HIGH PERFORMANCE PROGRAMMING
UPPSALA UNIVERSITY
SPRING 2019
LAB 3: MEMORY USAGE AND INSTRUCTION-LEVEL PARALLELISM (ILP)

This lab is about how a program's memory usage affects the performance and optimizations related to instruction-level parallelism (ILP). ILP allows the computer to perform several instructions at the same time, which of course can be very good for performance. However, this only works if the program contains independent instructions; if each instruction depends on the result of the previous instruction, they cannot run in parallel. Therefore, optimizing a program to make use of ILP typically means rewriting the code to get more independent instructions.

We will be using `valgrind` — a suite of tools for debugging and profiling programs. It is an extremely useful tool that can be used in several ways. In this lab we will see how it can detect memory errors.

If you do not have valgrind on the computer you are using (or if valgrind does not work correctly there), login to another computer where valgrind is available, and check your code for memory errors there. University Linux hosts that you can login to: http://www.it.uu.se/datordrift/maskinpark/linux. Login remotely using ssh and copy files using scp (see `getting_started_with_linux` tutorial in Student Portal).

There are a couple of extra tasks in the lab. Look at them if you are done with other tasks and have more time. If you need all your time for the non-extra tasks, then don't worry about the extra tasks.

Start by downloading and unpacking the `Lab03_MemUsage_ILP.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.

1. Debugging with valgrind

Task 1:

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

In this task we look at how `valgrind` can be used to find memory errors in our programs.

Date: February 6, 2019.

In the `Task-1` directory you find a small code that does some kind of small computation. Use the makefile to build it, and try running it. Note that in order to profile a program with valgrind you first need to generate debugging information by compiling the code with the `-g` option. This debugging information is stored in the object file. It contains the data type of each variable or function and the correspondence between source line numbers and addresses in the executable code.

Usage of valgrind: `valgrind [options] prog-and-args`

Valgrind can be used in different ways, with different so-called *tools*. In order to specify which tool to use you pass to valgrind the corresponding option. The default behavior of valgrind is that it is running its so-called "memcheck" tool, meaning that it is trying to check the program for memory-access-related errors. To find more information about options for valgrind tools type `valgrind --help`

Run you program using valgrind:

```
valgrind ./prog
```

or

```
valgrind --tool=memcheck ./prog
```

Then valgrind will run our program. We will get the regular output from our program, plus additional output from valgrind. Look at the output. Every row has prefix like `==12345==`, where the number is an identifier of the working process. If it found no errors, valgrind should say "ERROR SUMMARY: 0 errors" on the last line of its output.

To see how the memcheck functionality works, now introduce a bug in the code: add a line of code where you try to access memory outside of what was allocated. For example, in `fun1()` you could add a line trying to modify something outside what was allocated for the vector `tmp`, e.g. "tmp[k]=0.2;". Such a bug is not detected by the compiler, and may lead to wrong results and/or crashing your code. Valgrind can help us detect such bugs.

Introduce such a bug, build the program again and then run it using valgrind. Does valgrind detect the error? Are you able to see from the valgrind error message on what line in the code the error appears? What happens if you run the program normally, without using valgrind?

You can also use valgrind to detect memory leaks. To see this, introduce a memory leak by removing one of the `free()` calls. Then run the program using valgrind, and look for the "LEAK SUMMARY" output. There you see that there was some memory leak, and it says "Rerun with –leak-check=full to see details of leaked memory". That means you can run it like this:

```
valgrind --leak-check=full ./prog
```

Then valgrind will tell you where the problematic allocation was made. Note that the allocation in itself is not necessarily a problem, the problem is just that it was never freed.

When there are no memory leaks, valgrind should give you the message "All heap blocks were freed – no leaks are possible". Make a habit of always checking your programs with valgrind, and verify that you get the “All heap blocks were freed” message!

Running a program with valgrind usually takes a lot longer time than running it in the normal way, because of all the checking and extra work that valgrind is doing. To see this, increase the amount of work done in the program so that it takes e.g. 1 second to run normally, and then check how long time it takes to run it using valgrind. Does it become 2 times slower? 5 times? more?

2. Memory usage

Task 2:

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

3. Instruction-level parallelism (ILP)

Background reading:

ILP works largely because of a feature in the CPU called *pipelining*. In a pipelined CPU, the execution of an instruction is separated into several stages. Here is one possible set of such stages:

(1) Instruction Fetch (IF) — Instruction code is retrieved from program
(2) Instruction Decode (ID) — Instruction is decoded
(3) Execute (EX) — Operations are executed
(4) Memory access (MEM) — Memory is read or written
(5) Register write back (WB) — Result is written to the output register

For our purposes here, it is not important to know exactly what the stages are and what they are doing (and the list above is anyway just an example). The important thing for us is to understand the idea of a pipeline, that the work needed to carry out each instruction is usually split into a set of stages.

