Lecture 7. Instruction-Level Parallelism

Introductions to Data Systems and Data Design

Ce Zhang

What is ILP?

Instruction-Level Parallelism

ILP = the potential to execute multiple instructions simultaneously

Sequential: add r1,r2,r3 → sub r4,r5,r6 → mul r7,r8,r9

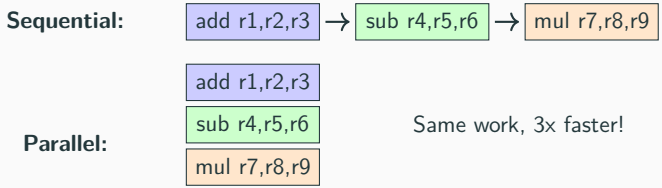
Parallel:

add r1,r2,r3
sub r4,r5,r6
mul r7,r8,r9

Same work, 3x faster!

Instruction-Level Parallelism

ILP = the potential to execute multiple instructions simultaneously



Key question: When can instructions execute in parallel?

Answer: When they are *independent* — no data dependencies between them.

Why ILP Matters for Data Systems

Modern CPUs can execute 4–8 operations per cycle:

Processor	Issue Width	ALU Ports	Load Ports
Intel Skylake	4 µOPs/cycle	4	2

Why ILP Matters for Data Systems

Modern CPUs can execute 4–8 operations per cycle:

Processor	Issue Width	ALU Ports	Load Ports
Intel Skylake	4 µOPs/cycle	4	2

But: Typical programs achieve only **1.5–2.5 IPC** (instructions per cycle).

Why? Dependencies between instructions prevent parallel execution.

The Data Systems Connection

Many data operations are inherently parallel:

- **Map:** Apply function to each element independently
- **Filter:** Test each element independently
- **Reduce:** Combine elements... *but how?*

The Data Systems Connection

Many data operations are inherently parallel:

- **Map:** Apply function to each element independently
- **Filter:** Test each element independently
- **Reduce:** Combine elements... *but how?*

```
// Map: embarrassingly parallel
for (int i = 0; i < n; i++)
    b[i] = f(a[i]);  // Each iteration independent!

// Reduce: sequential?
for (int i = 0; i < n; i++)
    sum += a[i];     // Each iteration depends on previous!
```

This lecture: How to expose ILP in reductions and similar operations.

Data Dependencies

Three Types of Dependencies

```
// Given these instructions:
r1 = r2 + r3    // I1
r4 = r1 + r5    // I2: uses r1 from I1
r1 = r6 + r7    // I3: writes r1 again
r8 = r1 + r9    // I4: uses r1 from I3
```

Type	Name	Example	Real Dependency?
RAW	Read After Write	I1 $\rightarrow$ I2	**Yes** (true)
WAR	Write After Read	I2 $\rightarrow$ I3	No (name only)
WAW	Write After Write	I1 $\rightarrow$ I3	No (name only)

Three Types of Dependencies

```
// Given these instructions:
r1 = r2 + r3    // I1
r4 = r1 + r5    // I2: uses r1 from I1
r1 = r6 + r7    // I3: writes r1 again
r8 = r1 + r9    // I4: uses r1 from I3
```

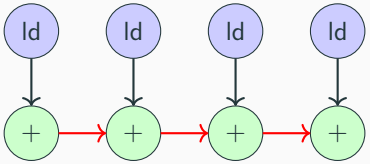
Type	Name	Example	Real Dependency?
RAW	Read After Write	I1 $\rightarrow$ I2	**Yes** (true)
WAR	Write After Read	I2 $\rightarrow$ I3	No (name only)
WAW	Write After Write	I1 $\rightarrow$ I3	No (name only)

RAW = True dependency: I2 *must wait* for I1's result. Cannot be avoided.

WAR, WAW = False dependencies: CPU eliminates these via register renaming.

The Reduction Problem

```
long sum = 0;
for (int i = 0; i < n; i++) {
    sum += a[i];
}
```



Sequential chain!

Each + waits for previous

ILP = 1 for the additions — complete serialization!

Breaking the Chain

Let's Measure It

```
// sum_v1.cpp – naive reduction (100M doubles)
double sum = 0;
for (int i = 0; i < n; i++) {
    sum += a[i];  // Dependency chain: each add waits for previous
}
```

```
$ cd examples && make sum_v1 && ./sum_v1
```

Baseline Performance

```
$ cd examples && make sum_v1 && ./sum_v1
Sum: 1e+08, Time: 237 ms
```

Why so slow?

