Vol.2016-HPC-155 No.13
2016/8/8

アウトオブコア・ステンシル計算に対する自動テンポラルブロッキングのためのアクセラレータ向けディレクティブ

三木 脩弘[1]　伊野 文彦[1]　萩原 兼一[1]

概要：本稿では，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を自動的に適用できる OpenACC ディレクティブの拡張およびそのトランスレータを提案する．提案するトランスレータは，OpenACC に類似する C プログラムおよびいくつかの実行パラメタを入力として，データ分割，パイプライン実行およびテンポラルブロッキングにより最適化された OpenACC プログラムを出力する．評価実験では，提案するトランスレータをヤコビ法，姫野ベンチマークおよび CIP（Constrained Interpolation Profile）法に適用し，これらの実効性能および性能可搬性の向上を分析する．

キーワード：OpenACC，アウトオブコア・ステンシル計算，テンポラルブロッキング

Accelerator Directives for Automated Temporal Blocking of Out-of-Core Stencil Computation

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

Abstract: In this paper, we propose an extension of OpenACC directives and its translator capable of automatically applying temporal blocking to out-of-core stencil computation. The proposed translator takes an OpenACC-like C program and some execution parameters as inputs to generate an OpenACC program optimized with data decomposition, pipelined execution, and temporal blocking. In experiments, where the effective performance and performance portability are analyzed, we apply our translator to the Jacobi method, Himeno benchmark, and constrained interpolation profile (CIP) scheme.

Keywords: OpenACC, out-of-core stencil computation, temporal blocking

1. はじめに

ステンシル計算とは，反復計算の一種であり，流体力学や画像処理などにおける時間発展問題の解法として頻出する．この計算では，データを格子点上に配置し，ある固定的な参照パターン（ステンシル）にしたがいデータを更新する．例えば，5 点ステンシルであれば，自身に加えてその上下左右の隣接点から次の時間ステップの値を計算する．各格子点は独立に更新できるため，ステンシル計算の並列性は高い．また，一般にステンシルは多数の近傍点を内包しているため，ステンシル計算の性能はメモリ帯域幅が律速する．したがって，数万個もの大量のスレッドを並列処理でき，高いメモリ帯域幅を実現する GPU（Graphics Processing Unit）[1] などのアクセラレータによる加速が試みられている．

一般に，アクセラレータ向けコードの実装には，計算性能を最大化できる固有のプログラミング言語を用いる．例えば，NVIDIA 社製 GPU 向けのプログラムを実装するためには，統合開発環境 CUDA（Compute Unified Device Architecture）[1] を用いる．CUDA においてデバイスの特

[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

性を活かすためには，計算コードやデータ構造の改変が不可欠である．しかし，これらの改変に起因する移植コストは大きい．そこで，同一の計算コードを用いつつ，様々な計算機環境において高い計算性能を実現できる性質（性能可搬性）を向上させることが重要となる．

ディレクティブによるプログラミングは，優れた性能可搬性を実現する手段の 1 つである．例えば，アクセラレータ向けのディレクティブとして，OpenACC が挙げられる．逐次コード中の並列化可能なコード片に OpenACC ディレクティブを挿入することで，そのコード片をアクセラレータへオフロードできる．ただし，OpenACC は，計算に用いるすべてのデータを，デバイスメモリに格納できると仮定している．したがって，計算に用いるデータの総量は，デバイスメモリ容量に制限される．

この制約を取り除くために，我々は PACC（Pipelined ACCerelator）[2] を開発した．PACC トランスレータは，PACC ディレクティブを挿入した逐次プログラムを入力とし，データ分割およびパイプライン実行を実現するよう書き換えた OpenACC コードを出力する．PACC は，デバイスメモリ容量を超える問題サイズに対して有用である．しかし，時間発展問題に対しては，ホスト・デバイス間のデータ転送が全体性能を律速してしまう．

