一般社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
IEICE Technical Report
CPSY2015-4,DC2015-4(2016-05)

動画像処理ライブラリ RaVioli におけるデータ転送と関数実行の効率化

古橋　一輝†　　小野　和馬†　　津邑　公暁†

† 名古屋工業大学

あらまし　静止画や動画像の処理を高速に実行可能な GPU の性能を十分に引き出すためには，その複雑なプロセッサ構成を理解した上でプログラムを記述する必要がある．動画像処理ライブラリ RaVioli の GPU 向け拡張である RaVioli/CUDA は，GPU 向け開発環境である CUDA 特有の記述をライブラリ内に隠蔽することで，このプログラマの負担を軽減している．しかし RaVioli/CUDA には，まだ GPU の性能を十分に引き出しきれていない部分がある．そこで本稿では，RaVioli/CUDA を改良し，GPU の利用効率を向上させる手法を提案する．提案手法では，GPU 使用時のデータ転送の効率化と関数同士の並行実行を可能にすることで，GPU の利用効率の向上を図る．提案手法を実装した RaVioli/CUDA を用いたプログラムの実行時間を，既存の RaVioli/CUDA と比較した結果，最大で約 3 倍の高速化を実現できることが確認できた．

キーワード　動画像処理ライブラリ，CUDA

Optimization of Data Transfer and Kernel Execution for Video Processing Library RaVioli

Kazuki FURUHASHI†, Kazuma ONO†, and Tomoaki TSUMURA†

† Nagoya Institute of Technology

Abstract　Programmers need to understand the complicated GPU structure deeply to utilize GPUs that can efficiently execute image and video processing. We have proposed a video processing library RaVioli/CUDA that is an improvement of RaVioli for supporting a GPU platform called CUDA. RaVioli/CUDA frees programmers from special notations for CUDA programming. However, it still leaves place for utilizing GPUs. For solving this problem, we propose improvements of RaVioli/CUDA for utilizing GPUs by optimizing data transfer and kernel function execution. By the result of experiments with a video processing program, we show that the proposed method improves performance 3-fold against the traditional RaVioli/CUDA.

Key words　image and video processing, CUDA

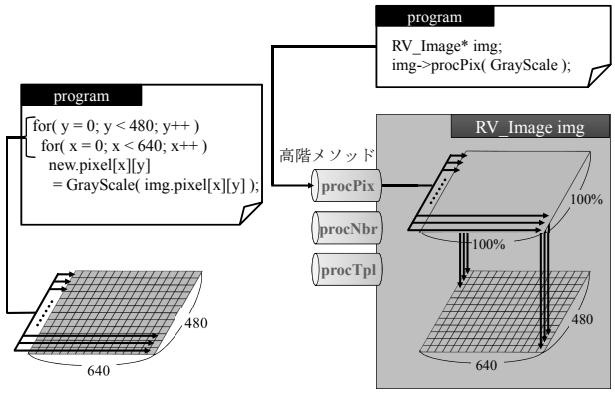
1. はじめに

携帯電話，デジタルカメラ，監視装置など静止画や動画像を扱える様々な情報機器が普及してきており，これに伴い，マルチメディアアプリケーションを開発する機会も増加している．このような背景を受けて，これまでに多くの動画像処理ライブラリが提案されている．その中でも **RaVioli**（**R**esolution-**A**daptable **V**ideo and **I**mage **O**perating **Li**brary）[1,2] はプログラマから解像度を隠蔽するプログラミングパラダイムを提供している．

一方で，GPU などに代表される，低消費電力で高い性能を実現可能なアクセラレータコアの普及が進んでいる．これに伴い，汎用プロセッサコアとアクセラレータコアとを搭載した，ヘテロジニアスな計算機環境も一般的となりつつあり [3]，その流れはモバイルプロセッサにまで波及してきている [4]．このような背景から，動画像処理にアクセラレータコアを利用する機会が増加している．しかしながら，アクセラレータコアを利用した高性能な動画像処理プログラムを開発するためには，その複雑なプロセッサ構造を理解した上でプログラムを記述し，さらにこれをチューニングする必要がある．そのため，プログラム開発コストの大きさが問題となっている．

