並列ステンシル計算における通信の自動最適化に向けた性能モデルの評価

河村 知輝[1]　丸山 直也[1,2,3]　松岡 聡[1,3,4]

概要：ステンシル計算を分散メモリ環境で並列化する場合には袖領域通信を行う必要がある．袖領域通信にかかるレイテンシを抑える方法としてテンポラルブロッキングと呼ばれる手法があるが，この手法には計算回数のパラメータが存在するため，最適な計算回数を求める必要が出てくる．本論文ではまずテンポラルブロッキングの効果を検証するために 7 点ステンシルと姫野ベンチマークを対象に最適化を行った結果，最大 21%の性能向上を確認した．さらに，最良計算回数を導出するための性能モデルを構築し，評価を行った．その結果，最適解を導出するためにはステンシル計算の計算量に応じて必要とする精度が異なることを確認した．

Performance Model for Automatic Optimization of Communication in Data-parallel Stencil Computations

Tomoki Kawamura[1]　Naoya Maruyama[1,2,3]　Satoshi Matsuoka[1,3,4]

Abstract: In order to solve stencil computations on distributed memory systems, each node must exchange ghost cells. Temporal blocking, which has a trade-off between the reduction in communication latency and the redundant computation cost by calculating ghost cells, is one of the optimization for stencil computations. Temporal blocking requires choosing the optimal number of calculation per communication. In this paper, we implement temporal blocking in 7-point stencil and Himeno Benchmark, and achieve 1.21x speedup. Moreover, we construct a performance model for selecting the optimal number of calculation. As a result, we confirm that the efficiency of the model depends on computation time of the stencil kernel.

1. はじめに

流体計算を数値計算によって解く際にステンシル計算は頻出する．ステンシル計算は各セルが近傍の値を用いて値を更新していく計算であり，全てのセルの更新を同一の計算規則で表すことができる．また，ダブルバッファリングと呼ばれる手法を用いることで各セルの計算は全て独立に行うことができるため，高い並列性を持つ．一般に，ステンシル計算は演算に対するメモリアクセスの割合（Byte/Flop 値）が高く，メモリバンド幅律速になる傾向にある [1]．このような性質があるため，演算性能が高いだけでなくメモリバンド幅も広い GPU を用いてステンシル計算を解く研究が盛んに行われている [2][3]．

ステンシル計算はセルの更新に近傍の値を使用するので，並列化をした場合にはタイムステップ毎に近傍の値を更新しなければならない．MPI のような分散メモリ並列モデルの場合には通信を行う必要が出てくるため，並列化を行うことでプログラミングコストが増加してしまう．この問題に対して，ステンシル計算を対象とした GPU クラスタ向け自動並列化フレームワーク（Physis）[4][5] が開発されている．Physis フレームワークを用いることで，ステンシル計算の計算規則とデータ配列のみを記述すれば，複数ノード環境で実行できるコードを自動生成することができる．現在，最適化として通信と計算のオーバーラップは行

[1] 東京工業大学
Tokyo Institute of Technology
[2] 理化学研究所
RIKEN AICS
[3] 科学技術振興機構 CREST
JST CREST
[4] 国立情報学研究所
National Institute of Informatics

ⓒ 2012 Information Processing Society of Japan

われているが，ステンシル計算特有の最適化はまだされていない．

本研究では通信コストを抑える既存手法であるテンポラルブロッキング [6][7] を Physis フレームワーク内に実装することを提案する．しかし，テンポラルブロッキングには「1 通信あたりの計算回数」というパラメータが存在し，回数によって性能が上下してしまう．そこで，本論文では計算回数を自動最適化するための準備として，最良となる計算回数を導出するための性能モデルを構築した．Physis フレームワークでは通信と計算のオーバーラップが行われているため，今回のモデルでは通信と計算のオーバーラップとテンポラルブロッキングを組み合わせたコードを対象としている．また，ステンシル計算は計算規則が比較的単純でメモリアクセスも規則性があるため，GPU 計算時間をコードの記述からモデル化している．

7 点ステンシル，姫野ベンチマークを対象に，オーバーラップとテンポラルブロッキングの 2 つの最適化を実装したところ，オーバーラップのみの場合と比べて 21%以上の性能向上が確認された．二つのステンシルカーネルの性能モデルを検証したところ，性能モデルの挙動を近くするためにはカーネルの計算量に応じて GPU 計算モデル式の必要精度が変わることがわかった．

