Programming Parallel Computers

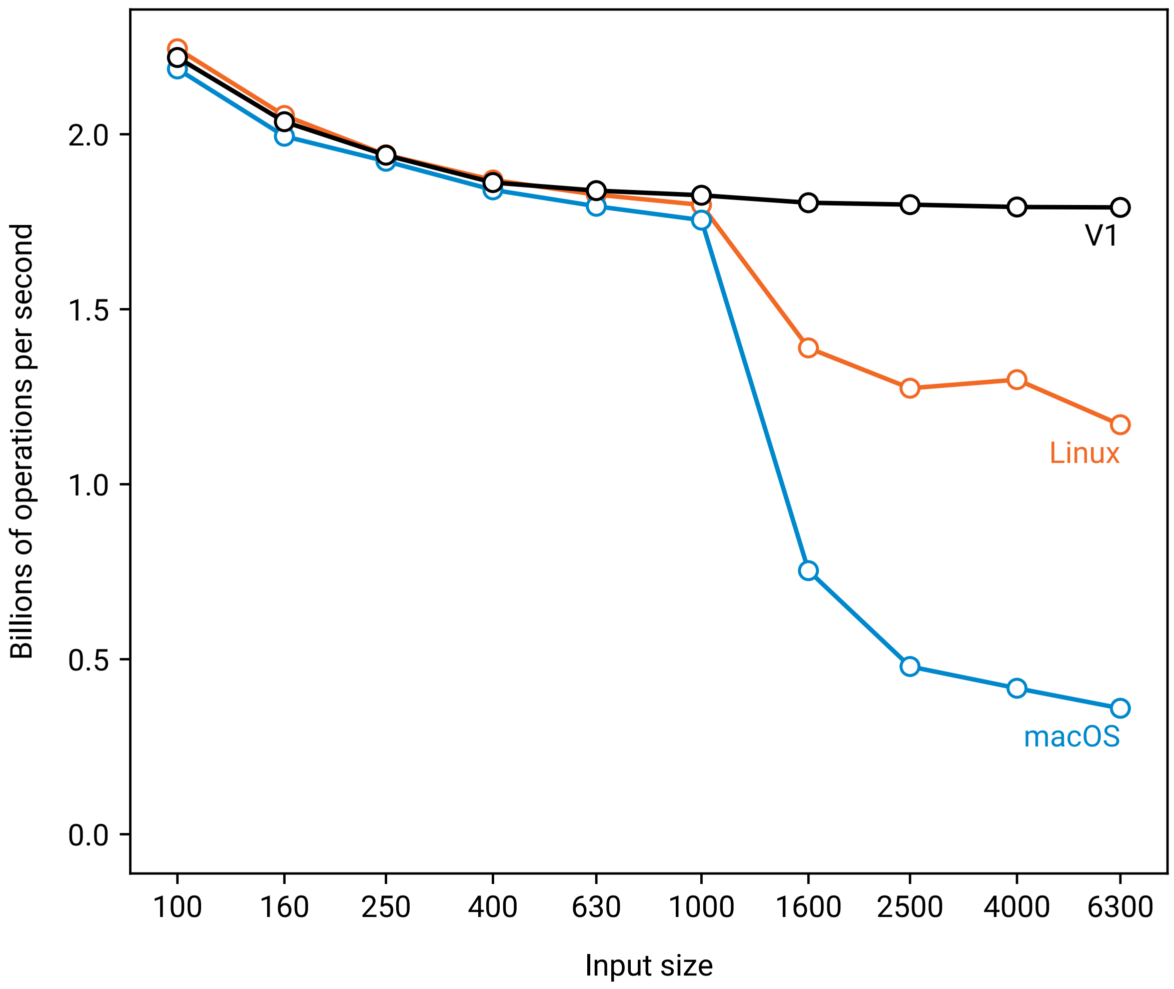
Jukka Suomela · Aalto University · ppc.cs.aalto.fi

Part 1D: Instruction-level parallelism

Current bottleneck?

