Parallel Computing in MATLAB

- As Scientific/Engineering problems grow in complexity, the need for computational and storage resources grow.

- There have been dramatic changes in hardware recently and parallelism has become mainstream. We now have clusters of machines (since late 1990’s) and multi-core processors on new machines (since mid 2000’s).

- Have you ever wondered how to effectively use your multi-core machine without having to learn a lot about parallel programming?

- MATLAB can solve “embarrassingly parallel” problems with minimal change to existing MATLAB code.
• For example, the Parallel Computing Toolbox (PCT) lets you computationally solve data intensive problems using multi-core processors (MATLAB workers), GPUs (Graphics Processing Units), and computer clusters.

• You can develop your program on a multi-core desktop computer using the Parallel Computing Toolbox and then scale up to many computers by running it on a MATLAB Distributed Computing Server. The server supports batch jobs, parallel computations, and distributed large data. However, this lecture (and course) will only focus on single machines with multi-core processors and a Graphical Processing Unit (GPU).
How to Speed up your MATLAB Programs?

• Besides the optimization techniques already covered (for example, vectorizing your code) we will look at how to use multi-core machines and GPUs with minimal changes to your existing code.

• A thread of execution is the smallest sequence of programmed instructions that can be executed in parallel.
• Much multi-threading is already built into many MATLAB functions. For example, the following operations use parallelism when the hardware is available:

  – the `fft` function (Fast Fourier Transform function);
  – various filtering functions such as `imfilter`; and
  – various matrix operations.

• Currently, the maximum number of computational threads is the number of physical cores on your machine.

• Typically, 12 cores can allow a speedup factor of about 10, but note that not everything is parallelizable and there are usually some overhead costs associated with parallelism.
Multi-Core Programming

• There is a tradeoff between ease of use and level of control.

• We can parallelize our programs and still run them on single core machines! That is, we can put parallel code in our program but select the maximum number of MATLAB workers to be 1 (such code will run on a single core machine). Of course, this code will not be run in parallel! We can set MATLAB to automatically select the maximum number of core available using `parfor` (with no parameters), as described below. This then means our code can run on many different multi-core machines without being changed.
• The command `parpool` allocates all the cores on your machine to your MATLAB session. You can also specify explicitly the number of cores you want to use, so `parpool('local',2)` uses 2 cores (your machine must have 2 cores). And, of course, using `parpool` or `parpool('local',1)` on a single core machines allocates the 1 existing core. We emphasize that `parpool` allocates all available cores on a machine and this could be just 1!

• If you have a quad-core machine, you might use `parpool('local',2)` so that the 2 other cores are available for other tasks currently running. Of course, `parpool('local',4)` or `parpool` allocates all 4 cores.
• For MATLAB 2013a or earlier, `matlabpool open` was used instead of `parpool`. Thus `matlabpool open 2` uses 2 cores (your machine must have 2 cores). This command was removed in MATLAB R2015a.

• Following the link\(^1\) gives a list of the functions in various toolboxes that are parallelized for you. These functions use Parallel Computing Toolbox functions to achieve this parallelism.

• Commands `parfor`, `batch` and `distributed` offer more control over parallelization but greater programming effort is required.

\(^1\)www.mathworks.com/products/parallel-computing/builtin-parallel-support.html
Using parfor to Parallelize Loops

- This type of loop is ideal for parallel programming as long as there are no dependencies or communication between various iterations of the loop. Such loops are said to be embarrassingly parallel. Since each iteration of the loop is independent of all other iterations the order of execution of any iteration is unimportant and obviously the iterations can be executed parallel.

- Setup a pool of MATLAB workers and convert for to parfor for all appropriate loops (details to come), thus interleaving (intermixing) serial and parallel code.
• Each core has its own memory and the data the cores operate on must be copied to these memories. There can be a major overhead cost, especially if the cost of transferring this data to and from the cores is on the same order of magnitude as the computational time savings achieved by parallelism.

• In theory, if there is little overhead required for a parfor computation, with 4 cores you should get a speed up factor of almost 4. Even if each iteration requires different times to execute, the speedup up will be almost 4 because, when one a core is finished with an iteration, another unexecuted iteration is assigned to it, even though the other cores might not yet be finished their current iterations (so no synchronization is required).
• For example: a problem requiring 25.13 seconds on a single core might require 6.71 seconds on 4 cores. In this case \((4*6.71-25.13=1.71)\) seconds are required for overhead.

• Notice that all the MATLAB code outside the `parfor` loop is just regular code running on a single core. So the parallel code is interleaved with the serial code.

• You can’t use `parfor` for a loop with a dependency between loop iterations (for example, iteration \(i\) requires the result of iteration \(i - 1\)). MATLAB will give a warning that such a loop cannot be parallelized if `parfor` is used.

• An example of a embarrassingly parallel code segment using `parfor` is given by:
% compute constants
two_pi=2*pi;
pi_over_3=pi/3;
parfor i=1:n
    argument=two_pi*i-pi_over_3;
    res(i)=sin(argument)*cos(argument)*...
        exp(argument);
end % parfor
fprintf('Maximum result is: %f
',max(res(:)));
Using Low Level Parallel Programming Constructs

- Sometimes you have a number of parallelizable tasks that don’t fit easily into a loop construct. In this case you can use MATLAB’s batch processing capabilities. Note that now you have more control over the parallelization but at increased complexity in programming.

- Use `createJob` to explicitly tell MATLAB what tasks (functions) can be run in parallel.
An example MATLAB pseudo code segment:

```matlab
% Create an independent job object for a cluster
% of processors. The cluster can be your machine -
% you could use createJob with no argument but that
% is the old way of doing thing and will be depreciated
% in the future
job=createJob(cluster);
set(job,'MyFiles',{'SDs.m','SDt.m','SDb.m','SDq.m'});
% Create tasks - pass the function handles, the number
% of output arguments and a cell array of the
% input parameters
createTask(job,@SDs,1,{n,k,b,totalTime});
createTask(job,@SDt,1,{n,k,b,totalTime});
createTask(job,@SDb,1,{n,k,b,totalTime});
createTask(job,@SDq,1,{n,k,b,totalTime});
% Submit job
submit(job); % fork
wait(job) % wait for all tasks to be completed, i.e. join

% Retrieve results
results=fetchOutputs(job);

% Compare results - a user written function
compareResults(m,k,b,totalTime,...
    results{1,1},results{2,1},...
    results{3,1},results{4,1})
```
• Some other MATLAB functionality:

1. `labSend` allows inter-worker communication (i.e., pass data, synchronize execution, etc).
2. `batch` runs the jobs in the background, if there are available cores the toplevel MATLAB session can continue executing. These are very similar to `createJob` and `createTask`, except all computation is completely offloaded.
3. `distribute` spreads parts of a (large) array onto different cores (MATLAB workers) so that these cores can access different parts of the data at the same time.

