Homework #8

Team 2:

Julian Carvajal Rico

Roberto Enriquez Vargas

Question 1:

Table 1 shows the total execution times for the 3 scripts used in part 1. As shown below, the pure python takes the longest to run. When the pure python script is analyzed with the line profiler, we find that there are two lines responsible for a majority of the run time. The first line is creating and filling in the time step values of variable t. The second line, which takes even more time than the first, is creating and filling in the values of variable y, which is being used to approximate the solution at the given time step. Using just in time compilation on the int_funct function reduces the total execution time by approximately 32% when compared to the pure python code. However, this only addressed one of the bottlenecks in our code, which explains why we only saw a 32% improvement. When both bottlenecks are addressed in the Numba2 script, we see an improvement of approximately 97% percent. This demonstrates the performance advantages of bypassing python API.

Table 1. Part 1 Results

Method	Dummy Execution Time (s)	Total Execution Time (s)
Pure Python	N/A	12.166441917419434
Numba1*	0.8864812850952148	8.329026222229004
Numba2*	0.5645143985748291	0.35559582710266113

Question 2:

The implementation of Numba's JIT compilation has significantly optimized performance, decreasing execution time from 86.92 seconds to 1.72 seconds, which is a substantial improvement over pure Python execution. Initial parallelization with two threads enhances performance noticeably, but as the number of threads increases, the speed up becomes less proportionate to the number of threads used. This is evident from the Speed Up values which increase from 1 (with 1 thread) to approximately 6.42 (with 20 threads), highlighting the benefits of parallel processing.

However, the Efficiency, which is the Speed Up per thread, shows a decline as more threads are introduced, indicating a bad use of parallelization. This suggests that while parallelization has benefits, there are practical limits to its efficacy. With 20 threads, the efficiency drops to 0.32092, suggesting that each thread is less effective than when fewer threads are used. This could be due to several factors, such as hardware limitations, where the CPU may not effectively handle many threads simultaneously, or the decreased amount of parallelizable work available for additional threads, leading to overhead from thread management.

Table 2. Part 2 Results

Method	Number Threads	Total Execution Time (s)	Speed Up	Efficiency
Pure Python	N/A	86.920012	N/A	N/A
Jit Decorator	N/A	1.722404	N/A	N/A
Jit + frange	1	1.794115	1	1
Jit + frange	2	1.102911	1.626709	0.813354
Jit + frange	4	0.838788	2.138937	0.534734
Jit + frange	8	0.586995	3.05644	0.382055
Jit + frange	16	0.300951	5.961485	0.372593
Jit + frange	20	0.279527	6.418396	0.32092

In conclusion, while Numba's JIT compilation with the 'frange' function significantly reduces execution time, the parallelization strategy shows that there is an optimal number of threads for this specific task and hardware environment. Beyond this optimal point, the efficiency of adding more threads decreases. It's essential to balance the number of threads against the nature of the task and hardware capabilities to achieve the best performance.

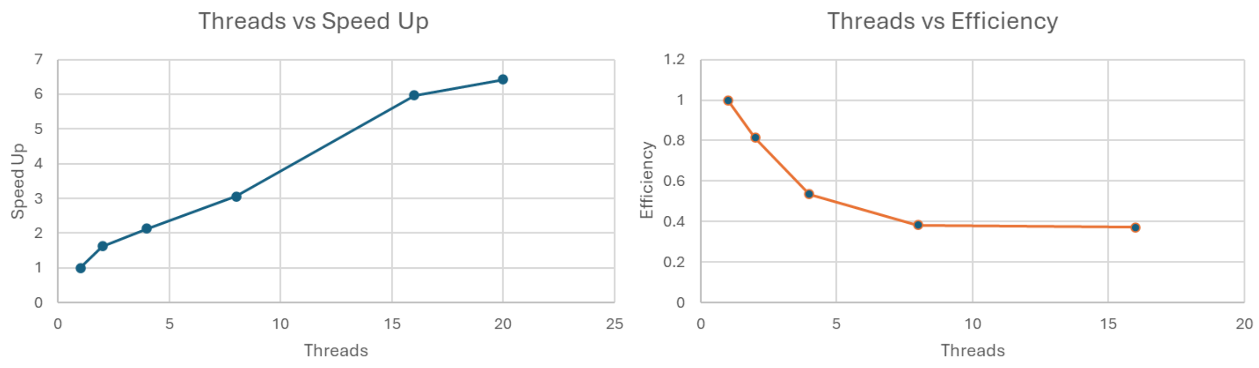

Figure 1. Threads vs Speed up/Efficiency

Question 3:

Table 3 shows the total execution times to multiply two matrices using both Cython and Numpy. For small matrices, it is clear Cython outperforms Numpy. However, once we reach a 10x10 matrix they seem to take approximately the same amount of time to execute. Surprisingly, for a 100x100 matrix, Numpy takes 10X as long as Cython. One would have expected Cython to take longer than Numpy to execute this size matrix given the results from the 10x10 matrix. However, for a 1000x1000 matrix, Cython takes approximately 10 times and long as Numpy to execute. The results from table 3 demonstrate that Cython is superior for smaller matrices up until a certain point, at which then, Numpy becomes the superior option for solving the matrix matrix multiplication problem.

Table 3. Part 3 Results

Matrix size	Cython Execution Time (s)	Numpy Execution Time(s)
3x3	3.43E-05	0.021492958
10x10	3.77E-05	3.86E-05
100x100	0.003282547	0.03038168
1000x1000	1.243914843	0.013671637