MATLAB Profiling and Performance Optimization

David Turner
Compiled vs Interpreted Languages

• General differences, not always the case.
• Compiled
  • Code is transformed by a compiler into machine level CPU instructions.
  • Advantages
    • Upfront cost of translating code to machine level instructions is paid once in advance at compile time.
    • Lower level control
    • Generally more efficient, lower overhead
    • Compilers can apply powerful optimizations at compile time
  • Disadvantages
    • Compiling\linking can be slow
    • Lower portability
• Examples: C\C++, Fortran, Pascal
C vs Assembly

```c
#include <stdlib.h>
int sub(int x, int y){
    return 2*x+y;
}

int main(int argc, char ** argv){
    int a;
    a = atoi(argv[1]);
    return sub(argc,a);
}
```

```assembly
.text:00000000  _sub:           push   ebp  
.text:00000001  mov   ebp, esp
.text:00000003  mov   eax, [ebp+8]
.text:00000006  mov   ecx, [ebp+0Ch]
.text:00000009  lea   eax, [ecx+eax*2]
.text:0000000C  pop   ebp
.text:0000000D  retn

.text:00000010  _main:         push   ebp
.text:00000011  mov   ebp, esp
.text:00000013  push   ecx
.text:00000014  mov   eax, [ebp+0Ch]
.text:00000017  mov   ecx, [eax+4]
.text:0000001A  push   ecx
.text:0000001B  call  dword ptr ds:__imp_atoi
.text:00000021  add   esp, 4
.text:00000024  mov   [ebp-4], eax
.text:00000027  mov   edx, [ebp-4]
.text:0000002A  push   edx
.text:0000002B  mov   eax, [ebp+8]
.text:0000002E  push   eax
.text:0000002F  call  __sub
.text:00000034  add   esp, 8
.text:00000037  mov   esp, ebp
.text:00000039  pop   ebp
.text:0000003A  retn
```
Compiled vs Interpreted Languages

• Interpreted Languages
  • Code is transformed into some intermediate representation (IR, huh?) and executed by an interpreter program
  • IR can be many things:
    • Abstract syntax tree – like a sentence diagram for computer languages
    • Most often: byte-code – low level code that can be run on a virtual machine
  • Advantages
    • Easier to develop an interpreter than a compiler
    • Platform independence
    • Dynamic typing
    • Reflection
  • Disadvantages
    • Slower
    • Dynamic typing
    • Easier to reverse engineer
• Examples
  • MATLAB, Python, Java, R, Ruby, Perl, JavaScript
  • What about Just in Time compiling (JIT)?
What language is MATLAB written in?

• As far as I can tell ...
  • Java (for the interface)
  • C++ (for most of the computational library)
  • FORTRAN (for matrix stuff, LAPACK, BLAS)

• Why care?
  • Language design can impact how we need think about optimizing it!
Optimization: Rules of thumb

• Rule One: Don’t do it.
• Rule Two: Don’t do it ... yet.
• Rule Three: Profile before optimizing

"We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%"

Donald Knuth.
Optimization Trade-offs

• Optimization can often (not always) have trade-offs:
  • Code readability
  • Code maintainability
  • Portability
  • Time
  • Performance – penalty in special case in return for benefit in another
Profiling

• What is profiling?
  • Profilers automatically instrument your code to measure time or memory spent within each function\command\instruction.
  • Useful for determining performance bottlenecks
  • Can provide insight on where to invest time in performance optimization.
  • Not good for benchmarking! Profiling, like debugging, will usually have a high performance overhead.
Before you profile!

• If possible: profile the real use case
  • Real data, real sizes, end-to-end from a user perspective
MATLAB Profiler

- MATLAB has a built-in profiler! Invoke with `profile` action.

