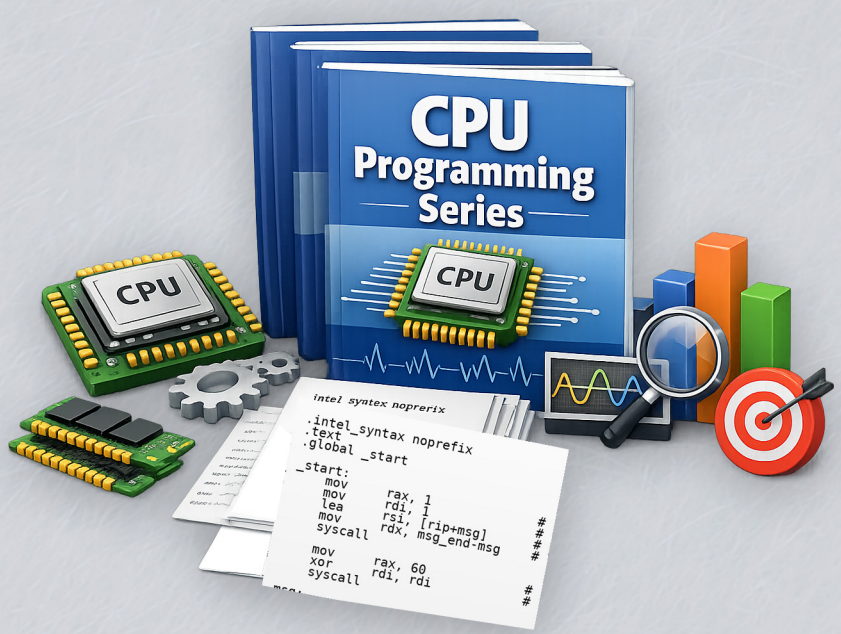
https://simplifycpp.org

CPU Programming Series

SIMD on x86: SSE

AVX Practical Patterns



20

Prepared by Ayman Alheraki

CPU Programming Series

SIMD on x86: SSE

AVX Practical Patterns

Prepared by Ayman Alheraki

simplifycpp.org

February 2026

Contents

Contents **2**

Preface **15**

Scope and Philosophy of This Booklet 15

Why SIMD Still Matters on Modern x86 16

What This Booklet Is Not 18

Required Background and Assumptions 19

How to Read and Apply the Patterns 20

1) Start with a scalar reference and a measurable baseline 21

2) Make data layout and loop shape SIMD-friendly 21

3) Apply the SIMD pattern with a clear tail strategy 21

4) Validate correctness with edge cases 22

5) Measure the right thing, the right way 23

6) Keep maintainability: isolate SIMD and allow multiple backends 23

1 SIMD Fundamentals on x86 **25**

1.1 What SIMD Really Means at the Microarchitectural Level 25

1.1.1 The canonical SIMD loop shape 26

1.1.2 What the core “sees” 26

1.2 Vector Registers vs Scalar Execution 27

1.2.1 Lane semantics . . . 27
1.2.2 Architectural width vs physical execution . . . 28
1.3 Data Parallelism vs Instruction-Level Parallelism . . . 28
1.3.1 Data Parallelism (DLP) . . . 28
1.3.2 Instruction-Level Parallelism (ILP) . . . 29
1.3.3 They complement each other . . . 29
1.4 SIMD Width Evolution: 128 → 256 → 512 . . . 31
1.4.1 Practical consequences of widening . . . 31
1.4.2 A robust width strategy . . . 31
1.4.3 Intel-syntax assembly intuition (GAS with Intel syntax) . . . 32
1.5 Costs and Benefits of Vectorization . . . 32
1.5.1 Benefits . . . 32
1.5.2 Costs . . . 33
1.5.3 A realistic "win/lose" checklist . . . 33
1.5.4 Extensive example: turning branches into masks . . . 34
1.5.5 Extensive example: layout impact (AoS vs SoA) . . . 35

2 x86 SIMD Execution Model 37
2.1 SIMD Pipelines and Execution Units . . . 37
2.1.1 What you should care about (portable mental model) . . . 37
2.1.2 A practical bottleneck example: load–compute–store . . . 38
2.2 Register Files and Rename Behavior . . . 39
2.2.1 Architectural vs physical registers . . . 39
2.2.2 Why renaming matters for SIMD . . . 39
2.2.3 Example: register pressure from "too many temporaries" . . . 39
2.3 Throughput vs Latency for Vector Instructions . . . 40
2.3.1 Definitions that matter in practice . . . 41
2.3.2 Example: latency-bound reduction . . . 41

