Vol.2021-HPC-180 No.16
2021/7/21

時間ブロッキングを用いたステンシル計算のHalide言語による高性能実装と評価

相川 洋貴[1,2,a)]　遠藤 敏夫[1,2]　幸 朋矢[1]　広渕 崇宏[2]

概要：ステンシル計算は演算回数に対してメモリアクセスがボトルネックとなる計算である．そこでメモリアクセスの空間的局所性を利用した最適化の手法として時間ブロッキングがよく用いられる．しかし時間ブロッキングはC言語などのforループを用いた実装ではコードが複雑になるため，実装が困難になってしまう問題がある．そこで本論文ではC言語などと比べると比較的にコードを簡素に記述できる利点があるドメイン特化言語Halideを用いて実装を行った．時間ブロッキングにはOverlapped tilingを用いて実装を行い，CPUではHalideでの時間ブロッキング実装によって，OpenMPでの時間ブロッキングなしでの実装と比べて1.3倍から2.9倍性能を向上することができた．また，GPUではHalideでの時間ブロッキング実装によって，CUDAによる時間ブロッキング実装の性能と比べて，0.95倍から1.54倍の性能を示した．

1. 序論

ステンシル計算は様々な科学技術計算において重要なカーネルであり，高性能に計算されることが求められている．ステンシル計算では，演算回数に対してメモリアクセス回数が多く，性能は計算機のメモリアクセス性能に依存する．現在の計算機ではメモリアクセス性能は演算性能と比べると相対的に低下してきているため，これを解決することが求められる．

ステンシル計算において，メモリアクセスを軽減する方法として時間ブロッキング [1–3] という手法が広く知られている．時間ブロッキングとは特定の範囲について一定の時間ステップ分をまとめて計算する方法である．これによりキャッシュメモリを活用し，キャッシュヒット率を上げることができる．

ステンシル計算において高性能に計算するためにあらゆる最適化方法を試すことはプログラマーにとって手間のかかる作業である．実際に計算順序を変えるとなると，コードを大幅に変える必要になることが多く，アーキテクチャごとに計算スケジュールの変更をするというのは大変である．

そこで本研究では時間ブロッキングを用いたステンシル計算の実装を，Halide言語というC++をベースとしたドメイン特化言語を用いることによって，計算スケジュールの簡略化を行いつつステンシル計算を高性能に実装するということが目的である．

Halide言語は計算アルゴリズムと計算スケジュールの記述が分離されているため，最適化のためのスケジュールを変更することが比較的容易になっている．またスケジュールの決定については最適な方法を探すことが容易ではないため，Halide言語に実装されているauto-schduler [4, 5] を用いることで良好なスケジュールを決定することができる．

本研究の成果としてCPUで56スレッド並列でステンシル計算の性能比較を行なった場合，Halideでの時間ブロッキング実装は，OpenMPでの時間ブロッキングなしでの実装と比べて1.3倍から2.9倍性能を向上することができた．またGPUでは1コアを用いて性能を比較し，Halideでの時間ブロッキング実装は，CUDAでの時間ブロッキングでの性能と比べて，0.95倍から1.54倍の性能を示した．

2. 背景

2.1 ステンシル計算

ステンシル計算とは流体計算などの数値シミュレーションなどにおいてよく使われる重要な計算である．2次元の場合5点ステンシルや9点ステンシルなどがあるが，本論文では9点ステンシル計算について取り扱う．ステンシル計算は通常各時間ステップで，キャッシュメモリサイズよりも大きい配列全体を走査する．そのため時間ステップ

[1] 東京工業大学
Tokyo Institute of Technology
[2] 国立研究開発法人 産業技術総合研究所
National Institute of Advanced Industrial Science and Technology
[a)] aikawa.h.ae@m.titech.ac.jp

が進むたびにメインメモリへのアクセスが起こる．ステンシル計算ではメモリアクセスに対する演算回数が少ないため，ステンシル計算の高速化のためには時間的局所性や空間的局所性などを考えたメモリアクセスのコスト削減が必要不可欠である．空間的局所性の観点で最適なものとしてタイリングという手法が用いられる．一定のサイズの空間ブロックに分けて計算することでメインメモリへのアクセスを減らすことができる．しかしこのタイリングのみの手法では，時間ステップごとに計算をする際に各時間ステップごとで全ての配列にアクセスすることになり，次の時間ステップの計算を行う際にはデータがキャッシュに残っておらず時間的局所性が利用できない問題がある．

2.2 時間ブロッキング

タイリングにおける時間的局所性を改善する方法として時間ブロッキングという手法が広く研究されている [1] [2] [3]．時間ブロッキングとは，領域ごとに時間ステップを数ステップ分まとめて計算することで時間的局所性を利用する手法である．これにより時間的局所性，空間的局所性を利用してメインメモリへのメモリアクセスコストの削減を行うことができる．特に，並列数を上げるほどこの時間ブロッキングは効果が大きくなる。時間ブロッキングはステンシル計算においてとても重要な手法であるが，実装面においてループ構造を記述する際プログラマーにとってとても手間のかかる作業であるという欠点がある．時間ブロッキングにはいくつか種類があるため代表的なものを挙げる．

