High Performance Programming, Lecture 7

Optimization IV: Memory space and profiling

Today

- Assignments
- ILP examples
- Memory usage
- `valgrind` tools
- Surprise :)

Progress in Student Portal

Labs If you were at the lab or sent me a report, you should have status Completed.

Assignments – If everything is fine you have status Compileted.
– In some cases I set the status "Incomplete, under supplementation". **Check your e-mail!**

Assignments 3-6: final report for grading

Assignments 3-6: no supplementation. If you submit before the deadline, you will recieve a feedback, which you will use for improving/fixing your final report. (feedback on first-in/first-out basis)

After feedback on the assignment 6 you submit the final report and all needed codes **for grading**.
Your grade is determined as
40% grade of the group assignment + 60% grade of the individual project (so both parts are important!)

Supplementation deadline for Final report and Individual assignment: Approved projects can not be resubmitted for higher grades. Only if assignment failed, the new solution can be re-submitted before the supplementation deadline. In this case the maximum number of points is 3(pass).

Assignments: hints

Optimization flow:

- make sure your code works correctly! (use debuggers: `gdb` (lab 1, 2, lecture 5 (slide 7)), `valgrind memcheck` (lab 3))
- use compiler optimization flags (`-O3`, `-funroll-loops`, `-march=native` ...)
- try to optimize the code yourself (check if your optimization helps!)
  - optimized first the most time consuming parts! (you can use profiling tool like `gprof` (lab 4))

Assignment 3 optimization hints:

- remind the Newton's third law
- don't forget about the optimization flags
- avoid expensive operations and complex conditional branches (lectures 4,6, labs 2,3)
- optimize cache usage (accessing consecutive locations is fastest, lecture 5)

Debugger and profiler

Debugger is a computer program that is used to test and debug other programs.

`https://en.wikipedia.org/wiki/Debugger`

Profiler is a computer program that is used to identify performance problems in other programs without changing their code. (Includes analysis of particular instructions usage, frequency and duration of function calls, memory usage, etc.)

`https://en.wikipedia.org/wiki/Profiling_(computer_programming)`

Instruction level parallelism

Pipeline: instructions are divided in stages, which can be executed in parallel

Most common way to increase the ILP is to exploit parallelism among iterations of a loop: loop unrolling (lecture 6, lab 3), SIMD instructions (lecture 8).

In-order vs out-of-order execution

In-order processors:

- execute instructions sequentially; do not start to execute the next instruction until current instruction is completed.

Out-of-order processors:

- the order in which the instructions are actually executed is not the order in which they were supplied to the CPU
  - hide memory latency: independent ready instructions can execute before earlier instructions that are stalled
  - instruction speculation: executing an instruction before it is known that it should be executed

Branch prediction

Branch means that the next instruction may not the one on the next memory location. Processsor does not know which direction branch will take untill the branch statement is executed.

Conditional vs unconditional branches.

Branch mispredictions: flush the pipeline, roll back and restart

valgrind manual: in a modern machine, an L1 cache miss will typically cost around 10 cycles, an LL (last level) cache miss can cost as much as 200 cycles, and a mispredicted branch costs in the region of 10 to 30 cycles.

Experiments on my computer

Intel(R) Core(TM) i7-4600U CPU @ 2.10GHz
Ubuntu 18.04.1 LTS

L1d cache: 32K
L1i cache: 32K
L2 cache: 256K
L3 cache: 4096K

gcc 8.2.0
clang 6.0.0

Example code

Look at the code in `branch_prediction` folder.

