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**" 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. For example, I/O can often be done independently of other work units.

- Much multi-threading is already built into many MatLab functions [for example, the **fft** function (Fast Fourier Transform function), various filtering functions such as **imfilter** and various matrix operations already use parallelism when the hardware is available].

- Currently, you can have up to 12 cores (12 MatLab workers) working on a parallel solution to your problem on a single machine.

- Typically, 12 cores (the maximum MatLab can handle now) 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.

- Note that if you select all the cores on a machine, nothing else will run on that machine until these cores are released!!! It may be difficult to interrupt running code and a **hard** reboot (turn your machine off and

then back on again) may be required to do so!!!

- 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 will eventually be deprecated in future versions of MatLab.

- The hyperlink *www.mathworks.com/products/parallel-computing/builtin-parallel-support.html* 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.

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 (2GB-4GB) 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\n',max(res(:)));
```

- Each `res(i)` is computed independently of other `res(i)` values, so the order of the iterations is unimportant and the loop is embarrassingly parallel

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:

```
% 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);
```

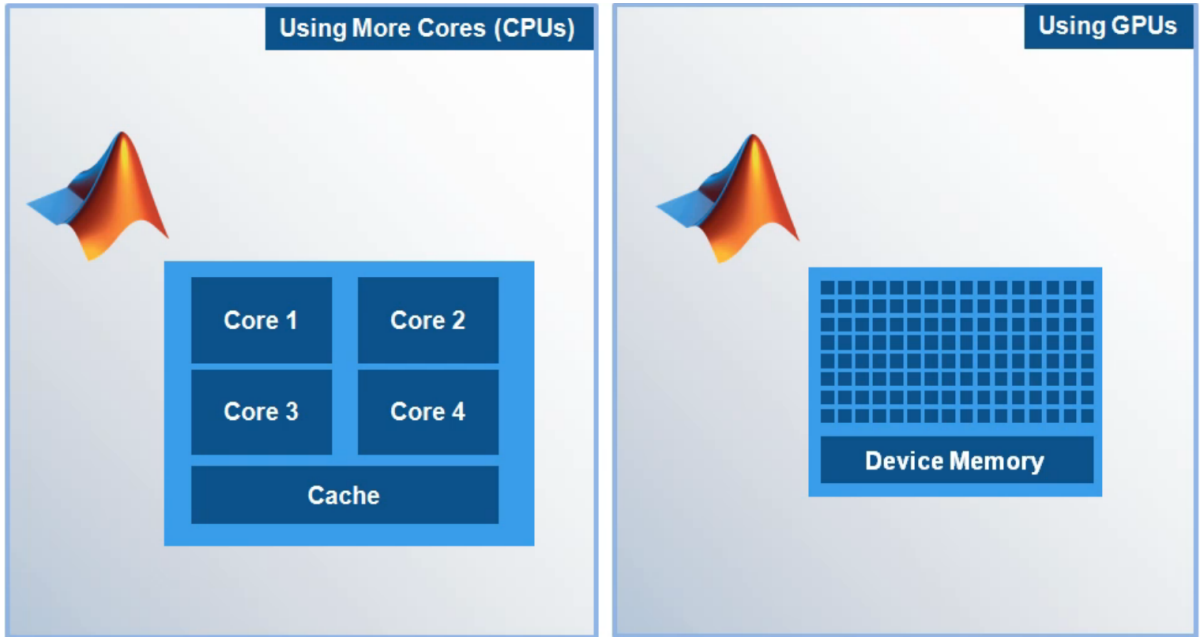
```
set(job,'MyFiles',{'SDs.m','SDt.m','SDb.m','SDq.m'});
% Create tasks - pass the function handles, the number
% of output arguments and a cell array of the
% input parameters
createTask(job,@SDs,1,{n,k,b,totalTime});
createTask(job,@SDt,1,{n,k,b,totalTime});
createTask(job,@SDb,1,{n,k,b,totalTime});
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 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 for the complete story on using these functions.

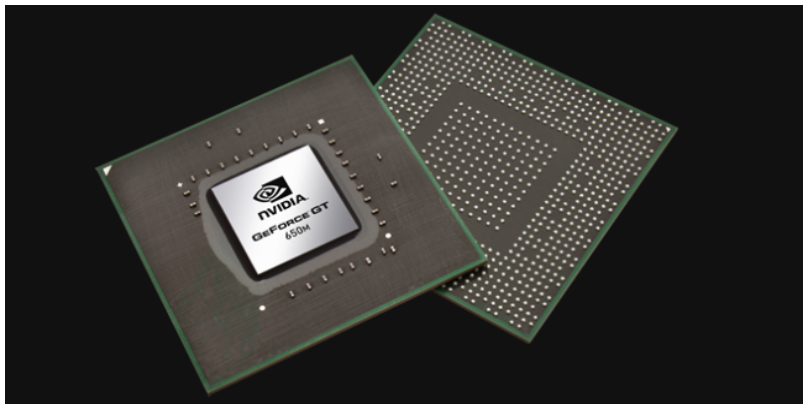
Using GPUs (Graphical Processing Units)

- 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.