<table>
<thead>
<tr>
<th>Option</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>on</td>
<td>Start the Profiler, clearing any previously recorded profile statistics.</td>
</tr>
<tr>
<td>off</td>
<td>Stop the Profiler.</td>
</tr>
<tr>
<td>resume</td>
<td>Restart the Profiler without clearing previously recorded statistics.</td>
</tr>
<tr>
<td>clear</td>
<td>Clear the recorded statistics.</td>
</tr>
<tr>
<td>viewer</td>
<td>Stop the Profiler and display the results in the Profiler window. For more information, see <a href="#">Profile to Improve Performance</a>.</td>
</tr>
<tr>
<td>info</td>
<td>Stop the Profiler and return a structure containing the results.</td>
</tr>
<tr>
<td>status</td>
<td>Return a structure with the Profiler status information.</td>
</tr>
</tbody>
</table>
Optimization

• Now that we have hopefully identified real bottlenecks in our code. Let's see what we can do to optimize them.
First, did you search Google?

• Seriously, look on Google and the MATLAB File exchange. Just type something like:
  • “fast <insert name of algorithm here> matlab”
  • or “slow <insert name of algorithm here> matlab”

• Examples
  • Fast Median:
    https://www.mathworks.com/matlabcentral/fileexchange/29453-nth-element
  • Fast Gaussian Blur

https://www.mathworks.com/matlabcentral/fileexchange/50848-fast-guassian-blur
Second, avoid these MATLAB Commands in bottlenecks

• Functions that query state, runtime introspection is slow:
  • `which`, `whos`, `exist(var)`, and `dbstack`
• MATLAB parsing and expression evaluation is slow
  • `eval`, `evalc`, `evalin`, and `feval(fname)`
• Manual memory management
  • Especially `clear all`, `clear functions`, or `clear classes`
• `cd`, `path`, and `rmpath`
  • Causes code recompilation (JIT)
• `ismember`
Pre-allocation: Memory Management

• MATLAB, and most interpreted languages manage memory for us, this is a blessing and a curse.
  • Advantages
    • Less boilerplate code
    • Less memory leaks
    • No double free bugs
    • No dangling pointers
    • Efficient implementations of persistent data structures
  • Disadvantages
    • Overhead
    • Unpredictable – although MATLAB is pretty good here
    • Easy to lose track of variable/object size in memory
Pre-allocation: Bad Examples

function data = example1_preallocate
% Example code from:

% all three different variables are growing inside the loop
% and all three are underlined in the MATLAB Editor
data2 = [];
data3 = [];
for idx = 1:100
    data1(idx) = fetchData();
data2(end+1) = fetchSomeOtherData();
data3 = [ data3 fetchYetMoreData() ];
end

data = { data1, data2, data3 };
end
No Pre-allocation: Why is it slow sometimes?

- Matrices are required to be non-fragmented (continuous memory)
- What happens under the hood when matrix needs to be grown?

```
Memory
<table>
<thead>
<tr>
<th>data1</th>
<th>data2</th>
<th>data3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Free</td>
<td></td>
<td>Space</td>
</tr>
</tbody>
</table>
```

Allocate

```
Memory
<table>
<thead>
<tr>
<th>data1</th>
<th>data2</th>
<th>data3</th>
</tr>
</thead>
<tbody>
<tr>
<td>new_data1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Copy

```
Memory
<table>
<thead>
<tr>
<th>data1</th>
<th>data2</th>
<th>data3</th>
</tr>
</thead>
<tbody>
<tr>
<td>data1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Free
Pre-allocation: Good Example

• If possible, allocate memory before hand and ensure it doesn’t grow within the loop:

```matlab
function data = example2_preallocate
% Example code from:

% all three different variables are growing inside the loop
% and all three are underlined in the MATLAB Editor
data1 = zeros(100,1);
data2 = zeros(100,1);
data3 = zeros(100,1);
for idx = 1:100
    data1(idx) = fetchData();
data2(idx) = fetchSomeOtherData();
data3(idx) = fetchYetMoreData();
end

data = { data1, data2, data3 };
end
```
Pre-allocation: Performance Comparison
Pre-allocation: Misunderstandings

