング

ageSmall を処理対象とする構成要素関数であり，procBox() の引数として渡すことで画像全体に処理を施すことが可能である．ここで，1 ウィンドウ画像 imageSmall に対する処理を記述するには，1 ウィンドウ内の 1 画素に対する処理を記述した関数を定義して，RV_DoppelImage インスタンスの高階メソッドに渡す必要がある．ここで RV_DoppelImage は RV_Image の情報に加えて，処理対象画像中の処理対象ウィンドウの開始位置の情報等を持つ．そのため procBox() を使用する場合，プログラマはこのように 2 つの関数を定義する必要があった．

テンプレートマッチングは並列化をした際にリダクション処理が必要となる．そのため変換後の Kernel 関数は，図 27 図 28 のようにリダクション処理を含む 3 つの関数となる．図 28 はリダクション処理が記述された Kernel 関数である．まずは図 27 に示した，リダクション処理以外の部分の変換について述べる．

変換前の構成要素関数である SAD と TPmatching は，変換後の SAD と TPmatching_kernel に対応する．まず，TPmatching_kernel は Host 側から呼ばれる Kernel 関数であるため，16〜18 行目のように__global__指定子が付けられて宣言される．一方 SAD は TPmatching_kernel 内から呼び出されていることから，Device 側から呼ばれ，Device 側で実行される Kernel 関数であるため，3〜4 行目のように__device__指定子が付けられて宣言される．TPmatching_kernel の引数は左から，処理対象画像 idata，thread ローカルな結果が格納される配列 data4reduction，処理対象画像の幅 wid，高さ hei，ウィンドウの幅 widBox，高さ heiBox である．また SAD の引数は左から処理対象画像 idata，処理対象画像の幅 wid，高さ hei，ウィンドウの幅 widBox，高さ heiBox，処理対象であるウィンドウの左上の座標 x，y である．ここで TPmatching_kernel の 20〜22 および 24，25，31 行目，SAD の 6〜9 および 12 行目はトランスレータによって高階メソッド名に応じて生成される．

変換前の 6 行目のコードは，変換後の 10 行目のコードへと変換することが可能である．変換前のコードである p1->absDiff(p2) は，画素 p1 と p2 の R，G，B の値の差の絶対値をそれぞれ求め，その和を返すメソッドである．変換後の absDiff は画素値 rgb，rgb2 に対して同様の処理を施す．また absDiff は本研究であらかじめ用意した Kernel 関数のひとつである．変換後の 26 行目では SAD 関数を呼び出し，その返り値である sad を受け取っている．モジュール内では大域変数を使用することができないため，このように返り値として受け渡しを行う必要がある．

次に変換前の 13〜17 行目のコードは，変換後の 27〜31 行目へと変換される．前節で述べた手法により，この部分はリダクション演算を施すことで並列化可能である．そ

