# Matlab Performance Tuning

## Open Day – Jan 8, 2013

Yair Altman

מכללת הי-טק

JOHN BRYCE
תלמד הי-טק. זה עובד!
a matrix company

http://UndocumentedMatlab.com/files/OpenDay.zip

# Is Matlab performance important?

- Not a trivial question at all
- Some examples:
  - Compiled application
  - Need for real-time processing
  - Optimizing parameters based on numerous re-runs
  - Development time is often cheaper than production time
  - Developers are cheap, customers are expensive ☺

# The iterative tuning cycle

1. Measure overall performance
   - Proceed only if out of spec
2. Profile the code to determine hotspots
   - Profiler, tic/toc, log files
3. Modify the code to fix only the top hotspots
4. Return to step #1

- Avoid premature optimization
- Avoid tendency to over-tune
- Avoid tendency to custom-tune

# Optimization techniques

- Perceived performance
- Standard programming techniques
- Data analysis techniques
- Matlab-specific techniques
- Parallelization techniques
- Graphics and GUI techniques
- Memory-related techniques

# Standard programming techniques

- Loop optimization
- Caching data
- Smart checks bypass
- Remove unused computations (mlint, profiler)
- Exception handling
- Processing smaller data subsets
- Inlining core algorithm based on profiling report
- Improve external systems (database, file system)

# Other standard techniques

- Place more common conditional branches at top
- Lazy loading/initializing
- Preload/prefetch data
- Minimize/unify data fetch requests
- Reduce CPU/memory-intensive external processes
- Use short-circuit operators (& => &&, | => ||)
  - Backward compatibility considerations => use nested ifs
- Dynamic self-tuning

# Data analysis techniques

- Selecting the right tool for the job
- Outliers removal
- Controlling the target accuracy
- Coordinate transformation
- Choosing correct parameters
- Reducing problem complexity

# Matlab specific techniques

- Effects of using different storage types
- Vectorization
- Object oriented Matlab
- Using internal helper functions
- Optimizing string operations
- Deployed (compiled) applications
- Optimizing I/O
- Object-oriented Matlab
- Using MEX (=*Matlab Executable*), Matlab Coder
- Many other techniques…

# Parallelization techniques

- Implicit/explicit parallelization
  - Within the core (hyper-threading)
  - Multi-threading
  - Across CPU cores
  - Across CPUs
  - Using GPU(s)
    - on-die (Intel's Ivy Bridge via OpenCL)
    - external (mainly NVidia CUDA 1.3+)
- Matlab add-ons:
  - MathWorks: Parallel Computing Toolbox, Distributed Computing Server
  - AccelerEyes: Jacket (*RIP…*)
  - CULA, GPULib, OpenCL, GPUMat, NVMex, JADE, OpenMP, MATLAB*P, MatlabMPI, pMATLAB

# Graphics and GUI techniques

- Initial graphs generation
- Updating graphs in real-time
- GUI preparation
- GUI responsiveness

# Memory-related techniques

- Why memory affects performance
- Profiling memory usage in Matlab
- Speed-up techniques:
  - Matlab's memory storage and looping order
  - Sub-indexing is sometimes slower (!)
  - Array memory allocation, pre-allocation
  - Minimizing memory allocations (COW, in-place)
  - Memory packing

# Why memory affects performance

- Data and code share the system memory
- CPU and disk caches constantly contend for room
- CPU performance outpaced memory throughput
  → programs are often "*memory-bound*"
- Paging in/out:
  - Disk (virtual memory)
  - Physical memory
  - CPU cache (L1,L2,…)
- All using the main system bus…
- Especially important for CPU/GPU parallelization

# Profiling memory usage in Matlab

- The Workspace browser
- The `whos` function
- The `memory` function
- `feature memstats`
- `feature dumpmem`
- `feature mtic/mtoc` (R2008a)
- `profile -memory on`
- Using 3rd-party tools (e.g., SysInternals ProcessExplorer)
- Using the OS tools
- `format debug`

# Dynamic array growth

- What happens here (performance-wise)?

```
fibonacci = [0, 1];
for idx = 3 : 100
   fibonacci(idx) = fibonacci(idx-1) + fibonacci(idx-2);
end
```

- Memory reallocations: allocate, copy, discard
  - Slow!
  - Potentially disastrous (thrashing)

# The quadratic cost of dynamic growth

- Theory: quadratic cost (N allocations x size N = $N^2$)

```
% This was ran on MATLAB 7.1 (R14 SP3):
tic, f=[0,1]; for idx=3:10000, f(idx)=f(idx-1)+f(idx-2); end, toc
⇒ Elapsed time is 0.149173 seconds. %baseline loop size & exec time

tic, f=[0,1]; for idx=3:20000, f(idx)=f(idx-1)+f(idx-2); end, toc
⇒ Elapsed time is 0.586088 seconds. %x2 loop size, x4 execution time

tic, f=[0,1]; for idx=3:40000, f(idx)=f(idx-1)+f(idx-2); end, toc
⇒ Elapsed time is 2.090217 seconds. %x4 loop size, x14 execution time
```

