Vol.2015-HPC-150 No.26
2015/8/5

OpenACCを用いたアウトオブコア・ステンシル計算に対するテンポラルブロッキングの適用

三木 脩弘[1]　伊野 文彦[1]　萩原 兼一[1]

概要：本稿では，ディレクティブによるテンポラルブロッキングの実現を目的として，OpenACC で記述されたアウトオブコア・ステンシル計算に対してテンポラルブロッキングを適用する．ここで，アウトオブコア・ステンシル計算とは，デバイスメモリ容量を超えるデータサイズを扱うステンシル計算のことである．開発した実装は，大きなデータを小さなチャンクに分割し，それらをパイプライン処理することにより，CPU 上のデータコピー時間を隠蔽する．また，ブロッキング段数やブロックサイズなどの実行パラメータを決定するために，実行時間を予測するモデルを示す．姫野ベンチマークを用いた評価実験では，約 15 GB のデータを扱う大きな問題サイズに対して，37.5 GFLOPS の実効性能を得た．この実効性能は，データ全体をデバイスメモリに格納できる小規模な問題サイズの結果と比較して，20%の低下にとどまっている．

キーワード：OpenACC，ステンシル計算，テンポラルブロッキング

Applying Temporal Blocking to Out-of-core Stencil Computation with OpenACC

Nobuhiro Miki[1]　Fumihiko Ino[1]　Kenichi Hagihara[1]

Abstract: In this paper, aiming at realizing directive-based temporal blocking, we present how stencil blocking can be applied to out-of-core stencil computation written with OpenACC. Out-of-core stencil computation here is stencil computation that deals with a data size larger than the capacity of device memory. Our implementation divides large data into small chunks, which are then processed in a pipeline manner to hide the copy overhead incurred on the CPU. We also show a model that predicts execution time to determine execution parameters such as a blocking factor and block size. In experiments, the effective performance of Himeno benchmark reached 37.5 GFLOPS for a large problem size that deals with 15 GB data. This achieved performance was 20% lower than the best performance obtained for a small problem that deals with small data that can be entirely stored in device memory.

Keywords: OpenACC, stencil computation, temporal blocking

1. はじめに

科学計算の加速を目的として GPU（Graphics Processing Unit）[1] や Xeon Phi[2] などのアクセラレータを搭載した高性能計算機が多く用いられている．例えば，偏微分方程式の数値解法である有限差分法やヤコビの反復法などのステンシル計算を高速化する試みが知られている．

一般に，アクセラレータ向けの計算コードは，固有のプログラミング言語を用いて記述する．例えば，NVIDIA 社の GPU に対しては，CUDA（Compute Unified Device Architecture）[1] が知られている．CUDA コードは，CPU 向けのホストコードおよび GPU 向けのデバイスコードからなる．したがって，アクセラレータ上で応用を加速するためには，元の逐次コードから性能ボトルネックを分離し

[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

たうえで，デバイスごとの最適化手法を適用する必要がある．結果，新しいアクセラレータへ計算コードを適応したいとき，あるいは元の計算機環境へ計算コードを戻したいときには大幅なコード修正が必要となる．そこで，これらの労力を長期的な観点から最小化するために，同一の記述で，複数の計算機が実行でき，かつ高い実効性能を達成できる性質（性能可搬性）が求められている．

高い性能可搬性を達成する手段として，OpenACC[3] などのディレクティブによるプログラミングが挙げられる．プログラマはディレクティブを追加・削除するだけで，アクセラレータ上に計算コードを容易に移植できる．つまり，アーキテクチャに特化した記述をディレクティブ内に留めることにより，その記述を計算の本質的な記述から分離できる．ただし，逐次プログラムをそのままアクセラレータ上で実行する OpenACC は，デバイスメモリがホストメモリと同程度の容量を持つことを仮定する．実際には，前者は後者よりも容量が 1 桁程度少ないために，問題規模を大きくしたときにプログラムの実行に失敗してしまう．

PACC（Pipelined ACCelerator）[4] は，デバイスメモリ容量を超えるデータサイズを扱うステンシル計算（アウトオブコア・ステンシル計算）を，単一アクセラレータ上で加速するためのディレクティブ仕様である．PACC は，処理対象のデータを複数のチャンクに分割し，それらに対する計算をパイプライン実行することにより，アウトオブコア計算を実現する．しかし，ホスト・デバイス間のデータ転送が全体性能を律速する．

