To benchmark all my optimization functions, I took benchmarks of the original applyBlur and applyGamma functions. I had found that applyGamma ran at **118,124** clock cycles over the entirety of the program, and that applyBlur ran at **114,461** cycles. Having to make three different optimized versions of the functions, I noticed that each function had some clear inefficiencies. For applyGamma, I noticed that the line 'res = pow(v, gamma)' was being called every iteration of the loop, and exponents typically are more taxing operations. However, I didn't see a way in which I could remove that line from the loop since v was being calculated in the loop prior to the pow() operation being ran. Instead, I tried to optimize other parts of the function. For applyBlur, it was very long, and there were many instances of the in_rows[][] 2-d array being accessed, so I wanted to change that.

For applyGamma, my first optimized function reduced the number of memory references by changing the bounds of both loops, creating variables height and width. After running the program with this optimized function, applyGamma ran at **107,918** clock cycles, resulting in a speed-up of **9.5%** compared to the original function. There was still more to be optimized from there, and for my next optimization I did a 2x1 loop unrolling of the inner for-loop. This optimization ran at **94,444** clock cycles, resulting in a speed-up of **25.1%** from the original function. For my last optimization, I tried to do a more complex loop unrolling, by doing a 2x1 loop unroll of the outer for-loop, and a 5x1 loop unroll of the inner for-loop. My goal in doing this was to achieve greater parallelism, but it didn't seem to affect much, clocking in at **93,579** clock cycles, resulting in a speed-up of **26.2%** from the original. I strongly believe that the limiting factor of this function is the use of the pow() function, but it couldn't be removed using code motion.

For applyBlur, I also started by reducing memory references, changing both loop conditions and all if-statement conditions by creating variables height and width. This function ran at a total of **103,090** clock cycles, resulting in a speed-up of **11.0%**. I mentioned previously that I believed the limiting factor of this function was the constant accessing of the in_rows array, so for my second optimized function, I used code motion to limit the references to the in_rows array. This speed-up was drastic, with the function clocking in at **59,541** clock cycles, with a speed-up of **92.2%**. My original guess was correct, and this simple use of code motion almost cut the runtime of applyBlur in half. For my last optimization, I did a 2x1 loop unroll of the previous optimized function. I thought that this would be effective in speeding up the function, however, it clocked in at **58,906** clock cycles, with a speed-up of **94.3%**. Although a slight improvement, nothing too drastic or notable.

In the 'image.h' header, you can see the max-dimensions of an image is 800x800 pixels. The pixel struct has an array of 4 unsigned char, so 4 bytes, and an unsigned int, another 4 bytes, for a total of 8 bytes per pixel. Therefore, each individual cache line of 64 bytes will be able to hold 8 pixels. For applyGamma and applyTint, the pixels are loaded first, and it is a cache miss, but the next 7 pixels are also loaded, and those will be hits. This results in a cache miss rate of 12.5%. For applyBlur, every in_rows[i][j] requires the 8 other pixels surrounding it, so for the first iteration, there are 3 misses, since 3 rows of pixels are being loaded, and 6 hits. For iterations 2-7 there are all hits (9x6 = 54). Then for iteration 8, which would be the end of the cache line from the previous iterations, there are 3 accesses of a pixel not loaded in the cache, which will all be misses, but the 6 other pixels will hit. This cycle repeats, and the cache miss rate for applyBlur is 8.3%.