I-062

マクロブロック間依存制約緩和による GPU H.264 動き推定の高速化
Relaxation of Inter-block Dependencies to Accelerate GPU H.264 Motion Estimation

鷹野 芙美代† 森吉 達治†
Fuimiyo Takano Tatsuji Moriyoshi

1. はじめに

H.264/MPEG-4 AVC[1]は、高い符号化効率を実現する動画像圧縮符号化の国際標準である。H.264 エンコーディングは符号化効率を高めるために非常に複雑であり、さらに年々画像サイズ拡大要求も高まっていることから、高速化が望まれている。

高速化手段の一つに GPGPU（General Purpose computing on Graphics Processing Unit）の利用がある。GPU は多数のコアを集積し、並列処理、特にベクトル処理に適した 3D グラフィック処理用ハードウェアであり、近年その性能向上は著しい。GPU を 3D グラフィック処理だけでなく物理シミュレーションや複雑な画像処理などの汎用処理にも用いる試みが GPGPU である。

そこで本稿では、H.264 エンコーディングで負荷が最も高い処理である動き推定を GPU 並列化する。H.264 の高い符号化効率を達成するには、動きベクトル予測を用いた動き推定が有効である。しかし、動きベクトル予測にはブロック間の依存関係があるため、並列に処理できるブロック数が少なくなるという課題がある。そこで、ブロック間の依存制約を緩和することで高い符号化効率を維持しつつ並列性を向上する擬似動きベクトル予測手法を提案する。さらに GPU では、ハードウェアリソース制約によりプロセッサ稼働率が高くしにくいという課題もある。そのため、本稿では GPU のプロセッサ稼働率を向上させる仮想スレッドブロック処理手法を提案する。

2. H.264 エンコーダの動き推定

H.264 は、MPEG-2 などと比較して高い圧縮効率を実現する国際符号化標準である。本稿では、H.264 エンコーダ内で特に演算量の多い動き推定を GPU 並列化する。

2.1 動き推定処理概要

H.264 は、動き補償を用いた画面間予測符号化を行うことで高圧縮率を実現している。一般的に動画像では時間的に連続なフレームの類似性が高いため、過去のフレーム画像から動きを補正（動き補償）して予測画像を生成し、予測画像と符号化対象フレームの差分のみを符号化することで大幅な圧縮を実現することができる。

動き推定は、フレーム間の動きを推定し、16x16 画素のマクロブロック毎に動きベクトルを算出する処理である。動き推定では、推定符号量をコストとしてコストが最小になるブロックを参照フレームから探索する。探索結果のブロック位置と処理ブロック位置の差がフレーム間の動きを表す動きベクトルである。コストは、参照フレームのブロックと処理ブロックの類似度コスト（例えばブロック内の画素同士の差分絶対値和 SAD: Sum of Absolute Difference）と、動きベクトル情報の符号化に必要な符号量であるベクトルコストの重み付け和によって算出する[1][2]。動き推定は候補となる多数の動きベクトルそれぞれに対してブロック類似度を計算する必要があるため、演算量が非常に多く高速化が望まれている。

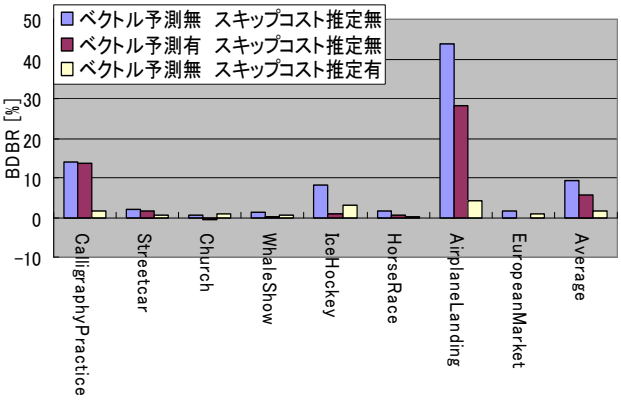


図 1 ベクトル予測とスキップ推定の効果

2.2 動きベクトル予測とスキップ推定

動き推定において符号化効率を向上させるための手法に動きベクトル予測とスキップ推定がある。

動画像は一般的に近隣のブロック同士の動きは似ていることから、画面間予測では、既に処理が終了している左と上と右上ブロックの動きベクトルのメディアンを予測ベクトルとし、処理ブロックの動きベクトルと予測ベクトルとの差分のみを符号化する。これにより、ベクトル符号量が削減され、符号化効率を向上できる。そのため、動き推定でも予測ベクトルを用いてベクトルコストを算出することで、精度の高い推定ができる。

また、H.264 では処理マクロブロックの情報を送らないスキップモードを用いて符号化効率を高めている。スキップモードはスキップベクトルの指すブロックの画像を符号化画像として用いる。通常スキップベクトルには予測ベクトルが用いられるが、左か上のブロックの動きベクトルがゼロベクトルだった場合にはゼロベクトルが用いられる[1]。スキップモードの符号量は非常に少ないため、動き推定では、スキップベクトルの類似度コストを通常の動きベクトルよりも小さく見積もることでスキップモードが選択されやすくし、符号化効率を向上させるスキップ推定が行われる[4]。

図 1に、フル HD サイズの ITE 標準動画像[3]8 種における、動きベクトル予測とスキップ推定の有無による符号化効率比較を示す。本稿では符号化効率は、BDBR (Bjontegaard Delta BitRate)[5]を用いて評価する。BDBR とは 2 つのエンコーダの同等画質における平均符号量差を表す指標で、この値が大きいほど基準エンコーダに比べて符号化効率が悪くなっていることを表す。基準エンコ

† 日本電気株式会社 システム IP コア研究所
System IP Core Research Laboratories, NEC Corporation

Copyright © 2011 by Information Processing Society of Japan and
The Instiute of Electronics, Information and Communication Engineers
All rights reserved.

