Vectorization Experiment on the IBM 3090 Vector Facility

Joerg Richard Weimar

TR 90-14

VECTORIZATION EXPERIMENT ON THE IBM 3090 VECTOR FACILITY

February 23rd, 1990

Joerg Richard Weimar

Technical Report
Computer Science Department, Virginia Tech
Email (Internet) : weimar@csgrad.cs.vt.edu

Vectorization Experiment on the IBM 3090

Abstract

A comparison between sequential code and vectorized code for matrix multiplication is made and the possible ways to vectorize matrix multiplication are examined. The result is that vectorization of the correct loop results in a speedup of almost 3 for vectors that are long enough, i.e., longer than or equal to half the vector length of the machine. The number and order of memory references is a very important factor.

Experimental Setup

The intent of the experiment is to see how different ways of doing matrix multiplication, with and without use of the vector facility, affect the speed. The code used for this is shown in Table 1 on page 2. There are six possible orderings of the 3 nested loops and this order affects the speed. All orderings are tested in scalar and vector mode. The orderings are named by the nesting order of the variables in the code. The ordering shown in Table 1 on page 2 is called ijk. The possible orderings are : ijk, jik, kji, kij, ikj, jki .

The parameter N, the size of the matrix, is varied to assume all powers of two from 8 to 1024. The time is measured using the FORTRAN call "cputime", which returns the cpu time in microseconds.

No efforts are made to control the multiprogramming level of the machine or the load factor. The times for the smaller matrices are based on a repetition of the matrix multiplication loops for 10 to 300 times.

Technicalities

The operating system used is IBM AI/X running on the IBM 3090 with a vector facility with vector length 128. The compiler is the VS Fortran II compiler called **fvs** in AI/X.

To enable the vectorization of the code, the compiler is called using the sequence

```
fvs mamu.f -o mamu -f " opt(2) vector( report ) " -xa
```

In order to control which loop is vectorized, it is necessary to introduce directives that specify which loop to vectorize. This is done by putting the statement

```
@PROCESS  DIRECTIVE "*VDIR"
```

Table 1. Code used for the experiment

```
      PROGRAM MAMULT
      PARAMETER ( N = 256 )

      REAL C(N,N), A(N,N), B(N,N)
      INTEGER I, J, K
      INTEGER  TIME1, TIME2 , TDIFF
      INTEGER RCODE

      DO 20 I = 1 , N
        DO 20 J = 1 , N
          A(I,J) = 3.14159265
          B(I,J) = 0.1234567890
          C(I,J) = 0.0
20      CONTINUE

      CALL CPUTIME(TIME1,RCODE)

      DO 10  I = 1 , N
        DO 10  J = 1 , N
C*VDIR  PREFER VECTOR
          DO 10  K = 1 , N
            C(I,J) = C(I,J)+ A(I,K) * B(K,J)
10      CONTINUE

      CALL CPUTIME(TIME2,RCODE)
      TDIFF = (TIME2 - TIME1 )/R
      WRITE (6,1) 'IJK',N,R,TDIFF, TDIFF/N/N,TDIFF/(N**3)
1     FORMAT(A3,' N= ',I4,' R= ',F3.0,' T= ',F15.3,
     +   ' T/N**2 ',F12.4,' T/N**3 ',F12.6)

      END
```

at the beginning of the program and the line

```
C*VDIR  PREFER VECTOR
```

immediately before the loop that is to be vectorized. Otherwise the compiler changes the order of the nested loops and vectorizes the loop which is best to vectorize, and therefore always produces the same result as the kji order.

Results

The resulting execution times are presented in bargraphs in the appendix, where the times are normalized by $\frac{1}{n^2}$ and $\frac{1}{n^3}$ respectively. Because matrix multiplication involves $O(n^3)$ operations, it can be expected that times normalized by $\frac{1}{n^3}$ are relatively equal for different n. This can indeed be observed for the "good" order (kji and jki) scalar version. The vector version shows decreasing $\frac{T}{n^3}$ for n < 128. For $n \geq 128$, which is the vector length on this machine, the normalized times do not change. It can also be seen that only the versions where the innermost loop runs over the variable i have a reasonable execution time and have this property of constant normalized time.

Table 2. Code used for the first autovector,attempt

```
      DO 10 repeat = 1 , R
        DO 10 I = 1 , N
          DO 10 J = 1 , N
            DO 10 K = 1 , N
              C(I,J) = C(I,J)+ A(I,K) * B(K,J)
10      CONTINUE
```

If the innermost loop runs over the variable k, the execution times are much higher, but the vector version is still faster than the scalar version. This is not the case if the innermost loop runs over the variable j, in which case both scalar and vector code are very slow, with the vector code being even slower that the scalar code for $n \geq 128$.

Automatic Vectorization

The compiler does an analysis of the cost of vectorization for each possible ordering, if the nesting can be reordered, and will choose the best possibility for the vectorization (as long as this is not overidden by a directive as in this experiment). This is observed in a test with auto vectorization for n = 256, which produced the same results as the kji or jki ordering for all possible orderings (see Figure 4).