The benefit of pipelining is that a different instruction can be present in each stage of the pipeline. With five stages, up to five instructions can be executed simultaneously. By the time the first instruction has finished, the second instruction is already at the WB stage, and a sixth instruction is already being fetched. The downside is that the minimal time to execute a single instruction is five clock cycles (one cycle per stage).

In a real processor nowadays, pipeline stages typically do not correspond exactly to the classical five stages above. At the height of the pipelining craze, the push for ever faster processor clocks had Intel designing CPU's with up to 31 stages! Today, pipelines are usually far shorter, with around 5-10 stages.

One reason for short modern pipelines is the problem of *hazards*, instructions that depend on each other. If one instruction uses the output of another, then they cannot follow directly after one another in the pipeline.

Example: say that our program has an "add" operation directly followed by a "move" operation. The add operation adds some numbers together and then, when it is done, places the result in a register R. The move operation is supposed to take the value in R and "move" (copy) it to a given memory address. This will not work properly if those two instructions were to be fed into the pipeline directly after eachother; the move operation would then try to get the value in the register R before the add operation had placed its result there. To make it work, the CPU would need to wait and not insert the move operation into the pipeline until it was certain the the result of the add was available.

Something that can be even more problematic for pipelining is *conditional branches.* A branch is a point in the program where execution is to be continued somewhere else, and a conditional branch is when the branch should be done or not depending on some condition. Thus, and if-statement in a C/C++ program will generally correspond to a "conditional branch" in the resulting machine code. When we have a loop that continues or stops depending on some condition, that will also give a conditional branch.

A conditional branch is very problematic for pipelining because the processor does not even know which instruction to fetch until the branch instruction is complete. Also in this case, it becomes impossible to keep the pipeline full. Some "empty space" or "no-operation" instructions must then be inserted into the pipeline instead of useful instructions. This is called a *bubble* in the pipeline. The longer the pipeline, the worse these problem become, and very long pipelines are essentially never filled by ordinary programs.

Compilers can try to generate machine code that minimizes these problems, and there are a few programming techniques that can help.

Task 5:

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

This task is about an optimization technique called *loop unrolling.* The idea of loop unrolling is to rewrite a loop so that the loop iterations are done in groups, like this:

```
for(i = 0; i < N; i++) {
  // do something related to i
}
-->
for(i = 0; i < N; i+=4) {
  // do something related to i
  // do something related to i+1
  // do something related to i+2
  // do something related to i+3
}
```

The idea is that the code can be faster this way because the instructions for i, i+1, i+2, i+3 can be done without interruptions for checking the loop condition.

Of course, it does not have to be 4 loop iterations that are grouped, it can be any number. This number is sometimes called the *unroll factor*. Note that if our code is written as above, we are assuming that N is divisible by the unroll factor. If N is not divisible by the unroll factor we can of course still use loop unrolling but the remaining iterations need to be handled separately, for example after the unrolled loop.

Loop unrolling is a kind of optimization that can often be done by the compiler, but there are also situations where you can benefit from doing it yourself.

Now, to try how loop unrolling works, start by looking at the code in the `Task-5` directory. The file `testfuncs.c` contains two routines, `f_std` and `f_opt`, which to begin with are identical. Your task is to change `f_opt` using loop unrolling, to see if you can make it faster in that way. Try your implementation and compare the run time to the unchanged `f_std` routine. Does loop unrolling make it faster? What seems to be a good choice of the unroll factor?

Remember to check correctness of the results! Make sure that you verify correctness also for cases when the number of loop iterations (`N1` in `main.c`) is not divisible by the unroll factor. For example, when N1 is 200, try the unroll factor 3 — make sure you get correct results also in that case.

To see if the compiler can do a better job, also try changing the compiler flags in the makefile. To ask the compiler to do loop unrolling, add the option `-funroll-loops`. You can also try changing from `-O2` to `-O3`. Do those changes make the `f_std` routine faster than your own loop-unrolled code?

Using "`man gcc`" you can read about the precise meaning of the compiler option `-funroll-loops` and compare it to the `-funroll-all-loops` option. Note that according to the man page, `-funroll-all-loops` often makes programs run more slowly.

Task 6:

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

In this task, we look at an example where it is difficult for the compiler to do loop unrolling, so it can be worthwhile do it manually.

Look at the code in `testfuncs.c`. As in the previous task, `testfuncs.c` contains two routines, `f_std` and `f_opt`, which to begin with are identical. This time, the loop is complicated by the fact that there is a variable called "counter" and an if-statement checking its value, so that in some iterations the value `x` changes.

