情報処理学会論文誌　コンピューティングシステム　Vol.9　No.4　25–37 (Nov. 2016)

ディレクティブに基づく ステンシル計算の性能パラメータ自動設定

角川 拓也[1,a)]　平澤 将一[1,b)]　滝沢 寛之[1,c)]　小林 広明[1,d)]

受付日 2016年4月22日, 採録日 2016年7月28日

概要：異種複数のシステム上で高性能を達成するためには，多数の性能パラメータをそれぞれのシステム向けに適切に設定する必要がある．現在，プログラムを実際に実行して得られる動的な性能プロファイル情報に基づいて性能パラメータを設定する研究がさかんに行われているが，システム上でそのような性能情報を得るためには長時間を要するという問題がある．そこで本論文では，科学技術計算で多用されるステンシル計算を対象とし，その性能最適化に必要な情報をディレクティブを用いて追記することによって，静的情報に基づいて性能パラメータを決定することを考える．ディレクティブとして与えられたステンシル計算に関する情報と，事前に用意されたシステムの情報を用いることにより，性能プロファイリングによらずにステンシル計算の性能パラメータを自動設定する手法を提案する．システムのアーキテクチャ的な特徴を考慮可能な規則を定義することにより，ステンシル計算の最適化で求められる性能パラメータを自動設定できることが評価結果より明らかになった．

キーワード：アクセラレータ，自動チューニング，性能最適化

Automatic Parameter Tuning of Stencil Computation Using Directives

Takuya Tsunogawa[1,a)]　Shoichi Hirasawa[1,b)]　Hiroyuki Takizawa[1,c)]　Hiroaki Kobayashi[1,d)]

Received: April 22, 2016, Accepted: July 28, 2016

Abstract: To achieve high performance on various computing systems, it is necessary to configure a lot of parameters for each system. Today, many approaches have been proposed for parameter configuration based on dynamic performance profiling information obtained by actual program execution. One problem is that it could take a very long time to obtain such performance information of each system. Considering stencil computation frequently used in scientific and technical computing, hence, this paper discusses parameter configuration based only on static information available at the compilation time while assuming that necessary information for performance optimization is given using directives. This paper proposes an automatic parameter configuration method for stencil computation that uses only static information about stencil computation and system configuration, i.e., it does not rely on performance profiling. The evaluation results show that performance parameter configuration can be automated by defining some rules considering architectural characteristics of the target system.

Keywords: accelerator, auto-tuning, performance optimization

1. はじめに

流体シミュレーションや電磁界解析等の科学技術計算は，空間格子の各格子点に対して同一の演算を適用する**ステンシル計算**と呼ばれる特徴的な計算パターンを使用して記述されることが多い．近年，ステンシル計算の高速化のために，汎用プロセッサ（Central Processing Unit,

[1] 東北大学大学院情報科学研究科
Graduate School of Information Sciences, Tohoku University, Sendai, Miyagi 980–8579, Japan
[a)] tsuno@sc.cc.tohoku.ac.jp
[b)] hirasawa@sc.cc.tohoku.ac.jp
[c)] takizawa@tohoku.ac.jp
[d)] koba@tohoku.ac.jp

CPU）に加えてアクセラレータを使用する事例が多数報告されている [13]．特に，NVIDIA 社製の描画処理ユニット（Graphics Processing Unit，GPU）や Intel 社製の Intel Xeon Phi（Xeon Phi）が，アクセラレータとして広く用いられている [1], [3]．

GPU と Xeon Phi のようにアーキテクチャの異なるアクセラレータで共通に使用可能なプログラムを開発するために，OpenCL [5] に代表される標準プログラミング環境の整備が進んでいる．そのような環境で高性能なプログラムを開発するためには，いくつかのパラメータ（**性能パラメータ**）を各システムに応じて適切に設定する必要がある．しかし，あるシステムに対して最適な性能パラメータ設定が，他のシステムでは性能を低下させる要因となる恐れがある．したがって，異種複数のシステム上で高い性能を達成できる性質である**性能可搬性**を高めるためには，それぞれのシステムに対して異なる性能パラメータ設定が必要となる．今後，システムのさらなる多様化が予想されていることから，性能可搬性の向上は今後ますます重要となる課題である [6], [8]．

性能可搬性の問題に対して，自動チューニング（Automatic performance Tuning，AT）技術がさかんに研究されている [17]．多くの AT 研究では，プログラムを実際に実行して得られる動的な性能プロファイル情報に基づいて，適切な性能パラメータを決定している．しかし，パラメータ探索の空間が大きい場合には，動的情報を取得するためのプログラム実行に長時間を要し，現実的な時間内に性能パラメータを決定できなくなる恐れがある．このため，コンパイル時に既知となる静的情報から性能パラメータを適切に決定することが求められている．

高い性能可搬性を達成するための別のアプローチとして，近年様々なドメイン特化型言語（Domain Specific Language，DSL）が提案されている [11]．DSL では，既存のプログラム中のカーネル部分を DSL を使用して記述し直す必要がある．しかしながら，今日の大規模アプリケーションのコード行数は数千から数万行におよぶため，使用するシステムに応じてプログラムを書き換える手間はプログラマにとって大きな負担となる．また，抽象度の高い記述によって実装の詳細を隠蔽し，その結果として性能可搬性の改善を期待できる一方で，パラメータ探索の所要時間の長さに関する問題は依然として解決されていない．

このように，今日のプログラミング環境では，性能可搬性の高さと既存のプログラムの移植容易性の高さの両立は実現されていない．そこで本論文では，科学技術計算で多用されるステンシル計算を対象とし，その性能最適化に必要な情報をディレクティブを用いて追記することによって，静的情報に基づいて性能パラメータを決定することを考える．ディレクティブとして与えられたステンシル計算に関する情報と，事前に用意されたアクセラレータの情報を用いることにより，性能プロファイリングによらずにステンシル計算の性能パラメータを自動設定する手法を提案する．

2. 関連研究

2.1 ステンシル計算の性能最適化手法

2.1.1 空間ブロッキング

一般に，ステンシル計算の素朴な実装では，メモリアクセスが性能ボトルネックとなる．このため，ステンシル計算の性能最適化手法として，低速なメインメモリへのアクセス回数を削減する空間ブロッキング（Spacial Blocking，SB）が広く使用されている [12]．

SB では，空間格子をオンチップメモリに保持可能な大きさに分割し，分割した格子（分割格子）に対して順番にステンシル計算が適用される．SB によってオンチップメモリ上のデータを再利用可能になり，メインメモリへのアクセス回数が削減される．分割格子のデータがオンチップメモリに保持されている間に，分割格子に含まれるすべての格子点に対して計算を適用することで高速なオンチップメモリを有効活用できる．

2.1.2 テンポラルブロッキング

一般に，科学技術計算に現れるステンシル計算では，時間軸方向へ繰返し処理が行われる．テンポラルブロッキング（Temporal Blocking，TB）は，SB に加えて時間軸方向への繰返し処理を複数回進めることで，格子点データの時間軸方向への再利用性を利用する．

TB の中でも，本論文ではオーバラップタイリング（Overlapped Tiling）[10] を対象として考える．TB では，SB によってオンチップメモリ上に格子データが保持されている間に，時間軸方向に複数ステップ繰返し処理を進めることで，メインメモリへのアクセス回数をさらに削減可能である．

