Student Name: Rohan Tiwari
AMATH 583
PS3

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?

This always contains the name of the program executable. On running ./repeat.exe, the output was:

```
argv[0]: ./repeat.exe
```

The "./" refers to the current directory where the program executable is located.

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

argv[1]

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

argv[2]

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

argc contains total number of arguments and the C++ arrays are indexed 0 to length-1 where length is number of elements in the array.

Last argument can be printed by `argv[argc-1]`

Student Name: Rohan Tiwari
AMATH 583
PS3

float vs double

---

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

Given below are some sample runs out of the many that were tried:

Input: 1000000

Construction time for double: 13

Construction time for float: 11

Ratio: 0.85

Input:10000000

Construction time for double: 107

Construction time for float: 88

Ratio: 0.82

Input:100000000

Construction time for double: 1139

Construction time for float: 959

Ratio: 0.84

Explanation: The ratio is around ~0.8 for most runs. Double is twice as large as a float. A float is 4 bytes whereas a double is 8 bytes. Without optimization, it takes a float almost the 70%-80% of time to get allocated as a double which is more than expected.

This means that the size difference is not being used well because a vector of floats should require half the amount of memory to be allocated than a double which should roughly be done in half the time.

Student Name: Rohan Tiwari
AMATH 583
PS3

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

Given below are some sample runs out of the many that were tried:

Input: 1000000

Multiplication time for double: 6

Multiplication time for float: 6

Ratio: 1

Input:10000000

Multiplication time for double: 55

Multiplication time for float: 49

Ratio: 0.90

Input:100000000

Multiplication time for double: 742

Multiplication time for float: 578

Ratio: 0.78

Explanation: The ratio is almost around 0.8-0.1 for most runs which means multiplication takes almost same time for float as it does for doubles on the CPU. Without optimization, both floats and doubles take almost the same time to multiply even though floats are half in size. This means that multiple floats are not getting loading into registers together to parallelize.

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

Given below are some sample runs out of the many that were tried:

Input:1000000

Construction time for double: 11

Student Name: Rohan Tiwari
AMATH 583
PS3
Construction time for float: 5

Ratio: 0.46

Input: 10000000

Construction time for double: 99

Construction time for float: 50

Ratio: 0.51

Input: 100000000

Construction time for double: 719

Construction time for float: 375

Ratio: 0.53

Explanation: Ratios are in the range 0.4-0.6. Floats (4 bytes) being half in size than double (8 bytes) are allocated much quicker in around half the time implies a single instruction is able to allocate almost 2 floats on an average at once compared to single double.

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

Given below are some sample runs out of the many that were tried:

Input:1000000

Multiplication time for double: 4

Multiplication time for float: 2

Ratio: 0.5

Input: 10000000

Multiplication time for double: 27

Multiplication time for float: 14

Ratio: 0.52

Input: 100000000

Multiplication time for double: 287

Multiplication time for float: 124

Ratio: 0.43

Explanation: The ratio is around 0.4-0.5 for most runs. This implies that 2 floats are multiplied in parallel or at the same time as they are smaller than a double. This indicates that 2 floats are being able to fit in a single register so they can be operated on in parallel. Generally, the timing is lower for both double and floats indicating the possibility of advanced optimizations being applied e.g. use of SIMD registers.

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

Input: 1000000

Multiplication time for double with optimization: 4

Multiplication time for double without optimization: 6

Ratio: 0.67

Input: 10000000

Multiplication time for double with optimization: 27

Multiplication time for double without optimization: 55

Ratio: 0.50

Input: 100000000

Multiplication time for double with optimization: 287

Multiplication time for double without optimization: 742

Ratio: 0.4

Explanation: Most runs yielded ratios between 0.4-0.6. There is possibility of using SIMD registers here to load multiple doubles in SIMD registers and process in parallel. Looks like we are able to load around 2 doubles (8*2 = 16 bytes) at once into the SIMD registers.

Efficiency (583 only)

---

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

Matrix-matrix product has a time complexity of O(N^3) where N is the dimension of the matrix.

When we double N, we get a complexity of 8 * O(N^3) which means 8 times more work or 7 times additional operations.

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?

If we keep time to perform a given operation constant, it will need 7 additional units of time or 8 times the original 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.

Resolution was increased to get sensible numbers by running core loop many times.

size = 128, time = 13ms

size = 256, time = 186ms, ratio = 14.31

size = 512, time = 5366ms, ratio = 28.85

size = 1024, time = 47756ms, ratio = 8.90

Ratio: around 9-30.

