ETS group meeting

intro to faster matlab code

by

Rob Young
overview

- motivation
- philosophy
- efficient Matlab techniques (tip of iceberg)
- GPU enabled Matlab functions
- parallel for loops
- MEX
- CUDA
motivation

- You don't want to wait for results
- Your labmates don't want to wait for your results
philosophy

“Premature optimization is the root of all evil (or at least most of it) in programming.” --Knuth

- readability is key
  - less errors
  - reusable
- only optimize bottlenecks
  - keep readable code commented
efficient Matlab - profiler

- find bottlenecks:
  1) > profile on
  2) run your code
  3) > profile viewer
Profiler – time spent per line

### nlmmeans_good (1 call, 2.794 sec)
Generated 20-Apr-2011 16:57:02 using cputime.
function in file /u/2/3/p3/pyoung/Talks/matlabOpt/nlmmeans_good.m
Copy to new window for comparing multiple runs.

#### Parents (calling functions)
No parent.

#### Lines where the most time was spent

<table>
<thead>
<tr>
<th>Line Number</th>
<th>Code</th>
<th>Calls</th>
<th>Total Time</th>
<th>% Time</th>
<th>Time Plot</th>
</tr>
</thead>
<tbody>
<tr>
<td>37</td>
<td>errB1k = nayBlock - repmat(x,...</td>
<td>2304</td>
<td>1.233 s</td>
<td>44.1%</td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>regNIWeights = exp(-regNIWeights...</td>
<td>2304</td>
<td>0.822 s</td>
<td>29.4%</td>
<td></td>
</tr>
<tr>
<td>41</td>
<td>regNIWeights - regNIWeights/su...</td>
<td>2304</td>
<td>0.283 s</td>
<td>10.3%</td>
<td></td>
</tr>
<tr>
<td>39</td>
<td>regNIWeights = sum(errB1k.A2);</td>
<td>2304</td>
<td>0.247 s</td>
<td>8.8%</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>RandStream.setDefaultStream (R...</td>
<td>1</td>
<td>0.082 s</td>
<td>2.9%</td>
<td></td>
</tr>
</tbody>
</table>

All other lines:
- 0.123 s  4.4%

Totals:
- 2.794 s  100%

#### Children (called functions)

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Function Type</th>
<th>Calls</th>
<th>Total Time</th>
<th>% Time</th>
<th>Time Plot</th>
</tr>
</thead>
<tbody>
<tr>
<td>nlmmeans</td>
<td>function</td>
<td>2304</td>
<td>1.088 s</td>
<td>38.2%</td>
<td></td>
</tr>
</tbody>
</table>
### Profiler – mlint (Code Analyzer)

<table>
<thead>
<tr>
<th>Line</th>
<th>Function Name</th>
<th>Calls</th>
<th>Total Time</th>
<th>% Time</th>
<th>Time Plot</th>
</tr>
</thead>
<tbody>
<tr>
<td>23</td>
<td>regNWeights = sum(errBklA2);</td>
<td>2304</td>
<td>0.247 s</td>
<td>8.8%</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>RandStream.setDefaultStream(R...)</td>
<td>1</td>
<td>0.082 s</td>
<td>2.5%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>All other lines</td>
<td>0.123 s</td>
<td>4.4%</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Totals</strong></td>
<td></td>
<td><strong>2.794 s</strong></td>
<td>100%</td>
<td></td>
</tr>
</tbody>
</table>

**Children (called functions)**

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Function Type</th>
<th>Calls</th>
<th>Total Time</th>
<th>% Time</th>
<th>Time Plot</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>remat</code></td>
<td>function</td>
<td>2304</td>
<td>1.068 s</td>
<td>38.2%</td>
<td></td>
</tr>
<tr>
<td><code>im2coll</code></td>
<td>function</td>
<td>2</td>
<td>0.041 s</td>
<td>1.5%</td>
<td></td>
</tr>
<tr>
<td><code>stream.RandStream&gt;RandStream&gt;RandStream</code></td>
<td>subfunction</td>
<td>1</td>
<td>0.041 s</td>
<td>1.5%</td>
<td></td>
</tr>
<tr>
<td><code>mkBump</code></td>
<td>function</td>
<td>1</td>
<td>0.041 s</td>
<td>1.5%</td>
<td></td>
</tr>
<tr>
<td><code>RandStream&gt;RandStream.setDefaultStream</code></td>
<td>subfunction</td>
<td>1</td>
<td>0 s</td>
<td>0%</td>
<td></td>
</tr>
</tbody>
</table>