2.4 Vector Dependency Chains . . . 43
2.4.1 Recognizing dependency-limited SIMD . . . 43
2.4.2 Breaking vector chains: the multi-accumulator rule . . . 44
2.5 SIMD and Out-of-Order Execution . . . 45
2.5.1 OoO-friendly SIMD pattern: software pipelining . . . 45
2.5.2 When OoO cannot save you . . . 46
2.5.3 Assembly intuition (GAS, Intel syntax) . . . 47

3 Data Layout for SIMD 48
3.1 Array of Structures vs Structure of Arrays . . . 48
3.1.1 AoS vs SoA: what changes for SIMD . . . 48
3.1.2 Example: scaling only `x` . . . 49
3.1.3 When AoS is acceptable . . . 49
3.1.4 Hybrid layouts: AoSoA (practical compromise) . . . 50
3.2 Alignment Requirements and Penalties . . . 50
3.2.1 Alignment is a performance contract, not just correctness . . . 50
3.2.2 Practical alignment targets . . . 51
3.2.3 Correct allocation for aligned SIMD buffers . . . 51
3.2.4 Aligned vs unaligned intrinsics . . . 52
3.3 Padding, Stride, and Cache-Line Interaction . . . 52
3.3.1 Stride determines whether SIMD feeds efficiently . . . 52
3.3.2 Example: strided access is hostile to SIMD . . . 53
3.3.3 Padding to avoid cache-line splits and simplify tails . . . 53
3.3.4 Cache-line interaction rule of thumb . . . 54
3.4 False Sharing and Vector Loads . . . 54
3.4.1 What false sharing is . . . 54
3.4.2 Bad pattern: per-thread counters in the same cache line . . . 55
3.4.3 Good pattern: pad each thread's writable data to a cache line . . . 55

3.4.4 SIMD partitioning rule 55
3.5 Designing SIMD-Friendly Memory Layouts 56
3.5.1 The design checklist 56
3.5.2 Alias control: separate input and output buffers 56
3.5.3 SoA + alignment + tail: a production-quality pattern 57
3.5.4 AoSoA block layout: maximizing locality and SIMD lanes 57
3.5.5 Assembly intuition: cache-line crossing matters 58

4 SSE Programming Model **60**
4.1 XMM Registers and Instruction Classes 60
4.1.1 XMM registers: the 128-bit SIMD baseline 60
4.1.2 Instruction classes (practical grouping) 60
4.2 Packed Integer vs Packed Floating-Point 61
4.2.1 Two different universes 61
4.2.2 Example: packed float add (4 floats) 61
4.2.3 Example: packed int32 add (4 int32) 62
4.2.4 FP comparisons produce masks, not booleans 62
4.3 Load / Store Semantics 63
4.3.1 Aligned vs unaligned 63
4.3.2 Typical intrinsics 63
4.3.3 Streaming stores (when appropriate) 63
4.4 Shuffle, Permute, and Blend Operations 64
4.4.1 Why shuffles matter 64
4.4.2 Core SSE lane rearrangement tools 64
4.4.3 Example: unpack interleave (transpose building block) 65
4.4.4 Example: shuffle to broadcast a lane 65
4.4.5 Example: blend using mask logic (portable across SSE) 65
4.4.6 Example: byte shuffle (SSSE3) for packed data 66

4.5 Common SSE Idioms 66
4.5.1 Idiom 1: “Load–Compute–Store” with scalar tail 66
4.5.2 Idiom 2: reduction with horizontal add (SSE3) or manual 67
4.5.3 Idiom 3: compare to mask, then use bitwise select 67
4.5.4 Idiom 4: integer lane widening and packing 68
4.5.5 Idiom 5: avoid slow paths by keeping data “lane-aligned” 68
4.5.6 Assembly intuition (GAS, Intel syntax) 68

5 AVX and AVX2 Extensions 71
5.1 YMM Registers and 256-bit Execution 71
5.1.1 From XMM (128) to YMM (256) 71
5.1.2 Practical consequences 72
5.2 VEX Encoding and Its Implications 72
5.2.1 What VEX changes (the programmer-visible effects) 72
5.2.2 Why three-operand matters 73
5.2.3 Practical coding guideline 73
5.3 Fused Multiply-Add (FMA) 73
5.3.1 What FMA does 73
5.3.2 AXPY and dot-product building blocks 74
5.3.3 FMA and “semantic compatibility” 74
5.3.4 Dot product with FMA and multiple accumulators 74
5.4 Integer SIMD with AVX2 75
5.4.1 What AVX2 adds 75
5.4.2 Example: 32-bit integer add 76
5.4.3 Example: byte-wise absolute difference (useful in image/audio) 76
5.4.4 Example: compare + mask extraction for filtering 77
5.5 Transition Costs Between SSE and AVX 78
5.5.1 The core issue: mixing legacy SSE with AVX in hot code 78

