Vectorization and Re-Use Issues

P. Sam Johnson

National Institute of Technology Karnataka (NITK)
Surathkal, Mangalore, India

Introduction

The matrix manipulations discussed in this book are mostly built upon dot products and saxpy operations. Vector pipeline computers are able to perform vector operations such as these very fast because of special hardware that is able to exploit the fact that a vector operation is a very regular sequence of scalar operations.

Whether or not high performance is extracted from such a computer depends upon the length of the vector operands and a number of other factors that pertain to the movement of data such as vector stride, the number of vector loads and stores, and the level of data re-use. Our goal is to build a useful awareness of these issues.

We are not trying to build a comprehensive model of vector pipeline computing that might be used to predict performance. We simply want to identify the kind of thinking that goes into the design of an effective vector pipeline code. We do not mention any particular machine. The literature is filled with case studies.

Pipelining Arithmetic Operations

The primary reason why vector computers are fast has to do with pipelining. The concept of pipelining is best understood by making an analogy to assembly line production. Suppose the assembly of an individual automobile requires one minute at each of sixty workstations along an assembly line. If the line is well staffed and able to initiate the assembly of a new car every minute, then 1000 cars can be produced from scratch in about $1000 + 60 = 1060$minutes. For a work order of this size the line has an effective "vector speed" of $1000/1060$ automobiles per minute. On the other hand, if the assembly line is understaffed and a new assembly can be initiated just once an hour, then 1000 hours are required to produce 1000 cars. In this case the line has an effective "scalar speed" of $1/60$th automobile per minute.

So it is with a pipelined vector operation such as the vector add $z = x + y$. The scalar operations $z_i = x_i + y_i$ are the cars. The number of elements is the size of the work order.

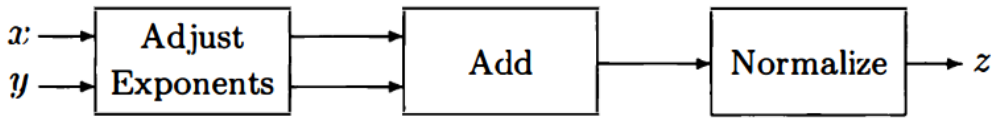
Pipelining Arithmetic Operations (Contd...)

If the start-to-finish time required for each z_i is τ, then a pipelined, length n vector add could be completed in time much less than $n\tau$. This gives vector speed. Without the pipelining, the vector computation would proceed at a scalar rate and would approximately require time $n\tau$ for completion.

Let us see how a sequence of floating point operations can be pipelined. Floating point operations usually require several cycles to complete.

Pipelining Arithmetic Operations (Contd...)

For example, a 3-cycle addition of two scalars x and y may proceed as in the following figure. To visualize the operation, continue with the above metaphor

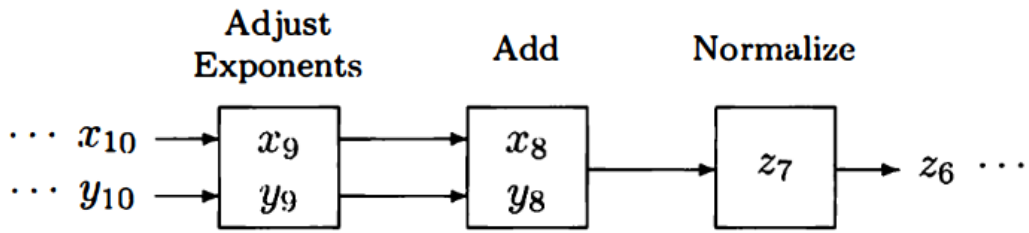


A 3-Cycle adder

and think of the addition unit as an assembly line with three "work stations".

The input scalars x and y proceed along the assembly line spending one cycle at each of three stations. The sum z emerges after three cycles.

Pipelining Arithmetic Operations (Contd...)



Pipelined addition

Note that when a single, "free standing" addition is performed, only one of the three stations is active during the computation.