そこで，本研究ではホスト・デバイス間のデータ転送回数および転送量の削減を目的として，テンポラルブロッキングを指示できるディレクティブを開発している．テンポラルブロッキングにより，ホスト・デバイス間のデータ転送 1 回あたりの演算回数が増加し，時間的局所性を向上できる．さらに，ホスト上のデータコピー，ホスト・デバイス間のデータ転送およびデバイス上の計算を並列処理できるよう，チャンクごとの計算をパイプライン実行する．このときの実効性能は，ブロックサイズおよびブロッキング段数などの実行パラメータに依存するため，これらをもとに性能を予測するためのモデルを構築する．

以降では，まず 2 章で関連研究を紹介し，3 章でステンシル計算の概要について述べる．次に，4 章で OpenACC によるアウトオブコア・ステンシル計算の実現について述べ，5 章で評価実験の結果を示す．最後に，6 章で本論文をまとめる．

2. 関連研究

河村ら [5] は，ドメイン特化言語 Physis[6] で記述されたアウトオブコア・ステンシル計算に対し，テンポラルブロッキングの自動変換を実現した．この変換は，Physis コードを入力として，CUDA コードを出力する．CUDA コードは NVIDIA 社の GPU 上でのみ動作するため，高い性能可搬性を追求することは難しい．また，元となる逐次コードをドメイン特化言語で書き直す労力は少なくない．

Jin ら [7] は，アウトオブコア・ステンシル計算に対する最適化手法を提案した．この既存手法は，テンポラルブロッキングに加えて，バッファコピーおよびメモリ節約を施す．バッファコピーの目的は，袖領域に起因する冗長な計算や通信の削減であり，処理中のチャンク間で計算結果を共有する．ただし，これらは CUDA コードに対する最適化であるため，高い性能可搬性を達成することは難しい．一方，PACC は OpenACC がサポートするすべてのアクセラレータ上で実行できる．

Endo ら [8] は，MPI と CUDA で記述されたプログラムを対象としたランタイムライブラリ Hybrid Hierarchical Runtime（HHRT）を設計した．HHRT では，独自の関数呼び出しにより計算コードを記述し，袖領域の大きさや計算コード領域を指定する．一方，PACC はディレクティブによりそれらを特定する．ディレクティブの利点は，計算の本質的な記述を環境依存の記述から分離できる点である．

XcalableACC（XACC）[9] は，Partitioned Global Address Space（PGAS）言語の一つである XcalableMP[10] と OpenACC を併用する．XACC では，各ノードへのデータ分散やノード間のデータ転送には XcalableMP を使い，ホスト・デバイス間のデータ転送には OpenACC を使う．いずれもディレクティブに基づくため，性能可搬性は高い．ただし，テンポラルブロッキングを実現するためには，ループ構造を含めた，計算の本質的な記述を改変する必要がある．

3. ステンシル計算

ステンシル計算とは，ある固定の計算パターン（ステンシル）にしたがい，配列内のすべての要素に対する更新を反復する計算である．例えば，偏微分方程式を解くための有限差分法やヤコビの反復法が該当し，これらは時間発展問題を解くときに有用である．

図 1 に，ステンシル計算の擬似コードを示す．この例では，大きさ $X \times Y$ の 2 次元配列に対して 5 点ステンシルを適用している．この場合，ある要素を更新するためには，自身に加えてその上下左右の要素値を必要とする．1 行目の最外ループが時間ステップの更新を担当している．

3.1 テンポラルブロッキング

テンポラルブロッキングは，ステンシル計算を対象とする最適化手法である．キャッシュ上に存在するデータを再利用することにより，高いキャッシュヒット率を得る．そのために，計算領域を小さなブロックに分割し，ブロック単位で k 時間ステップの計算を 1 度に進める．以降では，k のことをブロッキング段数と呼ぶ．

```
1  for (n=0; n<N; n++) { // 時間ステップ
2    // 要素の更新
3    for (x=1; x<X-1; x++)
4      for (y=1; y<Y-1; y++)
5        q[x][y] = p[x][y] + p[x][y+1] + p[x][y-1] +
             p[x+1][y] + p[x-1][y];
6    p = q;
7  }
```

図 1 5 点ステンシルの擬似コード

```
1  for (n=0; n<N; n+=k) { // ブロック外の時間ステップ
2    for (c=0; c<d; c++) // ブロックごとに更新
3      for (i=0;i<k;i++) // ブロック内の時間ステップ
4        for (x=1; x<(X-1)/d; x++)
5          for (y=1; y<Y-1; y++)
6            q[x][y] = p[x][y] + p[x][y+1] + p[x][y
                 -1] + p[x+1][y] + p[x-1][y];
7    p = q;
8  }
```