この問題を解決するために，RaVioli の GPU 向け拡張である **RaVioli/CUDA** [5] が提案されている．RaVioli/CUDA では GPU 向け開発環境である CUDA 特有の記述をライブラリ内に隠蔽することでプログラマの負担を軽減している．しかし，RaVioli/CUDA には，まだ GPU の性能を十分に引き出しきれていない部分がある．

そこで本稿では，RaVioli/CUDA を改良し，GPU の利用効



(a) 一般的なプログラム　(b) RaVioli を用いたプログラム

図 1　画像処理プログラム

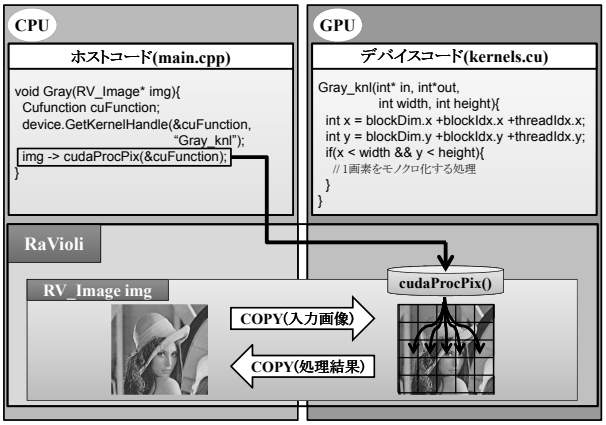


図 2　RaVioli/CUDA を用いたプログラム

率を向上させる手法を提案する．提案手法では，ストリーム数を自動調整することで GPU 使用時に必要となるデータ転送を効率化するのに加え，GPU 上で実行される関数であるカーネル関数の並行実行を可能にすることで，GPU の利用効率の向上を図る．

2. 動画像処理ライブラリ RaVioli

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

2.1 RaVioli 概要

一般に画像処理では，画像の構成要素に対する処理を，画像全体または任意の範囲に繰り返し適用するものが多い．例えば，カラー画像からモノクロ画像へ変換する処理や色を反転する処理では，処理単位は画素である．そして，**図 1(a)** の一般的な画像処理プログラムに示すようにループイテレーションを用いることで全ての画素に対して処理が適用されるが，その際プログラマは画像の幅と高さを意識してプログラムを記述しなければならない．

一方，RaVioli では画像の構成要素に対する処理のみを関数として定義し，その関数を RaVioli が提供しているメソッドに渡すことで，画像内の各構成要素に対して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．この高階メソッドは様々な処理単位ごとに用意されている．ここで，RaVioli を用いてカラー画像をグレースケールに変換する処理の様子を**図 1(b)** に示す．RaVioli は画像情報を RV_Image クラスにカプセル化している．カプセル化された画像に対して処理を適用するためには，RV_Image インスタンスの高階メソッドに構成要素関数を渡すのみでよい．図 1(b) の例では，RV_Image インスタンス img が持つ高階メソッド procPix() に，1 画素をグレースケール化する処理が記述された構成要素関数 GrayScale() を渡している．そして，高階メソッド procPix() は img が持つ画像の全ての画素に，GrayScale() を繰り返し適用する．このように RaVioli を用いる場合，プログラマは解像度や繰り返し処理を意識することなく画像処理プログラムを記述できる．