Explanation: mult_0 is the unoptimized implementation. The ratio is larger than pure operation count because larger matrices do not fit in the nearby caches leading to more accesses to the memory. Accessing the memory is very slow compared to accessing the caches, also the caches are ordered by size so accessing L1 is much faster than accessing L2 and so on. This increases the time beyond just the pure operation count that we get from the time complexity of O(N^3).

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?

Resolution was increased to get sensible numbers by running core loop many times.

size = 128, time = 3ms

size = 256, time = 26ms, ratio = 8.67

size = 512, time = 462ms, ratio = 17.76

size = 1024, time = 3841ms, ratio = 8.31

Ratio: around 8-20.

Explanation: Ratio is a bit lower than mult_0 due to Hoisting, tiling and blocking optimizations implemented. Locality is the most important requirement for HPC. Hoisting moves the innermost loop invariant code one level outside reducing the number of memory operations to boost efficiency. It reduces the number of reads and writes to C(i,j). Tiling with 2x2 tiles reduces the number of reads from A and B improving efficiency. Blocking splits the larger problem into sub problems so that they can fit completely in caches to boost efficiency.

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

mult1 (implements hoisting):

size = 128, time = 11ms

size = 256, time = 158ms, ratio = 14.36

size = 512, time = 3870ms, ratio = 24.50

size = 1024, time = 36362ms, ratio = 9.39

Ratio: around 10-25.

Explanation: Hoisting reduces the number of reads and writes to C(i,j). Ratio is about the same at mult_0 but the running times for larger matrices is much faster than mult_0.

Student Name: Rohan Tiwari
AMATH 583
PS3

mult2 (implements hoisting and tiling)

size = 128, time = 3ms

size = 256, time = 30ms, ratio = 10

size = 512, time = 774ms, ratio = 25.8

size = 1024, time = 11172ms, ratio = 14.43

Ratio: around 10-25.

Explanation: Hoisting reduces the number of reads and writes to C(i,j) to improve efficiency.

Tiling with 2x2 tiles reduces the number of reads from A and B improving efficiency. The running times are much faster particularly for larger matrices compared to mult_0 and mult_1.

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)?

#images mult_0(A,B) mult_1(A,B) mult_2(A,B) mult_3(A,B) m_t_0(A,C) m_t_1(A,C) m_t_2(A,C) m_t_3(A,C) m_t_0(A,A) m_t_1(A,A) m_t_2(A,A) m_t_3(A,A) m_t_4(A)

128 1.51118 1.51118 6.42253 8.56337 1.60563 1.71267 8.56337 8.56337 1.51118 1.71267 6.42253 8.56337 25.6901

256 -1 -1 3.31485 7.34003 -1 -1 5.70891 7.90465 -1 -1 5.40844 7.90465 25.6901

512 -1 -1 1.44225 2.85446 -1 -1 4.56713 7.34003 -1 -1 4.89335 6.96681 18.6837

1024 -1 -1 -1 2.80575 -1 -1 3.33502 7.24303 -1 -1 3.4908 7.11761 16.6077

2048 -1 -1 -1 2.56901 -1 -1 3.0113 6.90102 -1 -1 3.2736 7.03387 16.7345

Tried above runs many times for different image sizes and provided above is a sample output.

Student Name: Rohan Tiwari
AMATH 583
PS3

What do you observe about the different approaches to doing the similarity computation?

We have following types of approaches overall:

1. Mult_*(A,B): This is the approach uses A and transpose(A) as function arguments. This performs badly for larger sizes from 512 onwards. Mult_3 (A,B) is the most efficient because it implements hoisting, tiling and blocking. Also it can be seen for matrix sizes >= 1024 blocking is highly beneficial.
2. M_t_*(A,C): This approach uses A and copy of A as function arguments. This approach is more efficient than the first one particularly for larger sizes from 512 onwards. It is worth noting that m_t_3 which implements blocking (in addition to the optimizations in m_t_2) is always the most efficient and the gap increases for larger image sizes indicating that blocking is beneficial and increases cache efficiency.
3. M_t_*(A,A): This approach uses A passed in twice (pass by reference). This avoids making a copy of A. The efficiency is almost about the same as the second approach.
4. M_t_4(A): This passes A once. This gives a big jump in efficiency. Probably the reason is that the single matrices (A) can be easily cached as against the previous approaches that require caching of two matrices which may more easily lead to cache size constraints.

Within each approach, the most optimized version is generally the most efficient (most optimized version ending with _3). The _3 function has hoisting, tiling and blocking optimizations implemented.

m_t_*(A,C) and m_t_*(A, A) perform the better than mult_*(A, B) indicating that passing in a copy of A and A itself is better than passing in the transpose separately.

Passing in A just once is the best in m_t_4.

