Design and Evaluation of a Linear Algebra Package for Java

G. Almasi
galmasi@cs.uiuc.edu
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801

F. G. Gustavson
gustav@watson.ibm.com

J. E. Moreira
jmoreira@us.ibm.com

IBM T. J. Watson Research Center
P. O. Box 218
Yorktown Heights, NY 10598

ABSTRACT

This paper describes the design of a high-performance linear algebra library for Java. Linear algebra libraries such as ESSL and LAPACK are important tools of computational science and engineering, and have been available to C and Fortran programmers for quite a while. If Java is to become a serious language for the development of large scale numerical applications, it must provide equivalent functionality. From the many possible alternatives to accomplish this goal, we took the approach of designing a linear algebra library entirely in Java. This approach leads to good portability and maintainability of the code. It is also a good test of how far we can push Java performance. We adopted an object-oriented design in which the linear algebra operations are implemented as strategy design patterns. The higher level algorithms, optimized for the memory hierarchies of present-day machines, are described in a type independent manner. Type specific methods capture the lower level optimizations for operations on matrices of single-precision, double-precision, or complex numbers. We evaluate the performance of our linear algebra package on three different machines. Our preliminary results show that our Java library achieves up to 85% of the performance of the highly optimized ESSL.

1. INTRODUCTION

Scientists and engineers developing numerical applications in established languages such as Fortran and C have a vast collection of standard libraries available in their toolbox. Of particular importance are libraries for numerical linear algebra, such as LAPACK [1] and ESSL [17], libraries for discrete Fourier transform (FFT), and elementary functions libraries. If Java is to become a serious language for large scale numerical computing, it must provide a similar set of tools. Many efforts are under way to provide Java with such libraries [4, 5, 6, 7].

This paper describes and evaluates our particular design of a linear algebra package for Java. The goal of this package is to provide a portable and high performance Java analog to BLAS [11] and LAPACK. We chose to develop this package entirely in Java, with no native code components. We follow an object oriented design that leads to easy maintainability and portability of the code. Through an experimental evaluation, we show that it also leads to good performance. In fact, the package displays *performance portability*. That is, the same `class` files achieve a significant fraction of the machine peak performance on different architectures. In particular, for this paper we demonstrate good performance on two different memory architectures. We also accomplish reasonable performance across different instruction set architectures.

Our linear algebra package is an integral part of the Array package for Java, which we have developed to support true multidimensional arrays in Java [2, 19]. The linear algebra operations are static methods of the `Blas` class in that package. We refer to this library of linear algebra methods as our Java BLAS (even though it also contains LAPACK-style factorization methods). The methods mimic the functionality and interface of the well-known BLAS and LAPACK libraries for Fortran and C. Operations are only permitted on matrices and vectors of the same type, with a different method performing the operation for each type. At this point, we support three different elemental data types: single-precision floating-point numbers (`float`), double-precision floating-point numbers (`double`), and double-precision complex numbers (`Complex`). Although BLAS and LAPACK also support single-precision complex numbers, we chose not to include single-precision complex numbers at this time since they are not part of the standardization effort being conducted by the Java Grande Forum [18]. Support for single-precision complex numbers can be easily added to our package if necessary.

For the purpose of numerical linear algebra, vectors are implemented as one-dimensional arrays of one of the data types (`float`, `double`, and `Complex`), while matrices are implemented as two-dimensional arrays of those data types. A small set of basic operations are implemented by type-specific *kernels*. A kernel method performs a basic linear algebra operation optimized for data that fits well into a L1 data cache. More complicated operations, and memory-hierarchy conscious versions of the basic linear algebra operations, are implemented as type-independent *strategies* [13]. A strategy is a design pattern that specifies a generic algorithm as a series of operations (virtual method calls) to be performed, leaving the specific implementation of each operation to the particular concrete classes. Using this design methodology, we arrive at a linear algebra package that is significantly more concise than its Fortran or C counterparts. The conciseness facilitates maintainability and lets us focus on fewer performance-critical components.

Java 2000 San Francisco CA USA
Copyright ACM 2000 1-58113-288-3/00/6...$5.00

The rest of this paper is organized as follows. We start with a discussion of related work in Section 2. Section 3 gives a brief description of the Array package for Java, focusing on features and properties that are more relevant to numerical linear algebra. Section 4 describes the BlasVector and BlasMatrix Java interfaces, which allow the development of an object-oriented linear algebra package. Section 5 details the implementation of some of the linear algebra routines provided by the package. Section 6 shows some preliminary performance results that demonstrate that our approach can lead to high performance implementations. Finally, Section 7 presents our conclusions.

2. RELATED WORK

There are two basic approaches to providing Java with a linear algebra library. One can leverage existing high-performance linear algebra libraries by having Java programs perform native calls [4, 7]; the obvious advantage is that a relatively small amount of effort is needed. The disadvantage is that there are no guarantees of reproducibility of results, because of differences in each implementation. Also, improvements by one manufacturer do not benefit the users of other products.

The other possibility is to implement the linear algebra package entirely in Java [4, 5, 6, 7]. The obvious disadvantage is the amount of effort required, although the Java code could be generated by automatic translation from Fortran implementations [7, 12]. The advantage of a 100% Java approach lies in the portability of the package and reproducibility of results computed with it ("write once, run anywhere"). Also, all users benefit from enhancements to the package. In [4] it is shown that a hybrid approach, in which the higher levels of a linear algebra package are coded in Java and the lower levels in Fortran, can deliver very good performance.

Java was not originally designed for high-performance computation [14]. Some of the many design issues are handling of numerics-related exceptions, unnecessarily restrictive arithmetics, no direct support for complex numbers, and lack of standardized multidimensional arrays. These issues have been taken up by the members of the Java Grande Forum [18], but they are far from resolved. For example, in our approach we use the fused multiply-add capability of the POWER3 and PowerPC 604e processors to our advantage. We also tackle the array problem by targeting the Array package, which is part of the Java Grande Forum standardization effort.

Our library makes extensive use of recursive linear algebra algorithms, described in [10, 15]. Recursive algorithms are conceptually simple and powerful, and the resulting automatic blocking is an important feature to achieve performance portability. They also fit nicely in a strategy design pattern.

To compile the Array package (including the Java BLAS) we made use of the latest Java compiler technologies developed in our group [2, 3, 19]. We have already demonstrated the benefits of automatic compiler optimization and parallelization that result from programming with the Array package. The Java BLAS is one more step in providing a "Java solution" to scientific and engineering programmers.

3. THE ARRAY PACKAGE FOR JAVA

The Array package for Java implements true multidimensional arrays, which are not directly supported by the Java language. (Instead, Java supports arrays of arrays, which are less amenable to optimizations as shown in [2, 19].) A multidimensional array is characterized by three immutable properties: *type*, *rank*, and *shape*. The type of an array is the type of its elements. The rank (or *dimensionality*) of an array is the number of its axes. Finally, the shape of an array is the extent of each of its axes. The shape of a d-dimensional array A of type T can be represented by a d-element vector $n = (n_0, n_1, \ldots, n_{d-1})$, where n_k is the extent of axis k, $k = 0, \ldots, d-1$. Given an index $i = (i_0, i_1, \ldots, i_{d-1})$, with $0 \leq i_k < n_k \; \forall k$, then $A[i_0, i_1, \ldots, i_{d-1}]$ represents a unique element of the array A. Two elements $A[i_0, i_1, \ldots, i_{d-1}]$ and $A[j_0, j_1, \ldots, j_{d-1}]$ are the same if and only if $i_k = j_k, \forall k$.

