計算機アーキテクチャ 132－5
システムソフトウェアとオペレーティング・システム 80－5
ハイパフォーマンスコンピューティング 75－5
（1999．3．4）

複数のJava処理系における高性能計算の性能評価にむけて

伊藤茂雄† 松岡 聡†

Java言語はプラットフォーム間の可搬性が特徴であるが，JITコンパイラなどでの最適化は科学技術計算をポータブルな方法で高速に実行するためには，効果的ではない可能性がある．従って，Javaにおいて異なる処理系間の「性能の可搬性」は必ずしも保証されない．そこで我々は，複数のJava処理系のベンチマークを行える基盤を構築し，果たしてJavaが異なる処理系において性能の可搬性を維持できるかどうか検証して，将来Java言語やシステムにフィードバックすることを目指している．今回は，Cの最適化BLASコード生成システムであるATLASのJava版を開発し，複数のJava処理系のBLASのピーク性能を調べ，あらにソースレベルで可能な最適化と性能比較をすることによって，ソースレベル最適化の可搬性を検証した．

Towards Performance Evaluation of High-Performance Computing on Multiple Java Systems

SHIGEO ITOU† and SATOSHI MATSUOKA †

Despite claims of platform portability, it is not clear whether Java is suitable for high-performance scientific computing. In fact optimizations by e.g. JIT compilers may not be effective for achieving high performance in various scientific code, i.e., "performance portability" may not be guaranteed in current Java systems. To solve this situation, we are constructing a benchmarking platform for Java that candidly compares different Java systems. In particular, we have constructed a Java version of ATLAS, a program generator that outputs platform-specific optimized BLAS, to investigate the peak performance of each Java system. Then, we compared this performance to typical source-level optimizations that a user or a compiler might perform, to see how close such optimizations can approach the peak performance.

1. はじめに

Java言語はプラットフォームポータビリティが特徴であるが，現行のJavaは，高性能な科学技術計算に適切かどうかについて不透明な面が多く，JITコンパイラのようなバイトコード実行の高速化技術は，科学技術計算をポータブルに高速化できない可能性がある．特に，現在のJava処理系の実行環境は，インタプリタ・JITコンパイラ・ネイティブコンパイラと様々であり，ある特定のJavaのプラットフォーム向けの最適化は，仮にCPUやOSやプログラムが同一だとしても，実行環境の差異によって必ずしも有効になるかどうかは不明である．これに対し，Fortran, Cなどでは，コンパイラによる最適化のみならず，ユーザのソースレベルでの最適化技法が知られており，しかもそれらは異なるコンパイラやOS, CPUなどでも比較的に普遍的に適用可能である．

そこで我々は，複数のJava処理系ののベンチマーキングの環境を構築し，「性能の可搬性」の処理系間の維持の技術を研究している．その一環として，本論文では，Javaソースコードに従来の手続き型言語での最適化技法を用いることにより，各処理系でそれぞれの技法がどの程度ピーク性能にたいして有効であるか否か，処理系によって差異があるのか，を比較するための評価を行った．結果として，ある程度の共通性はあるものの，処理系によって最適化の効果に大きな差異が見られ，かつ単純な変換では(Javaプログラムによる)ピーク性能の20-30%程度にしか到達しないことが判明した

注意するのは，「性能の可搬性」は，単なる個々の処理系の絶対性能差ではなく，ある最適化を施したとき，それが全ての処理系上で同程度に効果的か否か，である．Fortran, Cにおいてはそのような前提はある程度満たされたが，Javaにおいては阻害要因が多く，それをシステマティックに検証した研究例はない．

また，本研究において個々のJava処理系のピーク性能を得ることが重要である．しかし，Javaにおいては特定の処理系で何MFlops達成したというような報告は幾つかあるものの，個々の処理系におけるピーク性能を客観的かつ普遍的に求める手法は知られていなかった．そこで，我々はTennessee大学で開発されたC用の最適

† 東京工業大学 情報理工学研究科 数理・計算科学専攻
Tokyo Institute of Technology

行列積探索ツールである ATLAS[5] システムを Java に移植し，個々の Java の処理系でのピーク性能を得，上記の比較が可能となった．これにより，Java における公正なベンチマークが可能であることも示した．