Look on the web or MATLAB search for the complete story on using these functions.
Using GPUs (Graphical Processing Units)

- GPUs were originally used to accelerate the display of graphics (i.e. fast 3D rendering for computer games) but these days they are also used for scientific calculations. The figure below shows the basic layout.
CPU cores versus GPU array. CPU cores and GPU cores are suitable for different calculations.
• GPUs have large array of integer/floating point processors (usually hundreds or thousands of processors per GPU with dedicated high speed memory). They are suitable for “number crunching” operations (filtering and matrix operations, for example). In some ways, GPU cores complement CPU cores.
• From the MATLAB documentation:

  – MATLAB now requires CUDA-enabled NVIDIA GPUs with compute capability 3.0 or higher.

  – For releases R2014a and earlier, compute capability 1.3 is sufficient.

  – For releases R2017b and earlier, compute capability 2.0 is sufficient for all features except for deep learning.

Values less than 1.3 mean the GPU cannot do double precision floating point arithmetic, a serious restriction for scientific computing!
• If your laptop does not have a NVIDIA GPU you cannot do GPU MATLAB programming!

• Note that you may be asked to update the CUDA driver associated with your GPU device. If this is the case, go to the NVIDIA website to download the appropriate driver.

• MATLAB command `gpuDeviceCount` return the number of GPU devices on your machine (this should usually be 1).

• GPU devices do not have MATLAB workers (as cores do), but rather you must push (spread) data onto them, do some calculation in parallel on this data using GPU enabled MATLAB functions like `fft` or `imfilter` and retrieve (gather) the output from them.
• Data is spread onto the GPU from the CPU using the command `gpuArray` and the results of the calculations are gathered back from the GPU onto the CPU using `gather`.

• All calculations on the GPU are done using overloaded MATLAB functions. An overloaded function, like `fft` or `imfilter`, use the GPU processors in parallel. You have to do nothing, other that use `gpuArray` to put the data on the GPU and `gather` at the end to bring the data back onto the CPU. Your MATLAB code looks like normal MATLAB code except it uses the overloaded functions. Some of these overloaded functions are listed on the next slide (this list is constantly growing).
Methods for class `gpuArray`:

<table>
<thead>
<tr>
<th>Method</th>
<th>Function</th>
<th>Method</th>
<th>Function</th>
<th>Method</th>
<th>Function</th>
<th>Method</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>abs</td>
<td>cdf2rdf</td>
<td>expm1</td>
<td>imerode</td>
<td>issorted</td>
<td>numel</td>
<td>sortrows</td>
<td></td>
</tr>
<tr>
<td>accumarray</td>
<td>ceil</td>
<td>eye</td>
<td>imfill</td>
<td>issparse</td>
<td>nzmax</td>
<td>sparse</td>
<td></td>
</tr>
<tr>
<td>acos</td>
<td>cgs</td>
<td>factorial</td>
<td>imfilter</td>
<td>issymmetric</td>
<td>ones</td>
<td>spdiags</td>
<td></td>
</tr>
<tr>
<td>acosd</td>
<td>chol</td>
<td>false</td>
<td>imgaussfilt</td>
<td>istrii</td>
<td>or</td>
<td>sprfun</td>
<td></td>
</tr>
<tr>
<td>acosh</td>
<td>circshift</td>
<td>fft</td>
<td>imgaussfilt3</td>
<td>istriu</td>
<td>padarray</td>
<td>spones</td>
<td></td>
</tr>
<tr>
<td>acot</td>
<td>classUnderlying</td>
<td>fft2</td>
<td>imgradient</td>
<td>kmeans</td>
<td>pagefun</td>
<td>sprand</td>
<td></td>
</tr>
<tr>
<td>acotd</td>
<td>complex</td>
<td>fftfilt</td>
<td>imgradientxy</td>
<td>knnsearch</td>
<td>pcg</td>
<td>sprandn</td>
<td></td>
</tr>
<tr>
<td>acoth</td>
<td>cond</td>
<td>ftn</td>
<td>imhist</td>
<td>ldivide</td>
<td>pdist</td>
<td>sprandsym</td>
<td></td>
</tr>
<tr>
<td>acsc</td>
<td>conj</td>
<td>filter</td>
<td>imlincomb</td>
<td>le</td>
<td>pdist2</td>
<td>sqrt</td>
<td></td>
</tr>
<tr>
<td>acsch</td>
<td>conv</td>
<td>filter2</td>
<td>imnoise</td>
<td>legendre</td>
<td>permute</td>
<td>sqrt</td>
<td></td>
</tr>
<tr>
<td>all</td>
<td>conv2</td>
<td>find</td>
<td>impop</td>
<td>length</td>
<td>planerot</td>
<td>sqrt</td>
<td></td>
</tr>
<tr>
<td>and</td>
<td>convn</td>
<td>fitcsvm</td>
<td>imreconstruct</td>
<td>linspace</td>
<td>std2</td>
<td>std2</td>
<td></td>
</tr>
<tr>
<td>angle</td>
<td>corr2</td>
<td>fix</td>
<td>imregdemons</td>
<td>log</td>
<td>stdfilt</td>
<td>stdf2</td>
<td></td>
</tr>
<tr>
<td>any</td>
<td>corrcoef</td>
<td>flip</td>
<td>imregionalmax</td>
<td>log10</td>
<td>sub2ind</td>
<td>sub2ind</td>
<td></td>
</tr>
<tr>
<td>applylut</td>
<td>cos</td>
<td>flipdim</td>
<td>imregionalmin</td>
<td>log1p</td>
<td>subsasgn</td>
<td>subsasgn</td>
<td></td>
</tr>
<tr>
<td>arrayfun</td>
<td>cosd</td>
<td>fliplr</td>
<td>imresize</td>
<td>log2</td>
<td>subsindex</td>
<td>subsindex</td>
<td></td>
</tr>
<tr>
<td>asec</td>
<td>cot</td>
<td>flipud</td>
<td>imrotate</td>
<td>logical</td>
<td>subsp</td>
<td></td>
<td></td>
</tr>
<tr>
<td>asecd</td>
<td>cotd</td>
<td>fprintf</td>
<td>imrotate_old</td>
<td>logspace</td>
<td>superiorfloat</td>
<td></td>
<td></td>
</tr>
<tr>
<td>asech</td>
<td>coth</td>
<td>full</td>
<td>int2sub</td>
<td>lsqr</td>
<td>sfd</td>
<td></td>
<td></td>
</tr>
<tr>
<td>asin</td>
<td>cov</td>
<td>gamma</td>
<td>inf</td>
<td>lu</td>
<td>svd</td>
<td></td>
<td></td>
</tr>
<tr>
<td>asind</td>
<td>csc</td>
<td>gammainc</td>
<td>inpolygon</td>
<td>mat2gray</td>
<td>svds</td>
<td></td>
<td></td>
</tr>
<tr>
<td>asinh</td>
<td>cscd</td>
<td>gammaincinv</td>
<td>int16</td>
<td>mat2str</td>
<td>svds</td>
<td></td>
<td></td>
</tr>
<tr>
<td>assert</td>
<td>csch</td>
<td>gammaln</td>
<td>int2str</td>
<td>max</td>
<td>tail</td>
<td></td>
<td></td>
</tr>
<tr>
<td>atan</td>
<td>ctranspose</td>
<td>gather</td>
<td>int32</td>
<td>mean</td>
<td>tan</td>
<td></td>
<td></td>
</tr>
<tr>
<td>atan2</td>
<td>cummax</td>
<td>ge</td>
<td>int64</td>
<td>medfilt2</td>
<td>tan</td>
<td></td>
<td></td>
</tr>
<tr>
<td>atan2d</td>
<td>cummin</td>
<td>gmem</td>
<td>int8</td>
<td>meshgrid</td>
<td>tanh</td>
<td></td>
<td></td>
</tr>
<tr>
<td>atand</td>
<td>cumprod</td>
<td>gop</td>
<td>interp1</td>
<td>min</td>
<td>real</td>
<td></td>
<td></td>
</tr>
<tr>
<td>atanh</td>
<td>cumsum</td>
<td>gpuArray</td>
<td>interp2</td>
<td>minres</td>
<td>reallog</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bandwidth</td>
<td>deg2grad</td>
<td>gradient</td>
<td>interp3</td>
<td>minus</td>
<td>realpow</td>
<td></td>
<td></td>
</tr>
<tr>
<td>besselj</td>
<td>del2</td>
<td>gt</td>
<td>interpn</td>
<td>mldivide</td>
<td>realsqrt</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bessely</td>
<td>det</td>
<td>head</td>
<td>intersect</td>
<td>mod</td>
<td>transpose</td>
<td></td>
<td></td>
</tr>
<tr>
<td>beta</td>
<td>detrend</td>
<td>histc</td>
<td>inv</td>
<td>mode</td>
<td>rem</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>repelem</td>
<td>triu</td>
<td></td>
</tr>
</tbody>
</table>
betainc | diag | histcounts | ipermute | movmean | repmat | true  
betaincinv | diff | histeq | iradon | movstd | reshape | typecast  
betainv | discretize | horzcat | isaUnderlying | movsum | rgb2gray | uint16  
bicg | disp | hsv2rgb | isbanded | movvar | rgb2hsv | uint32  
bicgstab | display | hypot | isdiag | mpower | rgb2ycbcr | uint64  
bicgstabl | dot | idivide | isempty | mrdivide | roots | uint8  
bitor | double | ifft | isequal | mtimes | rot90 | uminus  
bicgencode | edge | ifft2 | isequaln | nan | round | union  
bitorfind | eig | ifftn | isequalwithequalnans | ndgrid | sec | unique  
bitorindex | end | im2double | isfinite | ndims | secd | uniquetol  
bitorreduce | eps | im2int16 | isfloat | ne | sech | unwrap  
bitorselect | eq | im2single | ishermitian | nextpow2 | setdiff | uplus  
bitorselectsparse | erf | im2uint16 | isinf | nnz | setxor | var  
bsxfun | erfc | im2uint8 | isinteger | nonzeros | shiftdim | vertcat  
bwdist | erfcinv | imabsdiff | islogical | norm | sign | xcorr  
bwlabel | erf | imadjust | ismember | normest | sin | xor  
bwlookup | erfinv | imag | ismember | normxcorr2 | sind | ycbcr2rgb  
bwmorph | existsOnGPU | imbothat | ismissing | not | single | zeros  
cast | exp | imclose | isnan | nthroot | sinh |  
cat | expint | imcomplement | isnumeric | null | size |  
cconv | expm | imdilate | isreal | num2str | sort |  
  