そこで，本論文では，アウトオブコア・ステンシル計算における CPU・GPU 間のデータ転送量削減を目的として，データ分割およびパイプライン実行に加えて，テンポラルブロッキングを実現する PACC ディレクティブの拡張を提案する．さらに，提案ディレクティブを付加した逐次コードから OpenACC コードを生成するためのトランスレータの設計・実装について述べる．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大規模データに対するステンシル計算を指す．また，テンポラルブロッキングは，キャッシュ上のデータを再利用することにより，キャッシュヒット率の向上を図る最適化手法である．ホスト・デバイス間のデータ転送が全体性能を律速するステンシル計算では，テンポラルブロッキングによる転送量削減が性能向上につながる．ディレクティブ仕様およびトランスレータの特長を以下にまとめる．

- テンポラルブロッキングによるデータ転送量の削減
- データ分割による問題サイズの制約緩和
- パイプライン処理による CPU・GPU 間のデータ転送時間の隠蔽

以降では，まず 2 章で関連研究を紹介する．次に，3 章で提案するディレクティブの設計，4 章で OpenACC によるアウトオブコア・ステンシル計算の実現，5 章でトランスレータの実装について述べる．その後，6 章で評価実験の結果を示し，7 章で本論文をまとめる．

2. 関連研究

丸山らは，Domain Specific Language（DSL）によりステンシル計算を記述するヘテロジニアススパコン向けフレームワーク Physis [3] を開発した．Physis の入力は DSL で記述されたステンシル計算コードであり，その出力は MPI+CUDA である．また，テンポラルブロッキングを自動で適用する機構がある [4]．Physis では，既存のアプリケーションを DSL で書き換える必要があるが，計算性能と移植性の両立を目指す点は，我々の研究と共通する．

遠藤らは，MPI+CUDA で記述されたプログラムに対し，メモリスワップするランタイムライブラリ Hybrid Hierarchical Runtime（HHRT）[5] を開発した．Physis と同様にテンポラルブロッキングを適用できる [6]．HHRT は，並列化済のコードを前提としているため，あらかじめ並列プログラムを実装している場合には容易に利用できる．

XcalableACC[7] は，アクセラレータを搭載した大規模並列計算機向けに XcalableMP[8] と OpenACC を統合した並列プログラミング言語である．XcalableACC は，逐次プログラミング言語をディレクティブにより拡張した言語であるため，既存の逐次プログラムを再利用でき，移植性が高い．また，XcalableACC は，XcalableMP の特長を継承しているため，マルチノード環境で有用である．ただし，テンポラルブロッキングを適用するためには，コードの書き換えが不可欠である．

主記憶容量を超える大規模ステンシル計算を効率的に実行するために，緑川ら [9] はテンポラルブロッキングを主記憶・SSD 間に適用した．この手法は，各ノード上で実行可能な問題サイズの向上に役立つ．

3. ディレクティブの設計

テンポラルブロッキングを自動的に適用するためには，ステンシルの形状，配列名および配列サイズに関する情報をトランスレータに指示する必要がある．そこで，それらの情報を過不足なくトランスレータに指示できるようなディレクティブを設計する．

3.1 設計するディレクティブの位置づけ

図 1 に，並列計算を実現するディレクティブをまとめる．OpenMP および OpenACC と同様に，ユーザは，逐次コードに対してディレクティブを挿入する．ディレクティブを挿入したコードは，トランスレータによって OpenACC コードへ変換され，その後 OpenACC コンパイラによって実行ファイルへコンパイルされる．

3.2 プログラムの制約

ディレクティブ挿入対象のコードは，逐次ステンシル計

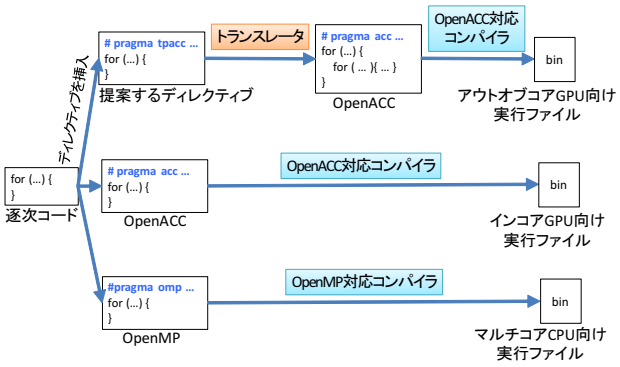


図 1 ディレクティブによるプログラミングの流れ
Fig. 1 Directive-based programming flow.