ーダには参照実装 JM16.0[4]を用いた。図 1より、ベクトル予測やスキップ推定を行わないと最大 44%、平均でも 9%と大きく符号化効率が低下することがわかる。特に、スキップ推定の効果は大きく、これを行わない場合は符号化効率は大きく低下する。

3. GPGPU 概要と動き推定実装の課題

本章では GPGPU とその開発環境である CUDA について簡単に説明し、従来の GPU における動き推定実装の課題について述べる。

3.1 GPGPU と CUDA

CUDA (Compute Unified Device Architecture)[6]は NVIDIA 社が提供する GPU で汎用処理を行うためのアーキテクチャ・ソフトウェア環境である。CUDA における並列処理は全てのスレッドが同じプログラムを実行する SPMD (Single Program Multiple Data)モデルである。

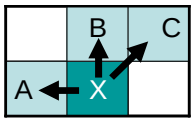
CUDA GPU のアーキテクチャは、1 スレッドを実行する CUDA コア が複数集まってストリーミングマルチプロセッサ(SM) を、SM が複数集まって全体の GPU を構成する階層構造となっている。これにあわせてプログラミングモデルも階層的であり、最小並列処理単位をスレッド、複数のスレッドの集まりをスレッドブロック、複数のスレッドブロックが集まった全体をグリッドと呼ぶ。同一スレッドブロックに属するスレッドは一つの SM 上で実行され、後述する共有メモリ空間を共有し、同期しながら処理を行うことができる。スレッドブロック、グリッドは、それぞれスレッド、スレッドブロックの多次元配列として定義される。メモリアーキテクチャは、全スレッドで共有する低速・大容量のグローバルメモリ、スレッドブロック内のみで共有するオンチップで高速・小容量の共有メモリ、といった複数種のメモリが存在している。特にグローバルメモリは共有メモリに比べてアクセスレイテンシが非常に長い。そのため、これらのメモリを適切に使い分けることが必要となる。

GPU は非常に多くのコアを持つメニコアプロセッサであり、今後もコア数は増加すると予想される。また、多くのスレッドが GPU 内のリソースを共有するため、ハードウェア的な制約を受けて各コアの稼働率が低くなることがある。そのため、コア数に見合った性能を得るためには、並列処理アルゴリズムと実行ハードウェア制御の両面から、1)処理の持つ並列性の向上、2)稼働率の向上、という 2 つを共に実現することが重要である。処理の並列性だけが向上してもそれを実際に高い稼働率で並列に処理できなければ効果はなく、稼働率を高めても処理自体の並列性が低ければ効率的な並列処理はできない。

3.2節では動き推定における処理の並列性について、3.3節では CUDA GPU の稼働率について述べる。

3.2 動き推定のブロック間依存による並列性低下

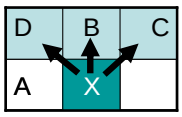
近年、GPU を用いて動き推定の速度向上を図る研究が多数行われている[7][8][9][10]。動き推定はマクロブロック単位で処理が行われることから、マクロブロックレベルで並列化されることが多い。文献[7][8]では、処理の並列性を重視し、全てのマクロブロックを並列に処理する。これらの手法は、周囲のマクロブロックの処理結果が使



1	2	3	4	5	6
3	4	5	6	7	8
5	6	7	8	9	10
7	8	9	10	11	12
9	10	11	12	13	14
11	12	13	14	15	16

(a) 参照ブロック位置　　(b) 並列処理可能ブロック

図 2　動きベクトル予測に用いるブロック位置



1	1	1	1	1	1
2	2	2	2	2	2
3	3	3	3	3	3
4	4	4	4	4	4
5	5	5	5	5	5
6	6	6	6	6	6

(a) 参照ブロック位置　　(b) 並列処理可能ブロック

図 3　横ブロック並列処理

用できないため、動きベクトル予測・スキップ推定を行うことができなかった。そのため、図 1に示すように符号化効率が低いという課題があった。

一方、文献[9][10]では、符号化効率を高めるため、動きベクトル予測が可能なようにマクロブロック間の依存制約を守って並列処理を行う。動きベクトル予測では図 2 (a)のブロック X に対して A, B, C 位置のブロックの動きベクトルを参照する[1]。文献[9]では、これらのマクロブロックの処理が終了してからマクロブロック X の処理を開始する。そのため、図 2(b)のように、1 列おき斜めのマクロブロックを並列に処理することになる。図 2(b)の番号は各マクロブロックの処理順を表し、同一番号のブロックは並列に処理可能であることを示す。文献[9]の手法は、正確な動きベクトル予測・スキップ推定を行えるため、符号化効率が低下することはない。しかし、並列に処理できるマクロブロック数はフレームあたり最大でもマクロブロック列数の半分と少ない。これは、多くても十コア程度からなるマルチコア CPU を用いた並列処理では充分であるが、数十から数百コアを持つメニコア GPU による並列処理ではプロセッサの持つ並列処理能力を使い切れず、十分な性能を発揮できない。

これに対し文献[10]では、図 2 のマクロブロック間の依存制約を緩和することで並列に処理できるマクロブロック数を増加させる。文献[10]の手法では、図 3(a)のブロック A の代わりにブロック D の動きベクトルを用いる。これにより、図 3(b)のように横に並んだブロック同士を並列に処理することができ、図 2 のように並列処理する場合に対して、並列に処理できるブロック数は 2 倍となる。この手法は、動きベクトル予測を全く行わない場合に比べると動き推定精度が高いが、正確ではない擬似的なベクトル予測を用いるため動き推定精度は低下する。さらに、スキップ推定において重要な左のマクロブロックを参照できない。スキップモードでは、左か上のブロックの動きベクトルがゼロだった場合にはゼロベクトルをスキッ

Copyright © 2011 by Information Processing Society of Japan and
The Instiute of Electronics, Information and Communication Engineers
All rights reserved.

