Vol.2016-HPC-157 No.18
2016/12/22

ディレクティブによる時空間ブロッキングの自動適用

黒田 勝汰[1,a)]　遠藤 敏夫[1,b)]　松岡 聡[1,c)]

概要：ステンシル計算向けのループ最適化である時空間ブロッキングは非常に高い効果があるが，ループの制御が複雑になるためプログラミングコストが大きく，汎用的な最適化ではない．そのためループ変換ツールやステンシル向け DSL コンパイラの機能として実装されてきた．しかし，これらはパラメータ設定の柔軟性や対象プログラムの大幅な書き換えが必要という点で問題を抱えている．そこで，我々はディレクティブによる時空間ブロッキングの適用を提案する．いくつかの条件を満たすループにディレクティブにより指定されたパラメータで時空間ブロッキングを適用するツールを実装した．ステンシルベンチマークを用いて提案システムの性能改善効果とプログラミングコストを評価する．

1. はじめに

ステンシル計算は流体計算や構造計算を含む様々な科学技術計算などで用いられている重要なカーネルの一つである．多くのステンシル計算は，データ量と計算量のオーダが同じであり，データインテンシブである傾向にある．そして，ナイーブな実装をされたステンシル計算においては，(時間ステップなどの) 各イテレーションごとに計算対象の配列全体を走査するので，キャッシュからデータを追い出してしまい，メインメモリへの負荷が高い．

メインメモリへの負荷を下げ性能を向上させるためには，データの再利用性を高めることによりメモリ階層を効率的に利用するブロッキング手法が一般的に有効である．ステンシル計算においては，空間方向にブロックを設ける空間ブロッキングが良く知られているが，さらに大幅に再利用性を高める手法として，時空間ブロッキング (Temporal Blocking, 以下 TB) がすでに提案されている [12], [13], [14]．これは，ひとたび配列のうちのある空間ブロックの計算を始めたら，局所的に複数の時間ステップの計算を一度に行うものである．TB はキャッシュとメインメモリ間のデータ移動の削減だけでなく，GPU メモリ-CPU メモリ間の通信の削減にも用いられている [7], [15]．

しかし，TB の導入によりループ構造が複雑になり，プログラミングコストが高くなるという課題がある．この大きな要因は，密行列演算などと異なりステンシル計算においては，隣接データ間の依存関係を守るために，時間に対して斜め方向のブロック分割が必要であることである．低プログラミングコストでの TB を実現するために，ステンシル向け DSL に TB を導入する研究も行われている [3], [16] が，この場合にはアプリケーションは DSL を用いて記述されている必要がある．本研究では C/C++などの汎用言語で記述されたプログラムを対象に想定する．そしてこれらを対象に TB のためのループ変換を行うために，Polyhedral compiler 技術 (Pluto コンパイラ [8] や LLVM-IR 向けの Polly ツール [1] などが代表的) を用いるアプローチを取る．

TB の導入が実現されたとしても，さらには空間方向・時間方向のブロックサイズをどのように設定すべきか，という課題がある．またブロック形状についても，wavefront タイリング，overlapped タイリング，trapezoid タイリング，diamond タイリングなど複数提案されている．このようなブロックサイズ・形状の最適解は，カーネルおよび計算機アーキテクチャ双方の性質に依存すると考えられるため，固定的な手法は好ましくない．本研究では，TB におけるブロックサイズ・形状などのパラメータおよび TB 適用の有無を，プログラマが容易に制御可能なように，ディレクティブに基づいた制御機構の提案を行う．

以上のような，(1) TB 向けパラメータの制御機構，(2) TB のためのループ変換機構を，コンパイラツールチェーン LLVM[2] への拡張として実現する．(1) については LLVM のための C コンパイラである clang[6] を，(2) については LLVM-IR 向け Polyhedral ループ最適化ツールである Polly を基に拡張を行った．これらの拡張コンパイラツールチェーンを用いた予備実験を，1 次元および 2 次元ステンシルベンチマークを用いて行い，その効果を実証した．

[1] 東京工業大学
[a)] kuroda.s.ac@m.titech.ac.jp
[b)] endo@is.titech.ac.jp
[c)] matsu@is.titech.ac.jp

2. 背景

2.1 時空間ブロッキング

時空間ブロッキング (以下 TB) はステンシル計算の空間ループと時間ループをブロック化する手法である．あるブロックの計算を始めたら，局所的に複数の時間ステップ（空間ブロックサイズ分）の計算を一度に行う．これにより，空間方向のブロッキングのみの場合よりも更にメモリアクセス局所性を向上させる．このようなブロッキングを導入する際，ブロッキングを使わない元々の計算に比べて計算順序が変更されるが，それでも元々の計算に内在する依存関係を壊さないようにしなければならない．単純な一次元 3 点ステンシルにおける，各点の計算式は以下のように表される．

$$f_{t+1,x} = a_{-1} f_{t,x-1} + a_0 f_{t,x} + a_1 f_{t,x+1}$$

このとき，ある点の計算のためには，ひとつ前の時間ステップにおける，自分自身の隣接点の計算が終わっている必要がある．上記の式は，一つ隣りの点を用いるステンシル計算（つまりステンシル半径が 1）である．