2. Java 処理系の高性能およびその可搬性の阻害要因

Java 処理系においては，Fortran, C と比較し，様々な高性能およびその可搬性の阻害要因が存在する．[4]

メモリモデル: Java では一次元配列のスカラ要素は隣接して格納されるものの，多次元配列は行ベクトルオブジェクトの配列オブジェクトの配列…といった，オブジェクト間接参照によって実現されるため，複数行が隣接して格納されているという保証はない．しかも，複数のサブ行列 (たとえば，二次元配列においては行ベクトル) がポインタ共有されていないという保証もない．さらに，Fotran や C のような COMMON やポインタによる部分行列の aliasing も不可能である．一般的な配列インデックスの依存性解析は困難になり，ブロック化アルゴリズムの効率がメモリマネージャの現在の状態と JVM の実装に高く依存し，かつブロック化した場合に部分行列のコピーが必須となるので，可搬性が保証しにくくなる[3]．さらに，ある時点でのメモリ配置の前提が GC により崩れる可能性もある．

配列境界チェック: Java は配列アクセスの境界チェックが必須である．近年ではコンパイラの静的解析で配列境界チェック除去の最適化を行ったり，スーパスカラプロセッサにおてい integer 演算であるインデックス境界チェックのコードをメインの浮動小数点演算とオーバーラップしてコードスケジューリングするなどの手法が開発されている．しかし，処理系によってはそのような最適化は行われておらず，生じる性能差により可搬性が損なわれる．

バイトコードの介在: Java の実行モデルは，共通の Java バイトコードのスタックマシンによる解釈実行であり，JIT コンパイラは，スタックマシン命令を RISC レジスタ命令にマッピングすることで最適化を行う．しかし，ソースコード上で施したコンパイラおよびユーザレベルの最適化 — アンロールやソフトウェアパイプライン等 — の構造が正しく残ったまま，ネイティブコードの最適化された構造にマッピングされるかどうかは明らかでない．

以上の問題のため，Java では性能の可搬性，特に高性能の可搬性を達成するのは必ずしも容易でない．先に述べた様に，ユーザあるいはコンパイラがソースレベルあるいはバイトコードレベルの最適化を行う場合，Fortran, C ならばせいぜいパラメタの変更 (例えば，unrolling の段数) 程度ですむが，Java ではその最適化技法そのものが，性能の低下につながることもあり得る．

3. 本研究の主眼

そこで，本研究では，いくつかの典型的な numerical kernel を Java で自然に記述し，そのソースに対して Fortran, C では確立した典型的な最適化技法でチューニングしたソースに変換することにより，それぞれの Java の処理系でどのような最適化の効果があるか，問題サイズによってどのような高速化の挙動を示すか，ピーク性能に対して何 % まで達成できるか，を測定するのを主眼とする．これにより，もし多くのプログラムにおいて，複数の Java 処理系が同一の傾向を示せば，その最適化は (高) 性能の可搬性に対して有効であるが，もし処理系によって大幅に異なれば，そのような最適化は少なくとも可搬ではなくなる．

高速化をソース変換で行うのは以下のとおり:

- Java ではバイトコードに十分ソースの情報が残っており，バイトコードのプログラムはソースコードとほぼ等価とみなせる．従ってそこでの最適化も，ほぼ等価とみなすことができる．
- 逆に，Java の場合は，ソースあるいはバイトコードレベルの最適化までしか可搬性がない．
- ソースレベルの変換だと 用いた技法によってプログラムがどのように変換され，結果としてどれだけ性能に影響を与えるかを確かめ易い．
- 一般ユーザがチューニングを行う際には，一般にはソースレベルでしか行えない．実際はこのようなユーザレベルの大局的なチューニングが性能向上に大きく寄与する場合が多い．

今回の測定は，行列積 (matrix multiply, BLAS Level 3) に対して行った．これは，Fortran, C における最適化手法が種々確立している，様々な数値計算と基本となる，などの性質のみならず，Fortran, C においてピーク性能を得る ATLAS システムの存在にもよる．この Java 版を開発することにより，それぞれの処理系のピーク性能に対する挙動の測定が可能となった．