プベクトルとする。そのため、左ブロックの動きベクトルが参照できないとスキップ推定の精度が大きく低下し、高い符号化効率を得にくい。

また、文献[10]では、複数のフレームにまたがるブロック同士の依存関係を守り、並列に処理する手法が述べられている。これにより、フレーム内のブロックのみを並列処理する場合に対して、並列に処理できるブロック数が多くなり、メニコア GPU による高速化が期待できる。

3.3 GPU コア稼働率の低下

GPGPU を効果的に使用するには、コアの稼働率が高くなるように並列処理することが重要である。

CUDA GPU では、コア数以上の多数のスレッドの命令をスケジューリングして実行することで、処理効率を上げる[6]。例えば、あるスレッドがメモリアクセスを待つ間に他のスレッドの演算を行うようにスケジューリングすることで、メモリアクセスレイテンシを隠蔽する。よって、同時実行可能なスレッド数が少ないと、レイテンシが隠蔽できず稼働率は低下する。一方で CUDA GPU では多くのスレッドが GPU 内のハードウェアリソースを共有するため、ハードウェア制約により同時実行可能なスレッド数が多くできない場合がある。制約の一つは、最大同時実行スレッドブロック数の制限である。スレッドはスレッドブロック単位で SM に割り当てられるため、SM あたりの最大同時実行スレッドブロック数に制限があると、充分な数のスレッドを同時実行できない。もう一つの制約は、共有メモリ量やレジスタ数といった SM 内のリソースによる制限である。各スレッドブロックの処理に必要なリソースが多ければ、多数のスレッドブロックを同時に実行できない。これらの制約を考慮してスレッドブロックあたりのスレッド数を調整することで、同時に実行出来るスレッド数を多くしコアの稼働率を向上させることができる。しかし、スレッドブロックあたりのスレッド数の最適値は GPU の機種・世代により一定ではない。そのため、環境に合わせたチューニングが必要になるが、通常の GPU 実装では、これらのパラメータは固定であり変更は容易ではない。例えば、CUDA による動き推定実装である文献[7]では、4x4 画素ブロックの動き推定処理を 1 スレッドブロック 256 スレッドで並列処理する。この場合、スレッドブロック数・スレッド数は画像サイズよって決まる。この対応を実行環境に応じて変更するのは容易ではない。

4. 提案する GPU 動き推定実装

本章では、符号化効率を保ちながら高速に GPU 処理できる動き推定手法を提案する。スキップ推定を考慮に入れてブロック間依存制約を緩和することで、符号化効率を低下させずに処理の並列性を向上する。さらに、同時に実行出来るスレッド数の最適化を容易にし、GPU コアの稼働率を向上させる。

以降の節では、まず基本的な GPU 並列化方針について述べ、ブロック間の依存制約を緩和した GPU 向け並列性向上手法と GPU 稼働率向上手法を述べる。

4.1 動き推定の GPU 並列化基本方針

動き推定はマクロブロックごとに動きベクトルを探索し、さらに、マクロブロック内の画素毎に参照フレームと現フレームの画素値の差分を計算する。そこで、図 4 に示すように CUDA のグリッド・スレッドブロック・スレッドという階層的なアーキテクチャに合わせ、マクロブロック・画素の各レベルにおける階層的に並列化する[11]。参照実装 JM[4]でも採用されている EPZS[12]などの高速な適応的ベクトル探索手法ではマクロブロックごとに処理パスが異なるため、各マクロブロックを独立実行可能なスレッドブロックで並列処理するのが適している。スレッドブロック毎にマクロブロックの並列処理を行い、スレッドブロック内のスレッド毎にマクロブロック内の画素の並列処理を行う。画素類似度計算は画素あたりの演算量が小さく、処理位置のアドレス演算が相対的に重い。そのため、各スレッドが 8 画素を処理することでこれを削減する。よって 1 マクロブロックを処理するスレッドブロックは 32 スレッドで構成されることとなる。

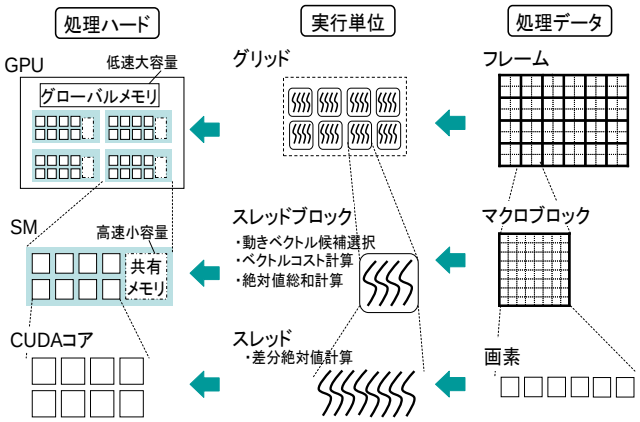


図 4　動き推定の階層並列処理

さらに CUDA のメモリアーキテクチャを効果的に使用するため、サイズの大きな現画像と参照画像は大容量のグローバルメモリに格納し、同一スレッドブロックで多数アクセスする原画像のうちの処理対象マクロブロックのデータは高速な共有メモリにコピーして使用する。

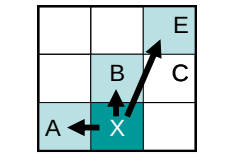
4.2 擬似ベクトル予測によるブロック間依存の緩和

本稿では符号化効率を高く保ちつつ並列性を向上させる擬似ベクトル予測手法を提案する。

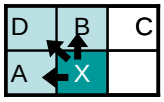
提案手法では、文献[10]のように左・上・右上ブロックのうち一部のみを参照することで、マクロブロック間依存を緩和する。文献[10]の手法では、図 3 の左ブロック A の動きベクトルが参照できないことから、符号化効率が低下するという課題があった。さらに、本稿では H.264 エンコーダのうち動き推定しか扱わないが、エンコーダ内の画面内予測やデブロックフィルタなどの処理でも左と上のマクロブロックの処理結果を用いる。そのため、これらの処理との親和性を高めるためには、動き推定においても左マクロブロックの処理と上マクロブロックの処理が終了してから当該マクロブロックの処理を行うことが重要である。

そこで本稿では、左と上のブロックの両方を参照し、かつ並列処理可能なマクロブロック数の多い手法を提案する。左と上のブロックの処理結果を参照するためには

Copyright © 2011 by Information Processing Society of Japan and
The Instiute of Electronics, Information and Communication Engineers
All rights reserved.



