Writing High Performance m-files

OctConf 2015
Darmstadt, Germany
Sep. 21, 2015
Overview

- Motivation for speed optimization
- Experimental approach
  - Design, Build, Test
- Design for performance
  - Structure of Octave
  - 4 General Performance Principles
- Testing performance
  - Goal and pitfalls of benchmarking
  - Benchmarking approaches in Octave
Don’t Optimize

- Life is short,
- Death is long,
- Spend your time wisely
Really, Don’t Optimize

- Base Google salary in Silicon Valley is $128K, approximately $65/hr
- More expensive to learn and implement optimization techniques than to
  - Buy faster CPUs
  - Buy more memory
  - “Rent” more hardware (AWS)
When to consider performance?

1) Doesn’t complete in a reasonable period
2) Real-time control
3) Core developer
Coding Priorities

1. Get it working
2. Make it readable

These two goals are often in conflict with better performance.
Engineering Performance

- Experimental approach to better performance
Structure of Octave

- Octave is an interpreted language
- Octave is a thin translation layer between m-files and powerful existing code libraries

```
X = fft (x);
DC = X(0);
...
```
Core Interpreter Operations

\[ y = \sin (x); \]

1. Parse m-file text
2. Gather inputs, outputs
3. Dispatch to correct library
A * B’

• Previously computed as 2 operations
  1. TMP = Transpose (B)
  2. ANS = A * TMP

• Now dispatched to BLAS as a single function call with appropriate flag settings

• Performance increase of ~30%
4 General Design Principles

1. Avoid parsing/translation
2. Use built-in functions
3. Manage memory
4. Stay within interpreter
Benchmarking
a.k.a. Testing

- Runtime is a complex function of multiple inputs

\[ \text{RunTime} = f \left( x_1, x_2, x_3, \ldots, x_n \right) \]

- Objective is to calculate partial derivative with respect to just code changes

\[ \frac{\partial}{\partial x_k} f \left( x_1, x_2, x_3, \ldots, x_n \right) \]
Benchmarking Best Practices

- Use data sets that match expected inputs
- Disable CPU frequency scaling
- Run on lightly loaded computer with enough memory to prevent swapping
- Run benchmarks multiple times; Use average and standard deviation to assess quality of benchmarking data
Pareto Principle

- The 80/20 rule
- Nearly always, 1 or 2 issues are the cause of all problems
- Use Pareto as a stopping criterion for optimization
Benchmarking in Octave

- tic / toc
- cputime
- profiler
Example BM Script

N = 50;
sz = [40, 40];

x = rand (sz);
y = zeros (sz);

bm = zeros (N, 1);

for i = 1:N
    tic;
    y = ftan (x);
    bm(i) = toc;
endfor
Sample function to be optimized

```matlab
function y = ftan (x)
    for i = 1:numel(x)
        y(i) = sin(x(i)) / cos(x(i));
    endfor
endfunction
```
Baseline Performance

- Mean = 0.148
- STD = .001
arrayfun ()

- Eliminates loops for single-valued (non-vector) functions

```matlab
fcn = @(x) sin (x) / cos (x);

for i = 1:N
    tic;
    y = arrayfun (fcn, x);
    bm(i) = toc;
endfor
```
arrayfun () performance

- Mean = 0.1220
- STD = .0006
- % change = -18%
- Not bad, but not outstanding
- In the future, this may improve
Vectorization

- Parse just once, eliminates multiple translations
- “Win-Win”
  - Increases performance drastically
  - Makes code more readable
Vectorized \( \text{ftan}() \)

```matlab
function y = ftan_vec(x)
    y = sin(x) ./ cos(x);
endfunction
```

- Remove looping structures
- Use vector operators, e.g., ‘./’
Vectorized Results

- Mean = .00039
- STD = .00002
- % change = -99.7%
- Well worth doing
Principle 1: Avoid Parsing/Translation

- Loops are abysmally slow
  - Band-aids such as arrayfun or cellfun don’t really work
  - Vectorization is most important strategy
    - Speeds up code and makes it more readable
    - ~100X improvement
Principle 2: Use Built-in Functions

- Don’t re-invent the wheel
- Built-in functions are often in a compiled language which is much faster
- Any m-file implementations have been optimized more than you can easily accomplish
Benchmark \texttt{tan()} \\

\begin{verbatim}
function y = ftan_tan(x)
    y = tan(x);
endfunction
\end{verbatim}