- 100M additions at 1.5 GHz = should be ~66 ms if 1 add/cycle
- But we're getting ~237 ms — about **0.25 adds/cycle**!

Baseline Performance

```
$ cd examples && make sum_v1 && ./sum_v1
Sum: 1e+08, Time: 237 ms
```

Why so slow?

- 100M additions at 1.5 GHz = should be ~66 ms if 1 add/cycle
- But we're getting ~237 ms — about **0.25 adds/cycle**!

The bottleneck: Each addition has 4-cycle latency.

With a single accumulator, we execute 1 add every 4 cycles → 0.25 IPC.

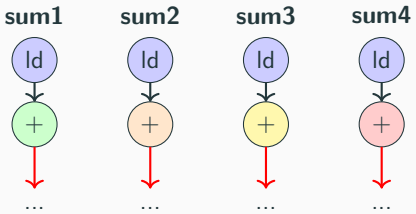
Loop Unrolling: The Idea

Instead of one accumulator, use **multiple independent accumulators**:

```
// Before: one chain
sum += a[0]; sum += a[1]; sum += a[2]; sum += a[3]; ...

// After: four parallel chains
sum1 += a[0]; sum2 += a[1]; sum3 += a[2]; sum4 += a[3];
sum1 += a[4]; sum2 += a[5]; sum3 += a[6]; sum4 += a[7];
...
sum = sum1 + sum2 + sum3 + sum4;
```

Visualizing Parallel Chains



No dependencies

between chains!

4 adds in parallel

Each chain has internal dependencies, but **chains are independent**!

4x Unrolled Code

```cpp
// sum_v2.cpp - 4x unrolled
double sum1 = 0, sum2 = 0, sum3 = 0, sum4 = 0;
for (int i = 0; i < n; i += 4) {
    sum1 += a[i];
    sum2 += a[i+1];
    sum3 += a[i+2];
    sum4 += a[i+3];
}
double sum = sum1 + sum2 + sum3 + sum4;
```

```
$ cd examples && make sum_v2 && ./sum_v2
Sum: 1e+08, Time: 60 ms      # ~ 4x speedup
```

Other Overheads?

Theoretical: 4 independent chains $\rightarrow$ 4$\times$ speedup

Other Overheads:

1. Loop overhead (increment, compare, branch)
2. Memory bandwidth starting to matter
3. Not perfectly overlapped

Other Overheads?

Theoretical: 4 independent chains $\rightarrow$ 4$\times$ speedup

Other Overheads:

1. Loop overhead (increment, compare, branch)
2. Memory bandwidth starting to matter
3. Not perfectly overlapped

Let's try more unrolling...

8x Unrolled

```
// sum_v3.cpp - 8x unrolled
double s1=0, s2=0, s3=0, s4=0, s5=0, s6=0, s7=0, s8=0;
for (int i = 0; i < n; i += 8) {
    s1 += a[i];   s2 += a[i+1]; s3 += a[i+2]; s4 += a[i+3];
    s5 += a[i+4]; s6 += a[i+5]; s7 += a[i+6]; s8 += a[i+7];
}
double sum = (s1+s2) + (s3+s4) + (s5+s6) + (s7+s8);
```

```
$ cd examples && make sum_v3 && ./sum_v3
Sum: 1e+08, Time: 34 ms      # 2x more
```

Performance Summary

```
$ cd examples && make sum    # Run all three versions
```

Version	Accumulators	Time	Speedup
Naive	1	237 ms	1×
4× unrolled	4	60 ms	3.5×
8× unrolled	8	34 ms	6.97×

Diminishing returns: Beyond 4–8 accumulators, starts to see diminishing returns.

More Examples

Example: Dot Product

```
// Naive dot product (dot_v1.cpp)
double dot = 0;
for (int i = 0; i < n; i++) {
    dot += a[i] * b[i];  // multiply then add to accumulator
}
```

Same problem: Single accumulator creates a dependency chain.