このような依存関係を満たすブロッキング手法の例を図 1 に示す．ブロックの時間方向の長さを TBSize，空間方向で最も大きい部分の長さを SBSize とすると，図は 1 次元 3 点ステンシルにおいて，TBSize = 3，SBSize = 9，ブロック形状が trapezoid の例を示す．依存関係を保つために，まず図の赤いブロック部分の計算を行う．この赤いブロックは時間ステップが進むたびに，空間方向に (ステンシル半径ずつ) 小さくなる．それが終了したら，青いブロックの計算を行う．その後，次の時間ブロックへ進むことができる．このような計算順序をプログラミングする際には，ブロックを時間に対して斜めに分割する必要があるため，ループが複雑になってしまう．また，対象プログラムのステンシル半径によって分割する際の分割面の傾きを変える必要がある．

TB のブロッキング形状は上記の trapezoid 型だけではなく，図 2 に示すように多数提案されている．本報告においては，trapezoid 型のブロッキングを用いるが，提案手法は overlapped 型を除いたほかの手法にも原理的に同様に適用可能である．Overlapped 型は，ループ構造が他のものより容易ではあるものの，ブロックが重なった部分の計算が冗長という短所があり，対応できなくても大きな支障はないと考える．

2.2 LLVM Compiler Infrastructure

LLVM Compiler Infrastructure(以下 LLVM) とは再利用可能なコンパイラツール、ライブラリである．NVIDIA GPU 向けコンパイラである nvcc や Polyhedral 最適化ツールである polly など多くの有用なツールが LLVM を用いて

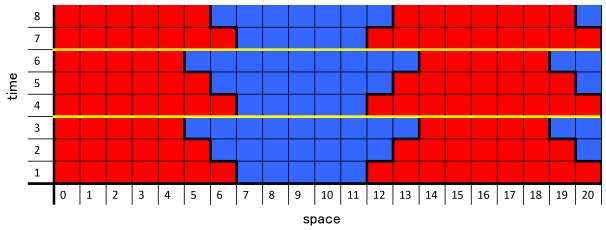

図 1 時空間ブロッキングの例．ブロック形状が trapezoidal の場合

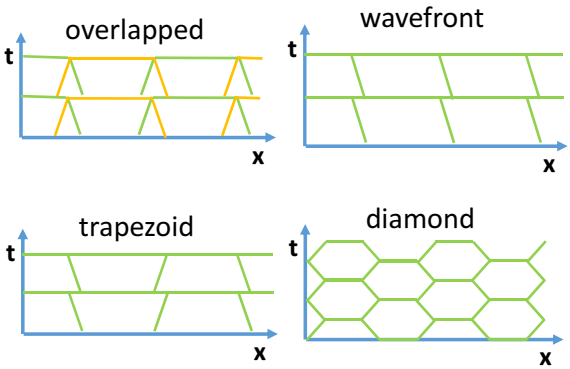

図 2 時空間ブロッキングの形状の例

実装されている．様々な言語やターゲットアーキテクチャに対応しているのが特徴で共通の中間表現 (以下 IR) を用いる．LLVM を用いたコンパイルは以下の 3 つのステップで行われる．

(1) フロントエンド:入力言語から IR を出力
(2) ミドルエンド:Pass が IR を最適化/解析し IR を出力
(3) バックエンド:IR から出力言語を出力

LLVM では入力言語にたいしてフロントエンドが IR に変換する．その際に，ディレクティブにより付加された情報や中間表現では表現されない情報 (入力言語でのデータ型など) などを metadata として IR 中に追加する．ミドルエンドやバックエンドでは metadata の情報を用いて最適化や解析を行う．本報告で扱った clang は LLVM の C フロントエンドである．

ミドルエンドでは Pass というモジュールごとに最適化/解析処理が管理され，最適化/解析が繰り返し行われる．Pass は他の Pass の解析結果を使うことができ，その場合は先に実行する解析 Pass を記述する必要がある．本報告で扱った Polly の最適化は全て Pass として実装されている．

LLVM は共通の中間表現を用いることにより，様々な入力/出力言語に対応している．LLVM では IR にフロントエンドや他の Pass で得た最適化/解析情報を metadata を用いて埋め込むことができる．本報告では C 言語のフロントエンドである clang を元にディレクティブを追加したフロントエンドを作成した．

2.3 Polly

Polly は Polyhedral モデルを採用した LLVM-IR 向け

ループ最適化ツールである．Polly の C++クラスを継承することによって Polyhedral 最適化を行う Pass を実装することができる．本報告では Polly を用いて Polyhedral モデル上で TB を適用する Pass を実装した．

2.3.1 Polyhedral モデル

Polyhedral モデルは n 重ループの各反復を n 次元整数凸集合内の点として扱うモデルである．整数凸集合として扱うことで，ループ表現，変換，依存性解析を統一的に扱うことができる．整数凸集合であるという条件を満たすために，扱う対象にいくつか制約を課している．制約を満たしたプログラムの部分のことを Static Control Part of program(SCoP) と呼ぶ．