• Pre-allocation has a cost. Make sure it is necessary.

```plaintext
function data = fillDataWithUnnecessaryPreallocation
    % note the Code Analyzer message
    % The value assigned to variable 'data' might be unused.
    data = zeros(1,100);
    data = fetchAllData();
end
```
Memory Management: Copy on write

- Functions receive references to arguments, unless they are modified!

```javascript
function foo(x, a, b)
    a(1) = a(1) + 12;
    y = a * x + b;
```

- `a` is modified, so it is copied!
- `x` and `b` are un-modified, so they are passed by reference!
Memory Management: Copy on write

• Members of structs are individually copied

```c
s.A = rand(3);
s.B = magic(3);
sNew = s;
sNew.A(3) = 14;
```

• s.B and sNew.B share memory, sNew.A and s.A do not
Memory Management: In place computation

- In place algorithms are those that process data without using auxiliary data structures (a small constant amount of memory overhead is allowed)

- Benefits don’t present themselves for small arrays.

- MATLAB’s JIT is getting better all the time, make sure you test things and don’t code in unnatural ways
## Memory Management: In-place computation

**function** `inplaceTest(x)`

% Call functions with either regular or in-place semantics.

% Call a Regular Function with the Same Left-Hand Side
\[ x = \text{myfunc}(x); \]

% Call an In-place Function with the Same Left-Hand Side
\[ x = \text{myfuncIP}(x); \]

% Call a Regular Function with a Different Left-Hand Side
\[ y = \text{myfunc}(x); \]

**function** `y` = `myfunc(x)`
\[ y = \sin(2^x.\^2+3^x+4); \]

**function** `x` = `myfuncIP(x)`
\[ x = \sin(2^x.\^2+3^x+4); \]
Memory Management: Use Appropriate Data Types

- Don’t use more than you need

<table>
<thead>
<tr>
<th>Class (Data Type)</th>
<th>Bytes</th>
<th>Supported Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>single</td>
<td>4</td>
<td>Most math</td>
</tr>
<tr>
<td>double</td>
<td>8</td>
<td>All math</td>
</tr>
<tr>
<td>logical</td>
<td>1</td>
<td>Logical/conditional operations</td>
</tr>
<tr>
<td>int8, uint8</td>
<td>1</td>
<td>Arithmetic and some simple functions</td>
</tr>
<tr>
<td>int16, uint16</td>
<td>2</td>
<td>Arithmetic and some simple functions</td>
</tr>
<tr>
<td>int32, uint32</td>
<td>4</td>
<td>Arithmetic and some simple functions</td>
</tr>
<tr>
<td>int64, int64</td>
<td>8</td>
<td>Arithmetic and some simple functions</td>
</tr>
</tbody>
</table>
Vectorization

- MATLAB is very fast with operations involving arrays and vectors.
- Vectorization is the process of revising loop-based, scalar oriented code to use matrix and vector operations.
- Vectorization MAY lead to concise and easily understandable code.
- In the past, vectorization was usually the go to approach speeding up loop based MATLAB code. MATLAB JIT is way better than it used to be though and now it can sometimes not be worth it. Test everything!
- Can lead to increased memory needs, not always.
Vectorization: Simple Examples

**Non-Vectorized**

```matlab
function y = example_nonvec(N)
    i = 0;
    ts = 0:.01:N;
    y = zeros(length(ts),1);
    for t = ts
        i = i + 1;
        y(i) = sin(t);
    end
```

**Vectorized**

```matlab
function y = example_vec(N)
    y = sin(0:.01:N);
```
Vectorization: Simple Examples

![Graph showing the comparison between No Vectorization and Vectorization with increasing N. The graph indicates that Vectorization reduces time significantly.]
Vectorization: Operators

- Placing a period (.) before the operators *, /, and ^ transforms them into array operators.

```matlab
V = zeros(10000,1);
for n = 1:10000
    V(n) = 1/12*pi*D(n)^2*H(n);
end
V = 1/12*pi*(D.^2).*H;
```
Vectorization: Array Expansion

• MATLAB 2016 and later added array expansion (or broadcasting for Python NumPy users). This largely replaces functionality covered by bsxfun.

