

SESSION 1

OVERVIEW



MASSIMILIANO FASI



Durham
University

One-slide course summary

Fundamental question

I would like this code to run faster: how do I know **what** to do?

One-slide course summary

Fundamental question

I would like this code to run faster: how do I know **what** to do?

Performance models & measurements

We can treat the computer as an experimental system:

1. Measure performance
2. Construct *models* that explain performance
3. Apply appropriate optimisations

Course overview

- ▶ Computer architecture overview
- ▶ Basics of performance engineering
- ▶ Tools: CPU topology and *affinity*
- ▶ Roofline performance model
- ▶ Tools: Performance counters
- ▶ Vectorisation (SIMD programming)
- ▶ Data layout transformations



`https://scicomp-durham.github.io/COMP52315/`

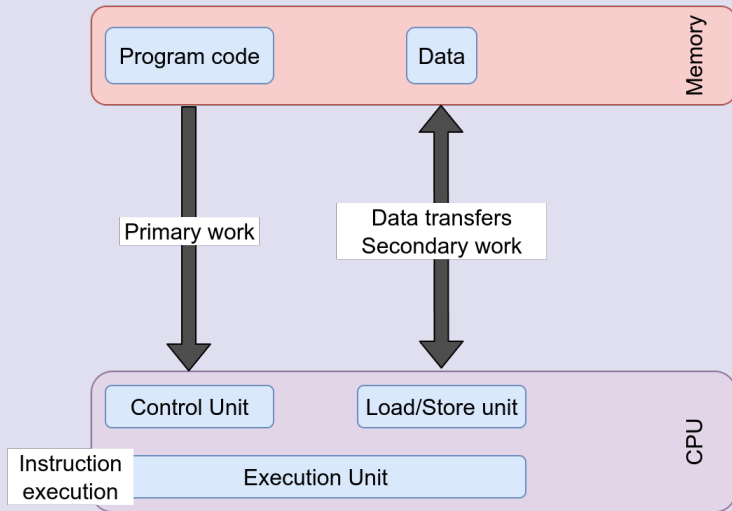
What you will need

- ▶ Hamilton account (which you should already have)
- ▶ familiarity with basic shell commands
- ▶ `likwid` tools, already available on Hamilton

Code?

This course is about **running** code, not writing it.

Stored-program architecture



Resource bottlenecks: instruction execution

- ▶ Primary resource of the processor.
- ▶ Measure is instruction throughput (instructions/second).
- ▶ First HW design goal is to *increase* instruction throughput.

Performance depends on how fast the CPU **retires** instructions.

Resource bottlenecks: instruction execution

- ▶ Primary resource of the processor.
- ▶ Measure is instruction throughput (instructions/second).
- ▶ First HW design goal is to *increase* instruction throughput.

Performance depends on how fast the CPU **retires** instructions.

Retired instruction

- ▶ CPUs execute more instructions than needed by program flow.
- ▶ “Retired instruction” are those whose results are **stored**.

Example: adding two arrays

```
for (int i = 0; i < N; i++)  
    a[i] = a[i] + b[i];
```

Example: adding two arrays

```
for (int i = 0; i < N; i++)  
    a[i] = a[i] + b[i];
```

User view

Work is N flops (additions)

Processor view

Work is $6N$ instructions

```
.top  
LOAD r1 = a[i]  
LOAD r2 = b[i]  
ADD  r1 = r1 + r2  
STORE a[i] = r1  
INCREMENT i  
GOTO .top IF i < N
```

Mismatch

User view

Work is N flops (additions)

