

ヒストグラム生成を高速化するための OpenACC オプティマイザの検討

池田 圭¹ 伊野 文彦¹ 萩原 兼一¹

概要：本稿では，OpenACC により記述されたヒストグラム生成のコードを GPU (Graphics Processing Unit) 向けに自動最適化するオプティマイザについて検討する．GPU によるヒストグラム生成では，局所ヒストグラムの生成により，アトミック演算による書き込みの競合を削減できる．この高速化手法は OpenACC で記述できる．しかし，OpenACC ディレクティブを逐次コードに挿入するだけでは記述できない．したがって，逐次コードを書き換える必要があるため，コードの性能可搬性が損なわれる．そこで，OpenACC コードからヒストグラム生成のコーディングパターンを抽出し，GPU 向けの高速化手法を適用するオプティマイザを提案する．性能可搬性を損なわずにコードを最適化するためには，コードの書き換えを自動化する機構が有用であると考えられる．予備実験では，オプティマイザによる最適化手順にしたがってコードを手動で書き換えた．NVIDIA 社の GPU を用いて得た結果により，局所ヒストグラムの生成による高速化を確認できた．

キーワード：ヒストグラム生成, 自動最適化, OpenACC, GPU

A Preliminary OpenACC Optimizer for Accelerating Histogram Generation

KEI IKEDA¹ FUMIHIKO INO¹ KENICHI HAGIHARA¹

Abstract: In this paper, we present a preliminary OpenACC optimizer that automatically optimizes a histogram generation code for a graphics processing unit (GPU). In GPU-accelerated histogram generation, atomic write conflicts can be reduced by using multiple local histograms. This acceleration method can be implemented as OpenACC code. However, OpenACC directives are not sufficient to apply the method to the sequential code. Owing to this restriction, the sequential code must be partially rewritten, and thus, the performance portability is eliminated from the code. To solve this issue, we propose an optimizer that automatically identifies histogram generation part from the OpenACC code to apply the acceleration method for the GPU. We think that automating code rewriting procedure is useful to optimize the code performance without losing the performance portability. In preliminary experiments, we manually rewrite four application code according to the optimization procedure. Results obtained on an NVIDIA GPU show that our optimization procedure successfully accelerates GPU-enabled histogram generation by using multiple local histograms.

Keywords: histogram generation, automatic optimization, OpenACC, GPU

1. はじめに

近年，高速な科学計算のために，GPU (Graphics Process-

ing Unit) や Xeon Phi[1] などのアクセラレータを搭載した計算機が広く用いられている．このようなアクセラレータ向けのプログラミングモデルとして OpenACC (Open ACCelaretor) [2] がある．OpenACC では，C や Fortran などにより記述した逐次コードに対して，アクセラレータ

¹ 大阪大学大学院情報科学研究科コンピュータサイエンス専攻
Department of Computer Science, Graduate School of Information Science and Technology, Osaka University

での動作を指示するディレクティブを挿入することにより、容易にアクセラレータ向けのコードを記述できる。

従来、アクセラレータ向けのコードを記述するためには、NVIDIA 社の GPU 向けのプログラミングモデルである CUDA (Compute Unified Device Architecture) [3] のように、アクセラレータごとに固有のプログラミング言語を用いる必要があった。一方、OpenACC は異なるアクセラレータ間で共通のコードを実行できる。さらに、アクセラレータでの動作をディレクティブにより記述することから、アクセラレータ向けの性能最適化は、原則としてディレクティブの挿入、修正および削除のみで完結し、逐次コードを書き換える必要がない。したがって、OpenACC により記述したコードは性能可搬性を備える。ここで性能可搬性とは、異なる計算環境において、同一のコードから得た実行形式プログラムを高い性能で実行できる性質のことである。

しかし、現状では、OpenACC ディレクティブの表現力はアクセラレータに固有のプログラミング言語と比較して低く、コンパイラによるコードの最適化も不十分である。そのため、現実的にはアクセラレータ向けの性能最適化のために、逐次コードを書き換える必要がある。これにより、コードの性能可搬性が損なわれている。

例えば、統計処理や画像処理などに広く用いられるヒストグラム生成を GPU 上で高速化する手法（以降、分散化手法）は、OpenACC ディレクティブのみで記述できない。したがって、分散化手法を適用するためには、逐次コードを書き換える必要がある。しかし、書き換え後のコードから得たプログラムを、GPU 以外の計算環境においても高い性能で実行できるとは限らない。

そこで、本研究では、OpenACC により記述されたヒストグラム生成のコードを、GPU 向けに自動最適化するオプティマイザについて検討する。このオプティマイザは、OpenACC コードからヒストグラム生成のコーディングパターンを抽出し、抽出したコード部分が GPU において高速に動作するように逐次コードを書き換え、OpenACC ディレクティブを挿入および修正する。OpenACC により記述されたヒストグラム生成のコードを分散化手法により最適化し、かつコードの性能可搬性を保つためには、コードを GPU 向けの実行形式プログラムにコンパイルする前に、分散化手法を自動で適用する機構があればよい。したがって、オプティマイザにより、元のコードの性能可搬性を保ったまま、GPU におけるプログラムの実行性能を高めることができる。

