HIGH PERFORMANCE PROGRAMMING
UPPSALA UNIVERSITY
SPRING 2018
LAB 5: MEMORY USAGE

This lab is mostly about how a program's memory usage affects the performance, but also includes some other code optimization aspects.

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

1. Main Tasks

Start by downloading and unpacking the `Lab05_MemUsage.tar.gz` file which contains the files needed for the tasks in this lab.

The code for each task is available in a separate directory for each task, including a makefile that you can use to compile. If you want to try different compiler optimization flags, you can do that by changing CFLAGS in the makefiles.

Task 0

The code for this task is in the `Task-0` directory.

Here we will look at how *data locality* can affect performance. 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.

The code in `cachetest.c` performs some operations on the data in a vector of structs, where each struct itself contains a vector of length `N`. The operations performed are done either by calling `ModifyWithStep8()` or `ModifyLow()`. Althogh

Date: January 29, 2018.

those functions perform the same number of operations, they are different when it comes to data locality.

Look at the code in `cachetest.c` and make sure you understand what it is doing. Note that you can choose if it will call `ModifyWithStep8()` or `ModifyLow()` by toggling between "`#if 1`" and "`#if 0`" in the main function. Considering data locality, which function should be faster? Run the code in both ways, measure timings and check if the result matches your expectations.

Note that the code in the main function repeats the j-loop `m` times. If all data operated on would fit in cache, we would expect the code to be almost equally fast regardless of the `ModifyWithStep8()` vs `ModifyLow()` choice, since all data would anyway be in cache after the first iteration of the outer loop. So if you see a clear performance difference, that indicates that all the data does not fit in the L1 data cache on the computer you are using. Try making `n` smaller in some steps and increase `m` correspondingly to keep the same number of operations, e.g. set `n` to $200 \longrightarrow 100 \longrightarrow 50 \longrightarrow 30 \longrightarrow 20 \longrightarrow 10$. Can you find a point where the `ModifyWithStep8()` code suddenly becomes a lot faster? The size of data where that happens should correspond to the size of the L1 data cache on the computer you are using.

Extra part: the cache typically works using a smallest unit called a cache line, so that when something is put in cache it is always a whole cache line at a time. The size of a cache line is usually 64 or 128 bytes. Think about how the cachetest.c code can be modified to help you test what the size of a cache line is on the computer you are using. Does it seem to be 64 or 128 bytes, or something else?

Task 1

The code for this task is in the `Task-1` directory.

The code for this task includes implementations of two sort routines: `bubble_sort` and `merge_sort`. They are declared in `sort_funcs.h` and implemented in `sort_funcs.c`.

The `bubble_sort` and `merge_sort` routines use different algorithms, with different scaling (computational complexity): `bubble_sort` scales as $\mathcal{O}(N^2)$ while `merge_sort` scales as $\mathcal{O}(NlogN)$. Therefore, `merge_sort` is expected to be much faster when the list to be sorted is long. Look through the code and try to understand how the algorithms work. Note that `merge_sort` is implemented recursively; it divides the list into smaller parts and then calls itself for each part.

The main program in `main.c` first creates a list with random numbers to be sorted, then calls a sort routine to sort the list, and finally verifies that the list was sorted properly.

To get an idea of the difference between the two sort routines, start by changing one line in `main.c` so that `bubble_sort` is called instead of `merge_sort`, and see how this affects the time it takes to sort the list for a few different list lengths. When you have tried this, change back to using `merge_sort` again.

Now, let us consider how memory leaks can affect the performance of a program. In the `merge_sort` implementation, `malloc` is called twice in the beginning and then there are two corresponding calls to `free` in the end. Now introduce a memory leak by removing the `free` calls. How does this affect the performance?

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.

Many small memory allocations can be a performance problem (even if there are no memory leaks). In case of our `merge_sort` implementation, an easy optimization is to replace the two `malloc` calls by allocating a single buffer and setting `list1` and `list2` by pointing into that buffer. This reduces the number of calls to malloc/free to half. Do this optimization and run the program again. Do you see any performance improvement?

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":

```
elias@elias-asus-N53SM:~$ ulimit -a
core file size          (blocks, -c) 0
data seg size           (kbytes, -d) unlimited
scheduling priority             (-e) 0
file size               (blocks, -f) unlimited
pending signals                 (-i) 31406
max locked memory       (kbytes, -l) 64
max memory size         (kbytes, -m) unlimited
open files                      (-n) 1024
pipe size            (512 bytes, -p) 8
POSIX message queues     (bytes, -q) 819200
real-time priority              (-r) 0
stack size              (kbytes, -s) 8192
cpu time               (seconds, -t) unlimited
max user processes              (-u) 31406
virtual memory          (kbytes, -v) unlimited
file locks                      (-x) unlimited
```

In the case above the stack size is 8192 kbytes, so if a program tries to put more than that on the stack, it will fail. Use the ulimit command to check the stack size limit on the computer you are using.