算コードである．以下の制約を満たすと仮定する．ただし，変数名，変数の数および条件分岐に関する制約はない．

- データ分割に起因する制約：トランスレータがデータを分割するため，ユーザが逐次コードにおいてデータを分割してはならない．さらに，全データの総量が主記憶容量以下である必要がある．
- テンポラルブロッキングに起因する制約：総ステップ数を静的に決める必要がある．さらに，デバイス上で数ステップまとめて計算を進めるため，計算中のデータを参照できない．したがって，計算中の誤差などを条件にコード実行を打ち切ることはできない．

3.3 ディレクティブ仕様

提案するディレクティブは，`init` 構文，`pipeline` 構文および `loop` 構文からなる．**図 2** に，ディレクティブを追加したステンシル計算コードの例を示す．OpenACC と同様に，ユーザは，主に `for` 文に対してディレクティブを挿入する．また，ディレクティブ内で，構文に応じた付加情報を記述する．

- `init` 構文：アウトオブコア・ステンシル計算では，ホストメモリおよびデバイスメモリ上に，バッファ領域を必要とする．そこで，`init` 構文により，バッファ領域を確保するコードの生成位置を指示する（1 行目）．したがって，`init` 構文は，`pipeline` 構文および `loop` 構文より前に実行される行に記述する．
- `pipeline` 構文：時間発展を担当する `for` 文（4 行目）に対して，`pipeline` 構文を挿入する（3 行目）．`pipeline` 構文は，`targetinout` 節，`targetin` 節，`size` 節，`halo` 節および `async` 節からなる．`targetinout` 節では，ステンシル計算により値を読み書きする配列を列挙する．読み込みだけの配列は，`targetin` 節に列挙する．`size` 節では，配列の開始インデックスとその大きさを次元ごとに指示する．例えば，サイズ $X \times Y$ の配列において全格子点が計算対象であれば，`size([0:X][0:Y])` とする．`halo` 節では，ステンシルの参照幅を次元ごとに指示する．さらに，後述するパイプライン実行を指示する場合には，`async` 節を付加する．
- `loop` 構文：ステンシル計算中の配列参照において，各次元の添字変数を担当する `for` 文に対して，それぞれ `loop` 構文を挿入する（5，7，11，13 行目）．

```
1  #pragma tpacc init
2
3  #pragma tpacc pipeline targetinout(work,a) size
       ([0:X][0:Y]) halo([1:1][1:1]) async
4  for(n=0;n<nn;n++){
5    #pragma tpacc loop dim(2)
6    for(x=1;x<X-1;x++)
7      #pragma tpacc loop dim(1)
8      for(y=1;y<Y-1;y++)
9        work[x][y] = (a[x-1][y] + a[x+1][y] + a[x
             ][y-1] + a[x][y+1])/4.0;
10
11   #pragma tpacc loop dim(2)
12   for(x=1;x<X-1;x++)
13     #pragma tpacc loop dim(1)
14     for(y=1;y<Y-1;y++)
15       a[x][y] = work[x][y];
16 }
```

図 2 提案するディレクティブを用いたプログラム例
Fig. 2 An example program with proposed directives.

