Session 4: Performance measurements
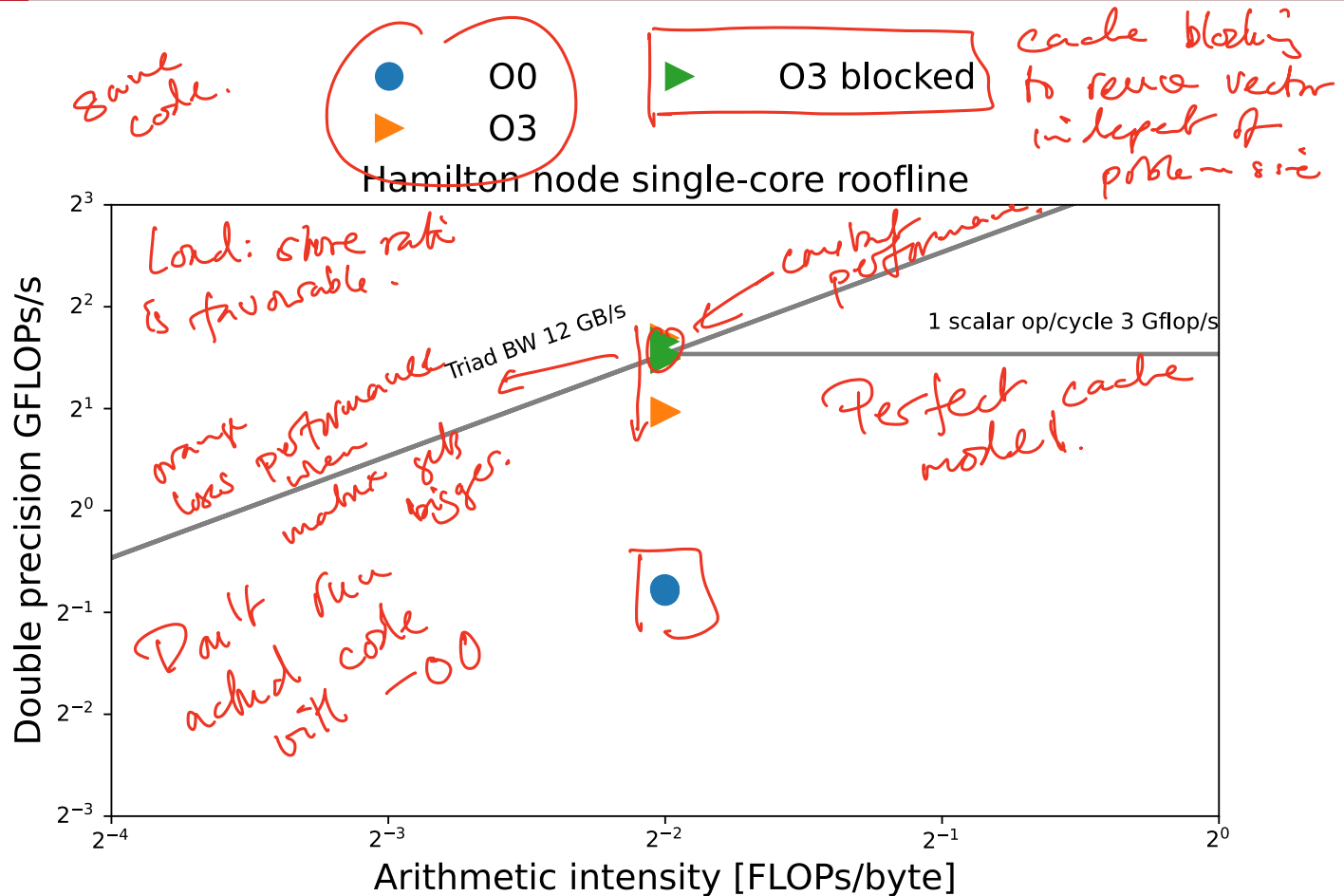
COMP52315: performance engineering

Lawrence Mitchell[*]

