# 対話的手法により最適化コードを生成する科学技術計算用 DSL

西田 秀之　千葉 滋

本研究は科学技術計算コード開発者のためにドメイン特化言語（DSL）に対話的コンパイラを組み込んだシステムを提案するものである．DSL は計算の本質と計算機上の効率化のための最適化を分離したデザインで設計されている．DSL は二段階の実行を採用しており，実行すると動的に最適化された科学技術計算コードを生成し，このコードを実行することで実際の計算を行う．同様のデザインを持つ従来のシステムでは，人的最適化と自動最適化の個別の限界により，コード最適化が困難なケースが存在した．本研究では，対話的手法を取り入れたコンパイルシステムを提案し，その課題を解決している．人的な解析が困難な情報はコンパイラが計算及び表示し，自動最適化が困難な複雑なケースはデータを参照しながら人間が指示する．

This study proposes a system that incorporates an interactive compiler in a domain-specific language (DSL) for scientific computing code developers. The DSL is designed to separate the nature of the computation from optimization for computational efficiency. The DSL employs a two-stage execution, where execution generates dynamically optimized scientific calculation code, and the actual computation is performed by executing this code. In conventional systems with similar designs, code optimization has been difficult in some cases due to the separate limitations of human and automatic optimization. In this study, we propose a compilation system that incorporates an interactive approach to solve this problem. The compiler computes and displays information that is difficult to analyze humanly, while complex cases that are difficult to optimize automatically are directed by humans while referring to the data.

## 1　はじめに

科学技術計算は計算によって科学の諸問題を解析し，解決する一分野である．特に近年のコンピュータ性能の向上や理論の深化に伴い，様々な計算対象を高精度・高速に計算できるため，実験的アプローチの先導的役割を担うようになってきている．しかし，コードの保守性・可読性を保ちつつ高速なアプリケーションを迅速に実装する難易度は依然として高く，計算対象やハードウェア毎の最適化なども合わせると多大な労力を要する．

ソフトウェア開発効率を向上させる手段として，ドメイン特化言語 (Domain Specific Language : DSL) [1] が近年注目されている．DSL は汎用言語 (C 言語や Python など) とは異なり，特定の領域での使用に特化している．DSL の実装形式の一つに内部ドメイン特化言語 (Embedded Domain Specific Language : EDSL) が存在する．EDSL はアプリケーションの主となる言語の機能を利用しつつ独自の記法や機能を実現する．科学技術計算コード開発者のための EDSL を設計することで，最適化されたコードを迅速に実装する環境を整えるアプローチが考えられる．

本論文ではまず科学技術計算と既存 EDSL の背景と問題点を説明し (2 節)，次に研究のアイデアと実例を説明する (3 節)．最後に結論 (4 節) を述べる．

## 2　Embedded DSL による半自動並列化

本節では，研究の基本的なアイデアを述べるに当たって必要な背景と既存研究の応用例を挙げる．科学

---

Interactive System to Generate Optimized Code for Scientific Calculation



Hideyuki Nishida, 東京大学, The University of Tokyo.
Shigeru Chiba, 東京大学, The University of Tokyo.

技術計算の一例として，量子化学計算を取り上げる．

### 2.1 量子化学計算

量子化学計算はコンピュータ技術の発展に伴い，研究面・工業面の広い範囲で取り扱われている．量子化学計算では古典的手法と異なり，電子の相関を露わに取り入れ，量子力学の振る舞いからエネルギー計算を行う．量子化学計算用アプリケーションの開発に当たって，実用性の観点からアプリケーションの高速性が求められる．しかし，高速なコードの記述には様々な難点があり，以下のように挙げられる．