この例からも分かるように，RaVioli を用いて記述されたプログラムでは，画像の構成要素に対する処理が関数として切り出されるため，並列処理の単位が明確になり，自動並列化の実現が比較的容易である．そこで RaVioli では汎用マルチコアプロセッサや GPU，Cell/B.E. [6] など様々な環境に対応した自動並列化が実現されている [2,5]．これらのうち，GPU に対応させた RaVioli の拡張が RaVioli/CUDA である．RaVioli/CUDA は，NVIDIA 社が提供している GPU コンピューティング向け統合開発環境である CUDA [7] をバックエンドとして実装されている．本稿では，この RaVioli/CUDA のさらなる改良について述べる．

2.2 RaVioli/CUDA 概要

RaVioli/CUDA は，画像情報やフレームレートに加え，CUDA を使用して画像処理プログラムを記述する場合に必要となるメモリ確保や実行構成の設定などもプログラマから隠蔽している．

RaVioli/CUDA を用いたプログラムの実行モデルを**図 2** に示す．プログラマはまず，GPU 上で実行される**デバイス・コード**内で，GPU が処理する関数である**カーネル関数**として構成要素関数を定義する．一方 CPU 上で実行される**ホスト・コード**では，モジュールをロードするための処理やカーネル関数のハンドルを取得するための処理といったカーネル関数を実行するための処理を記述する．そして，取得したハンドルを RV_Image インスタンスの高階メソッドの引数に渡す．なお，この高階メソッドは RaVioli と同様に，様々な処理単位ごとに用意されている．これにより，自動的にデバイス・メモリに領域が確保され，CPU から GPU へデータが転送される．さらに実行構成も自動的に設定され，カーネル関数が高階メソッドに応じて適切な処理対象に適用される．その後，画像に対する全ての処理が終わると処理結果が CPU に返される．このようにして RaVioli/CUDA は，メモリ確保や実行構成の設定などをライブラリ内部で自動的に行う．

ここで，RaVioli/CUDA では，動画像処理プログラムの性能を向上させるため，CUDA が提供しているストリームという機能を利用している．ストリームとは，GPU での処理の実

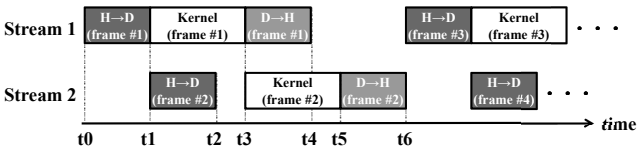


図 3　RaVioli/CUDA を用いた動画像処理プログラムの動作

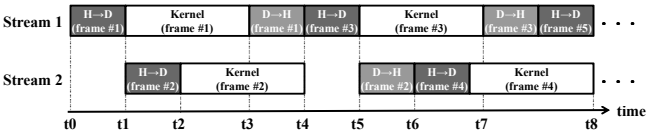


図 4　依存関係が存在しない場合の動作

行順序を定義できる枠組みのことである．RaVioli/CUDA ではデータ転送やカーネル関数の呼び出し時に，異なるフレームに対する処理を異なるストリームへ発行することで，データ転送とカーネル関数とをオーバラップ実行させている．**図 3** に，RaVioli/CUDA を用いて記述した動画像処理プログラムの動作を示す．ここで，図 3 中の “H→D”，“D→H” はそれぞれ CPU（Host）から GPU（Device）へのデータ転送と GPU から CPU へのデータ転送を表しており，“Kernel” はカーネル関数の実行を表している．また，“flame #N” は N 枚目のフレームに対する処理を表しており，最初のフレーム（frame #1）に対するデータ転送とカーネル関数の実行は Stream1 に，2 枚目のフレーム（frame #2）に対する処理は Stream2 に発行され，実行される様子を表している．このように別のフレームに対するカーネル関数とデータ転送とを並行実行させるためには，それらの並行実行させたい処理を別のストリームへ発行すればよい．以上のように RaVioli/CUDA ではストリームを用いることで，データ転送とカーネル関数の実行とを並行実行させ，処理を効率化している．ただしホスト-デバイス間のデータバスは 1 本しか存在しないため，データ転送同士は並行実行できない．また，既存の RaVioli/CUDA ではカーネル関数同士についても並行実行できない．これは RaVioli/CUDA では，あるフレームに対するカーネル関数を呼び出す処理はそれより前のフレームに対するカーネル関数の実行が終了するまでストリームに発行できないためである．

