Parallel Processing Techniques with MATLAB[1]

Sayed Mohammad Badiezadegan

MATHLAB Co. Funder, M. S. Cryptography (Applied Mathematics)
Email: *ceo[at]dataman.ca*

Abstract: ***Assuredly, the competency of MATLAB software in scientific and engineering research could not be ignored in this era. With its powerful modules, this software can perform a lot of modern industrial or even medical research efficiently. The biggest stunning point of this outstanding software is that it can process big data even without using supercomputers by taking advantage of multiple and successful parallel processing techniques. The following article will discuss the introduction of MATLAB parallel processing techniques. Afterward, some methods to achieve the best parallel processing performance using the graphics processing unit (GPU) are explained. Finally, the comparison between CPU parallel processing and GPU parallel processing will be discussed.***

Keywords: MATLAB, Parallel processing, Multicore processing, GPU processing, Big data processing

1. Introduction

Data processing in information - based systems can be done in two ways:

- Serial (Sequential)
- Parallel

In serial data processing, the CPU processes the information at each step and the result of each processing step enters the next processing step. This means that the second stage of a process waits until the result of the first stage is determined. This procedure continues and the nth step of a process is performed when the n - 1st step is completed.

For example, in processing the information of a network packet, the information of network layers from one to seven should be processed consecutively.

In parallel data processing, the CPU processor can start and finish several information processing steps together. This makes processing much faster than serial - only mode.

For instance, in the process of multiplying matrices, all the elements of the matrix C can be determined independently, For example, it is not necessary to know the C_{11} element to find the C_{12} matrix element.

$$[A_{ij}] \times [B_{jk}] = [C_{ik}]$$

Therefore, Matrix multiplication could be a suitable starting stage for discussing parallel processing with MATLAB.

Parallel processing should be discussed with CPU logical cores and compared with a Graphical Processing Unit (GPU). GPU cards are used for parallel processing in more programming languages such as Julia[2] and also CUDA[3] programming.

Now, it's time to start the parallel pool processing in MATLAB by pressing the left bottom button as shown in Figure 1 and Figure 2:

[1]MATLAB is a proprietary multi-paradigm programming language and numeric computing environment developed by MathWorks.
[2] Julia is a high-level, general-purpose dynamic programming language, most commonly used for numerical analysis and computational science. https://julialang.org/
[3] Compute Unified Device Architecture (CUDA) is a proprietary parallel computing platform and application programming interface (API) that allows software to use certain types of graphics processing units (GPUs) for accelerated general-purpose processing.

