日本ソフトウェア科学会第 29 回大会 (2012 年度) 講演論文集

HPC アプリケーションの OOP を用いたパフォーマンスチューニング

穂積 俊平　伊尾木 将之　千葉 滋

本論文では，高性能計算 (HPC) にオブジェクト指向プログラミング (OOP) を導入する事で，粒度の細かいパフォーマンスチューニングができる事を述べる．HPC では，良い実行性能を得るために実行環境に合わせたパフォーマンスチューニングが求められる．しかし，HPC で広く利用されている手続きライブラリを利用した場合，限られたチューニングしかできない．手続きライブラリはブラックボックスであり，ユーザはライブラリの実装に手を加えることができないためである．OOP を利用すれば，機能を関心事に分割する事で，粒度の細かいチューニングができる．しかし，OOP の利用には抽象化によるオーバーヘッドがかかるという問題点がある．そこで，この問題を解消するための機構として，我々は WootinJ を開発した．

1　はじめに

HPC では，アプリケーションの実行環境に合わせたパフォーマンスチューニングが求められる．HPC では大規模な問題を扱うため，実行性能が重要視される．実行性能をより高めるため，近年の HPC のハードウェアアーキテクチャは，CPU のマルチコア，GPU の利用など複雑化している．そのため，良い実行性能を得るためには，実行環境のハードウェアに合わせたチューニングが必要である．しかし，現状の HPC のプログラミング環境では，実行環境に合わせたチューニングを行う事は難しい．HPC では，広く手続きライブラリが利用されている．手続きライブラリはブラックボックスであり，ユーザは手続きライブラリの実装の内部に対して変更を加える事はできない．そのため，ユーザによる粒度の細かいチューニングができない．

本論文では，オブジェクト指向 (OOP) を利用する事で手続きライブラリよりも，粒度の細かいパフォーマンスチューニングができる事を述べる．OOP を利用する事で機能を関心事に分割する事ができ，ユーザが各関心事に対して実装を選択する事ができるようになるためである．一方で，OOP の利用には抽象化によるオーバーヘッドがかかるという問題点がある．そこで，この問題を解決する機構として我々は WootinJ を開発した．WootinJ は Java から CUDA [1] への実行時変換器である．WootinJ のユーザは，プログラムを OOP 言語である Java を利用して記述できる．記述されたプログラムは WootinJ によって，抽象化のオーバーヘッドが除去されたコードへと変換され，高速に実行される．

本論文の残りは次のような構成になっている．第 2 章では，ライブラリ利用時のパフォーマンスチューニングの限界について，第 3 章では，OOP を利用した際のパフォーマンスチューニングについて述べる．第 4 章では，OOP の抽象化によるオーバーヘッドを除去する機構として WootinJ を紹介する．第 5 章では関連研究を述べ，第 6 章でまとめと今後の課題を述べる．

Performance Tuning of HPC Applications by OOP

Shumpei Hozumi, Shigeru Chiba, 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.

Masayuki Ioki, 東京工業大学大学院情報理工学研究科, Graduate School of Information Science and Engineering, Tokyo Institute of Technology.

2 ライブラリ利用時のパフォーマンスチューニングの限界

HPC アプリケーションを作成する際には，実行環境に合った手続きライブラリを利用する事で良い実行性能を得る事ができる．例として，線形代数の手続きライブラリを考える．線形代数の手続きライブラリには Basic Linear Algebra Subprograms（BLAS）と呼ばれる標準 API が存在し，BLAS 準拠の手続きライブラリが多数存在する．例えば，BLAS 準拠の手続きライブラリの 1 つとして，Accelerate [2] がある．このライブラリは Apple によって提供されており，Mac OS X に最適化されている．したがって，Mac OS X を利用しているユーザは Accelerate を利用した方が他のライブラリを利用するよりも良い実行性能を得られるだろう．このように，実行環境にあった手続きライブラリを利用する事によってアプリケーションの性能を向上させる事ができる．

しかし，手続きライブラリ利用時に実行環境に合わせたチューニングを十分に行う事は難しい．手続きライブラリはブラックボックスであるため，ユーザが手続きライブラリの実装に手を加える事はできない．そのため，ユーザによる粒度の細かいチューニングが行えない．