Now consider a vector addition $z = x + y$. With pipelining, the x and y vectors are streamed through the addition unit. Once the pipeline is filled and steady state reached, *a* z_i is produced every cycle. In the above figure, we depict what the pipeline might look like once this steady state is achieved. In this case, vector speed is about three times scalar speed because the time for an individual add is three cycles.

Vector Operations

A vector pipeline computer comes with a repertoire of vector instructions, such as vector add, vector multiply, vector scale, dot product, and saxpy. We assume for clarity that these operations take place in vector registers. Vectors travel between the registers and memory by means of vector load and vector store instructions.

An important attribute of a vector processor is the length of its vector registers which we designate by v_L. A length-n vector operation must be broken down into subvector operations of length v_L or less.

Vector Operations (Contd...)

Here is how such a partitioning might be managed in the case of a vector addition $z = x + y$ where x and y are n-vectors:

$first = 1$
while $first \leq n$
  $last = \min n, first + v_L - 1$
  Vector load $x(first : last)$.
  Vector load $y(first : last)$.
  Vector add: $z(first : last) = x(first : last) + y(first : last)$.
  Vector store $z(first : last)$.
  $first = last + 1$
end

A reasonable compiler for a vector computer would automatically generate these vector instructions from a programmer specified $z = x + y$ command.

The Vector Length Issue

Suppose the pipeline for the vector operation *op* takes τ_{op} cycles to "set up." Assume that one component of the result is obtained per cycle once the pipeline is filled. The time required to perform an *n*-dimensional *op* is then given by

$$T_{op}(n) = (\tau_{op} + n)\mu \qquad n \leq v_L$$

where μ is the cycle time and v_L is the length of the vector hardware.

If the vectors to be combined are longer than the vector hardware length, then as we have seen the overall vector operation must be broken down into hardware-manageable chunks.

The Vector Length Issue

Thus, if

$$n = n_1 v_L + n_0 \qquad 0 \leq n_0 < v_L,$$

then we assume that

$$T_{op}(n) = \begin{cases} n_1(\tau_{op} + v_L)\mu & n_0 = 0 \\ (n_1(\tau_{op} + v_L) + \tau_{op} + n_0)\mu & n_0 \neq 0 \end{cases}$$

specifies the overall time required to perform a length-n *op*. This simplifies to

$$T_{op}(n) = (n + \tau_{op}\text{ceil}(n/v_L))\mu$$

where ceil(α) is the smallest integer such that $\alpha \leq$ ceil(α).

The Vector Length Issue (Contd...)

If ρ flops per component are involved, then the effective rate of computation for general n is given by

$$R_{op}(n) = \frac{\rho n}{T_{op}(n)} = \frac{\rho}{\mu} \frac{1}{1 + \frac{\tau_{op}}{n} \text{ceil}\left(\frac{n}{v_L}\right)}.$$

(If μ is in seconds, then R_{op} is in flops per second.) The asymptotic rate of performance is given by

$$\lim_{n \to \infty} R_{op}(n) = \frac{1}{1 + \frac{\tau_{op}}{v_L}} \frac{\rho}{\mu}.$$

The Vector Length Issue (Contd...)

As a way of assessing how serious the start-up overhead is for a vector operation, Hockney and Jesshope (1988) define the quantity $n_{1/2}$ to be the smallest n for which half of peak performance is achieved, i.e.,

$$\frac{\rho n_{1/2}}{T_{op}(n_{1/2})} = \frac{1}{2}\frac{\rho}{\mu}.$$

Machines that have big $n_{1/2}$ factors do not perform well on short vector operations.

The Vector Length Issue (Contd...)

