Vol.2014-HPC-143 No.32
2014/3/4

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

河村 知輝[1]　丸山 直也[2,1,3]　松岡 聡[1,3,4]

概要：偏微分方程式を解く際に差分法を用いるとステンシル計算に帰着する．この計算は高いメモリバンド幅を要求するため GPU を用いることで高速化が可能である．しかし GPU メモリ容量は小さく，大規模な問題を解く際に GPU メモリ容量が制限となってしまう．この問題に対して，テンポラルブロッキングを行うことで性能低下なく GPU メモリ容量以上の大きなドメインを解くことができることを示す先行研究があるが，プログラミングコストが高いという問題を抱えている．そこで，本研究ではこのテンポラルブロッキングをフレームワークに組み込むことで自動最適化を実現した．また，ブロッキング段数などのパラメータの最適値を導出するために性能モデルを構築した．

1. はじめに

ステンシル計算は流体や熱シミュレーション [1][2] などに頻出する科学技術計算において重要なカーネルの一つである．近傍の値を用いて時間発展していく計算であり，この計算式は全てのセルに対して同一の式で書くことができる．ステンシル計算の多くはメモリバンド幅律速であるため，高いメモリバンド幅を持つ GPU を用いることで高速化できることは現在では広く知られている．

大規模なシミュレーションを行う際に，シミュレーション範囲を拡大や高精度な計算を行うためには格子点数を増加させる必要があり，それに伴い必要なメモリ容量も増加していく．そのため，大規模なステンシル計算を行うためにはメモリバンド幅の値だけでなく，メモリ容量も必要となってくる．しかし，GPU のメモリ容量は現在の製品では 3GB〜12GB 程度しかなく，一般的に CPU よりもメモリ容量が小さい．東京工業大学のスーパーコンピュータ TSUBAME2.5 では 1 ノードあたりに CPU メモリは 54GB あるのに対して，GPU は NVIDIA Tesla K20X の 6GB が 3 基で合計 18GB しかない．GPU メモリサイズを超える問題を扱うためには，対象となる問題を 1GPU に収まる程度まで分割し，大量の GPU を用いることが一般的ではあるが，問題サイズが GPU メモリ容量依存となってしまう．

GPU メモリ容量を超える問題を GPU で解く方法として，GPU メモリサイズ以下のサブドメインに分割をし，各サブドメインを順番に GPU で計算を行う方法もある．この手法では PCI-Express 通信が頻繁に発生してしまい，実行性能が PCI-Express ネットワークバンド幅律速となってしまう．この問題に対して PCI-Express 通信を対象とするテンポラルブロッキングと呼ばれる手法を用いることで，GPU メモリ容量以上の問題に対して性能劣化なしに 1GPU で解くことができることを示した Jin らの研究 [3][4] がある．しかし，この手法を適用するにはプログラミングコストが高いという問題がある．

そこで，本論文ではこの PCI-Express 通信を対象とするテンポラルブロッキングの自動最適化を提案する．本研究では自動化を実現するためにステンシル DSL である Physis フレームワーク [5][6] を用いて，トランスレータによる変換先の 1 つとしてテンポラルブロッキングの自動最適化を実現した．また，このテンポラルブロッキングはサブドメイン数とブロッキング段数という二つのパラメータによって性能が変化する．そこで，本研究ではパラメータ変化による性能変化を事前に予測するための性能モデルも構築した．

2. 研究背景

2.1 ステンシル計算

ステンシル計算は近傍の値を用いて計算を行っていく時間発展型の計算である．各点の計算を行う際に，ダブルバッファリングを用いることで全点を独立に計算すること

[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

ⓒ 2014 Information Processing Society of Japan

ができるため，高い並列性を有している．そのため共有メモリ環境ではグリッドの任意の要素を参照できるため，容易に並列化を行うことができる．しかし，分散メモリ型の並列化を行うと，プロセス間の境界に位置するセルは更新するために隣のプロセスの値を参照する必要が出てくる．一般的には，各プロセスが計算担当領域だけではなく，さらに隣接した空間のデータも保持する形を取る．このような値の更新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際に最新の値である必要があるため，一度計算したら値を更新するため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