以降では、まず 2 章で関連研究について述べ、3 章で分散化手法について説明する。次に 4 章で OpenACC によるヒストグラム生成の現状と問題点を明らかにし、5 章でヒストグラム生成のコードを GPU 向けに最適化するオプティマイザについて説明する。6 章でオプティマイザによ

る高速化を確認し、最後に 7 章で今後の課題とともに本稿をまとめる。

2. 関連研究

性能可搬性を備えるコードを記述するための方法として、OpenACC の拡張を用いる方法がある。中野ら [4] は、アクセラレータのメモリ容量を超えるデータを処理するために、データの分割処理、および分割したデータのパイプライン処理を記述できる PACC (Pipelined ACCelerator) ディレクティブを提案している。Hoshino ら [5] は、メモリ参照効率が高いデータレイアウトが計算環境のアーキテクチャに依存することに着目し、データレイアウトの変換を記述できるディレクティブを提案している。Beyer ら [6] は、ポインタ型で宣言した構造体変数をアクセラレータに転送するとき、構造体のメンバが常にシャローコピーされる問題に着目し、転送するメンバを詳細に指定できるディレクティブを提案している。これらの拡張は、OpenACC ディレクティブの表現力を増すことにより、性能可搬性を備えるコードの記述を可能にする。一方、本研究は、すでに記述された OpenACC コードの性能可搬性を保つために、GPU 向けの性能最適化に伴うコードの書き換えを自動化する。

OpenACC と同様に計算の高速化を目的とするディレクティブとして、OpenMPC[7] や XcalableMP[8] などがある。OpenMPC は OpenMP の拡張であり、GPU 向けのコードを記述できる。一方、XcalableMP は PC クラスタなどの分散メモリ型並列計算環境向けのコードを記述できる。さらに、XcalableMP と OpenACC を統合することにより、GPU クラスタなどのアクセラレータクラスタ向けのコードを記述できる XcalableACC[9] が提案されている。これらに対しても、GPU 向けの最適化に伴う逐次コードの書き換えを自動化するオプティマイザの機構は有用だと思われる。

3. GPU によるヒストグラム生成の高速化

ヒストグラムとは、特定の集合から得た一連の観測値を、あらかじめ定めた区間ごとに分類し、計数することにより得られる度数分布表のことである。ヒストグラムは、区間ごとの度数を格納するビンにより構成される。以降では、観測値を得る集合を母集団と呼ぶ。

図 1 にヒストグラムを生成する逐次コードの例を示す。この例では、要素数 n の配列 `data` が母集団であり、`data` の要素の値を観測値として計数している。ヒストグラムを要素数 `bins` の配列 `hist` で表現し、ビンを `hist` の要素で表現している。図 1 では、2~4 行目でヒストグラムを初期化し、7~9 行目でヒストグラムを生成している。

GPU によるヒストグラム生成では、一般に、母集団を分割して各スレッドに割り当て、並列にヒストグラムを更

```

1:  /* ヒストグラムの初期化 */
2:  for (int i = 0; i < bins; i++) {
3:      hist[i] = 0
4:  }
5:
6:  /* ヒストグラム生成 */
7:  for (int i = 0; i < n; i++) {
8:      hist[data[i]]++;
9:  }

```

図 1 ヒストグラム生成を C により記述したコード例

新する。複数のスレッドがヒストグラムを共有する場合、同一のビンと同時に更新する可能性があるため、これらの更新を排他的に処理するための演算（アトミック演算）が必要である。アトミック演算による書き込み先が競合することにより、一連の書き込みが逐次処理され、実行効率が低下する。したがって、高速化を達成するためには、書き込み先の競合を削減する工夫が必要である。

単純手法では、すべてのスレッドが1つのヒストグラムを共有する。図 2 に単純手法によるヒストグラム生成の例を示す。この例では、data を分割して各スレッドに割り当て、並列に hist を更新している。

分散化手法は、ヒストグラムのコピー（局所ヒストグラム）を複数個用意し、母集団の局所領域ごとに局所ヒストグラムを生成する。その後、それらを1つに集約することにより最終的なヒストグラムを得る。図 3 に分散化手法によるヒストグラム生成の例を示す。この例では、2 個の局所ヒストグラムを用意し、局所ヒストグラムの集合を2次元配列 locals で表現している。

分散化手法は、局所ヒストグラムの個数を増やすことにより書き込み先を分散し、競合を削減できる。一方、局所ヒストグラムの個数が増えると、広範囲のメモリ領域に対するランダムな書き込みが発生し、性能を低下させる。つまり、(A) 競合の削減および (B) 書き込み領域の局所性はトレードオフの関係にあるため、高速化するためには局所ヒストグラムの個数を適切に決定する必要がある。しかし、我々の知る限り、任意のヒストグラム生成において最適な局所ヒストグラムの個数を決定する手法は確立されていないため、実験的に個数を決定する必要がある。