2.2.1 Overlapped tiling

時間ブロッキングの手法の一つとして本研究で用いる Overlapped tiling について説明する．Overlapped tiling とはタイリングをした領域ごとに任意に定めた一定の時間ステップ分を一度に計算する方法である．このとき定める時間ステップのサイズを時間ブロックサイズと呼ぶ．**図 1** に 1 次元のステンシル計算に対して時間ステップごとにどのような順序で計算するかを示す．Overlapped tiling では各時間ブロックサイズ後の計算に必要なデータの範囲が重複しながら計算を行うため冗長な計算が起こるという欠点がある．しかしメモリアクセスの時間的局所性，空間的局所性が利用されることで全体としての実行時間は減らすことができる．本研究ではこの Overlapped tiling による時間ブロッキングを実装する．

2.2.2 hexagonal tiling

本研究では用いないが，他の例として hexagonal tiling というものがある．hexagonal tiling とは六角形の形で計算していくことにより時間的局所性を利用しつつ，Overlapped tiling のような冗長な計算は起こらずに計算していく手法である．**図 2** に 1 次元のステンシル計算についての図を示す．hexagonal tiling は計算量は増えないが，Overlapped tiling と比較すると空間的局所性が悪くなってしまうといった欠点がある．

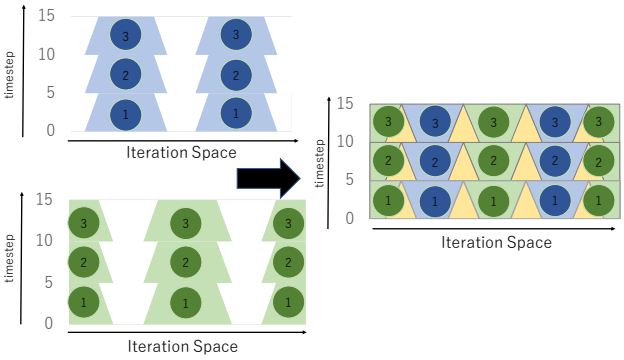

図 1 Overlapped tiling

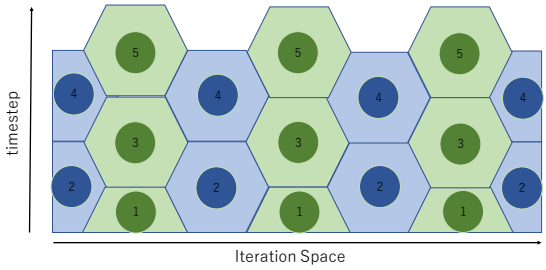

図 2 hexagonal tiling

2.3 Halide

2.3.1 Halide 言語の概要

Halide [6] とは画像処理 [7] や深層学習 [8] などに特化したドメイン特化言語であり，C++プログラムに組み込まれた言語である．Halide はコードを記述する部分が計算のアルゴリズムとスケジュールする部分に分かれているのが大きな特徴である．アルゴリズムの部分には計算の定義のみを記述をして，スケジュールの部分には並列化やループの変更，バッファの確保のしかたなどを記述する．スケジュールにはアーキテクチャごとに記述するということも可能である．Halide では時間ブロッキングの実装を簡潔に記述ができ，アーキテクチャごとにスケジュールを記述できるというのが利点である．

2.3.2 計算アルゴリズムの記述

Halide の実際の記述例について説明する．主要な部分のコード例を**図 3** に示す．Halide には関数の Func 型，変数の Var 型，式の Expr 型というものがある．図 3 の例では 1，2，4 行目でそれぞれの定義を行い，5 行目で計算の定義を記述している．この 5 行目の意味は，仮引数 x, y に対して値 x+y を返すという関数である．この例は x+y を式の値としてとっておきそれを代入するという形で記述しているが，6 行目のように一度にまとめて記述することでも同

```
1   Halide::Func f,g;
2   Halide::Var x, y;
3
4   Halide::Expr e = x + y;
5   f(x, y) = e;
6     //f(x, y) = x + y;
7
8  g(x,y) = f(x,y)+f(x+1,y);
9
10  f.compute_root();
11  //f.compute_at(g,y);
12
13  g.realize(NX, NY);
```

図 3 Halide のコード例

等の意味を持つ．同じように 8 行目は仮引数 x, y に対して値 x+y を返す関数を定義している．

2.3.3 計算スケジュールの記述

10 行目から記述しているのは, アルゴリズムの計算スケジュールについてである．ここに記述されている f.compute_root というのは，g(x,y) の計算を行う際に必要になる部分の f の値を全て先に計算をするという意味である．(同等な C 言語のソースコードを**図 4** に示す．) また，このスケジュール部分に f.compute_at(g,y) と記述をすると，g の y の値 1 つ分を計算するのに必要な f の値を前もって計算するというスケジュールになる．(同等な C 言語のソースコードを**図 5** に示す．) 他にもこの部分にはタイリング，ベクトル化，スレッド並列化などの記述をすることも可能であり，Halide 言語ではスケジュールについての記述をすることで計算の順序を容易に変更することが出来る．最後に 13 行目でここまでのアルゴリズムとスケジュールで計算を実行するサイズを決定して実行する．

