Lecture 7. Instruction-Level Parallelism

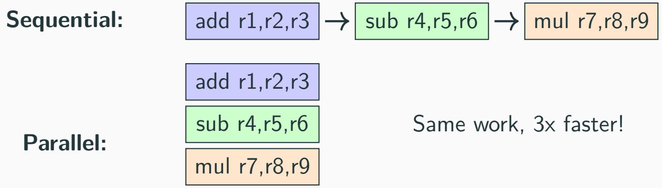
Introductions to Data Systems and Data Design

Ce Zhang

What is ILP?

Instruction-Level Parallelism

ILP = the potential to execute multiple instructions simultaneously

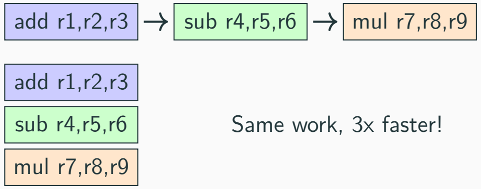


Instruction-Level Parallelism

ILP = the potential to execute multiple instructions simultaneously

Sequential:

Parallel:



Same work, 3x faster!

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
Apple M1	8 μOPs/cycle	6	3
AMD Zen 4	6 μOPs/cycle	4	3

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
Apple M1	8 µOPs/cycle	6	3
AMD Zen 4	6 µOPs/cycle	4	3

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
```

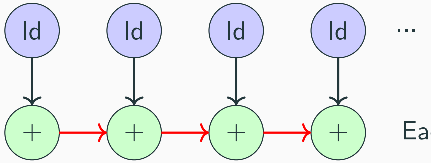
Type	Name	Example	Real Dependency?
RAW	Read After Write	I1 $\rightarrow$ I2	**Yes** (true)
WAR	Write After Read	I2 $\rightarrow$ I3	No (name only)
WAW	Write After Write	I1 $\rightarrow$ I3	No (name only)

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

```cpp
// 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: 152 ms
```

Why so slow?

- 100M additions at 3 GHz = should be ~33 ms if 1 add/cycle
- But we're getting ~150 ms — about **0.2 adds/cycle**!

Baseline Performance

```
$ cd examples && make sum_v1 && ./sum_v1
Sum: 1e+08, Time: 152 ms
```

Why so slow?

- 100M additions at 3 GHz = should be ~33 ms if 1 add/cycle
- But we're getting ~150 ms — about **0.2 adds/cycle**!

The bottleneck: Each addition has 4-cycle latency.

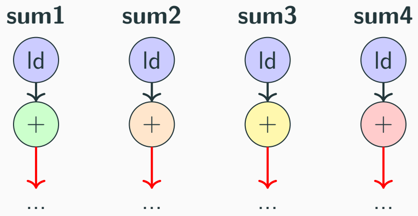
With a single accumulator, we execute 1 add every 4 cycles → 0.25 IPC.

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: 42 ms      # 3.6× speedup!
```

Why Not 4× Speedup?

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

Actual: 3.6× speedup

Why Not 4× Speedup?

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

Actual: 3.6× speedup

Reasons:

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

Why Not $4\times$ Speedup?

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

Actual: $3.6\times$ speedup

Reasons:

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

Let's try more unrolling...

8x Unrolled

```cpp
// 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);
```

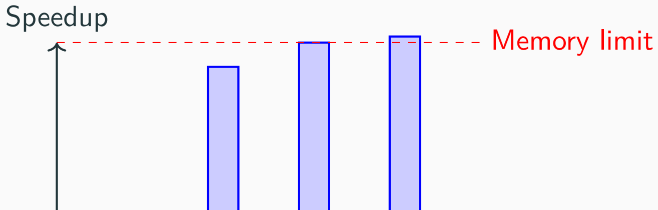
```
$ cd examples && make sum_v3 && ./sum_v3
Sum: 1e+08, Time: 38 ms      # Only 4× - hitting memory bandwidth!
```

Performance Summary

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

Version	Accumulators	Time	Speedup
Naive	1	152 ms	1×
4× unrolled	4	42 ms	3.6×
8× unrolled	8	38 ms	4.0×

Diminishing returns: Beyond 4–8 accumulators, memory bandwidth dominates.



More Examples

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.

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.

```
// 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: 168 ms   # naive
Dot: 1e+08, Time: 56 ms    # 4x unrolled
```

3× speedup — same pattern as sum!

Example: Finding Maximum

```
// 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, 2x faster
Max: 1, Time: 45 ms       # v3: Branchless + unrolled, 7x faster!
```

Combining branchless + unrolling = massive speedup on random data.

