COMPILING MATLAB

Stephen C. Johnson

Melismatic Software

Cleve Moler

The MathWorks, Inc.

ABSTRACT

MATLAB is a high level language oriented towards scientific and engineering applications. It has evolved over a ten year history to become a popular, flexible, powerful, but still simple language, and has served as an effective platform for over a dozen "toolboxes" supporting everything from symbolic computation to digital filter design, control theory, and neural nets. Designed to be used interactively, MATLAB also supports the ability to define functions and scripts, and dynamically link with C and FORTRAN programs. This paper discusses a project to provide a compiler for MATLAB. The focus of this paper is on techniques that might be useful for other high-level languages, and the parts of such languages that resist compilation. Our constructed MATLAB compiler is in beta test, and has demonstrated speedups over interpretation of up to 300 times on practical programs.

Introduction to MATLAB

MATLAB grew out of an attempt to provide a user-friendly front end for the LINPACK matrix programs developed in the late '70s and early '80s. Visualization methods were added in the late '80s, including a powerful 3-D graphing capability. Recent trends in the language have focused on an object-oriented graphics capability that permits a rich GUI construction, and the support of over a dozen toolboxes that support specialized scientific and engineering areas. A student edition of MATLAB is available for a low price, and over fifty textbooks have been written incorporating MATLAB in their course material.

MATLAB is, for all intents and purposes, a typeless language (or, if you wish, a strongly typed language), since every data object has the same type: a two dimensional array of complex numbers. Vectors are arrays with one dimension equal to 1, while scalars are 1x1 arrays. The sole exception is that strings have a flag bit set to control how they are printed: a string is represented as a vector of real numbers, one byte per element of the array. (This is in amusing contrast to many very very high-level languages that represent numbers as strings; MATLAB represents strings as numbers...). There is also support for sparse matrices, although they intermingle freely with regular matrices. The language is supported by an interpreter that supports interactive use, creation and execution of functions, dynamic loading of FORTRAN and C functions, and extensive debugging facilities.

Syntactically, MATLAB has few surprises. As you might expect, there is a notation for describing two dimensional arrays:

```
[   23, 2+3i, -5 ;  0, 1, 2 ]
```

describes an array with two rows and three columns. Most MATLAB expressions will print their values when typed--following the expression with a semicolon will suppress the printing. Because many numerical algorithms return multiple values, functions (but not ordinary assignments) can assign to several variables:

```
[L,U] = lu(X);
```

returns the standard L-U decomposition of a matrix X. Functions are also polymorphic:

```
[L,U,p] = lu(X);
```

returns a permutation in addition to the L and U matrices. Functions can interrogate how many inputs and outputs they are called with--in fact, the L matrix returned by lu is different in the two argument and three argument case.

MATLAB users have been encouraged to "vectorize" their computations--that is, to make use of built in array and matrix operations rather than writing scalar code. To this end, a number of vector-like extensions have been put into MATLAB. For example, if I is a vector of 0's and 1's, and A is a vector the same length as I, then A(I) is a vector consisting of just those elements of A where I is 1 (The apparent ambiguity is resolvable (just barely) because subscripts in MATLAB start at 1). Since relational operators and most Boolean operations produce vectors of 0's and 1's, there is no shortage of such index vectors. So you can say A(A>0) to refer to the vector of positive elements from the vector A. You can also delete elements by assigning the empty array to them:

```
A(A<0) = [];
```

deletes all the negative elements from the vector A.

