Step by step installation

This is a detailed guide on how to use the matrixMulMex.m function. These steps are required in order to successfully compile the function. They should be executed in the order written.

# System requirments

* At least 1 Nvidia Graphics card.
* Preferably a CPU with at least 2 cores.
* Windows 7 (Although newer versions should work)
* MATLAB R2014a/b or newer
* cuBLAS-XT premier license for more than 2 GPUs

# Installation steps

## Install MATALB

The project was tested on MATALB R2014a/b. So these versions should defiantly work. Newer versions should work just fine. MATLAB requires a license.

<https://www.mathworks.com/downloads/>

## Install VS 2010/2013

To compile the function successfully, Visual studio 2010/2013 must be installed. GPU integration works with both version whereas FPGA integration **only works with Visual Studio 2013 due to the driver provided to us, other FPGA drivers may not need this.**  A different .xml file is required for every version, more on that later. Download is available at <https://www.visualstudio.com/en-us/downloads/download-visual-studio-vs.aspx> and requires a license.

## Install CUDA

To successfully install and use the function, CUDA is required.

* Go to <https://developer.nvidia.com/cuda-downloads> or <https://developer.nvidia.com/cuda-toolkit-70>
* Select x64 operation system (We have tested on windows 7).
* Run the installation
* The system will check if an Nvidia graphics card is present. (The installation can work without the a graphics card but this is not the idea here).
* Click next and install CUDA in the default installation directory.

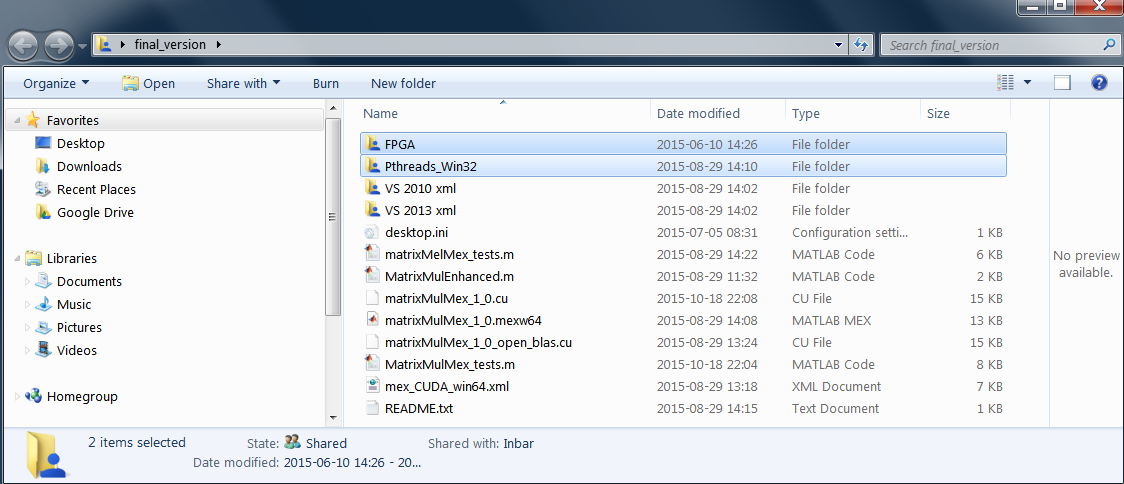
### Installing cuBLAS-XT premier license

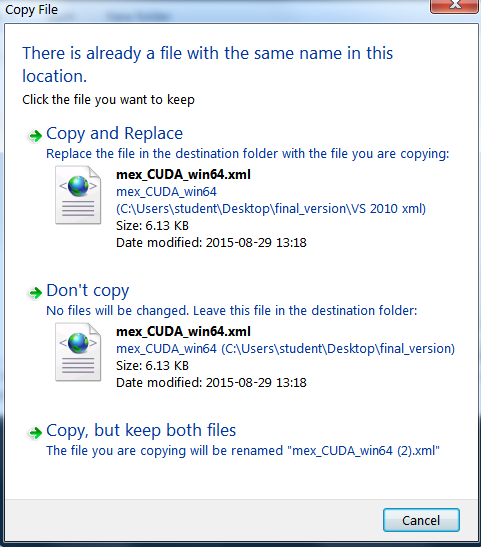
The license is required to perform calculations on more than 2 GPUs. Nvidia provides the license for free for development use.

