Performance of Linear Algebra Routines in Java and C

Franz Franchetti
Christoph Ortner
Christoph W. Ueberhuber

AURORA TR2002-06

Institute for Applied and Numerical Mathematics
Technical University of Vienna
Wiedner Hauptstrasse 8-10, A-1040 Wien, Austria

E-Mail: franz.franchetti@tuwien.ac.at
cortner@aon.at
christof@aurora.anum.tuwien.ac.at

The work described in this report was supported by the Special Research Program SFB F011 "AURORA" of the Austrian Science Fund FWF.

Abstract

This report describes the results of a performance assessment study of linear algebra packages in Java. Implementations of matrix multiplication, LU factorization and Cholesky factorization are compared to corresponding routines provided by CLapack and Atlas.

While the performance of Atlas remains unreachable for both the Java implementations as well as CLapack, for every benchmark there is a Java implementation which reaches CLapack's performance in the out-of-cache cases and often CLapack is even outperformed. For smaller problem sizes there are Java packages which reach 30 to 50 % of CLapack's performance.

Results like these show that after some further development, Java is seriously to be considered as the language of choice for many scientific applications.

Contents

Chapter 1

Introduction

1.1 Java's Deficiencies

In the first Java versions, the language had numerous deficiencies to overcome if it was to be used for numerical computing. The following list does not claim to be complete, but is intended to cover the most important issues, especially in terms of run time behaviour. Detailed discussions of these topics can be found in [9], [19, 20, 21, 22], [24, 25, 26, 27, 28, 29, 30, 31].

- The requirement of exact reproducibility of numerical results obtained on any platform was a great balk, which refused access to many performance relevant hardware features available nowadays. For instance, the fused multiply-add (FMA) operation, available on many systems, cannot be used, because it produces another result than a simple combination of a multiply and an add operation. The `strictfp` keyword, which has been introduced with Java 1.2, loosened this rule a little, but the issue is not entirely solved yet (see [17]).

- Complex arithmetic is essential in many areas of scientific computing. Java does not have a `complex` data type, although this is not a fatal flaw since new types are easy to define. However, since Java does not support operator overloading, one cannot make such types behave like the primitive types `float` or `double`. More important than syntactic convenience, however, is that not having complex arithmetic in the language can severely affect the performance of applications. This is because compilers, as well as the JVM, are not able to perform conventional optimization on complex arithmetic code because they are unaware of the semantics of the class. Since complex arithmetic is so pervasive it is necessary to establish community consensus on a Java interface for complex classes (see [19]).

- Array bounds checking produced an immense overhead on the first JVMs, especially, when algorithms consisted mainly of array accesses. There are already many techniques, called *array bounds check elimination*, that dramatically improve performance, but still stay behind unchecked code. The experimental Timber compiler [16], which allows array bounds checking to be turned off, gains on average 10 % performance speed-up using this feature. In extreme cases, the performance gain is around 30 % (Midkiff, Moreira, Snir [26]).

- The lack of operator overloading is mainly an issue of readability of code, but a very important one, especially since one of Java's main goals is to achieve user-friendliness (Veldhuizen [31]).

- The lack of some kind of lightweight classes, with less overhead, than Java's class model often forbids optimizing in terms of performance and program structure as well. This topic is obviously linked closely to the matters of operator overloading and the `complex` data type (Veldhuizen [31], Bacon [18]).

- Currently, multidimensional arrays in Java are implemented as *arrays of arrays*. The introduction of true multidimensional arrays is necessary ([13]).

1.2 Numerical Computing in Java

This section gives an overview of some of the most important attempts to introduce numerics to the Java platform.

1.2.1 Java Numerics

The Java Grande Numerics Working Group [9] made several official proposals for extending Java's specification, many of them addressing the issues mentioned in Section 1.1. Furthermore there are proposals suggesting framework classes, for instance, for numerical linear algebra. For a more detailed description of the working group's activities see `http://math.nist.gov/javanumerics/`.

The FASTFP Keyword

The introduction of the keyword `fastfp` into the Java specification would allow the use of modern floating-point hardware and optimization techniques to make Java's performance competitive with C and Fortran.

(see `http://java.sun.com/aboutJava/communityprocess/jsr/jsr_084_fpe.html`).

The Complex Data Type

The most obvious approach to address the lack of a `complex` data type is to introduce a new class. This was proposed by the Java Grande Numerics Working Group [9] and has been implemented by Visual Numerics [8]. Wu [32] and Moreira et al. [29] show, that this approach delivers completely satisfactory performance, if the compiler is able to recognize complex objects, and is aware of its semantics in order to allow the optimization of complex arithmetic code .

Multidimensional Arrays

The NINJA group described and implemented a multidimensional array package, capable of achieving almost Fortran-like performance using IBM's high performance compiler, which is not restricted to Java's rigid floating-point conventions (Moreira et al. [29], Ninja [13]).

BLAS, Linear Algebra Packages

JAMA [5] is the official Java Grande proposal of a numerical class library to be added to the Java specification. But there are many more packages available, that will be described in the followeing section. Detailed performance data are provided in Chapter 2.3.

1.2.2 Available Software

This section describes a representative sample of numerical linear algebra software written in Java. This freely available software has been used in the performance assessments described in this report.

JAMA

JAMA [5] is Java Grande's official proposal for a linear algebra package in Java. JAMA has been implemented at NIST [12]. It currently supports only real dense matrices which are stored internally as native Java arrays (i. e., `double[][]`). It also provides standard matrix decomposition routines.

