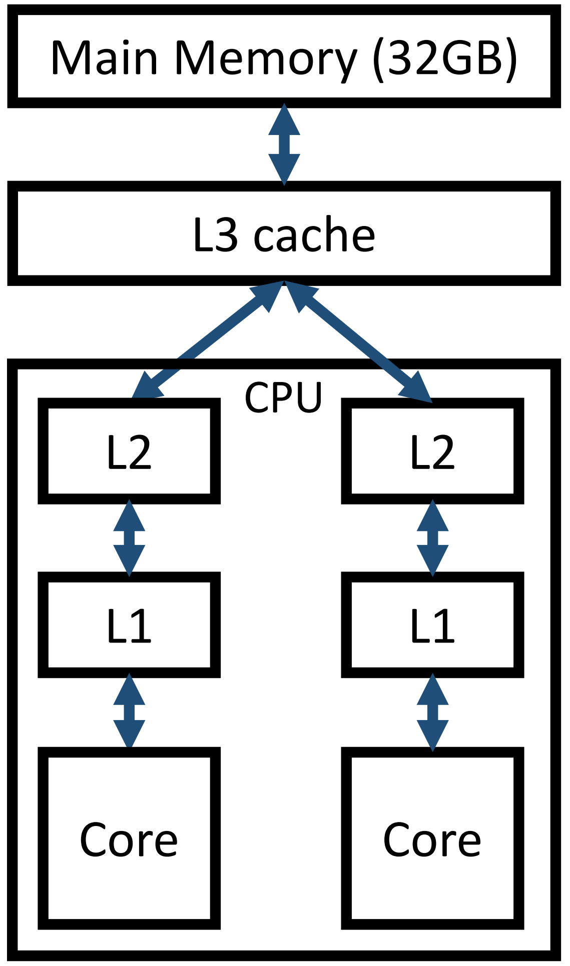
Parallel Programming

Previous weeks: Parallelism on the Java Level

Next: Parallelism on the Hardware Level

CPUs and Memory Hierarchies



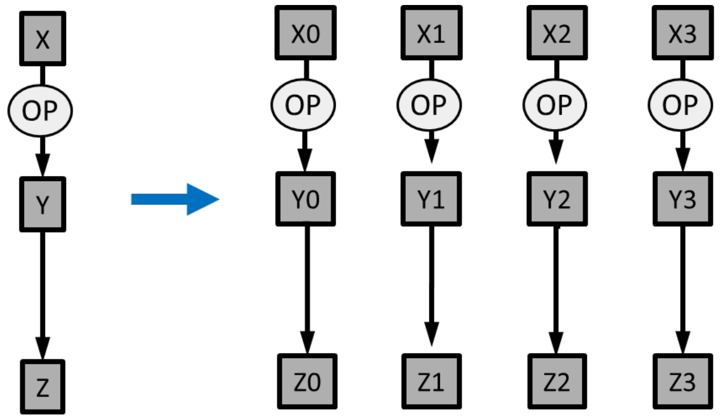
- Goal: allows cores to work in parallel, on their own, fast memory
- CPU reads/writes values from/to main memory, to compute with them, with a hierarchy of *memory caches* in between. *Faster* memory is more expensive, hence smaller: L1 is 5x faster than L2, which is 30x faster than main memory, which is 350x faster than disk
- Synchronisation *between* caches is taken care of by cache coherence protocols (e.g. MESI; typically implemented on the hardware level)
- Concurrency hazard: cores may pre-/postpone reads/writes from/to cache; memory barriers (special machine code instructions) needed to prevent problems with parallel code
  - Java: automatically inserted if, e.g. `synchronized` is used
  - C++: similar, but manual insertion also possible

3 approaches to apply parallelism to improve sequential processor performance

- *Vectorization:* Exposed to developers [last week]
- *Instruction Level Parallelism (ILP):* Inside CPU [last week]
- *Pipelining:* Also internal, but transfers to software [today]

Vectorization

- Goal: improve performance by using specialized *vector instructions*
- SIMD: **S**ingle **I**nstruction, applied to **M**ultiple **D**ata
- Requires *vectorised code:* code that uses the vector instructions provided by the target platform (CPU)
- Compilers (C++, JVM's JIT, …) attempt to detect vectorization opportunities → fully automated, but little or no control over if/where/how
- Platform-specific libraries (*intrinsics*, C/C++) expose vector instructions to developers → manual effort, but full control
- Poses no (additional) safety risks to concurrency