近年の HPC のハードウェアアーキテクチャは複雑化しているため，ユーザによる実行環境に合わせたチューニングが必要である．チューニングの例として行列積を考える．行列積は C 言語を用いると，単純には図 1 のように実装する事ができる．図 1 の緑色の線は行列積に含まれる関心事を表しており，以下のような意味を持つ．

1. C の要素のたどり方
2. C の要素の求め方
3. A と B から得た要素に作用させる演算
4. 行列の内部表現

このとき，関心事 2 に対して Listing 1 のように for 文を展開した実装 (loop unrolling) を考える事もできる．for 文の展開を行うと，終了条件の比較回数が減少するため，実行性能の向上が期待できる．しかし，展開にはローカル変数の増加やコード行数の増加などの負の面もあり，最適な展開数は単純に求める事はできない．

Listing 1 関心事 2 の for 文を展開した実装

```
for(int k = 0; k < M; k+=2) {
 C[i*N+j] += A[i*M+k] * B[k*N+j];
 C[i*N+j] += A[i*M+(k+1)] * B[(k+1)*N+j];
}
```

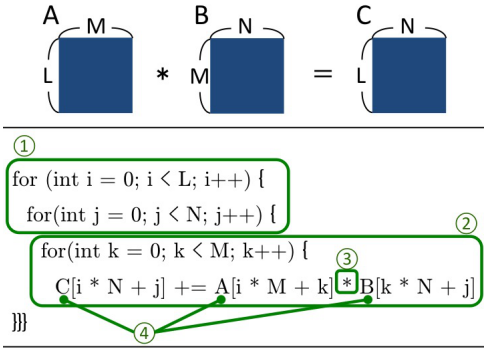

図 1 行列積の C 言語による実装

このように，実行環境に応じてユーザがライブラリの実装を最適化することが必要になるが，現状の HPC プログラミングでは，そのような最適化を許すライブラリを提供することは難しい．そのためにはライブラリの機能をモジュラーに分割する機能がプログラミング言語に必要だが，C 言語や Fortran 言語では十分でない．プリプロセッサの #ifdef やリンカを活用すれば，ある程度モジュラーな分割が可能になるが十分ではない．

3 OOP を利用したパフォーマンスチューニング

OOP を利用する事で，ユーザは手続きライブラリよりも粒度の細かい実行時性能の最適化が行えるようになる．例として，行列積を Java で実装する場合を考える．Java で実装した場合，図 2 のように，ユーザが関心事ごとに実装を指定する事ができる．図 2 中の緑色の線は図 1 の関心事に対応している．

このとき，図 2 中の関心事 2 の実装を変更する事を考える．図 2 で指定している Reduction クラスは

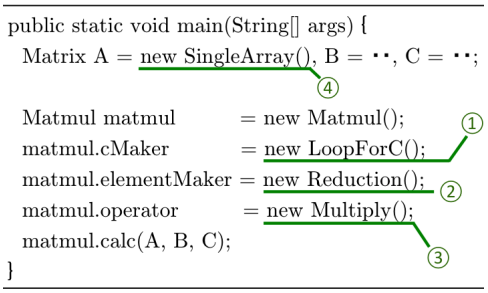

図 2　行列積の Java による実装

Listing 2　関心事 2 の通常の実装

```
class Reduction {
 void make(int i, int j, Matrix A, Matrix B,
           Matrix C, Operator operator) {
  float sum = 0;
    for(int k = 0; k < M; k++) {
     sum += operator.calc(A, B, i, k, k, j);
  } ‥

}}
```

Listing 3　関心事 2 の for 文を展開した実装

```
class UnrollingReduction extends Reduction{
 void make(int i, int j, Matrix A, Matrix B,
           Matrix C, Operator operator) {
  float sum = 0;
    for(int k = 0; k < B.getHeight(); k+=2) {
     sum += operator.calc(A, B, i, k, k, j);
     sum += operator.calc(A, B, i, k+1, k+1, j);
  } ‥

}}
```