2.2 性能パラメータの自動チューニング

OpenCL [5] や OpenACC [7] 等のアクセラレータを意識したプログラミング環境においては，アクセラレータのためにいくつかの性能パラメータを適切に設定する必要がある．プログラミング環境ごとに異なる用語で表現されているが，想定されている実行モデルは類似しているため，OpenCL の用語でアクセラレータの性能パラメータを説明する．まず OpenCL では，ホスト（CPU）が計算デバイス（アクセラレータ）を制御する．計算デバイスは複数の計算ユニット（Computing units）を備えており，各計算ユニットは複数の処理要素（Processing elements）から構成されている．各処理要素でワークアイテム（Work Items，WI）を実行することにより，計算デバイスは大規模な並列処理を実現する．また，いくつかの WI をまとめたワークグループ（Work Groups，WG）という概念が用意され

ており，WG の単位でそれぞれの計算ユニットに処理が割り当てられる．WI は n 次元（$1 \leq n \leq 3$）のインデックスを使って管理されており，n 次元以下の任意の形状と大きさで WG を構成することができる．ステンシル計算を OpenCL で記述する場合，簡単のために各格子点での計算を 1 つの WI に割り当てると仮定しても，WG の大きさと形状はアクセラレータごとに適切に決めなければならない．WG の大きさや形状は，アクセラレータの性能への影響の大きい重要な性能パラメータである [16]．

さらに SB や TB では，分割格子のサイズや一度に更新するステップ数（**時間ブロックサイズ**）を適切に決定する必要があり，それらもステンシル計算の性能最適化に求められる性能パラメータとして考える必要がある．これらの性能パラメータの適切な値は，ステンシル計算カーネルとそれを実行するシステムごとに異なる．このため，性能パラメータをシステムに合わせて調整しない限り，異種複数のシステム間で高い性能可搬性を達成することはできない．また，OpenCL でステンシル計算を実装する場合には，各 WG を 1 つの分割格子に対応させることが多い．この場合には，SB としての観点と OpenCL プログラムとしての観点の両方を同時に考えて，WG の大きさと形状を適切に決めなければならない．

そのような性能パラメータの設定が求められる条件下において性能可搬性を改善するために，AT 技術がさかんに研究されている [17]．多くの AT 研究では，性能パラメータを変化させながら性能プロファイリングを行い，性能の高くなる性能パラメータ設定を発見するアプローチがとられている．近年の高性能計算システムの性能を正確にモデル化することは容易ではないことから，性能プロファイリングに基づく経験的（empirical）な性能パラメータ探索が多くの場合必要である．しかし，性能パラメータの数が多い場合等には膨大な組合せで性能プロファイリングを行う必要があり，全探索では現実的な時間内に適切な性能パラメータ設定を決定できない恐れがある．

さらに近年では，高い抽象度で処理を記述することで実装の詳細を隠蔽し，プログラマに性能パラメータの存在を意識させない DSL も数多く提案されている．この場合，DSL の処理系に性能パラメータ設定を任せることができる．しかし，高性能計算アプリケーションの多くは C 言語や Fortran といった汎用のプログラミング言語で記述されているため，DSL を利用するためにはそのアプリケーションコードを DSL の提供する独自文法で書きなおす必要がある．DSL の処理系が内部的に性能プロファイリングによる経験的なパラメータ探索を行っている場合には，性能パラメータ探索の所要時間も依然として問題となる．

3. ステンシル計算専用ディレクティブによる性能パラメータ自動設定

2.2 節での議論から，現在では高い性能可搬性とコード移植性を両立させるプログラミング環境はいまだ確立されていないことが分かる．そこで本論文では，ステンシル計算のアクセラレータ向け性能最適化で求められる性能パラメータをコンパイル時に自動設定する手法を提案する．提案手法では，アーキテクチャ的な特徴を考慮可能な規則を設定することにより，性能プロファイリングによらずに性能パラメータを自動設定することが可能である．コード移植性を考えて，自動設定に必要な情報の記述にはディレクティブを用いるものとし，C 言語で記述された既存のステンシル計算カーネルに必要情報を追記することで，性能パラメータの自動設定を実現する．その結果として，高い性能可搬性とコード移植性を両立する．

本論文では，ステンシル計算カーネルの性能最適化を行うトランスレータを試作し，その実装を前提にして提案手法である性能パラメータ自動設定手法の有効性を議論する．トランスレータの概念図を**図 1** に示す．本実装は，専用のディレクティブが挿入された C 言語のプログラムを入力とする．入力プログラムは，ステンシル計算の中間表現に変換された後に，OpenCL プログラムへと変換される．本実装の詳細については，3.4 節で述べる．

図 1 の変換過程において，ディレクティブで指示された入力プログラム中のステンシル計算に対して SB および TB が適用される．OpenCL プログラムに変換するためには，アクセラレータに対して適切に WG の大きさと形状を決めなければならない．WG 形状は SB における分割格子に対応している．すなわち，OpenCL の性能パラメータと SB や TB の性能パラメータは密接に関係していることから，その両方の適切な設定が求められる．この変換過程を前提として，提案手法である性能パラメータ自動設定手法を以下で説明する．

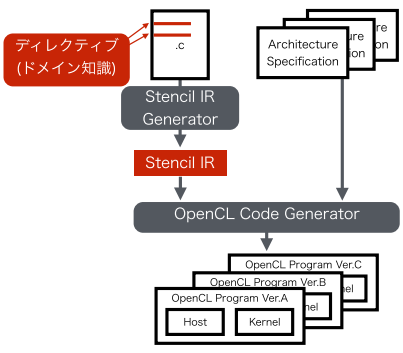


図 1　提案手法を用いたトランスレータ
Fig. 1　A translator with the proposed method.

3.1　ステンシル計算の性能パラメータ自動設定の概要

本提案手法では，SB における分割格子に対応する WG 形状と，時間ブロックサイズを自動設定する．WG は WI の集合である．本論文では，各 WI に対して 1 つの格子点に関する計算が割り当てられるものと仮定する．そのため，分割格子の形状は WG 形状と対応している．また，アクセラレータに関する情報は事前に与えられているものと仮定する．

以下，WG 形状および時間ブロックサイズを 4 つの性能パラメータと考え，4 次元の**性能パラメータベクトル**（Parameter Vector，PV）と表記する．たとえば (W, H, D, T) と表記されている場合，W，H，D がそれぞれ x 軸，y 軸，z 軸方向の WG のサイズを表しており，T は時間ブロックサイズを表している．時間軸を t 軸と表記する．また，WG 形状および時間ブロックサイズを拡大可能な方向を表すベクトルを，**成長ベクトル**（Evolution Vector，EV）と呼ぶ．たとえば EV が $(1, 0, 1, 1)$ の場合，WG を y 軸以外の方向に拡大でき，時間ブロックサイズも拡大できることを示している．