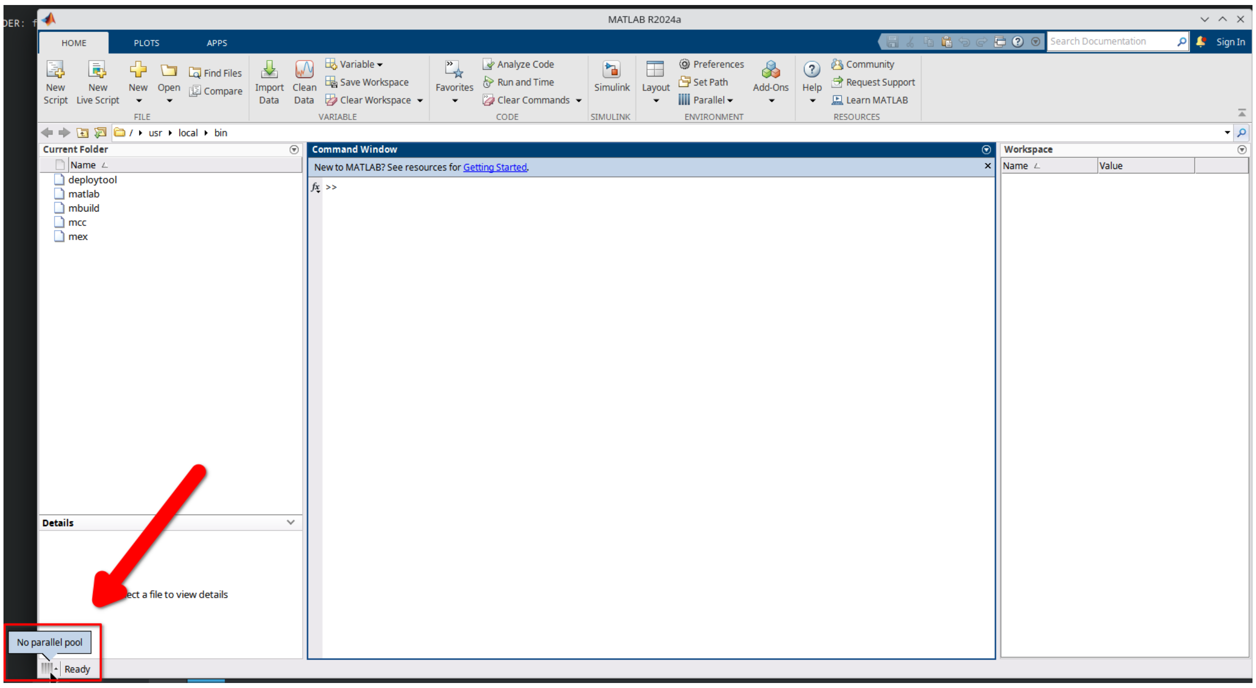


Figure 1: Start working with parallel pool

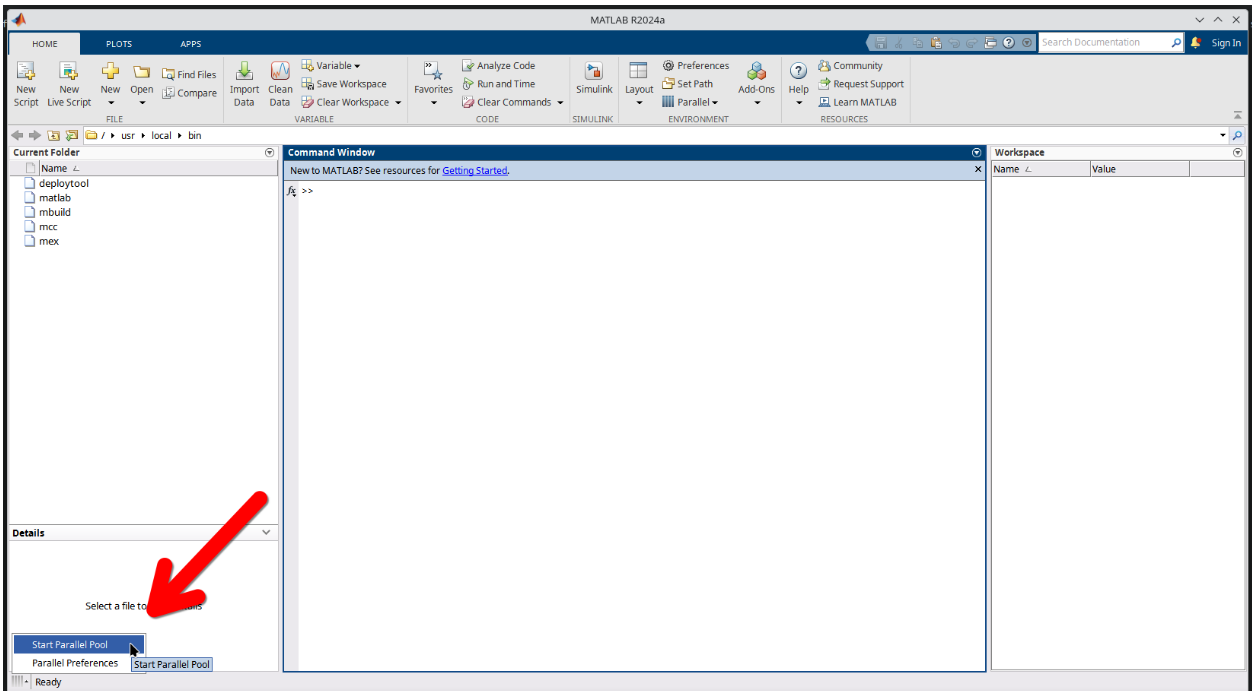
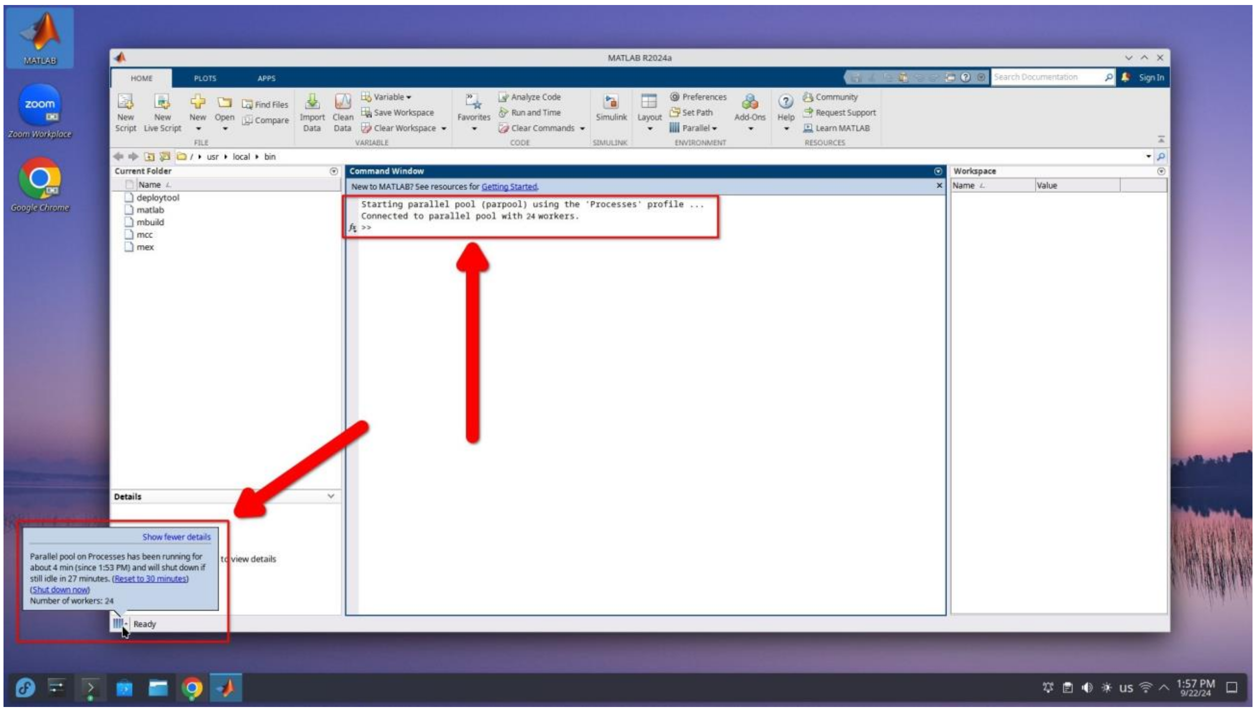


Figure 2: Start working with parallel pool

The parallel processing pool will start after some seconds with the following messages:

