Vol.2013-HPC-139 No.19
2013/5/30

チューニング対象の限定による効率の良い性能可搬性向上手法

平澤 将一[1,2,a)]　秋葉 諒[1]　滝沢 寛之[1,2]　小林 広明[1]

概要：計算システムの多様化に伴い，既存の科学技術計算プログラムを新たな計算システムへ移植し性能を最適化する作業がしばしば求められている．しかしながら大規模な科学技術計算プログラムの移植および性能最適化には多大な労力が必要となり，問題となっている．本研究では，性能可搬性向上を目的とした場合に優先的に性能最適化を行うべきソースコードの箇所を限定し，効率良くアプリケーション全体の性能可搬性を向上させる手法を提案する．ベンチマークプログラムおよび実アプリケーションによる評価の結果，提案手法はアプリケーション全体の性能可搬性を効率よく向上させるために，最適化すべきソースコードの部位を限定できることが示された．

1. 背景と目的

高性能科学技術計算において，特定の計算システムにおける実行性能が向上するようにプログラムを修正することを性能最適化と呼ぶ．近年の計算システムの構成は複雑化しているため，高い実行性能を達成するためにはシステムの構成を意識した性能最適化が必要である．一般的に科学技術計算プログラムは中長期的に利用されるため，その利用期間中に計算システムのアーキテクチャが変遷する可能性が高い．特に近年，GPU などを利用した複合型計算システムが普及し，科学技術計算プログラムの移植がしばしば求められている．特定の計算システム向けに性能最適化された科学技術計算プログラムは他の計算システムで高い実行性能を達成できないことが多いため，移植先となる計算システム向けに性能最適化を繰り返す多大な労力を必要とする．中長期的にプログラムの高い性能を維持することが求められる高性能科学技術計算分野において，複数の計算システムで高い実行性能を達成できる性質である性能可搬性が重要視されている．

プログラムの性能可搬性が高くなるようにソースコードを修正する作業を，本研究では性能リファクタリングと呼ぶ．科学技術計算プログラムの大規模化により，プログラム全体の性能改善を一度に行うことは困難である．そのためプログラムの性能改善の際には，gprof[1] に代表されるプロファイラを用いて， 関数やループなどの計算ブロックごとに性能ボトルネック解析が行われてきた．これにより，プログラムの性能低下要因となる計算ブロックを特定することができる．また今日では，ルーフラインモデル [2] に代表される性能低下要因解析手法を用いることで，計算システムの理論演算性能や最大メモリバンド幅を考慮してプログラムの性能ボトルネックを解析するアプローチも採られている．これらの性能解析手法を用いることで，性能改善のために修正すべき計算ブロックを限定することが可能であり，大規模プログラムを効率よく性能最適化できることが期待される．

一般的に，これらの手法は対象とする特定の計算システムにおける性能ボトルネックを解析する．一方，性能可搬性を向上させるためには，複数の計算システムにおける実行性能の比較に基づいて，性能可搬性低下の要因となっている計算ブロックを特定することが望まれる．しかしながら，性能可搬性の低下要因を定量的に解析する手法は確立されていない．

このため本研究では，性能可搬性低下要因となる計算ブロックを特定する指標を提案し，特定した計算ブロックに対する性能リファクタリングにより性能可搬性を効率的に向上可能であることを明らかにする．

2. 性能可搬性と性能リファクタリング

性能最適化により，特定の計算システムで高い実行性能を得られる一方，他の計算システムでは低い実行性能しか得ることができないプログラムは性能可搬性が低いプログラムである．科学技術計算プログラムの大規模化により，プログラム全体の性能可搬性を一度に改善することは困難

[1] 東北大学
[2] JST CREST
[a)] hirasawa@sc.isc.tohoku.ac.jp

である．そこで，プログラムの構成をより詳細に解析し，優先的に性能最適化すべき箇所を限定する必要がある．

2.1 性能可搬性の向上が期待できる計算ブロック

科学技術計算プログラムは，様々な計算を行う計算ブロックより構成されている．それぞれの計算ブロックは実行する計算量や並列度などの特徴が異なり，実行する計算システムによって実行性能に差が生じる．そのため性能最適化の際に想定されたシステム (以下，基準システムとする) において高い実行性能が得られる一方，それ以外の計算システムでは実行性能が大幅に低下する計算ブロックが存在する．プログラムの性能可搬性低下への影響が大きい計算ブロックから順に性能改善を行うことで，効率的にプログラム全体の性能可搬性を向上させることが期待できる．

移植先の計算システムにおいて実行性能が低い場合，以下の 2 つの状況が考えられる．(1) 基準システムと比較して移植先の計算システムの持つ理論最大性能が低い (2) 基準システムに特化した性能最適化が行われており，移植先の計算システム向けには性能最適化されていない．前者が主要因の場合には，移植先の計算システムにおいてさらなる実行性能の改善は困難である．一方，後者が主要因となっている場合には，移植先の計算システム向けに性能最適化を行うことで実行性能を向上することが可能であり，その結果として性能可搬性の改善を期待できる．この両者を区別するため，本研究では複数の計算システムで高い実行効率を達成することを考える．本研究における実行効率とは，計算システムの最大演算性能に対する実効性能の割合である．

