Durham University

Performance Analysis Workshop Series Revision: Scalability Laws

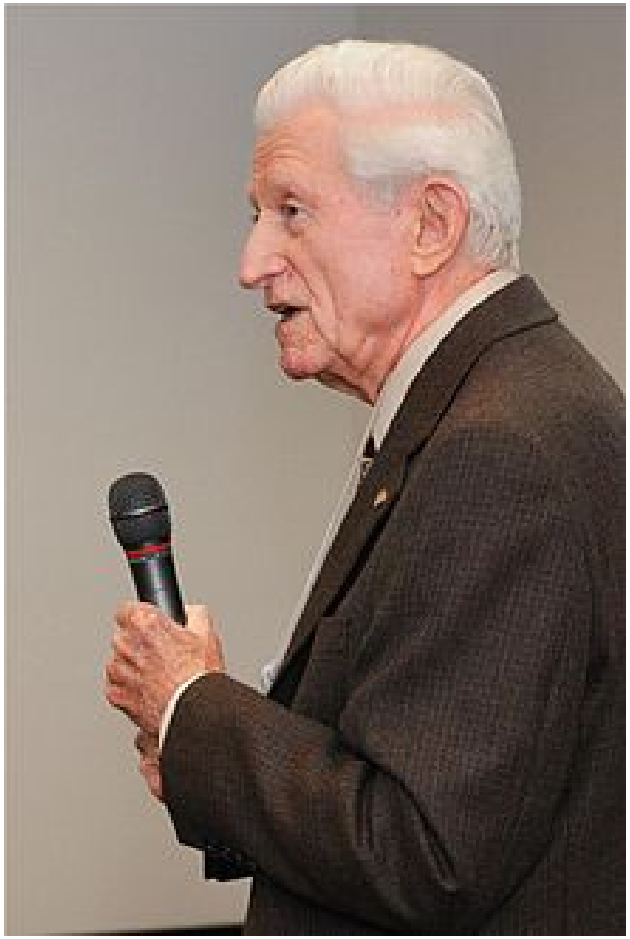
T. Weinzierl [1]

[1] Durham University

30 April 2024

Outline

Gene Amdahl



Gene Amdahl, 2008

- Gene Amdahl
- November 16, 1922 in South Dakota
- Computer architect (own company and IBM)
- Amdahl, G. M. (1967): *Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities*. AFIPS Conference Proceedings (30): 483–485

Definitions: Speedup and efficiency

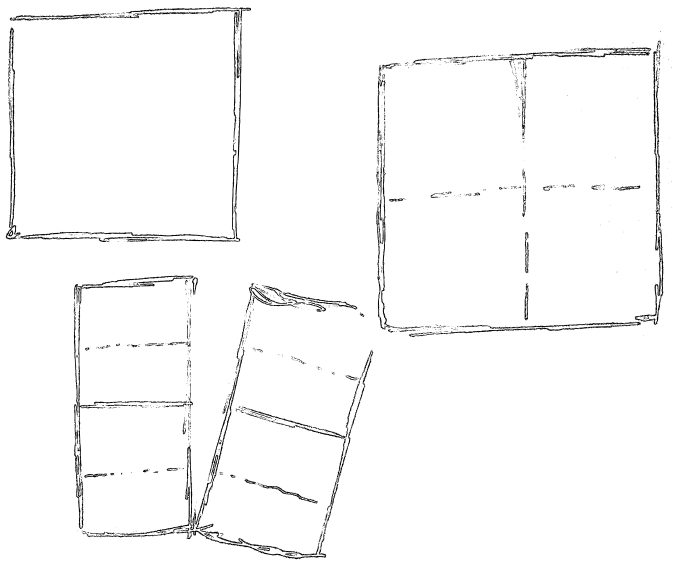
Let $t(1)$ be the time used by one processor. $t(p)$ is the time required by p processors. The *speedup* is