5.5.2 The standard fix: `vzeroupper` . . . 79
5.5.3 Assembly form (GAS, Intel syntax) . . . 79
5.5.4 A disciplined region policy . . . 80

6 Masking and Control Flow in SIMD **81**
6.1 Branchless Programming Fundamentals . . . 81
6.1.1 Why branches are a problem for SIMD . . . 81
6.1.2 Core identity: select by mask . . . 82
6.2 Comparison Instructions and Masks . . . 82
6.2.1 Comparisons produce masks, not booleans . . . 82
6.2.2 Example: clamp using compares indirectly (min/max) . . . 82
6.2.3 Example: explicit compare mask for “keep or zero” . . . 83
6.2.4 Mask extraction: turning lane masks into bits . . . 84
6.3 Conditional Execution via Blends . . . 84
6.3.1 Blend is selection, but mind the cost model . . . 84
6.3.2 Example: piecewise function without branches . . . 85
6.3.3 Example: saturating update (if condition then update else keep) . . . 86
6.4 Vectorized Loops with Partial Tails . . . 86
6.4.1 The tail problem . . . 86
6.4.2 Strategy 1: scalar tail (recommended default) . . . 87
6.4.3 Strategy 2: padding (fast when you control allocation) . . . 87
6.4.4 Strategy 3: masked tail for AVX (emulation) . . . 87
6.5 Eliminating Scalar Branches . . . 88
6.5.1 Replace per-element branches with vector masks . . . 89
6.5.2 Example: threshold + update (branchy to branchless) . . . 89
6.5.3 Example: filtering without branches (count + mark) . . . 90
6.5.4 Assembly intuition (GAS, Intel syntax) . . . 91