NVIDIA geForce XT650M GPU processing unit.

- MatLab requires NVIDIA (Tesla 1000 and 2000 series or better) GPUs, with a Computing Capacity of 1.3 or more. Use MatLab command `gpuDevice` to find out your value (my notebook has a compute capacity of 3.0). Values less than 1.3 usually mean the GPU cannot do floating point arithmetic, a serious restriction for scientific computing! If you 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). It is possible that in

the future, there will be one GPU for each core! A MatLab webinar on GPU computing has hinted at this!

- 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).

abs	complex	filter	ipermute	mldivide	sec
acos	cond	filter2	iscolumn	mod	sech
acosh	conj	find	isempty	mpower	shiftdim
acot	conv	fft	isequal	mrdivide	sign
acoth	conv2	fft2	isequaln	mtimes	sin
acsc	convn	fftn	isfinite	NaN	single
acsch	cos	fftshift	isfloat	ndgrid	sinh
all	cosh	fix	isinf	ndims	size
angle	cot	flip	isinteger	ne	sort
any	coth	fliplr	islogical	nnz	sprintf
arrayfun	cov	flipud	ismatrix	norm	sqrt
asec	cross	floor	ismember	normest	squeeze
asech	csc	fprintf	isnan	not	std
asin	csch	full	isnumeric	num2str	sub2ind
asinh	ctranspose	gamma	isreal	numel	subsasgn
atan	cumprod	gammaln	isrow	ones	subsindex
atan2	cumsum	gather	issorted	pagefun	subsref
atanh	det	ge	issparse	perms	sum
beta	diag	gt	isvector	permute	svd
betaln	diff	horzcat	kron	plot (and	tan
bitand	disp	hypot	ldivide	related)	tanh
bitcmp	display	ifft	le	plus	times
bitget	dot	ifft2	length	pow2	trace
bitor	double	ifftn	log	power	transpose
bitset	eig	ifftshift	log10	prod	tril
bitshift	eps	imag	log1p	qr	triu
bitxor	eq	ind2sub	log2	rank	true
blkdiag	erf	inf	logical	rdivide	uint16
bsxfun	erfc	int16	lt	real	uint32
cast	erfcinv	int2str	lu	reallog	uint64
cat	erfcx	int32	mat2str	realpow	uint8
ceil	erfinv	int64	max	realsqrt	uminus
chol	exp	int8	mean	rem	uplus
circshift	expm1	interp1	meshgrid	repmat	var
classUnderlying	eye	interp2	min	reshape	vertcat
colon	false	inv	minus	rot90	zeros
				round	

GPU overloaded functions.

- My laptop is a mid-2012 4 core Mac notebook with a NVIDIA Geforce GT650M GPU device with 1GB of memory and 384 shader cores.
- My desktop is a early-2014 4-core iMac with a NVIDIA Geforce GTX780M with 4GB of memory and 1536 shader cores.
- 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 i newer macs (alas)! There are rumors that Apple will return to NVIDIA in the future.
- The GPU cores are limited in what they can do (for example no direct access to I/O and limited memory [no virtual memory so you have to

manage the memory yourself]. 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. Since my laptop has 384 renders, my screen potentially could be overwhelmed by 384 figures plotted at roughly the same time (so don't do this)!!!
- 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 than in computations. Calculations that are iterative, like computing the flow of air around a moving car/flying airplane are great applications 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, `fft` or `ifft` compute the forward and inverse fast Fourier transforms 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 un-overloaded MatLab functions on the GPUs you can use the command **arrayfun**. For example, `A=arrayfun(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,...)=arrayfun(FUN,B(i,j,...))`.

- Use the **gather** command to bring the data back from the GPUs to the CPUs. 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 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.
- It is also easy to scale up (minimal code changes required) your multi-core CPU and GPU solutions to distributed clusters of computers. [In

Technology	Example	MATLAB Workers	Execution Target
matlabpool	`parfor`	Yes	CPU cores
user-defined tasks (batch processing)	`createJob` `createTask`	Yes	CPU cores
GPU-based parallelism	`GPUArray`	No	NVIDIA GPU with Compute Capability 1.3 or greater

Recap of multi-core CPU use versus GPU use.

this course, we only address the question of how to use the local resources on your individual machine to parallelize your code.]

- In a Mathworks webinar dedicated to GPU programming, the presenter mentioned at the end of her presentation something about assigning GPUs to cores. Typically, there is only one GPU device per machine so

this may be a hint about what the future holds. It would be interesting to write code that uses GPU programming on each up to 12 multi-cores in parallel.

- It should be noted that GPU programming is in a state of flux at Mathworks right now and many changes are in the pipeline now. For example, the 2013b release provides 23 new image processing overloaded GPU functions over the 2013a release. A hint: always update your MatLab to the latest version (if you have that kind of license) if you are interested in parallel programming.

