CS194-15 Assignment 1: Measuring CPU Performance
Vanessa Matalon
23126281 / cs194-ax

* 2.1 Vectorization *

My vectorized version ran ~1.6 times faster than the naïve implementation. This is not that significant of a speedup, but it is to be expected since the values that the radii take on do not allow for major SIMD advantages.

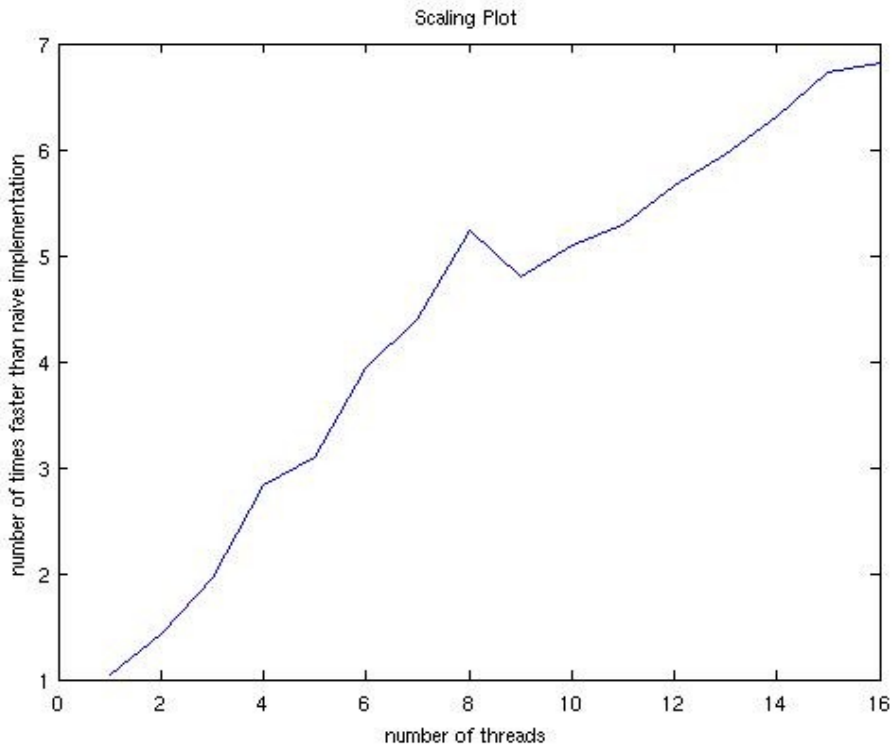
The intrinsic intructions I used were:

- `_mm_setzero_ps()`: to initialize the average vector to zeros.
- `_mm_loadu_ps()`: to load the values of my frame vector.
- `_mm_storeu_ps()`: to store the average vector into an average array which I could then use to factor them into the running average.
- `_mm_add_ps()`: to add the frame and average vectors.

I organized by kernel by vectorizing the values in the neighborhood of the current pixel (if there were enough to do this: some multiple of 4 of them) and added them to a running sum. I treated the remaining ones as if they were edge cases and executed them in the naïve, sequential way. Once I had iterated through all the pixels in the current pixel's neighborhood, this vector's values were added and converted into the average value that got sent to the output array.

I addressed unaligned/aligned loads by making sure that vectorization only occurred when loading 4 elements into a vector was did not pass the edge of the image or the kernel's dimensions. I find it worth mentioning that when I tried using `_mm_load_ps()` instead of `_mm_loadu_ps()` (the aligned version of the load instruction) I got a segmentation fault, but I did not fully understand its implications. Instead, I just made sure that vectorization never occurred when the values I needed to use were not a multiple of 4.

* 2.2 Parallelization *



I left my kernel as it was before with the exception of parallelizing and collapsing the outermost loops using: `#pragma omp parallel for collapse(2)`. I also specified which variables should be private or shared and I used the reduction keyword to let the compiler know which variables (num & avg: out[pixel]=avg/num) were being aggregated across iterations. I did this so that the assignment of current pixels was distributed among the threads while the innermost calculations took advantage of SIMD. Additionally, I made it so that the number of threads can manually be varied by the user.

I did not consider load balancing, but this deserves attention because of the randomization introduced with the radii. I would address this by using a scheduling clause to ensure that the amount of work that is done by each thread remains relatively constant.

* 3.1 Fastest *

My fastest version is 8.2398 times faster than the naïve blur.

The configuration that I used was the same one described in the previous part of the homework, except it always uses 16 threads because this is what is needed to get the best performance.