

Maximizing Code Performance by Optimizing Memory Access

BY STUART MCGARRITY

Most MATLAB® users want their code to be fast, especially when it is processing very large data sets. Because memory performance has not increased at the same rate as CPU performance, code today is often “memory-bound,” its overall performance limited by the time it takes to access memory.

Fortunately, with a little knowledge of how MATLAB stores and accesses data, you can avoid inefficient memory usage and improve the speed of your code.

This article describes three ways to improve the performance of memory-bound code:

- Preallocate arrays before accessing them within loops
- Store and access data in columns
- Avoid creating unnecessary variables

In each case we will compare the execution speed of a code segment before and after applying the technique. To ensure the best performance, the code segments are all timed in function M-files, not script M-files. As memory performance is machine-dependent, the performance tests were carried out on two different machines¹.

Preallocate Arrays Before Accessing them Within Loops

When creating or repeatedly modifying arrays within loops, always allocate the arrays beforehand. Of all three techniques, this familiar one can give the biggest performance improvement.

The code segments in Figure 1 use a `for` loop to calculate a sequence of numbers based on the equation, $x(k) = 1.05 * x(k-1)$, equal to the annual balance of a bank account earning a 5% interest rate.

Why Code Segment 2 is Faster

The MATLAB language does not require you to declare the types and sizes of variables before you use them. As a result, you can increase the size of an array merely by indexing into

Code segment 1

```
N = 10e3;

x(1)=1000;
for k=2:N
    x(k)=1.05*x(k-1);
end
Machine A = 0.1409 sec
Machine B = 0.1281 sec
```

Code segment 2

```
N = 10e3;
x=zeros(N,1); %Preallocate
locate
x(1)=1000;
for k=2:N
    x(k)=1.05*x(k-1);
end
Machine A = 0.00024 sec
Machine B = 0.00027 sec
```

Figure 1. Preallocating arrays. Code segment 2 executes in 99.8% less time (580 times faster) than segment 1 on machine A, and in 99.7% less time (475 times faster) than segment 1 on machine B.

it at a point larger than the current size. This approach is convenient for quick prototyping of code, but each time you use it, MATLAB must allocate memory for a new larger array and then copy the existing data into it. Code that repeats this procedure several times, as in a loop, is slow and inefficient.

In one step, code segment 2 preallocates the entire array `x` to the largest size that it needs to be. No more memory allocation is then required during the execution of the code. If the M-Lint code checker finds an opportunity to preallocate, it issues a warning.

¹ Machine A is a Lenovo T60 ThinkPad, 1.83GHz Intel Core Duo T2400 processor, 2MB L2 cache, 2GB RAM, 32-bit Windows XP.

Machine B is a Dual 2.1GHz AMD Opteron 248 processor machine, 1MB L2 Cache, 1GB RAM, 64-bit Linux.

Store and Access Data in Columns

When processing 2-D or N-D arrays, access your data in columns and store it so that it is easily accessible by columns.

The code segments in Figure 2 copy the positive values of the 2-D array x to a new array, y .

Code segment 1

```
N = 2e3;
x = randn(N);
y = zeros(N);

for r = 1:N % Row
    for c = 1:N % Column
        if x(r, c) >= 0
            y(r,c)=x(r, c);
        end
    end
end

Machine A = 0.1605 sec
Machine B = 0.2090 sec
```

Code segment 2

```
N = 2e3;
x = randn(N);
y = zeros(N);

for c = 1:N % Column
    for r = 1:N % Row
        if x(r, c) >= 0
            y(r,c)=x(r, c);
        end
    end
end

Machine A = 0.1083 sec
Machine B = 0.0946 sec
```

Figure 2. Storing and accessing data in columns. Code segment 2 executes in 33% less time than segment 1 on machine A, and in 55% less time than segment 1 on machine B.

Why Code Segment 2 is Faster

Modern CPUs use a fast cache to reduce the average time taken to access main memory. Your code achieves maximum cache efficiency when it traverses monotonically increasing memory locations. Because MATLAB stores matrix columns in monotonically increasing memory locations, processing data column-wise results in maximum cache efficiency.

Code segment 2 is faster because it traverses the elements of the 2-D array by going down the columns in the inner loop. If the algorithm permits, you can also maximize cache efficiency by using the single index method, $x(k)$, instead of $x(r,c)$.