How can we remove the branch?
Eliminate the branch and replace it with some bitwise operations (usually complicated and usually compiler will do it for you).

valgrind cachegrind

Cachegrind: a cache and branch-prediction profiler

- compile your code with `-g` flag
- `valgrind --tool=cachegrind --cache-sim=no --branch-sim=yes ./prog`
- You will see some output, also a `cachegrind.out.<pid>` file will be created
- Then use `cg_annotate cachegrind.out.<pid>` (numbers `<pid>` take from the previous cachegrind output)

Interpret the output:

Bc	conditional branches executed
Bcm	conditional branches mispredicted
Bi	indirect branches execute
Bim	indirect branches mispredicted

valgrind cachegrind

Output for the `branch_prediction` example from before with array size 10000 (compiled using `gcc -O3 -g sorting.c` and run as on previous slide):

Version 1 (unsorted array):

```
==12315== Branches:200,182,936 (200,172,546 cond + 10,390 ind)
==12315== Mispredicts:51,356,441( 51,356,270 cond +    171 ind)
==12315== Mispred rate:25.7% (       25.7%     +    1.6%   )
```

Version 2 (partitioned array):

```
==12318== Branches:200,182,946 (200,172,556 cond + 10,390 ind)
==12318== Mispredicts:26,426  (     26,255 cond +    171 ind)
==12318== Mispred rate:0.0% (        0.0%     +    1.6%   )
```

If you use `cg_annotate cachegrind.out.<pid>`, you can get a more detailed output for each function.

Compiler optimization

What often ends up being the fastest:

```
gcc -O3 -march=native -ffast-math
```

But:

- always try `-O2` (because `-O3` is not always better)
- always check that `-ffast-math` doesn't lead to unacceptably large errors

If possible: try another compiler (gcc, clang, Intel).

Example

Look at the code in `mmul` folder, test `mmul_loop_order.c`.
Compiled with gcc compiler and run with matrix size 500.

loop order	-O0	-O1	-O2	-O3	-O3 -march=native
mult_ijk	664.898	340.603	135.177	137.953	191.938
mult_ikj	507.716	90.221	84.842	51.791	43.745
mult_jik	618.859	341.089	133.290	132.135	193.585
mult_jki	699.036	324.984	332.679	328.016	331.401
mult_kij	511.692	98.943	96.233	66.858	58.530
mult_kji	733.099	333.376	333.635	333.307	331.054

`-ffast-math` did not have any effect

More examples: Lab 4

valgrind cachegrind

Cache misses are very expensive. The cache work most efficiently if pieces of data that are used together are stored near each other in memory.

Look at the code in `mmul` folder, test `mmul_loop_order_profiling.c`.

Run:

```
gcc -g -O3 -o mmul mmul_loop_order_profiling.c
valgrind --tool=cachegrind  ./mmul
cg_annotate cachegrind.out.<pid>
```

Interpret the output:

Ir	nr. of instructions executed	Dr	nr. of memory reads
I1mr	L1 instruction cache read misses	D1mr	L1 data cache read misses
ILmr	LL cache instruction read misses	DLmr	LL cache data read misses
Dw	nr. of memory writes		
D1mw	L1 data cache write misses		
DLmw	LL data cache write misses		

valgrind cachegrind

Run `mmul` with matrix size 600.

loop order	Ir	Dr	D1mr	DLmr
mult_kji	1,946,884,220	2 648,000,004	432,045,002	90,003
mult_jki	1,946,884,219	648,000,004	432,360,001	126,587
mult_jik	1,731,244,820	432,360,004	243,360,601	133,521
mult_ijk	1,730,885,422	432,360,004	243,404,403	128,931
mult_kij	874,449,638	217,084,807	27,405,602	90,003
mult_ikj	873,732,041	217,806,007	27,090,603	128,854

More about cachegrind:
`http://valgrind.org/docs/manual/cg-manual.html`
`https://accu.org/index.php/journals/1886`

`cg_annotate --auto=yes` annotates all source files containing functions. You may need larger monitor...
Visual graph: `kcachegrind cachegrind.out.<pid>`

Optimization overview

Program performance can be improved by:

- I: Doing less work
- II: Waiting less for data
- III: Doing the work faster
- IV: **Using less space (to fit a bigger problem)**

Memory space

Memory consumption can be an issue when:

- Doing things on embedded devices
- Solving large problems
- Observing reduced cache efficiency

The limit of code optimization

Memory footprint is the amount of memory needed by a program.

Fine code changes are much less effective at reducing memory footprints than improving speed.

First: improve your algorithmic *space complexity*

- Can you recalculate results instead of storing?
- Can you reorder calculations to save space?