Example: Count Elements

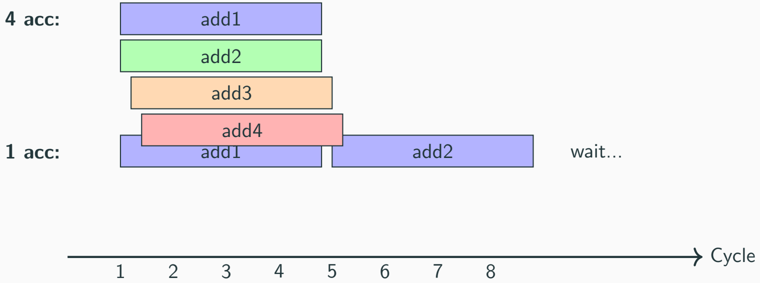
```
// Count elements > threshold (count.cpp has all versions)
int count = 0;
for (int i = 0; i < n; i++) {
    if (a[i] > threshold) count++;  // Naive: branch
}

count += (a[i] > threshold);        // Branchless: 0 or 1

// Branchless + 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);
```

Understanding the Hardware

Why Does Unrolling Help?

Add latency: 4 cycles (Skylake)

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



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

With 4 accumulators: 4 adds in 4 cycles = **1 add/cycle** (4× better!)

Latency vs Throughput

Operation	Latency	Throughput
Integer add	1 cycle	4/cycle
FP add	4 cycles	2/cycle
FP multiply	4 cycles	2/cycle
FP divide	10-20 cycles	1/14 cycles
Load (L1 hit)	4 cycles	2/cycle

Latency vs Throughput

Operation	Latency	Throughput
Integer add	1 cycle	4/cycle
FP add	4 cycles	2/cycle
FP multiply	4 cycles	2/cycle
FP divide	10-20 cycles	1/14 cycles
Load (L1 hit)	4 cycles	2/cycle

Key insight:

- **Latency** = time for one operation to complete
- **Throughput** = how many operations can be *in flight*

To achieve throughput, you need **latency** × **throughput** independent operations!

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 × throughput** to maximize ILP.

For most FP operations: **4–8× 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 vs -O3 -ffast-math
Time: 152 ms     # -O1
Time: 152 ms     # -O3: Same!
```

No! The compiler cannot change the order of FP additions (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×.

Beyond 8× rarely helps (memory becomes bottleneck).

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
```

Diagnosing ILP with PMU Counters

The First Metric: IPC

IPC (Instructions Per Cycle) = instructions retired / cycles

```
$ cd examples && make sum_v1 sum_v2
$ perf stat ./sum_v1
        481,234,567      cycles
        125,012,345      instructions    #  0.26 IPC

$ perf stat ./sum_v2
        132,456,789      cycles
        175,023,456      instructions    #  1.32 IPC
```

The First Metric: IPC

IPC (Instructions Per Cycle) = instructions retired / cycles

```
$ cd examples && make sum_v1 sum_v2
$ perf stat ./sum_v1
        481,234,567      cycles
        125,012,345      instructions    #  0.26 IPC

$ perf stat ./sum_v2
        132,456,789      cycles
        175,023,456      instructions    #  1.32 IPC
```

Low IPC ($<$ 1) often indicates ILP problems!

- Modern CPUs can retire 4+ instructions/cycle
- IPC of 0.26 means we're leaving ~90% performance on the table

Intel’s Top-Down Analysis

Intel CPUs provide **topdown** counters that categorize where cycles go:

```
$ cd examples && perf stat -M TopdownL1 ./sum_v1
    48.2%  backend_bound        # ← ILP problem!
    12.1%  frontend_bound
     8.3%  bad_speculation
    31.4%  retiring
```

Intel’s Top-Down Analysis

Intel CPUs provide **topdown** counters that categorize where cycles go:

```
$ cd examples && perf stat -M TopdownL1 ./sum_v1
    48.2%  backend_bound          # ← ILP problem!
    12.1%  frontend_bound
     8.3%  bad_speculation
    31.4%  retiring
```

Category	Meaning
Backend Bound	Waiting for data or execution resources
Frontend Bound	Instruction fetch/decode stalls
Bad Speculation	Wasted work from mispredicted branches
Retiring	Useful work (higher is better!)

Backend Bound: Digging Deeper

```
$ cd examples && perf stat -M TopdownL2 ./sum_v1
    Backend Bound:
        42.1%  core_bound              # ← Execution unit stalls
         6.1%  memory_bound            # ← Memory access stalls
```

Backend Bound: Digging Deeper

```
$ cd examples && perf stat -M TopdownL2 ./sum_v1
    Backend Bound:
        42.1%  core_bound           # ← Execution unit stalls
         6.1%  memory_bound         # ← Memory access stalls
```

Core Bound = waiting for execution resources

- Could be port contention (not enough ALUs)
- Could be **dependency chains** (our problem!)

