Java Signal Processing: FFTs with bytecodes

John Glossner,[1,2] Jesse Thilo,[1,2] and Stamatis Vassiliadis[2]
[1]Lucent / Bell Labs, Allentown, Pa
[2]Delft University of Technology, Delft, The Netherlands
{glossner,jthilo,stamatis}@cardit.et.tudelft.nl

March 31, 1998

Abstract

This paper investigates the possibility of using Java as a language for Digital Signal Processing. We compare the performance of the Fast Fourier Transform using Java interpreters, compilers, and native execution. To characterize the Java language as a platform for signal processing, we have implemented a traditional FFT algorithm in both C and Java and compared their relative performance. Additionally, we have developed a Tensor algebra FFT library in both Matlab and Java. Each of the Tensor libraries has been coded to exploit the characteristics of the source language. Our results indicate that the latest Sun Solaris 2.6 Java platform can provide performance within 20% to 60% of optimized C code on short FFT computations. On longer FFT computations, Java is about a factor of 2 to 3 times less efficient than optimized C code. We anticipate this gap to narrow with better compiler technology and direct execution on Java processors such as the DELFT-JAVA multithreaded processor.

1 Introduction

Java is a new C++-like programming language designed for general-purpose, concurrent, class-based object-oriented programming[1]. The language includes a number of useful programming features including: 1) programmer defined parallelism in the form of synchronized threads, 2) strong typing, 3) garbage collection, 4) classes, 5) inheritance, and 6) dynamic linking. An appeal of the language is a "write once, run anywhere" philosophy[2]. This is accomplished by providing a Java Virtual Machine (JVM) interpreter and runtime for each platform supported [3]. In theory, any platform that supports the Java runtime environment will produce the same execution results independent of the platform.

Because of Java's interpreted execution, it has been asserted that Java is inappropriate for Signal Processing applications. We investigate this hypothesis by comparing the performance of a traditional Fast Fourier Transform (FFT) algorithm originally written in C and translated to Java. We then extend our analysis to a Tensor algebra FFT library coded in both Matlab and Java.

We have also designed the DELFT-JAVA processor[4]. An important feature of this architecture is that it has been designed to efficiently execute JVM bytecodes. The architecture has two logical views: 1) a JVM Instruction Set Architecture(ISA) and 2) a RISC-based ISA. The JVM is a stack-based ISA with support for standard datatypes, synchronization, object-oriented method invocation, arrays, and object allocation[3]. An important property of Java bytecodes is that statically determinable type state enables simple on-the-fly translation of bytecodes into efficient machine code[5]. We utilize this property to dynamically translate Java bytecodes into DELFT-JAVA instructions. Because the bytecodes are stored as pure Java instructions, Java programs generated by Java compilers execute on a DELFT-JAVA processor without modification.

We have previously developed a Matlab implementation of a Tensor algebra library[6]. For this analysis, we developed an equivalent library in Java. A key point of each implementation is that it is written to utilize important features of the source language. The Matlab implementation utilizes a vector encoding and takes advantage of the built-in Complex datatype. The Java implementation uses a Complex class that operates on the Java datatype double. This allows us to treat a complex number as any other number thereby mimicking the built-in features of the Matlab system. In addition, the Java library is reentrant to allow for multithreaded programming. Each Tensor algebra implementation is built from basic functions. A stage of an FFT is built by executing a number of parallel FFTs followed by a Permutation phase. The mathematical description of the algorithm is a parallel decomposition based on Pease[7] and is only briefly summarized along with one possible mapping to a multithreaded processor. More detailed definitions of the Kronecker product (denoted by $\otimes$) can be found in [8, 9].

In Section 2 we give a brief introduction to Tensor algebra and the Fast Fourier Transform. In Section 3 we present the results of two studies. The first study compares optimized C and Java FFT routines. The second study compares a Tensor library implementation in both Matlab and Java. Finally, in Section 4 we summarize our findings and present conclusions.

2 Tensor Algebra

In this section, we provide an introduction to the Fast Fourier Transform and Tensor algebra. We begin by distinguishing between software *threads* and hardware thread units (referred to as *contexts*). If t is the number of software threads and c is the number of hardware contexts on which the threads may execute, a possible mapping of the threads to hardware units is given by a tensor decomposition.