Multi-Core and GPU Programming Examples

- Finally, we consider some MatLab parallel programming! Consider a MatLab program that computes the $y = \sin(x)$ for 50,000,000 values of x ranging from 0 to 2π. 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 **J**ust **I**n **T**ime solution,

4. the multi-core **J**ust **I**n **T**ime 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):

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Serialized Solution                            %
% compute y=sin(x) for N values of x             %
% ranging from 0 to 2*pi                         %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [serialized_elapsed_time]=L14serialized_sin(N)

serialized_elapsed_time=tic;

% 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
```

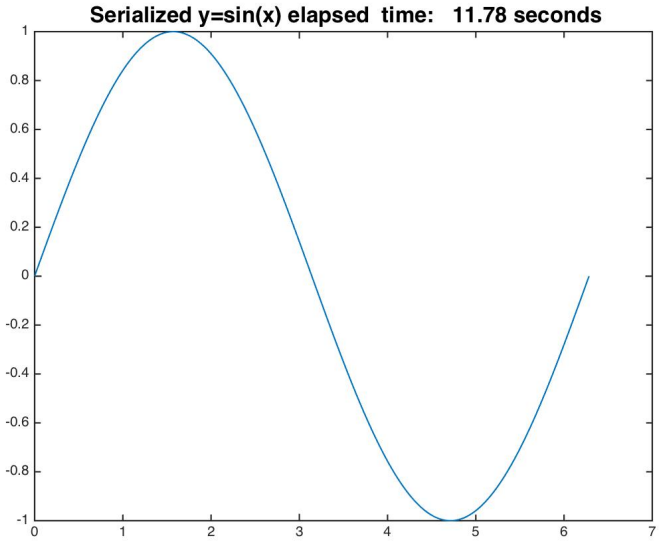
```
plot(x,y)
stg=sprintf('%8.4f',serialized_elapsed_time);
title(['\fontsize{16}\bf Serialized y=sin(x) elapsed ',...
       ' time:' stg ' seconds']);
filename=['serialized_sin_curve.jpg'];
print(filename,'-djpeg');
```

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

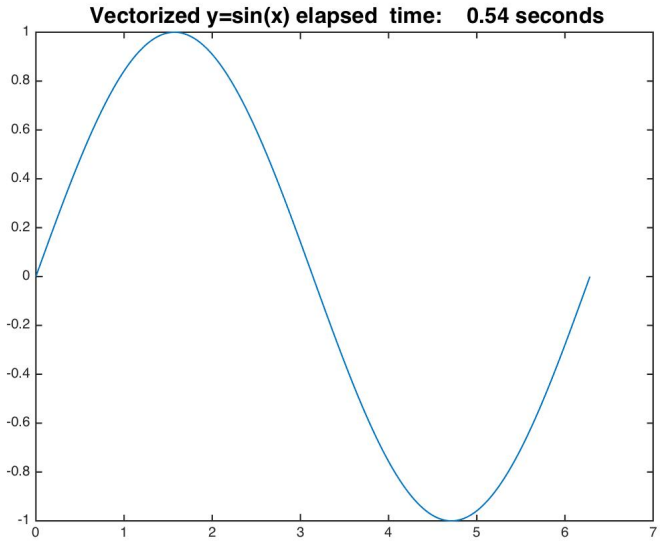
```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Vectorized Solution                           %
% compute y=sin(x) for N values of x            %
% ranging from 0 to 2*pi                        %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [vectorized_elapsed_time]=L14vectorized_sin(N)

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);
```

```
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{16}\bf Vectorized y=sin(x) elapsed ',...
       ' time:' stg ' seconds']);
filename=['vectorized_sin_curve.jpg'];
print(filename,'-djpeg');
```

- 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 of the loop.



(a) Serialized solution



(b) Vectorized solution

Plot of $y = \sin(x)$ from the serialized function (13.6536 seconds) and the vectorized solution (0.9645 seconds)

- We present the single core **L14jit_sin.m** function below:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% JIT Solution                                   %
% compute y=sin(x) for N values of x             %
% ranging from 0 to 2*pi                         %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
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
jit_elapsed_time=toc(jit_elapsed_time);
fprintf('JIT time:        %8.4f seconds\n',...
         jit_elapsed_time);
% see if it really is a sin curve
figure
plot(x,y)
```

```
stg=sprintf('%8.4f',jit_elapsed_time);
title(['\fontsize{16}\bf JIT y=sin(x) elapsed ',...
       ' time:' stg ' seconds']);
filename=['jit_sin_curve.jpg'];
print(filename,'-djpeg');
```