• Array expansion can be more efficient because temporary storage is not created under the hood.

```
>> A = rand(3,3)
A =
 0.8147  0.9134  0.2785
 0.9058  0.6324  0.5469
 0.1270  0.0975  0.9575

>> ma = mean(A)
ma =
 0.6158  0.5478  0.5943

>> ma_expanded = ma(ones(3,1),:)
ma_expanded =
 0.6158  0.5478  0.5943
 0.6158  0.5478  0.5943
 0.6158  0.5478  0.5943

>> A - ma_expanded
ans =
 0.1989  0.3656  -0.3158
 0.2900  0.0846  -0.0474
 -0.4888  -0.4502  0.3632

```
Vectorization: Array Expansion

- MATLAB 2016 and later added array expansion (or broadcasting for Python NumPy users). This largely replaces functionality covered by bsxfun.

- Array expansion can be more efficient because temporary storage is not created under the hood.

```matlab
>> A = rand(3,3)
A =
    0.8147    0.9134    0.2785
    0.9058    0.6324    0.5469
    0.1270    0.0975    0.9575

>> ma = mean(A)
ma =
    0.6158    0.5478    0.5943

>> A - repmat(ma,3,1)
ans =
    0.1989    0.3656   -0.3158
    0.2900    0.0846   -0.0474
   -0.4888   -0.4502    0.3632

>> bsxfun(@minus,A,ma)
ans =
    0.1989    0.3656   -0.3158
    0.2900    0.0846   -0.0474
   -0.4888   -0.4502    0.3632
```

Somewhat Newer School Method!
Vectorization: Array Expansion

• MATLAB 2016 and later added array expansion (or broadcasting for Python NumPy users). This largely replaces functionality covered by bsxfun.

• Array expansion can be more efficient because temporary storage is not created under the hood.

\[
\begin{array}{ccc}
0.8147 & 0.9134 & 0.2785 \\
0.9058 & 0.6324 & 0.5469 \\
0.1270 & 0.0975 & 0.9575 \\
\end{array}
\]

\[
\begin{array}{ccc}
0.6158 & 0.5478 & 0.5943 \\
\end{array}
\]

\[
\begin{array}{ccc}
0.1989 & 0.3656 & -0.3158 \\
0.2900 & 0.0846 & -0.0474 \\
-0.4888 & -0.4502 & 0.3632 \\
\end{array}
\]
Vectorization: Array Expansion

- All singleton dimensions are expanded

Be careful. Used to be a common error in MATLAB codes. Now no error is generated!

- Extends beyond two dimensions
Vectorization: Array Expansion

- These operators/functions support array expansion currently:

<table>
<thead>
<tr>
<th>+</th>
<th>-</th>
<th>.*</th>
<th>./</th>
</tr>
</thead>
<tbody>
<tr>
<td>.\</td>
<td>.^</td>
<td>&lt;</td>
<td>&lt;=</td>
</tr>
<tr>
<td>=&gt;</td>
<td>==</td>
<td>~=</td>
<td></td>
</tr>
<tr>
<td>xor</td>
<td>bitor</td>
<td>bitand</td>
<td>bitxor</td>
</tr>
<tr>
<td>min</td>
<td>max</td>
<td>mod</td>
<td>rem</td>
</tr>
<tr>
<td>hypot</td>
<td>atan2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Vectorization: Logical Array Operations

• If statements can be vectorized as well.

```matlab
>> D = [-0.2 1.0 1.5 3.0 -1.0 4.2 3.14];
>> D >= 0
ans =
   0 1 1 1 0 1 1

>> D(D >= 0)
ans =
   1.0000  1.5000  3.0000  4.2000  3.1400

>> D((D >= 0) & (D < 3))
ans =
   1.0000  1.5000
```
Vectorization: Indexing