図 1 で示したような通常のリダクション処理を意図しており，Listing 2 のように実装されている．一方，2 章で述べたように，関心事 2 の実装として for 文を展開したものも考える事ができる．Listing 3 の UnrollingReduction クラスは関心事 2 に含まれる for 文を展開した実装になっている．ユーザが，実行環境を考慮し，for 文を展開した実装に変更すると判断した場合には，以下のように図 2 中の関心事 2 に関する new 式を変更するだけでよい．

```
   :
```

Listing 4　OOP を用いたオートチューニング

```
List<Reduction> ems;
ems.add(new Reduction());
ems.add(new UnrollingReduction());
  :
for(Reduction red : ems) {
 matmul.elementMaker = red;
 matmul.calc(A, B, C);
}
```

```
matmul.elementMaker = new UnrollingReduction();
   :
```

関心事ごとに実装を容易に切りかえられる事を利用すれば，オートチューニングへの応用もできる．オートチューニングとは，同一の機能を実現する複数の実装をあらかじめ準備しておき，実際の性能を比較することで，優れた実装をソフトウェア的に決定する手法である．OOP を利用した方法であれば，Listing 4 のように 1 つの関心事に対して複数の実装で自動的に動作させ，実行性能を比較する事が手軽にできる．

4　WootinJ による OOP と実行性能の両立

OOP を利用する事には，抽象化によるオーバーヘッドがかかるという問題点がある．HPC では実行性能が重要視されるため，このオーバーヘッドを無視することはできない．そこで，我々はこの問題点を解決するために，OOP と実行性能を両立する機構として WootinJ を開発している．

WootinJ は Java から CUDA への実行時変換器である．WootinJ を利用する事でユーザは Java でコードを記述することができる．また，記述されたコードは WootinJ によって最適化が施された CUDA コードへ変換され高速に実行される．

ユーザは WootinJ が提供する CUDARunner.invoke メソッドを呼ぶことで，メソッド単位でコードの変換をおこなう事ができる．CUDARunner.invoke のそれぞれの引数は，メソッドのレシーバ，メソッド名，メソッドに渡す実引数である．

```
CUDARunner.invoke(Object reciever,
                  String methodName,
                  Object... args);
```

Listing 5 WootinJ を利用したカーネル関数呼び出し

```
public static void main(String[] args) {
  int L, M , N;
  CUDARunner.invoke(new Calc(), ‘‘target’’,
                 L, M, N);
}

class Calc {
  void target(int L, int M , int N) {
    float[] A, B, C;

    dim3 grid = new dim3(N/BS, L/BS);
    dim3 block = new dim3(BS, BS);
    make(grid, block, A, B, C);
  }

  kernelvoid make(dim3 grid, dim3 block,float[] A,
float[] B,float[] C) :
```

CUDA のカーネル関数呼び出しを記述したい場合は，WootinJ が提供する@kernel を利用する．WootinJ は，変換されるメソッド中に@kernel が付与されたメソッドの呼び出し式がある場合は，その呼び出し式をカーネル関数呼び出しへと変換する．例えば，Listing 5 のようなコードが与えられた場合には，Listing 5 中の target メソッドを Listing 6 のように変換する．Listing 5 で make メソッドの第一，第二引数として渡しているのは CUDA のスレッド管理の単位であるグリッドとブロックのサイズを指定するための値である．WootinJ が提供する dim3 クラスを利用することでこれらの値を生成する事ができる．

WootinJ は抽象化によるオーバーヘッドを除去するために，実行時情報を利用した 2 つの最適化を行なっている．1 つ目は，メソッド呼び出しの非仮想化である．CUDARunner.invoke に渡されたレシーバと実引数の実行時の型を利用することで，メソッド呼び出しを非仮想化している．2 つ目は，オブジェクト構造の除去である．オブジェクトを CUDA で素直に表現しようとした場合，構造体の利用が考えられる．しかし，構造体を作成すると，間接参照が発生するためオーバーヘッドが発生する．そこで WootinJ はオブジェクトを展開し，プリミティブな値の利用へと変換している．

