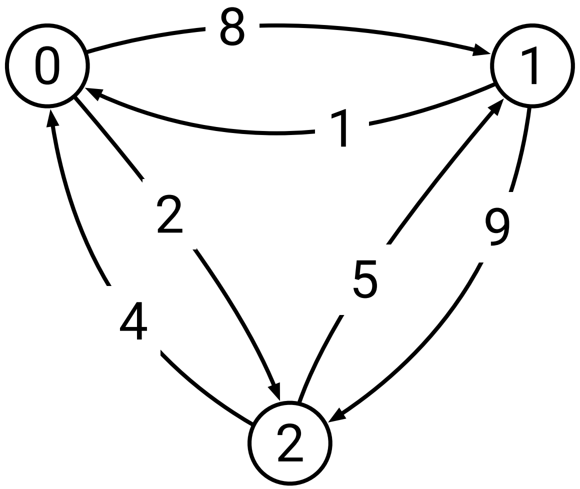
Programming Parallel Computers

Jukka Suomela · Aalto University · ppc.cs.aalto.fi

Part 1C:
Sample application · Memory access pattern

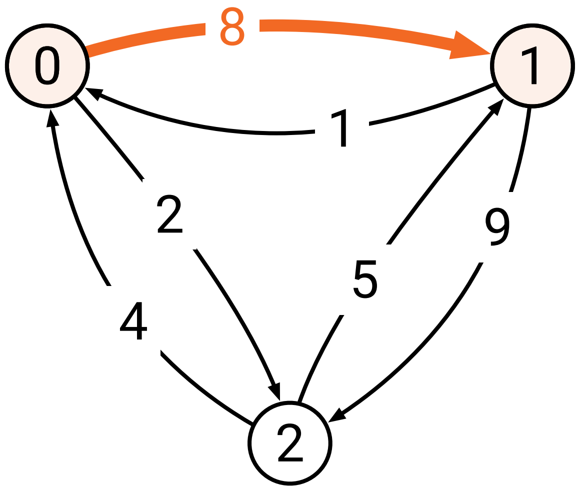
Sample application: cheapest 2-hop path

d (input):



Sample application: cheapest 2-hop path

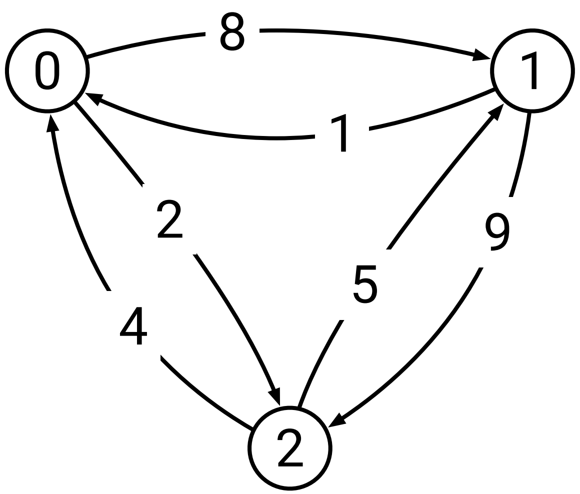
d (input):

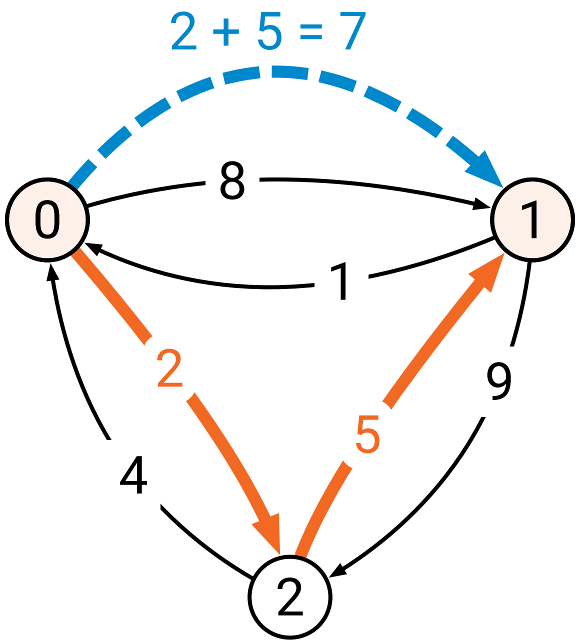


Cost of traveling directly 0 → 1

Sample application: cheapest 2-hop path

d (input):