2.3　RaVioli/CUDA の問題点

本来，発行先ストリームの異なる関数同士は，お互いの処理順に依存がない場合，並行実行可能である．しかし既存の RaVioli/CUDA では，処理順依存の有無にかかわらず関数同士を並行実行できていない．よって図 3 の t2〜t3 のように，無駄な待機時間が発生してしまう．

また，ホスト-デバイス間のデータ転送に用いられるデータバスは 1 本しかなく，効率良くデータを転送するには，この 1 本のデータバスをできるだけ空き時間が生じないように使用する必要がある．しかし，既存の RaVioli/CUDA ではデータバスが使用されない時間が存在する（t2〜t3，t4〜t5）．以上の 2 点において，既存の RaVioli/CUDA では GPU の性能を十分に引き出しきれていない．

3.　GPU 利用効率の向上手法

本章では，2.3 節で述べた問題点を解決するために，RaVioli/CUDA を改良し，GPU の利用効率を向上させる手法を提案する．まず 3.1 節でカーネル関数の並行実行について述べ，その後 3.2 節でストリーム数の調整について述べる．

3.1　カーネル関数同士の並行実行

2.3 節で述べたように，異なるフレームに対するカーネル関数実行同士に処理順依存が存在しない場合，それらは本来並行実行可能である．しかし，既存の RaVioli/CUDA ではフレーム間の依存関係の有無にかかわらず，それらのフレームに対するカーネル関数同士は並行実行されない．そこで，提案手法ではフレーム間の依存関係の有無に応じて，ストリームへの処理の発行の仕方を変えることで，依存関係が無い場合にカーネル関数を並行実行させる．

まず，フレーム間に依存関係が存在する場合，既存の RaVioli/CUDA と同様に，データ転送とカーネル関数は並行実行させるが，異なるフレームに対するカーネル関数は逐次実行させる．

一方で，フレーム間に依存関係が存在しない場合，データ転送とカーネル関数だけでなく，カーネル関数同士も並行実行させる．依存関係が存在しない場合の動作を**図 4** に示す．図 3 と対比すると，Stream2 による frame #2 の GPU へのデータ転送が終わり次第（t2），Stream2 によるカーネル関数の実行がすぐに始まっており，t2〜t3 において複数のカーネル関数を並行実行することで，図 3 において発生している t2〜t3 の待機時間を解消することができている．

3.2　ストリーム数の調整

2.3 節で述べたように，効率良くデータを転送するためには，一本しかないデータバスを絶え間なく使用し続けることが望まれる．しかし従来の RaVioli/CUDA のように，ストリームを 2 本だけ使用した場合，カーネル関数実行に要する時間が長いプログラムでは，一方のストリームでカーネル関数を実行している間のデータバスの空き時間を，他方のストリームによるデータ転送だけでは埋めきれない．そこで，データバスに空きが発生しない必要十分な数のストリームを生成・使用することで，データバス上で絶え間なくデータ転送が行われるように RaVioli/CUDA を改良する．なお，適切なストリーム数はデータ転送とカーネル関数実行に要する時間比に依存して決まるため，これを考慮してストリーム数を自動的に調整する機構も提案する．

それでは，ストリーム数を調整しない場合とする場合の動作をそれぞれ**図 5**，**図 6** に示し，ストリーム数を調整することでデータ転送が効率化される様子を説明する．図 5 は既存の RaVioli/CUDA を用いた場合の動作をタイミングチャートで表したものであり，ストリーム数は 2 である．一方で，図 6 ではストリーム数を調整した結果，その数が 5 つになった場合を仮定している．図 5 の場合ではまず，Stream1 において GPU へのデータ転送を実行する． その後，Stream1 に発行された