$$S(p) = \frac{t(1)}{t(p)}.$$

The *efficiency* of a parallel application is

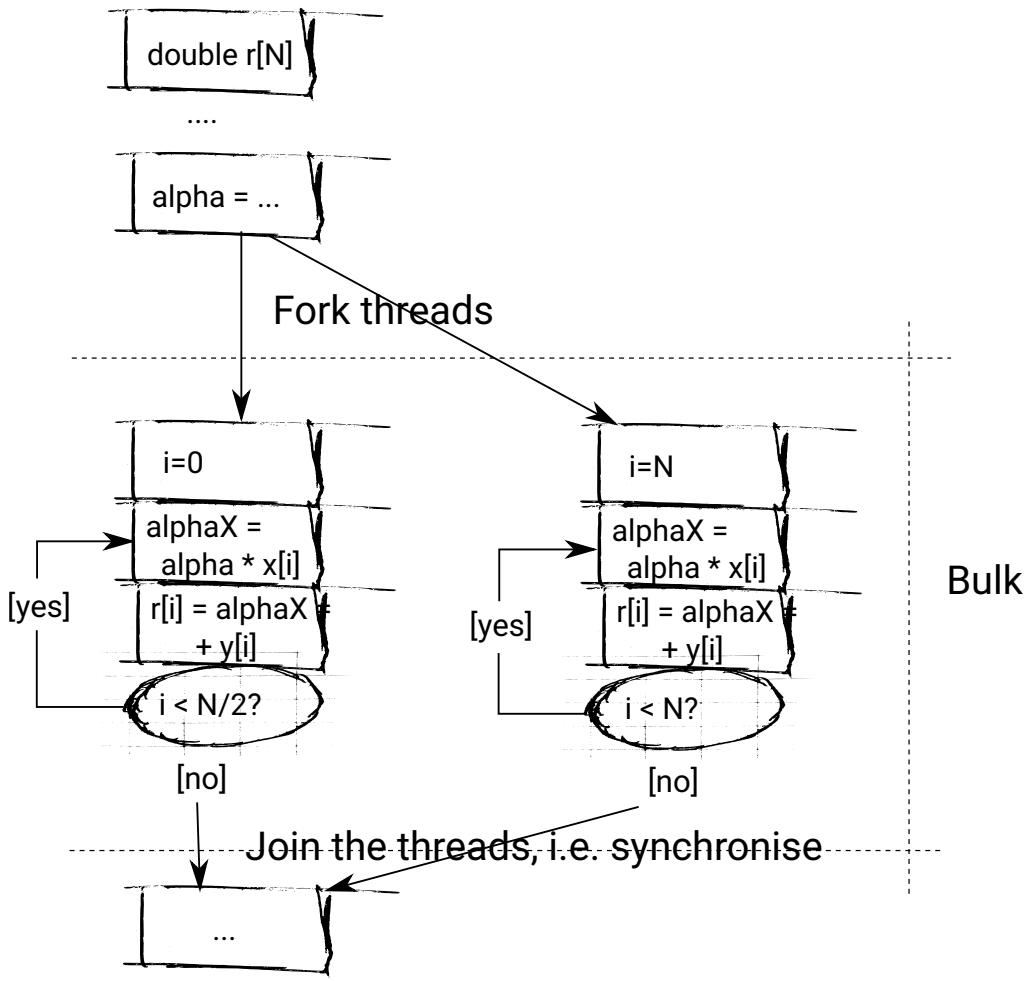
$$E(p) = \frac{S(p)}{p}.$$

Gedankenexperiment



- A single-core computer requires 64s to complete a job.
- How much time should a dual core computer require?
- How much time should a quadcore require?
- How would you define the terms speedup and efficiency?
- Write down three reasons why the speedup might actually be lower than expected.

Fork-join in action



- Some stuff runs serially
- Fraction of code parallelises nicely
- Real-world is not day and night

Amdahl's Law

$$t(p) = f \cdot t(1) + (1 - f)\frac{t(1)}{p}$$

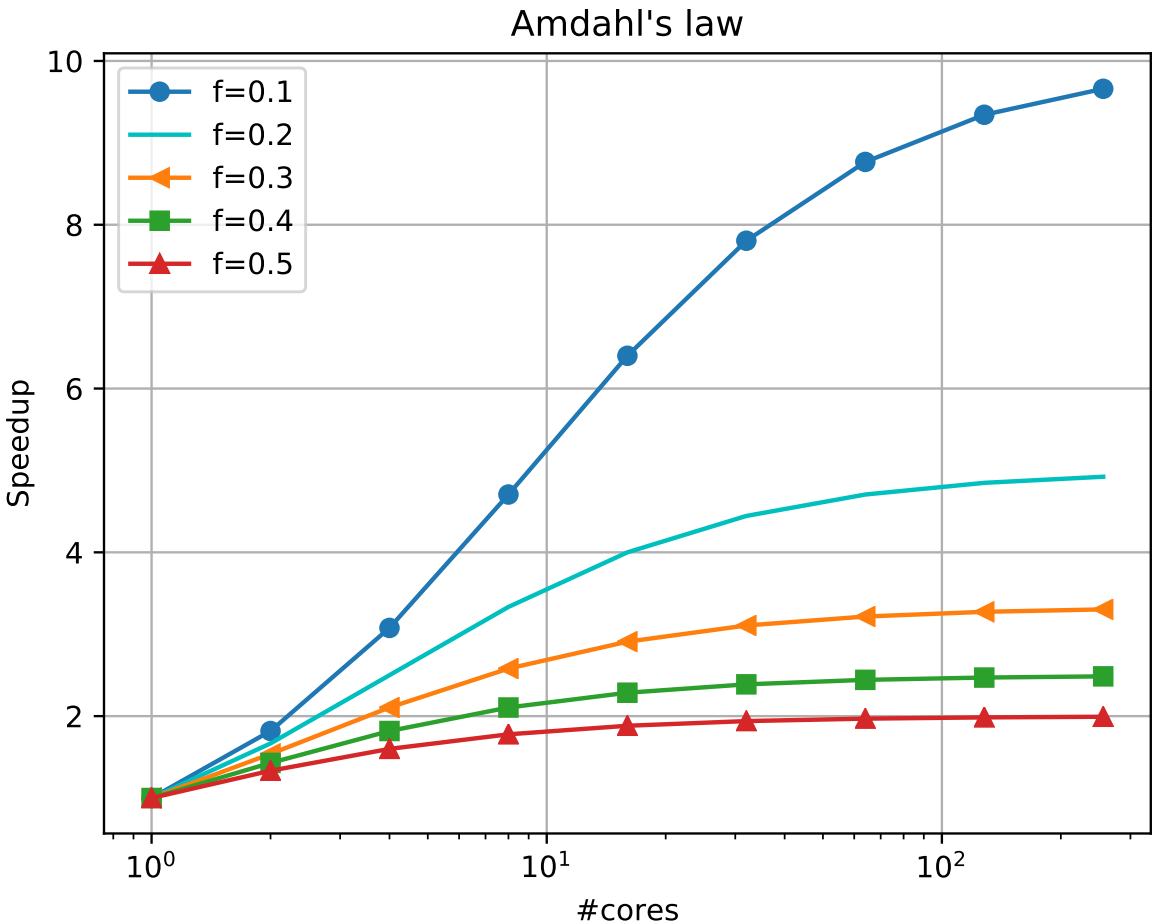
Ideas:

- Focus on simplest model, i.e. neglect overheads, memory, ...
- Assume that code splits up into two parts: something that has to run serially (f) with a remaining code that scales
- Assume that we know how long it takes to run the problem on one core.
- Assume that the problem remains the same but the number of cores is scaled up

Remarks:

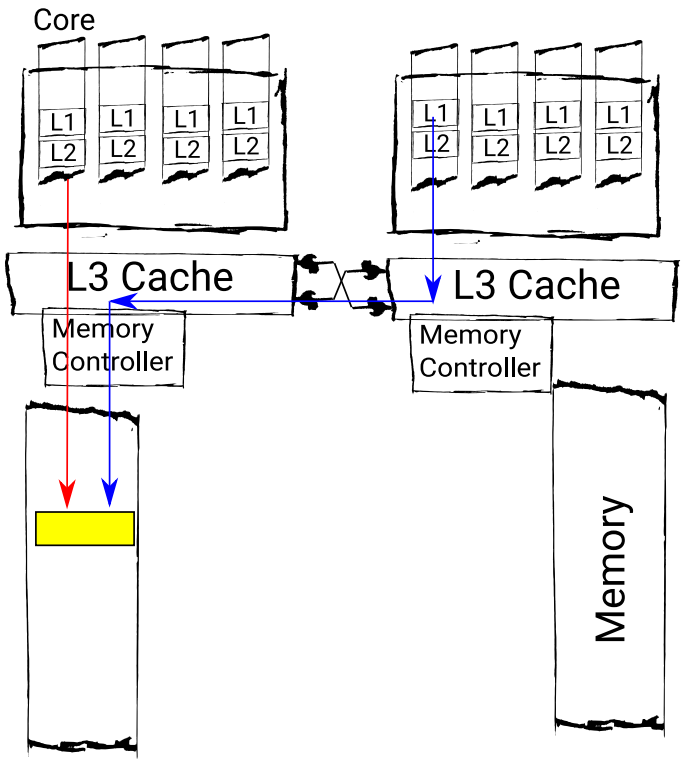
- We do not change anything about the setup when we go from one to multiple nodes. This is called **strong scaling**.
- The speedup then derives directly from the formula.
- In real world, there is some concurrency overhead (often scaling with p) that is neglected here.

Examples


Amdahl's law
f=0.1
f=0.2
f=0.3
f=0.4
f=0.5
Speedup
#cores
10
8
6
4
2
10^0
10^1
10^2