# Effects of incremental JIT releases

- JIT Accelerator introduced in Matlab 6.5 (R13)
  - Dramatic automatic performance boost
  - Incrementally improved over the years (releases)
    - Matlab 7.11 (R2010b) is ~30% faster than Matlab 7.1 (R14SP3) for the fibonacci code above
    - However, still $O(N^2)$
- Matlab 7.12 (R2011a): significant optimization to dynamic memory allocation => O(N)

```
% This was ran on MATLAB 7.12 (R2011a):
tic, f=[0,1]; for idx=3:10000, f(idx)=f(idx-1)+f(idx-2); end, toc
⇒ Elapsed time is 0.004924 seconds. %baseline loop size & exec time

tic, f=[0,1]; for idx=3:20000, f(idx)=f(idx-1)+f(idx-2); end, toc
⇒ Elapsed time is 0.009971 seconds. %x2 loop size, x2 execution time

tic, f=[0,1]; for idx=3:40000, f(idx)=f(idx-1)+f(idx-2); end, toc
⇒ Elapsed time is 0.019954 seconds. %x4 loop size, x4 execution time
```

# Alternatives for dynamic array growth

- Using cell arrays
- Dynamic advanced allocation (factor growth)
- Using FEX: ***growdata*** utility
- Reuse existing data array
- Wrap data in a referential object
- Naïve resizing using the new JIT:

```
% This was ran on Matlab 7.12 (R2011a):
% Variant #1: direct assignment into a specific out-of-bounds index
data=[]; tic, for idx=1:100000; data(idx)=1; end, toc
⇒ Elapsed time is 0.075440 seconds.

% Variant #2: direct assignment into an index just outside the bounds
data=[]; tic, for idx=1:100000; data(end+1)=1; end, toc
⇒ Elapsed time is 0.241466 seconds. % x3 slower

% Variant #3: concatenating a new value to the array
data=[]; tic, for idx=1:100000; data=[data,1]; end, toc
⇒ Elapsed time is 22.897688 seconds. % x300 slower!!!
```

# Pre-allocation

- Even with the new JIT, pre-allocation still faster:

```
% This is ran on MATLAB 7.12 (R2011a)

% Regular dynamic array growth
tic, f=[0,1]; for idx=3:40000, f(idx)=f(idx-1)+f(idx-2); end, toc
⇒ Elapsed time is 0.019954 seconds.

% Now use preallocation: x5 faster than dynamic array growth
tic, f=zeros(40000,1); f(1)=0; f(2)=1;
for idx=3:40000, f(idx)=f(idx-1)+f(idx-2); end, toc
⇒ Elapsed time is 0.004132 seconds.
```

- Compare Matlab 7.1 (R14 SP3): 2.1s, 0.06s (=x35)

# Pre-allocation variants – D/C value

```
% Variant #1: explicit preallocation of data1
data1 = zeros(1000,3000);
for colIdx = 1 : 3000
   for rowIdx = 1 : 1000
      data1(rowIdx,colIdx) = someValue;
   end
end
```

⇒ 17 mSecs

```
% Variant #2: implicit preallocation of data2 (much faster, see below)
data2=[]; data2(1000,3000) = 0;
for colIdx = 1 : 3000
   for rowIdx = 1 : 1000
      data2(rowIdx,colIdx) = someValue;
   end
end
```

⇒ 0.03 mSecs (~x500 faster)

```
% Variant #3: implicit preallocation of data3 using loop reversal
for colIdx = 3000 : -1 : 1
   for rowIdx = 1000 : -1 : 1
      data3(rowIdx,colIdx) = someValue;
   end
end
```

⇒ 0.03 mSecs (~x500 faster)

# Pre-allocation variants – scalar value

```
scalar = 7;  % for example...

data = scalar(ones(1000,3000));              % Variant A: 87.680 msecs
data(1:1000,1:3000) = scalar;                % Variant B: 28.646 msecs
data = repmat(scalar,1000,3000);             % Variant C: 17.250 msecs
data = scalar + zeros(1000,3000);            % Variant D: 17.168 msecs
data(1000,3000) = 0; data = data+scalar;     % Variant E: 16.334 msecs
```

# Preallocation of non-double data

- Prevent implicit data conversions and reallocations

```
% Bad idea: allocates 8MB double array, then converts to 1MB int8 array
data = int8(zeros(1000,1000));    % 1M elements
⇒ Elapsed time is 0.008170 seconds.

% Better: directly allocate the array as a 1MB int8 array - x80 faster
data = zeros(1000,1000,'int8');
⇒ Elapsed time is 0.000095 seconds.


% Another example:
text = char(fread(…));     % reads doubles, then converts to chars
text = fread(…,'*char');   % better - reads chars directly
```