WootinJ は実行時変換器であると述べた．この特徴を利用すると，パラメータの決定をより柔軟におこなうことができる．行列積では，パラメータとして行列のサイズを考える事ができる．C 言語で実装する場合，行列のサイズは定数としてコード中に記述するのが一般的である．なぜならば，定数として記述することでコンパイラによる最適化を期待することができるためである．具体的には，定数伝播やループアンローリングなどの最適化が期待できる．実行時にコンパイルをおこなう WootinJ であれば，コンパイル時にはパラメータは定数扱いになるため，パラメータをハードコーディングする必要はない．これにより，柔軟にパラメータの変更をおこなうことができる．

Listing 6 Listing 5 のコードから WootinJ によって生成される CUDA コード

```
void target(int L, int M , int N) {
  float[] A, B, C;

  dim3 grid(N/BS, L/BS);
  dim3 block(BS, BS);
  make<<<grid, block>>>(A, B, C);
}

__global__
void make(dim3 grid, dim3 block,
         float[] A, float[] B,
         float[] C) {
   :
}
```

5 関連研究

5.1 手続きライブラリにおける実行環境に合わせたチューニング

ATLAS [4] は BLAS 準拠の線形代数ライブラリである．インストール時にパラメータによる自動最適化を行なっており，実行環境に合わせたチューニングが可能となっている．また，ユーザが CPU の種類などの情報を ATLAS に与える事で適切な実装を選択するようになっている．しかし，実際に利用できるメモリ量など実行してみないとわからない要素があるため，ライブラリ作成時に最適な実装を決めるのは難しい．

5.2 関数ポインタ

C 言語や C++に存在する関数ポインタを利用する事で，ユーザによって内部の実装の一部を変更可能なライブラリを作成することができる.. 例えば，クイックソートをライブラリで実装した qsort 関数がある．qsort 関数は，ユーザによって実装された比較関数を使いソーティングを行なっている．このように関数ポインタを利用すれば，OOP のように処理を分割する事ができる．しかし，関数ポインタを利用すると OOP と同様のオーバーヘッドがかかるという問題が発生する．解決のためには WootinJ 同様の変換器が必要である．

5.3 HPC における OOP の利用

Firepile [3] は Scala から OpenCL [5] への実行時変換器である．Firepile を用いる事で，GPU で実行される関数内で，動的メソッドディスパッチを利用することができる．しかし，動的メソッドディスパッチは，巨大な swtich 文を使い表現されている．そのため，動的メソッドディスパッチを利用するとオーバーヘッドがかかってしまう．

CUDA C は NVIDIA の GPU 向けに開発されたプログラミング言語である．CUDA C は C 言語をベースとした言語であるが，C++の機能を利用する事もできる．例えば，クラスやテンプレートなどの利用が可能である．しかし，関数ポインタや仮想関数の利用には制限がある．

6 まとめ

OOP を用いる事で，粒度の細かいパフォーマンスチューニングができる事を述べた．HPC で広く利用されている手続きライブラリではブラックボックスであり，ユーザによるチューニングが行えなかった．それに対して，OOP を利用する事で，ユーザは粒度の細かいパフォーマンスチューニングができる．これにより，実行環境に応じた最適化ができるようになり，アプリケーションのパフォーマンスをより高める事ができる．OOP を利用すると抽象化によるオーバーヘッドがかかってしまうが，それを除去するための機構として WootinJ を開発した．WootinJ は，実行時情報を利用して，動的メソッドディスパッチとオブジェクトのインライン化を行っている．それによって，抽象化によるオーバーヘッドが除去されたコードを生成する．今後の課題は，WootinJ を利用したフレームワークの作成である．

参 考 文 献

[1] CUDA http://developer.nvidia.com/category/zone/cuda-zone

[2] Accelerate https://developer.apple.com/ performance/acceleratframework.html

[3] Firepile Nystrom, N. and White, D. and Das, K. : Firepile: run-time compilation for GPUs in scala Proceedings of the 10th ACM international conference on Generative programming and component engineering

[4] ATLAS Clint Whaley, R. and Petitet, A. and Dongarra, J.J. : Automated empirical optimizations of software and the ATLAS project Parallel Computing, 2001

[5] OpenCL Stone, J.E. and Gohara, D. and Shi, G. : OpenCL: A parallel programming standard for heterogeneous computing systems Computing in science & engineering, 2010