4. OpenACC によるヒストグラム生成

3 章で説明した単純手法および分散化手法を、OpenACC により記述する方法について説明する。以降では、CPU をホスト、GPU をデバイスと呼ぶ。

図 4 に、単純手法によるヒストグラム生成の記述例を示す。このコードは、図 1 に以下の OpenACC ディレクティブを挿入するだけで記述できる。

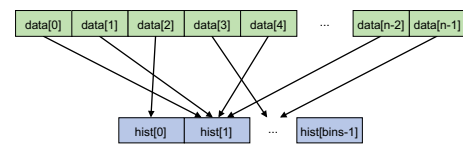


図 2 単純手法によるヒストグラム生成の例

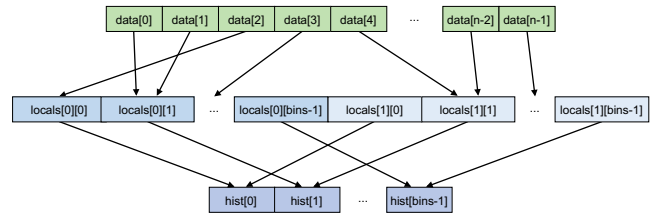


図 3 分散化手法によるヒストグラム生成の例

- #pragma acc data : ホスト・デバイス間のデータ転送を指示
- #pragma acc parallel loop : for ループの並列化を指示
- #pragma acc atomic update : アトミック演算による変数の更新を指示

すなわち、逐次コードの書き換えが不要であるため、このコードは性能可搬性を備える。

図 4 では、1 行目でホスト・デバイス間のデータ転送を指示し、4～7 行目でヒストグラムを初期化し、10～14 行目でヒストグラムを生成している。なお、data はホストからデバイスに転送するだけでよいいため、copyin 節を用いている。同様に、hist はデバイスからホストに転送するだけでよいいため、copyout 節を用いている。

図 5 に、分散化手法によるヒストグラム生成の記述例を示す。この例では、8 個の局所ヒストグラムを用意し、gang ごとに局所ヒストグラムを割り当てている。ここで gang とは、デバイスにおいて論理的に同時実行されるスレッドの集合であり、CUDA におけるスレッドブロックに相当する。このコードは、図 1 に OpenACC ディレクティブを挿入するだけでは記述できないため、逐次コードを書き換える必要がある。したがって、このコードは性能可搬性を持たない。

図 5 では、局所ヒストグラムの個数を num_locals とし、局所ヒストグラムの集合を要素数 num_locals×bins の配列 locals として表現している。locals の先頭から bins 個ずつの要素を、各局所ヒストグラムに割り当てている。まず、2 行目で局所ヒストグラムのためのメモリ領域をホストに確保し、4 行目でホスト・デバイス間のデータ転送およびデバイスのメモリ領域の確保を指示している。次に、13～18 行目で局所ヒストグラムを初期化し、21～28 行目で局所ヒストグラムを生成し、31～37 行目で局所ヒストグラムを集約している。なお、locals はデバイスにメモリ領域を確保するだけでよいいため、create 節を用いている。

```

1:  #pragma acc data copyin(data[0:n]) copyout(hist
    [0:bins])
2:  {
3:  /* ヒストグラムの初期化 */
4:  #pragma acc parallel loop
5:  for (int i = 0; i < bins; i++) {
6:      hist[i] = 0
7:  }
8:
9:  /* ヒストグラム生成 */
10: #pragma acc parallel loop num_gangs(GANG)
    vector_length(VECTOR)
11: for (int i = 0; i < n; i++) {
12:     #pragma acc atomic update
13:     hist[data[i]]++;
14: }
15: }
```

図 4 OpenACC による単純手法の記述例

また、局所ヒストグラムを初期化および集約する for 文は、`#pragma acc parallel loop tile` ディレクティブにより並列化している。なお、集約にはアトミック演算を用いている。

局所ヒストグラムの生成において、gang ごとに局所ヒストグラムを割り当てるためには、各 gang のインデックスを取得する必要がある。しかし、OpenACC のランタイムライブラリは、gang のインデックスを取得する関数を提供していない。そのため、計算によりインデックスを取得する必要がある。

インデックスを計算するためには、各 gang が処理するイテレーションに固有の値を用いればよい。しかし、OpenACC では、並列化するループの各イテレーションを gang に割り当てる方法を定めていない。そこで、ループをブロック状に分割して各 gang に割り当てていると仮定する。この仮定の元で、ループ回数が n である 1 重ループを並列化するとき、生成する gang の個数を g 、各ブロックに含まれるイテレーションの個数を c とすると、 $c = \lceil n/g \rceil$ である。各 gang が持つループ変数の値は連続しているため、各 gang のインデックスは、ループ変数の値を c で割った商である。

図 5 では、各 gang のインデックスを局所ヒストグラムの個数で割った余りを、各 gang が更新する局所ヒストグラムのインデックスとしており、 c を `num_iters`、各 gang のインデックスを `gang_id`、各 gang が更新する局所ヒストグラムのインデックスを `charge` で表現している。