2. 研究背景

2.1 ステンシル計算

近傍の値を用いて計算を行なっていく時間発展型の計算をステンシル計算と呼ぶ．ステンシル計算ではグリッドの各点に行う計算は全て同一の計算式で表すことができ，各点の計算はダブルバッファリングを用いることで全て独立に行うことができるため高い並列性を有している．OpenMP のような共有メモリ並列モデルでは，全てのプロセスがデータ配列の任意の要素を参照できるため，比較的容易に並列化を行うことができる．しかし，分散メモリ環境の場合にデータ配列を単純に等分割してしまうと，各プロセス間の境界に位置するセルを計算する際に毎回通信を行わなければならなくなる．そこで，一般的には各プロセスは担当データ配列だけでなく，隣接した空間のデータも一部保持する形をとる．このような，計算自体は行わない隣接空間の領域を袖領域と呼ぶ．袖領域は計算を行う前に最新の値に更新する必要があるため，通常は計算を行う度に隣接通信を行わなければならない．

2.2 テンポラルブロッキング

袖領域通信のコストを抑える最適化としてテンポラルブロッキングと呼ばれる手法がある．袖領域は各プロセスの境界位置にあるセルが計算を行えるようにするための領域なので，計算カーネルが必要とする近傍幅だけあればよい．しかし，**図 1** のように 1 回の通信で送る袖領域の量を増やし，袖領域も計算することで，1 回の通信に対して複数回分の計算を行うことができる．これにより通信するデータ量は変化せずに，総通信回数を減らすことができるようになるため，通信レイテンシコストを抑えることができる．一方，テンポラルブロッキングを行うと計算の総回数は変化しないが，本来は計算をする必要がない袖領域を計算しなければならないため計算コストは増加してしまう．

このように，1 通信あたりの計算回数を変化させると通信レイテンシ減少量と計算コスト増加量が変化するため，テンポラルブロッキングを適用する場合には最適な計算回数を求める必要性が出てくる．

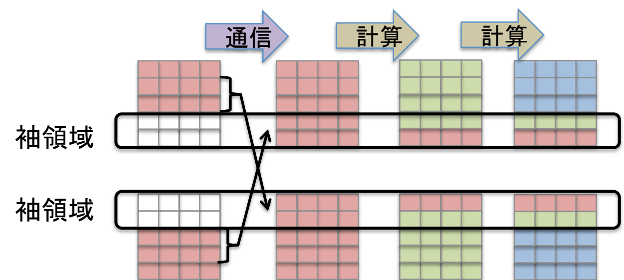

図 1 1 通信あたりの計算回数が 2 回のテンポラルブロッキング．

2.3 通信と計算のオーバーラップ

袖領域は境界位置のセルが計算を行うために必要な領域である．そのため，各プロセスが担当する計算領域を，計算を行う際に袖領域を必要とする領域（境界領域）と袖領域を必要としない領域（内部領域）の二つに分けることで袖領域通信と内部領域計算が独立化できる．

3. 性能モデルの構築

テンポラルブロッキングの最適な計算回数を導出するために性能モデルを構築する．

3.1 計算手順

通信と計算のオーバーラップとテンポラルブロッキングを組み合わせた場合のプログラム概要について述べる．

テンポラルブロッキングは 1 回の通信で複数回の計算を行う手法であるが，通信と計算のオーバーラップを行うために計算領域を二つに分けると，二つの領域でそれぞれ複数回の計算を行う必要が出てくる．そのため，通信と計算のオーバーラップとテンポラルブロッキングを同時に行うと，**図 2** のような流れになる．また，今回は GPU を用いた実装を行なっているため，通信をする際に CPU と GPU の通信を行う必要がある．CPU-GPU 間の通信と MPI による CPU 間の通信は非同期通信を行うことにより 2 つの通信をオーバーラップすることも可能ではあるが，今回の実装では行なっていない．そのため，GPUtoCPU 通信，MPI 通信，CPUtoGPU 通信は全て依存性があることになる．

ⓒ 2012 Information Processing Society of Japan

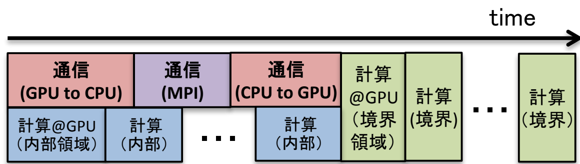

図 2 オーバーラップ+テンポラルブロッキングの流れ.

図 2 のような，1 回の通信と複数回の計算を行う流れを本論文では 1 ステップと呼ぶ．計算する領域が 1 つであれば，二つのバッファを用意することで 1 ステップ前の計算結果の入っている A_{in} と，そのステップで行った計算結果を入れる A_{out} の二つを用意すれば各セルの計算は独立に行うことができた．しかし，二つの領域が複数回の計算を行うためには，それぞれの計算開始時に 1 ステップ前のデータをバッファとして確保しなければならない．この問題を解決する方法の 1 つに計算を行う度に新しいバッファに書き込んでいく方法がある．この方法では 1 通信あたりの計算回数に比例してメモリ使用量が増えていく．現在の GPU はメモリ量が CPU ほど大きくないため，搭載されているメモリ量に応じて計算回数の上限が決まってしまう．そこで，今回の実装ではメモリ使用量を抑えるために，バッファを 3 つ用意して使いまわすトリプルバッファリングを採用した．トリプルバッファリングにおけるバッファの使い方は**図 3** と**図 4** のようになる．

```
for ( cur = 1; cur <= calcMax; cur++){
   // 最初の計算は Aout に書き込む
   if (cur == 1){
      kernel(Ain, Aout);
   }
   // Atmp から Aout へのダブルバッファリング
   else{
      swap(Aout, Atmp);
      kernel(Atmp, Aout);
   }
}
```

