Accelerating MATLAB

The MATLAB JIT-Accelerator

Introduction

Most engineers and scientists use two types of computer languages for the analysis, design, and implementation of technical applications: third-generation languages (3GLs), such as C, C++, Fortran, and Basic, and fourth-generation languages (4GLs) such as MATLAB. While 4GLs offer tremendous ease-of-use and productivity benefits, certain types of code have typically executed more quickly using third-generation languages. The MathWorks has developed technology that combines the ease-of-use of a 4GL with the fast performance of a 3GL. The MATLAB JIT-Accelerator, introduced in MATLAB 6.5, includes several technological innovations that accelerate the execution of MATLAB code. This technology backgrounder describes MATLAB language execution, explains how the new JIT-Accelerator speeds up MATLAB code, and identifies which types of code will see the greatest performance benefit from the JIT-Accelerator.

Background: 3GLs vs. 4GLs and MATLAB Language Execution

Third-generation languages are sometimes referred to as high-level languages because they add a layer of abstraction to hard-to-use lower-level languages, such as assembly and machine code. 3GLs are translated into assembly or machine language to execute very quickly. To use 3GLs effectively requires a good deal of programming experience and knowledge.

Fourth-generation languages are much less procedural in nature than 3GLs and consist of statements similar to those in human language. For this reason, 4GLs are typically much easier to use than 3GLs. However, due to the way that 4GL code is interpreted, execution time is often slower than with 3GLs. MATLAB is a 4GL that was developed specifically for engineers and scientists. While many common functions, such as vector and matrix math, are highly optimized to execute quickly, other operations have incurred the overhead common in 4GLs.

MATLAB Language Execution

Before MATLAB 6.5, the MATLAB language was processed in two steps. First, the MATLAB code was converted into a linear stream of p-code, the instruction set that is executed by the MATLAB interpreter. Second, the interpreter executed each p-code instruction in sequence. Each p-code execution incurred a small amount of overhead. Some p-code instructions were high-level and took much longer to execute than the overhead, so the execution overhead was insignificant. In some cases however, the p-code operation ran very quickly and the interpreter overhead was the majority of the total execution time. The most common examples are codes that deal with scalar values and for loops.

MATLAB Type Handling

An important benefit of MATLAB is that users do not have to declare variables to be of certain data types, as is required with 3GLs. In MATLAB, any variable can be assigned a value of any type, and that type can be changed implicitly at will because of an assignment to a new value of a different type. As a result, the MATLAB interpreter is prepared to deal with the most complicated data types (such as an n-dimensional array of complex doubles) and is capable of performing operations no matter what the actual data types turn out to be at run-time. Prior to M 6.5, the p-code specified the most complicated case. As a result, code that operated on scalar values incurred additional overhead in execution time and storage.

MATLAB JIT-Accelerator: Fast Execution of MATLAB Code

The JIT-Accelerator is a built-in feature of MATLAB that lets users automatically take advantage of increased code execution speed. MATLAB 6.5 is the first release of MATLAB to include the JIT-Accelerator. This first release focuses on improving the speed of loops and scalar math. Subsequent releases of MATLAB will contain additional performance improvements.

The MATLAB 6.5 JIT-Accelerator speeds up execution of MATLAB code using two primary methods: Just-In-Time Code Generation and Run-time Type Analysis.

Just-In-Time Code Generation

The JIT-Accelerator converts many p-code instructions into native machine instructions. These instructions suffer no interpreter overhead, and therefore run very quickly. In some cases, code generated by the JIT-Accelerator can run several thousand times faster than was possible in prior versions of MATLAB. While most programs will not have speed increases of that magnitude, some scalar code running in large loops can achieve speedups of several hundred times. Users are likely to experience greater productivity on Intel X86-based Linux and Windows systems than on other platforms due to some additional optimization for these systems.

Run-time Type Analysis

Prior to MATLAB 6.5, some additional overhead was generated during p-code execution due to the way that MATLAB handled the typing of variables. Run-time type analysis eliminates this overhead, significantly speeding up execution of many p-code operations.

