High Performance Programming, Lecture 5

Optimization II: memory usage and some other things

Follow up on lab 2: Time measurement

```
time ./regularcode
              real     0m1.869s
              user     0m1.868s
              sys      0m0.000s
time ./sleepycode
              real     0m7.044s
              user     0m2.040s
              sys      0m0.000s
time ./mallocycode
              real     0m2.387s
              user     0m2.176s
              sys      0m0.208s
time ./threadedcode
              real     0m1.933s
              user     0m3.860s
              sys      0m0.000s
```

Intel Core i5-3210M CPU @ 2.50GHz, gcc 8.1.0

gdb debugger: stack frame, local addresses

Stack frame is the collection of all data on the stack associated with a function call

Corresponding registers:

Stack pointer (`%rsp`) stores the address of the top of stack

Frame pointer (`%rbp`) stores the address of the stack pointer when a function begins executing

`https://godbolt.org/z/-YxQq5`

gdb debugger: stack frame, local addresses

```
Breakpoint 2, gg (a=0) at littlecode.c:7
7       int p = 55;
(gdb) info locals                          # at the start
   of gg
p = 0
q = 0
r = 0
c = 0 '\000'
(gdb) c
Continuing.

Breakpoint 4, gg (a=0) at littlecode.c:17
17      return r;
(gdb) info locals                          # at the end
  of gg
p = 55
q = 66
r = 621
c = 100 'd'
```

gdb debugger: stack frame, local addresses

```
Breakpoint 1, hh (a=0) at littlecode.c:22
22      int s = 1;
(gdb) info locals
i = 55   # garbage value (leftovers from the gg
 call)
s = 66   # garbage value (leftovers from the gg
 call)
(gdb)
```

gdb debugger: ascii characters

```
(gdb) p c
$7 = 43 '+
      # or
(gdb) p c
$14 = 16 '\020
```

From the gdb manual:
Regard as an integer and print it as a character constant. This prints both the numerical value and its character representation. The character representation is replaced with the octal escape `'\nnn'` for characters outside the 7-bit ASCII range.

See ASCII table: `http://www.asciitable.com/`

More gdb debugger options

(gdb) continue (or just type letter c)
means proceed to the next breakpoint (or untill the end of the program if there are no breakpoints)

(gdb) next
execute just the next line of code
treat function calls as one instruction, do not step inside

(gdb) step
execute just the next line of code
step inside each function call

(gdb) finish
run until the current function is finished

(gdb) break myfunc
set a breakpoint
after running the program, the execution will stop in this function

(gdb) delete myfunc delete a specified breakpoint

Comment: typing “step” or “next” a lot of times can be tedious. If you just press ENTER, gdb will repeat the same command you just gave it.

Memory is one-dimensional

Any 2D, 3D, etc. array must somehow be mapped to 1D memory address space.

Example: matrix A (2D array $N \times N$), matrix element A_{ij} can be accessed as $A[i * N + j]$ (or as $A[j * N + i]$ depending on convention used).

Optimization overview

Program performance can be improved by:

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

Outline

Things covered in this lecture:

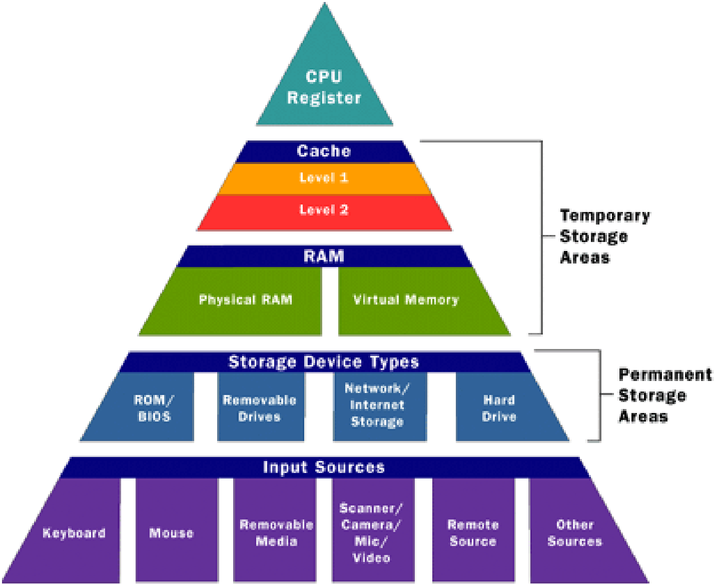
- Memory usage
- Cache, data locality, blocking techniques
- Function side effects, pure functions
- The "`const`" keyword
- The "`restrict`" keyword

Reading

Reading for this module:

- Fog 7.1 : “Different kinds of variable storage”
- Fog 8.3 : “Obstacles to optimization by compiler”
- Fog 9 : “Optimizing memory access”
- Pacheco 2.2.1: “The basics of caching”
- Pacheco 2.2.2: “Cache mappings”
- Pacheco 2.2.3: “Caches and programs: an example”

Example of memory hierachy



C language works with memory addresses

- The C language syntax lets us work with pointers to memory addresses.
- However, a C program does typically not explicitly decide what is stored in cache and in registers.
  - `register` keyword hints the compiler to use very fast memory if possible. i.e. `register int a;`
  - Usually doesn't make much difference. Compilers don't care any more.