4. ATLAS for Java

4.1 オリジナルの ATLAS の概要

オリジナルの ATLAS(Automatically Tuned Linear Algebra Software)[5] は，Tennessee 大の R. C. Whaley, Jack Dongarra らによって開発されたソースレベルの自動最適化ツールである．ATLAS は与えられたハードウェアプラットフォームに対し，ブロック数やアンロール数等の最適なパラメータを探索し，C による行列積 (BLAS Level 3) プログラムを生成する．

ATLAS の測定で用いるパラメータは，muladd, NB, MU, NU, KU, LAT などがある．Muladd は，ATLAS 内でのパラメータで 0 の場合は積算と和を分割，1 の場合は積と和を組み合わて計算する．NB はブロック数，MU, NU, KU は各ループのアンロール数，LAT はレイ

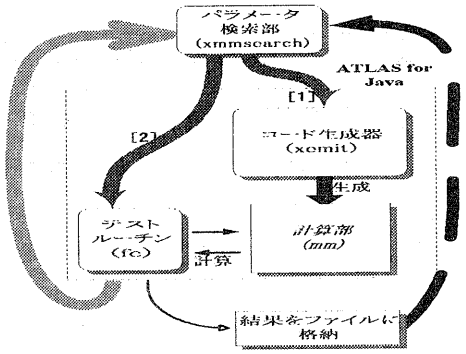

図1 ATLASのパラメータ検索部

```
m0 = a0 * b0;   m0 = a0 * b0;
m1 = a1 * b0;
                c00_0 += m0;
                m0 = a1 * b0;
c00_0 += m0;    c10_0 += m0;
m0 = a0 * b1;   m0 = a0 * b1;
c10_0 += m1;    c01_0 += m0;
m1 = a1 * b1;   m0 = a0 * b1;
   LAT = 2         LAT = 1
```

図2 レイテンシの例

テンシの値を意味する．例えば，レイテンシの値が1と2の時のソースは，図2のように出力される．

ATLASのパラメータ検索部は，これらのパラメータを決定するために以下の処理を行う

(1) L1キャッシュを推定するプログラムを起動し，複数の最適なNBの候補を得る．

(2) コード生成器(図1[1])を呼出し，MU = 1，NU = 1とする．

(3) チェックルーチン(図1[2])を起動し，結果をファイルに格納する．

(4) (2)(3)を繰り返し，ファイルに格納された結果からNBを仮に決定する．

(5) 同様に，MU = 2，NU = 2として，xemitを起動しlatecy factor(最大6)を決定する．

(6) これらをもとに，いくつかの候補のNBに対して，全てのLATに適応して最適解を得る．

このような全ての場合を探索するような最適化法は，通常のコンパイラは行なわない．実際，探索には表1に示すような多大な時間がかかる．しかし，その結果ATLASではCベースではあらゆるプロセッサでベンダが提供する最適化BLASライブラリとほぼ同様の性能を得ることが報告されている．

また，ATLAS内部では行列積を全て1次配列で表現していることに注意されたい．

4.2 ATLAS for Javaの実装

我々は，各Java処理系の行列積のピーク性能をシステマティックに得るために，ATLASをJavaに適用できるように変更したATLAS for Javaを開発した．Javaの介在により，C版のATLASのようにそのアーキテクチャにおける最高性能を得るわけではないが，生成されたプログラムは少なくともそのJava処理系上で実行できるJavaプログラムによるピーク性能である可能性は非常に高い，といえる．のみならず，生成されたJavaプログラムを観測することで，Javaにおける有効な数値計算の最適化や可搬性の指針ともなりうる．