• Accessing single elements

```matlab
>> A = magic(4)
A =
  16   2   3  13
   5  11  10   8
   9   7   6  12
   4  14  15   1
>> A(4,2)
ans =
   14
```

• Linear indexing
  • MATLAB stores matrices in column major order

```matlab
>> A(:)'
ans =
   16   5   9   4   2  11   7  14   3  10   6  15  13   8  12   1
```
Vectorization: Indexing

Uses formula,

\[(j-1) \times d1 + i\]

Where \(\text{size}(A) = [d1 \ d2]\)

Note: These functions can be slow, they are generalized to \(N\) dimensions and perform bound checking. Can get a speedup by using formulas.
Vectorization: Other tips

- `repmat`

```matlab
>> A = [1 2; 3 4]
A =
    1  2
    3  4
>> repmat(A, [2, 2])
ans =
    1  2  1  2
    3  4  3  4
    1  2  1  2
    3  4  3  4
```
Vectorization: Other tips

• reshape

```python
>> A = magic(4)
A =
     16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1
>> reshape(A, [2, 8])
ans =
     16     9     2     7     3     6    13    12
     5     4    11    14    10    15     8     1
```
Vectorization: Other tips

- `permute`

```matlab
a = rand(1,2,3,4);
size(permute(a,[3 2 1 4])) % now it's 3-by-2-by-1-by-4.
```
Vectorization: Other tips

• `ndgrid`

```matlab
>> [X Y Z] = ndgrid(1:2,1:2,1:2)
X(:,:,1) =
    1  1
    2  2
X(:,:,2) =
    1  1
    2  2
Y(:,:,1) =
    1  2
    1  2
Y(:,:,2) =
    1  2
    1  2
Z(:,:,1) =
    1  1
    1  1
Z(:,:,2) =
    2  2
    2  2
```
Vectorization: A Lesson

• Lets put it together!

function L = example2_nonvec(N)

    delt = 5/10;
    A = rand(N, N, N);
    L = zeros(N^3, 4);
    ind = -250*delt:delt:250*delt;

    for di=1:N
        for dj=1:N
            for dk=1:N
                for dk=1:N
                    L((N^2)*(di-1)+N*(dj-1)+dk,:) = ...
                        [ind(di), ind(dj), ind(dk), A(di, dj, dk)];
                end
            end
        end
    end
end
Vectorization: A Lesson

• Vectorize! Who cares if its unreadable!

```matlab
function L = example2_vec(N)

delt = 5/10;
A = rand(N,N,N);
ind = -250*delt:delt:250*delt;

X = repmat({1:N},3,1);
[X, Y, Z] = ndgrid(X{:});
L = [ind([Z(:) Y(:) X(:)]) reshape(permute(A,[3 2 1]),[],1)];

end
```
Vectorization: A Lesson

• Vectorize! Who cares if its unreadable!

```matlab
function L = example2_vec(N)

    delt = 5/10;
    A = rand(N,N,N);
    ind = -250*delt:delt:250*delt;

    X = repmat({1:N},3,1);
    [X, Y, Z] = ndgrid(X{:});
    L = [ind([Z(:) Y(:) X(:)]) reshape(permute(A,[3 2 1]),[],1)];

end
```
Vectorization: A Lesson
Vectorization: A Lesson

• We have sacrificed code readability and maintainability for performance. Is this necessary?
Vectorization: A Lesson

• Simplify the loop

```matlab
function L = example2_nonvec(N)

delt = 5/10;
A = rand(N,N,N);
L = zeros(N^3,4);
ind = -250*delt:delt:250*delt;

for di=1:N
    for dj=1:N
        for dk=1:N
            L((N^2)*(di-1)+N*(dj-1)+dk,:) = ...
                [ind(di),ind(dj),ind(dk),A(di,dj,dk)];
        end
    end
end
```

```matlab
function L = example2_nonvec2(N)

delt = 5/10;
A = rand(N,N,N);
L = zeros(N^3,4);
ind = -250*delt:delt:250*delt;

cnt = 0;
for di=1:N
    for dj=1:N
        for dk=1:N
            cnt = cnt + 1;
            L(cnt,1) = ind(di);
            L(cnt,2) = ind(dj);
            L(cnt,3) = ind(dk);
            L(cnt,4) = A(di,dj,dk);
        end
    end
end
end
```
Vectorization: A Lesson