JAMPACK

JAMPACK [6] is a sibling of JAMA. JAMPACK has been developed at NIST and the University of Maryland in order to examine alternate designs of matrix implementations in Java. It fully supports complex matrices (but lacks implementations optimized for *real* arithmetic), diagonal matrices, and includes many decomposition algorithms.

NINJA

IBM's NINJA project [13] provides an array package with highly optimized multidimensional arrays and support for some of the BLAS routines.

JNL

JNL [11] is a numerical library developed by Visual Numerics [8]. JNL provides a `double` precision `Complex` class as well as linear algebra routines which operate on Java arrays of type `double[][]` and `Complex[][]`. Additionally it contains

classes which provide statistical functions and special functions (Bessel functions, etc.). JNL makes intensive use of the `Complex` class even for plain arithmetic and therefore its performance in the complex benchmarks is very poor. For this reason some of JNL's routines are not shown in some of the figures.

COLT

The COLT library [2] provides fundamental general-purpose data structures optimized for numerical data, such as resizable arrays, dense and sparse matrices (multi-dimensional arrays). COLT contains many algorithms for numerical linear algebra, statistics, data analyis, etc.

OR Objects

OR Objects [14] is a collection of Java classes for developing operations research, scientific and engineering applications. Among many others it provides complex and matrix packages as well as the BLAS and some linear algebra routines.

JAVA LAPACK

Java LAPACK [7] from the University of Tennessee was obtained by automatically translating the original Fortran code into Java using `f2j` [3]. Currently it supports only real arithmetic.

JLAPACK

JLAPACK [10] is an implementation of the general linear equation solver of LAPACK and parts of the BLAS. JLAPACK takes advantage of object-oriented features of Java. It was produced in the HARPOON [4] project of the University of North Carolina.

1.2.3 ATLAS

All linear algebra packages described in Section 1.2.2 are implemented and optimized manually, as is the LAPACK library. Considering the enormous performance gain of the automatically tuned ATLAS routines compared to standard LAPACK routines (see Section 2.3), it would be interesting to see whether similar results can be achieved using a Java version of ATLAS.

AJAPACK

The AJAPACK is the only Java ATLAS implementation that could be found. It was developed at the Tokyo Institute of Technology. It uses CATLAS and a modified JLAPACK to produce optimized code. AJAPACK especially targets

parallel implementations of LAPACK and BLAS routines. Unfortunately, due to a lack of English documentation it was not possible to include AJAPACK in the performance assessment. Depending on the system, the authors claim to achieve 20 to 50 % of CATLAS' performance in serial benchmarks and say that they sometimes even outperform it in parallel ones [23].

1.2.4 Java Dialects

There are several extensions and/or dialects of Java, which target the problems described in Section 1.1 by changing the language specification.

BORNEO

To address Java's problems, the Borneo language (Borneo [1], Darcy [21]) changes and extends Java so that all IEEE 754 features can be expressed and new numeric types can be easily created. Borneo allows either better hardware utilization than Java or (nearly) exact reproducibility while in all cases being predictable [1].

Currently Borneo exists only as a specification. It hasn't been implemented yet.

SPAR

Apart from a few minor modifications, Spar [15] is a superset of Java. This provides Spar with a modern, solid, language as basis, and makes Spar more accessible. Spar extends Java with constructs for parallel programming, extensive support for array manipulation, and a number of other powerful language extensions.

Timber [16] is a modified gcc compiler which is able to handle Spar/Java code.

KAVA

Kava chooses a far more aggressive but uniform approach. In Kava there are no primitive types; instead, object-oriented programming is provided down to the level of single bits. Types such as `int` can be explicitly programmed within the language. While the language maintains a uniform object reference semantics, efficiency is obtained by making heavy use of unboxing and semantic expansion (see [18]).

Kava can be compiled with a modification of the Jikes compiler (Bacon [18]), but it is still in an experimental state.

Chapter 2

Performance Assessment

2.1 The Test Environment

All performance experiments were run on a PC with an Athlon XP 1800+ (1533 MHz) processor and 768 MB DDR-SDRAM main storage.

The tests in Java were performed using IBM's JDK 1.2.2 under WindowsXP. In addition to the software packages a simple three loop version of the real matrix-matrix multiplication was implemented to gain an impression of the quality of the algorithms used in the Java packages (see Fig. 2.13 and 2.14).

As standard of comparison CLapack and Atlas were chosen. The respective routines were run under Red Hat Linux 7.1 (Kernel 2.4.2-2) using the gcc 2.96 compiler.

The names of the tested software packages will be abbreviated in this chapter as shown in Table 2.1.

Package	Abbreviation
Atlas	atlas
CLapack	c
Colt	colt
Java Lapack	f2j
Jama	jama
Jampack	jp
JLapack (HARPOON)	harp
JNL	jnl
Ninja	ibm
OR-Objects	or
trivial implementation	tr

Table 2.1: Abbreviations denoting the various software packages of the assessment study.

2.2 Benchmark Description

All presented benchmarks are performed in real and complex double precision arithmetic.

All results are presented in two graphics, one showing the absolute performance measured in Mflop/s and one showing the relative performance compared to CLapack. In the absolute performance graphics it becomes obvious how superior the Atlas is compared to all other packages. Therefore it is not included in the relative graphics. This enables better readability of the Java performance.

The size n of the matrices was chosen close to the powers of 2 from $2^6 = 64$ to $2^{10} = 1024$, i.e., $n = 50, 100, 200, 500, 1000$; values for which the algorithms won't suffer as badly from cache associativity problems as they would for 2^k values. Both values were tested, in order to gain an idea how well the packages perform optimizations needed in such cases. In the main figures the powers of two are not included, because many packages deliver a far worse performance for these values (see Section 2.3.7).

