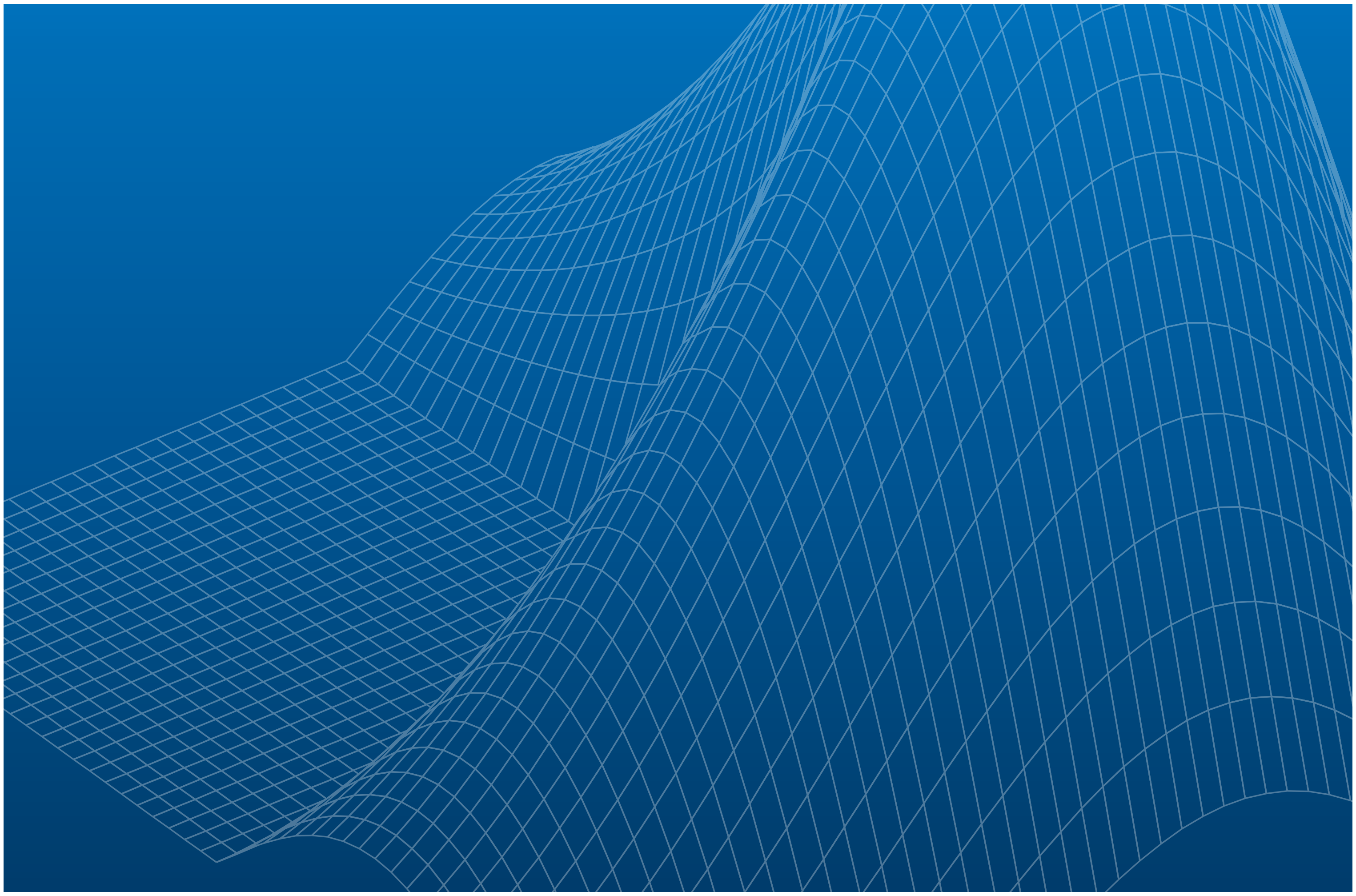


# Speeding Up MATLAB Applications

WHITE PAPER

MathWorks®

## Contents

## I. Introduction

MATLAB® is a high-level programming language that enables scientists and engineers to move quickly from idea to implementation. MATLAB provides a flexible syntax and the ability to speed application development. This flexibility and speed can sometimes lead to poor programming patterns, resulting in suboptimal performance. Fortunately, MATLAB offers tools that significantly improve application performance.

This paper shows how to diagnose performance issues, select the right tools, and make tradeoffs between ease of use, readability, and performance. It covers the following topics:

- Improving code performance with vectors and matrices
- Optimizing performance with MATLAB Profiler and MATLAB Code Analyzer
- Parallel computing

## II. Improving Code Performance with Vectors and Matrices

Optimized for operations on vectors and matrices, MATLAB stores and operates on data in patterns. MATLAB therefore operates more efficiently on chunks of data than on individual data elements. Follow these best practices to exploit the way MATLAB stores and operates on data in memory:

- Pre-allocate memory for vectors and matrices.
- Optimally access elements of large multidimensional matrices.
- Vectorize code.

You can improve program performance with relatively little programming effort by using these practices.

### 1. Preallocate Memory for Vectors and Matrices

MATLAB performs best when working with vectors and matrices of specified sizes. To make the most of this characteristic of MATLAB, follow these guidelines:

- Allocate memory to vector and matrix variables before using them in your program.
- Avoid repeatedly changing the size of vector and matrix variables in your program—for example, within loops.

