Very simple things we've seen.
→ work out details

:(⇒ not time for real code.

~ Williams et al.
2009.

Session 3: Roofline models

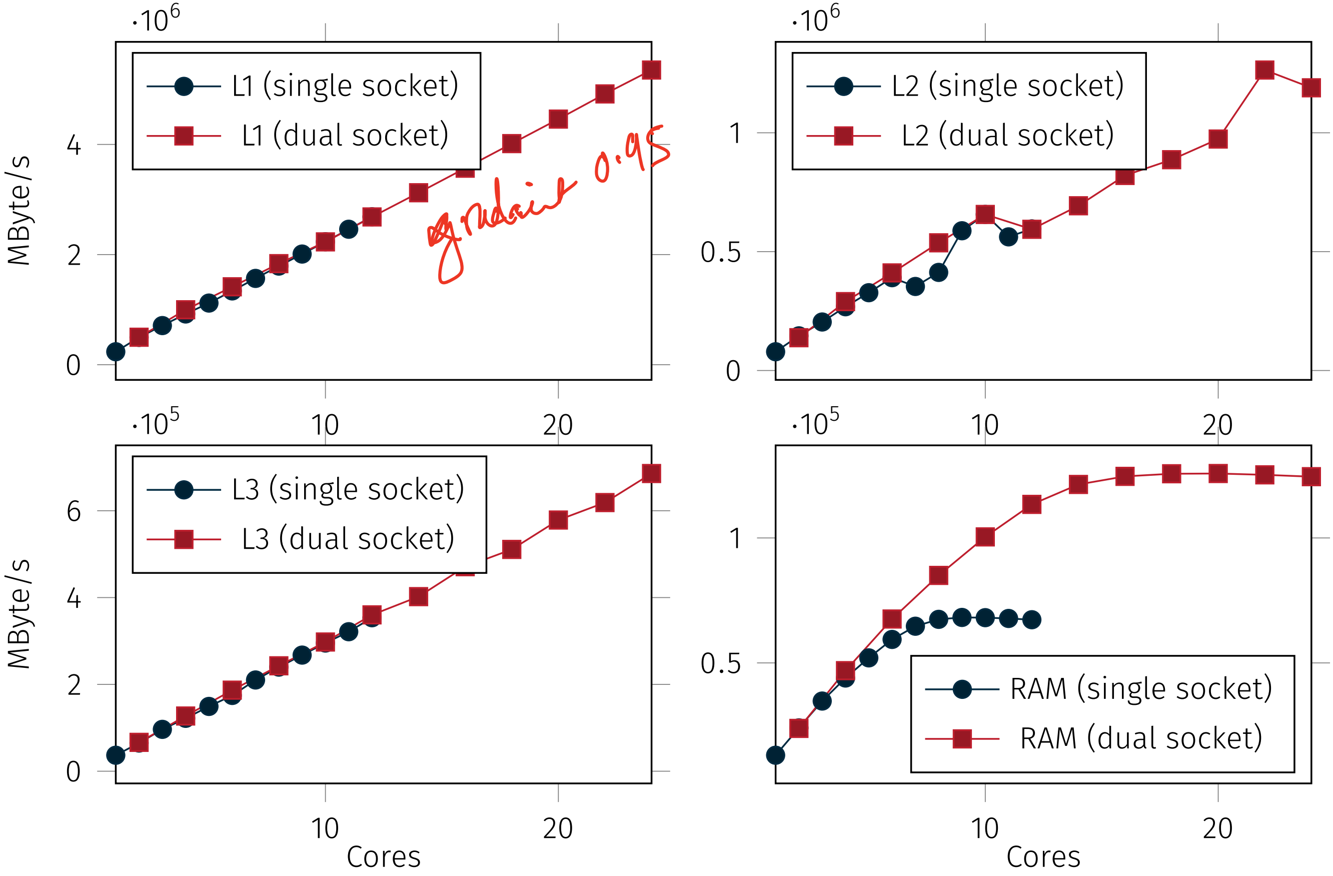
COMP52315: performance engineering

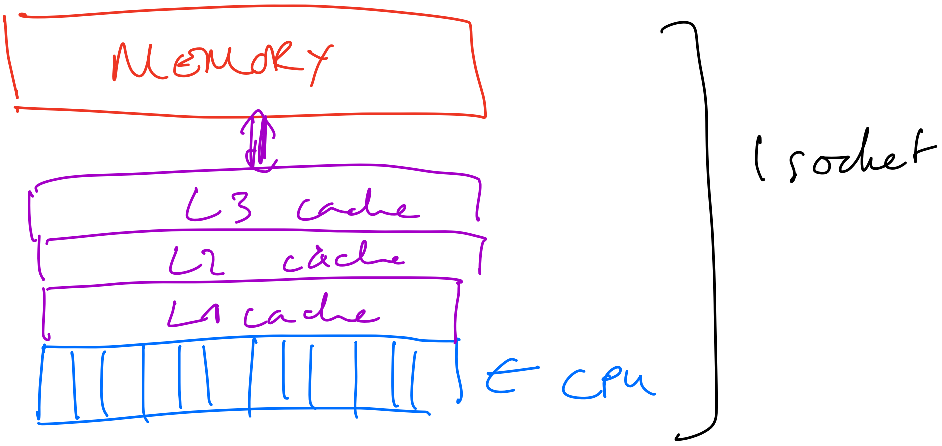
Lawrence Mitchell*

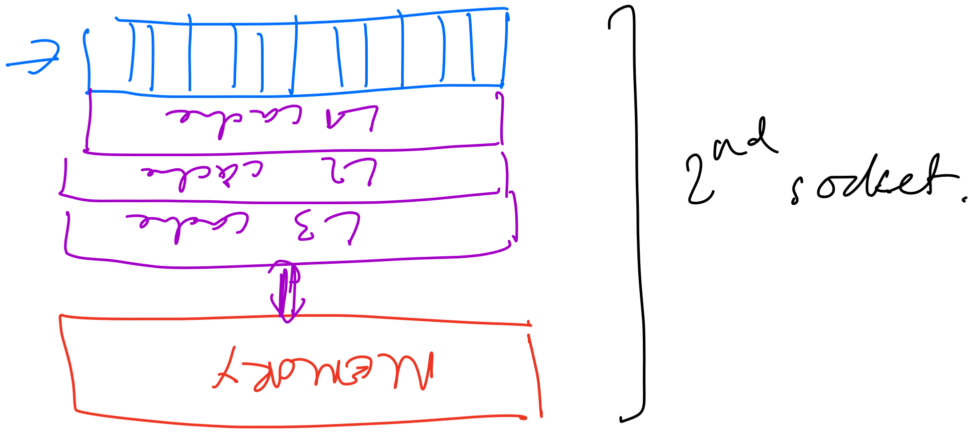