Let us see what the above performance model says about the design of the matrix multiply update $C = AB + C$ where $A \in \mathbb{R}^{m \times p}$, $B \in \mathbb{R}^{p \times n}$, and $C \in \mathbb{R}^{m \times n}$. Recall that there are six possible versions of the conventional algorithm and they correspond to the six possible loop orderings of

```
for i=1:m
    for j=1:n
        for k=1:p
            C(i,j) = A(i,k)B(k,j) + C(i,j)
        end
    end
end
```

The Vector Length Issue (Contd...)

This is the *ijk* variant and its innermost loop oversees a length-p dot product. Thus, our performance model predicts that

$$T_{ijk} = mnp + mn \cdot \text{ceil}(p/v_L)\tau_{dot}$$

cycles are required. A similar analysis for each of the other variants leads to the following table:

Variant	Cycles
ijk	$mnp + mn \cdot \tau_{dot}(p/v_L)$
jik	$mnp + mn \cdot \tau_{dot}(p/v_L)$
ikj	$mnp + mp \cdot \tau_{sax}(n/v_L)$
jki	$mnp + np \cdot \tau_{sax}(m/v_L)$
kij	$mnp + mp \cdot \tau_{sax}(n/v_L)$
kji	$mnp + np \cdot \tau_{sax}(m/v_L)$

The Vector Length Issue (Contd...)

We make a few observations based upon some elementary integer arithmetic manipulation.

Assume that τ_{sax} and τ_{dot} are roughly equal. If m, n, and p are all less than v_L, then the most efficient variants will have the longest inner loops.

If m, n, and p are much bigger than v_L, then the distinction between the six options is small.

The "layout" of a vector operand in memory often has a bearing on execution speed. The key factor is stride. The stride of a stored floating point vector is the distance (in logical memory locations) between the vector's components.

The Stride Issue

Accessing a row in a two-dimensional Fortran array is not a unit stride operation because arrays are stored by column. In C, it is just the opposite as matrices are stored by row. Nonunit stride vector operations may interfere with the pipelining capability of a computer degrading performance.

To clarify the stride issue we consider how the six variants of matrix multiplication "pull up" data from the A, B, and C matrices in the inner loop. This is where the vector calculation occurs (dot product or saxpy) and there are three possibilities:

The Stride Issue (Contd...)

jki or *kji*:

for $i = 1 : m$
$C(i,j) = C(i,j) + A(i,k)B(k,j)$
end

ikj or *kij*:

for $j = 1 : n$
$C(i,j) = C(i,j) + A(i,k)B(k,j)$
end

ijk or *jik*:

for $k = 1 : p$
$C(i,j) = C(i,j) + A(i,k)B(k,j)$
end

The Stride Issue (Contd...)

Here is a table that specifies the A, B, and C strides associated with each of these possibilities:

Variant	A Stride	B Stride	C Stride
jki or *kji*	Unit	0	Unit
ikj or *kij*	0	Non-Unit	Non-Unit
ijk or *jik*	Non-Unit	Unit	0

Storage in column-major order is assumed. A stride of zero means that only a single array element is accessed in the inner loop. From the stride point of view, it is clear that we should favor the *jki* and *kji* variants. This may not coincide with a preference that is based on vector length considerations. Dilemmas of this type are typical in high performance computing. One goal (maximize vector length) can conflict with another (impose unit stride).

The Stride Issue (Contd...)

Sometimes a vector stride/vector length conflict can s be resolved through the intelligent choice of data structures. Consider the gaxpy $y = Ax + y$ where $A \in \mathbb{R}^{n \times n}$ is symmetric. Assume that $n \leq v_L$ for simplicity. If A is stored conventionally and Algorithm 1.1.4 is used, then the central computation entails n, unit stride saxpy's each having length n:

for j=1:n
$y = A(:, j)x(j) + y$
end

Our simple execution model tells us that

$$T_1 = n(\tau_{sax} + n)$$

cycles are required.

The Stride Issue (Contd...)