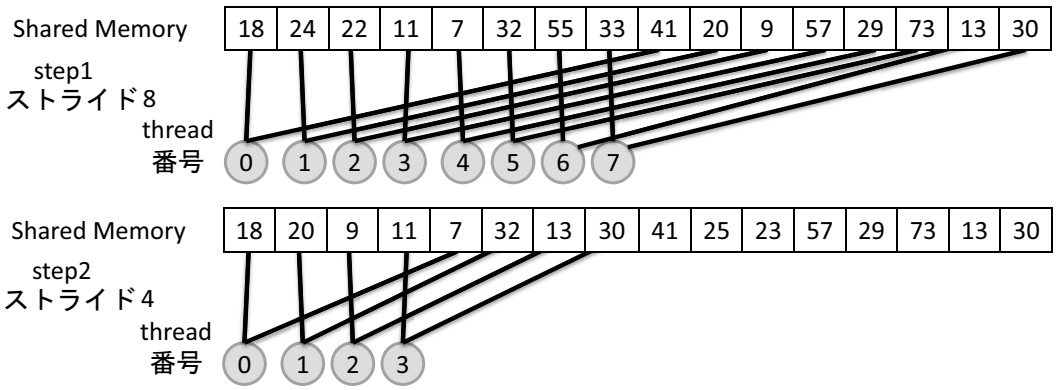

図 29: 連続した Shared Memory へのアクセス

こで thread ローカルな結果を格納する配列に，当該 thread の結果を格納するように変換する．

最後に，図 28 に示した thread ローカルな結果をまとめる処理の生成について述べる．図 28 は Shared Memory を用い，Block 内の thread 間でデータのやりとりをすることで Block 内での最小値とそのときの座標を求める Kernel 関数である．この Kernel 関数を 3.2.1 節の図 14 のように複数回呼び出すことで，thread ローカルな結果をまとめ，最終的な答えを導く．

この処理は Host 側から呼ばれるため，__global__指定子を付けて宣言する．引数は左から thread ローカルな結果が格納されている配列 data4reduction，Block 内でまとめた結果を格納する配列 g_odata である．ここで thread ローカルな結果が格納されている配列は int4 型で宣言されている．これは CUDA に存在するベクトル型の変数である．「int4 data」と宣言された場合には，data.x，data.y，data.z，data.w とすることでそれぞれの値にアクセス可能である．そのため，リダクション用の配列の型を int4 とすることで，4 つまでの値をリダクション処理用に用いることが可能である．4〜13 行目では，Block 内の thread 間で共有するシェアード・メモリを確保し，Global Memory 上にある thread ローカルな結果をそこに格納している．また Shared Memory へのアクセスは 32 ビット単位で行われるため，int 型の配列へ data4reduction の要素をそれぞれ代入する必要がある．Shared Memory にデータを格納したあとは，Block 内の全 thread で同期を取る．15〜24 行目が Shared Memory 上でのリダクション計算になる．トランスレータは変換前のコードの 13〜17 行目を元にこのコードを生成する．

ここで 2.3.3 節で述べたように，CUDA の最適化の際には Warp ダイバージェントの回避と，Shared Memory を使用する際にはバンクコンフリクトの回避が必要である．

表 4: 評価環境

OS	Fedora9
CPU	Core2Quad
Frequency	2.83GHz
Memory	3GB
GPU	GeForce GTX280
Number of multiprocessors	30
Number of cores(SP)	240
CUDA version	2.2(Driver API)
compute capability	1.3
コンパイラ	gcc
最適化オプション	-O3

これらを回避するために，15〜24 行目のようなコードを生成する．このとき 1Block 中の thread のシェアード・メモリに対するアクセスは**図 29** のようになる．このとき，16 thread が連続する Shared Memory にアクセスするため，バンクコンフリクトは回避できたといえる．また，Warp ダイバージェントも回避できている．

TPmatching_kernel() の関数ポインタを受け取る高階メソッドである cudaProcBox() は，さらにこの reduction_kernel() を受け取る．cudaProcBox() の内部では TPmatching() を実行した後，ここで求められた thread ローカルな結果を元に，reduction_kernel() を複数回呼び出すことによって最終的な結果を求める．

5 評価

RaVioli+CUDA を用いて記述したプログラムの処理速度と，トランスレータの適用範囲について評価を行った．評価環境は**表 4** に示す．GPU として NVIDIA 社の GeForce GTX280 を使用した．GeForce GTX280 は 30 個のストリーミング・マルチプロセッサ (SM) を搭載している．さらに各 SM 上にはそれぞれ 8 個のストリーミング・プロセッサ (SP) が搭載されており，計 240 の SP を持つ．

表 5: 画像処理の速度比較 (ms)

プログラム名	w/o RaVioli	w/ RaVioli	w/ RaVioli+CUDA
GrayScale	0.841	8.208	1.322
EmbossFilter	1.497	118.327	1.432
TPmatching	1898.223	10453.549	63.460