Example: Dot Product

```cpp
// Naive dot product (dot_v1.cpp)
double dot = 0;
for (int i = 0; i < n; i++) {
    dot += a[i] * b[i];  // multiply then add to accumulator
}
```

Same problem: Single accumulator creates a dependency chain.

```cpp
// 4x unrolled dot product (dot_v2.cpp)
double d1=0, d2=0, d3=0, d4=0;
for (int i = 0; i < n; i += 4) {
    d1 += a[i]   * b[i];
    d2 += a[i+1] * b[i+1];
    d3 += a[i+2] * b[i+2];
    d4 += a[i+3] * b[i+3];
```

Dot Product Performance

```
$ cd examples && make dot && ./dot_v1 && ./dot_v2
Dot: 1e+08, Time: 240.54 ms   # naive
Dot: 1e+08, Time: 61.5316 ms    # 4x unrolled
```

4× speedup — same pattern as sum!

Example: Finding Maximum

```cpp
// Naive max (max_v1.cpp)
double max_val = a[0];
for (int i = 1; i < n; i++) {
    if (a[i] > max_val) max_val = a[i];
}
```

```
$ cd examples && make max_v1 && ./max_v1
```

Two problems: (1) Dependency chain on `max_val`, (2) Unpredictable branch!

Max: Branchless + Unrolled

```
// Branchless max (max_v2.cpp) — ternary → cmov
double max_val = a[0];
for (int i = 1; i < n; i++)
    max_val = (a[i] > max_val) ? a[i] : max_val;

// 4x unrolled branchless max (max_v3.cpp)
double m1=a[0], m2=a[1], m3=a[2], m4=a[3];
for (int i = 4; i < n; i += 4) {
    m1 = (a[i]   > m1) ? a[i]   : m1;
    m2 = (a[i+1] > m2) ? a[i+1] : m2;
    m3 = (a[i+2] > m3) ? a[i+2] : m3;
    m4 = (a[i+3] > m4) ? a[i+3] : m4;
}
double max_val = max(max(m1,m2), max(m3,m4));
```

Max Performance

```
$ cd examples && make max
Max: 1, Time: 312 ms       # v1: Branch mispredictions!
Max: 1, Time: 156 ms       # v2: Branchless
Max: 1, Time: 45 ms        # v3: Branchless + unrolled
```

Combining branchless + unrolling = massive speedup on random data.

Example: Count Elements

```cpp
// count_v1.cpp: Naive with branch
for (int i = 0; i < n; i++)
    if (a[i] > threshold) count++;

// count_v2.cpp: Branchless
for (int i = 0; i < n; i++)
    count += (a[i] > threshold);  // 0 or 1

// count_v3.cpp: Branchless + 4x unrolled
int c1=0, c2=0, c3=0, c4=0;
for (int i = 0; i < n; i += 4) {
    c1 += (a[i]   > threshold);
    c2 += (a[i+1] > threshold);
    c3 += (a[i+2] > threshold);
    c4 += (a[i+3] > threshold);
}
```

Understanding the Hardware

Why Does Unrolling Help?

FP Add latency: 4 cycles (Skylake)

FP Add throughput: 2 per cycle (ports P0, P1)

```
1 accumulator (2 ports, but only 1 dependency chain):
Cycle:      0   1   2   3   4   5   6   7
Port 0:    [===|===|===|===]
Port 1:                    [===|===|===|===]
                           ↑ must wait for s1 to finish

8 accumulators (saturate both ports):
Cycle:      0   1   2   3   4   5   6   7
s1 (P0):   [===|===|===|===][===|...
s2 (P1):   [===|===|===|===][===|...
s3 (P0):       [===|===|===|===][...
s4 (P1):       [===|===|===|===][...
s5 (P0):           [===|===|===|===]
s6 (P1):           [===|===|===|===]
s7 (P0):               [===|===|===|===]
s8 (P1):               [===|===|===|===]
```

With 1 accumulator: 1 add every 4 cycles = **0.25 adds/cycle**

With 8 accumulators: 2 adds every cycle = **2 adds/cycle** (8× better!) ## Latency vs

Why Does Unrolling Help?