*lawrence.mitchell@durham.ac.uk

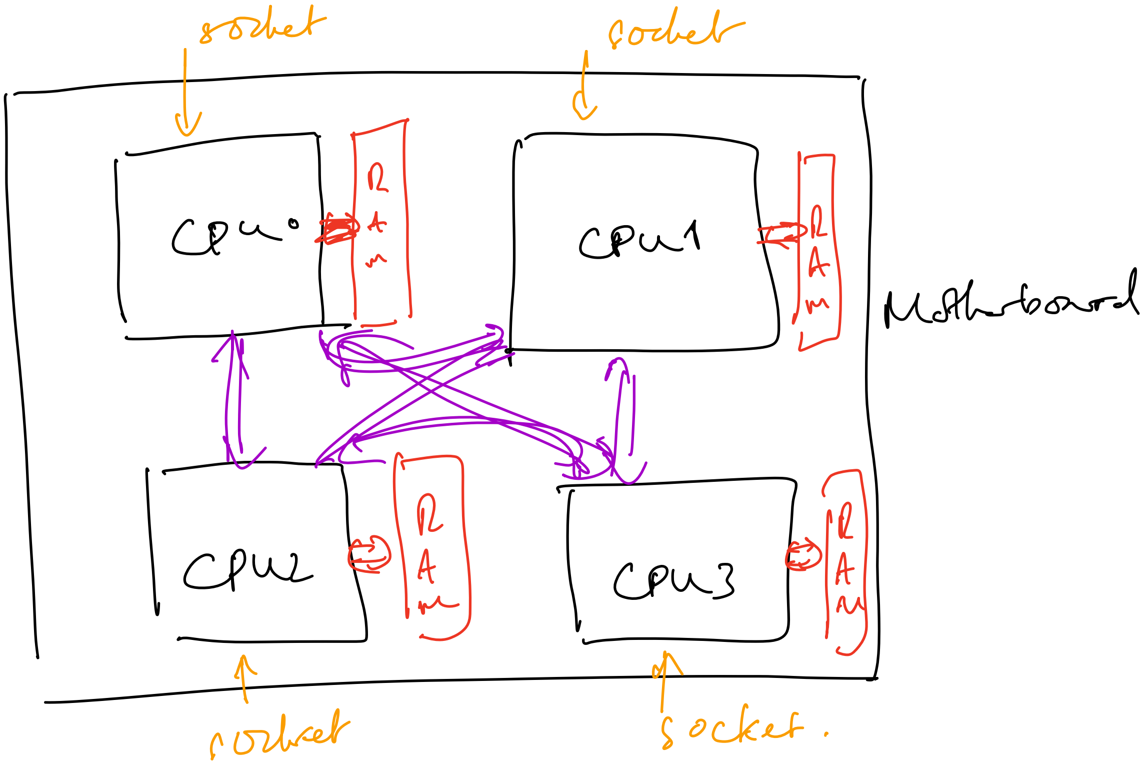
Parallel load bandwidth

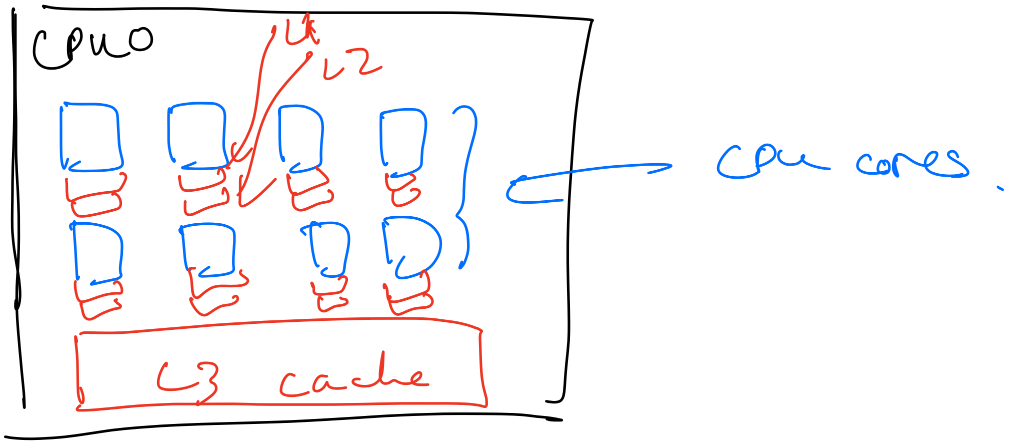
In exercise 3, you hopefully produced plots similar to these. Hamilton is 2-socket.




MEMORY
L3 cache
L2 cache
L1 cache
← CPU
1 socket


L1 cache
L2 cache
L3 cache
MEMORY
2nd socket.


socket
socket
CPU0
RAM
CPU1
RAM
Motherboard
CPU2
RAM
CPU3
RAM
socket
socket.



```
mpiexec --bind-to core
        --map-by socket -n 8./main
```

launch 4 processes
→ stick them each to one core, but spread them out over the sockets.