Static methods:
  
colon | loadobj | randi | randperm |  
freqspace | rand | randn | speye |
• My laptop is a mid-2018 4 core PC notebook with a NVIDIA Geforce MX150 GPU device with 2GB of memory and 384 shader cores and a performance of 1,127 GFLOPS (1.13 TFLOPS) single precision, 35.23 GFLOPS double.

• As an aside: Apple switched from NVIDIA gpus to AMD gpus circa 2014. This means the MATLAB Parallel processing toolbox’s gpu programming tools no longer work on newer macs (alas)!

• The GPU cores are limited in what they can do (for example no direct access to I/O and limited memory. These cores are often called “renders” or “shaders” as they were originally were used (and still are used) to do graphic display on computers.
• You can run MATLAB functions like `plot` on the GPUs.

• For smaller problems, the CPU may actually be faster than the GPU solution because the GPU spends more time transferring the data to and from the GPU and main memory (RAM) than in computations. Calculations that are iterative, like computing the flow of air around a moving car/flying airplane are great computations to run on a GPU (as the air flow is computed by iterative equations).

• The `x=gpuarray(x)` command pushes the data in array `x` onto the GPU memory. Initially `x` was on the CPU but after it has been distributed on the GPU. Now, any operation done on `x` occurs on the GPU.
• No modifications are required for your code, MATLAB just executes code using `x` on the GPU (if it can) and the other code on the CPU. Any MATLAB function overloaded to work on the GPUs now executes on the GPUs.

• For example, `gradient` or `del2` compute the gradient and Laplacian on the GPU and save their results there. Note that if a non-overloaded function is used on the GPU data, than that data has to be explicitly transferred to the CPU memory, processed and then explicitly transferred back to the GPU. This is expensive time-wise and probably removes any advantage of doing GPU programming.
If you want to run an non-overloaded MATLAB function on a GPU you can use the command `arrayfun`. For example, 

\[ A = \text{arrayfun}(\text{FUN}, B) \]

applies the function specified by anonymous function handle `FUN` to each element of `gpuArray B` and returns the result to `A` on the GPU. `A` and `B` must have the same size as 

\[ A(i,j,...) = \text{arrayfun}(\text{FUN}, B(i,j,...)). \]

Use the `gather` command to bring the data back from the GPU to the CPU. You might do this because you need to use an non-overloaded function on all the data or you want to save it at the end of the GPU computations. Remember, data transfer is expensive overhead, so it is to be minimized. Always try to program your GPU code using only overloaded GPU functions and only have one spread and gather operation (at the beginning and the end).
• There are some overloaded MATLAB commands that will create data directly on the GPUs (avoiding data transfer overhead), for example, `ones`, `zeros`, `rand`, `randn`, `linspace`, `logspace`, etc. Use these if you can.

• The GPU overloaded functions include most element wise numeric operations (yes, you can vectorize your GPU code!!!).

• We can also run kernels from existing CUDA code and PTX (compiled CUDA code) files on MATLAB (with the Parallel Computing Toolbox). CUDA is the non-MATLAB programming environment for NVIDIA devices. Such code can be imported into MATLAB and run directly on the GPUs.
• CUDA offers you much more control over how GPU programming is done but at a greater cost in complexity. CUDA and PTX are beyond the scope of this course.

• Some criteria for determining whether a problem is suitable to be solved on the GPU:
  1. One is able to break the algorithm down into hundreds or thousands of independent work units.
  2. All GPU processors are kept busy for the same amount of time (avoid idle GPU processors).
  3. The computational time should significantly exceed the time required for data transfer time to and from the GPU. Data transfer is expensive because the GPU are connected to the CPU via a single PCI express bus.
Recap of multi-core CPU use versus GPU use.