Another possible optimization of the `merge_sort` implementation is to reduce the number of function calls by avoiding recursive calls when the list is very short. An easy way of doing this is to simply call `bubble_sort` if the list N is smaller than some number n. Although `bubble_sort` is very inefficient for long lists, it can be faster for very short lists since it avoids many small function calls. Perform this optimization, and try different values of n. What seems to be the optimal value of n, and how much faster can you make the code in this way?

Extra part: can you optimize further by removing all memory allocation from the merge sort routine, and instead use a work buffer allocated only once, in the main program? How much performance do you gain?

Task 2

The code for this task is in the `Task-2` directory.

In this task, we will look at a similar merge sort program as in the previous task, but this time the code is written using a `typedef` for the integer type `intType` used for the numbers to be sorted, so that we can easily change between different integer types. Look through the code to see how `intType` is used. Note that the `typedef` line must appear before `intType` is used. What happens otherwise? You can test this for example by commenting out the `typedef` line in `sort_funcs.h` and then trying to compile.

The numbers to be sorted are now in the range 0 to 99, so it is not really necessary to represent them using the `int` type; it is possible to represent them also using `short int` or even `char`.

The type used can be changed by changing the typedef statement defining `intType` on the first line in `sort_funcs.h`.

Type	Size in bits on 64-bit Linux	Size in bits on 32-bit SunOS
char	8	8
short int	16	16
int	32	32
long int	64	32
long long	64	64

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?

Note that `intType` is not used for all integers in the code, only for the integers referring to list elements. What would happen if we used `intType` for everything?

Task 3

The code for this task is in the `Task-3` directory.

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

When a function has several pointers as input, the compiler must take into account that those pointers may point to the same (or nearly the same) memory addresses. This means that the compiler cannot do some optimizations that involve changing the order of execution.

The code in `testfunc1.c` and `testfunc2.c` contains two variants of a function doing some simple computations using two input data buffers, and storing the result in a third buffer. This is an example of a case where the compiler could be able to optimize more if it was certain that the pointers did not point to the same memory regions (this is sometimes referred to as no "pointer aliasing").

We can tell the compiler that each pointer points to separate data by using the `restrict` keyword. To see the effect of this, add the `restrict` keyword to the pointer arguments of the `transform_opt` routine in `testfunc2.c`, like this:

```
void transform_opt (float * __restrict dest,
    const float * __restrict src,
    const float * __restrict params,
    int n) {
```

(Depending on the compiler version, the restrict keyword may be written simply as `restrict` or as `__restrict` with two underscore characters in the beginning).

After adding restrict, run the program again to see the performance effect.

Depending on which compiler and which compiler version you are using, `restrict` may have a larger effect when the input argument `n` to the `transform_opt` routine has a smaller value. To test this, try changing the `N1` and `N2` values in `main.c` so that `N1` becomes small, for example 20. Do a corresponding increase of `N2` so that you get a long enough runtime to give meaningful timings. Does the `restrict` keyword have a bigger effect when `n` is small?

Task 4

The code for this task is in the `Task-4` directory.

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

The code for this task is similar to the previous task, but now there is an extra argument `np` giving the length of the `params` list.

First, optimize by adding `__restrict` for the input pointers for `transform_opt` in `Task-4/testfuncs.c` in the same way as you did in the previous task. Does `__restrict` improve performance in this case?

Now assume that in practice we know what the `np` value will be when the program runs; we know that the `np` value will be 2. However, the compiler does not know this, so the compiler must generate code that can handle any `np` value.

We can try to optimize the code by using the `NP` value, defined two lines above the `transform_opt` routine:

```
int NP = 2
```

Change the j loop limit from `np` to `NP`, and run the code again. Did this improve the performance?

Now use the `const` keyword to tell the compiler that the NP value will never change:

```
const int NP = 2
```

Run the code again. Did you get any performance improvement from adding the `const` keyword?

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?

Task 5

The code for this task is in the `Task-5` directory.

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 main program in `main.c` creates a matrix `A` and then tries two different routines for transposing it: `do_transpose_standard` and `do_transpose_optimized`, implemented in `transpose.c`.

The matrix transpose implementation in `do_transpose_standard` is completely straightforward, while the `do_transpose_optimized` variant uses blocking to achieve a better memory access pattern.