[*]`lawrence.mitchell@durham.ac.uk`

Hamilton node single-core roofline

Handwritten annotations on figure:

- Same code.
- cache blocking to reuse vector independent of problem size
- O0
- O3
- O3 blocked
- Load: store ratio is favorable.
- constant performance
- 1 scalar op/cycle 3 Gflop/s
- Triad BW 12 GB/s
- orange loses performance when matrix gets bigger.
- Perfect cache model.
- Don't run actual code with -O0

Double precision GFLOPs/s

Arithmetic intensity [FLOPs/byte]

- Roofline gives us a high-level overview of what to try next.
- How to drill down and get more information about what is causing the bottleneck?
- How to confirm the hypothesis formed through the roofline analysis?
- ⇒ *Measure* things about the code.

Performance measurements

- Modern hardware comes with some special purpose *registers* that you can prod to measure low level performance events.
- Can use this to characterise performance of a piece of code

Caveats

- Measurements can only tell you about the algorithm you're using
- e.g. Counts the data you moved, not the data you could have moved.
- Do not tell you about potential better algorithms
- Need to work hand in hand with models.

Recognise/exploit high level structure in the problem.
⟹ this comes from the maths.

Since 1960.

10^{21} increase in size of
PDE problems we can
solve.

~ 10^{9} has come from hardware
10^{12} has come from better algorithms

- An almost overwhelming number of different things like:
    - Number of floating point instructions of different type (scalar, sse, avx)
    - Cache miss/hit counts at various levels
    - Branch prediction success rate
    - …

$\Rightarrow$ Best used to confirm hypothesis from some model

Abstract metrics

- Can read low-level hardware counters directly (e.g. how many floating point instructions were executed?)
- More useful to group into abstract metrices
- $\Rightarrow$ easier to compare across hardware, easier to interpret.
- For example, measure "Instructions per cycle" rather than instructions.

$\rightarrow$ independent of problem size (we hope)

$\rightarrow$ machine portable

$\rightarrow$ on a given machine "useful #".

prods kernel-level stuff
IO needs to be installed as root.

- Use `likwid-perfctr` (installed on Hamilton via the `likwid` module).
- Offers a reasonably friendly command-line interface.
- Provides access both to counters directly, and many useful predefined "groups".

module load likwid/5.0.1

- Will use `likwid-perfctr` to measure memory references in different implementations of the same loop.

(handwritten) $c[i] = a[i] + a[i] * b[i]$

(handwritten) pseudo assembly for

Scalar

```
for i from 0 to n:
  load a[i:i+1] reg1
  load b[i:i+1] reg2
  load c[i:i+1] reg4
  mul reg1 reg2 reg3
  add reg4 reg3 reg4
  store reg4 c[i:i+1]
```
(handwritten) 3 regs i-egistes

(handwritten) 3×10^6 loads
10^6 stores

SSE

```
for i from 0 to n by 2:
  vload a[i:i+2] vreg1
  vload b[i:i+2] vreg2
  vload c[i:i+2] vreg4
  vmul vreg1 vreg2 vreg3
  vadd reg4 reg3 reg4
  vstore reg4 c[i:i+2]
```
(handwritten) 1.5×10^6
5×10^5

AVX

```
for i from 0 to n by 4:
  vload a[i:i+4] vreg1
  vload b[i:i+4] vreg2
  vload c[i:i+4] vreg4
  vmul vreg1 vreg2 vreg3
  vadd reg4 reg3 reg4
  vstore reg4 c[i:i+4]
```
(handwritten) 7.5×10^5
2.5×10^5

AVX2

```
for i from 0 to n by 4:
  vload a[i:i+4] vreg1
  vload b[i:i+4] vreg2
  vload c[i:i+4] vreg3
  vfma vreg1 vreg2 vreg3
  vstore reg3 c[i:i+4]
```

Model

For each loop choice, if we choose $n = 10^6$, how many load and store instructions do we expect to measure?

Measurement

Model

For each loop choice, if we choose $n = 10^6$, how many load and store instructions do we expect to measure?