本提案手法の WG 形状と時間ブロックサイズの決定アルゴリズムを**図 2** に示し，その手順を以下で説明する．

(1) PV と EV を初期値 $(1, 1, 1, 1)$ に設定してパラメータ探索を開始する．
(2) 3.3 節で後述する規則に基づいて PV を暫定的に変更する．
(3) 暫定値が 3.2 節で後述する分割格子サイズ変更条件を満たす場合には，暫定 PV を新しい PV として用いる．一方，分割格子サイズ変更条件を満たさない場合には，暫定 PV が拡大した方向に対応する EV の要素を 0 に設定する．
(4) EV に非ゼロ要素が含まれている場合には，手順 (2) に戻る．それ以外の場合にはパラメータ探索を終了する．

以上の手順の擬似コードを本論文の付録に示す．

3.2　分割格子サイズ変更条件

オンチップメモリに保持される格子点の集合を**タイル**と呼ぶ．タイル領域の大きさは，WG 形状に袖領域を加えることで算出可能である [10]．袖領域の大きさは各格子点計算でアクセスされる領域を基に算出される．TB を行う場合，**図 3** に示されるように，時間ブロックサイズの拡大にともなって各 WG のタイル領域も拡大する．

分割格子サイズ変更条件は，アクセラレータの性能およびアーキテクチャ的な特徴を考慮して設定する必要がある．そこで本提案手法では，分割格子サイズ変更条件を以下の 3 条件と定義する．

- タイルをオンチップメモリ（あるいはキャッシュ）上に保持可能であること

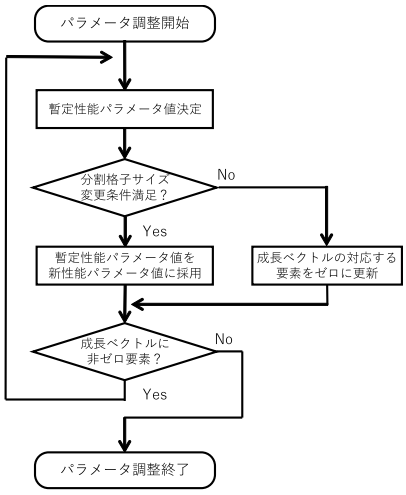


図 2　性能パラメータ自動設定アルゴリズム
Fig. 2　The proposed automatic parameter tuning algorithm.

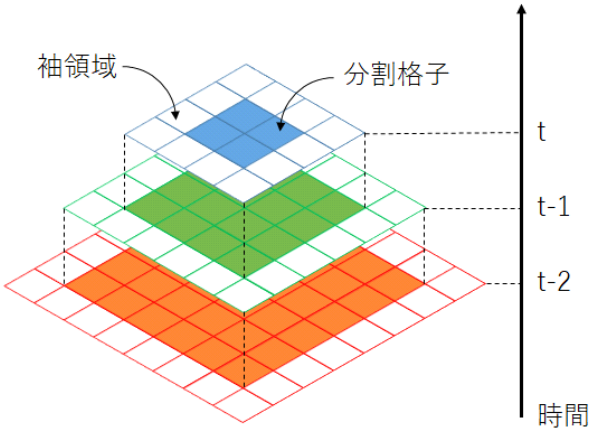


図 3　テンポラルブロッキング時にワークグループが計算する領域
Fig. 3　The region computed by a work group in the case of temporal blocking.

- メモリバンド幅律速であること
- WG 数が一定数以上であること

WG の拡大にともなって，オンチップメモリに保持されることが期待されるデータサイズも増大する．オンチップメモリ上のデータをプログラムが明示的に管理している場合，オンチップメモリ容量を超えるデータの管理には複雑な制御が必要となる．オンチップメモリをキャッシュとして使用している場合でも，キャッシュ容量を超えるデータサイズでは容量性のキャッシュミスが頻発し，性能の著しい低下を引き起こす．そのため，タイルサイズがアクセラレータに搭載されるオンチップメモリのサイズを超えないように性能パラメータを調整する．

また，時間ブロックサイズにはトレードオフがあり，その増加にともなってオフチップメモリへのアクセス回数が削減される一方で，必要なオンチップメモリ容量が増え，さらには冗長な計算も急激に増加する．そのため本提案手法では，メモリバンド幅律速状態が維持される範囲に時間ブロックサイズを限定することで，冗長計算による性能低

下を回避する．以下，システムの理論メモリバンド幅と理論演算性能との比を**システムの BF 値**と呼ぶ．同様に，ステンシル計算中のデータアクセス回数と演算回数の比を**アルゴリズムの BF 値**と呼ぶ．システムの BF 値とアルゴリズムの BF 値を比較することで，演算律速またはメモリバンド幅律速を判定する．アルゴリズムの BF 値がシステムの BF 値を上回る範囲内で時間ブロックサイズを拡大することで，ステンシル計算が演算律速になることを回避する．

本論文では，配列アクセスが初回のみキャッシュミスしてメモリからのデータ転送となり，それ以後のアクセスはキャッシュ上のデータにヒットすることを仮定している．この仮定の下で入力プログラムのステンシル計算中の配列数 N_{array} と演算回数 N_{op} をそれぞれ数えることで，以下の式からアルゴリズムの BF 値を推定している．

$$BF_{\mathrm{alg}} = \frac{N_{\mathrm{array}} \times D_{\mathrm{field}}/N_{\mathrm{WI}}}{N_{\mathrm{op}}} \tag{1}$$

ここで，D_{field} はステンシル計算でアクセスする配列全体のサイズをバイト単位で示しており，N_{WI} は WI 総数を表している．メインメモリから転送されるデータサイズを演算回数で割ることによってアルゴリズムの BF 値を算出する．

各 WI が 1 つの格子点に対応しているため，計算格子のサイズが変わらなければ WI の総数は一定である．このため，WG の拡大にともなって WG の数が減少し，WG 間で独立に並列処理可能な並列性が低下する．その結果，WG の数が少ない場合に発生するテイルエフェクトによって性能が低下する恐れがある．本論文で対象とする GPU，Xeon Phi ともに，WG の数は 1000 以上であることが推奨 [2], [4] されている．このため，本論文では WG 数の下限の閾値を設定し，テイルエフェクトによる性能低下を軽減する．

3.3　成長ベクトルの決定方法

アーキテクチャの違いに起因して，EV の決定方式もアクセラレータごとに考える必要がある．以下に示す理由から，本論文では 4 次元の EV(x, y, z, t) の決定に際し，x 軸方向，t 軸（時間軸）方向，y 軸方向，z 軸方向の順に，拡大方向に優先度を設ける．本論文では，各方向のサイズが 2 のべき乗になるように WG を拡大する．

GPU に関しては，はじめに SMX ユニット [18] 内のコアの稼働率の向上を期待して，x 軸方向に WG を拡大する．アーキテクチャ的な特徴から，WG の x 軸方向の大きさが 32 の倍数のときに高性能となるため，本手法では x 軸方向の大きさをまず 32 に拡大する．次に，アルゴリズムの BF 値に基づいてメモリバンド幅律速と判断される場合に時間ブロックサイズを 1 時間ステップ分だけ拡大する．その結果として冗長計算が増加し，アルゴリズムの BF 値が低下する．その後，y 軸および z 軸方向の WG サイズを拡大する．GPU における WG 拡大方向の決定方法は，付録の擬似コードのサブルーチン `compute_temp_gpu` に対応している．