A MATLAB array is represented in computer memory by a series of consecutive memory locations. When you use an array without initializing it, MATLAB must first determine the amount of memory that is required for the array data based on the information provided and then place the variable in memory in such a way that all the data is held in contiguous memory locations.

When an array is resized—for example, to accommodate more elements—MATLAB must find and allocate a new chunk of memory to hold the larger array. This is because the data elements of an array must occupy consecutive memory locations, and there is no guarantee that the memory next to the existing array data is not already in use. Once MATLAB finds and allocates memory for the new array, it copies the contents of the old array over to the new array and

```
function dynamicMatrixAlloc
tic
Lx = 10; Nx = 1000;
Ly = 10; Ny = 1000;

x = linspace(1, Lx, Nx);
y = linspace(1, Ly, Ny);

%% Dynamically expand mysurf
in each iteration
for i = 1:length(x)
  for j = 1:length(y)
    yp = y(j);
    xp = x(i);
    mysurf(i,j) =
5*cos((xp+yp)*2*pi) +
2*sin(xp*2*pi) + 2*cos(xp*2*pi);
  end
end
toc
```

```
function preallocatedMatrix
tic
Lx = 10; Nx = 1000;
Ly = 10; Ny = 1000;

x = linspace(1, Lx, Nx);
y = linspace(1, Ly, Ny);

%% Preallocate memory for
mysurf
mysurf = zeros(length(x),
length(y));

for i = 1:length(x)
  for j = 1:length(y)
    yp = y(j);
    xp = x(i);
    mysurf(i,j) =
5*cos((xp+yp)*2*pi) +
2*sin(xp*2*pi) + 2*cos(xp*2*pi);
  end
end
toc
```

Figure 1. *Left: Dynamic matrix allocation. Right: Preallocated matrix space.*

discards the old memory. Overhead in this process is incurred at each step: finding a new location for array data, copying data to the new location, and discarding the old data. In addition, frequent resizing leads to memory fragmentation because of repeated allocation and de-allocation.

While these operations are not expensive when performed infrequently on small matrices, the cumulative cost becomes significant for larger arrays, numerous arrays, or numerous iterations. In fact, a sharp increase in execution time in such cases indicates that memory allocation and reallocation overhead is an issue.

By pre-allocating memory whenever possible, you can minimize resizing as well as the associated overhead and memory fragmentation.

To better understand this concept, consider the code samples in Figures 1. In Figure 1 left, the matrix `mysurf` grows one element at a time within a double nested `for` loop. In Figure 1 right, the matrix is preallocated and initialized to zero after determining the length of vector `x` and the length of vector `y`. When `Nx` and `Ny` values are set to `1000`, the code takes approximately 4 seconds to execute without preallocation. With pre-allocation it takes about 1.7 seconds. You can run this code yourself, varying the size of the vectors to determine how much time pre-allocation can save as your matrix size increases.

## 2. Optimally Access Elements of MATLAB Matrices

Another easy way to improve the performance of your MATLAB code is to access large multi-dimensional matrices in the same order that MATLAB stores them internally.

Consider the following matrix:

$$A = \begin{bmatrix} a11 & a12 & a13 \\ a21 & a22 & a23 \\ a31 & a32 & a33 \end{bmatrix}$$

While this is a 2-D matrix, the data it holds is stored sequentially in computer memory. To convert multidimensional arrays to sequential representations, MATLAB uses *column-major format*, a technique that lays out the matrix data so that elements within a single column are adjacent to each other. Thus, in column-major format, `A` will be stored as follows:

[a11 a21 a31 a12 a22 a32 a13 a23 a33) ]

More generally, matrix elements belonging to the same column are located in consecutive locations of memory, while elements belonging to the same row of the matrix are located in nonconsecutive locations of memory. Code written to access elements sequentially executes faster than code that is not.

For example, the code in Figure 2 that accesses matrices in row-major order takes approximately 1.7 seconds to execute. In contrast, the code that accesses matrices in column-major order takes only 1.25 seconds to execute. The larger the matrix, the more pronounced the difference in execution times.

You can often achieve additional performance gains by *vectorizing*, or eliminating loops.

```
% Row-based Access                    % Column-based Access
N = 5e3;                              N = 5e3;
X = randn(N);                         X = randn(N);
Y = zeros(N);                         Y = zeros(N);

for row = 1:N % Row                   for col = 1:N % Column
  for col = 1:N % Column                for row = 1:N % Row
    if X(row,col) >= 0                    if X(row,col) >= 0
      Y(row,col) = X(row,col);              Y(row,col) = X(row,col);
    end                                   end
  end                                   end
end                                   end
```

Figure 2. *Row-based and column-based matrix access.*

## 3. Vectorize Code

Vectorized code operates on entire vectors and matrices with a single command rather than performing scalar operations within a loop. Vectorization in MATLAB provides two advantages: First, the code can use optimized linear algebra libraries and so execute faster. Second, the code is closer to its corresponding mathematical expression, and so is more readable.

Loops such as `for` and `while` introduce overhead that adversely affects the performance of matrix operations. For example, at the beginning of each iteration it is necessary to check that the iteration variable has not exceeded a particular value. `for` loops are represented in the underlying machine code as ‘jumps’ from one program location to another at the end of each iteration.