Figure 1 is an architectural overview of the Array package for Java. It shows, among other things, the inheritance graph of the classes in the Array package. The virtual base class Array is specialized by type, and then specialized again by rank to obtain the concrete classes. Concrete class names have the form

<*type*>Array<*rank*>D,

where *type* denotes the elemental type (either float, double, or Complex) and *rank* denotes the rank of the array. The shape of the array is defined at object instantiation time, and is immutable once created. The immutability and rectangularity of the shape, together with the specific type and rank of the concrete classes, are important for optimization of code using the Array package. The type- and rank-specific concrete classes facilitate devirtualization and semantic expansion [21], while the rectangular and immutable shape helps bounds checking optimization and alias disambiguation [2, 3, 19]. We also note, from Figure 1, that one-dimensional (vector) concrete classes implement the BlasVector interface, while two-dimensional (matrix) classes implement the BlasMatrix interface. These interfaces are discussed in Section 4.

Also shown in Figure 1 are some methods of the Blas class. This class implements basic linear algebra operations (such as defined by BLAS), as well as some factorization operations (such as defined by LAPACK). The public methods of the Blas (exemplified in Figure 1 by sgemm, dgemm, and zgemm) mimic the interfaces specified by BLAS and LAPACK. The class also includes some internal methods that are used in the implementation of the linear algebra operations. These methods (declared as protected in Figure 1) are not exposed to the application programmer.

3.1 Internal representation

Standard BLAS and LAPACK routines operate on data arranged in memory in column-major format, the standard layout for Fortran arrays. By contrast, the routines in Java BLAS (and Array package code in general) are independent of data layout. Thus, the internal representation (memory layout) of arrays in the Array package is purposely not exposed to the programmer, opening the possibility of future enhancements and compiler optimizations. However, in this section we explain the internals of the current reference implementation of the Array package, since they impact the efficiency of our Java BLAS.

The representation of a d-dimensional array of floats, doubles, or Complexes consists of:

- a data storage pointer (data),
- a shape descriptor $(n_0, n_1, \ldots, n_{d-1})$, and
- a mapping descriptor $(w_0, w_1, \ldots, w_d)$.

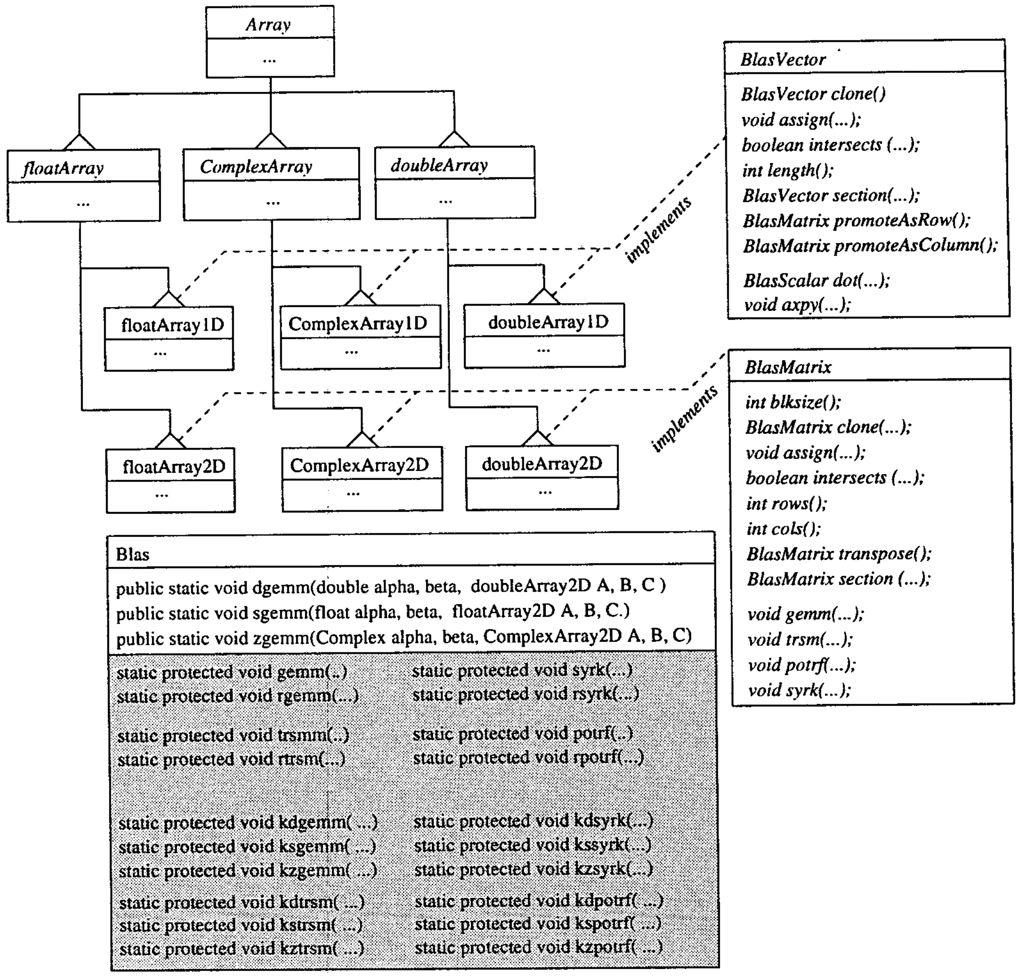
Array
...
floatArray
...
ComplexArray
...
doubleArray
...
floatArray1D
...
ComplexArray1D
...
doubleArray1D
...
floatArray2D
...
ComplexArray2D
...
doubleArray2D
...
implements
implements
BlasVector
BlasVector clone()
void assign(...);
boolean intersects (...);
int length();
BlasVector section(...);
BlasMatrix promoteAsRow();
BlasMatrix promoteAsColumn();
BlasScalar dot(...);
void axpy(...);
BlasMatrix
int blksize();
BlasMatrix clone(...);
void assign(...);
boolean intersects (...);
int rows();
int cols();
BlasMatrix transpose();
BlasMatrix section (...);
void gemm(...);
void trsm(...);
void potrf(...);
void syrk(...);
Blas
public static void dgemm(double alpha, beta, doubleArray2D A, B, C)
public static void sgemm(float alpha, beta, floatArray2D A, B, C.)
public static void zgemm(Complex alpha, beta, ComplexArray2D A, B, C)
static protected void gemm(..)
static protected void rgemm(...)
static protected void syrk(...)
static protected void rsyrk(...)
static protected void trsmm(..)
static protected void rtrsm(...)
static protected void potrf(..)
static protected void rpotrf(...)
static protected void kdgemm(...)
static protected void ksgemm(...)
static protected void kzgemm(...)
static protected void kdsyrk(...)
static protected void kssyrk(...)
static protected void kzsyrk(...)
static protected void kdtrsm(...)
static protected void kstrsm(...)
static protected void kztrsm(...)
static protected void kdpotrf(...)
static protected void kspotrf(...)
static protected void kzpotrf(...)

