データレイアウト最適化のためのコード変換規則の自動生成

山田 剛史[1,a)] 平澤 将一[2,b)] 須田 礼仁[3,c)] 滝沢 寛之[2,d)]

概要：メモリアクセスパターンは高性能計算アプリケーションの性能に大きく影響するため，データレイアウト最適化は重要である．しかし，データレイアウト最適化のためにソースコードを書き換えることで，可読性や性能可搬性を損なう恐れがある．こうした問題は，アプリケーションにソースコード変換を適用することで解決できるが，性能チューニングを行うプログラマがソースコード変換の変換規則を記述する必要があることが課題となっている．本研究では，変換前後のデータ構造の定義から構造体メンバへのアクセス表現を自動で生成が可能であることに着目し，生成したアクセス表現をもとにアクセス表現の変換規則を自動で生成する手法を提案する．提案手法により，既存のソースコード変換手法と比較して少ない記述量で変換規則を定義し，データレイアウト変換の適用が可能になる．また，提案手法をアプリケーションに適用し，実行条件に適したデータレイアウトでアプリケーションを実行できることを示す．

キーワード：
データレイアウト最適化，ソースコード変換

1. はじめに

メモリアクセスパターンは，高性能計算 (High Performance Computing，HPC) アプリケーションの性能に大きく影響する．将来的には，HPC システムのメモリ階層はより複雑になり，メモリアクセスパターンの性能への影響は大きくなると予想される．

プログラミング言語が提供する配列や構造体などは，構造化されたデータ (**データ構造**) をソースコード中で表現するために使用される．HPC アプリケーションの開発で用いられる Fortran などの比較的低水準なプログラミング言語では，データ構造の表現が特定のメモリ空間内の配置である**データレイアウト**と厳密に対応づけられている．データレイアウトはメモリアクセスパターンに密接に関係しており，アプリケーションの性能に大きく影響するため，データレイアウトによっては HPC システムの性能を十分に引き出せない場合がある．例えば，特定の HPC システム向けにデータレイアウトが最適化されたアプリケーションを，HPC システムやデータサイズなどの条件を変更して実行すると性能が低下する場合がある．したがって，様々な実行条件で高い実行性能を達成するためには，データレイアウト最適化を実行条件に応じて適用する必要がある．

また，多くの場合において，プログラマにとって理解しやすい表現と，プロセッサが高速にアクセスできるデータレイアウトは異なっている．プログラマにとって理解しやすい表現は物理的な現象や理論に基づいた表現であり，プロセッサが高速にアクセスできるデータレイアウトはメモリアクセスの効率が高いデータレイアウトである．しかし，ソースコード中の表現と異なるデータレイアウトでのアプリケーションの実行は難しいため，プログラマにとって理解しやすい表現とプロセッサが高速にアクセスできるデータレイアウトの両立は困難である．

以上の背景から，可読性と保守性を維持したまま様々な実行条件で高い実行性能を達成するには，データレイアウト最適化をソースコード変換で実現する必要がある．本稿では，プログラマが定義した変換規則を用いて行うソースコード変換を対象として，変換規則の定義を容易にすることを目的とする．変換規則の一部を自動で生成することでプログラマからの入力減らし，誤った変換規則が入力される危険性を抑制する．本稿の目的のために，Fortran で書かれた HPC 向けアプリケーションの最適化を行うプログラマ (**性能エンジニア**) を対象として，性能エンジニアが容

1 東北大学 大学院 情報科学研究科
Graduate School of Information Sciences, Tohoku University
2 東北大学 サイバーサイエンスセンター
Cyberscience Center, Tohoku University
3 東京大学 大学院 情報理工学系研究科
Graduate School of Information Science and Technology, The University of Tokyo
a) tyamada@sc.cc.tohoku.ac.jp
b) hirasawa@sc.cc.tohoku.ac.jp
c) reiji@is.s.u-tokyo.ac.jp
d) takizawa@tohoku.ac.jp

© 2017 Information Processing Society of Japan

易にデータレイアウト変換を定義できるインターフェースを提案する．また，性能エンジニアの入力からソースコード変換の規則を生成する手法を提案する．

2. データレイアウト

データレイアウトの代表的なものとして AoS (Array of Structures) と SoA (Structure of Arrays)，DA (Discrete Array) がある．ステンシル計算のような特定のメンバを連続してアクセス表現する計算では，SoA および DA の方がキャッシュにデータが乗りやすい．その結果として，キャッシュヒット率が改善することで，AoS を用いた場合と比較して高速なデータアクセスを期待できる．一方，AoS は計算格子上の各点の物理量を 1 つの構造体内にまとめて保持することができる．そのため，計算格子上の各点を構造体単位で扱うことができ，プログラマにとって理解や管理がしやすいという利点がある．一方，SoA や DA でデータ構造を定義した方が高性能を達成できる特性をもつアプリケーションが多いため，可読性や保守性の低下を許容した上で SoA や DA を用いてアプリケーションを開発する場合が多い．したがって，可読性と保守性を考慮しつつ高性能を達成するためには，開発時のソースコード上の表現と実行時のデータレイアウトを分離し，実行前にデータレイアウト変換することで双方の利点を両立させる必要がある．