ATLAS for Javaは主にJavaで記述されている．オリジナルのATLASに対し，新たに作成した部分は，図1の点線内である．ブロック内コード生成器がxemit.java，性能のチェックモジュールがfc.javaである．xemit.javaは指定されたパラメータに応じて行列最内周ループ計算を最適化したソースを出力する．具体的にはATLASと同じアルゴリズムでブロッキング，アンローリング，ソフトウェアパイプライニングを行う最適化されたJavaのソースを生成する．fc.javaはテストルーチンであり，ATLASのxmmsearchによって呼び出され，行列を格納するための配列を生成し，必要数のループを行い，実行時間を測定した後Mflops値を得る．

先に述べたCとJavaの違いにより，ATLAS for Javaの実装では単純なソースの書き換えのみならず，以下のような変更が必要であった:

- Javaでは配列ポインタがないため，配列のインデックスと内容を保持するクラスを定義した．Javaではオブジェクトは参照渡しのため，このクラスのインスタンスを渡すことで，配列全体およびそのインデックスを一度に参照渡しすることが可能となる．
- 関連するが，Javaではスカラ値への参照渡しができない．しかし，ATLASでは探索の結果の複数のパラメタを参照渡しで得てさらに返すことをしている(i.e., inout変数として機能している)．そこで，変数を参照渡しのためのWrapperクラスを定義した．
- 本質的ではないが，オリジナルのATLASはfloat，double型をサポートしているが，ATLAS for Javaで使用するデータ型はdouble型に制限する．

5. 性能評価

5.1 ATLASによるピーク性能の推定

ATLASで行う測定の結果は，xmmsearchが最終的に出力したデータである．ATLASが計測対象とする行列のサイズは，2次キャッシュサイズの75%を，各パラメータでのブロック数で割り切れるようにした値が用いられる．上記のようにxmmsearchは，xemitが生成するmmのルーチンをfcにチェックさせることを繰り返すことによって探索を行い，最終的に最適化されたmmのソースを出力する．本来は，上位の行列積のルーチンがATLASの生成するmmを部分行列積の求解ルーチンとして呼び出すのだが，本ベンチマークでのチェックルーチンは最内周ループのみを計算し，行列積全体の計算結果を保持していない．そのため，表 1は，最内周ルー

マシン名	処理系	Muladd	ブロック数	I-Unroll	J-Unroll	K-Unroll	レイテンシ	性能 (Mflops)	検索時間 (秒)
Ultra2	C	0	44	6	2	1	4	340.6	1383
	JDK1.2	0	24	2	2	24	4	27.4	4460
Ultra1	C	0	36	3	2	36	5	155.2	1702
	JDK1.2	0	24	2	2	24	4	14.8	7447
PC	C	0	40	2	1	40	5	197.7	1732
	Jview(MS JIT)	0	24	2	3	24	3	96.0	4445
	Jacc(HBC)	0	24	3	2	24	3	192.0	5090
	HPJ	0	28	3	2	28	2	24.0	18414
	JDK1.2	0	24	1	1	24	3	96.0	12122

Ultra2 = Sun Ultra2 model 2170 (UltraSparc 200MHz)
Ultra1 = Sun Ultra1 model 140 (UltraSparc 143MHz)
PC = PC (Pentium II 266MHz)

表 1 ATLAS での測定結果一覧

プの計算性能を測定したものである.

5.2 測定環境

測定した計算機は, 以下のものである.

- Sun Ultra1(UltraSPARC 143MHz) + Solaris 2.6
- Sun Ultra2(UltraSPARC 200MHz) + Soloris 2.6
- Intel Pentium II 266MHz + Windows NT 4.0

これらは全て, 以下である.

- 1 次キャッシュ 16KB(命令)+16KB(データ)
- 2 次キャッシュ 512KB

今回, 実験に使った Java の処理系は, 以下である.

- JDK1.2 for Solaris (JIT)
- JDK1.2 for Win32 (JIT)
- IBM High Performance Compiler for Java v.Beta A12(Native)
- 富士通 J Accelerator v1.1L10 (Native)
- Microsoft Jview Version 6.00.8167 (JIT)

コンパイラあるいはユーザが行うと想定したプログラム変換によるハンドチューニングは以下のものを行った. これらは, ATLAS の場合と異なり, Java の 2 次元配列を直接用いた. 測定行列サイズは, 4 行 4 列から 512 行 512 列まで 4 刻みで測定した.

