# Matlab on GPUs

Jerry Ebalunode

Senior Researcher

HPE DSI

uh.edu/datascience

# Objective

- Learn how to run MATLAB codes on NVIDIA CUDA-enabled GPUs

# Overview

- Review
  - Basics
  - Parallel MATLAB
- MATLAB on GPUs
  - Built-in functions
  - Elementwise operations
    - gpuArrays
  - GPU Code porting
- Conclusion

# Review

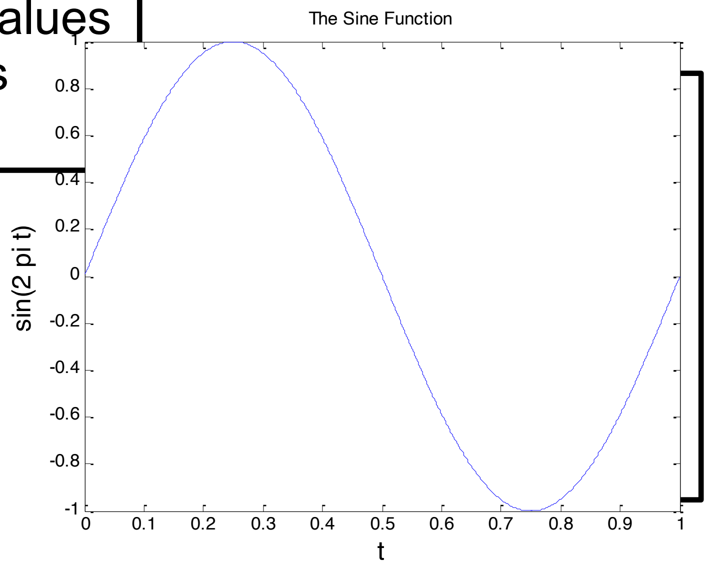
- MATLAB Introduction
  - Variables, arrays, matrices, etc
    - Arrays are the fundamental units of data
  - Operators
  - Flow control
    - if, while, for
  - Using M files
    - Scripts and functions
  - Plots

```
>>sum(M)
```

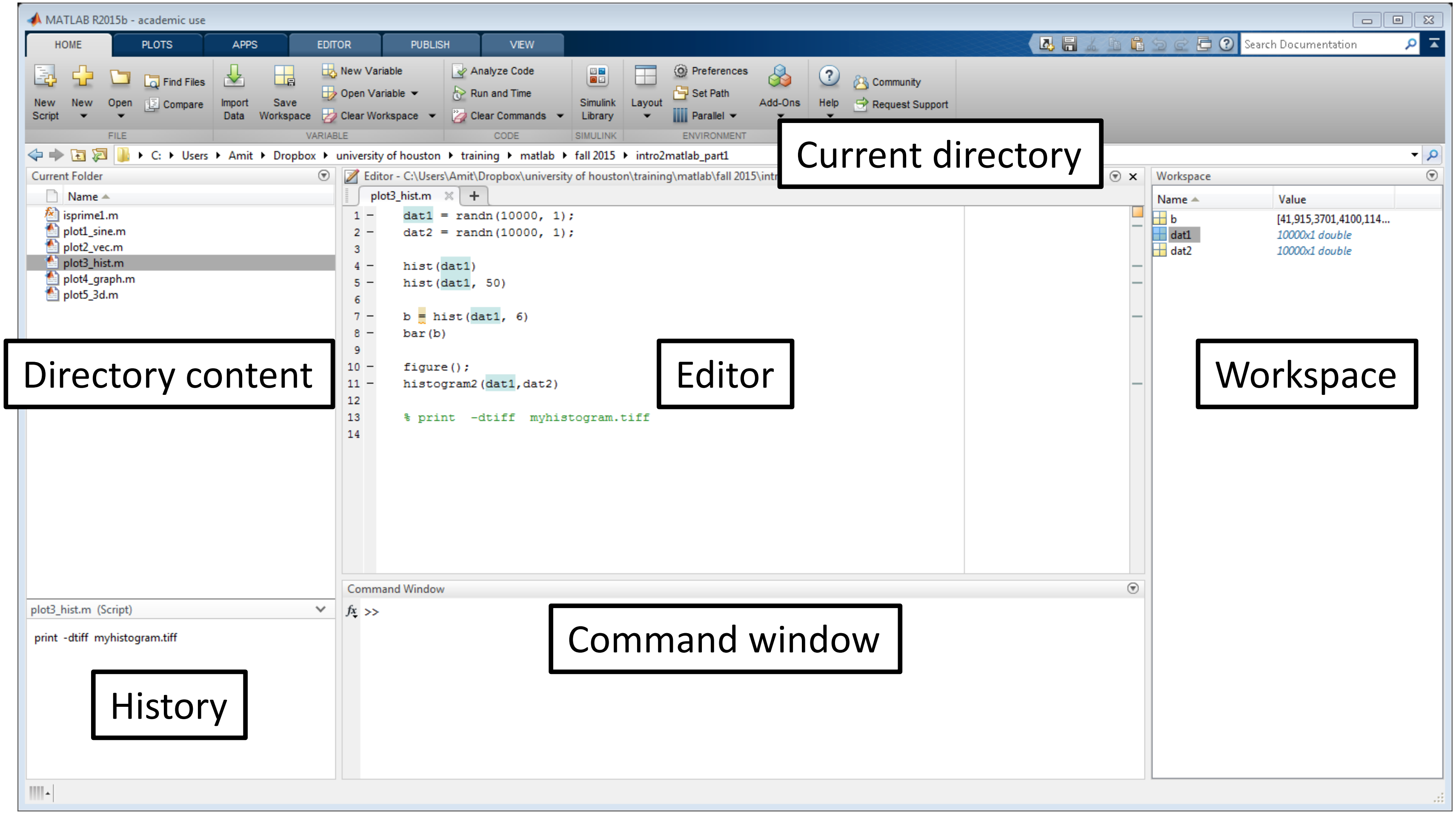
```
>>a= [1 2 3; 4 5 6; 7 8 9; 10 11 12];
```