It no longer matters where the input data is

Problem: calculations done in a sequential order



Innermost loop

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

Innermost loop

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

```
x0 = d[... + 0];
y0 = t[... + 0];
z0 = x0 + y0;
v  = min(v, z0);
x1 = d[... + 1];
y1 = t[... + 1];
z1 = x1 + y1;
v  = min(v, z1);
x2 = d[... + 2];
y2 = t[... + 2];
z2 = x2 + y2;
v  = min(v, z2);
...
```

Innermost loop

- Dependency chain
    - cannot start the next “min” operation until we know the result of the previous “min” operation
- Cost of each iteration:
  ≥ latency of the “min” operation

```
x0 = d[... + 0];
y0 = t[... + 0];
z0 = x0 + y0;
v = min(v, z0);
x1 = d[... + 1];
y1 = t[... + 1];
z1 = x1 + y1;
v = min(v, z1);
x2 = d[... + 2];
y2 = t[... + 2];
z2 = x2 + y2;
v = min(v, z2);
...
```

Innermost loop

- Dependency chain
- Cost of each iteration:
  ≥ latency of the "min" operation
- Benchmarks:
  ≈ 4 clock cycles per iteration
- Latency of the "**`vminss`**" instruction:
  4 clock cycles

```
x0 = d[... + 0];
y0 = t[... + 0];
z0 = x0 + y0;
v = min(v, z0);
x1 = d[... + 1];
y1 = t[... + 1];
z1 = x1 + y1;
v = min(v, z1);
x2 = d[... + 2];
y2 = t[... + 2];
z2 = x2 + y2;
v = min(v, z2);
...
```

Dependency chain

"**min**": associative, commutative

Freedom to rearrange operations:

$$\min(\min(\min(z_0, z_1), z_2), z_3)$$

Inherently sequential, no room for parallelism

$$=$$

$$\min(\min(z_0, z_2), \min(z_1, z_3))$$

Two independent operations, could be computed in parallel

Dependency chain

Accumulate two minimums:

- `v0` = minimum of even elements
- `v1` = minimum of odd elements

We could at least in principle do **two “min” operations in parallel?**

```
v = min(v, z0);
v = min(v, z1);
v = min(v, z2);
...
```

```
v0 = min(v0, z0);
v1 = min(v1, z1);
v0 = min(v0, z2);
v1 = min(v1, z3);
v0 = min(v0, z4);
v1 = min(v1, z5);
...
v = min(v0, v1);
```

Dependency chain

Accumulate three minimums:

- `v0` = minimum of elements 0 mod 3
- `v1` = minimum of elements 1 mod 3
- `v2` = minimum of elements 2 mod 3

We could at least in principle do **three "min" operations in parallel?**

```
v = min(v, z0);
v = min(v, z1);
v = min(v, z2);
...
```

```
v0 = min(v0, z0);
v1 = min(v1, z1);
v2 = min(v2, z2);
v0 = min(v0, z3);
v1 = min(v1, z4);
v2 = min(v2, z5);
...
v = min(v0,v1,v2);
```

```
float w[4] = ...
for (int k = 0; k < n/4; ++k) {
    for (int m = 0; m < 4; ++m) {
        float x = d[n*i + k*4 + m];
        float y = t[n*j + k*4 + m];
        float z = x + y;
        w[m] = min(w[m], z);
    }
}
v = min(w[0], w[1],
        w[2], w[3]);
```

4 times more potential for parallelism

```
float w[4] = ...
for (int k = 0; k < n/4; ++k) {
    for (int m = 0; m < 4; ++m) {
        float x = d[n*i + k*4 + m];
        float y = t[n*j + k*4 + m];
        float z = x + y;
        w[m] = min(w[m], z);
    }
}
v = min(w[0], w[1],
        w[2], w[3]);
```