図 3 内部領域計算のトリプルバッファリング.

```
for ( cur = 1; cur <= calcMax; cur++){
   // Ain から Atmp へのダブルバッファリング
   if (cur < calcMax){
      kernel(Ain, Atmp);
      swap(Ain, Atmp);
   }
   // 最後の計算は Aout に書き込む
   else{
      kernel(Ain, Aout);
   }
}
```

図 4 境界領域計算のトリプルバッファリング.

3.2 性能モデルの構築

図 2 で示した 1 ステップの流れをもとに構築した性能モデルの詳細について述べる．

3.2.1 モデル全体

1 通信あたりの計算回数を k 回，GPU から CPU への通信時間を T_{G2C}，MPI による CPU 間の通信時間を T_{MPI}，CPU から GPU への通信時間を T_{C2G}，k 回の内部領域計算時間を T_{inner}， k 回の境界領域計算時間を $T_{boundary}$ とする．袖領域通信と内部領域計算の間でオーバーラップをするため，通信と通信のどちらか大きい方の値を取ることになる．以上より，1 ステップあたりの実行時間 t_{step} は次のように表すことができる．

$$\max(T_{inner}, T_{G2C} + T_{MPI} + T_{C2G}) + T_{boundary} \quad (1)$$

さらに，テンポラルブロッキングを行わないオリジナルコードでの反復回数を $loop$ 回とすれば，次のような k に関する最小化問題になる．

$$\text{Minimize}: \frac{loop}{k} \times t_{step} \quad (2)$$

3.2.2 各フェーズの実行時間

ステンシル計算は各セルに行う計算を全て同一の式で書くことができ，メモリアクセスも規則的であるためコードを書いた段階で事前に性能予測できると考えられる．

まず，使用する GPU の実効 FLOPS 値を $F_{machine}$，実効メモリバンド幅値を $B_{machine}$ とする．計算カーネルの計算規則から数え上げた演算数を $flop$，要求されるメモリ量（Byte）を $byte$ とする．ただし，アーキテクチャによってキャッシュが搭載されているものがあり，要求 Byte 値を求める場合にキャッシュに入っているかなども考慮するものとする．さらに，演算とメモリアクセスはオーバーラップされるため，1 セルの計算にかかる時間は，

$$\max\left(\frac{flop}{F_{machine}}, \frac{byte}{B_{machine}}\right) \quad (3)$$

全体の問題サイズを $datasize$，GPU のカーネル起動コストを $const$ とすれば，GPU の計算にかかる時間は次のように表すことができる．

$$\max\left(\frac{flop}{F_{machine}}, \frac{byte}{B_{machine}}\right) \times datasize + const \quad (4)$$

一方，通信時間である T_{G2C}，T_{MPI}，T_{C2G} については現在の段階ではモデル式を作らず，実行環境マシンにて別途でデータサイズ別に測定したものを使用している．

4. 性能評価

通信と計算のオーバーラップとテンポラルブロッキングを組み合わせた最適化の性能評価と，構築した性能モデルの評価を行うために実験を行う．

4.1 実験環境

4.1.1 実験環境マシン

東京工業大学学術国際情報センター（GSIC）のスーパー

© 2012 Information Processing Society of Japan

コンピュータ TSUBAME2.0 を使用した．TSUBAME2.0 は Thin ノード，Medium ノード，Fat ノードの 3 種類を利用することができるが今回は Thin ノードを利用している．Thin ノードは，ノード内に Intel 社製 Xeon X5670 が 2 ソケット，NVIDIA 社製 Tesla M2050 の GPU が 3 ソケット搭載されている．また，各計算ノード間の Infiniband は QDRx4 にて接続されている．今回，MPI を用いた並列実行を行う際には，1 ノードに 1 プロセスを割り当てており，ノードあたりに CPU と GPU は 1 つずつ使用している．なお，MPI は OpenMPI の 1.4.2，CUDA ドライバは 4.1，CUDA ランタイムは 4.1，CUDA Capability は 2.0 を使用している．実装を行う際にシェアードメモリを使っていないため，L1 キャッシュ/シェアードメモリの割り当ては 48KB/16KB に設定している．

性能モデルの GPU 計算の理論値を求めるために GPU の実効値を求める必要があるため，TSUBAME2.0 上に搭載されている Tesla M2050 にて計測を行ったところ，実効 Flops 値は 1000[GFlops], 実効メモリバンド幅は 120[GB/s] となった．また，GPU のカーネル起動コストも計測したところ 12[us] となった．