2.3.4 auto-scheduler

Halide には自動でスケジュールを決定する auto-scheduler という機能がある．auto-scheduler とは 2.3.3 節で説明したスケジュールの決定を Halide 内部のアルゴリズムで決定してくれるものである．本論文では提案する手法のスケジュールの決定について，この auto-schduler によるスケジューリングを用いる．スケジュールの決定には全ての入出力サイズの情報が必要であるため，スケジュールを記述する代わりに Halide の auto-scheduler に対して入出力サイズの情報を示す記述をする．

3. Halide による時間ブロッキング実装

3.1 計算アルゴリズム

Halide 言語でのステンシル計算の実装を時間ブロッキングを用いて行う．時間ブロックサイズが 5 の場合の主要部分のソースコードを部分的に**図 6** に示す．初めに Func と Var を定義する．入出力に関しては Func とは別のものとして定義を行う必要があるため別途定義を記述している．計算アルゴリズムの初めに入力の境界条件について，12 行目では境界の外側は 0 となるように定義している.13 行目からは実際の時間ブロックサイズが 5 となるように 5 つ関数を連ねることで時間ブロッキングを実装している．ここでは Halide での計算アルゴリズムとしての定義であるため，通常のような逐次計算が行われているわけではない．ここでは省略をしているが，スケジュールには 2 章で説明した auto-schduler を用いているということに注意が必要である．また，この計算アルゴリズムの部分は GPU でも同じアルゴリズムであるため GPU でも全く同じ記述を用いている．この計算アルゴリズムの書き換えが必要ないことによって，コードを記述する際の負担を大きく減らすことができる．

```
1   for(y=0; y<NY; y++){
2     for(x=0; x<NX+1; x++){
3       f[y][x] = x+y;
4     }
5   }
6   for(y=0; y<NY; y++){
7     for(x=0; x<NX; x++){
8       g[y][x] = f[y][x]+f[y][x+1];
9     }
10  }
```

図 4 図 3 で f.compute_root() と記述した場合の動作を示す C 言語コード

```
1  for(y=0; y<NY; y++){
2    for(x=0; x<NX+1; x++){
3      f[0][x] = x+y;
4    }
5
6    for(x=0; x<NX; x++){
7      g[y][x] = f[0][x]+f[0][x+1];
8    }
9  }
```

図 5 図 3 で f.compute_at(g, y) と記述した場合の動作を示す C 言語コード

3.2 CPU でのスケジュール

今回用いる Halide の auto-scheduler には種類があり，その中で本論文では Mullapudi2016，Adams2019 の二つのスケジューラーを用いる．auto-scheduler には実際に Halide のコードでどのようなスケジュールをして計算が行われたかを確認できるものがあり，Mullapudi2016 についてのスケジュールのソースコードを**図 7** に示す．

スケジュールは calc_1，calc_2，calc_3，calc_4，output の 5 つについて記述されているが，calc_1，calc_2，calc_3，calc_4 は同じ記述がされているため記述を省略している．

まず calc 部分のスケジュールについて説明する．compute_at(output,x_o) というのは output で x_o 一つ分について必要な部分の calc を前もって計算するという意味である．x_o というのは output のスケジュールの際にも説明するが，タイル番号の x の変数である．よってこの compute_at では output のタイル 1 枚に必要な計算について calc 部分の計算を行うという意味である．split，vectorize はまとめてベクトル化を行っている．split は x の変数のループ部分を 8 要素ごとの二重ループに変更し，外側の変数を x_vo，内側の変数を x_vi と分け，vectorize で x_vi について SIMD 命令を用いるということである．このスケジュールが calc_1，calc_2，calc_3，calc_4 について全て記述されている．

次に 11 行目からの output のスケジュールについて，12 行目の compute_root は，全ての配列について計算をおこなうという意味である．13，14 行目の split 部分では x，y のそれぞれを 256 要素ごとに分割を行い，15 行目の reorder で変数の順序を内側から順に並べ直している．この部分の記述で配列を 256×256 のタイルサイズで分割していることを示している．16 行目，17 行目は calc と同様にベクトル化を行っている 18 行目の parallel(y_o) では y_o ごとにスレッド並列化を行うということを示している．これらの記述から auto-scheduler で Overlapped tiling を実装しているということがわかる．

3.3 GPU でのスケジュール

また，GPU でのスケジュールについてのソースコードをソースコード**図 8** に示す．1 行目の gpu_tile 関数では CPU の時と同様にタイル化を行うものをこの 1 行で記述しており，タイルサイズを 16×16 としている.2 行目から 5 行目は CPU 同様 output のタイル 1 枚の計算に必要な calc の計算部分のみを計算することを示している．compute_at の記述は CPU と GPU で同じ記述であり，GPU スレッドを用いることを表すために gpu_thread(x,y) というのを全ての calc 関数について記述している．