- Mean = 0.00028
- STD = 0.00002
- % change over ftan = -99.8%
- % change over vectorized ftan = -26%
## Benchmark Summary

<table>
<thead>
<tr>
<th>Function</th>
<th>Relative Speed</th>
</tr>
</thead>
<tbody>
<tr>
<td>tan ()</td>
<td>1</td>
</tr>
<tr>
<td>vectorized ftan</td>
<td>1.36</td>
</tr>
<tr>
<td>arrayfun</td>
<td>436</td>
</tr>
<tr>
<td>looping ftan</td>
<td>529</td>
</tr>
</tbody>
</table>
Memory Management

- General Problem
  - Octave hides details like garbage collection
  - BUT, Octave is not an optimizing compiler
  - Still necessary to manage memory and avoid bad code constructs
- **Must** have enough memory to avoid swapping
Growing Arrays

- Forces multiple memory allocations, fragments system memory

```matlab
function y = ftan_mem (x)
    y = [];
    for i = 1:numel (x)
        y(end+1) = sin (x(i)) / cos (x(i));
    endfor
    y = reshape (y, size (x));
endfunction
```
Pre-Declare Arrays

• Single memory allocation

```matlab
function y = ftan_mem_declare (x)
    y = zeros (size (x));
    for i = 1:numel (x)
        y(i) = sin (x(i)) / cos (x(i));
    endfor
endfunction
```
Memory Benchmarking

<table>
<thead>
<tr>
<th>Method</th>
<th>RunTime</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array growth</td>
<td>.167</td>
</tr>
<tr>
<td>Pre-declared array</td>
<td>.143</td>
</tr>
<tr>
<td>% change</td>
<td>-14%</td>
</tr>
</tbody>
</table>
In-Place Operators 1

\[
A = A + 1
\]

is equivalent to

\[
\text{TMP} = A + 1
\]

\[
A = \text{TMP}
\]
In-Place Operators 2

A += 1

Does not create a temporary array!
### In-Place Benchmarks

<table>
<thead>
<tr>
<th>Method</th>
<th>RunTime</th>
<th>% Change</th>
<th>Relative RunTime</th>
</tr>
</thead>
<tbody>
<tr>
<td>A = A + 1</td>
<td>.111</td>
<td>--</td>
<td>1</td>
</tr>
<tr>
<td>A++</td>
<td>.110</td>
<td>-1%</td>
<td>.99</td>
</tr>
<tr>
<td>++A</td>
<td>.111</td>
<td>0%</td>
<td>1</td>
</tr>
<tr>
<td>A += 1</td>
<td>.041</td>
<td>-60%</td>
<td>.40</td>
</tr>
</tbody>
</table>

- Octave core functions already use in-place operators
- Use built-in functions and get optimization for free
Copy-on-Write (COW)

- Octave conserves memory by using Copy-on-Write
- A copy of a variable, such as $y = x$, creates a link to the original variable without using additional memory
- Modifications to a copy of a variable, such as $y = y + 1$, require allocation of new memory
Accidental Memory Consumption

function retval = tst_cow (x)
    tmp = x + 1;
    retval = 2 * tmp;
endfunction

- Use 3*sizeof (x) memory to store x, tmp, and retval
- Minimum memory allocation of 2*sizeof (x) is possible through simple recoding
Avoiding COW I

- Strategy 1: Avoid COW by using a single intermediate variable for all calculations

```plaintext
function retval = tst_cow (x)
    tmp = x + 1;
    tmp = 2 * tmp;
    retval = tmp;
endfunction
```
Avoiding COW II

• Strategy 2: Avoid COW by using the output variable for intermediate calculations

```plaintext
function retval = tst_cow(x)
    retval = x + 1;
    retval = 2 * retval;
endfunction
```
Principle 3 : Manage memory

- Pre-declare large variables
- Clear large, unnecessary variables before calculations begin
- Use in-place operators
- Avoid accidental COW variables
4 General Design Principles

1. Avoid parsing/translation
2. Use built-in functions
3. Manage memory
4. Stay within interpreter
Performance Expectations

- Vectorization: ~100X
- Built-in Functions: ~2-100X
- Memory Management: ~25%
- Stay within interpreter: < 10%
What if it isn’t enough?

- Use the 80/20 rule
- Accelerate only the bottleneck
- Look at the external code interface in Appendix A