Figure 1: Array package for Java: architectural overview.

For a d-dimensional array, n_k is the shape of its k-th axis, and w_k is the *weight* of its k-th axis. Element references are translated to locations in the data storage space as follows:

$$\mathtt{A}[i_0, i_1, \ldots, i_{d-1}] \Rightarrow \mathtt{A.data}\left[\begin{array}{c} i_0 \times w_0 + i_1 \times w_1 + \\ \ldots + i_{d-1} \times w_{d-1} + w_d \end{array}\right]$$

We note that w_d, for a d-dimensional array, represents the displacement of the first element of the array. The data storage pointer (`data`) is of type `float[]` for single-precision arrays and of type `double[]` for double-precision and complex arrays. A complex number is represented as two consecutive `doubles`.

3.2 Manipulators

Array manipulators are operations that extract regular subsets of arrays according to certain rules. Because these operations are used intensively by the canonicalization and recursive descent phases of linear algebra routines (see Section 5.1), they are all implemented as $O(1)$ operations that involve no data copying. That is, an array manipulator operation on an array A returns a new array A' that has the same data storage pointer as A (A.**data** $= A'$.**data**), but has (possibly) different shape and mapping descriptors.

Table 1 shows the definition and implementation of some of the manipulators for one- and two-dimensional arrays (*i.e.*, vectors and matrices). The operations are expressed in Fortran 90-like notation. Implementation is shown in terms of new n' and w' values for the descriptors of the resulting array A', as a function of n and w for array A. Sectioning manipulators are used extensively in our Java BLAS implementation to decompose linear algebra operations into smaller sub-operations. Vector promotion to matrix is also very useful, as it allows matrix-vector (BLAS-2) operations to be converted to matrix-matrix (BLAS-3) operations.

Particularly important for the implementation of the linear algebra operations in Section 5 are the *transposition* and *axis reversal* manipulators. The transpose of an $n \times m$ matrix A is an $m \times n$ matrix A^T such that $A^T[i_0, i_1] = A[i_1, i_0]$. Axis reversal is an operation that can be applied to each axis of a matrix independently. For an $n \times m$ matrix A, A^0, A^1, and $A^{0,1}$ are all $n \times m$ matrices with the properties $A^0[i_0, i_1] = A[n - i_0 - 1, i_1]$, $A^1[i_0, i_1] = A[i_0, m - i_1 - 1]$, and $A^{0,1}[i_0, i_1] = A[n - i_0 - 1, m - i_1 - 1]$. Axis reversal is represented in the Array package as a special case of sectioning. As we will shown in Section 5, transposition and axis

Table 1: Vector and matrix manipulators: definition and implementation.

definition	$n'_0 =$	$n'_1 =$	$w'_0 =$	$w'_1 =$	$w'_2 =$
vector section $v' = v[f_0 : l_0 : s_0]$	$\left\lfloor \frac{l_0-f_0}{s_0} \right\rfloor + 1$		$w_0 \times s_0$	$w_0 \times f_0 + w_1$	
vector row promotion $A' = spread(v, 2, 1)$	n_0	1	w_0	1	w_1
vector column promotion $A' = spread(v, 1, 1)$	1	n_0	1	w_0	w_1
vector reversal $v^0 = v[n_0 - 1 : 0 : -1]$	n_0		$-w_0$	$w_0 \times (n_0 - 1)$ $+w_1$	
matrix section $A' = A[f_0 : l_0 : s_0, f_1 : l_1 : s_1]$	$\left\lfloor \frac{l_0-f_0}{s_0} \right\rfloor + 1$	$\left\lfloor \frac{l_1-f_1}{s_1} \right\rfloor + 1$	$w_0 \times s_0$	$w_1 \times s_1$	$w_0 \times f_0 +$ $w_1 \times f_1 + w_2$
matrix projection (col) $v' = A[i_0, f_1 : l_1 : s_1]$	$\left\lfloor \frac{l_1-f_1}{s_1} \right\rfloor + 1$		$w_1 \times s_1$	$w_0 \times i_0 +$ $w_1 \times f_1 + w_2$	
matrix projection (row) $v' = A[f_0 : l_0 : s_0, i_1]$	$\left\lfloor \frac{l_0-f_0}{s_0} \right\rfloor + 1$		$w_0 \times s_0$	$w_0 \times f_0 +$ $w_1 \times i_1 + w_2$	
matrix transpose $A^T = transp(A)$	n_1	n_0	w_1	w_0	w_2
matrix column (up/down) reversal $A^0 = A[n_0 - 1 : 0 : -1, :]$	n_0	n_1	$-w_0$	w_1	$w_0 \times (n_0 - 1)$ $+w_2$
matrix row (left/right) reversal $A^1 = A[:, n_1 - 1 : 0 : -1]$	n_0	n_1	w_0	$-w_1$	$w_1 \times (n_1 - 1)$ $+w_2$
matrix row and column reversal $A^{0,1} = A[n_0 - 1 : 0 : -1,$ $n_1 - 1 : 0 : -1]$	n_0	n_1	$-w_0$	$-w_1$	$w_0 \times (n_0 - 1)$ $+$ $w_1 \times (n_1 - 1)$ $+w_2$

reversal simplify the development of numerical linear algebra operations, by reducing the various forms of the operation to a single canonical form on which the solution can be focused.

Efficient processing of matrices in our Java BLAS relies on two key features of modern processors. First, L1 data caches must behave as true uniform random access memories. That is, a load from any cache position can be accomplished in the same amount of time. Second, the processor instruction set must efficiently support array traversals of arbitrary stride. Both features are present in the IBM POWER3 and PowerPC family of processors. These two processor features, coupled with the memory-hierarchy conscious blocking of our Java BLAS, combine to deliver good performance in a simple design.

4. THE LINEAR ALGEBRA INTERFACES

The BlasVector and BlasMatrix interfaces (see Figure 1) define the methods that a class has to implement in order to function as either a vector or a matrix, respectively, in the numerical linear algebra sense. These interfaces are what make possible the use of a strategy design pattern for the linear algebra operations. High-level algorithms are expressed in terms of the BlasMatrix and BlasVector methods. The actual implementation of those methods is deferred to the concrete classes.

The BlasVector interface defines some manipulators shown in Table 1: a sectioning operation (method section), and two promotion operations to convert an n-element vector to an $1 \times n$ or $n \times 1$ matrix (methods promoteAsRow and promoteAsColumn, respectively) . The BlasVector interface also defines methods that implement vector-vector (BLAS-1) operations, like dot-product $x^T y$ (method dot) and addition of vectors with scaling $y = \alpha x + y$ (method axpy).

