Programming Parallel Computers 2019

Jukka Suomela · Jaakko Lehtinen · Samuli Laine

Aalto University

ppc.cs.aalto.fi

week 3

Quick recap

Three forms of parallelism

Parallel computing resources

- ***Multicore:*** factor **4**
  - 4 cores, each of them can run independent threads
- ***Superscalar:*** factor **2**
  - each core can initiate 2 multiplications per clock cycle
- ***Pipelining:*** factor **4**
  - no need to wait for operations to finish before starting a new one
- ***Vectorization:*** factor **8**
  - each multiplication can process 8-wide vectors

Parallel computing resources

- ***Multicore:*** factor **4**
  - 4 cores, each of them can run independent threads
- ***Superscalar:*** factor **2**
  - each core can initiate 2 multiplications per clock cycle
- ***Pipelining:*** factor **4**
  - no need to wait for operations to finish before starting a new one
- ***Vectorization:*** factor **8**
  - each multiplication can process 8-wide vectors

OpenMP (week 2)

Parallel computing resources

- ***Multicore:*** factor **4**
  - 4 cores, each of them can run independent threads
- ***Superscalar:*** factor **2**
  - each core can initiate 2 multiplications per clock cycle
- ***Pipelining:*** factor **4**
  - no need to wait for operations to finish before starting a new one
- ***Vectorization:*** factor **8**
  - each multiplication can process 8-wide vectors

Instruction-level parallelism (week 1)

Parallel computing resources

- ***Multicore:*** factor **4**
  - 4 cores, each of them can run independent threads
- ***Superscalar:*** factor **2**
  - each core can initiate 2 multiplications per clock cycle
- ***Pipelining:*** factor **4**
  - no need to wait for operations to finish before starting a new one
- ***Vectorization:*** factor **8**
  - each multiplication can process 8-wide vectors

Vector instructions (week 2)

OpenMP parallel for loop

week 2

```
#pragma omp parallel for
for (int i = 0; i < 10; ++i) {
    c(i);
}
```

thread 0: `c(0)` `c(1)` `c(2)`

thread 1: `c(3)` `c(4)` `c(5)`

thread 2: `c(6)` `c(7)`

thread 3: `c(8)` `c(9)`

Instruction-level parallelism

week 1

Bad: dependent

```
a1 *= a0;
a2 *= a1;
a3 *= a2;
a4 *= a3;
a5 *= a4;
```

Good: independent

```
b1 *= a1;
b2 *= a2;
b3 *= a3;
b4 *= a4;
b5 *= a5;
```

Vector types

week 2

```
float8_t a, b, c;

a = ...;
b = ...;
c = a + b;
```

≈

```
float a[8], b[8], c[8];

a = ...;
b = ...;
c[0] = a[0] + b[0];
c[1] = a[1] + b[1];
c[2] = a[2] + b[2];
c[3] = a[3] + b[3];
c[4] = a[4] + b[4];
c[5] = a[5] + b[5];
c[6] = a[6] + b[6];
c[7] = a[7] + b[7];
```

Similar behavior, but much more efficient code: one vector addition

Performance in practice?

Is this enough?

Example

- "*Mandelbrot iteration*" for 512 values, *N* times:
  - `x = 0`
  - `c = input[i]`
  - ***repeat N times*:** `x = x * x + c`
  - `result[i] = x`
- Calculation for `input[i]`:
  - very long dependency chain, cannot parallelize
- Calculation for `input[i]` and `input[j]`:
  - ***independent, can be parallelized!***

```
for (int i = 0; i < 512; ++i) {

    float x = 0.0;
    float c = input[i];

    for (long long n = 0; n < N; ++n) {

            x = x * x + c;

    }

    result[i] = x;

}
```

Naive sequential version

```
#pragma omp parallel for
for (int i = 0; i < 8; ++i) {
    float8_t c[8], x[8];
    for (int j = 0; j < 8; ++j) {
        x[j] = float8_0; c[j] = input[i][j];
    }
    for (long long n = 0; n < N; ++n) {
        for (int j = 0; j < 8; ++j) {
            x[j] = x[j] * x[j] + c[j];
        }
    }
    for (int j = 0; j < 8; ++j) {
        result[i][j] = x[j];
    }
}
```

Fully parallelized version

```
#pragma omp parallel for
for (int i = 0; i < 8; ++i) {
    float8_t c[8], x[8];
    for (int j = 0; j < 8; ++j) {
        x[j] = float8_0; c[j] = input[i][j];
    }
    for (long long n = 0; n < N; ++n) {
        for (int j = 0; j < 8; ++j) {
            x[j] = x[j] * x[j] + c[j];
        }
    }
    for (int j = 0; j < 8; ++j) {
        result[i][j] = x[j];
    }
}
```

Using 4 threads evenly

Plenty of room for instruction-level parallelism here

8-wide vector operations

"input" and "result" are here 8×8×8 arrays

Performance?