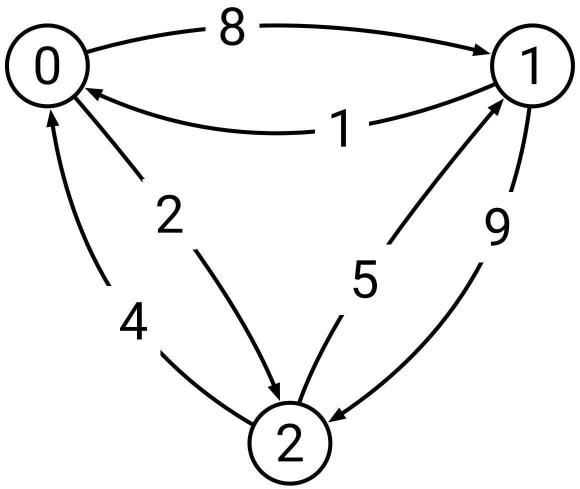


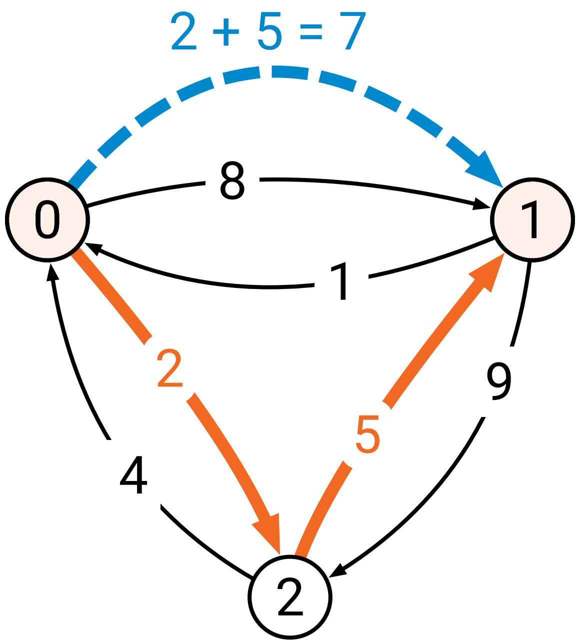


Cost of traveling
0 → 2 → 1

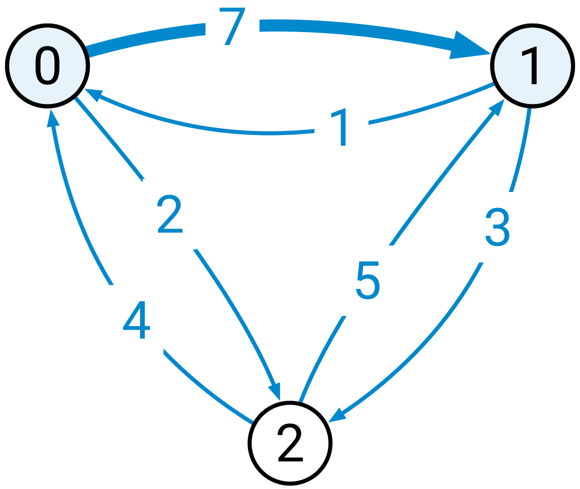
Sample application: cheapest 2-hop path

d (input):



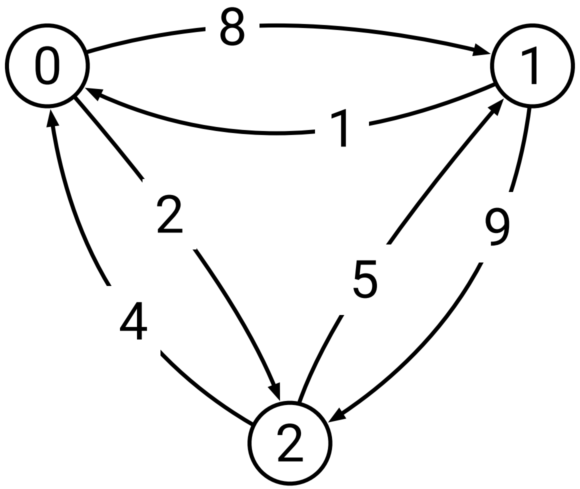


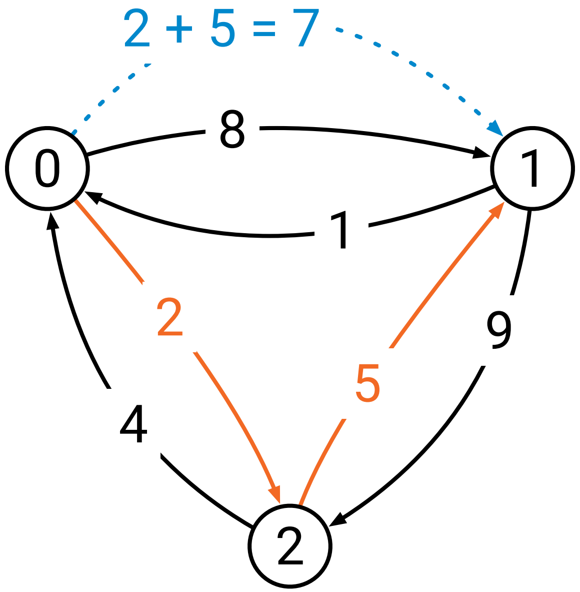
r (output):



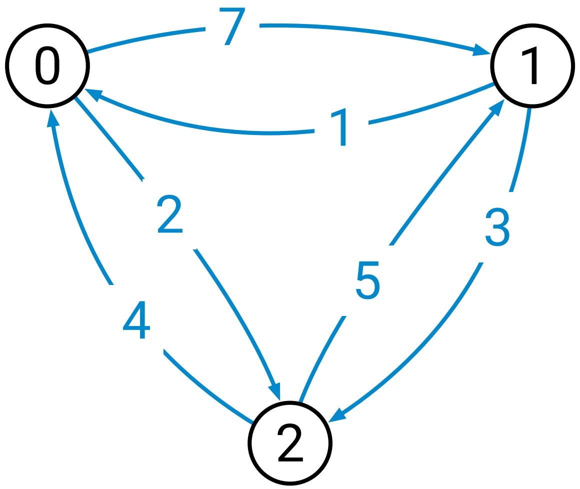
Sample application: cheapest 2-hop path

d (input):





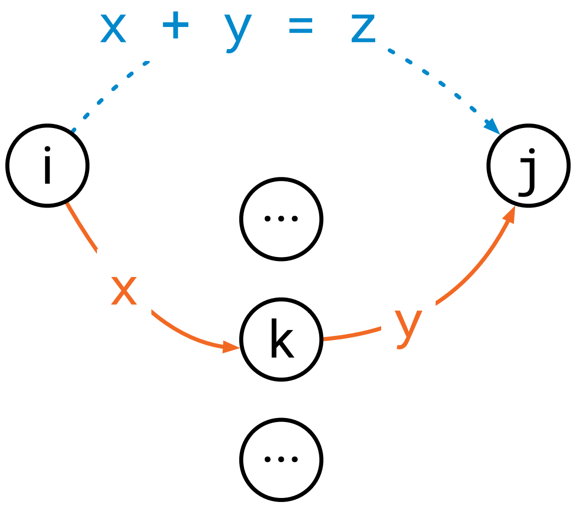
r (output):




```
d[] = { 0, 8, 2,
        1, 0, 9,
        4, 5, 0 }
```

```
r[] = { 0, 7, 2,
        1, 0, 3,
        4, 5, 0 }
```

```
void step(float* r, const float* d, int n) {
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            float v = infinity;
            for (int k = 0; k < n; ++k) {
                float x = d[n*i + k];
                float y = d[n*k + j];
                float z = x + y;
                v = min(v, z);
            }
            r[n*i + j] = v;
        }
    }
}
```



Is it fast?

- Benchmark platform: **4-core Intel “Skylake” CPU**
  - 3.2–3.6 GHz
  - Linux, GCC, `g++ -O3 -march=native`
- Benchmark instance: ***n* = 4000**
  - 64 billion “+” operations and 64 billion “min” operations
- Running time: ***99 seconds***
  - 1.3 billion useful operations per second
  - ***0.36 useful operations per clock cycle***

Is it fast?