2.3.2 SCoP 条件

Polly[1] が最適化対象に課している制約は以下のものである．

(1) ループ誘導変数が 1 つ存在し，下限から上限まで 1 づつ増加する
(2) 上限と下限は周りのループのループ誘導変数とパラメータのアフィン表現で表されている (パラメータは SCoP 内で変更されない整数変数)
(3) SCoP 内のステートメントは配列要素に対する式の代入のみ
(4) 代入する式はパラメータ，配列要素，ループ誘導変数をオペランドにとる副作用のない演算子による表現
(5) 配列の添字はループ誘導変数とパラメータによるアフィン表現

SCoP 条件を満たすプログラムの例を図 3 に示す．ステンシル計算においては特に，ポインタ交換やポインタ配列などが扱えないので注意する必要がある．

2.3.3 Polyhedral 表現

SCoP はステートメントの集合として表現され，ステートメントは以下の要素で定義される．

- domain:ステートメントを囲むループのループ誘導変数のとりうる範囲を表す集合
- schedule:ステートメント同士の実行順序を表す写像
- memory accesses の集合:ステートメントが行うメモリアクセスの集合

domain はループ誘導変数がとりうる値の範囲を表す整数集合である．ステートメントを囲むループが n 重ループの場合，n 次元の整数集合となる．

schedule は domain から実行順序を表す k 次元の整数 (タイムスタンプ) に移す写像である．二つの反復をそれぞれタイムスタンプに写像し，タイムスタンプ同士を辞書順で比較し，若い方が先に実行される．同一のタイムスタンプに移された場合，二つの反復は並列に実行可能であることを示す．

memory accesses の集合はある反復でステートメントがアクセスするメモリアドレスとそのアクセス種類を表す写像の集合である．

```
void calc(float *d,float *s,const int nx){
  const float a = 1.0f / 3.0f;
  int x;
  for(x=0 ; x<nx ; ++x){
S1: d[x] = a * (s[x-1] + s[x] + s[x+1]);
  }
}
```

図 3 SCoP 条件を満たすコードの例

```
"statements" : [ {
  "accesses" : [ {
    "kind" : "read",
    "relation" : "[nx] ->
                        { S1[x] -> s[-1 + x] }"
   },{
    "kind" : "read",
    "relation" : "[nx] ->
                        { S1[x] -> s[x] }"
   },{
    "kind" : "read",
    "relation" : "[nx] ->
                        { S1[x] -> s[1 + x] }"
   },{
    "kind" : "write",
    "relation" : "[nx] ->
                        { S1[x] -> d[x] }"
   }
  ],
  "domain" : "[nx] -> { S1[x] : 0 <= x < nx }",
  "name" : "S1",
  "schedule" : "[nx] -> { S1[x] -> [x] }"
 }
]
```

図 4 Polly による図 3 の Polyhedral 表現

Polly によって作られた Polyhedral 表現を図 4 に示す図 4 に示されているようにループ誘導変数が動く範囲は domain で不等式によって表現される．schedule ではループ誘導変数 x がタイムスタンプ x に移されているので，x の値が若い方が先に実行される．accesses ではメモリアクセスの種類（read, write）とそのアドレスが写像として表されている．

3. 関連研究

ステンシル計算に対して TB を適用することによってメモリ階層を活用する研究が多くなされてきた．ステンシルプログラムに対する TB の適用に関しては多くの研究がなされてきた．GPU メモリ-CPU メモリ間 [7], [15] や，CPU メモリ-SSD 間 [9] に適用され，効果をあげている．

また，ステンシル計算の DSL コンパイラに TB が実装

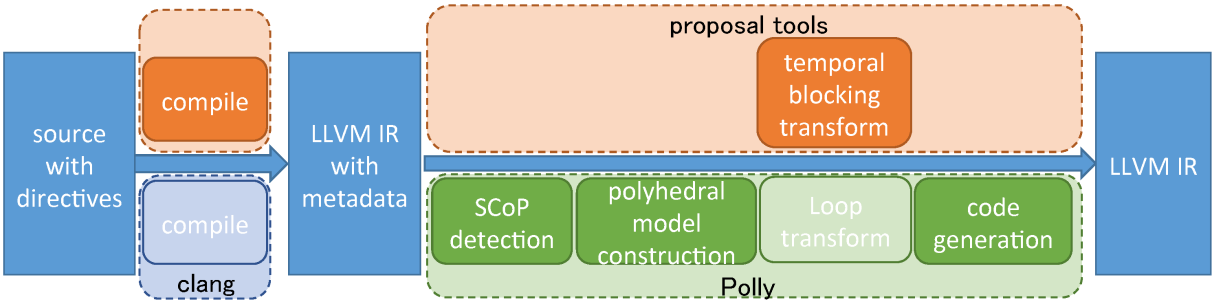

図 6 処理の流れ

されている [3], [16]. DSL を用いることによってプログラムの記述を容易にし，なおかつ TB などの最適化を使用者が大きなコストをかけず適用できる．しかし，既存アプリに TB をかけたい場合 (本報告で対象にする) はプログラムの書き換えが必要になる．