5. OpenACC オプティマイザ

OpenACC コードを GPU 向けに自動最適化するオプティマイザについて検討する。オプティマイザは、まず (1) 入力として与えられた OpenACC コードからヒストグラム生成のコーディングパターンを抽出し、次に (2) 局所

```

1:  const int num_locals = 8;
2:  int *locals = (int*)malloc(sizeof(int)*bins*
    num_locals);
3:
4:  #pragma acc data copyin(data[0:n]) copyout(hist
    [0:bins]) create(locals[0:bins*num_locals])
5:  {
6:  /* ヒストグラムの初期化 */
7:  #pragma acc parallel loop
8:  for (int i = 0; i < bins; i++) {
9:      hist[i] = 0
10: }
11:
12: /* 局所ヒストグラムの初期化 */
13: #pragma acc parallel loop tile(256,1)
14: for (int j = 0; j < num_locals; j++) {
15:     for (int i = 0; i < bins; i++) {
16:         locals[bins*j + i] = 0;
17:     }
18: }
19:
20: /* 局所ヒストグラム生成 */
21: #pragma acc parallel loop num_gangs(GANG)
    vector_length(VECTOR)
22: for (int i = 0; i < n; i++) {
23:     int num_iters = (int)ceil((double)n/GANG);
24:     int gang_id = i/num_iters;
25:     int charge = gang_id%num_locals;
26:     #pragma acc atomic update
27:     locals[bins*charge + data[i]]++;
28: }
29:
30: /* 局所ヒストグラムの集約 */
31: #pragma acc parallel loop tile(256,1)
32: for (int j = 0; j < num_locals; j++) {
33:     for (int i = 0; i < bins; i++) {
34:         #pragma acc atomic update
35:         hist[i] += locals[bins*j + i];
36:     }
37: }
38: }
39:
40: free(locals);
```

図 5 OpenACC による分散化手法の記述例

ヒストグラムを初期化、生成および集約する逐次コードに書き換え、OpenACC ディレクティブを挿入および修正する。ただし、逐次コードは C により記述されているものとする。

ユーザは、OpenACC コードを GPU 向けの実行形式プログラムにコンパイルする前に、オプティマイザを用いて最適化する (図 6)。最適化の機能がオプティマイザとしてコンパイラから分離されているため、コンパイル環境に依存せずにコードを最適化できる。

以降では、(1) および (2) の手順について説明する。

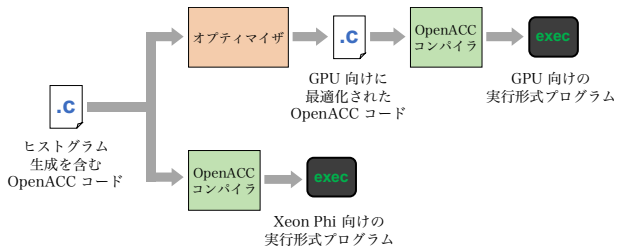


図 6 OpenACC コードから実行形式プログラムを得るまでの流れ

5.1 ヒストグラム生成のコーディングパターンの抽出

最適化は、入力として与えられた OpenACC コードから、以下の 2 つの条件を満たす for 文をヒストグラム生成のコーディングパターンとして抽出する。

- (C-1) デバイスが並列に実行する
- (C-2) ループ内でアトミック演算により配列の要素をインクリメントしている

このとき、(C-2) における配列がヒストグラムを表現しているとみなす。

OpenACC コードにおいて、for 文に `#pragma acc loop` ディレクティブが付加されているとき (C-1) を満たす。このディレクティブの代わりに、複合ディレクティブである `#pragma acc kernels loop` ディレクティブまたは `#pragma acc parallel loop` ディレクティブが付加されていてもよい。一方、配列の要素をインクリメントするコードに `#pragma acc atomic update` ディレクティブが付加されているとき (C-2) を満たす。ただし、`update` 節は省略されていてもよい。

5.2 分散化手法を適用するためのコードの書き換え

分散化手法を適用するために、抽出したコード部分に対して以下の処理を追加する。

- (a) 局所ヒストグラムのためのメモリ領域の確保
- (b) 局所ヒストグラムの初期化および集約
- (c) 局所ヒストグラムの生成

まず (a) のために、抽出した for 文を含み、かつ `#pragma acc data` ディレクティブが付加されたブロックの前後に、ホストに局所ヒストグラムのためのメモリ領域を確保および解放するコードを挿入する。さらに、デバイスにメモリ領域を確保するために、`#pragma acc data` ディレクティブに `create` 節を追加する。`create` 節の引数には、局所ヒストグラムを表現する配列を指定する。すでに `create` 節が存在する場合、局所ヒストグラムを表現する配列を引数に追加する。

最適化は、局所ヒストグラム全体の大きさがアクセラレータのメモリ容量を超えない範囲で、用意する局

所ヒストグラムの個数を自動的に設定する。また、抽出した for 文において、ヒストグラムを表現する配列が複数個存在する場合、それぞれのヒストグラムに対して局所ヒストグラムを用意する。

次に (b) のために、抽出した for 文の前後に局所ヒストグラムを初期化および集約するコードを挿入する。初期化および集約はいずれも for 文による 2 重ループで記述し、外側のループで各局所ヒストグラムを走査し、内側のループで局所ヒストグラムの各ビン走査する。この for 文に `#pragma acc parallel loop tile` ディレクティブを付加し、デバイスで並列実行する。なお、集約にはアトミック演算を用いる。

(c) では、4 章で説明したように、gang ごとに局所ヒストグラムを割り当てる。そのためには、抽出した for 文をデバイスで並列実行するために生成する gang の個数を得る必要がある。並列化を指示するディレクティブにおいて、生成する gang の個数が明示的に指定されていない場合、最適化が `gang` 節または `num_gangs` 節などをディレクティブに追加し、自動的に個数を指定する。

(c) のために、抽出した for 文の内側に、各 gang が更新する局所ヒストグラムのインデックスを取得するコードを挿入する。さらに、配列の要素をアトミック演算によりインクリメントしているコードを、局所ヒストグラムのビンを更新するコードに置換する。