- It is also easy to scale up (minimal code changes required) your multi-core CPU and GPU solutions to distributed clusters of computers. (In this course, we only address the question of how to use the local resources on your individual machine to parallelize your code.)
Multi-Core and GPU Programming Examples

- Finally, we consider some MATLAB parallel programming! Consider a MATLAB program that computes the \( y = \sin(x) \) for 100,000,000 values of \( x \) ranging from 0 to \( 2\pi \). This program is very simple and often the parallel overhead costs exceed the computational costs. The example is for illustrative purposes only. We will measure the computational costs with and without overhead costs in our MATLAB code below. We will code:

1. the single core serialized solution,
2. the single core vectorized solution,
3. the single core Just In Time solution,
4. the multi-core Just In Time solution,

5. the multi-core `parfor` solution,

6. the multi-core `vectorized` solution and

7. the gpu solution.

- We consider the serialized single core solution first. To ensure that the JIT compiler is not invoked, we do not pre-allocate the space for the $y$ array, but instead allow this array to be dynamically grow in size by 1 element for each iteration of a loop. Dynamically reallocation of the array for each loop iteration is an expensive thing to do and a bit unfair for comparison purposes but, on the other hand, JIT compilation will not occur.
Below we present the serial solution (see L14serialized_sin.m), which we consider the base case (we compare the performance of all other solutions to this solution):

```matlab
function [serialized_elapsed_time]=L14serialized_sin(N)

% Don’t allocate y array outside the loop, %
% this forces the code to be serial %
% and JIT not to be used
x=linspace(0,2*pi,N);
for i=1:N
    y(i)=sin(x(i));
end
serialized_elapsed_time=toc(serialized_elapsed_time);
fprintf(’Serialized time: %8.4f seconds\n’,serialized_elapsed_time);
% see if it really is a sin curve
figure
```
plot(x,y)
stg=sprintf('%8.4f',serialized_elapsed_time);
title(['
fontsize{12}\bf Serialized y=sin(x) elapsed ',
    ' time:' stg ' seconds']);
filename=['serialized_sin_curve.png'];
print(filename,'-dpng');

- We can also vectorize this solution (see L14vectorized_sin.m) on a single core using:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Vectorized Solution
% compute y=sin(x) for N values of x
% ranging from 0 to 2*pi
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [vectorized_elapsed_time]=L14vectorized_sin(N)

vectorized_elapsed_time=tic;

% allocate vectors x and y
x=zeros(N,1,'double');
y=zeros(N,1,'double');

% compute x and y values
x=linspace(0,2*pi,N);
```
\begin{verbatim}
\begin{verbatim}
y=sin(x);

vectorized_elapsed_time=toc(vectorized_elapsed_time);
fprintf('Vectorized time: %8.4f seconds\n',
    vectorized_elapsed_time);

\% see if it really is a sin curve
figure
plot(x,y)
stg=sprintf('%8.4f',vectorized_elapsed_time);
title(['\fontsize{12}\bf Vectorized y=sin(x) elapsed ',
    ' time:' stg ' seconds']);
filename=['vectorized_sin_curve.png'];
print(filename,'-dpng');
\end{verbatim}

- To demonstrate solution correctness, we plot out the \((x,y)\) coordinates to ensure the serialized and vectorized sin function values are the same. We also labelled each plot with the execution time required for that solution. This time is large for the serial solution because the arrays are continually being increased at each iteration
\end{verbatim}
of the loop.

(a) Serialized solution  
(b) Vectorized solution

Plot of $y = \sin(x)$ from the serialized function (7.897986 seconds) and the vectorized solution (0.986834 seconds)
We present the single core L14jit_sin.m function below:

```matlab
function [jit_elapsed_time]=L14jit_sin(N)

[jit_elapsed_time]=tic;

% Allocate x and y outside the loop,
% this allows the JIT (Just In Time)
% compiler to be used as the data
% structures inside the do not change

x=linspace(0,2*pi,N);
y=zeros(N,1,'double');

% compute x and y values
for i=1:N
    y(i)=sin(x(i));
end

j= toc(jit_elapsed_time);
fprintf('JIT time: %8.4f seconds
',j);

figure
plot(x,y)
```
% We present the multi-core JIT solution (L14jit_parallel_parfor_sin.m below. Note that we now have a second parameter for this function, num_cores. The idea is to run this program using both 2 and 4 cores to see the performance difference.

\begin{verbatim}
function [jit_parallel_parfor_elapsed_time_with_overhead,...
    jit_parallel_parfor_elapsed_time_without_overhead]=...
L14jit_parallel_parfor_sin(N,num_cores)

    stg=sprintf('%8.4f',jit_elapsed_time);
    title(['\textbf{JIT y=sin(x) elapsed }',...
         ' time: ' stg ' seconds']);
    filename=['jit_sin_curve.png'];
    print(filename,'-dpng');
\end{verbatim}
% Allocate the available cores
parpool('local',num_cores);

x=linspace(0,2*pi,N);
y=zeros(N,1,'double');

jit_parallel_parfor_elapsed_time_without_overhead=tic;

% Allocate x and y outside the loop,
% this allows the JIT (Just In Time)
% compiler to be used as the data
% structures inside the do not change

% compute the y values
parfor i=1:N
    y(i)=sin(x(i));
end
jit_parallel_parfor_elapsed_time_without_overhead=...
    toc(jit_parallel_parfor_elapsed_time_without_overhead);
fprintf('JIT parallel time without overhead: %10.6f seconds\n',...
    jit_parallel_parfor_elapsed_time_without_overhead);

% close worker pool
% gcp - get current pool
delete(gcp('nocreate'))

jit_parallel_parfor_elapsed_time_with_overhead=...
    toc(jit_parallel_parfor_elapsed_time_with_overhead);
fprintf('JIT parallel time with overhead: %10.6f seconds
',jit_parallel_parfor_elapsed_time_with_overhead);

% see if it really is a sin curve
figure
plot(x,y)
stg=sprintf('%8.2f',jit_parallel_parfor_elapsed_time_with_overhead);
title(['\textbf{JIT y=sin(x) elapsed } ',...'
  time for parfor:' stg ' seconds']);
filename=['jit_parfor_sin_curve.png'];
print(filename,'-dpng');

• delete(gcp('nocreate')) deletes the current pool. gcp is a MATLAB function that returns the current pool handle. Thus, p=gcp('nocreate') returns the current pool handle if one exists. If no pool exists, the 'nocreate' option ensures that a pool is not created.
• We can compare the code for the serialized and jit solutions using the compare tool on the MATLAB toolbar:

![Compare button on the MATLAB main toolbar](image)

Compare button on the MATLAB main toolbar (4th from the left, 2nd row).

• Clicking compare gives a window where you can type the full path-names and filenames of the 2 files you want to compare:
Compare files window with serialized and jit file names.

- Clicking the **compare** button on this window yields the screen shot captured on the next slide. Purple lines indicate changes, cyan lines indicate new lines and un-coloured lines mean common text.
Compare serialized and jit solutions.
We present `L14parallel_parfor_sin.m` below. Again we have a parameter for `num_cores`.