- Benchmark platform: **4-core Intel “Skylake” CPU**
  - 3.2–3.6 GHz
  - Linux, GCC, `g++ -O3 -march=native`
- Benchmark instance: ***n* = 4000**
  - 64 billion “+” operations and 64 billion “min” operations
- Running time: ***99 seconds***
  - 1.3 billion useful operations per second
  - ***0.36 useful operations per clock cycle***

We are using roughly 0.6% of the performance of the CPU

```
void step(float* r, const float* d, int n) {
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            float v = infinity;
            for (int k = 0; k < n; ++k) {
                float x = d[n*i + k];
                float y = d[n*k + j];
                float z = x + y;
                v = min(v, z);
            }
            r[n*i + j] = v;
        }
    }
}
```

What went wrong here?

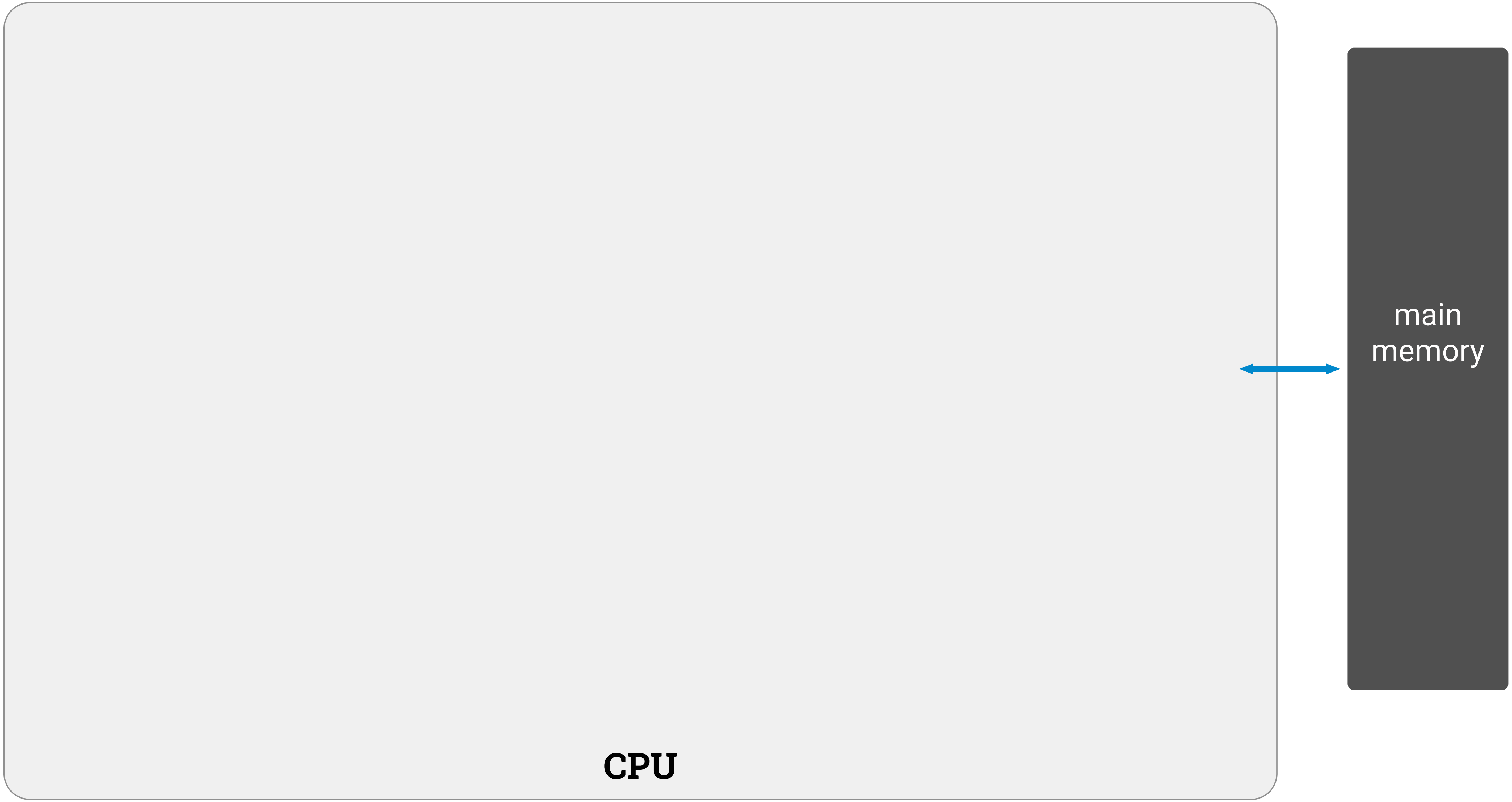
What went wrong?