図 2 5 点ステンシルの擬似コード（テンポラルブロッキング後）

図 2 に，図 1 のコードに対してテンポラルブロッキングを適用した結果を示す．この例では，計算領域を d 個のブロックに分割し，ブロックごとに時間ステップを更新するために，元のループを 2 つに分割している．内側のループ（3 行目）はブロックごとの k 回の反復を担当し，外側のループ（1 行目）は k とびの反復を担当している．

このように，テンポラルブロッキングは，主記憶よりも高速に参照できるキャッシュに着目する．同様の関係は，ホストメモリとデバイスメモリ間に存在する．したがって，多くの GPU 実装はテンポラルブロッキングを用い，デバイスメモリに転送したブロックを可能な限り再利用する．つまり，k 時間ステップをデバイス上で 1 度に計算し，ホスト・デバイス間のデータ転送量およびデータ転送回数を $1/k$ に削減する．

ここで，ブロックごとに k 時間ステップの計算を独立に進めるためには，計算領域の周囲に袖領域が必要であることに注意されたい．例えば，ある要素から h 行以内の要素を参照するステンシルに対して k 段のテンポラルブロッキングを適用する場合，行方向に hk 個の要素からなる袖領域が必要である（**図 3**）．これにより，あるブロックの計算を他のブロックとは独立に処理できる．しかし，袖領域間の冗長な計算が実行効率を低下させてしまい，高い並列性は少ない計算量と両立しない．すなわち，並列性および計算量はトレードオフの関係にある．以降では，分割後のデータのことをチャンクと呼ぶ．ここで，チャンクはブロック（計算領域）だけでなく，袖領域を含むことに注意されたい．

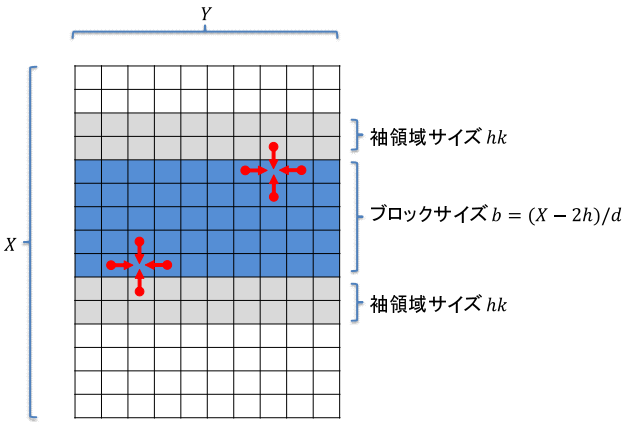


図 3 k 段のブロッキングに対して必要な袖領域

4. OpenACC によるアウトオブコア・ステンシル計算

OpenACC を用いてアウトオブコア・ステンシル計算を実現するためには，元の逐次コードに対して以下の実現が必要である．

データ分割 処理対象の大規模データを，デバイスメモリに格納できる大きさのチャンクに分割する．

さらに，テンポラルブロッキングを適用し，実行効率を高めるために，以下の実現が必要である．

ループ再構成 時間ステップに関するループを分割し，ブロック内およびブロック外のループとして構成する．

パイプライン実行 カーネル実行とホスト・デバイス間のデータ転送をオーバラップする．

以降では，OpenACC ディレクティブを用いてこれらを実現する方法を示す．**図 4** に，テンポラルブロッキングを適用した後の擬似コードを示す．

4.1 データ分割

OpenACC は，原則としてホストメモリおよびデバイスメモリの双方に同じ名前の変数を確保し，それらの間でデータを転送する．したがって，デバイスメモリ容量よりも大きな配列に対しては，ホストメモリ上にチャンクと同じ大きさのバッファを確保し，配列の一部をバッファにコピーすればよい（**図 5**）．図 5 の例では，ホストメモリ上の配列 p に対して，3 個のバッファ `buf_p[0]`〜`buf_p[2]` を用意している．これら小さなバッファのいずれかにチャンクをコピーしたうえで，データを転送しているため，配列 p の大きさが原因でデバイスメモリが枯渇することはない．なお，複数のバッファを用いる理由は，後述するパイプライン処理のためである．