4. 実験

4.1 実験概要

Halide 言語でステンシル計算の高性能計算の実装を CPU，GPU についてそれぞれ行い，性能について評価した．実験環境を**表 1** に示す．9 点ステンシルについてそれぞれかかった時間を測り考察した．時間ステップ数は 100 に設定し，行列サイズを 4000×4000，8000×8000，16000×16000，32000×32000 のそれぞれについて性能を比較した．1 点を計算するのに必要な演算回数は足し算が 8 回，掛け算が 1 回で合計 9 回であるため，行列サイズ n × n の時の実行時間を T としたとき，Flops は $n^2 \times 9 \times 100/T$ で計算できる．性能評価は GFlops で比較している．スレッド並列化について，CPU については今回用いた CPU の最大コア数の 56 と並列化なしの場合と比較した．Halide での

```
1  #include "Halide.h"
2  #define stencil(f1,f0) f1(x, y) = (f0(x-1,y-1)+
       f0(x,y-1)+f0(x+1,y-1)+f0(x-1,y)+f0(x,y)+f0(x
       +1,y)+f0(x-1,y+1)+f0(x,y+1)+f0(x+1,y+1))/9
3
4  using namespace Halide;
5
6  Func calc_1, calc_2, calc_3, calc_4;
7  Var x, y;
8
9  Input<Buffer<float>> input{"input", 2};
10 Output<Buffer<float>> output{"output", 2};
11
12 Func input_ = BoundaryConditions::
       constant_exterior(input, 0);
13 stencil(calc_1, input_);
14 stencil(calc_2, calc_1);
15 stencil(calc_3, calc_2);
16 stencil(calc_4, calc_3);
17 stencil(output, calc_4);
```

図 6 時間ブロッキングの計算パイプラインの定義部分のコード (時間ブロックサイズが 5 であるとき)

```
1  {
2    Var x = calc_i.args()[0];
3    calc_i
4      .compute_at(output, x_o)
5      .split(x, x_vo, x_vi, 8)
6      .vectorize(x_vi);
7  }
8  {
9    Var x = output.args()[0];
10   Var y = output.args()[1];
11   output
12     .compute_root()
13     .split(x, x_o, x_i, 256)
14     .split(y, y_o, y_i, 256)
15     .reorder(x_i, y_i, x_o, y_o)
16     .split(x_i, x_i_vo, x_i_vi, 8)
17     .vectorize(x_i_vi)
18     .parallel(y_o);
19 }
```

図 7 Mullapudi2016 によるスケジュール

```
1    calc5.GPU_tile(x, y, xo, yo, xi, yi, 16, 16);
2    calc4.compute_at(calc5, xo);
3    calc3.compute_at(calc5, xo);
4    calc2.compute_at(calc5, xo);
5    calc.compute_at(calc5, xo);
6
7    calc.GPU_threads(x, y);
8    calc2.GPU_threads(x, y);
9    calc3.GPU_threads(x, y);
10   calc4.GPU_threads(x, y);
```

図 8 GPU でのスケジュール

表 1　実験環境

CPU	Intel Xeon E5-2680 v4 × 2
CPU メモリ容量	128 GB
CPU コア数	28
CPU スレッド数	56
GPU	Tesla P100-SXM2
GPU メモリ容量	16GB
Halide	version 8.0.0 Mullapudi2016 scheduler(default)
auto-scheduler	version 10.0.0 Adams2019 scheduler
OS	CentOS 7.2

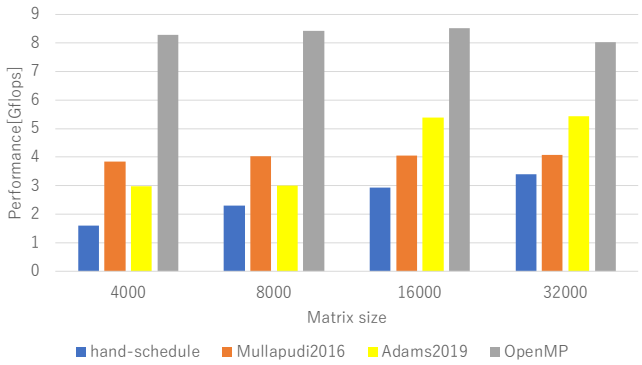

図 9　CPU の各実装での時間ブロッキングなし，スレッド数 1，各行列サイズごとの性能

スレッド数は環境変数 HL_NUM_THREADS を，OpenMP では OMP_NUM_THREADS をそれぞれ変えて測定した．GPU については 1 コアでの性能を比較している．