4. OpenACC によるアウトオブコア・ステンシル計算の実現

アウトオブコア・ステンシル計算を実現するためには，データ分割が不可欠である．さらに，高い実効性能を実現するためには，テンポラルブロッキングおよびパイプライン実行を実現する必要がある．

- データ分割：1 次元ブロック分割により，計算領域を複数のチャンクに分割する（**図 3**）．例えば，大きさ $X \times Y \times Z$ の計算領域を，x 方向について d 個のチャンクに分割することを考える．チャンクの x 方向の大きさは $b = (X - 2h)/d$ となり，全体で $b \times Y \times Z$ となる．ここで，h はステンシルの x 方向の参照幅を表す．
- テンポラルブロッキング：1 度のデータ転送に対して，チャンクをデバイス内で k ステップ時間発展させる．以降，k をブロッキング段数と呼ぶ．k ステップの時間発展において，チャンクは計算領域外の袖領域を参照する．袖領域は，チャンクの x 方向に隣接し，その大きさは正負方向にそれぞれ $hk \times Y \times Z$ である．つまり，チャンクを独立に計算するためには，チャンクごとに大きさ $2hk \times Y \times Z$ の袖領域が必要である．そこで，1 つのチャンクの計算に対し，大きさ $(b + 2hk) \times Y \times Z$ のデータをアクセラレータへ転送する．
- パイプライン実行：袖領域が付加されたチャンクは，他のチャンクから独立して計算を進行できる．そこで，

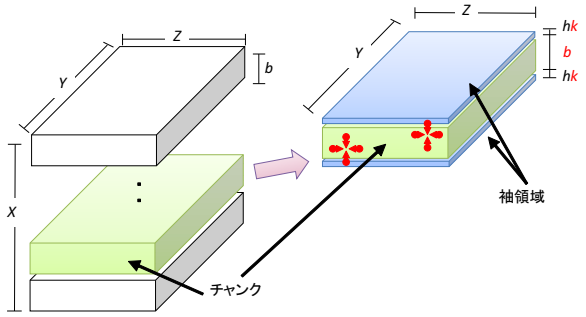


図 3 1 次元ブロック分割
Fig. 3 1-D block decomposition scheme.

ホストにおける配列・バッファ間のデータコピー，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それぞれパイプラインステージとみなし，チャンクごとの計算をパイプライン実行する．

図 2 に対して，これらを実現するよう書き換えた OpenACC コードを，**図 4** に示す．まず，データ分割を実現するために，チャンクおよび袖領域を格納するためのバッファ `buf_a` を，ホストメモリおよびデバイスメモリに確保する（1，2 行目）．さらに，チャンクを順次計算するため `for` 文を追加し（5 行目），元の配列 `a` への参照をすべてバッファ `buf_a` への参照に置換する（22 行目）．ホストメモリ上の配列，バッファおよびデバイスメモリ上のバッファ間におけるデータ転送を記述する（8, 9, 28, 29 行目）．

次に，テンポラルブロッキングを実現するために，時間発展を担当する `for` 文を 2 つに分割する（4，11 行目）．チャンク内の `for` 文（11 行目）ではアクセラレータによりチャンクの計算を k ステップ独立に進め，チャンク外の `for` 文（4 行目）では k ステップごとにホスト・デバイス間でチャンクのスワップイン・アウトを繰り返す．

最後に，パイプライン実行を実現するために，`num_stream` 個の非同期ストリームを，チャンクにサイクリックに割り当てる（7 行目）．さらに，OpenACC ディレクティブの `kernels` 構文および `update` 構文に対し，`async` 節を付加することで，カーネル実行をホスト・デバイス間のデータ転送とオーバラップさせる．