There are, however, some problems with the automatic vectorization: In a first attempt at auto vectorization, the code as in Table 2 was used. Here the loops for the matrix multiplication were embedded in another loop to repeat the whole matrix multiplication several times (this is used to get nonzero timing results for small matrices). In that version, this outermost loop of R identical computations is vectorized by the compiler into a vector of R identical computatations, the results of which are then copied one after the other into the variable c(i,j). This happens, because this "vector" has stride 0 (see following section), so it is cheapest to vectorize. The compiler does not recognize the problem that the computations are identical, nor does it take the vector length into account. For the larger matrices, this parameter R is always 1, so this essentially results in worse than scalar code, because the vector unit is used for scalar operations.

In the case of the scalar code, which is also affected by the ordering of the nested loops, no such analysis is done and therefore the programmer has to take care of that.

Explanation

The results of vectorizing different loops of the three nested loops necessary to perform the matrix multiplication can be explained in terms of stride of the memory accesses. Stride of memory accesses is defined as the distance between succesive elements of one vector. In order for the vector unit to work efficiently, this stride needs to be small. This is the criterion applied by the autovectorizer, which minimizes the cost calculated as a function of the strides inside the vectorized

loop. Vectorizing the i-loop results in three accesses with stride 1 and one access with stride 0, because the matrices are stored in column order. Vectorizing the k-loop results in two accesses with stride 0 (the c(i,j) access), one access with stride 1 and one access with stride N. Vectorizing the j-loop results in one access with stride 0 and three accesses with stride N. Therefore the best choice is to vectorize the i-loop. This only holds if the vector length is roughly the same for all loops. The compiler does not try to take the vector length into account, as could be seen in the problem described in the previous paragraph.

Optimization

In the experiment the optimization level is 2. This is choosen, because the vector option of the compiler requires the optimization level 2 or 3 (out of 0..3). To observe the effect of these optimization levels, another test was performed to observe the effect of optimization on the code. In this test, the scalar code for n = 256 is tested with each optimization level from 0 (no optimization) to 3 . Only for level 0 there was a severe decrease in speed (factor 6 slower), as can be seen in Figure 4.

Conclusion

For vectors of at least half the hardware vector length, vectorization results in a speedup of 2 to 3 over scalar execution. The matrix multiplication execution time is very sensitive to the order of the nested loops in the algorithm, although all orderings have the same semantics. The absolute speed of the 3090 vector code is 25 MFLOPS, or 12.5 million times the operations "multiply" and "add" per second. This is for 1024 * 1024 matrices. For 64 * 64 matrices the speed is only 22.4 MFLOPS. The scalar execution speed is 7.91 MFLOPS for 1024 * 1024 and 8.51 MFLOPS for 64 * 64 matrices.