4.2　CPU での実験結果

はじめに時間ブロッキングを用いない CPU 実装として，Halide の auto-scheduler である Mullapudi2016，Adams2019 の二つについての実装，Halide で tiling を用いた実装 (hand-schedule)，OpenMP での実装について性能を比較した．CPU では時間ブロックサイズ 1，2，5，10，20 について性能を比較した．**図 9**，**図 10** にスレッド数 1 と 56 のそれぞれについての比較を示す．スレッド数 1 の場合図 9 から OpenMP での実装と比べるとでの実装は 46%から 50%程度，Adams2019 での実装は 36%から 68%程度に性能が落ちていることがわかる．また hand-schdule での実装と比べると，Mullapudi2016 での実装は 1.2 倍から 2.4 倍程度，Adams2019 での実装は 1.3 倍から 1.9 倍程度性能が向上していることがわかる．スレッド数 56 の場合図 10 から OpenMP での実装と比べると Mullapudi2016 での実装は 20%から 50%程度，Adams2019 での実装は 44%から 60%程度に性能が落ちていることがわかる．また hand-schdule での実装と比べると，Mullapudi2016 での実装は 63%から 98%程度に性能が落ちており，Adams2019 での実装は 1.0 倍から 1.7 倍程度性能が向上していることがわかる．OpenMP での性能と比較すると性能は及ばないが，hand-schedule と比べると同等かそれ以上に良好なスケジュールが実装できていることがわかる．

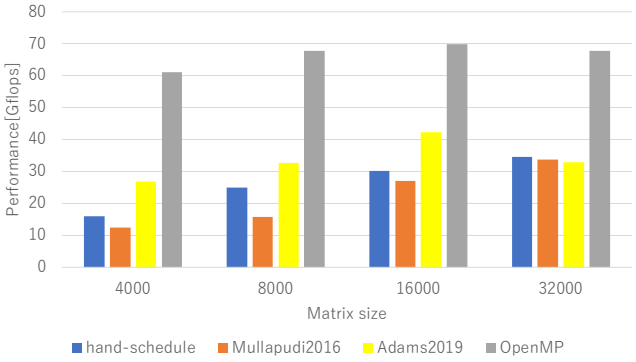

図 10　CPU の各実装での時間ブロッキングなし，スレッド数 56，各行列サイズごとの性能

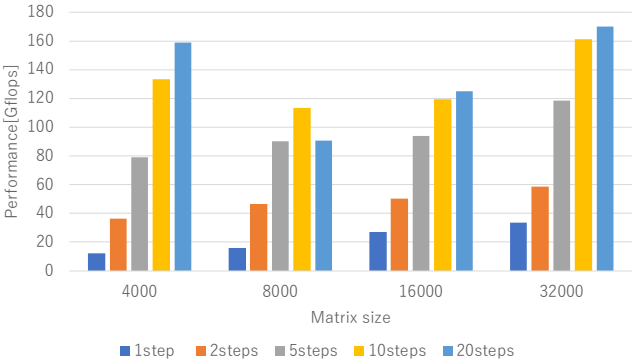

図 11　Mullapudi2016 での時間ブロッキングあり，スレッド数 56，各行列サイズ，各時間ブロックサイズごとの性能

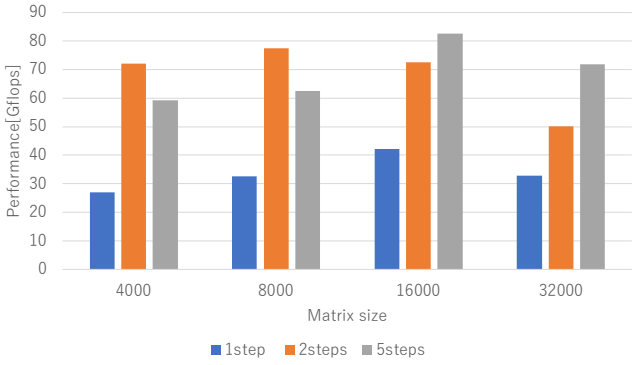

図 12　Adams2019 での時間ブロッキングあり，スレッド数 56，各行列サイズ，各時間ブロックサイズごとの性能

次に時間ブロッキングを用いた CPU 実装として，同様に Mullapudi2016，Adams2019 を用いた実装，OpenMP での実装を比較した．**図 11** に Mullapudi2016 での性能を示す．図から時間ブロックサイズを 1step から 20steps にしたときに 4.6 倍から 12.8 倍程度の性能向上が見られる．同様に**図 12** に Adams2019 での性能を示す．Adams2019 では時間ブロックサイズ 10steps 以上のスケジュールを探索することが不可能であったため，時間ブロックサイズは 5steps までの性能を比較している．図から時間ブロックサイズを 1step から 5steps にしたときに 1.9 倍から 2.2 倍程