1. 入力データの人的な把握管理が困難である．計算量は基本的に入力データのサイズに依存する．高速なコードを記述するためには入力データに合わせた実装が必要となる．量子化学計算では，入力データには主に計算手法・原子番号・座標・全体の電子数などの情報が使われる．しかし，実際の計算に用いるデータはそれだけでなく，原子情報から求められる電子の配置や関数の情報など原子数の数倍 ∼ 数十倍のデータが用いられる．つまり，暗黙的で巨大な入力データが存在するため，人的な計算量の把握が難しい．例えば，計算の高速化を目的として分子を部分系に分割して計算する場合を考える．部分系毎の計算量は属する電子数の累乗で求められるが，それに特化したコードは入力データ毎に異なり，データ自体の管理の困難さから最適な負荷分散も困難である．
2. 計算式自体が複雑である．量子化学計算で扱う電子は量子的な振る舞いをする．確率によって計算するため，n ** 2 を前提としている．n 対計算とも捉えることができ，ループの入れ子が深い構成になりがちである．行う計算も複雑であるため，どのループに対して負荷分散を行うべきかの判断が困難である．一般にループの入れ子は 10 以上に重なるため，総当たりで最適な負荷分散を求める試行は現実的ではない．
3. 高速化可能性が多岐にわたる．他の研究分野と同様に，量子化学計算においても高速化手法は様々開発されているため，計算毎に適した高速化手法を実装・選択する必要がある．例えば，OpenMP と MPI のハイブリッド並列を行うケースを考える．簡単のため，OpenMP と MPI はそれぞれ単一のループを並列化するためのマクロとする．特定の計算手法 A を実現するための並列化された擬似コードは以下となる．

```
#pragma MPI
for(y=0;y<MAXLEN2;y++){
  #pragma MP
  for(x=0;x<MAXLEN1;x++){
    --- inside algorithm ---
  }
}
```

計算手法 A の高速化のため，分割して計算する手法を取り入れたとする．(例えば分子を分割して部分系毎に計算する) その擬似コードは以下となる．

```
#pragma MPI
for(z=0;z<MAXLEN3;z++){
  #pragma MP
  for(y=0;y<MAXLEN2;y++){
    for(x=0;x<MAXLEN1;x++){
      --- inside algorithm ---
    }
  }
}
```

for 文で用いている変数を見ると分かるように，異なる for 文に対してマクロが使われている．最適化のためにプログラムを修正すると，計算手法毎に pragma を入れる位置を見直さなければならず，コードの管理の負担が大きい．Work stealing などの計算機の側面からの負荷分散手法を取り入れる場合でも，計算手法毎・負荷分散手法毎のコードが必要となる．

### 2.2 既存 EDSL による実装と問題点

高品質なコードの記述を補助するため，特定の領域に特化した EDSL が開発されている．例えば，Halide [2] は画像処理用の EDSL であり，可搬性の高いインターフェイスで知られている．Halide では，計算そのものの本質を示すアルゴリズム部と計算機への最適化のためのスケジューリング部を分離した記法をもつ．その記法は二段階コンパイル形式を取ることで実現されている．まず，ユーザが Halide を用いて

書いたコードは C++コードとして汎用コンパイラにコンパイルされる．出力されたバイナリの実行時に，ヒントを元に最適化された C++コードが出力され，Halide 独自のコンパイラが 2 回目のコンパイルを行う．例えば，Halide を簡略化した形で OpenMP を用いた並列化を考える．一般的な C++コードでは，以下のように記述する．

```
#pragma omp parallel
for(int y = 0; y < 100; y++){
  for(int x = 0; x < 100; x++){
    f[x][y] = x + y;
  }
}
```

対して，それと等価な Halide 風の記述は以下のコードとなる．

```
Halide::Func f;
Halide::Var x, y;
Halide::Buffer<int> output;

f(x, y) = x + y;
f.parallel(y);
output = f.realize(100, 100);
```

まず，Halide の特別な型 Func に，x + y の計算式自体を代入する．この時点では計算は行われない．次に parallel を用いて計算の並列化のヒントを与えている．realize 関数が呼ばれた時点でこれまでの記述を元に上に示された一般的な C++コードが出力され，一般の C++コンパイラでコンパイルして実行される．

このような Halide 風のシステムを科学技術計算向けに応用することを考える時に問題点が現出する．

1. 人的なヒント指定が困難である．2.1 節で述べたように，暗黙的な入力データが巨大で複雑な構造をもつため，人的な計算量の予測が困難である．Halide のような二段階コンパイル形式でも，最初にコードを書く時点で人間がヒント付けを行う必要がある．
2. 自動最適化が現実的な時間で実行できない．Halide には自動で最適化可能性を探索して並列化などがなされたコードを出力する機構がある．しかし，複雑なデータ・複雑な計算手法を用いているケースでは探索時間が爆発してしまうため，複雑な科学技術計算への適用は困難である．

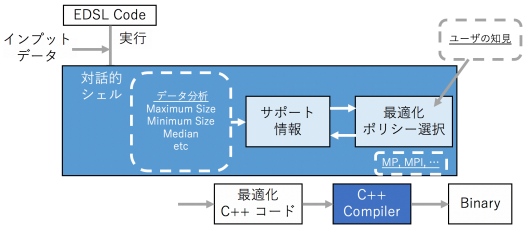


**図 1　システム概念図**

## 3　対話的コンパイラを組み込んだ DSL

本研究では，ユーザとコンパイラの知見を相補的に取り込むことでコード最適化を支援する環境構築を目指す．具体的には，二段階コンパイル形式の EDSL に対話的手法を取り込む．