The BlasMatrix interface also defines manipulators, like sectioning (section) and transposition (transpose). It also defines method blksize that returns the ideal block size for a particular matrix. This ideal block size should lead to in-cache computations. The value of the block size is used by the numerical linear algebra algorithms to efficiently partition the computation. The ideal block size typically depends on the type of the data (blocks for floats, for example, are larger than for doubles). The concrete array classes implement this method to return the appropriate value. The value can be further refined to be architecture dependent, in a particular implementation of the Array package. (Java provides an API – through the System class [8] – by which an executing program can find out on which type of architecture it is running.)

BlasMatrix also defines kernel-level methods for BLAS-3 and factorization operations. In Figure 1 we show three BLAS-3 methods (gemm, trsm, and syrk), and one factorization (potrf) method. These kernel-level methods do not have to worry about memory hierarchy optimizations (they are intended to operate on matrices no larger than the ideal block size) and need only to implement one version of the operation, instead of the multiple versions typically defined by BLAS and LAPACK. This issue is detailed in the next section.

5. IMPLEMENTATION OF BLAS-3 AND FACTORIZATION OPERATIONS

The BLAS-3 [11] operations are at the core of high-performance linear algebra algorithms. Because of their importance, in this section we discuss the implementation of a few BLAS-3 operations. We also discuss one factorization algorithm. We use recursive formulations to implement all our BLAS-3 and factorization operations. The recursive formulation has two very useful properties. First, it leads to automatic blocking of data [15]. Second, a large fraction of the computation ends up expressed in terms of matrix multiplication (GEMM) operations.

Automatic blocking is important as we move to machines with deeper and deeper memory hierarchies. With a recursive formulation, different levels of the recursion have different matrix sizes that are appropriate for the various levels of the memory hierarchy. Recursion also helps by making the choice of the block size, as defined by method `blksize`, less critical. The automatic blocking from recursion leads to good performance even when the cache is much larger than the particular block size.

Expressing linear algebra operations in terms of matrix multiplications is important because the GEMM operation can be performed very efficiently on a variety of architectures. Its n^3/n^2 computation to data ratio makes it possible to tolerate the large computation/memory performance gap in present day machines. It is common practice to maximize the use of GEMM operations when developing linear algebra algorithms [11].

5.1 Operation stages

The example call graph of Figure 2, for the TRSM (triangular system solve) operation, details the stages in the execution of a linear algebra operation. The first step consists of an invocation of a type-specific (in this case, for `doubles`) public method which mimics the standard BLAS interface. This method (`Blas.dtrsm`) simply calls the type-generic method (`Blas.trsm`). The first stage in each operation is a *sanity check*. The operation will not proceed until the data are shown to be consistent and there is a guarantee that the operation can terminate with no exceptions. Also, before proceeding with the operation, its many different forms (as defined by BLAS or LAPACK) are all transformed into a single *canonical form*. This canonical form is then used by the *recursive descent* phase, which decomposes one operation into smaller sub-operations. Canonicalizing the operation allows us to write just one version of the recursive descent phase. When recursive descent bottoms out by reaching the ideal block size defined by method `blksize` (*i.e.*, when the problem size becomes small enough to fit into L1 cache), an *optimized kernel* is invoked through a virtual function call. Thus, the extra overhead of recursion is kept at a minimum.

The sanity check, canonicalization, and recursive descent phases are written in a type-generic fashion. The optimized kernels are type-aware. (There exists one properly optimized kernel for each data type.) The appropriate kernel is found by means of a virtual function call (as detailed by the call graph in Figure 2). Type-specific optimized kernels exist for all operations for double-precision, single-precision, and complex data.

In the next sections we present implementation details of four linear algebra operations. These serve as illustrations of the general approach adopted in implementing Java BLAS. We discuss three operations that are part of BLAS-3: GEMM (generic matrix multiply), TRSM (triangular system solve), and SYRK (symmetric rank-k update). We also discuss POTRF (Cholesky factorization).

5.2 GEMM

GEMM implements the operation $C = \beta C + \alpha A^* \times B^*$, where X^* denotes either the matrix X or its transpose X^T. Consequently, there are four different forms of GEMM and typical BLAS code in Fortran or C will explicitly code all four forms [1]. In the Java BLAS, all four forms can be easily reduced to the canonical form $C = \beta C + \alpha A \times B$ with the use of the transposition manipulator. The canonical form of GEMM is implemented recursively. The operation can be decomposed in three different ways at each recursive step:

$$\begin{gathered}\begin{bmatrix} C_1 \\ C_2 \end{bmatrix} = \beta \begin{bmatrix} C_1 \\ C_2 \end{bmatrix} + \alpha \begin{bmatrix} A_1 \\ A_2 \end{bmatrix} \times B \\ \Downarrow \\ C_1 = \beta C_1 + \alpha A_1 \times B, C_2 = \beta C_2 + \alpha A_2 \times B\end{gathered} \tag{1}$$

$$\begin{gathered}\begin{bmatrix} C_1 & C_2 \end{bmatrix} = \beta \begin{bmatrix} C_1 & C_2 \end{bmatrix} + \alpha A \times \begin{bmatrix} B_1 & B_2 \end{bmatrix} \\ \Downarrow \\ C_1 = \beta C_1 + \alpha A \times B_1, C_2 = \beta C_2 + \alpha A \times B_2\end{gathered} \tag{2}$$

$$\begin{gathered}C = \beta C + \alpha \begin{bmatrix} A_1 & A_2 \end{bmatrix} \times \begin{bmatrix} B_1 \\ B_2 \end{bmatrix} \\ \Downarrow \\ C = \beta C + \alpha A_1 \times B_1 + \alpha A_2 \times B_2\end{gathered} \tag{3}$$

The particular decomposition is chosen at each recursive step so that the larger axis at that step is split. This approach tends to create more "square" subproblems, which lead to a better computation to data ratio, and therefore to better performance. The recursion continues until the problem is no larger than the block size defined for each data type. At this point, the type-specific kernel is invoked.

5.3 TRSM

TRSM solves a triangular system of equations with multiple right-hand sides. The canonical form of the system is

$$LX = \alpha B \tag{4}$$

where L denotes an $n \times n$ lower triangular matrix, X (the unknown) and B are $n \times m$ rectangular matrices, and α is a scalar. In addition to the canonical form, the triangular system can take seven other forms as shown in Figure 3, where U denotes an $n \times n$ upper triangular matrix. The figure also shows the sequence of operations needed to bring the equations to the canonical form. We note that the four forms with the coefficient (L or U) matrix on the right can be transformed to forms with the coefficient matrix on the left with the application of transposition. The forms with U^T or L^T are directly equivalent to forms with L and U, respectively. Finally, the form $UX = \alpha B$ can be transformed to form $LX = \alpha B$ with the use of the axis reversal manipulators discussed in Section 3.2. $U^{0,1}$ is a lower triangular matrix, and equation $UX = \alpha B$ is equivalent to the equation $U^{0,1} X^0 = B^0$.