Xeon Phi に関しては，はじめにベクトルユニットの活用を期待して x 軸方向に WG を拡大する．次に，アルゴリズムの BF 値を算出して，メモリバンド幅律速である場合には時間ブロックサイズを増加させて，アルゴリズムの BF 値を低下させる．その後，さらなるベクトルユニットの使用を期待して，x 軸方向に WG を拡大する．タイルをオンチップメモリに保持できなくなり，x 軸方向に対して制限された際には，y 軸および z 軸方向へ WG を拡大する．Xeon Phi における WG 拡大方向の決定方法は，付録の擬似コードのサブルーチン `compute_temp_phi` に対応している．

以上のように EV という概念を導入し，EV の決定方法をアーキテクチャごとに設計することにより，それぞれのアーキテクチャに対して適切に WG を決定することが可能である．本論文では以上のように EV の決定方法を定義したが，さらに多くの条件を加えて WG サイズを調整することにより，より適切なパラメータ選択を実現できる可能性も残されている．

オンチップメモリにタイルデータを保持しきれない場合には，暫定 WG を計算した際の EV を以後選択しないように制限する．これは付録の擬似コードの `update_ev` で行われているように，EV の要素の値を変更することで実現されている．また，BF 値の比較により演算律速へ遷移すると予測される場合には，時間方向への EV を選択できないように制限する．WG 数が閾値を下回る場合には，空間方向の EV を選択しないように制限する．すべての方向に対して EV 選択を制限したとき，WG 形状および時間ブロックサイズの拡大を停止し，WG 形状および時間ブロックサイズを確定する．

3.4　ステンシル専用ディレクティブの設計

本論文では，比較的少ない追加情報をディレクティブで指示することにより，C 言語で書かれたステンシル計算を OpenCL のプログラムに変換するトランスレータを実装して評価に用いる．本実装の主目的は，提案手法によってステンシル計算の性能パラメータを自動設定可能であることを示すことである．

一般に，OpenCL のプログラムはホストコードとデバイスコードで構成されている．ホストコードには CPU で動作する処理が記述されており，アクセラレータの制御もホストコードに記述される．一方，デバイスコードにはアクセラレータで行う処理が記述されている．本実装は Scala [14] で記述されており，以下のような制限があるものの，ディレクティブの挿入された C 言語のプログラムから OpenCL のホストコードとデバイスコードを生成すること

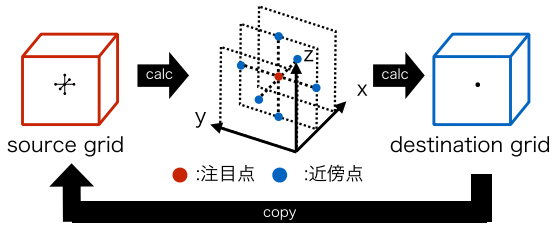


図 4　本実装で想定するステンシル計算
Fig. 4　Stencil computation assumed by the implementation.

表 1　ステンシル計算の中間表現に必要な情報
Table 1　Information for stencil intermediate representation.

中間表現	情報	ディレクティブによる指示
計算格子	始点	不要
	終点	不要
	方向	必要
計算場	データ型	必要
	データサイズ	必要
	次元	必要
	名前	不要
	役割	必要
ステンシル関数	演算式	不要
	同期のタイミング	必要

ができる．本実装が生成するホストコードには，入力プログラムの内容に加えてアクセラレータの制御に必要な配列の確保やアクセラレータとのデータ転送等の処理も記述されている．

本実装で想定する入力プログラムを**図 4** で説明する．計算格子のデータは固定長の配列になっており，ステンシル計算の後に入力配列と出力配列の入れ替えを行っている．また，時間発展のループを最外ループとし，その中に計算格子にステンシル計算を適用するための多重ループと配列コピーが記述されている．さらに，式 (1) によるアルゴリズムの BF 値の推定を容易にするため，現在の実装ではステンシル計算部分に関数呼び出しはないものと仮定し，演算子の数から演算回数 N_{op} を取得している．

本研究では，C 言語のプログラムからは取得が困難なステンシル計算に関する情報をディレクティブを用いて記述する．本論文では，計算格子，計算場，ステンシル関数の 3 つの概念 [10] でステンシル計算を表す．本論文では，これを**ステンシル計算の中間表現**と呼ぶ．本論文で使用するステンシル計算の中間表現に使用する情報を**表 1** に示す．

計算格子は空間格子および時間方向への繰返しを表現する概念である．ループ文の制御構文より，計算格子の始点と終点を取得可能である．しかしながら，プログラム中のループ文の制御には変数が使用されるため，各ループ文と対応する計算格子の方向を指示する必要がある．

計算場は，ステンシル計算に使用するデータを表現する概念である．C 言語においては，ポインタを使用することで配列の動的な領域確保や 1 次元配列による多次元配列の

表 2　本実装で利用するディレクティブの仕様
Table 2　Directives supported by the translator.

ディレクティブ名	指示内容
step	時間方向のループ変数名とその最大値
dimension	各空間軸方向のループ変数名
calc	ステンシル計算部分
copy	コピー部分
data	入力配列名と出力配列名
begin	ステンシル計算部の開始点
end	ステンシル計算部の終了点

表現が可能であり，そのような場合には計算場の情報を入力プログラムのみから正確に取得することは困難である．このため，計算場の取得には配列のデータ型と大きさ，次元，役割を明示的に指示する必要があり，本実装では固定長配列を用いることを仮定している．

ステンシル関数は，計算場に対して行う計算を表現する概念である．逐次処理を並列化してアクセラレータで実行するためには，データの依存関係に基づいて同期処理を挿入する必要がある．本実装では，そのようなデータの依存関係解析をトランスレータ自身は行わない．その代わりに，必要な情報がディレクティブとして入力プログラム中に記述されていることを仮定している．

本論文では，**表 2** に示すディレクティブを使用することで以上の情報を記述し，ステンシル中間表現を取得する．提案手法によるプログラム例を**図 5** に示す．図 5 に示すように，本実装では既存のプログラムに対してステンシル計算の情報を記述したディレクティブを挿入する．本実装によって，図中の 10〜12 行目に示されている式文が，デバイスコードのカーネル関数内でも利用される．また，必要な一時変数がカーネル関数内で定義され，式文中のループ変数が各 WI の ID を示す変数に置換される．2 行目のディレクティブ `step` は，変数 `t` が時間方向のループ変数であり，その最大値が 1000 であることを示している．3 行目のディレクティブ `data` は，図 4 で表現されるステンシル計算を想定した場合の入力配列と出力配列を指示しており，本実装では固定長配列を想定している．4 行目のディレクティブ `dimension` は，各空間軸方向に対応するループ変数を指示している．このディレクティブに与えられた変数の数によって，計算格子の次元も判定している．ディレクティブ `calc` および `copy` は，図 4 の calc および copy に相当する処理であることを示している．

本実装には入力プログラムに関して制約があるものの，ディレクティブによる追記のみでアクセラレータを使用することが可能であり，提案手法による性能パラメータ自動設定の有用性を評価するという目的で必要となる機能を実現できている．

```
#pragma stencil begin
#pragma stencil step(t:1000)
#pragma stencil data source(float A[NZ][NY][NX]) destination(float B[NZ][NY][NX])
#pragma stencil dimension(i, j, k)
for(t=0; t<total_step; t++){
#pragma stencil calc
  for(i=1; i<NZ-1; i++)
    for(j=1; j<NY-1; j++)
      for(k=1; k<NX-1; k++)
        B[i][j][k] = C_0*A[i][j][k] + C_1*( A[i-1][j][k] + A[i+1][j][k] +
                                            A[i][j-1][k] + A[i][j+1][k] +
                                            A[i][j][k-1] + A[i][j][k+1] );
#pragma stencil copy
  for(i=1; i<NZ-1; i++)
    for(j=1; j<NY-1; j++)
      for(k=1; k<NX-1; k++)
        A[i][j][k] = B[i][j][k];
 }
#pragma stencil end
```

図 5　ステンシル計算専用ディレクティブを用いたコードの例
Fig. 5　A sample code using directives for stencil computation.