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

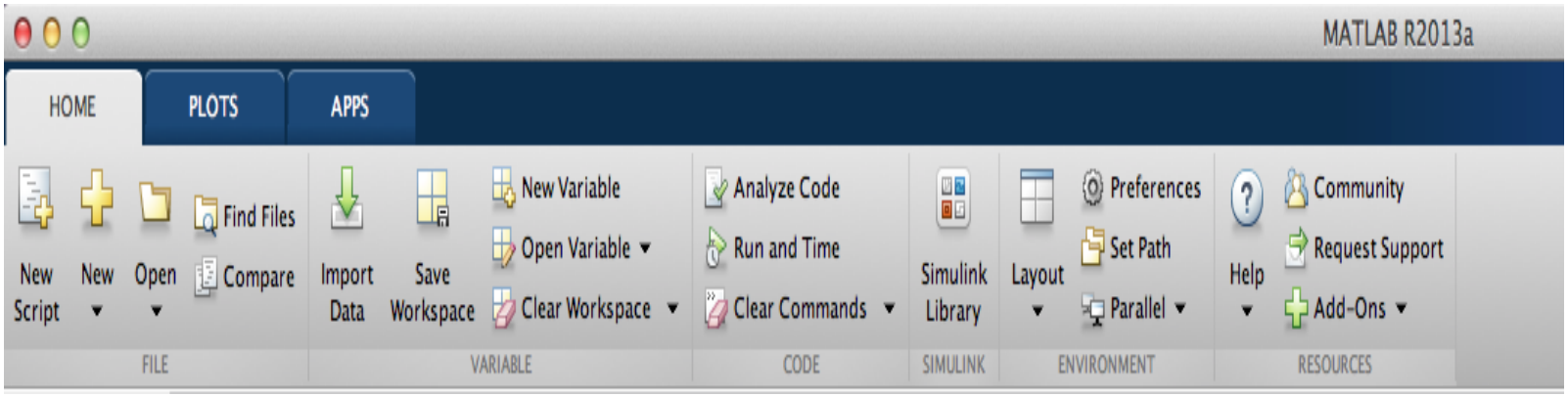
```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% JIT Solution                               %
% compute y=sin(x) for N values of x         %
% ranging from 0 to 2*pi                     %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [jit_parallel_parfor_elapsed_time_with_overhead,...
          jit_parallel_parfor_elapsed_time_without_overhead]=...
         L14jit_parallel_parfor_sin(N,num_cores)

jit_parallel_parfor_elapsed_time_with_overhead=tic;
```

```
% 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\n',...
         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(['\fontsize{16}\bf JIT y=sin(x) elapsed ',...
       ' time for parfor:' stg ' seconds']);
filename=['jit_parfor_sin_curve.jpg'];
print(filename,'-djpeg');
```

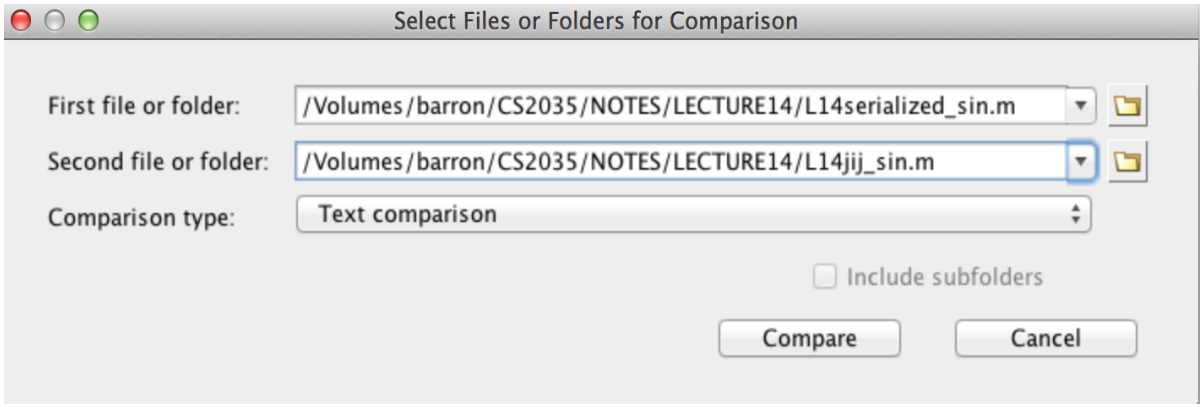
- `delete(gcp('nocreate'))` deletes the current pool. `gcp` is a MatLab function that returns the current pool handle. Thus, `p=gcp('nocreate')` returns the current pool handle if one exists. If no pool exists, the 'nocreate' option ensures that a pool is not created.

- 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 (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. Red lines indicate changes, green lines indicate new lines and un-coloured lines mean common text.

L14serialized_sin.m vs. L14jit_sin.m

8 differences found. Use the toolstrip buttons to navigate to them.