6. 予備実験

5 章で説明した手順によりコードを最適化できることを確認する。OpenACC で記述された以下のアプリケーションのコードを書き換え、プログラムの実行時間を測定した。表 1 に測定環境を示す。なお、最適化は未実装のため、手動でコーディングパターンを抽出し、コードを書き換えた。

- (A-1) 結合重み分布計算 [11]
- (A-2) カラー画像の画素値ヒストグラム生成
- (A-3) ハフ変換による直線検出 [12]
- (A-4) 相互情報量計算

(A-1) の結合重み分布は、ネットワーク符号化 [13] における誤り訂正符号の評価などに用いられる。結合重み分布計算では、1 個のヒストグラムを生成し、母集団の 1 要素に対して 1 個のビンを更新する。なお、実験に用いたコードは、安藤ら [11] の計数方式を用いている。

(A-2) は、画素値をビンとするヒストグラムを生成し、画像処理において広く用いられる。カラー画像の画素値ヒストグラム生成では、1 個のヒストグラムを生成し、画素を構成する各色ごとに値を計数する。すなわち、母集団の

表 1 測定環境

項目	仕様
OS	Ubuntu 14.04.2 LTS
CPU	Intel Xeon E5-2680 x2 2.80 GHz
メインメモリ容量	512 GB
GPU	NVIDIA Tesla K40c
ビデオメモリ容量	12 GB
CUDA	6.5
コンパイラ	PGI コンパイラ 15.5[10]
コンパイラオプション	-acc -ta=tesla,cc35 -O3

1 要素につき複数個のビンを更新する。

(A-3) は画像認識などに用いられる。ハフ変換では 1 個のヒストグラムを生成し、母集団の 1 要素に対して複数個のビンを更新する。なお、実験に用いたコードは、OpenCV [14] における標準ハフ変換のコードを基に記述した。

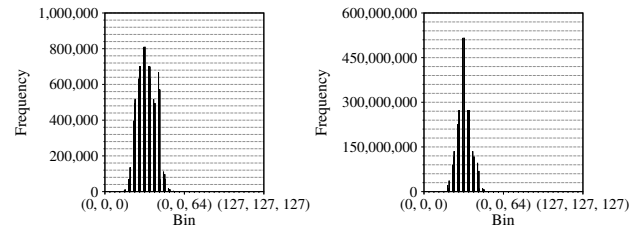
(A-4) の相互情報量は、画像位置合わせにおいて画像間の類似度の尺度として用いられる [15]。相互情報量を計算するためには、ビン数の異なる 2 種類のヒストグラムを合計 3 個生成する必要がある。ヒストグラム生成においては、いずれも母集団の 1 要素につき 1 個のビンを更新する。

これらのアプリケーションでは、一部のビンに対して計数値が集中する母集団を入力として用いることが多い。したがって、分散化手法により性能を向上できる可能性が高い。後述する実験結果では、局所ヒストグラムの個数を適切に選択することにより、ヒストグラム生成の実行時間を短縮できた。したがって、5 章で説明した手順により局所ヒストグラムを生成できた。

6.1 結合重み分布計算

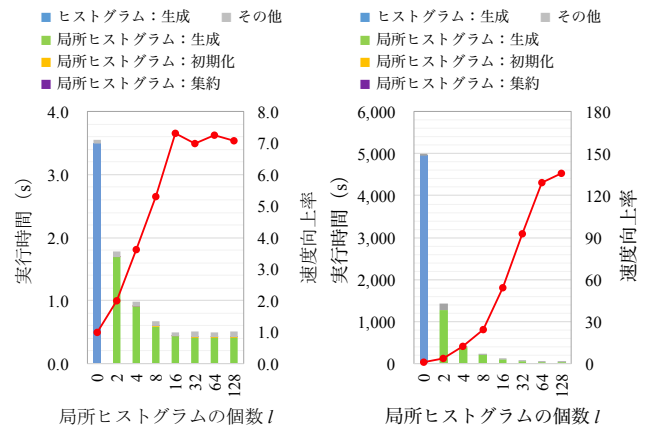
実験では、符号長 n が 127 であり、情報ビット（基底）の数 k が 22 である 2 元 (127, 22) BCH 符号 [16] から得た $k = 13$ および $k = 17$ の部分符号を用いた。符号語対の個数は 2^{2k} 個であるため、 $k = 13$ のときの符号語対の個数は 67,108,864 個であり、 $k = 17$ のときの符号語対の個数は 17,179,869,184 個である。1 つの符号語の大きさは 32 ビットである。ヒストグラム（結合重み分布）のビン数は $128 \times 128 \times 128$ 個であり、ビンの大きさは 64 ビットである。図 7 に入力符号から計算した結合重み分布を示す。これらのヒストグラムでは、一部のビンに対して計数値が集中している。したがって、ヒストグラム生成ではアトミック演算による書き込みの競合が性能低下の原因となる。

図 8 に、局所ヒストグラムの個数 l を変えたときの結合重み分布計算の実行時間を示す。実行時間は、デバイスでの実行時間に加え、ホストでの実行時間およびホスト・デバイス間のデータ転送時間などを含む。 $l = 0$ は最適化前（単純手法）に対応し、 $l \geq 2$ は最適化後（分散化手法）に対応する。実験結果から、 $k = 13$ では $l = 16$ のときに単純手法に対する速度向上率が最も高く、7.31 倍の速度向上



