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

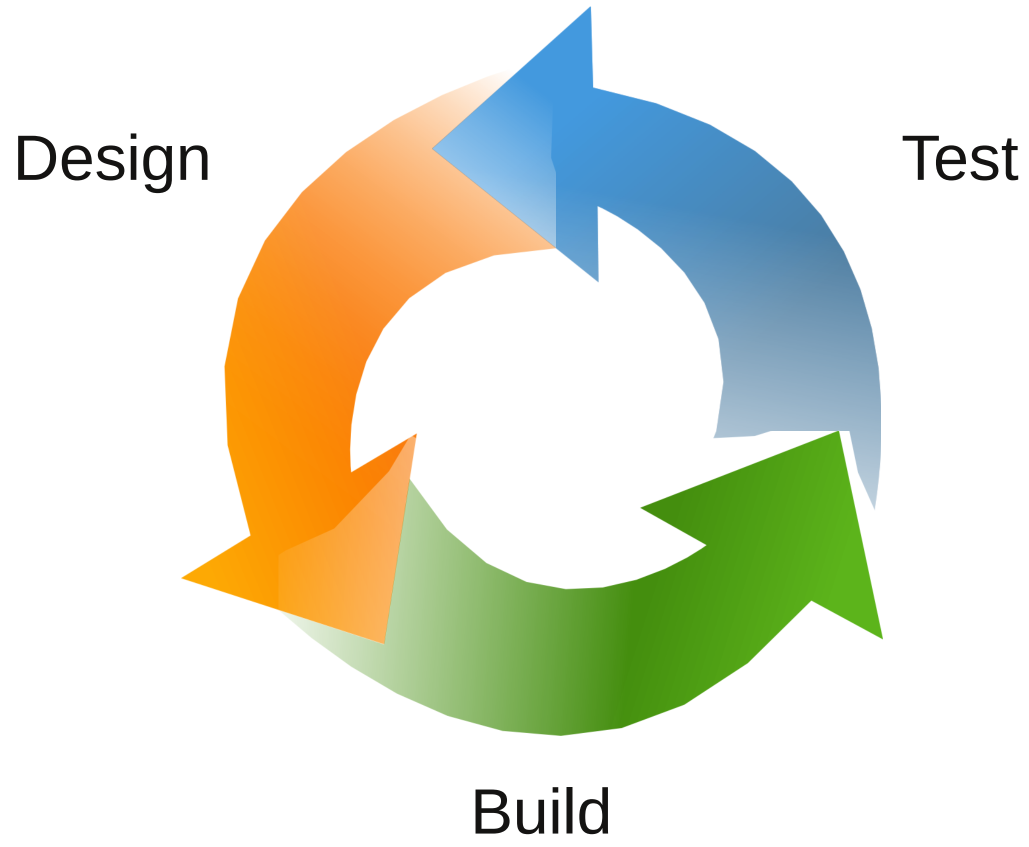
Coding Priorities

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



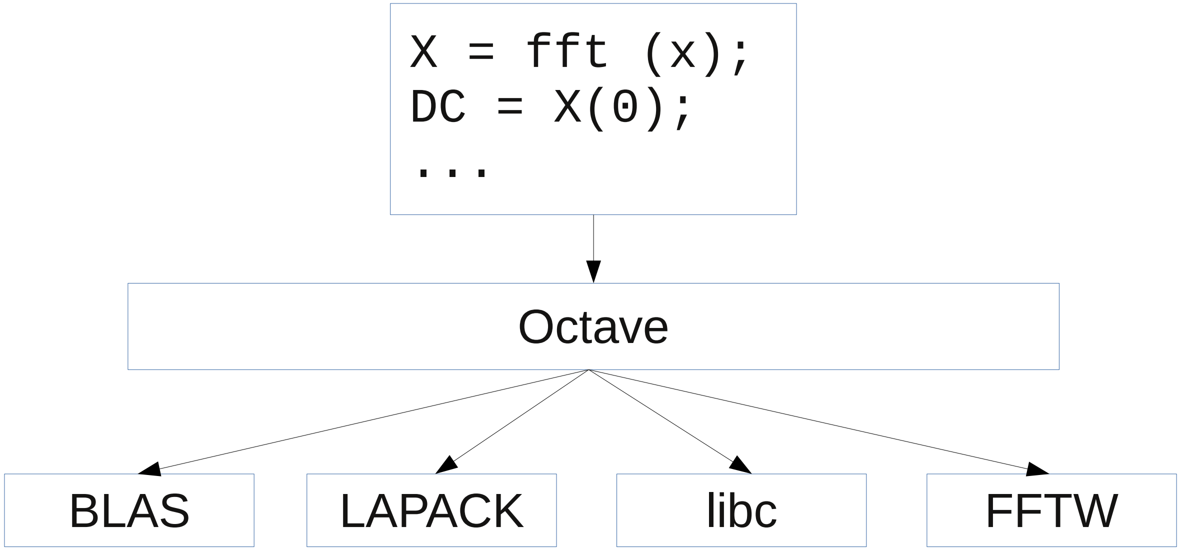
Design

Test

Build

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);
...
```

Octave

BLAS

LAPACK

libc

FFTW

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

$$RunTime = f\left(x_{1,}\, x_{2,}\, x_{3,} \ldots, x_n\right)$$

- Objective is to calculate partial derivative with respect to <u>just</u> 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
```

ftan () demonstration function

Sample function to be optimized

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

Baseline Performance

0.15062
0.14942
0.14847
0.14894
0.14864
...

- Mean = 0.148
- STD = .001

arrayfun ()

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

```
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 ftan ()

```
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 <u>and</u> 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 tan ()

```
function y = ftan_tan (x)
  y = tan (x);
endfunction
```

- Mean = .00028
- STD = .00002
- % change over ftan = -99.8%
- % change over vectorized ftan = -26%

Benchmark Summary

Function	**Relative Speed**
tan ()	1
vectorized ftan	1.36
arrayfun	436
looping ftan	529

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

```
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

```
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

Method	**RunTime**
Array growth	.167
Pre-declared array	.143
% change	-14%

In-Place Operators 1

```
A = A + 1
```

is equivalent to

```
TMP = A + 1
A = TMP
```

In-Place Operators 2

```
A += 1
```

Does not create a temporary array!

In-Place Benchmarks

Method	RunTime	% Change	Relative RunTime
A = A + 1	.111	--	1
A++	.110	-1%	.99
++A	.111	0%	1
A += 1	.041	-60%	.40

- 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

```
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

```
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