* Sign up for CUDA developer program at:  
  <https://developer.nvidia.com/cuda-registered-developer-program>
* Go to developer area
* Download cuBLAS-XT Premier Evaluation Version
* Follow license agreement and installation process

## Preparing the function

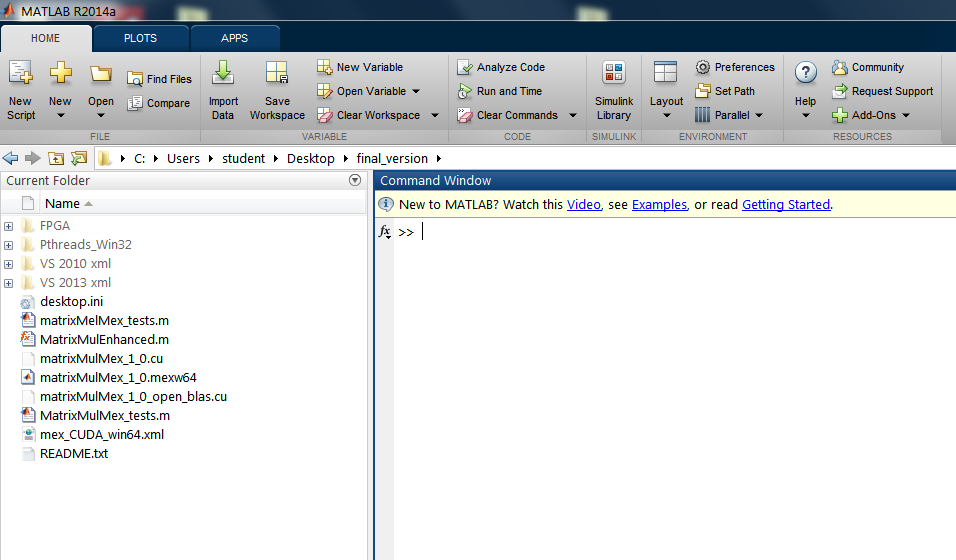
### Unzip the provided file to a local directory on your pc

* The open directory should look like this: 
* Copy the provided .xml file out of your corresponding visual studio version to the main directory i.e if you are using visual studio 2010, copy the file out of VS 2010 xml to the mail directory. Replace is needed.



### MATLAB setup.

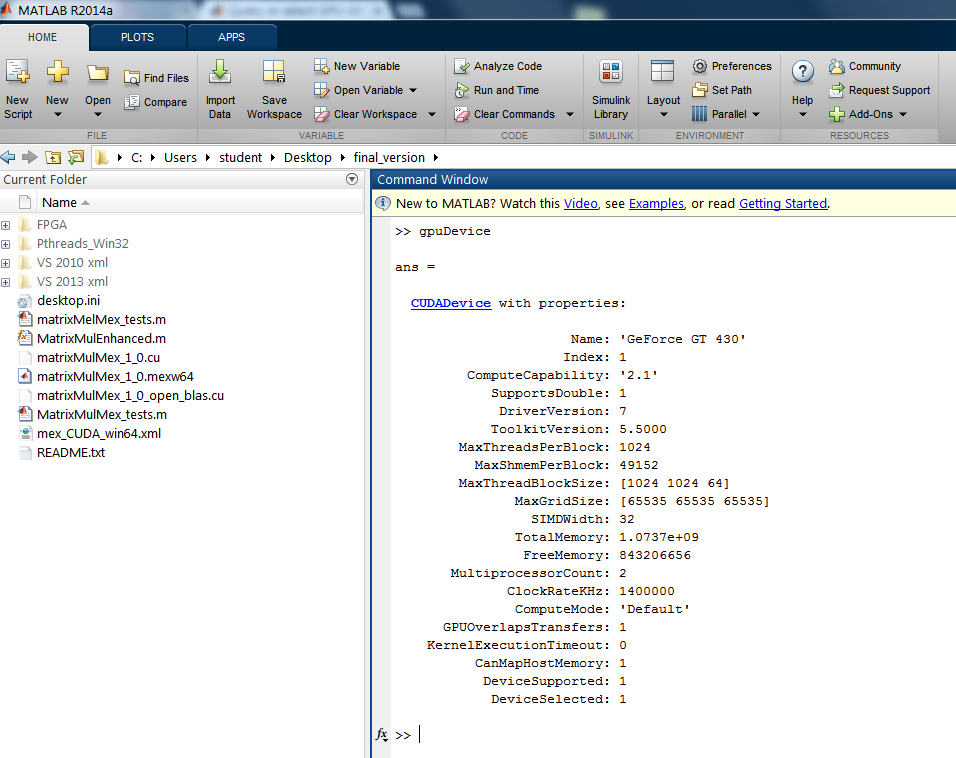
* Open MATLAB and change the directory to the local main folder you have unzipped.