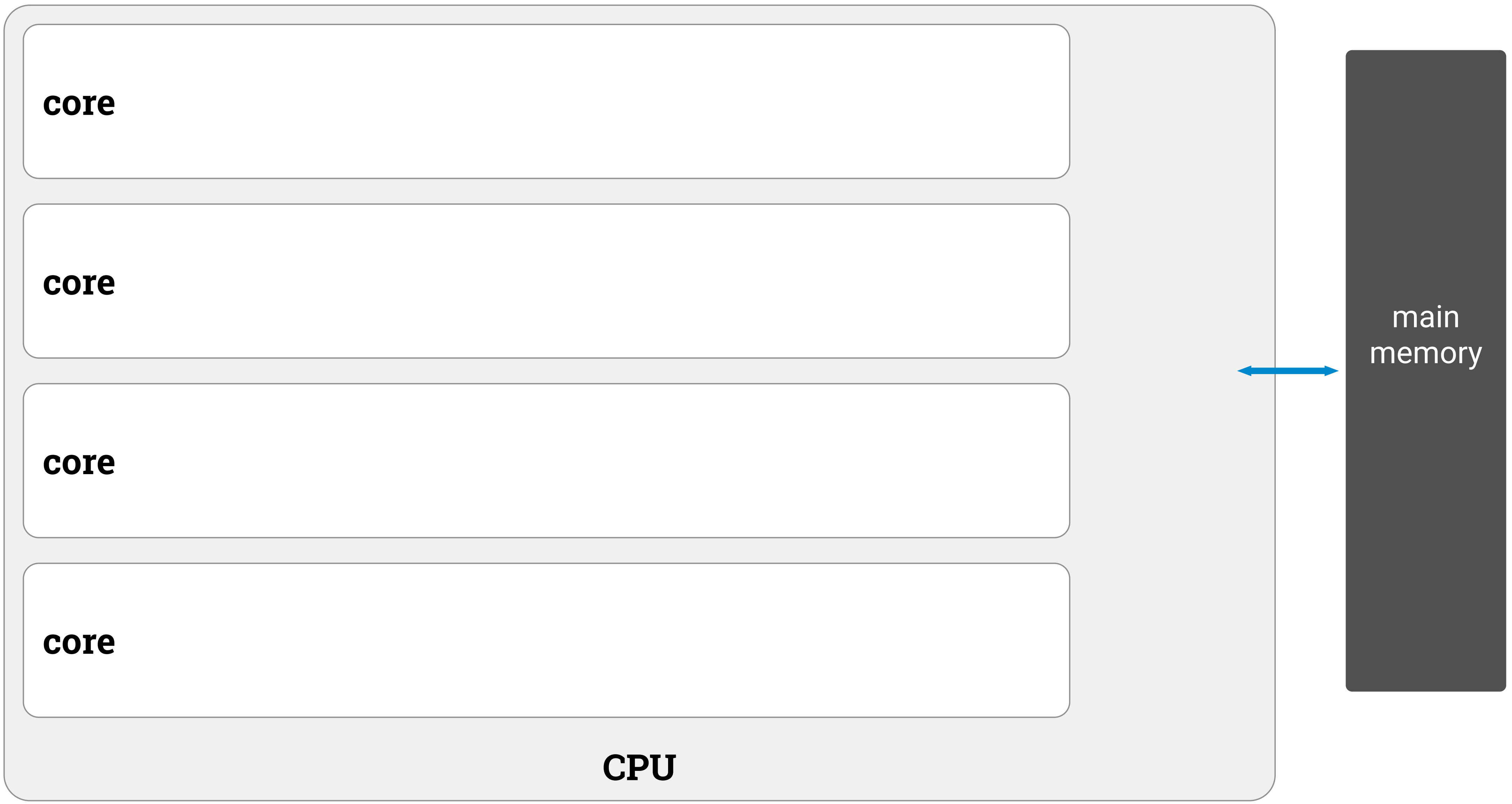
- It is ***not any single thing***
  - there is no magic quick fix
  - take care of one bottleneck and there is another one
- But it ***does not need to be hard***
  - not that much work to improve running time from minutes to seconds
  - it can really be worth the effort!
- And ***almost everything is possible***
  - if we really want, we can engineer a solution that is **150 times faster** and uses **93%** (or more?) of the processing power of the CPU