MATLAB provides techniques for operating on vectors and matrices of known sizes that are more efficient than using loops. For example, Basic Linear Algebra Subprograms (BLAS) libraries are tuned for fast operations on vectors and matrices. MATLAB uses a BLAS library as well as other optimizations to support vectorization. When you use vectorized code, MATLAB can take advantage of these optimizations.

Consider the two code samples in Figure 3. In the example on the left, the output vector is computed one element at a time. In the example on the right, the MATLAB code has been vectorized, and computes the output in a single function call. The vectorized version avoids the overhead of 1000 conditional checks and 1000 code jumps (within internal machine code) inside the `for` loop. Finally, note that intent is much more apparent in the vectorized example.

```
% A poorly constructed loop
tic
i = 0;
for t = 0.01:0.01:10
  i = i + 1;
  y(i) = sin(t);
end
toc

% Alternate, better loop
construction
tic
t = 0.01:0.01:10;
for i = 1:length(t)
  y(i) = sin(t(i));
end
toc
```

```
% Vectorized code
tic
t = 0.01:0.01:10;
y = sin(t);
toc
```

Figure 3. *Computing sine using a loop (left) and using vectorization (right).*

While vectorization usually leads to more readable code, this is not always the case. If vectorization produces complex code, then you may decide that the readability of `for` loops outweighs the performance gains afforded by vectorization.

When you are creating higher-dimensional matrices from existing vectors or matrices, your first inclination may be to use `for` loops. Often, vectorization is a better solution. For example, you can use the `meshgrid` and `repmat` functions to build appropriate matrices for use with vector operations.

Consider an algorithm that computes all possible volumes of a right circular cylinder given two vectors containing values for radius (`R`) and height (`H`). Figure 4 shows two solutions to this problem, one using a `for` loop and one using vectorized code. Note that using `meshgrid` not only makes the code more succinct and readable, it also enables the use of vectorization through element-wise operations in MATLAB. Note also that the vectorized expression `V = pi.*matR.^2.*matH` resembles linear algebra expressions found in textbooks for computing the volume of a cylinder, and is easier to understand than the `for` loop.

```
% Using a loop                      % Vectorized code
tic                                 tic
R = 1:10000;                        R = 1:10000;
H = 1:4000;                         H = 1:4000;
V = zeros(10000,4000);
for i = 1:numel(H)                  [matR, matH] = meshgrid(R,H);
  for j = 1:numel(R)                V = pi.*matR.^2.*matH;
    V(j,i) = pi * R(j).^2 * H(i);
  end                               toc
end
toc
```

Figure 4. *Code vectorization with* `meshgrid`.

In general, vectorization improves performance but incurs memory overhead. In the example described here, the matrices `matR` and `math` require memory that the unvectorized example does not. When vectorization requires too much memory, consider using the `bsxfun` function instead of `repmat` or `meshgrid`. The `bsxfun` enables you to take advantage of vectorization through *virtual* matrix expansion.

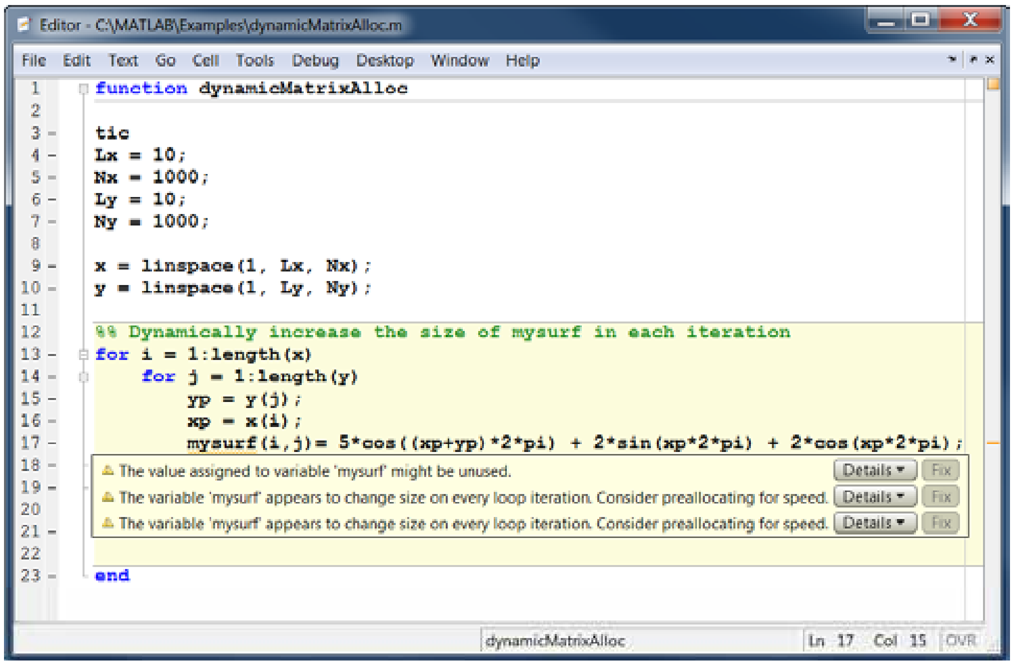
## III. Improving Code Performance with MATLAB Code Analyzer and MATLAB Profiler

Ideally, programs are developed with pre-allocation, optimal matrix access, and vectorization in mind. However, your application may include legacy code that does not take advantage of these techniques and, as a result, performs poorly on production data. Manually locating and fixing performance bottlenecks in such code can be a daunting challenge.