Run-time type analysis is based on the following premise: If a line of M-code has been processed before, it is very likely that the variables have the same types and shapes that they had the last time the system saw this line. The first time that a line of code is executed, the system examines the variables and generates specific code for the data types and shapes that were found. Subsequent executions of the line can reuse this code as long as the system verifies that the variable types and sizes have not changed. Since the types rarely change, subsequent executions run as quickly as possible. If the types do change, the code is regenerated.

Performance Benefits of the JIT-Accelerator and Supported MATLAB Features

Programming Style

The JIT-Accelerator gives you the flexibility to run your code faster without having to perform vectorization. Vectorization, the process of structuring MATLAB code to work on matrices, serves two purposes: It enables algorithms to be expressed more succinctly and provides a mechanism for improving MATLAB execution speed. Today with the JIT-Accelerator, you no longer need to vectorize code to speed up the execution of many applications.

MATLAB users write the MATLAB code that is the most understandable or that best fits their application. The JIT-Accelerator then ensures optimal performance. Vectorization can be used if it results in code that is clearer and more concise. Loops processed by the JIT-Accelerator often execute at least as fast as vectorized loops. Programs that are already vectorized may experience minor improvements with the JIT-Accelerator. However, since the internal vectorized funtions have already been optimized, acceleration of this code may be modest.

Supported MATLAB Features

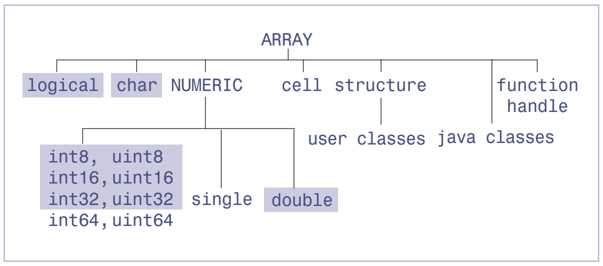
The MATLAB JIT-Accelerator supports the following aspects of the MATLAB language:

- Data types and array shapes
- `for` loops
- Conditional statements
- Array size

These supported features will automatically take advantage of the JIT-Accelerator.

Data Types and Array Shapes

MATLAB accelerates code that uses the data types that are shaded in the class hierarchy diagram below. Both real and complex doubles are accelerated. All arrays are accelerated except sparse arrays and those array shapes with more than three dimensions.



MATLAB data types.

`for` Loops

Loops controlled by a `for` statement execute faster in MATLAB as long as they meet the following conditions:

- Indices of the `for` loop are set to a range of scalar values.
- Code in the `for` loop uses only the supported data types and array shapes.
- Any functions called within the loop are MATLAB built-in functions.

Loop performance is optimal when every line of code in the loop can take advantage of the JIT-Accelerator. When this is the case, MATLAB speeds up execution of the entire loop, including the `for` and `end` statements. If this is not the case, then acceleration of the loop is temporarily interrupted on each iteration of the loop.

Conditional Statements

`if, elseif, while,` and `switch` statements execute faster as long as the expression in the statement evaluates to a scalar value.

Array Size

The execution overhead represents a small percentage of the total execution time when handling large arrays. For small arrays, however, the reverse is true. Since the JIT-Accelerator reduces the overhead incurred with data handling, there is a greater relative performance improvement in programs that use smaller arrays.

Code that Benefits Most from the JIT-Accelerator

Programs consisting of large, contiguous portions of code that contain only those elements of MATLAB that take advantage of the JIT-Accelerator will show the greatest speed improvement. This is because whenever the interpreter encounters an element not supported by the JIT-Accelerator, it handles the instruction through the nonaccelerated interpreter. The more often this happens, the less opportunity there is to speed up the code. For this reason, the most significant performance improvements are achieved in functions and scripts that primarily consist of self-contained loops, particularly loops that make no calls to M-file functions and operate on scalar data.

MATLAB is optimized for vector and matrix math. Most of the built-in functions in MATLAB (such as `fft, eig,` and matrix multiply functions) are already running as fast as possible. As a result, this type of code will not be affected by the JIT-Accelerator.