We introduced the lower triangular storage scheme for symmetric matrices and obtained this version of the gaxpy:

for $j = 1 : n$
  for $i = 1 : j - 1$
    $y(i) = A.vec((i-1)n - i(i-1)/2 + j)x(j) + y(i)$
  end
  for $i = j : n$
    $y(i) = A.vec((j-1)n - j(j-1)/2 + i)x(j) + y(i)$
  end
end

The Stride Issue (Contd...)

Notice that the first i-loop does not define a unit stride saxpy.

If we assume that a length n, nonunit stride saxpy is equivalent to n unit-length saxpys (a worst case scenario), then this implementation involves

$$T_2 = n\left(\frac{n}{2}\tau_{sax} + n\right)$$

cycles.

The Stride Issue (Contd...)

We developed the store-by-diagonal version:

for i=1:n
  $y(i) = A.diag(i)x(i) + y(i)$
end
for $k = 1 : n - 1$
  $t = nk - k(k-1)/2$
  $\{y = D(A,k)x + y\}$
  for $i = 1 : n - k$
    $y(i) = A.diag(i+t)x(i+k) + y(i)$
  end
  $\{y = D(A,k)^T x + y\}$
  for $i = 1 : n - k$
    $y(i+k) = A.diag(i+t)x(i) + y(i+k)$
  end
end

The Stride Issue (Contd...)

In this case both inner loops define a unit stride vector multiply (vm) and our model of execution predicts

$$T_3 = n(2\tau_{vm} + n)$$

cycles.

The example shows how the choice of data structure can effect the stride attributes of an algorithm. Store by diagonal seems attractive because it represents the matrix compactly and has unit stride.

However, a careful which-is-best analysis would depend upon the values of τ_{sax} and τ_{vm} and the precise penalties for nonunit stride computation and excess storage.

The complexity of the situation would call for careful benchmarking.

Thinking About Data Motion

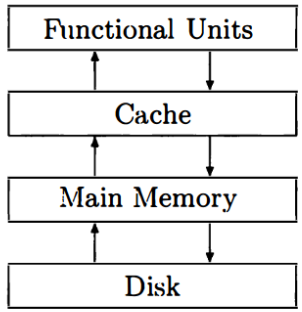
Another important attribute of a matrix algorithm concerns the actual volume of data that has to be moved around during execution. Matrices sit in memory but the computations that involve their entries take place in functional units.

The control of memory traffic is crucial to performance in many computers.

To continue with the factory metaphor used at the beginning of this section: Can we keep the superfast arithmetic units busy with enough deliveries of matrix data and can we ship the results back to memory fast enough to avoid backlog?

Thinking About Data Motion (Contd...)

The following figure depicts the typical situation in an advanced uniprocessor environment. Details vary from machine



A memory hierarchy

to machine, but two "axioms" prevail :

- Each level in the hierarchy has a limited capacity and for economic reasons this capacity is usually smaller as we ascend the hierarchy.
- There is a cost, sometimes relatively great, associated with the moving of data between two levels in the hierarchy.

The design of an efficient matrix algorithm requires careful thinking about the flow of data in between the various levels of storage. The vector touch and data re-use issues are important in this regard.

The Vector Touch Issue

In many advanced computers, data is moved around in chunks, e.g., vectors.

The time required to read or write a vector to memory is comparable to the time required to engage the vector in a dot product or saxpy.

Thus, the number of vector touches associated with a matrix code is a very important statistic. By a "vector touch" we mean either a vector load or store.

The Vector Touch Issue

Let's count the number of vector touches associated with an m-by-n outer product. Assume that $m = m_1 v_L$ and $n = n_1 v_L$ where v_L is the vector hardware length.

In this environment, the outer product update $A = A + xy^T$ would be arranged as follows:

for $\alpha = 1 : m_1$
  $i = (\alpha - 1)v_L + 1 : \alpha v_L$
  for $\beta = 1 : n_1$
    $j = (\beta - 1)v_L + 1 : \beta v_L$
    $A(i,j) = A(i,j) + x(i)y(j)^T$
  end
end

The Vector Touch Issue (Contd...)