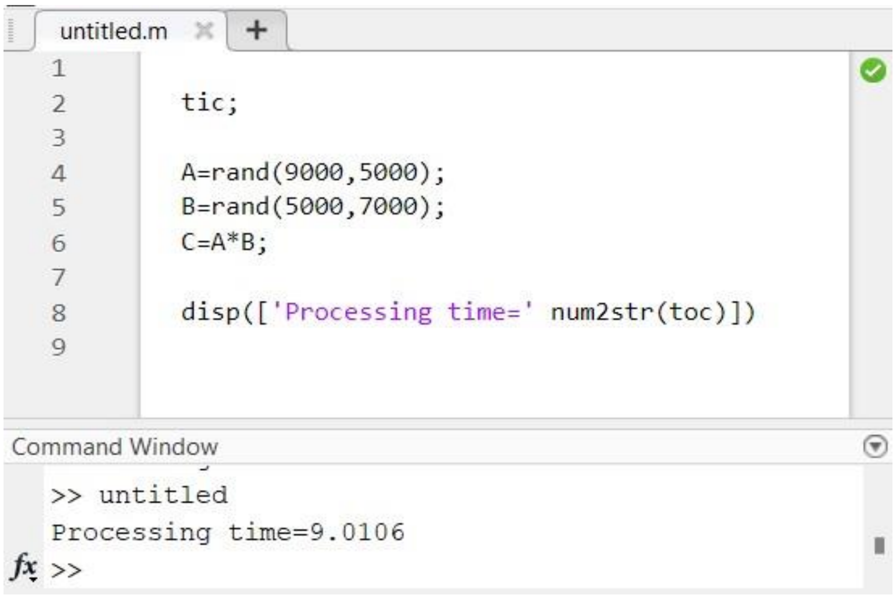


```
Starting parallel pool (parpool) using the 'Processes' profile ...
Connected to parallel pool with 24 workers.
>>
```

Parallel pool on Processes has been running for about 4 min (since 1:53 PM) and will shut down if still idle in 27 minutes. (Reset to 30 minutes)
(Shut down now)
Number of workers: 24

Figure 3: Start parallel pool processing after a while

Parallel processing can be recognized better by measuring the processing time. To measure the processing time in MATLAB, using the **tic** & **toc** commands with the following structure is recommended:

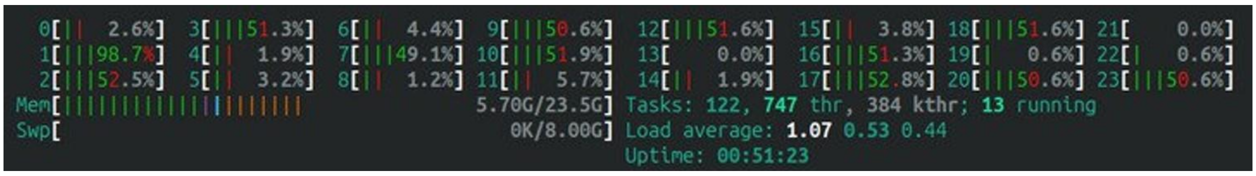


```
tic;

A=rand(9000,5000);
B=rand(5000,7000);
C=A*B;

disp(['Processing time=' num2str(toc)])
```

```
>> untitled
Processing time=9.0106
>>
```

Figure 4: Matrices multiplication processing time

CPU usage can also be monitored while MATLAB is processing. The **htop** command in Linux Operating systems is useful for this measurement. For example, It shows the following information about CPU core usage when MATLAB is processing:



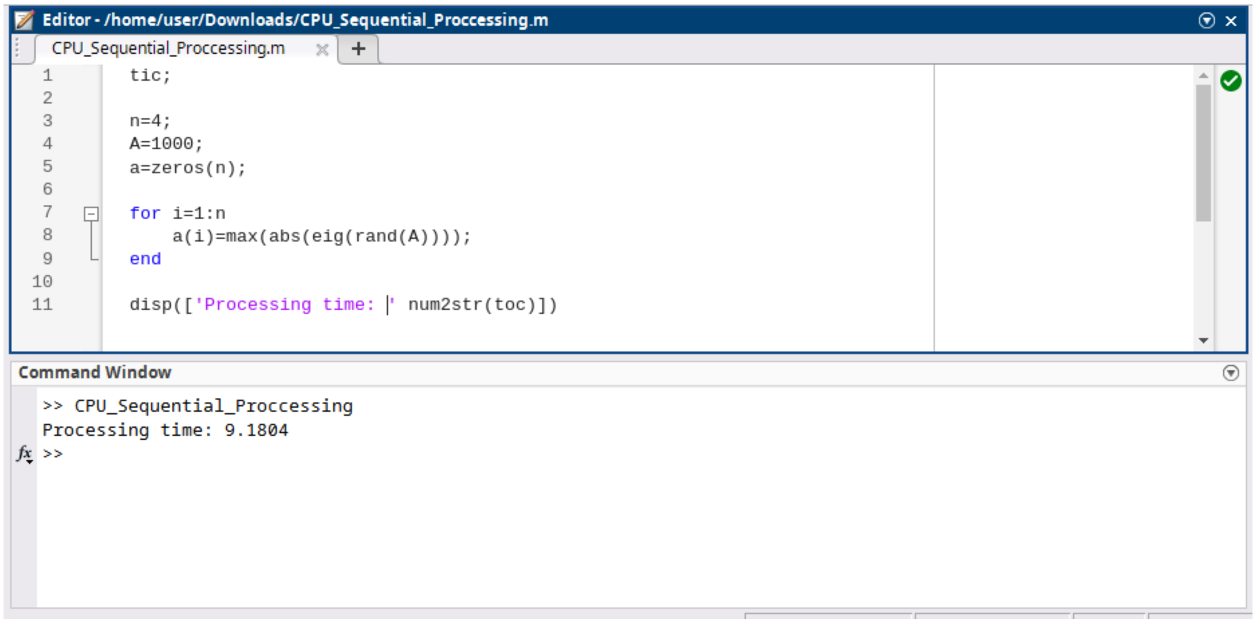
```
  0[||   2.6%]  3[|||51.3%]  6[||   4.4%]  9[|||50.6%]  12[|||51.6%]  15[||   3.8%] 18[|||51.6%] 21[     0.0%]
  1[|||98.7%]  4[||   1.9%]  7[|||49.1%] 10[|||51.9%]  13[     0.0%]  16[|||51.3%] 19[|    0.6%] 22[|    0.6%]
  2[|||52.5%]  5[||   3.2%]  8[||   1.2%] 11[||   5.7%]  14[||   1.9%]  17[|||52.8%] 20[|||50.6%] 23[|||50.6%]
Mem[||||||||||||||||||||||                5.70G/23.5G] Tasks: 122, 747 thr, 384 kthr; 13 running
Swp[                                        0K/8.00G] Load average: 1.07 0.53 0.44
                                                      Uptime: 00:51:23
```

Figure 5: CPU processing measurement by Linux's htop application

Figure 5 shows that the matrices multiplication process is assigned to some CPU cores while others are idle. Also, there is no control over which processor runs the process.

The above commands were executed on a relatively powerful computer with an Intel Xeon CPU, which has 24 logical cores and is relatively expensive. Now, the following commands are executed on a cheaper computer with an AMD CPU including 4 logical cores and then the processing result is compared with when the graphics card is used on it.

The following MATLAB script code is best for starting work with CPU sequential processing as described in the MATLAB Parallel Computing Toolbox User's Guide[4].