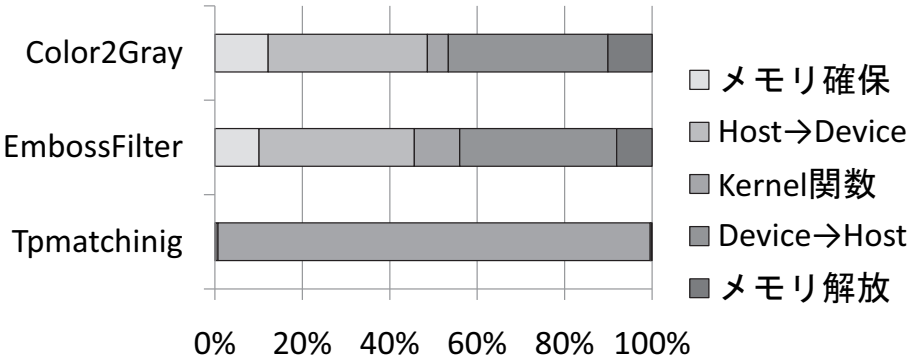

図 30: RaVioli+CUDA の処理時間の内訳

5.1 画像処理の速度比較

サンプルプログラムを用いて，本研究で提案した RaVioli+CUDA で記述した静止画像処理の処理時間を，RaVioli 不使用時，従来の RaVioli 使用時それぞれの場合の処理時間と比較して，評価を行った．その結果を**表 5** に示す．用いたサンプルプログラムは上から，グレースケール化，エンボスフィルタ，およびテンプレートマッチングの 3 種類である．w/o RaVioli は RaVioli 不使用時，w/ RaVioli は従来の RaVioli 使用時，w/ RaVioli+CUDA は本研究で拡張後の CUDA に対応した RaVioli 使用時の処理時間を表す．ここで w/o RaVioli の処理時間は，画像の入出力の時間は含めず，画像に対する処理時間のみを計測したものであり，w/ RaVioli および w/ RaVioli+CUDA の処理時間は高階メソッド呼び出しの実行時間である．またグレースケール化，エンボスフィルタでは 512×512 の画像，テンプレートマッチングでは 395×372 の処理対象画像，70×72 のテンプレート画像を使用した．

表 5 に示すように，すべてのプログラムにおいて既存の RaVioli からの速度向上が達成でき，グレースケール化，エンボスフィルタ，テンプレートマッチングそれぞれにおいて，約 6.2 倍，82.6 倍，164.7 倍の処理速度の向上が確認できた．しかしグレースケール化おいては，C 言語で記述したプログラムよりも速度が低下してしまってい