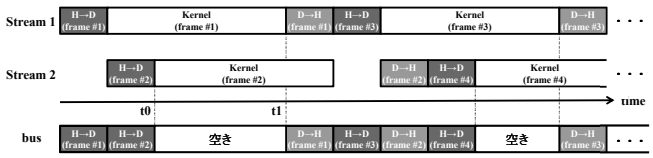


図 5　ストリーム数を調整していない場合の動作

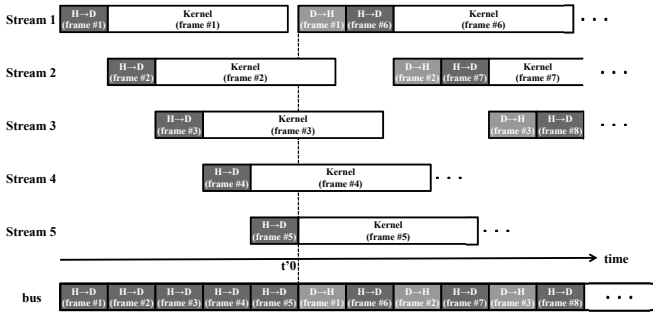


図 6　ストリーム数を調整した場合の動作

カーネル関数と Stream2 に発行された GPU へのデータ転送とを並行実行させる．次に，Stream2 に発行された GPU へのデータ転送が終わり次第，Stream2 に発行されたカーネル関数の実行を開始する（t0）．このとき，データバスが未使用状態となるため，次のフレームを GPU へ転送可能となる．しかし，Stream1，Stream2 ともに GPU へのデータ転送の次にカーネル関数を実行中であるため，どちらのストリームでも新なデータ転送を行うことができず，データバスに空きが生じてしまう（t0〜t1）．これに対し，図 6 ではまず，ストリームを増やしたことで，図 5 の場合で生じていたデータバスの空き時間を解消できている．ここで，一般的にデータバスの空き時間を発生させないためには，あるストリームによるカーネル関数の実行に要する時間よりも他のすべてのストリームが一度ずつ GPU へデータ転送する際にかかる合計時間が長くなればよい．図 6 では，Stream5 によるデータ転送が行われている間に Stream1 によるカーネル関数の実行が終了し，Stream5 のデータ転送が終わり次第，Stream1 がデータ転送を開始できている（t'0）．

4. GPU 利用効率向上手法の実装

既存の RaVioli/CUDA を拡張し，3 章で述べた GPU 利用効率向上手法を実装した．まず，4.1 節で，RaVioli/CUDA に追加したメソッドとその用法について述べ，その後，4.2 節でメソッドの動作について述べる．

4.1 メソッドの概要と用法

提案手法を実現するために，RaVioli/CUDA に用意されている，動画像の情報をカプセル化する機能を持つ RV_Video クラスを拡張し，メソッド runOverlap() を追加した．プログラマは，フレーム間の依存関係の有無をこの runOverlap() に引数として与えることにより指定する．このメソッド runOverlap() の概要と用法について，**図 7** に示すプログラムを例に説明する．なお，このプログラムは 2 値化とエッジ抽出を動画像の各フレームへ順に適用するプログラムである．RaVioli/CUDA

```
1  void BinarizeStage(RV_Data* data){  /* 2 値化 */
2    CUfunction cuFunction;
3    device.GetKernelHundle(&cuFunction,"Binarize_knl");
4    data -> image -> cudaProc(cuFunction,data);
5  }
6  void EdgeStage(RV_Data* data){  /* エッジ抽出 */
7    CUfunction cuFunction;
8    device.GetKernelHundle(&cuFunction,"Edge_knl");
9    data -> image -> cudaProc(cuFunction,data);
10 }
11 void main(int argc, char argv[]){  /* main 関数 */
12   RV_Video video;
13   RV_StageVector DetectEdge;
14   //ステージを登録
15   DetectEdge.push(BinarizeStage);
16   DetectEdge.push(EdgeStage);
17   //動画像処理の実行
18   //  video.runOverlap(DetectEdge, 0);
19   video.runOverlap(DetectEdge, 1);
20 }
```