実行効率を解析するためには，各計算システムにおける達成可能な最大の演算性能 (以下，最大演算性能とする) を知る必要がある．最大演算性能は，ハードウェアのもつ理論的な最大演算性能 (以下，理論演算性能とする) だけではなく，複数の要因で決まる．特に，現在の計算システムの最大演算性能は最大メモリバンド幅によって律速されることが多い．プログラムの実行効率を解析するためには，理論演算性能，最大メモリバンド幅など計算システムの性能ボトルネックとなる要因を考慮した最大演算性能を用いる必要がある．

図 1 に計算ブロックのそれぞれの計算システムにおける実行効率の例を示す．グラフの縦軸は実行効率，横軸は計算ブロックである．実行効率はそれぞれの計算システムにおける実行効率を用いることで，計算システムの持つ絶対性能が低い場合と性能最適化が不十分な場合を区別することが可能である．計算ブロック 1 に着目すると，基準システムにおける実行効率 (以下，基準実行効率とする) と比較して，他の計算システムにおける実行効率が同等かそれ以上であることがわかる．そのため本研究では，計算ブロック 1 の性能可搬性は高いと考える．一方，計算ブロック 2

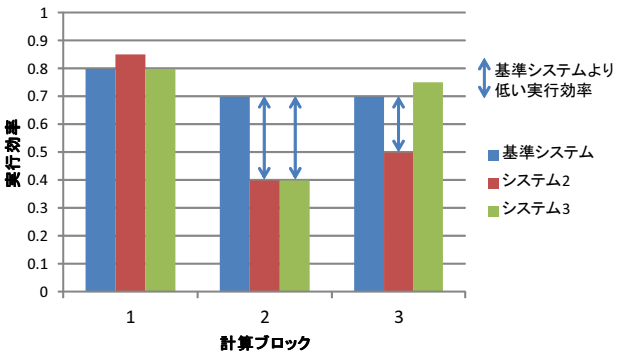

図 1 各計算ブロックにおける実行効率例

に着目すると，基準実行効率と比較して，他のすべての計算システムにおける実行効率が低いことがわかる．また，計算ブロック 3 は基準実行効率に対し，システム 2 での実行効率が低く，システム 3 では高くなっている．本研究では，計算ブロック 2，3 のように，実行効率が基準実行効率に達していない計算システムが存在する場合，その計算ブロックの性能可搬性が低いと考える．性能可搬性の低い計算ブロックに対して，基準システム以外での実行効率を改善することにより，複数の計算システムで高い実行性能を達成することを考える．

基準システムの実行効率に達していない計算システムが複数存在する計算ブロックや，基準システム以外の実行効率と基準実行効率の差が大きい計算ブロックは，性能最適化による実行効率の向上幅が大きく，結果として性能可搬性が大きく向上することが期待できる．図 1 では，計算ブロック 3 におけるシステム 2 の実行効率と基準システムの実行効率の差，計算ブロック 2 における計算システム 2 の実行効率と基準システムの実行効率の差は大きい．このため，これらの計算ブロックに対して十分な性能最適化が行われていないと推測することが可能であり，これらの計算ブロックの最適化によって性能可搬性が大きく向上することが期待できる．また，計算ブロック 3 では実行効率が基準実行効率に達していない計算システムが 1 つであるのに対して，計算ブロック 2 では 2 つの計算システムの実行効率が基準実行効率に達していない．そのため，プログラムの性能可搬性向上を考える場合，計算ブロック 3 より計算ブロック 2 を性能最適化したほうが，より大ききな性能可搬性向上を期待できる．したがって，計算ブロック 3 より計算ブロック 2 を優先的に性能リファクタリングすべきであると判断できる．一方，絶対性能が低い場合であっても十分な性能最適化がされている場合は，その計算システムにおいて高い実行効率が発揮できていると考えられる．このような場合，その計算ブロックを最適化しても，性能可搬性の大幅な改善は期待できない．また，プログラム全体に占める実行時間の割合の大きい計算ブロックには，実行時間の割合の小さい計算ブロックと比較して大きな性能可

```
for (int i = 0; i < k; ++i) {
    float a = A[m + i * lda];
    float b = B[n + i * ldb];
    c += a * b;
}
```

図 2 ループアンローリングによる性能最適化前のソースコード

```
for (int i = 0; i < k; i=i+2) {
    float a = A[m + i * lda];
    float b = B[n + i * ldb];
    c += a * b;

    a = A[m + (i+1) * lda];
    b = B[n + (i+1) * ldb];
    c += a * b;

}
```

図 3 ループアンローリングによる性能最適化後のソースコード

搬性向上を期待できる．そのためプログラム全体に占める実行時間の割合が大きい計算ブロックについても，プログラマが優先的に性能リファクタリングすべき計算ブロックであるといえる．

図 1 の例においては，上記の議論より性能可搬性向上が期待できる計算ブロックを優先度の高いものから並べることは容易である．しかし，プログラムの大規模化により，多数の計算ブロックを有するプログラム中の，各計算ブロックのそれぞれの計算システムにおける実行効率の比較や，実行時間割合の大小比較を同時に行うことは困難である．そのため，上記の議論を考慮した性能可搬性を指標を用いて定量的に示し，その大小より各計算ブロックの性能可搬性を比較することを考える．これにより，プログラマは大きな性能可搬性向上が期待できる計算ブロックを優先的に性能リファクタリングすることができる．