Second: store only the things you really need to store.

Third: store things efficiently.

valgrind massif

Massif: a heap profiler.
It measures how much heap memory your program uses.

Read a short tutorial here:
`http://valgrind.org/docs/manual/ms-manual.html`
`https://accu.org/index.php/journals/1884`

Example: run your program with
`valgrind --tool=massif --time-unit=B ./prog`

All of Massif's profiling data is written to a file. By default, this file is called `massif.out.<pid>`, where `<pid>` is the process ID.

`-time-unit=B` option means that we want the time unit to instead be the number of bytes allocated/deallocated on the heap and stack(s). By default, Massif uses "instructions executed" as the unit of time.

valgrind massif

Massif: a heap profiler.

Use `ms_print` command to read created output file:

`ms_print massif.out.5821` (numbers you will see in the valgrind output: ex. ==5821==)

or

`ms_print massif.out.5821 > tmp.txt` (redirect the output to a file with name tmp.txt)

Note: by default Massif records a peak whose size is within 1% of the size of the true peak

`massif-visualizer` presents visual graph

Memory usage

See code in the `memory_usage` folder.

There are three versions of merge sort (in `sort_funcs.c`):

- (1) `merge_sort`, allocated buffers in each recursive call using malloc
- (2) `merge_sort_nofree`, allocated buffers in each recursive call using malloc, do not free them
- (3) `merge_sort_buffer`, only allocated buffers in the main function, re-use in recursive calls (no extra malloc calls)

valgrind massif

Massif: a heap profiler.

Compare massif output files for three versions (try to create your own output files):

- (1) `ms_print massif.out.version1`
- (2) `ms_print massif.out.version2`
- (3) `ms_print massif.out.version3`

How to interpret:

- bars consisting of ':' characters are normal memory snapshots (only basic info)
- bars consisting of '@' characters are detailed snapshots (with information about where allocations happened)
- bar consisting of '#' characters is a peak snapshot where the memory consumption was greatest

Storage alignment

Alignment: Variables of fundamental types like int, float, double etc are given (virtual) addresses that are a multiple of the size of the type. Compiler takes care of the alignment.

Alignment can allow hardware to work more efficiently, but a downside is that some memory space is wasted as padding.

Examples:

- use of intrinsic vectors requires alignment to addresses divisible by 16
- in some cases alignment of data structures to address divisible by the cache line size can improve performance

Alignment in structs

Data members in a struct are stored consecutively in the order in which they are declared.

The compiler inserts padding between members to ensure natural alignment if necessary.

By declaring members in order of decreasing size, padding is avoided.

Structs themselves are aligned according to the size of their largest member.

Alignment in structs

```
struct foo2 {
    short s;      // 2 bytes
    char c;       // 1 byte
};
```

In memory:

```
short s;            // 2 bytes
char c;       // 1 byte
padding       // 1 byte
```

Effectively same as:

```
struct foo2 {
    short s;      // 2 bytes
    char c;       // 1 byte
    char dummy;   // 1 byte
};
```

Alignment in structs

```
struct foo4 {
    char c1;            // 1 byte
    char* p1;           // 8 bytes
    char c2;            // 1 byte
    char* p2;           // 8 bytes
};
```

Alignment in structs

```
struct foo4 {
    char c1;            // 1 byte
    char* p1;           // 8 bytes
    char c2;            // 1 byte
    char* p2;           // 8 bytes
};
```

$\longrightarrow$ size 32 bytes. (7+7 bytes of padding)

We can improve this by simply reordering the members:

```
struct foo4 {
    char* p1;           // 8 bytes
    char* p2;           // 8 bytes
    char c1;            // 1 byte
    char c2;            // 1 byte
};
```

$\longrightarrow$ size 24 bytes. (still 6 bytes of padding in the end)

packed attribute in gcc

For gcc, the *packed* attribute can be used to remove natural alignment.

In some cases there is a performance cost. On the other hand, there may be a performance gain due to improved cache usage.