Each column of the submatrix $A(i, j)$ must be loaded, updated, and then stored. Not forgetting to account for the vector touches associated with x and y we see that approximately

$$\sum_{\alpha=1}^{m_1} \left(1 + \sum_{\beta=1}^{n_1} (1 + 2v_L) \right) \approx 2m_1 n$$

vector touches are required. (Low order terms do not contribute to the analysis.)

The Vector Touch Issue (Contd...)

Now consider the gaxpy update $y = Ax + y$ where $y \in \mathbb{R}^m$, $x \in \mathbb{R}^n$ and $A \in \mathbb{R}^{m \times n}$. Breaking this computation down into segments of length v_L gives

for $\alpha = 1 : m_1$
  $i = (\alpha - 1)v_L + 1 : \alpha v_L$
  for $\beta : 1 : n_1$
    $j = (\beta - 1)v_L + 1 : \beta v_L$
    $y(i) = y(i) + A(i, j)x(j)$
  end
end

The Vector Touch Issue (Contd...)

Again, each column of submatrix $A(i, j)$ must be read but the only writing to memory involves subvectors of y. Thus, the number of vector touches for an m-by-n gaxpy is

$$\sum_{\alpha=1}^{m_1} \left(2 + \sum_{\beta=1}^{n_1} (1 + v_L) \right) \approx m_1 n.$$

This is half the number required by an identically-sized the outer product. Thus, if a computation can be arranged in terms of either outer products or gaxpys, then the former is preferable from the vector touch standpoint.

Blocking and Re-Use

A cache is a small high-speed memory situated in between the functional units and main memory. Cache utilization colors performance because it has a direct bearing upon how data flows in between the functional units and the lower levels of memory.

To illustrate this we consider the computation of the matrix multiply update $C = AB + C$ where $A, B, C \in \mathbb{R}^{n \times n}$ reside in main memory[1].

All data must pass through the cache on its way to the functional units where the floating point computations are carried out. If the cache is small and n is big, then the update must be broken down into smaller parts so that the cache can "gracefully" process the flow of data.

[1] The discussion which follows would also apply if the matrices were on a disk and needed to be brought into main memory.

Blocking and Re-Use (Contd...)

One strategy is to block the B and C matrices,

$$B = \underset{\ell}{[B_1} \quad , \ldots , \quad \underset{\ell}{B_N]} \qquad C = \underset{\ell}{[C_1} \quad , \ldots , \quad \underset{\ell}{C_N]}$$

where we assume that $n = \ell N$. From the expansion

$$C_\alpha = AB_\alpha + C_\alpha = \sum_{k=1}^{n} A(:,k)B_\alpha(k,:) + C_\alpha$$

we obtain the following computational framework:

for $\alpha = 1 : N$
  Load B_α and C_α into cache.
  for k=1:n
    Load $A(:,k)$ into cache and update C_α:
    $C_\alpha = A(:,k)B_\alpha(k,:) + C_\alpha$
  end
  Store C_α in main memory.
end

Blocking and Re-Use (Contd...)

Note that if M is the cache size measured in floating point words, then we must have

$$2n\ell + n \leq M. \tag{1}$$

Let Γ_1 be the number of floating point numbers that flow (in either direction) between cache and main memory.

Note that every entry in B is loaded into cache once, every entry in C is loaded into cache once and stored back in main memory once, and every entry in A is loaded into cache $N = n/\ell$ times. It follows that

$$\Gamma_1 = 3n^2 + \frac{n^3}{\ell}.$$

Blocking and Re-Use (Contd...)

In the interest of keeping data motion to a minimum, we choose ℓ to be as large as possible subject to the constraint (1.4.1). We therefore set

$$\ell \approx \frac{1}{2}\left(\frac{M}{n} - 1\right)$$

obtaining

$$\Gamma_1 \approx 3n^2 + \frac{2n^4}{M - n}.$$

(We use "$\approx$" to emphasize the approximate nature of our analysis.) If cache is large enough to house the entire B and C matrices with room left over for a column of A, then $\ell = n$ and $\Gamma_1 = 4n^2$. At the other extreme, if we can just fit three columns in cache, then $\ell = 1$ and $\Gamma_1 \approx n^3$.