Figure 1 : Time / N^3 for Good Order

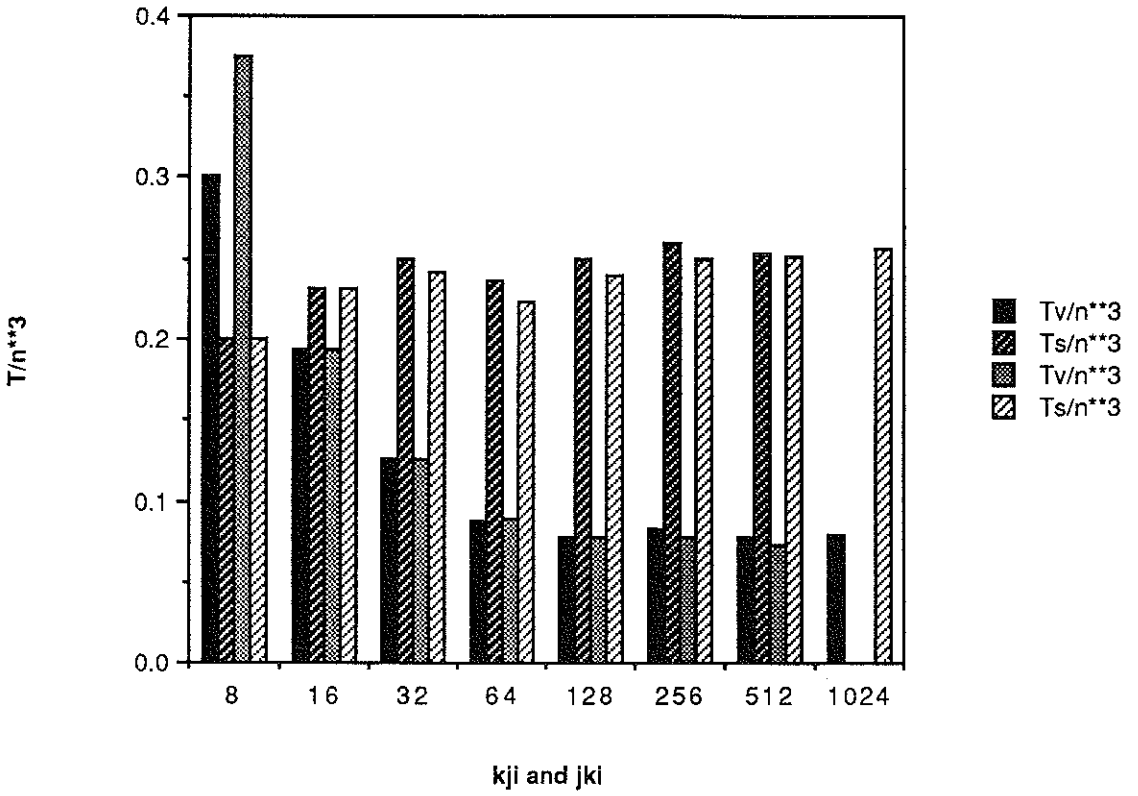

Figure 2 : Time / N^3 for Medium Good Order

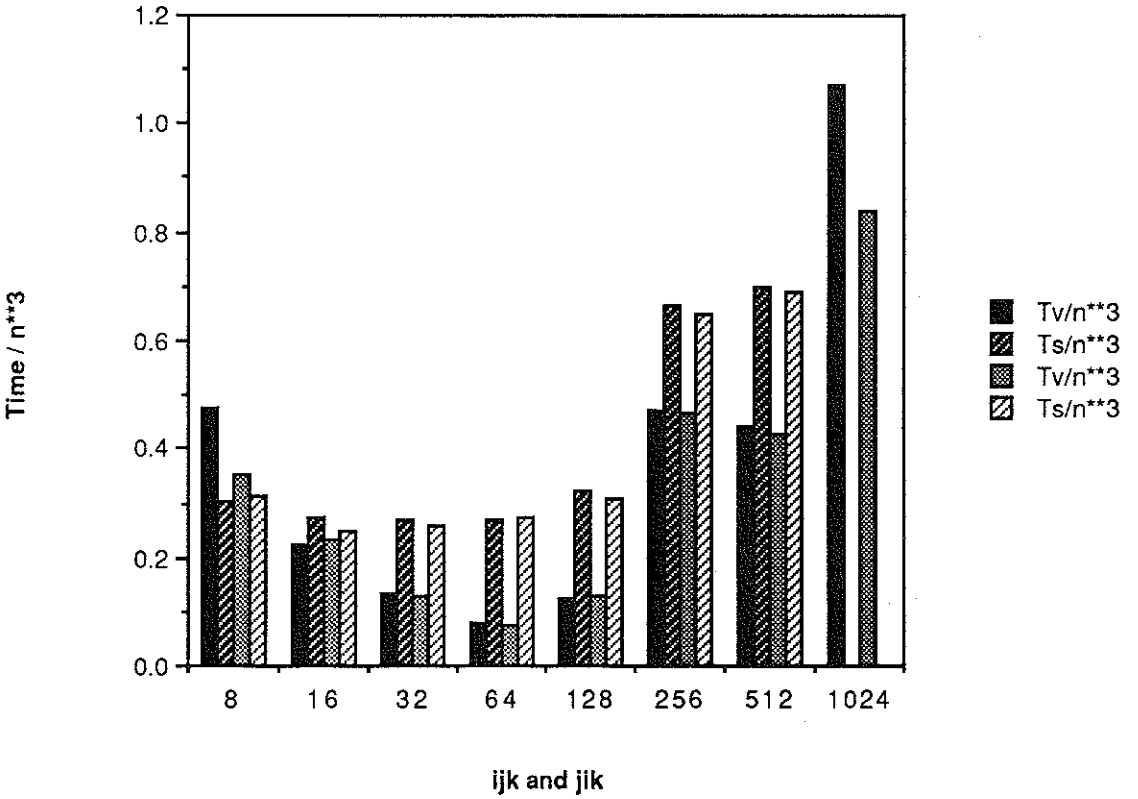

