A Note on Native Level 1 BLAS in Java*

Aart J.C. Bik and Dennis B. Gannon
Lindley Hall 215, Computer Science Department, Indiana University
Bloomington, Indiana 47405-4101, USA
ajcbik@cs.indiana.edu

Abstract
In this research note, we explore the potential of extending the Java Application Programming Interface with some mathematical primitives to improve the performance of certain operations in Java programs while maintaining portability. In particular, we show that providing straightforward native implementations of primitives from Level 1 BLAS can already improve the performance substantially. On multiprocessors, combining this native Level 1 BLAS with the multi-threading mechanism of Java may even provide a simple and portable way to obtain a Java program that runs faster than compiled serial C code.

1 Introduction
The portability of the Java programming language [13] is obtained by compiling Java programs into architectural neutral instructions (bytecode) of the Java Virtual Machine (JVM) [16], rather than into native machine code. Bytecode runs on any platform that supports an implementation of the JVM. Although the interpretation of bytecode is substantially faster than the interpretation of most high level languages, still a performance penalty must be paid for this portability. Clearly, for Java applications that are computational intensive, it would be desirable to reduce this performance penalty without sacrificing the portability of the language. One approach, for example, is to optimize the Java bytecode [3, 5], either at compile-time (where a machine independent bytecode to bytecode optimization is added as additional phase to the Java compiler), or at runtime (where optimizations that require knowledge of the target machine are applied before execution). Some implementations of the JVM further improve performance by means of ‘just-in-time compilation’ (JITC), where, at runtime, bytecode is compiled into native machine code.

In this note, we explore a way to speedup certain operations in Java programs using ideas from the mathematical software community. Here, it has been widely accepted that adopting a set of basic routines for problems in linear algebra can help in improving the clarity, portability, modularity, maintenance, robustness, and even the efficiency of mathematical software. The most well-known example of such a set of routines is formed by the Basic Linear Algebra Subprograms [10, ch.5]. The original set of vector-vector operations is now commonly referred to as Level 1 BLAS [14, 15]. The set has been extended to Level 2 BLAS [8, 9] and Level 3 BLAS [6, 7] to provide more opportunities to exploit vector processing facilities for matrix-vector operations and memory hierarchies or parallelism for matrix-matrix operations, respectively. Once an efficient implementation of BLAS is available, new mathematical software can be easily build on top of the primitives.

Obviously, a similar approach can be taken for Java by extending the Java API (Application Programming Interface) with an appropriate set of mathematical primitives. In first instance, a Java implementation can be provided for all these primitives to preserve the portability of all Java programs in which the mathematical primitives are used. On a particular machine, however, the performance of all Java software that uses these primitives is simply improved by providing native implementations of the mathematical primitives. Although providing a broad range of highly optimized mathematical primitives would offer the best potential to exploit all characteristic a particular target machine, this approach would also require the most programming efforts to port the mathematical primitives in the API to different machines. Therefore, in this research note, we explore the potential of extending the API with straightforward native implementations of Level 1 BLAS only. We will see that this extension alone already can improve performance substantially, while combining these native Level 1 BLAS with multi-threading in Java may even provide a simple and portable way to outperform compiled serial C code on multi-processors.

In section 2, we briefly discuss how native methods are integrated in Java. In section 3, we present the results of a series of experiments, followed by conclusions in section 4.

2 Native BLAS
In Java, if the keyword native appears as part of a method definition (without an implementation), then this implies that the method is implemented in another language. A set of primitives from BLAS, for instance, can be defined using the following class that provides a shared library loader that will load the actual implementation when required as well as definitions of all primitives in the set:

*This project is supported by DARPA under contract ARPA F19628-94-C-0057 through a subcontract from Syracuse University.
class Blas {
    // Shared Library Loader
    static {
        System.loadLibrary("blas");
    }
    // Level 1 BLAS
    native static double ddot(int n, double x, int xoff, int incx, double y, int yoff, int incy);
    native static void daxpy(int n, double alpha, double x, int xoff, int incx, double y, int yoff, int incy);
}

Above, method definitions of DDOT ($d \leftarrow \alpha x^T y$ in double precision) and DAXPY ($y \leftarrow \alpha x + y$ in double precision) are shown. Alternatively, as stated earlier, Java implementations of the primitives could be given in first instance.

In figure 1, the integration of native methods in Java is illustrated. Using JDK1.0.2, first the source file Blas.java is compiled into Blas.class using the Java compiler javac. Subsequently, the tool javah is used to generate a stubs file Blas.c and a header file Blas.h. The latter file contains C prototypes for all native methods in the class. The actual implementation of these methods is given in a file BlasImp.c.