2.1 データレイアウト最適化

Fortran のような比較的低水準なプログラミング言語においてデータレイアウトを最適化するためには，ソースコードに大きく分けて 2 種類の変更を加える必要がある．

1 種類目は，*データ構造の宣言 (構造体宣言)* の変更である．本稿では，Fortran の派生型を含めてデータ構造を定義している型を構造体型を呼ぶ．構造体宣言によってデータ構造を定義し，データレイアウトが決定されるため，データレイアウトを変更するためには構造体宣言を変更する必要がある．厳密な型付けを行うプログラミング言語の場合，特定の変数やデータ構造の宣言は 1 箇所しか存在せず，他のコンパイル単位からのアクセスを許可をするインターフェース宣言も最大でコンパイル単位の数しか存在しないため，書き換え箇所は少ない．

2 種類目は，*構造体メンバへのアクセスの表現 (アクセス表現)* の変更である．科学技術シミュレーションなどの HPC アプリケーションでは，アクセス表現はソースコード上に数多く存在している．アクセス表現を変更するためには，ソースコードの中の多くの箇所を適切に書き換える必要があり，性能エンジニアに要求される労力が大きい．また，ソースコードの行数が増加するにつれ，アクセス表現の数も増加することが想定される．したがって，大規模アプリケーションでデータレイアウト最適化のためのソースコード書き換えを行う場合，性能エンジニアに要求される労力がさらに大きくなる．

3. 関連研究

特定の用途や対象システムという仮定の下でのデータ構造の変換は研究されており，データ構造の抽象化・変換を行う手法が数多く提案されている [1] [2]．データ構造の抽象化により，コンパイル時に自由にデータレイアウトを決定することが可能になる．しかし，既存のアプリケーションに適用する場合にはデータレイアウトと対応付けられたデータ構造の定義を抽象化するために，ソースコードの書き換えが必要となる．また，抽象化のために元のソースコードの言語から逸脱した文法での記述が必要となる場合は，性能エンジニアは新しい言語を取得する必要がある [3]．

昨今アクセラレータとして広く用いられている Graphics Processing Unit (GPU) のメモリアクセス性能は，メモリアクセスパターンに大きく影響を受けるため，GPU を対象としたデータレイアウト最適化に関する研究は数多く存在する [4] [5] [6]．こうした実行時にデータレイアウト最適化を行う手法では，データレイアウト変換のオーバヘッドをある程度隠蔽することはできるが，ソースコードの変更によってデータレイアウト最適化された理想的な実行性能と比較して，隠蔽しきれなかったオーバヘッドの分の性能損失が発生する．

以上の考察から，一般的な性能エンジニアがデータレイアウト最適化のためにアプリケーションのコードのデータ表現を変更する手法が必要とされていることがわかる．LLVM [7] や ROSE [8] 等のコンパイラ基盤では，ソースレベルの変換ツールを開発する目的には有用であるが，コンパイラの専門家ではない性能エンジニアがそうした変換ツールを開発するのは難しい．

既存のフレームワークを用いてデータレイアウト最適化のためのソースコード変換を実現する手法を，予備評価 [9] として行った結果を以下に示す．予備評価 [9] では，データレイアウト最適化のための指示を XSLT (XSL Transformations) [10] で記述する必要がある．しかし，この手法を用いるには，性能エンジニアが ROSE の抽象構文木 (Abstract Syntax Tree, AST) と XSLT の文法を熟知している必要があり，難易度が高い．また，XSLT による指示は冗長であるため，性能エンジニアの負担が大きい．

こうした問題を踏まえて，さらに抽象度の高い指示をディレクティブで実現する手法を予備評価 [11] として行った結果を以下に示す．この手法でデータレイアウト変換を実現するためには，構造体宣言の変換規則とアクセス表現の変換規則の 2 種類を記述する必要がある．構造体宣言の変換規則は，変換前後の構造体宣言を記述するだけであり，簡単なディレクティブで指示できる．一方，アクセス表現

© 2017 Information Processing Society of Japan

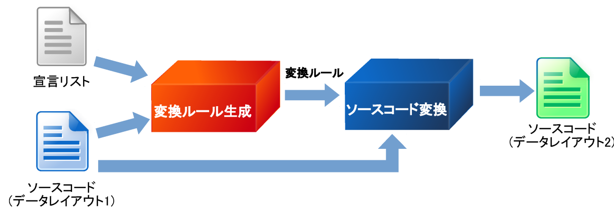

図 1　ソースコード変換の概要

の変換規則は，ソースコード内のアクセス表現に広く適用されるように考慮された記述を要求するため，高度なディレクティブ指示が必要になる．

4. データレイアウト最適化のためのコード変換

データレイアウト最適化に関する既存手法では用途や対象システムが限定されているため，本稿ではソースコード変換によってデータレイアウトを変換することを考える．既存のソースコード変換の手法 [9] [11] では，変換ルールの技術に高度な知識を要求するという問題があるため，本稿では一般的な性能エンジニアでも容易に利用可能なソースコード変換の実現を考える．