本章ではステンシル計算の最適化手法の一つであるテンポラルブロッキング [7][8] について説明する．

ステンシル計算でメモリ領域が分かれるような並列化を行うと袖領域が生まれ，隣接通信が発生する．通常であれば 1 回の通信で 1 ステップの計算しか行わないが，通信するときの袖領域の幅を増やすことによって，複数回の計算を行うことができるようになり，通信回数が減少する．しかし，複数タイムステップを更新するためには本来計算を行う必要がない袖領域の計算も行う必要があり，計算コストが増加してしまう．

テンポラルブロッキングは CPU-GPU 間の PCI-Express 通信に適用することも可能である．Jin らの研究により PCI-Express 通信のテンポラルブロッキングに対して様々な最適化を行うことで，GPU メモリ容量以上のドメインに対して性能劣化することなしに 1GPU で計算できることが示されている．

1D-N 手法（naive method）
CPU 側で初期化したドメインをサブドメインに分割し，各サブドメインを 1 つずつ GPU に転送し GPU 上で計算を行う手法である．この手法では CPU-GPU 間の通信が頻繁に発生するため性能を出すことが困難となる．

1D-T 手法（temporal blocking method）
1D-N 手法と同様にサブドメインに分割をするが, GPU への転送をする際に袖領域を多めに送ることで複数ステップの計算を GPU 上でできるようになり通信回数を減らすことができる．

1D-TB 手法（+Buffer copy）
1D-T 手法では袖領域の冗長な計算が多いため，GPU 上にバッファを用意することで計算コストの増加を抑えることができる．

1D-TBM 手法（+Memory saving）
メモリ確保をシフトさせることによって，メモリ確保量を抑えることができる．これによりブロッキング段数の上限を上げることができるようになる．

3. テンポラルブロッキングの自動最適化

3.1 自動化方針

自動化を達成するためにはコンパイラによる変換やランタイムとして実装等の様々な方法がある．本研究においてはステンシル計算を対象とした自動並列化フレームワークである Physis を用いてテンポラルブロッキングの自動最適化を実現する．

3.1.1 自動並列化フレームワーク Physis

ステンシル計算はカーネルの記述が簡潔であり, メモリアクセスなども規則的であるため GPU による恩恵を受けやすい．しかし, GPU を利用できる CUDA や OpenCL などのプログラミング言語があるが, 並列プログラミングであるため効率を出すことは難しい．さらに, ステンシル計算は分散メモリ並列を行うと計算以外にデータの分割や袖領域の交換などをしなければならず，それらを MPI などを用いて記述しなければならない．このような背景のもとで, ユーザーに並列化を意識させずに, GPU 搭載クラスタを含む分散メモリ環境で実行できるコードに変換するフレームワークの研究が行われている．

汎用的なアプリケーションに対応させるのではなく, ステンシル計算のみを対象とすることで, 本質的に必要なカーネルの記述のみの簡潔な記述をすることが可能になっている．また，このフレームワークは新しい言語で作られているのではなく，広い目的で使われている C 言語をベースに作られている．そのため, Physis を使用するために新しくプログラミング言語を覚える必要はなく，既に記述されているアプリケーションなどに適用させることも比較的容易になっている．

3.1.2 目標コード

PCI-Express 通信のテンポラルブロッキングの自動生成を目標とする．先行研究により様々な最適化が提案されているが，本研究においてはまずカーネルの最適化がされていないナイーブな実装となっている 1D-T 手法の自動生成を目標とする．

全体のドメインをサブドメインに分割を行い，各サブドメインで cudaMemcpy を用いて GPU へ転送する．GPU 上で複数回計算を行うために袖領域を拡張させなければならないが，GPU メモリサイズはあまり大きくないため，サブドメインのサイズが大きすぎるとブロッキング段数を大きくできなくなってしまう．そのため解く問題のサイズや使用している GPU 環境に応じて，サブドメインの数とブロッキング段数は変化していく．

今回，PCI-Express 通信のテンポラルブロッキング最適化で最もナイーブな実装である 1D-T 手法の自動生成を実装した．通常のステンシル計算であれば**図 1** のようにカーネルを呼ぶだけで済む処理であるが，テンポラルブロッキングを実装するには**図 2** のようにコードを変更しなければ