汎用言語向けではループ最適化ツール Pluto[8] に TB が実装されている．Pluto では変換対象のプログラムをディレクティブで指定し，コンパイラオプションで変換を指定することによって適用できる．しかし Pluto でも TB のパラメータの指定ができず，対象プログラムにも疑似多次元配列が扱えないなどの制約がある．

4. 実装

本章では作成したツールの使用方法と実装について述べる．

4.1 ディレクティブの設計

提案システムでは適用対象ごとにブロックサイズなどのパラメータを柔軟に指定するという目的を果たすため，パラメータ情報を対象の時間ループの直前に指定するようにした．現在実装されている指示句を以下に示す．

- tile_size：TB のブロックサイズの指定
- radius：ステンシル半径の指定

tile_size 指示句では時間ループとそれぞれの空間ループのブロックサイズをコンマ区切りで指定する．一番左に指定された整数を時間ループのブロックサイズとする．それに続く数値は時間ループの内側の空間ループのブロックサイズとする．これによって指定されたパラメータをブロックサイズとして変換する．

radius 指示句では各空間次元のステンシル半径を指定する．対象ステンシルをステンシル半径を外側の空間ループから順に指定する．これによって指定されたステンシル半径に従って，依存性を壊さないように TB を適用する．

図 5 にディレクティブの使用例を示す．このディレクティブの指定によって時間ブロックサイズを 16，y 軸のブロックサイズを 64，x 軸のブロックサイズを 128，y 軸のステンシル半径を 1，x 軸のステンシル半径を 2 として変換する．また現在はディレクティブが指定されたループが含むすべてのループ（空間ループ）に対して TB を適用する．

4.2 ツール構成

提案システムは LLVM clang，Polly を用いて実装した．ツールの処理の流れを図 6 に示す．提案システムは以下の処理を行う．

(1) ディレクティブに指定されたパラメータ情報を含む中間表現の生成 (custom clang)
(2) SCoP 条件に適合する部分を検出 (Polly SCoP 検出 Pass)
(3) Polyhedral 表現を生成 (Polly Polyhedral 表現作成 Pass)
(4) パラメータ情報を含む SCoP であった場合，Polyhedral 表現に TB を適用 (TB-Pass)
(5) 変更された SCoP を IR に適用 (Polly codegen Pass)

Polly が扱うには SCoP 条件を満たさなくてはならないので，本システムの適用対象のコードも先述の SCoP 条件を満たしたものでなくてはならない．

4.3 TB-Pass の実装

提案システムでは TB の適用は TB-Pass が行なっている．TB-Pass は Polly の ScopPass クラスを継承することで実装されている．ScopPass クラスは Polly が検出した SCoP に解析/最適化を行う Pass を実装するためのクラス．ScopPass クラスの runOnScop(Scop &S) 関数に解析/最適化処理を実装する．

Polly の SCoP は図 4 で示されている形式で管理していて，その中の schedule は integer set library(以下 isl)**??**で用意されている isl_map 型として表されている．isl は整数集合/写像/関数を扱う汎用のライブラリである．isl では写像を扱う API を用意しているが，API の汎用性のため TB のような複雑な式を扱おうとすると多くの API 呼び出しが必要になってしまう．TB-Pass では実装を単純にするため，一度元のスケジュールを表す isl_map を文字列にし，文字列操作で TB を適用した後，isl_map 型に再度変換するようにした．

```
#define IDX(x,y) ( (y) * stride + (x) )

float *f[2];
const float a;

f[0] = malloc( ... );
f[1] = malloc( ... );

#pragma tb tile_size(16,64,128) radius(1,2)
// 時間ループ
for(int t=0 ; t<nt ; ++t){
  // f[(t + 1) % 2][IDX(x,y)] とすると
  // 多重ポインタになってしまい条件SCoP (3) に違反するので
  // t を偶数と奇数で分け、単純な配列アクセスにする
  if(t % 2 == 0){
    // Y ループ
    for(int y=0 ; y<ny ; ++y){
      // X ループ
      for(int x=0 ; x<nx ; ++x){
        f[1][IDX(x,y)] =
                a * f[0][IDX(x,y-1)]
              + a * f[0][IDX(x-2,y)]
              + a * f[0][IDX(x-1,y)]
              + a * f[0][IDX(x,y)]
              + a * f[0][IDX(x+1,y)]
              + a * f[0][IDX(x+2,y)]
              + a * f[0][IDX(x,y+1)];
  } else {
    for(int y=0 ; y<ny ; ++y){
      for(int x=0 ; x<nx ; ++x){
        f[0][IDX(x,y)] =
                a * f[1][IDX(x,y-1)]
              + a * f[1][IDX(x-2,y)]
              + a * f[1][IDX(x-1,y)]
              + a * f[1][IDX(x,y)]
              + a * f[1][IDX(x+1,y)]
              + a * f[1][IDX(x+2,y)]
              + a * f[1][IDX(x,y+1)];
      }
    }
  }
}
```

