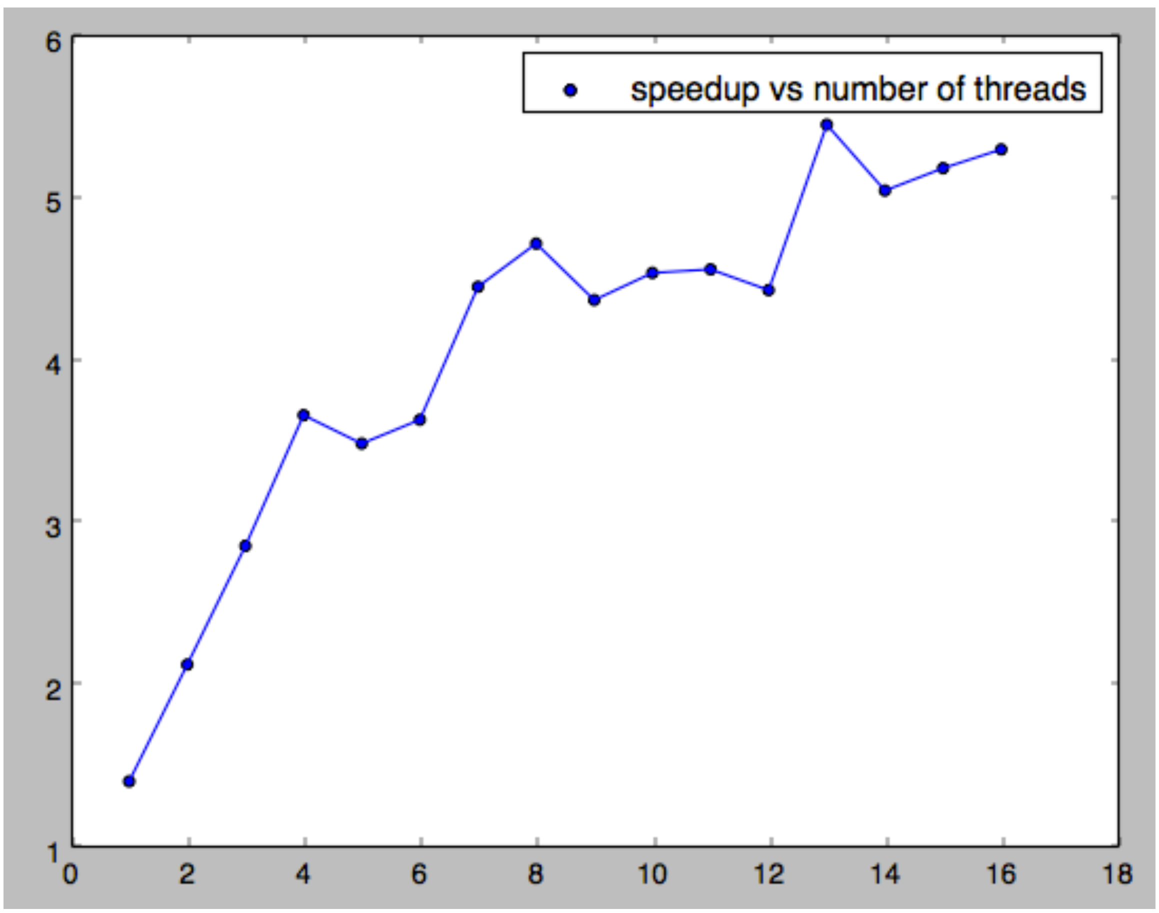
Assignment 3

2.1.1

1.
2. On average, my program got about a 1.8x speedup. It took a lot of the tweaking of how wide I wanted to unroll the loop, but I settled on jumping 4, rather than something wider for the radius.
3. I used _mm_add_ps onto my 128-bit accumulator inside the two for loops. I used _mm_loadu_ps to load 4 floats at a time from memory. And I used _mm_prefetch to look ahead. Finally, I used SSE3 instructions _mm_hadd_ps and _mm_cvtss_f32 to "fold" the vector into the final "avg" value.
4. I only affected the innermost loop, since it is the bottleneck (it's called the most times). I have one vector load per iteration, and I prefetch forward. Finally, I fold the vector and add the 4 values into a single value, and do the usual avg/num calculation.

2.3.3

1. 

2.
3. I did nothing fancy. Since all of my vectorization speedup was done on the inner loop, it allowed me to just use the OpenMP "#pragma omp parallel for

collapse(2)" to speed up the outer loops since they're "embarrassingly parallel."

4. I did consider load balancing, but I figured it was not worth the overhead. It's clear that there's an imbalanced load, since the corners of the photo have less blurring than the direct center. A possible solution would be to first divide up the inner pixels, so that all of the jobs are CPU intensive, and equally distributed. Then a free CPU could tackle the easier jobs after it's done.

<u>2.3</u>

1. 
2. My final implementation is about 6x faster than the naïve implementation.
3. This is done by combining my OpenMP with vectorization for the inner loop blur calculation. I also did simple algorithmic enhancements, like instead of counting num for every iteration, you can just say that num has to be added for the total of (loop upper limit – loop lower limit), which saves the CPU some wasted calculation.