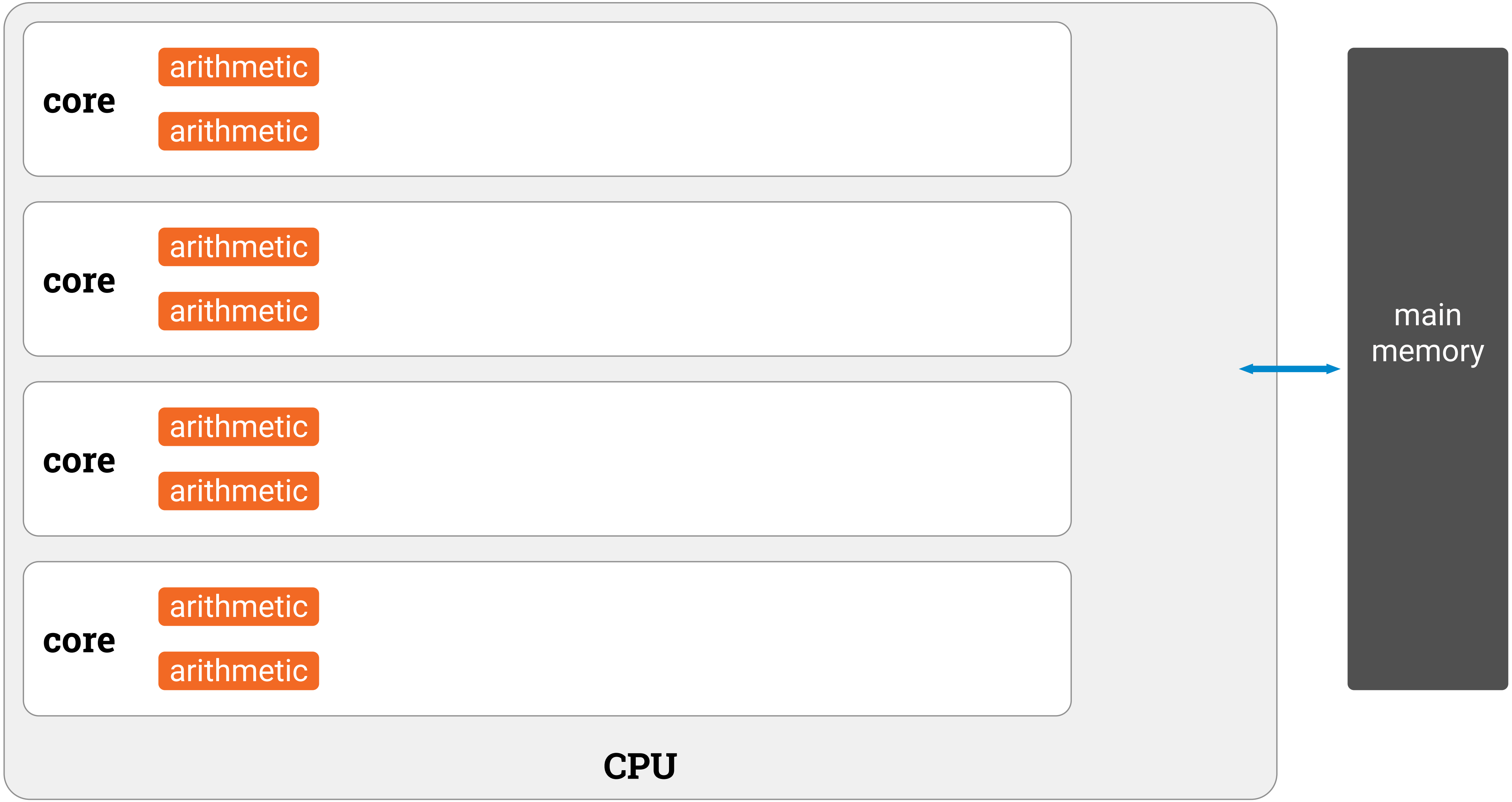
Two main challenges

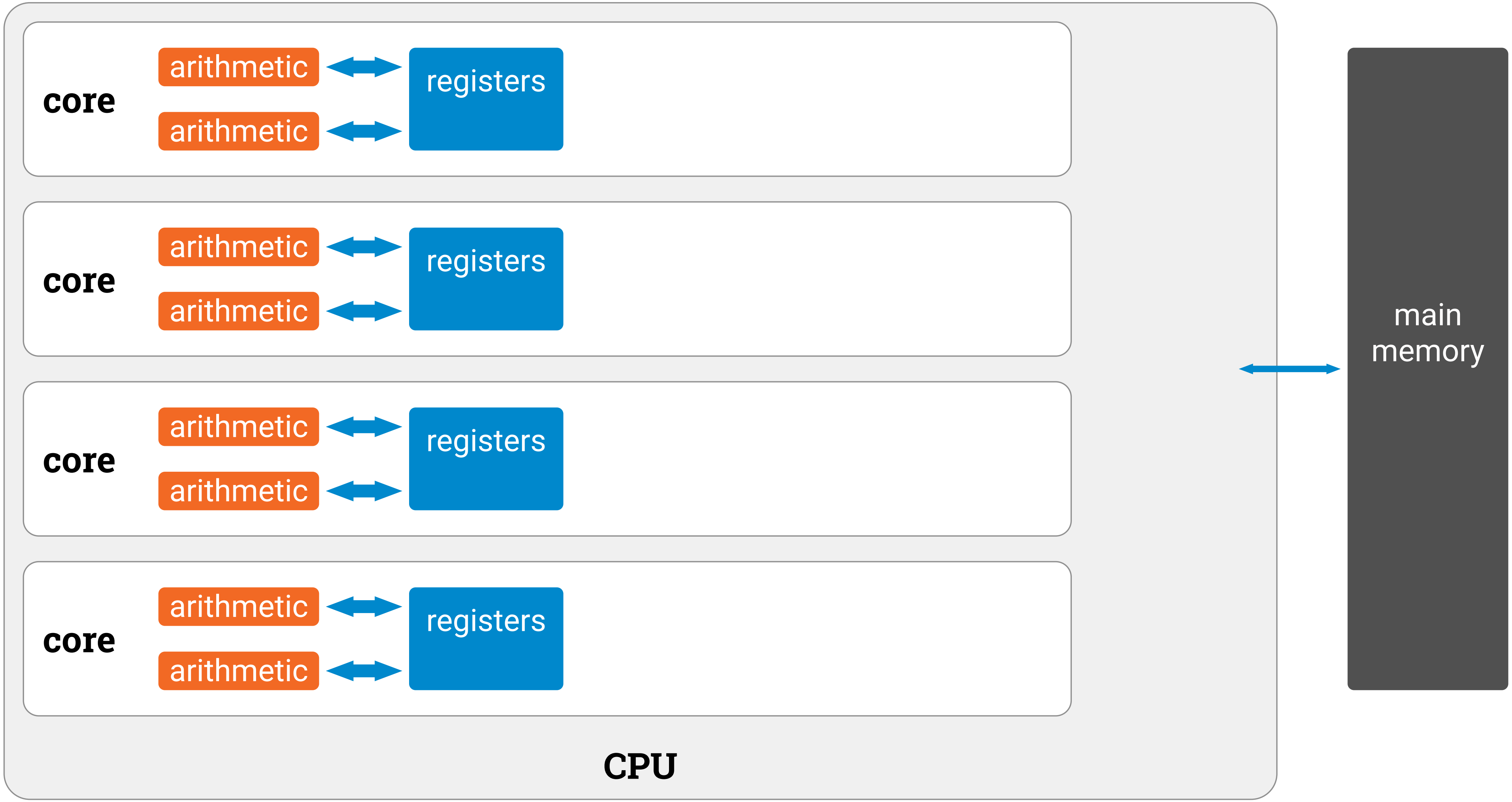
- How to get data fast enough ***from main memory to CPU?***
- Once the data is there, how to ***do lots of things in parallel?***