Blocking and Re-Use (Contd...)

Now let us regard $A = (A_{\alpha\beta})$, $B = (B_{\alpha\beta})$, and $C = (C_{\alpha\beta})$ as N-by-N block matrices with uniform block size $\ell = n/N$. With this blocking the computation of

$$C_{\alpha\beta} = \sum_{\gamma=1}^{N} A_{\alpha\gamma} B_{\gamma\beta} \qquad \alpha = 1 : N, \beta = 1 : N$$

can be arranged as follows:

for $\alpha = 1 : N$
  for $\beta = 1 : N$
    Load $C_{\alpha\beta}$ into cache.
    for $\gamma = 1 : N$
      Load $A_{\alpha\gamma}$ and $B_{\gamma\beta}$ into cache.
      $C_{\alpha\beta} = C_{\alpha\beta} + A_{\alpha\gamma} B_{\gamma\beta}$
    end
    Store $C_{\alpha\beta}$ as in main memory.
  end
end

Blocking and Re-Use (Contd...)

In this case the main memory/cache traffic sums to

$$\Gamma_2 = 2n^2 + \frac{2n^3}{\ell}$$

because each entry in A and B is loaded $N = n/\ell$ times and each entry in C is loaded once and stored once. We can minimize this by choosing ℓ to be as large as possible subject to the constraint that three blocks fit in cache, i.e.,

$$3\ell^2 \leq M$$

Setting $\ell \approx \sqrt{M/3}$ gives

$$\Gamma_2 \approx 2n^2 + 2n^3\sqrt{\frac{3}{M}}.$$

Blocking and Re-Use (Contd...)

A manipulation shows that

$$\frac{\Gamma_1}{\Gamma_2} \approx \frac{3n^2 + \frac{2n^4}{M-n}}{2n^2 + 2n^3\sqrt{\frac{3}{M}}} \geq \frac{3 + 2\frac{n^2}{M}}{2 + 2\sqrt{3}\sqrt{\frac{n^2}{M}}}.$$

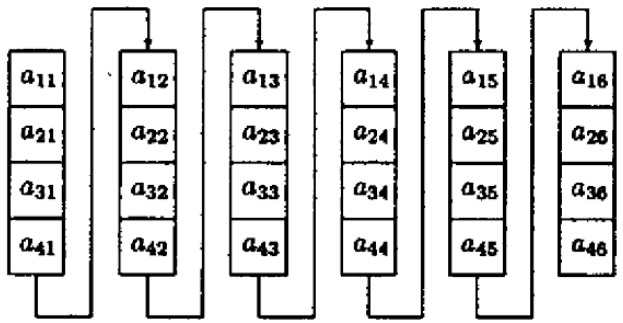
The key quantity here is n^2/M, the ratio of matrix size (in floating point words) to cache size. As this ratio grows the we find that

$$\frac{\Gamma_1}{\Gamma_2} \approx \frac{n}{\sqrt{3M}}$$

showing that the second blocking strategy is superior from the standpoint of data motion to and from the cache. The fundamental conclusion to be reached from all of this is that blocking effects data motion.