A Simple Example

The following code example demonstrates the optimizations provided by the JIT-Accelerator:

```
a = 2.5;
b = 3.33;
c = 6.23;
for i = 1:10000000
    a = b + c;
    b = a - c;
    c = b * a;
    if(a > b)
        a = b - c;
    else
        b = b + 12.3;
    end
end
```

This piece of code cycles through a loop ten million times. The loop performs some scalar math and a comparison. Tests show this code running approximately 550 times faster in MATLAB 6.5 with the JIT-Accelerator than with MATLAB 6.1. This code was developed with code acceleration in mind, and represents one of the larger improvements that one is likely to experience. However, other examples executed as much as 3,400 times faster than with MATLAB 6.1.

Summary

The MATLAB 6.5 JIT-Accelerator is the first step towards the MathWorks ultimate goal of eliminating the performance difference between MATLAB and 3GLs. Without any difference in performance, users can skip the additional work of recoding their programs in C, for example. With MATLAB 6.5, programs that include a lot of scalar math and loops for iterative numeric algorithms will experience the greatest performance benefits. While it is still the case that MATLAB functions operate on matrices quickly, it is no longer necessary to vectorize code to achieve optimum speed. The JIT-Accelerator enables MATLAB users to write programs in a style that is both comfortable and fits the application at hand.

With its flexible development environment, intuitive language, and the performance optimization of the JIT-Accelerator, MATLAB 6.5 is a compelling alternative to 3GLs for technical computing applications. It is the only tool available that maximizes both application performance and user productivity.

Sample Accelerated Programs

The following programs demonstrate how to make the best use of the MATLAB JIT-Accelerator:

- Program 1 – Bayes' Rule
- Program 2a – Vector Comparison, with Loop
- Program 2b – Vector Comparison, Vectorized
- Program 3 – Relaxation Algorithm

Each program shows a sizeable performance improvement over earlier versions of MATLAB. The table shows hardware specifications for the systems used in measuring performance on the program examples.

Operating System	CPU Type	Processor Speed	Main Memory
Windows	Intel x86	1 GHz	256 MB
Linux	Pentium III	550 MHz	256 MB
Solaris	Sparc	270 MHz	128 MB

Program 1—Bayes' Rule

This program implements Bayes' Rule for computing probability based on prior probability and updated information. The program contains nested `for` loops that, unless they were vectorized, would be quite costly to execute without the JIT-Accelerator.

The table below compares the performance of this program in MATLAB 6.1 (with no acceleration) and in MATLAB 6.5 (with acceleration). The increase in performance is shown in the far right column for the three operating systems tested.

Operating System	MATLAB 6.1	MATLAB 6.5	Performance Gain
Windows	16 min., 0.5 sec.	2.7 sec.	x 355.7
Linux	38 min., 15.8 sec.	5.9 sec	x 389.1
Solaris	1 hr., 47 min.,32.7 sec.	57.9 sec.	x 111.4

Here is the Bayes.m program.
function score = Bayes(Seq, Matrix, priorProbability)

```
% BAYES  Use Bayes' rule to determine
signal probabilities.
%
%   score = BAYES(Seq, Matrix,
% priorProbability) is the
%   probability that the signal whose
% probability is expressed in Matrix
%   occurs at each position in Seq,
%   where priorProbability is the probability
% of seeing the signal at any position.

% Using Bayes' rule:
% P(a|b) = P(b|a) * P(a) / P(b), where Pa
% is the probability that whatever signal
% we're looking for occurs at a given posi-
% tion, Pb|a is the probability that we would
% see this specific nucleotide if the signal
% were here, and Pb is the unconditional
% probability of seeing this nucleotide here.

% Initially, we'll assume each nt is
% equally likely.
 Pb = 1/4;

% Initialize storage for the result.
% score = zeros(1, length(Seq));

 lm = length(Matrix);
 ls = length(Seq) - length(Matrix);

 for m = 1:ls
    Pa = priorProbability;
    k = m - 1;

    for n = 1:lm
       nt = Seq(k + n);
       if (nt > 0) && (nt < 5)
          PbGa = Matrix(nt, n);
          Pb = Pa * PbGa + (1 - Pa) * 0.25;
          Pa = PbGa * Pa / Pb;
       end
    end

    score(m) = Pa;
 end
```

The times shown in the previous table were recorded by running the Bayes' program on data stored in a double matrix and a large 8-bit integer vector, with a prior probability of 0.0001.