7 Practical SIMD Loop Patterns **92**

7.1 Vectorized Reduction (Sum, Min, Max) . . . 92

7.1.1 Reduction fundamentals . . . 92

7.1.2 Sum reduction with AVX and two accumulators . . . 92

7.1.3 Min/Max reduction with AVX . . . 93

7.1.4 Reduction correctness note . . . 95

7.2 Vectorized Search and Filtering . . . 95

7.2.1 Search: find the first element matching a predicate . . . 95

7.2.2 Filtering: count matches fast (predicate density unknown) . . . 97

7.2.3 Filtering strategy note . . . 97

7.3 Vectorized Copy and Transform Loops . . . 98

7.3.1 Copy and scale: the bread-and-butter kernels . . . 98

7.3.2 Vectorized copy (float) with AVX . . . 98

7.3.3 Transform: y = a*x + b (FMA when available) . . . 98

7.3.4 Byte transforms with AVX2 (integer SIMD) . . . 99

7.4 Horizontal vs Vertical Computation . . . 100

7.4.1 Definitions . . . 100

7.4.2 Example: vertical vs horizontal . . . 100

7.4.3 Design rule . . . 101

7.5 Loop Unrolling with SIMD . . . 101

7.5.1 Why unroll SIMD loops . . . 101

7.5.2 Unroll patterns . . . 101

7.5.3 Example: unrolled map kernel (two independent vectors per iteration) . . . 101

7.5.4 Example: unrolled reduction (four accumulators) . . . 102

7.5.5 Unrolling limits . . . 103

7.5.6 Assembly intuition (GAS, Intel syntax) . . . 104

8 SIMD and Memory Bandwidth **105**

8.1 Load/Store Bottlenecks . . . 105

8.1.1 The bandwidth reality . . . 105

8.1.2 Roofline-style reasoning (practical) . . . 105

8.1.3 Example: SAXPY is often bandwidth-bound for large arrays . . . 106

8.1.4 Practical bottleneck checklist . . . 106

8.2 Streaming Loads and Stores . . . 107

8.2.1 Temporal vs non-temporal (streaming) stores . . . 107

8.2.2 AVX non-temporal stores . . . 107

8.2.3 Streaming loads . . . 108

8.2.4 When streaming stores hurt . . . 108

8.3 Prefetching Strategies . . . 108

8.3.1 Hardware prefetch usually wins for unit stride . . . 108

8.3.2 Software prefetch basics . . . 109

8.3.3 Example: prefetch ahead in a streaming transform . . . 109

8.3.4 Prefetch for two-stream kernels . . . 110

8.4 Write Combining and Store Forwarding . . . 111

8.4.1 Write combining (practical meaning) . . . 111

8.4.2 Store forwarding (practical meaning) . . . 111

8.4.3 Avoid store-forwarding stalls: do full-width stores . . . 112

8.4.4 Example: structured full-width store . . . 112

8.5 Measuring Memory-Bound SIMD Code . . . 112

8.5.1 Measure bandwidth, not just time . . . 112

8.5.2 Avoid common benchmark traps . . . 113

8.5.3 A minimal, robust bandwidth benchmark harness . . . 113

8.5.4 Example: measuring a copy kernel . . . 114

8.5.5 Assembly intuition: a bandwidth kernel is mostly moves . . . 115

9 Compiler Vectorization vs Manual SIMD **116**

9.1 Auto-Vectorization Capabilities 116

9.1.1 What modern compilers can do well 116

9.1.2 Baseline example: a vectorizable transform loop 117

9.1.3 Make the compiler's job easier: noalias + alignment contracts 117

9.2 When Compilers Fail to Vectorize 118

9.2.1 1) Pointer aliasing uncertainty 118

9.2.2 2) Non-unit stride / irregular access 118

9.2.3 3) Loop-carried dependencies 118

9.2.4 4) Hard-to-predicate branches 119

9.2.5 5) Function calls inside the loop 119

9.3 Pragmas, Hints, and Assumptions 119

9.3.1 General principle 119

9.3.2 Common categories of hints 120

9.3.3 Portable alignment declaration with C++ 120

9.3.4 OpenMP SIMD pragma (widely supported idea) 120

9.3.5 "Assume" rules of thumb 121

9.4 Manual Intrinsics vs Compiler Output 121

9.4.1 When intrinsics are justified 121

9.4.2 The cost of intrinsics 121

9.4.3 Side-by-side example: branchless clamp (compiler vs intrinsics) 122

9.4.4 Correctness and FP semantics 123

9.5 Reading Compiler-Generated Assembly 123

9.5.1 What to look for (a deterministic checklist) 123

9.5.2 Example: expected AVX pattern in assembly (conceptual) 124

9.5.3 Example: recognizing reduction structure 124

9.5.4 Practical workflow 125

10 SIMD Precision, Accuracy, and Pitfalls **126**

10.1 Floating-Point Reordering Effects 126

10.1.1 Why SIMD changes results 126

10.1.2 Example: scalar sum vs vector sum 126

10.1.3 Engineering rule 128

10.2 Denormals and Flush-to-Zero 128

10.2.1 What denormals do to performance 128

10.2.2 FTZ and DAZ 128

10.2.3 Controlling FTZ/DAZ via MXCSR 128

10.2.4 Safe usage pattern 129

10.2.5 Rule 129

10.3 Precision Loss in Vectorized Math 129

10.3.1 Where precision issues come from 129

10.3.2 Example: accumulating many floats into float vs double 130

10.3.3 Vector pattern: accumulate float lanes, reduce into double 130

10.3.4 Rule of thumb 131

10.4 Integer Overflow and Saturation 131

10.4.1 Overflow is a correctness problem, not a performance problem 131

10.4.2 Use saturating arithmetic when the domain requires it 132

10.4.3 Widen–compute–narrow (avoid overflow) 132