MATLAB provides two tools to help you with these tasks: MATLAB Code Analyzer and MATLAB Profiler.

### 1. Using MATLAB Code Analyzer to Find Errors and Inefficient Code Constructs

Code Analyzer launches when you open MATLAB code in the MATLAB Editor. It identifies easy-to-find bottlenecks for faster program execution, detects syntax errors, suggests potential improvements, and displays information in the editor window. MATLAB code is classified into one of three categories: green (no issues found), orange (potential code optimizations found), and red (code errors found). When you hover over the marked code, you are presented with information about optimizations that you can perform to improve your code; for example, Code Analyzer may identify and note a variable that can be preallocated to improve performance (Figure 5).


Editor - C:\MATLAB\Examples\dynamicMatrixAlloc.m
File Edit Text Go Cell Tools Debug Desktop Window Help
Details
Fix
dynamicMatrixAlloc
Ln 17 Col 15 OVR

```
function dynamicMatrixAlloc

tic
Lx = 10;
Nx = 1000;
Ly = 10;
Ny = 1000;

x = linspace(1, Lx, Nx);
y = linspace(1, Ly, Ny);

%% Dynamically increase the size of mysurf in each iteration
for i = 1:length(x)
    for j = 1:length(y)
        yp = y(j);
        xp = x(i);
        mysurf(i,j)= 5*cos((xp+yp)*2*pi) + 2*sin(xp*2*pi) + 2*cos(xp*2*pi);
```

The value assigned to variable 'mysurf' might be unused.
The variable 'mysurf' appears to change size on every loop iteration. Consider preallocating for speed.
The variable 'mysurf' appears to change size on every loop iteration. Consider preallocating for speed.

```
end
```

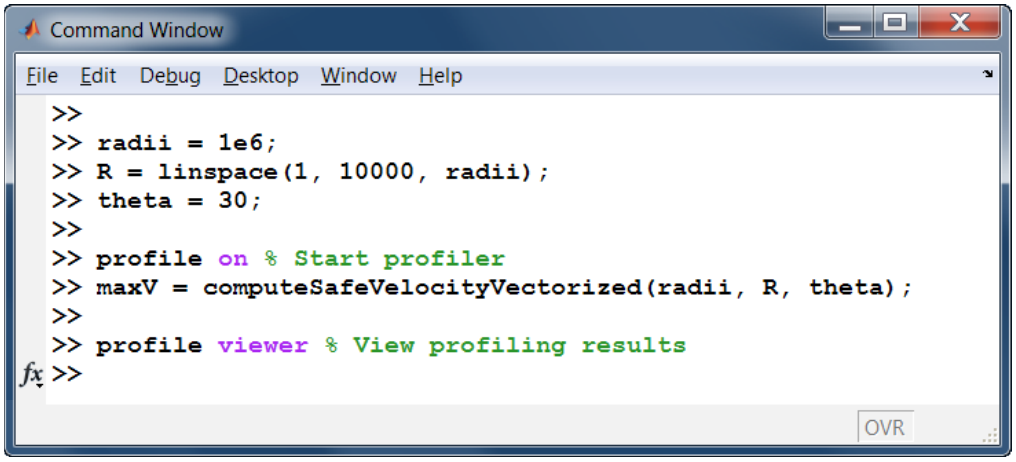
Figure 5. *Sample MATLAB Code Analyzer results.*

You can launch Code Analyzer by using the MATLAB `mlint` command at the MATLAB Command Window prompt. For the file you specify, `mlint` displays all Code Analyzer warnings and errors in the Command Window. Each warning and error is accompanied by a hyperlink to the relevant line of code.

## 2. Using MATLAB Profiler to Identify Code Bottlenecks at Run Time

MATLAB Code Analyzer performs static code analysis, and is therefore best used while you are developing your program. However, many programs do not reveal performance bottlenecks until the program is executed or until a large data set is processed. You can detect and locate these bottlenecks by running the program with MATLAB Profiler. MATLAB Profiler identifies the parts of your program that take the most execution time so that you know where to target further code optimizations. You can launch MATLAB Profiler from the MATLAB Editor or from the MATLAB Command Window.