Figure 3 : Time /N^3 for Bad Order

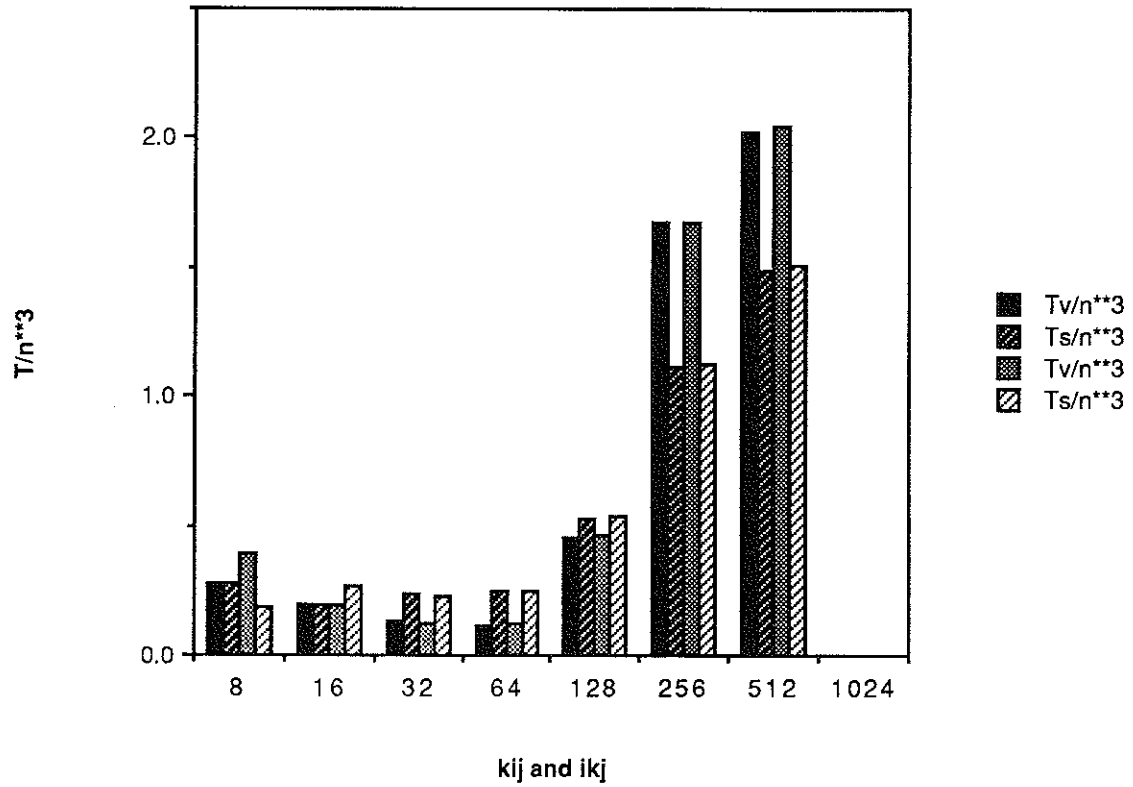

Figure 4 : Additional Tests for n=256

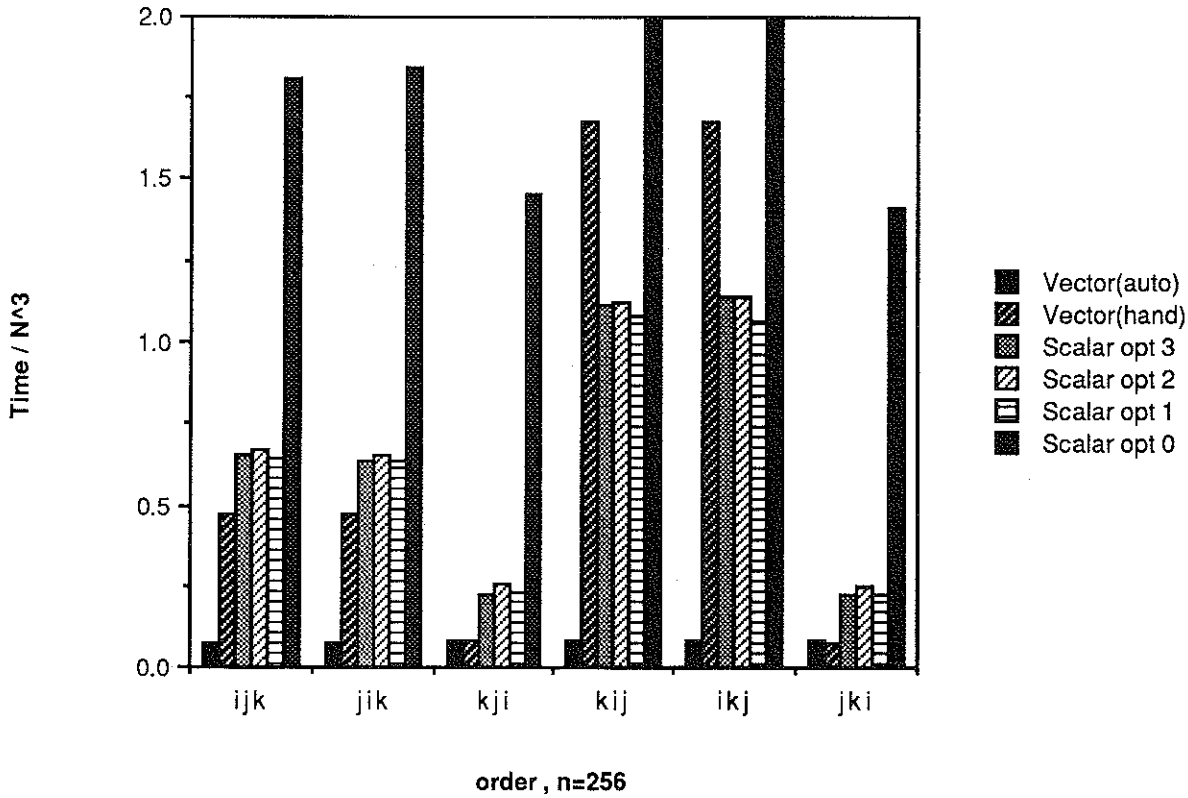