4.1 コード変換の概要

図 1 に，あるデータレイアウト 1 から別のデータレイアウト 2 へデータレイアウト変換を行うソースコード変換の概要を示す．変換ルールを生成するための入力は変換前のデータレイアウト 1 のデータ構造をもつソースコードと，**宣言リスト**である．宣言リストには，変換前のデータレイアウト 1 の構造体宣言と，変換後のデータレイアウト 2 の構造体宣言が Fortran の文法に基づいて記述されている．宣言リストをアプリケーションと同じ言語で記述することで，性能エンジニアが容易にデータレイアウト変換を定義できる．データレイアウト変換は宣言リストに基づいて行われるため，変換後のデータレイアウトを性能エンジニアが指定できる．

また，宣言リストはソースコードとは別のファイルに記述されており，ソースコードを直接変更することなくデータレイアウトを最適化することができる．したがって，実行条件に応じてデータレイアウト最適化の適用・不適用を選択したり，特定の実行条件の時に異なる宣言リストに基づいたデータレイアウト変換を適用する等の使い方が可能である．宣言リストの詳細は 4.2 節で述べる．

本稿では，ソースコードの変換を行うためにソースコード変換の規則 (**変換ルール**) を生成する．生成される変換ルールには，構造体宣言を置換するルールと，アクセス表現を変換するルールが含まれる．変換ルールの生成についての詳細は 4.3 節で述べる．

```
1  program datalayout1_to_datalayout2
2
3  !データレイアウト 1のデータ構造の宣言
4  !$xev dlopt before begin
5   !データレイアウト 1のデータ構造
6   ...
7  !$xev end dlopt before
8
9  !データレイアウト 2のデータ構造の宣言
10 !$xev dlopt after begin
11  !データレイアウト 2のデータ構造
12  ...
13 !$xev end dlopt after
14
15 end program datalayout1_to_datalayout2
```

図 2　宣言リスト

4.2 宣言リスト

宣言リストには，変換前の構造体宣言と変換後の構造体宣言の 2 種類を記述する．図 2 に，データレイアウト 1 からデータレイアウト 2 への変換を定義する宣言リストを示す．宣言リストには変換前の構造体宣言を記述し，その後に変換後の構造体を記述する．変換前後の構造体宣言において，構造体のメンバ数とそれぞれのメンバの名前は同じである必要がある．また，データ構造が次元をもつ場合，次元の数は変換前後で等しい必要がある．

変換前後を区別するための情報は，図 2 の 4，7，10，13 行目のようにディレクティブで指定する．

本来の Fortran の文法から逸脱した情報をディレクティブとして分離することで，性能エンジニアが主な編集箇所を視認しやすくなる効果を期待でき，ディレクティブ以外の部分は Fortran の文法に基づいて記述できる．さらに，宣言リストが Fortran の文法に対応することで，宣言リスト自体にソースコード変換を適用する等の応用が可能になる．

宣言リストに用いるディレクティブの一覧を表 1 に示す．表 1 のように，3 種類のディレクティブで変換前後の構造体宣言を記述する．`!$xev dlopt before begin` は，変換前の構造体宣言が始まることを示す．`!$xev end dlopt before` は，変換前の構造体宣言が終了することを示す．`!$xev dlopt after begin` は変換後の構造体宣言が始まることを示す．`!$xev end dlopt after` は，変換後の構造体宣言が終了することを示す．これらのディレクティブから，変換前の構造体宣言を検出し，変換後の構造体宣言に置換する．

4.3 コード変換ルールの生成

図 3 に宣言リストと変換ルールの概要を示す．入力として宣言リストとソースコードを用い，ソースコード変換のための変換ルールを生成する．生成される変換ルールには 2 種類のルールが含まれる．1 種類目は構造体宣言の置換

© 2017 Information Processing Society of Japan

表 1　ディレクティブ一覧

ディレクティブ	説明
!$xev dlopt before begin	変換前の宣言
!$xev end dlopt before	変換前の宣言の終了
!$xev dlopt after begin	変換後の宣言
!$xev end dlopt after	変換後の宣言の終了