4. 性能評価

4.1 評価環境

本節では，3.4 節の実装を用いて提案手法による性能可搬性改善，プログラムの移植容易性，および性能パラメータ決定に要する時間（チューニングオーバヘッド）を定量的に評価する．

本評価で用いるシステムおよびアクセラレータの仕様を**表 3** および**表 4** に示す．コンパイラの最適化オプションには `-O3` を用いるものとする．表 4 に示すように，GPU と Xeon Phi ではオンチップメモリの階層関係とサイズが異なることから，オンチップメモリのサイズの違いによるブロッキングの効果を評価することが可能である．各アクセラレータのメモリバンド幅と理論演算性能から，システムの BF 値を計算可能である．また，WG の適切な数もアクセラレータごとに異なることから，本評価では WG 数の下限値をアクセラレータごとに設定している．これらの値はシステム情報として提案手法で用いられる．付録の擬似コードにおいては，それぞれ `S_cache`, `BF_sys`，および `N_ng` というパラメータ名で与えられている．

また本評価には熱伝導シミュレーションを想定した 1 次元の 3 点ステンシル，2 次元の 5 点ステンシル，3 次元の 7 点ステンシル，および姫野ベンチマークのプログラムを使用する．各ベンチマークのパラメータを**表 5** に示す．比較対象として，OpenCL および OpenACC によって記述されたコードを使用する．OpenCL プログラムの性能測定には，OpenCL 標準のプロファイリング機能を用いており，`clGetEventProfilingInfo` 関数でカーネル実行時間を取得している．OpenACC プログラムの性能測定では，環境変数 `PGI_ACC_TIME` を 1 に設定して表示されるカーネル実行時間を利用している．

表 3　評価に使用するシステムの構成
Table 3　System configurations.

	PC0	PC1
CPU	Intel Core i7 930	Intel Xeon E5-2690
アクセラレータ	NVIDIA Tesla C2070	Intel Xeon Phi 5110P
メインメモリ	24 GBytes	32 GBytes
Linux	2.6.32-573.3.1. el6.x86_64	2.6.32-573.22.1. el6.x86_64
コンパイラ	gcc 4.4.7 PGI compiler 15.7	gcc 4.4.7
OpenCL	NVIDIA OpenCL 1.1	Intel OpenCL 1.2
CUDA	7.0	-

表 4　評価に使用するアクセラレータ
Table 4　Accelerators used for the evaluation.

アクセラレータ	Tesla C2070	Intel Xeon Phi 5110P
演算コア数	448	60
理論演算性能	1.03 TFLOPS	1.01 TFLOPS
デバイスメモリ	6 GBytes	8 GBytes
共有 L2 キャッシュ	768 KBytes	30 MBytes
L1 キャッシュ	16 KBytes (per SMX)	32 KBytes (per core)
共有メモリ	48 KByte (per SMX)	-
メモリバンド幅	144 GBytes/s	320 GBytes/s
システムの BF 値	0.14	0.32
WG 数の下限値	8192	1024

4.2 評価と考察

4.2.1 OpenACC および OpenCL との性能比較

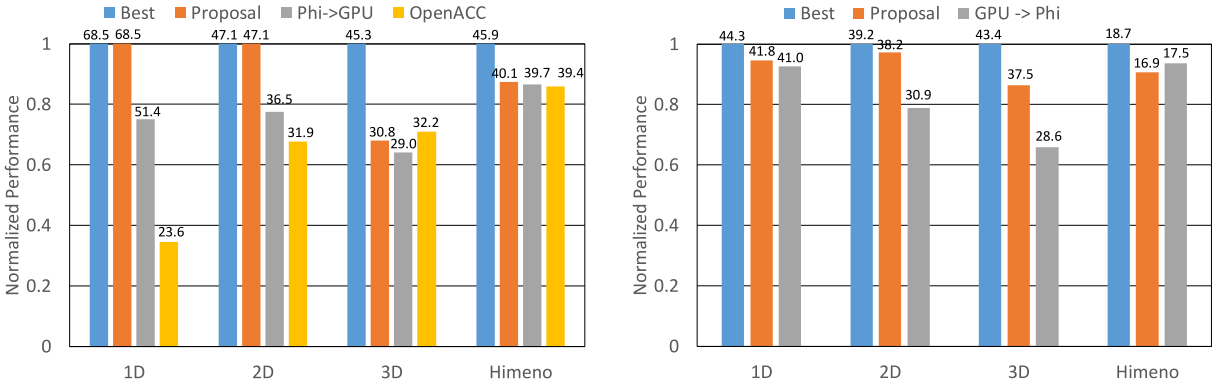
本評価では，提案手法と OpenACC および OpenCL プログラムの実行性能および書き換えに必要な行数を使用して，提案手法の性能可搬性および移植容易性を評価する．性能可搬性を議論するために，実行するアクセラレータとは異なるアクセラレータ向けに全探索を用いてパラメータチューニングされた OpenCL プログラムと提案手法を比較する．また OpenACC プログラムには data，parallel お

表 5　評価に使用するベンチマーク
Table 5　Benchmark programs used for the evaluation.

	熱拡散シミュレーション			姫野ベンチマーク
	1D	2D	3D	
データ型	float			
空間格子の形状	4194304 ($=2^{22}$)	2048×2048	256×256×256	256×128×128
1 格子点あたりの演算数	4	6	8	34
配列数	2	2	2	14
アルゴリズムの BF 値	2.0	1.33	1.0	1.65

表 6　最良の性能パラメータと自動設定された性能パラメータ
Table 6　Best performance parameters and automatically-selected parameters.

	熱拡散シミュレーション			姫野ベンチマーク
	1D	2D	3D	
NVIDIA Tesla C2070				
最良のパラメータ設定（全探索）	(256,1,1,3)	(32,8,1,2)	(32,2,8,1)	(128,2,4,1)
提案手法によるパラメータ設定	(256,1,1,3)	(32,8,1,2)	(32,8,4,1)	(32,2,2,1)
Intel Xeon Phi 5110P				
最良のパラメータ設定（全探索）	(2048,1,1,2)	(2048,1,1,1)	(256,8,1,1)	(256,4,1,1)
最良のパラメータ設定（GPU 実行可能）	(1024,1,1,2)	(1024,1,1,1)	(256,4,1,1)	(256,4,1,1)
提案手法によるパラメータ設定	(1024,1,1,2)	(1024,1,1,1)	(256,1,1,1)	(32,2,1,1)



(a) GPU　　(b) Xeon Phi

図 6　実行性能の比較
Fig. 6　Performance comparison.