図 5 ダブルバッファリングされたコードでのディレクティブの使用例

図 7 に 1 次元ステンシルでの schedule を表す isl_map の変更の例を示す．タイムスタンプ [i0, 1, i1] に T, 0/1,BlockID を追加している．それぞれ時間ブロックループ，ブロックの種類（ブロック形状による順序），空間ブロックループを表している．

TB-Pass は schedule を変換した後 Polly の依存性チェックを行い，依存性が守られる schedule だった場合，変換した schedule を適用する．

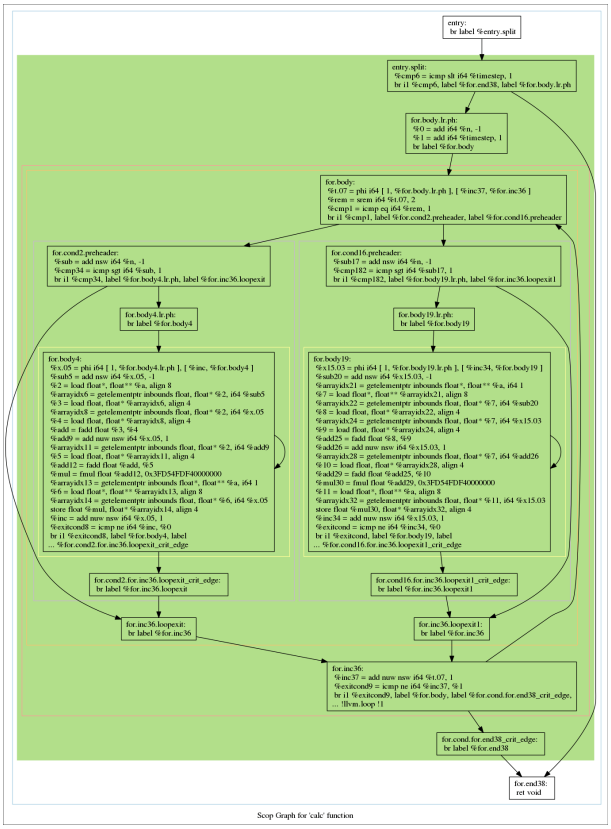

図 8 TB 適用前の 1 次元 3 点ステンシルの IR

図 8,9 に 1 次元 3 点ステンシルの TB 適用前の IR と適用後の IR を示す．適用後の IR には適用されたコードだけでなく適用前のコードも残っていて，SCoP に入った時のパラメータによって適用後のプログラムが適正かをチェックするコードが Polly によって追加されている．

5. 評価

提案システムの変換コストと性能向上効果を評価するため，ダブルバッファリングされた 1 次元 3 点ステンシルと 2 次元 5 点ステンシルに対して，手動で時空間ブロッキングをかけた場合と比較した．計測には 2 種類の計算機を用いた (Sandy,KNL)．表 1 にその詳細を示す．

コンパイルは clang 4.0.0 を用いた．最適化に関するコンパイラオプションは”-O3 -march=native -fno-vectorize”を指定した．現在提案システムを用いた場合，clang による自動ベクトル化機能は使えないため比較対象のコンパイル時にはベクトル化機能をオフにした．

変換前のプログラムは最外空間ループに OpenMP による並列化ディレクティブを用いて並列化している (collapse 節は不使用)．手動で空間ブロッキング/時空間ブロッキングを適用したプログラムではブロックループを並列化ディレクティブを用いて並列化している (collapse 節を使用)．提案システムでは Polly の自動並列化機能を用いて並列化した．

それぞれの時間ブロックサイズに対して最適なブロック

```
// 適用前
[nx, nt] -> { S1[i0, i1] -> [i0, 1, i1] }
// [nx, nt] : パラメータ(内で変更されない整数変数SCoP)
// S1 : ステートメントの名前
// [i] -> [i'] : ループ誘導変数がの反復をタイムスタンプiiに移す'

// 適用後
[nx, nt] ->
 { S1[i0, i1] ->
     [T, 0, BlockID1, i0, 1, i1] :
      ( T = floor(i0 / 2) and
        BlockID1 = floor( ( i1 + 5 * (1 - ( i0 - 2 * T))) / 16 ) and
           floor( ( i1 + 5 * (1 - ( i0 - 2 * T))) / 16 )
        =  floor( ( i1 - 13 + 5 * ( 1 + ( i0 - 2 * T)) + 16) / 16 ) ) ;
   S1[i0, i1] ->
     [T, 1, BlockID1, i0, 1, i1] :
      ( T = floor(i0 / 2) and
        BlockID1 = floor( ( i1 + 5 * (1 - ( i0 - 2 * T))) / 16 ) and
           floor( ( i1 + 5 * (1 - ( i0 - 2 * T))) / 16 )
        != floor( ( i1 - 13 + 5 * ( 1 + ( i0 - 2 * T)) + 16) / 16     ) )
 }
```

図 7 1 次元ステンシルでの SCoP の schedule に対する TB-schedule 適用の例. 時間ブロックを 2, 空間ブロックを 13, ステンシル半径を 5 とした.