The canonical equation in the form $LX = \alpha B$ is solved using a recursive algorithm. The particular decomposition adopted in each recursive step depends on the relative values of n and m. If $m > n$ then we can write

$$L \begin{bmatrix} X_1 & X_2 \end{bmatrix} = \alpha \begin{bmatrix} B_1 & B_2 \end{bmatrix} \tag{5}$$

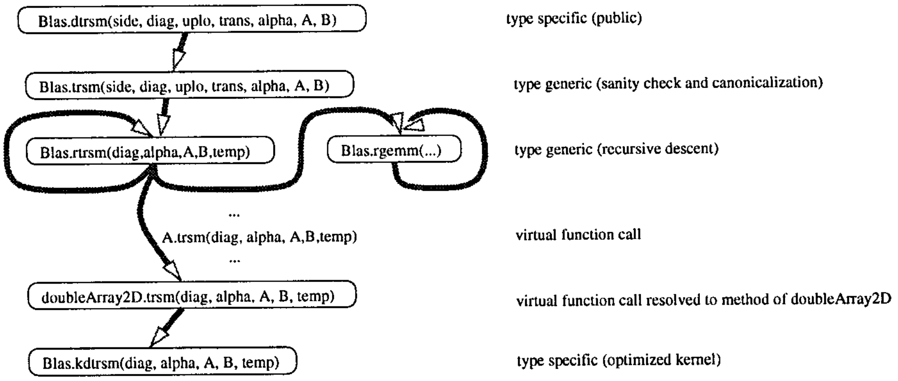
Blas.dtrsm(side, diag, uplo, trans, alpha, A, B)
type specific (public)
Blas.trsm(side, diag, uplo, trans, alpha, A, B)
type generic (sanity check and canonicalization)
Blas.rtrsm(diag,alpha,A,B,temp)
Blas.rgemm(...)
type generic (recursive descent)
...
A.trsm(diag, alpha, A,B,temp)
...
virtual function call
doubleArray2D.trsm(diag, alpha, A, B, temp)
virtual function call resolved to method of doubleArray2D
Blas.kdtrsm(diag, alpha, A, B, temp)
type specific (optimized kernel)

Figure 2: Call graph for TRSM, showing the various stages of execution of a Java BLAS operation.

$$\begin{tikzcd}
XL^T = \alpha B \arrow[r, Leftrightarrow, "T"] & \boxed{LX = \alpha B} \arrow[r, Leftrightarrow, "R"] \arrow[d, Leftrightarrow, "E"] & UX = \alpha B \arrow[r, Leftrightarrow, "T"] \arrow[d, Leftrightarrow, "E"] & XU^T = \alpha B \\
XU = \alpha B \arrow[r, Leftrightarrow, "T"] & U^T X = \alpha B & L^T X = \alpha B \arrow[r, Leftrightarrow, "T"] & XL = \alpha B
\end{tikzcd}$$

transformation	effect
axis reversal	$UX = \alpha B \overset{R}{\Leftrightarrow} U^{0,1} X^0 = \alpha B^0$
transposition	$XL^T = \alpha B \overset{T}{\Leftrightarrow} LX^T = \alpha B^T, XU = \alpha B \overset{T}{\Leftrightarrow} U^T X^T = \alpha B^T, \ldots$
equivalence	$LX = \alpha B \overset{E}{\Leftrightarrow} U^T X = \alpha B, UX = \alpha B \overset{E}{\Leftrightarrow} L^T X = \alpha B$

Figure 3: The eight forms of TRSM, which can all be reduced to $LX = \alpha B$ through manipulators.

and we decompose it into two smaller TRSM operations $LX_1 = \alpha B_1$ and $LX_2 = \alpha B_2$. If $n \geq m$ we can write $LX = \alpha B$ as

$$\begin{bmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{bmatrix} \begin{bmatrix} X_1 \\ X_2 \end{bmatrix} = \begin{bmatrix} \alpha B_1 \\ \alpha B_2 \end{bmatrix} \tag{6}$$

where L_{11} is an $n_1 \times n_1$ lower triangular matrix, L_{21} is an $n_2 \times n_1$ rectangular matrix, and L_{22} is an $n_2 \times n_2$ lower triangular matrix. We note that $n_1 + n_2 = n$, both X_1 and B_1 are $n_1 \times m$ rectangular matrices and both X_2 and B_2 are $n_2 \times m$ rectangular matrices. From this formulation we derive

$$L_{11} X_1 = \alpha B_1 \tag{7}$$
$$L_{22} X_2 = \alpha B_2 - L_{21} X_1 \tag{8}$$

Equation 7 is just a smaller triangular system of equations, and can be solved with a recursive call to TRSM. To evaluate X_2 in Equation 8, we first need to compute the new right-hand side $\alpha B_2 - L_{21} X_1$. This can be accomplished with a GEMM operation. After that, we just need to solve another triangular system of equations with TRSM.

5.4 SYRK

The SYRK operation performs an update of a triangular matrix. It can take one of four forms:

$$\begin{array}{ll} L = \beta L + \alpha A \times A^T & U = \beta U + \alpha A \times A^T \\ L = \beta L + \alpha A^T \times A & U = \beta U + \alpha A^T \times A \end{array} \tag{9}$$

The forms with $A^T \times A$ can be reverted to the forms with $A \times A^T$ by a transposition of A. Also the forms with upper triangular matrix U can be converted to forms with lower triangular matrix L by a transposition. Therefore, we only have to consider the canonical form $L = \beta L + \alpha A \times A^T$.

For the recursive descent phase, the canonical form can be decomposed in two different ways, depending on which axis of matrix A is larger. Let L be $n \times n$ and let A be $n \times m$. If $n \geq m$ we can decompose the canonical form as

$$\begin{bmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{bmatrix} = \beta \begin{bmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{bmatrix} + \alpha \begin{bmatrix} A_1 \\ A_2 \end{bmatrix} \times \begin{bmatrix} A_1^T & A_2^T \end{bmatrix} \tag{10}$$

Which leads to the relations:

$$L_{11} = \beta L_{11} + \alpha A_1 \times A_1^T \tag{11}$$
$$L_{21} = \beta L_{21} + \alpha A_2 \times A_1^T \tag{12}$$
$$L_{22} = \beta L_{22} + \alpha A_2 \times A_2^T \tag{13}$$

Equations 11 and 13 are just smaller SYRK operations, while Equation 12 is a GEMM operation. If $m > n$ then we decompose as:

$$\begin{aligned} L &= \beta L + \alpha \begin{bmatrix} A_1 & A_2 \end{bmatrix} \times \begin{bmatrix} A_1^T \\ A_2^T \end{bmatrix} \\ &= \beta L + \alpha A_1 \times A_1^T + \alpha A_2 \times A_2^T \end{aligned} \tag{14}$$

which can be implemented as two recursive calls to SYRK.