Look at the code and try to understand how it works. Then run some tests and check if the performance is improved by the blocking implementation. Try different values of the `blockSz` constant in `do_transpose_optimized`.

You can change the size of the matrix by setting the constant `N` in the beginning of the main program. Hint: the benefits of blocking are usually greater for large

matrices, since for small matrix sizes most of the matrix may already fit in cache so that blocking has little or no effect.

Does the blocking give any improvement? What seems to be the optimal block size on the computer you are running on?

There is also an alternative code in `transpose_x.c` with the corresponding main program `main_x.c`, where the resulting matrix transpose is stored in several copies, so that the memory access pattern becomes even worse. This should make the benefits of cache blocking more clear.

The effect of blocking techniques, as well as the optimal block sizes and other parameters, depend a lot on the design of the cache(s) in the hardware. If the `do_transpose_optimized` blocked transpose implementation does not seem to give any improvement on the machine you are running on, try the code in `transpose_x.c` with the corresponding main program `main_x.c` instead.

Task 6

The code for this task is in the `Task-6` directory.

This task is about pure functions, see the section "Pure functions" on page 80 in the book by Agner Fog.

The main program has two loops, an outer loop over i and an inner loop over j. In the inner loop a function call `f(i)` is made. In this situation, if the compiler does not know what the function `f` is doing, it will have to generate code for calling `f(i)` each iteration in the iner loop, even though the argument `i` is the same each time.

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.

Another way of optimizing the code is to tell the compiler that the function is pure, so that the compiler can do the optimization. Change the main program back to its original form, and instead try to speed up the program by adding `__attribute__((const))` in the function declaration in the header file `func.h`:

```
int f(int k);
-->
int f(int k) __attribute__((const));
```

Are you able to speed up the program in this way?

Unfortunately, explicitly telling the compiler that a function is pure is currently only supported by the `gcc` compiler, so if you are using some other compiler you will not be able to do this optimization. The "Pure functions" section in the Fog book shows how the code can be written so that it compiles anyway, using a preprocessor macro.

Another way of allowing the compiler to do this optimization is to put the code of the function in side the same c source code file where the function is called. Try this: instead of doing the `__attribute__((const))` declaration, simply move the contents of `func.c` into `main.c`. Does this make the code faster?

Compare the "Function call counter" output from the program for the two cases: the `__attribute__((const))` variant and the solution of moving everything into `main.c`. Is the "Function call counter" output different? If so, can you understand why?

Task 7

The code for this task is in the `Task-7` directory.

In this task, we look at a small experiment related to virtual memory and *swapping*.

Almost all modern computers use a "virtual memory" system, which means that each program can run as if it had access to a very large memory space, even though in reality the physical memory is much more limited and is shared by all running processes. The virtual memory system achieves this by letting programs work with virtual memory addresses, which are automatically translated to physical addresses when the actual memory access happens.

Most of the time, the virtual memory system works very well and as a programmer you do not need to care about it. However, in case your program (together with other programs running at the same time) is using nearly all the computer's physical memory, then there are some interesting performance implications of how the virtual memory system works.

When all of the computer's physical memory is used, and some program is still trying to allocate more memory, the virtual memory system will do something called *swapping* — the data in parts of the physical memory will be written to disk (that disk space is often called swap space) in order to make some physical memory available. This disk access is typically very, very slow compared to the physical memory access, so when swapping occurs some memory access that is normally fast can become very, very slow.

In Linux, you can check the amount of physical memory and swap space available using the command "`free -h`".

The `alloc_all_mem.c` program in the `Task-7` directory performs an experiment to show the effect of swapping.

NOTE: since this is an experiment where we deliberately use all physical memory available, it will affect other programs running on the same computer. So if you run this on the lab computers or on some of the university's other computers, only do it once or a few times, to not disturb other users too much. If you use your own computer, you can of course play with this as much as you want.

Look at the code to figure out what it is doing, then run it and see what happens. Look at the difference between the `time_taken_min` and `time_taken_max` values

— that gives you an indication of how much slower memory access can become due to swapping. How many times slower does it get when swapping happens? 100 times? 1000 times? more?

Note that this test is designed to detect when memory access becomes very slow, and to stop when that happens. If it did not stop at that point, the swapping problems would get a lot worse, perhaps causing the computer to appear to "hang". If you use your own computer you can try that if you want, but please do not do it on the university computers.

The risk of swapping is one reason why memory leaks are very bad; memory leaks lead to larger memory usage and increses the risk that the physical memory will run out, leading to swapping, making everything very, very slow.