(a) $k = 13$ (b) $k = 17$

図 7 入力符号から計算した結合重み分布



(a) $k = 13$ (b) $k = 17$

図 8 結合重み分布計算の実行時間

を得た。一方、 $k = 17$ では $l = 128$ のときに速度向上率が最も高く、136 倍の速度向上を得た。

6.2 カラー画像の画素値ヒストグラム生成

実験では、Wikimedia Commons[17] から得た 4 つの 24 ビットカラー画像を用いた。画像の大きさは $1,920 \times 1,080$ 画素である。1 画素は RGB の 3 色からなり、各色は 256 階調（8 ビット）である。ヒストグラムのビン数は 256×3 個であり、R、G および B にビン番号の先頭から 256 個ずつを割り当てている。ビンの大きさは 32 ビットである。図 9 に入力画像から生成した画素値ヒストグラムを示す。これらのヒストグラムでは、一部のビンに対して計数値が集中している。したがって、ヒストグラム生成ではアトミック演算による書き込みの競合が性能低下の原因となる。

図 10 に各画像の画素値ヒストグラム生成の実行時間を示す。実行時間はデバイスでの実行時間のみであり、ホスト・デバイス間のデータ転送時間などは含まない。これは、画像のデータがすでにデバイスのメモリに存在し、生成した画素値ヒストグラムをホストで利用しない状況を想定しているためである。実験結果から、 $l = 12$ (animals)、 $l = 10$ (austin)、 $l = 6$ (nature) および $l = 12$ (sunset) のときに最も速度向上率が高く、それぞれ 2.45 倍、2.59 倍、2.17 倍および 3.29 倍の速度向上を得た。