開発した実装は，1 次元ブロック分割を前提としていて，最上位の次元に関して多次元配列を分割する（図 3）．例えば，大きさ $X \times Y$ の 2 次元配列に対して，x 方向のブロックサイズ b は $(X-2h)/d$，チャンクサイズは $b+2hk$ で表

```
1  buf_p[0]〜buf_p[num_stream]をホストメモリに確保;
2  #pragma acc create (buf_p [0:num_stream] [0:b+2*h*k], ...)
3
4  for (n=0; n<N; n+=k) { // ブロック外の時間ステップ
5    for (c=0; c<d; c++) { // ブロックごとに更新
6
7      ストリームsiを選択; // 0 <= si < num_stream
8      pからbuf_p[si]へチャンクをコピー;
9      #pragma acc update device (buf_p [si:1][0:b+2*h*k], ...) async (si)
10
11     for (i=0; i<k; i++) { // ブロック内の時間ステップ
12
13       #pragma acc kernels present (buf_p[si:1][0:b+2*h*k], ...) async(si)
14       {
15         offset = h*(i+1);
16         xsize  = b+2*h*(k-1-i);
17
18         #pragma acc loop independent
19         for (x=offset; x<offset+xsize; x++)
20           #pragma acc loop independent
21           for (y=1; y<Y-1; y++)
22             #pragma acc loop independent
23             for (z=1; z<Z-1; z++)
24               buf_q[si][x*Y*Z+y*Z+z] = buf_p[si][x*Y*Z+y*Z+z] + buf_p[si][(x+1)*Y*Z+y*Z+z] + (略);
25       }
26
27       buf_p[si] = buf_q[si];
28     }
29
30     #pragma acc update host (buf_p [si:1][0:b+2*h*k], ...) async (si)
31     buf_p[si]からpへチャンクをコピー;
32
33   }
34 }
```

図 4 テンポラルブロッキング後のステンシル計算コード

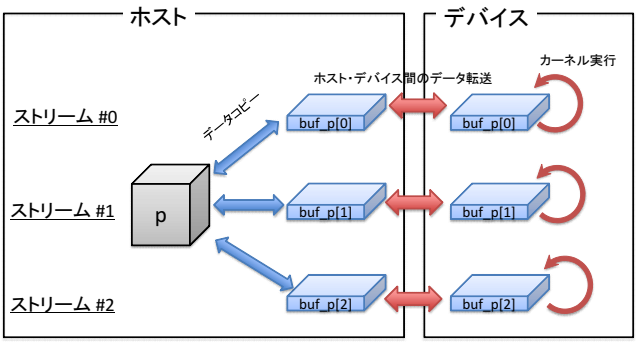


図 5 バッファを介したホスト・デバイス間のデータ転送

せる．ここで，d はブロック数である．

図 4 の例では，チャンクを格納するためのバッファをホストメモリに確保している（1 行目）．その後，OpenACC の create 節を用いて，同名のバッファをデバイスメモリに確保する．これらは，いずれも時間ステップに関するループの外側に存在することに注意されたい．その後，チャンクをホストからデバイスへ転送する直前に，元の配列 p からバッファ buf_p へ，該当チャンクをコピーし，update 節によりデバイスメモリへ転送する．GPU 上の更新が終われば，update 節によりホストメモリへバッファ（計算結果）を転送し（30 行目），元の配列 p へコピーする（31 行目）．

4.2 ループ再構成

テンポラルブロッキングを適用するために，図 2 の例と同様に，ループ構造を再構成する．k 時間ステップごとに反復するループに加え（4 行目），カーネル内でブロックごとに k ステップだけ更新を進める（11〜28 行目）．

なお，更新部分のループに対しては，kernels 構文を用いて GPU による計算を指示する（13 行目）．loop 構文では，independent 節により，直後のループがデータ依存に関して独立であることを明示する．また，present 節により，buf_p がデバイスメモリ上に転送済みであることを明示し，再転送を回避する．

4.3 パイプライン実行

ソフトウェアパイプラインを実現するために，一連の

データ転送やカーネル起動は非同期で処理する必要がある．そこで，OpenACC が提供する非同期キューを用い，現在のチャンクを更新しながら，次のチャンクを転送する．非同期キューは，自身の持つ一連の処理をインオーダで逐次実行する．ただし，異なる非同期キューは互いに独立に実行できるため，上記のデータ転送と GPU 上の計算をオーバラップできる．これにより GPU 上の計算時間と比べて長くなりがちなデータ転送時間の一部を隠蔽できる．

図 4 の例では，`num_stream` 個の非同期キューを用意し，それらに各チャンクをサイクリックに割り当てている．例えば，9 行目のデータ転送に対しては `async` 節を用いて非同期実行したうえで，`si` 番目の非同期キューを指定している．また，各々の非同期キューが独立にバッファを読み書きできるよう，非同期キュー `si` ごとに専有のバッファ `buf_p[si]` を用意している．

