High Performance Programming,
Lecture 7

Optimization IV: Memory space and profiling
Today

- Assignments
- ILP examples
- Memory usage
- valgrind tools
- Surprise :)}
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 Compiled.
– 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 receive 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.


**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.)

**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. Processor does not know which direction branch will take until 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
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:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bc</td>
<td>conditional branches executed</td>
</tr>
<tr>
<td>Bcm</td>
<td>conditional branches mispredicted</td>
</tr>
<tr>
<td>Bi</td>
<td>indirect branches execute</td>
</tr>
<tr>
<td>Bim</td>
<td>indirect branches mispredicted</td>
</tr>
</tbody>
</table>
**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.

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

-ffast-math did not have any effect

More examples: Lab 4
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:

<table>
<thead>
<tr>
<th>Ir</th>
<th>Dr</th>
<th>Lr</th>
<th>Dmr</th>
<th>Dm</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>nr. of instructions executed</td>
<td>nr. of memory reads</td>
<td></td>
</tr>
<tr>
<td>L1mr</td>
<td>D1mr</td>
<td>L1 instruction cache read misses</td>
<td>L1 data cache read misses</td>
<td></td>
</tr>
<tr>
<td>ILmr</td>
<td>DLmr</td>
<td>LL cache instruction read misses</td>
<td>LL cache data read misses</td>
<td></td>
</tr>
<tr>
<td>Dw</td>
<td></td>
<td>nr. of memory writes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D1mw</td>
<td></td>
<td>L1 data cache write misses</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DLmw</td>
<td></td>
<td>LL data cache write misses</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Run `mmul` with matrix size 600.

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

More about cachegrind:
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 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:
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

```c
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

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

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

→ size 32 bytes. (7+7 bytes of padding)

We can improve this by simply reordering the members:

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

→ 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

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

→ size 32 bytes. (7+7 bytes of padding)

We can improve this by simply reordering the members:

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

→ 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

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

→ size 12 bytes. (3+1 bytes of padding)

Better to change order:

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

→ 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:

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

```c
//} __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

<table>
<thead>
<tr>
<th>loop order</th>
<th>-O3</th>
<th>-O3 no packed (16B)</th>
<th>-O3 packed (15B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>mult_ijk</td>
<td>137.953</td>
<td>311.376</td>
<td>372.050</td>
</tr>
<tr>
<td>mult_ikj</td>
<td>51.791</td>
<td>107.641</td>
<td>113.794</td>
</tr>
<tr>
<td>mult_jik</td>
<td>132.135</td>
<td>180.998</td>
<td>181.790</td>
</tr>
<tr>
<td>mult_jki</td>
<td>328.016</td>
<td>401.947</td>
<td>639.855</td>
</tr>
<tr>
<td>mult_kij</td>
<td>66.858</td>
<td>126.896</td>
<td>133.669</td>
</tr>
<tr>
<td>mult_kji</td>
<td>333.307</td>
<td>412.621</td>
<td>650.706</td>
</tr>
</tbody>
</table>
**Something strange (??)**

Run on matrix with size 500.

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

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

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

(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)

gdb for multi-threaded programs
https://sourceware.org/gdb/onlinedocs/gdb/Threads.html

Valgrind

20 Command Line Tools to Monitor Linux Performance
In particular: top, htop and vmstat