Definition 1 *Tensor Product. The tensor product of two matrices $A = [a_{t_1,c_1}]$ of order $(t_1 \times c_1)$ and $B = [b_{t_1,c_2}]$ of order $(t_2 \times c_2)$, denoted by $A \otimes B$, is*

$$A_{t_1,c_1} \otimes B_{t_2,c_2} = \begin{bmatrix} a_{00}B & \cdots & \cdots & a_{0(c_1-1)}B \\ a_{10}B & \ddots & & a_{1(c_1-1)}B \\ \vdots & & \ddots & \vdots \\ a_{(t_1-1)0}B & \cdots & \cdots & a_{(t_1-1)(c_1-1)}B \end{bmatrix} \tag{1}$$

The resultant matrix is of order $(t_1 t_2 \times c_1 c_2)$ and each submatrix of the form $a_{t_i c_j}B$ is of the order $(t_2 \times c_2)$. If I_n is the identity matrix of order n, this can also be expressed as

$$A_{t_1c_1} \otimes B_{t_2c_2} = (A_{t_1c_1} \otimes I_{t_2})(I_{c_1} \otimes B_{t_2c_2}) = (I_{t_1} \otimes B_{t_2c_2})(A_{t_1c_1} \otimes I_{c_2}) \tag{2}$$

The tensor product is not commutative. However, we can use a class of permutations called stride permutations which will commute the tensor product. Furthermore, these stride permutations govern the load and store data addressing between stages of the tensor product decompositions of DSP algorithms[9]. The permutation matrix partitions the computations and communications by changing the dataflows from vector to parallel or from parallel to vector dataflows. Given a multithreaded processor with multiple contexts, it is possible to optimize the computations such that the thread level parallelism available in the algorithm can be exploited as Instruction Level Parallelism (ILP) in the processor. The tensor permutation matrix P_t^{tc} is a $(tc \times tc)$ matrix where the elements $(P_t^{tc})_{i,j}$ are equal to 1 for $i = j = (tc-1)$ and $j = it$ $mod\ (tc-1)$, $0 \leq i \prec (tc-1)$ and equal to 0 for all other i, j elements. As an example, P_2^4 is generated by $(P_2^4)_{i,j} = 1$ for $(i,j) = (0,0), (1,2), (2,1), (3,3), (P_2^4)_{i,j} = 0$ for other i, j elements. Given that P is a tensor permutation matrix, then if A is a $(t \times t)$ matrix and B is a $(c \times c)$ matrix, we can commute the tensor equation using $A_t \otimes B_c = P_t^{tc}(B_c \otimes A_t)P_c^{tc}$.

Some additional properties of tensor algebra include scalar multiplication $A \otimes \alpha B = \alpha(A \otimes B)$, distributive law $(A+B) \otimes C = A \otimes C + B \otimes C$, associative law $A \otimes (B \otimes C) = (A \otimes B) \otimes C$, identity product $I_{tc} = I_t \otimes I_c$, matrix transpose $(AB)^t = B^t A^t$, tensor transpose $(A \otimes B)^t = A^t \otimes B^t$ and mixed product rule $(A \otimes B)(C \otimes D) = AC \otimes BD$.

2.1 Fourier Transform

The N-point Fast Fourier Transform(FFT) is given by Equation (3).

Definition 2 *Fast Fourier Transform. The N-point FFT is given as*

$$FFT(x) = y_k = \sum_{j=0}^{N-1} \omega^{jk} x_j \ \textit{for}\ 0 \leq k \prec N\ \textit{where}\ \omega = e^{\frac{-2\pi i}{N}},\ i = \sqrt{-1} \tag{3}$$