Figure 5 : Vector and Scalar Times for n = 8

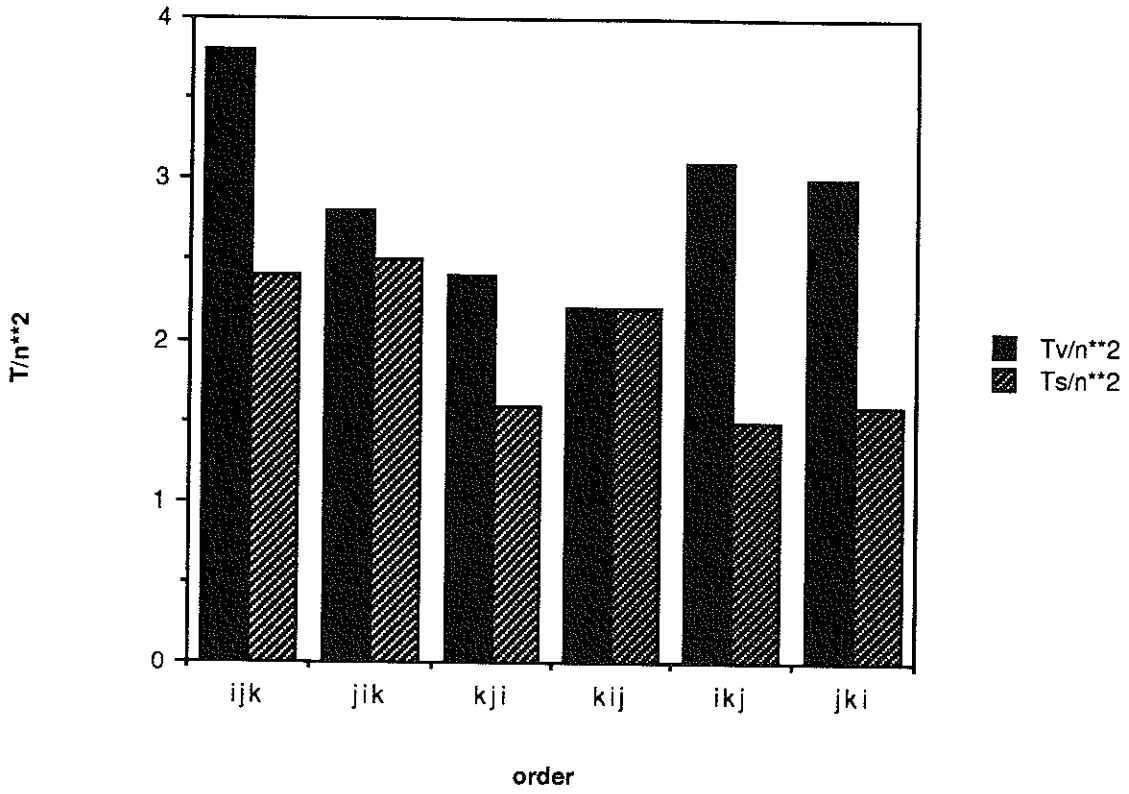

Figure 6 : Vector and Scalar Times for n = 16

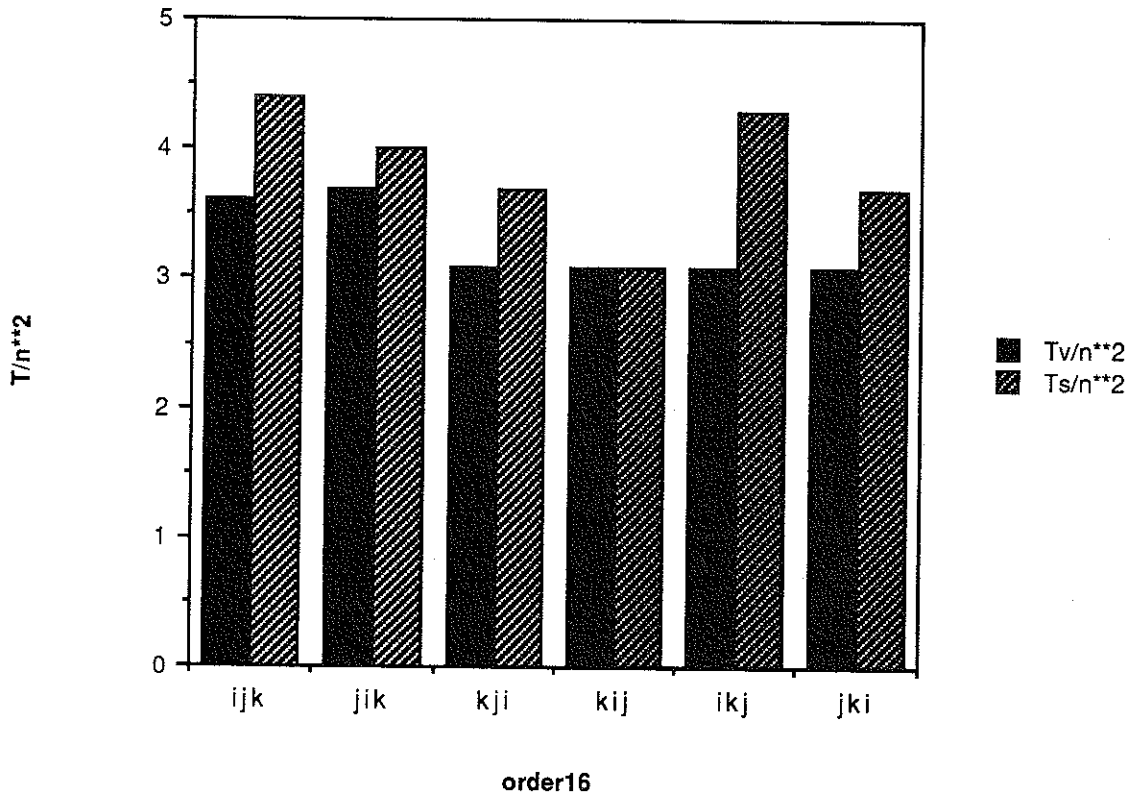