Why?
Vectorization: A Lesson

• MATLAB’s JIT engine is getting better. Sometimes it is able to compile the MATLAB code into machine code and avoid the interpreter altogether! We need to test things always.
• But this is annoying, a proprietary closed mystery JIT engine that decides to silently invoke itself randomly.
• Can we get around this without losing all that MATLAB does have to offer?
MATLAB executable: MEX

• MEX is a way to efficiently call C\C++\FORTRAN from MATLAB

• Advantages
  • Best of both worlds
  • Access to features of C\C++\FORTRAN
  • Speed

• Disadvantages
  • Learning new language
  • Mixed code base – software engineering overhead
  • Compiling\building
  • Loss of portability
  • Marshalling data back and forth
  • Memory management responsibility
  • Annoying to debug/profile
MEX: MATLAB “built-in” functions!

• Don’t feel bad! Mathworks does this all the time.
• MATLAB has the concept of built-in functions. These are just MEX functions. Most fast functions in MATLAB are built-in. To check a function just us type:

```matlab
>> type cumsum
'cumsum' is a built-in function.
```
MEX: Building

• Setup a compiler within MATLAB environment
  • Use command `mex -setup`
• Supported compilers
  • Linux
    • GCC C/C++ 6.3.x, GNU gfortran 6.3.x
  • Windows
    • Microsoft Visual C++ Family, Intel Parallel Studio XE for C\C++ or Fortran
  • Mac
    • Xcode, Intel Parallel Studio XE for Fortran
MEX: Write C code

• Start with pure C++ code

```c
void arrayProduct(double x, double *y, double *z, int n)
{
    int i;

    for (i=0; i<n; i++) {
        z[i] = x * y[i];
    }
}
```
MEX: Write C code

• Every C++ program has a main() function, MEX replaces this with mexFunction(), this is the entry point.

/* The gateway function */
void mexFunction( int nlhs, mxArray *plhs[],
                 int nrhs, const mxArray *prhs[])
{

• nlhs – The number of output arguments (lhs = left hand side)
• plhs – Array of pointers to the output arguments
• nrhs – The number of input arguments (rhs = right hand side)
• prhs – Array of pointers to the output arguments
MEX: Write C Code

- This is C, we need to declare every variable we are working with:

```c
double multiplier;    /* input scalar */
double *inMatrix;     /* 1xN input matrix */
size_t ncols;         /* size of matrix */
double *outMatrix;    /* output matrix */
```
# MEX: Write C Code

- Check our input arguments

```c
/* check for proper number of arguments */
if(nrhs!=2) {
    mexErrMsgIdAndTxt("MyToolbox:arrayProduct:nrhs","Two inputs required.");
}
if(nlhs!=1) {
    mexErrMsgIdAndTxt("MyToolbox:arrayProduct:nlhs","One output required.");
}

/* make sure the first input argument is scalar */
if( !mxIsDouble(prhs[0]) ||
    mxIsComplex(prhs[0]) ||
    mxGetNumberOfElements(prhs[0])!=1 ) {
    mexErrMsgIdAndTxt("MyToolbox:arrayProduct:notScalar","Input multiplier must be a scalar.");
}

/* make sure the second input argument is type double */
if( !mxIsDouble(prhs[1]) ||
    mxIsComplex(prhs[1])) {
    mexErrMsgIdAndTxt("MyToolbox:arrayProduct:notDouble","Input matrix must be type double.");
}
```
MEX: Write C Code

• Marshall input values, create output matrix, and call

/* get the value of the scalar input */
multiplier = mxGetScalar(prhs[0]);