The tensor form of the N-point FFT is given as $F_{tc} = P_t^n(I_c \otimes F_t)P_c^n T_c^n(I_t \otimes F_c)P_t^n$ where F is the Fourier Matrix $F_n = [\omega^{jk}]$, n is equal to tc, P is the tensor permutation matrix, and T is a twiddle factor matrix whose elements along the diagonal of the matrix are given by $T_c^n = diag(I_c, D_c(n), ..., D_c(n)^{t-1})$ and $D_c(n) = diag(1, \omega, ...\omega^{c-1})$. This has the interpretation of decomposing the FFT into an initial data permutation P_t^n followed by t parallel c-point FFTs. The twiddle factor is then applied followed by a P_c^n data permutation. Then c independent t-point FFTs are performed. Finally, the data is permuted to its final non-bit reversed form. We can interpret t as the number of software threads in a system and c as the number of hardware contexts (e.g. thread units). On the DELFT-JAVA processor, we try to optimize the number of threads t to be an integer multiple of the number of hardware contexts c. When this is possible, all of the hardware thread units make progress computing the FFT. Furthermore, since all inter-thread communications take place through shared memory, the permutations serve as a barrier synchronization point. We summarize the FFT equations used in this paper in Equations (4-8). We note that based on identity product, it is possible to decompose the first stage of Equations (5-8) into multiple iterations of $I_c \otimes I_{\frac{t}{c}}$ so that each thread unit is active for all computations. It is also possible to rearrange the tensor formulation so that communication through shared memory is minimized[10]. For the purposes of this paper, Equations (4-8) provide adequate comparison performance.

$$
\begin{aligned}
F_4 &= \text{direct implementation} && (4)\\
F_{16} &= P_4^{16}(I_4 \otimes F_4)P_4^{16}T_4^{16}(I_4 \otimes F_4)P_4^{16} && (5)\\
F_{64} &= P_{16}^{64}(I_4 \otimes F_{16})P_4^{64}T_4^{64}(I_{16} \otimes F_4)P_{16}^{64} && (6)\\
F_{256} &= P_{64}^{256}(I_4 \otimes F_{64})P_4^{256}T_4^{256}(I_{64} \otimes F_4)P_{64}^{256} && (7)\\
F_{1024} &= P_{256}^{1024}(I_4 \otimes F_{256})P_4^{1024}T_4^{1024}(I_{256} \otimes F_4)P_{256}^{1024} && (8)
\end{aligned}
$$

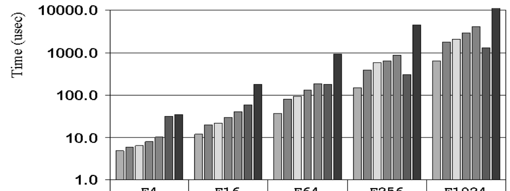
3 Results

In this section we describe the performance models and results of our experiments. All experiments were conducted using a 300MHz Sun Ultra-Sparc II with 128MB of memory running Solaris 2.6. For all experiments, multiple iterations were performed to minimize start-up effects. The average time is reported based on a time-of-day function in a root window at maximum priority in an otherwise nearly idle system. We measure the execution time of the algorithms and not the time to load the interpreters or the time to compile the C code.

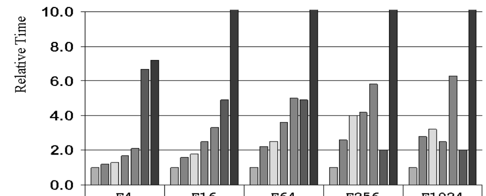
The first experiment investigates a traditional FFT algorithm from Press[11]. The *C code* is compiled using gcc 2.7.2. Results are presented for unoptimized and -O3 optimized execution. The C program was then hand-translated into an equivalent Java program. The resulting code does not take advantage of most of the Java programming constructs. It is essentially the same C code with a Java class wrapper. Results are presented for -O optimized Java code. There were no practical differences between unoptimized and -O optimized Java results. The *Matlab* numbers are the times for the built-in FFT routines. The *Sun Solaris 2.6* results are for the built-in Java interpreter with JIT that comes standard as part of the Solaris 2.6 installation. The *Sun JDK 1.1.4* results are reported for comparison. The *Kaffe* 0.9.2 just-in-time compiler results are also presented for comparison . The *Toba*[12] results are for the 1.0b6 beta release. Toba *off-line translates* Java bytecodes to C code. The C code was then compiled with gcc 2.7.2 with -O3 optimization.

Table 1 shows that the best performance in all cases was for optimized C code. A notable point is that the Sun Solaris 2.6 platform came within 20% of the optimized C code for the smallest FFT but diverged to about 3 times slower for larger FFTs. This is a significant improvement over the JDK 1.1.4 platform. The new results improve Java's performance up to 12 times versus the JDK 1.1.4. In addition, the Solaris 2.6 Java platform is comparable in performance to the Toba off-line compiler which produced very impressive results. Matlab's built-in FFT libraries are also very competitive with optimized C - particularly as the FFT size increases. The Kaffe JIT is competitive when compared to the JDK 1.1.4 but does not perform as well as the Sun Solaris 2.6 platform.