よび loop ディレクティブの挿入のみを行い，WG 形状や WI といった性能パラメータには PGI コンパイラのデフォルト値を使用する．また，移植容易性の評価においては，SB や TB が行われていない C 言語プログラムを基準として用いる．

提案手法で自動設定された性能パラメータ，および全探索によって発見された最良の性能パラメータを**表 6** に示す．表中の数字は，性能パラメータを (x, y, z, t) の形式で表している．また，性能評価の結果を**図 6** に示す．図 6 では，横軸にベンチマークが示されており，OpenCL プログラムで最良の性能パラメータを設定した場合に達成される実行性能に対して，各プログラムがどの程度の実行性能を達成できていたかを示す割合（**正規化性能**）が縦軸に示されている．棒グラフに添えられている数字は，実行性能を Gflop/s で示している．また，図中の Phi->GPU は，Xeon Phi 向けに調整された性能パラメータを用いて GPU 上でベンチマークを実行した場合の性能を示している．同様に GPU->Phi は，GPU 向けに調整された性能パラメータを用いて Xeon Phi 上でベンチマークを実行した場合の性能を示している．熱伝導シミュレーションを Xeon Phi 向けに最適化する場合，3 次元では $(x, y, z) = (256, 8, 1)$，それ以外では $(x, y, z) = (2048, 1, 1)$ が最良の WG の形状であることが全探索の結果より分かっている．Tesla C2070 では WG 内の WI 数を 1024 以下にする必要があり，Xeon Phi 向けに最適化された性能パラメータでは同プログラムを実行することはできない．このため，図 6 (a) の熱伝導シミュレーションの Phi->GPU は，GPU で動作可能な範囲で Xeon Phi の性能が最も高くなる性能パラメータ（表 6 参照）を用い，GPU で実行した場合の性能が示されている．

Xeon Phi での最良の性能パラメータで GPU が動作し

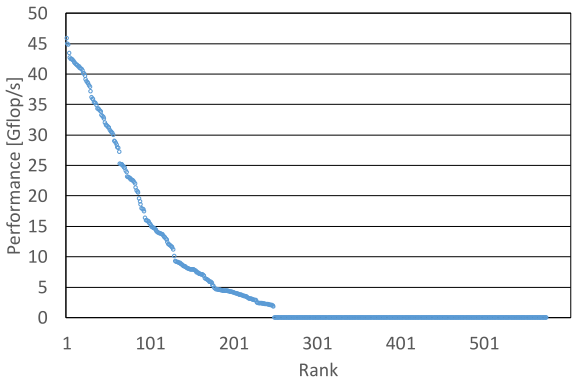


図 7　ワークグループサイズと性能との関係

Fig. 7　Relashionship between the work group size and performance.

ないことからも分かるとおり，特定のアクセラレータ向けに性能パラメータを設定した場合，他のアクセラレータでは動作しなくなる恐れがある．しかし，提案手法では各アクセラレータに合わせて性能パラメータを調整しているため，動作可能な性能パラメータを設定できる．また，提案手法ではアクセラレータの情報を考慮して WG の形状や TB のブロックサイズを調整しているため，図 6 (b) の熱伝導シミュレーションに見られるように，多くの場合，調整しない場合と比較して実行性能が向上している．これらの結果から，提案手法は特定のアクセラレータ向けに最適化された OpenCL プログラムよりも高い可搬性および性能可搬性を実現できていることが分かる．

図 6 (a) より，提案手法は OpenACC と比較しても高い実行性能を達成可能であることが分かる．これは，提案手法では TB の適用と性能パラメータチューニングを自動で行う一方で，OpenACC では SB しか適用されないためである．表 6 にも示されているとおり，本評価で用いた熱伝導シミュレーションの場合，3 次元では TB を用いない方が高性能であることが全探索の結果より分かっている．同様に，姫野ベンチマークでも TB を行わない方が高い性能を達成できる．これらのベンチマークに関しては，提案手法は TB を用いないという判断を適切に行い，OpenACC と同等の性能を達成している．これらの結果として，提案手法は OpenACC と比較して平均 1.58 倍の実行性能を達成できている．

GPU で姫野ベンチマークを実行する場合の，WG 形状と性能との関係を**図 7** に示す．縦軸は実行性能を示しており，横軸はその実行性能を達成した WG 形状の順位を示している．すなわち，図 7 は各 WG 形状による実行性能を左から降順に並べたものである．GPU 上で実行することのできない WG 形状の性能は 0 で表されている．この結果から，提案手法と同等の 40 Gflop/s を超える性能を達成できるのは上位の WG 形状を選択した場合のみに限られており，提案手法は準最適な性能パラメータを選択できていることが分かる．

これらの結果より，提案手法である性能パラメータ自動設定手法はアクセラレータおよびステンシル計算の性能パラメータを適切に調整できており，その自動調整によって性能可搬性を改善できることが明らかになった．

ただし，姫野ベンチマークにおいても可搬性と性能可搬性は実現できているものの，熱伝導シミュレーションよりも他手法との性能差が小さく，提案手法によって達成された実行性能が他よりも低い事例も見られる．これは，姫野ベンチマークがカーネル内でアクセスするデータ量が多いためである．姫野ベンチマークのカーネルには 14 個もの配列が引数として渡され，それぞれの配列の複数要素が各時間ステップでアクセスされている．このため必要データのオンチップメモリやキャッシュでの保持が困難であり，姫野ベンチマークはオーバラップタイリング [10] による TB では性能向上を期待できない計算であるといえる．このような場合，WG の形状を x 軸方向に拡大することによってオフチップのメモリアクセスが効率化され，性能が改善される傾向にあることが分かっている．GPU でも Xeon Phi でも同様の傾向が見られるため，Phi->GPU や GPU->Phi での性能低下は小さい．しかし，袖領域の増大を抑制するために，現在の提案手法では WG が細長くならないように形状を調整している．特に，本論文では GPU における WG 形状の x 軸方向への拡大の優先度は，Xeon Phi における優先度と比較して低くなっている．その結果として，姫野ベンチマークでは提案手法の優位性が顕著には見られない．この問題に関しては，ステンシル計算のデータサイズがオンチップメモリサイズと比較して大きい場合に WG 形状の x 軸方向への拡大を優先することによって比較的容易に解消できると考えられる．このように，本提案手法ではアーキテクチャの特徴をさらに考慮して成長ベクトルの設定規則を改良していくことができる．ただし，本論文執筆時点ではそのような改良は行っておらず，今後の課題である．

また，Xeon Phi は姫野ベンチマークにおいて比較的低い性能しか達成できていない．本論文で試作したトランスレータを Xeon Phi 向けに実用的に使うためには，本提案手法を用いて WG 形状や時間ブロックサイズを調整するだけでなく，他の性能最適化技法もコード変換の中に取り入れていく必要があると考えられる．

熱伝導シミュレーションの OpenCL プログラムの作成には，最適化されていない C 言語プログラムからの書き換えが 79 行から 163 行必要であり，元のプログラムのカーネル部分がわずか数行であることを考えると大規模なコード修正が必要である．一方で，提案手法では 7 行のディレクティブの挿入によって各アクセラレータ向けに最適化でき，最良の性能パラメータ設定と比較すると全ベンチマークプログラムの平均で 88.8%および 92.2%の実行性能を GPU