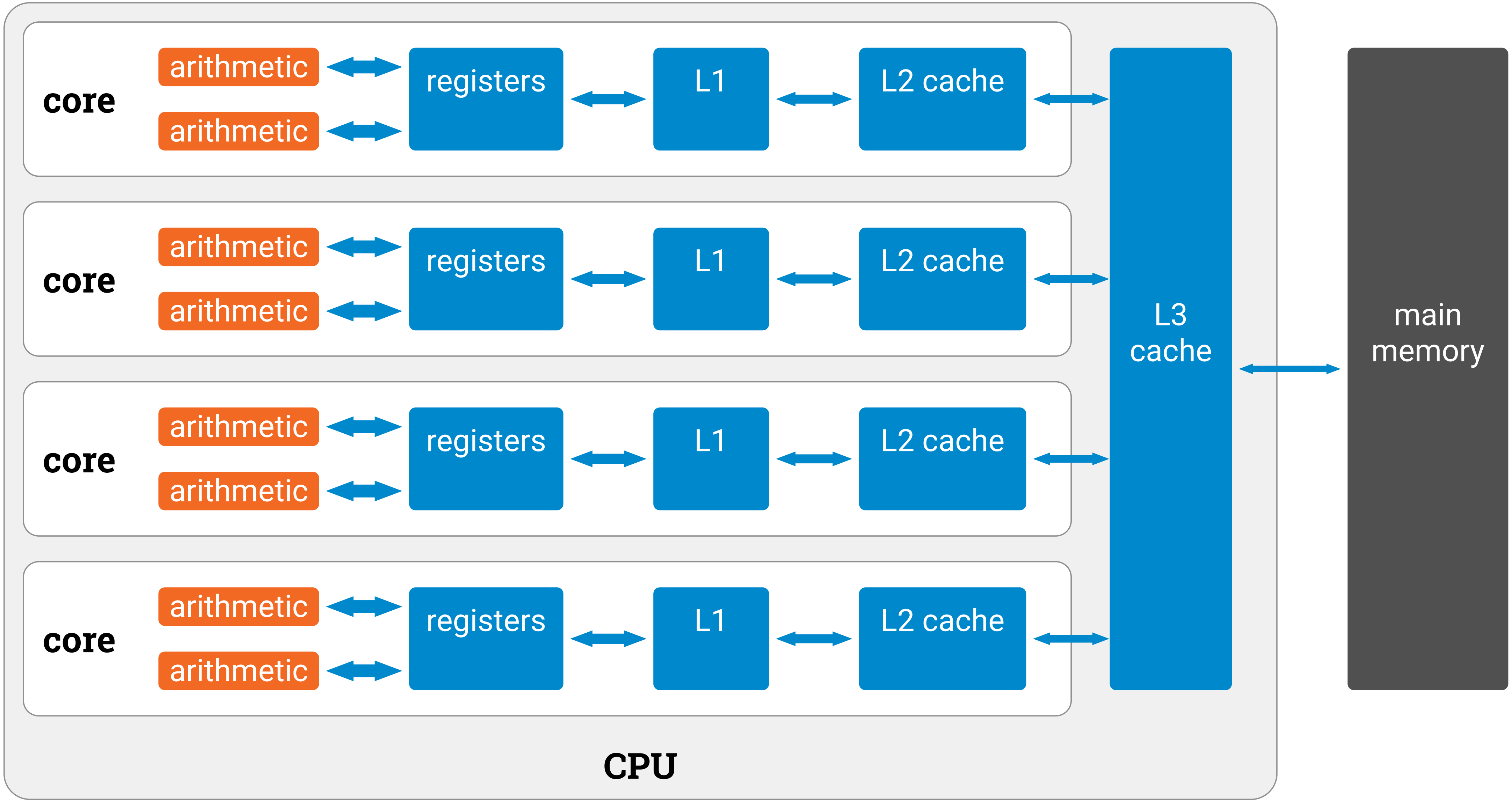
CPU

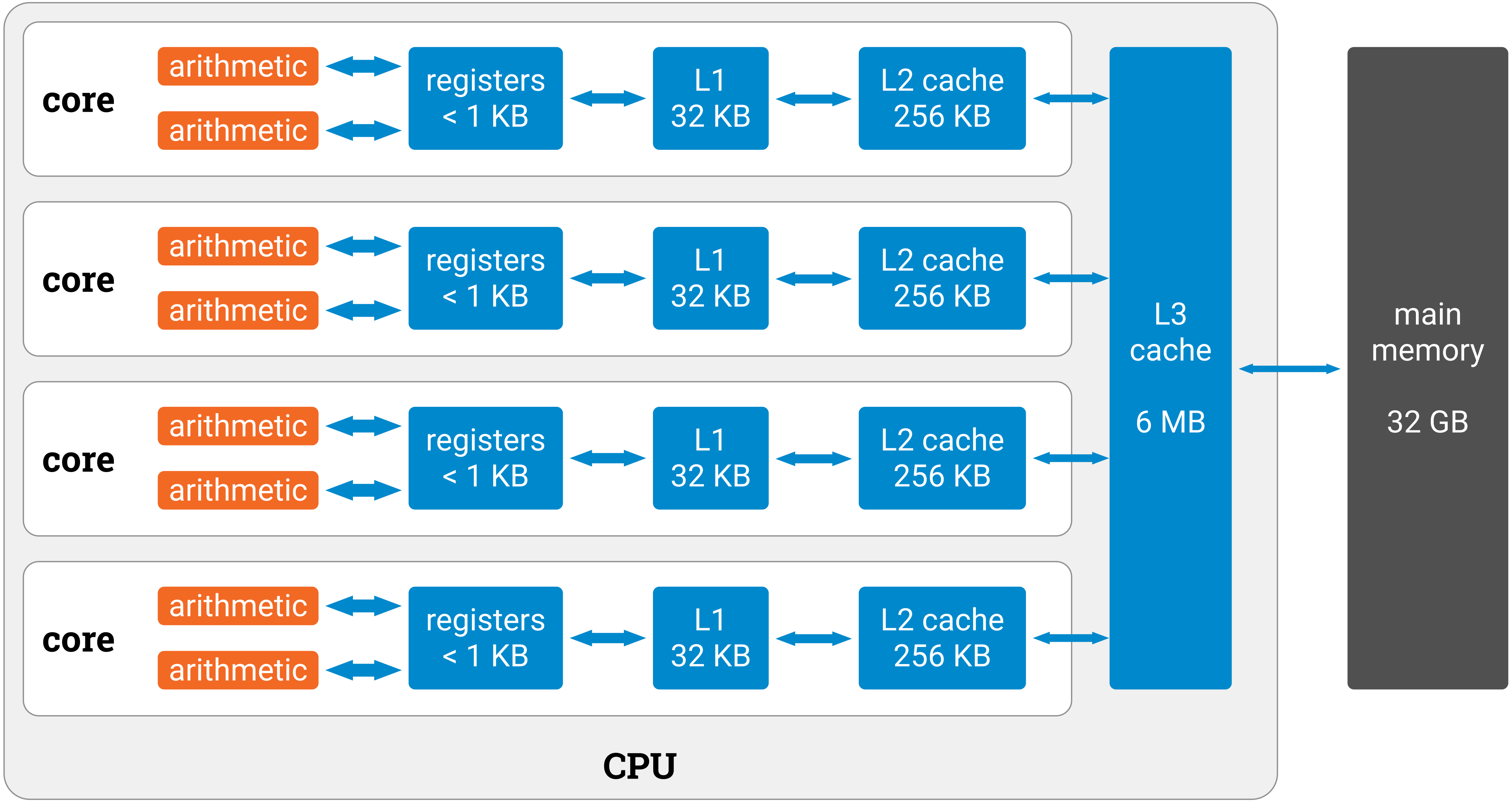

main
memory
CPU


core
core
core
core
CPU
main
memory


core
arithmetic
arithmetic
core
arithmetic
arithmetic
core
arithmetic
arithmetic
core
arithmetic
arithmetic
CPU
main memory


core
arithmetic
arithmetic
registers
core
arithmetic
arithmetic
registers
core
arithmetic
arithmetic
registers
core
arithmetic
arithmetic
registers
CPU
main memory


core
arithmetic
arithmetic
registers
L1
L2 cache
core
arithmetic
arithmetic
registers
L1
L2 cache
core
arithmetic
arithmetic
registers
L1
L2 cache
core
arithmetic
arithmetic
registers
L1
L2 cache
L3 cache
main memory
CPU