Outline

Revision: Two most important code flaws

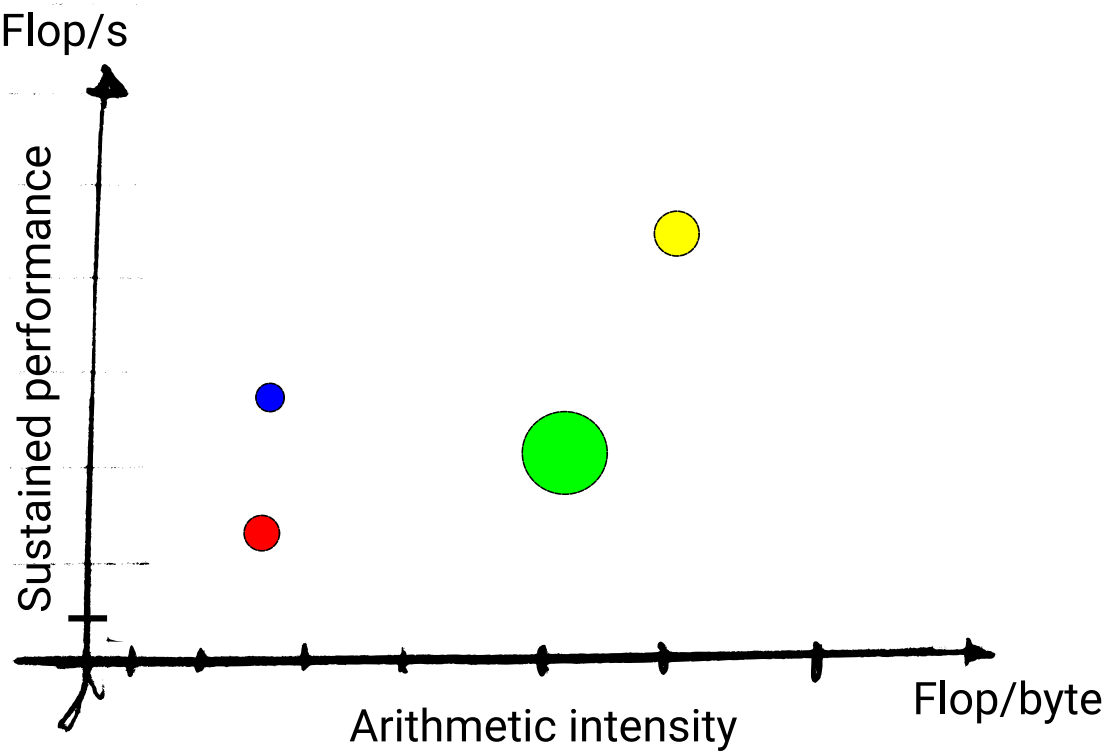


Compute-bound Code waits for floating-point calculations to finish.

Memory-bound Code waits for floating-point data to arrive in ALU.

$\Rightarrow$ More cores (strong scaling) makes only limited sense for ...

Function characterisation



- x-axis: arithmetic intensity (computations per byte)

$\Rightarrow$ implementation characterisation

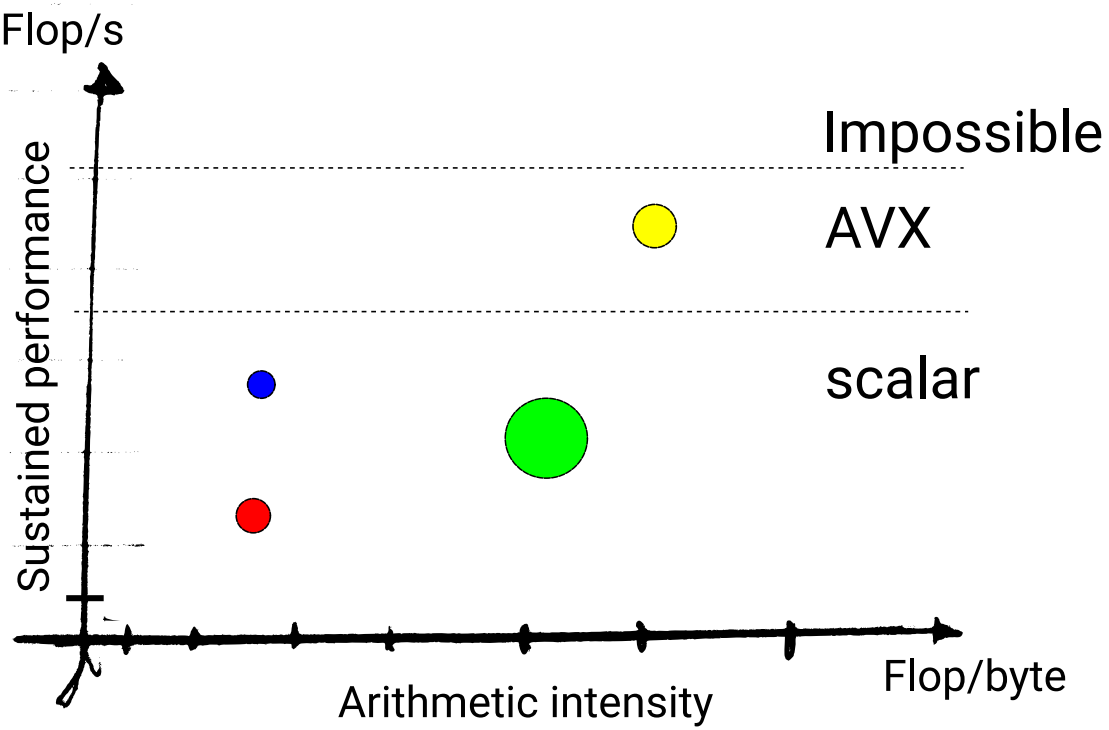
- y-axis: delivered floating point operations per second (FLOPS, flops or flop/s)

$\Rightarrow$ higher is better

- size of dot = relative runtime

(not part of original model)