Column major

```
for index = values
    statements
end
```



# Review – MATLAB GUI

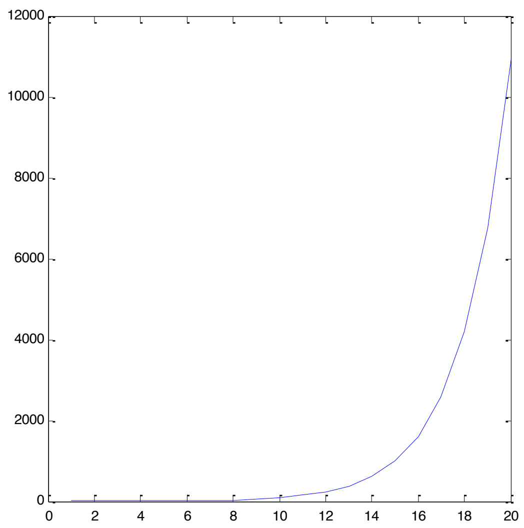


# Review exercise

- Write a program to plot the Fibonacci series – first 20 numbers in the series

- Hint: use the fibonacci.m function file

- >>plot(fibonacci(20));



# Review – parallel MATLAB

- Parallel MATLAB
  - extension of MATLAB that takes advantage of multicore desktop machines, GPUs and clusters.

- The Parallel Computing Toolbox or PCT runs on a desktop, and can take advantage of cores (R2014a has no limit, R2013b limit is 12, ...). Parallel programs can be run interactively or in batch.

- The MATLAB Distributed Computing Server (MDCS) controls parallel execution of MATLAB on a cluster with tens or hundreds of cores.