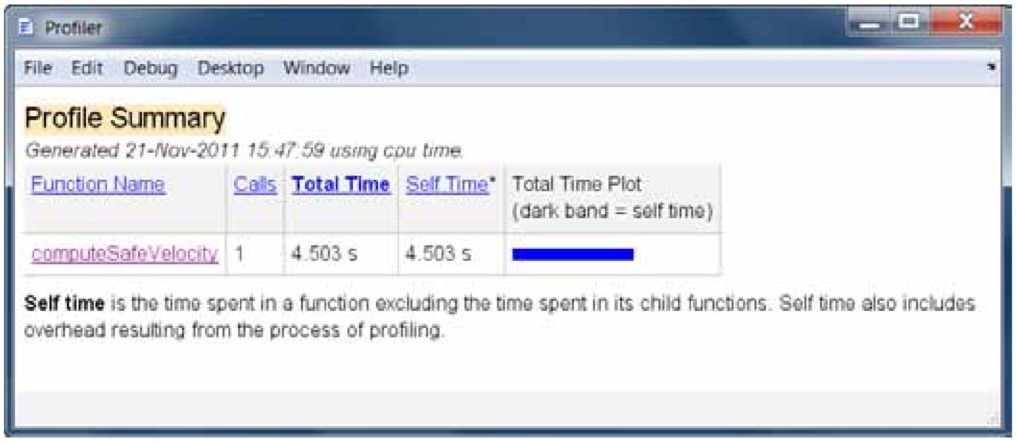
The following example profiles a program named `computeSafeVelocity`. To begin, execute the commands shown in Figure 6.



```
>>
>> radii = 1e6;
>> R = linspace(1, 10000, radii);
>> theta = 30;
>>
>> profile on % Start profiler
>> maxV = computeSafeVelocityVectorized(radii, R, theta);
>>
>> profile viewer % View profiling results
fx >>
```

Figure 6. *Steps for profiling a program in MATLAB.*

The `profile viewer` command displays the performance profile (Figure 7).



Profile Summary

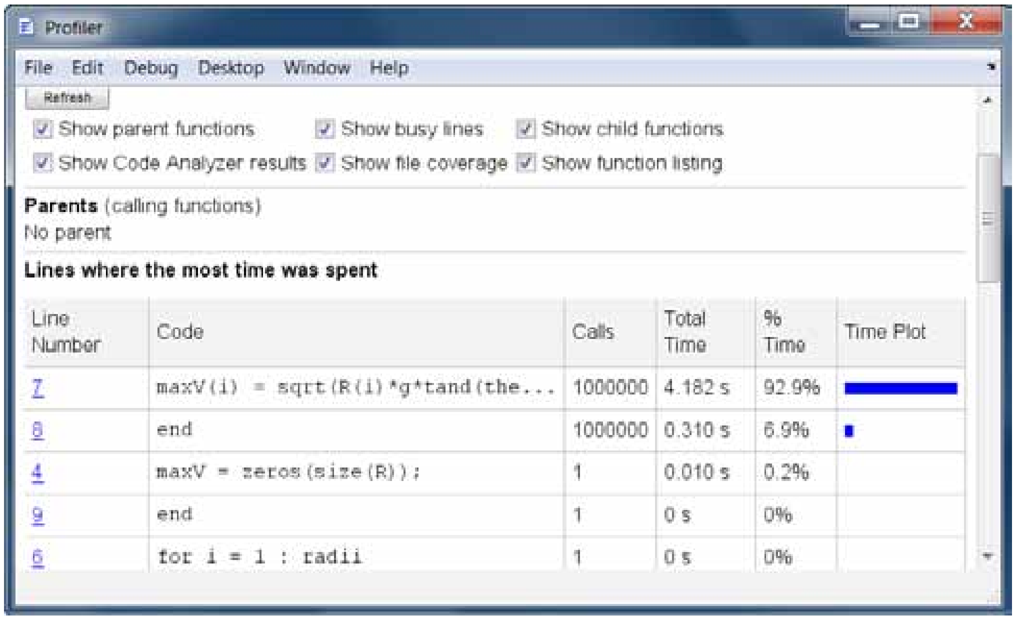
Generated 21-Nov-2011 15:47:59 using cpu time.

| Function Name | Calls | Total Time | Self Time* | Total Time Plot (dark band = self time) |
|---|---|---|---|---|
| computeSafeVelocity | 1 | 4.503 s | 4.503 s | 🟦 |

**Self time** is the time spent in a function excluding the time spent in its child functions. Self time also includes overhead resulting from the process of profiling.

Figure 7. *Performance profile summary.*

The profile summary reveals that most of the execution time is spent in the `computeSafeVelocity` function. When you select this function, you can examine it for opportunities to improve its performance (Figure 8).



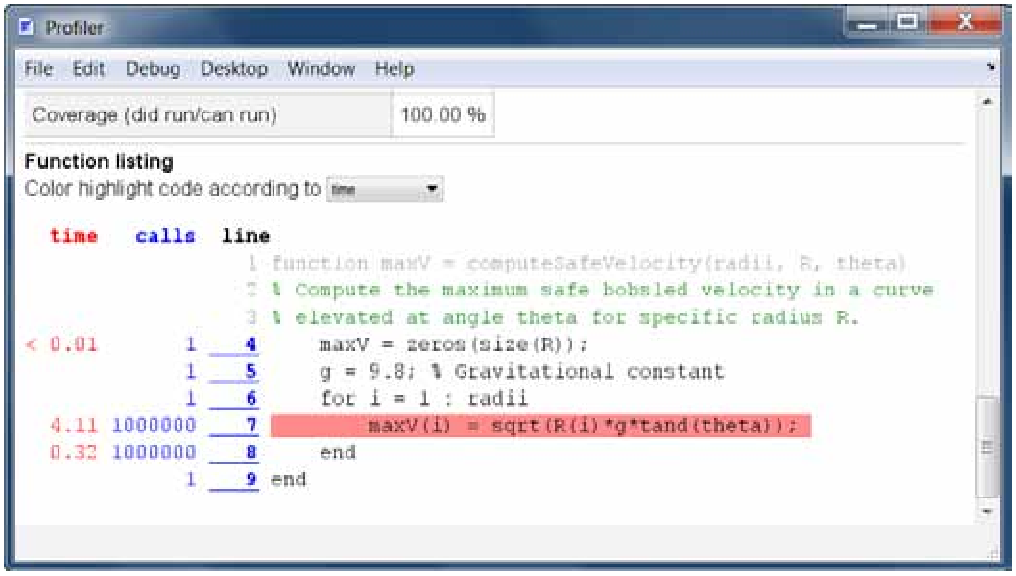
**Parents** (calling functions)
No parent