0 4 1 5

2 6 3 7

OMP_PROC_BIND

A more realistic measure of memory throughput

- The cache line copy benchmark we've seen provides upper bounds, but doesn't simulate *realistic* workloads.
- It only touches one byte in each cache line, but remember, optimised code works on *all* the bytes in a cache line.

⇒ STREAM benchmark `https://www.cs.virginia.edu/stream/`

- Most commonly used is TRIAD.
- Implemented in `likwid-bench` as `stream_triad_XXX` with a few different options.

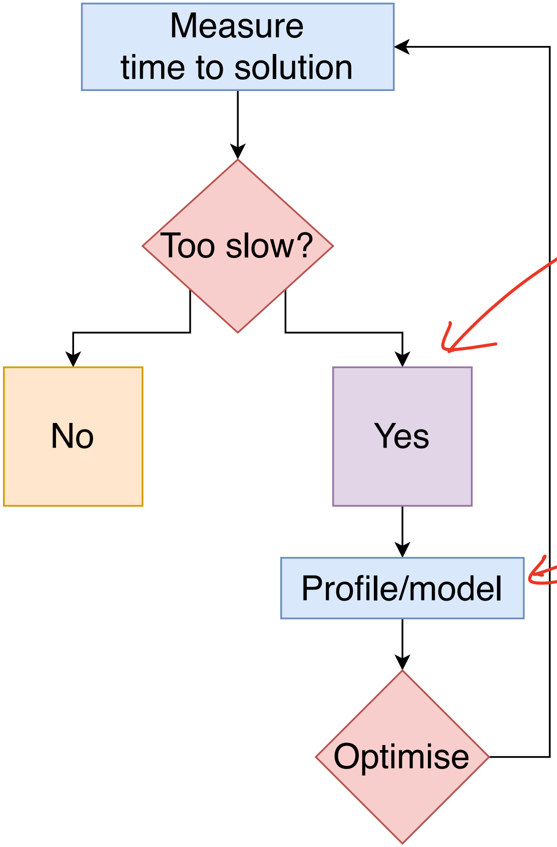
TRIAD loop

```
double *a, *b, *c;
double alpha = 1;
...
for (int i = 0; i < N; i++)
    a[i] = b[i]*alpha + c[i];
```

Limited by memory throughput at all cache levels, on all hardware post 1960.

This is the measure of memory bandwidth I use.

Code optimisation



Measure time to solution

Too slow?

No

Yes

Profile/model

Optimise

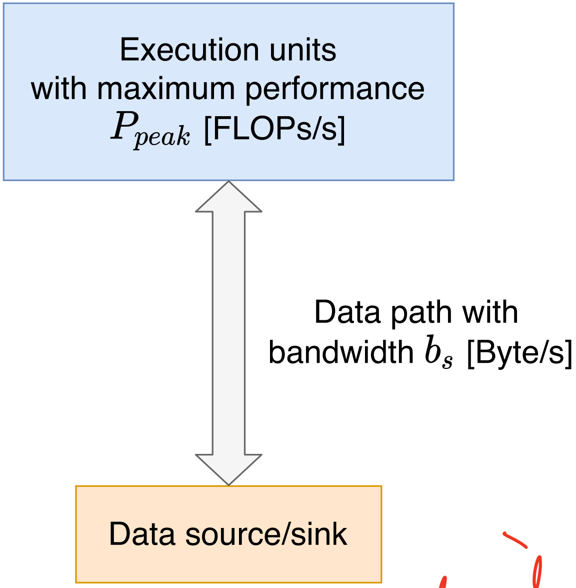
code running slower than we would like

How fast could the code run?

Simple model for loop heavy code

Roofline: simple model
→ can tell you what your code is doing at a high level.