```matlab
% Parallel Serialized Solution
% compute y=sin(x) for N values of x
% ranging from 0 to 2*pi

function [parallel_parfor_elapsed_time_with_overhead,...
        parallel_parfor_elapsed_time_without_overhead]=...
        L14parallel_parfor_sin(N,num_cores)

parallel_parfor_elapsed_time_with_overhead=tic;

% Allocate the available cores
parpool('local',num_cores);

x=linspace(0,2*pi,N);

parallel_parfor_elapsed_time_without_overhead=tic;

parfor i=1:N
    y(i)=sin(x(i));
end

parallel_parfor_elapsed_time_without_overhead=... 
toc(parallel_parfor_elapsed_time_without_overhead);
```
% close worker pool
% gcp - get current pool
delete(gcp('nocreate'))

parallel_parfor_elapsed_time_with_overhead = ... 
    toc(parallel_parfor_elapsed_time_with_overhead);

fprintf('Parallel parfor elapsed time without overhead: %10.6f seconds\n', ...
    parallel_parfor_elapsed_time_without_overhead);
fprintf('Parallel parfor elapsed time_with_overhead: %10.6f seconds\n', ...
    parallel_parfor_elapsed_time_with_overhead);
%
% see if it really is a sin curve
figure
plot(x,y)
stg=strprintf('%8.2f', parallel_parfor_elapsed_time_with_overhead);
title(['\fontsize{12}\bf Parallel parfor y=sin(x) elapsed ', ...
    ' time: ' stg ' seconds']);
filename=['parallel_parfor_sin_curve.png'];
print(filename,'-dpng');
- We present `L14_parallel_vectorized_sin.m` function next:

```matlab
% Parallel Vectorized Solution
% compute y=sin(x) for N values of x
% ranging from 0 to 2*pi

function [parallel_vectorized_elapsed_time_with_overhead,...
    parallel_vectorized_elapsed_time_without_overhead]=...
L14_parallel_vectorized_sin(N,num_cores)

parallel_vectorized_elapsed_time_with_overhead=tic;

parpool('local',num_cores);

parallel_vectorized_elapsed_time_without_overhead=tic;

close worker pool
% gcp - get current pool
delete(gcp('nocreate'))
```
parallel_vectorized_elapsed_time_with_overhead=...
    toc(parallel_vectorized_elapsed_time_with_overhead);

fprintf('Parallel vectorized elapsed time without overhead: %10.6f seconds\n',...
    parallel_vectorized_elapsed_time_without_overhead);
fprintf('Parallel vectorized elapsed time with overhead:   %10.6f seconds\n',...
    parallel_vectorized_elapsed_time_with_overhead);
% see if it really is a sin curve
figure
plot(x,y)
stg=sprintf('%8.2f',parallel_vectorized_elapsed_time_with_overhead);
title(['
\textbf{Parallel vectorized } y=\sin(x) \text{ elapsed }',...
    ' time:' stg ' seconds']);
filename=['parallel_vectorized_sin_curve.png'];
print(filename,'-dpng');
Finally, we present the GPU sin function (L14gpu_sin.m) last:

```matlab
function [gpu_elapsed_time_with_overhead,...
    gpu_elapsed_time_without_overhead] = ...
    L14gpu_sin(N)

gpu_elapsed_time_with_overhead=tic;

% clear GPU memory
gpudev=gpuDevice(1);
reset(gpudev);

% Allocate x and y outside the loop,
% Distribute x to GPU (Graphical Processing Unit)
% x is increment*(0:N)
x=linspace(0,2*pi,N);
x=gpuArray(x);

% no need to preallocate y and spread to GPU cores
gpu_elapsed_time_without_overhead=tic;
y=sin(x);
gpu_elapsed_time_without_overhead=...
toc(gpu_elapsed_time_without_overhead);
```
% Use the gather function to
% bring data back from the GPU;
x=gather(x);

gpu_elapsed_time_with_overhead=...
    toc(gpu_elapsed_time_with_overhead);

fprintf('
');
fprintf('gpu time without overhead: %10.6f seconds
',...
    gpu_elapsed_time_without_overhead);
fprintf('gpu time with overhead:  %10.6f seconds
',...
    gpu_elapsed_time_with_overhead);

% see if it really is a sin curve
figure
plot(x,y)
stg=sprintf('%8.2f',gpu_elapsed_time_with_overhead);
title(['
fontsize{12}\bf GPU y=sin(x) elapsed ',...
    ' time: ' stg ' seconds']);
filename=['gpu_sin_curve.png'];
print(filename,'-dpng');
• These 4 functions all produce the same sinusoid curve as shown below:
Plots of $y = \sin(x)$ for: JIT (2.496367 seconds), parallel JIT with overhead (15.911441 seconds), parallel parfor with overhead (19.328429 seconds), parallel vectorized with overhead (18.703938 seconds) and GPU with overhead (2.110412 seconds) sin curves.
Finally, we can run `statistics_sin.m` that runs these 6 functions, uses the serialized sin solution as the base case and prints the execution times and speedup factors for the last 5 functions versus the serialized function. For the multi-core solutions, we report times for both 2 and 4 core calculations. The code that does this (see `L14statistics_sin.m`) is:

```matlab
N=1e8;

[gpu_elapsed_time_with_overhead,...
     gpu_elapsed_time_without_overhead] = ...;
L14gpu_sin(N);