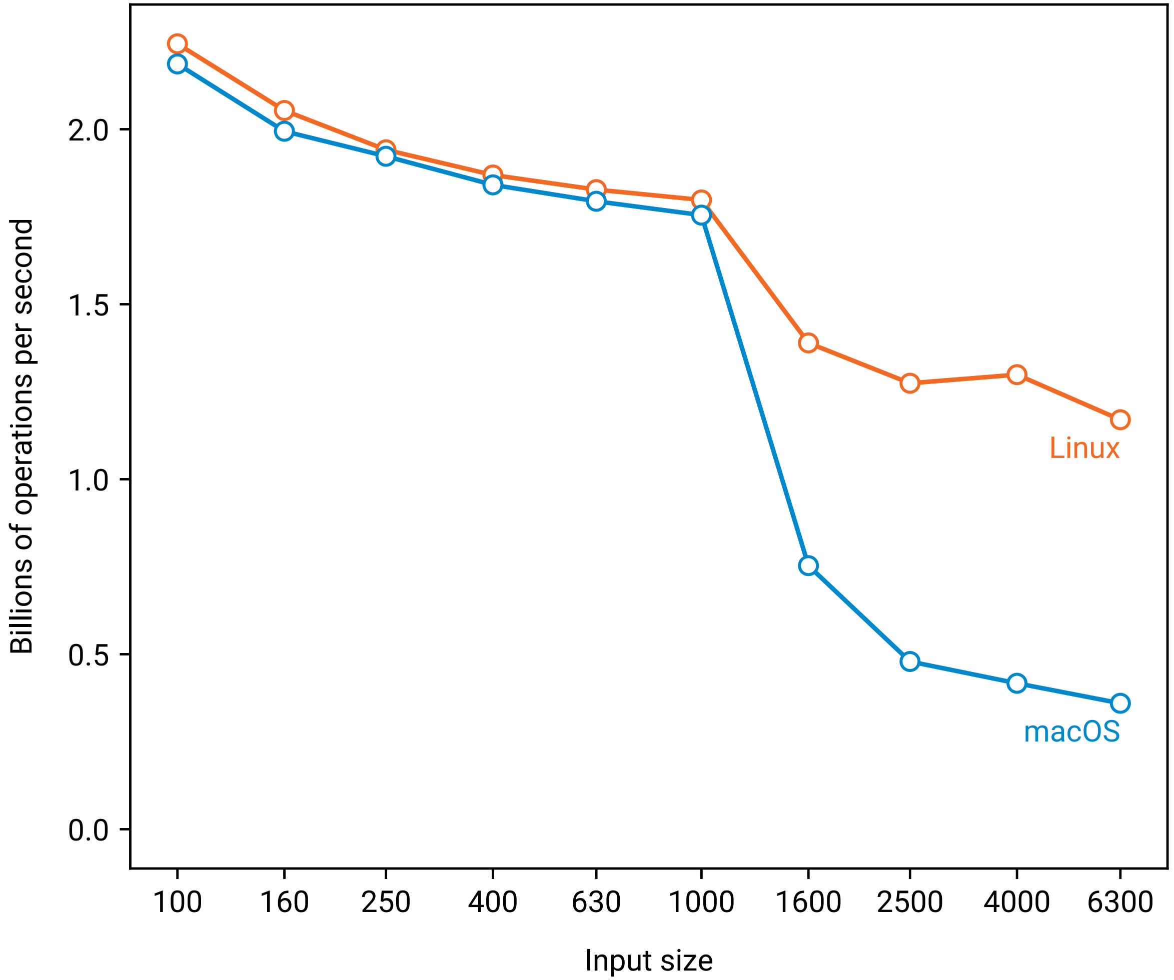

core
arithmetic
arithmetic
registers
< 1 KB
L1
32 KB
L2 cache
256 KB
core
arithmetic
arithmetic
registers
< 1 KB
L1
32 KB
L2 cache
256 KB
core
arithmetic
arithmetic
registers
< 1 KB
L1
32 KB
L2 cache
256 KB
core
arithmetic
arithmetic
registers
< 1 KB
L1
32 KB
L2 cache
256 KB
L3
cache
6 MB
main
memory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

Two main challenges

- How to get data fast enough ***from main memory to CPU?***
  - **high latency:** fetching one unit of data takes a lot of time
  - **low throughput:** there is not that much bandwidth available
- Once the data is there, how to ***do lots of things in parallel?***
  - high arithmetic throughput, but **how to exploit it**?
  - a typical C++ program might use just one arithmetic unit at a time, in a highly sequential manner
  - how to use all arithmetic units efficiently?

Current bottleneck?

