Advance HPC - Project

Hoang Manh Truong

2023-11-05

1 Versions

1. CPU (using numpy)
2. CPU (using pure python)
3. GPU (non-shared memory)
4. GPU (shared memory)

2 Steps

Given parameter ω as window size

1. Convert RGB to HSV (SCATTER)
2. For each pixel $\Phi(i, j)$
3. Define use 4 windows W^k, $k \in [1..4]$ of size $(\omega + 1) \times (\omega + 1)$
   - $W_x^1 \in [i - \omega, i]$, $W_y^1 \in [j - \omega, j]$
   - $W_x^2 \in [i, i + \omega]$, $W_y^2 \in [j - \omega, j]$
   - $W_x^3 \in [i - \omega, i]$, $W_y^3 \in [j, j + \omega]$
   - $W_x^4 \in [i, i + \omega]$, $W_y^4 \in [j, j + \omega]$
4. Find W_l, $l \in [1..4]$ having the lowest standard deviation of brightness.
   - Use V in HSV color space to calculate SD.
5. Assign mean (R, G, B) value of this window $|W_l|_{\text{RGB}}$ as new color (REDUCE, MAP)

$$\Phi(i, j)_{\text{RGB}} = |W_l|_{\text{RGB}}$$

3 Results

3.1 CPU methods

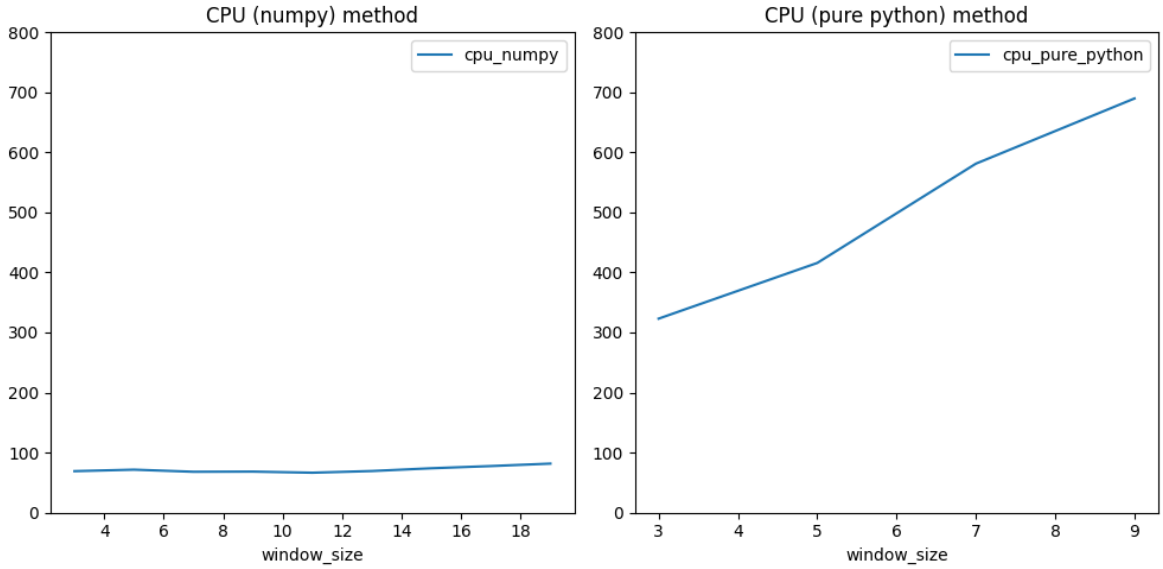
For this project, two approaches are used that utilize CPU:

- Using numpy for the computation
- Using pure python for the computation

The results are shown in the following table:

	type	time	window_size
0	cpu_numpy	68.953223	3
1	cpu_numpy	71.472559	5
2	cpu_numpy	67.938109	7
3	cpu_numpy	68.209134	9
4	cpu_numpy	66.368311	11
5	cpu_numpy	69.186078	13
6	cpu_numpy	73.795276	15
7	cpu_numpy	77.442831	17
8	cpu_numpy	81.551760	19
9	cpu_pure_python	323.052644	3

	type	time	window_size
10	cpu_pure_python	415.845123	5
11	cpu_pure_python	581.276988	7
12	cpu_pure_python	689.937136	9



As shown in the figure and table:

- **The CPU numpy approach** on average takes about 70 seconds to run. This is much faster than the pure python approach (which takes from 300 to 700 seconds), and has an approximately constant runtime for different window sizes. This is probably due to the nature of numpy, which is optimized for vectorized computation.
- **The pure Python approach** is much slower than the numpy approach, and have a runtime that increases with the window size. This is due to the use of nested loop, meaning that the time complexity scales with the window size and the sizes of the image.