Answer

Each loop iteration has 3 loads and 1 store.

Vector width v and n iterations we need $\frac{3n}{v}$ loads and $\frac{n}{v}$ stores.

$\Rightarrow$ let's attempt to verify this with measurements.

- Goal is to convince ourselves that measurement works!
- $\Rightarrow$ Exercise 5 from the usual place.

Larger code

Problem

What if you don't know which part of the code takes all the time?

Answer

Use *profiling* to determine hotspots (regions of code where all the time is spent).

$\Rightarrow$ allows us to focus in on important parts.

Profiling: types

- Goal is to gather information about what a code is doing
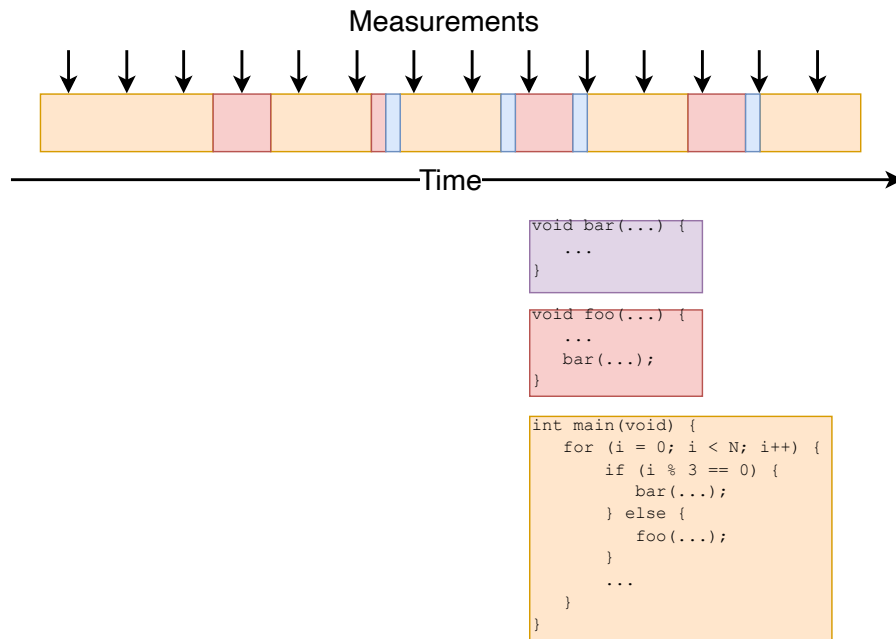 - *Sampling*
  - or *code instrumentation*

Sampling

- Works with unmodified executables
- Only a statistical model of code execution
- ⇒ not very detailed for volatile metrics
- ⇒ needs long-running application