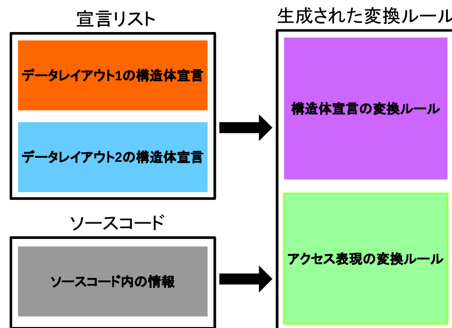

図 3　宣言リストと変換ルールの概要

を行う変換ルールである．構造体宣言の変換ルールによって，ソースコード内の構造体宣言は，変換後の構造体宣言へ差し替えられる．

2 種類目はアクセス表現の変換ルールである．ソースコード上に散在するアクセス表現は，アクセス表現の変換ルールによって変換後の構造体宣言に適合するアクセス表現へと変換される．本研究の予備評価 [9] [11] では，アクセス表現の変換ルールを性能エンジニアが定義していた．構造体宣言の置換ルールは，宣言リストの変換前の構造体宣言とソースコードを AST に基づいて厳密に照合するが，アクセス表現の変換ルールとして一般性の高いルールを定義することで，単一のルールでソースコード内の多くの箇所を変換することができる．しかし，こうした変換ルールの定義は，ソースコード変換の知識のない性能エンジニアにとって難易度が高い場合がある．このため本稿では，アクセス表現の変換ルールは変換前後の構造体宣言から生成できることに着目し，アクセス表現の変換ルールを自動生成する．アクセス表現の変換ルールを自動で生成することで，性能エンジニアの入力を変換前後の構造体宣言のみに絞り，容易にデータレイアウト変換を定義できるようにする．また，構造体宣言からアクセス表現の変換は一意に決定できるため，性能エンジニアが誤った変換ルールを定義する恐れを軽減できる．

4.3.1　構造体宣言の置換のための変換ルールの生成

構造体宣言を置換する変換ルールは，宣言リストから容易に生成できる．入力された宣言リストから変換前と変換後にあたる構造体宣言の部分を見つけ出し，変換前の部分と変換後の部分をソースコードで置換するような変換ルールを生成する．生成される変換ルールは，変換前の構造体宣言とソースコードとを両者の AST に基づいて比較し，変換前の構造体宣言の検出と削除，および変換後の構造体宣言を挿入する．

4.3.2　アクセス表現の変換のための変換ルールの生成

アクセス表現の変換ルールを生成は，アクセス表現の生成と変換ルールの生成の 2 段階に分けて行う．これは，アクセス表現を変換するための変換ルールは，変換前後のアクセス表現から生成されるためである．したがって，まずアクセス表現を生成し，生成されたアクセス表現からアクセス表現の変換ルールを生成する必要がある．変換ルールは，単一アクセス，範囲アクセス，組み込み関数それぞれに対して生成する．

4.3.2.1　アクセス表現の生成

アクセス表現は，宣言リストとソースコードの情報を用いて生成される．このアクセス表現は，変換前後それぞれについて構造体のメンバの数だけ生成される．アクセス表現の生成は，データ構造の取得とすべてのアクセス表現の列挙の 2 段階で行う．

4.3.2.1.1　データ構造の取得

アクセス表現を生成するためには，変換前後のデータ構造を解析する必要がある．本稿では，データ構造の各メンバが持つ属性として，データ構造のメンバの名前を `name`，データ構造のメンバの次元を `dim`，データ構造のメンバの型の名前を `type` と定義する．これらの属性をもとに，データ構造のメンバは構造体，配列，スカラ変数の 3 種類に分類される．

この定義に基づいて宣言リストとソースコードからデータ構造の各メンバの属性を取得するさらに，データ構造を取得し，データ構造の属性に基づいてデータ構造を 3 種類に分類する．データ構造が `type` を持っている場合，そのデータ構造は構造体である．データ構造が `type` を持たず，`dim` が 1 以上の場合，データ構造は配列である．データ構造が `type` を持たず，`dim` が 0 の場合，データ構造はスカラ変数である．データ構造が構造体である場合，構造体内部にもデータ構造が含まれているため，データ構造の取得は再帰的に行われる．

4.3.2.1.2　すべてのアクセス表現の列挙

取得したデータ構造の情報から，アクセス表現を生成する．データ構造のメンバがスカラ変数である場合，アクセス表現は変換する必要がない．データ構造が変数定義されている配列の場合は，アクセス表現は `name(dim)` となる．また，データ構造が構造体の場合は，アクセス表現は `name(dim)%`となり，構造体内のメンバに対応したアクセス表現を `type` に応じて追記していく．構造体の `dim` が 0 の場合は，`name%`のようなアクセス表現になる．また，構造体の中に構造体があるような入れ子構造を持つデータ構造の場合は，アクセス表現は `name(dim)%name(dim)%...` と続いていく．

データレイアウト変換されたデータ構造のアクセス表現

全てにソースコード変換が適用される変換ルールを生成するためには，取得したデータ構造の取りうる全てのパターンに対してアクセス表現を列挙する必要がある．そのため，データ構造が構造体の場合，アクセス表現は再帰的に生成する．構造体のメンバ変数との関係は木構造で表現することができ，宣言リストで変数宣言されている構造体が木構造の根のノード，構造体のメンバが子ノードとなる．アクセス表現を生成するために，木構造で最も深いノードから根ノードに到達するまでに辿ったノードの経路を求める．そして，求めた経路をもとに根ノードから最も深いノードへたどる．このとき，それぞれのノードに対応するデータ構造からアクセス表現を生成される．

データ構造が構造体かつ内部の要素に同じ type をもつ構造体が存在しているリスト構造をもつ場合は，データ構造の取得とアクセス表現の列挙は無限に実行されてしまう．そのため，自己の参照を検出した場合は，データ構造の取得とアクセス表現の列挙をそこで停止する．

4.3.2.2 変換ルールの生成

アクセス表現のための変換ルールは生成されたアクセス表現と，ソースコードの情報に基づいて生成される．変換後のデータ構造が取りうるアクセス表現のそれぞれについて，生成した変換前のアクセス表現と対応付けることによって，変換ルールが生成される．

