GPU vs CPU For Matrix Multiplication

Usage:

The shebang is set to #!/home/605/sincomb/anaconda3/bin/python3 with the permissions allowed for execution. There is also a cool debug tool to help grade to see if everything is working "-d" option for debug. I reference a built-in matrix multiplication to check if my code is correct for the debugging mode.

Complete Options:

```
> ./project.py  [--CUDA | --THREADS=<numeric_value>] [--DEBUG] M N O
> ./project.py [-c | -t <numeric_value> ] [-d] M N O
```

Example CPU:

```
>  threads=12
>  M=3; N=4; O=7 # for a (MxN)*(NxO) matrix multiplication
> ./project.py -d -t $threads $M $N $0
# or simply
> ./project.py -d -t 12 3 4 7
```

Example GPU:

```
>  M=3; N=4; O=7 # for a (MxN)*(NxO) matrix multiplication
> ./project.py -d -c $M $N $0
```

Help Menu:

```
> ./project --help
```

Goal Abstract:

The aim of this project was to see how the GPU (using CUDA) and CPU (using dynamic multithreading) will speed up a traditional 1 process matrix multiplication. The rules were that the matrix multiplication had to cover rectangular matrices while also dealing with any GPU grid/block sizes to accompany such dynamic iterations.

Code Methods:

The coding language used was Python and the focused modules were PyCUDA, Numpy, and Multiprocessing (contains multithreading parameters). PyCUDA links C kernel code through a wrapper interface to simplify the CUDA experience by removing some boilerplate. The same rules still apply for the CUDA kernel where it needs to be treated as a for loop itself while allocating the appropriate memory for the array. With a fixed block size of 16 by 16, the grid was dynamically changed to fit the resulting matrix dimensions. CPU multithreading was simpler if the correct modules were used. Particularly, Python has a tricky 1 GIL issue that basically forces a 1 thread per CPU core at a time. Any more threads per core and it would wait in line sequentially. It would not help speed up the code as significantly as it would have otherwise. This will show results in why the scalability isn't as linear as it would be if this assignment were coded C or another language without this 1 GIL limitation.

Results:

At the start of the testing for GPU matrix multiplication took longer at about 3 seconds for a (50x50) matrix multiplication while the single CPU thread took .16 seconds. The bifurcation of the CPU at 16 threads is more and more significant the larger the matrices, but the most distinct result is the GPU. The fact that the runtime didn't change in any matrix size honestly worried me for a second and I had to triple check the output to make sure everything was working correctly. It was amazing, the GPU matrix multiplication was working perfecting and it didn't change until I increased the matrix multiplication to (10,000 x 10,000); it only took 10 seconds. I couldn't increase the matrix multiplication sizes for CPU past (400 x 400) due to courtesy of not hogging the nodes on Tuckoo. I ran 1 test of (1,000 x 1,000) for both GPU and 16 CPU threads to show just how slower the CPU is becoming in comparison. The CUDA code only took 3 seconds while the 16 CPU threads took 24 seconds.