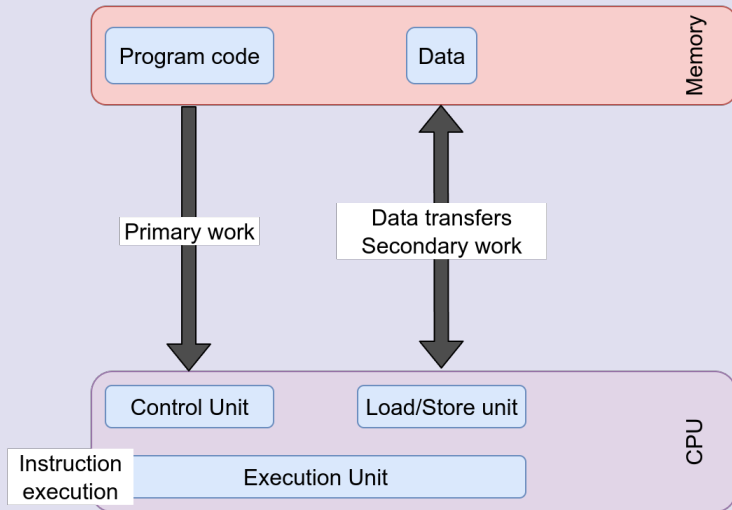
Processor view

Work is $6N$ instructions

Mismatch

- ▶ Processor designers: all instructions are “work”.
- ▶ Code developers: instructions I write are “work”.

Hardware for programmers



Resource bottlenecks: data transfer

- ▶ From memory to CPU and back.
- ▶ Consequence of instruction execution.
- ▶ Secondary resource.
- ▶ Measure is bandwidth (bytes/second).
- ▶ Bandwidth determined by load/store rate and HW limits.

Example: adding two arrays

```
for (int i = 0; i < N; i++)  
    a[i] = a[i] + b[i];
```

Data transfers (double precision floats):

```
LOAD   r1 = a[i]  /* 8 bytes */  
LOAD   r2 = b[i]  /* 8 bytes */  
STORE  a[i] = r1  /* 8 bytes */
```

24 bytes of data movement per loop iteration.

Understanding the performance of some code

Core question

What is the resource bottleneck?

- ▶ Instruction execution?
- ▶ Data transfer?

Understanding the performance of some code

Core question

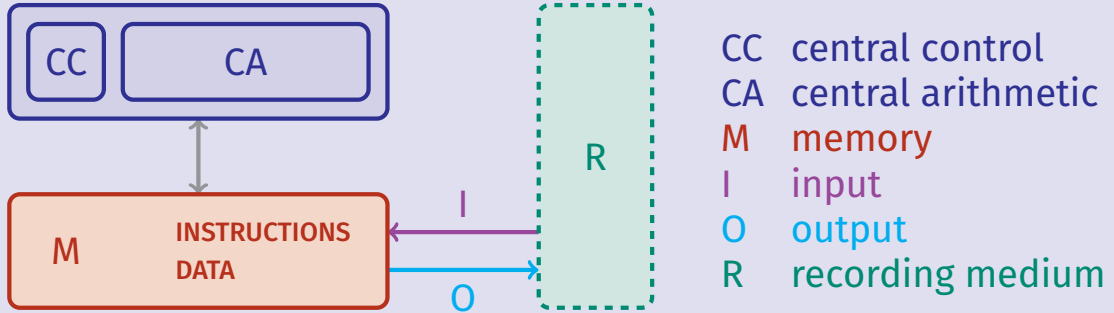
What is the resource bottleneck?

- ▶ Instruction execution?
- ▶ Data transfer?

Tools to find an answer

- ▶ Measurements
- ▶ Models

The “Princeton” architecture



- John von Neumann. *First draft of a report on the EDVAC*. Incomplete report, 1–101, 30 June 1945.

The “Princeton architecture”

- ▶ Used by programming languages
- ▶ Sequential model
- ▶ In-order execution
- ▶ Simple

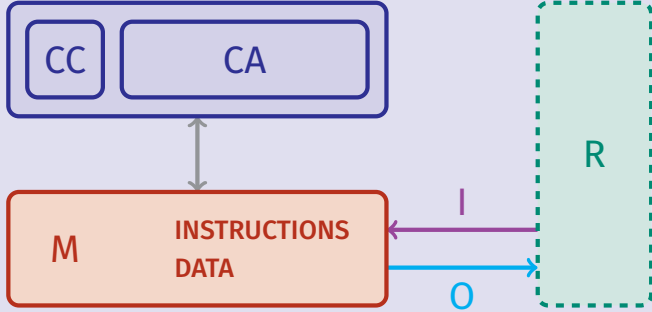
The “Princeton architecture”

- ▶ Used by programming languages
- ▶ Sequential model
- ▶ In-order execution
- ▶ Simple
- ▶ Realistic for 1945

📄 John von Neumann. *First draft of a report on the EDVAC*.
Incomplete report, 1–101, 30 June 1945.