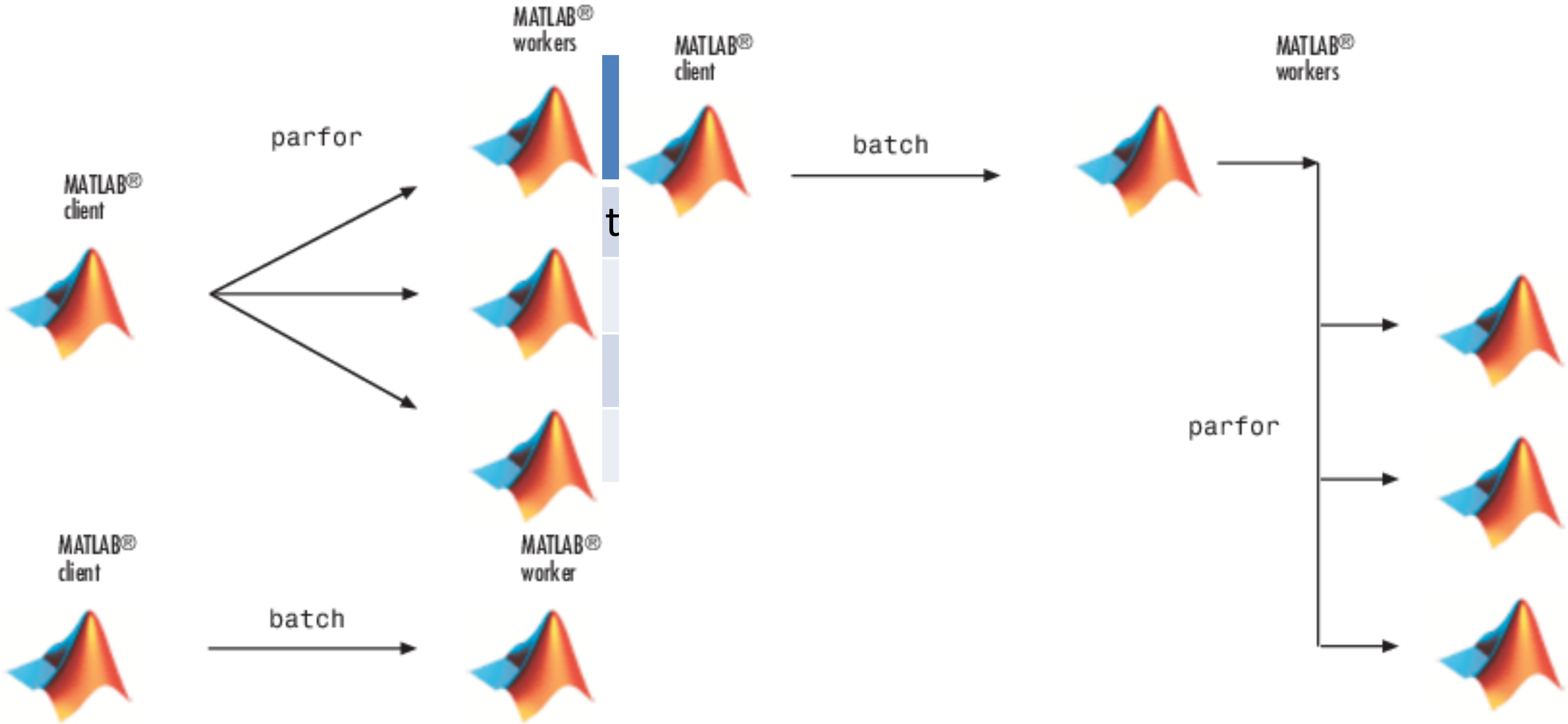
# Review – parallel MATLAB

- Three ways to write a CPU parallel MATLAB program:
  - suitable for loops can be made into parfor loops;
  - the spmd statement can define cooperating synchronized processing;
  - the task feature creates multiple independent programs.
- GPU Computing

- The parfor approach is a limited but simple way to get started.
- spmd is powerful, but may require rethinking the program/data.
- The task approach is simple, but suitable only for computations that need almost no communication.

# Review – parallel MATLAB

- There are several ways to execute a parallel MATLAB program:



# Review exercise

- Parallelize the following function and measure its execution time for n=10,000,000

```
function total = prime_fun ( n )
%% PRIME returns the number of primes between 1 and N.
    total = 0 ;
    for i = 2 : n
        prime = 1 ;
        for j = 2 : i-1
            if ( mod ( i , j ) == 0 )
            prime = 0 ;
            end
        end
        total = total + prime ;
    end
    return
end
```

Use the prime_fun.m function file

# Access Your Account

- UHVPN connection may be required if you are not on campus network
- Make sure that you are added to the classroom cluster access
- If you have confirmed enrollment then you should have access
- Ask the instructor if you have trouble

```
ssh -XY -l username aerb202.cacds.e.uh.edu
```