MPI によってデータ並列を行う際には，Z 軸方向 1 次元分割を行っている．そのため，並列数が 3 以上になると Z 軸正負の方向に袖領域が発生する．各領域内のデータは連続領域ではあるが，2 つの袖領域自体は連続していない．そのため今回，CPU-GPU 通信では 2 回の通信を行い、MPI 通信では各プロセスは先に非同期の MPI_Irecv を 2 方向に発行したのち，各方向に MPI_Isend を発行する形で実装している．また，テンポラルブロッキングは通信レイテンシを抑える手法であるため，本論文では Strong Scaling 性能の評価を行う．

4.1.2 対象とするステンシルアプリケーション

7 点ステンシルと姫野ベンチマーク [8]（以下，姫野 BMT とする）の二つのステンシル計算にて評価を行う．7 点ステンシルはサンプルとして作成したコードとなっており， 各点の係数はメモリを使用しない定数としている．姫野 BMT は非圧縮流体解析コードの性能評価を目的として理化学研究所の姫野龍太郎氏により作られたものである．姫野 BMT には計算サイズとして S（$128 \times 64 \times 64$），M（$256 \times 128 \times 128$），L（$512 \times 256 \times 256$），XL（$1024 \times 512 \times 512$）が用意されている．通信データ量を小さくしたほうが通信レイテンシコストの影響が大きいため，Z 軸に大きいサイズを割り振っている．

次に，7 点ステンシルの GPU 計算モデル式について考える．7 点ステンシルの計算規則は**図 5** のようになっている．7 点ステンシルを CUDA で実装するにあたり，XY 平面 2 次元分割にてスレッドブロックを割り当てているため，1 スレッドは Z 軸の幅分だけ計算を担当することになる．

まずコードから推測されるメモリ要求回数を算出する．

```
v2[i][j][k] = a1*v1[i][j][k]
            + a2*v1[i-1][j][k] + a3*v1[i+1][j][k]
            + a4*v1[i][j-1][k] + a5*v1[i][j+1][k]
            + a6*v1[i][j][k-1] + a7*v1[i][j][k+1];
```

図 5 7 点ステンシルの計算カーネル

```
s0 = a0[i][j][k] * p[i+1][j][k]
    + a1[i][j][k] * p[i][j+1][k]
    + a2[i][j][k] * p[i][j][k+1]
    + b0[i][j][k] * ( p[i+1][j+1][k] - p[i+1][j-1][k]
                     - p[i-1][j+1][k] + p[i-1][j-1][k] )
    + b1[i][j][k] * ( p[i][j+1][k+1] - p[i][j-1][k+1]
                     - p[i][j+1][k-1] + p[i][j-1][k-1] )
    + b2[i][j][k] * ( p[i+1][j][k+1] - p[i-1][j][k+1]
                     - p[i+1][j][k-1] + p[i-1][j][k-1] )
   + c0[i][j][k] * p[i-1][j][k]
   + c1[i][j][k] * p[i][j-1][k]
   + c2[i][j][k] * p[i][j][k-1]
   + wrk[i][j][k];
ss = (s0 * a3[i][j][k] - p[i][j][k]) * bnd[i][j][k];
wrk2[i][j][k] = p[i][j][k] + omega * ss;
```

図 6 姫野 BMT の計算カーネル

各点にかかる係数は全てメモリからの読み込みを必要としない点，今回実験で使用する GPU にはキャッシュが搭載されている点を考慮すると，v1[i][j][k], v1[i-1][j][k], v1[i+1][j][k], v1[i][j-1][k]，v1[i][j+1][k] はキャッシュに入っていると仮定できる．また，Z 軸方向の v1[i][j][k] と v1[i][j][k] は今回レジスタを使用しループで使いまわしているため，メモリバンド幅は消費しないと仮定できる．よって，読み込み側の配列 v1 のメモリ要求数は 1 回となり，書き込み側の v2[i][j][k] と合わせると，コードから推測できるメモリ要求数は 2 回となり，データ配列は単精度を用いているため要求 Byte 値は $4 \times 2 = 8$ となる．計算規則からは演算数が 13 となるが，スレッドが計算するセルを変更するときにインデックス計算を行うために 5 回の和算を要する．そのため，コードから算出できる演算数は 18 となる．

次に姫野 BMT の GPU 計算モデル式について考える．姫野 BMT の計算規則は**図 6** のようになっている．7 点ステンシルと異なり，1 スレッドが 1 セルを担当するナイーブな実装をしている．

読み込み側の配列である p は全部で 19 点読み込むことになる．X 軸方向に連続となっているデータは同一のキャッシュラインに入ると推測できる．Y 軸，Z 軸方向に関してはキャッシュが効かないと仮定する．以上より，読み込み側の配列に対するメモリアクセス要求回数は 9 となり，係数配列の分も合わせると 22 回となる．計算規則から演算数は 23 となり，スレッドのインデックス計算での演算数が 5 であるため，コードから算出できる演算数は 28 となる．