- アンローリング (i 重 j 重 k 重) (i, j, k)=(1, 1, 4), (1, 1, 8) , (2, 1, 4) , (2, 1, 8), (2, 2, 1) , (2, 2, 2) , (2, 2, 4) , (2, 2, 8) , (2, 4, 1) , (4, 2, 1) , (4, 4, 1) , (4, 4, 4) , (4, 4, 8)
- ブロック化 2x2, 4x4, 8x8, 16x16, 32x32, 64x64, 128x128 重
- ストリップマイニング 2, 4, 8, 16, 32, 64, 128 重
- キャッシュコピー
- ソフトウェアパイプライン 1 重, 4 重, 8 重

また, 比較として, C 言語で同内容のプログラムを書いたものを用いた. 処理系とオプションは, 以下の通り:

- Sun's cc -dalign -fsingle -xtarget=ultra2/2170 -xO5 -xarch=v8plusa
- Sygnus cygwin-b20.1 gcc -fomit-frame-pointer -funroll-loops

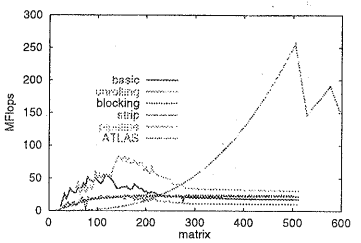

図 3 J Acceleratorfor Win32

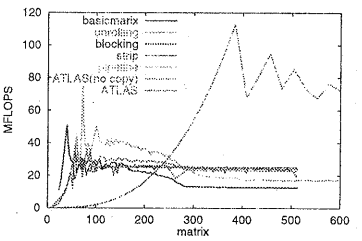

図 4 MS jview for Win32

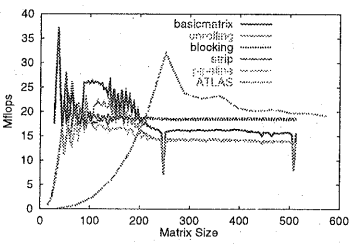

図 5 IBM High Performance Compiler for Java for Win32

6. 測定結果

手でチューニングした行列積のなかで, 一般的に最も

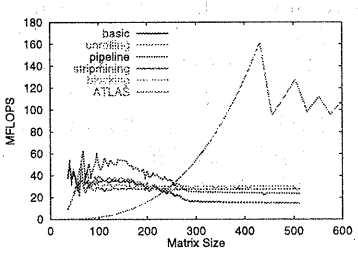

図6 JDK1.2 for Win32

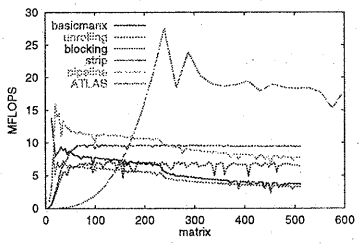

図7 jdk1.2 for Solaris

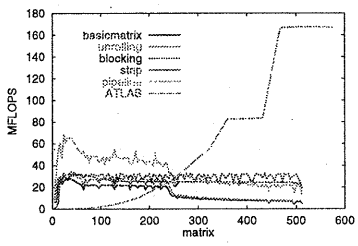

図8 C for Solaris

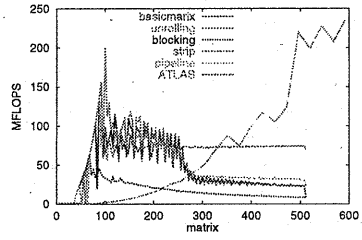

図9 C for Win32

高い性能を示したものは, ソフトウェアパイプライン技法である. HPJ を除く多くの処理系で最も高い性能をしめす. 一方, ブロック化とストリップマイニングは, C や Fortran の結果と同様に行列サイズが大きくなっても性能が低下しない. 両者ともにほぼ同じ結果となる. しかし, パイプライニングあるいは何もしない場合と比較して, 性能が低い場合が多い.