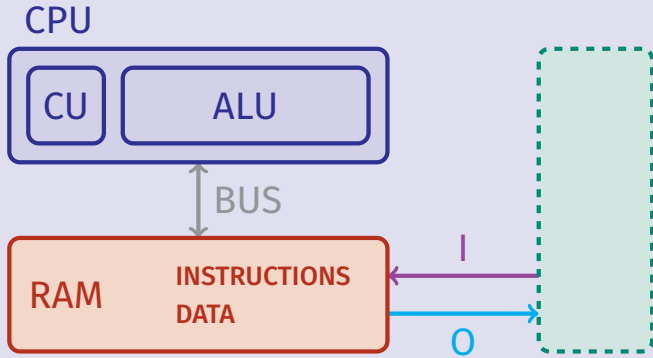
What has changed in the last 77 years?

The “Princeton” architecture today

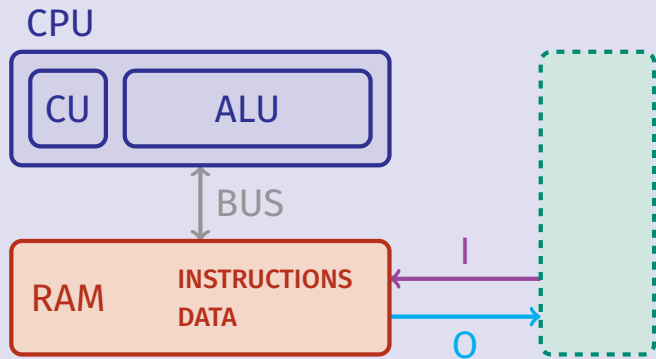


CC central control
CA central arithmetic
M memory
I input
O output
R recording medium

The “Princeton” architecture today

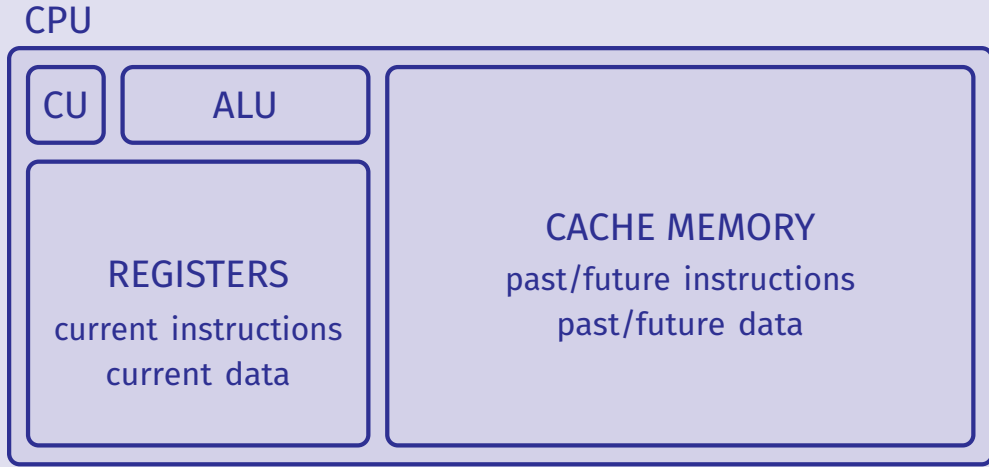


The “Princeton” architecture today



THE ONE FEATURE: both instructions and data reside in memory.
But CPUs are **much** more complicated today!

On-chip memory



Definitions

Cycle	unit of execution of CPU
Frequency	# cycles per second (measured in Hz)
Latency	# cycles to execute given instruction
Throughput	# instructions that can run simultaneously

Problem

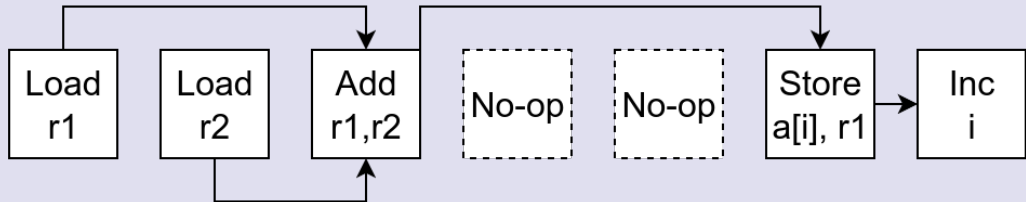
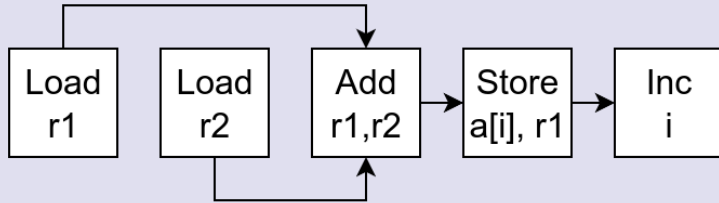
Most instructions have a *latency of more than one* clock cycle.

```
LOAD r1 = a[i]
LOAD r2 = b[i]
ADD  r1 = r1 + r2
STORE a[i] = r1
INCREMENT i
```