4.2 二つの最適化を組み合わせたコードの性能評価

オーバーラップとテンポラルブロッキングの最適化コー

ドの性能評価を行った．MPI によるデータ分割は Z 軸方向 1 次元分割の Strong Scaling にて計測を行なっている．各グラフは 1 通信あたりの計算回数を表し，計算回数が 1 回のグラフはオーバーラップのみの最適化を表すことになる．今回の計測ではステンシル計算の反復回数を 3000 回前後行なっている．グラフの縦軸には 1 イテレーションあたりの実行時間と用いている．また，計測はそれぞれ 10 回行い，最小の値を使用している．

図 7 は 7 点ステンシルを対象に，問題サイズ 256 × 256 × 256 にて計測した結果となっている．プロセス数が 2 のときには計算回数を増やすと性能が悪化し，プロセス数が 4 以上のときには性能向上することが確認できる．性能向上しているプロセス 4 以上において 8.7%から 12.3%の性能向上となった．1 通信あたりの計算回数が 2 回であればレイテンシは 1/2，計算回数が 3 回であれば 1/3 のようになっていくため，計算回数が 2 回のときに大きく減少していることが説明できる．

一方，プロセス数が 2 のときには計算回数増加に従い性能が悪化してしまっている．テンポラルブロッキングは通信のコストを下げる代わりに計算コストが増加する最適化であるが，プロセス数が 2 のときには通信が計算に隠蔽されてしまい，計算コストのみが反映されてしまっているのではないかと考えられる．そこで，内部領域計算時間，袖領域通信時間をそれぞれ計測しグラフにしたものが**図 8** になる．これより，プロセス数が 2 のときには通信が隠蔽されており，プロセス数が 4 以上のときには内部計算が隠蔽されていると確認できる．また，オーバーラップを行なっている箇所の時間も計測したところ，図 8 よりほぼ完全にオーバーラップされていることがわかる．そのため，今回構築したモデルにおいてオーバーラップを表す MAX 関数が正しいことになる．

また，図 8 では，Strong scaling にも関わらずプロセス数が 4 以降では実行時間が減っていないが，今回の実装した 1 次元分割では並列数を増加させても通信データ量は変化しないことによるものとなる．

図 9 は姫野 BMT を対象に，問題サイズ 128 × 128 × 256 にて計測した結果となっている．プロセス数が 16 以降においては計算回数が 3 回のときに最良の値となり，13%から 21%の性能向上となった．通信データ量が減ることで，通信レイテンシの影響が大きくなり，姫野 BMT のように計算量が大きい場合でも性能向上を期待できることがわかる．