- Log into your accounts
- Username or login = Cougarnet ID
- **Password = Cougarnet password**

# Interactive Job on GPU node

ssh -XY -l username aerb202.cacds.e.uh.edu

srun -N 1 -n 8 --x11 --pty /bin/bash

# Interactive Job on GPU node

ssh -XY -l username aerb202.cacds.e.uh.edu

srun -N 1 -n 8 --x11 --pty /bin/bash

squeue -u $USER

# Accessing an Allocated GPU node

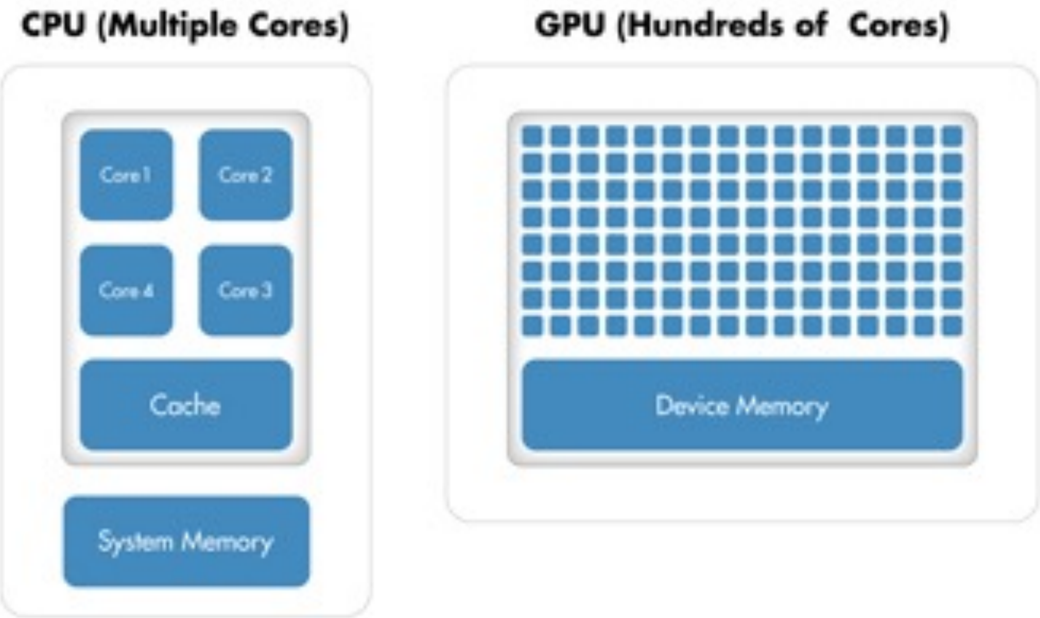
ssh -XY -l username aerb202.cacds.e.uh.edu

srun -N 1 -n 8 --x11 --pty /bin/bash

squeue -u $USER

ssh -XY aerb-202?

# GPU acceleration




- CPU
  - fast
  - general-purpose
- GPU
  - highly parallel
  - handles specific tasks with large amount of data
  - **memory transfers needed**

# What/Why GPU computing?

- Serial portions of the code run on the CPU while parallel portions run on the GPU
- From a user's perspective, applications in general run significantly faster

# Basic setup

- **CUDA-enabled NVIDIA GPUs** with compute capability 2.0 or higher. For releases 2014a and earlier, compute capability 1.3 is sufficient.
- Latest CUDA driver with 64 bit OS
- Your MATLAB must be version 2010b or later:
  - Go to the HELP menu, and choose About MATLAB.
- You must have the Parallel Computing Toolbox (PCT):
  - At UH, the concurrent (& student) license includes the PCT.
  - The standalone license does not include the PCT.
- To list all your toolboxes, type the MATLAB command **ver**.
- Limited availability on UH research cluster

# Application requirement

- **Will Execution on a GPU Accelerate My Application?**
- A GPU can accelerate an application if it fits both of the following criteria:
  - **Computationally intensive**—The time spent on computation significantly exceeds the time spent on transferring data to and from GPU memory.
  - **Massively parallel**—The computations can be broken down into hundreds or thousands of independent units of work.
- Applications that do not satisfy these criteria might actually run slower on a GPU than on a CPU.

