Code Optimisation

Advanced Programming Tutorials

Erik Wannerberg
Supervisors: Roland Wittmann, Philipp Neumann

January 13, 2016

Content

1. Contents

2. Repetition: Slide Karaoke...

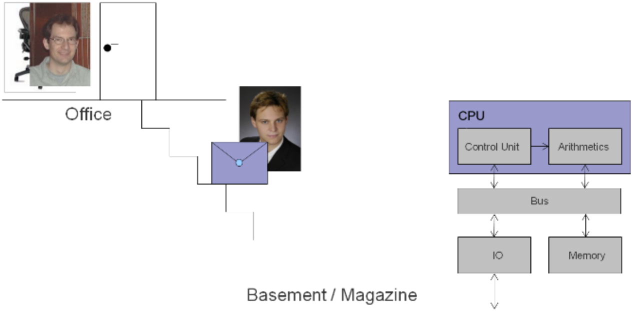
3. Code optimisation tools
   - 3.2 gprof
   - 3.3 Valgrind, Kcachegrind
   - 3.4 vampir

4. Hints and tricks

Repetition: Slide Karaoke...

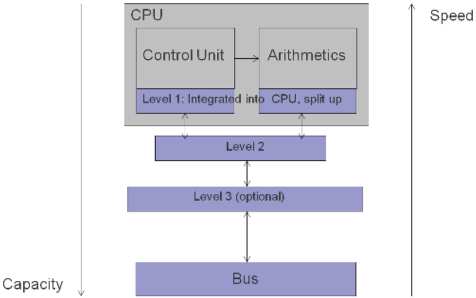
Hold your breath for what comes next...

Accessing the Memory: The Neumann Bottleneck



- Running into the basement is time consuming, and
- The bigger the basement (memory), the slower the search becomes.
- The faster the processor, the more annoying the slow search in the memory is.
- Can we study this effect?

Cache Levels

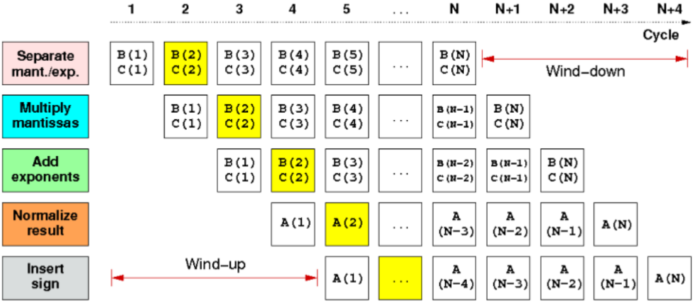


Example: My Notebook (Dual-Core IvyBridge)

- L1: 32 kB
- L2: 256 kB
- L3: 4 MB

- Computers have a hierarchy of caches and lots of registers.
- The time to finish one operation depends significantly on where the data is located right now.
- It is always whole cache lines that are read (e.g., 64 bytes= 8 doubles)
- It is important for many algorithms to exhibit spatial locality and temporal locality.

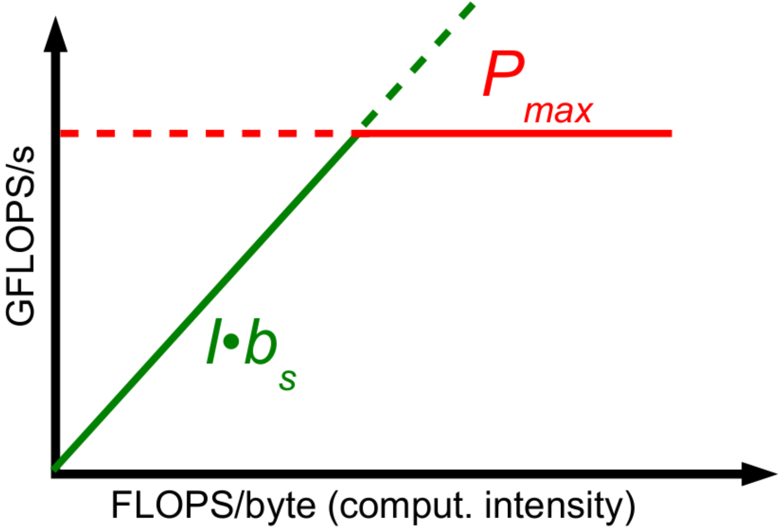
Pipelining (2)



Wellein/Hager: Node Level Performance Engineering, Lugano 2014

- Avg. execution time/ multiplication: $\approx$ 1 cycle (for large N)
- Challenge for programmer: exploit this, i.e. write programme such that the pipeline is always full!
  $\rightarrow$ help: out-of-order execution
  $\rightarrow$ help the compiler and (potentially) check what it does (assembly code etc)

The Roofline Model (2)



The Roofline Model (1)

- Two major upper bounds for performance
  1. Applicable peak performance (assuming data in L1 cache) P_{max} [Op/s]
  2. Memory throughput; define
     - Computational intensity I [Op/Byte]: work per byte (transferred via slowest data path)
     - Applicable peak bandwidth b_s [Byte/s]

Roofline model principles, cont...