FP Add latency: 4 cycles (Skylake)

FP Add throughput: 2 per cycle (ports P0, P1)

```
1 accumulator (2 ports, but only 1 dependency chain):
Cycle:      0   1   2   3   4   5   6   7
Port 0:    [===|===|===|===]
Port 1:                    [===|===|===|===]
                           ↑ must wait for s1 to finish

8 accumulators (saturate both ports):
Cycle:      0   1   2   3   4   5   6   7
s1 (P0):   [===|===|===|===][===|...
s2 (P1):   [===|===|===|===][===|...
s3 (P0):       [===|===|===|===][...
s4 (P1):       [===|===|===|===][...
s5 (P0):           [===|===|===|===]
s6 (P1):           [===|===|===|===]
s7 (P0):               [===|===|===|===]
s8 (P1):               [===|===|===|===]
```

With 1 accumulator: 1 add every 4 cycles = **0.25 adds/cycle**

With 8 accumulators: 2 adds every cycle = **2 adds/cycle** (8× better!) ## Latency vs

The Magic Number

FP add: 4 cycles latency $\times$ 2/cycle throughput $=$ **8 operations in flight**

$\rightarrow$ Need at least 8 independent adds to saturate the FP adders!

The Magic Number

FP add: 4 cycles latency $\times$ 2/cycle throughput $=$ **8 operations in flight**

$\rightarrow$ Need at least 8 independent adds to saturate the FP adders!

FP multiply: 4 cycles $\times$ 2/cycle $=$ **8 operations in flight**

Load: 4 cycles $\times$ 2/cycle $=$ **8 operations in flight**

The Magic Number

FP add: 4 cycles latency $\times$ 2/cycle throughput $=$ **8 operations in flight**

$\rightarrow$ Need at least 8 independent adds to saturate the FP adders!

FP multiply: 4 cycles $\times$ 2/cycle $=$ **8 operations in flight**

Load: 4 cycles $\times$ 2/cycle $=$ **8 operations in flight**

Rule of thumb: Unroll by **latency $\times$ throughput** to maximize ILP.

For most FP operations: **4–8$\times$ unrolling** is the sweet spot.

Compiler Optimizations

Does the Compiler Do This Automatically?

```
// Will -O3 unroll this for us?
double sum = 0;
for (int i = 0; i < n; i++) {
    sum += a[i];
}
```

```
$ cd examples && make fast    # Compare -O1 vs -O3 -ffast-math
Time: 238 ms     # -O1
Time: 22 ms     # -O3: Same!
```

No! The compiler cannot change the order of FP additions without `-ffast-math` (not associative).

Enabling Compiler Optimizations

```
$ cd examples && make sum_v1_fast && ./sum_v1_fast
Time: 25 ms      # -O3 -ffast-math works!
```

`-ffast-math` tells the compiler:

- FP operations are associative (can reorder)
- No NaN/Inf checks needed
- Allows aggressive optimizations

Enabling Compiler Optimizations

```
$ cd examples && make sum_v1_fast && ./sum_v1_fast
Time: 25 ms      # -O3 -ffast-math works!
```

`-ffast-math` tells the compiler:

- FP operations are associative (can reorder)
- No NaN/Inf checks needed
- Allows aggressive optimizations

Warning: May change numerical results slightly!

What Does -O3 -ffast-math Generate?

```
$ cd examples
$ g++ -O3 -ffast-math -std=c++14 -S sum_v1.cpp
$ cat sum_v1.s | grep -A 20 "\.L3:"
```

The compiler generates **SIMD code** with multiple accumulators!

- Uses `vaddpd` (vectorized add, 4 doubles at once)
- Unrolls the loop
- Multiple vector accumulators

What Does -O3 -ffast-math Generate?

```
$ cd examples
$ g++ -O3 -ffast-math -std=c++14 -S sum_v1.cpp
$ cat sum_v1.s | grep -A 20 "\.L3:"
```

The compiler generates **SIMD code** with multiple accumulators!

