Vy-An Phan
26159888
cs195-15-aii

Assignment 3

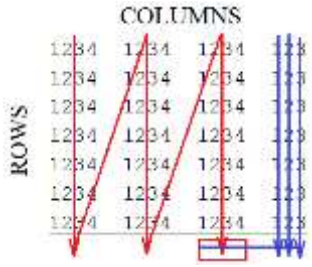
2.1 Vectorization

a) see vector_blur.cpp

b) 0.928s / 0.345s = 2.7x faster

c) `_mm_setzero_ps` for initializing vectors
`_mm_loadu_ps` for transferring pixel values from frame into my vectors
`_mm_add_ps` for summing up my vectors

d) I split the image into vectors of size 4 and summed down the rows. The leftovers I saved for last and summed those sequentially.



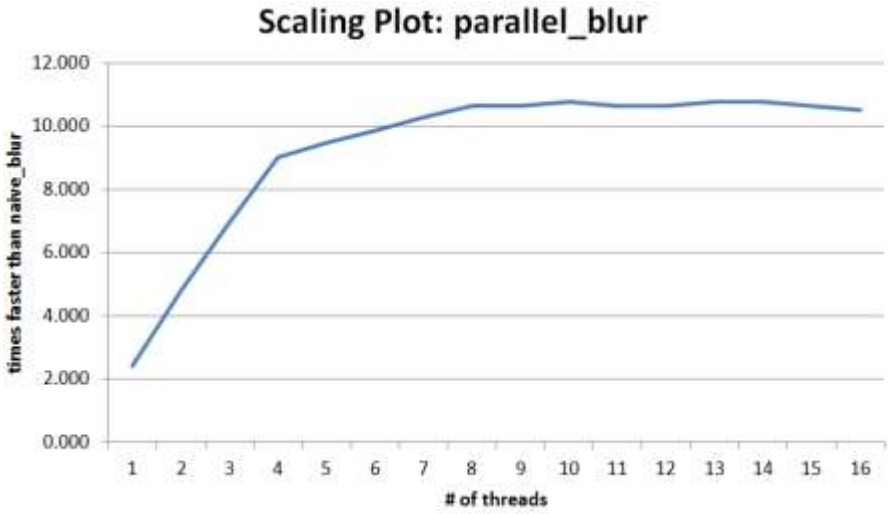
Initialize a "sum vector" to zeros (`_mm_setzero_ps`)
`_mm_loadu_ps` each "1234" frame chunk into temp vector
`_mm_add_ps` the vectors into the sum vector (red arrows)
Sequentially sum the leftovers (blue arrows)

e) By grouping all the "leftover" ends of the rows (the remainder after blocking by 4) together, I ensured that the loads would all be aligned. In the code, I implemented this by incrementing my column ends by size 4 only. As the m128 vectors are size 4, this ensures that when the loading pointer finally reaches the end of the row and realizes that there are less than 4 elements left, it skips that area. An alternate way to do this would have been to go through the frame array and pad it with zeros.

2.2 Parallelization

a) see parallel_blur.cpp

b)



c) I used parallel for with dynamic scheduling. Since the averaging of each r,c pixel is independent, I can run the averaging functions on each pixel simultaneously. The scaling increases until around 8 or 9 threads; the speed levels off after that because the hive computers only have 8 threads, and any attempts at hyperthreading will just lead to competition for cache space.

d) Yes, I considered load balancing. As a program is only as fast as its slowest thread, it's best to split tasks as evenly as possible to minimize the time any of the threads spend idle. I used dynamic scheduling to ensure the threads would request new work once they completed their assignments. In this way there would be less idle threads and the loads will be shared more equally.

2.3 Fastest Implementation

a) see fastest_blur.cpp

b) My best performance gave me 10.761x faster than the naïve implementation. This occurred at 10 threads. Anything above 8 threads broke 10x faster.

c) I just vectorized the code according to the pattern shown in 2.1.d and parallelized it using parallel for with dynamic scheduling. The greatest performance increases are due to the separate threads operating on different independent vectors simultaneously, and the load balancing from the dynamic scheduling.