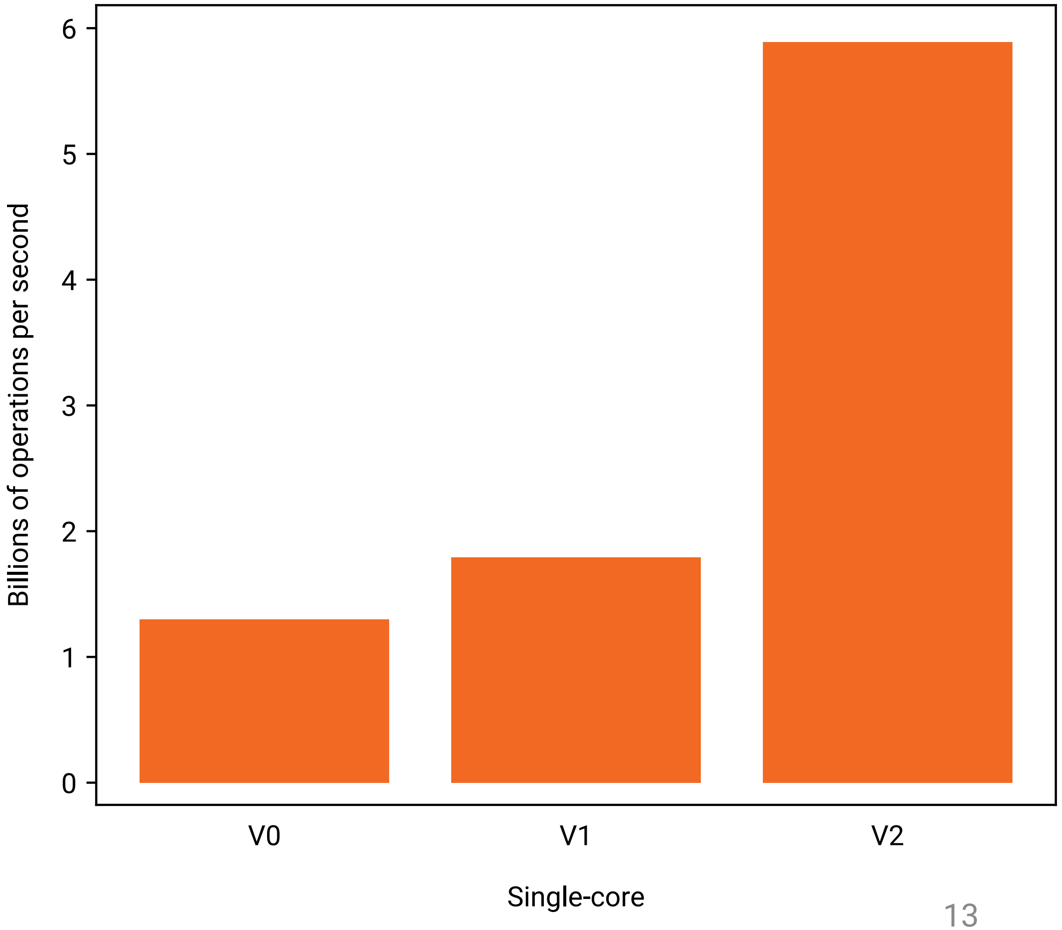
How to tell CPU that it should parallelize this?

```
float w[4] = ...
for (int k = 0; k < n/4; ++k) {
    for (int m = 0; m < 4; ++m) {
        float x = d[n*i + k*4 + m];
        float y = t[n*j + k*4 + m];
        float z = x + y;
        w[m] = min(w[m], z);
    }
}
v = min(w[0], w[1],
        w[2], w[3]);
```

It is done!
Here it is!
Nothing else needed!

```
float w[4] = ...
for (int k = 0; k < n/4; ++k) {
    for (int m = 0; m < 4; ++m) {
        float x = d[n*i + k*4 + m];
        float y = t[n*j + k*4 + m];
        float z = x + y;
        w[m] = min(w[m], z);
    }
}
v = min(w[0], w[1],
        w[2], w[3]);
```

≥ 3 times faster



Instruction-level parallelism

- CPU will look at the instruction stream further ahead
- It will try to find operations that are ***ready for execution***
  - their operands are already known
  - there are execution units available for them
- Example – "**`vminss`**" instruction:
  - two execution ports in each CPU core that can run this operation
  - each of them can start a new operation at each clock cycle
  - if there are lots of ***independent*** "`vminss`" operations in the code, then we can get a throughput of 2 operations / clock cycle / core

Instruction-level parallelism

Bad: dependent

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

Good: independent

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

Instruction-level parallelism

Bad: dependent

```
a1 = x[a0];
a2 = x[a1];
a3 = x[a2];
a4 = x[a3];
a5 = x[a4];
```

Good: independent

```
b1 = x[a1];
b2 = x[a2];
b3 = x[a3];
b4 = x[a4];
b5 = x[a5];
```

Instruction-level parallelism

Bad: dependent

```
a1 = min(b1, a0);
a2 = min(b2, a1);
a3 = min(b3, a2);
a4 = min(b4, a3);
a5 = min(b5, a4);
```

Good: independent

```
b1 = min(b1, a1);
b2 = min(b2, a2);
b3 = min(b3, a3);
b4 = min(b4, a4);
b5 = min(b5, a5);
```