サイズを使用して計測した. スレッド数はコアあたりのスレッド数を Sandy で 1 から 2, KNL で 1 から 4 まで変えて最適だったデータを用いた.. 計測プログラムでは単精度浮動小数点数を用いている.

TB 適用での変換コストを表 2 に示す. 演算子は C 言語での加減乗除と MIN/MAX 演算を 1 演算とした. 提案システムでのコストは SCoP 条件に適合させるための変換とディレクティブの追加を含んでいる. 手動で TB を適用した場合はループ誘導変数の下限と上限の計算のために演算子数が大きく増えてしまっている.

図 10,11 に Sandy と KNL において 1 次元 3 点ステンシルに対して, 最適化を施さない場合 (TB-auto), 提案システムを用いて TB を適用した場合と手動で TB を適用した場合 (TB-manual) の性能を示す. 問題サイズは空間サイズを 16777216, 時間ステップ数を 2048 とした.

提案システムで TB を適用することにより性能が向上することが確認できる. 提案システムによって TB を適用していないものと比べて Sandy では最大 1.4 倍, KNL では最大 1.8 倍の性能が得られた. TB-manual と比較すると性能はやや落ちるが全体の傾向としてはおおよそ同じである. TB-auto の性能が TB-manual のものと比べて低くなっている理由は, TB のループが複雑なため Polly により生成されたコードも複雑になってしまい, 他のコンパイラ最適化がかかりづらくなってしまったものと考えられる. 最適なブロックサイズは 1 スレッド分のブロックのデータが L1D-cache に収まるもの程度のサイズが最適になった.

図 12,13 に Sandy と KNL において 2 次元 5 点ステンシルに対して, 最適化を施さない場合, 提案システムを用いて TB を適用した場合, 手動で空間ブロッキング/時空間ブロッキングを適用した場合 (SB-manual/TB-manual) の性能を示す. 問題サイズは空間サイズを 4096x4096, 時間ステップ数を 2048 とした.

Sandy においては自動 TB 適用による性能向上が得られなかった. 手動で TB をかけた場合は最大 1.17 倍の性能向上が得られている. KNL では Sandy の場合と比べて TB の効果が高い. TB-manual で最大 1.8 倍, TB-auto で最大 1.05 倍の性能が得られた.

Sandy と KNL において自動 TB の性能は単純な空間より悪くなっている. 自動 TB による効果が低い原因は, TB による性能向上が小さいため, 他の最適化がかかりづらいことによる性能差の方が大きくなってしまったことが考えられる. また, そもそもなぜ TB の効果が低いのかについては, X 方向 (メモリ上で連続) の問題サイズが小さいため, X 軸の TB の効果がなく, ループ制御のオーバーヘッドが TB の効果を低くしているのではないかと考えた. そこで X 方向の問題サイズを十分大きくして Sandy で計測した.

図 14 に Sandy において 2 次元 5 点ステンシルに対して, 最適化を施さない場合, 提案システムを用いて TB を適用した場合, 手動で空間ブロッキング/時空間ブロッキングを適用した場合 (SB-manual/TB-manual) の性能を示す. 問題サイズは空間サイズを 42097152x512, 時間ステップ数を

表 1 計測環境

	環境 1(Sandy)	環境 2(KNL)
CPU	Intel Core i7-3930K	Intel Xeon Phi 7210
動作周波数	3.2GHz	1.3GHz
コア数	6	64
L1D cache size	32kB	32kB
last level cache size	12MB	32MB
メモリ帯域	51.2GB/s	102GB/s

表 2 TB 適用のコスト (変換前カーネルは 6 行)

	提案システム		手動	
	1D	2D	1D	2D
編集/追加した行数	7	9	18	44
追加した演算子数	2	2	70	270

32 とした．TB 適用前と比べて TB-manual では最大 2.1 倍，TB-auto では最大 1.4 倍の性能が得られた．

6. おわりに

本報告では既存のプログラムに少ないコストで TB を適用するために，ディレクティブによるテンポラルブロッキング (TB) の適用を提案し，変換ツールを LLVM への拡張の形で実装し，初期評価を行った．提案システムを用いることにより 1 次元 3 点ステンシルにおいては Sandy で最大 1.4 倍, KNL で最大 1.8 倍，2 次元 5 点ステンシルにおいては Sandy で最大 1.4 倍，KNL で最大 2.1 倍の性能が得られた．提案システムを用いた場合，C 言語コードの多重ループ部分に，TB の空間ブロックサイズや時間ブロックサイズをディレクティブのオプションとして記述することにより，ユーザによる容易なチューニングを可能にする．

今後の方向性について議論する．ディレクティブによるパラメータの制御機構については，現在未対応のブロック形状の指定に対応する計画である．また空間ブロックサイズについても，次元によってはあえてブロック分割をしない方が高性能になる場合も考えられ，そのような場合も容易に制御できることが望ましい．