(a-1) 参照ブロック位置①



(a-2) 参照ブロック位置②

1	2	3	4	5	6
2	3	4	5	6	7
3	4	5	6	7	8
4	5	6	7	8	9
5	6	7	8	9	10
6	7	8	9	10	11

(b) 並列処理可能ブロック

図 5　斜めブロック並列処理

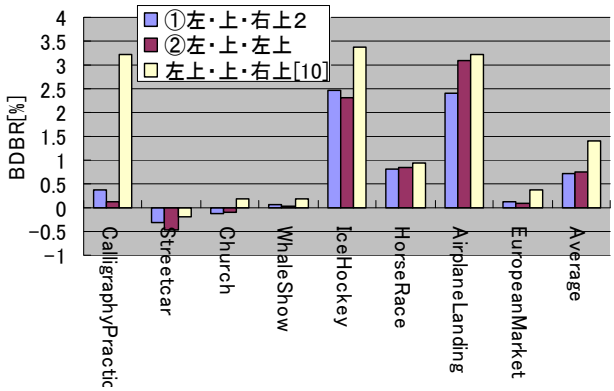


図 6　擬似ベクトル予測による符号化効率

図 5(b)のように斜めに隣接したブロックを並列に処理出来れば良い。この時並列に処理できるマクロブロック数は、画像のマクロブロック列数となり、図 2 のように 1 列おきに並列処理する場合に比べて 2 倍となる。図 5(b)のように斜めに隣接するマクロブロックを並列処理するためには、右上のブロック C の処理結果は参照できない。一方で図 5(a-1) (a-2)のブロック D やブロック E などは参照可能である。そこで、本提案手法ではブロック A, B, E もしくはブロック A, B, D の動きベクトルのメディアンを擬似予測ベクトルと定義し、擬似予測ベクトルを使用することで図 5(b)のように斜めに連続するブロックを並列処理する。

まず、これらの擬似ベクトル予測を用いた場合の符号化効率を評価する。図 6 に、左・上・右上ブロックによる正確なベクトル予測（図 2）に対する、①ブロック A, B, E を参照した擬似ベクトル予測(図 5(a-1))、②ブロック A B, D を参照した擬似ベクトル予測(図 5(a-2))、ブロック B, C, D を参照し水平並列処理する場合[10](図 3)の BDBR を示す。評価画像には ITE 標準動画像 8 種類を用いた。図 6 より、画像により最適な擬似予測ベクトルは異なるが、左のブロックを参照できない手法[10]に比べて、左のブロックを参照できる①②の手法は符号増加量が低いことがわかる。平均値では①の方が符号増加量がより低いため、本稿では①の擬似ベクトル予測を用いる。

4.3 フレームパイプライン処理による並列性向上

4.2節のようにマクロブロック間依存制約を緩和して並列処理しても、並列に処理できるマクロブロック数はフル HD 画像でも最大で列数である 120 であり、大きな並列処理能力をもつ GPU に対して並列処理可能数が充分であるとは言えない。そこで、文献[10]と同様に、フレーム間の並列性を利用してパイプライン処理を行い、複数のフレームにまたがるマクロブロック同士を並列に処理する。

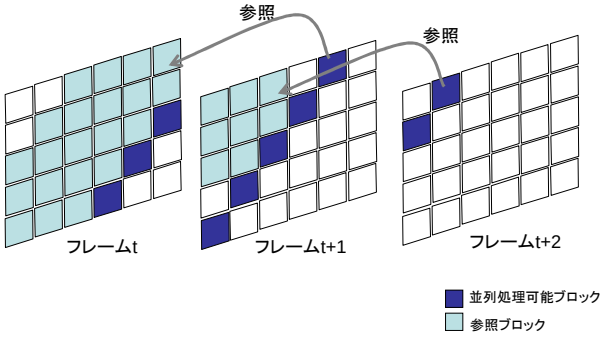


図 7　フレームパイプライン処理

H.264 エンコーダでは、既に処理が終了した過去フレームを参照フレームとして動き推定する。そのため、前フレームにおける参照範囲の処理が終了すれば、次のフレームの処理を行うことができる。そこで、パイプラインのように、前フレームの参照範囲の処理が終了したら次のフレームの処理を開始し、図 7 に示すように依存関係の無い複数のフレームのマクロブロック同士を並列に処理することで処理の並列性を向上させることができる。

予測ベクトルを算出するためには左や上のブロックの処理結果が必要になるため、フレーム内マクロブロック間の依存関係解消のためには、斜め 1 ライン毎に同期処理を行い処理の終了を確認する必要がある。フレーム間の依存関係解消のためには、参照フレームの参照範囲のマクロブロックの処理が終了してから、次のフレームの処理を始めるようにブロックの処理順を制御する。参照範囲を広くすれば並列処理可能フレーム数は少なくなり、参照範囲を狭くすれば並列処理可能数を高めることができる。但し、参照範囲を狭くし過ぎると動き推定精度は低下する。

4.4 仮想スレッドブロックを用いた稼働率向上

これまでに述べたように、擬似動きベクトル予測とフレームパイプラインにより、並列処理可能マクロブロック数を増加させる、即ち処理の並列性を向上させることができた。次に、CUDA アーキテクチャの並列処理能力を向上させ、GPU の稼働率を向上させる手法を提案する。

本稿の実装では、図 4 のようにスレッドブロックとマクロブロックを対応付けて並列処理している。これは処理の並列性や独立性を考えると妥当であるが、CUDA では SM あたりの最大同時実行スレッドブロック数が規定されていることから SM あたりで同時に処理できるマクロブロック数が制限されることになる。そのため、各 SM で同時に実行されるスレッド数を増加させて稼働率を向上させるには、各マクロブロックをより多くのスレッドで処理するしかない。しかし、本稿で扱う動き推定処理では、前述のようにマクロブロックあたりの処理スレッド数は少ないほうが有利である。

Copyright © 2011 by Information Processing Society of Japan and
The Instiute of Electronics, Information and Communication Engineers
All rights reserved.