Throughout, the emphasis in MATLAB has been on supporting the user with "no surprises", and the result has been quite successful. However, while the user has few surprises, there are plenty in the implementation. As a simple example, consider the colon operator that produces regularly spaced vectors:

```
0 : .1 : 1
```

produces the vector

```
[ 0 .1 .2 .3 .4 .5 .6 .7 .8 .9 1.0 ]
```

However, in the general case, getting the colon operator to work properly is surprisingly subtle, and quite wound up with the properties of floating-point arithmetic. Simply determining the correct number of elements in the vector is a subtle exercise in roundoff error. For some filtering applications, a colon operator with endpoints $-x$ and x must be exactly symmetrical around the origin, to prevent systematic buildup of roundoff error: contemplating the difficulties with

```
-2*pi : pi/10 : 2*pi
```

where pi is our favorite transcendental will help you appreciate why the colon operator takes several pages of code.

Functions may be defined with a keyword and syntax that gives names to the input and output arguments. A function is written into a file with the extension .m; if the file called xxx.m, and is written into a directory on the search path of the MATLAB interpreter, a function called xxx has just been defined. Note that the function name is derived from the name of the file containing the definition--while the syntax for function declarations allows a name to be specified, this name is ignored. For obvious reasons, these files are called *M-files*.

Mexfiles and the MATLAB Compiler

MATLAB has coexisted with FORTRAN and C code from its earliest days. So pressing was this need that MATLAB has supported dynamic linking and execution for many years, often orginally through special code written before this feature was available as part of standard operating systems. These dynamically loadable "MATLAB executable" files are called *mexfiles*.

An *external interface* was defined to allow Fortran and C programs to be written, compiled, and then dynamically loaded into MATLAB and executed. This interface consists primarily of a set of library functions and an associated header file that allow the Fortran and C programs to determine the dimensions of an array argument, find the real and imaginary parts, construct output arrays, *etc*. Typically, mexfiles consist of a "wrapper" that reads the MATLAB data structures, finds their pieces, and puts together the MATLAB

results, and one or more core functions that do the desired computation, often lifted with little change from an existing Fortran or C application.

Mexfiles work like M-files; if a file named `xxx.mex` is on the search path of the interpreter, it is taken to be the definition of the function `xxx`. (Actually, the suffix depends on the system the file is compiled on--for Sun 4's, the actual suffix is `mex4`.). If there is both an `xxx.m` and an `xxx.mex`, the mexfile wins.

The MATLAB compiler compiles M-files into mexfiles, primarily to obtain improved execution speed--when an M-file contains few matrix or vector operations, interpreter overhead can be a considerable fraction of the total runtime, so speedups of orders of magnitude are possible. In writing the compiler, it was a great advantage that there was already an existing dynamically loadable format for mexfiles that could be used. Also, while compiling an M-file to run without the interpreter is sometimes useful (and, in fact, MATLAB supported a toolkit for embedded control that compiled a limited class of M-files into C in a standalone environment), our target for the compiler was to turn M-files into mexfiles, preserving the flexibility and power of the interpeter.

MATLAB's Overhead and How to Remove It

To understand how the compiler speeds up programs, we need first to understand where MATLAB spends its time. The traditional rule of thumb suggests that the "cost" of interpreting is an order of magnitude in execution speed, but this rule of thumb is very misleading when applied to MATLAB. The data objects in MATLAB can be very large, containing thousands of elements. The fundamental operations on these data objects (for example, matrix multiplication) and many of the standard library operations (such as solving linear equations) are built into the interpreter, and use state-of-the-art algorithms. Since these operations can perform millions of floating point operations the overhead of interpretation can be insignificant in these cases.

However, when the MATLAB function consists largely of scalar operations, the overhead can be considerable. Not only do the operations have to be interpreted, but storage needs to be allocated for the matrices, sizes need to be checked, etc. Looking at such a scalar MATLAB program, it appears at first that its translation into C or FORTRAN would be straightforward. For example, consider the simple function that creates an n-element vector of squares:

```
function a=squares1(n)
for i=1:n
   a(i) = i*i;
end
```

This is actually very poor MATLAB code, because each vector element assignment causes the array `a` to grow, leading to $n-1$ reallocations of the original space, each time a little larger. Many storage allocators display quadratic behavior with such an allocation pattern, since the previous values must be copied into the larger space for each element of the array. Far better is to allocate a row vector of the right size at the beginning of the loop, and then just stuff the elements:

```
function a=squares2(n)
a=zeros(1,n);
for i=1:n
    a(i) = i*i;
end
```

On a Sun SPARC 10, the second function interprets almost 15 times faster than the first when n is 5000. Not surprisingly, the majority of serious MATLAB functions preallocate such arrays.