2.2 性能可搬性向上手法: 性能リファクタリング

複数の計算システムで高い実行性能を得るためには，計算システムごとに異なる性能最適化を行う必要がある．そのため，計算システムごとに実行性能の高いソースコードがそれぞれ存在する．本研究では，近年普及している自動チューニング機構 [4] などの，実行する計算システムごとにプログラム内のソースコードを自動修正し，1 個のプログラムで複数のソースコードを生成可能な機構を仮定する．これにより，各計算システムで実行性能の高い性能最適化コードでプログラムを実行することができるため，プログラムの性能可搬性を向上させることができる．

ループアンローリングは繰り返し演算を行うループ回数を減らし，ループ終了判定時間を減少させることにより実行時間を短縮する性能最適化手法である．計算システムごとに，ループ内の処理を複製する回数であるアンロール段数の最適な値が異なる．**図 2**，**図 3** にループアンローリングによる性能最適化前後のソースコードの例を示す．図 3 はアンロール段数が 2 である．各段数でアンロールしたプログラムの，各計算システムにおける演算性能の違いを**図 4** に示す．グラフの縦軸は演算性能 (GFlops/s)，横軸はアンロール段数である．アンロール段数によって各計算システムにおける演算性能の振る舞いが異なり，各計算システムで最適なアンロール段数を持つ最適化コードが存在する．アンロール段数が計算システムごとに適切に調整されるように自動チューニングすることで，高い実行性能でプログラムを実行することが可能となり，性能可搬性を向上させることができる．

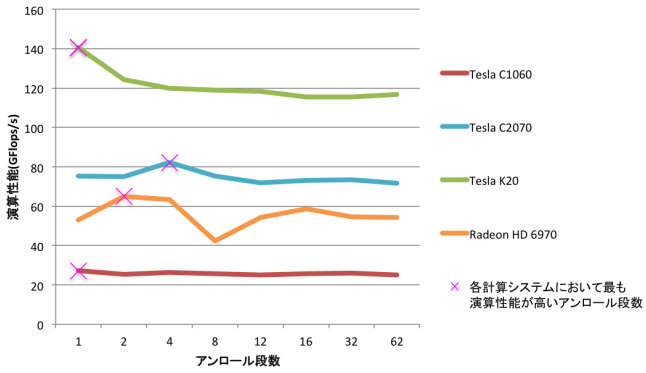

図 4 各計算システムにおけるアンロール段数による演算性能

3. 性能リファクタリングによる性能可搬性向上手法

3.1 性能可搬性向上手法

本手法では，プログラムの性能可搬性が低下する要因であり，性能可搬性の大きな向上につながる計算ブロックを定量的に示す性能可搬性指標を提案する．指標に基づいて性能リファクタリングを適用することで，プログラムの性能可搬性を効率的に向上できることが期待できる．

性能可搬性の高低を示す性能指標として，実行時間や演算性能など様々な性能指標が考えられる．しかし前章の議論に基づき，本手法では実行効率に基づいて性能可搬性を定義する．実行効率の解析には，計算システムの理論演算性能と最大メモリバンド幅の双方を考慮した性能解析手法を用いる．各計算ブロックのそれぞれの計算システムにおける実行効率から性能可搬性が相対的に低い計算ブロックを特定し， 性能リファクタリングを適用することにより，プログラム全体の性能可搬性を大きく向上させることが期待できる．

3.2 性能可搬性低下要因となる計算ブロックの特定

本指標では，基準実行効率とプログラムの移植先である計算システムでの実行効率の差，およびプログラム全体に占める実行時間の割合を考慮する．移植元と移植先の計算システムでの実行効率の差の分散を求め，性能可搬性を評

価する．ただし，基準実行効率より高い実行効率を得られている計算システムでは，性能可搬性が十分に高いと考え，分散を最小の 0 として扱う．

あらかじめ定められた，プログラムを実行する N 個の計算システムをシステム $n(n=1, 2, 3, \cdots N)$，プログラム中の M 個の計算ブロックを計算ブロック $m(m=1, 2, 3, \cdots M)$ とする．システム n における計算ブロック m の実行効率を p_n^m，基準実行効率を p_s^m とする．p_s^m は各計算システム固有のパラメータである理論演算性能および最大メモリバンド幅を考慮した実行効率とする．各計算システムと基準実行効率の差を $\{\min(p_n^m, p_s^m) - p_s^m\}$ と定義する．ここで，実行効率が基準実行効率よりも高いシステムでは，実行効率の差が 0 となる．システム n におけるプログラム全体の実行時間を 1 とし，プログラム中の計算ブロック m の実行時間の割合を t_n^m とする．また，実行不可能である計算システムにおける実行効率の差を，最も大きい値として扱う必要がある．そのため実行不可能な計算システムにおける実行時間の割合を 1 とする．以上の値を用いて，計算ブロック m における s_m を次式で定義する．

$$s_m = \frac{\sum_{n=1}^{N} \{\min(p_n^m, p_s^m) - p_s^m\}^2 t_n^m}{N} \quad (1)$$

s_m が大きい計算ブロックでは性能可搬性が低いと考えることができる．$\{\min(p_n^m, p_s^m) - p_s^m\}$ がすべて最大値 1 であり，t_n^m がすべてのシステムで 1 であった場合に，s_m は最大値である 1 となる．そのため，性能可搬性指標 S_m は s_m の最大値と s_m の差として定義される．

$$S_m = 1 - s_m \quad (2)$$

プログラム内で相対的に S_m が小さく，性能可搬性の低下要因となっている計算ブロックを特定し，性能リファクタリングを適用することでプログラムの性能可搬性が効率よく向上することが期待できる．