Figure 7 : Vector and Scalar Times for n = 32

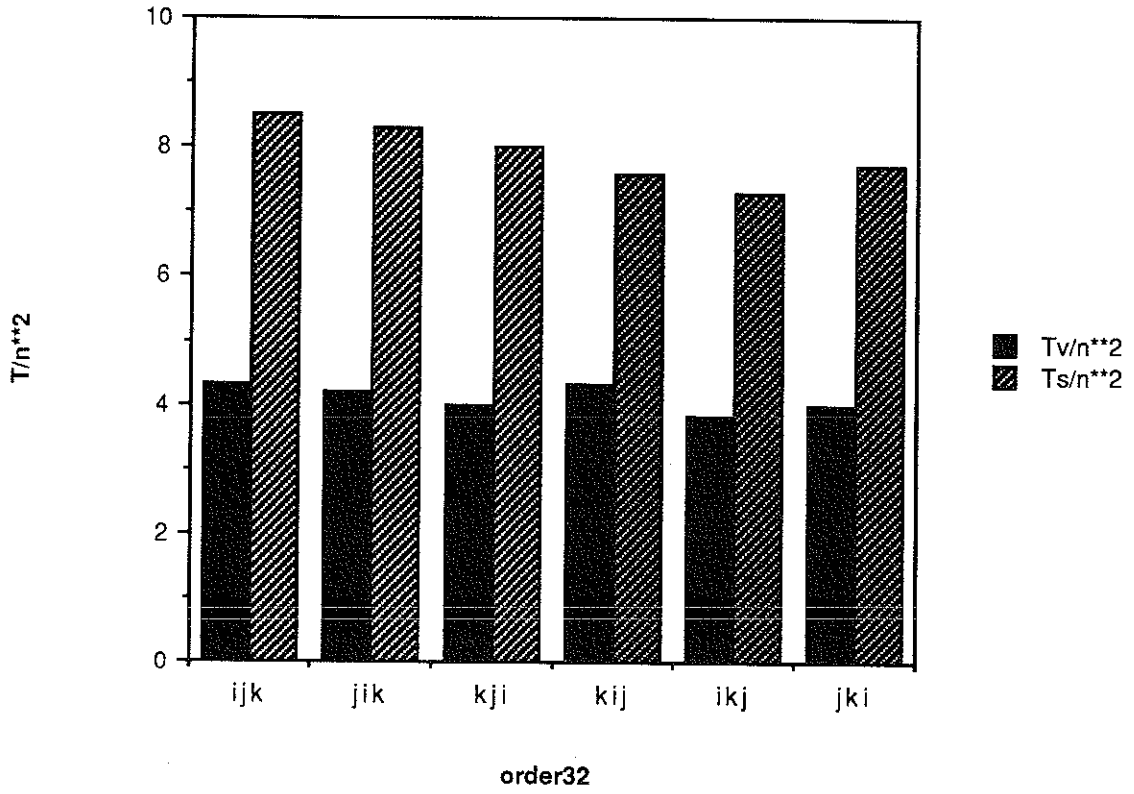

Figure 8 : Vector and Scalar Times for n = 64

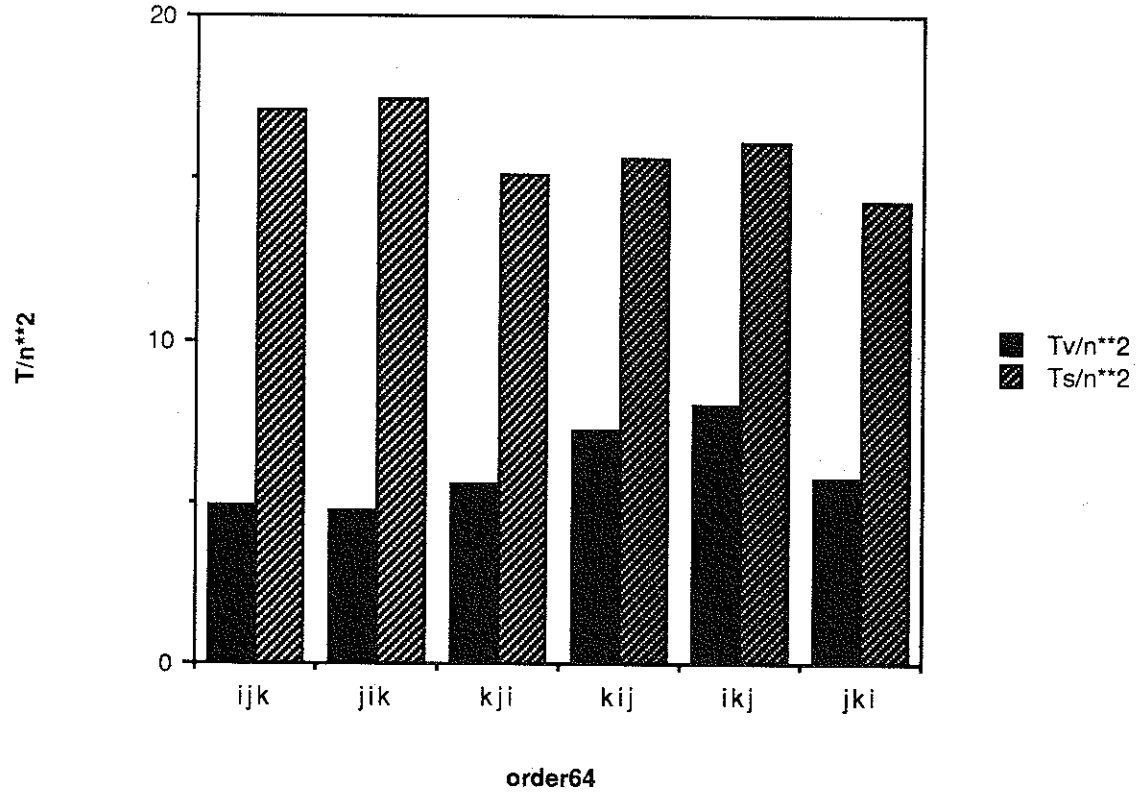