それぞれの非同期キューは，以下の手順を処理する．

(1) 元の配列から計算対象のチャンクを取り出し，ホストメモリ上のバッファへコピーする（8 行目）．
(2) ホストメモリ上のバッファを，デバイスメモリ上のバッファへ転送する（9 行目）．
(3) k 時間ステップだけチャンク内の要素を更新する（11〜28 行目）．
(4) デバイスメモリ上のバッファ（計算結果）を，ホストメモリ上のバッファへ転送する（30 行目）．
(5) ホストメモリ上のバッファを，元の配列にコピーする（31 行目）．

なお，現在の実装は CPU のスレッドを 1 個だけ用いている．したがって，ホスト上の手順（1）および（5）を同時に処理することはできない．

4.4 性能予測モデル

ブロックサイズ b およびブロッキング段数 k を決定するための性能予測モデルを構築する．モデル化の対象は，全体の実行時間 T を支配する手順（1），（3）および（5）である．これらの実行時間は，配列の要素数に依存しているため，要素数を配列の大きさ $X \times Y$ などのパラメータを用いて表す．

まず，手順（3）のカーネル実行時間 T_3 について考える．T_3 は，ブロック内の時間ステップに関するループ（11〜28 行目）が要する時間の和である．その i $(0 \leq i < k)$ 番目の反復において，参照領域が内包する要素数を $f(i)$ とする．i 番目の反復では，24 行目の代入文は $f(i)$ 個の要素を参照し，$f(i+1)$ 個の要素に結果を格納する．$f(i)$ は式 (1) のように表せる．

$$f(i) = (b + 2h(k-i))\,(Y - 2h) \qquad (1)$$

ここで，$b = (X - 2h)/d$ である（4.1 節）．さらに，11 行目，5 行目および 4 行目のループにおけるすべての反復についての和を考えると，T_3 は式 (2) で表せる．

$$T_3 = \sum_{i=0}^{k-1} f(i+1) \cdot d \cdot N/k \cdot F/P \qquad (2)$$

ここで，N および F はそれぞれ時間ステップ数および要素あたりに必要な浮動小数点演算数である．また，P は実効 FLOPS 値であり，あらかじめ小規模な問題サイズを用いて取得しておく．

次に，手順（1）および（5）のデータコピー実行時間 $T_1 + T_5$ について考える．チャンクの大きさは $f(0)$ であり，同じ大きさのチャンクがホスト・デバイス間を往復する場合，$T_1 + T_5$ は式 (3) で表せる．

$$T_1 + T_5 = 2f(0) \cdot d \cdot N/k \cdot D/B \qquad (3)$$

ここで，B はホストメモリのメモリバンド幅であり，D は要素を格納するデータ型の大きさである．

最後に，全体の実行時間 T は $T = \max(T_3, T_1 + T_5)$ で与えられる．

表 1 実験環境

項目	仕様
CPU	Intel Xeon E5-2680v2
主記憶容量	512 GB
GPU	NVIDIA Tesla K40c
ビデオメモリ容量	12 GB
OS	Ubuntu 15.3
コンパイラ	PGI C Compiler 15.5 [12]
コンパイルオプション	`-O3`

5. 評価実験

テンポラルブロッキングおよびパイプライン実行の効果を評価するために，姫野ベンチマーク [11] に対して OpenACC ディレクティブを追加し，GPU 上の実効性能を計測した．姫野ベンチマークは，非圧縮性流体解析において頻出するポアソン方程式をヤコビの反復法で解いている．配列サイズを $X \times Y \times Z$ としたとき，その実効性能 E は $E = 34N(X-2)(Y-2)(Z-2)/T$ で得られる．**表 1** に，実験環境を示す．

5.1 大規模データにおける評価

デバイスメモリ容量を超えるデータを扱う問題サイズとして XL（$X = Y = 512$ および $Z = 1024$）を用い，時間ステップ数 $N = 256$ としてホスト・デバイス間のデータ転送量および実効性能を計測した．

図 6 および**図 7** に，ホスト・デバイス間のデータ転送量および実効性能を示す．ブロッキング段数 k およびブロックサイズ b の増大に伴い，データ転送量は単調に減少し，実効性能は単調に向上する．しかし，これらの増大に伴い，チャンクが大きくなり，デバイスメモリ容量を超え