最後に，S_m を用いてプログラム全体の性能可搬性を定義する．これによりプログラム全体の性能可搬性を評価することができる．計算ブロックに包含関係がなく各計算ブロックが独立している場合，プログラム全体の性能可搬性指標を，すべての計算ブロックの S_m の合計と定義する．

$$S = \sum_{m=1}^{M} S_m \quad (3)$$

また，計算ブロックに包含関係が存在する場合の例として，**図 5** のように計算ブロック内で他の計算ブロックの呼び出しが存在する場合を考える．例に示す計算ブロックは関数とする．図 5 に着目すると，メインソースコード内で計算ブロック func1，func2，func3 が呼ばれている．図 5 の func2 のように計算ブロック内で他の計算ブロックを呼んでいる場合，func2 の実行時間が func4 の実行時間を含んでいる．そのため各計算ブロックの実行時間は，計算ブロック内で呼び出した他の計算ブロックにおける実行時間を除いた実行時間と定義する．これにより，計算ブロックに包含関係が存在している場合でも，上記の式 (3) と同様にプログラムの性能可搬性指標 S をすべての S_m の合計で求めることができる．プログラムの性能可搬性指標 S を用いて，提案手法によりプログラムの性能可搬性を効率的に向上できることを次章で評価する．

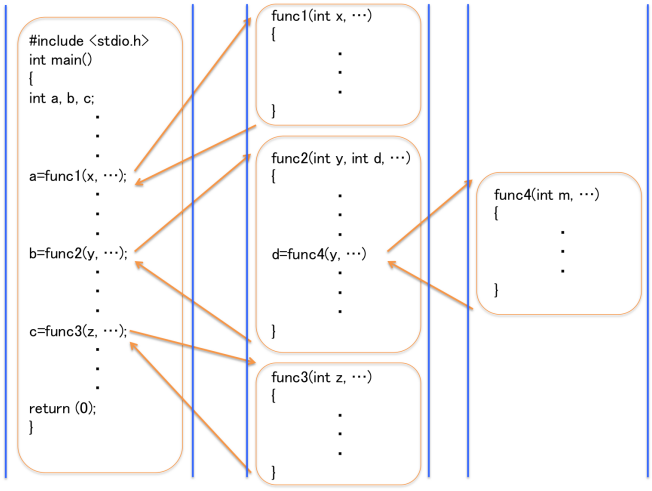

図 5 関数内で他の関数を実行するソースコードの例

ここで，プログラムの性能可搬性向上を考える際，包含関係が存在する計算ブロックを有するプログラムでは，呼び出し順により定まる計算ブロック群ごとに各計算ブロックの性能可搬性指標を考慮する．はじめにメインソースコード内で呼び出される計算ブロック群 (図 5 の例における func1，func2，func3) の中で性能可搬性が最も低い計算ブロックを特定する．次に，性能可搬性が低い計算ブロック内に他の計算ブロックの呼び出しが複数存在している場合，その計算ブロック群内で性能可搬性が最も低い計算ブロックを選定する．このように，性能可搬性が低い計算ブロックを各計算ブロック群ごとに求め，相対的に性能可搬性が最も低い計算ブロックを特定し，性能リファクタリングを優先的に適用することでプログラムの性能可搬性を効率的に向上できる．

4. プログラムの性能可搬性向上手法の評価

4.1 評価手法

本研究の提案手法により，プログラムの性能可搬性が効率的に向上することを評価する．基準システム向けに性能最適化されたプログラムが，カーネル a，b，c，d，e，f から構成されていると仮定する．カーネルとは関数単位で演算を行う計算ブロックである．性能可搬性指標によりプログラムの性能可搬性が低下する要因となるカーネルを特定し，特定したカーネルに対して性能リファクタリングを適用する．本評価実験で用いる性能リファクタリングは，実行する計算システムごとに実行効率の高い性能最適化パラ

メータを選定する自動チューニング手法 [5] の利用を前提とした性能リファクタリングである．他のカーネルと比較して，提案手法により特定されたカーネルを性能リファクタリングした場合に，性能可搬性がより大きく向上することが期待される．

本評価では，プログラムの性能可搬性向上幅を，性能リファクタリング前後での S の減少幅 ΔS で評価する．ΔS_m が最大のカーネル m を性能リファクタリングした場合の性能可搬性向上幅 $\Delta S_{\max}$ と，提案手法により特定されたカーネルを性能リファクタリングした場合の性能可搬性向上幅 ΔS を比較することで，提案手法によるカーネル選択の妥当性を評価する．

4.2 評価環境

評価環境は以下の通りである．評価システムには，4 種類の異なる GPU が搭載されている．用いる GPU は NVIDIA 社 GPU である Tesla C1060，Tesla C2070，Tesla K20 と AMD 社 GPU である Radeon HD 6970 である．以下，各 GPU を搭載する計算システムをそれぞれ Tesla C1060, Tesla C2070，Tesla K20，Radeon HD 6970 と表記する．

対象とするプログラムとして，Parboil benchmark[6] に含まれるカーネルを用いる．用いるカーネルは sgemm(sgemm_t，sgemm_u)，dgemm(dgemm_t, dgemm_u)，stencil(stencil_single，stencil_double) である．sgemm, dgemm は行列積の計算カーネル, stencil は 3 次元グリッド上で Jacobi 演算を行うカーネルである．sgemm_t, dgemm_t，stencil はそれぞれスレッドコースニング [7]，sgemm_u，dgemm_u はループアンローリングにより，基準システムに対して性能最適化されている．スレッドコースニングとは，実行するスレッド数をマージし，必要以上に確保したスレッド数を減らすことにより実行時間を短縮する性能最適化である．