Figure 9 : Vector and Scalar Times for n = 128

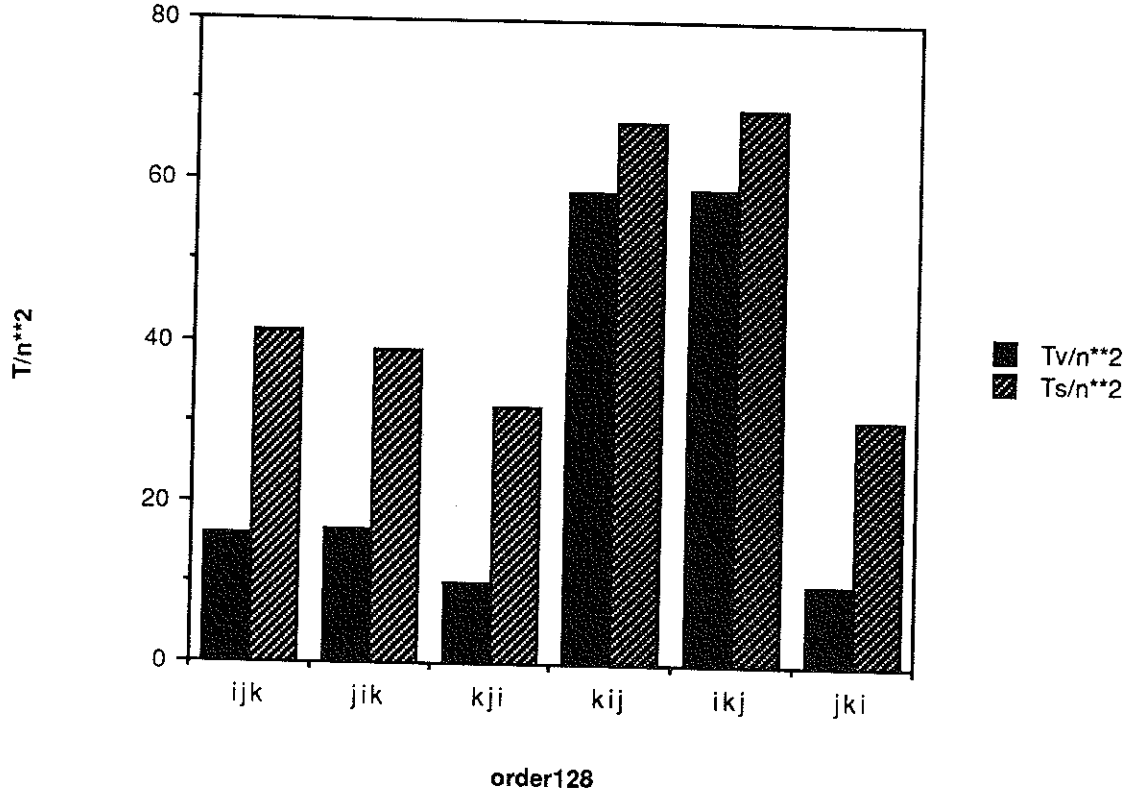

Figure 10 : Vector and Scalar Times for n=256