What happens if:

- ▶ all instructions have latency 1?
No “wasted” cycles.
- ▶ ADD has latency 3?
Two “wasted” cycles before STORE.

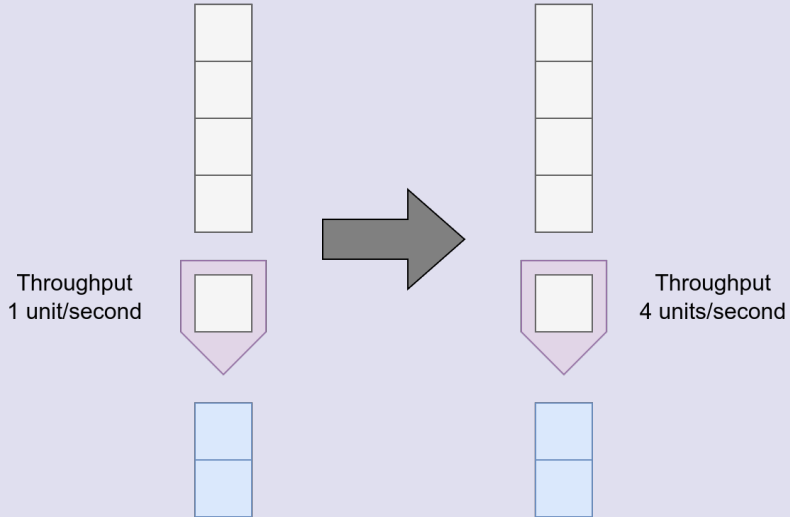
In pictures



Strategies for faster chips

1. Increase clock speed (more cycles per second)
2. Parallelism
 - ▶ data-level parallelism
 - ▶ instruction-level parallelism
3. Specialisation (optimised hardware units)

Increasing clock speed



Increasing clock speed

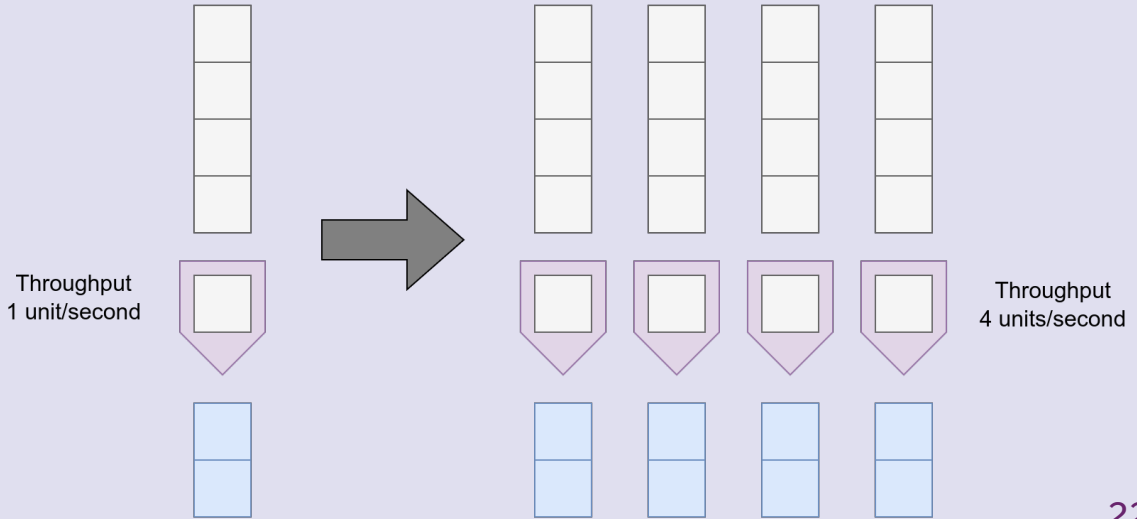
Easy for the programmer

Architecture is unchanged, everything just happens faster!