serialized_elapsed_time = L14serialized_sin(N);
vectorized_elapsed_time = L14vectorized_sin(N);
jit_elapsed_time = L14jit_sin(N);
[jit_parallel_parfor_elapsed_time_with_overhead2,...
     jit_parallel_parfor_elapsed_time_without_overhead2] = ...;
L14jit_parallel_parfor_sin(N,2);
[jit_parallel_parfor_elapsed_time_with_overhead4,...
     jit_parallel_parfor_elapsed_time_without_overhead4] = ...;
```
L14jit_parallel_parfor_sin(N,4);
[parallel_parfor_elapsed_time_with_overhead2,...
parallel_parfor_elapsed_time_without_overhead2]=...
L14parallel_parfor_sin(N,2);
[parallel_parfor_elapsed_time_with_overhead4,...
parallel_parfor_elapsed_time_without_overhead4]=...
L14parallel_parfor_sin(N,4);
[parallel_vectorized_elapsed_time_with_overhead2,...
parallel_vectorized_elapsed_time_without_overhead2]=...
L14parallel_vectorized_sin(N,2);
[parallel_vectorized_elapsed_time_with_overhead4,...
parallel_vectorized_elapsed_time_without_overhead4]=...
L14parallel_vectorized_sin(N,4);

% Assume parallel_serialized_elapsed_time
% is the worst elapsed time - compute
% the speedup factor with respect to
% this time.

fprintf('
Times for programs:
');
fprintf('serialized_elapsed_time: %14.6f
',...
serialized_elapsed_time);
fprintf('vectorized_elapsed_time: %14.6f
',...
vectorized_elapsed_time);
fprintf('jit_elapsed_time: %14.6f
',...
jit_elapsed_time);
fprintf('jit_parallel_parfor_elapsed_time_without_overhead2: %14.6f
',...
jit_parallel_parfor_elapsed_time_without_overhead2);
fprintf('jit_parallel_parfor_elapsed_time_with_overhead2  %14.6f
',
   jit_parallel_parfor_elapsed_time_with_overhead2);
fprintf('jit_parallel_parfor_elapsed_time_without_overhead4: %14.6f
',
   jit_parallel_parfor_elapsed_time_without_overhead4);
fprintf('jit_parallel_parfor_elapsed_time_with_overhead4: %14.6f
',
   jit_parallel_parfor_elapsed_time_with_overhead4);
fprintf('parallel_parfor_elapsed_time_without_overhead2: %14.6f
',
   parallel_parfor_elapsed_time_without_overhead2);
fprintf('parallel_parfor_elapsed_time_with_overhead2: %14.6f
',
   parallel_parfor_elapsed_time_with_overhead2);
fprintf('parallel_parfor_elapsed_time_without_overhead4: %14.6f
',
   parallel_parfor_elapsed_time_without_overhead4);
fprintf('parallel_parfor_elapsed_time_with_overhead4: %14.6f
',
   parallel_parfor_elapsed_time_with_overhead4);
fprintf('parallel_vectorized_elapsed_time_without_overhead2: %14.6f
',
   parallel_vectorized_elapsed_time_without_overhead2);
fprintf('parallel_vectorized_elapsed_time_with_overhead2: %14.6f
',
   parallel_vectorized_elapsed_time_with_overhead2);
fprintf('parallel_vectorized_elapsed_time_without_overhead4: %14.6f
',
   parallel_vectorized_elapsed_time_without_overhead4);
fprintf('parallel_vectorized_elapsed_time_with_overhead4: %14.6f
',
   parallel_vectorized_elapsed_time_with_overhead4);
fprintf('gpu_elapsed_time_without_overhead: %14.6f
',
   gpu_elapsed_time_without_overhead);
fprintf('gpu_elapsed_time_with_overhead: %14.6f
',
   gpu_elapsed_time_with_overhead);
fprintf('

');
% Compute speedup factors
serialized_factor=1.0;
vectorized_factor=serialized_elapsed_time/vectorized_elapsed_time;

jit_factor=serialized_elapsed_time/jit_elapsed_time;
jit_parfor_factor_with_overhead2=... 
    serialized_elapsed_time/jit_parallel_parfor_elapsed_time_with_overhead2;
jit_parfor_factor_without_overhead2=... 
    serialized_elapsed_time/jit_parallel_parfor_elapsed_time_without_overhead2;
jit_parfor_factor_with_overhead4=... 
    serialized_elapsed_time/jit_parallel_parfor_elapsed_time_with_overhead4;
jit_parfor_factor_without_overhead4=... 
    serialized_elapsed_time/jit_parallel_parfor_elapsed_time_without_overhead4;

parallel_parfor_factor_without_overhead2=... 
    serialized_elapsed_time/parallel_parfor_elapsed_time_without_overhead2;
parallel_parfor_factor_with_overhead2=... 
    serialized_elapsed_time/parallel_parfor_elapsed_time_with_overhead2;

parallel_parfor_factor_without_overhead4=... 
    serialized_elapsed_time/parallel_parfor_elapsed_time_without_overhead4;
parallel_parfor_factor_with_overhead4=... 
    serialized_elapsed_time/parallel_parfor_elapsed_time_with_overhead4;

parallel_vectorized_factor_without_overhead2=... 
    serialized_elapsed_time/parallel_vectorized_elapsed_time_without_overhead2;
parallel_vectorized_factor_with_overhead2=...
serialized_elapsed_time/parallel_vectorized_elapsed_time_with_overhead2;

parallel_vectorized_factor_without_overhead4=...
    serialized_elapsed_time/parallel_vectorized_elapsed_time_without_overhead4;
parallel_vectorized_factor_with_overhead4=...
    serialized_elapsed_time/parallel_vectorized_elapsed_time_with_overhead4;

gpu_factor_without_overhead=...
    serialized_elapsed_time/gpu_elapsed_time_without_overhead;
gpu_factor_with_overhead=...
    serialized_elapsed_time/gpu_elapsed_time_with_overhead;

fprintf('Factors:
');
fprintf('serialized_factor: %14.6f
',...
    serialized_factor);
fprintf('vectorized_factor: %14.6f
',...
    vectorized_factor);
fprintf('jit_factor: %14.6f
',...
    jit_factor);
fprintf('jit_parfor_factor_without_overhead2: %14.6f
',...
    jit_parfor_factor_without_overhead2);
fprintf('jit_parfor_factor_with_overhead2: %14.6f
',...
    jit_parfor_factor_with_overhead2);
fprintf('jit_parfor_factor_without_overhead4: %14.6f
',...
    jit_parfor_factor_without_overhead4);
fprintf('jit_parfor_factor_with_overhead4: %14.6f
',...
    jit_parfor_factor_with_overhead4);
fprintf('parallel_parfor_factor_without_overhead2: %14.6f
',...
parallel_parfor_factor_without_overhead2);
fprintf('parallel_parfor_factor_with_overhead2: %14.6f\n',...
parallel_parfor_factor_with_overhead2);
fprintf('parallel_parfor_factor_without_overhead4: %14.6f\n',...
parallel_parfor_factor_without_overhead4);
fprintf('parallel_parfor_factor_with_overhead4: %14.6f\n',...
parallel_parfor_factor_with_overhead4);
fprintf('parallel_vectorized_factor_without_overhead2: %14.6f\n',...
parallel_vectorized_factor_without_overhead2);
fprintf('parallel_vectorized_factor_with_overhead2: %14.6f\n',...
parallel_vectorized_factor_with_overhead2);
fprintf('parallel_vectorized_factor_without_overhead4: %14.6f\n',...
parallel_vectorized_factor_without_overhead4);
fprintf('parallel_vectorized_factor_with_overhead4: %14.6f\n',...
parallel_vectorized_factor_with_overhead4);
fprintf('gpu_factor_without_overhead: %14.6f\n',...
gpu_factor_without_overhead);
fprintf('gpu_factor_with_overhead: %14.6f\n',...
gpu_factor_with_overhead);
fprintf('\n\n');

• For one run (each run produces slightly different results) L14statistics_sin.m prints (for my 2015 Linux laptop):

gpu time without overhead: 0.001314 seconds
gpu time with overhead: 2.110412 seconds
Serialized time: 7.8980 seconds
Vectorized time: 0.986834 seconds
JIT time: 2.496367 seconds
Starting parallel pool (parpool) using the 'local' profile ... connected to 2 workers.
JIT parallel time without overhead: 3.120174 seconds
Parallel pool using the 'local' profile is shutting down.
JIT parallel time with overhead: 20.398485 seconds
Starting parallel pool (parpool) using the 'local' profile ... connected to 4 workers.
JIT parallel time without overhead: 2.900986 seconds
Parallel pool using the 'local' profile is shutting down.
JIT parallel time with overhead: 15.911441 seconds
Starting parallel pool (parpool) using the 'local' profile ... connected to 2 workers.
Parallel pool using the 'local' profile is shutting down.
Parallel parfor elapsed time without overhead: 3.222454 seconds
Parallel parfor elapsed time_with_overhead: 14.159103 seconds
Starting parallel pool (parpool) using the 'local' profile ... connected to 4 workers.
Parallel pool using the 'local' profile is shutting down.
Parallel parfor elapsed time without overhead: 5.711277 seconds
Parallel parfor elapsed time_with_overhead: 19.328429 seconds
Starting parallel pool (parpool) using the 'local' profile ... connected to 2 workers.
Parallel pool using the 'local' profile is shutting down.
Parallel vectorized elapsed time without overhead: 1.319532 seconds
Parallel vectorized elapsed time with overhead: 13.951076 seconds
Starting parallel pool (parpool) using the 'local' profile ...
connected to 4 workers.
Parallel pool using the 'local' profile is shutting down.
Parallel vectorized elapsed time without overhead: 3.715435 seconds
Parallel vectorized elapsed time with overhead: 18.703938 seconds

Times for programs:
serialized_elapsed_time: 7.897986
vectorized_elapsed_time: 0.986834
jit_elapsed_time: 2.496367