/* create a pointer to the real data in the input matrix */
inMatrix = mxGetPr(prhs[1]);

/* get dimensions of the input matrix */
ncols = mxGetN(prhs[1]);

/* create the output matrix */
plhs[0] = mxCreateDoubleMatrix(1, (mwSize) ncols, mxREAL);

/* get a pointer to the real data in the output matrix */
outMatrix = mxGetPr(plhs[0]);

/* call the computational routine */
arrayProduct(multiplier, inMatrix, outMatrix, (mwSize) ncols);
MATLAB: Compile the MEX

• With all of the code presented in a single file called arrayProduct.c Run the command:

```
>> mex arrayProduct.c
Building with 'Microsoft Visual C++ 2013 Professional (C)'.
MEX completed successfully.
```

• If successful, this will create a file in the current working directory called arrayProduct.mexw64, or arrayProduct.mexa64 on linux. This is really just dynamic link library (DLL) on windows.
MATLAB: Use it

• With the .mexw64 file in the current directory or on the MATLAB path, we can simply run it like a normal function.

```matlab
>> B = arrayProduct(5, [1.5, 2, 9])
B =
    7.5000    10.0000    45.0000
```
MEX: Tips

• Be careful with types
  • C\C++\Fortran are statically type languages, check types before passing into mex code, or check within.

• Memory management – be careful with memory. If you are causing MATLAB to crash then you are probably screwing up with memory.

• Separate MEX API code from standard C\C++\Fortran code. This will make your other code more extensible and portable

• Debugging can be done, but it is trickier.

• OpenMP can be used within a MEX function to create multi-threaded code.
MATLAB: Parallelization – built in multithreading

Arrays and matrices
- Basic information: ISFINITE, ISINF, ISNAN, MAX, MIN
- Operators: +, -, *, ./, .\, .^, *, ^, \ (MLDIVIDE), / (MRDIVIDE)
- Array operations: PROD, SUM
- Array manipulation: BSXFUN, SORT

Linear algebra
- Matrix Analysis: DET, RCOND
- Linear Equations: CHOL, INV, LDL, LINSOLVE, LU, QR
- Eigenvalues and singular values: EIG, HESS, SCHUR, SVD, QZ

Elementary math
- Trigonometric: ATAN2, COS, CSC, HYPOT, SEC, SIN, TAN
- Exponential: EXP, POW2, SQRT
- Complex: ABS
- Rounding and remainder: CEIL, FIX, FLOOR, MOD, REM, ROUND
- LOG, LOG2, LOG10, LOG1P, EXPM1, SIGN, BITAND, BITOR, BITXOR

Special Functions
- ERF, ERFC, ERFCINV, ERFCX, ERFINV, GAMMA, GAMMALN

Data Analysis
- CONV2, FILTER, FFT and IFFT of multiple columns or long vectors, FFTN, IFFTN
MATLAB: Parallel – use on Princeton clusters

• Running jobs with a single node and single core (SBATCH –N 1 –c 1)
  • Invoke MATLAB command with arguments:
    • -singleCompThread -nodisplay -nosplash -nojvm

• Running a multi-core jobs, when SBATCH –c N, N is greater than 1
  • Invoke MATLAB without -singleCompThread and -nojvm
  • On Princeton RC clusters this should be enough. On other clusters check the result of the command maxNumCompThreads
  • If greater than the number of cores you requested the set maxNumCompThreads(N)
    • This will ensure that MATLAB doesn’t use more threads than you have requested
MATLAB: Parallel For Loops

• A parallel for loop can be useful if you have a slow for loop
  • Loop iterations take a long time to execute, make sure you have more iterations than cores
• A parallel for loop might not be useful if you have:
  • Highly vectorized code – vectorized code tends to already be multi-threaded because of built in commands
  • Loop iterations that are very short – parallel overhead
• A parallel for loop almost definitely won’t be useful if you have:
  • A need for iteration order guarantees
  • Any dependency between iterations
MATLAB: Parallel parfor Example

```matlab
function a = parfor_example(n, num_workers)

A = 500;
a = zeros(n);
parfor (i = 1:n, num_workers)
    a(i) = max(abs(eig(rand(A))));
end
```

num_workers not needed on Princeton RC clusters
MATLAB: Distributed Computing

• In my humble opinion, if you require truly distributed computing that involves inter-node communication, don’t use MATLAB.