Method ddot(), for example, can be implemented in C as follows, where the macro unhand() is used to dereference an object handle:

```
#include <Stubs/impl.h>
#include "Blas.h"

double Blas_ddot(struct Blas *this, long n, Double x, long xoff, long incx, Double y, long yoff, long incy) {
    double x = unhand(x)->body;
    double y = unhand(y)->body;
    double d = 0;
    if (n > 0) {
        if ((incx == 1) && (incy == 1)) { /* Unit strides */
            for (i = 0; i < n; i++)
                d += x[i] * y[i];
        } else { /* Non-Unit strides */
            ...}
    return d;
}
```

Eventually, the files Blas.c and BlasImp.c are compiled into a shared library. The way in which this is done and the naming convention of shared libraries depends on the target architecture (under Solaris, for instance, the shared library is created as cc -G -O -I $JAVA_HOME/include -I $JAVA_HOME/include/solaris Blas.c BlasImp.c).

As illustrated in the second picture in figure 1, integrating native methods in JDK1.1 is slightly different. Only a header file Blas.h which contains the appropriate prototypes is generated using javah -jni. The actual implementation of methods is, again, given in a file BlasImp.c.

```
    @include "Blas.h"
    JNIEXPORT jdouble JNICALL Java_Blas_ddot
    (JNIEnv *env, jclass class, jint n, jlong x, jint xoff, jint incx, jlong y, jint yoff, jint incy) {
        return xoff == 0 && yoff == 0 ? mjdot(env, n, x, y) : ddot(env, n, x, y, xoff, yoff, incx, incy);
    }
```

Eventually, this file is compiled into a shared library. More details on integrating native methods in Java can be found on the Web [17].

![Figure 1: Integration of Native Methods](image)

Method ddot(), for example, can be implemented in C as follows, where functions GetDoubleArrayElements() and ReleaseDoubleArrayElements() are used to obtain and release a double array that cannot be moved by the garbage collector:

```
JNIEXPORT jdouble JNICALL Java_Blas_ddot
    (JNIEnv *env, jclass class, jint n, jlong x, jint xoff, jint incx, jlong y, jint yoff, jint incy) {
        return (n > 0) ?
            (incx == 1 && (incy == 1)) ? { /* Unit strides */
                 int i;
                 for (i = 0; i < n; i++)
                     d += x[i] * y[i];
            } : { /* Non-Unit strides */
            ...}
    return d;
```

Eventually, this file is compiled into a shared library. More details on integrating native methods in Java can be found on the Web [17].

3 Experiments

In this section, we present the results of a series of experiments that have been conducted on an IBM RS/6000 G30 with four PowerPC 604 processors using the AIX4.2 JDK1.0.2B (with JITC) and the AIX4.2 JDK1.1beta (without JITC), a Sun with two 175MHz ultra SPARC processors using the Solaris 2.5 JDK1.0.2dp (with JITC), and an SGI Indy 4600 with one 133 MHZ IP22 Processor using the IRIX6.2 JDK1.0.2 (without JITC). All Java and C programs are compiled using the flag ‘-O’. Bytecode is interpreted using the flag ‘-nosync’ and, if available, with ‘just-in-time compilation’ (JITC) enabled.

---

1 Because Java does not support passing of arbitrary sub-arrays, offsets into arrays are added as additional parameters (cf. [11]).
3.1 Some Mathematical Operations

In the first series of experiments, we compare the performance of a pure Java implementation, a Java implementation that uses native Level 1 BLAS (and possibly parallelism by means of Java multi-threading), and a compiled C implementation of an \( \Theta(N) \) DAXPY-operation, an \( \Theta(N^2) \) matrix times vector operation, and an \( \Theta(N^3) \) matrix times matrix operation.

Below we show the pure Java implementation of the DAXPY operation \( \bar{y} \leftarrow \alpha \bar{x} + \bar{y} \), and the Java implementation that uses a primitive from Level 1 BLAS:

```java
pure Java:
for (int i = 0; i < n; i++)
    y[i] += alpha * x[i];
Java + Level 1 BLAS:
Blas.daxpy(n, alpha, x, 0, 1, y, 0, 1);
```

The inner product Java implementation of matrix times vector and a Java implementation that uses a call to Level 1 BLAS DDOT \( d \leftarrow \bar{x}^T \bar{y} \) are shown below:

```java
pure Java:
for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
        b[i] += a[i][j] * x[j];
Java + Level 1 BLAS:
for (int i = 0; i < n; i++)
    b[i] = Blas.ddot(n, a[i], 0, 1, x, 0, 1);
```

Finally, the product of two matrices can be computed using either the pure Java fragment shown below, or a similar Java fragment that uses a call to DAXPY:

```java
pure Java:
for (int i = 0; i < n; i++)
    for (int k = 0; k < n; k++)
        c[i][j] += a[i][k] * b[k][j];
Java + Level 1 BLAS:
for (int i = 0; i < n; i++)
    for (int k = 0; k < n; k++)
        Blas.daxpy(n, a[i][k], b[k], 0, 1, c[i], 0, 1);
```

The performance of the fragments for matrix times vector and matrix times matrix with Level 1 BLAS may be further improved by parallelization of the outermost \( i \)-loop. In earlier work \cite{1, 2}, we have shown how loop parallelization can be expressed in Java by means of multi-threading. In this manner the transformed Java program remains portable (i.e. the parallelized version still runs on uni-processors with only a slight overhead), while the programming efforts of the parallelization are reduced substantially with respect to exploiting parallelism in a native language. The versions with a parallel outermost loop are labeled as ‘parallel’ in the subsequent figures.