1	%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%	.	%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% 1
2	% Serialized Solution %	x	% JIT Solution % 2
3	% compute y=sin(x) for N values of x %	.	% compute y=sin(x) for N values of x % 3
4	% ranging from 0 to 2*pi %	.	% ranging from 0 to 2*pi % 4
5	%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%	.	%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% 5
6	function [serialized_elapsed_time]=L14serialized_sin(N)	x	function [jit_elapsed_time]=L14jit_sin(N) 6
7		.	7
8	serialized_elapsed_time=tic;	x	jit_elapsed_time=tic; 8
9		.	9
10	% Don't allocate x or y outside the loop,	x	% Allocate x and y outside the loop, 10
11	% this forces the code to be serial	x	% this allows the JIT (Just In Time) 11
12	% and JIT not to be used	x	% compiler to be used as the data 12
-		>	% structures inside the do not change 13
-		>	14
13	x=linspace(0,2*pi,N);	.	x=linspace(0,2*pi,N); 15
-		>	y=zeros(N,1,'double'); 16
-		>	% compute x and y values 17
14	for i=1:N	.	for i=1:N 18
15	y(i)=sin(x(i));	.	y(i)=sin(x(i)); 19
16	end	.	end 20
17	serialized_elapsed_time=toc(serialized_elapsed_time);	x	jit_elapsed_time=toc(jit_elapsed_time); 21
18	fprintf('Serialized time: %8.4f seconds\n',...	x	fprintf('JIT time: %8.4f seconds\n',... 22
19	serialized_elapsed_time);	x	jit_elapsed_time); 23
20	% see if it really is a sin curve	.	% see if it really is a sin curve 24
21	figure	.	figure 25
22	plot(x,y)	.	plot(x,y) 26
23	stg=sprintf('%8.4f',serialized_elapsed_time);	x	stg=sprintf('%8.4f',jit_elapsed_time); 27
24	title(['\fontsize{16}\bf Serialized y=sin(x) elapsed ',...	x	title(['\fontsize{16}\bf JIT y=sin(x) elapsed ',... 28
25	' time:' stg ' seconds']);	.	' time:' stg ' seconds']); 29
26	filename=['serialized_sin_curve.jpg'];	x	filename=['jit_sin_curve.jpg']; 30
27	print(filename,'-djpeg');	.	print(filename,'-djpeg'); 31

Number of matching lines: 15

Number of unmatched lines in left file: 12

Number of unmatched lines in right file: 16

Compare serialized and jit solutions.

- We present **L14parallel_parfor_sin.m** below. Again we have a parameter for `num_cores`.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% 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=sprintf('%8.2f',parallel_parfor_elapsed_time_with_overhead);
title(['\fontsize{16}\bf Parallel parfor y=sin(x) elapsed ',...
       ' time:' stg ' seconds']);
filename=['parallel_parfor_sin_curve.jpg'];
print(filename,'-djpeg');
```

- We present **L14parallel_vectorized_sin.m** function next:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% 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]=...
          L14parallel_vectorized_sin(N,num_cores)

parallel_vectorized_elapsed_time_with_overhead=tic;

% If the number of cores if not specified
% it is the total number of cores available
parpool('local',num_cores);

parallel_vectorized_elapsed_time_without_overhead=tic;

% compute x and y values
x=linspace(0,2*pi,N);
y=sin(x);

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

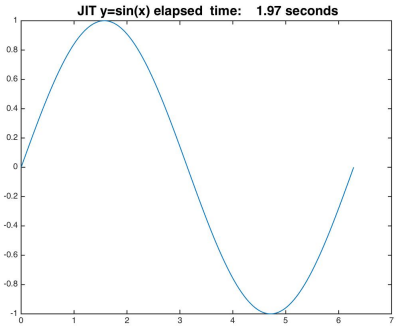
```
parallel_vectorized_elapsed_time_with_overhead=...
        toc(parallel_vectorized_elapsed_time_with_overhead);

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(['\fontsize{16}\bf Parallel vectorized y=sin(x) elapsed ',...
       ' time:' stg ' seconds']);
filename=['parallel_vectorized_sin_curve.jpg'];
print(filename,'-djpeg');
```

- Finally, we present the GPU sin function (L14gpu_sin.m) last:

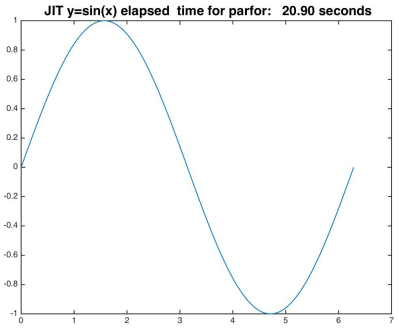
```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% GPU Solution                                  %
% compute y=sin(x) for N values of x            %
% ranging from 0 to 2*pi                        %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [gpu_elapsed_time_with_overhead...
          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('\n');
fprintf('gpu time without overhead: %10.6f seconds\n',...
         gpu_elapsed_time_without_overhead);
fprintf('gpu time with overhead:    %10.6f seconds\n',...
         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{16}\bf GPU y=sin(x) elapsed ',...
       ' time:' stg ' seconds']);
filename=['gpu_sin_curve.jpg'];
print(filename,'-djpeg');
```

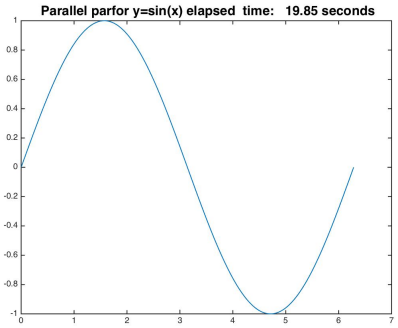
- These 4 functions all produce the same sinusoid curve as shown below:



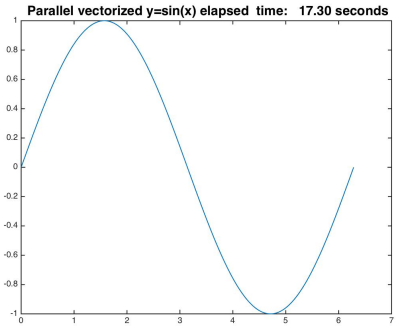
JIT sin



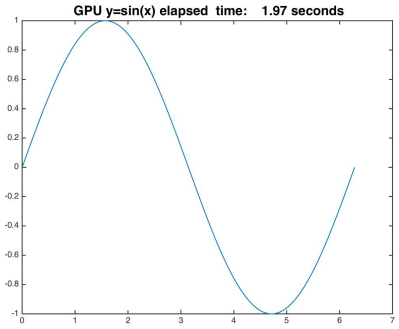
JIT parfor sin



Parallel parfor sin



Parallel vectorized sin



GPU sin

Plots of $y = \sin(x)$ for: JIT (1.97 seconds), parallel JIT with overhead (20.90 seconds), parallel parfor with overhead (19.05 seconds), parallel vectorized with overhead (17.30 seconds) and GPU with overhead (1.97 seconds) sin curves.

- Finally, we can run **statistics_sin.m** that runs these 6 functions, uses the 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:

```
N=50000000;
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);
[gpu_elapsed_time_with_overhead,...
 gpu_elapsed_time_without_overhead]=...
         L14gpu_sin(N);


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

fprintf('\nTimes for programs:\n');
fprintf('serialized_elapsed_time:                              %14.6f\n',...
         serialized_elapsed_time);
fprintf('vectorized_elapsed_time:                              %14.6f\n',...
         vectorized_elapsed_time);
fprintf('jit_elapsed_time:                                     %14.6f\n',...
         jit_elapsed_time);
fprintf('jit_parallel_parfor_elapsed_time_without_overhead2: %14.6f\n',...
         jit_parallel_parfor_elapsed_time_without_overhead2);
fprintf('jit_parallel_parfor_elapsed_time_with_overhead2     %14.6f\n',...
         jit_parallel_parfor_elapsed_time_with_overhead2);
fprintf('jit_parallel_parfor_elapsed_time_without_overhead4: %14.6f\n',...
```

```
        jit_parallel_parfor_elapsed_time_without_overhead4);
fprintf('jit_parallel_parfor_elapsed_time_with_overhead4:    %14.6f\n',...
        jit_parallel_parfor_elapsed_time_with_overhead4);
fprintf('parallel_parfor_elapsed_time_without_overhead2:     %14.6f\n',...
        parallel_parfor_elapsed_time_without_overhead2);
fprintf('parallel_parfor_elapsed_time_with_overhead2:        %14.6f\n',...
        parallel_parfor_elapsed_time_with_overhead2);
fprintf('parallel_parfor_elapsed_time_without_overhead4:     %14.6f\n',...
        parallel_parfor_elapsed_time_without_overhead4);
fprintf('parallel_parfor_elapsed_time_with_overhead4:        %14.6f\n',...
        parallel_parfor_elapsed_time_with_overhead4);
fprintf('parallel_vectorized_elapsed_time_without_overhead2: %14.6f\n',...
        parallel_vectorized_elapsed_time_without_overhead2);
fprintf('parallel_vectorized_elapsed_time_with_overhead2:    %14.6f\n',...
        parallel_vectorized_elapsed_time_with_overhead2);
fprintf('parallel_vectorized_elapsed_time_without_overhead4: %14.6f\n',...
        parallel_vectorized_elapsed_time_without_overhead4);
fprintf('parallel_vectorized_elapsed_time_with_overhead4:    %14.6f\n',...
        parallel_vectorized_elapsed_time_with_overhead4);
fprintf('gpu_elapsed_time_without_overhead:                  %14.6f\n',...
        gpu_elapsed_time_without_overhead);
fprintf('gpu_elapsed_time_with_overhead:                     %14.6f\n',...
        gpu_elapsed_time_with_overhead);
fprintf('\n\n');