- N = 1 billion
  - we do **1024 billion** arithmetic operations
  - running time on **3.3 GHz 4-core** Skylake CPU: **2.44** seconds
- Got: **420** billion single-precision arithmetic operations / second

Happy?

Performance!

- N = 1 billion
  - we do **1024 billion** arithmetic operations
  - running time on **3.3 GHz 4-core** Skylake CPU: 2.44 seconds
- Got: **420** billion single-precision arithmetic operations / second
- Theoretical maximum for this CPU: ≈ **422** billion / second

Yes!

Cheating?

- Tiny input, tiny output
- Everything in inner loops fits in CPU registers
- ***No memory accesses in inner loops***

- Also: CPUs are very good at this kind of operations
  - key operation: **FMA**, *fused multiply and add*
  - single instruction for **`d = a * b + c`**

Main memory is slow

Large latency, small throughput

Throughput example

CPU

- 64 additions per clock cycle
  - one addition: two floats
  - float: 4 bytes
- ***512 bytes of input data*** per clock cycle to arithmetic units

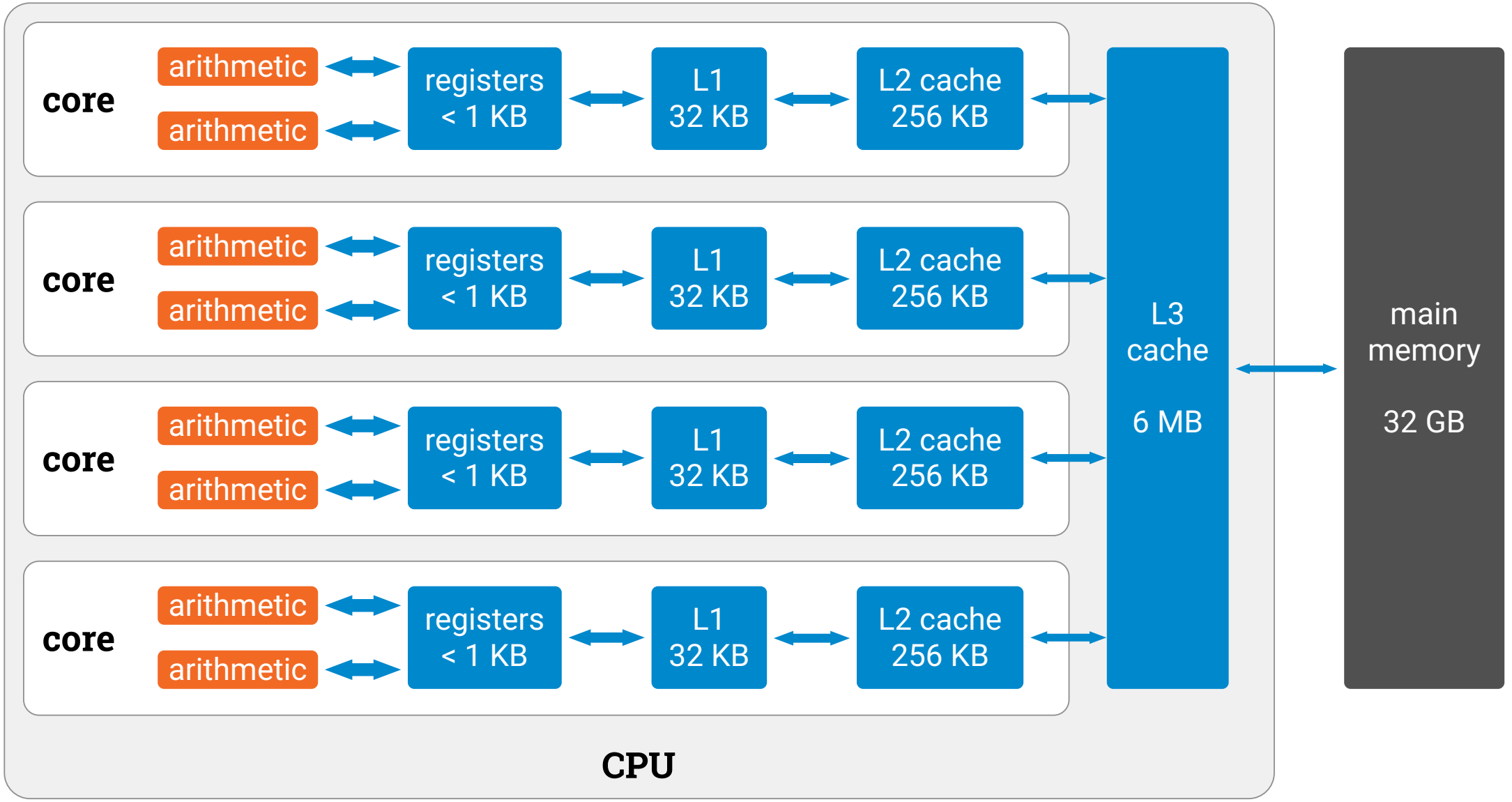
Main memory

- Bandwidth 34.1 GB/s
  - clock speed 3.3 GHz
- ***10 bytes of input data*** per clock cycle from memory

Factor 50 difference!

Main memory is slow

- Typical: **factor-50 difference** between these:
  - how much data CPU could “consume”
  - how much data main memory could “provide”
- We can only afford to fetch ***2% of our input*** from main memory!
- Everything else has to come from:
  - **CPU registers**
  - **caches** (preferably those close to the CPU)


core
arithmetic
arithmetic
registers
< 1 KB
L1
32 KB
L2 cache
256 KB
core
arithmetic
arithmetic
registers
< 1 KB
L1
32 KB
L2 cache
256 KB
core
arithmetic
arithmetic
registers
< 1 KB
L1
32 KB
L2 cache
256 KB
core
arithmetic
arithmetic
registers
< 1 KB
L1
32 KB
L2 cache
256 KB
L3
cache
6 MB
main
memory
32 GB
CPU

How do registers work?