評価手順は以下の通りである．はじめに各カーネルの実行時間と演算密度を基に，ルーフラインモデルを用いて実行効率を解析する．得られた実行効率から性能可搬性指標 S_m を求め，対象プログラム内で性能可搬性が低下する要因となるカーネルを特定する．次に対象カーネルの 6 個をすべて用いる組み合わせ 1 通り，5 個を用いる組み合わせ 6 通り，4 個を用いる組み合わせ 15 通りの全 22 通りのプログラムを対象に，性能可搬性向上の評価を行う．プログラム内のすべてのカーネルの実行時間の合計に対する各カーネルの実行時間を，カーネルの実行時間の割合 t_n^m とする．本評価プログラムが有するカーネルは包含関係がないものとする．そのため，性能リファクタリング前後の S の差 ΔS を S_m の差の合計より求める．$\Delta S_{\max}$ と提案手法による性能可搬性向上幅 $\Delta S1$，および提案手法による選択を除いたカーネルによる最大性能可搬性向上幅 $\Delta S2$ を比較し，性能可搬性向上幅が大きいカーネルを特定できていることを評価する．

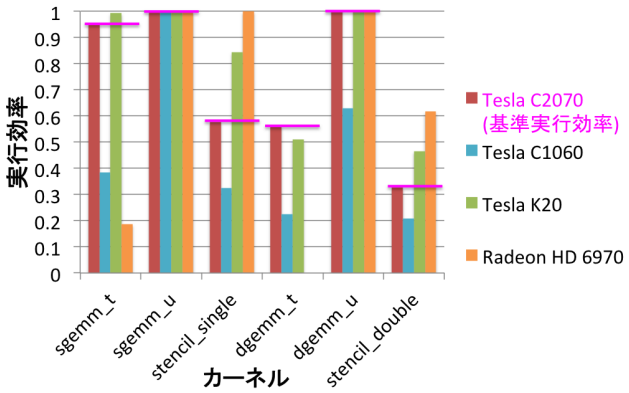

図 6 各カーネルにおける実行効率解析結果

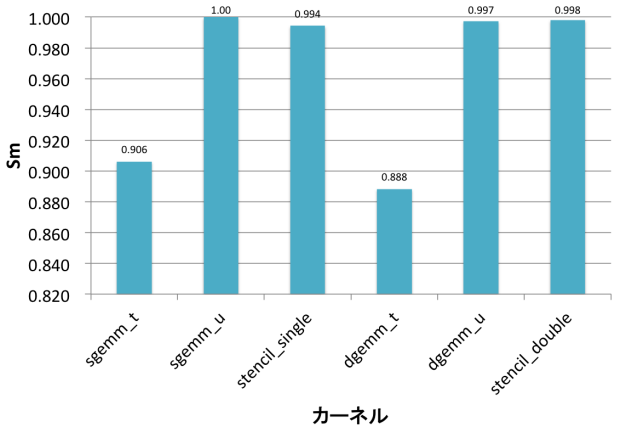

図 7 各カーネルにおける S_m

4.3 プログラムの性能可搬性向上幅の評価結果

ルーフラインモデルを用いて各カーネルのそれぞれの計算システムにおける実行効率を解析した結果を**図 6** に示す．

図 6 に着目すると sgemm_u は基準実行効率，他の計算システムにおける実行効率ともに 1 であり，性能可搬性が高いことがわかる．一方，dgemm_t では基準実行効率と比較して他の計算システムにおける実行効率が低く，さらに Radeon HD6970 の実行効率が 0 である．このことから，dgemm_t の性能可搬性は低いことがわかる．図 6 を基に各カーネルにおける S_m を求めた結果を**図 7** に示す．グラフの縦軸は S_m，横軸はカーネルである．

図 7 より，性能可搬性が高いと考えられた sgemm_u における S_m が最も大きく，性能可搬性が低いと考えられた dgemm_t における S_m が最も小さいことがわかる．そのため，6 個のカーネルすべてを有するプログラムを想定した場合，性能可搬性が低下する要因となり，性能リファクタリングによる性能可搬性向上が期待できるカーネルは dgemm_t となる．各カーネルの性能リファクタリ