**Self time (built-ins, overhead, etc.)**

|                  |               |       | 1.603 s    | 57.4% |           |

**Totals**

|                  |               |       | **2.794 s**| 100%   |           |

### Code Analyzer results

<table>
<thead>
<tr>
<th>Line number</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>The function return value 'T' might be unset.</td>
</tr>
<tr>
<td>11</td>
<td>The value assigned to variable 'mypos' might be unused.</td>
</tr>
<tr>
<td>23</td>
<td>The value assigned to variable 'orgBlockSize' might be unused.</td>
</tr>
<tr>
<td>43</td>
<td>The variable 'denInNL' appears to change size on every loop iteration. Consider preallocating for speed.</td>
</tr>
<tr>
<td>46</td>
<td>The value assigned to variable 'Tcpu' might be unused.</td>
</tr>
<tr>
<td>48</td>
<td>The value assigned to variable 'denInNL' might be unused.</td>
</tr>
</tbody>
</table>

### Coverage results

<table>
<thead>
<tr>
<th>Line number</th>
<th>Coverage</th>
</tr>
</thead>
</table>

| 1           | 100%     |
| 11          | 100%     |
| 23          | 100%     |
| 43          | 100%     |
| 46          | 100%     |
| 48          | 100%     |
For loops are slow in Matlab, so replace with colon (:) or repmat:

```matlab
i = 0;
for t = 0:0.001:1
    i = i + 1;
    y(i) = sin(t);
end
```

with:

```matlab
t = 0:0.001:1;
y = sin(t);
```
efficient Matlab – pre-allocation

• If you are stuck with a for loop then make sure you preallocate:

```matlab
foo = zeros(1,N);
for i = 1:N
    foo(i) = baz(i);
end
```

• otherwise you're reallocating a new array at each iteration
efficient Matlab - In-place operations

• Many Matlab functions support in-place operation on data:
  \[ x = \text{myfunc}(x) \]
  • No memory overhead and no time overhead for allocation.
efficient Matlab – single precision

• Do you really need double precision?
• If not allocate as single precision:  
  foo = single(rand(N));
• quick way to cut execution time in half.  
  (almost anyway)
• cuts internal representation of variables in half
parallel threads of execution

• Matlab >= 7.4 supports CPU multithreading
  • CPU usage > 100%  ==  CPU multithreading
• Matlab >= 7.11 supports GPU multithreading
• example: independent iterations of for loop
  • pass each job to its own processing core (CPU or GPU)
  • Multiple iterations done at each time step
efficient Matlab – GPU functions

- latest versions of Matlab have limited GPU support:
  - arrayfun, conv, dot, filter, fft, ifft, ldivide, lu, mldivide, ...
- data transfer to and from card is slow
- works best with vectorized code
% move data to GPU
X_gpu = gpuArray(im_cpu);
Y_gpu = gpuArray(filt_cpu);
< perform operations on the GPU >
Z_gpu = ifft( fft(X_gpu) .* fft(Y_gpu) );
Z_cpu = gather(Z_gpu);% pull data off the GPU
faster for loops - parfor

• have a for loop that you can't vectorize?
• if each loop iteration is independent:
  
  ```
  matlabpool open;
  parfor i=1:N
      < loop body >
  end
  matlabpool close;
  ```