Optimize the `f_opt` code using loop unrolling. *Hint: since the "counter" is reset every 4 iterations, it is probably beneficial to unroll the loop by 4. Can you even get rid of the if-statement?*

Try changing the compiler flags as in the previous task, to ask the compiler to do loop unrolling automatically. Does this make the code faster? If not, can you understand why it may be more difficult for the compiler to optimize the code in this case?

Task 7:

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

In this task you will try a technique called *loop fusion.* Loop fusion essentially works like this:

```
for(i = 0; i < N; i++) {
  // do "work A" related to i
}
for(i = 0; i < N; i++) {
  // do "work B" related to i
}
-->
for(i = 0; i < N; i++) {
  // do "work A" related to i
  // do "work B" related to i
}
```

As with loop unrolling, this gives a code where more instructions can be carried out without checking the loop condition in between.

Look at the code in `stencil.c`. There are three versions of the `apply_stencil` routine, labeled v1, v2, v3. To start with, try optimizing `apply_stencil_v2` using loop fusion. Does this make `apply_stencil_v2` faster than `apply_stencil_v1`?

Try changing the `STENCIL_SZ` value in `stencil.h` from 20 to 4. How does this affect the timings of `apply_stencil_v2` compared to `apply_stencil_v1`? Can you understand why changing `STENCIL_SZ` has this effect? *Hint: the compiler may do some optimizations automatically for very short loops, but not for longer loops.*

Extra part: The third version of the `apply_stencil` routine, `apply_stencil_v3`, is a blocked implementation. To get a scenario where blocking is more likely to help, you can change the `GridPt` structure in `stencil.h` by including the previously commented-out "`double v[5]`" line. This means that the data structure becomes larger (the `v[5]` part is not used, it just takes up space), so there are more problems with cache, so that the blocking implementation may be faster than the others. Now try improving `apply_stencil_v3` also using loop fusion, and see if the blocking makes it faster than `apply_stencil_v2`.

4. Extra tasks

Task 8:

The code for this task is in the `Task-8` 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-8` 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.

Task 9:

The code for this task is in the `Task-9` directory. The program for this task is doing some operations with an array. The array is initialised with random integers and a result is printed out. Your tasks is to find memory errors using `valgrind` and fix them.

Try to compile and run the `ex1.c` program. Do not forget to specify the `-g` option to the compiler.

The program may seem to run without problems but it contains some memory errors. Code with memory errors can be tricky to debug since it can appear to work for small tests but if applied to large problems or if routines are called many times, memory errors will likely cause the program to crash or to become extremely slow.

Run `valgrind` using

```
valgrind ./ex1
```

or

```
valgrind --tool=memcheck ./ex1
```

The memory errors are described after the line

```
==12345== Command: ./ex1
```

Here you should see lines similar to:

```
==12345== Invalid write of size 4
==12345==    at 0x400872: main (ex1.c:35)
==12345==  Address 0x520307c is 0 bytes after a block of size 60 alloc'd
==12345==    at 0x4C2DB8F: malloc (vg_replace_malloc.c:299)
==12345==    by 0x40081B: main (ex1.c:29)
```

This output indicates that there is no storage beyond the end of the array of 60 bytes, which means that program is trying to reach a location outside the bounds of the allocated memory. The `(ex1.c:35)` tells you the location in the code: line 35 in `ex1.c`. Valgrind also shows where the allocated memory buffer came from: memory was allocated using `malloc` in the main function (in `ex1.c` line 29).

Look also under LEAK SUMMARY.

If there is something definitely lost, that means the program has memory leaks.

Rerun with `--leak-check=full` to see details of leaked memory. If you fix all memory leaks you should see the following message:

```
All heap blocks were freed -- no leaks are possible
```

Next, uncomment the line with the call to `function66(N)` and again use valgrind to check for memory errors. Now you will see a few more lines for each error. This is because now there are errors inside some function calls and valgrind is showing the call stack which lead to each error, e.g. `main` called `function66` which in turn called `function77` where the error occurred.

Valgrind can detect when your program tries to access memory outside what was allocated, giving "invalid read" or "invalid write" messages. However, this does not mean that valgrind can detect all errors where a program is going outside of array bounds. To see this, use the example code `ex2.c`. Look at the code to see what it does: it works with a struct type A that includes an array `arr` followed by a long int `x`.

Compile the program and run it with valgrind to check for errors. Then introduce a bug by changing from `i < 8` to `i < 9` in the loop where `a->arr` is set. This means that the program is going beyond the array size. Compile and run it with valgrind. Is valgrind able to detect this error? What happens with the `a->x` output value? Can you understand why?

Then introduce the same bug for `b`, changing from `i < 8` to `i < 9` in the loop where `b->arr` is set. Is valgrind able to detect that error? Can you understand why?

Task 10:

The code for this task is in the `Task-10` 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.