ⓒ 2014 Information Processing Society of Japan

```
//時間発展ループ
for(int i=0; i<iter; i++){
  CUDA_kernel<<<grid, block>>>(args);
}
```

図 1 通常のステンシル計算の時間更新ループ

```
// 時間発展ループ
for(int i=0; i<iter/nblocking; i++){

// サブドメインに分割
  for(int j=0; j<sub_domains; j++){
    cudaMemcpyHostToDevice();
    // GPU 上での複数回の計算
    for(int k=0; k<nblocking; k++){
      CUDA_kernel<<<grid, block>>>(args);
    }
    cudaMemcpyDeviceToHost();
  }
}
```

図 2 テンポラルブロッキング実装時の時間更新ループ

ならない．

3.1.3 フレームワークへの実装

Physis フレームワークでは現在トランスレータにより自動生成できるコードが逐次 C，MPI，CUDA，MPI+CUDA の 4 種類ある．今回実装するテンポラルブロッキングは CUDA を用いた 1 ノード環境で実装を行うことができるため，CUDA 版をベースにテンポラルブロッキングの実装を行った．

CUDA 版では手動による CUDA 実装の基本と同様に，計算の前処理として必要領域を GPU デバイスメモリ上に通信しておき，時間発展ループの中の計算は全て GPU デバイスメモリ上で行い，後処理として CPU ホストメモリに計算結果を通信する形を取っている．そのため，Physis が計算を行うときに扱うグリッドはデバイスメモリ上にのみ宣言されている．しかし，今回実装を行うテンポラルブロッキングはサブドメインに分割した後，時間発展ループの中でそれぞれを計算する度に GPU メモリへの転送を行なってから計算を行う必要がある．そのため，扱うグリッドが GPU デバイスメモリ上だけでなく CPU ホストメモリ上にも宣言をする必要が出てくる．

テンポラルブロッキングを行う際にサブドメイン数とブロッキング段数の値が必要になってくる．この値はユーザーにとっては必要がないものであるため，これらの値は Physis の内部で管理を行い，現在の実装ではトランスレータにより自動的に宣言されるようになっている．

3.1.4 自動化の制限

今回実装したテンポラルブロッキングの自動生成に際していくつかの制限がある．

一つ目はテンポラルブロッキングの実装する際に必要となるサブドメイン数とブロッキング段数は最適値を取れているわけではないことが挙げられる．テンポラルブロッキングの性能はサブドメインの数とブロッキング段数によって大きく変動するが，現在は使用している GPU デバイスメモリ量と問題サイズから一意に指定される．トランスレータによる変換後のコードでこれらのパラメータを変更することができるので，最適な値を取るためにはユーザーがパラメータサーベイを行わなければならない．これらのパラメータの値は Physis のトランスレータにより生成されたコードの中で宣言されているため，値を変える事自体は容易に行うことができる．

二つめは複数のステンシルカーネルによって構成されているアプリケーションには対応していない．データに依存性のあるカーネルが複数あった場合にはテンポラルブロッキングができないケースが存在する．現在は 1 つの CUDA カーネルに対してテンポラルブロッキングを行うように実装を行っているため，データ依存性のあるカーネルではコード変換は成功してしまうが，計算結果合致の保証はされていない．

3.2 性能モデルの構築

PCI-Express 通信のテンポラルブロッキングを行うためには GPU のメモリ容量以下となるようにサブドメインに分割する必要があるため，最適化の伴い，サブドメイン数とブロッキング段数の二つを指定する必要がある．