% 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:\n');
fprintf('serialized_factor:                              %14.6f\n',...
         serialized_factor);
fprintf('vectorized_factor:                              %14.6f\n',...
         vectorized_factor);
fprintf('jit_factor:                                     %14.6f\n',...
         jit_factor);
fprintf('jit_parfor_factor_without_overhead2:            %14.6f\n',...
         jit_parfor_factor_without_overhead2);
fprintf('jit_parfor_factor_with_overhead2:               %14.6f\n',...
         jit_parfor_factor_with_overhead2);
fprintf('jit_parfor_factor_without_overhead4:            %14.6f\n',...
         jit_parfor_factor_without_overhead4);
fprintf('jit_parfor_factor_with_overhead4:               %14.6f\n',...
         jit_parfor_factor_with_overhead4);
fprintf('parallel_parfor_factor_without_overhead2:       %14.6f\n',...
         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 2014 Mac desktop):

```
Serialized time:  11.7753 seconds
Vectorized time:   0.541531 seconds
JIT time:          1.966768 seconds
Starting parallel pool (parpool) using the 'local' profile ... connected to 2
JIT parallel time without overhead:          3.903425 seconds
Parallel pool using the 'local' profile is shutting down.
JIT parallel time with overhead:         17.907350 seconds
```

```
Starting parallel pool (parpool) using the 'local' profile ... connected to 4
JIT parallel time without overhead:          2.614843 seconds
Parallel pool using the 'local' profile is shutting down.
JIT parallel time with overhead:         17.145282 seconds
Starting parallel pool (parpool) using the 'local' profile ... connected to 2
Parallel pool using the 'local' profile is shutting down.
Parallel parfor elapsed time without overhead:   3.561287 seconds
Parallel parfor elapsed time_with_overhead:     16.586685 seconds
Starting parallel pool (parpool) using the 'local' profile ... connected to 4
Parallel pool using the 'local' profile is shutting down.
Parallel parfor elapsed time without overhead:   2.611197 seconds
Parallel parfor elapsed time_with_overhead:     19.851586 seconds
Starting parallel pool (parpool) using the 'local' profile ... connected to 2
Parallel pool using the 'local' profile is shutting down.
Parallel vectorized elapsed time without overhead:   0.574648 seconds
Parallel vectorized elapsed time with overhead:     12.714834 seconds
Starting parallel pool (parpool) using the 'local' profile ... connected to 4
Parallel pool using the 'local' profile is shutting down.
Parallel vectorized elapsed time without overhead:   0.577902 seconds
Parallel vectorized elapsed time with overhead:     17.302423 seconds
gpu time without overhead:   0.001035 seconds
gpu time with overhead:      1.973697 seconds

Times for programs:
serialized_elapsed_time:                                  11.775339
vectorized_elapsed_time:                                   0.541531
jit_elapsed_time:                                          1.966768
jit_parallel_parfor_elapsed_time_without_overhead2:        3.903425
```

```
jit_parallel_parfor_elapsed_time_with_overhead2          17.907350
jit_parallel_parfor_elapsed_time_without_overhead4:       2.614843
jit_parallel_parfor_elapsed_time_with_overhead4:         17.145282
parallel_parfor_elapsed_time_without_overhead2:           3.561287
parallel_parfor_elapsed_time_with_overhead2:             16.586685
parallel_parfor_elapsed_time_without_overhead4:           2.611197
parallel_parfor_elapsed_time_with_overhead4:             19.851586
parallel_vectorized_elapsed_time_without_overhead2:       0.574648
parallel_vectorized_elapsed_time_with_overhead2:         12.714834
parallel_vectorized_elapsed_time_without_overhead4:       0.577902
parallel_vectorized_elapsed_time_with_overhead4:         17.302423
gpu_elapsed_time_without_overhead:                        0.001035
gpu_elapsed_time_with_overhead:                           1.973697


Factors:
serialized_factor:                                        1.000000
vectorized_factor:                                       21.744541
jit_factor:                                               5.987152
jit_parfor_factor_without_overhead2:                      3.016668
jit_parfor_factor_with_overhead2:                         0.657570
jit_parfor_factor_without_overhead4:                      4.503268
jit_parfor_factor_with_overhead4:                         0.686798
parallel_parfor_factor_without_overhead2:                 3.306484
parallel_parfor_factor_with_overhead2:                    0.709927
parallel_parfor_factor_without_overhead4:                 4.509556
parallel_parfor_factor_with_overhead4:                    0.593169
parallel_vectorized_factor_without_overhead2:            20.491393
```

```
parallel_vectorized_factor_with_overhead2:          0.926110
parallel_vectorized_factor_without_overhead4:      20.376002
parallel_vectorized_factor_with_overhead4:          0.680560
gpu_factor_without_overhead:                    11381.416359
gpu_factor_with_overhead:                           5.966134
```