The C versions of the three mathematical operations are similar to the pure Java implementations.

In figures 2–4 and figures 5–7 we show the execution times for varying values of \( N \) on the IBM using the AIX 4.2 JDK 1.02B (with JITC) and the AIX 4.2 JDK 1.1beta (without JITC), respectively. Here we see that with JITC, providing Level 1 BLAS primitives only suffices to obtain performance that is close to the performance of native C code. Moreover, because IBM’s implementation of the JVM supports the actual parallel execution of threads, the parallel versions with native Level 1 BLAS even outperform serial C code. Without JITC, however, the \( \Theta(N^3) \) operation still suffers from much overhead, and here it would probably be desirable to provide primitives from Level 2 BLAS as well.

In figures 8-10 the execution times on the Sun using the Solaris 2.5 JDK 1.02B (with JITC) are shown. Now, in all cases the performance is even slightly better than the performance of native C code. Obviously, Sun’s implementation of the JVM does not support the actual parallel execution of threads yet.

In figures 11–13 the execution times on the SGI using the IRIX 6.2 JDK 1.0.2 (without JITC) are shown. Again, without JITC, the performance of \( \Theta(N^3) \) matrix times matrix operations is substantially less than the performance of compiled C code. Obviously, on this uni-processor, no speedup can be expected from loop parallelization.

3.2 Linpack Benchmark

In this section, we present some performance numbers for a Java Linpack benchmark \cite{11} that solves a system of linear equations (i.e. factorization followed by forward and back substitution).

<table>
<thead>
<tr>
<th>( n )</th>
<th>JDK 1.0.2</th>
<th>JDK 1.0.2 + native Level 1 BLAS</th>
<th>JDK 1.0.2 + native Level 1 BLAS (parallel factorize)</th>
</tr>
</thead>
<tbody>
<tr>
<td>500</td>
<td>3.3 MFlops</td>
<td>8.1 MFlops</td>
<td>9.7 MFlops</td>
</tr>
<tr>
<td>1000</td>
<td>3.2 MFlops</td>
<td>10.3 MFlops</td>
<td>8.6 MFlops</td>
</tr>
</tbody>
</table>

Table 1: Linpack benchmark on the IBM

In tables 1–3, we show the results on the IBM, Sun, and SGI, respectively, for a pure Java implementation of this benchmark and an implementation that uses native implementation of the primitives DDOT, DAXPY, DSCAL and IDAMAX from Level 1 BLAS. For the IBM, we also present the performance of a version in which loop parallelization has been applied to the elimination step in the factorization.

Again, it is clear that providing native Level 1 BLAS can improve the performance substantially. Note, however, that the mapping between Java and C data types may cause a change in precision of the computed result. Loop parallelization, on the other hand, does not affect the semantics of the program.
Table 2: Linpack benchmark on the Sun

<table>
<thead>
<tr>
<th>n</th>
<th>JDK1.0.2</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>500</td>
<td>Mflops: 8.5 Time: 9.9 s. Norm Res: 5.17</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>Mflops: 8.5 Time: 78.6 s. Norm Res: 10.10</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>JDK1.0.2 + native Level 1 BLAS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>500</td>
<td>Mflops: 18.4 Time: 4.6 s. Norm Res: 5.77</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>Mflops: 18.6 Time: 35.9 s. Norm Res: 10.44</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3: Linpack benchmark on the SGI

<table>
<thead>
<tr>
<th>n</th>
<th>JDK1.0.2</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>500</td>
<td>Mflops: 0.3 Time: 266.4 s. Norm Res: 5.17</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>JDK1.0.2 + native Level 1 BLAS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>500</td>
<td>Mflops: 7.3 Time: 11.6 s. Norm Res: 5.77</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>Mflops: 7.0 Time: 95.9 s. Norm Res: 10.44</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Figure 2: DAXPY on the IBM

Figure 5: DAXPY on the IBM

Figure 3: Matrix x Vector on the IBM

Figure 6: Matrix x Vector on the IBM

Figure 4: Matrix x Matrix on the IBM

Figure 7: Matrix x Matrix on the IBM