4.3 性能モデルの検証

オーバーラップにより通信律速となるか，計算律速になるかによってテンポラルブロッキングで 1 通信あたりの計算回数を増加させたときに次の 2 通りの挙動がある．

(a) 計算律速では，レイテンシ減少が隠蔽されてしまうため常に性能悪化．

(b) 通信律速では，レイテンシ減少と計算コスト増加のトレードオフが発生．

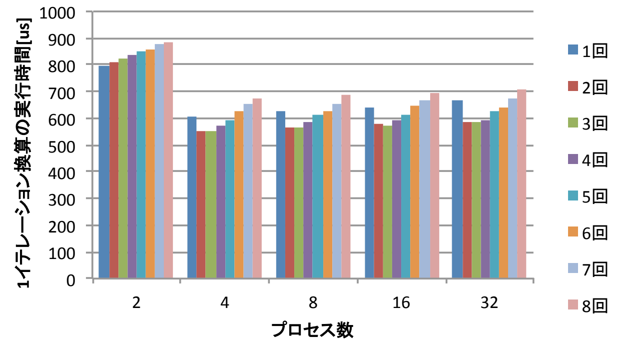

図 7 7 点ステンシルを Strong Scaling で計測．全体の問題サイズは 256x256x256．グラフはそれぞれ 1 通信あたりの計算回数を表す．

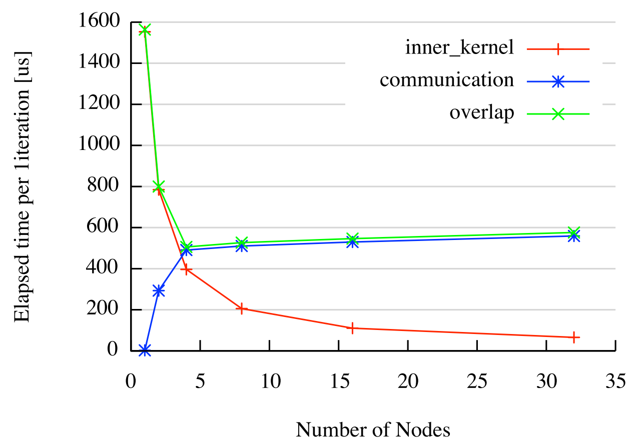

図 8 7 点ステンシルでの内部領域計算時間と袖領域通信の実行時間．1 通信あたりの計算回数は 1 回としている．

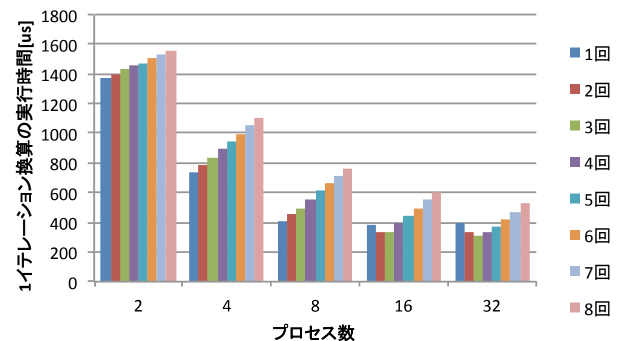

図 9 姫野 BMT を Strong Scaling で計測．全体の問題サイズは 128x128x256．グラフはそれぞれ 1 通信あたりの計算回数を表す．

4.3.1 7 点ステンシルでの性能モデル検証

(a) について検証するために，プロセス数は 2 を選択した．性能モデルによる理論値（Model）と実測値（Measured）

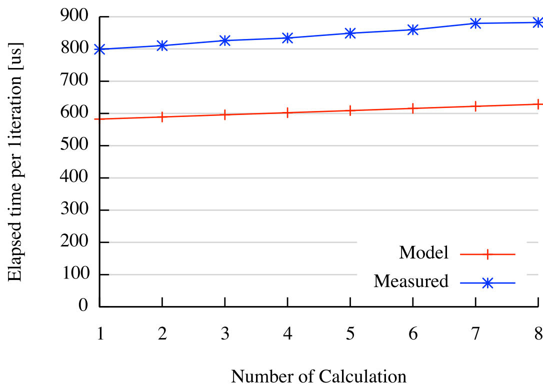

図 10　7 点ステンシルでの性能モデル評価（プロセス数 2）．オーバーラップにより計算律速となっている場合

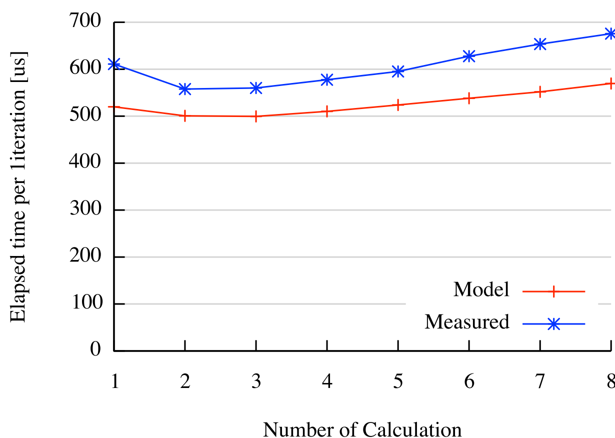

図 11　7 点ステンシルでの性能モデル評価（プロセス数 4）．通信律速となっている場合．

を比較したものが**図 10** となる．二つのグラフで 200[us] 程度の絶対誤差が生じている．内部領域計算部分を対象に実測値とモデル式による理論値を別途計測したところ，同じく 200[us] 程度の絶対誤差が生じていたため，図 10 における誤差の原因は GPU 計算モデルの精度が低かったためと考えられる．

(b) について検証するために，プロセス数 4 にてグラフ化したものが**図 11** となる．性能モデルと実測値の両者にて最適計算回数が 3 回となり，グラフの挙動は比較的近いものとなった．グラフの挙動に関係するのは計算コストや通信レイテンシの変化量であるため，GPU 計算のモデルと実測値との間に絶対誤差が生じていても影響が小さかったと考えられる．

4.3.2　姫野 BMT での性能モデル検証

(a) について検証するために，プロセス数は 8 を選択し，理論値と実測値を比較したものが**図 12** となる．計算回数が 1 回のときにはほぼ理論値と実測値が一致している．7 点ステンシルと同様に別途 GPU 計算のモデルと実測値を計測したところ，姫野 BMT においてもグラフの誤差の原因は GPU 計算モデルの影響であった．また，計算回数が