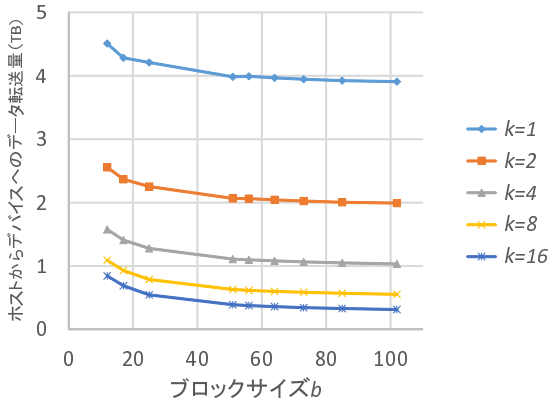


図 6　大規模データにおけるデータ転送量

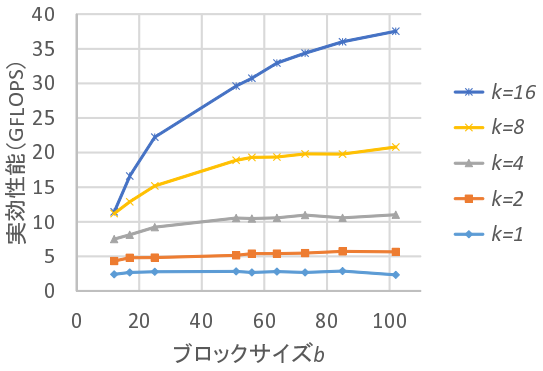


図 7　大規模データにおける実効性能

表 2　大規模データにおける実行時間の内訳（秒）

項目	非同期版	同期版
T_1：元の配列からバッファへのコピー時間	53.8	59.6
T_2：ホストからデバイスへの転送時間	30.0	29.9
T_3：カーネルの実行時間	26.3	26.3
T_4：デバイスからホストへの転送時間	4.3	4.3
T_5：バッファから元の配列へのコピー時間	6.7	6.0
T：全体の実行時間	61.6	126.4

ると実行に失敗してしまう．今回の実験環境では，$k = 16$ かつ $b = 102$ のときが実行に成功する最大のチャンクであり，実効性能が最大であった．つまり，デバイスメモリ容量の制限下で，できるだけ大きなブロッキング段数 k およびブロックサイズ b を選べばよい．この際，ブロックサイズ b よりもブロッキング段数 k の増大による実効性能の向上が大きいため，k を優先して大きくすればよい．

表 2 に，実効性能 E が最大となる実行パラメータ（$k = 16$ および $b = 102$）における実行時間の内訳を示す．実効性能 E は，非同期版で 37.5 GFLOPS，同期版で 18.3 GFLOPS であった．非同期版では，データコピー時間 $T_1 + T_5$ が残りの実行時間 $T_2 + T_3 + T_4$ とほぼ同様であり，ホスト上のデータコピー時間を隠蔽できている．結果，同期版と比較して，パイプライン実行により全体の実行時間をほぼ半減できている．なお，姫野ベンチマークでは T_1 は T_5 よりも大きい．この理由は，要素あたりの入力配列が出力配列よりも多いことに起因する．姫野ベンチマークは 14 個の配列を参照しているため，それぞれの配列ごとにチャンクをバッファへコピーし，ホストからデバイスへ転送する必要がある．

次に，アウトオブコア計算の実行効率を評価するために，デバイスメモリに乗り切るデータ（$X = Y = 128$ および $Z = 256$）を扱う OpenACC 実装を用意した．結果，ホスト・デバイス間のデータ転送時間を除外して，47.5 GFLOPS の実効性能を得た．したがって，アウトオブコア計算はホスト・デバイス間のデータ転送を必要とするにも関わらず，その性能低下を 20%に抑えられている．

5.2　実行パラメータの選定

4.4 節で示した性能予測モデルを評価するために，モデルを用いて適切なブロッキング段数 k ならびにブロックサイズ b を推定した．まず，実効性能 P およびホストメモリのメモリバンド幅 B を計測するために，データ分割やテンポラルブロッキングを施していない姫野ベンチマーク（OpenACC 版）を用意した．その後，表 1 の環境において，デバイスメモリに乗り切る小規模なデータ（$X = Y = 128$ および $Z = 256$）を用い，P および B を計測した．結果，$P = 47.5$ GFLOPS，$B = 5.2$ GB/s であった．これらの値を基に，問題サイズ L（$X = Y = 256$ および $Z = 512$）に対して適切な k および b を推定した．