Memory allocation Optimizations

Avoid too many small allocations

$\longrightarrow$ Avoid having many `malloc`/`free` calls for small memory buffers.

Ways to do this:

- Allocate few larger blocks and get smaller blocks by pointing into large blocks
- Use a pre-allocated work buffer instead of doing `malloc`/`free` each time a function is called
- Put small things on stack instead of calling `malloc`/`free`

Cache and data locality

Programs often re-use nearby memory locations (locality principle):

- Temporal locality: the same address is likely to be needed again soon
- Spatial locality: nearby addresses likely to be needed soon

Cache is a collection of memory locations that can be accessed in less time than some other memory locations.

This is why **CPU caches** are built into the hardware.

Hardware is complex

Example of info from AMD opt. manuel

In the "Software Optimization Guide for AMD Family 17h Processors" we can for example find info like this:

Pages 23-25: detailed info about L1, L2, L3 caches.

The AMD Family 17h processor uses five caches at three hierarchy levels to accelerate instruction execution and data processing:

- Dedicated L1 instruction cache
- Dedicated L1 data cache
- Dedicated L1 op cache
- Unified (instruction and data) L2 cache per core
- 4-Mbyte or 8-Mbyte L3 cache (depending on configuration)

Caches

Several levels of cache with varying latency

- L1 (approx. 10-100 kB) 1-5 cycles
- L2 (256 kB-) 10-20 cycles
- L3 ($>$ 2 MB), slower

Caches organized into units called "cache lines", often 64 or 128 bytes each: a whole cache line is fetched from memory.

L1 cache of two types: L1d (data) and L1i (instruction) cache.

Example

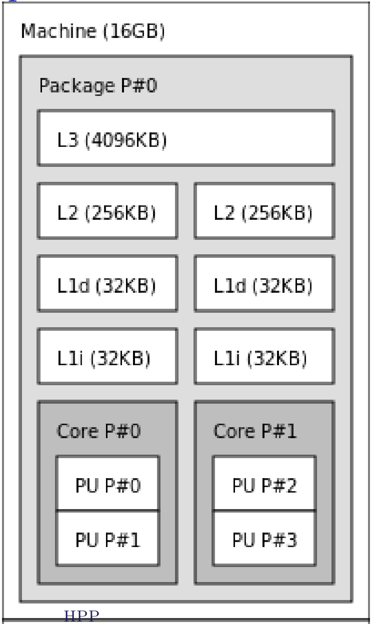
Intel Core i7-4600U CPU @ 2.10GHz

```
cat /proc/cpuinfo
```

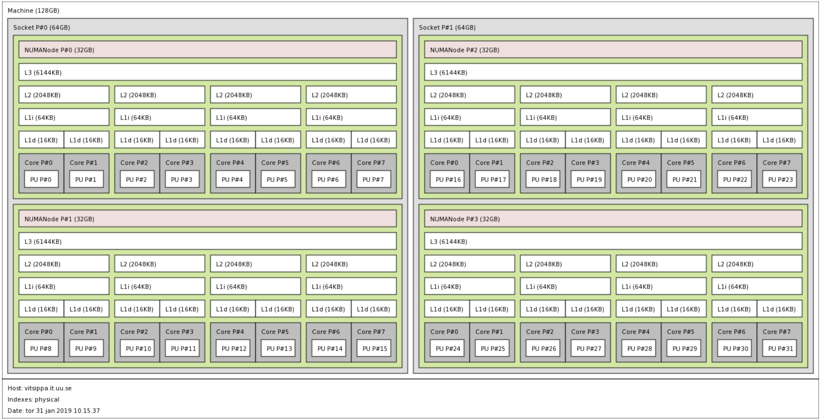
```
lscpu
```

To get this picture run:

```
lstopo --output-format
    png -v --no-io > cpu
    .png
```



Example: computer at vitsippa.it.uu.se



Example of costs for cache/memory access

To where	Cycles
Register	≤ 1
L1d	3
L2	14
Main Memory	240

(Numbers listed by Intel for a Pentium M)

How caches work – cache lines

- Cache divided into cache lines
- When memory is read into cache, a whole cache line is always read at the same time
- Good if we have data locality: nearby memory accesses will be fast
- Typical cache line size: 64-128 bytes

So for example, if using double precision, `sizeof(double) = 8` means that 16 `double` numbers will fit in the same cache line if the cache line size is 128 bytes.