Which algorithm (optimizations) are most effective?

m_t_4(A) is the most effective - it passes A once and applies hoisting, tiling and blocking on the multiplication. This also saves making a copy of A.

Though this approach tends to lose some efficiency for larger image counts (1024 and above), it still performs the best.

Does it pay to make a transpose of A vs a copy of A vs just passing in A itself?

Yes. it pays to just pass in A itself but not by much. Passing in A itself saves the time to make a copy of A.

What about passing in A twice vs passing it in once (mult_trans_3 vs mult_trans_4)?

Passing in A only once performs better and is more efficient because it does not have to operate on two matrices probably able to use cache better for locality.

Student Name: Rohan Tiwari
AMATH 583
PS3

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?

What is the best performance over all the algorithms that you observed for the case of 1024 images?

m_t_4(A) with 16.60 gflops/s is mostly the best when tried multiple times.

What would the execution time be for 10 times as many images?

The time complexity of matrix multiplication is O(N*N*K). N is number of images and K is number of pixels per image (28*28=784). If the number of images is increased to 10 times, the execution should increase to 100 times. Size of A matrix = 10240 * 768 = 7.86 MB which should fit in L3 cache which is 8 MB on my machine but not in L1 (512KB on my machine) and L2 (4MB on my machine) caches. Similarity matrix size would be 10240 * 10240= 105 MB which won't fit in cache and will require trips to memory.

For 60 times as many images?

With the same logic as that for 10 times as many images, the execution time would be 3600 times for 60 times as many images. With 60 times as many images, A matrix will not fit in caches nor will the similarity matrix. Size of A matrix = 60*1024*784 = 48 MB. Size of Similarity matrix = 3600*1024*1024= 4 GB. Similarity may not even fit in the memory on some machines but will fit in mine whose RAM is 8 GB.

What if we wanted to do, say 56 by 56 images instead of 28 by 28?

This increases the size of vectorized image in matrix A by 4 i.e. number of columns of A increase 4 times. Time complexity of matrix multiplication is O(N*N*K). N is number of images and K is number of pixels per image (56*56=3136). K increases by a factor of 4 which means execution time should increase 4 times. Size of matrix A = 1024 * 3136 = 3.2MB which should fit in L2 cache (4 MB on my machine). Size of Similarity matrix = 1024*1024 = 1.04 MB which also fit in L2 cache. But both A and Similarity matrix won't fit together in L2 cache and will require L3 cache (8 MB on my machine) as well. Appropriate size of blocking should help improve performance in this case to get the data to closer caches.

Student Name: Rohan Tiwari
AMATH 583
PS3

Extra Credit: There is an optimization that you can make to the last algorithm that can significantly cut the running time. Implement that optimization as a new function mult_trans_5 and report its performance (you will need to add a call to it to the benchmark driver). (Hint: This optimization involves doing less work – not doing the same work faster.)

#images mult_0(A,B) mult_1(A,B) mult_2(A,B) mult_3(A,B) m_t_0(A,C) m_t_1(A,C) m_t_2(A,C) m_t_3(A,C) m_t_0(A,A) m_t_1(A,A) m_t_2(A,A) m_t_3(A,A) m_t_4(A) m_t_5(A,A)

128 1.37292 1.47323 5.68869 7.47457 1.69942 17.0472 25.1864 22.6744 1.70914 18.0155 26.161 22.7145 21.4084 30.8775

256 -1 -1 2.8566 4.40276 -1 2.61311 22.3247 18.9141 -1 15.2826 24.2588 19.5139 20.8101 30.822

512 -1 -1 1.43258 2.8662 -1 10.5693 16.4885 18.0662 -1 12.562 19.8398 16.4134 17.6952 28.5129

1024 -1 -1 -1 2.9102 -1 3.25113 7.47254 16.5547 -1 3.10313 7.27646 17.2097 17.845 26.6555

2048 -1 -1 -1 2.97485 -1 3.12453 6.89798 16.2654 -1 2.81861 6.07995 15.1561 17.2509 25.1944

Explanation: m_t_5(A,A) is always most efficient than other implementations. It is almost always about >= 50% efficient than m_t_3(A,A) and similarly for m_t_4(A).

M_t_5(A,A) implements hoisting, tiling, blocking to increase overall efficiency. It also implements algorithm optimization to take advantage of the second argument being the transpose of A. The optimization allows the function to operate essentially on half the problem set and then apply the results to the corresponding transposed location in the result matrix. This makes the function do less work compared to other optimizations that implement hoisting, tiling and blocking (m_t_3 and m_t_4).

About PS3

---

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

Concept of locality and why it is important for achieving high performance.

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

How does CPU branch prediction work?