Limitations

- ▶ Limited by physical impossibility to cool chip.
- ▶ Clock speeds have been approximately constant for 10 years.

Increasing parallelism



Increasing parallelism

Problems

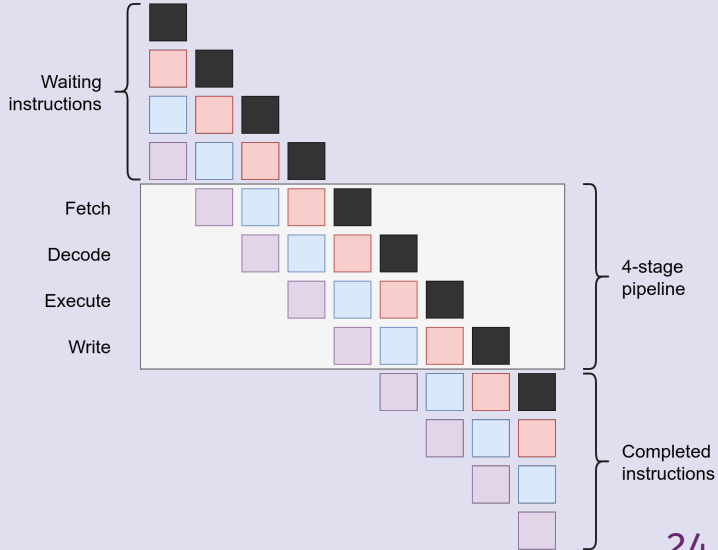
- ▶ Need enough parallel work
- ▶ No dependencies between work
- ▶ Mostly pushes problem onto programmer

Instruction-level parallelism: pipelining

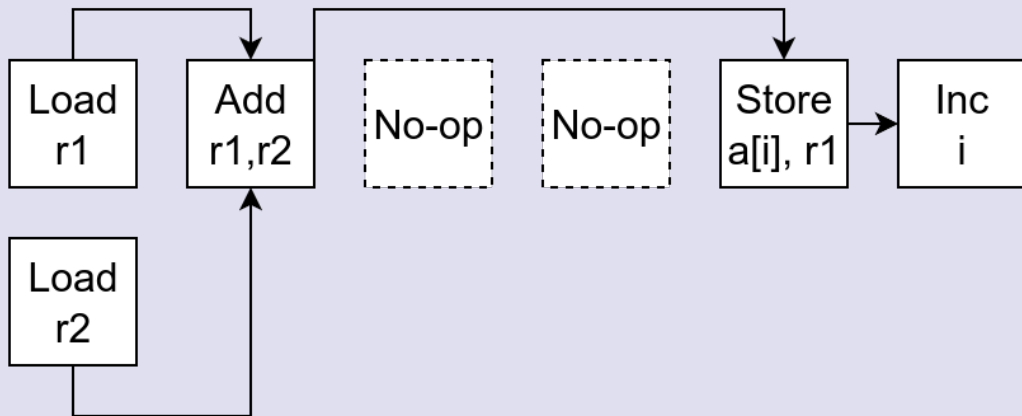
Split each instruction into

- ▶ fetch
- ▶ decode
- ▶ execute
- ▶ write

and use a pipeline.



Instruction-level parallelism: superscalar



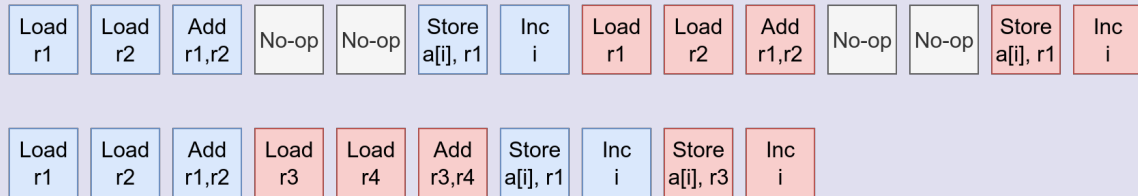
Instructions with no dependencies can be issued **simultaneously**.