Simple view of hardware



Execution units
with maximum performance
P_{peak} [FLOPs/s]

Data path with
bandwidth b_s [Byte/s]

Data source/sink

Hardware characterised by 2 numbers.

Simple view of software

```
/* Possibly nested loops */
for (i = 0; i < ...; i++)
 /* Complicated code doing */
 /* N FLOPs causing
 /* B bytes of data transfer */
```

Computational intensity [FLOPs/byte]
Arithmetic

$$I_c = \frac{N}{B}$$

Software characterised by 1 number.

Roofline

What is the performance P of a code?

How fast can work be done? P measured in FLOPs/s

Bottleneck

Either

- execution of work P_{peak} [FLOPs/s];
- or the data path $I_c b_s$ [FLOPs/byte × byte/s].

flops/s.

$$P = \min\left(P_{\text{peak}}, I_c b_s\right)$$

This is the simplest form of the roofline model. It is *optimistic*, everything happens at “light speed”.

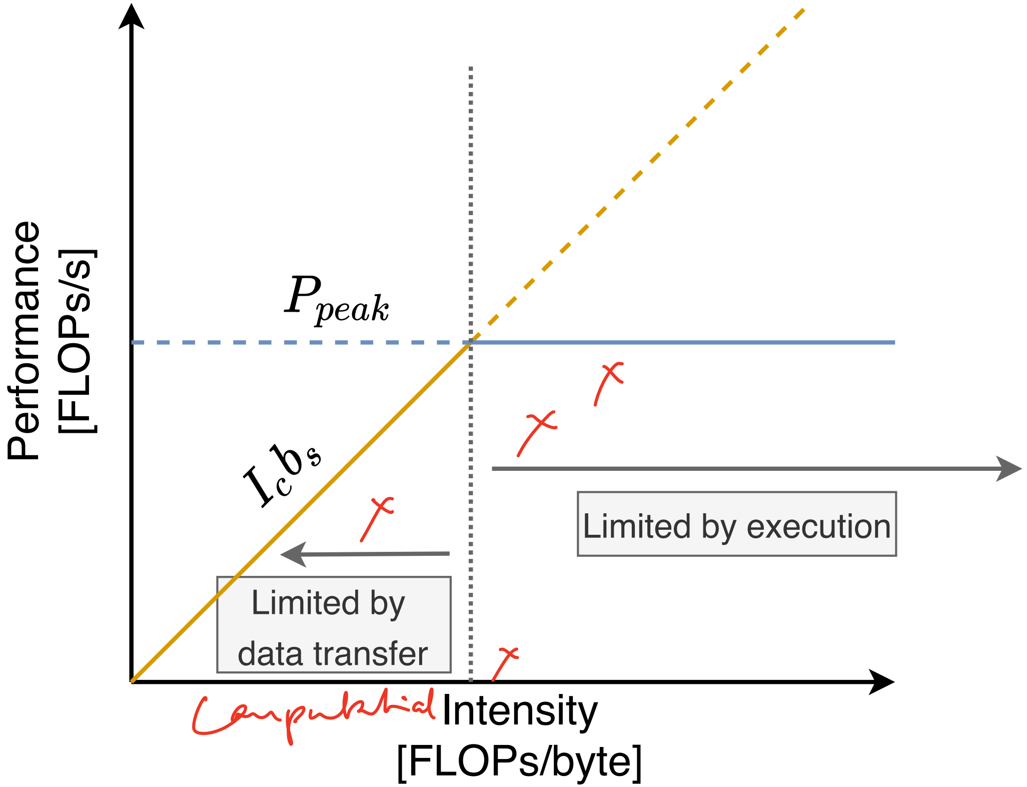
Introduced in Williams et al. *Roofline: An Insightful Visual Performance Model for Multicore Architectures*, CACM (2009). `https://doi.org/10.1145/1498765.1498785`

Read this paper.

come up with 1/2 questions comments. ⇒ for discussion tomorrow.