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

```
Starting 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.

- We summarize the results from **L14statistics_sin.m** into a two table below for a 2012 Mac laptop and a 2014 iMac desktop:

Method	Time (seconds)	Speedup Factor
Serialized	11.78	1.00
Vectorized	0.54	21.44
JIT (single core)	1.97	5.99
JIT (2 cores, no overhead)	3.90	3.02
JIT (2 cores, overhead)	17.91	0.66
JIT (4 cores, no overhead)	2.61	4.50
JIT (4 cores, overhead)	17.15	0.69
parallel parfor (2 cores, no overhead)	3.56	3.31
parallel parfor (2 cores, overhead)	19.85	0.71
parallel parfor (4 cores, no overhead)	2.61	4.51
parallel parfor (4 cores, overhead)	19.85	0.59
parallel vectorized (2 cores, no overhead)	0.58	20.49
parallel vectorized (2 cores, overhead)	12.71	0.93
parallel vectorized (4 cores, no overhead)	0.58	20.38
parallel vectorized (4 cores, overhead)	17.30	0.68
GPU (no overhead)	0.001036	1138.42
GPU (overhead)	1.97	5.97

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.

Method	Time (seconds)	Speedup Factor
Serialized	13.90	1.00
Vectorized	0.57	24.53
JIT (single core)	2.04	6.81
JIT (2 cores, no overhead)	5.37	2.58
JIT (2 cores, overhead)	32.47	0.43
JIT (4 cores, no overhead)	3.18	4.36
JIT (4 cores, overhead)	22.10	0.67
parallel parfor (2 cores, no overhead)	3.17	3.26
parallel parfor (2 cores, overhead)	22.09	0.63
parallel parfor (4 cores, no overhead)	4.26	4.38
parallel parfor (4 cores, overhead)	26.51	0.52
parallel vectorized (2 cores, no overhead)	0.60	23.22
parallel vectorized (2 cores, overhead)	17.87	0.78
parallel vectorized (4 cores, no overhead)	0.62	22.27
parallel vectorized (4 cores, overhead)	23.07	0.60
GPU (no overhead)	0.000592	22364.63
GPU (overhead)	1.33	9.95

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:

- 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 accounted for most of this overhead.]

- The multi-core vectorized solution (without overhead) is only slightly more efficient than the single-core vectorized solution.

- If we look at the speedup factors (anything greater than 1.0 is better than the serialized solution). We can see that the vectorized and JIT solutions are the best. Without overhead, the parallel vectorized solution is also quite good. The gpu solution also gives respectable performance. If more

intensive computations were done on the GPU relative to the amount of data transferred its speedup factor would be significantly higher.

- The parallel solutions with overhead are completely infeasible for this problem. The vectorized solution without overhead is pretty good.

Simple Edge Detection on CPUs versus GPUs

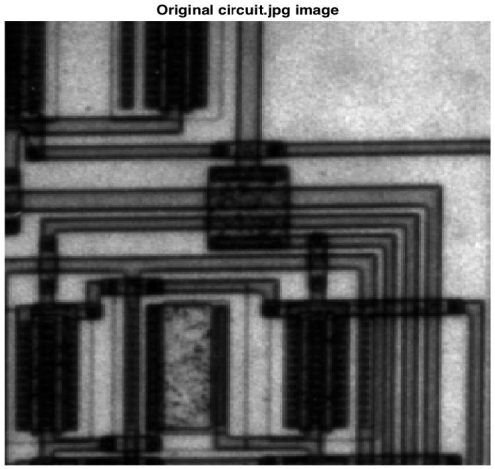
- Most of the edge detectors (except got the Canny edge detector, which happens to be the best edge detector in MatLab) are overloaded so that they execute on the GPUs. Consider using the Prewitt edge operator, called using the **edge** function with **'prewitt'** specified.

```
% 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,
```

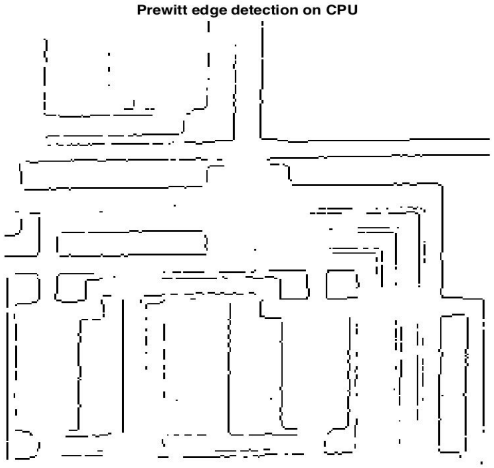
```
% one of which is circuit.tif
% Look in /Applications/MATLAB_R2014b.app/toolbox/images/imdata
% for the images for MatLab 2014b.

I = imread('circuit.tif');
imshow(I,[]);
time1=tic;

% 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 circuit.jpg image');
print original_circuit.jpg -djpeg
% 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.jpg -djpeg

% Compute edges on GPU
time2=tic;
% spread the image onto the GPU - notice
% that another read is done
```

```
I = gpuArray(imread('circuit.tif'));
BW2 = edge(I,'prewitt');
time2=toc(time2);
figure,
imshow(~BW2)
title('Prewitt edge detection on GPU');
print prewitt2.jpg -djpeg

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);
```

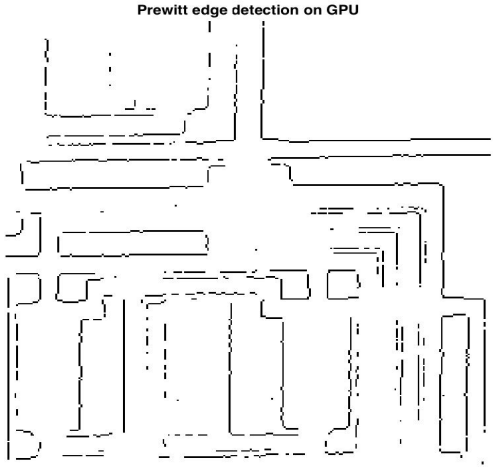
- This function produces three images:



circuit.tif



CPU edges



GPU edges

MatLab supplied circuit.tif image of a circuit board, its CPU edge map and its GPU edge map (which are identical).

- The program produces the following output:

```
CPU time for prewitt edge calculation: 0.063212
GPU time for prewitt edge calculation: 0.024766
Speedup of GPU (without overhead) over CPU: 2.552396
```

- 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.