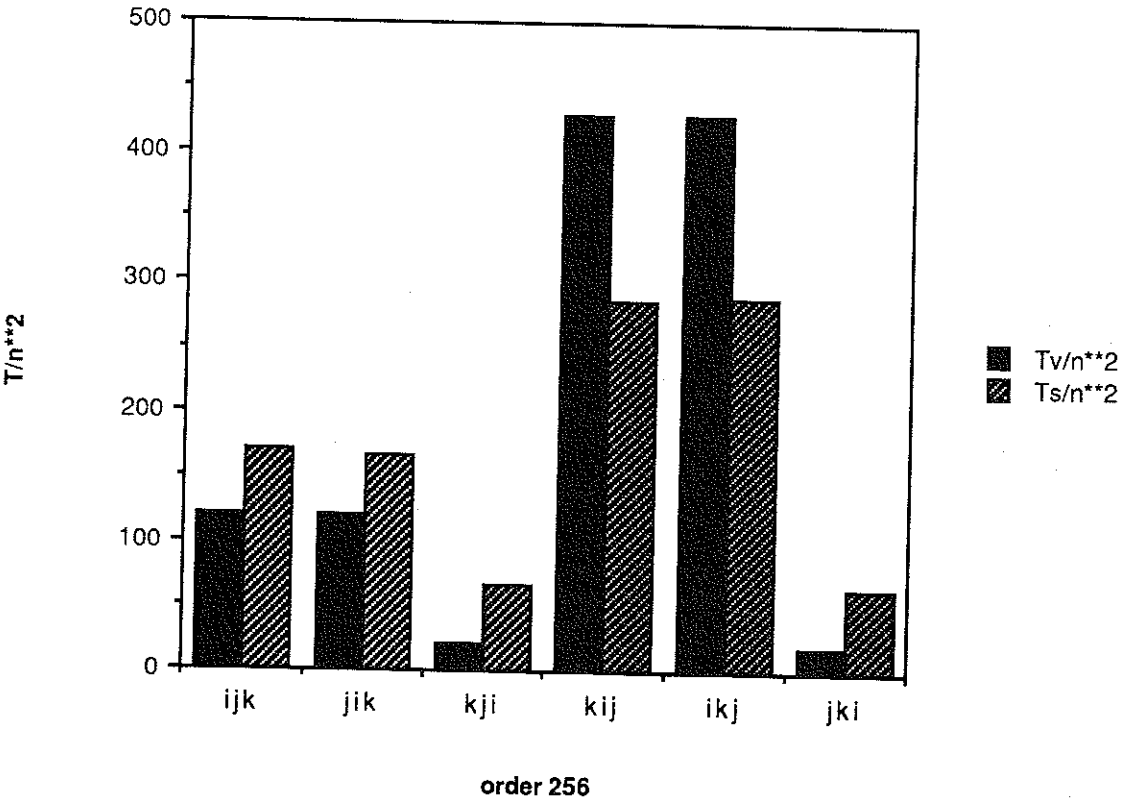

Figure 11 : Vector and Scalar Times for n = 512

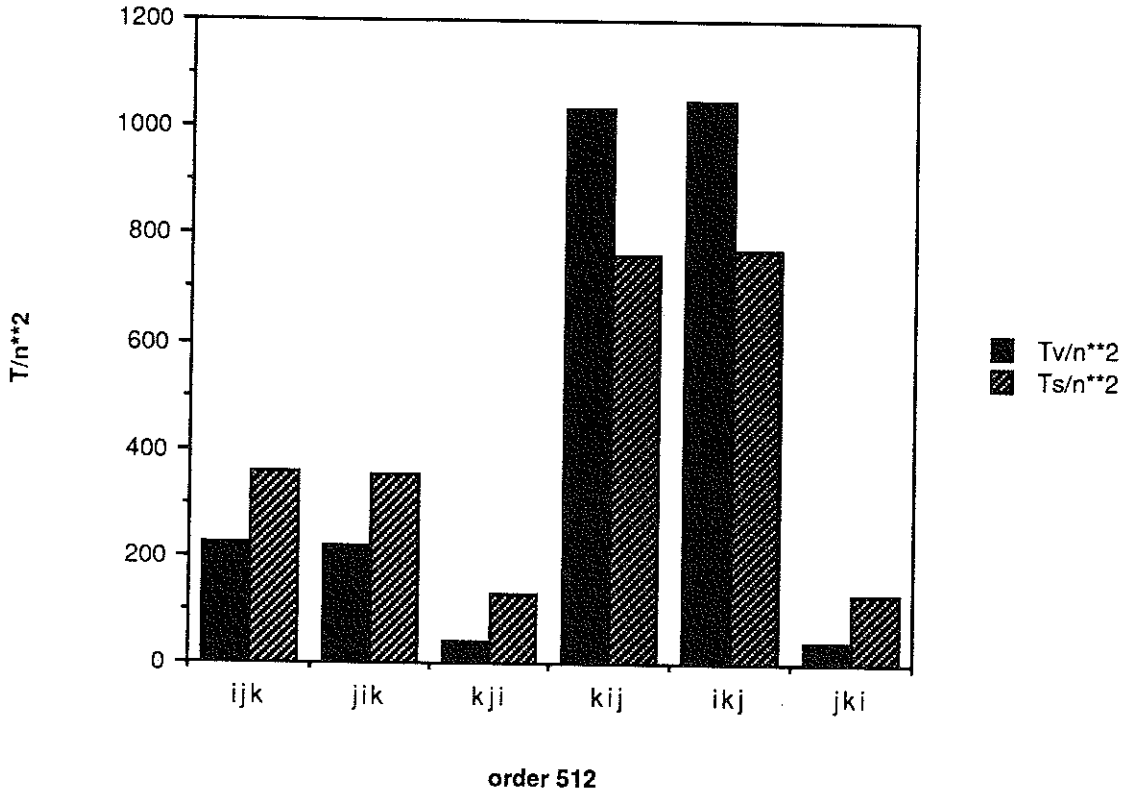

Figure 12 : Some Vector and Scalar Times for n = 1024