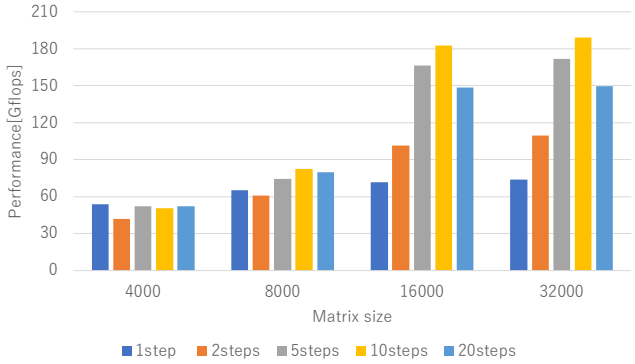

図 13 OpenMP での時間ブロッキングあり，スレッド数 56，各行列サイズ，各時間ブロックサイズごとの性能

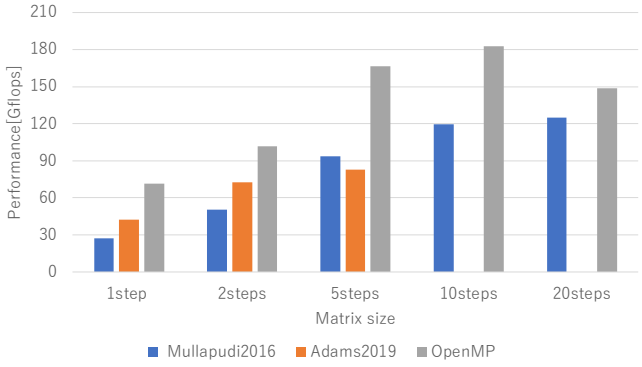

図 14 CPU の各実装での時間ブロッキングあり，スレッド数 56，行列サイズ 16000×16000，各時間ブロックサイズごとの性能

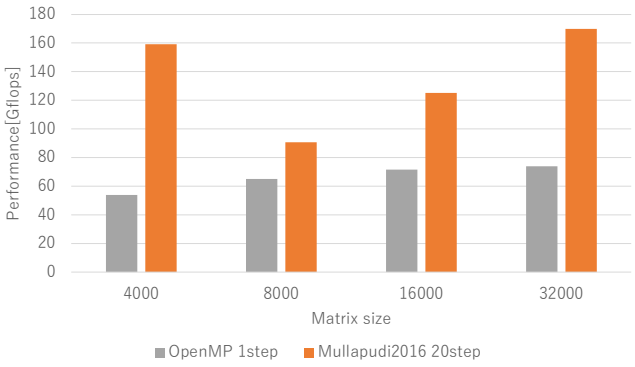

図 15 OpenMP の時間ブロッキングなしと mullapudi で時間ブロックサイズ 20steps での各行列サイズごとの性能

度の性能向上が見られる．よって Halide で時間ブロッキングによって性能向上を達成できたと考えられる．また比較のため**図 13** に OpenMP での時間ブロッキング実装の性能比較を示す．OpenMP ではタイルサイズを 200×200 で分割している．図から 10steps のときに最大になり行列サイズが 16000，32000 では 2.5 倍から 2.6 倍性能が向上していることがわかる．また，**図 14** で行列サイズが 16000 でのそれぞれの比較を示す．この図から，それぞれの時間ブロックサイズごとでは OpenMP を用いた実装の 56% から 84%の性能にとどまっているが，**図 15** に示すように

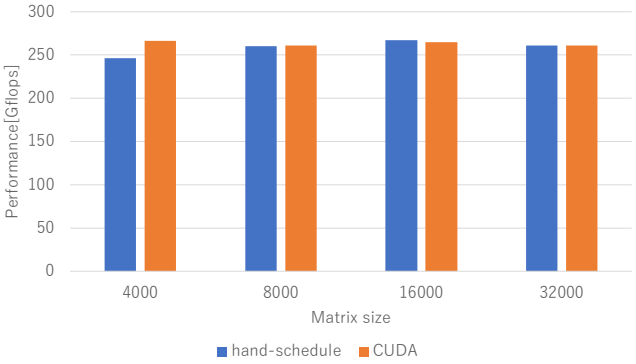

図 16 GPU の各実装での時間ブロッキングなし，各行列サイズごとの性能

OpenMP の時間ブロッキングなしの場合と比較を行うと，Mullapudi2016 での 20steps での時間ブロッキングを用いた性能は 1.3 倍から 2.9 倍の性能向上が見られた．Halide で時間ブロッキングを実装することは，OpenMP での時間ブロッキング実装と比べると比較的容易なため，Halide での時間ブロッキングによる性能が OpenMP の時間ブロッキングなしの性能を上回っているのは非常に有益なことであると言える．

4.3 GPU での実験結果

GPU での実装として，Halide で tiling を用いた実装 (hand-schedule)，CUDA での実装について性能を比較した．GPU については時間ブロックサイズ 1，2，4，5 について実験した．まず時間ブロッキングなしでの性能比較を**図 16** に示す．図から Halide での実装は CUDA での実装と比べて 0.9 倍から 1.0 倍程度の性能であることがわかる．