Roofline

Typically on a log-log scale.



Applying roofline

Performance model

Roofline characterises performance using three numbers:

1. P_{peak} the peak floating point performance;
2. b_s the streaming memory bandwidth;
3. I_c the computational (or arithmetic) intensity of the code.

measure these once per hardware.

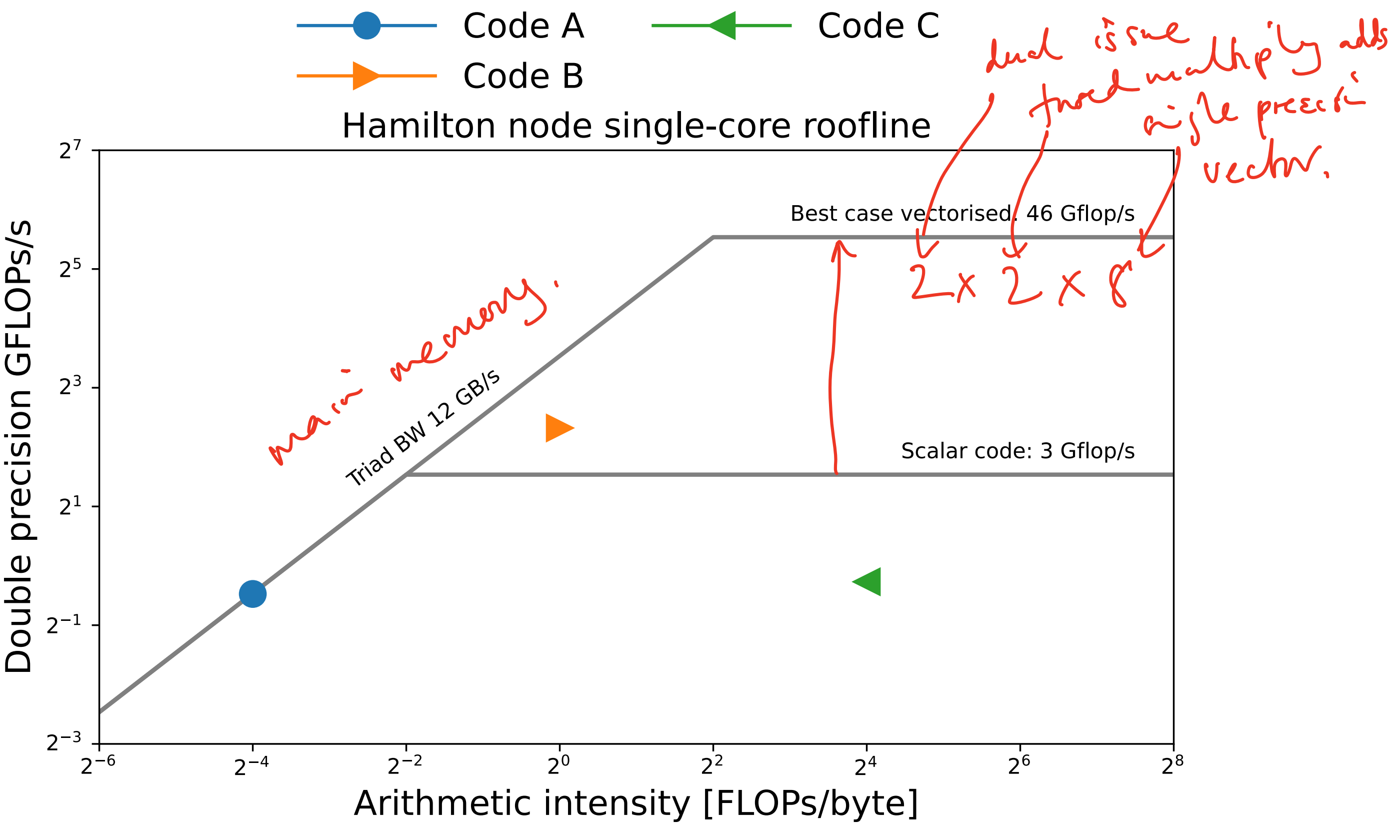
measure this per individual of code.