10.5 Debugging SIMD Bugs 133

10.5.1 The common bug categories 133

10.5.2 Debug strategy: scalar reference + randomized tests 133

10.5.3 Debug strategy: isolate tails and boundaries 135

10.5.4 Debug strategy: inspect intermediate vectors 135

10.5.5 Assembly intuition: silent bugs are often tails or masks 135

11 Performance Measurement and Analysis **137**
11.1 Benchmarking SIMD Code Correctly . . . 137
11.1.1 What "correct" means for SIMD benchmarks . . . 137
11.1.2 A minimal, robust timing harness . . . 137
11.1.3 Preventing dead-code elimination . . . 138
11.1.4 Example: benchmark a dot product kernel . . . 139
11.2 Cycle Counting and Hardware Counters . . . 140
11.2.1 Cycle counting: what you can and cannot assume . . . 140
11.2.2 RDTSC for quick, local measurement (x86) . . . 140
11.2.3 Hardware performance counters (conceptual workflow) . . . 141
11.3 Identifying Compute-Bound vs Memory-Bound Code . . . 142
11.3.1 Two quick tests . . . 142
11.3.2 Example: adding math to detect memory-bound behavior . . . 142
11.3.3 Bytes and arithmetic intensity . . . 143
11.4 Scaling Behavior with Vector Width . . . 144
11.4.1 What scaling you should expect . . . 144
11.4.2 A controlled scaling experiment . . . 144
11.4.3 Example: same kernel in SSE vs AVX . . . 145
11.4.4 Interpretation . . . 145
11.5 Avoiding Misleading Benchmarks . . . 146
11.5.1 The classic benchmark traps . . . 146
11.5.2 A disciplined checklist for SIMD benchmarks . . . 146
11.5.3 Assembly intuition: measure the loop you think you measure . . . 147

12 Real-World SIMD Design Guidelines **149**
12.1 Choosing the Right SIMD Width . . . 149
12.1.1 Start from the bottleneck . . . 149
12.1.2 Practical baseline policy on x86 . . . 149

12.1.3 Runtime dispatch skeleton (multi-versioning) . . . 150
12.1.4 Design rule . . . 150
12.2 When SIMD Hurts Performance . . . 151
12.2.1 Common reasons SIMD gets slower . . . 151
12.2.2 Example: shuffle-heavy “bad vectorization” . . . 151
12.2.3 Example: too-small n . . . 151
12.2.4 Rule . . . 152
12.3 Maintainability vs Raw Speed . . . 152
12.3.1 Prefer stable abstractions . . . 152
12.3.2 Example: single-purpose kernel interface . . . 153
12.3.3 Keep intrinsics local, not everywhere . . . 153
12.3.4 Correctness-first rule . . . 153
12.4 SIMD in Large Codebases . . . 153
12.4.1 Architecture patterns that scale . . . 153
12.4.2 Avoid ISA mixing hazards . . . 154
12.4.3 Testing strategy for large systems . . . 154
12.5 Long-Term Portability Considerations . . . 155
12.5.1 Portability is an architectural decision . . . 155
12.5.2 Design for multiple implementations . . . 155
12.5.3 Avoid over-specialization . . . 155
12.5.4 A practical “SIMD contract” for kernel APIs . . . 156
12.5.5 Assembly intuition: portable performance comes from stable loop shapes 156

Appendices **158**

Appendix A — SIMD Instruction Reference Overview . . . 158
Common SSE Instruction Groups . . . 158
Common AVX / AVX2 Instruction Groups . . . 160
Load, Compute, Shuffle, Store Categories . . . 163

Appendix B — SIMD Performance Rules of Thumb 165
Alignment and Access Rules 165
Instruction Selection Heuristics 167
Loop Structure Guidelines 168
Register Pressure Considerations 170
Appendix C — Common SIMD Anti-Patterns 172
Over-Vectorization 172
Scalar-SIMD Transitions 174
Excessive Shuffling 175
Memory-Dominated Vector Code 177
Appendix D — Tooling and Inspection Techniques 179
Compiler Flags for SIMD Visibility 179
Disassembly and Analysis Workflow 181
Microbenchmark Framework Design 183

References **187**
x86 Architecture Manuals (Conceptual) 187
SIMD and Vectorization Literature 188
Compiler Optimization Documentation 189
Performance Engineering Foundations 189

Preface

Scope and Philosophy of This Booklet

This booklet is a practical, systems-oriented guide to SIMD on modern x86, focusing on **repeatable patterns** that survive compiler changes, microarchitectural differences, and real-world constraints. The goal is not to list instructions, but to teach you how to:

- recognize when a loop is SIMD-friendly,
- choose a **data layout** and **loop shape** that vectorizes cleanly,
- write **manual SIMD** with intrinsics when it matters,
- validate performance with correct measurement,
- keep SIMD code maintainable in large codebases.

SIMD is treated here as an engineering tool. Each chapter is structured around:

- **Constraints** (alignment, aliasing, tails, NaNs/denormals, cache effects),
- **Pattern** (load–compute–store template),
- **Correctness hooks** (edge cases, fallbacks, invariants),
- **Performance validation** (what to measure and why).

Core policy: this booklet covers **SSE, AVX, AVX2** practical patterns. It does not require AVX-512, and it does not mix concepts from unrelated domains (e.g., atomics/memory ordering).