How do caches work?

"Direct-mapped cache":
– Hash function maps to a single place for every unique address (multiple sets with a single cache line per set)

"Set associative cache":
– Cache space is divided into n sets – Address is distributed modulo n – k-way set associative ($k = 2...24$)

"Fully associative cache":
– Maps to any slot (single cache set with multiple cache lines)

In practice, k-way set associative commonly used, e.g. L1d cache commonly 2-way set associative.

How do caches work?

Example

Example: 2-way set associative cache, with 4 sets. Sets A, B, C, D.

Each memory address belongs to one of the 4 sets, this is fixed.

Either of the two "ways" can be used (slot1 or slot2).

```
Memory: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx...
   Set: ABCDABCDABCDABCDABCDABCDABCDABCD...

Cache:      Set A : slot1 slot2
            Set B : slot1 slot2
            Set C : slot1 slot2
            Set D : slot1 slot2
```

General caching concepts

- "Cache hit": memory address already in cache — good!
- "Cache miss:"
  - Compulsory: On the first access to a block. (also called cold start misses, first reference misses).
  - Capacity: When blocks are being discarded from cache because cache cannot contain all blocks.
  - Conflict: In set associative or direct mapped strategies, conflict misses occur when several blocks are mapped to the same set or block frame; also called collision misses or interference misses.
- "Hit rate" and "miss rate"

Cache optimization – how?

Try to increase the chances that the needed memory locations are already in cache:

Nearby memory accesses in time should be to nearby locations in memory

Caches and performance

- Caches are extremely important for performance
- Level 1 latency is usually only 1-3 cycles
- Work well for problems with nice locality properties
- Caching can be used in other areas as well, example: web-caching (proxies)
- Modern CPUs have two or three levels of cache
- In the hardware, most of the chip area is used for caches

Cache optimization example

Loop order for matrix-matrix multiplication

```
for (i=0; i<n; i++)
  for (j=0; j<n; j++)
    for (k=0; k<n; k++)
      c[i][j] += a[i][k] * b[k][j];

for (k=0; k<n; k++)
  for (j=0; j<n; j++)
    for (i=0; i<n; i++)
      c[i][j] += a[i][k] * b[k][j];

for (k=0; k<n; k++)
  for (i=0; i<n; i++)
    for (j=0; j<n; j++)
      c[i][j] += a[i][k] * b[k][j];
```

Test problem, matrices of size 300

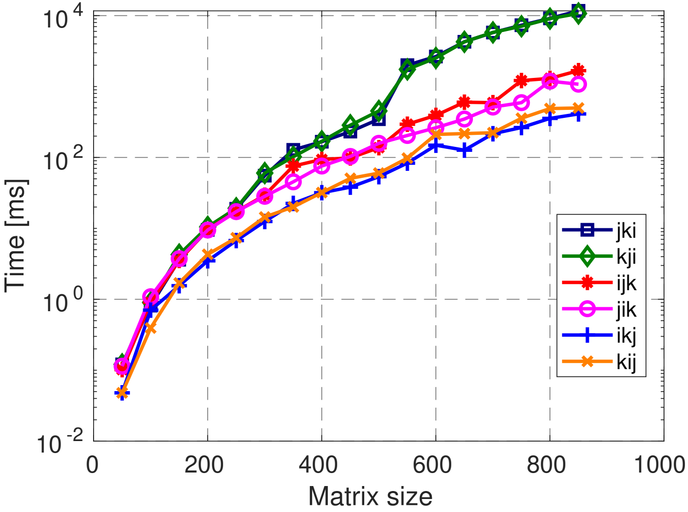
CPU used: AMD Opteron 6220 “Bulldozer” @ 3.0 GHz (Tintin cluster at UPPMAX). Compiled using `-O3`.

Timings in milli-seconds:

	Loop order					
Compiler	ijk	ikj	jik	jki	kij	kji
gcc 4.4.7 (2012)	62	34	64	128	37	128
gcc 4.8.3 (2013)	61	29	63	128	31	128
gcc 5.3.0 (2015)	61	19	60	128	20	127
gcc 5.3.0 `-march=native`	65	**13**	66	128	**16**	127
Intel icc 13.0.1 (2012)	49	19	61	129	21	129
Intel icc 16.0.2 (2016)	50	**14**	61	131	**17**	131

Test code `mmul_loop_order.c` available in Student Portal.
Try it!

Test problem, matrices of size 50:50:850



CPU info: Intel(R) Core(TM) i7-4600U CPU @ 2.10GHz L1d cache: 32K L1i cache: 32K L2 cache: 256K L3 cache: 4096K gcc 8.2.0

Cache optimizations

Improve data locality!