jit_parallel_parfor_elapsed_time_without_overhead: 3.120174
jit_parallel_parfor_elapsed_time_with_overhead: 20.398485
jit_parallel_parfor_elapsed_time_without_overhead: 2.900986
jit_parallel_parfor_elapsed_time_with_overhead: 15.911441
parallel_parfor_elapsed_time_without_overhead: 3.222454
parallel_parfor_elapsed_time_with_overhead: 14.159103
parallel_parfor_elapsed_time_without_overhead: 5.711277
parallel_parfor_elapsed_time_with_overhead: 19.328429
parallel_vectorized_elapsed_time_without_overhead: 1.319532
parallel_vectorized_elapsed_time_with_overhead: 13.951076
parallel_vectorized_elapsed_time_without_overhead: 3.715435
parallel_vectorized_elapsed_time_with_overhead: 18.703938
gpu_elapsed_time_without_overhead: 0.001314
gpu_elapsed_time_with_overhead: 2.110412
Factors:
serialized_factor: 1.000000
vectorized_factor: 8.003358
jit_factor: 3.163792
jit_parfor_factor_without_overhead2: 2.531265
jit_parfor_factor_with_overhead2: 0.387185
jit_parfor_factor_without_overhead4: 2.722518
jit_parfor_factor_with_overhead4: 0.496372
parallel_parfor_factor_without_overhead2: 2.450923
parallel_parfor_factor_with_overhead2: 0.557803
parallel_parfor_factor_without_overhead4: 1.382876
parallel_parfor_factor_with_overhead4: 0.408620
parallel_vectorized_factor_without_overhead2: 5.985445
parallel_vectorized_factor_with_overhead2: 0.566120
parallel_vectorized_factor_without_overhead4: 2.125723
parallel_vectorized_factor_with_overhead4: 0.422263
gpu_factor_without_overhead: 6010.643836
gpu_factor_with_overhead: 3.742391

- Note that we allocate 4 cores using `parpool` (the same as `parpool('local',4)`).
- For `parpool('local',4)` we get:

  Starting parallel pool (parpool) using the 'local' profile ... connected to 4 workers.
or

Parallel pool using the local profile is shutting down.

These commands can take considerable time to execute.

- You will notice that the 4 core times are usually worse on my machine. This is because MATLAB reserves the cores that it uses, and my machine only has 4 cores, so it has to reserve processors that are being used for other tasks, such as running MATLAB and my operating system.

- We summarize the results from L14statistics_sin.m into three tables below. for N=1e8 on my 2015 laptop and for N=5e7 for a 2012 Mac laptop and a 2014 iMac desktop:
<table>
<thead>
<tr>
<th>Method</th>
<th>Time (seconds)</th>
<th>Speedup Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Serialized</td>
<td>7.90</td>
<td>1.00</td>
</tr>
<tr>
<td>Vectorized</td>
<td>0.99</td>
<td>8.00</td>
</tr>
<tr>
<td>JIT (single core)</td>
<td>2.50</td>
<td>3.16</td>
</tr>
<tr>
<td>JIT (2 cores, no overhead)</td>
<td>3.12</td>
<td>2.53</td>
</tr>
<tr>
<td>JIT (2 cores, overhead)</td>
<td>20.40</td>
<td>0.39</td>
</tr>
<tr>
<td>JIT (4 cores, no overhead)</td>
<td>2.90</td>
<td>2.72</td>
</tr>
<tr>
<td>JIT (4 cores, overhead)</td>
<td>15.91</td>
<td>0.50</td>
</tr>
<tr>
<td>parallel parfor (2 cores, no overhead)</td>
<td>3.22</td>
<td>2.45</td>
</tr>
<tr>
<td>parallel parfor (2 cores, overhead)</td>
<td>14.16</td>
<td>0.56</td>
</tr>
<tr>
<td>parallel parfor (4 cores, no overhead)</td>
<td>5.71</td>
<td>1.38</td>
</tr>
<tr>
<td>parallel parfor (4 cores, overhead)</td>
<td>19.33</td>
<td>0.41</td>
</tr>
<tr>
<td>parallel vectorized (2 cores, no overhead)</td>
<td>1.32</td>
<td>5.99</td>
</tr>
<tr>
<td>parallel vectorized (2 cores, overhead)</td>
<td>13.95</td>
<td>0.57</td>
</tr>
<tr>
<td>parallel vectorized (4 cores, no overhead)</td>
<td>3.71</td>
<td>2.13</td>
</tr>
<tr>
<td>parallel vectorized (4 cores, overhead)</td>
<td>18.70</td>
<td>0.42</td>
</tr>
<tr>
<td>GPU (no overhead)</td>
<td>0.001314</td>
<td>6010.64</td>
</tr>
<tr>
<td>GPU (overhead)</td>
<td>2.11</td>
<td>3.74</td>
</tr>
</tbody>
</table>

Execution times for the serial/vectorized/jit/parallel methods (when appropriate both overhead and no overhead times using 2 and 4 cores are given) for a mid 2015 Linux laptop.
<table>
<thead>
<tr>
<th>Method</th>
<th>Time (seconds)</th>
<th>Speedup Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Serialized</td>
<td>11.78</td>
<td>1.00</td>
</tr>
<tr>
<td>Vectorized</td>
<td>0.54</td>
<td>21.44</td>
</tr>
<tr>
<td>JIT (single core)</td>
<td>1.97</td>
<td>5.99</td>
</tr>
<tr>
<td>JIT (2 cores, no overhead)</td>
<td>3.90</td>
<td>3.02</td>
</tr>
<tr>
<td>JIT (2 cores, overhead)</td>
<td>17.91</td>
<td>0.66</td>
</tr>
<tr>
<td>JIT (4 cores, no overhead)</td>
<td>2.61</td>
<td>4.50</td>
</tr>
<tr>
<td>JIT (4 cores, overhead)</td>
<td>17.15</td>
<td>0.69</td>
</tr>
<tr>
<td>parallel parfor (2 cores, no overhead)</td>
<td>3.56</td>
<td>3.31</td>
</tr>
<tr>
<td>parallel parfor (2 cores, overhead)</td>
<td>19.85</td>
<td>0.71</td>
</tr>
<tr>
<td>parallel parfor (4 cores, no overhead)</td>
<td>2.61</td>
<td>4.51</td>
</tr>
<tr>
<td>parallel parfor (4 cores, overhead)</td>
<td>19.85</td>
<td>0.59</td>
</tr>
<tr>
<td>parallel vectorized (2 cores, no overhead)</td>
<td>0.58</td>
<td>20.49</td>
</tr>
<tr>
<td>parallel vectorized (2 cores, overhead)</td>
<td>12.71</td>
<td>0.93</td>
</tr>
<tr>
<td>parallel vectorized (4 cores, no overhead)</td>
<td>0.58</td>
<td>20.38</td>
</tr>
<tr>
<td>parallel vectorized (4 cores, overhead)</td>
<td>17.30</td>
<td>0.68</td>
</tr>
<tr>
<td>GPU (no overhead)</td>
<td>0.001036</td>
<td>1138.42</td>
</tr>
<tr>
<td>GPU (overhead)</td>
<td>1.97</td>
<td>5.97</td>
</tr>
</tbody>
</table>

Execution times for the serial/vectorized/jit/parallel methods (when appropriate both overhead and no overhead times using 2 and 4 cores are given) for a late 2013 Mac desktop.
<table>
<thead>
<tr>
<th>Method</th>
<th>Time (seconds)</th>
<th>Speedup Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Serialized</td>
<td>13.90</td>
<td>1.00</td>
</tr>
<tr>
<td>Vectorized</td>
<td>0.57</td>
<td>24.53</td>
</tr>
<tr>
<td>JIT (single core)</td>
<td>2.04</td>
<td>6.81</td>
</tr>
<tr>
<td>JIT (2 cores, no overhead)</td>
<td>5.37</td>
<td>2.58</td>
</tr>
<tr>
<td>JIT (2 cores, overhead)</td>
<td>32.47</td>
<td>0.43</td>
</tr>
<tr>
<td>JIT (4 cores, no overhead)</td>
<td>3.18</td>
<td>4.36</td>
</tr>
<tr>
<td>JIT (4 cores, overhead)</td>
<td>22.10</td>
<td>0.67</td>
</tr>
<tr>
<td>parallel parfor (2 cores, no overhead)</td>
<td>3.17</td>
<td>3.26</td>
</tr>
<tr>
<td>parallel parfor (2 cores, overhead)</td>
<td>22.09</td>
<td>0.63</td>
</tr>
<tr>
<td>parallel parfor (4 cores, no overhead)</td>
<td>4.26</td>
<td>4.38</td>
</tr>
<tr>
<td>parallel parfor (4 cores, overhead)</td>
<td>26.51</td>
<td>0.52</td>
</tr>
<tr>
<td>parallel vectorized (2 cores, no overhead)</td>
<td>0.60</td>
<td>23.22</td>
</tr>
<tr>
<td>parallel vectorized (2 cores, overhead)</td>
<td>17.87</td>
<td>0.78</td>
</tr>
<tr>
<td>parallel vectorized (4 cores, no overhead)</td>
<td>0.62</td>
<td>22.27</td>
</tr>
<tr>
<td>parallel vectorized (4 cores, overhead)</td>
<td>23.07</td>
<td>0.60</td>
</tr>
<tr>
<td>GPU (no overhead)</td>
<td>0.000592</td>
<td>22364.63</td>
</tr>
<tr>
<td>GPU (overhead)</td>
<td>1.33</td>
<td>9.95</td>
</tr>
</tbody>
</table>

Execution times for the serial/vectorized/jit/parallel methods (when appropriate both overhead and no overhead times using 2 and 4 cores are given) for a mid 2012 Mac laptop.
Some comments are in order (based on the mid 2015 Linux machine results):

- The parallel overheads are quite significant. The computation of \( \sin(x) \) is too small of a problem to make a multi-core solution efficient. (According to a MathWorks MATLAB technician, data transfer accounts for most of this overhead.)

- The multi-core vectorized solution (without overhead) is less efficient than the single-core vectorized solution on my machine.

- If we look at the speedup factors (anything greater than 1.0 is better than the serialized solution). We can see that the vectorized solution is the overall winner here, with the JIT and GPU solutions also being very good.
• Without overhead, the parallel vectorized solution is also quite good.

• If more intensive computations were done on the GPU relative to the amount of data transferred its (with overhead) speedup factor would be significantly higher.

• The parallel solutions with overhead are completely infeasible for this problem.
Simple Edge Detection on CPUs versus GPUs

- Most of the edge detectors (except got the Canny edge detector, which happens to be the best edge detector in MATLAB) are overloaded so that they execute on the GPUs. Consider using the Prewitt edge operator, called using the `edge` function with `'prewitt'` specified.