Memory Bound = waiting for data from cache/memory

Specific Counters for ILP

```
$ cd examples
$ perf stat -e cycles,instructions,uops_executed.thread,\
uops_issued.any,resource_stalls.any ./sum_v1
```

Counter	Meaning
`uops_executed.thread`	µOPs actually executed
`uops_issued.any`	µOPs sent to execution
`resource_stalls.any`	Cycles stalled on resources
`cycle_activity.stalls_total`	Total stall cycles

Specific Counters for ILP

```
$ cd examples
$ perf stat -e cycles,instructions,uops_executed.thread,\
uops_issued.any,resource_stalls.any ./sum_v1
```

Counter	Meaning
`uops_executed.thread`	µOPs actually executed
`uops_issued.any`	µOPs sent to execution
`resource_stalls.any`	Cycles stalled on resources
`cycle_activity.stalls_total`	Total stall cycles

Key ratio: `uops_executed / cycles`

- Should be 2–4 on modern CPUs
- If < 1, you have an ILP problem

Example: Diagnosing Our Sum

```
$ cd examples && make perf   # Runs all perf comparisons

$ perf stat -e cycles,uops_executed.thread ./sum_v1
        480,000,000      cycles
        125,000,000      uops_executed.thread  # 0.26 uops/cycle!

$ perf stat -e cycles,uops_executed.thread ./sum_v2
        130,000,000      cycles
        180,000,000      uops_executed.thread  # 1.38 uops/cycle
```

Example: Diagnosing Our Sum

```
$ cd examples && make perf   # Runs all perf comparisons

$ perf stat -e cycles,uops_executed.thread ./sum_v1
        480,000,000      cycles
        125,000,000      uops_executed.thread  # 0.26 uops/cycle!

$ perf stat -e cycles,uops_executed.thread ./sum_v2
        130,000,000      cycles
        180,000,000      uops_executed.thread  # 1.38 uops/cycle
```

Diagnosis: Naive version executes only 0.26 μops/cycle

Cause: Dependency chain — each add waits 4 cycles for previous

Solution: Multiple accumulators → 1.38 μops/cycle (5× better!)

Port Utilization

For advanced diagnosis, check which execution ports are busy:

```
$ cd examples
$ perf stat -e uops_dispatched_port.port_0,\
uops_dispatched_port.port_1,uops_dispatched_port.port_5 ./sum_v2
```

Port Utilization

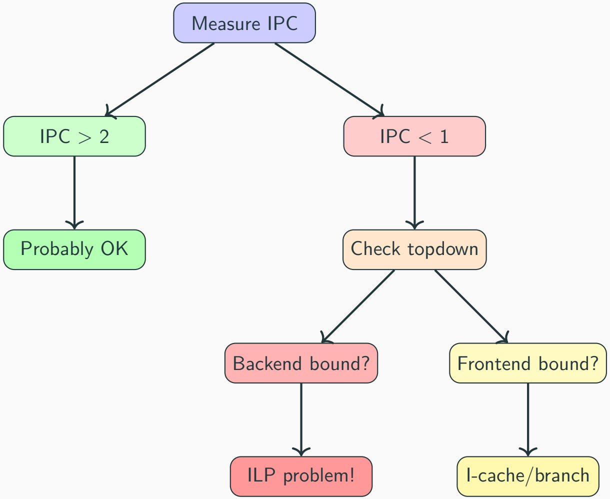
For advanced diagnosis, check which execution ports are busy:

```
$ cd examples
$ perf stat -e uops_dispatched_port.port_0,\
uops_dispatched_port.port_1,uops_dispatched_port.port_5 ./sum_v2
```

If one port is saturated while others are idle → restructure code

If all ports have low utilization → dependency problem (our case!)

Quick Diagnosis Flowchart



Practical Commands

```
$ cd examples

# Quick IPC check
$ perf stat ./sum_v1

# Top-down analysis (Intel)
$ perf stat -M TopdownL1 ./sum_v1

# Detailed execution metrics
$ perf stat -e cycles,instructions,uops_executed.thread ./sum_v1

# Compare two versions
$ perf stat ./sum_v1 && perf stat ./sum_v2
```

Practical Commands

```
$ cd examples

# Quick IPC check
$ perf stat ./sum_v1

# Top-down analysis (Intel)
$ perf stat -M TopdownL1 ./sum_v1

# Detailed execution metrics
$ perf stat -e cycles,instructions,uops_executed.thread ./sum_v1

# Compare two versions
$ perf stat ./sum_v1 && perf stat ./sum_v2
```

Rule of thumb: If IPC < 1 and backend_bound $> 40\%$, try loop unrolling!

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** — $4\times$ or $8\times$
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 versions
$ make dot          # Compare dot product versions
$ make max          # Compare max versions
$ 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!