

Session 1: Introduction & Overview

COMP52315: performance engineering

Lawrence Mitchell^{*}

^{*}`lawrence.mitchell@durham.ac.uk`

Introduction

What is the course about?

Saw (in Core I: High Performance Computing) different parallel programming paradigms.

Parallelism helps to improve performance (runtime) of a code.

Question

Given some code, which I would like to make faster, how do I know *what* to do?

What is the course about?

Saw (in Core I: High Performance Computing) different parallel programming paradigms.

Parallelism helps to improve performance (runtime) of a code.

Question

Given some code, which I would like to make faster, how do I know *what* to do?

Performance models & measurements

We can treat the computer as an experimental system

- ⇒ perform measurements of the performance
- ⇒ construct *models* that explain performance
- ⇒ apply appropriate optimisations

Course overview (not in order, approximate)

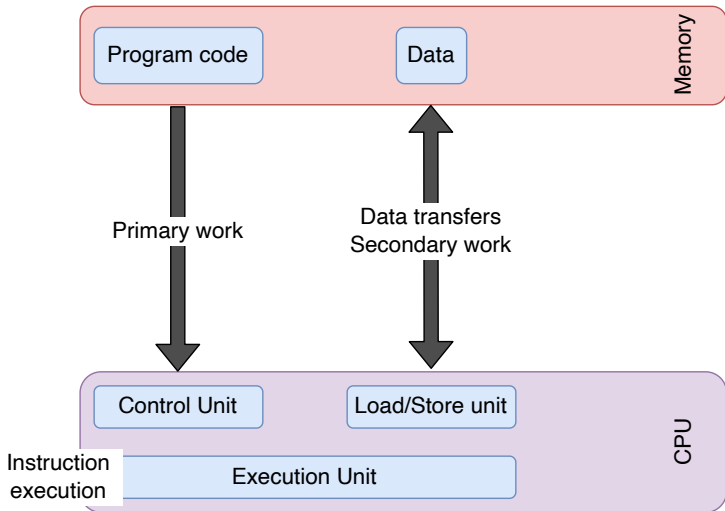
- Computer architecture overview
- Fundamentals of performance engineering
- Tools: CPU topology and *affinity*
- Roofline performance model
- Tools: Performance counters
- Vectorisation (SIMD programming)
- Data layout transformations

What you need

- The toolchain we'll use is available on Linux machines (but not Windows/Mac)
 - I recommend using Hamilton (you should already have accounts from Core I)
- ⇒ Short howto from Core I website. Exercises will also recap some of this material.

Resources in stored program computers

Hardware for programmers



Resource bottlenecks: instructions

Instruction execution

How fast the CPU retires instructions.

Primary resource of the processor. Primary hardware design goal is to *increase* instruction throughput (instructions/second).

Mismatch

Instructions are “work” as seen by processor designers.

Not all instructions are considered “work” by software developers (you!).