PCI-Express 通信のテンポラルブロッキングのメインとなる時間発展部分のループは図 2 のような擬似コードとなり，主に cudaMemcpyHostToDevice と CUDA Kernel と cudaMemcpyDeviceToHost の 3 つから構成される．

cudaMemcpy による実行時間は PCI-Express のネットワークバンド幅の値とデータサイズから算出した．この時に，バンド幅の値はハードウェアの理論ピーク値を用いるのではなく，事前に cudaMemcpy 用の実効バンド幅を計測するためのベンチマークを作成し，データサイズを変えてバンド幅の値を取得している．DeviceToHost と HostToDevice では実効バンド幅の値が異なるので，それぞれ別の値を用いている．

$$\frac{byte}{BW_{PCI-Express}} \tag{1}$$

CUDA Kernel による実行時間は，ステンシル計算がメモリバンド幅律速であることから，演算性能ではなくメモリバンド幅の値が重要となる．ステンシル計算は同じ計算を何回も行う計算であり，各セルの計算も同一のものであるため，対象とするステンシル計算で一度ベンチマークを計測し，その実行時間からバンド幅の値を計算し，その値を実効メモリバンド幅として扱った．この実効メモリバンド幅の値を元に以下の計算式で実行時間を算出した．

$$\frac{byte}{BW_{stencil}} \tag{2}$$

ⓒ 2014 Information Processing Society of Japan

4. 性能評価

フレームワークにより自動生成したテンポラルブロッキングコードを中心に性能評価を行っていく．

4.1 実験環境

今回の実験では，東京工業大学のスーパーコンピュータ TSUBAME2.5 を使用した．TSUBAME2.5 は Thin ノード，Medium ノード，Fat ノードの 3 種類の計算ノードを使用することができるが，今回は Thin ノードを利用しており，ノード内に Intel Xeon X5670 の CPU が 2 ソケット，NVIDIA Tesla K20X が 3 基搭載されている．NVIDIA K20X は Kepler アーキテクチャであり，性能は単精度演算性能 3.94TFLOPS，倍精度演算性能 1.31TFLOPS，メモリバンド幅は 250GB/s となっている．CPU メモリは 54GB，GPU メモリは 1 基あたり 6GB で 3 基合計 18GB 搭載されている．3 基の GPU は PCI-Express2.0 x16 で接続されており，バンド幅は片方向 8.0GB/s となる．Thin ノード中の一部にはメモリが 96GB 搭載されており，今回 CPU メモリを大量に使う場合には CPU メモリが 96GB 搭載されているノードにて実験を行った．

CUDA ドライババージョンは 5.0，CUDA ランタイムは 5.0，CUDA Capability は 3.5 となる．

4.2 自動生成コードの性能評価

7 点ステンシルを Physis フレームワークで変換を行い，問題サイズを変更して性能測定を行ったものが図 3 となる．Physis にて変換を行ったものだけではなく，Jin らが手動で最適化したコードを同実験環境にて計測したもの，さらには GPU メモリ内に収まるサイズに対して，Kepler アーキテクチャにて使える Read-only cache を用いた Hand-tuning にて計測したものも載せている．テンポラルブロッキングを適用しているグラフではサブドメイン数とブロッキング段数を変化させてパラメータサーベイした結果，最も性能が良かったものを各問題サイズにおける性能として使用している．

今回計測した際の問題設定として，扱うデータは float，問題サイズは各辺の要素数が 256，512，768，1024，1280，1536，1792，全体のイテレーション数は 1000 回前後となるようにしている．また，問題サイズとサブドメイン数から，GPU メモリ容量を超えないための最大ブロッキング段数の値を計算することができるので GPU メモリに入りきらないサブドメイン数とブロッキング段数の組み合わせは事前に弾いている．

1GPU 内に収まる問題サイズの場合には手動実装による 1D-T 手法と比べて最大で性能が 145%となった．一方で，1GPU では容量が足りない問題サイズの場合には手動で実装されたものと比べて性能がわずかに低いものの，問題サイズが大きい場合には性能がほぼ一致することが確認できる．

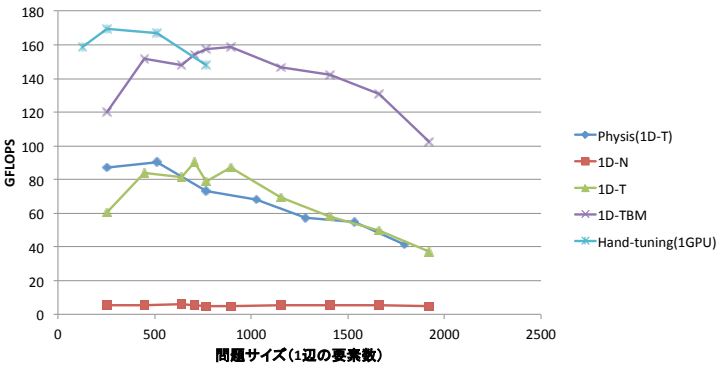

図 3 GFLOPS 比較

	手動最適化コード	Physis コード
カーネル記述	14 行	9 行
前処理 (init, malloc, memcpy)	18 行	19 行
時間発展ループ	60 行	3 行
後処理 (memcpy, free)	4 行	6 行