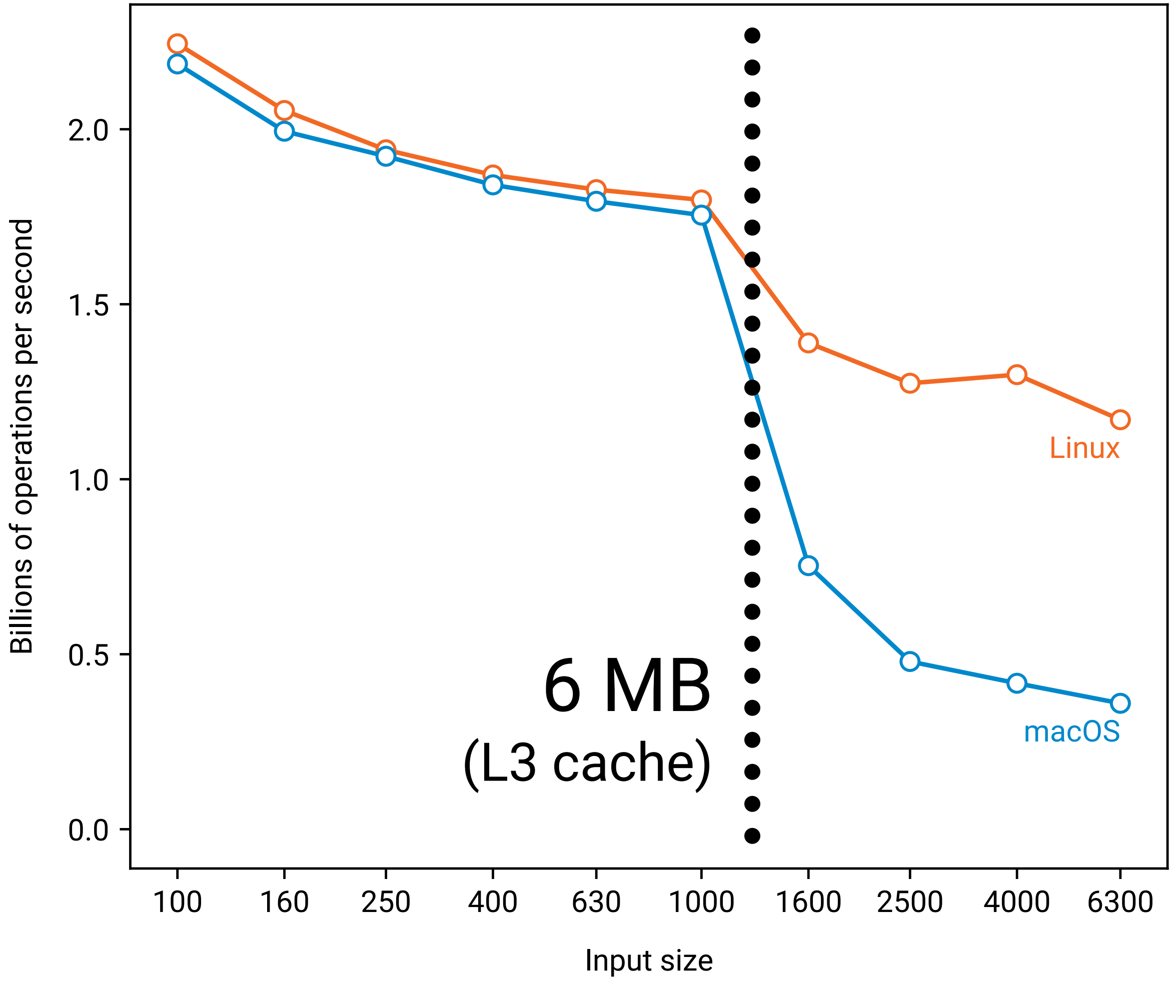
Performance as a function of input size



Current bottleneck?

Performance as a function of input size

Difficulties getting data from memory to CPU once we run out of L3 cache



```
void step(float* r, const float* d, int n) {
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            float v = infinity;
            for (int k = 0; k < n; ++k) {
                float x = d[n*i + k];
                float y = d[n*k + j];
                float z = x + y;
                v = min(v, z);
            }
            r[n*i + j] = v;
        }
    }
}
```

Innermost loop

Memory access pattern

```
for (int k = 0; k < n; ++k) {
    float x = d[n*i + k]; // d[0], d[1], d[2], ...
    float y = d[n*k + j]; // d[0], d[4000], d[8000], ...
    float z = x + y;
    v = min(v, z);
}
```

Memory access pattern

```
for (int k = 0; k < n; ++k) {
    float x = d[n*i + k]; // d[0], d[1], d[2], ...
    float y = d[n*k + j]; // d[0], d[4000], d[8000], ...
    float z = x + y;
    v = min(v, z);
}
```

Rule of thumb: linear scanning is good

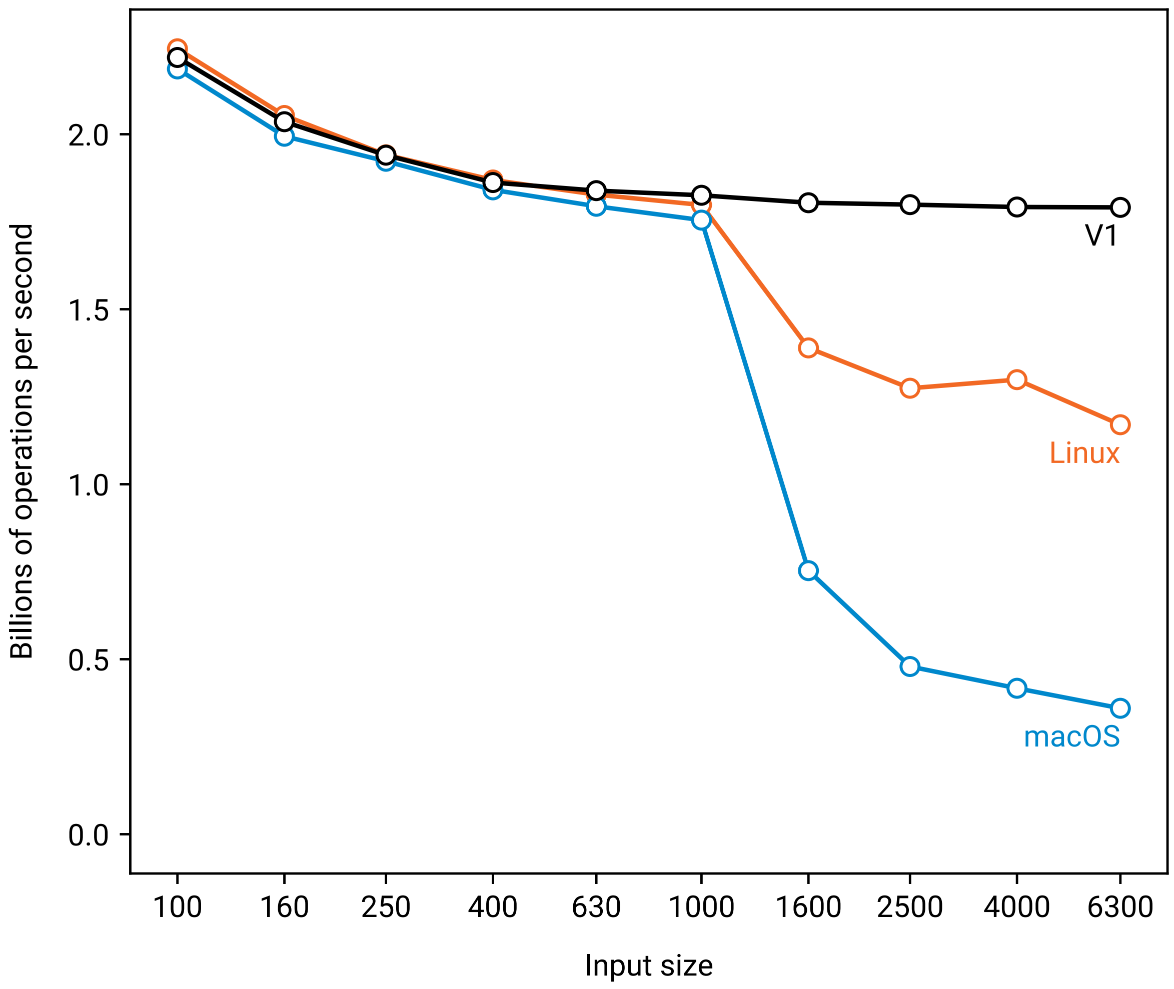
Memory access pattern

```
for (int k = 0; k < n; ++k) {
    float x = d[n*i + k]; // d[0], d[1], d[2], ...
    ~~float y = d[n*k + j]; // d[0], d[4000], d[8000], ...~~
    float y = t[n*j + k]; // t[0], t[1], t[2], ...
    float z = x + y;
    v = min(v, z);
}
```

Array t = transpose of array d

Current bottleneck?

It no longer matters where the input data is



Current bottleneck?

It no longer matters where the input data is

Problem: calculations done in a ***sequential*** order