Editor - /home/user/Downloads/CPU_Sequential_Proccessing.m

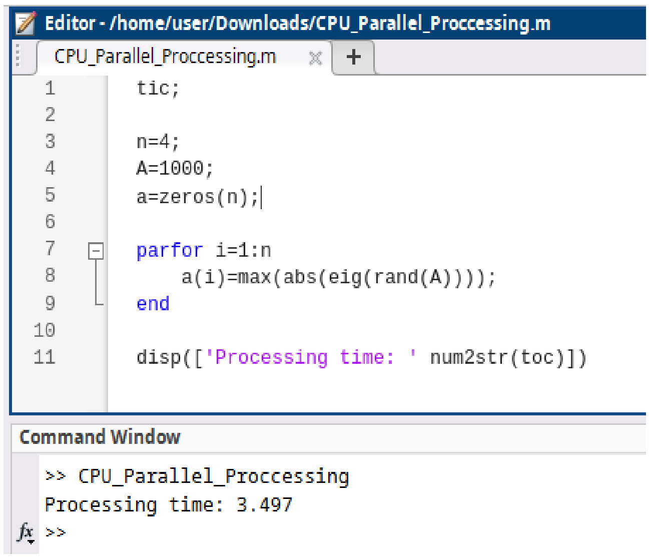
```
tic;

n=4;
A=1000;
a=zeros(n);

for i=1:n
    a(i)=max(abs(eig(rand(A))));
end

disp(['Processing time: ' num2str(toc)])
```

Command Window

```
>> CPU_Sequential_Proccessing
Processing time: 9.1804
>>
```

Figure 6: CPU Processing time (Logical cores running sequentially)

The CPU sequential processing time is about 9 sec as Figure 6 is shown. But, what is the CPU processing time if its logical cores are processing parallel instead of sequential? To answer this question, the **for** - loop on the script should be replaced with the **parfor** - loop as shown below:


Editor - /home/user/Downloads/CPU_Parallel_Proccessing.m

```
tic;

n=4;
A=1000;
a=zeros(n);

parfor i=1:n
    a(i)=max(abs(eig(rand(A))));
end

disp(['Processing time: ' num2str(toc)])
```

Command Window

```
>> CPU_Parallel_Proccessing
Processing time: 3.497
>>
```

Figure 7: CPU Processing time (Logical cores running parallel)

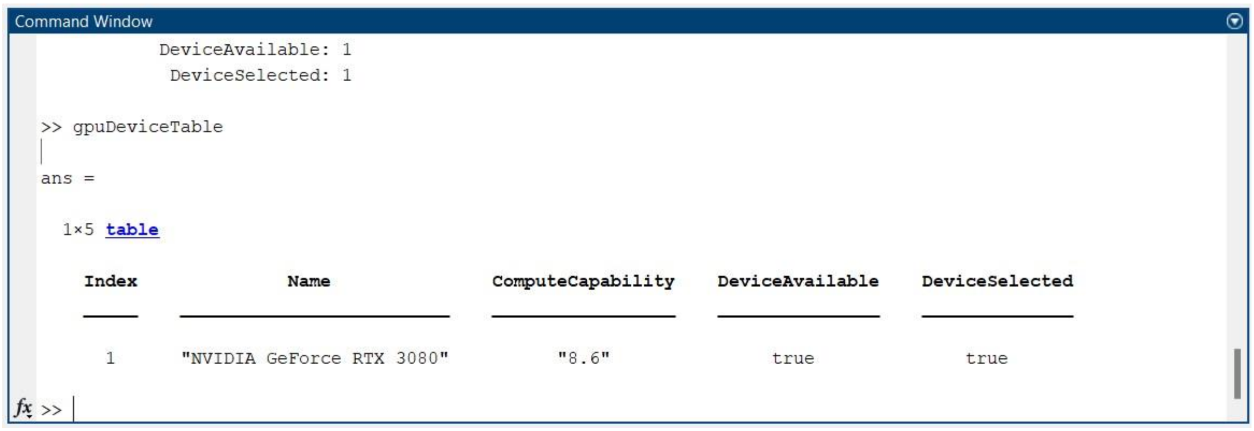
The CPU processing time is decreased seriously by around %38.

Now, It's time to use a GPU card for better parallel processing. For instance, it is essential to make sure whether MATLAB supports the current GPU card or not. The MATLAB Parallel Computing Toolbox User's Guide[5] on pages 6 - 5 explains how to find an appropriate GPU card that is more efficient for MATLAB parallel processing. Also, the GPU cards should be installed in the OS by its driver in advance.