Table 2.2 shows which packages provide working implementations of the benchmarked algorithms.

Package	MatMul	LU	Cholesky
Atlas	**r**, **c**	**r**, **c**	**r**, **c**
CLapack	**r**, **c**	**r**, **c**	**r**, **c**
Colt	**r**	**r**	**r**
Java Lapack	**r**	**r**	**r**
Jama	**r**	**r**	**r**
Jampack	**c**	**c**	**c**
JLapack (HARPOON)	**r**, **c**	**r**, **c**	
JNL	**r**, **c**	**r**, **c**	**r**, **c**
Ninja	**r**		
OR-Objects	**r**, **c**	**r**	
trivial implementation	**r**		

Table 2.2: Packages included in the numerical tests. **r** marks a real arithmetic implementation, **c** a complex one.

Matrix Multiplication

The matrix multiplication routines of Java are compared to the `gemm` routines of Lapack. As test matrices random matrices were used.

LU Factorization

In this test the LAPACK routine `getrf` was compared to any existing LU factorization routines from the Java packages. Again random matrices were used as test matrices.

Cholesky Factorization

Tha Java routines were compared to the LAPACK routine `potrf`. As test matrix $A = (\min(i, j))_{i,j=1..n}$ was used, which factorizes to $L \times L^T$ with L being lower triangular and its elements being all 1.

2.3 Numerical Results

2.3.1 Real Matrix Multiplication

Except for medium matrix sizes, JAMA leads the Java group. It delivers 80 % of CLAPACK's performance for the bigger matrix sizes.

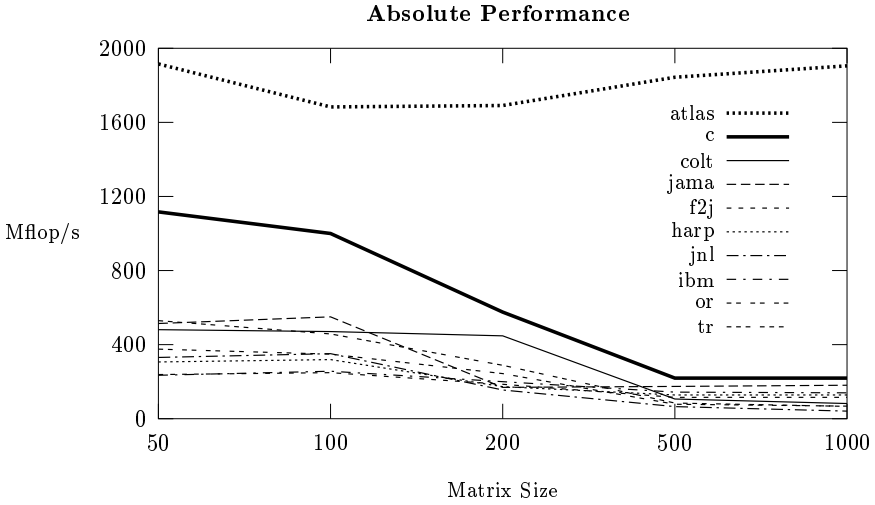

Figure 2.1: Performance (Mflop/s) of the real matrix multiplication on an AthlonXP 1800+ comparing CLAPACK and ATLAS (using gcc) to several Java implementations (using IBM JDK 1.2.2). The ATLAS and LAPACK routine `dgemm` was used for comparison.

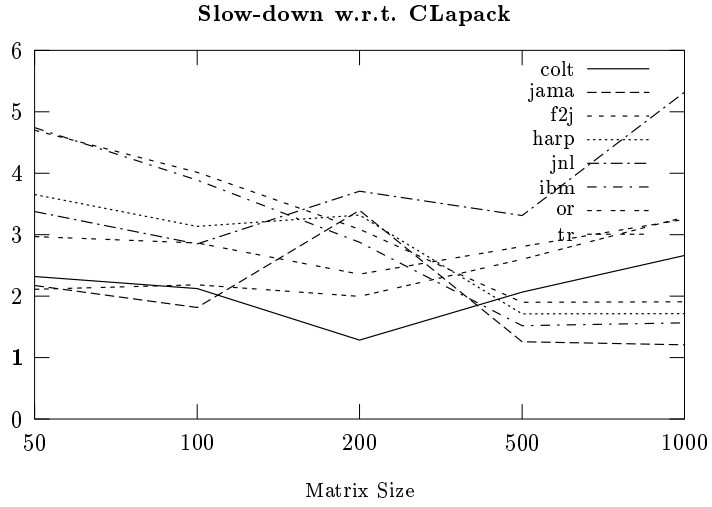

Figure 2.2: Slow-down of several implementations of the real matrix multiplication in Java (using IBM JDK 1.2.2) with respect to the `dgemm` routine of CLAPACK (using gcc) on an AthlonXP 1800+.

2.3.2 Complex Matrix Multiplication

JAMPACK delivers an extraordinary performance, reaching more than 80 % of CLAPACK through the whole test.