Drawback of *packed* attribute: not standard, so your code will be less portable (although most compilers have something similar)

packed attribute in gcc

```
struct foo4 {
    char c1;            // 1 byte
    char* p1;           // 8 bytes
    char c2;            // 1 byte
    char* p2;           // 8 bytes
};
```

$\longrightarrow$ size 32 bytes. (7+7 bytes of padding)

We can improve this by simply reordering the members:

```
struct foo4 {
    char c1;            // 1 byte
    char* p1;           // 8 bytes
    char c2;            // 1 byte
    char* p2;           // 8 bytes
} __attribute__((__packed__));
```

$\longrightarrow$ size 18 bytes. (no padding) **note: may be slower**

Always order structs: largest members first

A good and portable way of reducing the size of structs is to simply order the members inside the struct so that the largest members come first.

If the total size is a multiple of the largest member size, there is no padding if ordered in this way.

Always order structs: largest members first

```
struct foo {
  char c1;  // 1 byte
  int i;    // 4 bytes
  char c2;  // 1 byte
  char c3;  // 1 byte
  char c4;  // 1 byte
};
```

$\longrightarrow$ size 12 bytes. (3+1 bytes of padding)

Better to change order:

```
struct foo {
  int i;    // 4 bytes
  char c1;  // 1 byte
  char c2;  // 1 byte
  char c3;  // 1 byte
  char c4;  // 1 byte
};
```

$\longrightarrow$ size 8 bytes. (no padding)

Example

See `data_alignment/reading_file` example.

Does code works? Why? Can you fix the program by adding *packed* attribute to the structure?

Example

See `data_alignment/mmul_alignment` example.

In `mmul_struct_alignment.c` we store data into the structure:

```
typedef struct {
 double x;  // only this is used!
 int i;
 char c0;
 char c1;
 char c2;
} valStruct;
//} __attribute((packed)) valStruct;
```

Example

Compiled with gcc and run on matrix with size 500.

-O3 – standard mmul
-O3 no packed (16B) – use structure, no attribute
-O3 packed (15B) – use structure, with packed attribute

loop order	**-O3**	**-O3 no packed (16B)**	**-O3 packed (15B)**
mult_ijk	137.953	311.376	372.050
mult_ikj	51.791	107.641	113.794
mult_jik	132.135	180.998	181.790
mult_jki	328.016	401.947	639.855
mult_kij	66.858	126.896	133.669
mult_kji	333.307	412.621	650.706

Something strange (??)

Run on matrix with size 500.

-O3 – standard mmul

-O3 (str) – use structure storing only one double:

```
typedef struct {
 double x;
} valStruct;
```

loop order	clang -O3	gcc -O3	clang -O3 (str)	gcc -O3 (str)
mult_ijk	139.324	137.953	134.023	137.764
mult_ikj	47.553	51.791	47.161	41.717
mult_jik	131.048	132.135	138.687	136.589
mult_jki	334.745	328.016	335.610	182.155
mult_kij	67.641	66.858	65.598	60.312
mult_kji	332.995	333.307	334.654	196.298

(Compared with gcc 7.3, no such speed up observed.)

Want to learn more?

Advanced Computer Architecture course, 10 credits

"Really good" course!

Computer Architecture free online course:
`https://www.coursera.org/learn/comparch`

Exploiting out-of-order execution to steal data which is currently processed on the computer: Meltdown and Spectre
`https://meltdownattack.com/`

Extra links: for multi-threaded programs.

Set a watchpoint (stop execution whenever the value of an expression changes)
`https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/4/html/Debugging_with_gdb/stopping.html#SET-WATCHPOINTS`

gdb for multi-threaded programs
`https://sourceware.org/gdb/onlinedocs/gdb/Threads.html`
`https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/4/html/Debugging_with_gdb/threads.html`
`http://www.drdobbs.com/cpp/multithreaded-debugging-techniques/199200938?pgno=6`

Valgrind
`http://valgrind.org/docs/manual/manual-core.html#manual-core.pthreads`

20 Command Line Tools to Monitor Linux Performance
`https://www.tecmint.com/command-line-tools-to-monitor-linux-performance/`
In particular: top, htop and vmstat