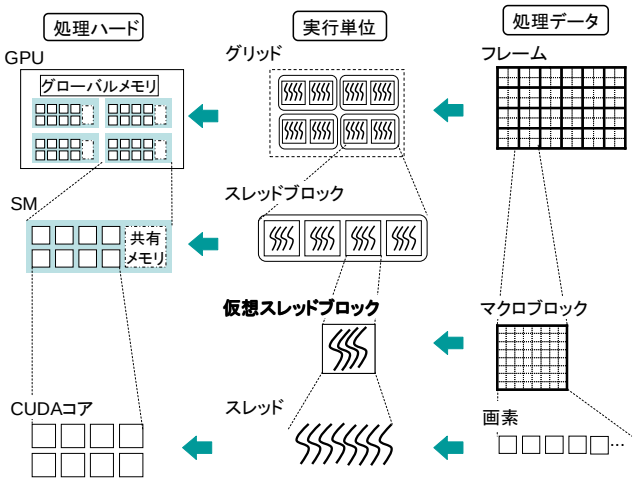


図 8　仮想スレッドブロックによる並列処理

そこで本稿では、SM、スレッドブロック、マクロブロックの対応付けを柔軟にするために、仮想スレッドブロックという概念を導入する。図 8 のように、1 マクロブロックの処理を行うのは 1 仮想スレッドブロックであり、1 スレッドブロックの中に複数の仮想スレッドブロックを持つことで、1SM 上で同時に実行されるマクロブロック数を増加させる。これにより、1 マクロブロックの処理を行うスレッド数が少なくとも、SM あたりの同時実行スレッド数を多くすることができる。

しかし、処理内容とスレッド・スレッドブロックの対応付けを変更したり、さらにはスレッドブロック内で同時処理するマクロブロック数を変更したりするのは容易ではない。図9 に簡易化した動き推定処理の CUDA 擬似コードを示す。1 スレッドブロックで 1 マクロブロック(MB)を処理するには、図9 の 27 行のようにスレッドブロックインデックスからマクロブロック番号を求めればよく、これによりマクロブロック・スレッドブロックレベルの並列処理が出来る。マクロブロック内のスレッドごとの並列処理を行うには 7、8 行のようにスレッドインデックスを用いて処理する画素位置を決定する。これを、SM あたりの同時実行スレッド数を増やすためにスレッドブロックとマクロブロックの対応を変更し同一スレッドブロックで複数のマクロブロックを処理するためには、多くの変更が必要となる。例えばスレッドブロック内のスレッド数を倍にしインデックスの奇偶で異なるマクロブロックを処理しようとすると、27 行のマクロブロック番号の計算にスレッドインデックスを加味する必要があるし、7、8 行のスレッドインデックスを用いた処理データ位置の計算にも変更が必要となる。特に後者はスレッドレベルで並列処理する部分の全てに変更が必要となり、書き換えはプログラマにとって大きな手間となる。

そこで、プログラムの大きな変更を必要とせずに仮想スレッドブロックを実現する手法を提案する。本手法によりスレッドブロックあたりの仮想スレッドブロック数の変更を簡易に行うことができ、最適化が容易になる。

CUDA では各スレッドブロックを構成するスレッドに x, y, z の 3 次のインデックスを付与することができる。本手法では、スレッドブロック構成の定義に使用していないスレッドインデックス次元を仮想スレッドブロックのインデックスとして使用することで、マクロブロック内のスレッド毎の処理を変えずに済む。例えば 1 マクロブロックの処理あたり x=16, y=2 の 2 次元 32 スレッド構成であれば、使用されていない 3 次元目の z インデックスを仮想スレッドブロックのインデックスとして扱いマクロブロック選択に用いることで、マクロブロック処理内には影響なく、同時に実行出来るスレッド数を増やすことができる。図 10 に仮想スレッドブロックを用いて書き換えた場合の擬似コードを示す。本手法を用いると図 10 の下線部

```
01  //GPU側の処理
02  __device__ computeSAD(){
03
04                    ⋮
05    //現画像ブロックと参照ブロックの画素値の差分絶対値計算
06    sad +=
07     abs(curr[MBPosY + threadIdx.y][MBPosX + threadIdx.x]
08         - ref[MVy + threadIdx.y][MVx + threadIdx.x]);
09                    ⋮
10  }

21  __global__ kernel1 ( ){
22
23      //共有メモリ上に使用リソースを確保
24      __shared__ resource_t resource;
25
26     //処理するMB番号
27      MBnum = blockIdx.x;
28
29      // MBnum番のMBをresourceを使って処理
30      ME ( MBnum, &resource );
31  }

41  //CPU側の処理
42  main ( ){
43
44      numThreads = {x, y};  // x, yの2次元のブロック構成
45      numBlocks = {numMBs}; // ブロック数はMB数
46
47      //GPU処理をcall
48      kernel1 <<< numBlocks, numThreads>>> ();
49  }
```

図 9　動き推定の擬似コード

```
01  //GPU側の処理
02  __device__ computeSAD(){
03                ⋮
04
05    //現画像ブロックと参照ブロックの画素値の差分絶対値計算
06    sad
07      += abs(curr[MBPosY + threadIdx.y][MBPosX + threadIdx.x]
08             - ref[MVy + threadIdx.y][MVx + threadIdx.x]);
09                ⋮
10  }

21  __global__ kernel2( ){
22
23      //処理するMB数分リソース確保
24      __shared__ resource_t resource[blockDim.z];
25
26      //仮想スレッドブロック数を加味してMB番号を決定
27      MBnum = blockIdx.x * blockDim.z + threadIdx.z
28
29      // ME()の内部には影響なし
30      ME ( MBnum, &resource[threadIdx.z] );
31  }

40  //CPU側の処理
41  main ( ){
42
43      numThreads = {x, y, z}; //スレッドブロック構成にz方向を追加
44      numBlocks = {numMBs / z}; //スレッドブロック数は（MB数/z）
45
46      //GPU処理をcall
47      kernel2 <<< numBlocks, numThreads>>> ();
48  }
```

図 10　仮想スレッドブロックを用いた擬似コード

Copyright © 2011 by Information Processing Society of Japan and
The Instiute of Electronics, Information and Communication Engineers
All rights reserved.