Resource bottlenecks: instructions

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
```

Resource bottlenecks: data transfer

Data Transfer

Data movement (from memory to CPU and back) is a *consequence* of instruction execution and considered a secondary resource. Maximum bandwidth (bytes/second) determined by rate at which load/store instructions can be executed and hardware limits.

Data movement 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.

To understand the performance of some code we must answer

Question

What is the resource bottleneck?

- Data transfer?
- Instruction execution?

To understand the performance of some code we must answer

Question

What is the resource bottleneck?

- Data transfer?
- Instruction execution?

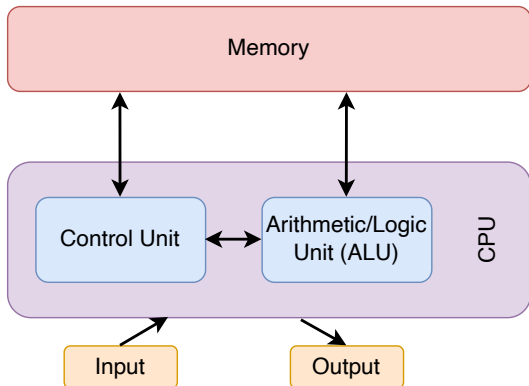
Answer

We will see how to answer these questions in this course through a combination of measurements and models.

Real hardware vs. models

Model of hardware as presented by programming languages is von Neumann model.

Sequential execution of instructions, each instruction completes before next one starts.



Problem

- CPUs operate at a certain frequency, we will count time in terms of clock “cycles”. For example, a 1GHz processor runs at one billion cycles per second.
- Due to the complexity of modern chips, most instructions have a *latency of more than one clock cycle*.

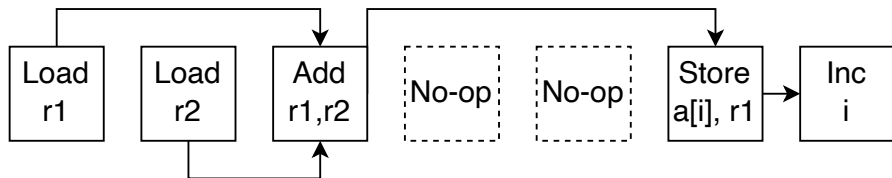
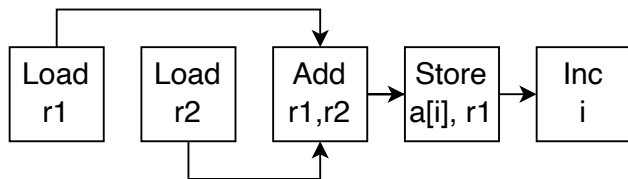
Example: addition loop

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

Suppose that the CPU can execute one instruction per cycle. If every instruction has a latency of one cycle, then there are no “wasted” cycles.

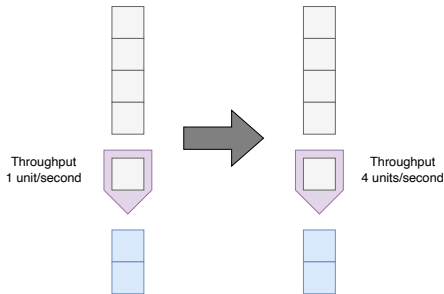
If **ADD** has latency of three cycles, then there are two wasted cycles (between the **ADD** and the **STORE**).

A picture



1. Increase clock speed (more cycles per second)
2. Parallelism (of various kinds)
3. Specialisation (for example optimised hardware for computing divisions)

Increasing clock speed



Easy for the programmer

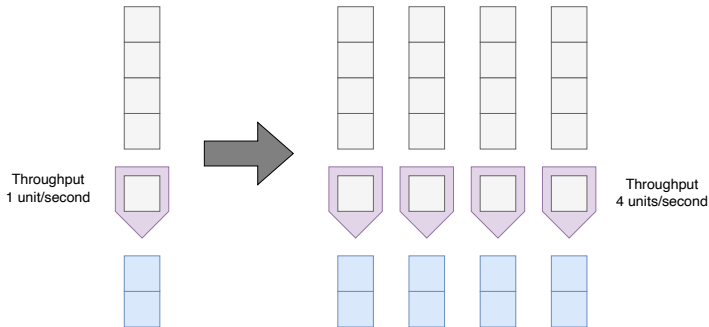
Architecture is unchanged, everything just happens faster!

Limitations

Limited by physical limitations on cooling.

Clock speeds have been approximately constant for 10 years.

Increasing parallelism



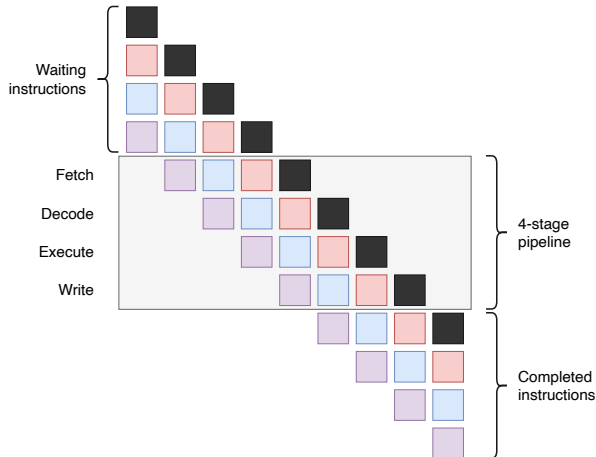
Problems

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

Instruction-level parallelism: pipelining

Pipelining

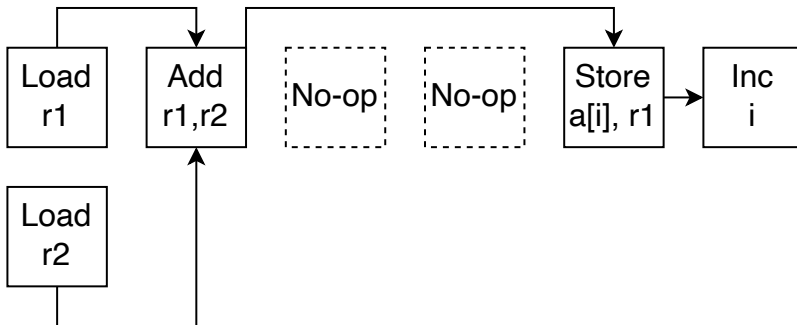
Rather than performing instruction fetch, decode, execute, and writeback in one go, separate them into a pipeline.



Instruction-level parallelism: superscalar

Superscalar execution

Most modern chips can issue more than one instruction per cycle.
Instructions with no dependencies can be issued simultaneously.



Instruction-level parallelism: out-of-order

Out-of-order execution

Execute instructions in an ordering based on availability of *input data* and *execution units* rather than the order in the program.

Keeps more of the execution units busy.



Data parallelism: SIMD vectorisation

SIMD

We mostly consider “single-core” performance in this course.
Vectorisation is critical for single-core performance.

Summing arrays again

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

We’ve seen that instruction throughput can be a bottleneck here.

One way chip designers have “fixed” this is to make individual instructions operate on more data at once \Rightarrow vectorisation.

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