• MATLAB has a distributed computing server but its license is cost prohibitive so it is rare to find on HPC clusters in my experience.

• MatlabMPI\pMatlab seems like another option, but I have never used it so I can’t comment.
GPU

Three main ways to use GPU for accelerating MATLAB code:

1. Use built-in MATLAB functions that support GPU capability
2. Using `arrayfun` to perform the algorithm on each element independently
3. Using the MATLAB/CUDA interface to run some existing CUDA/C++ code

<table>
<thead>
<tr>
<th>Key Functions and Classes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>gpuArray</code></td>
<td>Array stored on GPU</td>
</tr>
<tr>
<td><code>gather</code></td>
<td>Transfer distributed array or <code>gpuArray</code> to local workspace</td>
</tr>
<tr>
<td><code>gpudevice</code></td>
<td>Query or select a GPU device</td>
</tr>
<tr>
<td><code>GPUDeviceManager</code></td>
<td>Manager for GPU Devices</td>
</tr>
</tbody>
</table>
CPU vs GPU Example

CPU

```matlab
maxIterations = 500;
gridSize = 1000;
xlim = [-0.748766713922161, -0.748766707771757];
ylim = [0.123640844894862, 0.123640851045266];

% Setup
t = tic();
x = linspace( xlim(1), xlim(2), gridSize );
y = linspace( ylim(1), ylim(2), gridSize );
[xGrid,yGrid] = meshgrid( x, y );
z0 = complex(xGrid, yGrid);
count = ones(size(z0));

% Calculate
z = z0;
for n = 0:maxIterations
    z = z.*z + z0;
    inside = abs(z) <= 2;
count = count + inside;
end
count = log(count);
```

GPU

```matlab
maxIterations = 500;
gridSize = 1000;
xlim = [-0.748766713922161, -0.748766707771757];
ylim = [0.123640844894862, 0.123640851045266];

% Setup
t = tic();
x = gpuArray.linspace( xlim(1), xlim(2), gridSize );
y = gpuArray.linspace( ylim(1), ylim(2), gridSize );
[xGrid,yGrid] = meshgrid( x, y );
z0 = complex(xGrid, yGrid);
count = ones(size(z0), 'gpuArray');

% Calculate
z = z0;
for n = 0:maxIterations
    z = z.*z + z0;
    inside = abs(z) <= 2;
count = count + inside;
end
count = log(count);
count = gather(count); % Fetch the data back from the GPU
```
GPU Example

3.5x faster on GeForce GTX 960M vs Intel Core i7 6700HQ CPU @ 2.6 GHz 4-core
GPU: Even Better

• Algorithm is operating equally on every input. Put the main process in a helper function and call it using arrayfun. For GPU array inputs, the function used with arrayfun gets compiled into native GPU code.

```matlab
% Calculate
count = arrayfun(@processMandelbrotElement, ... 
    xGrid, yGrid, maxIterations);

% Show
count = gather(count); % Fetch the data back from the GPU

function count = processMandelbrotElement(x0,y0,maxIterations)
    z0 = complex(x0,y0);
    z = z0;
    count = 1;
    while (count <= maxIterations) && (abs(z) <= 2)
        count = count + 1;
        z = z*z + z0;
    end
    count = log(count);
end
```

39x faster than CPU Version.
GPU: Use CUDA

• We can do even better by writing our own CUDA code and invoking it from MATLAB. This is too in depth for this tutorial though, if interested see:

Sources

1. **Loren on the Art of MATLAB**

2. MATLAB Documentation

   http://www-h.eng.cam.ac.uk/help/tpl/programs/Matlab/tricks.html