Natural upper bounds



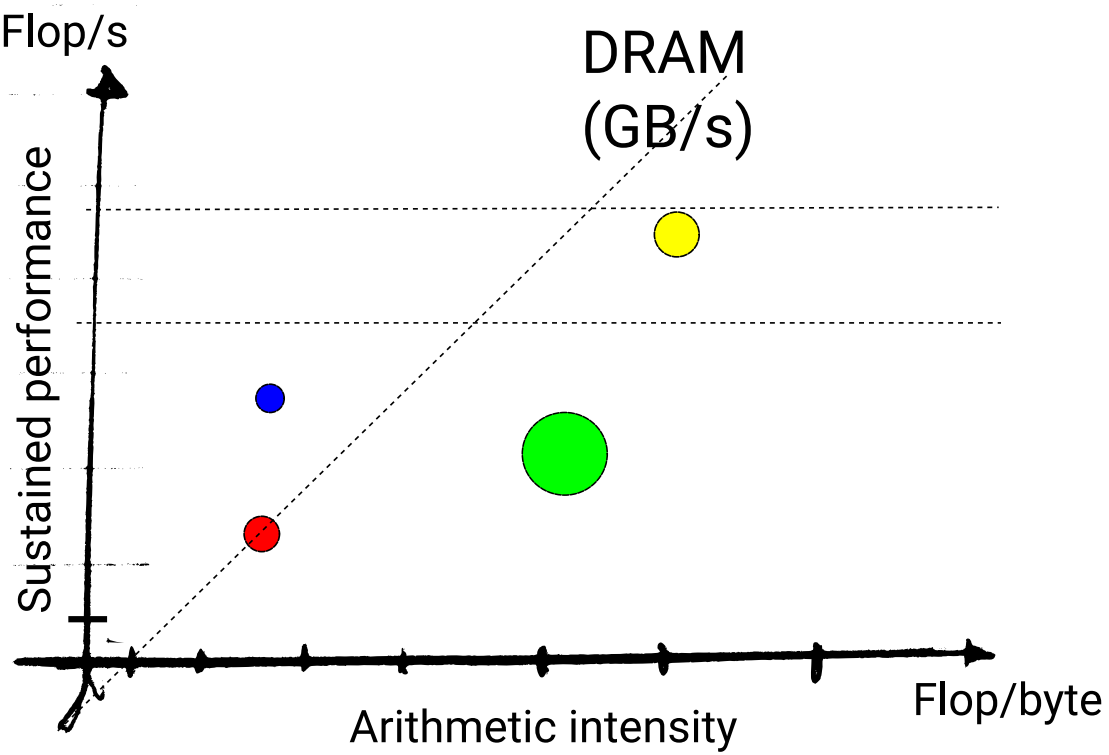
- Lower line: Scalar operations
- Upper line: Vectorisation limit

$\Rightarrow$ With different vectorisation flavours, there might be more lines

- Only yellow function vectorises
- No function/loop consists 100% of vector instructions

$\Rightarrow$ Hard upper limit

Memory bandwidth constraints



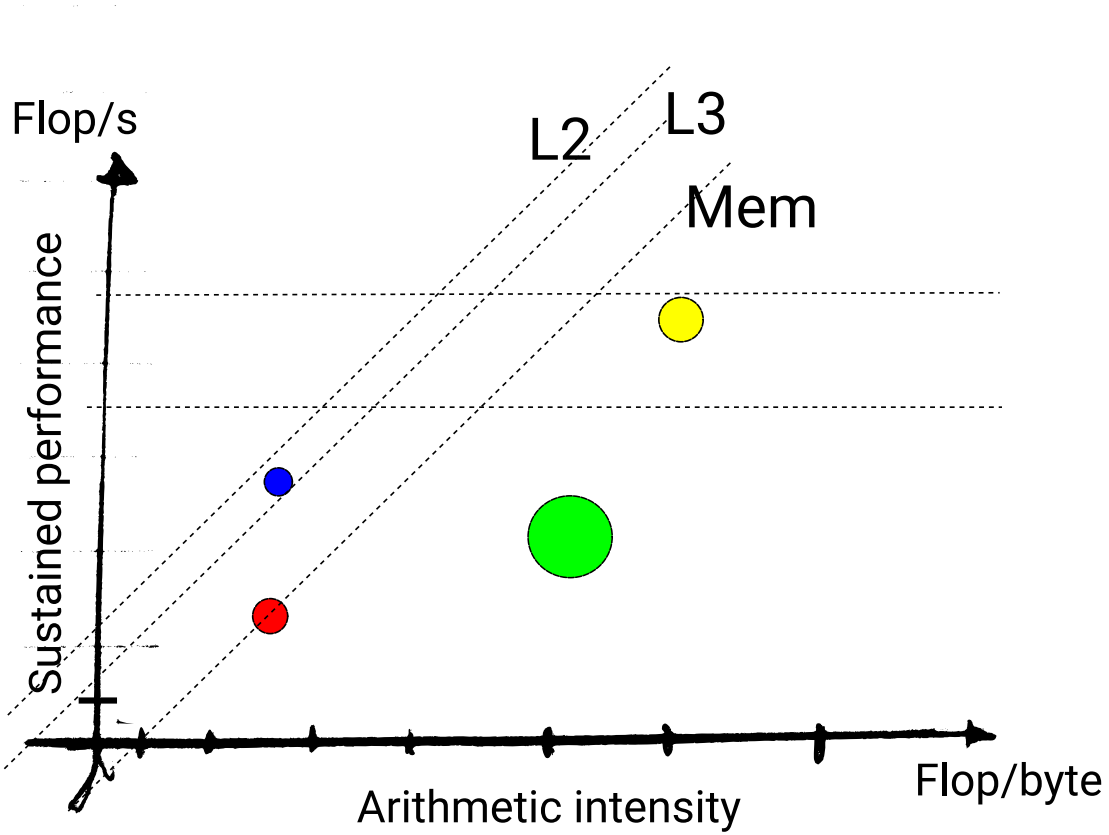
- Memory controller can only deliver certain GB/s
- Twice the operations per byte $\Rightarrow$ twice the performance