### 3.1　対話的コンパイラによるコード出力

システムの概念図を図 1 に示す．本システムは 2.2 節で取り上げた二段階実行方式を採用する．一段階目の実行では，対象の計算は行われず計算式のみが保持され，二段階目の実行コードを出力する．二段階目の実行で初めて計算結果の取得ができる．本システムの特徴は，一段階目の実行時に起動する独自コンパイラに対話的シェルを用いることである．コンパイラは事前に入力データを読み込むことで，対話的シェルを介して推測される計算量やデータサイズなどをユーザに提供する．これは，巨大で複雑なデータなどの人的な取り扱いが困難な部分をコンパイラが吸収する機構である．ユーザは自身の知見とコンパイラのサポート情報から，どの計算部分にどの最適化を組み込むかをポリシーを選択する形で決定する．これにより，コンパイラ単独では最適化探索が爆発してしまうケースにも適用が可能となる．

システムの実行は次の 3 ステップで行う．

1. EDSL を用いたコードの記述と実行．科学技術計算コード開発者は EDSL をライブラリとして利用し，EDSL の記法でコードの記述を行う．Halide と同様に，計算の本質を示す部分と計算機のための最適化ポリシー部分を分離した記述を行う．これにより，コード改修の試行錯誤を容

$$result[grand] = (\sum_{child1 \in parent1} \sum_{child2 \in parent2} child1.x + child2.x : parent1,2 \in grand)$$

**図 2　計算式**

易とする．このコードの実行時点では，対象となる計算式自体が読み込まれ，計算の実行は行われていない．

2. 対話的コンパイラを用いた最適化ポリシー選択．コードの実行により，独自に組み込まれたコンパイラが計算式と最適化ポリシー情報を読み込む．対話的最適化を行うケースの場合に対話的シェルが起動する．このシェルでは計算式や入力データ情報，各種サポート情報を確認することができる．確認した情報を元にユーザは最適化ポリシーを選択する．
3. コード出力と計算の実行．対話的シェルの実行が終了した後，読み込まれた計算式と選択された最適化ポリシーから適した C++の最適化コードが出力される．そのコードを改めてコンパイルして実行することで，計算の結果を得られる．

### 3.2　プロトタイプ DSL の設計

本研究では，システムの設計と要点確認のためにプロトタイプとなる DSL を設計した．プロトタイプ DSL は Ruby の EDSL の形で実装を行なった．出力するのは実行速度の観点から C++コードである．構文の説明のため，3 次元の array of array で要素数が配列毎に異なるケースで，計算式図 2 を実装する場合を考える．

このステップで記述するコード例を以下に示す．

```
1  input.set_data("./data.dat")
2  input.has(grand.has(
3           parent.has(child)))
4  child.is(child1).is(child2)
5  parent.is(parent1).is(parent2)
6
7  # ---- algorithm ----
8  calc1[child1, child2] ==
9  child1.attr("x")
10 + child2.attr("x")
11 calc2[parent1, parent2] ==
12 calc1[child.in(parent1),
13         child.in(parent2)]
14 + calc1[child.in(parent1),
15          child.in(parent2)]
16 calc3[grand] ==
17 calc2[parent.in(grand),
18        parent.in(grand)]
19 # ---- end algorithm ----
20
21 # ---- scheduling ----
22 calc1.interactive_optimize()
23 calc2.interactive_optimize()
24 # ---- end scheduling ----
25
26 # ---- compile ----
27 calc3.compile()
28 # ---- end compile ----
```

独自にデザインした記法を用いているが，要点のみ説明する．1 から 5 行目はデータ準備を行なっている．特別な型に入力データを読み込ませ，使用するデータの次元情報の構築を行う．8 から 18 行目は計算式の記述である．図 2 の内側部分から順に記述を行う．calc1 が child1.x + child2.x に，calc2 が Σ に相当する．22, 23 行目は最適化ポリシーの選択となる．対話的な最適化を行いたい場合には，計算式毎に interactive_optimize() を呼び出すことで，独自コンパイラの実行時に対話的シェルを起動させる．27 行目は組み込まれた独自コンパイラの起動命令である．

