The following optimization strategies covered in this lab:

1. Improving data locality / cache usage
2. Memory allocation / deallocation
3. Avoiding too many small function calls
4. The keyword restrict and its effect on performance
5. The keyword const and its effect on performance
6. Blocking to improve use of cache memory
7. Function side effects and pure functions

Task-0:

Since practically all modern processors use cache memories, the cost of memory access depends a lot on data locality; if we have recently accessed data at a certain memory location, nearby memory accesses will be cheap since that memory location is likely to be in cache.

n = 30, gave the best result.

Task-1:

bubble_sort scales as O(N^2) while merge_sort scales as O(N*logN). Therefore, merge_sort is expected to be much faster when the list to be sorted is long.

Note that merge_sort is implemented recursively; it divides the list into smaller parts and then calls itself for each part.

merge_sort:

- N = 1000000
- Before sort: the number 7 occurs 1051 times in the list.
- Sorting list with length 1000000 took 0.094 wall seconds.
- After sort : the number 7 occurs 1051 times in the list.
- OK, list is sorted!
- ./sort_test 1000000 0.10s user 0.01s system 97% cpu 0.116 total

bubble_sort:

- N = 1000000
- Before sort: the number 7 occurs 1051 times in the list.
- Sorting list with length 1000000 took 162.615 wall seconds.
- After sort : the number 7 occurs 1051 times in the list.
- OK, list is sorted!
- ./sort_test 1000000 161.78s user 0.40s system 99% cpu 2:42.64 total

Without the free() calls:

- N = 1000000
- Before sort: the number 7 occurs 1051 times in the list.

- Sorting list with length 1000000 took 0.087 wall seconds.
- After sort : the number 7 occurs 1051 times in the list.
- OK, list is sorted!
- ./sort_test 1000000 0.10s user 0.01s system 41% cpu 0.266 total

There are several reasons why memory leaks in a case like this can lead to worse performance. One reason is that the memory allocations (malloc() calls) will probably become more expensive because there is a larger strain on the operating system's memory management system. Another reason is less efficient cache usage: if we never call free, each malloc call will give us a completely new memory block that has not been used before, so it will not be in cache. On the other hand, if we free the old memory blocks before allocating new ones, it is likely that we will often be working with partly the same memory addresses, so there is a better chance that the memory addresses we are using will already be in cache.

Code with buffer (merge_sort):
- N = 10000000
- Before sort: the number 7 occurs 10027 times in the list.
- Sorting list with length 10000000 took 0.843 wall seconds.
- After sort : the number 7 occurs 10027 times in the list.
- OK, list is sorted!
- ./sort_test 10000000 0.83s user 0.13s system 98% cpu 0.974 total

Code without buffer (merge_sort):
- N = 10000000
- Before sort: the number 7 occurs 10027 times in the list.
- Sorting list with length 10000000 took 0.902 wall seconds.
- After sort : the number 7 occurs 10027 times in the list.
- OK, list is sorted!
- ./sort_test 10000000 0.89s user 0.12s system 80% cpu 1.259 total

Answer:
- Yes, there are improvements

An alternative approach to get rid of the memory allocations in our merge_sort implementation would be to place the list1 and list2 buffers on the stack instead of calling malloc. This should work provided that the list to be sorted is not too big, so that we do not run out of stack space, usually giving "segmentation fault" errors. Do this change and check how it works. Did it make the code faster? How long lists can you sort in this way, before running into the problem with the stack size? In Linux, you can check the stack size and other system limits using the command "ulimit -a":

Fixing so that it is stored on the stack instead of using malloc:
I changed to this code instead:
int list1[n1]
int list2[n2]

- N = 10000000 (This gave the segmentation error)

- Before sort: the number 7 occurs 10027 times in the list.
- zsh: segmentation fault ./sort_test 10000000
- ./sort_test 10000000 0.09s user 0.01s system 37% cpu 0.270 total

Running ulimit -a on my linux environment on my computer:

- stack size (kbytes, -s) 8192

Task-2:

Check how the performance is affected by the used datatype. Compare long long, long int, int, short int, and char. Which one gives best performance? Can you understand why?