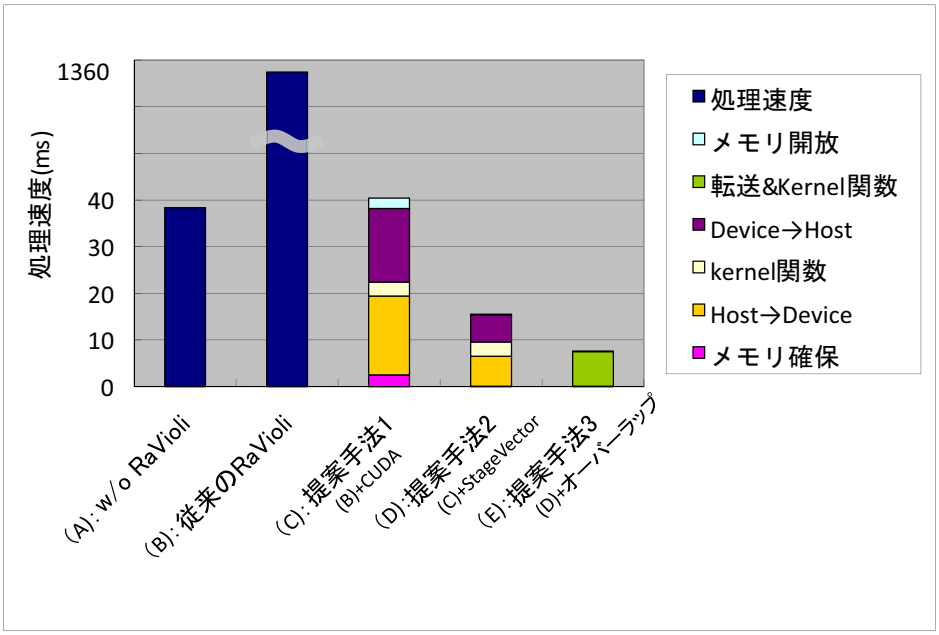

図 31: 10 フレームの処理速度の比較

る．これは Device 側のメモリ確保や画像データの転送時間を無視できないくらいに，サンプルプログラムの画像に対する処理自体の実行時間が非常に小さいためだと考えられる．**図 30** に RaVioli+CUDA で記述されたプログラムそれぞれの処理時間の割合を示した．グレースケール化およびエンボスフィルタでは，実際の画像に対する処理である Kernel 関数の実行時間は全体の約 10 分の 1（0.1ms）前後であり，高速化されていることが確認できた．しかし CUDA 使用前のサンプルプログラムの処理時間が非常に小さかったため，メモリ確保や Host-Device 間のデータ転送に掛かる時間を無視できず，全体の処理時間で見ると CUDA を使用した処理の高速化を確認することができなかった．一方テンプレートマッチングでは，CUDA 使用前のプログラムでの処理時間が非常に大きかったため，CUDA を使用することで効果的に高速化を行えたことが確認できた．

5.2 動画像処理の速度比較

動画像処理の処理速度について，1 枚のフレームに対してグレースケール化，2 値化，エッジ抽出を連続して行うプログラムを用いて評価を行った．また今回の評価では，10 枚のフレームに対する処理時間とその内訳を調べた．評価結果を**図 31** に示す．(A) は RaVioli 不使用時の場合であり，(B) は従来の RaVioli 使用時の場合であり，(C) から (E) は CUDA に対応した RaVioli を用いた場合である．さらに (C) は 3.3.1 で述べ

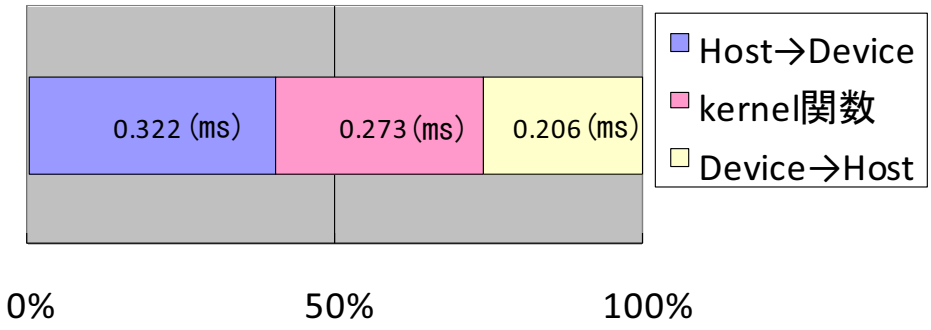

図 32: 1 ストリームにおける処理の内訳

た従来の RaVioli 記法を用いた場合であり，(D) は 3.3.2 で述べた提案記法を用いた場合であり，(E) は (D) に 3.3.3 で述べたオーバーラップを追加実装した場合である．

(C) では，メモリの確保や解放，Host-Device 間のデータ転送に掛かる時間が処理の大半を占めている．これは RV_Image インスタンスの高階メソッドが呼び出される度に，メモリの確保や解放，Host-Device 間のデータ転送が行われていたためだと考えられる．一方 (D) では，これらの冗長な操作がなくなり処理時間が 2 倍以上高速化されているのが確認できた．