Instruction-level parallelism: out-of-order

Instruction ordering is based on availability of

- ▶ input data
- ▶ execution units

rather than order in the program.



Data parallelism: SIMD vectorisation

Summing arrays again

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

Instruction throughput can be a bottleneck here.

Vectorisation: make instructions operate on more data at once.

Vectorisation is critical for **single-core** performance.

SIMD execution


```
double *a, *b, *c;
```


```
...
```

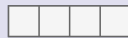
```
for (i = 0; i < N; i++)
```

```
    c[i] = a[i] + b[i];
```

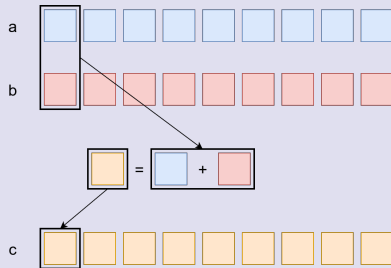
Register widths:

 1 operand (scalar)

 2 operands (SSE)

 4 operands (AVX)

 8 operands (AVX512)



Scalar addition

SIMD execution


```
double *a, *b, *c;
```

```
...
```


```
for (i = 0; i < N; i++)
```

```
    c[i] = a[i] + b[i];
```

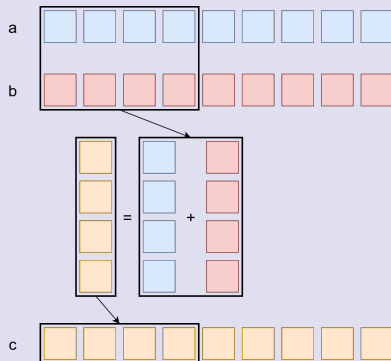
Register widths:

 1 operand (scalar)

 2 operands (SSE)

 4 operands (AVX)

 8 operands (AVX512)



AVX addition

Example: sum reduction

How fast can this code run if all data are in L1 cache?

```
float c = 0;  
for (i = 0; i < N; i++)  
    c += a[i];
```

Notes

- ▶ AVX-capable core (vector width: 8 floats)
- ▶ Loop-carried dependency on summation variable
- ▶ Execution stalls at every add until the previous one completes

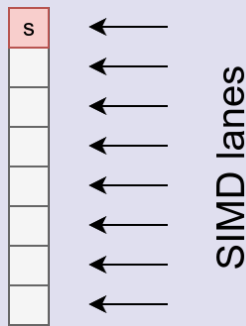
Applicable peak (scalar execution)

```
float c = 0;  
for (i = 0; i < N; i++)  
    c += a[i];
```

Assembly pseudo-code

```
LOAD r1.0 ← 0  
i ← 0  
loop:  
    LOAD r2.0 ← a[i]  
    ADD  r1.0 ← r1.0 + r2.0  
    i ← i + 1  
    if i < N: loop  
result ← r1.0
```

Only one SIMD lane.



Runs at $\frac{1}{8}$ of possible ADD peak.

Applicable peak (SIMD execution)

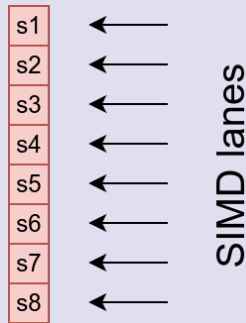
Scalar code

```
float c = 0;
for (i = 0; i < N; i++)
    c += a[i];
```

Assembly pseudo-code

```
LOAD [r1.0, ..., r1.7] ← [0, ..., 0]
i ← 0
loop:
    LOAD [r2.0, ..., r2.7] ← [a[i], ..., a[i+7]]
    ADD r1 ← r1 + r2 // SIMD ADD
    i ← i + 8
    if i < N: loop
result ← r1.0 + r1.1 + ... + r1.7
```

Using all eight SIMD lanes



Runs at ADD peak.

Exercise: benchmarking sum reduction

1. Split into small groups
2. Make sure one person per group has access to Hamilton
3. Benchmark sum reduction to confirm this “theoretical” effect.
4. Ask questions!

Conclusions

- ▶ Modern computer hardware is quite complex
- ▶ For simple things we can try to figure our performance limits
- ▶ Typically we must benchmark to confirm hypotheses
- ▶ We must find bottlenecks before starting to optimise

Next: memory hierarchy and first models of performance.