Variables in the Bayes.m program

Name	Size	Bytes	Class
Matrix	4x20	640	double array
Seq	1x912211	912211	int8 array

What Makes It Faster

The program spends most time in a nested `for` loop that calculates the `Pb|a, Pb,` and `Pa` values. When run with the data shown in the previous table, the inner loop executes more than 18 million times. Because all the code in the program uses elements of MATLAB that support acceleration, the entire program runs much faster.

MATLAB accelerates this code as follows:

Supported Data Types and Array Shapes. All the statements within the inner and outer loops use double and 8-bit integer data in scalar, vector, and two-dimensional matrix form. These data types and array shapes support performance acceleration.

Scalar Loop Indices. Both `for` loops operate on a range of scalar values, a criteria for JIT-acceleration. For example, `ls` used in this statement is scalar.

```
for m = 1:ls
```

Function Calls and Overloading. Calling functions that are MATLAB built-ins execute very quickly, but calling functions implemented in M-files can use up considerable time. This program calls only built-in functions, such as `zeros` and `length,` with no function calls in the nested loops. In addition to direct function calls, there are no overloaded operations that implicitly call M-file functions.

Conditional Statements. The one conditional statement residing in the inner loop evaluates scalar terms, enabling it to be JIT-accelerated This statement uses the scalar, `nt,` as shown here.

```
if (nt > 0) && (nt < 5)
```

Small Array Size. The second argument, matrix, is a 4-by-20 array. Small arrays perform faster. The overhead of array handling, which is more noticeable with smaller arrays, is insignificant in accelerated versions of MATLAB. The first argument, `Seq,` is quite large and does not speed up much due to its size alone.

Program 2a—Vector Comparison, with Loop

This program scans two sorted input vectors and finds the elements that are common to both. It returns the indices of these common elements.

There are two versions of this program: Program 2a processes the vectors using a `while` loop. Program 2b replaces the loop with vectorized code. When run on MATLAB without acceleration, there is a big difference in performance between the two. When run with acceleration, there is no significant difference.

Operating System	MATLAB 6.1	MATLAB 6.5	Performance Gain
Windows	10.6 sec.	0.1 sec.	x 106.0
Linux	24.0 sec.	0.1 sec.	x 240.0
Solaris	1 min., 4.7 sec.	1.5 sec.	x 43.1

```
function [aIndex, bIndex] = vfind_scalar(avec, bvec)

 avecLen = length(avec);
 bvecLen = length(bvec);

% Size aIndex and bIndex to be large enough
 outlen = min(avecLen, bvecLen);
 aIndex = zeros(outlen,1);
 bIndex = zeros(outlen,1);

 n  = 0;
 ai = 1;
 bi = 1;

 while (ai <= avecLen || bi <= bvecLen)
% Get vector elements at indices ai and bi
    A = avec(ai);
    B = bvec(bi);

% If equal, record indices where elements match
    if A == B
       n = n + 1;
       aIndex(n) = ai;
       bIndex(n) = bi;
    end

% Advance index of avec, when appropriate
    if A <= B
       if ai < avecLen
          ai = ai + 1;
       else
          break;
       end
    end

% Advance index of bvec, when appropriate
    if A >= B
       if bi < bvecLen
          bi = bi + 1;
```

```
        else
            break;
        end
    end
end

% Snip aIndex and bIndex to correct size
 aIndex = aIndex(1:n);
 bIndex = bIndex(1:n);
```

In preparation for running the program, you must create two sorted vectors that have some common elements. The following statements create vectors `a` and `b`, append the elements from a third vector, `c`, to both, and then sort. This gives vectors `a` and `b` at least 20,000 common elements.

```
 a = rand(200000,1);
 b = rand(260000,1);
 c = rand(20000,1);
 a = sort([a;c]);
 b = sort([b;c]);
```

Now pass `a` and `b` into the function shown above. Use the `tic` and `toc` functions to track how much time it takes to execute.

```
 tic;  [ia, ib] = vfind_scalar(a, b); toc
```

What Makes It Faster

MATLAB accelerates every line in this program. The code in the `while` loop matters most, since this takes up nearly all the program execution time. Consider the following factors:

Supported Data Types and Array Shapes. All the code in the program operates on vectors of type double. This is one of the data types and array shapes that supports acceleration.