Cache effects are data-size- and machine-dependent, making it difficult to predict performance. For example, column size affects memory access time when you operate on row vectors because it determines the stride, or step through memory. Figure 3 shows the time taken per element on the two machines to perform element-wise multiplication of a 10k element row vector by a matrix of varying column size. The operation is most efficient for a column size equal to 1, a 1-D vector. For other column sizes, code can run up to an order of magnitude more slowly.

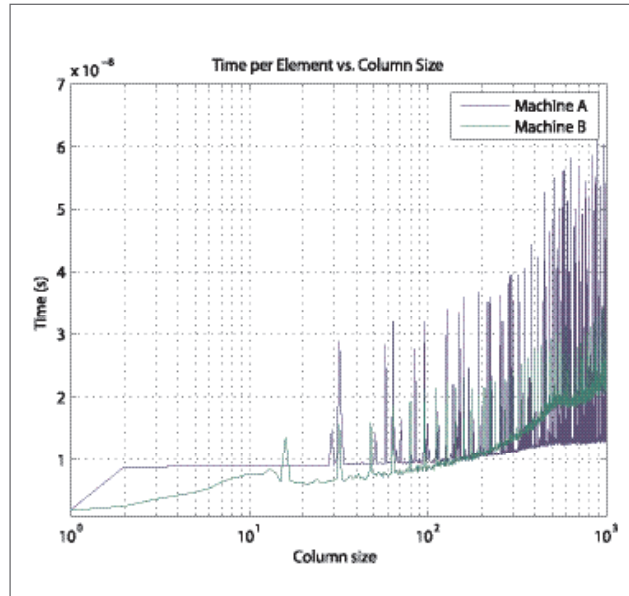


Figure 3. Time vs. column size for multiplying a 10k row vector.

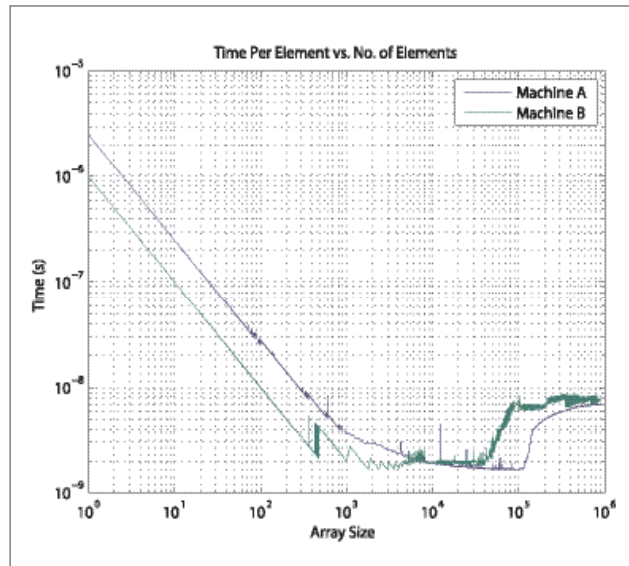


Figure 4. Time per element vs. number of elements for elementwise multiplication.

Even array size affects efficiency. Figure 4 shows the time taken per element on the two machines to perform element-wise multiplication on a 1-D array of varying size.

Depending on the cache and memory characteristics of the computer, processing large arrays is orders of magnitude more efficient than processing small arrays.

In some cases, the time taken to process arrays that differ slightly in size can vary widely, and optimizing code for a problem size on one machine may not yield the same results on another.

Avoid Creating Unnecessary Variables

When creating new variables or variables that are functions of existing data sets, ensure that they are essential to your algorithm, particularly if your data set is large.

The code segments in Figure 5a and 5b multiply the matrix x , by a scalar via an operator in segments 1 and 2 (Figure 5a), and via an M-function in segments 3 and 4 (Figure 5b).

Code segment 1	Code segment 2
<pre>N = 3e3; x = randn(N); y=x*1.2; Machine A = 0.0847 sec Machine B = 0.1115 sec</pre>	<pre>N = 3e3; x = randn(N); x=x*1.2; % In place Machine A = 0.0506 sec Machine B = 0.00461 sec</pre>

Figure 5a. Avoiding unnecessary variable creation by calling operators in-place. Code segment 2 executes in 40% less time than code segment 1 on machine A and in 96% less time on machine B.

