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);
}
Innermost loop

• Dependency chain
  • cannot start the next “min” operation until we know the result of the previous “min” operation

• Cost of each iteration:
  \( \geq \text{latency of the “min” operation} \)
**Innermost loop**

- Dependency chain
- Cost of each iteration: \( \geq \) latency of the “min” operation
- Benchmarks: \( \approx 4 \) clock cycles per iteration
- Latency of the “vminss” instruction: 4 clock cycles

```c
v0 = 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:

\[
\begin{align*}
\min(\min(\min(z_0, z_1), z_2), z_3) &= \\
\min(\min(z_0, z_2), \min(z_1, z_3))
\end{align*}
\]

Inherently sequential, no room for parallelism

Two independent operations, could be computed in parallel
Dependency chain

Accumulate two minimums:

- $v_0 = \text{minimum of even elements}$
- $v_1 = \text{minimum of odd elements}$

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

```plaintext
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:
- \(v_0\) = minimum of elements 0 mod 3
- \(v_1\) = minimum of elements 1 mod 3
- \(v_2\) = minimum of elements 2 mod 3

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

\[
v = \min(v, z0); \\
v = \min(v, z1); \\
v = \min(v, z2); \\
\ldots \\
v_0 = \min(v_0, z0); \\
v_1 = \min(v_1, z1); \\
v_2 = \min(v_2, z2); \\
v_0 = \min(v_0, z3); \\
v_1 = \min(v_1, z4); \\
v_2 = \min(v_2, z5); \\
\ldots \\
v = \min(v_0, v_1, v_2);
\]
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 \textit{ready for execution}
  • their operands are already known
  • there are execution units available for them

• Example – “\texttt{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 \textit{independent} “\texttt{vminss}” operations in the code, then we can get a throughput of 2 operations / clock cycle / core
Instruction-level parallelism

Bad: dependent

\[
a_1 *= a_0; \\
a_2 *= a_1; \\
a_3 *= a_2; \\
a_4 *= a_3; \\
a_5 *= a_4;
\]

Good: independent

\[
b_1 *= a_1; \\
b_2 *= a_2; \\
b_3 *= a_3; \\
b_4 *= a_4; \\
b_5 *= a_5;
\]
Instruction-level parallelism

Bad: dependent

\[
\begin{align*}
  a_1 &= x[a_0] ; \\
  a_2 &= x[a_1] ; \\
  a_3 &= x[a_2] ; \\
  a_4 &= x[a_3] ; \\
  a_5 &= x[a_4] ;
\end{align*}
\]

Good: independent

\[
\begin{align*}
  b_1 &= x[a_1] ; \\
  b_2 &= x[a_2] ; \\
  b_3 &= x[a_3] ; \\
  b_4 &= x[a_4] ; \\
  b_5 &= x[a_5] ;
\end{align*}
\]
**Instruction-level parallelism**

**Bad: dependent**

\[
\begin{align*}
a_1 &= \min(b_1, a_0); \\
a_2 &= \min(b_2, a_1); \\
a_3 &= \min(b_3, a_2); \\
a_4 &= \min(b_4, a_3); \\
a_5 &= \min(b_5, a_4);
\end{align*}
\]

**Good: independent**

\[
\begin{align*}
b_1 &= \min(b_1, a_1); \\
b_2 &= \min(b_2, a_2); \\
b_3 &= \min(b_3, a_3); \\
b_4 &= \min(b_4, a_4); \\
b_5 &= \min(b_5, a_5);
\end{align*}
\]