(E) では，(D) にオーバーラップの処理が追加実装されている．Host-Device 間のデータ転送と Kernel 関数の実行のオーバーラップにより，実際に処理時間が短縮されていることが確認できた．ここで 1 つのストリームにおける，1 枚のフレームに対する処理時間の内訳を**図 32** に示す．2 枚のフレームの処理のオーバーラップを考えた場合，1 枚目のフレームに対する Kernel 関数の実行と 2 枚目のフレームの Host から Device への転送，およびその後の 1 枚目のフレームの Device から Host へのデータ転送と 2 枚目のフレームに対する Kernel 関数の実行はオーバーラップされる．また今回の処理では，Host-Device 間のデータ転送に掛かる時間と 1 枚のフレームの処理に掛かる時間がほぼ同じ程度であった．そのため，全体として処理時間を 3 分の 2 程度に減少できたと考えられる．またオーバーラップの際には，Host 側のページロック・メモリを使用していることで，実際にはさらに処理時間を短縮することができていることが確認できる．

5.3 トランスレータの適用範囲

本研究で提案してきたトランスレータは，実際にはまだ未実装であり，検討段階である．しかし，本論文でいくつかの例を挙げることにより，従来の RaVioli から CUDA に対応した拡張後の RaVioli へと変換することが可能であることを確認した．本論文では，グレースケール化とテンプレートマッチングの変換の例を掲載した．グレースケール化の変換例を通して，procPix() を用いる，処理対象が画素である画像処理プログラムの変換が可能であることを示した．また，テンプレートマッチングの変換例を通して，procBox() を用いる，処理対象がウィンドウである画像処理プログラムの変換が可能であることを示した．またテンプレートマッチングでは，ループイテレーション間にまたがる依存関係がある場合にも，その依存関係を解析し，ある条件を満たした処理はリダクション処理を適用することで変換可能であることを示した．RaVioli は，提供するインターフェースにより，プログラムの記述段階では全順序化されないという利点がある．これは画像の幅や高さ，画素配列データといった画像に関する情報をプログラマから隠蔽することによって，ループイテレーションの処理をライブラリ内で行っているからである．このことによって，依存解析が容易である．またリダクション処理の生成を含む画像処理プログラムの変換では，Shared Memory を使うことで，GPU を有効に利用可能なプログラムへと変換することが可能であると考えられる．

また，エッジ抽出やエンボスフィルタ等の近傍画素を用いた画像処理プログラムの変換も可能であると考えている．これはグレースケール化と同様に画素毎に独立した処理を行っているためである．一方近傍画素を用いた処理では，隣り合う thread が同じ画素にアクセスする．そのため，Shared Memory を有効に使用可能であると考えている．これについては今後の課題として検討していく必要がある．

また動画像処理プログラムの変換についても検討段階である．しかし RaVioli で記述される動画像処理プログラムは，フレーム 1 枚もしくは時系列で隣り合うフレーム 2 枚を用いた処理を行うため，画像処理に対する変換手法と同じアルゴリズムを用いればよい．よって動画像処理への適用は容易に実現可能である．

6 関連研究

6.1 画像処理の抽象化

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に VIGRA[6] がある．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．また MAlib[7] や OpenCV[8][9] は，画像処理の一般的なアルゴリズムを C の関数や C++のメソッドとして提供している．特に OpenCV は Video4Linux2 を使用することにより，IEEE1394 カメラ経由のデータに対するリアルタイム処理も提供している．

一方 RaVioli は，処理量に直接関係する 1 フレームの構成画素数やフレームレートをプログラマから隠蔽することを目的としている．このため，従来の抽象化ライブラリのように単純に処理内容を抽象化してユーザライブラリの形で提供するものとは完全に異なっている．プログラマは画像に対する処理内容を，解像度を考慮することなく記述できることから，従来の抽象化ライブラリと比べて，動画像処理の詳細なアルゴリズムを簡単に実現することが可能になる．

また金井らは数式エディタを用いて記述できる独自の記述言語を用いることにより，画像処理プログラミングを抽象化している [10][11]．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しループレスな記述ができるという点は RaVioli と似ているが，この記述言語は構成画素数を明示的に指定する必要があるため，RaVioli で行っている動的な構成画素数の変動には対応していない．