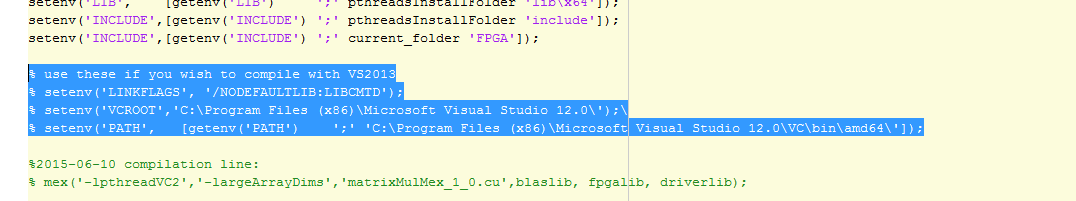
In this example the directory was unzipped on the desktop.

* Run >gpuDevice

Do this to make sure you have a detected GPU in your system.

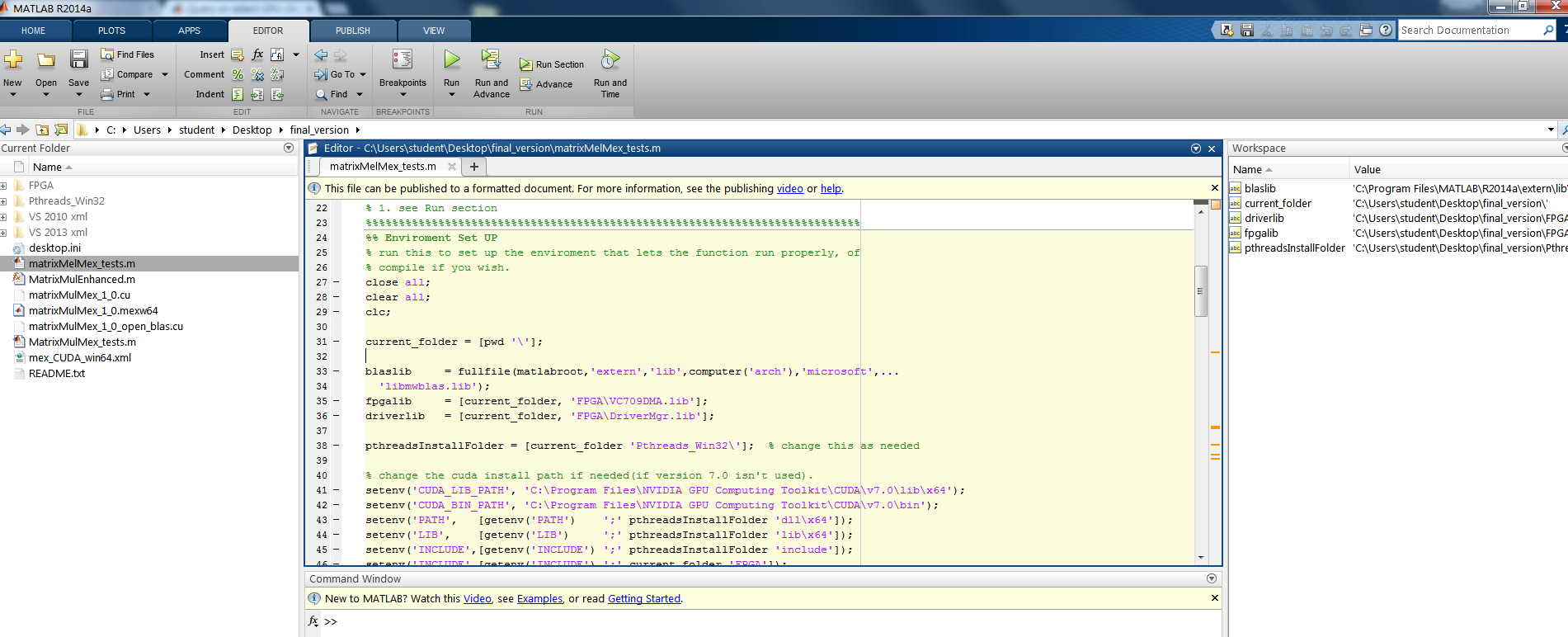


* Read the comments in the beginning, they contain the same information as this this tutorial.
* Select the environment setup “chunk” by clicking anywhere in the code (the chunk will be heighted when selected).
* If you are compiling with Visual Studio 2013, uncomment the commented lines:



and check that the information in them is correct, i.e. the visual studio installation directory.