There is a command in MATLAB that can monitor the status of the current connected GPU card which is **gpuDeviceTable** as follows:

4 https://www.mathworks.com/help/pdf_doc/parallel-computing/parallel-computing.pdf page 2-3

5 https://www.mathworks.com/help/pdf_doc/parallel-computing/parallel-computing.pdf

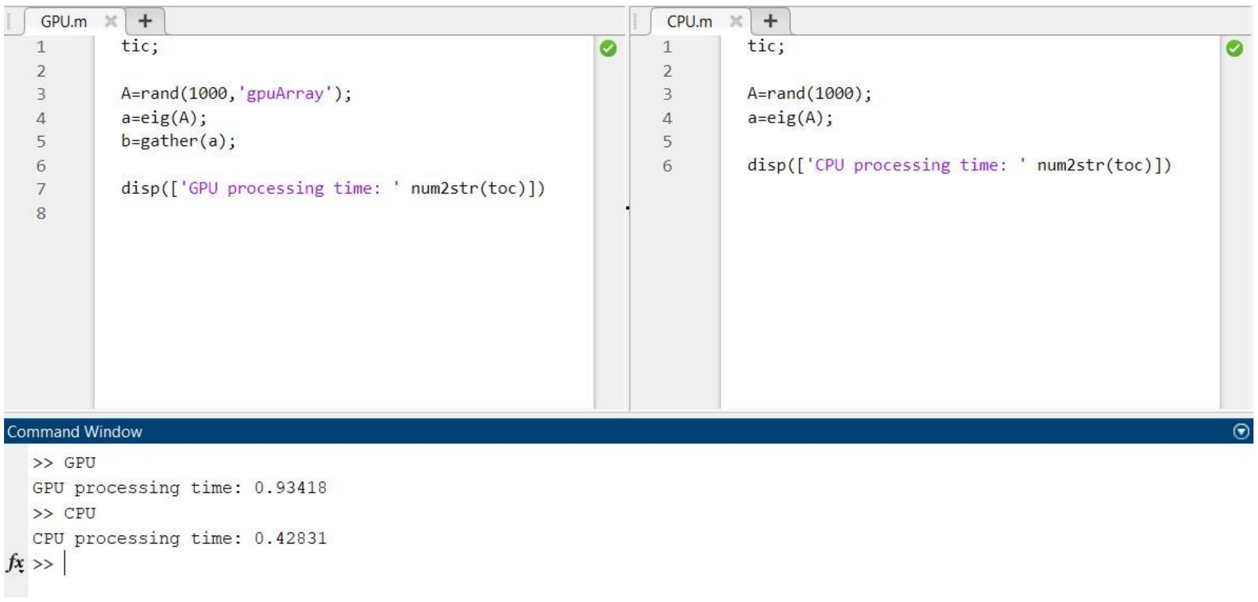


```
Command Window
            DeviceAvailable: 1
             DeviceSelected: 1

>> gpuDeviceTable

ans =

  1×5 table

    Index              Name              ComputeCapability    DeviceAvailable    DeviceSelected
    _____    ________________________    _________________    _______________    ______________

      1      "NVIDIA GeForce RTX 3080"         "8.6"               true               true

fx >>
```

Figure 8: Running the gpuDeviceTable command

There are some commands used to explain the advantages of using a GPU card:

First, all of the processing variables should be called by the **'gpuArray'** parameter to send from memory to the GPU card. Likewise, all GPU processing results should be returned to memory by the **gather** command as follows:



GPU.m

```
tic;

A=rand(1000,'gpuArray');
a=eig(A);
b=gather(a);

disp(['GPU processing time: ' num2str(toc)])
```