6.2 処理時間の自動調整

トレードオフの関係にある処理精度と処理時間を動的に調整するアプローチとして，複数アルゴリズムの切り換え手法がある．たとえば，指定した計算時間を超過すると，その時点で得られている不完全な中間結果を計算結果として採用するといった，計算時間の長さに応じて精度が向上するモデル（Imprecise Computation Model）[12] が提案されている．またこのモデルに基づき，処理精度および処理時間に関して経験的に得た知識を利用することで，プログラマがあらかじめ記述した複数のアルゴリズムから，状況に応じて適したアルゴリズムを動的に選択する信頼度駆動アーキテクチャも提案されている [13][14]．しかしこの方法では，処理を計算負荷の異なる複数のアルゴリズムで実装する必要があり，依然プログラマに対する負担は大きい．

6.3 CUDA

CUDA プログラムの自動最適化のために，コンパイル時に変換する方法が提案されている [15]．この手法により，効率のよいグローバル・メモリへのアクセスを行うプログラムへと変換させることが可能である．ここで提案されているコンパイラの枠組は，多体面コンパイラモデルを使用してアフィンループネストを最適化する．本研究でも，

まだイレギュラーなループには対応していないが，RaVioli はループイテレーション自体の管理をライブラリ内で行っているため，イレギュラーなループにも対応しやすい．

また CUDA-lite[16] というトランスレータが提案されている．これはグローバル・メモリデータの最適なタイル表示のためのコードを生成する．CUDA-lite はプログラマがプラグマによって提供する情報に基づいて変換を行う．また同様にプラグマを使用して変換する方法 [17] がいくつか提案されている．しかしこれらを用いる場合には，並列化可能な部分を考慮してプログラムを作成する必要があり，またリダクション演算などを意識しなければならない．一方で RaVioli はループ処理をライブラリ内で行っているので，並列化可能な箇所を抽出することが容易であり，また本研究で示したように，イテレーション間にまたがる依存関係の解析も可能である．

7　おわりに

本論文では，動画像処理ライブラリ RaVioli を CUDA に対応させることにより高速化する手法を提案した．RaVioli はプログラマから解像度を隠蔽する抽象度の高いライブラリであるが，その抽象度の高さゆえに動作速度の遅さが問題となっていた．そこで本研究では，RaVioli から CUDA を使用可能となるように既存クラスの拡張や新しいクラスおよび高階メソッドの追加実装を行った．

拡張後の RaVioli を使用する場合，まずプログラマは画像の構成要素（画素やウィンドウ）に対する処理を記述した関数を定義する．この関数は Kernel 関数と呼ばれ，1Thread の処理を記述した関数である．その後，その Kernel 関数のハンドルを拡張後の RaVioli の持つ高階メソッドに渡すだけで，全 thread に Kernel 関数の実行を命令し，画像全体に処理を施すことができた．高階メソッド内では，自動的に Device 側のメモリ確保や，Host-Device 間の転送，thread の実行構成の設定が行われる．また動画像処理では，連続する 2 枚のフレームの Host-Device 間の転送と Kernel 関数の実行を自動的にオーバーラップさせることが可能になった．

拡張後の RaVioli を使用した場合には，Host-Device 間の転送や実行構成の設定，CUDA プログラムの一部の最適化は意識せずに動画像処理プログラムを記述可能になった．しかし，プログラマは Device 側で実行される関数を記述する必要があり，この関数を記述するには，プログラマは 1Thread あたりに使用する Register 数や Shared Memory の量等を意識する必要がある．そこで本研究では，従来の RaVioli プログラムを自動的に CUDA 対応の RaVioli プログラムに変換するトランスレータを提供することで，プログラミングレベルにおける抽象度を維持し，さらには従来プログラムを

自動的に高速化できることを示した．

RaVioli 不使用の場合，従来の RaVioli 使用の場合，CUDA 対応の RaVioli 使用の場合のサンプルプログラムを用いて処理速度の比較を行うことで，画像処理および動画像処理における本研究の有効性を示した．処理時間が小さな一部の画像処理プログラムでは，Device メモリの確保や Host-Device 間の転送時間を無視することができず，処理速度の向上は確認することができなかったものの，本来高速化が必要となるような処理時間が大きな画像処理プログラムにおいて，大幅な処理時間の削減が確認できた．また動画像処理プログラムにおいては，オーバーラップ等による処理時間の削減が確認できた．