アンローリングは, Java において最内周を 4 段で展開すると最も性能が向上する報告[2] もあるが, 今回の測定では, アンロール段数が多い程高い性能を出すことを確認した. これは, 次に述べる Atlas の結果と一致していて, 興味深い. しかし, どの技法でも小さい行列サイズでピークになったり, その後フラットになる.

図 3, 5, 4, 6, 7, 8, 9での ATLAS の xmmsearch の結果のパラメータは, 全て最内周をブロック数分アンローリングしているものとなっている. これは, 一つには Java におけるループを構成するブランチのコストが高いことに起因していると推測されるが, 詳細は定かではない. また, 最内周を全てアンロールするため, ブロック化やアンロールでコードが肥大化し, そのコストがブロッキングによる性能を上回ってしまい, 性能が低下する. この現象はオリジナルの ATLAS でも報告されているが, JIT コンパイラで特に顕著というわけでもなく, スタティックコンパイラでもほぼ同様に現れていることが興味深い. 行列サイズがある程度大きくなると, 本来の性能が現れる; しかし, 処理系によって行列サイズが異なるが, 行列サイズが 500 以降で性能低下が現れる. 後者の現象は C 言語では現れず, その原因は調査中である. PentiumII/WindowsNT では全体的に ATLAS による性能は C と比較しても大変高く, 100MFlops から 200MFlops 程度出ている. 一方, Ultra ではパラメタの傾向は類似しているが, 性能は C の 1/10 程度である.

ATLAS の検索時間は, 表 1によると, Java の場合は C と比べて遅い. NT でも性能差以上に違いが出ている. これは検索時間が各処理系のコンパイル時間に依存した結果となっていることに起因すると考えられる.

ATLAS とソース変換を比較すると, 後者の性能は, IBM の HPJ を除いて, 小さい問題サイズでは ATLAS より優位だが, すぐに飽和し, 大きい問題サイズでは ATLAS の敵ではない. どの処理系でも 1/2 から 1/5 程度の性能しか達成できないのが現状である. したがって, 通常のプログラム変換では, ピーク性能は到底到達できないが, この傾向は C でも実は見られる.

以下, それぞれの処理系における結果の特徴を述べる:

6.1 富士通 J Accelerator(HBC)

J Accelerator は富士通が開発したネイティブコンパイラ HBC(High speed Bitecode Compiler) を含むパッケージである. C++, C や FORTRAN90 のコンパイラで用いた共通のバックエンドを使用し, バイトコードをコンパイラの中間言語に変換することで, HBC を構成している. HBC は Java 言語仕様を完全に満たしたものではなく, 速度向上のため一部の Java の安全性の機能などが満たされていない (コンパイラオプションによって互換モードにすることも可能である.). 実際, 我々は HBC では配列の境界チェックや Exception の正しいハンドリングを行っていないことをテストにより確認した.

一方, このような工夫により, HBC は大変高速である. プログラム変換では, ソフトウェアパイプライン法

を用いることにより, 性能が最大 99Mflops まで引き出せた. さらに, ATLAS for Java で得たソースを HBC で実行するとピーク性能が CPU 最大性能に達する (ブロックサイズ 504x504 のとき, 256Mflops で, 一般的にも 200MFlops 程度). これは, C のピーク性能に近い.

6.2 Microsoft Jview

Microsoft の Java 処理系である. ここでは, ブロック内の行列積を C=AxB における C に代入する時, 単純に計算のみで代入しない時と比較して性能が 1/3 程度以下になってしまうという, ほかの処理系と比較して特異な現象を示した. 原因は調査中だが, 本稿ではひとまず代入する時 (ATLAS と書かれたもの) と代入しない時 (ATLAS(no copy) と書かれたもの) のグラフを表示している. 両者ともに, 行列サイズが 264x264 までは差がないが, それ以降 Mflops 値は変化しなくなる.

6.3 IBM HPJ

HPJ は IBM のネイティブコンパイラである. 用いたのは Web での公開バージョンで, 現在の商用バージョンではない. HPJ は行列サイズが 200 前後まで手を加えないほうが速い. 300 以上での同じ行列サイズでは性能比はたかだが 2 倍である. (他の処理系では, ATLAS 以外の最大性能より 3, 4 倍以上). 全体的に, 性能は HBC のみならず JIT コンパイラと比較しても大変悪い. 最近の同一研究グループによる実験的な HPJ では, RS6000 で 100MFlops 程度の行列積の性能が報告されている. これは Java の本来の配列と異なる特殊な配列クラスを定義し, それをビルトインの配列とコンパイラがみなして通常の最適化を施す.