Conditional Expression Evaluates to Scalar. The expressions in the `while` and `if` statements all evaluate to scalar values. For example,

```
while (ai <= avecLen || bi <= bvecLen)
```

No Disqualifying Statements in Loop. Every line of code in the `while` loop qualifies for JIT-acceleration. This means that every iteration of the loop can execute at a higher speed without being interrupted to separately process any disqualifying lines.

Function Calls and Overloading. The only functions called are MATLAB built-ins. No M-file or MEX-file functions are called, and no operations are overloaded for the data types being used.

Program 2b—Vector Comparison, Vectorized

Here is the vectorized version of the program 2a. Note that there is little difference in performance between the accelerated `while` loop, shown in Program 2a, and the vectorized code shown below. This means that with accelerated MATLAB, you can choose the style of coding that you prefer for each MATLAB application without affecting performance.

The table below also shows little difference in the times measured for running the vectorized code on unaccelerated and accelerated versions of MATLAB. You will not see a significant performance improvement in vectorized programs when run with the JIT-Accelerator.

Operating System	MATLAB 6.1	MATLAB 6.5	Performance Gain
Windows	0.7 sec.	0.6 sec.	x 1.2
Linux	1.0 sec.	0.9 sec.	x 1.1
Solaris	1.2 sec.	1.2 sec.	x 1.0

Here is the vectorized version of the `vfind` program:

```
function [aIndex, bIndex] =
vfind_vector(avec, bvec)

avecLen = length(avec);
bvecLen = length(bvec);

avec = reshape(avec, avecLen, 1);
bvec = reshape(bvec, bvecLen, 1);

[c, pc] = sort([avec; bvec]);
cIndex = find(diff(c) == 0);
aIndex = pc(cIndex);
bIndex = pc(cIndex + 1) - avecLen;
```

Program 3 —Relaxation Algorithm

This program starts by creating a sharply contrasted graphics display in a figure window. When you press any key after starting the program, it runs a relaxation algorithm on the figure, gradually blurring the color boundaries.

When the algorhithm is run on earlier versions of MATLAB, you can see the algorithm taking effect in distinct steps that are spaced over time. When it is run on MATLAB 6.5, the display changes smoothly over a much shorter period.

This table shows comparative execution times for 300 iterations of the program. (The version of this program used in these measurements differs slightly from what is shown below. The final image handling functions (`set` and `drawnow`) were moved to the outside of the three nested `for` loops, and thus refreshed the image only once, at the very end.)

Operating System	MATLAB 6.1	MATLAB 6.5	Performance Gain
Windows	1 min., 25.9 sec.	1.1 sec.	x 78.1
Linux	3 min., 13.8 sec.	1.7 sec.	x 114.0
Solaris	8 min., 51.0 sec.	23.5 sec.	x 22.6

Here is the program:

```
function relax(iterations)
sz = 102;

plate = magic(sz) * 64 / (sz * sz);
newPlate = plate;

im = image(plate);
axis off
set(gcf, 'DoubleBuffer', 'on')

% Wait for user to press a key to kick
% off the image processing
pause

for i = 1:iterations
   for j = 2:(sz-1)
      jm1 = j - 1;
      jp1 = j + 1;
      for k = 2:(sz-1)
         km1 = k - 1;
         kp1 = k + 1;
         newPlate(j,k) = (plate(jm1,km1)/2 + ...
           plate(jm1,k) + plate(jm1,kp1)/2 + ...
           plate(j,km1) + plate(j,kp1) + ...
           plate(jp1,km1)/2 + plate(jp1,k) + ...
           plate(jp1,kp1)/2)/6;
     end
  end
   plate = newPlate;

% Refresh the image once every 5 times
% through the loop.
   if (0 == rem(i,5))
      set(im, 'cdata', plate)
      drawnow
   end

end
```

You can see the visual effect of the faster execution by running the program yourself. Put the code into an M-file named relax.m, and run it for 300 iterations by typing

```
relax(300);
```

A new window is created showing the initial image. Once you see this, reselect the MATLAB command window, and then press any key to start processing the image.