4.3.2.2.1 単一アクセスの変換ルールの生成

単一アクセスは，特定のメンバ変数 1 個に対してアクセスする表現であり，ソースコード内で頻出する傾向にある．単一アクセスのアクセス表現は，ソースコード内の変換前のアクセス表現を探し出し，変換後のアクセス表現に置き換えることで変換できる．このとき，アクセス表現のインデックスは実数や変数を含めた任意の式を取りうる．そのため，インデックスは任意の式として，変換前のインデックスを変換後のインデックスに代入する変換ルールを生成する．

4.3.2.2.2 範囲アクセスの変換ルールの生成

範囲アクセスは，配列全体に対してアクセスする表現で，配列の全要素に同一の値を代入する際などに用いられる．こうした処理を構造体を含むデータ構造で実現する場合，ループ生成が必要である．また，ループ生成をする際，ループの上限・下限が必要となる．ループの上限・下限は構造体もしくは配列のデータサイズであるため，ソースコード中のデータ構造の定義もしくは allocate 文から取得する．

取得したループの上限，下限の情報に基づいて，ループ生成を行う変換ルールを生成する．変換ルールには，変換前の演算対象を，変換後の演算対象として代入する機能が含まれる．

4.3.2.2.3 組み込み関数向けの変換ルールの生成

データのメモリ領域を動的に確保するアプリケーションの場合，メモリ確保と開放のための組み込み関数が含まれる．allocate 文には，データ構造の変数名とデータサイズの情報が必要であり，deallocate 文には，データ構造の名前が必要である．データ構造の名前は宣言リストとソースコードから取得する．また，allocate 文と deallocate 文はメモリ領域を確保する構造体や配列の数に応じて存在するため，変換前後でメモリ確保する構造体や配列の数が異なる場合は，allocate 文と deallocate 文の数が異なる．そこで，変換前後のデータ構造に基づいて allocate 文と deallocate 文の数を決定し，allocate 文と deallocate 文を置換する変換ルールを生成する．allocate 文に必要なデータサイズの情報は，変換前の allocate 文に含まれている．したがって，変換ルールには変換前の allocate 文からデータサイズの情報を変換後の allocate 文に代入する機能が含まれる．

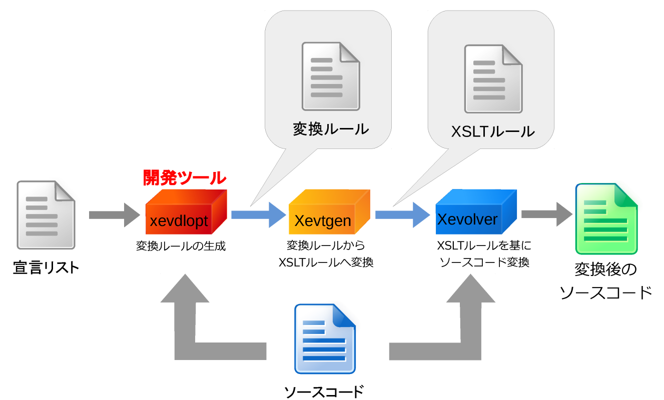

図 4 提案手法の実装の概要

4.4 実装

図 4 に提案手法の実装の概要を示す．まず，宣言リストとソースコードは，変換ルール生成ツールとして新規開発した xevdlopt に入力される．xevdlopt は，入力されたファイルを AST 解析し，4.3 節で述べた手法に基づいて Xevtgen[12] 向けの変換ルールを生成する．この時，出力された変換ルールには構造体宣言の置換とアクセス表現の変換を行う変換ルールが含まれる．この変換ルールは Xevtgen に入力することで，XSLT ルールへと変換される．その後，XSLT ルールと変換前のソースコードを Xevolver[13] に入力して，データレイアウト変換されたソースコードが出力される．

xevdlopt は Xevtgen を含むツール群 Xevtools のひとつのツールとして，C で実装されている．Fortran 形式で入力される宣言リストとソースコードは，Xevtools を利用して AST 化される．AST 化された入力ファイルは，AST 解析されて変換ルールとして出力される．

構造体宣言の置換ルールの生成処理では，宣言リストのディレクティブを変更する．Xevtgen で用いられるディレクティブ仕様に基づいたディレクティブを変更することで，変換前の構造体宣言から変換後の構造体宣言へ置換す

表 2　評価アプリケーション

アプリケーション	データサイズ	データレイアウト
姫野ベンチマーク	XS, S, M, L, XL	DA / AoS
PolyBench syr2k	32^2,128^2,1024^2,2000^2,4000^2	DA / AoS

表 3　評価環境

Intel Xeon E5-2630	コンパイラ	GNU gfortran 4.4.7
	最適化オプション	-O3
NVIDIA Tesla K20c	コンパイラ	pgfortran 15.7
	最適化オプション	-acc
NEC SX-ACE	コンパイラ	sxf90
	最適化オプション	-P auto -C hopt
	ジョブクラス	sx32 / 1 ノード