- char:
    - N = 100000000
    - Before sort: the number 7 occurs 1000986 times in the list.
    - Sorting list with length 100000000 took 10.319 wall seconds.
    - After sort : the number 7 occurs 1000986 times in the list.
    - OK, list is sorted!
    - ./sort_test 100000000 10.53s user 0.63s system 98% cpu 11.354 total
- int:
    - N = 100000000
    - Before sort: the number 7 occurs 1000986 times in the list.
    - Sorting list with length 100000000 took 9.362 wall seconds.
    - After sort : the number 7 occurs 1000986 times in the list.
    - OK, list is sorted!
    - ./sort_test 100000000 9.89s user 0.21s system 97% cpu 10.329 total
- long int:
    - N = 100000000
    - Before sort: the number 7 occurs 1000986 times in the list.
    - Sorting list with length 100000000 took 10.813 wall seconds.
    - After sort : the number 7 occurs 1000986 times in the list.
    - OK, list is sorted!
    - ./sort_test 100000000 10.67s user 0.89s system 96% cpu 11.953 total
- long long:
    - N = 100000000
    - Before sort: the number 7 occurs 1000986 times in the list.
    - Sorting list with length 100000000 took 11.287 wall seconds.
    - After sort : the number 7 occurs 1000986 times in the list.
    - OK, list is sorted!
    - ./sort_test 100000000 10.72s user 1.33s system 97% cpu 12.342 total

Smaller datatypes may perform better due to better cache locality and lower memory bandwidth usage.

Task-3:

Here we will investigate how the keyword restrict can affect performance.

Pointer aliasing:

- It occurs when two or more pointers refer to the same memory location. This can prevent the compiler from making certain optimizations because it must assume that modifying one pointer's data might affect another pointer's data.

We can tell the compiler that each pointer points to separate data by using the restrict keyword.

Does the restrict keyword have a bigger effect when n is small?

- There is a small change when compiling with small n and increasing N2.

Task-4:

In this task, we will look at an example of how the const keyword can improve performance

The keyword restrict did not really change the performance.

When declaring NP as a const, huge improvements were made. From 0.152 to 0.035 seconds.

The reason why declaring NP as const can help performance is that when it is not const, the compiler must assume that the value can change while the program is running

Try moving the line int NP = 2 inside the transform_opt function definition. Now check the performance with and without const. Does const seem to make any difference now? Can you understand why?

- There was no difference in performance. This is likely because NP is now a local variable inside `transform_opt`, allowing the compiler to handle it efficiently. Since its value is known at compile time, it does not need to be recomputed, and `const` does not provide any additional optimization benefit in this case.

Task-5:

Now we will look at how a matrix transpose operation can be improved by blocking, to make better use of the computer's cache memory.

the do_transpose_optimized variant uses blocking to achieve a better memory access pattern.

Without changing block_size:

- do_transpose_mod_standard 1 times for N=5000 took 0.111 wall seconds.

- do_transpose_mod_optimized 1 times for N=5000 took 0.042 wall seconds.
- ./transpose_test 0.50s user 0.10s system 67% cpu 0.891 total

in transpose.c, changing the blockSz gives very little change. Still only 1000 blockSz before segmentation error.

In transpose_x.c when changing blockSz gives a difference, but after blockSz of 1000, we get segmentation fault.

Task-6:

This task is about pure functions

Now, if the programmer knows that the function f is pure, meaning that it has no side effects, the code can be optimized by moving the function call f(i) out of the inner loop and storing the value in a variable, and simply using that variable inside the inner loop. Try doing this optimization and verify that it really makes the program run faster, and that the result is the same.

Without optimization (f(i) inside the inner loop)
- Result sum: 838134976
- Function call counter: 400000000
- ./pure_test 0.85s user 0.02s system 75% cpu 1.141 total

With optimization (f(i) stored in a variable outside the innerloop, in the outer loop)
- Result sum: 838134976
- Function call counter: 20000
- ./pure_test 0.00s user 0.01s system 6% cpu 0.174 total

When adding this: int f(int k) __attribute__((const)); (explicitly telling the compiler that a function is pure)
We managed to get this time, which is a lot faster than without changing place of f(i):
- Result sum: 838134976
- Function call counter: 20000
- ./pure_test 0.01s user 0.01s system 6% cpu 0.263 total

Putting all the code in the same file gave better results:
- Result sum: 838134976
- Function call counter: 400000000
- ./main 0.06s user 0.01s system 22% cpu 0.292 total

Task-7:

Version kij, time = 0.111701
Version ijk, time = 0.892775
Version jik, time = 0.941909
./matmul 1.94s user 0.03s system 39% cpu 4.995 total

According to the time, kij performs much faster.