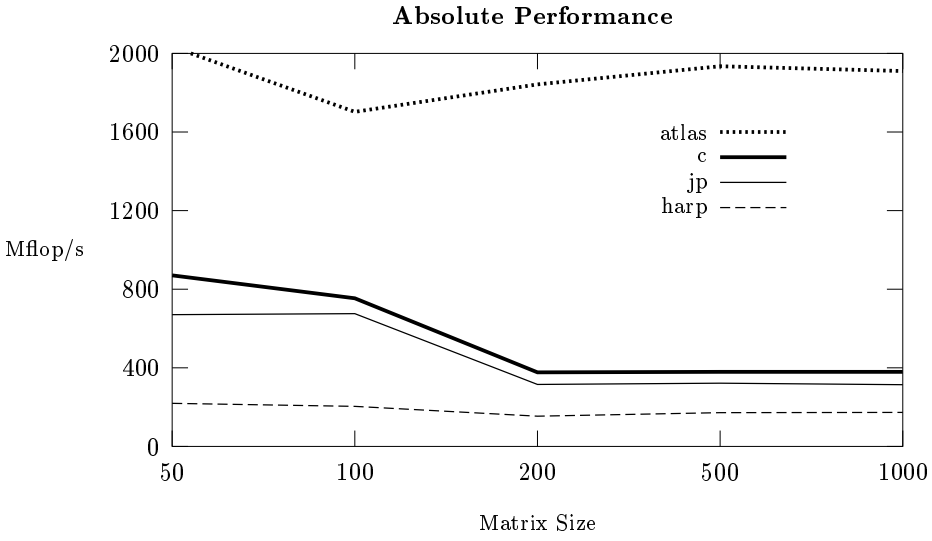

Figure 2.3: Performance (Mflop/s) of the complex matrix multiplication on an AthlonXP 1800+ comparing CLAPACK and ATLAS (using gcc) to several Java implementations (using IBM JDK 1.2.2). The ATLAS and LAPACK routine `zgemm` was used for comparison.

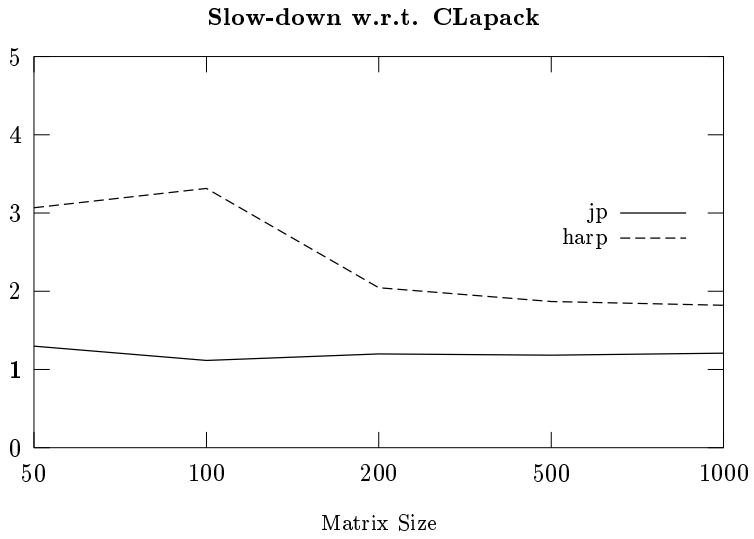

Figure 2.4: Slow-down of several implementations of the real matrix multiplication in Java (using IBM JDK 1.2.2) with respect to the `zgemm` routine of CLAPACK (using gcc) on an AthlonXP 1800+.

2.3.3 Real LU Factorization

In this test Java cannot really reach up to CLapack. Jama and Java Lapack are barely able to reach 50 % of CLapack's performance together.

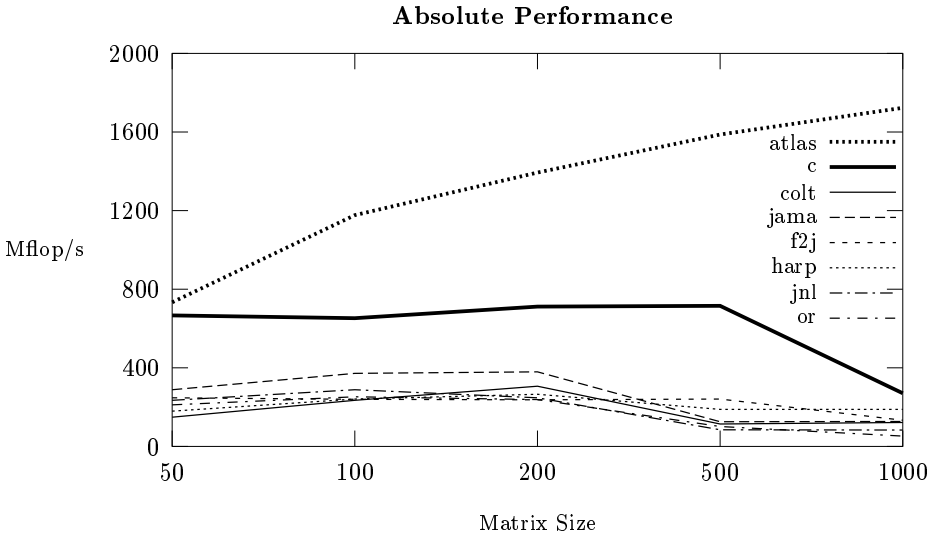

Figure 2.5: Performance (Mflop/s) of the real LU factorization on an AthlonXP 1800+ comparing CLapack and Atlas (using gcc) to several Java implementations (using IBM JDK 1.2.2).