図 7　エッジ検出プログラム

を用いて記述された動画像処理プログラムは，動画像の単一フレームを処理するための関数（1 行目〜10 行目）と main 関数（11 行目〜20 行目）で構成される．main 関数ではまず，RV_Video インスタンスと RV_StageVector インスタンスを生成する．この例では，RV_Video インスタンスとして video を（12 行目），RV_StageVector インスタンスとして DetectEdge を生成している（13 行目）．この RV_StageVector インスタンスは，動画像の各フレームに対して段階的に段階的に適用したい処理を，一元的に管理するために用いられる．ユーザはこのインスタンスが持つメソッド push() に，これらの処理を定義した関数を順に渡すことで（15 行目〜16 行目），それらの処理が RV_StageVector インスタンスに登録され，この登録された処理が各フレームに順に適用される．ここで，これらの RV_StageVector インスタンスによって管理される，フレームを処理する関数内には，カーネル関数の呼び出し側の処理を記述する．なお，この関数内で使用されている RV_Data 型は，画像データとあわせてデバイスメモリへのポインタといった GPU 側の情報などをメンバとして保持している．これにより，動画像のフレームを処理するための各関数内では，画像データ用の配列や中間データ用の配列，スカラ変数など異なる入出力データを使用する場合であっても，RV_Data インスタンスのメンバを参照することでデバイス側の情報や必要となる入出力データにアクセスできる．この例では，フレームにエッジ抽出を適用する際に RV_Data インスタンスのメンバを参照することで 2 値化が適用された画像データにアクセスできる．RV_StageVector インスタンスに対し処理を全て登録した後，メソッド runOverlap() にこの RV_StageVector インスタンス

```
1  void RV_Video::runOverlap
2  (RV_StageVector* StageV, int dep){
3    int loop;
4    if(dep == 0){
5      /* 既存の RaVioli/CUDA と同じ動き */
6    }else{
7      /* ストリーム数の計算と更新 */
8      int stream_num = CalcStreamNum();
9      /* stream 生成とメモリ確保 */
10     CUstream st[stream_num];
11     CUdeviceptr device[stream_num];
12     for(int i=0; i < stream_num; i++){
13       cudaStreamCreate(&st[i]);
14       cuMemAlloc(&device[i], MEM_SIZE);
15     }
16     /* 実行開始 */
17     for(int i=0; i < stream_num; i++, loop++){
18       i %= stream_num;
19       cuMemcpyHtoDAsync(device[i], frame[loop],
20                                    MEM_SIZE, st[i]);
21       kernelCallAsync(StageV -> Kernel(frame[loop],st[i]));
22       cuMemcpyDtoHAsync(frame[loop], device[i],
23                                    MEM_SIZE, st[i]);
24     }
25     /* メモリ解放と stream 破棄 */
26     for(int i=0; i < stream_num; i++){
27       cuMemFree(device[i]);
28       cudaStreamDestroy(st[i]);
29     }
30   }
31 }
```

図 8　メソッド runOverlap() の定義を簡略化したコード

を渡すことで，それら登録した処理が複数ストリームによりオーバラップ実行される．この際プログラマは，runOverlap() の第 2 引数を用いて，フレーム間の処理依存の有無を明示することができる．異なるフレームに対するカーネル関数の実行同士に処理順依存が存在する場合，18 行目に示すようにプログラマは引数に 0 を指定する．このとき，runOverlap() は，既存の RaVioli/CUDA と同様に，異なるフレームに対するカーネル関数を逐次的に実行する．一方，依存関係が存在しない場合は，19 行目に示すようにプログラマは引数に 1 を指定する．この場合，runOverlap() 内で最適なストリーム数を算出後，その数のストリームが生成され，動画像処理が実行される．runOverlap() の詳しい動作については次節で述べる．