The first two are characteristics of *the hardware*. The last is a characteristic of the *code*.

Idea

Measure these numbers and plot, gives idea of what performance optimisations are likely to pay off.

Example



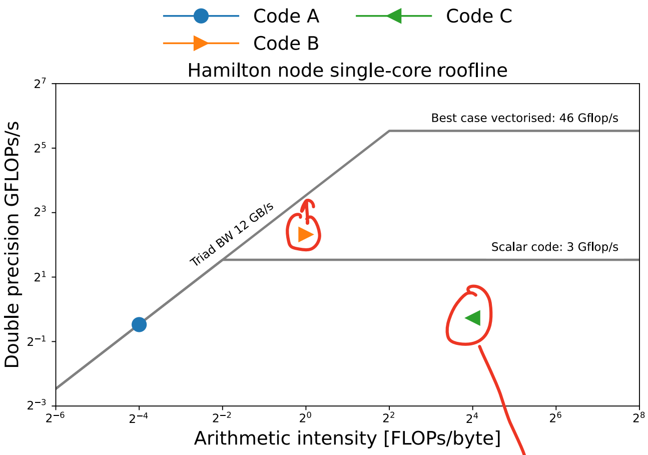
dual issue fused multiply adds single precision vector.

2x 2x 8

main memory.

Guides optimisation choices

Last: time to solution.



Which codes might benefit from vectorisation?

How much improvement could we expect?

Which codes might benefit from refactoring to increase arithmetic intensity?

Why is the floating point throughput so low?

Determining machine characteristics

Memory bandwidth

Roofline models data movement with streaming memory bandwidth.

Two ways of computing it.

1. Know what speed of memory you have, and look up number of memory channels on spec sheet. For example, 4-channel 2.4GHz RAM delivers at best 4 × 2.4GHz × 8Byte = 76.8GByte/s.
   ⇒ Needs knowledge of installed memory, typically not achieved in practice.
2. *Measure* using STREAM.
   ⇒ we will typically do this (see exercise 4).

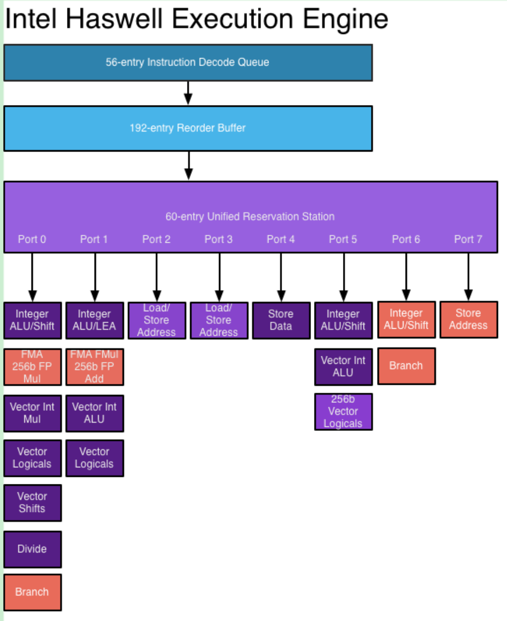
Need to do this for the level of parallelism you're using.

Determining machine characteristics

Floating point throughput

Absolute peak can be determined from spec sheet frequency and some knowledge of hardware.

wikichip.org



- Floating point instructions execute on port 0 and port 1
- Up to 4 “micro-ops” issued per cycle $\Rightarrow$ up to 2 floating point instructions per cycle
- FMA ($y \leftarrow a + b \times c$); MUL execute on both ports.
- ADD only executes on port 1. Divide only executes on port 0.

Determining machine characteristics

Example: best case

Code only contains double precision SIMD FMAs, clock speed is 2.9GHz.