5.5 POTRF

POTRF computes the Cholesky factor L of an $n \times n$ symmetric positive definite matrix A. That is, it computes the lower triangular matrix L such that $L^T \times L = A$. Alternatively, POTRF can compute the upper triangular factor U such that $U^T \times U = A$. The latter problem can be reverted back to the former with a transposition. We decompose $L \times L^T = A$ as

$$\begin{bmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{bmatrix} \times \begin{bmatrix} L_{11}^T & L_{21}^T \\ 0 & L_{22}^T \end{bmatrix} = \begin{bmatrix} A_{11} & A_{21}^T \\ A_{21} & A_{22} \end{bmatrix} \quad (15)$$

from which we write

$$\begin{aligned} L_{11} \times L_{11}^T &= A_{11} && (16) \\ L_{21} \times L_{11}^T &= A_{21} && (17) \\ L_{22} \times L_{22}^T &= A_{22} - L_{21} \times L_{21}^T && (18) \end{aligned}$$

L_{11} can be computed with Equation 16 through a recursive call to POTRF. L_{21} can be computed with Equation 17 through a call to TRSM. Finally, to compute L_{22} we first have to compute $A_{22} - L_{21} \times L_{21}^T$ with a call to SYRK, and then we can evaluate L_{22} with another recursive call to POTRF.

6. EXPERIMENTAL RESULTS

We evaluate the performance of our Java BLAS on three machine architectures (POWER3, PowerPC, and IA-32) and in two different execution environments (static and dynamic compilation). We used the following machines: (i) An IBM RS/6000 model F50, with a 332 MHz PowerPC 604e processor, 32 kB of level 1 data cache, 256 kB of level 2 data cache, and a peak floating-point performance of 664 Mflops. (ii) An IBM RS/6000 model 260, with a 200 MHz POWER3 processor, 64 kB of level 1 data cache, 4 MB of level 2 data cache, and peak performance of 800 Mflops. (iii) A Dell Precision 610, with a 550 MHz Xeon processor, 32kB of level 1 data cache, 512kB of level 2 data cache, and peak performance of 550Mflops. The POWER3 is optimized for scientific computing, while the 604e and the Xeon are more general-purpose processors.

We start by comparing the performance of statically compiled Java BLAS against native libraries optimized for each architecture. On the RS/6000 machines, Java BLAS was statically compiled into native code with the Ninja prototype optimizing Java compiler [2, 3, 19]. The Ninja compiler performs bounds checking elimination, loop invariant motion, strength reduction, semantic expansion [21], and various loop transformations. Since Ninja is only available for RS/6000, in the Xeon machine we used IBM HPCJ with the runtime checks disabled. Although this does not correspond to a legal Java execution, it gives us an idea of the performance we could achieve by porting Ninja to IA-32. (The Ninja compiler is based on HPCJ.)

We compared the compiled Java code with the performance of native libraries on the target machines as follows. On the RS/6000 machines the basis for comparison was IBM's Engineering and Scientific Subroutine Library (ESSL). ESSL is mostly implemented in Fortran, and has been fine tuned for IBM machines over the course of many product generations. On the Intel machine the frame of reference was the Math Kernel Library [16], which contains a set of linear algebra routines optimized for Intel CPUs. Equivalent driver programs in Java, Fortran, and C were used to test Java BLAS and the native libraries respectively.

Results for each of the three different instances of GEMM and TRSM (one each for `float`, `double`, and `Complex` types) are shown in Figure 4 for the RS/6000 machines. Each plot shows the performance of one particular routine (SGEMM and STRSM for `floats`, DGEMM and DTRSM for `doubles`, ZGEMM and ZTRSM for `Complexes`) when operating on $n \times n$ square matrices of the same size. Performance in Mflops is shown as a function of n, the problem size. Each plot has four lines: ESSL on POWER3 and 604e, and Java BLAS on POWER3 and 604e.

We first note that the performance of Java BLAS is quite stable with respect to problem size, indicating that our recursive approach is effective in exploiting the memory hierarchy. Our Java BLAS implementation is not as efficient as ESSL, but it achieves good absolute performance. For `float` operations the Java BLAS achieves approximately 85% of the corresponding ESSL on the POWER3. The results for the `double` operations are slightly worse, 80% of the corresponding ESSL routines on the POWER3. The corresponding numbers for the PowerPC 604e are 85% and 70% of ESSL for `floats` and `doubles`, respectively. We have measured the `doubleArray2D` gemm kernel at approximately 95% of ESSL performance when operating on small, in-cache, matrices. The remaining performance degradation results from recursion and virtual function call overhead.

In the case of the `Complex` operations ZGEMM and ZTRSM, our Java BLAS achieves only 65% and 70% of ESSL performance on POWER3 and PowerPC 604e, respectively. Operations on complex numbers have a better computation-to-data ratio than equivalent operations on real numbers. Therefore, ZGEMM and ZTRSM are expected to perform even better than the corresponding DGEMM and DTRSM. This is the case for the ESSL versions, but not so for the Java BLAS. The results from POWER3 in particular indicate that we need more work to improve the `ComplexArray2D` gemm and `trsm` kernels.

Results for DGEMM in the Xeon machine are shown in Figure 5. Again, the performance of Java BLAS is quite stable with respect to problem size, indicating effective exploitation of the memory hierarchy. However, on that platform, the relative performance of Java BLAS is only around 30% of Intel's MKL. (The same ratio holds for the other versions of GEMM and TRSM.) The performance of the `doubleArray2D` gemm kernel when operating on small, in-cache, matrices, yields about 40% of MKL performance. This demonstrates the need to tune the block size parameter, and probably some of the kernel code, for the Xeon processor.

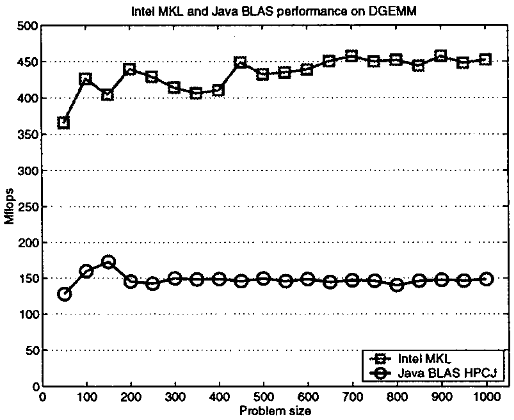

Figure 5: IA-32 (Xeon) performance on DGEMM.