- Your ***C++ compiler*** will keep all kinds of ***local variables*** and temporary values in registers whenever it can
  - it tries very hard, just make sure there are no obstacles for that
- **Main limitations:**
  - you cannot have ***"pointers to registers"***
    - `float x; float *p = &x; something(p);`
  - you cannot do ***"array indexing with registers"***
    - `float y[4]; int i = calculate(); float z = y[i & 3];`
  - there are not ***that many registers***
    - `float8_t a[1000];`

What is kept in the main memory?

- ***Everything that is not in the registers***
- Everything that you allocate from the **heap** (malloc, new …)
  - input data, output data, temporary storage …
- Everything that is in the **stack**
  - e.g. return addresses in recursive functions …
- **Program code** that the CPU is running
- **Page tables**: mapping between virtual memory addresses and physical memory addresses

How do caches work?

- When your program reads **anything** that is in the main memory:
  - ***CPU tries to get it from L1 cache***
  - if not there, try L2 cache
  - if not there, try L3 cache
  - if not there, get from main memory
  - ***CPU automatically stores it in caches*** if it was not there yet
  - makes space by throwing away some not-so-recently-used values
- Smallest meaningful unit of data: **cache line = 64 bytes**

Data reuse

Read once, use many times

Reuse data in registers

- **Bad:**
  - read x and y from memory to registers
  - calculate $x + y$
  - throw away x and y
- **Slightly better:**
  - read x_1, x_2, y_1, y_2 from memory to registers
  - calculate $x_1 + y_1$, $x_1 + y_2$, $x_2 + y_1$, $x_2 + y_2$
  - throw away x_1, x_2, y_1, y_2

Arithmetic intensity:

1 op : 2 reads

Arithmetic intensity:

4 ops : 4 reads

Sample application: 2-hop paths

```
// a = pointer to a row of data
// b = pointer to a row of transpose
for (int k = 0; k < n; ++k) {
    float x = a[k];
    float y = b[k];
    v = min(v, x + y);
}
```