$\Rightarrow$ Diagonal line

- Red function exploits all the bandwidth (and latency)
- Maximum performance is minimum of lines

$\Rightarrow$ Yellow functions cannot be "better"

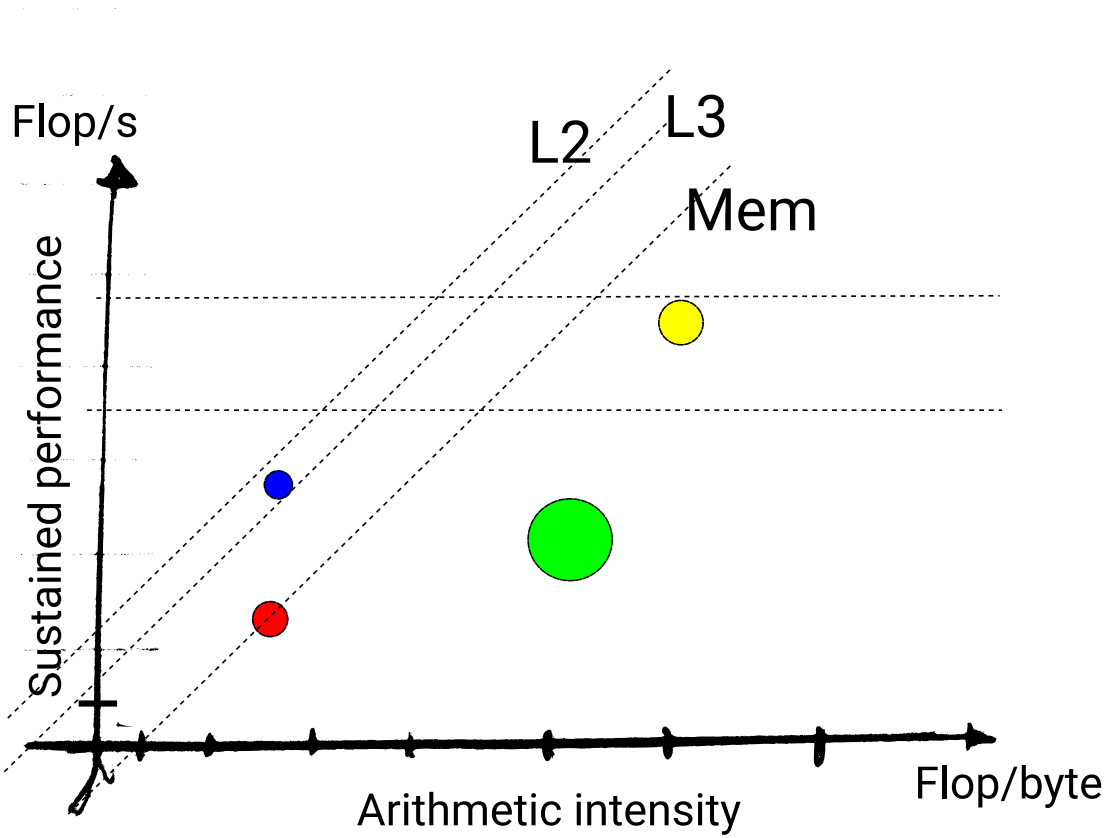
Caches



- Faster than memory controller
- Again, certain GB/s

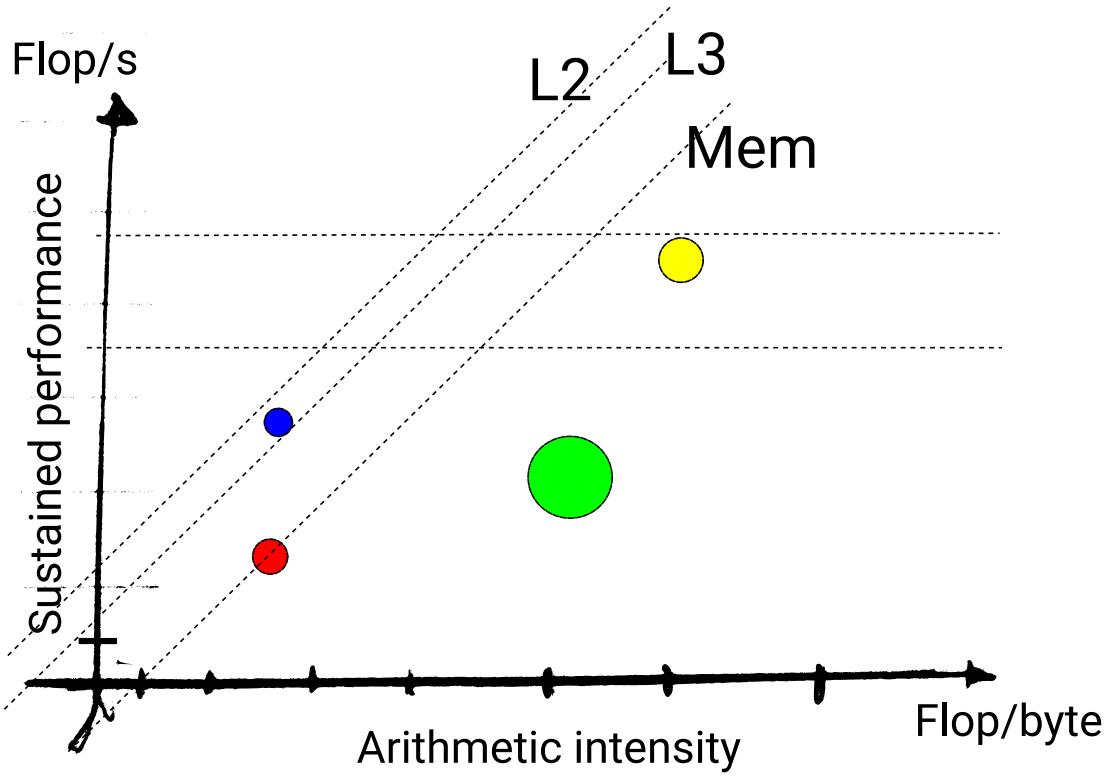
$\Rightarrow$ Blue function works with data in cache

Objectives



- Higher is better
- Bigger means more important to get fast
- Visualisation shows direction of travel

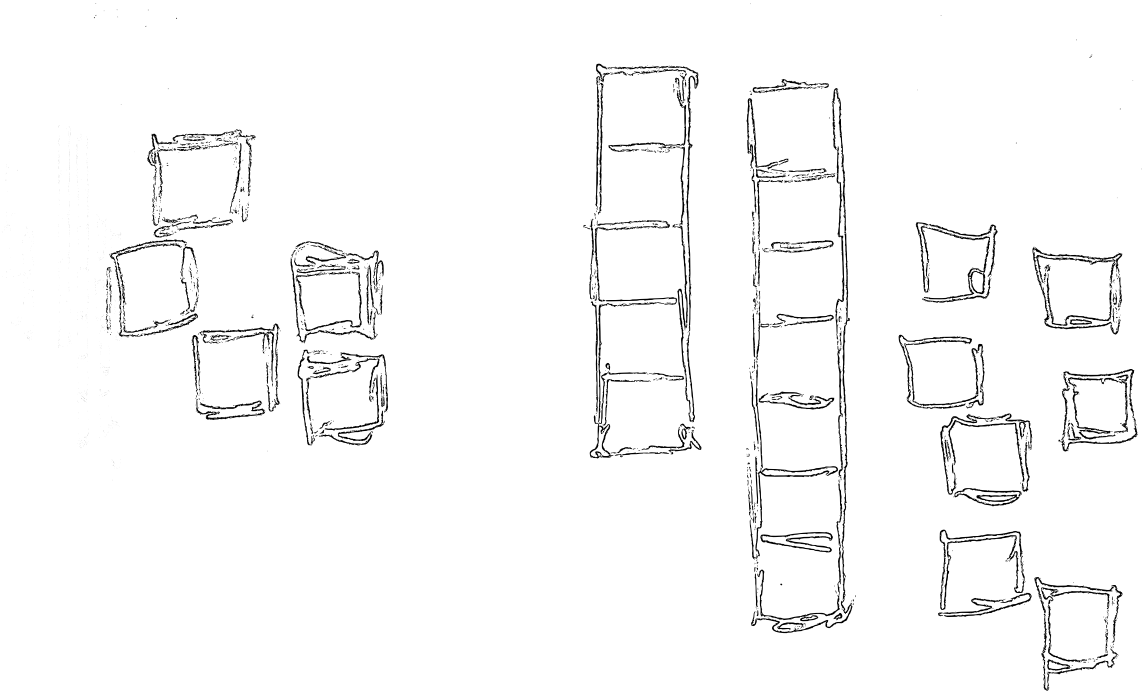
Actions



- If below vectorisation line and not memory-bound ⇒ can we vectorise?
  (remove ifs, reorder operations, hard-code loop counts, inline functions, . . .)
- If memory-bound ⇒ can we localise operations?
  (flatten into arrays, avoid jumping around in memory, fuse loops, . . .)
- Rule of thumb: Move right, then up

Outline

Gedankenexperiment



- A supercomputer with 64 nodes requires 2s to complete a job.
- How much time would a single node computer require if the program has a sequential fraction of instructions f?
- Write down the speedup.