Block Matrix Data Structures

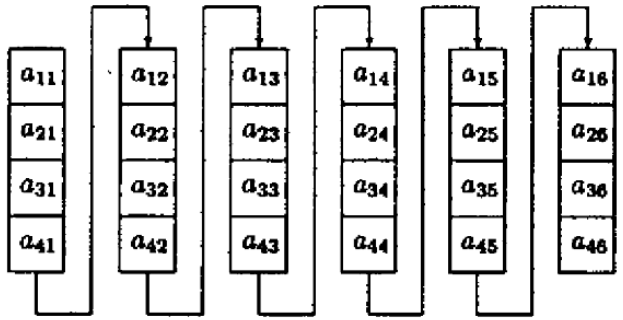
We conclude this section with a discussion about block data structures. A programming language that supports two-dimensional arrays must have a convention for storing such a structure in memory. For example, Fortran stores two-dimensional arrays in column major order. This means that the entries within a column are contiguous in memory. Thus, if 24 storage locations are allocated for $A \in \mathbb{R}^{4\times 6}$, then in traditional store-by-column format the matrix entries are "lined up" in memory as depicted in the following figure. In other words, if $A \in \mathbb{R}^{m\times n}$ is stored in $v(1:mn)$, then we identify $A(i,j)$ with $v((j-1)m+i)$.



Store by Column (4-by-6 case)

Block Matrix Data Structures

For algorithms that access matrix data by column this is a good arrangement since the column entries are contiguous in memory.



Store by Column (4-by-6 case)

In certain block matrix algorithms it is sometimes useful to store matrices by blocks rather than by column. Suppose, for example, that the matrix A above is a 2-by-3 block matrix with 2-by-2 blocks. In a store-by-column block scheme with store-by-column within each block, the 24 entries are arranged in memory as shown in the above figure. This data structure can be attractive for block algorithms because the entries within a given block are contiguous in memory.

Problems

1. Consider the matrix product $D = ABC$ where $A \in \mathbb{R}^{m \times r}$, $B \in \mathbb{R}^{r \times n}$ and $C \in \mathbb{R}^{n \times q}$. Assume that all the matrices are stored by column and that the time required to execute a unit-stride saxpy operation of length k is of the form $t(k) = (L + k)\mu$ where L is a constant and μ is the cycle time. Based on this model, when is it more economical to compute D as $D = (AB)C$ instead of as $D = A(BC)$? Assume that all matrix multiplies are done using the *jki*, (gaxpy) algorithm.
2. What is the total time spent in *jki* variant on the saxpy operations assuming that all the matrices are stored by column and that the time required to execute a unit-stride saxpy operation of length k is of the form $t(k) = (L + k)\mu$ where L is a constant and μ is the cycle time? Specialize the algorithm so that it efficiently handles the case when A and B are n-by-n and upper triangular. Does it follow that the triangular implementation is six times faster as the flop count suggests?

Problems (Contd...)

3. Give an algorithm for computing $C = A^T BA$ where A and B are n-by-n and B is symmetric. Arrays should be accessed in unit stride fashion within all innermost loops.
4. Suppose $A \in \mathbb{R}^{m \times n}$ is stored by column in $A.col(1 : mn)$. Assume that $m = \ell_1 M$ and $n = \ell_2 N$ and that we regard A as an M-by-N block matrix with ℓ_1-by-ℓ_2 blocks. Given i, j, α, and β that satisfy $1 \leq i \leq \ell_1$, $1 \leq j \leq \ell_2$, $1 \leq \alpha \leq M$, and $1 \leq \beta \leq N$ determine k so that $A.col(k)$ houses the (i, j) entry of $A_{\alpha\beta}$. Give an algorithm that overwrites $A.col$ with A stored by block an in Figure 1.4.5. How big of a work array is required?

Reference Books

1. Gene H. Golub and Charles F. Van Loan, Matrix Computations, 3rd Edition, Hindustan book agency, 2007.
2. A.R. Gourlay and G.A. Watson, Computational methods for matrix eigen problems, John Wiley & Sons, New York, 1973.
3. W.W. Hager, Applied numerical algebra, Prentice-Hall, Englewood Cliffs, N.J, 1988.
4. D.S. Watkins, Fundamentals of matrix computations, John Wiley and sons, N.Y, 1991.
5. C.F. Van Loan, Introduction to scientific computing: A Matrix vector approach using Matlab, Prentice-Hall, Upper Saddle River, N.J, 1997.