4.1 テンポラルブロッキングにおけるトレードオフ

テンポラルブロッキングにおいて，計算コストとデータ転送コストは，トレードオフの関係にある．

まず，チャンクに対する k ステップの時間発展（11〜26 行目）における計算コストを考える．i（$0 \le i < k$）ステップ目の時間発展において，x および y 方向を担当する `for` 文の反復回数はそれぞれ $b + 2h(k-1-i)$ および $Y-2$ である（19，21 行目）．つまり，i ステップ目の時間発展では，$b(Y-2) + 2h(k-1-i)(Y-2)$ 個の格子点を更新する．ここで，$b(Y-2)$ 回の更新処理は，原理的に

```
1  buf_a[0]～buf_a[num_stream-1]およびbuf_work[0]～
       buf_work[num_stream-1]をホストメモリに確保;
2  #pragma acc create (buf_a [0:num_stream][0:(b+2*h*
       k)*Y], ...)
3
4  for (n=0; n<N; n+=k) { // チャンク外の時間発展
5    for (c=0; c<d; c++) { // チャンクごとに更新
6
7      ストリームsiを選択; // 0 <= si < num_stream
8      aからbuf_a[si]へチャンクをコピー;
9      #pragma acc update device (buf_a [si:1][0:(b
           +2*h*k)*Y], ...) async (si)
10
11     for (i=0; i<k; i++) { // チャンク内の時間発展
12
13       #pragma acc kernels present (buf_a[si:1][0:(
             b+2*h*k)*Y], ...) 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             buf_work[si][x*Y+y] = buf_a[si][(x-1)*
                   Y+y] + buf_a[si][(x+1)*Y+y] + (略
                   );
23       }
24
25       buf_a[si] = buf_work[si];
26     }
27
28     #pragma acc update host (buf_a [si:1][0:b+2*h*
           k], ...) async (si)
29     buf_a[si]からaへチャンクをコピー;
30   }
31 }
```

図 4 アウトオブコア・ステンシル計算の OpenACC 実装
Fig. 4 An OpenACC-based implementation of out-of-core stencil computation.

必要な計算である．一方，$2h(k-1-i)(Y-2)$ 回の更新処理は，逐次コードでは現れなかった冗長な計算である．まとめると，チャンクに対する k ステップ時間発展では，$\sum_{i=0}^{k-1} 2h(k-1-i)(Y-2) = hk(k-1)(Y-2)$ 回の冗長な更新処理が実行される．また，$k \ge 1$ において，この冗長な計算の回数は，単調に増加する．

次に，データ転送コストについて考える．あるチャンクサイズ b において，ブロッキング段数 k を大きくすると，より長期間に渡ってチャンクをデバイスメモリに留めることができ，プログラム全体のデータ転送量を削減できる．まとめると，k の増加に伴い，データ転送コストは小さくなるが，計算コストは大きくなってしまう．

例として，5 点ステンシルに対するテンポラルブロッキングを考える．k を 1 から 3 に増加させると，チャンクが

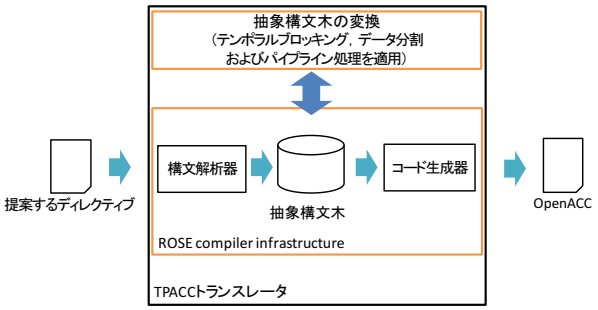


図 5 トランスレータの概要
Fig. 5 An overview of the proposed translator.

デバイスメモリに存在する時間が 1 ステップから 3 ステップに延び，時間局所性が増す．ただし，k の増加に伴い，冗長な計算が増加してしまう．

つまり，ブロッキング段数 k およびチャンクサイズ b は実効性能を決定する重要なパラメタである．そこで，実行時にこれらを環境変数により指示し，パラメタスイープにより性能を最大化する．

5. トランスレータの実装

トランスレータは，2 つのモジュールからなる（**図 5**）．1 つ目のモジュールは，ソースコードと抽象構文木（AST：Abstract Syntax Tree）を相互に変換する構文解析器およびコード生成器である．2 つ目のモジュールは，テンポラルブロッキング，データ分割およびパイプライン処理を AST の変換によって実現する．

