PS3 Questions

Add your answers to this file in plain text after each question. Leave a blank line between the text of the question and the text of your answer.

argv

1. What does *argv[0]* always contain?

It always conatains *./filename.exe*. It's the name of the executable.

2. Which entry of *argv* holds the first argument passed to the program?

The second entry, indexed with *[1]*.

3. Which entry of *argv* holds the second argument passed to the program?

The third entry, indexed with: *[2]*.

4. How would you print just the last argument passed to a program?

The number of parameter is stored in the first argument of the *main* function of type int, usually in the name of *argc*, therefore, to find the last input argument, I will do *argv[argc - 1]*, which will give the pointer to the string of the last element.

float vs double

5. What is the difference (ratio) in execution times between single and double precision for *construction with no optimization*? Explain.

For experiment of this part, we are testing on 5 instances for each input size of 2^23, 2^24, and 2^25. Each is run 5 times to get a better estimate. The CPU I had for this experiment is ryzen 4900HS.

The between the construction time with no optimization with different sizes of input between the floats and the double is averaged out to be: *0.614760292436114*.

This is close to 0.5 and slightly above, this expected because floats has only 32 bits, 4 bytes to store in the memory while double takes 64 bits, 8 bytes to store. So the times it takes to allocated double is twice as much as the floats.

6. What is the difference (ratio) in execution times between single and double precision for *multiplication with no optimization*? Explain.

Without optimization, avaeraged ratio between the time it takes to multiply floats over doubles is *0.923198319385292*, averaged out over different inputs.

The floating point operations takes the same number of clock cycles on x86. And it means that the throughput for my CPU, zen 2 is the same for floats and doubles.

7. What is the difference (ratio) in execution times between single and double precision for *construction with optimization*? Explain.

The ratio between the time takes to construct floats over double with compiler optimization is *0.4493755147684468*.

This is slightly below a half which is expected because double has twice the memory bandwidth. The means the fetching of the instructions for the CPU will take double the amount of time for double datatypes.

8. What is the difference (ratio) in execution times between single and double precision for *multiplication with optimization*? Explain.

The ratio of floats over double for multiplication with optimization is *0.3936700287599035*, meaning that the time takes for float to mulitply is less than a half of the time taken for double.

This ratio is slightly below a half. This is expected and I think all the perfomance improvement is gained via cache optimizaiton for both data types, and because occupies half of what double is, it has less cache misses when computing.

Take note that, because both has optimization of *O3*, it's likely that both will get SIMD vectorization.

9. What is the difference (ratio) in execution times for double precision multiplication with and without optimization? Explain.

The ration is: *0.6662649953910553*. It's less and the performance gain is obtained by cache optimization and SIMD optimization at the same time.

I am not sure what to expect exactly becaues in this cause the perfomance gain is obtained from better SIMD instructions AND cache optimization.

Efficiency

10. If you double the problem size for matrix-matrix product, how many more operations will matrix-matrix product perform?

The complexity is N^3, therefore, we will do 7 times more works because when scaling up the problem size by 2, the complexity scaled up by 8.

11. If the time to perform a given operation is constant, when you double the problem size for matrix-matrix product, how much more time will be required to compute the answer?

7 Times more, becase we assume that each operations is constant amount of time.

12. What ratio did you see when doubling the problem size when mmtime called *mult_0*? (Hint, it may be larger than what pure operation count would predict.) Explain.

Size of 256 yields *28ms* on mult_0, 512 yields *998*, the ratio is *35.6*. If, things are perfect, we should expect 7 times, but it's not perfect because the Matirix is Column major, the memory locality is reduced. Each multiplication requires moving to find memory addresses that are in totally different places and it will take longer for a larger matrix where the memory addresses are spread out to 4 times the space in the memory.

13. What raio did you see when doubling the problem size when mmtime called *mult_3*? Was this the same for *mult_0*? Referring to the function in amath583.cpp, what optimizations are implemented and what kinds of performance benefits might they provide?

Optimization used for *mul_3* involved 32 by 32 blocks on the matrices, which reduces the Spatial and Temporal locality at the same time, Hoisting is used for better vectorization on the compiler level, which introduces some micro optimization. This reduces the amount of time increased for each flops when the size of the matrix increases. It won't change the big O, 2 method gets closer when inputs gets larger, but they will be a constant differece because of the cache misses and lack of advance SIMD for *mul_0*.

Inputs size of 256, 512 is used for testing. *5 ms* and *88 ms* for the *mul_3* and *26* and *954 ms* for *mul_0*. The ratio is about: 17.6 for *mul_3* and 36 for *mul_0*. Notice how the use of Hoisting and Blocks Tiling improve the constant growth factor for each flops as the size of the matrix grows from small that it can be stored in the L3 cache to when, the matrix can't fit there.

14. (Extra credit.) Try also with *mult_1* and *mult_2*.

All-Pairs

15. What do you observe about the different approaches to doing the similarity computation? Which algorithm (optimizations) are most effective? Does it pay to make a transpose of A vs a copy of A vs just passing in A itself. What about passing in A twice vs passing it in once (mult_trans_3 vs mult_trans_4)?

Different approaches results in different flops/s. However all of the mappraoches appraoche the the same amount of flops as the input size of the matrix gets larger.

Thst most affective appraoch is *m_t_2*, and *m_t_5*

Passing A directly results in better performance, but it's not extremely significant.

m_t_3(A, A) and *m_t_4(A)* has very similar performance.

16. What is the best performance over all the algorithms that you observed for the case of 1024 images? What would the execution time be for 10 times as many images? For 60 times as many images? (Hint: the answer is not cubic but still potentially a problem.) What if we wanted to do, say 56 by 56 images instead of 28 by 28?

The best performance for 2014 is *m_t_5*. The implementations is tiling, without blocks, but we only focous on the upper trigular part first, and then copying the lower triagular from the upper triangular, because the resulting matrix is symmetric.

The execution time grows quadratiacally. Becauese VV^T is the action of taking cross product between every 2 images. Because each images is fixed sized, therefore complexity grows quadratically wrt to the number of images, assuming dot produt between each image is a constant.

When the size of the image increased, we have bit problem because the number of pixels grows qudratically as the scale (or size). If the number of images is fixed, then the complexity grows quadratiacally.

About PS3

17. The most important thing I learned from this assignment was ...

O3 improve performance and I don't undersand the detail.

Declares variables outside the forloops and assigning them after the for loop speeds things up, especially it accumulates a lot of values.

Tiling and seems to be the best strategy.

Spatial locality can be made better by considering matrix multiplications as blocks of sub matrices.

18. One thing I am still not clear on is.

why not:

I don't understand compiler, c++, and the details, but I think I have idea on what the assignment is trying to teach.

Why since *C(i, j)* returns a reference, why not declare *double& t = C(i, j)* before we start accumulating on *t*.