The second experiment investigates the effects on performance when many of the features of the Java language are involved. Unlike the code of the first experiment, the Java Tensor library makes use of many



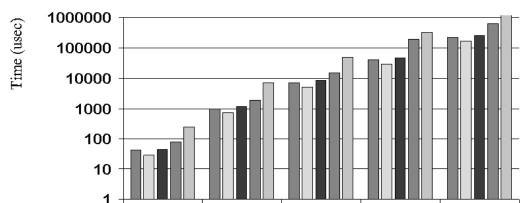
	F4	F16	F64	F256	F1024
gcc -O3	4.8	12	37	149	642
Solaris 2.6 w/JIT	5.8	20	81	384	1780
Toba 1.0b6 -O3	6.3	22	93	591	2070
gcc unopt	8.0	30	132	630	2890
Kaffe 0.9.2	10.2	41	185	867	4070
Matlab built-in	32.0	60	180	300	1300
Java 1.1.4 Interp	34.7	181	918	4520	22000



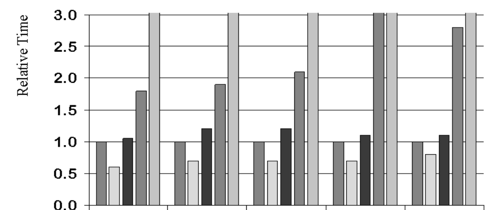
	F4	F16	F64	F256	F1024
gcc -O3	1.0	1.0	1.0	1.0	1.0
Solaris 2.6 w/JIT	1.2	1.6	2.2	2.6	2.8
Toba 1.0b6 -O3	1.3	1.8	2.5	4.0	3.2
gcc unopt	1.7	2.5	3.6	4.2	2.5
Kaffe 0.9.2	2.1	3.3	5.0	5.8	6.3
Matlab built-in	6.7	4.9	4.9	2.0	2.0
Java 1.1.4 Interp	7.2	15.0	25.0	30.0	34.0

Table 1: Press FFT Analysis.

of the Java programming constructs. In particular, inherited class objects are used and the code is reentrant to allow for multithreaded subroutines. Because of this, many objects are created which require garbage collection.



	F4	F16	F64	F256	F1024
Solaris 2.6 w/JIT	44	1000	7200	41000	221000
Toba 1.0b6 -O3	28	730	5100	29000	166000
Java 1.1.4 Interp	46	1200	8400	47000	254000
Kaffe 0.9.2	81	1900	15000	196000	629000
Matlab Interp	250	7100	50000	318000	3470000



	F4	F16	F64	F256	F1024
Solaris 2.6 w/JIT	1.0	1.0	1.0	1.0	1.0
Toba 1.0b6 -O3	0.6	0.7	0.7	0.7	0.8
Java 1.1.4 Interp	1.1	1.2	1.2	1.1	1.1
Kaffe 0.9.2	1.8	1.9	2.1	4.8	2.8
Matlab built-in	5.7	7.1	6.9	7.8	15.7

Table 2: Tensor FFT Analysis.

Table 2 shows the results of the Tensor algebra library for Matlab and Java. In all cases, the Toba compiler performs better than any other alternative. Most interestingly, the performance difference between the Solaris 2.6 platform and the JDK 1.1.4 are within 20% for all cases. This is significantly different than the 12x improvement noted in the first experiment. We postulate that the JIT compiler may not have been able to perform as many optimizations due to the large number of objects created. A profile of each execution stream showed computations to be the predominant time factor in the first experiment while object creation and method invocation were much larger percentages of the execution time for the second experiment. Toba, on the other hand, may take as long as necessary to compile (although our experience is that it is not much more than a few minutes) and therefore produces more optimized code. Also, it may not be fair to compare these FFT results with the results presented in Table 1. The tensor libraries were written to assist partitioning of parallel code onto multiprocessors and multithreaded processors and not particularly for absolute performance on a uni-processor.