Calculating 1 unit of result

No reuse of memory reads

```
// a0, a1 = pointers to two rows of data
// b = pointer to a row of transpose
for (int k = 0; k < n; ++k) {
    float x = a0[k];
    float y = b[k];
    v0 = min(v0, x + y);
}
for (int k = 0; k < n; ++k) {
    float x = a1[k];
    float y = b[k];
    v1 = min(v1, x + y);
}
```

Calculating 2 units of result

No reuse of memory reads

```
for (int k = 0; k < n; ++k) {
    float x = a0[k];
    float y = b[k];
    v0 = min(v0, x + y);
}
for (int k = 0; k < n; ++k) {
    float x = a1[k];
    float y = b[k];
    v1 = min(v1, x + y);
}
```

```
for (int k = 0; k < n; ++k) {
    float x0 = a0[k];
    float x1 = a1[k];
    float y = b[k];
    v0 = min(v0, x0 + y);
    v1 = min(v1, x1 + y);
}
```

Calculating 2 units of result

```
for (int k = 0; k < n; ++k) {
    float x = a0[k];
    float y = b0[k];
    v00 = min(v00, x + y);
}
for (int k = 0; k < n; ++k) {
    float x = a0[k];
    float y = b1[k];
    v01 = min(v01, x + y);
}
for (int k = 0; k < n; ++k) {
    float x = a1[k];
    float y = b0[k];
    v10 = min(v10, x + y);
} ...
```

```
for (int k = 0; k < n; ++k) {
    float x0 = a0[k];
    float x1 = a1[k];
    float y0 = b0[k];
    float y1 = b1[k];
    v00 = min(v00, x0 + y0);
    v01 = min(v01, x0 + y1);
    v10 = min(v10, x1 + y0);
    v11 = min(v11, x1 + y1);
}
```

Calculating
2 × 2 units of result

```
for (int k = 0; k < n; ++k) {
    float x = a0[k];
    float y = b0[k];
    v00 = min(v00, x + y);
}
for (int k = 0; k < n; ++k) {
    float x = a0[k];
    float y = b1[k];
    v01 = min(v01, x + y);
}
for (int k = 0; k < n; ++k) {
    float x = a1[k];
    float y = b0[k];
    v10 = min(v10, x + y);
} ...
```

```
for (int k = 0; k < n; ++k) {
    float x0 = a0[k];
    float x1 = a1[k];
    float y0 = b0[k];
    float y1 = b1[k];
    v00 = min(v00, x0 + y0);
    v01 = min(v01, x0 + y1);
    v10 = min(v10, x1 + y0);
    v11 = min(v11, x1 + y1);
}
```

50 % less memory reads

Also much better for ILP!

Reuse data in registers

- **Version 4** in course material:
  - read 3 + 3 vectors
  - update 3 × 3 vector results
  - 6 + 9 = 15 vector registers
  - CPU: 16 vector registers
- ***Number of memory reads*** decreases by factor of 3
- Also plenty of opportunities for ***instruction-level parallelism***

```
for (int ka = 0; ka < na; ++ka) {
    float8_t y0 = vt[...];
    float8_t y1 = vt[...];
    float8_t y2 = vt[...];
    float8_t x0 = vd[...];
    float8_t x1 = vd[...];
    float8_t x2 = vd[...];
    vv00 = min8(vv00, x0 + y0);
    vv01 = min8(vv01, x0 + y1);
    vv02 = min8(vv02, x0 + y2);
    vv10 = min8(vv10, x1 + y0);
    vv11 = min8(vv11, x1 + y1);
    vv12 = min8(vv12, x1 + y2);
    vv20 = min8(vv20, x2 + y0);
    vv21 = min8(vv21, x2 + y1);
    vv22 = min8(vv22, x2 + y2);
}
```