およびXeon Phiのそれぞれにおいて達成できている．元のC言語プログラムから比較すれば大幅な性能向上であり，わずかな労力で既存のプログラムを各アクセラレータ向けに性能最適化できることから，高い移植容易性を実現できているといえる．

OpenACCでは4行のディレクティブの挿入によってアクセラレータを利用できるものの，ステンシル計算に関する情報が与えられていないことから，TBのようなステンシル計算特有の性能最適化を行うことはできず，提案手法と比較して実行性能が低くなっている．これらの結果から，提案手法は性能パラメータ設定の自動化による高い性能可搬性，およびディレクティブの使用による高い移植容易性を両立できることが示された．

4.3　チューニングオーバヘッドに関する評価

以下では，提案手法によるチューニングオーバヘッドを評価する．性能パラメータを変化させながらプログラムを実行して性能を計測し，性能が最も高くなる性能パラメータの値を見つけることは理論上は可能である．しかし，性能パラメータ数の増加にともなって性能計測に要する時間も急速に増大し，実用的な時間内にパラメータ空間全体を探索することは困難な場合が多い．また，性能最適化の目的のみでは，システムを長時間利用することはできないという状況も考えられる．このため，性能パラメータ決定に要する時間であるチューニングオーバヘッドは，本研究の重要な評価項目である．

本評価では，提案手法によってコード生成をする場合と，最大実行性能を達成するプログラムを全探索によってコード生成する場合で，性能パラメータ決定までに要する時間を評価する．評価には，1次元空間から3次元空間における熱伝導シミュレーションをGPU向けに最適化するものとする．WG形状の全探索には時間がかかりすぎるため，それぞれの方向について2のべき乗の大きさ（2^n）であることを仮定して，計算格子サイズやアクセラレータのWGのサイズの上限を超えないようにnを変化させるパラメータ探索の時間を全探索のチューニングオーバヘッドと定義する．表3のPC0を用いてチューニングオーバヘッドを評価した結果を**図8**に示す．図8では，横軸にベンチマークおよび提案手法，縦軸にWG形状および時間ブロックサイズのパラメータを決定する場合に必要となる所要時間を示している．

各ベンチマークのチューニングオーバヘッドは，1次元，2次元，3次元ステンシル計算でそれぞれ7.0秒，365.9秒，28294.3秒である．一方，提案手法によるOpenCLプログラム生成の所要時間は，静的情報のみから性能パラメータを設定するため平均5.2秒程度である．言語処理系がScalaで実装されていることを考えれば，チューニングオーバヘッドをさらに削減するための実装上の工夫の余地も残されている．

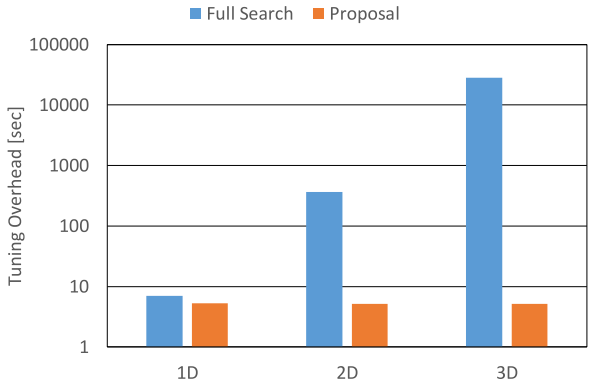


図8　チューニングオーバヘッドの評価結果
Fig. 8　Tuning overhead evaluation results.

本論文の提案手法では，全探索によって得られる最大実行性能に対して平均88.8%および92.2%の実行性能をGPUおよびXeon Phiのそれぞれで達成可能である．高性能アプリケーションの総開発期間という観点では，チューニングの所要時間が大きいことはアプリケーション開発の生産性を低下させる．また，限られた開発期間の中で，許容可能な実行性能を達成しなければならないような開発状況もしばしば考えられる．一方，提案手法ではチューニング時間が全探索する場合と比較して小さいため，提案手法は高性能プログラムの開発期間短縮に貢献可能であり，性能最適化に費やすことのできる期間や労力が限られた状況において特に有用である．

5.　おわりに

本論文では，ステンシル計算の性能最適化に必要な情報をディレクティブを用いて追記することによって，静的情報に基づいて性能パラメータを自動設定する手法を提案した．提案手法では，成長ベクトルと分割格子サイズ更新条件という，アーキテクチャ的な特徴を考慮可能な規則を定義して用いている．評価の結果，ステンシル計算の最適化で求められる性能パラメータの準最適な値を，静的情報のみに基づいて自動設定できることが明らかになった．静的情報の記述にディレクティブを用いることにより，通常のOpenCLと比べて約1/20の書き換え行数で高い実行性能を達成することが可能であり，提案手法によってプログラムの高い移植容易性を達成可能であることも示された．

本論文では独自の言語処理系を実装して提案手法の有効性を議論したが，性能パラメータ自動設定手法自体は他の言語処理系でも利用可能である．たとえばOpenACCの実装に提案手法を取り入れることで，OpenACCディレクティブとその拡張文法でテンポラルブロッキング等の性能最適化を適用できる可能性があり，本研究における今後の重要な課題である．

謝辞　本研究の一部は，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」および科研費基盤研究（B）#25280041 の支援を受けている．

参考文献

[1] Micikevicius, P.: 3D Finite Difference Computation on GPUs using CUDA, *GPGPU-2 Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, pp.79–84 (2009).

[2] Micikevicius, P.: GPU Performance Analysis and Optimization (2012), available from ⟨http://on-demand.gputechconf.com/gtc/2012/presentations/S0514-GTC2012-GPU-Performance-Analysis.pdf⟩.

[3] Saini, S., Jin, H., Jespersen, D., Feng, H., Djomehri, J., Arasin, W., Hood, R., Mehrotra, P. and Biswas, R.: An Early Performance Evaluation of Many Integrated Core Architecture Based SGI Rackable Computing System, *Proc. International Conference on High Performance Computing, Networking, Storage and Analysis, SC'13*, pp.94:1–94:12 (2013).

[4] OpenCL* Design and Programming Guide for the Intel Xeon Phi Coprocessor (2014), [Online] available from ⟨https://software.intel.com/en-us/articles/opencl-design-and-programming-guide-for-the-intel-xeon-phi-coprocessor/⟩.

[5] Khronos OpenCL Working Group: The OpenCL Specification version 1.1., available from ⟨http://www.khronos.org/opencl/⟩.

[6] Seo, S., Jo, G. and Lee, J.: Performance characterization of the NAS parallel benchmarks in OpenCL, *Proc. 2011 IEEE International Symposium on Workload Characterization* (*IISWC'11*), pp.137–148 (2011).

[7] OpenACC-standard.org, OpenACC Home, available from ⟨http://www.openacc.org⟩.

[8] Sabne, A., Sakdhnagool, P., Lee, S. and Vetter, J.S.: Evaluating performance portability of OpenACC, *International Workshop on Languages and Compilers for Parallel Computing* (*LCPC'11*), pp.51–66 (2014).

[9] The OpenMP API specification for parallel programming, available from ⟨http://openmp.org/wp/⟩.

[10] Holewinski, J., Pouct, L.-N. and Sadayapp, P.: High-Performance Code Generation for Stencil Computations on GPU Architectures, *Proc. 26th ACM International Conference on Supercomputing* (*ICS'12*), pp.311–320 (2012).