What is performance P?

$\Longrightarrow$ We like to calculate, want many calculations fast!
([Floating Point] Ops/s)

What is the limiting factor?

- ..is it the theoretical highest speed of the processor P_{max} [Ops/s]?
- ..or is it the speed that we can load and store data that the processor works on?

...determined by how many operations we can do with the loaded data I [ops/byte], multiplied by rate/bandwidth of data delivery b_s [bytes/s]

$$\boxed{P = \min(I \cdot b_s, P_{max})}$$

Some inspirational quotes...

premature optimization is the root of all evil.

– Donald Knuth

Some inspirational quotes...

"Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. ***We should forget about small efficiencies, say about 97% of the time:*** *premature optimization is the root of all evil.* ***Yet we should not pass up our opportunities in that critical 3%.****"* – Donald Knuth

"*90% of the execution time of a program is spent executing only 10% of the code.*" – Software Engineering Pareto Principle (Wikipedia)

Some inspirational quotes...

"Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%." – Donald Knuth

"*90% of the execution time of a program is spent executing only 10% of the code.*" – Software Engineering Pareto Principle (Wikipedia)

Do optimize – but only where it makes sense!

gprof

Program *profiling* tool.

- compile using `-p` or `-pg`
- run program $\Longrightarrow$ profiling information in `gmon.out`
- run `gprof <program>`

Generates *call graph* with times of each function call, and how big part of that time is in the function and called inner functions respectively.

Export to directed graph-supporting `.dot` using for example `gprof2dot`, and view using `xdot`.

Valgrind, Kcachegrind

Valgrind has multiple tools for optimization:

- `callgrind` – produces call graph similar to `gprof`, with source code line information
- `cachegrind` – produces information on cache usage

Invocation:

- `valgrind --tool=<tool name> <program name>`
- program compiled with `-g` for source code information
- options include tracking of conditional branches `--branch-sim=yes` etc...

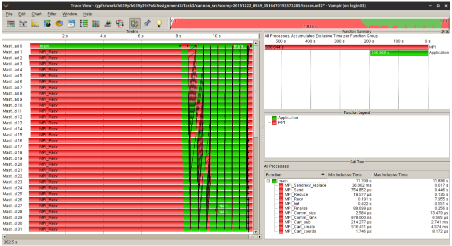
Visualisation using for example `kcachegrind`.

vampir

Analysis tool for parallel applications.

- multiple processes side-by-side
- clear illustration of interprocess communication and synchronisation
- works with other parallel performance analysis framworks like SCOREP

(useful for example for programming on the SuperMUC, i.e. in Programming of Supercomputers, 3rd semester CSE!)



Hints and tricks

- Can your compiler do some work for you?
    - Optimization flags (`-O3` combines many, but long list of other compiler-specific ones! see `g++ --help=optimizers`)
    - Help vectorisation and out-of-order execution by eliminating dependencies (later)
- Have you choosen the right algorithm? (trying to multiply by 1000000 will never be fast if implemented by a million additions...)
- Spatial and temporal locality – use your caches!
- Reducing branches (`ifs`) – especially in loops (taking the wrong branch is expensive..)
- Allocate loop–independent variables *once* outside the loop
- ...but most of all, *only optimize the things that take time*!!

Links

Roofline model:
http://crd.lbl.gov/assets/pubs_presos/parlab08-roofline-talk.pdf
Quotes by Donald Knuth: https://en.wikiquote.org/wiki/Donald_Knuth
Friendly tutorial on gprof: ⌣̈
http://www.thegeekstuff.com/2012/08/gprof-tutorial/
Unix manual on gprof: http://www.unix.com/man-page/FreeBSD/1/gprof/
gprof2dot on github: https://github.com/jrfonseca/gprof2dot
Valgrind: http://valgrind.org
Kcachegrind: http://kcachegrind.sourceforge.net
Vampir: https://www.vampir.eu