4.2　メソッドの実行時の動作

本節では，runOverlap() の動作を，**図 8** に示す定義に沿って説明する．runOverlap() は 4.1 節で述べたように，動画像の各フレームに対する処理を保持する RV_StageVector インスタンス StageV と，フレーム間の依存関係の有無を表す整数 dep を仮引数として呼び出される．runOverlap() 内では，dep の値に応じて，処理が切り替わる．引数として 0 が指定されたとき，フレーム間に依存関係が存在するとみなし，既存の RaVioli/CUDA と同じように動画像処理を実行していく（4〜6 行目）．一方で，1 が指定されたときは依存関係がないとみなし，ストリーム数を調整して動画像処理を実行していく．まず，最適なストリーム数を算出するために，RV_Video クラス内で定義されている関数 CalcStreamNum() を呼び出す（8 行目）．この関数 CalcStreamNum() 内では，まず 1 フレーム目に対する処理のみを，他のフレームの処理と並行実行させることなく実行し，GPU へのデータ転送に要した時間とカーネル関数の実行に要した時間を計測する．そして計測した時間に基づき，最適なストリーム数を算出する．計算に使用する式は以下のとおりである．

$$\left\lceil \frac{\text{カーネル関数の実行に要する時間}}{\text{GPU へのデータ転送に要する時間}} \right\rceil + 1$$

この計算式では，あるストリームによるカーネル関数の実行に要する時間よりも他のすべてのストリームが一度ずつ GPU へデータ転送する際にかかる合計時間が長くなるためにはいくつのストリームが必要になるのかを算出できる．この計算結果をもとに必要となる数のストリームを生成し，生成したストリーム数に応じて必要となる GPU 側のメモリを確保する（11〜15 行目）．その後，各フレームに対する GPU へのデータ転送，カーネル関数の呼び出し処理，CPU へのデータ転送をストリームへ発行し，すべてのフレームに処理を施す（17〜24 行目）．このとき，各フレームの発行先となるストリームはラウンドロビンに割り振る．最後に，メモリ解放，ストリームの破棄を順に行う（26〜29 行目）．このようにすることで，自動的に最適なストリーム数で動作させることが可能となる．

5.　評　　価

これまでに述べた提案手法を実装し，既存の RaVioli/CUDA と実行時間を比較した．

5.1　評 価 環 境

評価環境を**表 1** に示す．本評価には，Kepler2 アーキテクチャを採用している GTX TITAN を使用した．プログラムには，グレースケール化，エッジ抽出，直線検出の 3 つを使用し，実行時間を評価した．入力には，解像度が VGA（640×480）で，長さが 1 分間の動画像を使用した．なお，この動画像のフレームレートは 30fps であり，総フレーム数は 1,800 枚である．

5.2　評価結果と考察

評価結果を**図 9** に示す．各プログラムの実行結果を表すバーは，既存の RaVioli/CUDA で実行した場合を 1 として正規化したときの提案手法の実行時間をそれぞれ表している．

評価結果から，提案手法を用いた場合では既存の RaVioli/CUDA を用いた場合よりも実行時間を削減できており，カーネル関数同士の並行実行を可能にし，データバスの使用率を向

表 1　評 価 環 境

OS	CentOS 6.4
CPU	Intel Core i7-4770
Frequency	3.4 GHz
Memory	16 GB
GPU	GTX TITAN
Core Clocks	837 MHz
Memory Clocks	6,008 MHz
SMs	14
Compiler	gcc 4.4.7
Compile options	-O3