5.1 AST の変換

AST は，対象コードの構成要素を木構造で表現する．まず，トランスレータは，AST 全体を探索し，ディレクティブの挿入行に基づいて，ステンシル計算を構成する `for` 文のノード（図 2 における 4，6，8，12，14 行目）を抽出する．同時に，ディレクティブに記述された各節を構文解析し，ステンシルの参照幅，配列名および配列サイズなどディレクティブの情報を取得する．

次に，`init` 構文に対応するノードに対し，ホストメモリおよびデバイスメモリ上にバッファ領域を確保する部分木を追加する（図 4 における 1，2 行目）．ここで，バッファの大きさをブロッキング段数 k およびチャンクサイズ b から算出するために，これら 2 つの実行パラメタを環境変数から取得する部分木を，兄弟として追加する．

続いて，`pipeline` 構文に関するノードを変換する．全格子点に対する 1 ステップの更新を担当する部分木（図 2 における 5〜9 行目）を，1 つのチャンクに対する更新に変換する（図 4 における 13〜23 行目）．この変換は，インデックス計算の書き換えによって実現できる．さらに，変換した部分木の親に，チャンク内の時間発展を担当する `for` 文を追加する（図 4 における 11 行目）．まとめると，ここまでの変換によって，チャンク内の時間発展に関する部分木を生成する．

さらに，ホスト・デバイス間のデータ転送および主記憶内のデータコピーを担当する部分木を，兄弟として追加する（図 4 における 8，9，28，29 行目）．その後，チャンクの選択およびチャンク外の時間発展を担当する `for` ノードを，親に追加する（図 4 における 4，5 行目）．

最後に，データ転送やカーネル実行を指示する OpenACC ディレクティブを適切なノードに挿入する（図 4 における 9，13，18，20，28 行目）．

表 2 実験環境
Table 2 Experimental environment.

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`

6. 評価実験

ディレクティブとそのトランスレータを評価するために，**表 1** に示す 3 つのステンシル計算コードをトランスレータにより自動変換し，実効性能およびディレクティブの挿入行数を評価した．ヤコビ法は，連立一次方程式を反復法で解く．姫野ベンチマーク（姫野 BMT）[10] は，非圧縮流体解析において頻出するポアソン方程式を解く．CIP 法 [11] は，双曲型偏微分方程式を差分法により解く．

以降では，演算強度 G を，$G = f/(l + s)$ と定義する．ここで，l は 1 つの格子点における読み込みバイト数，s は書き込みバイト数，f は浮動小数点演算回数である．**表 2** に，実験環境を示す．

6.1 実効性能

各ステンシル計算コードに対してディレクティブを挿入し，その後トランスレータにより OpenACC コードへ自動変換した．変換で得られた出力コードの実効性能を**図 6** に示す．

ヤコビ法では，チャンクサイズ $b = 8000$，ブロッキング段数 $k = 32$ のとき，実効性能は最大になり，28.5 GFLOPS であった．テンポラルブロッキングにおけるトレードオフの関係から，単純に k を大きくしても，性能は向上しないことが分かる．また，チャンクサイズ b は，ブロッキング段数 k と比較すると，実効性能に与える影響が小さい．

$k = 32$ では，カーネル実行時間とデータ転送時間が一致し，パイプライン処理におけるデータハザードが発生しないため，最も効率よく実行できる．一方，$k < 32$ ではチャ

表 1 実験に用いたテストコード
Table 1 Experimental test codes.

テストコード	次元	配列サイズ	配列数	総データ量 (GB)	総ステップ N	Float 演算 f (FLOP)	読み込み l (B)	書き込み s (B)	演算強度 G (FLOP/B)
ヤコビ法	2	48000×48000	2	18.4	2048	4	16	4	0.20
姫野 BMT	3	$512 \times 512 \times 1024$	14	15.0	256	34	128	4	0.26
CIP 法	2	22000×22000	8	15.5	256	91	120	12	0.69