**Lines where the most time was spent**

| Line Number | Code | Calls | Total Time | % Time | Time Plot |
|---|---|---|---|---|---|
| 7 | `maxV(i) = sqrt(R(i)*g*tand(the...` | 1000000 | 4.182 s | 92.9% | ▬ |
| 8 | `end` | 1000000 | 0.310 s | 6.9% | ▪ |
| 4 | `maxV = zeros(size(R));` | 1 | 0.010 s | 0.2% | |
| 9 | `end` | 1 | 0 s | 0% | |
| 6 | `for i = 1 : radii` | 1 | 0 s | 0% | |

Figure 8. *Detailed performance analysis of the* `computeSafeVelocity` *function.*

The profiler window shows the total time that each line in the function takes to execute. In this example, line 7 is clearly the bottleneck. When you click the line 7 link, you can see this line in its context within the function (Figure 9).



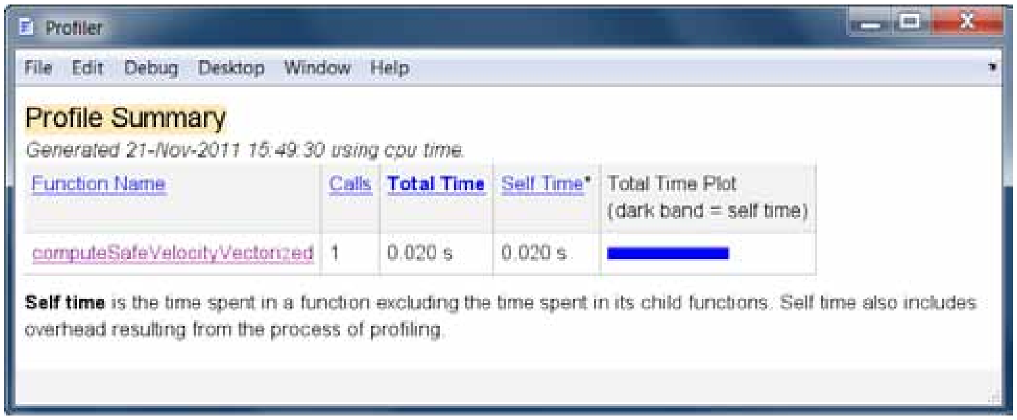
| Coverage (did run/can run) | 100.00 % |
|---|---|

**Function listing**
Color highlight code according to time

```
  time     calls   line
                     1 function maxV = computeSafeVelocity(radii, R, theta)
                     2 % Compute the maximum safe bobsled velocity in a curve
                     3 % elevated at angle theta for specific radius R.
< 0.01         1     4    maxV = zeros(size(R));
               1     5    g = 9.8; % Gravitational constant
               1     6    for i = 1 : radii
  4.11   1000000     7        maxV(i) = sqrt(R(i)*g*tand(theta));
  0.32   1000000     8    end
               1     9 end
```

Figure 9. *Code with bottleneck highlighted.*

The highlighted line is inside a `for` loop. Because each iteration of this `for` loop is independent of the others, it is an ideal candidate for vectorization. By vectorizing this loop using the techniques described in this paper, you can significantly improve performance, cutting execution time from about 4 seconds to 0.02 seconds (Figure 10).



Profile Summary

Generated 21-Nov-2011 15:49:30 using cpu time.

| Function Name | Calls | Total Time | Self Time* | Total Time Plot (dark band = self time) |
|---|---|---|---|---|
| computeSafeVelocityVectorized | 1 | 0.020 s | 0.020 s | ▬ |

**Self time** is the time spent in a function excluding the time spent in its child functions. Self time also includes overhead resulting from the process of profiling.

Figure 10. *Performance profile for updated vectorized code.*

## IV. Using Parallel Computing to Accelerate Serial MATLAB Computations

When even highly optimized code does not execute fast enough to meet requirements, parallel computing may further enhance performance. Parallel Computing Toolbox™ enables you to parallelize your application and execute it using:

- Local computational resources (multiple cores and processors on a single machine)
- Remote computational resources (clusters)
- GPU computing resources (both local and remote)

### 1. Use `parfor` to Accelerate `for` loops with Independent Loop Iterations

When you are working on a single machine, one of the easiest parallel constructs to employ is the `parfor` command. `parfor` lets you parallelize and speed up programs with minimum changes to your code. It is best suited to `for` loops for which there are no dependencies between loop iterations, when the final result does not depend on the order of the iterations.