次に，時間ブロッキングを Halide で実装したものについて**図 17** に示す．図より時間ブロックサイズを 4 にしたときに性能が上がっており，それぞれの行列サイズで 1.9 倍から 2.0 倍性能が向上していることがわかる．このことから GPU の場合でも Halide で時間ブロッキングを実装することで性能が向上することが確認できる．また，比較として CUDA で時間ブロッキングの実装を行なったものを**図 18** に示す．CUDA では時間ブロックサイズを 2 にしたときに性能が向上し，それぞれの行列サイズで 1.1 倍から 1.4 倍性能が向上していることがわかる．最後に，**図 19** に行列サイズを 16000 にしたときのそれぞれの性能について比較したものを示す．図から Halide での実装は CUDA での実装と比べて 0.95 倍から 1.54 倍の性能となっており，Halide での時間ブロッキング実装の有用性を示している．

5. 関連研究

本研究と同様にステンシル計算の Halide 上での実装についての研究がある [9]．Liao らは本研究と同様に Halide でステンシル計算を行う際に最適化手法として Overlapped tiling を用いている．本研究では一般的な Halide のコード

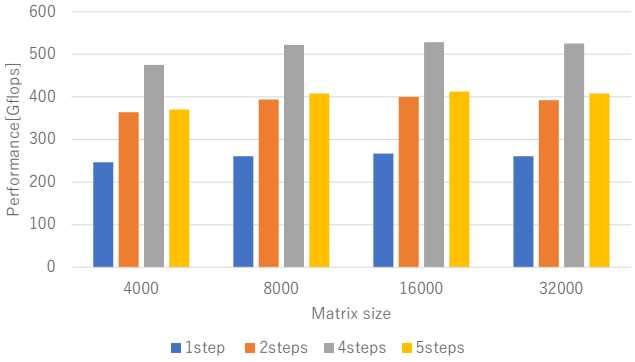

図 17 Halide を用いた hand-schedule での時間ブロッキングあり，各行列サイズ，各時間ブロックサイズごとの性能

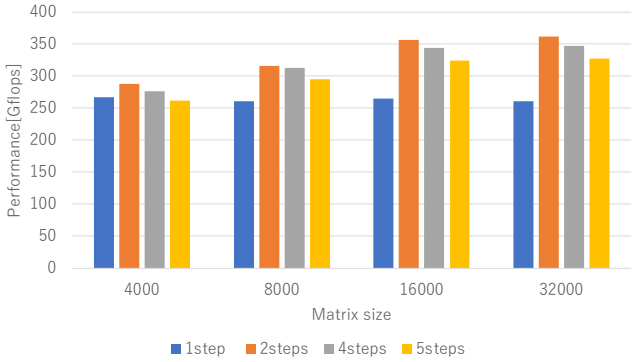

図 18 CUDA での時間ブロッキングあり，各行列サイズ，各時間ブロックサイズごとの性能

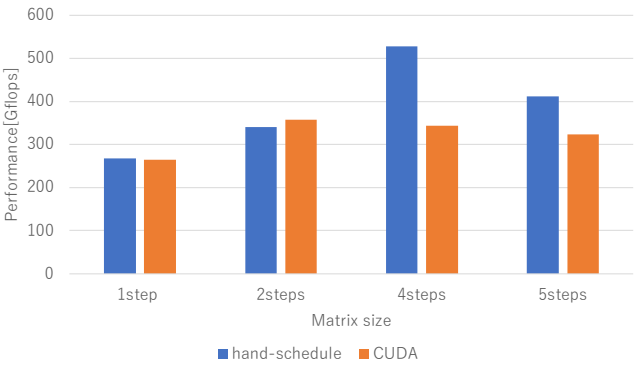

図 19 hand-schedule と CUDA での時間ブロッキングあり，行列サイズ 16000×16000，各時間ブロックサイズごとの性能

に対して良好なスケジュールを決定する auto-scheduler を用いているが，Liao らはステンシル計算に特化して最適なタイルサイズを理論的に計算し探索するスケジューラの提案を行っている．本研究は CPU で 28 コアでの実装，GPU 1 コアでの実装を行なっているが，Liao らは CPU4 コアでの実装を行なっている。

この論文の著者は最適なタイルサイズを求める方法を 2 つのルールによって決定している．まず 1 つ目のルールは (TW1 + TW2) ×num_threads < (0.8 × cache_size) の式を満たすようなタイルサイズにすることである．こ

```
/*** pseudo code***/
1.  for( int tY = output_height / threads ; tY >= 1 ; tY = tY / 2 )
        Heights.add(tY);
2.  for(int tY : Heights){
        for( int tX = 4 ; tX <= output_width ; tX = tX*2 )
            Widths.add(tX);
        int max_TW = 0;
        int picked_tX;
        for(int tX : Widths){
            if(getTopmostW(tH, tW)*threads > (80% * Cache size )
                next;
            int TW = getTW(tH, tW);
            if( TW >=  max_TW){
                max_TW = TW;
                int picked_tX = tX;
            }
        }
        Candidate.add( (tY, picked_tX, max_TW) );
    }
3.  min_WR = ∞
    picked_tile = null
    for( tile(tY, tX, TW): Candidate){
        int tiles_amount = (output.height+tY-1)/tY  *  (output.width+tX-1)/tX;
        int W = TW*tiles_amount;
        int WR = W*100 / baseline_W;  // in percentage form
        if(WR < min_WR){
            min_WR = WR;
            picked_tile = (tY, tX);
        }
    }
```