表 1　評価環境

		GeForce GTX 580	Tesla S1070
GPU	コア数/SM	32	8
	SM 数	16	30
	動作周波数	1.54 GHz	1.44 GHz
	メモリ	1.6 GB	4 GB
	メモリバンド幅	192 GB/sec	102 GB/sec
	ComputeCapability	2.0	1.3
	CUDA	3.2	3.2
CPU		Intel Core i7	Intel Xeon
	コア数	4	4
	動作周波数	3.07 GHz	2.83 GHz
	メモリ	3 GB	4GB
	OS	Windows XP	Linux 2.6.18

表 2　評価画像

	概要	開始フレーム
Calligraphy Practice	和室で習字をする女性のタイトショット	100
Streetcar	街並を背景に走る電車の風景	100
Church	中世建築の教会の風景	100
Whale Show	水族館のショーでジャンプするシャチと観客の風景	100
IceHockey	フェイスオフの場面	300
HorseRace	スタート直後の競馬の場面	100
Airplane Landing	旅客機が飛行場に着陸する場面	150
European Market	花売りなど市場のルーズショット	100

のみを書き換えれば良い。図 10 では、27 行目のマクロブロック番号の計算部分に仮想スレッドブロックインデックス（スレッドインデックス z）を加味し、7、8 行目のスレッド並列処理部分は変更を必要としない。さらに、スレッドブロックあたりの仮想スレッドブロック数を変更する時には、43, 44 行の z の値を変更するだけで良い。

ここで、共有メモリ変数はスレッドブロック内の全スレッドで共有されるので、同一スレッドブロック内の他の仮想スレッドブロックとも共有されてしまう。そこで、仮想スレッドブロック内のみで共有したい変数は、図 10 の 24 行のように仮想スレッドブロック数分の配列とし、図 10 の 30 行のように仮想スレッドブロックインデックス(threadIdx.z)を添字としてアクセスすることで、仮想スレッドブロック毎に異なる領域を使用することができる。

なお、仮想スレッドブロック内のスレッド数が 32 以外の場合にはスレッド間同期処理に注意が必要である。現在の CUDA では、同期処理はスレッドブロック内の全てのスレッドとしか行えない。そのため、syncthreads による明示的な同期処理はスレッドブロック内の全仮想スレッドブロック、全スレッドが同期されてしまう。しかし、CUDA では特に同時に処理を実行する 32 スレッドをワープと呼び、ワープ内のスレッド群は明示的に同期処理を行わなくても、順序制約が守られることが保証されている[6]。そのため、仮想スレッドブロック内のスレッド数が 32 の場合は同期処理に関して気にせずとも良い。仮想スレッドブロック内のスレッド数が 32 より多い場合には、共有メモリを使うなどして、同一仮想スレッドブロック内のみで同期を行うなどの工夫が必要になる。

表 3　エンコードパラメータ

Profile	Baseline Profile
QP	16, 20, 24, 28
I フレーム間隔	30
参照フレーム数	1
RD-optimization	無効
エンコードフレーム数	30
動き推定アルゴリズム	EPZS
探索範囲	32

表 4　動き推定パラメータ

	JM（高画質）	JM（高速）	GPU
整数精度 ME	16x16, 16x8, 8x16, 8x8, 8x4, 4x8, 4x4	16x16, 16x8, 8x16	16x16, 16x8, 8x16
小数精度 ME	16x16, 16x8, 8x16, 8x8, 8x4, 4x8, 4x4	16x16, 16x8, 8x16 のうち最良のみ	16x16, 16x8, 8x16 のうち最良のみ
動きベクトル予測	○	○	擬似予測
スキップ推定	○	○	擬似推定
探索初期点候補数	68	8	8

5. 性能評価

本章では、提案する GPU 動き推定の符号化効率・処理速度性能について評価を行う。

5.1 評価環境

表 1 に評価に用いた NVIDIA GeForce GTX580 と Tesla S1070 の詳細を示す。Tesla S1070 は GPU4 基が搭載されたマルチ GPU であるが、本評価では 1 基のみを用いた。評価画像には表 2 に示すフル HD（1920x1080 画素）サイズの ITE 標準動画像[3]8 種を用いた。評価に際して、参照実装ソフトウェア JM16.0[4]を、CUDA を用いて GPU 並列化した。GPU エンコーダの処理速度・符号化効率比較対象として CPU エンコーダを用いるが、これには JM16.0 を使用し、並列化・SIMD 化などの高速化は行っていない。エンコードパラメータを表 3 に、評価に用いた H.264 エンコーダの動き推定パラメータを表 4 に示す。GPU 動き推定の符号化効率はオリジナルの JM である JM(高画質)と比較する。しかし、オリジナル JM は符号化効率が高い代わりに演算量が非常に多い。そこで、演算量と符号化効率のトレードオフを検討し、演算量を削減してもある程度の符号化効率を維持できる事を事前実験により確認した。提案手法は、演算量を削減した JM(高速)をベースに GPU 実装し、速度評価も JM(高速)と比較する。JM(高速)と GPU 実装では、小数精度動き推定は整数精度動き推定後に最適ブロックサイズのみで行う。また、EPZS における探索初期点候補算出においても search range dependent predinctor[12]など、評価点数の多い手法は除いている。さらに、GPU 実装では動きベクトル予測とスキップ推定で右上ブロックの代わりに右 2 つ上ブロックの処理結果を使用する。

Copyright © 2011 by Information Processing Society of Japan and
The Instiute of Electronics, Information and Communication Engineers
All rights reserved.