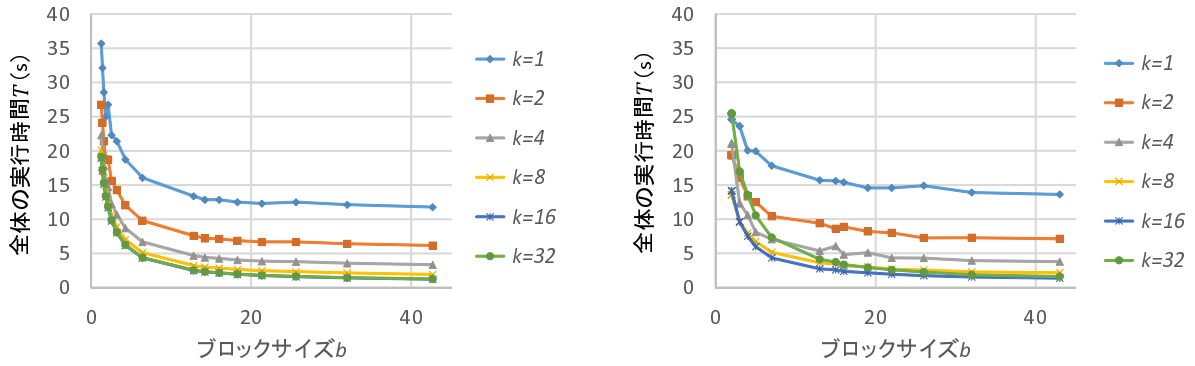
図 8 に，全体の実行時間 T の予測値および実測値を示す．性能予測モデルは，全体の実行時間 $T = \max(T_3, T_1 + T_5)$ を最小にする k および b を適切な実行パラメータとして返す．この例では，$k = b = 32$ の場合に T が最小であり，$T = 1.46$ と予測する．一方，実測による結果では，$k = 16$ および $b = 32$ の場合に T が最小であり，$T = 1.56$ であった．したがって，b の推定には成功したが，k の推定には失敗した．この理由は，実測による結果の内訳が $T_3 = 1.29$，$T_1 + T_5 = 1.07$ であることから，カーネル実行時間 T_3 の予測における誤差のためである．

図 9 に，カーネル実行時間 T_3 の予測値および実測値を示す．予測結果は，実測結果におけるグラフの概形をとらえているものの，実測値のおよそ半分程度の値を示した．この大きな誤差は，モデルがカーネル内の条件分岐を考慮していないことに起因する．

同様に，ホスト上のデータコピー時間 $T_1 + T_5$ の予測値および実測値を，**図 10** に示す．ブロックサイズ $b \geq 10$ では，高い精度で予測できている．ブロックサイズ b の変化による影響は少なく，ブロッキング段数 k の増加に伴いデータコピー時間 $T_1 + T_5$ は単調に減少している．

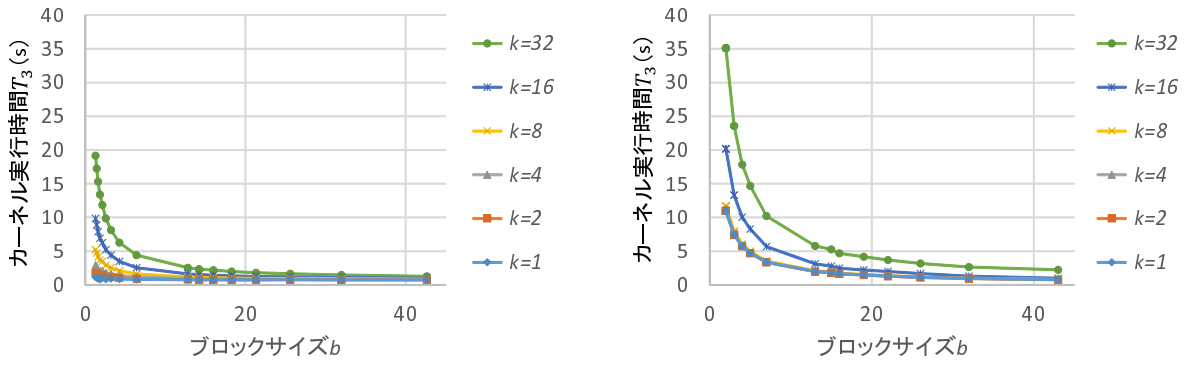
6.　まとめ

本論文では，OpenACC を用いてアウトオブコア・ステ



(a) 予測　　(b) 実測

図 8　全体の実行時間 T の比較



(a) 予測　　(b) 実測

図 9　カーネル実行時間 T_3 の比較

ンシル計算にテンポラルブロッキングを適用する手法を示した．開発した OpenACC 実装は，大きなデータを小さなチャンクに分割したうえで，非同期キューを用いてそれらをパイプライン処理する．また，ブロッキング段数やブロックサイズを決定するための性能予測モデルを示した．