4 Conclusions

We have compared FFT performance in Java, Matlab, and C. We have found that Java may offer sufficient performance for FFTs. However, when compared against native C code, the best Java off-line translation and JIT's may still execute 2 to 3 times slower than an equivalent algorithm written in C. However, Sun's latest JVMs have significantly closed the gap between native C performance and Java. The JVM shipped as part of the Solaris 2.6 operating system is nearly 10 times more efficient on C-like Java code than the JDK 1.1.4 and produces results within 20% to 60% of optimized C for small FFTs. The Kaffe JIT runs about 2 to 5 times slower than the interpreted code for the tensor model. We attribute this to the large number of objects created and the requirement for an efficient garbage collector. We also note that the state of Java compilation is relatively immature compared to C compilers. As Java compilers become more sophisticated, the gap may narrow further. Furthermore, direct compilation of Java to native machine code may provide performance closely rivaling C. In addition, raw performance is not the only metric influencing the success of a product. For example, Matlab has an exceptionally rich library of efficient signal processing routines. The ease with which these are integrated with other Matlab programs makes Matlab an excellent DSP development environment. In addition, the number of lines of code written for the Matlab program was less than for Java due to the built-in Complex type. Finally, we conclude that for applications that make predominant use of Java, application specific processors may accelerate Java execution to be at least on par with and potentially better than C code execution on a traditional processor. Depending upon parameters chosen (e.g. issue rates, clock speed, and functional unit latencies), we have estimated that a DELFT-JAVA processor can also execute in comparable time relative to native C code. However, at this point our models do not take into account I/O or garbage collection. Our current work is focussing on Java multithreaded performance which may provide the possibility of significant gains on multithreaded processors.

References

[1] James Gosling, Bill Joy, and Guy Steele, editors. *The Java Language Specification.* The Java Series. Addison-Wesley, Reading, MA, USA, 1996.

[2] James Gosling and Henry McGilton. The Java Language Environment: A White Paper. Technical report, Sun Microsystems, Mountain View, California, October 1995. Available from ftp.javasoft.com/docs.

[3] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification.* The Java Series. Addison-Wesley, Reading, MA, USA, 1997.

[4] C. John Glossner and Stamatis Vassiliadis. The Delft-Java Engine: An Introduction. In *Lecture Notes In Computer Science. Third International Euro-Par Conference (Euro-Par'97 Parallel Processing)*, pages 766–770, Passau, Germany, Aug. 26 - 29 1997. Springer-Verlag.

[5] James Gosling. Java Intermediate Bytecodes. In *ACM SIGPLAN Notices*, pages 111–118, New York, NY, January 1995. Association for Computing Machinery. ACM SIGPLAN Workshop on Intermediate Representations (IR95).

[6] C. John Glossner, Gerald G. Pechanek, Stamatis Vassiliadis, and Joe Landon. High-Performance Parallel FFT Algorithms on M.f.a.s.t. Using Tensor Algebra. In *Proceedings of the Signal Processing Applications Conference at DSPx'96*, pages 529–536, San Jose Convention Center, San Jose, Ca., March 11-14 1996.

[7] Marshall C. Pease. An Adaptation of the Fast Fourier Transform for Parallel Processing. *Journal of the ACM*, 15(2):252–264, April 1968.

[8] J.R. Johnson, R.W. Johnson, D. Rodriguez, and R. Tolimieri. A Methodology for Designing, Modifying, and Implementing Fourier Transform Algorithms on Various Architectures. *Circuits Systems Signal Process*, 9(4):449–500, 1990.

[9] R. Tolimieri, M. An, and C. Lu. *Algorithms for Discrete Fourier Transform and Convolution.* Springer-Verlag, New York, 1992.

[10] S. K. S. Gupta. *Synthesizing Communication-Efficient Distrubted-Memory Parallel Programs for Block Recursive Algorithms.* PhD dissertation, The Ohio State University, Computer Science Department, 1995.

[11] William H. Press, Saul A. Teukolskey, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing.* Cambridge University Press, Cambridge, UK, 2 edition, 1992.

[12] Todd A. Proebsting, Gregg Townsend, Patrick Bridges, John H. Hartman, Tim Newsham, and Scott A. Watterson. Toba: Java For Applications - A Way Ahead of Time (WAT) Compiler. In *Proceedings Third Conference on Object-Oriented Technologies and Systems (COOTS'97)*, 1997.