"fused multiply add"

Peak floating point throughput is

$$y = \underbrace{a +}_{\text{add}} \underbrace{b * c}_{\text{mul}} \rightarrow \text{FMA}$$

$$\overbrace{2.9}^{\text{clock speed}} \times \underbrace{2}_{\text{dual issue}} \times \overbrace{4}^{\text{vector width}} \times \underbrace{2}_{\text{FMA}} = 46.4\text{GFLOPs/s}$$

issue 2 FMAs/cycle.

Example: only ADDs

Code only does double precision SIMD ADDs, clock speed is 2.9GHz.

$y = a+b.$

$$\overbrace{2.9}^{\text{clock speed}} \times \underbrace{1}_{\text{single issue}} \times \overbrace{4}^{\text{vector width}} = 11.6\text{GFLOPs/s}$$

only issue 1 ADD/cycle.

Determining machine characteristics

- Often useful to put multiple "roofs" on the roofline, corresponding to different instruction mixes.

when all cores are running frequency drops.

- Calculations are complicated by frequency scaling as well.

⇒ can add measured limit by running LINPACK (see exercises)

Big dense matrix-matrix multiplications.

More details

`https://uops.info` has all the information you could ever want on micro-op execution throughput.

`https://travisdowns.github.io/blog/2019/06/11/speed-limits.html` discusses in much more detail how to find limiting factors in (simple) code.

Computing arithmetic intensity

Two options:

1. Measure using performance counters (see later);
2. Read code, count floating point operations and data accesses.

Both options have their pros and cons.

Difficult if code is very big.

Counting operations

3N flops.

```
double *a, *b, *c, *d;
...
for (i = 0; i < N; i++) {
  a[i] = b[i]*c[i] + d[i]*a[i];
}
```

1 2 3

3 DP FLOPs/iteration. 3N total DP FLOPs. (Notice how we don't care about what type of FLOPs these are).

Counting data accesses

Each read counts as one access. Each write counts as two (one load, one store). Only care about array data (ignore loop variables)

```
double *a, *b, *c, *d;
...
for (i = 0; i < N; i++) {
  a[i] = b[i]*c[i] + d[i]*a[i];
}  4,5.     1     2       3
```

$$I_c = \frac{3N}{40N} = \frac{3}{40}.$$

3 DP reads, 1 DP write per iteration. 8 × 5N total bytes.

↳ sizeof(double) = 8.

Complication

```
double *a, *b, *c, *d;
for (i = 0; i < N; i++)
  for (j = 0; j < M; j++)
    a[j] = b[i]*c[i] + d[i]*a[j];
```

What about caches & data reuse.

For actual data moved, need a *model* of cache.

Bounds on movement

Real code

Perfect cache

Provides lower bound.

Each array entry moved from main memory once.

Counts *unique* memory accesses.

$8 \times 2M + 8 \times 3N$ total bytes.

Pessimal cache

Provides upper bound

Each array access misses cache.

Counts *total non-unique* memory accesses.

$8 \times 2MN + 8 \times 3MN$ total bytes.

Complication

Bounds on movement

Perfect cache

Provides lower bound.

Each array entry moved from main memory once.

Counts *unique* memory accesses.

$8 \times 2M + 8 \times 3N$ total bytes.

Pessimal cache

Provides upper bound

Each array access misses cache.

Counts *total non-unique* memory accesses.

$8 \times 2MN + 8 \times 3MN$ total bytes.

These bounds are typically not tight. If you want better bounds normally have to work harder in the analysis.

Best employed in combination with measurement of arithmetic intensity.

Exercise: roofline plot for dense matrix-vector multiplication

- Goal is to produce a roofline plot for dense matrix-vector multiplication, which computes

$$\vec{y} = A\vec{x} = \sum_j A_{ij}\vec{x}_j$$

`teaching.wence.uk/comp52315/exercises/exercise04/`