However, the really expert MATLAB programmer would just write

```
a = (1:n).^2;
```

which is in turn nearly sixty times faster than `squares2`. (`.^` is an operator that performs the power operation elementwise across the array, calling built in functions).

When the two squares functions are compiled, the generated C code also has to respect the semantics of growing the array with every assignment. The compiled C code for `squares1` runs about 4.5 times faster

than the interpreted code for $n = 5000$, due in large part to a somewhat better reallocation scheme for growing the arrays. For `squares2`, the compiled code runs about six times faster than the interpreted code. The C code generated has to check each reference and assignment to ensure that it is in range--a special 'fast' flag allows this subscript checking code (and the resulting semantics of growing the array) to be suppressed, consistent with the spirit of most C code. When this is used, the time is cut by another factor of 6.5, bringing the code quality close to the optimal MATLAB solution. Summarizing the times (in sec.) for n=5000:

```
4.5824  squares1, interpreted
1.0079  squares1, compiled
0.2622  squares2, interpreted
0.0422  squares2, compiled
0.0065  squares2, compiled with 'fast' flag
0.0056  (1:5000).^2
```

Type Imputation

When the `squares2` function is compiled, `i` and `n` are allocated to C integers, and `a` is known to be a real matrix. This saves a great deal of overhead, since the integers are allocated on the stack and no dynamic allocation is needed. Correctly figuring out the "true" types for a MATLAB program is the key part of compiling it effectively. This is especially true when we attempt to determine whether a variable is a scalar, and whether a variable is non-complex. For example, even something as straightforward as `i*i` could take a huge overhead if it was viewed as a matrix multiplication of two complex matrices--hundreds of instructions of overhead would surround the one multiplication that really mattered.

There are three main methods used to gain information about MATLAB types, which we can characterize "bottom up", "top down", and "recursive". Bottom up methods start with a simple known type and follow its possible transformations. For example, in `squares2`, the expression `1:n` is an array of integers, so the `for` statement that defines `i` always sets `i` to an integer. Knowing that `i` is an integer, `i*i` must be, and thus `a(i)` is. Since the elements of `a` are only set to integer values, the array `a` must be an integer array.

The bottom up method is made a bit more complicated because of the dynamic typing of MATLAB. The variable `i` may be the square root of -1 at one place in the program, an integer at another, a string at a third, and a full complex matrix at a fourth use. This means that we cannot have a traditional symbol table that tracks "the" type of `i`; we must associate a type with every use of `i`, and follow the implications of this through the program flow.

An interesting situation arises when we have iterative control flow. For example, suppose the variable `X` is set in three places:

```
X = 1;

for ...
    . . .
    X = X + 1;
    . . .
    X = 2*X;
end
```

It is reasonably clear that `X` is always going to be an integer. However, the `X` on the right side of `X=X+1` has its value set in two places: the initial `X=1`, and the `X=2*X` from the previous loop iteration. Since we know nothing about the type of the previous `X`, we tentatively make `X` an integer. Continuing in this way, we see that `X` always retains an integer value. Had the third equation been `X=2.5*X`, we would have had to upgrade the type of `X` from integer to floating point when we had completed the analysis.