[11] Devito, Z., Joubert, N., Palacios, F., Oakley, S., Medina, M., Barrientos, M., Elsen, E., Ham, F., Aiken, A., Duraisamy, K., Darve, E., Alonso, J. and Hanrahan, P.: Liszt: A Domain Specific Language for Building Portable Mesh-based PDE Solvers, *Proc. 2011 International Conference for High Performance Computing, Networking, Storage and Analysis* (*SC'11*), pp.9:1–9:12 (2011).

[12] Datta, K., Murphy, M., Volkov, V., Williams, S., Carter, J., Oliker, L., Patterson, D., Shalf, J. and Yelick, K.: Stencil Computation Optimization and Auto-tuning on State-of-the-Art Multicore Architectures, *Proc. 2011 International Conference for High Performance Computing, Networking, Storage and Analysis* (*SC'08*), pp.1–12 (2008).

[13] Phillips, E.H. and Fatica, M.: Implementing the Himeno benchmark with CUDA on GPU clusters, *Proc. 2010 IEEE International Parallel & Distributed Processing Symposium* (*IPDPS'10*), pp.1–10 (2010).

[14] Odersky, M., Spoon, L. and Venners, B.: *Programming in Scala, 2nd Edition*, Artima Press (2010).

[15] Nugteren, C. and Codreanu, V.: CLTune: A Generic Auto-Tuner for OpenCL Kernels, *Proc. 9th International Symposium on Embedded Multicore-Many-core Systems-on-Chip*, pp.195–202 (2015).

[16] Komatsu, K., Sato, K., Arai, Y., Koyama, K., Takizawa, H. and Kobayashi, H.: Evaluating Performance and Portability of OpenCL Programs, *Proc. 5th International Workshop on Automatic Performance Tuning* (2010).

[17] Naono, K., Teranishi, K., Cavazos, J. and Suda, R.: *Software Automatic Tuning: From Concepts to the State-of-the Art Results*, Springer (2010).

[18] Kirk, D.B. and Hwu, W.W.: *Programming Massively Parallel Processors: A Hands-on Approach*, Morgan Kaufmann (2012).

付　　録

A.1　性能パラメータ設定手法の擬似コード

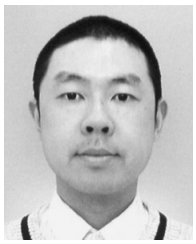
本論文で提案している性能パラメータ設定手法の擬似コードを以下に示す．

```
# system-specific information
INPUT BF_sys  # system B/F ratio
INPUT S_cache # cache size
INPUT N_wg    # lower bound of No. WGs

# parameters being tuned
pv  = {1,1,1,1}

ev  = {1,1,1,1} # evolution vector
dir = x         # direction of ev

# Repeat until all elements of ev become zero
REPEAT
  # compute temporal parameters
  CASE (device type)
    GPU: temp = compute_temp_gpu(pv,ev,dir)
    PHI: temp = compute_temp_phi(pv,ev,dir)
  ENDCASE

  # check if temp satisfies all the conditions
  # described in Section 3.2.
  IF (check_condition(temp)==true) THEN
    # use temp as new parameters
    pv = temp
  ELSE
    # don't repeat computing the same temp
    update_ev(ev,dir)
  ENDIF
WHILE (ev has a non-zero element)

STOP

# Compute temp vector for GPU
# See Section 3.3 for details.
```

```
compute_temp_gpu(pv,ev,dir)
  temp = pv

  IF (ev[x]!=0 && pv[x]<32) THEN
      temp[x] = 32
      dir = x
  ELSEIF (ev[t]!=0 && BF_sys>BF_alg) THEN
      temp[t] = pv[t] + 1
      dir = t
  ELSEIF (ev[x]!=0 && pv[x]==pv[z]) THEN
      temp[x] = pv[x]*2
      dir = x
  ELSEIF (ev[y]!=0 && pv[y]==pv[z]) THEN
      temp[y] = pv[y]*2
      dir = y
  ELSE
      temp[z] = pv[z]*2
      dir = z
  ENDIF

RETURN temp

# Compute temp vector for Xeon Phi
# See Section 3.3 for details.
compute_temp_phi(pv,ev,dir)
  temp = pv

  IF (ev[x]!=0 && pv[x]<32) THEN
      temp[x] = 32
      dir = x
  ELSEIF (ev[t]!=0 && BF_sys>BF_alg) THEN
      temp[t] = pv[t] + 1
      dir = t
  ELSEIF (ev[x]!=0) THEN
      temp[x] = pv[x]*2
      dir = x
  ELSEIF (ev[y]!=0 && pv[y]==pv[z]) THEN
      temp[y] = pv[y]*2
      dir = y
  ELSE
      temp[z] = pv[z]*2
      dir = z
  ENDIF

RETURN temp

# Check if temp satifies the conditions.
# See Section 3.2 for details.
check_condition(temp)
  COMPUTE S_tile     # Tile size (Figure 3)
  IF (S_tile > S_cache) THEN
    RETURN false
  ENDIF

  UPDATE BF_alg      # See Eq.(1)
  IF ( BF_sys < BF_alg) THEN
    RETURN false
  ENDIF

  COMPUTE number of work groups
  IF (number of work groups < N_wg) THEN
    RETURN false
  ENDIF

RETURN true

# update evolution vector
update_ev(ev,dir)
  ev[dir] = 0
  dir = undefined
STOP
```

角川 拓也

平成 26 年東北大学工学部機械知能・航空工学科卒業．平成 28 年東北大学大学院情報科学研究科博士前期課程修了．高性能計算，アクセラレータプログラミングとその応用に関する研究に従事．



平澤 将一　（正会員）

平成 12 年東京大学理学部情報科学科卒業．平成 14 年東京大学大学院理学系研究科情報科学専攻修了．平成 17 年東京大学大学院情報理工学系研究科コンピュータ科学専攻博士課程単位取得退学．平成 18 年電気通信大学大学院情報システム学研究科助手．平成 19 年同助教．平成 23 年同産学官連携研究員．平成 24 年東北大学大学院情報科学研究科産学官連携研究員．高性能計算システムとプログラミング言語処理に関する研究に従事．ACM，IEEE CS 各会員．

滝沢 寛之　（正会員）

平成 11 年東北大学大学院情報科学研究科博士課程修了．博士（情報科学）．同年新潟大学総合情報処理センター助手，平成 15 年東北大学情報シナジーセンター助手，平成 16 年東北大学大学院情報科学研究科講師を経て，平成 21 年より同研究科准教授．高性能計算システム．コンピュータアーキテクチャとその応用に関する研究に従事．電子情報通信学会，IEEE CS，ACM 各会員．

小林 広明　（正会員）

昭和 63 年東北大学大学院博士課程修了．同年東北大学助手．平成 3 年東北大学講師．平成 5 年東北大学助教授．平成 13 年東北大学教授．平成 20～28 年サイバーサイエンスセンター長兼任．平成 18 年より NII 客員教授併任．コンピュータアーキテクチャ，並列処理システムとその応用に関する研究に従事．工学博士．IEEE Senior Member，ACM，電子情報通信学会各会員．日本学術会議連携会員．