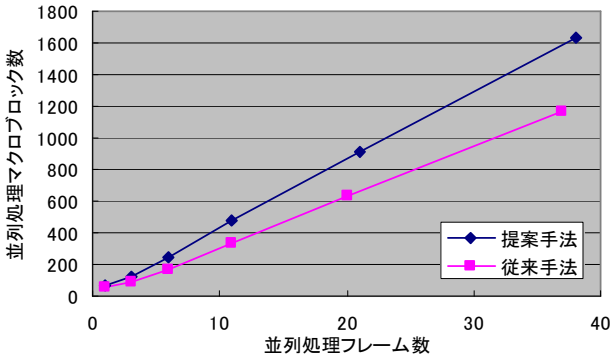


図 11　擬似ベクトル予測動き推定の並列処理 MB 数

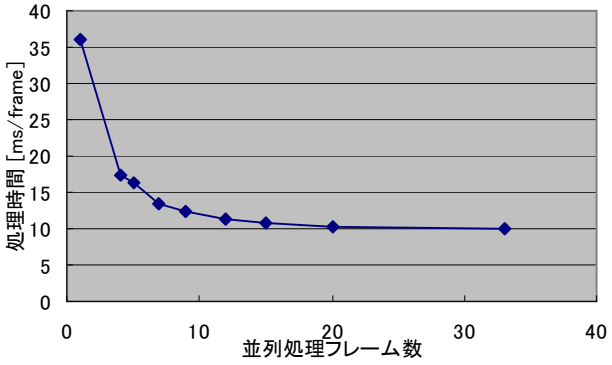


図 12　並列処理フレーム数と処理時間

5.2 処理時間評価

はじめに、従来の正確なベクトル予測を用いる場合と提案する擬似ベクトル予測を用いる場合の最大並列処理可能マクロブロック数を図 11 に示す。図 11 の横軸はフレームパイプライン処理した場合の並列処理フレーム数である。図 11 より、従来手法に比べて提案手法は並列処理できるマクロブロック数が 1.5 倍多いことがわかる。これにより、メニコア GPU の並列処理能力をより活用することが可能であり、さらに、GPU のコア数が増えた場合にもよりスケーラブルな性能向上が可能になる。また、並列処理マクロブロック数が同じであれば、提案手法の方が並列処理フレーム数が少ない。これは提案手法の方がフレーム画像を保持するメモリ領域が少なく済むことを示す。さらに、並列処理フレーム数が少ないと、多くのフレームの入力を待つ必要がなく遅延を短くすることができる。

次に、提案擬似ベクトル予測手法において、並列に処理するフレーム数を変化させた場合の処理時間を評価する。並列処理フレーム数を変化させた場合の GTX580 による動き推定処理時間(8 画像平均)を図 12 に示す。図 12 より、並列処理フレーム数が増加すると処理時間が削減できることがわかる。これは、並列処理フレーム数が増えると並列処理可能なマクロブロック数も増え、遊休するコアが削減できたためであると考えられる。しかし並列処理フレーム数がある程度以上になると GPU コア数の限界から高速化効果は少なくなる。さらに、並列に処理するフレーム数が増えるとメモリ使用量も増えるため、並列処理フレーム数は GPU のグローバルメモリ容量も考慮して決定する必要がある。

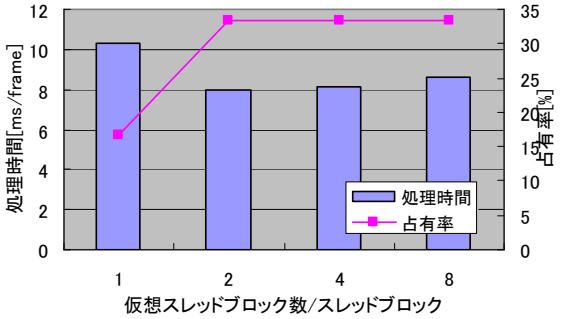


図 13　仮想スレッドブロックによる処理時間変化 (GTX580)

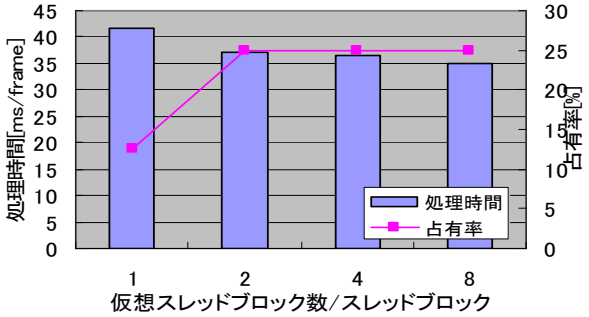


図 14 仮想スレッドブロックによる処理時間変化 (Tesla S1070)

次に、仮想スレッドブロックの効果を評価する。GTX580 と Tesla S1070 それぞれにおいて、スレッドブロックあたりの仮想スレッドブロック数を 1 から 8 まで変化させた場合の平均処理時間を図 13、図 14 にそれぞれ示す。図 13、図 14 における占有率とは、最大同時実行スレッド数に対する実同時実行スレッド数の比率である[6]。占有率が高いほどコアの稼働率が高くできることを表す。図 13、図 14 を見ると、仮想スレッドブロック数を 2 以上にすることで占有率が向上し、処理速度も向上していることがわかる。これは、同時実行スレッドブロック数で制限されていた同時実行スレッド数を増やすことができたため、コアの稼働率が向上し高速化したと考えられる。しかし、処理時間が最短になる仮想スレッドブロック数は GTX580 では 2、Tesla S1070 では 8 と、環境によって異なることがわかる。これには、占有率以外にも GPU のレジスタ数や共有メモリ量、スレッドスケジューラなどアーキテクチャの違い、動作周波数やメモリバンド幅など様々な要因の影響が考えられる。そのため、環境によってスレッドブロックあたりのスレッド数・処理マクロブロック数を最適化する必要がある。提案する仮想スレッドブロック処理を用いることで、このような環境依存の最適化を容易に行うことができるようになる。

なお、図 13、図 14 では仮想スレッドブロック数を増やしても、使用レジスタ数・共有メモリ量の制約により占

Copyright © 2011 by Information Processing Society of Japan and
The Instiute of Electronics, Information and Communication Engineers
All rights reserved.