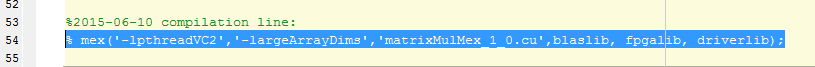
* Run the code by using Ctrl-Enter. (Note that CUDA installation directory is currently the default installation directory. You must update it in case you have installed CUDA in a different location.



* You are now ready to use and compile the function.

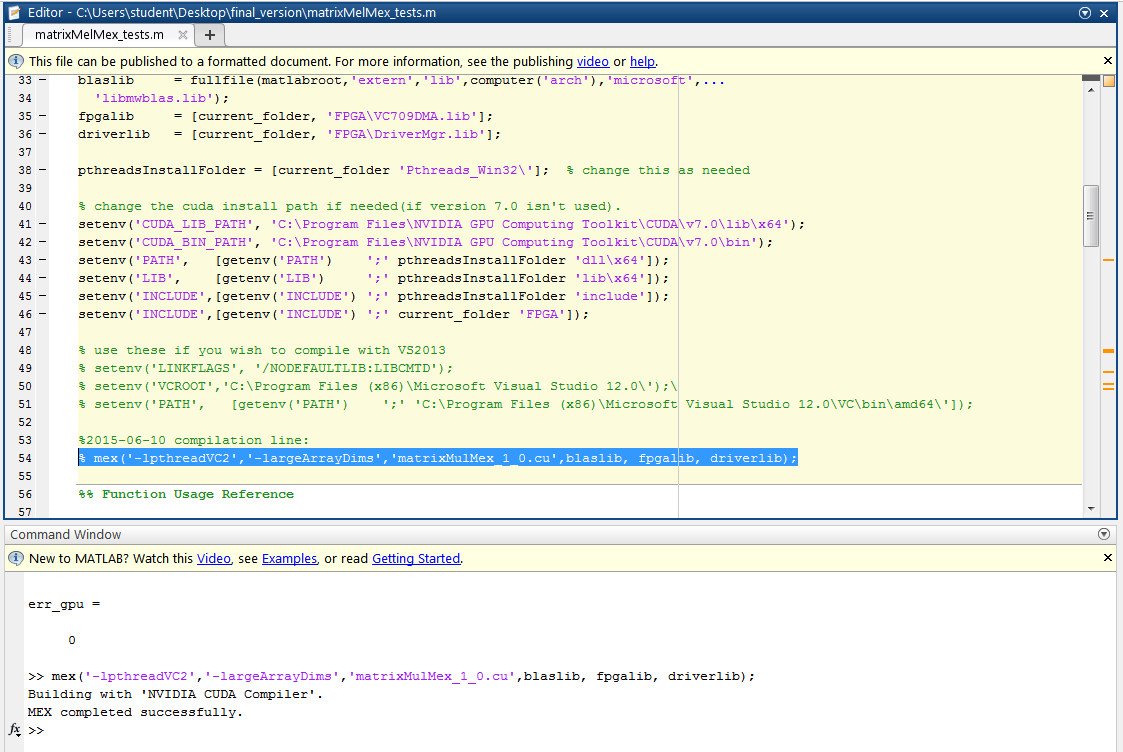
### Compile the function

* Compile the function by running the quoted out compilation line in the end of the system setup script.



mex('-lpthreadVC2','-largeArrayDims','matrixMulMex\_1\_0.cu',blaslib, fpgalib, driverlib);

If all steps where completed successfully, MATLAB should compile the function and return a successful compilation message



# Use case scenarios

## The first script %% Function Usage Reference

This script gives an example of how to use the function.

Say we want to calculate the result of and store it in .

For this example:

* A, B and c\_gpu are random floating point matrices .
* The matrix sizes used are for both matrices A and B.
* Only 1 thread is used because we have now FPGA’s connected. We can add a thread for every available FPGA. E.g if 1 FPGA is connected, use 2 threads.