姫野ベンチマークを用いた実験の結果，ホスト・デバイス間のデータ転送を伴うアウトオブコア・ステンシル計算を，データ転送を除外したときの実効性能と比べて 20%の性能低下で実現できた．また，性能予測モデルを用いることにより，小規模な問題サイズにおける測定結果を基に，アウトオブコア・ステンシル計算の実行性能を最大化できるブロックサイズを推定できたが，ブロッキング段数の推定には失敗した．

今後の課題としては，手動で適用したコード変換を自動化することが挙げられる．

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] NVIDIA Corporation: CUDA C Programming Guide Version 6.5 (2014). http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf.

[2] Intel Corporation: Intel Xeon Phi Product Family. http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-detail.html.

[3] OpenACC-Standard.org: The OpenACC Application Programming Interface, Version 2.0 (2013).

[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 (2013). 8 pages.

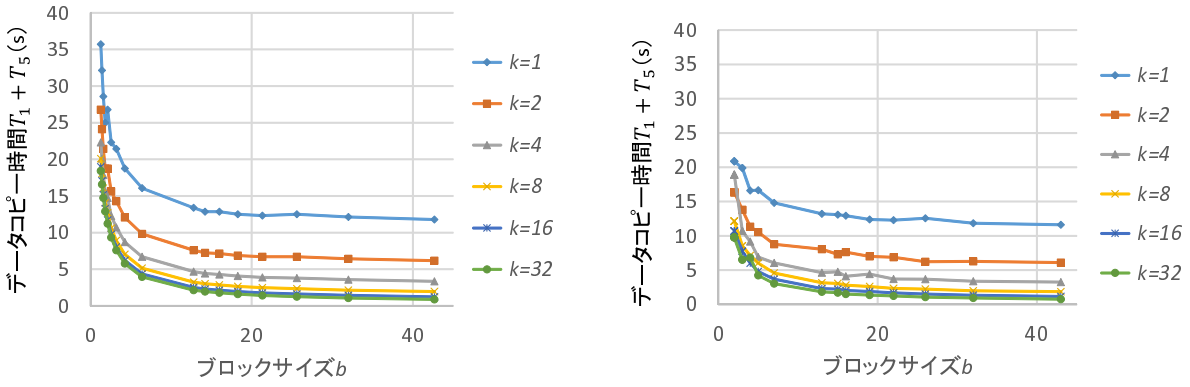
[5] 河村知輝，丸山直也，松岡　聡：自動テンポラルブロッキングによる大規模ステンシル計算の実現，情処研報，2014-HPC-143 (2014). 6 pages.

[6] Maruyama, N., Nomura, T., Sato, K. and Matsuoka, S.: Physis: An Implicitly Parallel Programming Model for Stencil Computations on Large-Scale GPU-Accelerated Supercomputers, *Proc. Int'l Conf. High Performance Computing, Networking, Storage and Analysis (SC'11)* (2011). 12 pages.

[7] Jin, G., Endo, T. and Matsuoka, S.: A Parallel Optimization Method for Stencil Computation on the Domain that is Bigger than Memory Capacity of GPUs, *Proc. 15th IEEE Int. Conf. Cluster Computing (CLUSTER'13)* (2013). 8 pages.

[8] Endo, T. and Jin, G.: Software Technologies Coping with Memory Hierarchy of GPGPU Clusters for Stencil Computations, *Proc. 16th IEEE Int. Conf. Cluster Computing (CLUSTER'14)*, pp. 132–139 (2014).

[9] Nakao, M., Murai, H., Shimosaka, T., Tabuchi, A.,



(a) 予測　　(b) 実測

図 10　データコピー時間 $T_1 + T_5$ の比較

Hanawa, T., Kodama, Y., Boku, T. and Sato, M.: XcalableACC: Extension of XcalableMP PGAS Language using OpenACC for Accelerator Clusters, *Proc. 1st Workshop Accelerator Programming using Directives (WACCPD'13)*, pp. 27–36 (2014).

[10] Nakao, M., Lee, J., Boku, T. and Sato, M.: Productivity and Performance of Global-View Programming with XcalableMP PGAS Language, *Proc. 12th IEEE/ACM Int'l Symp. Cluster, Cloud and Grid Computing (CCGRID'12)*, pp. 402–409 (2012).

[11] Himeno, R.: Himeno benchmark (2014). `http://accc.riken.jp/2444.htm`.

[12] NVIDIA Corporation: PGI Compiler (2015). `http://www.pgroup.com/`.