This kind of analysis is well known in compiler writing circles (but not too well known outside), and a well-known technique is used to address this. We analyze the control flow for the program and associate with every variable a list of expressions that might define it (so-called *use-def* chains). When analyzing the type of the `X` on the left side of `X=X+1`, we see that it is set from the right hand side of the assignment, so

we need to understand the type of the right-hand X. There are two possible definition points for this X: X=1 and X=2*X. The first definition point has a known type, integer scalar. The second requires us to recursively ask about the type of the left side of X=2*X. This in turn requires that we ask about the right side X of X=2*X. And this has a single definition point, the left side of X=X+1. At this point, we appear to have reached an impasse, but in fact we are nearly done. As we investigate each of these types, we mark each definition of the variable we are recursively examining by setting a flag. When we encounter a definition with the flag set, we know (or, rather, those gifted in such matters can prove) that nothing new will be added from this definition, so we need take no action. At this point, we can unwind the recursion, recognizing that nothing other than integers is ever seen in the type structure for X, so X is an integer everywhere.

This kind of analysis (called a "least fixed point" by the theorists) is easy to apply when the type structure has a natural way of joining two types: for example, joining a floating point number and an integer array gives a floating-point array. We do this analysis with a type hierarchy consisting of integer, float, and complex, and a shape hierarchy consisting of scalar, vector (array with one dimension equal to 1), and array.

The second kind of type imputation arises from top down constraints, e.g., a variable is used in such a way that it implies things about its type. As an example, consider

```
function a=f(x)
    . . .
    y(3) = 4 + x;
    . . .
```

Here, the input variable might *a priori* be any shape, but we know that y(3) is a scalar, so 4 + x must be, so x must be. Unlike the previous case, this type information is not built bottom up from components, but is constrained top down by its uses. As another example, in the squares2 function, we recognize that the arguments to the "zeros" function are integers, so the input n must be.

The challenge comes in correctly blending these two forms of type imputation. For example, suppose we have

```
A = B * C;
```

and we know that A is scalar. Then, if we know that B is scalar, we can conclude that C must be scalar. This turns into a top-down requirement on C. If it later develops that we have underestimated the type of B, and B can be a vector, than we must also reexamine C as well.

The third form of type imputation is recursive examination of functions. MATLAB has several hundred library functions in reasonably common use, everything from bookeeping functions like zeros and size to mathematical functions like bessel to decompositions like lu and packages to do entire digital filter and control theory design. These functions are a rich and vital source of type information, and few MATLAB functions go for very long without calling another function, so it is essential not to lose information. Compounding the difficulty is the large number of built in functions.

Because the compiler is viewed as being called from the interpreter the interpreter search path is available, so we can recursively determine the type of all functions called by a given M-file (except for built-ins, that must be kept in a table). To avoid doing this multiple times, we save the computed output types for a given function called with a given number of input and output arguments. For built in functions, we have a table that makes use of the fact that, for most MATLAB functions, the type and shape of the output of a function is related to the type and shape of the inputs. Since most mathematical operations do not cause inputs to become less complicated, the type system we chose is one that relates the types and shapes of the outputs to the types and shapes of the inputs, together with a "lower bound" type and shape.

For example, the absolute value function abs always returns real values, but the shape of the result is the same as the shape of the input. The function prod takes the product of the rows of a matrix or the elements of a vector. The type is the same as the type of the input, but the dimension is one less: prod of a matrix yields a vector, while prod of a vector yields a scalar.

This simple but effective calculus of types is sufficient to handle nearly all built in functions accurately. When a function is used, it is first searched for in the table. If it is not found, the M-file is referenced and the type of the outputs determined recursively. Because the types of the outputs of a function sometimes

depend on the number of input and output arguments, the type imputation system makes different table entries for differing numbers of arguments, and can "prune" the parse tree, recognizing conditionals that check the number of incoming or outgoing arguments and eliminating unused code from the function in a particular case.

Wrinkles and Blemishes

As with every language that has been useful and used for a decade, MATLAB has developed some quirks that need careful attention when compiling. For example, in common with many interpreters, MATLAB has an `eval` function that takes a string and evaluates it in the context of the calling function. This causes the compiler so many problems that we just give up on it. To write an `eval` properly we would need to access all local variables and be able to set them. More seriously, `eval` could cause a variable to change type at runtime, totally breaking the assumptions of the compiler.