### 3.3　対話的最適化の例

対話的シェルを用いた最適化を 3.2 節のコードを実行したと仮定して逐次的に説明する．27 行目の calc3.compile() が実行された時点で対話的シェルが起動し，以下の出力がなされる．

```
----
compile start
----

calc1 interactive optimize :
  argument : child1, child2
  usage : calc2
  max child size : child1:leaf,
    child2:leaf
  minimum child size : child1:
    leaf, child2:leaf
  median : child1:leaf, child2:
    leaf
```

```
calc2 interactive optimize :
  argument : parent1, parent2
  usage : calc3
  max child size : parent1:1000,
    parent2:1000
  minimum child size : parent1:5,
     parent2:5
  median : parent1:7, parent2:7

I suppose MP to calc1
I suppose MPI, work_steal to
  calc2

calc2>
```

対話的シェルの開始時には計算式や入力データの基本情報が出力される．ここで，使用する引数や計算式の使用元，子となる配列データの基本情報を確認する．基本情報の後には推奨する最適化ポリシーが出力される．特にこのケースでの calc2 では，array of array の要素数に大きな隔たりが確認されているため，Work steal の推奨がされる．

```
calc2> pwd
calc3(grand)
  |
  |-calc2(parent1, parent2) <-
      |
      |-calc1(child1 in parent1,
        child2 in parent2)
      |
      |-calc1(child1 in parent1,
        child2 in parent2)
```

現在作業している計算式と計算式の木構造を pwd で確認する．対話的コンパイルを指定した計算式の中から，より根に近い計算式を作業計算式として始まる．

```
calc2> select parent1[0]
{
  child: object[30],
         argument of calc1
  x: int(10),
  y: int(20),
}
```

作業計算式の引数に対して，詳細情報を確認する．parent1 の一つの要素に select を用いて構成要素を表示すると，child，x，y の三つの要素を保持した引数であることが分かる．この parent1 の child が calc1 の引数として用いられてるオブジェクトであることが分かる．

```
calc2> search maxlen child from
  parent1
parent1[20] has child[1000]

calc2> search maxlen child from
  parent1 except parent1[20]
parent1[19] has child[960]

calc2> search group child from
  parent1 where len > 500
total number 200
parent1[1, 14, 19, 20, 50, 74,
  ...]

calc2> use MPI with work_steal
```

要素数の偏りを search を用いて確認する．基本情報で確認した中央値 (median) よりも非常に大きいものが複数確認できる．単純な MPI で calc2 を並列化すると作業量に偏りが出ると予想できるので，Work steal を用いた並列化を use で指定する．

```
calc2> cd calc1
calc3-calc2-calc1

calc1> select child1[0] from
  parent1[0]
{
  x: int(1),
  y: int(2),
}

calc1> use MP
```

次に calc1 に対して作業するために cd で移動する．child1 の要素を確認してみると，x と y の要素を保持していることが分かる．計算量に差が生じづらく，最内面の計算式であるので MP の並列化を指定した．

```
calc1> end

calc1 optimize : MP
calc2 optimize : MPI, work_steal
----
compile end
----
```

end によって対話的シェルを終了する．calc1 と calc2 にそれぞれ MP，MPI work_steal が適用されていることを確認する．対話的シェルの終了後は最適

化ポリシーに則った C++コードが出力される．

## 4 まとめと今後の課題

科学技術計算の中でも量子化学計算を取り上げ，高速なコードの取得に関する問題点を挙げた．既存の DSL を拡張し，対話的にヒント付けができる形式を取り入れることで，人間とコンパイラが相補的にコード最適化を行えるシステムを提案した．このシステムでは二段階のコンパイル形式を採用しており，人的なヒント付が困難な部分をコンパイラの機能で補助し，コンパイラの自動最適化が困難な部分に人的なヒントを取り込む形で最適化コード生成への利便性を向上させている．今後の課題としては，コンパイラの補助部分の利便性の議論を行う必要がある．特に，どの最適化を推奨するかのアルゴリズムは新規開発と検証が必須と考えている．また，現在は EDSL の開発者があらかじめ組み込んだ最適化のみ用いることができるが，EDSL のユーザが独自最適化を組み込める機構のデザインも議論すべきである．

## 参 考 文 献


[1] Arie Van Deursen, Paul Klint, and Joost Visser.: Domain-specific languages: An annotated bibliography. *Sigplan Notices*, Vol. 35, No. 6(2000), pp. 26–36.

[2] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe.: Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *ACM SIGPLAN Notices*, Vol. 48, No. 6(2013), pp. 519-530.