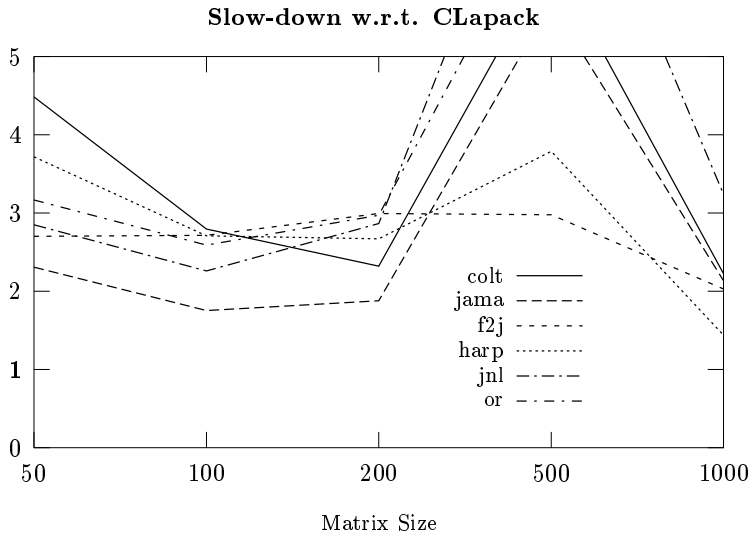

Figure 2.6: Slow-down of several implementations of the real LU factorization in Java (using IBM JDK 1.2.2) with respect to the `dgetrf` routine of CLapack (using gcc) on an AthlonXP 1800+.

2.3.4 Complex LU Factorization

This test compares Java more favorable than the real LU test. JLapack stays as close as 50 % to CLapack for all sizes and almost reaches it for the 1000×1000 test.

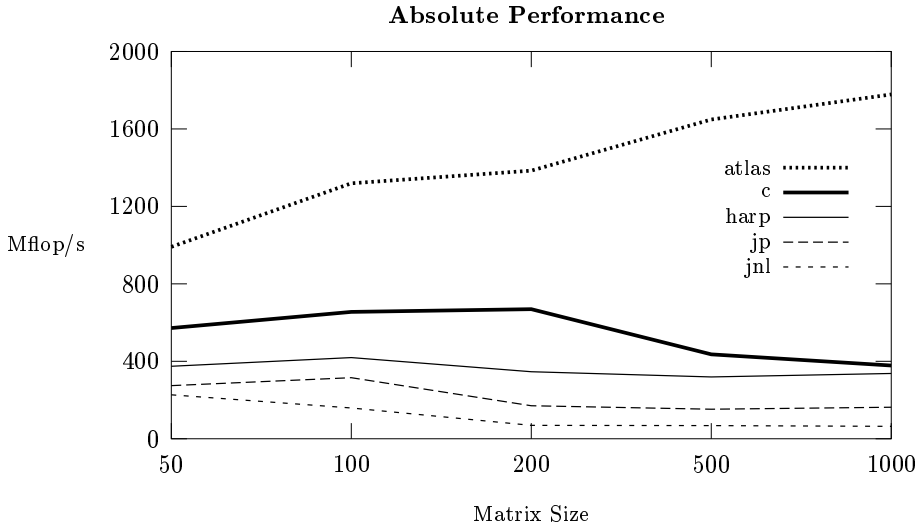

Figure 2.7: Performance (Mflop/s) of the complex LU factorization on an AthlonXP 1800+ comparing CLapack and Atlas (using gcc) to several Java implementations (using IBM JDK 1.2.2).

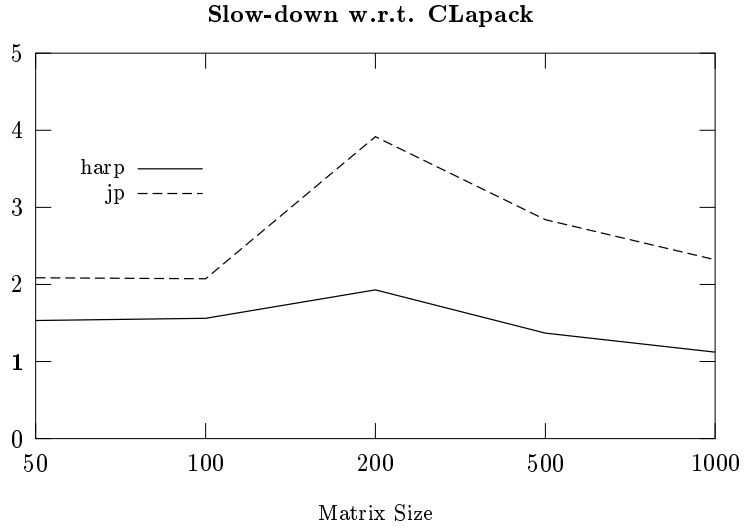

Figure 2.8: Slow-down of several implementations of the complex LU factorization in Java (using IBM JDK 1.2.2) with respect to the `zgetrf` routine of CLapack (using gcc) on an AthlonXP 1800+.

2.3.5 Real Cholesky Factorization

This is another test, where Java does not perform so well. Surprisingly, JNL delivers the best Java performance.