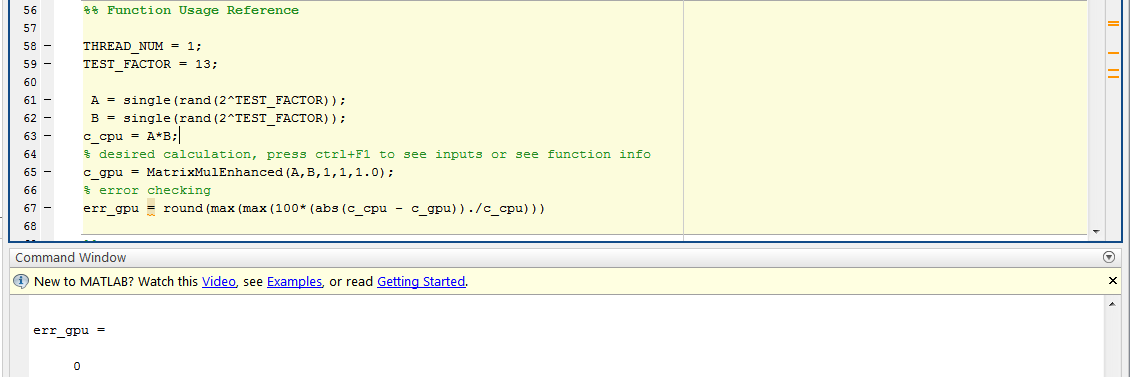
(Currently any reasonable number of threads could be selected and all will be calculated on the CPU).

* 1 Nvidia GPU is connected so we choose 1 GPU.
* We want 20% of the workload to be calculated on the CPU.
* We use the function in the following format

c\_gpu = MatrixMulEnhanced(A,B,1,1,0.2);

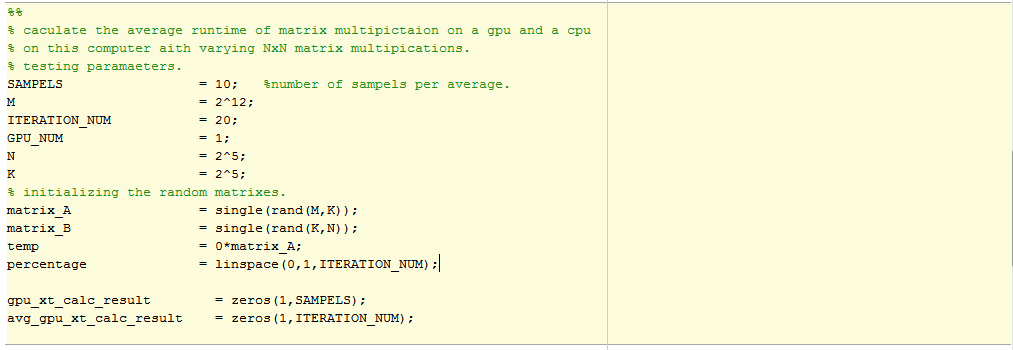
* You can also use Ctrl-Enter to run the selected code.
* Try experimenting with different sizes and ratios.
* Use F1 while determining the parameters to get MATALB help for the function.

The result is stored in c\_gpu. The result is also stored for reference in c\_cpu. The latter is calculated by MATLAB on the CPU only. The last line calculates the **maximum** error between the GPU and CPU calculations. An error of 0 means the results are exactly the same!



## Assign parameters.

The next part of the script allows to determine the parameters for the tests. The purpose of the test is to find the optimal operation point for the CPU and GPU rations. The operation point will vary depending on matrix size and available hardware.



The available parameters are:

SAMPELS: The average calculation time is more accurate with a high number

ITERATION\_NUM: How many fractions will be tested

GPU\_NUM: Number of GPU’s to test

M: Matrix dimensions

K = 2^5;

N = 2^5;

% initializing the random matrixes.

matrix\_A = single(rand(M,K));

matrix\_B = single(rand(K,N));

temp = 0\*matrix\_A;

percentage = linspace(0,1,ITERATION\_NUM):