CPU.m

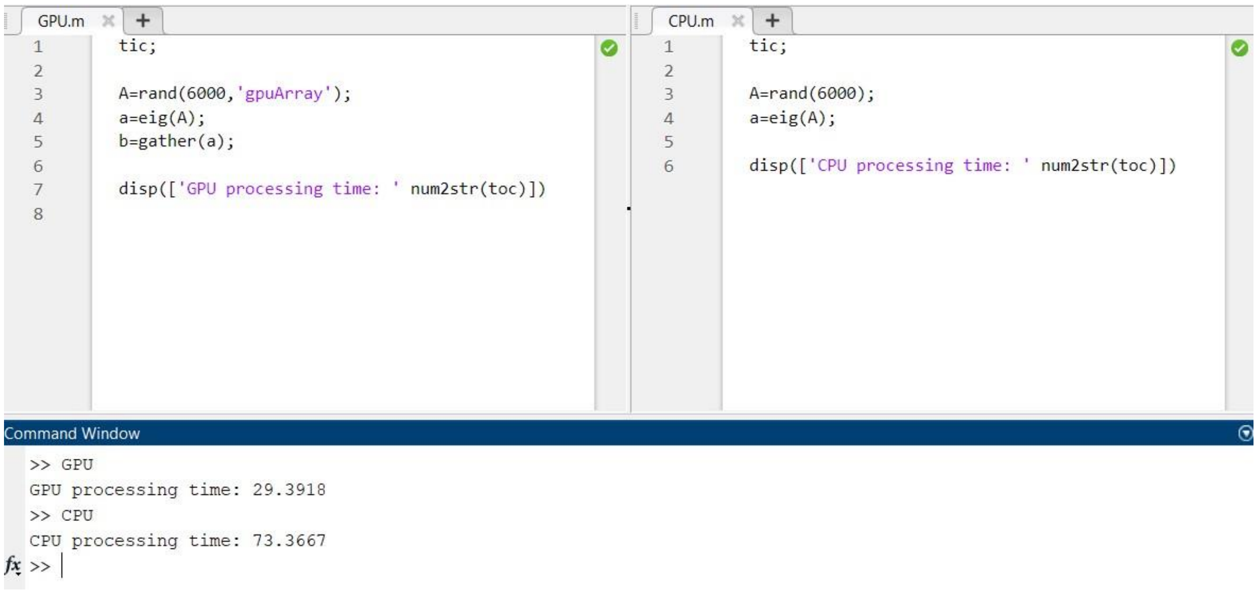
```
tic;

A=rand(1000);
a=eig(A);

disp(['CPU processing time: ' num2str(toc)])
```

```
Command Window
>> GPU
GPU processing time: 0.93418
>> CPU
CPU processing time: 0.42831
fx >>
```

Figure 9: GPU vs CPU processing time for 1000x1000=10^6 numbers

Figure 9 shows that the CPU processing time for 10^6 numbers is around half the amount of GPU processing time. So, using a CPU is more efficient for less data.

But when the amount of data increases, the superiority of GPU becomes evident. Figure 10 shows that the GPU processing time is less than half of the CPU processing time when data increases by 36 times. (6, 000x6, 000=36, 000, 000)



```
tic;

A=rand(6000,'gpuArray');
a=eig(A);
b=gather(a);

disp(['GPU processing time: ' num2str(toc)])
```

```
tic;

A=rand(6000);
a=eig(A);

disp(['CPU processing time: ' num2str(toc)])
```

```
>> GPU
GPU processing time: 29.3918
>> CPU
CPU processing time: 73.3667
>> 
```

Figure 10: GPU vs CPU processing time for 6000x6000=36x10^6 numbers

2. Conclusion

1) Parallel processing could be used instead of sequential processing when data processing is completely independent. For example, finding a C_{11} element is completely independent of the finding of C_{12} in Matrix multiplication.
2) In the case of using parallel processing, GPU processing data is more efficient than CPU processing only for large data processing.