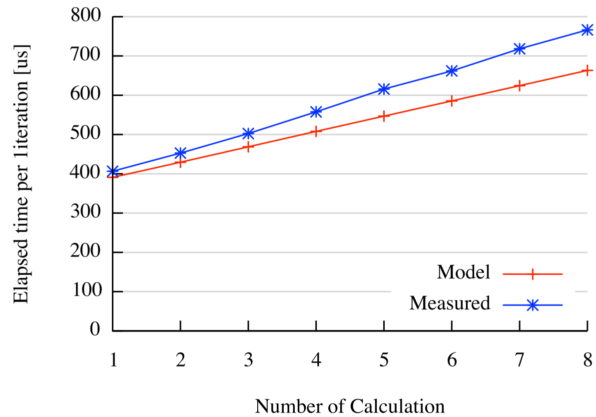

図 12　姫野 BMT での性能モデル評価（プロセス数 8）．計算律速となっている場合．

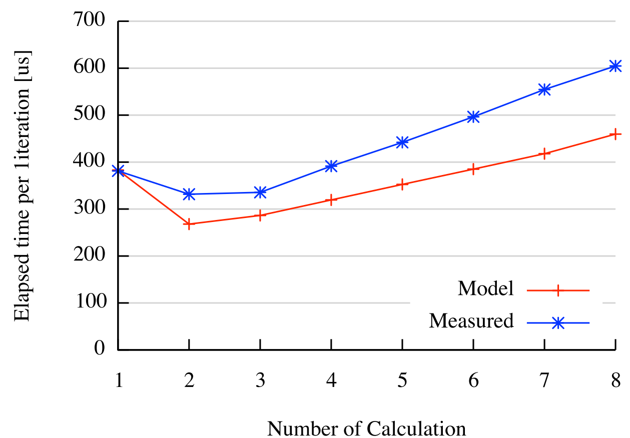

図 13　姫野 BMT での性能モデル評価（プロセス数 16）．通信律速となっている場合．

増加するにつれて誤差は大きくなっている．

(b) を検証するために，プロセス数を 16 としグラフにしたものが**図 13** となる．性能モデルでは最適値が 2 回，実測値では 2 回,3 回付近となった．(a) と同様に，計算回数増加につれて誤差が大きくなっていくことがわかる．

4.3.3　性能モデル評価

7 点ステンシルと姫野 BMT の結果から，最適な計算回数を求める際には実行時間の絶対誤差はあまり重要ではなく，計算回数によって誤差が大きくなっていくかどうかが重要であることがわかる．そこで，GPU 計算モデル式により算出した理論値とデータサイズを変えて計測した実測値をグラフにしたものが**図 14** になる．今回 Z 軸分割で行なっているため，テンポラルブロッキングにて 1 通信あたりの計算回数を変化を変化させると XY 平面は変化せず，Z 軸の大きさだけ変化するため横軸には Z 軸の幅を設定している．実行時間に注目すると，7 点ステンシルは精度が低く，姫野 BMT は精度が高いと言える．しかし，最適な計算回数を求めるためにはデータ量を変化させたときの実行時間の変化量，つまり図 14 での傾きが重要となる．ま

ⓒ 2012 Information Processing Society of Japan

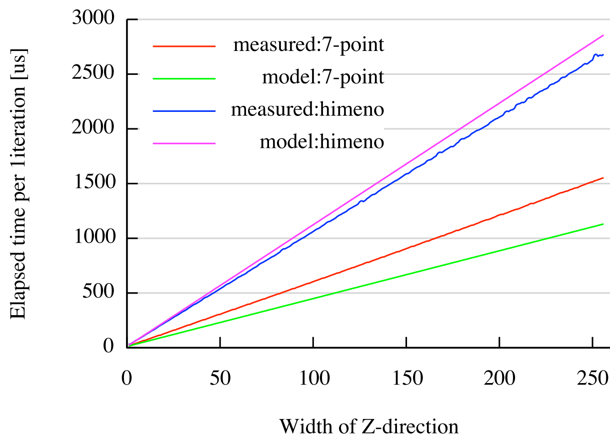

図 14 GPU 計算モデル式と実測値の比較．7 点ステンシルは 256x256xZ．姫野 BMT は 128x128xZ．

た，この傾きはステンシル計算のカーネルの計算量に依存する．

以上より，最適な計算回数を求めるために構築した性能モデルに対して次のことが言える．カーネルの計算量が小さい場合，実行時間の変化量も小さくなるため，GPU 計算モデルの精度が低い場合にも通信律速時の挙動は近いものとなる．対して，カーネルの計算量が大きい場合には GPU 計算モデルの精度が高い必要がある．

5. 関連研究

多くの科学技術計算に頻出するステンシル計算を最適化する研究は多くなされている．特に，ステンシル計算はメモリバンド幅律速であることが多いことよりメモリアクセスを減らすためにキャッシュを効果的に利用するための手法が多い．Meng らの研究 [9] では，共有メモリ環境においてキャッシュブロッキングを行う際に生まれる袖領域の最適化の性能予測を行うために，NVIDIA の Tesla アーキテクチャを対象に性能モデルを構築している．Nguyen らの研究 [10] では，キャッシュヒット率を高める分割方法と，メモリアクセス回数を減らすことができる手法を組み合わせた 3.5D ブロッキング手法を提案している．