表 1 手動最適化版と自動生成版のコード行数比較

GPU メモリに入りきるサイズを実行した際の性能と比べると 53%〜30%程度となった．しかし，先行研究で示されている最適化を施した PCI-Express 通信のテンポラルブロッキング（1D-TBM）では，ピーク性能比較でも 93%程度となるため，現在の自動生成コードを最適化することにより同程度の性能が得られることが見込める．

4.3 プログラミングコストの評価

ステンシル計算は主要な計算カーネル自体は非常に簡単な記述で済む．そのため，最適化を行うことによる増える追加処理等のプログラミングコストは無視できない．そこで，PCI-Express 通信を対象とするテンポラルブロッキングを全て手動で実装したコードと，Physis フレームワークを用いる際の入力コードの行数を比較し，プログラミングコストの評価を行った．主要な処理毎にそれぞれの行数を比較したのが**表 1** となる．

Physis ではデータ管理するためにグリッド情報などを設定しなければならないため，前処理や後処理の行数がわずかに多い．時間発展ループ内については手動で最適化を行うには次のような点を考慮する必要がある．

- CPU メモリ，GPU メモリのデータ移動をユーザーが把握する必要がある
- 各サブドメイン処理時に，通信のデータサイズやオフセット値を計算しなければならない
- ブロッキング段数に応じてカーネルの計算範囲を計算しなければならない

ⓒ 2014 Information Processing Society of Japan

これに対して，Physis による自動生成では CPU メモリ，GPU メモリの概念を完全に隠蔽しているため，これらの処理をユーザーが考える必要がない．

現在，サブドメイン数やブロッキング段数の自動チューニングは達成できていないが，これらの計算処理が追加されているコードの生成が行えて，パラメータの値を変更するのもそれぞれ 1 行変更するだけで値を変更ができる．

4.4 性能モデル評価

PCI-Express 通信を対象とするテンポラルブロッキングを実装したときの性能を予測する性能モデルの評価を行った．

問題サイズを $1024 \times 1024 \times 11024$，サブドメイン数とブロッキング段数を変化させたときの性能モデルによる予測値と，Physis フレームワークにより自動生成されたコードによる実測値を比較したものが図 4 となる．ブロッキング段数によって 1 イテレーション内での計算回数は変化してしまうため，計算 1 ステップあたりの実行時間を縦軸にとっている．

実測値では実行時間にばらつきがあるが，サブドメイン数，ブロッキング段数変化に伴う実行時間変化の挙動は表現できていると思われる．サブドメイン数が 2 のときには誤差が大きくなっているため，原因を調査する必要がある．サブドメイン数が 2 と 4 以上では分割によるデータ通信量変化が大きく異なり，モデルに反映できていないことも考えられる．

5. 関連研究

テンポラルブロッキングについての研究は多いが，ステンシル計算がメモリバンド幅律速であるため，特にメモリアクセスを対象とする研究が多くなされている．Nguyen ら [9] は，キャッシュヒット率を高める 2.5D スペーシャルブロッキングと，メモリアクセスを対象とした 1D テンポラルブロッキングを組み合わせた 3.5D ブロッキング手法を提案している．Meng ら [10] はメモリアクセスを対象に，ブロッキング段数の最適値の予測を行うために GPU アーキテクチャ性能から性能モデルを構築している．

ステンシル計算はカーネルが単純であるため，ステンシル計算を対象とした自動並列，自動チューニングの研究も多くなされている．Kamil ら [11] は Fortran95 で書かれたコードを C や Fortran，CUDA のような並列言語に変換するフレームワークを提案している．最適化としては多段ブロック化を行っており，ハードウェアのキャッシュサイズなどに合わせて適切なパラメータを設定している．

6. おわりに

本論文では GPU メモリ容量を超える問題サイズを解くことが可能になる PCI-Express 通信のテンポラルブロッキングに対して自動最適化を提案し，ステンシル DSL の Physis フレームワークに組み込むことで実現した．同程度の最適化を行った手動実装版とほぼ同等であることを示し，最適なパラメータを出すために性能モデルを構築し評価を行ったところ，パラメータ変化に伴う実行時間の変化が表現できていることも確認できた．

参考文献

[1] 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 '12, pp. 3:1–3:11, New York, NY, USA, 2011. ACM.