図 20 タイルサイズ選択のアルゴリズム (Liao らの論文 [9] から引用)

こで，TW1，TW2 は stage1，stage2 の時のタイルのサイズ，num_threads はキャッシュを共有するスレッドの数を表す．この式は stage1 と stage2 のタイルサイズの合計にスレッド数をかけたものが，キャッシュサイズの 80%を超えないようにするといったるルールである．タイルサイズはステージが進むにつれて小さくなっていくため，最初の stage1，stage2 の部分のデータが全てキャッシュ上にのるような条件にしておくと残りの stage は必ずキャッシュサイズの 80%を超えないことになる．

2 つ目のルールは計算を最小限にするためにタイルサイズを最大化する方法についてである．この部分のコードを**図 20** に示す．3 つの部分に分かれているためそれぞれについて説明する．コードの 1 の部分はタイルの高さの候補を作るコードになっている．一番大きいものは出力全体のタイルサイズの高さ/スレッド数でそこから 1 を下回らないものまで 1/2 したものを全て候補としている．コードの 2 の部分は 1 で生成したそれぞれの高さ候補について幅の候補を生成する．幅の候補については一番小さい候補を 4 としてそこから出力全体のタイルサイズの幅を超えないところまで 2 倍していったものを全て候補としている．また 1 つ目のルールであるキャッシュサイズの 80%を超えないという条件をここで判別して最大のタイル幅と高さの組を求める．コードの 3 の部分は仕事率が最小になるペアを選択している．

この論文では y 軸方向のループを並列化し，各ステージでループ x をベクトル化して，ステージごとに実行する方法をベースラインスケジュールと定義している．提案

しているアルゴリズムによって決定したタイルサイズでOverlapped tiling を用いた場合，ベースラインスケジュールと比較すると約 8 倍高速化できたことを成果としている．

6. 結論

本論文では Halide 言語を用いてステンシル計算の高性能実装を行った．メモリアクセスコスト削減のために時間ブロッキングを用いた実装を提案した．Overlapped tiling による時間ブロッキングを Halide で実装し，CPU では Halide での実装によって，OpenMP での時間ブロッキングなしでの実装と比べて 1.3 倍から 2.9 倍性能を向上することができた．また，GPU では Halide での実装によって，CUDA での時間ブロッキングでの性能と比べて，0.95 倍から 1.54 倍の性能を示した．

謝辞 本研究の一部は JSPS 科研費 20H04165 の助成による．

参考文献

[1] Matsumura, K., Zohouri, H. R., Wahib, M., Endo, T. and Matsuoka, S.: AN5D: automated stencil framework for high-degree temporal blocking on GPUs, *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, pp. 199–211 (2020).

[2] 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, *2013 IEEE International Conference on Cluster Computing (CLUSTER)*, IEEE, pp. 1–8 (2013).

[3] Endo, T.: Realizing out-of-core stencil computations using multi-tier memory hierarchy on gpgpu clusters, *2016 IEEE International Conference on Cluster Computing (CLUSTER)*, IEEE, pp. 21–29 (2016).

[4] Adams, A., Ma, K., Anderson, L., Baghdadi, R., Li, T.-M., Gharbi, M., Steiner, B., Johnson, S., Fatahalian, K., Durand, F. et al.: Learning to optimize halide with tree search and random programs, *ACM Transactions on Graphics (TOG)*, Vol. 38, No. 4, pp. 1–12.

[5] Mullapudi, R. T., Adams, A., Sharlet, D., Ragan-Kelley, J. and Fatahalian, K.: Automatically scheduling halide image processing pipelines, *ACM Transactions on Graphics (TOG)*, Vol. 35, No. 4, pp. 1–11 (2016).

[6] Ragan-Kelley, J. M.: Decoupling algorithms from the organization of computation for high performance image processing, PhD Thesis, Massachusetts Institute of Technology (2014).

[7] Ragan-Kelley, J., Barnes, C., Adams, A., Paris, S., Durand, F. and Amarasinghe, S.: Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines, *SIGPLAN Not.*, Vol. 48, No. 6, pp. 519–530 (online), DOI: 10.1145/2499370.2462176 (2013).

[8] Li, T.-M., Gharbi, M., Adams, A., Durand, F. and Ragan-Kelley, J.: Differentiable programming for image processing and deep learning in Halide, *ACM Trans. Graph. (Proc. SIGGRAPH)*, Vol. 37, No. 4, pp. 139:1–139:13 (2018).

[9] Liao, S.-W., Tsai, S.-J., Yang, C.-H. and Lo, C.-K.: Locality-aware scheduling for stencil code in halide, *2016 45th International Conference on Parallel Processing Workshops (ICPPW)*, IEEE, pp. 72–77 (2016).