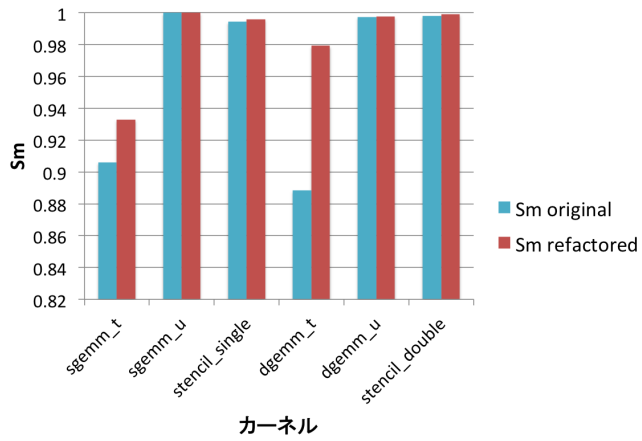

図 8 各カーネルにおける性能リファクタリング前後の S_m の比較

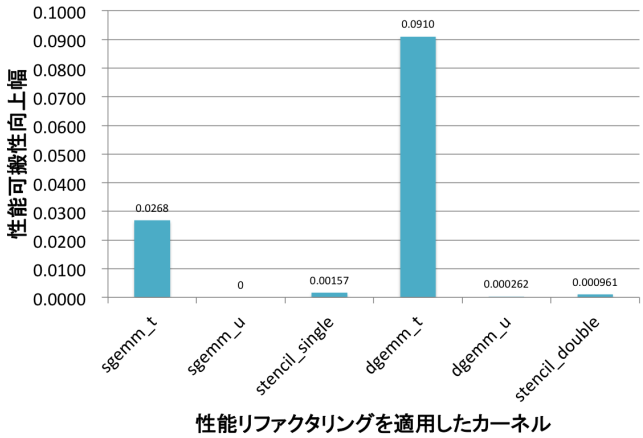

図 9 各カーネルへ性能リファクタリングを適用した場合のプログラムの性能可搬性向上幅

ング適用前の S_m を S_m original，適用した後の S_m を S_m refactored とする．**図 8** に，S_m original と S_m refactored を比較した結果を示す．グラフの縦軸は S_m，横軸はカーネルである．dgemm_t は性能リファクタリングの適用によって S_m が向上したことがわかる．さらにプログラムの性能可搬性を向上させるため性能リファクタリングをする際，最も大きな性能可搬性が向上期待できるカーネルはその時点で S_m が最小の sgemm_t となる．

性能リファクタリングを適用するカーネルが 1 つの場合，各カーネルの性能可搬性向上幅 ΔS_m とプログラム全体の性能可搬性向上幅 ΔS は一致する．そのため，図 8 における各カーネルの ΔS_m は ΔS となる．図 8 より各カーネルに性能リファクタリングを適用した場合のプログラム全体の性能可搬性向上幅を**図 9** に示す．グラフの縦軸はプログラムの性能可搬性向上幅，横軸は性能リファクタリングを適用したカーネルである．

図 7，図 9 に着目すると，dgemm_t は S_m が最も小さい計算ブロックであり，性能リファクタリングによるプログラムの性能可搬性向上幅が最も大きいカーネルである．そのため，プログラム全体の性能可搬性向上幅が最も大きい計算ブロックを S_m により特定できていることがわかる．

同様に 22 通りのプログラムを想定して S_m が最も小さい計算ブロックを特定し，その計算ブロックに性能リファクタリングを適用する．また，プログラムの性能可搬性向上幅を ΔS より評価する．各プログラムの $\Delta S_{\max}$，$\Delta S1$，$\Delta S2$ を**図 10** に示す．グラフの縦軸は性能可搬性向上幅，横軸はカーネル a からカーネル f を含むプログラムである．図 10 より，全 22 通りのプログラム中 17 通りにおいて，提案手法は最も大きく性能可搬性が向上するカーネルを特定できている．以上の結果から，本提案手法を用いることで，プログラム内で最も大きく性能可搬性が向上するカーネルを高い確度で特定できることが示された．

S_m に基づいて特定されたカーネルに性能リファクタリングを適用した場合，性能可搬性向上幅 $\Delta S1$ の平均は 0.0749 である．一方，最も大きく性能可搬性が向上するカーネルを理想的に特定できた場合の性能可搬性向上幅 $\Delta S_{\max}$ の平均は，0.0791 である．このため，提案手法は理想的に特定できた場合とほぼ同等の ΔS を達成できているといえる．提案手法は 5 個のプログラムにおいて最適な選択をできなかったため，$\Delta S1$ と $\Delta S_{\max}$ の差が生じている．提案手法が 5 個のプログラムにおいて最適な選択をできなかったため，この差が生じている．以下，この差が生じた原因について考察する．提案手法では，S_m の小さい計算ブロックに性能リファクタリングを適用することで，性能可搬性が大きく向上すると仮定している．しかしながら，S_m が大きいにも関わらず性能リファクタリングによる性能可搬性向上が大きい計算ブロックが存在する場合，大きな性能可搬性向上を期待できるにも関わらず，提案手法ではその計算ブロックを特定することができない．このような計算ブロックも特定し，より効果的に性能可搬性を向上させるためには，計算システムの特徴や実行性能を考慮して性能可搬性向上幅を予測する必要がある．

5. 実アプリケーションに対する性能可搬性向上

以下，実アプリケーションに対する提案手法の有効性を実証実験により確認する．本評価では，実アプリケーションの例として数値タービン [9] を用いる．数値タービンは，タービン多段流路内の非定常 3 次元流れをシミュレートするために，東北大学山本研究室で開発されている実アプリケーションである．東北大学サイバーサイエンスセンターに設置されている SX-9 ベクトル型スーパーコンピュータで 2013 年現在において運用されている大規模アプリケーションであり，SX-9 向けにさまざまな最適化が施されてきた．