# Basic usage

- Send data to GPU
  – either allocate there or transfer from workspace
- Run Matlab functions
  – GPU acceleration is used automatically
  – Built-in and custom functions
- Retrieve the output data

# GPU Capabilities and Performance

- Identify and Select a GPU Device
- Transfer or Create arrays on GPU
- Run built-in functions on GPU
- Run Element-wise MATLAB Code on GPU
- Run CUDA or PTX Code on GPU
- Run MEX-Functions Containing CUDA Code

# Identify and Select a GPU Device

- GPU functions
  - compute capability 2.0 and above

| | |
|---|---|
| gpuDevice | Query or select GPU device |
| gpuDeviceCount | Number of GPU devices present |
| gputimeit | Time required to run function on GPU |
| reset | Reset GPU device and clear its memory |
| wait (GPUDevice) | Wait for GPU calculation to complete |

```
>> gpuDeviceCount
>> gpuDevice
```

# Transfer Arrays Between Workspace and GPU

- Transfer Arrays Between Workspace and GPU
  - X = rand(100);
  - G = gpuArray(X);
- Create GPU Arrays Directly
  - G = rand(100,'gpuArray');
- Transfer data back to MATLAB workspace
  - C = gather(G);

# Try these examples:

Create on GPUs

```
>> identity = eye(1024,'int32','gpuArray');
>> Z = zeros(8192,1,'gpuArray');
```

Send to GPUs

```
>> N = 6;
>> M = magic(N);
>> G = gpuArray(M);
>> G = gpuArray(single(X));
>> G = gpuArray(ones(100,'uint32'));
```

Retrieve from GPUs

```
>> G = gpuArray(ones(100,'uint32')); %Array stored on GPU
>> D = gather(G); % Get G from GPU to D in MATLAB workspace
>> OK = isequal(D,ones(100,'uint32')) % check if G on GPU is same as D
on CPU
```

# Examine gpuArray Characteristics

| Function | Description |
|---|---|
| `classUnderlying` | Class of the underlying data in the array |
| `existsOnGPU` | Indication if array exists on the GPU and is accessible |
| `isreal` | Indication if array data is real |
| `length` | Length of vector or largest array dimension |
| `ndims` | Number of dimensions in the array |
| `size` | Size of array dimensions |

# GPUArray class

`gpuArray`

- main data class for GPU computations
- stored in the GPU memory
- create directly using static methods

| | | | | |
|---|---|---|---|---|
| `zeros` | `nan` | `eye` | `rand` | `linspace` |
| `ones` | `true` | `colon` | `randi` | `logspace` |
| `inf` | `false` | | `randn` | |

- copy from existing data

```
gpuArray(img)
```

# GPUArray class

- Supported data types:

  (u)int8, (u)int16, (u)int32, (u)int64, single, double, logical

  - determine the type using

    ```
    classUnderlying(gpuVar)
    ```

- Retrieve the data using

  ```
  workspaceVar = gather(gpuVar)
  ```

# Run Built-In Functions on a GPU

- If at least one of the input arguments is a gpuArray, the function executes on the GPU and returns a gpuArray
- Common functions on GPUs - discrete Fourier transform (fft), matrix multiplication (mtimes), left matrix division (mldivide)
- Full list - `methods('gpuArray')` or http://www.mathworks.com/help/distcomp/run-built-in-functions-on-a-gpu.html
- help gpuArray/*functionname*
- Explicit definition for Complex numbers if output is expected to be complex
  - G = gpuArray(complex(p))

# Functions with gpuArray

Compare the execution time of the fft function on GPU vs CPU

gfun.m

```
Ga = rand(1000,'single','gpuArray');
Gfft = fft(Ga);
Gb = (real(Gfft) + Ga) * 6;
G = gather(Gb);
whos
```

cfun.m

```
Ga = rand(1000,'single');
Gfft = fft(Ga);
Gb = (real(Gfft) + Ga) * 6;
whos
```

## Sparse matrices on GPUs

sparsefun.m

```
x = [0 1 0 0 0; 0 0 0 0 1]
s = sparse(x)
g = gpuArray(s);   % g is a sparse gpuArray
gt = transpose(g); % gt is a sparse gpuArray
f = full(gt)
whos
```