Unfortunately, there is one common use of `eval` that is reasonably common: using it to simulate a `varargs` functionality in MATLAB. There is some hope that we might recognize this special case and compile it efficiently, but there are also language changes afoot to make this feature a first-class part of the language, so compiling good code will become easy.

Another area where the compiler has difficulty with straightforward code is in dealing with the semantic ambiguity of subscripting. In an expression like

```
A(U) = X;
```

where `U` is an index vector, such vectors may either have values in the range from 1 to the length of `A`, or may have values that are 0 or 1 (as mentioned above), selecting a subvector through a boolean condition. The compiler must, in general, call a function and generate a temporary index vector if needed. In most cases, it should be possible to recognize at compile time which of these cases is present, but this is not currently done in the compiler.

Strings is another area where the compiler struggles. A string datatype was conspicuously missing from the type system described above. Luckily, most MATLAB applications spend little time in string processing. The difficulty comes from many library functions that allow certain arguments to be either strings or values. It can be very difficult to determine at compile time whether such an argument is really a string. This means that we need to carry around a dynamic "string flag" with our matrix structure, as MATLAB does. Given that we need do this anyway, there is little incentive to do a better job recognizing strings at compile time, since string handling rarely accounts for more than a couple of percent of runtime in MATLAB applications.

The operation of removing elements from a vector is another where the compiler needs to be careful in generating code. In MATLAB, an expression like

```
A(U) = X;
```

has totally different semantics if `X` is the empty array than if `X` is nonempty. This would *a priori* require the compiler to insert a test and two totally different pieces of code around a large class of such assignments. In practice, virtually 100% of the time when it is desired to remove elements, the matrix `X` is written as explicitly empty. The compiler chooses in this case to restrict the language compiled: we will just not correctly compile the general case when `X` is empty (more precisely, in most cases we will complain dynamically that the dimensions do not agree across the assignment...). This language restriction has caused no problem up to the present.

Unexpected Spinoffs

One interesting result of the compiler project has been a continual flow of bug reports about library routines, some of which have been in constant use for many years. The compiler has found some obsolete but still tolerated usage, some inefficiencies, some failures to check input arguments, and a couple of improper uses (for example, uses of the library routine `size` where `length` was intended).

The compiler is slowly growing a set of *lint*-like features to comment on MATLAB constructions. As an example, consider the expressions

```
if A == B
```

and

```
if A ~= B
```

in MATLAB. The first does pretty much what you would expect: the two matrices must agree in shape (or one must be a scalar), and the result of the equality operator is a matrix of 0's and 1's, describing the equality or inequality on an elementwise basis. The semantics of the MATLAB if statment is that the boolean argument is examined, and the positive consequent is taken only if every element of the boolean array is one. So there are no surprises.

The second expression would appear to behave in a similar fashion, but in practice is almost always a bug when `A` or `B` is nonscalar. A boolean array is constructed with 1's if the corresponding elements are unequal, but the if passes only if `all` elements of `A` differ from the corresponding elements of `B`, a decided surprise. Good MATLAB style should really demand one of the library functions `all` or `any` in this case (or, for two dimensional arrays, `all(all(A==B))` and `any(any(A~=B))`. When the compiler is invoked in verbose mode, users are told about any conditionals that appear to have nontrivial dimension (this also helps the compiler writer find places where the type imputation has missed a trick).