SX-9 での実行性能を維持しつつ他の計算システム上での実行性能も高める移植は，同アプリケーションの性能可

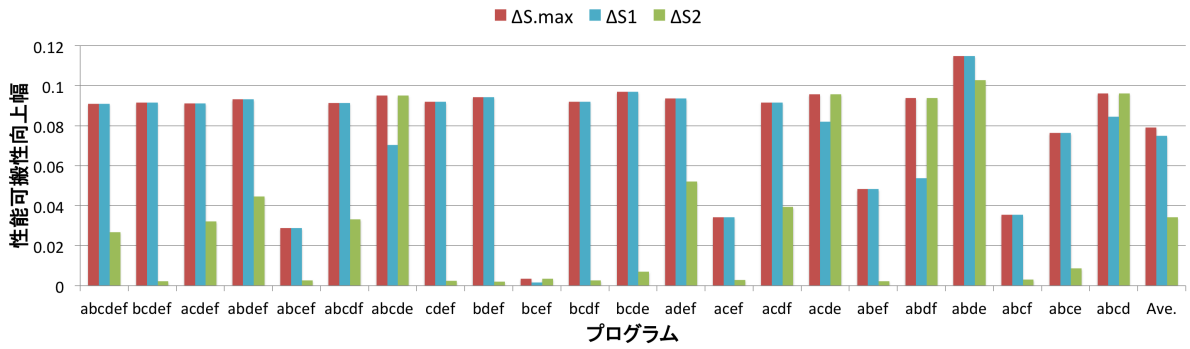

図 10 各プログラムにおける性能可搬性向上幅

搬性を高めるために必要である．近年，OpenACC による GPU コンピューティングが普及しつつあり，適切にディレクティブを挿入することで既存のアプリケーションを GPU に移植可能となっている．そこで本評価では，数値タービンを OpenACC を用いて GPU 計算システムに移植し，さらに性能リファクタリングした場合の性能可搬性を評価する．本評価で用いる GPU 計算システムは，CPU が Core i7 930 2.8GHz，メインメモリが 24GB，GPU が Tesla C2070 である．OS は Linux 2.6.32 であり，コンパイラとして PGI アクセラレータコンパイラ 12.10 を用いる．

移植に際して，まず SX-9 において実行時間割合が 47%と最大を占めるサブルーチン EXPLICIT に注目し，OpenACC ディレクティブの挿入により GPU 計算システムで動作可能とした．

サブルーチン EXPLICIT は基準システムである SX-9 に最適化されているため，SX-9 においては高い性能を発揮する．しかし，OpenACC ディレクティブの単純な挿入では GPU 計算システムで高い性能を発揮できない．また，サブルーチン EXPLICIT は約 1700 行のコードであり，16 個の DO ループが含まれている．限られた工数で効率よく性能可搬性を高めるためには，性能可搬性向上にもっとも効果的な DO ループを優先的に性能リファクタリングする必要がある．このため本評価では，DO ループを計算ブロックとして扱い，提案手法により優先的に性能リファクタリングすべき計算ブロックを特定できることを確認する．

対象とした全 16 個の計算ブロックのうち，SX-9 における実行時間割合が 2%以上であった 6 個の計算ブロックの SX-9 における実行時間割合 t_{sx}，実行効率 p_{sx} および，GPU 計算システムにおける実行時間割合 t_{gpu}，実行効率 p_{gpu} およびこれらを用いて計算した性能可搬性指標 S_m を**図 11** に示す．すべての計算ブロックは DO ループであり，ループ名に従ってこれらを Loop200，Loop210，Loop300，Loop310，Loop400，Loop410 と表記する．すべての計算ブロックは SX-9 で高い実行効率 p_{sx} を発揮する一方，GPU 計算システムにおける実行効率 p_{gpu} が低いことがわかる．

提案手法では，S_m の小さい計算ブロックから優先的に選択される．その結果，選択された順番は Loop200，Loop400，Loop300，Loop210，Loop310，Loop410 となった．本評価では，選択された順番に従って，各計算ブロックが GPU 計算システム向けに最適化された．ここで行った最適化として，主にループ交換による GPU スレッド並列数増加が挙げられる．

各計算ブロックの最適化後の，対象サブルーチン全体の実行時間と，選定した計算ブロックの最適化による実行時間の削減量 (Δ 実行時間) を**図 12** に示す．各計算ブロック最適化によりサブルーチン全体の実行時間が減少していることがわかる．また，Δ 実行時間が単調減少していることから，提案手法により最適化する効果が大きい計算ブロックから順番に選定できていることがわかる．この結果より，提案手法により数値タービンの GPU 移植に際して，最適化による効果が大きい計算ブロックを限定することによって効率よく性能可搬性を向上させることが示された．

ただし本評価においては，提案指標の値 S_m が小さく優先的に選択された計算ブロックは，実行時間に占める割合が大きい計算ブロックと一致した．その原因として，これらの計算ブロックは最適化前に一律に低い実行効率だったことが挙げられる．GPU 計算システムにおける実行時間割合が高い場合でも，最適化する以前にすでに高い実行効率を発揮する計算ブロックが存在した場合には，本提案手法による優先順位が低くなり，実行時間割合のみを用いて選定する場合より効率的に限定することができると考えられる．

6. まとめ

本研究では性能可搬性が低下する要因となり，性能リファクタリングにより性能可搬性の大きな向上が期待できる計算ブロックを特定する性能可搬性指標を提案した．基準システムの実行効率に，他の計算システムにおける実行効率が達していない計算ブロックと，プログラム全体に占める実行時間の割合が大きい計算ブロックは，プログラム全体の性能可搬性向上が期待できる計算ブロックであると考えられる．そのため，上記の計算ブロックを性能可搬性