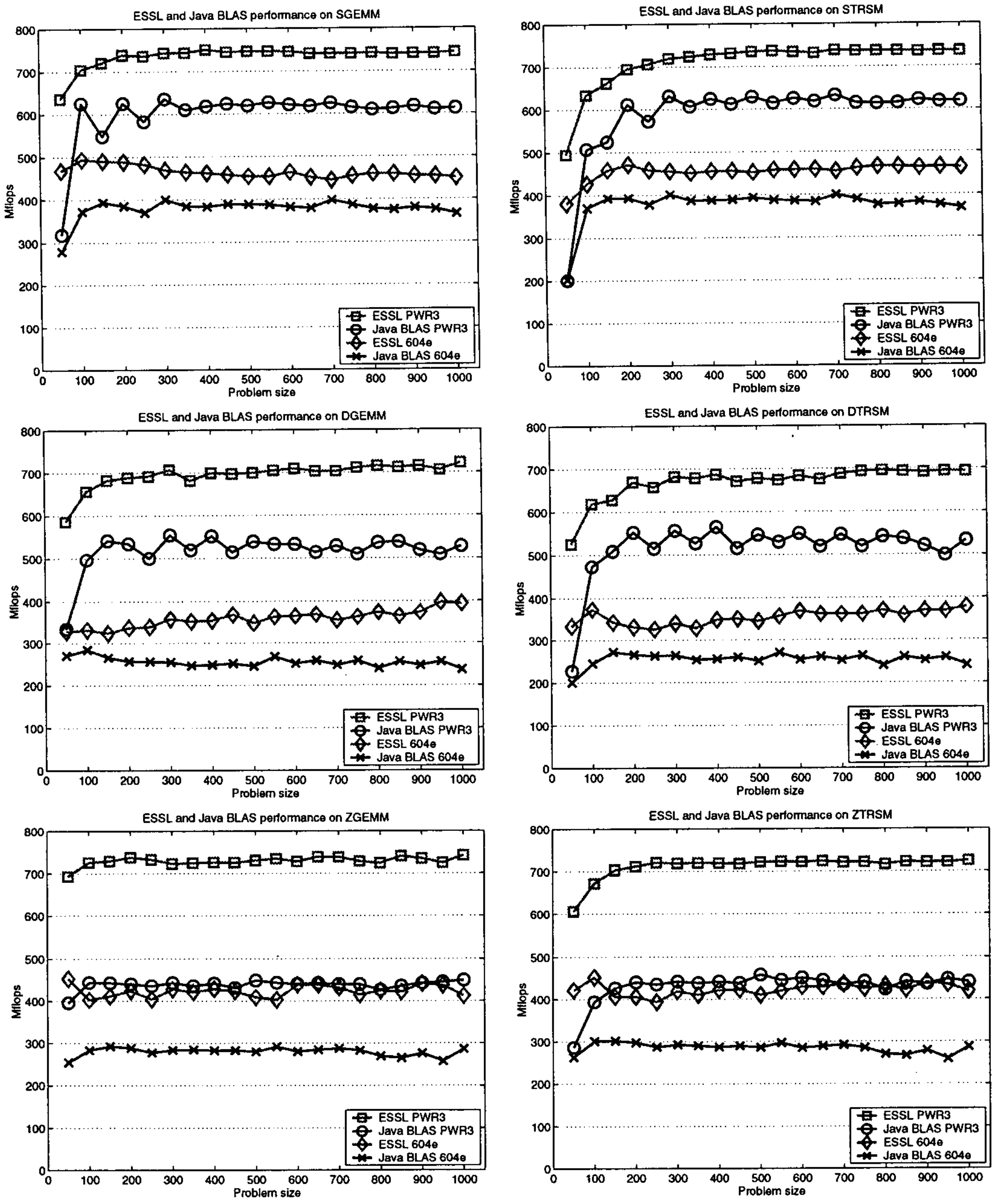

Figure 4: Performance results for ESSL and Java Blas on POWER3 and PowerPC 604e machines.

We note, in Figure 4, that there is not much difference between the plots for GEMM and the corresponding (same data type) plots for TRSM. Although not shown, the same holds true for the Xeon machine. Both operations can be implemented with the same level of efficiency. This is expected (at least in the Java BLAS versions) for two reasons. First, the kernel levels `gemm` and `trsm` methods use identical loop unrolling and register blocking optimizations, with the final results that the kernels look remarkably similar. Second, and most importantly, our recursive approach to implementing TRSM leads to a large fraction of computation in a call to TRSM actually being performed inside GEMM operations. That fraction, as a function of problem size, is plotted in Figure 6 for the three different instances of TRSM. Block sizes for `Complex` are smaller than for `double`, which in turn are smaller than for `float`. (A larger element results in a smaller block size for the a fixed cache size.) That means that, for the same problem size, the `Complex` version will have a deeper recursion tree than the `double` version, which will have a deeper recursion tree than the `float` version. A deeper recursion tree results in a higher fraction of GEMM operations.

The numbers for fraction of computation performed by GEMM for SYRK are identical to those for TRSM, and thus not shown in Figure 6. For completeness, Figure 6 also shows the fraction of computation performed by GEMM in a Cholesky factorization (POTRF). For a given problem size, the fraction in POTRF is not a high as in TRSM or SYRK, since POTRF has to be first decomposed into those operations before they are in turn decomposed into GEMM operations. Nevertheless, for large problem sizes the fraction of computation in GEMM can be very significant, and the asymptotic performance of POTRF is the same as that of GEMM.

Finally, to analize performance on a dynamic compilation environment, we compare the speed of our Java BLAS with the Java Numerical Library from Visual Numerics [20]. We ran our experiments on the Xeon platform using IBM's DK (version 1.1.8), again using equivalent driver programs for the two libraries. Figure 7 compares the respective performances of the two libraries for DGEMM. Visual Numerics' code implements matrices using Java arrays of arrays. Our Array package Java BLAS displays higher and more stable performance.

7. CONCLUSIONS

Optimized linear algebra libraries are important tools in computational science and engineering. Their availability in Java is absolutely necessary if this language is to become a serious candidate for the development of large numerical applications. We have adopted one particular approach to address this issue by developing a linear algebra package entirely in Java.

The benefits of our approach can be summarized as follows. First and foremost, we have achieved *high performance.* Even though it is still in earlier stages of development, our Java linear algebra package has already achieved 65-85% of the performance of one of the most respected industrial-strength numerical libraries (ESSL). This is an excellent starting point, and with it we have demonstrated the performance feasibility of our approach. Now we need to continue to optimize our implementation and broaden the evaluation with other operations, particularly factorization.

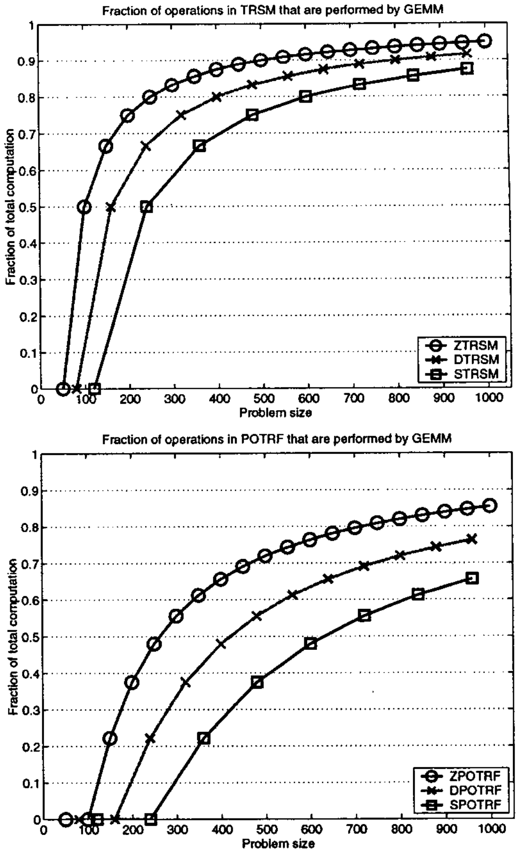

Figure 6: Fraction of computations performed by GEMM.