• current maximum # workers (threads) == 8
faster code - MEX

- Running C code in Matlab
- Standard C except for matlab interface.
faster for loops - CUDA

```c
#include <mex.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include "cuda.h"

void mexFunction(int NInLhs, mxArray *plhs[], int nrhs, const mxArray *prhs[])
{
    int i, nr, nc, blksize, npix;
    double *denInNL, *denInNL gpu, *org gpu, *nsy gpu;
    double *sumRegNlWeights, *sumRegNlWeights gpu, *orgIn, *nsyIm;
    double gaussianWidth;
    size_t dims[1] = [1];

    /* Input parameters */
    nr = (int) mxGetScalar(prhs[0]);
    nc = (int) mxGetScalar(prhs[1]);
    blksize = (double) mxGetPr(prhs[2]);
    orgIn = (double *) mxGetPr(prhs[3]);
    nsyIm = (double *) mxGetPr(prhs[4]);
    gaussianWidth = (double) mxGetScalar(prhs[5]);

    /* Output parameter */
    dims[1] = (size_t)npix;
    plhs[0] = mxCreateNumericArray(2, dims, mxDOUBLE_CLASS, mxREAL);
    denInNL = (double *) mxGetPr(plhs[0]);

    /* malloc vars */
    sumRegNlWeights = (double*)malloc(npix*sizeof(double));
    for(i=0;i<npix;i++)
        denInNL[i] = 0.0;
    sumRegNlWeights[1] = 0.0;

    /* CUDA code */
    cudaMemcpy(sumRegNlWeights gpu, sizeof(double)*npix);
    cudaMempyHostToDevice(sumRegNlWeights gpu, sumRegNlWeights, sizeof(double)*npix);
    cudaMemcpyHostToDevice(sumRegNlWeights gpu, sumRegNlWeights, sizeof(double)*npix);
    cudaMemcpy(org gpu, orgIn, sizeof(double)*npix, cudaMemcpyHostToDevice);
    cudaMemcpy(syt gpu, nsyIm, sizeof(double)*npix, cudaMemcpyHostToDevice);
    cudaMemcpy(devInNL gpu, denInNL, sizeof(double)*npix, cudaMemcpyHostToDevice);
    gpuFunc<<<npix/128,128>>>(nr, nc, blksize, denInNL gpu, org gpu, nsy gpu, gaussianWidth, sumRegNlWeights); // cudaThreadSynchronize();

    /* normalize */
    for(i=0;i<npix; i++)
        denInNL[i] = denInNL[i] / sumRegNlWeights[i];

daFree(denInNL gpu);
daFree(org gpu);
daFree(syt gpu);
}
```
when is CUDA the right answer?

• Loop with large number of iterations
• Few if any temporary variables in loop
  • Large temporary variables must be duplicated
• For example: summary statistics
  • Only memory transfer on to card
  • Small temporary variable
  • Temporary variable can be shared by threads
nlmeans speed comparison
nlmeans speed comparison
nlmeans speed comparison
nlmeans speed comparison
Summary

1. Profile code.
2. Does simple fix fix problem fast enough?
   - Yes: Success!
   - No: Go to bottleneck.
3. Is bottleneck from for loop?
   - Yes: Parfor fast enough?
     - Yes: Success!
     - No: Go to Matlab GPU.
   - No: Matlab GPU functions fast enough?
     - Yes: Success!
     - No: Go to MEX.
4. Is MEX fast enough?
   - Yes: Success!
   - No: Go to CUDA.
5. Is CUDA fast enough?
   - Yes: Success!
   - No: Fail!
Resources

- me – my door’s always open!
- Matlab blogs (especially Loren & Steve):
  http://blogs.mathworks.com
- general Matlab optimization:
- profiler:
  http://blogs.mathworks.com/desktop/2010/02/01/speeding-up-your-program-through-profiling/
- parfor:
  http://www.mathworks.com/help/toolbox/distcomp/brb2x2l-1.html
- GPU:
  http://www.mathworks.com/discovery/matlab-gpu.html
  http://www.mathworks.com/help/toolbox/distcomp/bsic3by.html
- MEX:
Thanks!

Let's talk about your code!
nlmeans code comparison

The image shows a graph plotting execution time (seconds) against image size (pixels) for different code implementations:
- CUDA
- MEX
- parfor
- matlab

The graph indicates how each implementation's performance scales with increasing image size.