In many cases, a program can be parallelized with minimal code changes. For example, the algorithm in Figure 11 computes the maximum velocity at which a car can travel on a tilted curve of varying radius *r*. The coefficient of friction between the road and the tires is constant, but the radius of the curve under consideration ranges from 10 meters to 10,000 meters.

```
function speed = MaxSafeSpeedTable(cFriction, elements, rStart,
rEnd, theta)
g = 9.8;
r = linspace(rStart, rEnd, elements);
speed = zeros(size(r));
for ii = 1:elements
  num = (sind(theta) + cFriction*cosd(theta));
  den = (cosd(theta) - cFriction*sind(theta));
  speed(ii) = sqrt(r(ii) * g *num/den);
end
end
```

Figure 11. *Serial code for computing the maximum velocity of a car traveling on a tilted curve.*

The main part of this computation inside the `for` loop computes a maximum permissible velocity of the car for a curve of radius `r`. Since each loop iteration is independent of the others, this task is said to be *embarrassingly parallel*, and is perfect for parallelization using the `parfor` construct. The only code change necessary is switching `for` to `parfor` (Figure 12).

```
function speed = parMaxSafeSpeedTable(cFriction, elements, rStart,
rEnd, theta)
g = 9.8;
r = linspace(rStart, rEnd, elements);
speed = zeros(size(r));
parfor ii = 1:elements
  num = (sind(theta) + cFriction*cosd(theta));
  den = (cosd(theta) - cFriction*sind(theta));
  speed(ii) = sqrt(r(ii)*g* num/den);
end
end
```

Figure 12. *Parallel code for computing the maximum velocity of a car traveling on a tilted curve.*

Accelerating this algorithm with `parfor` requires just four steps:

1. Replace the `for` loop with `parfor`.
2. Create a MATLAB pool of workers using the `matlabpool open` command.
3. Execute the program.
4. Close the MATLAB pool using the `matlabpool close` command.

You can compare the performance of the parallel and the serial code by:

- Keeping the number of iterations the same and increasing the number of workers (*strong scaling*)
- Increasing the number of iterations proportionally with the number of workers (*weak scaling*)

Table 1 shows the performance differences between the serial code and the parallel code for strong scaling. The number of workers increases, but the amount of computation is fixed. As workers are added, performance improves significantly. Note, however, that the increase is not linear: the execution time of parallel runs on *N* workers is not *exactly N* times faster than serial execution. The small difference is due to communication overhead between the client MATLAB instance and the MATLAB pool workers in the parallel runs.

| Number of Workers (N) | Serial | 2 | 4 | 8 |
|---|---|---|---|---|
| Execution Time (seconds) | 27.89 | 13.19 | 9.28 | 5.69 |

Table 1. *Strong scaling performance comparison between serial and parallel implementations.*

Table 2 shows the results using weak scaling. This method illustrates the scalability potential of `parfor` as the number of iterations increases in proportion to the number of MATLAB workers. While the `parfor` execution times are slightly higher than the serial execution times, remember that the `parfor` solutions complete *N* times more velocity computations than the serial version of the application (where *N* is the number of workers). Again, the small difference between the serial and parallel version execution times is due to communication overhead between the client MATLAB instance and the MATLAB pool workers.

| Number of Workers (N) | Serial | 2 | 4 | 8 |
|---|---|---|---|---|
| Execution Time (seconds) | 27.89 | 28.22 | 27.52 | 30.5 |

Table 2. *Weak scaling performance comparison between serial and parallel implementations.*

## 2. Accelerate Computations on Computer Clusters

In addition to running your MATLAB programs on local workers, Parallel Computing Toolbox lets you execute jobs on distributed computing resources, such as computer clusters running MATLAB Distributed Computing Server™. With Parallel Computing Toolbox installed on your local machine and MATLAB Distributed Computing Server installed on a cluster or in the cloud, you can submit MATLAB programs to clusters as batch jobs for execution using one of several approaches. The easiest employs the `batch` command. With `batch` you submit existing MATLAB programs for off-line processing on clusters and local machines. These programs can be strictly serial or can include parallel constructs, such as `parfor` loops. In addition to the `batch` command, Parallel Computing Toolbox includes lower-level functionality for submitting jobs to clusters. This approach offers more control than the `batch` command. For more details, see the Parallel Computing Toolbox documentation.

## 3. Accelerate Computations on GPUs

Support for the graphics processing unit (GPU) in Parallel Computing Toolbox enables you to accelerate your programs by executing some operations on NVIDIA GPUs with compute capability of 1.3 or higher. There are several ways to use the GPU to accelerate your code directly from MATLAB.

`GPUArray` is a special array type provided by Parallel Computing Toolbox that enables you to execute more than 100 built-in MATLAB functions directly on the GPU. In most cases, you simply invoke `gpuArray()` to transfer data to the GPU device memory, perform the calculations, and then invoke `gather()` to transfer the data back to the MATLAB workspace from device memory. Figures 13 and 14 show the code for solving a system of linear equations `(Ax = b)` on the CPU and on a GPU for a range of matrix sizes.

```
function cpuSolveLinEqs
max_size = 6*1e3;
matrix_sizes = round(linspace(1,max_size,10))

for i = 1:length(matrix_sizes)
  tic
  A = rand(matrix_sizes(i));
  b = ones(matrix_sizes(i),1);
  x = A\b;
  time_cpu(i) = toc
end
end
```

Figure 13. *Code for solving a system of linear equations on the CPU.*

```
function gpuSolveLinEqs
max_size = 6*1e3;
matrix_sizes = round(linspace(1,max_size,10))

for i = 1:length(matrix_sizes)
  tic
  A = rand(matrix_sizes(i));
  b = ones(matrix_sizes(i),1);
  gA = gpuArray(A); % Transfer data to GPU
  gb = gpuArray(b);
  gx = gA\gb; % Perform computations on GPU
  x = gather(gx ); % Bring results back from GPU
  time_gpu(i) = toc
end
end
```

Figure 14. *Code for solving a system of linear equations on the GPU.*

Figure 15 shows the execution times for both algorithms as a function of matrix dimension. Note that the GPU version performs much better than the CPU version for larger matrices.