プログラムコードから性能解析を行う研究として南らの研究 [11] がある．「京」コンピュータ上にて疎行列とベクトルの積を行うコードを対象にコーディングによる性能予測の方法を提示しており，実測値により比較し有効性を確認している．また，複数の計算カーネルを対象に B/F 値に基づくチューニングも行われている．

6. まとめと今後の課題

本論文では，7 点ステンシルと姫野 BMT を対象に， 並列計算でよく用いられる通信と計算のオーバーラップと，ステンシル計算の最適化手法であるテンポラルブロッキングの二つの最適化を実装し性能評価を行った．その結果，通信データ量にも依存するが 7 点ステンシルでは最大で 12%，姫野 BMT では 21%の性能向上を確認した．さらに，テンポラルブロッキングの最適な計算回数を求めるために性能モデルを構築した．その際，GPU を用いた計算部分に関しては実験マシンの演算性能とメモリバンド幅値，計算カーネルの計算規則と CUDA の実装方法から実行時間の予測を行った．手動最適化コードによりモデルの評価を行ったところ，扱うステンシル計算の計算量が小さい場合には，GPU 計算のモデル式の精度の影響をあまり受けずに最適計算回数を導出することが可能であった．一方，計算量が大きい場合には，最適計算回数を導出するためには GPU 計算モデル式の精度が重要であることがわかった．

今後の課題として，まずは本研究の提案である Physis フレームワークにテンポラルブロッキングを実装することが挙げられる．今回構築した性能モデルでは GPU 計算はモデル式を作成したが，通信は別途計測した値を用いているため，通信に対してもモデル式を立てることも挙げられる．通信時間は 1 ノードに 1 プロセス割り当てでも実行時間にばらつきが生じ，TSUBAME2.0 でも可能な 1 ノードに複数プロセス割り当てる環境での通信時間の挙動も調査する必要がある．

参考文献

[1] S. Kamil, Cy Chan, L. Oliker, J. Shalf, and S. Williams. An auto-tuning framework for parallel multicore stencil computations. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pp. 1 –12, april 2010.

[2] Jose Maria Cecilia, Jose Manuel Garcia, and Manuel Ujaldon. CUDA 2D Stencil Computations for the Jacobi Method. In *State of the Art in Scientific and Parallel Computing*, 2010.

[3] Takashi Shimokawabe, Takayuki Aoki, Tomohiro Takaki, Toshio Endo, Akinori Yamanaka, Naoya Maruyama, Akira Nukada, and Satoshi Matsuoka. Peta-scale phase-field simulation for dendritic solidification on the TSUBAME 2.0 supercomputer. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pp. 3:1–3:11, New York, NY, USA, 2011. ACM.

[4] Naoya Maruyama, Tatsuo Nomura, Kento Sato, and Satoshi Matsuoka. Physis: an implicitly parallel programming model for stencil computations on large-scale GPU-accelerated supercomputers. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pp. 11:1–11:12, New York, NY, USA, 2011. ACM.

[5] 野村達雄, 丸山直也, 遠藤敏夫, 松岡聡. ステンシル計算を対象とした大規模 GPU クラスタ向け自動並列化フレームワーク. 情報処理学会研究報告. 計算機アーキテクチャ研究会報告, Vol. 2010, No. 7, pp. 1–9, 2010-12-09.

[6] M. Wittmann, G. Hager, and G. Wellein. Multicore-aware parallel temporal blocking of stencil codes for shared and distributed memory. In *Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pp. 1 –7, april

2010.
[7] 深沢圭一郎, 梅田隆行, 南里豪志. 超並列惑星磁気圏電磁流体シミュレーションに向けた隣接通信の効率化. ハイパフォーマンスコンピューティングと計算科学シンポジウム論文集, 第 2012 巻, pp. 101–106, jan 2012.
[8] 姫野ベンチマーク. http://accc.riken.jp/HPC/HimenoBMT.html.
[9] Jiayuan Meng and Kevin Skadron. Performance modeling and automatic ghost zone optimization for iterative stencil loops on GPUs. In *Proceedings of the 23rd international conference on Supercomputing*, ICS '09, pp. 256–265, New York, NY, USA, 2009. ACM.
[10] Anthony Nguyen, Nadathur Satish, Jatin Chhugani, Changkyu Kim, and Pradeep Dubey. 3.5-D Blocking Optimization for Stencil Computations on Modern CPUs and GPUs. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pp. 1–13, Washington, DC, USA, 2010. IEEE Computer Society.
[11] 南一生, 井上俊介, 堤重信, 前田拓人, 長谷川幸弘, 黒田明義, 寺井優晃, 横川三津夫.
「京」コンピュータにおける疎行列とベクトル積の性能チューニングと性能評価. ハイパフォーマンスコンピューティングと計算科学シンポジウム論文集, 第 2012 巻, pp. 23–31, jan 2012.

ⓒ 2012 Information Processing Society of Japan