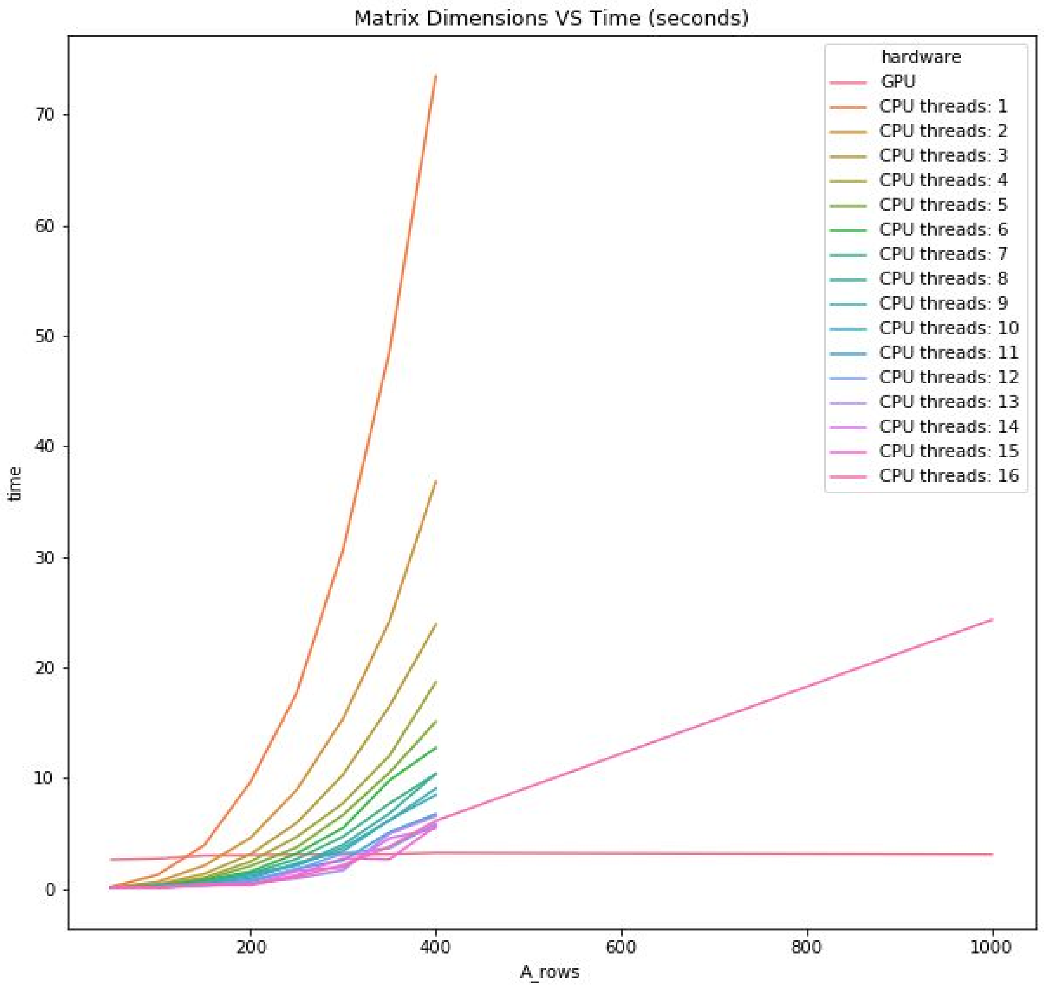
Conclusion:

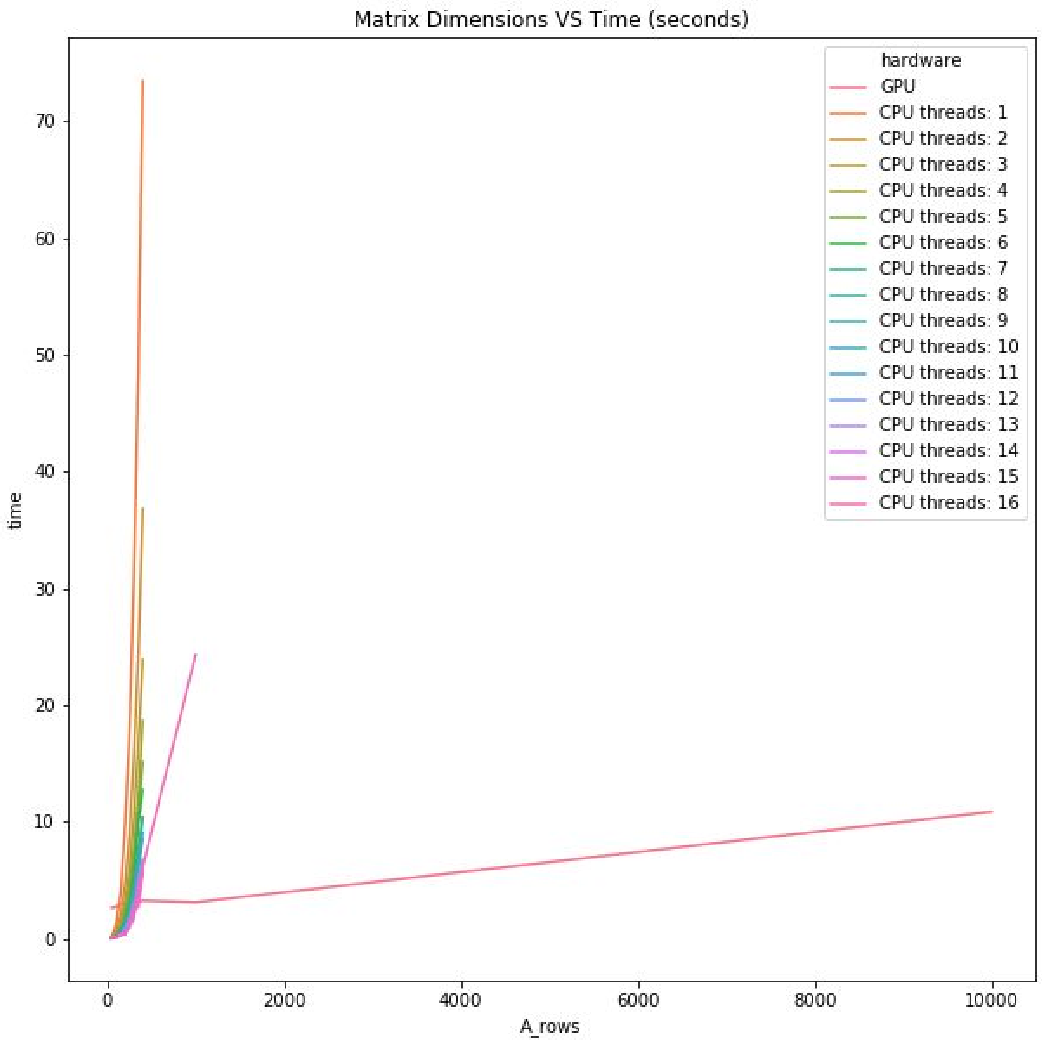
The CPU computation for many simple tasks shines for small input data due to the GPU's massive overhead. However, it is very clear that GPU computing for matrix multiplication is far superior for large matrix multiplication. If a person has all the data ahead of time to do a linear algebra related task, it would be no question that they should opt into exploring CUDA as a primary option.

Discussion:

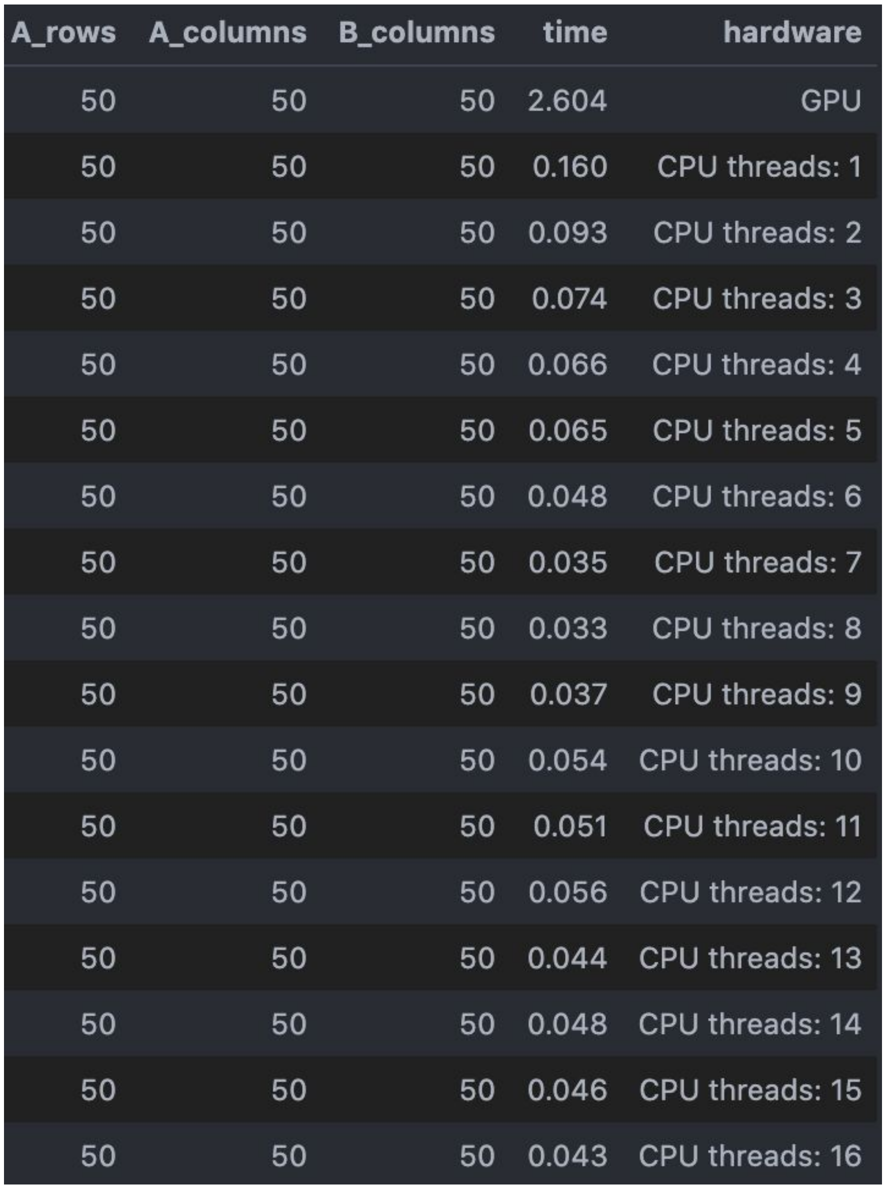
If I were to spend more time in this project I would research deeper into memory stability, specifically in shared memory tools for CUDA. There were inconsistent errors in about .01% error of the dot product when the matrices where larger than 1000 rows by 1000 columns and I think it had to do with byte stability. However I also noticed the same phenomenon with CPU multithreading at that same size. However, according to some sources this might be a difference in how the CPU and GPU handle their bytes and not an error at all. This is outside this project's scope, but worth noting nonetheless.