3.2 GPU methods

For the GPU approach, one method are used:

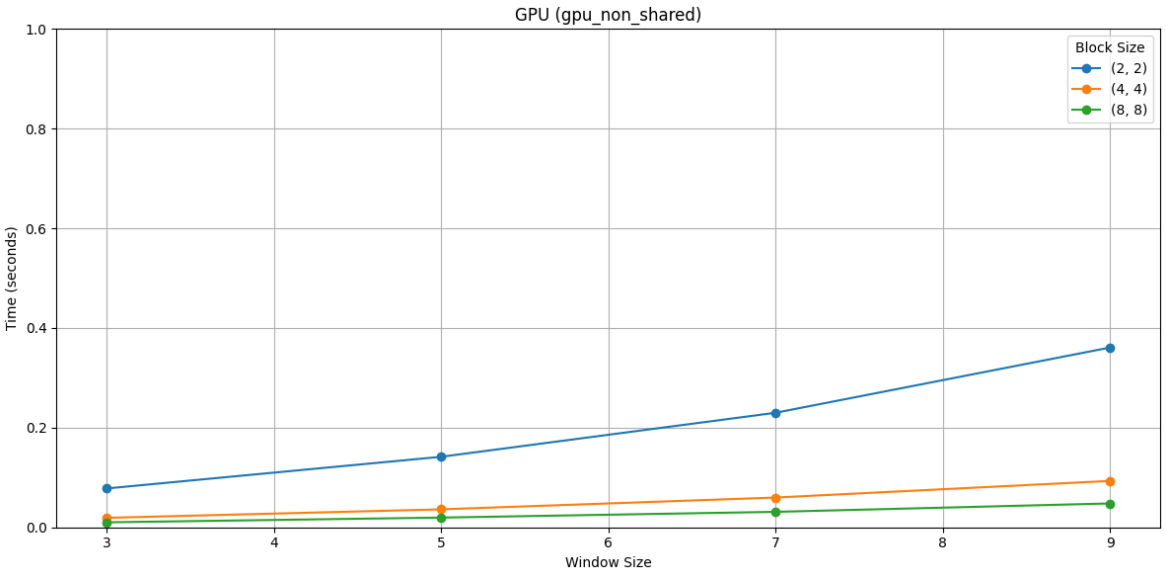
- **Using GPU without shared memory**: Result included.
- **Using GPU with shared memory**: The code is included, however it is not working properly and as such the result is omitted.

Each method is also run with different block sizes: *(2,2), (4,4), (8,8)*. Moreover, the results also exclude the time for initializing the GPU, which is done only once, so that the results are more accurate and not biased by the initialization time.

The results are shown in the following table:

	type	time	window_size	block_size
13	gpu_non_shared	0.078021	3	(2, 2)
14	gpu_non_shared	0.141573	5	(2, 2)
15	gpu_non_shared	0.229896	7	(2, 2)
16	gpu_non_shared	0.360766	9	(2, 2)
17	gpu_non_shared	0.019103	3	(4, 4)
18	gpu_non_shared	0.036170	5	(4, 4)
19	gpu_non_shared	0.059818	7	(4, 4)
20	gpu_non_shared	0.093136	9	(4, 4)
21	gpu_non_shared	0.010099	3	(8, 8)

	type	time	window_size	block_size
22	gpu_non_shared	0.019615	5	(8, 8)
23	gpu_non_shared	0.031119	7	(8, 8)
24	gpu_non_shared	0.047993	9	(8, 8)



As shown in the table and figure above, the runtime across different block sizes increases with the window sizes. Moreover, the runtime becomes smaller as the block size increases.