る変換ルールとなる．

アクセス表現の変換ルールの生成では，アクセス表現の生成と変換ルールの生成の 2 段階に分けて行う．まず，データ構造の解析を行い，解析されたデータ構造の情報に基づいてアクセス表現を列挙することでアクセス表現を生成する．本稿の実装は，生成したアクセス表現をもとに単一アクセスのための変換ルールを生成する．変換ルールは，Xevtgen のディレクティブを用いて`!$xev tgen trans exp src`（‘変換前のアクセス表現‘）`dst`（‘変換後のアクセス表現‘）と定義できる．このディレクティブを生成するためには，アクセス表現すべてに対してデータ構造の名前に基づいて変換前後のアクセス表現を対応付ける必要がある．本稿の実装では，変換前後でデータ構造が同じ名前を持っていると仮定して，変換前のアクセス表現と同じ名前の変換後のアクセス表現を対応付けている．対応付けられたアクセス表現の組み合わせから，ディレクティブ内にそれぞれのアクセス表現を出力することで変換ルールを生成する．

5.　評価

5.1　評価環境

性能評価するアプリケーションを表 2 に示す．アプリケーションは動的メモリ確保版の姫野ベンチマーク [14] と，PolyBench [15](Fortran 版) の syr2k を使用する．姫野ベンチマークと syr2k において，配布されているソースコードのデータレイアウトは DA である．評価では，提案手法を用いてデータレイアウトを AoS に変換したソースコードも用いて性能を比較する．

評価環境を表 3 に示す．姫野ベンチマークでは，Intel Xeon E5-2630 と NEC SX-ACE を用いて評価を行う．また，syr2k では Intel Xeon E5-2630 と NVIDIA Tesla K20c を用いて評価を行う．NVIDIA Tesla K20c では，syr2k のカーネル部分に OpenACC [16] のディレクティブを挿入して実行する．

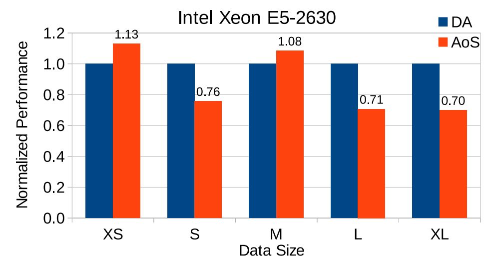

図 5　姫野ベンチマークの Intel Xeon E5-2630 での実行性能

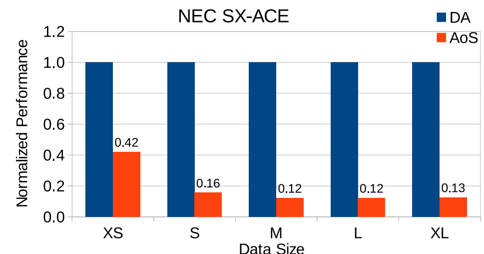

図 6　姫野ベンチマークの NEC SX-ACE での実行性能

5.2　評価結果

5.2.1　姫野ベンチマーク

姫野ベンチマークでソースコード変換前後での性能変化を評価する．図 5 と図 6 に姫野ベンチマークの評価結果を示す．図 5 と図 6 の横軸はデータサイズであり，縦軸は DA の実行性能を 1.0 として正規化した実行性能を示している．

図 5 の Intel Xeon E5-2630 において，AoS がデータサイズ XS では 13%，データサイズ M では 8%高速であり，他のデータサイズでは DA が高速である．図 6 の SX-ACE では DA がすべてのデータサイズで高速であり，Intel Xeon E5-2630 と比較して性能差が大きい．これは，DA ではメモリのバンクコンフリクトの時間が大きく削減され，メモリアクセス効率が向上しているためである．

姫野ベンチマークにおいては，Intel Xeon E5-2630 のデータサイズ XS，M は元のデータレイアウトの DA から AoS にソースコード変換を行い，他の実行条件ではソースコード変換を行わずにアプリケーションを実行することで，全ての実行条件で高い実行性能を達成できる．このように，実行条件に応じてデータレイアウト変換を適用することで，全ての実行条件で適切なデータレイアウトでアプリケーションを実行できる．

また，2 節で議論したように，AoS は計算格子上の各点の物理量を 1 つの構造体内にまとめて保持することができるため，可読性・保守性に優れ物理的な事象を直感的に表現できる．しかし，SX-ACE のようなシステム上で姫野ベンチマークのようなステンシル計算を行うアプリケーションを実行する場合，データレイアウトは DA でアプリケーションを開発することが好ましい．したがって，AoS のソースコード表現でアプリケーションを開発し，実行前に提案手法を用いてメモリアクセス効率の高いデータレイア