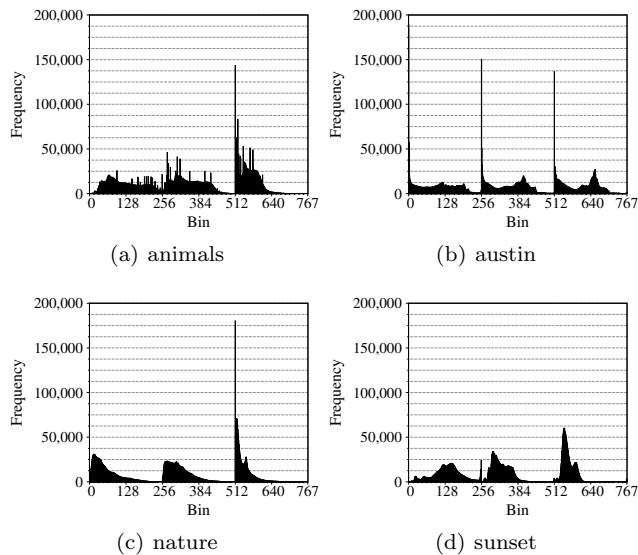


図 9 入力画像から得た画素値ヒストグラム

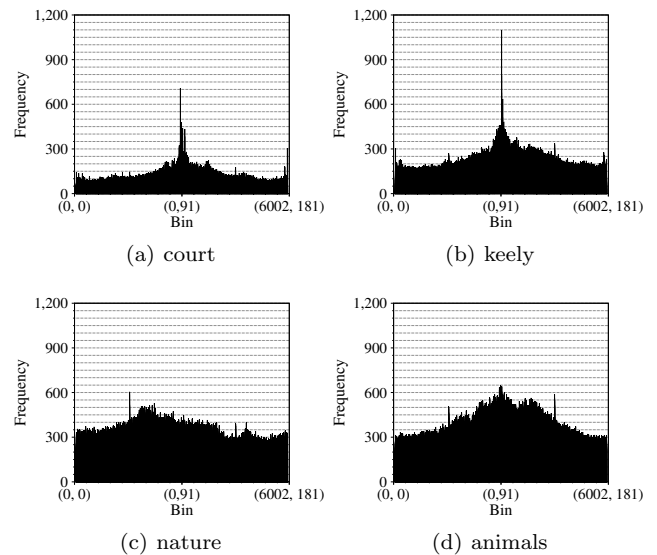


図 11 入力画像から得たハフ空間

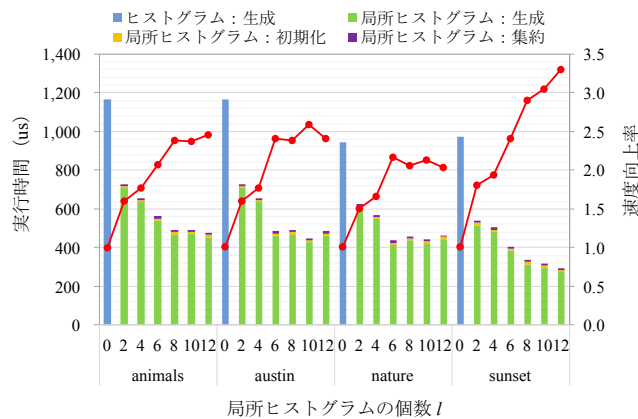


図 10 カラー画像の画素値ヒストグラム生成の実行時間

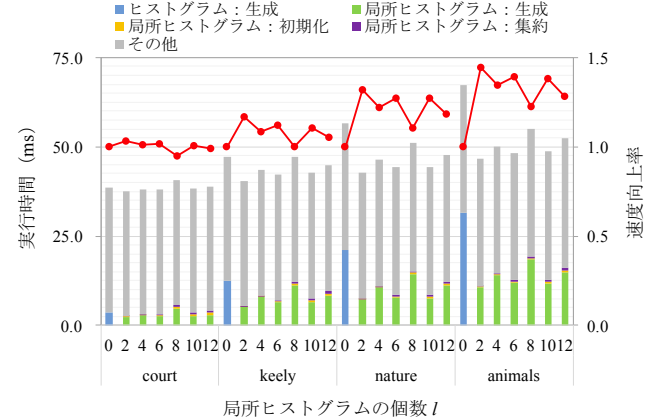


図 12 ハフ変換による直線検出の実行時間

6.3 ハフ変換による直線検出

実験では、Wikimedia Commons[17] から得た 4 つの画像を用いた。画像の大きさは $1,920 \times 1,080$ 画素であり、1 画素の大きさは 32 ビットである。各画像におけるエッジ画素の割合は、5.26% (court), 10.74% (keely), 16.15% (nature) および 21.35% (animals) である。ヒストグラム (ハフ空間) のビン数は $6,003 \times 182$ 個であり、ビンの大きさは 32 ビットである。ヒストグラム生成では、1 個のエッジ画素に対して 180 個のビンを更新する。図 11 に入力画像から生成したハフ空間を示す。このヒストグラムでは一部のビンに対して計数値が集中している。同時に、多数のビンに対して計数値が分散している。したがって、アトミック演算による書き込みの競合および広範囲のメモリ領域に対するランダムな書き込みが性能低下の原因となる。

図 12 にハフ変換による各画像中の直線検出の実行時間を示す。実行時間は、デバイスでの実行時間に加えて、ホストでの実行時間およびホスト・デバイス間のデータ転送時間などを含む。実験結果から、いずれの実行でも $l = 2$

のときに単純手法に対する速度向上率が最も高く、1.03 倍 (court), 1.17 倍 (keely), 1.32 倍 (nature) および 1.45 倍 (animals) の速度向上を得た。ランダムな書き込みによる性能低下のため、分散手法の効果は小さく、いずれも l の値が最も小さいときに速度向上率が最も高かった。

6.4 相互情報量計算

実験では、Vanderbilt Database [18] から得た脳の CT 画像、T1 強調 MR 画像および T2 強調 MR 画像を用いた。画像の大きさは $256 \times 256 \times 89$ 画素であり、各画素は 256 階調 (8 ビット) である。したがって、画素値ヒストグラムのビン数は 256 個であり、結合画素値ヒストグラムのビン数は 256×256 個である。ビンの大きさはいずれも 32 ビットである。図 13 に入力画像から生成した画素値ヒストグラムおよび結合画素値ヒストグラムを示す。これらのヒストグラムでは、一部のビンに対して計数値が集中している。したがって、ヒストグラム生成ではアトミック演算による書き込みの競合が性能低下の原因となる。