Code segment 3	Code segment 4
<pre>y=myfun(x); function y=myfun(x) y=1.2*x; Machine A = 0.0858 sec Machine B = 0.1019 sec</pre>	<pre>x=myfun_ip(x); function x=myfun_ip(x) x=1.2*x; % In place Machine A = 0.0508 sec Machine B = 0.0421sec</pre>

Figure 5b. Avoiding unnecessary variable creation by calling Mfunctions in-place. Code segment 4 executes in 40% less time than code segment 3 on machine A and in 59% less time on machine B.

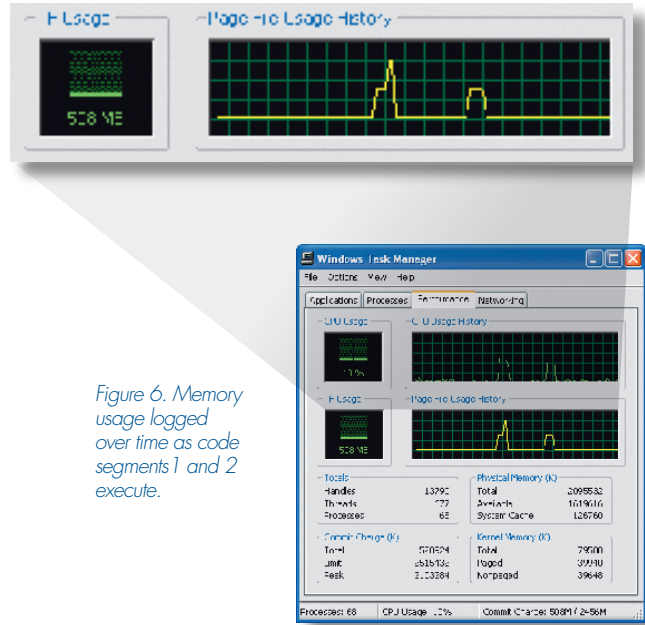


Figure 6. Memory usage logged over time as code segments 1 and 2 execute.

Why Code Segments 2 and 4 are Faster

In MATLAB, it is easy to inadvertently make copies of your data. For a large data set this uses a lot of memory, and the allocation and copying of memory can itself be time-consuming.

Code segments 2 and 4 are faster because they use in-place operations to avoid creating new variables that are modified versions of the existing ones. This capability is available with element-wise operators (such as `.*`, `+`), some MATLAB functions (such as `sin` and `sqrt`), and your own M-functions. To create a function M-file that can be called in-place, you must ensure that the output argument matches one of the input arguments. The ability to call functions in-place (introduced in MATLAB 7.3) is available only when the function itself is called from a function M-file and not from a script.

As mentioned earlier, in-place operations result in reduced memory consumption as well as reduced computation time. Figure 6 shows the memory usage in the Windows Task Manager during the execution of code segments 1 and 2 processing a 100-million element array (stepped through manually with the debugger to make memory changes more obvious).

Reduced memory usage is particularly important when dealing with data sets that are larger than the available RAM. Operating systems supplement physical RAM memory with a swap or page file stored on disk. Accessing data stored on disk is orders of magnitude slower than accessing data stored in RAM, however. For maximum performance, keep memory usage low enough to prevent your system from running out of RAM when MATLAB is executing. ◀◀

Memory Performance Glossary

Cache. The fast memory on a processor or on a motherboard, used to improve the performance of main memory by temporarily storing data.

RAM. Main physical memory, usually in the range of 1GB to 4GB on 32-bit operating systems.

Memory-bound code. Code whose performance is limited by memory speed, not by CPU speed.

Stride. Memory step size taken by the code when accessing matrix rows.

Vectorized code. Code that operates on arrays and does not use `for` loops.

For More Information

- Code Segments Download www.mathworks.com/res/code_segments
- Memory Management Guide www.mathworks.com/res/memory

Resources

VISIT

www.mathworks.com

TECHNICAL SUPPORT

www.mathworks.com/support

ONLINE USER COMMUNITY

www.mathworks.com/matlabcentral

DEMOS

www.mathworks.com/products/demos

TRAINING SERVICES

www.mathworks.com/training

THIRD-PARTY PRODUCTS

www.mathworks.com/connections

WORLDWIDE CONTACTS

www.mathworks.com/contact

E-MAIL

info@mathworks.com



©1994-2007 by The MathWorks, Inc.
MATLAB, Simulink, Stateflow, Handle Graphics, Real-Time Workshop, SimBiology, SimHydraulics, and xPC TargetBox are registered trademarks and SimEvents is a trademark of The MathWorks, Inc. Other product or brand names are trademarks or registered trademarks of their respective holders.