An Example

A prototypical example of the effective use of the MATLAB Compiler is provided by the solution of a tridiagonal system of linear equations. A tridiagonal linear system is n equations in n unknowns where the j-th equation explicitly involves only three unknowns, x_{j-1}, x_j, and x_{j+1}.

$$
\begin{aligned}
b_1 x_1 + c_1 x_2 &= d_1 \\
a_1 x_1 + b_2 x_2 + c_2 x_3 &= d_2 \\
&\cdots \\
a_{j-1} x_{j-1} + b_j x_j + c_j x_{j+1} &= d_j \\
&\cdots \\
a_{n-1} x_{n-1} + b_n x_n &= d_n
\end{aligned}
$$

The data for the problem consists of two vectors, b and d, of length n, and two vectors, a and c, of length $n-1$. The resulting solution, x, is then a vector of length n.

There are several ways to solve a tridiagonal system in MATLAB. One could create a full, n-by-n matrix, T, with a, b and c on the subdiagonal, diagonal and superdiagonal, and then use the linear system solution operator:

```
x = T\d
```

As a function of the order, n, this approach requires n^2 storage and n^3 time, so it is impractical for n larger than a few hundred. It is much more efficient to create the sparse form of the same matrix and use the same backslash operator. The time and storage requirements for the sparse linear equation solver are both linear in n, so this approach is reasonable for much larger systems.

Alternatively, one could use the following M-file. This has optimal storage efficiency because it involves only the relevant vectors. But it is slower than the sparse matrix approach because, although the input data and the output result are all vectors, the computation itself cannot be "vectorized". Each iteration of the loops involves only a few scalar operations. (This sample M-file can only be used when the elements of the diagonal coefficient vector, b, are nonzero and, in fact, are large enough that no pivoting for numerical stability is required during the elimination processes.)

```
function x = tridi(a,b,c,d)
%TRIDI  Solve tridiagonal system of equations.
%  x = TRIDI(a,b,c,d) solves the system of linear
%  equations x = T\d where T is the tridiagonal
%  matrix with a on the subdiagonal, b on the diagonal,
%  and c on the superdiagonal, and d is the right hand
%  side.  The input vectors b and d, and the output
%  vector x, all have the same length, say n, and the
%  input vectors a and c have length n-1.

n = length(b);
x = zeros(n,1);

for j = 2:n
    p = a(j-1)/b(j-1);
    b(j) = b(j) - p*c(j-1);
    d(j) = d(j) - p*d(j-1);
end

x(n) = d(n)/b(n);
for j = n-1:-1:1
    x(j) = (d(j)-c(j)*x(j+1))/b(j);
end
```

For this M-file, the compiler's most difficult task is determining that the variables `j` and `n` are integer scalars. This conclusion is possible here because `n` is the output of the length function and `j` is involved in loops which start at integer values and have integer increments. The result is the crucial declaration

```
int j,n;
```

in the resulting C program.

The compiler generates two versions of the body of the function, one for real arithmetic and one for complex arithmetic. Here is the real version.

```
for( j = 2; j <= n; j = j + 1 )
   {
      p = ((a.pr[((j-1)-1)]) / (b.pr[((j-1)-1)]));
      b.pr[(j-1)] = ((b.pr[(j-1)]) - (p * (c.pr[((j-1)-1)])));
      d.pr[(j-1)] = ((d.pr[(j-1)]) - (p * (d.pr[((j-1)-1)])));
   }

   x.pr[(n-1)] = ((d.pr[(n-1)]) / (b.pr[(n-1)]));
   for( j = (n - 1); j >= 1; j = j -1 )
   {
      x.pr[(j-1)] = (((d.pr[(j-1)]) -
         ((c.pr[(j-1)]) * (x.pr[((j+1)-1)]))) / (b.pr[(j-1)]));
   }
```

In the following graph, the circles are the actual measured speedup and the solid line is obtained by fitting the measured execution times with linear functions of n and then taking the ratio. We see that, as n increases, the compiled version approaches a speedup of 150 over the interpreted version.

Summary

It is very rewarding to write a compiler that speeds programs up by orders of magnitude, especially when the incoming programs are of practical importance. The compiler has not only made MATLAB a more useful product, but has also encouraged an examination of the language and many of the existing programs from a new and interesting perspective.