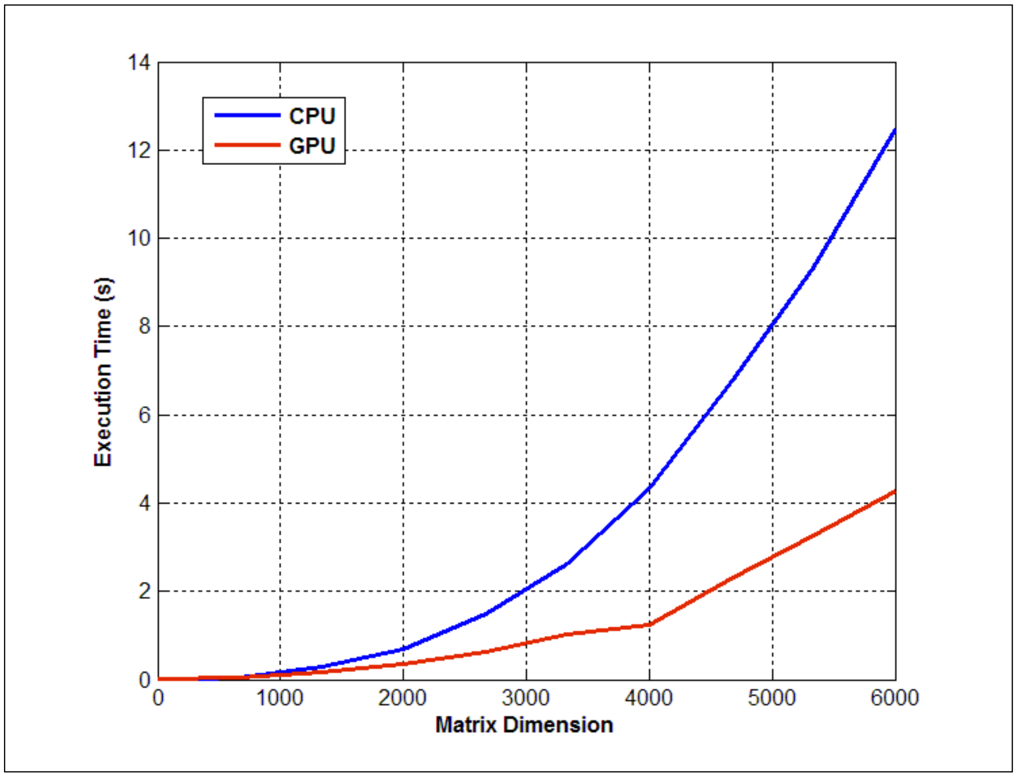


Figure 15. *Performance comparison: solving a system of linear equations on a CPU and on a GPU.*

The GPU-enabled functions in MATLAB are *overloaded*; they operate differently depending on the data type of the arguments passed to them. If the default set of overloaded functions does not provide all the functionality you need, you can write your own MATLAB functions and execute them on the GPU. In addition, if you have your own CUDA kernels, you can run them natively on the GPU from MATLAB.

## V. Conclusion

This paper introduced three techniques for speeding up MATLAB code: preallocation, memory access optimization, and code vectorization. It also showed how to use MATLAB Profiler and MATLAB Code Analyzer to identify performance bottlenecks. Lastly, it demonstrated techniques for using Parallel Computing Toolbox to further accelerate MATLAB computations on multiple workers, clusters, and GPUs.

## VI. Learn More

Analyzing Your Program's Performance
*mathworks.com/help/techdoc/matlab_prog/f8-790895.html*

Code Vectorization Guide
*mathworks.com/support/tech-notes/1100/1109.html*

GPU Computing
*mathworks.com/help/toolbox/distcomp/bsic3by.html*

How do I Preallocate Memory When Using MATLAB?
*mathworks.com/support/solutions/en/data/1-18150/?solution=1-18150*

Maximizing Code Performance by Optimizing Memory Access
*mathworks.com/company/newsletters/news_notes/june07/patterns.html*

Parallel `for` Loops (`parfor`)
*mathworks.com/help/toolbox/distcomp/bq9u0a2.html*

Techniques for Improving Performance
*mathworks.com/help/techdoc/matlab_prog/f8-784135.html*

Using `GPUArray`
*mathworks.com/help/toolbox/distcomp/bsic4fr-1.html*

© 2013 The MathWorks, Inc. MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See mathworks.com/trademarks for a list of additional trademarks.
Other product or brand names may be trademarks or registered trademarks of their respective holders.

MathWorks®

91991v01 03/13