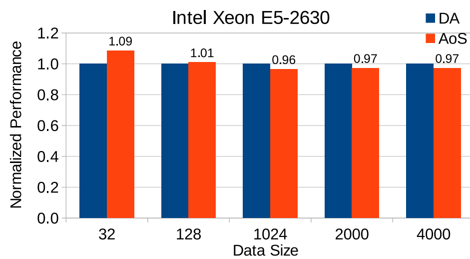

図 7 syr2k の Intel Xeon E5-2630 での実行性能

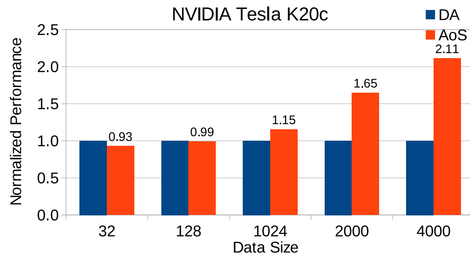

図 8 syr2k の NVIDIA Tesla K20c での実行性能

ウトに変換することでソースコードの可読性・保守性を維持したまま高性能を達成できる．

5.2.2 syr2k

次に，syr2k でソースコード変換前後の性能変化を評価する．図 7 と図 8 に syr2k の評価結果を示す．図 7 と 8 の横軸はデータサイズであり，縦軸は DA の実行性能を 1.0 として正規化した実行性能を示している．

図 7 の Intel Xeon E5-2630 において，AoS がデータサイズ 32 では 9%，データサイズ 128 では 1%高速であり，データサイズ 1024 以上では DA が高速である．図 8 の NVIDIA Tesla K20c において，データサイズ 1024 以上では，AoS が高速である．これは，カーネルの実行性能がデータサイズ 32 を除いて向上をしているためである．

syr2k において，Intel Xeon E5-2630 のデータサイズ 32 と 128，NVIDIA Tesla K20c のデータサイズ 1024 では，提案手法を用いてデータレイアウト変換を適用し，AoS で実行すると DA よりも高い実行性能を達成できる．実行条件によって適切なデータレイアウトは異なるが，提案手法を用いることで複数のデータ構造を条件に合わせて切り替えてアプリケーションを実行できる．

5.3 考察

5.3.1 データレイアウト変換による性能可搬性の向上

5.2 節で述べたように，アプリケーションの実行性能は実行条件とデータレイアウトに大きく影響される．特定のデータサイズやシステム向けにデータレイアウト最適化されたアプリケーションの実行条件を変更した時に，システムの潜在性能を引き出せない恐れがある．こうした場合，新しい実行条件のためのデータレイアウト最適化を行う必要があるが，手作業でソースコードを書き換えるのは性能エンジニアへの負担が大きい．本稿で示した手法では，性能エンジニアは少ない負担でアプリケーションにデータレイアウト最適化を適用することができる．

表 4 性能エンジニアからの入力の比較

	Xevolver[9]	Xevtgen[11]	提案手法
文法	XSLT	Fortran	Fortran
構造体宣言の置換指示	必要	必要	必要
アクセス表現の変換指示	必要	必要	不要

また，宣言リストはソースコードから分離されているため，アプリケーション本体のソースコードは編集せずにデータレイアウト変換を適用できる．例えば，特定の実行条件への最適化を行っていないアプリケーションのソースコードを性能エンジニアが提供を受ける場合，性能エンジニアが HPC システムやデータサイズに合わせて任意のデータレイアウトの最適化を適用できる．また，性能エンジニアがアプリケーションを拡張して再配布するような場合，元のソースコードとデータレイアウトは変わっていないため，他の利用者は元のソースコードと同じデータレイアウトのソースコードを受け取ることができる．このように，データレイアウト最適化とソースコードを分離することで性能可搬性の向上を期待できる．

5.3.2 変換ルール生成の生産性

Xevolver と Xevtgen，提案手法を用いて同じデータレイアウト変換を行う際に記載が必要な事項について表 4 にまとめる．Xevolver を用いた場合では変換ルールを XSLT で指示する必要があるが，提案手法で入力される性能エンジニアの指示は，アプリケーションと同様の Fortran の文法と 4.2 節で述べた単純なディレクティブのみである．宣言リストの変換前の部分はソースコードから複製してディレクティブを挿入するだけで指示できる．また，宣言リストの変換後の部分は，アプリケーションのソースコードと同じ文法で変換ルールを記述できるため，性能エンジニアの負担を削減することを期待できる．

また，Xevtgen を用いた場合では，性能エンジニアによるアクセス表現の変換ルールの指示が必要である．アクセス表現の変換ルールは，Xevtgen のディレクティブの文法に沿って指示することが必要であり，構造体宣言を置換する変換ルールと比較して指示が難しい．しかし，表 4 に示すように，提案手法ではアクセス表現を変換する指示を性能エンジニアが指示する必要がなく，アクセス表現を変換する変換ルールを自動で生成することで性能エンジニアの負担を減らしている．

例えば，姫野ベンチマークのデータレイアウトを DA から AoS に変換する場合，アクセス表現は 7 パターンとなり，単一アクセスの変換ルールは 7 パターンが自動で生成される．また，syr2k のデータレイアウトを DA から AoS に変換する場合，アクセス表現は 3 パターンとなり，単一アクセスの変換ルールは 3 パターンが自動で生成される．

© 2017 Information Processing Society of Japan

構造体メンバの数の増加などでデータ構造が複雑化した場合，取りうるアクセス表現のパターンは増加するが，提案手法はアクセス表現の変換ルールを書く必要がないため，性能エンジニアの負担を大きく削減できる．