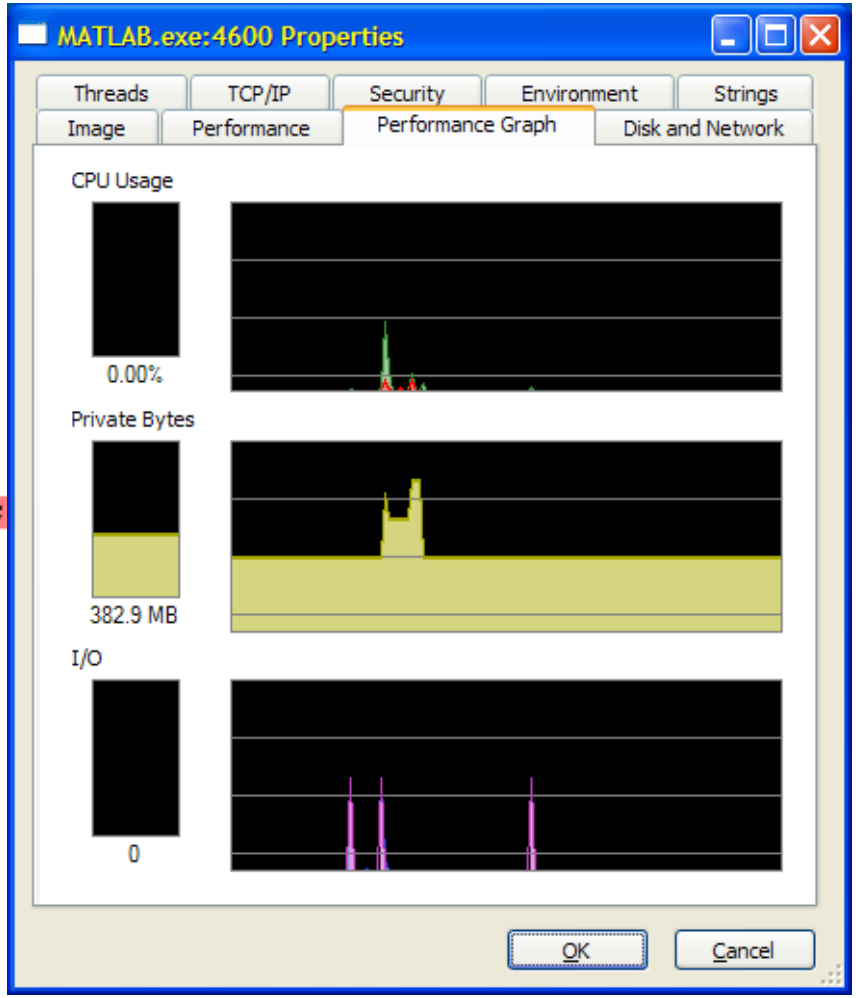
# Minimizing memory allocations

- Matlab's *Copy-on-Write* (COW) mechanism
- In-place data manipulations
- Reusing exiting variables (*with utmost care!*)
- Clearing unused workspace variables
- `global` and `persistent` variables
- Scoping rules and nested functions
- Passing handle references to functions
- Reducing data precision/type

# Matlab's Copy-on-Write mechanism

- Pass-by-value vs. pass-by-reference
- Lazy copying

| time | calls | mem | line | |
|---|---|---|---|---|
| | | | 1 | function perfTest |
| 0.890 | 1 | 215m/23.9m/191m | 2 | data1 = magic(5000); |
| | 1 | | 3 | data2 = data1; |
| 0.172 | 1 | 191m/0b/191m | 4 | data2(1,1) = 0; |
| | 1 | | 5 | end |



# In-place data manipulations

- Within the code:

```
% In-place data manipulation, no memory allocation
>> tic, m = m * 0.5; toc
⇒  Elapsed time is 0.056464 seconds.

% Regular data manipulation (122MB allocation) - 50% slower
>> clear m2; tic, m2 = m * 0.5; toc;
⇒  Elapsed time is 0.084770 seconds.
```

- Calling functions:

```
% In-place data manipulation, no memory allocation
>> d=0:1e-7:1; tic, d = sin(d); toc
⇒ Elapsed time is 0.083397 seconds.

% Regular data manipulation (76MB allocation) - 50% slower
>> clear d2, d=0:1e-7:1; tic, d2 = sin(d); toc
⇒ Elapsed time is 0.121415 seconds.
```

- Called function:

```
% Suggested practice: use in-place optimization within functions
function x = function1(x)
   x = someOperationOn(x);    % temporary variable x is NOT allocated
end

% Standard practice: prevents future use of in-place optimizations
function y = function2(x)
   y = someOperationOn(x);    % new temporary variable y is allocated
end
```

# MEX in-place

- Working, but not officially supported for RHS params
- Use `mxDuplicateArray()` to create copies
- Use `mxUnshareArray()` to mimic COW behavior
- Warning: can easily crash Matlab upon SEGV

# Conclusion

- Matlab application performance can be dramatically improved
- Numerous different techniques can be used
- Need to pay managerial attention to
  - performance tradeoffs
  - apply the most promising techniques
  - stick to the measure-profile-tune-remeasure cycle
  - prevent premature tuning, over-tuning