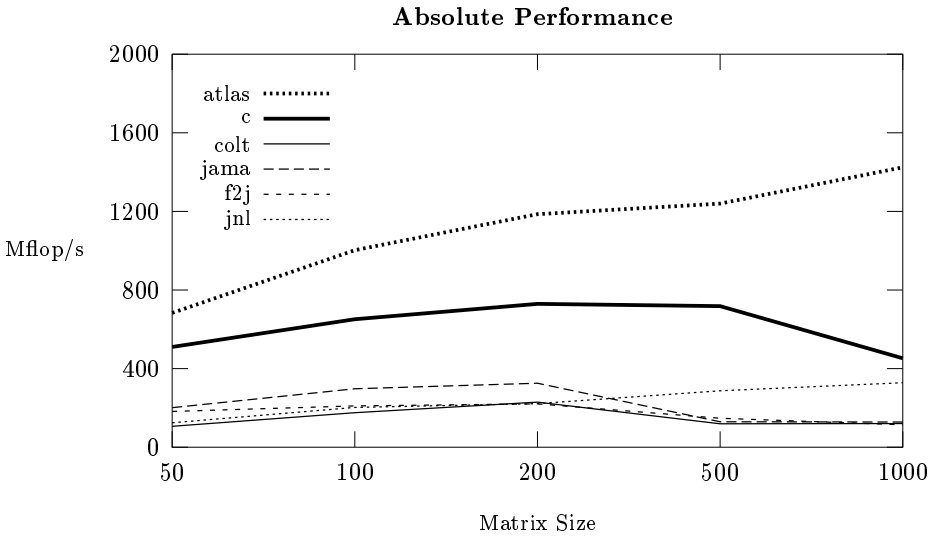

Figure 2.9: Performance (Mflop/s) of the cholesky algorithm on an AthlonXP 1800+ comparing CLapack and Atlas (using gcc) to several Java implementations (using IBM JDK 1.2.2).

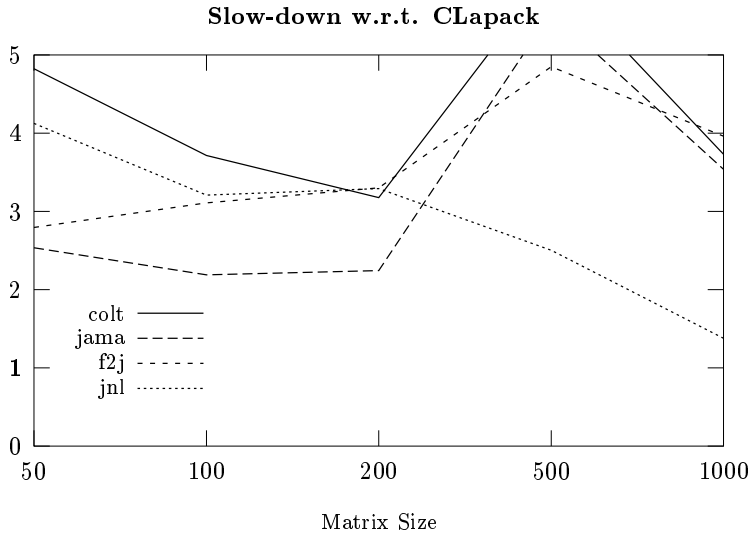

Figure 2.10: Slow-down of several Java implementations (using IBM JDK 1.2.2) of the cholesky algorithm with respect to the `dpotrf` routine of CLapack (using gcc) on an AthlonXP 1800+.

2.3.6 Complex Cholesky Factorization

Even outperforming CLapack for some tests, again Jampack is able to deliver a performance that is absolutely satisfactory.

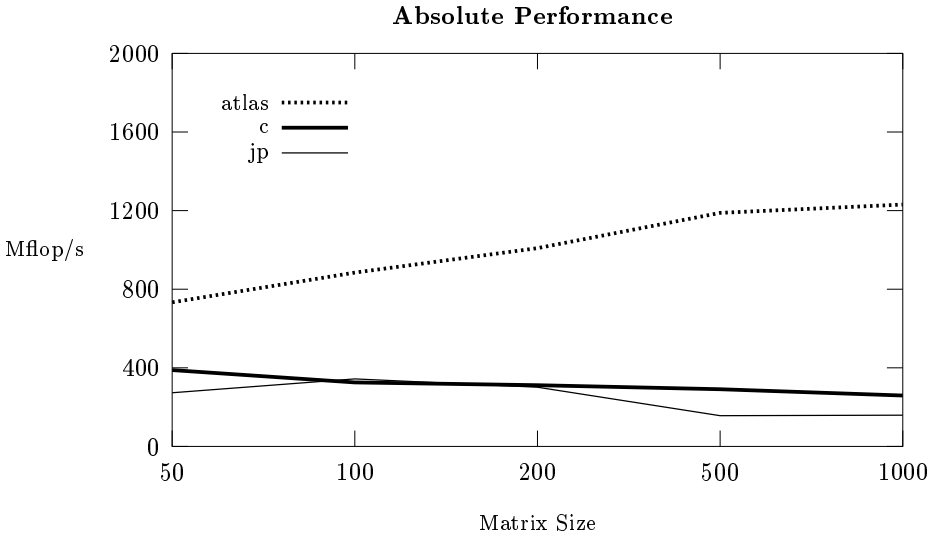

Figure 2.11: Performance (Mflop/s) of the complex cholesky algorithm on an AthlonXP 1800+ comparing CLapack and Atlas (using gcc) to Jampack's implementation in Java (using IBM JDK 1.2.2).

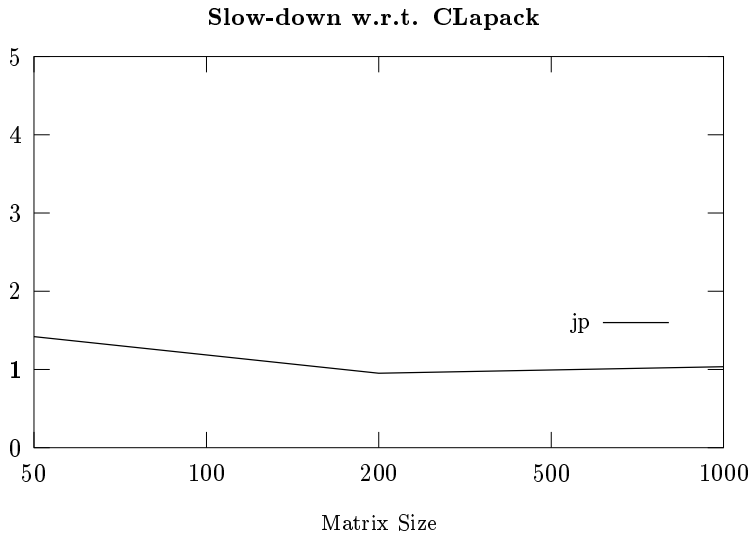

Figure 2.12: Slow-down of Jampacks's implementation of the complex cholesky algorithm in Java(using IBM JDK 1.2.2) with respect to the `zpotrf` routine of CLapack (using gcc) on an AthlonXP 1800+.