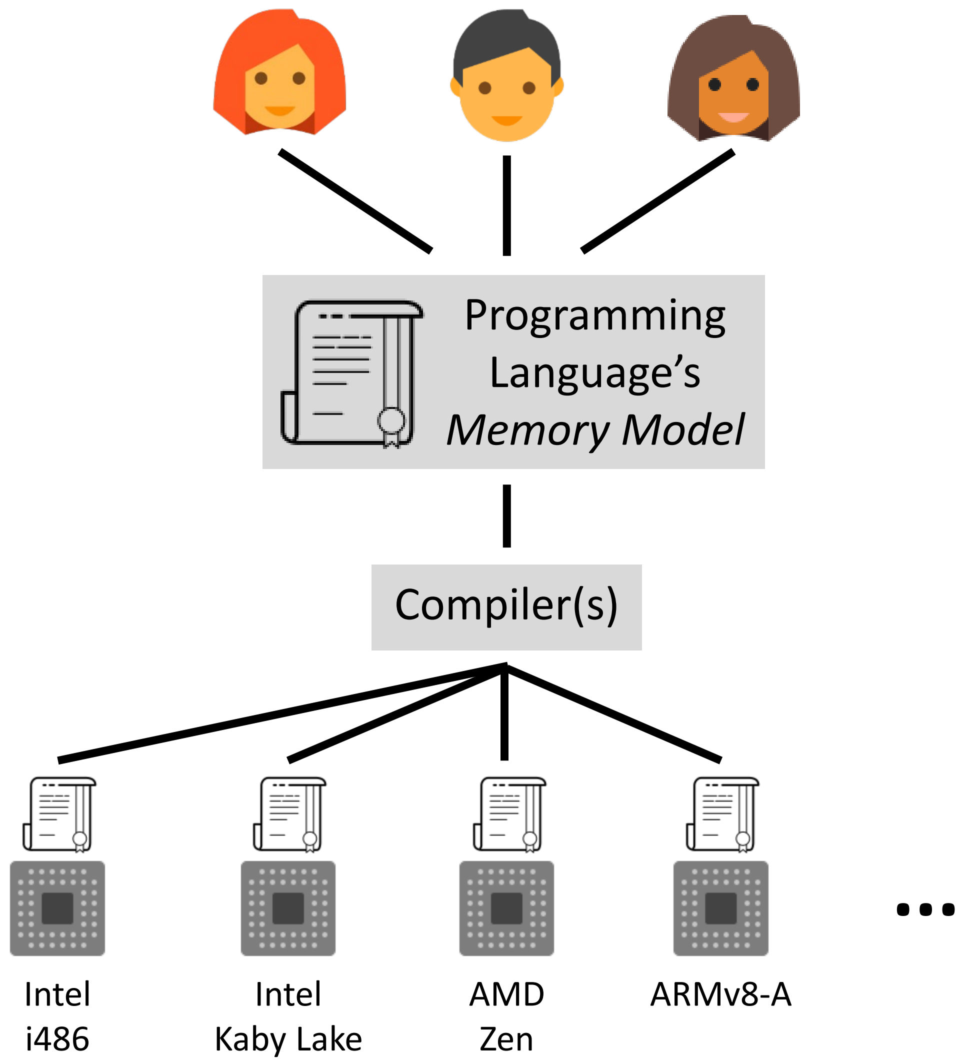
Instruction Level Parallelism (ILP)

```
1: x = a+b
2: y = c+d
3: if (p)
4:   z = e*f
5: else
6:   z = x*y
```

- Goal: improve CPU performance by internal parallelisation
- CPU/Core detects independent operations in its instruction stream (left: lines 1, 2)
- These may be executed in parallel inside the CPU, if enough functional units (e.g. floating-point unit, …) are available

- Various measures to increase potential for instruction parallelization. E.g. *speculatively execute* instructions in parallel (left: line 4, together with 1, 2), even if the result may not be used (left: if `p`, which may depend on lines 1,2, turns out to be false)
- Concurrency hazard: cores only locally consider dependencies in their instruction stream, not globally across all cores
    - Java: e.g. `synchronized` automatically adds memory barriers to prevent problematic reordering
    - C++: similar, but manual insertion also possible
- Compilers may also reorder instructions; similar problems, same solution (e.g. use `synchronized`)

Thread Safety Across Compilers and Platforms



- CPU designers specify CPU behaviour and guarantees (e.g. when do reorder, and how)
- A programming language's *memory model* specifies concurrency behaviour and guarantees (e.g. of a `synchronized` block)
- Developers program against the memory model
  - *Regular developers* "only" know guidelines, e.g. that shared data should only be accessed inside a synchronized block
  - *Expert developers* know the memory model and use special instructions (e.g. volatile), in particular in C++, to squeeze out performance
- Compilers know CPU specifications and enforce PL's memory model guarantees on each platform
- Bottom line:
  - Memory model protects developers from lower levels (compilers, hardware)
  - Concurrency guidelines sufficient for majority of applications/projects

`Icons (contract, people, cpu) by Icons8 (https://icons8.com)`