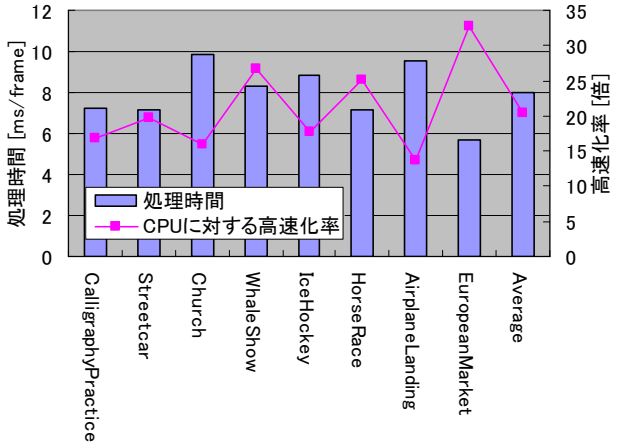


図 15　各画像の処理時間と高速化率

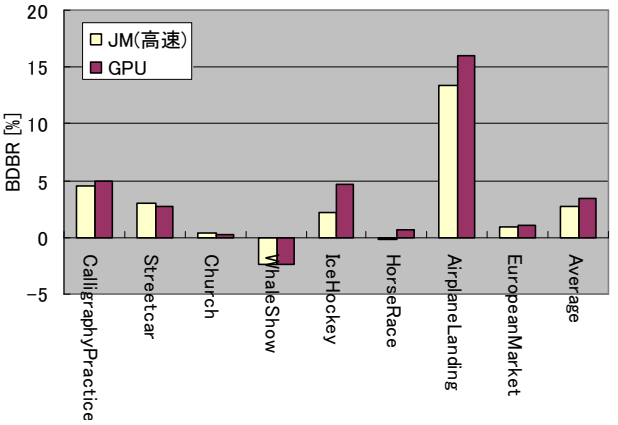


図 16　提案手法の符号化効率

有率の向上は頭打ちになっている。よって、使用レジスタ数・共有メモリ量を削減出来れば更なる高速化も可能だと考えられる。

次に GTX 580 での各画像の処理時間と CPU に対する高速化率を図15 に示す。フレームパイプラインの並列処理フレーム数は 20、仮想スレッドブロック数は 2 とした。図15 より、動き推定の処理時間はフレームあたり 5.7msec から 9.8msec の平均 8.0msec となり、CPU 処理に対して平均 20.3 倍の高速化となった。フレームレートは平均 125fps であり、フル HD 動画像のリアルタイム動き推定処理が充分可能である。

5.3 符号化効率評価

最後に、本 GPU 動き推定実装の符号化効率を評価する。図16 に、JM(高画質)に対する JM(高速)と GPU 実装の BDBR を示す。GPU 実装は表 4 に示すように、JM(高速)に対して擬似ベクトル予測を用いるなどの違いがある。図16 より、提案 GPU 実装は JM(高速)と同等の符号化効率であることがわかる。IceHockey, AirplaneLanding などでは JM(高速)よりも BDBR が高いが、これは動きベクトル予測等において右上のブロックの代わりに右 2 つ上のブロックを使用したためであると考えられる。提案 GPU 実装の JM(高画質)に対する符号量増加は平均で 3.5%であり、目視にはほとんど影響がない程度であると考えられる。

6. おわりに

本稿では、動きベクトル予測を用いた GPU H.264 動き推定の高速化を行った。GPU のようなメニコアアーキテクチャでは、最大限の性能を得るためには、処理の並列性を多く引き出し、かつ、プロセッサの稼働率を向上させるために実行環境に応じた最適化することが重要である。H.264 動き推定処理において、動きベクトル予測は符号化効率を向上させるのに効果的だが、並列に処理できるブロック数を減少させてしまう。そこで本稿では、ブロック間の依存制約を緩和して符号化効率を維持しつつ並列性を向上させる擬似ベクトル予測手法を提案した。さらに、実行環境適応を容易にして CUDA GPU のプロセッサ稼働率を向上させる仮想スレッドブロック処理手法を提案した。仮想スレッドブロック処理によるプロセッサ稼働率向上手法は、動き推定に限らず、他の多くの処理にも適用可能であると期待できる。

提案手法を評価した結果、フル HD 画像において GPU 動き推定は CPU 処理に比べて 20 倍の高速化となり、処理時間は平均 8.0 msec/frame となった。この時フレームレートは 125 fps でありリアルタイム処理が充分可能である。符号化効率においては、本実装は基準エンコーダ JM に対して平均 3.5 %の符号量増加に抑えることができた。

ただし、動きの大きい画像などで符号量増加が大きくなる場合もあり、今後、更なる符号化効率向上手法の検討が必要である。また、プロセッサの稼働率をさらに向上させて、より一層の高速化を実現するためには、使用する共有メモリ量やレジスタ数を削減できるアルゴリズムの検討も必要である。

参考文献

[1]ITU-T Recommendation H.264 “Advanced video coding for generic audiovisual services,” May 2003.
[2]G.J. Sullivan, T. Wiegand, "Rate-distortion optimization for video compression," Signal Processing Magazine, IEEE , vol.15, no.6, pp.74-90, Nov 1998.
[3]映像情報メディア学会, “ITE 標準動画像”, http://www.nes.or.jp/gaiyo/hanpu.html
[4]A. M Tourapis, K. Sühring, G. Sullivan, “H.264/MPEG-4 AVC Reference Software Manual” , ITU-T SG16 Q.6, 2007.
[5]G. Bjontegaard, “Calculation of average PSNR differences between RD-curves,” ITU-T Q.6/16, 2001.
[6]NVIDIA , “CUDA Programing Guide” , 2010.
[7]W.N. Chen1, H.M. Hang1, “ H.264/AVC Motion Estimation Implementation on Compute Unified Device Archtecturer (CUDA)” , Multimedia and Expo, 2008 IEEE International Conference on, pp.697-700, 2008.
[8]D. Ailawadi, M. K. Mohapatra, A. Mittal, “Frame-Based Parallelization of MPEG-4 on Compute Unified Device Architecture (CUDA), IEEE Advance Computing Conference (IACC), pp.267-272, 19-20 Feb. 2010.
[9]M. Kung, O. Au, P. Wong, and C. Liu, “Block based parallel motion estimation using programmable graphics hardware,” in Proc. Int. Conf. Audio, Language and Image Processing, pp. 599–603, 2008.
[10] A. Obukhov, “GPU-Accelerated Video Encoding”, NVIDIA GPU Technology Conference, 2010.
[11] 鷹野, 森吉, "GPGPU における動き探索の空間的ベクトルサイズ相関性を活かしたスケジューリング手法", 電子情報通信学会 第 25 回信号処理シンポジウム, C5-4, Nov., 2010.
[12] A. M. Tourapis, “Enhanced predictive zonal search for single and multiple frame motion estimation,” in Proc. VCIP, pp. 1069–1079, 2002.

Copyright © 2011 by Information Processing Society of Japan and
The Instiute of Electronics, Information and Communication Engineers
All rights reserved.