2.3.7 Real Matrix Multiplication for Power of 2 Sizes

JAMA's performance does lives up to its promises in the previous MatMul test, but other packages, like COLT or JNL suffer from a great performance deterioration for the 2^k matrix sizes.

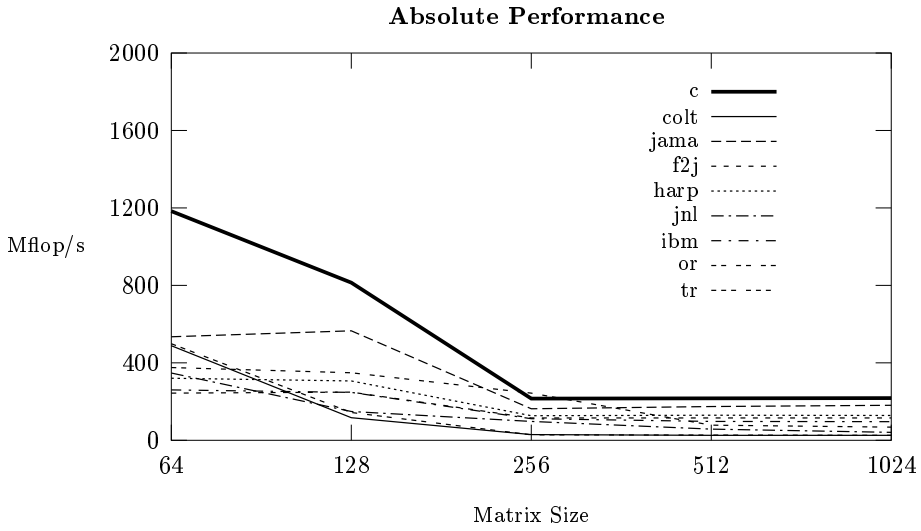

Figure 2.13: Performance (Mflop/s) of the real matrix multiplication on an AthlonXP 1800+ comparing CLAPACK and ATLAS (using gcc) to several Java implementations (using IBM JDK 1.2.2) for power of 2 matrix sizes.

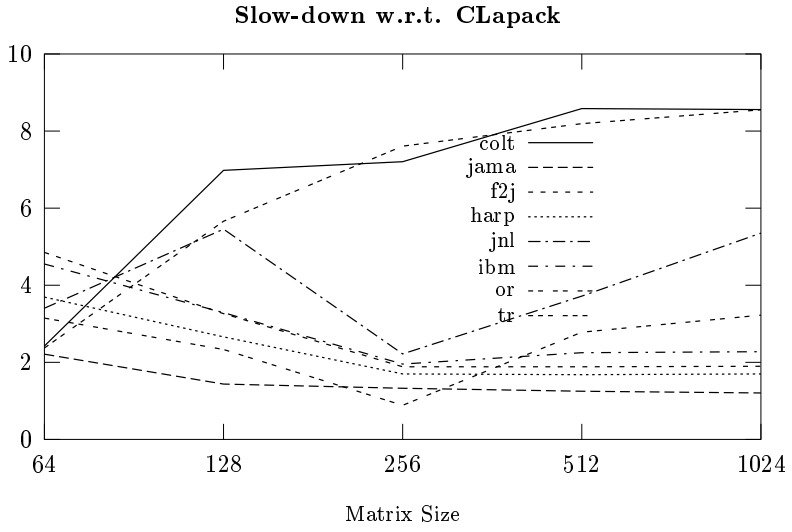

Figure 2.14: Slow-down of several implementations of the real matrix multiplication in Java (using IBM JDK 1.2.2) with respect to the `dgemm` routine of CLAPACK (using gcc) on an AthlonXP 1800+ for power of 2 matrix sizes.

Conclusion

While ATLAS remains unreachable for both the Java implementations as well as CLAPACK, for every Benchmark there is a Java implementation, which reaches CLAPACK's performance for the 500×500 and the 1000×1000 problems, and in some cases even outperforms CLAPACK. For smaller problem sizes there are always Java packages which stay as close as 40 to 50 %.

The best results for Java were obtained in the complex benchmarks. Here JLAPACK and JAMPACK deliver an excellent performance. JAMPACK's matrix multiplication is able to stay as close as 80 %. JLAPACKS's LU factorization holds 75 % of CLAPACK for small problems and even reaches CLAPACK for the bigger ones. Also the complex Cholesky factorization shows similiarly promising results. JAMPACK beats CLAPACK for some vector lengths and stays as close as 75 % for most of the rest (see Sections 2.3.2, 2.3.4 and 2.3.6).

Most of the packages though, do not deliver competitive performance. Especially the bad results for large problem sizes which are powers of two (see Section 2.3.7 and the similarity to the results of the trivial matrix multiplication show how little effort has been invested into sophisticated optimization.