Graphs & Tables:



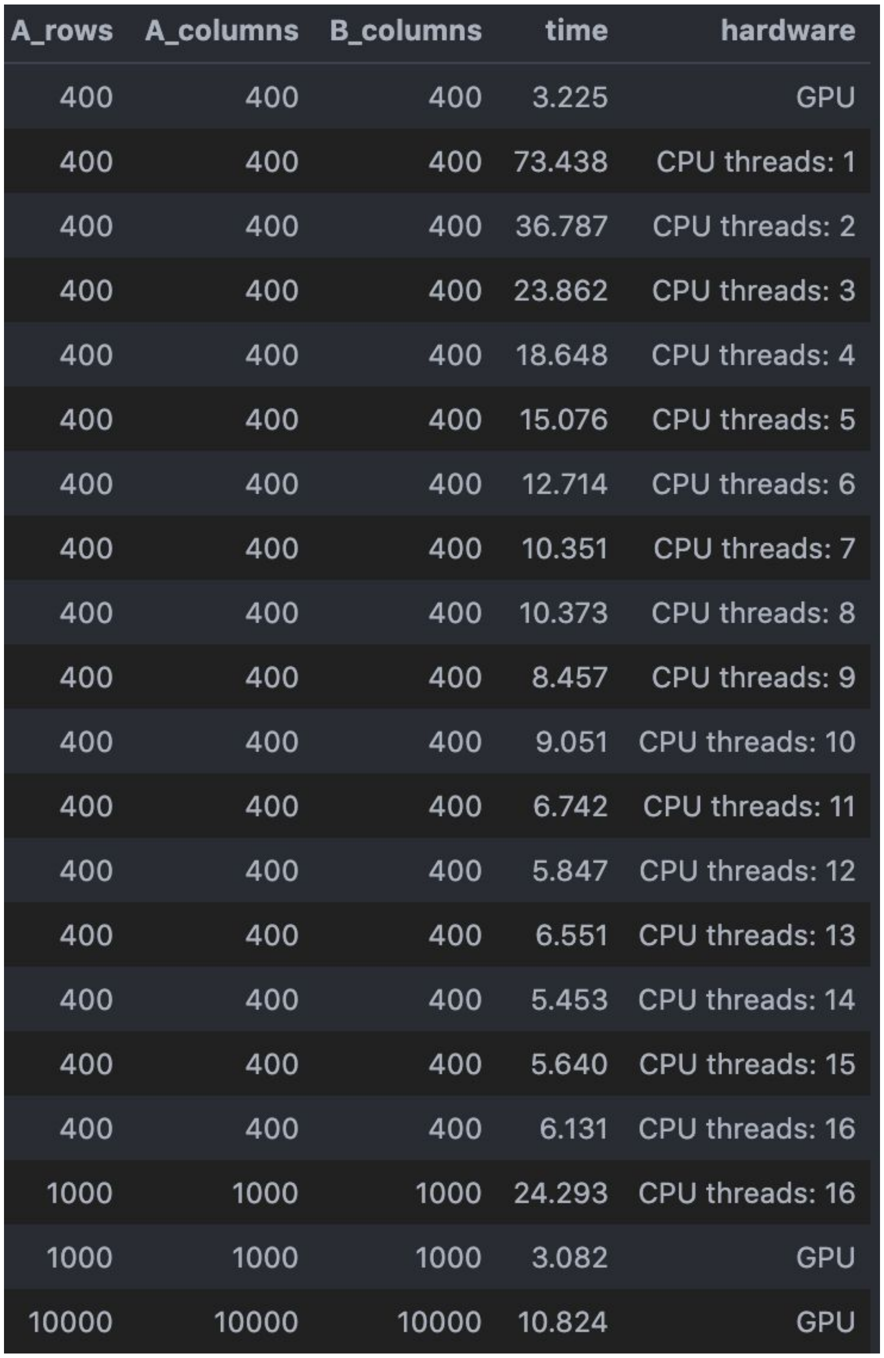

Matrix Dimensions VS Time (seconds)
time
A_rows
hardware
GPU
CPU threads: 1
CPU threads: 2
CPU threads: 3
CPU threads: 4
CPU threads: 5
CPU threads: 6
CPU threads: 7
CPU threads: 8
CPU threads: 9
CPU threads: 10
CPU threads: 11
CPU threads: 12
CPU threads: 13
CPU threads: 14
CPU threads: 15
CPU threads: 16
0
10
20
30
40
50
60
70
0
2000
4000
6000
8000
10000

First Matrix Multiplication (50 x 50) * (50 * 50):



A_rows	A_columns	B_columns	time	hardware
50	50	50	2.604	GPU
50	50	50	0.160	CPU threads: 1
50	50	50	0.093	CPU threads: 2
50	50	50	0.074	CPU threads: 3
50	50	50	0.066	CPU threads: 4
50	50	50	0.065	CPU threads: 5
50	50	50	0.048	CPU threads: 6
50	50	50	0.035	CPU threads: 7
50	50	50	0.033	CPU threads: 8
50	50	50	0.037	CPU threads: 9
50	50	50	0.054	CPU threads: 10
50	50	50	0.051	CPU threads: 11
50	50	50	0.056	CPU threads: 12
50	50	50	0.044	CPU threads: 13
50	50	50	0.048	CPU threads: 14
50	50	50	0.046	CPU threads: 15
50	50	50	0.043	CPU threads: 16

Last Matrix Multiplication:



A_rows	A_columns	B_columns	time	hardware
400	400	400	3.225	GPU
400	400	400	73.438	CPU threads: 1
400	400	400	36.787	CPU threads: 2
400	400	400	23.862	CPU threads: 3
400	400	400	18.648	CPU threads: 4
400	400	400	15.076	CPU threads: 5
400	400	400	12.714	CPU threads: 6
400	400	400	10.351	CPU threads: 7
400	400	400	10.373	CPU threads: 8
400	400	400	8.457	CPU threads: 9
400	400	400	9.051	CPU threads: 10
400	400	400	6.742	CPU threads: 11
400	400	400	5.847	CPU threads: 12
400	400	400	6.551	CPU threads: 13
400	400	400	5.453	CPU threads: 14
400	400	400	5.640	CPU threads: 15
400	400	400	6.131	CPU threads: 16
1000	1000	1000	24.293	CPU threads: 16
1000	1000	1000	3.082	GPU
10000	10000	10000	10.824	GPU