今後の課題として，高階メソッドやトランスレータの改良による更なる最適化が挙げられる．thread の実行構成の設定において，現在は Block のサイズを 16×16 としている．CUDA における制約として，x 軸方向のサイズは 16 の倍数でなければならないというものがあるが，y 軸方向のサイズは特に決まりはない．また x 軸方向のサイズも 16 が最適な値とはいえない．そこで RaVioli 内で実行構成を動的に変動させて，最も効率的に Kernel 関数の実行を行える実行構成を決定させることができるのではないかと考えている．また，今回の研究では十分に活用することができていない Shared Memory を有効活用できるように，トランスレータを改良する必要がある．これについては，まずエッジ抽出やエンボスフィルタ等の近傍処理で有効活用できると考えている．さらに今回は使用していない Texture Memory や Constant Memory についても，有効に使用できる方法を考える必要がある．また記述可能なプログラムの範囲を広げるために，RaVioli およびトランスレータを拡張する必要がある．

また RaVioli 全体の課題としては,「メモ化手法」[18] を追加実装することが挙げられる．メモ化手法とは，関数の入力値に対応した出力値を表に記憶しておき，再度同じ関数が同じ入力で呼び出された場合に，出力値を表から呼び出すことにより計算を省略し，処理の高速化および省電力化を行う手法である．画像処理関数の入力値は RGB 色情報という限られた範囲の値であるため，この手法は有効であると考える．

謝辞

本研究のために多大な御尽力を頂き, 日頃から熱心な御指導を賜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 · 津邑研究室ならびに齋藤研究室の皆様に深く感謝致します．

参考文献

[1] 有田大作, 濱田義雄, 米元聡, 谷口倫一郎: PC クラスタを利用した実時間並列画像処理環境 RPV, 電子情報通信学会論文誌 D-Ⅱ, Vol. J84-D-Ⅱ, No. 6, pp. 965–975 (2001).

[2] 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).

[3] 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).

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

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

[6] Köthe, U.: *VIGRA - Vision with Generic Algorithms*, 1.6.0 edition (2008).

[7] 飯尾淳, 谷田部智之, 比屋根一雄, 米元聡, 谷口倫一郎: 動画像処理ライブラリ MAlib を利用した 3 次元ユーザインタフェースの実装, オブジェクト指向 2002 シンポジウム (OO2002), 情報処理学会, pp. 117–120 (2002).

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

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

[10] 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).

[11] 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).

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

[13] 吉本廣雅, 有田大作, 谷口倫一郎: 実時間分散画像処理システムのための信頼度駆動アーキテクチャ, 情処研報 2004-ARC-149 (HOKKE 2004), pp. 85–90 (2004).

[14] Yoshimoto, H., Date, N., Arita, D. and Taniguchi, R.: Confidence-Driven Architecture for Real-time Vision Processing and Its Application to Efficient Vision-

based Human Motion Sensing, *Proc. of the 17th Int'l. Conf. on Pattern Recognition (ICPR'04)*, Vol. 1, pp. 736–740 (2004).

[15] Baskaran, M. M., Bondhugula, U., Krishnamoorthy, S., Rountev, A. and P.Sadayappan: A Compiler Framework for Optimization of Affine Loop Nests for GPGPUs, *ACM International Conference on Supercomputing(ICS)* (2008).

[16] Ueng, S.-Z., Lathara, M., S.Baghsorkhi, S. and mei W.Hwu, W.: CUDA-lite: Reducing GPU Programming Complexity, *International Workshop on Languages and Compilers for Parallel Computing(LCPC)* (2008).

[17] Lee, S., Min, S.-J. and Eigenmann, R.: OpenMP to GPGPU: A Compiler Framework for Automatic Translation and Optimization, *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming(PPoPP)*, pp. 101–110 (2009).

[18] Norvig, P.: *Paradigms of Artificial Intelligence Programming*, Morgan Kaufmann (1992).