Gustafson's Law

$$t(1) = f \cdot t(p) + (1 - f)t(p) \cdot p$$

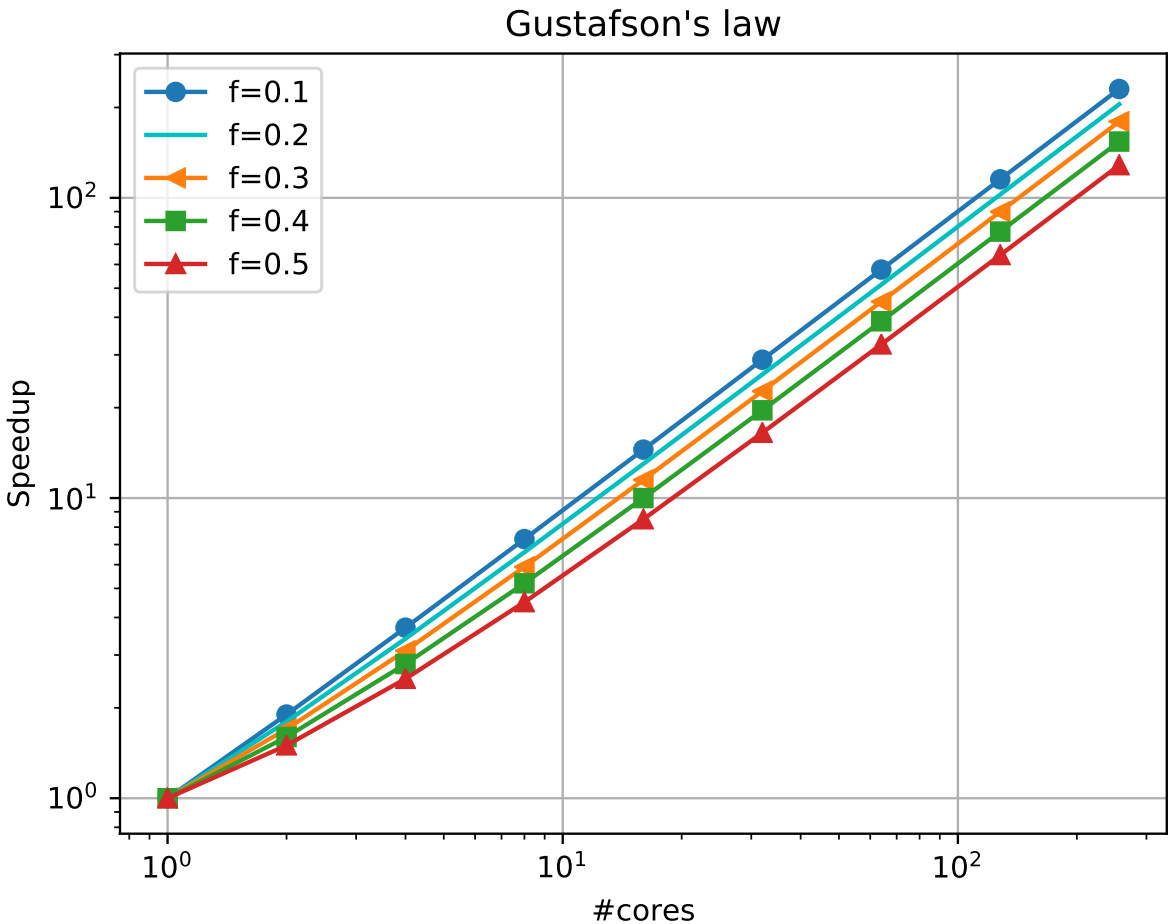
Ideas:

- Focus on simplest model, i.e. neglect overheads, memory, ...
- Assume that code splits up into two parts (cf. BSP with arbitary cardinality): something that has to run serially (f) and the remaining code that scales.
- Assume that we know how long it takes to run the problem on p ranks.
- Derive the time rerquired if we used only a single rank.

Remarks:

- Single node might not be able to handle problem and we assume that original problem is chosen such that whole machine is exploited, i.e. problem size is scaled This is called **weak scaling**.
- The speedup then derives directly from the formula.
- In real world, there is some concurrency overhead (often scaling with p) that is neglected here.

Examples


Gustafson's law
f=0.1
f=0.2
f=0.3
f=0.4
f=0.5
Speedup
10^0
10^1
10^2
10^0
10^1
10^2
#cores

Comparison of the speedup laws

$$t(p) = f \cdot t(1) + (1-f)\frac{t(1)}{p} \qquad t(1) = f \cdot t(p) + (1-f)t(p) \cdot p$$

$$S(p) = \frac{t(1)}{t(p)} \qquad S(p) = \frac{t(1)}{t(p)}$$

$$S(p) = \frac{1}{f + (1-f)/p} \qquad S(p) = f + (1-f)p$$

- It depends on whether you fix the problem size.
- It hence depends on your purpose.
- It is crucial to clarify assumptions a priori.
- It is important to be aware of shortcomings.