Also no Java library provides a set of numerical routines that can be compared to CLAPACK, which is another argument against using Java for numerical computing right now. The only Java package which provides all benchmark routines (JNL) has such a bad performance, that it was not included in all of the tests (see Table 2.2).

However, if equivalent results could be achieved with a Java version of ATLAS, Java might become the language of choice for many scientific applications, mainly due to its most outstanding advantage over traditional environments, like Fortran or C: a maximum degree of portability.

References

[1] *Borneo Homepage:*
`http://www.jddarcy.org/Borneo`.

[2] COLT *Homepage:*
`http://nicewww.cern.ch/ hoschek/colt/index.htm`.

[3] `f2j` *Homepage:*
`http://www.cs.utk.edu/f2j`.

[4] HARPOON *Homepage:*
`http://www.cs.unc.edu/Research/HARPOON/`.

[5] JAMA *Homepage:*
`http://math.nist.gov/javanumerics/jama/`.

[6] JAMPACK *Homepage:*
`ftp://math.nist.gov/pub/Jampack/Jampack/AboutJampack.html`.

[7] *Java* LAPACK *Homepage:*
`http://www.cs.utk.edu/f2j/download.html`.

[8] *Java Research at Visual Numerics:*
`http://www.vni.com/corner/garage/grande/index.html`.

[9] *JavaNumerics Homepage:*
`http://math.nist.gov/javanumerics/`.

[10] JLAPACK *Homepage:*
`http://www.cs.unc.edu/Research/HARPOON/jlapack/`.

[11] *JNL — A Numerical Library for Java — Homepage:*
`http://www.vni.com/products/wpd/jnl/`.

[12] *National Institute of Standards and Technology Homepage:*
`http://math.nist.gov/`.

[13] NINJA *(Numerically Intensive Java) Homepage:*
`http://www.research.ibm.com/ninja/`.

[14] *OpsResearch, OR-Objects Homepage:*
`http://opsresearch.com/`.

[15] *SPAR/Java Homepage:*
`http://pds.twi.tudelft.nl/timber/spar/index.html`.

[16] *Timber Compiler Homepage:*
`http://pds.twi.tudelft.nl/timber/`.

[17] *Updates to the Java Language Specification for JDK Release 1.2 Floating Point:*
`http://java.sun.com/docs/books/jls/strictfp-changes.pdf`.

[18] D.F. Bacon: KAVA*: A Java dialect with a uniform object model for lightweight classes.*
`http://aspen.ucs.indiana.edu/CandCPandE/jg2001/C559bacon/c559Bacon02Kava.pdf`.

[19] R.F. Boisvert, J.J. Dongarra, R. Pozo, K.A. Remington, , G.W. Stewart: *Developing Numerical Libraries in Java*. ACM 1998 Workshop on Java for High-Performance Network Computing (1998).
`http://www.cs.ucsb.edu/conferences/java98/papers/jnt.pdf`.

[20] R.F. Boisvert, J. Moreira, M. Philippsen, R. Pozo: *Java and Numerical Computing.*
`www.javagrande.org/leapforward/cacm-ron.pdf` .

[21] J.D. Darcy: *Evolving Java's Floating Point Support: The Good, the Bad, and the Ugly.*
`http://www.sonic.net/~jddarcy/Research/cascon.pdf`.

[22] J.D. Darcy, W. Kahan: *Analysis of the Proposal for Extension to Java Floating Point Semantics, Revision 1.*
`http://www.sonic.net/~jddarcy/Research/jgrande.pdf`.

[23] S. Itou, S. Matsuoka, H. Hasegawa: AJAPACK*: Experiments in Performance Portable Parallel Java Numerical Libraries.* Java Grande (2000).
`http://www.ipsj.or.jp/members/SIGNotes/Eng/12/1999/080/article015.html` .

[24] B. Joy: *The Design of the Java Language: Towards a Science of Reliable Programming:*
`www.cs.ucsb.edu/conferences/java99/slides/81-joy.ppt`.

[25] W. Kahan, J.D. Darcy: *How Java's Floating-Point Hurts Everyone Everywhere.*
`http://www.cs.berkeley.edu/~wkahan/JAVAhurt.pdf`.

[26] S.P. Midkiff, J.E. Moreira, M. Snir: *Optimizing Array Reference Checking in Java Programs.* IBM System Journal 37 (1998)(3).

[27] J. Moreira, S. Midkiff: *A Comparison of Java, C/C++, and Fortran for Numerical Computing*. IEEE Antennas and Propagation Magazine 40 (1998)(5). pp 102-105.

[28] J. E. Moreira, S. P. Midkiff, M. Gupta: *From Flop to Megaflops: Java for Technical Computing* (1998).
`http://www.research.ibm.com/journal/sj/391/moreira.html`.

[29] J. E. Moreira, S. P. Midkiff, M. Gupta, P. V. Artigas, M. Snir, R. D. Lawrence: *Java Programming for High-performance Numerical Computing*. IBM System Journal 39 (2000)(1).
`http://www.research.ibm.com/journal/sj/391/moreira.html`.

[30] M. Snir, J. Moreira, M. Gupta, L. Haibt, S. Midkiff: *Java for High-Performance Computing*.
`www.npac.syr.edu/javagrande/ibmgrande.pdf`.

[31] T. L. Veldhuizen: *Just When You Thought Your Little Language Was Safe: "Expression Templates" in Java* (2000).
`http://www.cs.indiana.edu/l/www/hyplan/tveldhui/papers/2000/gcse00.ps`.

[32] P. Wu et al.: *Efficient Support for Complex Numbers in Java*.
`http://polaris.cs.uiuc.edu/~pengwu/publication.htm`.