Polyhedral モデルによるループ変換機構については下記の通りである．現時点では単純なステンシルベンチマークへの適用を行ったが，より実用的なアプリケーションにおける容易な TB の実現に向けて，調査・改良を行う予定である．現在，上記の目的を妨げる問題の多くは Polly の SCoP 条件の厳しさに由来する．まず，SCoP としてみなされるためにはループ内のステートメントが制限され，特に関数呼び出しがあると変換ができない．しかし実用的なアプリケーションでは，時間ループが複数の関数にまたがることは珍しくない．単純な方法としてはそのような関数をコンパイル段階でインライン化してしまうことが考えられるが，それが困難な場合にも対応できる手法を考案が望ましい．また，演算対象の配列が構造体の中からポインタとして参照されているような場合に，SCoP の判定が失敗することが分かっている．実際のアプリケーションで頻発するパターンについて，SCoP 条件に反する場合を洗い出し，変換可能なケースを増加する改良を行っていく必要もある．

謝辞 本研究は JST-CREST「EBD: 次世代の年ヨッタバイト処理に向けたエクストリームビッグデータの基盤技術」および JST-CREST「ポストペタスケール時代のメモリ階層の深化に対応するソフトウェア技術」の支援を受けている．

参考文献

[1] Grosser, T., et al.: Polly-Polyhedral optimization in LLVM, Proceedings of the First International Workshop on Polyhedral Compilation Techniques (IMPACT). Vol. 2011. 2011.
[2] Lattner, C. A.: LLVM: An infrastructure for multi-stage optimization, Diss. University of Illinois at Urbana-Champaign, 2002.
[3] 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，情報処理学会研究報告，Vol. 2014-HPC-143, No. 32 (2014).
[4] Bondhugula, U., et al.: PLuTo: A practical and fully automatic polyhedral program optimization system, Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI 08), Tucson, AZ (June 2008). 2008.
[5] Malas, T., et al.: Multicore-optimized wavefront diamond blocking for optimizing stencil updates, SIAM Journal on Scientific Computing 37.4 (2015): C439-C464.
[6] Lattner, C.: LLVM and Clang: Next generation compiler technology, The BSD Conference. 2008.
[7] Jin, Guanghao, Toshio Endo, and Satoshi Matsuoka.: A parallel optimization method for stencil computation on the domain that is bigger than memory capacity of GPUs, 2013 IEEE International Conference on Cluster Computing (CLUSTER). IEEE, 2013.
[8] Bondhugula, Uday, et al.: PLuTo: A practical and fully automatic polyhedral program optimization system, Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI 08), Tucson, AZ (June 2008). 2008.
[9] 緑川 博子, 丹 英之: 大規模ステンシル計算のための Flash SSD 向けテンポラルブロッキングの性能評価，情報処理学会 研究報告, Vol. 2014-HPC-145, No.22 (2014).
[10] Tang, Yuan, et al.: The pochoir stencil compiler, Proceedings of the twenty-third annual ACM symposium on Parallelism in algorithms and architectures. ACM, 2011.
[11] Unat, Didem, Xing Cai, and Scott B. Baden.: Mint: realizing CUDA performance in 3D stencil methods with annotated C, Proceedings of the international conference on Supercomputing. ACM, 2011.
[12] M. E. Wolf and M. S. Lam: A Data Locality Optimizing Algorithm. ACM PLDI 91, pp. 30–44 (1991).
[13] M. Wittmann, G. Hager, and G. Wellein: Multicore-aware parallel temporal blocking of stencil codes for shared and distributed memory. Workshop on Large-Scale Parallel Processing (LSPP10), in conjunction with IEEE IPDPS2010, 7pages (2010).
[14] A. Nguyen, N. Satish, J. Chhugani, C. Kim, and P.

Dubey: 3.5-D blocking optimization for stencil computations on modern CPUs and GPUs. IEEE/ACM SC'10, 13 pages (2010).
[15] L. Mattes, S. Kofuji: Overcoming the GPU memory limitation on FDTD through the use of overlapping subgrids. International Conference on Microwave and Millimeter Wave Technology (ICMMT), pp.1536–1539 (2010).
[16] T. Muranushi, H. Hotta, J. Makino, S. Nishizawa, H. Tomita, K. Nitadori, M. Iwasawa, N. Hosono, Y. Maruyama, H. Inoue, H. Yashiro, Y. Nakamura: Simulations of Below-Ground Dynamics of Fungi: 1.184 Pflops Attained by Automated Generation and Autotuning of Temporal Blocking Codes, IEEE/ACM SC'16, 11pages (2016).
[17] Verdoolaege, Sven: Integer Set Library, an integer set library for program analysis (2008).