[2] 小野寺直幸, 青木尊之, 下川辺隆史, 小林宏充. 格子ボルツマン法による 1m 格子を用いた都市部 10km 四方の大規模 les 気流シミュレーション. 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 2013 巻, pp. 123–131, jan 2013.

[3] Guanghao Jin, Toshio Endo, and Satoshi Matsuoka. A parallel optimization method for stencil computation on the domain that is bigger than memory capacity of gpus. In *Cluster Computing (CLUSTER), 2013 IEEE International Conference on*, pp. 1–8, 2013.

[4] Guanghao Jin, Toshio Endo, and Satoshi Matsuoka. A multi-level optimization method for stencil computation on the domain that is bigger than memory capacity of gpu. In *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, IPDPSW '13, pp. 1080–1087, Washington, DC, USA, 2013. IEEE Computer Society.

[5] 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.

[6] 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.

[7] 河村知輝, 丸山直也, 松岡聡. 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. Technical Report 32, 東京工業大学, 東京工業大学／理化学研究所／科学技術振興機構 CREST, 東京工業大学／科学技術振興機構 CREST ／国立情報学研究所, jul 2012.

[8] 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.

[9] 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,

ⓒ 2014 Information Processing Society of Japan

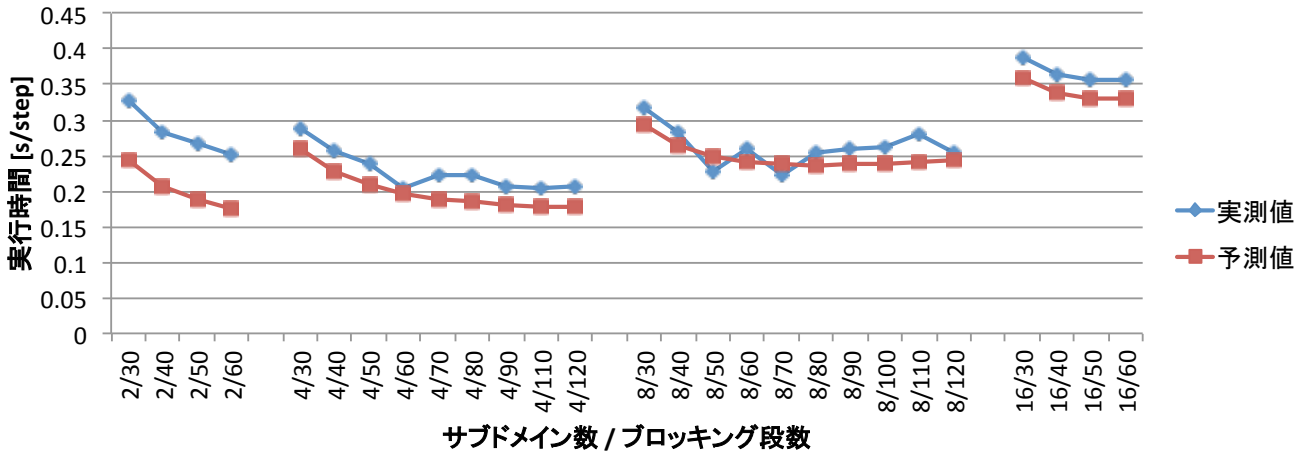

図 4 問題サイズ 1024x1024x1024，サブドメイン数とブロッキング段数を変化させたときのモデルによる予測値と実測値の比較

Washington, DC, USA, 2010. IEEE Computer Society.

[10] 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.

[11] 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.

ⓒ 2014 Information Processing Society of Japan