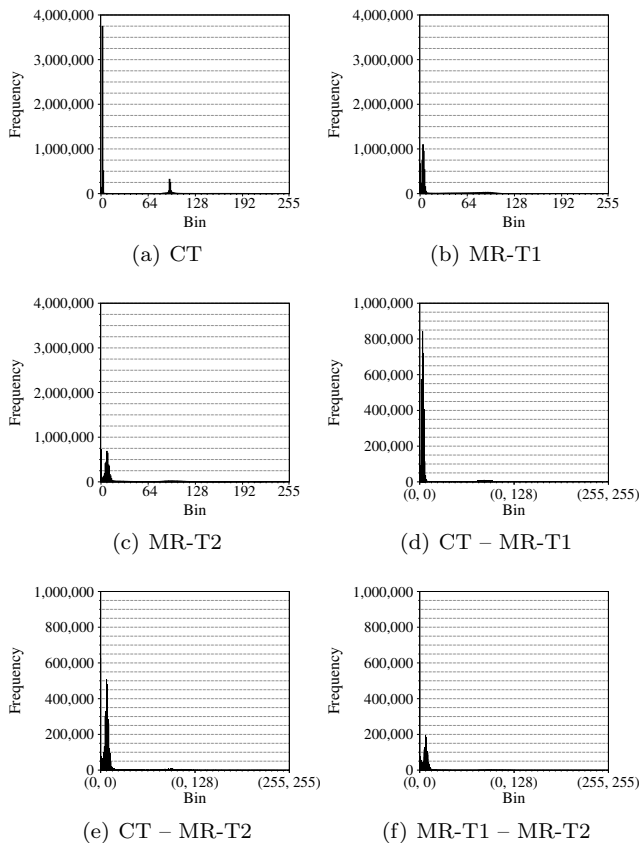


図 13 入力画像から得た画素値ヒストグラムおよび結合画素値ヒストグラム

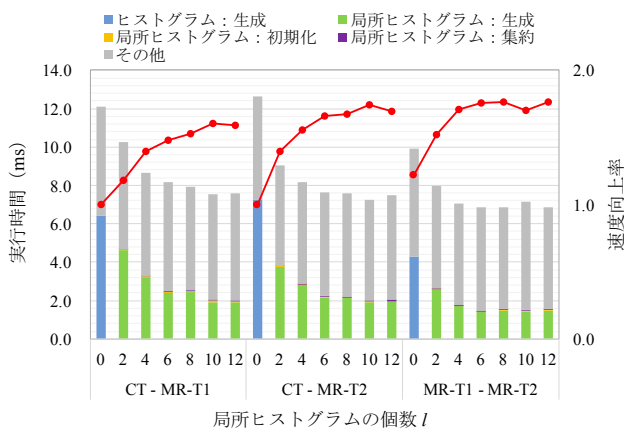


図 14 相互情報量計算の実行時間

図 14 に画像の各組の相互情報量計算の実行時間を示す。実行時間は、デバイスでの実行時間に加えて、ホストでの実行時間およびホスト・デバイス間のデータ転送時間などを含む。実験結果から、 $l = 10$ (CT 画像および T1 強調 MR 画像)、 $l = 10$ (CT 画像および T2 強調 MR 画像) および $l = 8$ (T1 強調 MR 画像および T2 強調 MR 画像) のときに速度向上率が最も高く、それぞれ 1.60 倍、1.75 倍および 1.76 倍の速度向上を得た。

7. まとめ

本稿では、OpenACC により記述されたヒストグラム生成のコードを GPU 向けに最適化するオプティマイザについて検討した。開発中のオプティマイザは、OpenACC コードからヒストグラム生成のコーディングパターンを抽出し、分散化手法を適用する。分散化手法の適用に伴う逐次コードの書き換えを自動化することにより、コードの性能可搬性を保ちながら、GPU において高い実行性能を達成できる。

予備実験では、オプティマイザによる最適化の手順にしたがって 4 種類のアプリケーションを手動で最適化し、性能を GPU 上で測定した。結果として、アトミック演算による書き込みの競合が性能低下の原因であるヒストグラム生成を高速化できた。

今後の課題として、局所ヒストグラムの適切な個数を決定する手法の確立が挙げられる。

謝辞 本研究の一部は、科研費 15K12008, 15H01687 および JST CREST 「進化的アプローチによる超並列複合システム向け開発環境の創出」の補助による。

参考文献

- [1] Intel Corporation: Intel Xeon Phi Product Family. <http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-detail.html>.
- [2] OpenACC-Standard.org: The OpenACC Application Programming Interface Version 2.0 (2013). http://www.openacc.org/sites/default/files/OpenACC.2.0a_1.pdf.
- [3] NVIDIA Corporation: CUDA C Programming Guide Version 6.5 (2014). http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf.
- [4] 中野瑛仁, 伊野文彦, 萩原兼一: アクセラレータのメモリ容量を超えるデータをパイプライン処理するためのディレクティブ, 情処研報, 2013-HPC-142, 8 pages (2013).
- [5] T. Hoshino, N. Maruyama, and S. Matsuoka: An OpenACC Extension for Data Layout Transformation, Workshop on accelerator programming using directives, In Proceedings of First Workshop on Accelerator Programming using Directives (WACCPD 2014), pp.12–18 (2014).
- [6] J. Beyer, D. Oehmke, and J. Sandoval: Transferring user-defined types in OpenACC, In Proceedings of CUG 2014 Conference, 15 pages (2014). https://cug.org/proceedings/cug2014_proceedings/includes/files/pap122.pdf
- [7] S. Lee and R. Eigenmann: OpenMPC: Extended OpenMP Programming and Tuning on GPUs, International Journal of Computational Science and Engineering, Vol.8, No.1, pp.4–20 (2012).
- [8] XcalableMP Specification Workgroup: XcalableMP Language Specification Version 1.2.1 (2014). <http://www.xcalablemp.org/download/spec/xmp-spec-1.2.1.pdf>.
- [9] 中尾昌広, 村井均, 下坂健則, 田淵晶大, 塙敏博, 児玉祐悦, 朴泰祐, 佐藤三久: XcalableACC:OpenACC を用いたアクセラレータクラスタのための PGAS 言語 XcalableMP

- の拡張, 情処研報, 2014-HPC-146, 11 pages (2014).
- [10] NVIDIA Corporation: PGI Compiler & Tools (2015). <http://www.pgroup.com>.
 - [11] 安藤翔平, 伊野文彦, 藤原融, 萩原兼一: 結合重み分布を高速に計算するための並列手法, 電子情報通信学会論文誌, Vol.J97-D, No.9, pp.1471–1480 (2014).
 - [12] P.V.C. Hough :Method and Means for Recognizing Complex Patterns, U.S.Patent No.3069654 (1962).
 - [13] R. Ahlswede, N. Cai, S.-Y.R. Li, and R.W.Yeung: Network information flow, IEEE Transactions of Information Theory, vol.46, no.4, pp.1204–1216 (2000).
 - [14] Itseez: OpenCV (2015). <http://opencv.org/>.
 - [15] J.P.W. Pluim, J.B.A. Maintz, and M.A. Viergever: Mutualinformation-based registration of medical images: A survey, IEEE Transaction on Medical Imaging, vol.22, Issue 8, pp.986–1004 (2003).
 - [16] S. Lin and D.J. Costello, Jr.: Error Control Coding Second Edition, Upper Saddle River, NJ: Prentice Hall (2004).
 - [17] Wikimedia Foundation: Wikimedia Commons (2015). <https://commons.wikimedia.org>.
 - [18] J.M. Fitzpatrick: The retrospective image registration evaluation project (2008). <http://www.insight-journal.org/rire/>.