# Simple example

- Solve system of linear equations (Ax = b)

```
A = gpuArray(A);
b = gpuArray(b);
x = A\b;
x = gather(x);
```

# Simple example

- Compute convolution using FFT (convolution.m)

```
img = gpuArray(img);
msk = padarray(msk,size(img)-size(msk),0,'post');
msk = gpuArray(msk);
I = fft2(img);
M = fft2(msk,size(img,1),size(img,2));
res = real(ifft2(I.*M));
res = gather(res);
```

# Run Element-wise MATLAB Code on GPU

- You can run your own MATLAB function of element-wise operations on a GPU
  - Define arrays on GPU
  - Use arrayfun or bsxfun to execute the custom functions
- List of supported MATLAB
  - http://www.mathworks.co n-
    element-wise-matlab-code-
    gpu.html#bsnx7h8-1

```
function Y = myfun(X)
    R = rand();
    Y = R.*X;
end
G = 2*ones(4,4,'gpuArray')
H = arrayfun(@myfun, G)
```

# Random numbers on GPUs

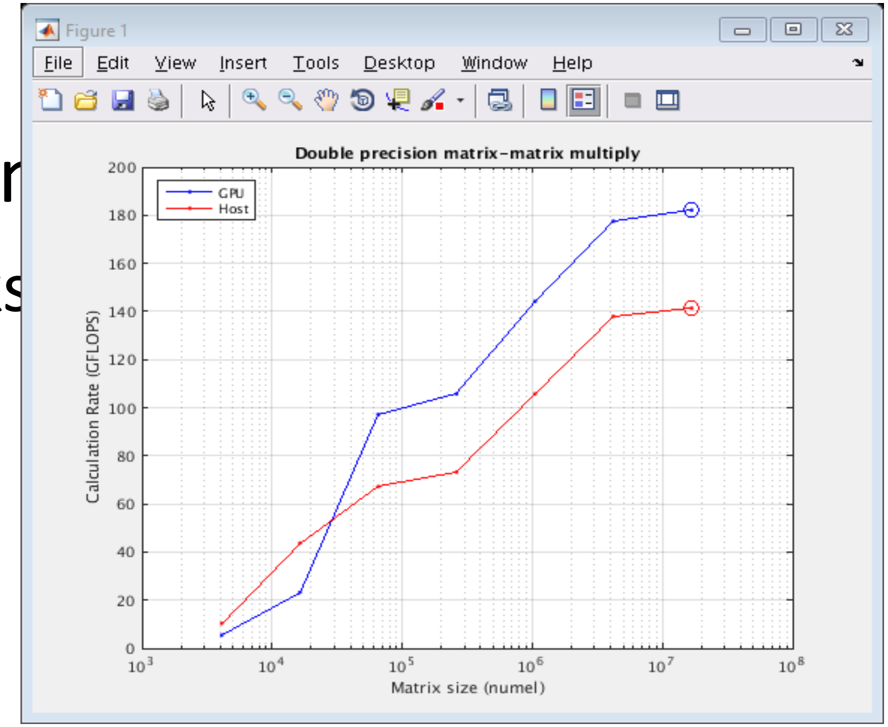
- rand, randi, and randn partially available on GPUs
- Default random stream on CPU and GPU are not same
  - parallel.gpu.rng
  - parallel.gpu.RandStream
- To set the streams equal
  - Set the same stream and seed
  - Use the random_equal.m

# MEX-Functions with CUDA Code

- Write a MEX-File Containing CUDA Code
- Compile a GPU MEX-File
- Run the Resulting MEX-Functions

# Benchmarking

- Run the MATLAB benchmark for GPU
  - Use script gpu_benchmark.m
- Run the MATLAB ber
  - Use script gpu_backs



# Exercise - The Mandelbrot Set

- Using CPU only
  - mandelbrot_cpu.m
- Using the existing algorithm but with GPU data as input
  - mandelbrot_gpuArray.m
- Using arrayfun to perform the algorithm on each element independently
  - mandelbrot_arrayfun.m
- *Using the MATLAB/CUDA interface to run some existing CUDA/C++ code*