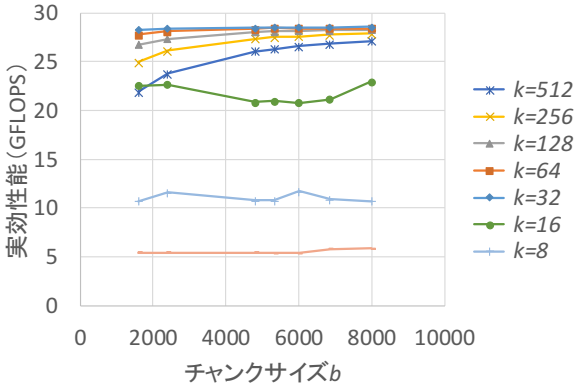
ンクの生存期間が短く，カーネル実行時間と比較してデータ転送時間が長くなり，結果としてデータ転送が性能を律速する．$k > 32$ では，チャンクの生存期間が延びデータ転送時間を削減できるが，冗長な計算に起因してカーネル実行時間が長くなる．

姫野 BMT では，$1 \leq k \leq 16$ において，実効性能が k とともに単調に増加した．したがって，チャンクサイズ $b = 102$，ブロッキング段数 $k = 16$ のとき，実効性能が最大になり，37.5 GFLOPS であった．一方，$k > 16$ とすると，デバイスメモリが枯渇して実行に失敗した．ここで，トレードオフの関係をふまえると，最適なブロッキング段数は $k > 16$ の範囲に存在する．すなわち，この評価実験では，ブロッキング段数 k を選択できなかった．

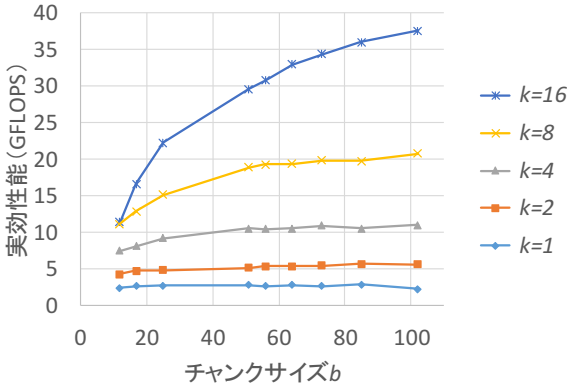
このようにデバイスメモリ容量の不足が原因で，最適なブロッキング段数 k を選択できない場合がある．1 次元ブロック分割を採用している現在の実装では，1 つのチャンクに対して大きさ $(b + 2hk) \times Y \times Z$ のデータをデバイスメモリに格納する．ここで，デバイスメモリ容量の制約から，Y あるいは Z が増大すると，選択できる k の上限が小さくなる．つまり，Y あるいは Z が増大する状況において，大きな k を選択するためには，他のデータ分割手法を検討する必要がある．

CIP 法では，チャンクサイズ $b = 2750$，ブロッキング段数 $k = 16$ のとき，実効性能は最大になり，73.4 GFLOPS であった．$8 \leq k \leq 256$ において，いずれも実効性能は 70 GFLOPS 前後であった．CIP 法は，他のテストコードと比べて演算強度 G が高いため，テンポラルブロッキングを適用せずとも，データ転送時間に対するカーネル実行時間の比率が高い．したがって，ブロッキング段数 k とともに比率を高めることの効果が小さく，小さな k で向上の度合いが頭打ちとなった．$k > 16$ では，冗長な計算に起因してカーネル実行時間が長くなり，実効性能が低下した．

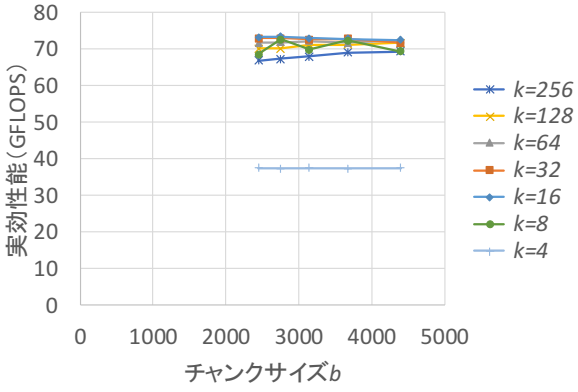
表 3 に，すべてのデータをデバイスメモリに格納できるインコア版 OpenACC 実装との実効性能の比較を示す．いずれのテストコードにおいても，実効性能の低下は 20%程度に留まった．この性能低下は，メモリ帯域幅律速な計算において，大規模データを扱えるディレクティブとしては，許容範囲である．