```
...
```


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

Register widths:

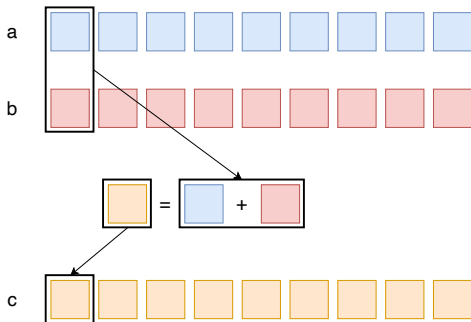
 1 operand (scalar)

 2 operands (SSE)

 4 operands (AVX)

 8 operands (AVX512)

Scalar addition, 1 output element per instruction.




AVX addition, 4 output elements per instruction.


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

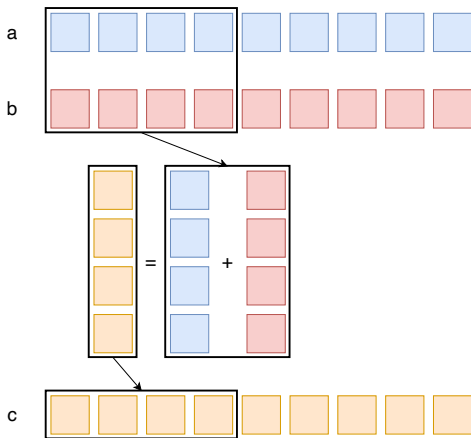
Register widths:

 1 operand (scalar)

 2 operands (SSE)

 4 operands (AVX)

 8 operands (AVX512)



Example and exercise

A “simple” example: sum reduction

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

Single precision sum of all values in a vector,
on an AVX-capable core (vector width 8).

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

- Loop-carried dependency on summation variable
- Execution stalls at every add until the previous one completes.

Applicable peak

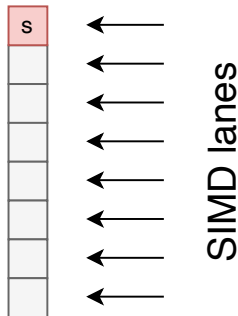
Scalar code

```
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
```

ADD has latency of 1 cycle (per Intel), but we're only using one of the eight SIMD lanes for each instruction.



Runs at 1/8 of possible ADD peak.

SIMD vectorisation

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

- Split into small groups, each group should have at least one person with a Hamilton account.
- Goal is to benchmark sum reduction to see if we observe this “theoretical” effect.
- \Rightarrow over to you. Please ask questions!

Exercises, and notes, live at

<https://teaching.wence.uk/comp52315/>

Conclusions

- Modern computer hardware is quite complex
- For simple things we can work out what the performance limits will be
- Typically must benchmark to confirm hypotheses
- Next, we'll look at the memory hierarchy and start constructing models of performance.