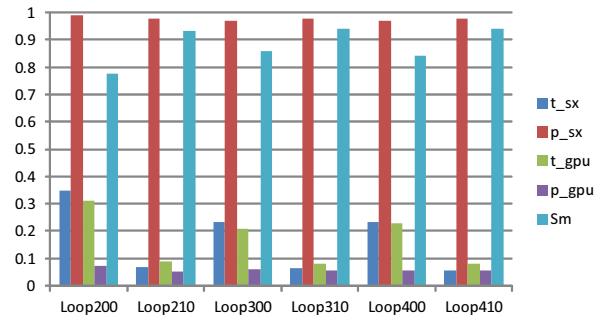

図 11 SX-9 および GPU 計算システムにおける各計算ブロックの指標

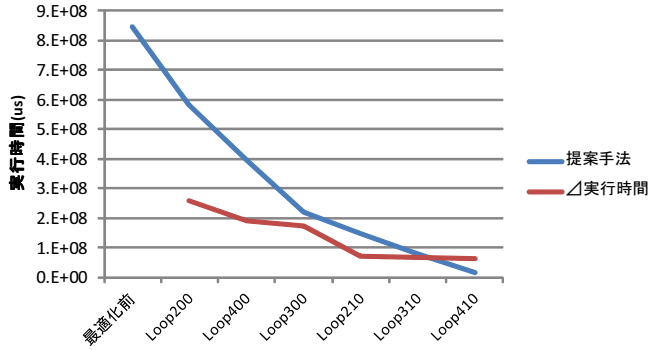

図 12 選択結果に従った性能最適化によるサブルーチンの実行時間短縮

が低い計算ブロックであると定義した．本指標は基準実行効率と他の計算システムにおける実行効率との差，計算ブロックのプログラム全体に占める実行時間の割合の大きさを考慮し，性能可搬性が低い計算ブロックを相対的に示すことができる．この指標の値が小さい計算ブロックから順に性能リファクタリングを適用することでプログラムの性能可搬性を効率的に改善する手法を提案した．

提案手法を用いたプログラムの性能可搬性向上の評価を行った結果，提案指標を用いることで性能可搬性向上幅が大きい計算ブロックを特定し，効率よくプログラムの性能可搬性を高めることが可能であった．また，実アプリケーションにおいても提案手法による選択手法はアプリケーションを効率よく性能向上させることが分かった．以上から，本研究により性能可搬性を評価するための指標を定義し，その指標に基づいて性能リファクタリングすべき計算ブロックを効率よく特定することができた．

今後の課題として，計算ブロックに包含関係が存在する実アプリケーションにおいて本提案手法の有用性を高めることが挙げられる．本研究において，計算ブロック内で他の計算ブロックの呼び出しが存在する場合，呼び出し元の計算ブロックの実行時間を，呼び出し先の計算ブロックの実行時間を除く実行時間としている．そのため，計算ブロックの包含関係をコールグラフで表し，その呼び出し順を考慮した性能ボトルネック解析を行う VTune[8] 等の性能解析ツールを用いて性能可搬性の解析を行う必要がある．上記のような性能解析ツールと本特定指標の双方を用いることで，性能可搬性低下要因となる計算ブロックを限定することが必要である．

謝辞

本研究を遂行するにあたって数値タービンアプリケーションを提供していただいた東北大学大学院情報科学研究科の山本悟教授に深く感謝します．本研究の一部は，JST CREST 研究課題「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」，および文部科学省科研費基盤研究 (B)(25280041) の支援によります．

参考文献

[1] Susan L. Graham, Peter B. Kessler, Marshall K. Mckusick, “Gprof: A call graph execution profiler, ” SIGPLAN ’82 Proceedings of the 1982 SIGPLAN symposium on Compiler construction, Pages 120-126, 1982.

[2] Samuel Williams, Andrew Waterman, and David Patterson, “Roofline: an insightful visual performance model for multicore architectures,” Communications of the ACM, Volume 52 Issue 4, Pages 65-76, 2009.

[3] Sally A. McKee, “Reflections on the memory wall,” In Proceedings of the 1st conference on Computing frontiers, CF ’ 04, Pages 162–167, 2004. ACM.

[4] 弓場敏嗣，片桐孝洋，“ソフトウェア自動チューニングの枠組み”，情報処理学会 会誌「情報処理」 Vol. 50 No. 6，Pages 478-482，2009.

[5] Peng Du, Rick Weber, Piotr Luszczek, Stanimire Tomov, Gregory Peterson, Jack Dongarra, “From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming,” Parallel Computing, Volume 38 Issue 8, Pages 391-407, 2012.

[6] John A. Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, Wen-mei W. Hwu, “Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing,” IMPACT Technical Report IMPACT-12-01, 2012.

[7] Gary M. Zoppetti, Gagan Agrawal, Lori Pollock, Jose Nelson Amaral, Xinan Tang, Guang Gao, “Automatic Compiler Techniques for Thread Coarsening for Multithreaded Applications,” Proceedings of the 2000 International Conference on Supercomputing (ICS 2000), Pages 306-315, 2000.

[8] Jack Donnell, “Java Performance Profiling using the VTune Performance Analyzer,” Intel Corporation 2200 Mission College Blvd, 2004.

[9] Satoru Yamamoto, Yasuhiro Sasao, Shoichiro Sato and Kentaro Sano, “Parallel-Implicit Computation of Three-dimensional Multistage Stator-Rotor Cascade Flows with Condensation,” 18th AIAA CFD Conference-Miami, AIAA Paper 2007-4460, June, 2007.