# GPU Code porting

- Profile the existing code
  - Identify the hotspots using MTLAB profile tool
    - Custom functions
    - Parfor loops
    - Custom MEX functions
    - Built-in functions without corresponding gpuArray version
    - High frequency of function calls for many functions
    - Short execution time
  - Use the MATLAB debugger to get memory requirements
    - Large input/output data
    - Simple data types (arrays and matrices)

# Code porting strategies

- Use the built-in gupArray functions
  - Check the results for correctness
  - Speedup might vary
    - Faster than CPU – good
    - Slower than CPU – not bad!
      - At least it works and other optimizations might improve performance
- Matrix manipulations tend to be faster on GPUs
  - Move operations like reshape, subsref, etc…

# Code porting strategies

- Eliminate loops/calls in the code
  - Vectorize the code

```
A = rand(4);
output = zeros(4);
for n=1:4
   output(n,:) = input(n,:) / n;
end
```

```
A=rand(4);
output=zeros(4);
output = A ./ repmat([1:4].',[1,4]);
```

  - Minimize the calls to gpuArray functions to avoid data transfers

# Code porting strategies

- Wisely use the limited GPU memory
  - Keep reuse data on GPUs

```
??? Error using ==> gpuArray at 37
Out of memory on device. You requested: 651.73Mb, device has 1.27Gb free.
```

  - use “clear” to flush unnecessary variables from GPU memory *or*
  - divide your problem into smaller chunks
  - Use host memory as buffer for data transfer

# Code porting strategies

- Recast for arrayfun
  - element-wise operations
  - operations across a large number of scalar values
  - handle bulk processing of small arrays

```
R1 = rand(2,5,4,'gpuArray');
R2 = rand(2,1,4,3,'gpuArray');
R3 = rand(1,5,4,3,'gpuArray');
R = arrayfun(@(x,y,z)(x+y.*z),R1,R2,R3);
size(R)
```

  - Combine vectorized statements

```
>> result = (a1 + a2 + a3) ./ 3;
```

```
function out=littleaaafun(a1,a2,a3)
out = (a1+a2+a3) / 3;
```

# GPU arrayfun example

- Define a MATLAB function

```
function [o1,o2] = aGpuFunction(a,b,c)
o1 = a + b;
o2 = o1 .* c + 2;
```

- Evaluate on GPU

```
s1 = gpuArray(rand(400));
s2 = gpuArray(rand(400));
s3 = gpuArray(rand(400));
[o1,o2] = arrayfun(@aGpuFunction,s1,s2,s3);
whos
```

- Retrieve the data to MATLAB workspace on CPU

```
d = gather(o2);
```

# Code porting strategies

- Create a CUDA kernel
  - big speed-up and a big headache
  - Simple functions that are task independent
  - translate an existing MEX function that is very simple to run on the GPU
  - complex code that include a lot of branching, task dependencies, and/or serialized output

# Code porting strategies for multiple GPUs

- Use GPUs from a parpool on CPUs
  - GPU function call from a parfor could create conflicts in worker processes so GPU computations in parfor are serialized across workers.
  - Use spmd to pin a GPU per worker process

```
parpool('local',4)
spmd
  gpuDevice( labindex );
  % customer GPU code goes here
end
delete(gcp)
```

  - Alternatively, toggle between CPU and GPU
    - See code in testgpu.m

# Code porting strategies

- Test code changes
  - Use ‘whos’ to see if GPUs are actually being used
  - Due to changes in handling of arithmetic on CPU and GPU a small amount of variance in results (difference of O(1e-7) percent may be reasonable)
- Find and fix new bottlenecks
  - Incrementally add GPU parallelism
  - Scale up the input data size to find other issues
  - Iteratively fix the bottlenecks
  - Rethink the algorithm if necessary

# References


- http://www.mathworks.com/discovery/matlab-gpu.html
- http://www.mathworks.com/company/newsletters/articles/gpu-programming-in-matlab.html
- http://its2.unc.edu/divisions/rc/training/scientific/short_courses/ParallelGPUMATLAB.pptx
- http://zoi.utia.cas.cz/files/GPU%20acceleration%20in%20Matlab.pptx
- http://www.cac.cornell.edu/matlab/TechDocs/Examples/BestPracticesGPU.aspx
- http://www.mathworks.com/moler/exm/book.pdf