gpu\_xt\_calc\_result = zeros(1,SAMPELS): results vector

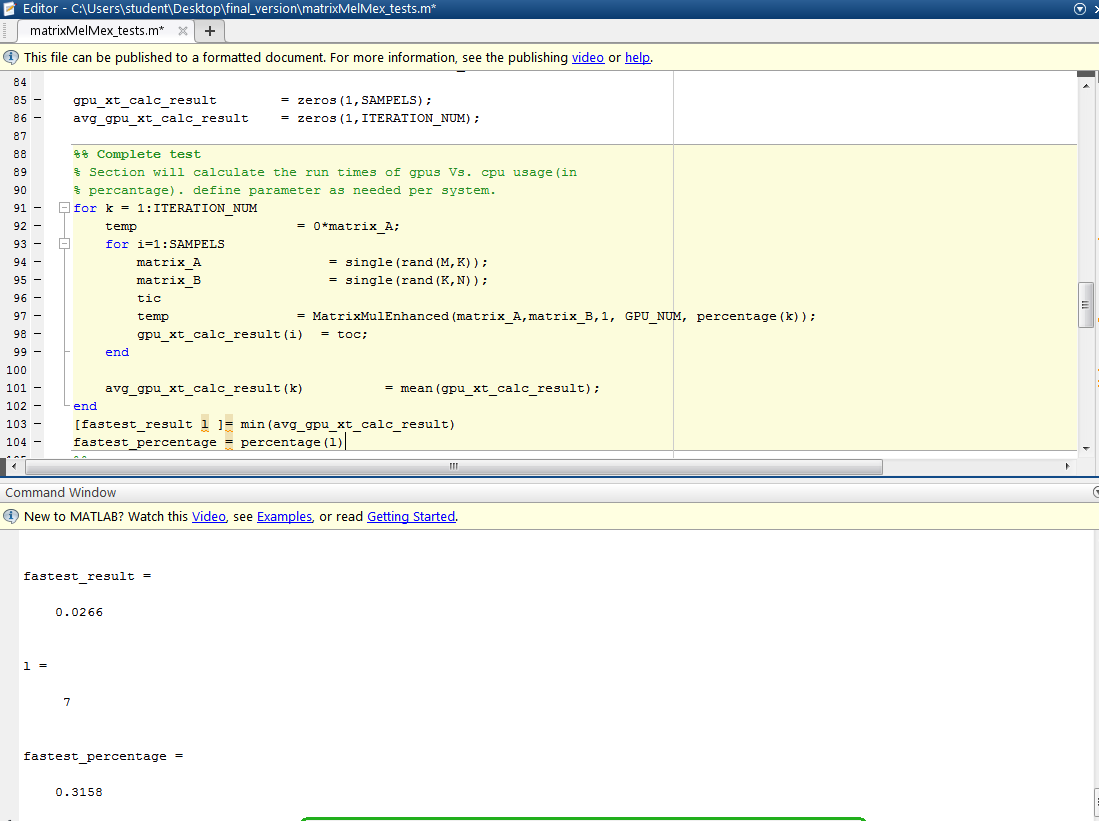
avg\_gpu\_xt\_calc\_result = zeros(1,ITERATION\_NUM): results vector

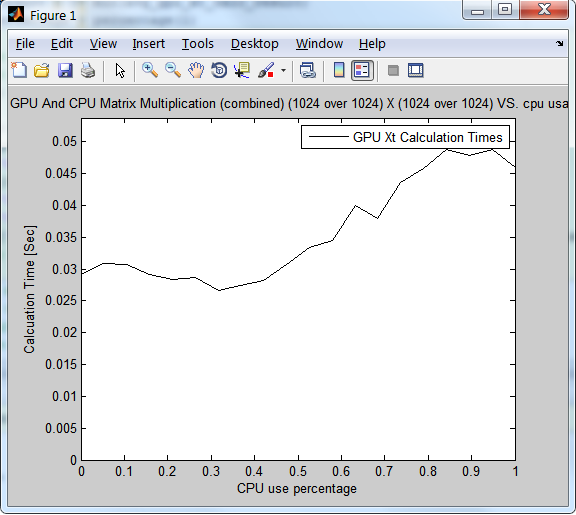
## Running the full test

Once these have been defined, we can run the full test.

The test runs on all the rations in the “percentage” vector, “SAMPLES” amount of time. Once the results are calculated, the next part of code would create a graph to demonstrate the results. Running this test might take some time, depending on the varying parameters.

In this example, the fastest calculation was for a ratio of 31% calculations on the CPU.

The graph give an idea on of the CPU only and GPU only calculation time at the edges of the graph.



## Binary minimum search on the slope

Usually, the full test creates a graph that looks like 2 linear lines. Assuming there is a global minima, we tried to find the optimal operation point by performing a binary search on the results. This test give good result on the optimal operation point, most of the time.

For the same example as before, we get a similar result for the binary search test. The results tend to be more accurate with larger matrices. This can also perform SAMPELS amount of iterations and calculate their average.