(a) ヤコビ法



(b) 姫野 BMT



(c) CIP 法

図 6 実行パラメタを変えたときの実効性能
Fig. 6 Effective performance with different execution parameters.

表 3 実効性能の比較（GFLOPS）
Table 3 Comparison of effective performance in GFLOPS.

テストコード	インコア実装	アウトオブコア実装
ヤコビ法	32.2	28.5
姫野 BMT	47.5	37.5
CIP 法	83.9	73.4

表 4 テストコードの長さの比較（行）
Table 4 Comparison of test code lengths in lines.

テストコード	C	OpenACC	入力コード	出力コード
ヤコビ法	98	104	104	271
姫野 BMT	220	235	236	541
CIP 法	149	164	160	402

6.2 ディレクティブの挿入行数

表 4 に，逐次コード，OpenACC コード，トランスレータの入力コードおよび出力コードのコード行数を示す．ここで，逐次コード，OpenACC コードおよびトランスレータの入力コードは，それぞれ手動で実装した．これら 4 種類のコードは，ステンシル計算コードに加えて，ヘッダファイルの読込および配列データの初期化などを含み，単独で実行できるものとする．いずれのテストコードにおいても，OpenACC コードと入力コードの行数の差が小さい．したがって，提案するディレクティブの挿入行数は，OpenACC におけるそれと同程度である．

7. まとめ

本論文では，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自動で適用するディレクティブおよびそのトランスレータの実装を示した．開発したトランスレータは，ユーザが挿入したディレクティブに基づき，データ分割，テンポラルブロッキングおよびパイプライン処理を実現するよう入力コードを変換し，OpenACC コードを出力する．

ヤコビ法，姫野 BMT および CIP 法による評価実験では，実効性能はそれぞれ 28.5，37.5 および 73.4 GFLOPS であった．いずれの場合でも，インコア実装と比較して 20%程度の実効性能の低下に留まっている．

今後の課題としては，実行パラメタ k および b の自動推定が挙げられる．

謝辞 本研究の一部は，科研費 15K12008，15H01687，16H02801 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 7.5 (2015). http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf.
[2] Kato, T., Ino, F. and Hagihara, K.: PACC: An Extension of OpenACC for Pipelined Processing of Large Data on a GPU, *Poster 27th Int'l Conf. High Performance Computing, Networking, Storage and Analysis (SC'14)* (2014).
[3] 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.
[4] 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, p. 6 pages (2014).
[5] 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).
[6] Endo, T., Takasaki, Y. and Matsuoka, S.: Realizing Extremely Large-Scale Stencil Applications on GPU Supercomputers, *IEEE Trans. Parallel and Distributed Systems*, pp. 625–632 (2015).
[7] Nakao, M., Murai, H., Shimosaka, T., Tabuchi, A., 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'14)*, pp. 27–36 (2014).
[8] xcalablemp.org: XcalableMP. http://www.xcalablemp.org/.
[9] Midorikawa, H. and Tan, H.: Locality-Aware Stencil Computations Using Flash SSDs as Main Memory Extension, *Proc. 15th IEEE/ACM Int'l Symp. Cluster, Cloud and Grid Computing (CCGRID'15)*, pp. 1163–1168 (2015).
[10] Himeno, R.: Himeno benchmark (2015). http://accc.riken.jp/en/supercom/himenobmt/.
[11] 矢部　孝，尾形陽一，滝沢研二：CIP 法と Java による CG シミュレーション，森北出版 (2007).
[12] NVIDIA Corporation: PGI Compiler (2015). http://www.pgroup.com/.