```matlab
% L14prewitt_edge_cpu_vs_gpu.m
%
% BW = edge(I,'prewitt') specifies the Prewitt method.
% BW = edge(I,'prewitt',thresh) specifies the sensitivity
% threshold for the Prewitt method. The edge function
% ignores all edges that are not stronger than thresh.
% If you do not specify thresh, or if thresh is empty ([]),
% edge chooses the value automatically.
% BW = edge(I,'prewitt',thresh,direction) specifies the
% direction of detection for the Prewitt method. direction
% is a string specifying whether to look for 'horizontal'
% or 'vertical' edges or 'both' (default).
```
% [BW,thresh] = edge(I,'prewitt',...) returns the threshold value.

% MatLab supplies a number of ‘‘standard’’ images, 
% one of which is circuit.tif

time1=tic;
I = imread('european-parliament.jpg');
I = rgb2ntsc(I);
I = I(:,:,1);
imshow(I,[]);
% We don’t want to time for the title and print 
% statements as we don’t do this for the GPU 
% calculation: comment out these statements 
% for timing purposes 
title('Original Parliament.png image');
print original_parliament.png -dpng
% Compute edges on CPU

BW1 = edge(I,'prewitt');
time1=toc(time1);
% make edges black and background white by ~ operator 
figure
imshow(~BW1);
title('Prewitt edge detection on CPU');
print prewitt1.png -dpng

% Compute edges on GPU 
time2=tic;
% spread the image onto the GPU - notice
% that another read is done
I = imread('european-parliament.jpg');
I = rgb2ntsc(I);
I = I(:,:,1);
BW2 = edge(I,'prewitt');
time2=toc(time2);
figure,
imshow(~BW2)
title('Prewitt edge detection on GPU');
print prewitt2.png -dpng

fprintf('CPU time for prewitt edge calculation: %f\n',time1);
fprintf('GPU time for prewitt edge calculation: %f\n',time2);
fprintf('Speedup of GPU (without overhead) over CPU: %f\n',time1/time2);

- This function produces three images:
The european_parliament.jpg image of a parliament building in Europe, its CPU edge map and its GPU edge map (which are identical).
• The program produces the following output:

CPU time for prewitt edge calculation: 0.882636
GPU time for prewitt edge calculation: 0.097697
Speedup of GPU (without overhead) over CPU: 9.034423

• The GPU time is faster than the CPU time as indicated by the GPU speedup factor (the title and print statements were commented out for the CPU timing). Running this program a multiple number of times and taking the average is a better way to draw a meaningful conclusion.