Instrumentation

- Requires source code annotations to capture "interesting" information
- Much more details and focused
- ⇒ Preprocessing of source required
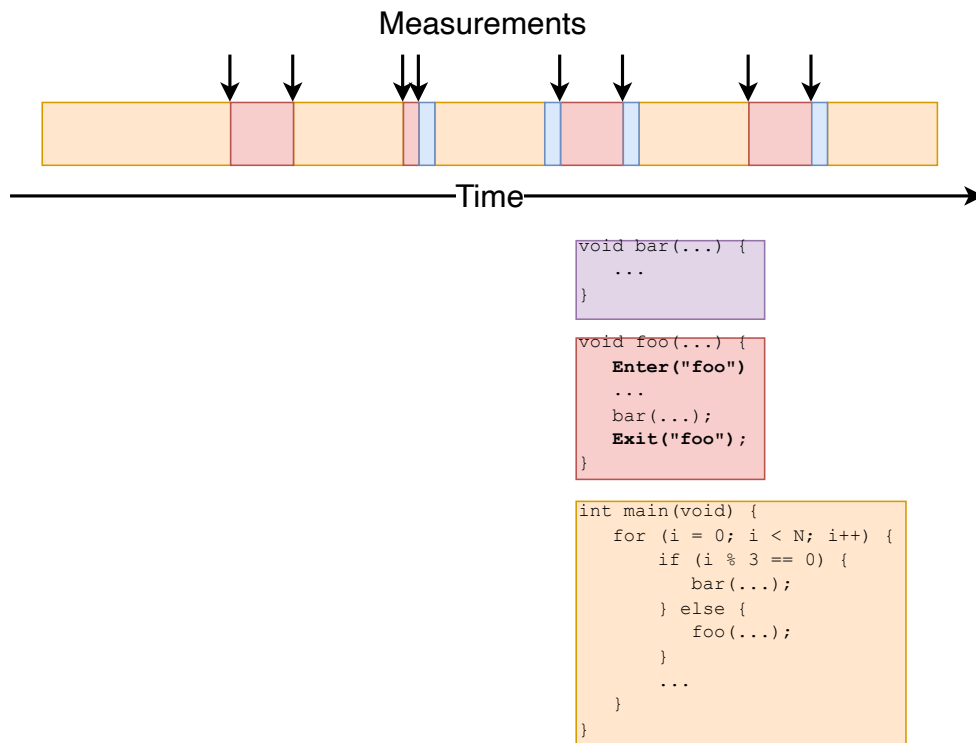- ⇒ Can have large *overheads* for small functions.

Sampling

- Running program is periodically interrupted to take a measurement.
- Records which function we are in.

Tracing

- Measurement code is inserted to capture all the events we care about

Measurements

Time

```
void bar(...) {
    ...
}
```

```
void foo(...) {
    Enter("foo")
    ...
    bar(...);
    Exit("foo");
}
```

```
int main(void) {
    for (i = 0; i < N; i++) {
        if (i % 3 == 0) {
            bar(...);
        } else {
            foo(...);
        }
        ...
    }
}
```

Sampling profiles with gprof

Workflow

1. Compile *and link* code with symbols (add `-g`) and profile information (`-p`).

2. Run code ⇒ produces file `gmon.out`

3. Postprocess data with `gprof`

4. Look at results

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
76.14      5.71      5.71      102     0.06     0.06  ForceLJ::compute(Atom&, Neighb
17.07      6.99      1.28        6     0.21     0.22  Neighbor::build(Atom&)
 2.80      7.20      0.21        3     0.07     0.07  void ForceLJ::compute_halfnei
 1.47      7.31      0.11        1     0.11     7.05  Integrate::run(Atom&, Force*,
 0.93      7.38      0.07                              __intel_avx_rep_memcpy
 0.40      7.41      0.03       11     0.00     0.00  Neighbor::binatoms(Atom&, int
 0.40      7.44      0.03        6     0.01     0.01  Comm::borders(Atom&)
 0.40      7.47      0.03        1     0.03     0.04  create_atoms(Atom&, int, int,
 0.13      7.48      0.01   285585     0.00     0.00  Atom::unpack_border(int, doub
```

- Code is instrumented (instructions inserted so we know which function we're in), triggering of measurement is sampling based (not every call).
- GProf provides profile with some tracing information
- Gives both *inclusive* and *exclusive* timings.

- Blue box shows "inclusive" time for `main`
- `foo` and `bar` calls (orange) excluded for "exclusive" time.

⇒ exclusive time measures execution in function that is not attributable to some other function.

```
int main(void) {
    for (i = 0; i < N; i++) {
        if (i % 3 == 0) {
            bar(...);
        } else {
            foo(...);
        }
        ...
    }
}
```

- After we have identified the hotspot that takes all the time, we'd like to determine if it is optimised

$\Rightarrow$ need more detailed insights.

1. Find relevant bit of code
2. Determine algorithm
3. Add instrumentation markers (see exercise)
4. Profile with more detail/use performance models.

$\Rightarrow$ guidance for appropriate optimisation.

Exercise: finding the hotspot

- So far, we've looked at very simple code. Now, your task will be to find the hotspot and do some exploration in a larger one.

⇒ Exercise 6 from the usual place.