What Makes It Faster

The program spends most of its time in a nested `for` loop that modifies the image data. The newPlate calculation in the inner loop executes 3 million times when iterations `is` set to 300, as it is in the above test.

MATLAB accelerates the two inner `for` loops for the reasons explained below. The outer loop does not accelerate (see The Outer `for` Loop), yet that has little effect on the overall execution time, as nearly all of the time spent is in the inner loops.

Scalar loop indices. The `for` loops operate on a range of scalar values, a criteria for JIT-acceleration. For example,

```
for j = 2:(sz-1)
```

Supported Data Types and Array Shapes. All of the statements within the inner loops use a double data type with either a scalar value or a two-dimensional matrix. These are among the data types and array shapes that MATLAB accelerates.

Higher Complexity Operations. MATLAB usually shows a noticeable performance gain for statements containing multiple operators and/or functions. The plate computation is an example of this.

```
newPlate(j,k) = (plate(jm1,km1)/2 + ...
  plate(jm1,k) + plate(jm1,kp1)/2 + ...
  plate(j,km1) + plate(j,kp1) + ...
  plate(jp1,km1)/2 + plate(jp1,k) + ...
  plate(jp1,kp1)/2)/6;
```

Function Calls and Overloading. One factor that enables the acceleration of the two inner loops is that the only function calls made in this code are to built-in functions. The program performs a number of mathematical operations, but as long as none of the math operators used (+, -, and /) is overloaded for the data type being operated on (double), these math operations execute quickly, not having to make M-file calls.

The Outer `for` Loop

The outer `for` loop is not sped up by the JIT-Accelerator. One reason is that the leading `for` statement relies on an ambiguous data type for the maximum index value. The value for iterations is passed into the program and thus may not necessarily be one of the data types or array shapes supported for JIT-acceleration.

```
for i = 1:iterations
```

If you include the final set and `drawnow` calls in the loop, these will not be sped up by the JIT-Accelerator because they operate on Handle Graphics® objects, which are not among the data types supported for performance acceleration.

```
        set(im,'cdata',plate)
        drawnow
```

The fact that the outer loop does not accelerate is not that important in this case, as nearly all of the execution time is spent in the inner loops.

Will my MATLAB code run faster in MATLAB 6.5?

The JIT-Accelerator will result in the greatest performance improvements to MATLAB code that:

- Is loop intensive
- Does a lot of scalar math
- Does not include sparse arrays or 3-D arrays
- Includes `if`, `elseif`, `while`, and `switch` statements within the code that has expressions that evaluate to a scalar value
- Includes `for` loops that resolve to scalar values
- Does not call other functions (M-file or MEX) or subfunctions
- Has large, contiguous portions of code with the characteristics mentioned above
- Is not highly vectorized
- Is executed on Intel X86 platforms

For demos, application examples, tutorials, user stories, and pricing:

- Visit **www.mathworks.com**
- Contact The MathWorks directly

US & Canada	508-647-7000
Benelux	+31 (0)182 53 76 44
France	+33 (0)1 41 14 67 14
Germany	+49 (0)241 470 750
Italy	+39 (011) 2274 700
Spain	+34 93 362 13 00
Switzerland	+41 (0)31 954 20 20
UK	+44 (0)1223 423 200

Visit **www.mathworks.com** to obtain contact information for authorized MathWorks representatives in countries throughout Asia Pacific, Latin America, the Middle East, Africa, and the rest of Europe.

The MathWorks

www.mathworks.com

09/02 91076v00