6.4 JDK1.2

NT と Solaris のグラフの形状はほとんど似ているが, ATLAS for Java のグラフの立上りの位置が異なる. ソフトウェアパイプラインが行列サイズ 200 後半まで上回り, ATLAS for Java を除けばブロック化が最も性能が良くなる. また Solaris の JDK1.2 では, ブロック化がストリップマイニングより 1.5 倍の性能差がでている. 先に延べたように, 全体的に Ultra の Java は性能が大変悪いが, その原因は調査中である.

7. 関連研究

Java Numerical compuing in Java[3] は Java での数値計算のライブラリである. 初期バージョンでは, BLAS, LU や QR 分解などをサポートしている. B. Blount[2] では, LAPACK を Java で実装した JLAPACK を用いて性能評価している. 他にも NIST の Pozo らが Java における BLAS の実装や, SciMark などのベンチマークの結果を報告しているが☆, 性能の可搬性や, 処理系のピーク性能に言及した研究はまだない.

また, ソース上での最適行列積生成ツールの関連研究として, PHiPAC がある. PHiPAC[1] は, ポータブルで高性能な ANSI C の数値ライブラリを開発するための GEMM 生成ルーチンである. ATLAS が各アーキテクチャ毎に最適化ソースを供給するのとは異なり, 様々なアーキテクチャで使用可能なように分岐を挿入した GEMM を供給する. しかし, ATLAS と比べると, あらゆるルーチンに同種の測定を繰り返すため, パラメータ発見にかかる時間が長くかかる.

☆ http://gams.nist.gov/javanumerics/

8. まとめ

本稿では, Java ソースコードに従来の最適化技法を用いることにより, どの程度高速化が達成されるのか, 性能の可搬性は達成できるのか, を種々の処理系を比較する環境を構築し計測, 評価を行った. この測定を行うにあたり, ATLAS for Java[5] を開発し, Java 処理系のピーク性能を調べた.

ATLAS for Java を用いて, ネイティブコンパイラでは C 並の性能を, JIT コンパイラでも行列積で 100Mflops を超える性能を引き出すことが可能であることを確認した. 一方, 処理系によっては, ピーク性能は C の 1/5 から 1/10 程度であり, 処理系の質の差が見られた. これにより, 今後の最適化コンパイラ構築の指針を得た.

同じ変換によるチューニングを施したソースでも, 場合によっては処理系により極端に異なる結果になることがあるため, 現状では一般的な性能の可搬性を達成するのは難しいことが判明した. 今後, より詳しい知見を得るために, 他の科学技術計算でも同様な調査をする予定である. また, 得られた知見をもとに, 性能の可搬性を達成するコンパイラフロントエンドを開発する予定である.

参考文献

1) Jeff Bilmes, Krste Asanovic, Chee-Whye Chin, and Jim Demmel. Optimizing matrix multiply using PHiPAC: a portable,high-performance, ANSI C coding methodology. ACM ICS '97, July 1997.

2) Brian Blount and Siddhartha Chatterjee. An evaluation of java for numerical computing. Second International Symposium ISCOPE '98, July 20-21 1998.

3) Ronald F.Boisvert, Jack J.Dongarra, Rlodan Pozo, Karin A.Remington, and GG.W.Stewart. Developing numerical libraries in java. Proceedings of ACM 1998 Workshop on Java for High-Performance Network Computing, 1998.

4) Satoshi Matuoka and Shigeo Itoh. Is java suitable for portable high-performance computing?-preliminary reports on benchmarking diffent java platforms-. POOSC'98 Workshop, July 20-21 1998.

5) R.Clint Whaley and Jack J.Dongarra. Automatically tuned linear algebra software. *IEEE Supercomputing '98 (CD-ROM).*