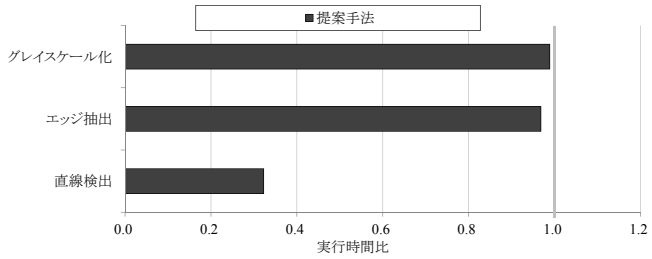


図 9　評 価 結 果

上させたことによって，GPU の利用効率が向上することを確認できた．ここで，グレースケール化プログラムでは約 1.01 倍，エッジ抽出プログラムでは約 1.03 倍速度が向上したのに比べ，直線検出プログラムでは約 3 倍と他の 2 つのプログラムより，大きく速度が向上している．ストリーム数の自動調整結果を確認すると，グレースケール化プログラムでは 2〜3，エッジ抽出プログラムでは 3〜4，直線検出プログラムでは 8〜10 のストリームが生成・使用されていた．これは直線検出プログラムでは，他の 2 つのプログラムと比べ，データ転送に要する時間よりも，カーネル関数の実行に多く時間を要するため，既存の RaVioli/CUDA ではデータバスが空きとなる時間が多いからである．提案手法では，この空き時間を削減することで，実行時間を削減できた．

以上の結果から，提案手法では GPU の利用効率を向上させることで既存の RaVioli/CUDA よりも処理の高速化を実現できることが確認できた．特に，データ転送に要する時間よりも，カーネル関数の実行に多く時間を要するようなプログラムで大きく効果を得ることができることも確認できた．提案手法を用いることでプログラマは，GPU を使用した動画像処理においてプログラムの処理内容や処理を行う環境を意識することなく，効率の良い動画像処理を実現可能である．

6. お わ り に

本稿ではまず，動画像処理ライブラリ RaVioli の GPU 向け拡張である RaVioli/CUDA の問題点として関数の並行実行ができず，またデータバスに空きが生じてしまうため，GPU を効率よく使用できていないことを示した．この問題を解決するために，関数の並行実行とストリーム数の調整による GPU の利用効率向上手法を提案した．既存の RaVioli/CUDA と提案手法を実装した RaVioli/CUDA のそれぞれを用いて記述した動画像処理プログラムの実行時間を比較した結果，提案手法による性能向上を確認することができた．特に，直線検出を行うプログラムで比較すると，提案手法を用いたプログラムでは，既存の RaVioli/CUDA を用いたプログラムに対して約 3 倍の速度向上を達成することができた．今後の課題として，フレーム間依存関係の自動検出機構の提案が挙げられる．

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，Mar. 2009.

[2] H. Sakurai, M. Ohno, T. Tsumura, and H. Matsuo, “RaVioli: a Parallel Video Processing Library with Auto Resolution Adjustability,” Proc. IADIS Int'l. Conf. Applied Computing 2009, vol.1, pp.321–329, Nov. 2009.

[3] Advanced Micro Devices, “What is Heterogeneous System Architecture (HSA)?,” , http://developer.amd.com/resources/heterogeneous-computing*ash*what-is-heterogeneous-system-architecture-hsa/.

[4] Whitepaper: NVIDIA Tegra X1 : NVIDIA's New Mobile Superchip, NVIDIA Corp. 1.0 edition, 2015.

[5] 稲葉崇文，大野将臣，桜井寛子，津邑公暁，松尾啓志，“GPU 及び Cell/B.E. に対応した解像度非依存型動画像処理ライブラリ RaVioli の提案と実装,” 信学技報，vol.IEICE-110 (SWoPP2010)，no.IEICE-CPSY-167，pp.7–12 電子情報通信学会，Aug. 2010.

[6] Sony Computer Entertainment, Cell Broadband Engine Architecture, , 1.01 edition, Oct. 2006.

[7] NVIDIA Corp. , CUDA C Programming Guide, , 7.0 edition, Mar. 2015.