Example: blocking

Optimizations can have different effects depending on e.g. prefetching capabilities in the hardware.

Function side effects are bad

When a function modifies something other than its output value or output arguments, the function has side effects.

Examples: modification of global variables or writing to files.

Function side effects are:

- **bad** for flexibility
- **bad** for performance

Function side effects are bad

What to do with global variables

When you need global variables Don't use global variables. Avoid it as much as possible.

Ideas:

- Pass as input argument.
- Gather in a structure.

More on function side effects

What does the compiler know?

```
int func1(int x) {
  return f(x)+f(x)+f(x)+f(x);
}
```

```
int func2(int x) {
  return 4 * f(x);
}
```

Are these two equivalent?

Do you know? Does the compiler know?

Side effects in loops

What does the compiler know?

```
char *str = "Hello World!";
int i;
for (i = 0; i < strlen(str); i++)
{
  if (str[i] == '!') str[i] = '?';
}
```

What is the complexity?
Can you improve it?
Can the compiler improve it?
Does the compiler know the implementation of `strlen(...)`?
Check `https://godbolt.org/z/UTHeiS`

Pure functions

`strlen()` is a pure function while `rand()` is an impure one

Can use gcc `__attribute__((pure))` in function declaration to tell the compiler that a function is pure (free from side effects):

```
int f(int k);
```

$\longrightarrow$

```
int f(int k) __attribute__((pure));
```

- `__attribute__((const))` also exists and is more strict; then the function is not allowed to use pointer arguments. See gcc documentation for details.

Pure functions

```
int f(int x) __attribute__((pure));

int func1(int x)
{
  return f(x)+f(x)+f(x)+f(x);
        }
```

Check at `https://godbolt.org/z/h3M0R0`

Pure functions, example

See `text`

`https://godbolt.org/z/bhYLYw`

- Try to specify the function f as pure using `__attribute__((const))`
- Try instead of doing the `__attribute__((const))` declaration, simply move the contents of func.c into main.c.

The "const" keyword

Letting the compiler know more

Use the "const" keyword to tell the compiler that the value will never change.

Allows compiler to optimize more because the compiler knows more.

Using "const" also helps avoid programming errors, like modifying something by mistake.

$\longrightarrow$ Using "const" is both **good for program design, and good for performance!**

The "const" keyword

Example

```
int M = 50;
```

Variable not declared as const $\longrightarrow$ compiler must assume that the value may change.

```
const int M = 50;
```

Variable declared as const $\longrightarrow$ compiler knows value will not change. Compiler may be able to optimize more thanks to this info.

Pointer aliasing

Any pointer of unknown origin can reference a value that is accessed through another variable

Any pointer might be used as an array – of unknown size

Multiple "aliases" for the same memory location

Makes compile-time optimization very hard

$\longrightarrow$ need ways to inform compiler that aliasing does not occur.

The "strict aliasing" rule

Default mode in C99 and recent GCC

Pointers of different types are not allowed to refer to the same memory

$\longrightarrow$ Significant compilation benefits

The "restrict" keyword

Available in many C/C++ compilers, including recent gcc (sometimes as `__restrict`)

Within this context, any memory locations accessed by a restricted (pointer) variable will only be accessed through that pointer

Example:

```
void copyString(char * __restrict dest,
                char * __restrict from) { ... }
```

The `restrict` keyword can have different effect depending on compiler version.

The “restrict” keyword

Example

```c
void f(double * v, int N, double* x)
{
  int i;
  for(i = 0; i < N; i++)
    v[i] += ( A + B*x[0] + C*x[1] ) * D;
}
```

Performance problem due to pointer aliasing?

The "restrict" keyword

Example

```
void f(double * v, int N,
       double* x) {
  int i;
  for(i = 0; i < N; i++)
    v[i] += ( A + B*x[0] + C*x[1] ) * D;
}
```

Optimize by adding "restrict" keyword:

```
void f(double * __restrict v, int N,
       double * __restrict x) {
  int i;
  for(i = 0; i < N; i++)
    v[i] += ( A + B*x[0] + C*x[1] ) * D;
}
```

No change in function body, only added `__restrict` in argument

The “restrict” keyword

Example timings

The `restrict` keyword can have different effect depending on compiler version.
Code on previous slide, compiled with `-O3` for different gcc versions:

Compiler	Standard	Optimized
gcc 4.4.7	1.58	1.19
gcc 4.8.3	1.07	0.46
gcc 5.3.0	0.52	0.43

CPU used: AMD Opteron 6220 “Bulldozer” @ 3.0 GHz (Tintin cluster at UPPMAX).

Preparation for the lab on Thursday

We will use `valgrind` tool: `http://valgrind.org/`

If you are using your personal computer, check if it exists.

That's all

Questions?