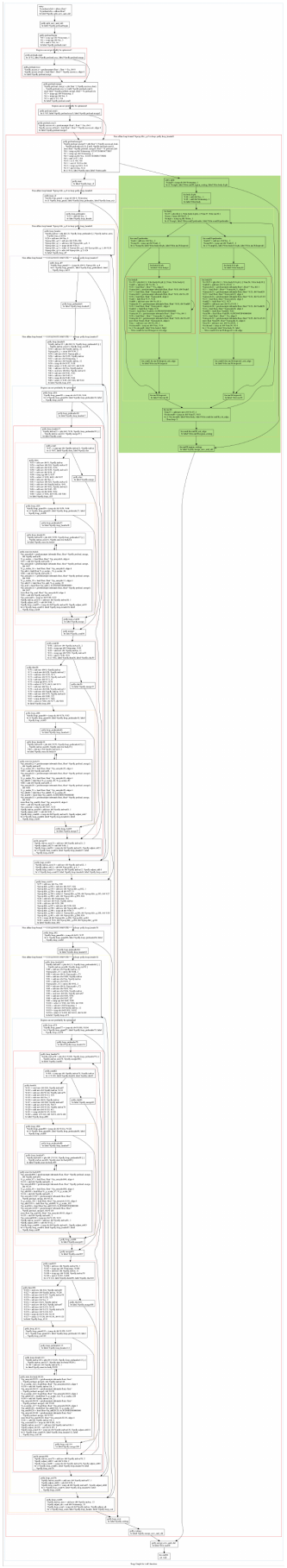

図 9 TB 適用後の 1 次元 3 点ステンシルの IR

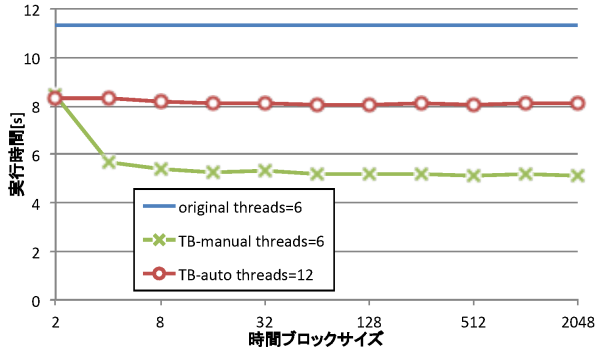

図 10 Sandy での 1 次元 3 点ステンシルでの TB の効果 (nx=16777216,nt=2048)

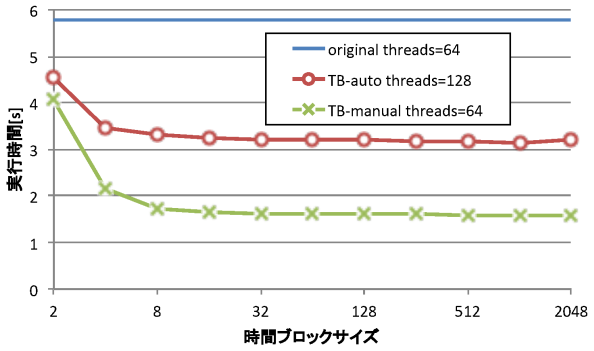

図 11 KNL での 1 次元 3 点ステンシルでの TB の効果 (nx=16777216,nt=2048)

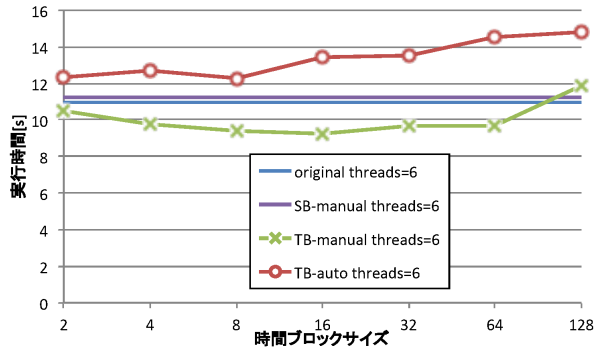

図 12 Sandy での 2 次元 5 点ステンシルでの TB の効果 (nx=4096,ny=4096,nt=2048)

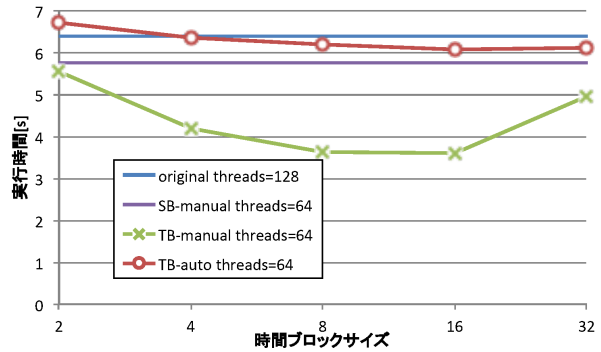

図 13 KNL での 2 次元 5 点ステンシルでの TB の効果 (nx=4096,ny=4096,nt=2048)

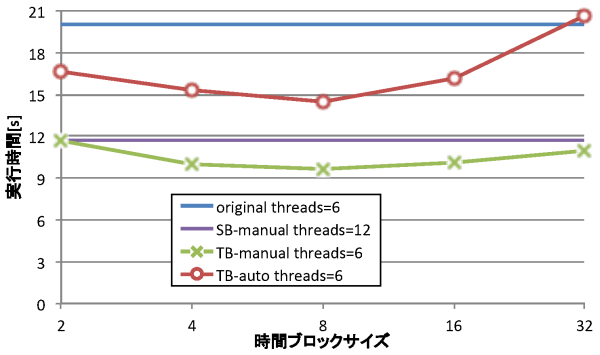

図 14 Sandy での 2 次元 5 点ステンシルでの TB の効果 (nx=2097152,ny=512,nt=32)