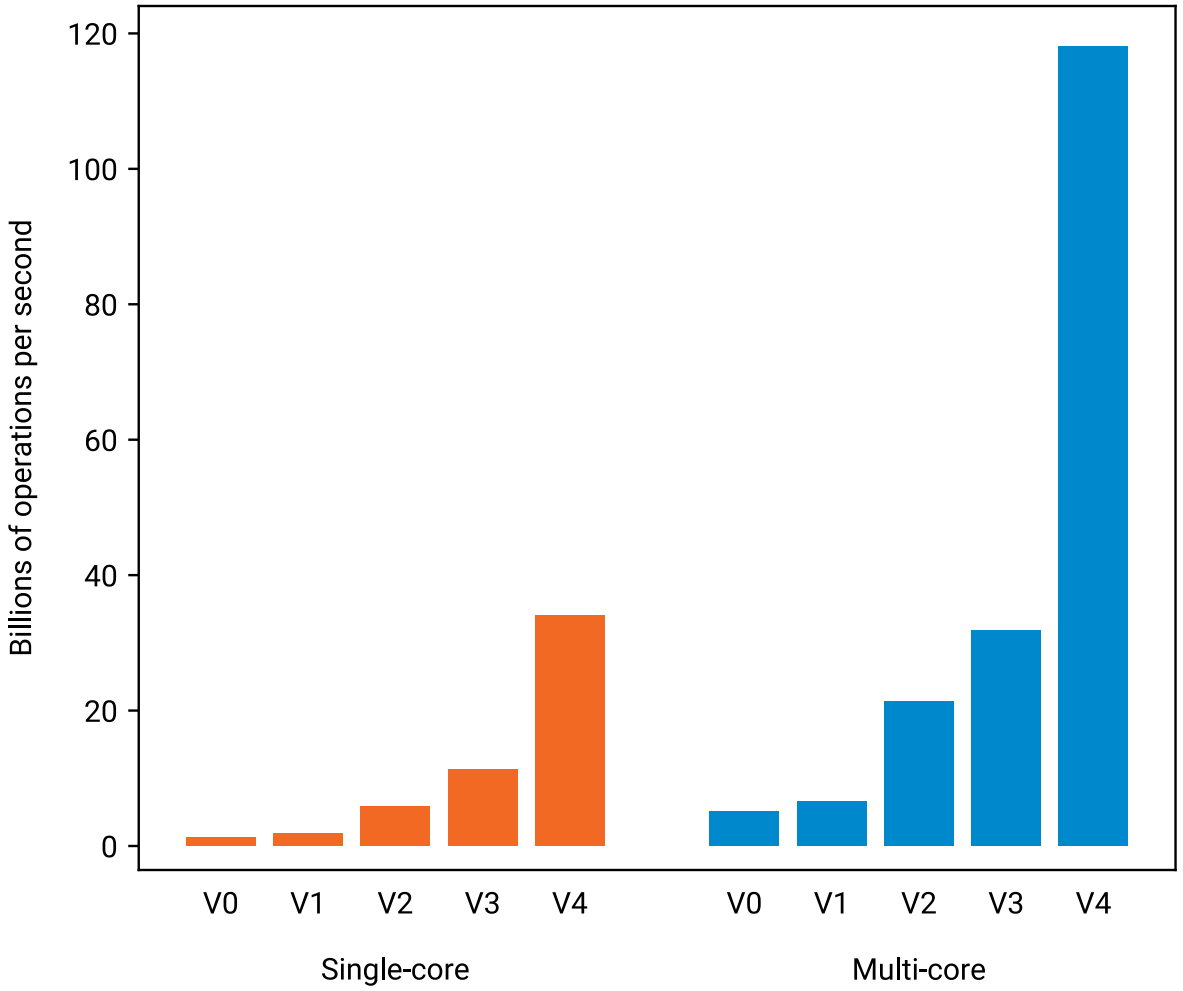
Reuse data in registers

V2: instruction-level parallelism

V3: vectorization

V4: reuse data

Running time improved from **99 s** to **1 s**



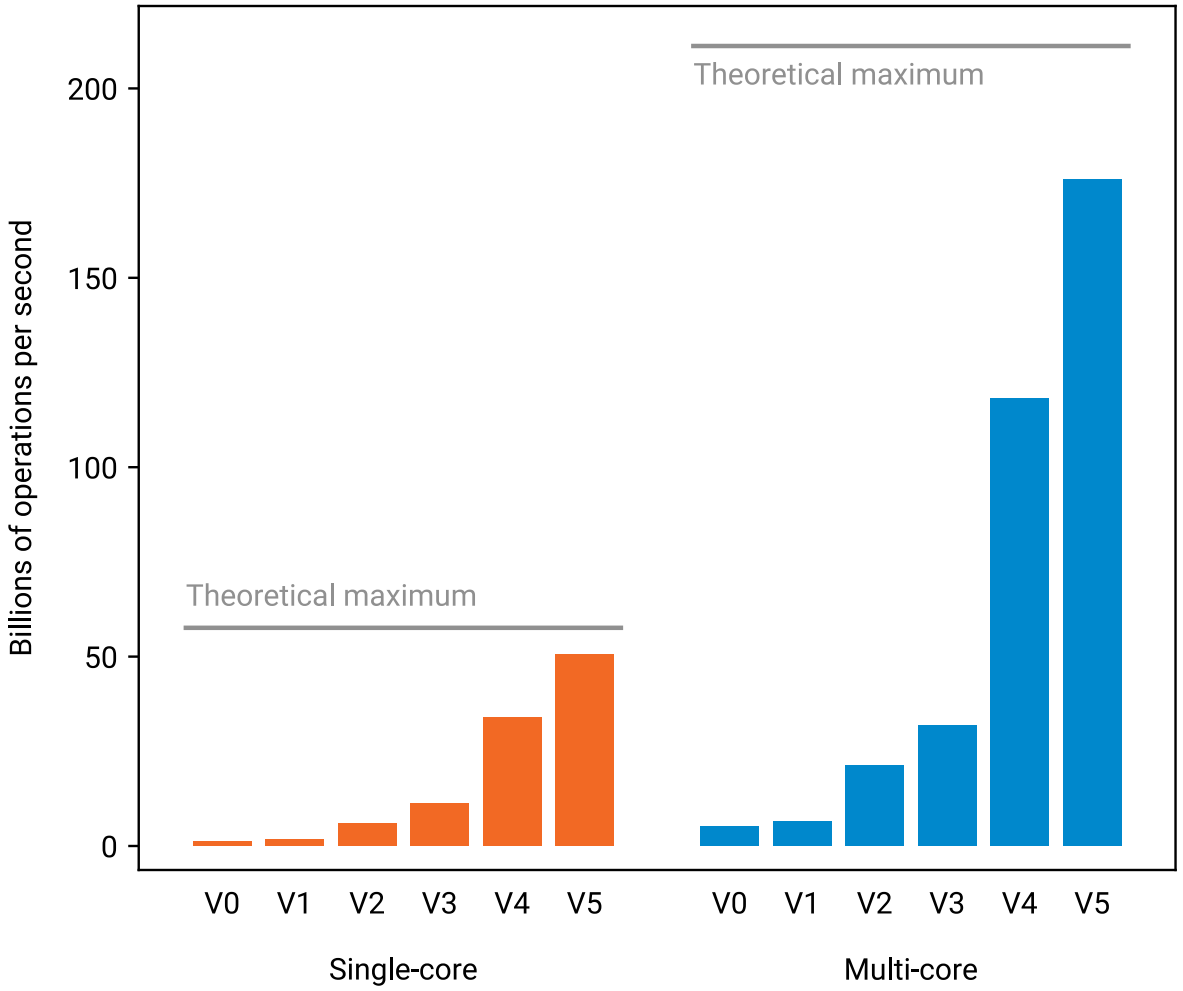
Reuse data in registers

V2: instruction-level parallelism

V3: vectorization

V4: reuse data

V5: more data reuse…



Reusing data in caches

- We got nice performance by using registers as very fast “cache”
  - **V5:** 8 arithmetic operations : 1 memory read
- We already “accidentally” benefit from caches, too
  - repeatedly scan the same block of rows
- Could we **design** it so that we explicitly take cache memory into account?

Animations:

ppc.cs.aalto.fi/cache1/
ppc.cs.aalto.fi/cache2/

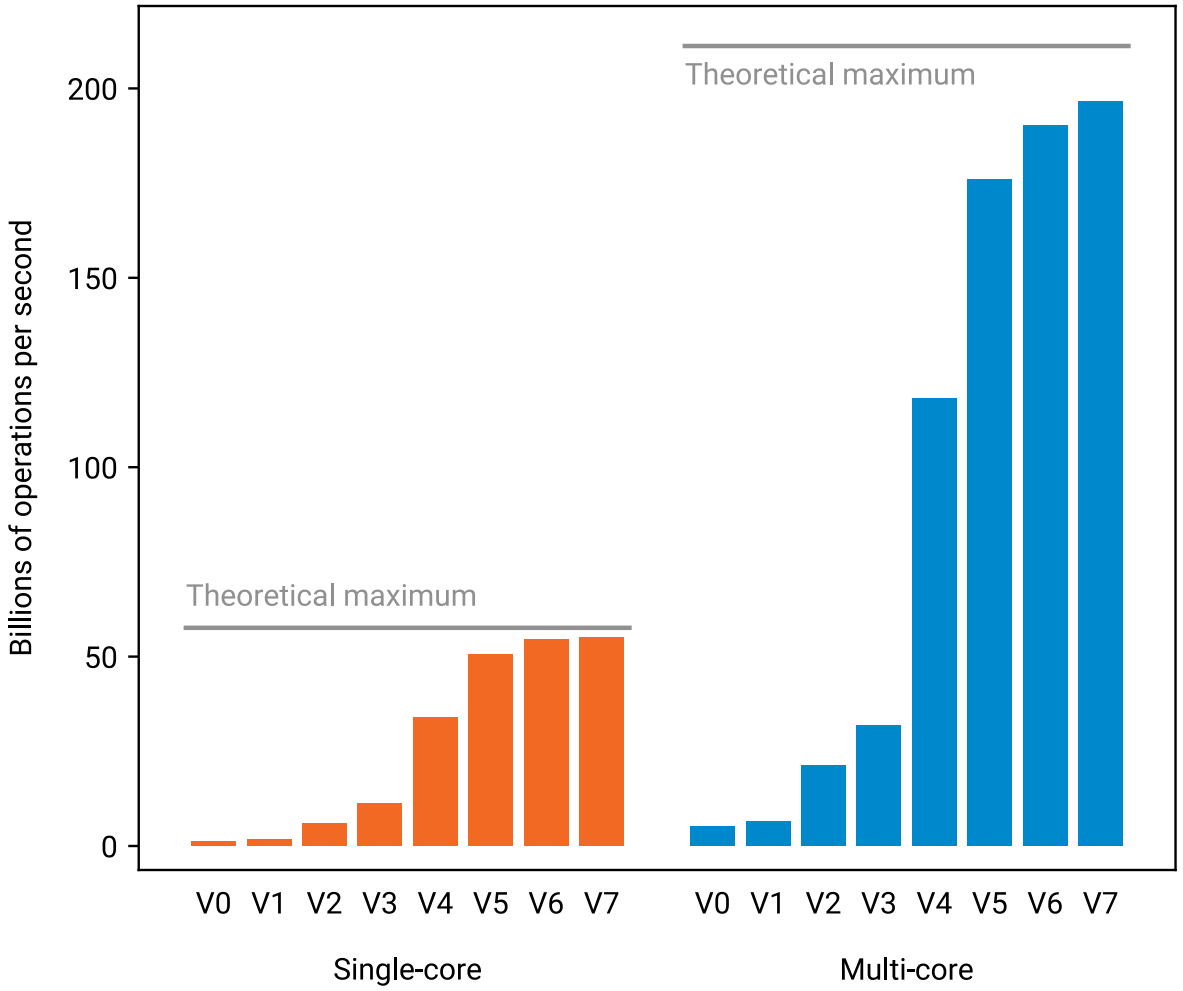
Putting it together

Baseline: **99 s**

Final: **0.7 s**

Factor-151 speedup

93% of theoretical maximum



Summary

Pay attention to memory lookups

Performance engineering challenges

- Using all parallel resources
  - ***multicore parallelism***: multithreading with OpenMP
  - ***instruction-level parallelism***: independent operations
  - ***vector units***: vector types
- Making sure CPU has enough data to process
  - ***reusing data*** in registers
  - choosing memory access patterns that benefit from caches
  - linear reading
  - using entire cache line for something useful

Course: what next

- **Chapter 2:** case study of everything we have covered so far
  - all forms of parallelism + reusing data
  - enough to solve e.g. **CP1**, **CP2abc**, **CP3ab**, **MF1**, **MF2** …
- **Chapter 3:** more about OpenMP
  - what else is there beyond **#pragma omp parallel for**
  - very helpful in e.g. **SO4**, **SO5**
- **Chapter 4:** GPU programming
  - our topic for ***next week's lecture***
  - needed for e.g. **CP4**, **CP5** …

Good for self-study

Course: what next

- All modern computers have at least two processors:
  - ***CPU***: what we have been using so far
  - ***GPU***: lots of more computing power available and we can use it, too!
- CPU on Maari computers :
  - **460 billion** single-precision FLOPS (floating-point operations / second)
  - 230 billion double-precision FLOPS
- GPU on Maari computers:
  - **1400 billion** single-precision FLOPS
  - 45 billion double-precision FLOPS

How to use it? Week 4...

Questions?