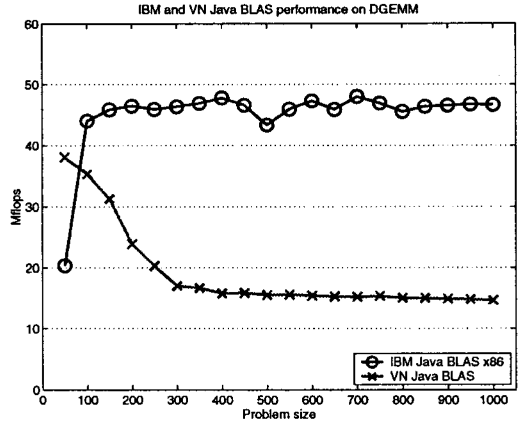

Figure 7: Java BLAS performance with JIT.

Other benefits of our approach are *performance portability*, *code portability*, and *reproducibility of results.* The library is not optimized for any particular architecture/cache layout. We have demonstrated that it yields very good results on two different memory architectures. The recursive formulation of the linear algebra operations leads to good exploitation of memory hierarchy. Code portability is intrinsic to any Java program. Our object oriented design is also more *concise* and *easier to maintain* than Fortran and C implementations of BLAS and LAPACK. Finally, the performance results are also a demonstration of the capabilities of state-of-the-art Java compilers. We have demonstrated reasonable performance with the Array package on another instruction-set architecture, namely IA-32. Those results, however, were not as good as for the RS/6000 machines. At the moment, our optimizing compiler infrastructure is only available on RS/6000.

As we have emphasized earlier, the data layout in the Array package is in no way exposed to programmers, giving us the freedom to explore new data organizations. We have started experimenting with blocked and recursive blocked data layouts [9, 10], which nicely match the recursive decomposition algorithms. We can introduce these new data structures without having to rewrite a single line of Java BLAS. At this point, we only have very preliminary results that show performance gains of approximately 10% over the current data organization in the Array package. We will continue to pursue this line of research, as these techniques potentially lead to Java programs achieving better performance than their Fortran counterparts.

Acknowledgements: The authors wish to thank Sid Chatterjee for fruitful discussions during the development of the linear algebra package for Java.

8. REFERENCES

[1] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK User's Guide*. Society for Industrial and Applied Mathematics, 1995.

[2] P. V. Artigas, M. Gupta, S. P. Midkiff, and J. E. Moreira. High performance numerical computing in Java: Language and compiler issues. In J. Ferrante et al., editors, *12th International Workshop on Languages and Compilers for Parallel Computing*. Springer Verlag, August 1999. IBM Research Division report RC21482.

[3] P. V. Artigas, M. Gupta, S. P. Midkiff, and J. E. Moreira. Automatic loop transformations and parallelization for Java. In *International Conference on Supercomputing*, May 2000. IBM Research Division report RC21629.

[4] B. Blount and S. Chatterjee. An evaluation of Java for numerical computing. In *Proceedings of ISCOPE'98*, volume 1505 of *Lecture Notes in Computer Science*, pages 35–46. Springer Verlag, 1998.

[5] R. F. Boisvert, J. J. Dongarra, R. Pozo, K. A. Remington, and G. W. Stewart. Developing numerical libraries in Java. *Concurrency, Pract. Exp. (UK)*, 10(11-13):1117–29, September-November 1998. ACM 1998 Workshop on Java for High-Performance Network Computing. URL: `http://www.cs.ucsb.edu/conferences/java98`.

[6] R. F. Boisvert, J. Hicklin, B. Miller, C. Moler, R. Pozo, K. Remington, and P. Webb. JAMA: A Java matrix package. URL: `http://math.nist.gov/javanumerics/jama/`, 1998.

[7] H. Casanova, J. Dongarra, and D. M. Doolin. Java access to numerical libraries. *Concurrency, Pract. Exp. (UK)*, 9(11):1279–91, November 1997. Java for Computational Science and Engineering - Simulation and Modeling II Las Vegas, NV, USA 21 June 1997.

[8] P. Chan, R. Lee, and D. Kramer. *The Java Class Libraries*, volume 1 of *The Java Series*. Addison-Wesley, 1998.

[9] S. Chatterjee, V. V. Jain, A. R. Lebeck, S. Mundhra, and M. Thottethodi. Nonlinear array layouts for hierarchical memory systems. In *Proceedings of the 1999 International Conference on Supercomputing*, pages 444–453, Rhodes, Greece, 1999.

[10] S. Chatterjee, A. R. Lebeck, P. K. Patnala, and M. Thottenthodi. Recursive array layouts and fast parallel matrix multiplication. In *Proceedings of the 11th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 222–231, Saint-Malo, France, June 1999.

[11] J. J. Dongarra, I. S. Duff, D. C. Sorensen, and H. A. van der Vorst. *Solving Linear Systems on Vector and Shared Memory Computers*. Society for Industrial and Applied Mathematics, 1991.

[12] G. Fox, X. Li, Z. Qiang, and W. Zhigang. A prototype of Fortran-to-Java converter. *Concurrency, Pract. Exp.*, 9(11):1047–61, Nov 1997. Java for Computational Science and Engineering - Simulation and Modeling II Las Vegas, NV, USA 21 June 1997.

[13] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Publishing Company, 1995.

[14] James Gosling. The evolution of numerical computing in Java. URL: `http://java.sun.com/people/jag/FP.html`, 1997. Sun Microsystems.

[15] F. G. Gustavson. Recursion leads to automatic variable blocking for dense linear algebra algorithms. *IBM Journal of Research and Development*, 41(6):737–755, November 1997.

[16] Intel Corporation. *Intel Math Kernel Library*. URL: `http://www.intel.com/vtune/perflibst/mkl/index.htm`.

[17] International Business Machines Corporation. *IBM Engineering and Scientific Subroutine Library for AIX – Guide and Reference*, December 1997.

[18] Java Grande Forum. *Java Grande Forum Report: Making Java Work for High-End Computing*, November 1998. Java Grande Forum Panel, SC98, Orlando, FL. URL: `http://www.javagrande.org/reports.htm`.

[19] J. E. Moreira, S. P. Midkiff, M. Gupta, P. V. Artigas, M. Snir, and R. D. Lawrence. Java programming for high performance numerical computing. *IBM Systems Journal*, 39(1):21–56, 2000. IBM Research Division report RC21481.

[20] Visual Numerics Inc. *JNL 1.0 - A Numerical Library for Java*. URL: `http://www.vni.com/products/wpd/jnl/`.

[21] P. Wu, S. P. Midkiff, J. E. Moreira, and M. Gupta. Efficient support for complex numbers in Java. In *Proceedings of the 1999 ACM Java Grande Conference*, 1999. IBM Research Division report RC21393.