- Uses vaddpd (vectorized add, 4 doubles at once)
- Unrolls the loop
- Multiple vector accumulators

But: Understanding manual unrolling helps you:

1. Know when compiler can't help (complex reductions)
2. Debug performance issues
3. Write code the compiler can optimize

Practical Guidelines

When to Unroll Manually

Do unroll when:

- Reduction over large arrays (sum, max, dot product)
- Compiler can't optimize (no `-ffast-math`, complex operations)
- Performance-critical inner loops

Don't unroll when:

- Code is not in a hot path
- Loop body is already complex
- Memory bandwidth is the bottleneck (unrolling won't help)

How Much to Unroll

Data Type	Latency	Throughput	Unroll Factor
int add	1	4/cycle	4×
int64 add	1	4/cycle	4×
float add	4	2/cycle	8×
double add	4	2/cycle	8×
double mul	4	2/cycle	8×

How Much to Unroll

Data Type	Latency	Throughput	Unroll Factor
int add	1	4/cycle	4×
int64 add	1	4/cycle	4×
float add	4	2/cycle	8×
double add	4	2/cycle	8×
double mul	4	2/cycle	8×

Practical advice: Start with 4×, measure, try 8×.

The Complete Pattern

```
// Template for any associative reduction
T reduce(T* a, int n, T init, T (*op)(T, T)) {
    // Multiple accumulators
    T acc1 = init, acc2 = init, acc3 = init, acc4 = init;

    int i = 0;
    for (; i + 4 <= n; i += 4) {
        acc1 = op(acc1, a[i]);
        acc2 = op(acc2, a[i+1]);
        acc3 = op(acc3, a[i+2]);
        acc4 = op(acc4, a[i+3]);
    }

    // Handle remainder
    for (; i < n; i++) {
        acc1 = op(acc1, a[i]);
```

Diagnosing ILP with PMU Counters

IPC

IPC (Instructions Per Cycle) = instructions retired / cycles

```
$ cd examples && make sum_v1 sum_v2
$ perf stat ./sum_v1
     1,998,092,185      cycles                    #    4.966 GHz                      (
     3,014,800,364      instructions              #    1.51  insn per cycle

                                                  #    0.03  stalled cycles per insn  (

$ perf stat ./sum_v2
     1,086,025,270      cycles                    #    4.848 GHz                      (
     2,903,562,007      instructions              #    2.67  insn per cycle

                                                  #    0.06  stalled cycles per insn  (
```

IPC

IPC (Instructions Per Cycle) = instructions retired / cycles

```
$ cd examples && make sum_v1 sum_v2
$ perf stat ./sum_v1
     1,998,092,185      cycles                    #    4.966 GHz                     (
     3,014,800,364      instructions              #    1.51  insn per cycle

                                                  #    0.03  stalled cycles per insn  (

$ perf stat ./sum_v2
     1,086,025,270      cycles                    #    4.848 GHz                     (
     2,903,562,007      instructions              #    2.67  insn per cycle

                                                  #    0.06  stalled cycles per insn  (
```

Low IPC often indicates ILP problems!

Summary

Key Takeaways

1. **ILP** = executing multiple instructions per cycle
   - Limited by data dependencies between instructions
2. **Reductions create dependency chains** — single accumulator = serialization
3. **Loop unrolling with multiple accumulators** breaks the chain
   - 4–8 accumulators typically optimal
   - Matches latency $\times$ throughput
4. **Compiler may help** with `-O3 -ffast-math`
   - But understanding the technique helps write better code

Performance Checklist

When optimizing a reduction:

1. **Measure baseline** — is it actually slow?
2. **Check for branches** — make branchless if unpredictable
3. **Unroll with multiple accumulators** — $4\times$ or $8\times$
4. **Measure again** — did it help?
5. **Check memory bandwidth** — if no improvement, you're memory-bound

Performance Checklist

When optimizing a reduction:

1. **Measure baseline** — is it actually slow?
2. **Check for branches** — make branchless if unpredictable
3. **Unroll with multiple accumulators** — 4× or 8×
4. **Measure again** — did it help?
5. **Check memory bandwidth** — if no improvement, you're memory-bound

Same patterns apply to:

- Database aggregations (SUM, COUNT, AVG, MAX)
- ML operations (dot products, norms, reductions)
- Compression (checksums, hashing)

Try It Yourself!

All examples available in the `examples/` folder:

```
$ cd examples
$ make run          # Run all examples
$ make sum          # Compare sum_v1, sum_v2, sum_v3
$ make dot          # Compare dot_v1, dot_v2
$ make max          # Compare max_v1, max_v2, max_v3
$ make count        # Compare count_v1, count_v2, count_v3
$ make fast         # Compare -O1 vs -O3 -ffast-math
$ make perf         # Run with perf stat (Linux)
```

Next lecture: SIMD — doing 4–8 operations with *one* instruction!