6. おわりに

HPC アプリケーションにおけるデータレイアウト最適化は重要である．しかし，データレイアウト最適化にはソースコードの書き換えが伴い，可読性や性能可搬性を損なう恐れがある．こうした問題はソースコード変換で解決が可能であるが，予備評価では全ての変換ルールを性能エンジニアが記述する必要があることが課題となってきた．本稿では，変換前後の構造体宣言からアクセス表現の変換ルールを自動で生成が可能であることに着目し，アクセス表現の変換ルールをアクセス表現から生成する手法を提案した．

評価により，データレイアウト最適化を実行条件に応じて適用することで高い実行性能を達成できることを示した．また，可読性と保守性に優れた表現でアプリケーションの開発を行い，実行する前にソースコード変換によるデータレイアウト最適化を適用することで，高い実行性能を達成できることを示した．さらには，予備評価と比較して性能エンジニアへの負担の少ない方法でデータレイアウト最適化を指示できることが示された．

今後の課題として，多次元配列のインデックス順を入れえかるなどのデータレイアウト変換のパターンの拡張が挙げられる．また，OpenACC などに用いられるディレクティブでは，データ転送のためにディレクティブ内にデータ構造の名前を記述する．したがって，転送するデータ構造が変化した場合は，ディレクティブを変更する必要があり，ディレクティブ変換を行う変換ルールが必要になる．

7. 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) 16H02822 の支援を受けている．また，本研究の性能評価には，東北大学サイバーサイエンスセンターの NEC SX-ACE システムが用いられた．

参考文献

[1] Xiong Fu, Yu Zhang, and Yiyun Chen, “Data-layout optimization using reuse distance distribution,emerging directions in embedded and ubiquitous computing,” vol. 4097, pp. 858–867, 2006.
[2] Qing Yi,“Optimizing and tuning scientific codes,“in Scalable computing and communications: Theory and Practice,” John Wiley&Sons, 2011.
[3] Jonathan Ragan-Kelley, Andrew Adams, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Frédo Durand, “Decoupling Algorithms from Schedules for Easy Optimization of Image Processing Pipelines,” ACM Transactions on Graphics (TOG) - Proceedings of ACM SIGGRAPH 2012, vol.31(4), pp. 1–32, 2012.
[4] I-Jui Sung, Geng Daniel Liu, and Wen-Mei W. Hwu, “Dl: A data layout transformation system for heterogeneous computing,”in Innovative Parallel Computing (InPar), pp. 1–11, 2012.
[5] Shuai Che, Jeremy W.Sheaffer, and Kevin Skadron, “Dymaxion: Optimizing Memory Access Patterns for Heterogeneous Systems,” Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC11), pp.1-11, 2011.
[6] Deepak Majeti, Kuldeep S. Meel, Rajkishore Barik, and Vivek Sarkar, “Automatic Data Layout Generation and Kernel Mapping for CPU+GPU Architectures,” Proceedings of the 25th International Conference on Compiler Construction (CC 2016), pp. 240-250, 2016.
[7] Chris Lattner, “LLVM: An infrastructure for multi-stage optimization,” Master’s thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, 2002.
[8] Dan Quinlan, “ROSE: Compiler support for object-oriented frameworks,” Parallel Processing Letters, vol. 10, no. 02n03, pp. 215–226, 2000.
[9] Takeshi Yamada, Shoichi Hirasawa, Hiroyuki Takizawa, and Hiroaki Kobayashi,“A Case Study of User-Defined Code Transformations for Data Layout Optimizations,” in Third International Symposium on Computing and Networking (CANDAR’15), pp. 535-541, 2015.
[10] Michael Kay,“XSLT 2.0 and XPath 2.0 Programmer’s Reference (Programmer to Programmer),” 4th ed. Wrox Press Ltd., 2008.
[11] Hiroyuki Takizawa, Takeshi Yamada, Shoichi Hirasawa, and Reiji Suda, “A Use Case of a Code Transformation Rule Generator for Data Layout Optimization”, The 24th Workshop on Sustained Simulation Performance, 2016.
[12] Reiji Suda, Hiroyuki Takizawa, and Shoichi Hirasawa, “Xevtgen: Fortran code transformer generator for high performance scientific codes,” in Third International Symposium on Computing and Networking (CANDAR’15), pp. 528–534, 2015.
[13] Hiroyuki Takizawa, Shoichi Hirasawa, Yasuharu Hayashi, Ryusuke Egawa, and Hiroaki Kobayashi, “Xevolver: An XML-based Code Translation Framework for Supporting HPC Application Migration,” IEEE International Conference on High Performance Computing (HiPC’14), pp. 1-11, 2014.
[14] Himeno Ryutaro,“Himeno benchmark,” RIKEN, [Online] Available: `http://accc.riken.jp/supercom/himenobmt/`, 2011.
[15] Louis-Noel Pouchet, Mohanish Narayan, “PolyBench the Polyhedral Benchmark suite,” [Online] Available: `http://web.cse.ohio-state.edu/~pouchet/software/polybench/`, 2012.
[16] OpenACC-standard.org,“The OpenACC Application Programming Interface Version 2.0,” [Online] Available: `http://www.openacc.org/`, 2013.

© 2017 Information Processing Society of Japan