A note about portability inside x86: You should always assume that your executable may run on different generations of x86 CPUs. Therefore, patterns are written so they can be:

- compiled to a baseline (e.g., SSE2),
- optionally dispatched to faster variants (AVX2),
- validated with identical output behavior (within defined FP tolerances).

```
/* Minimal pattern philosophy: keep scalar reference + one or more SIMD
↪  paths. */

float dot_scalar(const float* a, const float* b, int n) {
    float sum = 0.0f;
    for (int i = 0; i < n; ++i) sum += a[i] * b[i];
    return sum;
}

/* SIMD paths must match scalar semantics within your accepted FP tolerance
↪  policy. */
```

Why SIMD Still Matters on Modern x86

Modern x86 CPUs are wide, superscalar, and out-of-order, yet most real workloads are bottlenecked by one (or more) of these:

- **memory bandwidth** (moving data dominates),
- **instruction throughput** (not enough work per cycle),

- **front-end pressure** (too many instructions / poor locality),
- **branching and dependencies** (pipeline disruptions).

SIMD helps by increasing **useful work per instruction** and reducing loop overhead. Even when a loop is memory-bound, SIMD can still matter because:

- fewer instructions are decoded and retired per element,
- address generation and loop control overhead shrink,
- better use of load/store units is often possible,
- the code becomes more predictable and easier to optimize.

Example: "map" transform (multiply by constant). The SIMD version does 8 floats per iteration with AVX, reducing control overhead and using vector FP pipelines efficiently.

```
#include <immintrin.h>

void scale_avx(float* x, float c, int n) {
    const __m256 vc = _mm256_set1_ps(c);

    int i = 0;
    for (; i + 8 <= n; i += 8) {
        __m256 vx = _mm256_loadu_ps(x + i);
        vx = _mm256_mul_ps(vx, vc);
        _mm256_storeu_ps(x + i, vx);
    }

    /* Tail: keep it simple and correct. */
    for (; i < n; ++i) x[i] *= c;
}
```

Example: reduction (sum). Reductions stress dependency chains; SIMD can still win by accumulating partial sums in registers and reducing once per loop chunk.

```
#include <immintrin.h>

float sum_avx(const float* x, int n) {
    __m256 acc = _mm256_setzero_ps();

    int i = 0;
    for (; i + 8 <= n; i += 8) {
        __m256 vx = _mm256_loadu_ps(x + i);
        acc = _mm256_add_ps(acc, vx);
    }

    /* Horizontal reduce acc -> scalar. */
    __m128 lo = _mm256_castps256_ps128(acc);
    __m128 hi = _mm256_extractf128_ps(acc, 1);
    __m128 s  = _mm_add_ps(lo, hi);
    s = _mm_hadd_ps(s, s);
    s = _mm_hadd_ps(s, s);

    float result = _mm_cvtss_f32(s);

    for (; i < n; ++i) result += x[i];
    return result;
}
```

Key takeaway: SIMD remains a first-class performance tool because it aligns with how x86 hardware executes: fewer instructions, more work per cycle, and clearer data-parallel intent.

What This Booklet Is Not

To keep the booklet focused and usable, it intentionally excludes:

- **An instruction encyclopedia:** you do not need to memorize every opcode to write fast SIMD.
- **A compiler manual:** we use compilers as tools; we do not re-document their full behavior.
- **A microarchitecture benchmark catalog:** we teach *methods* for measurement, not a table of numbers that goes stale.
- **A floating-point treatise:** we cover only what affects SIMD correctness/performance (NaNs, denormals, associativity, flush-to-zero behavior).
- **Cross-ISA SIMD:** ARM NEON/SVE and RISC-V V are not covered here.
- **Concurrency and atomics:** those belong to a separate booklet in this phase.

Also not included: "clever" patterns that are hard to review, hard to debug, or brittle across compilers. The preferred patterns are those you can confidently deploy in production.

Required Background and Assumptions

This booklet assumes you already know:

- C/C++ loops, pointers/references, and basic performance terminology,
- the difference between throughput and latency (at least conceptually),
- what alignment means and why misalignment may cost,
- how to compile and inspect assembly output.

Tooling assumptions:

- You can compile with optimization enabled (e.g., `-O2` or `-O3`).

- You can enable ISA targets when needed (e.g., AVX2).
- You can read **Intel-syntax** assembly when inspecting output.

A practical compilation workflow (example):

```
# Inspect auto-vectorization and generated assembly
g++ -O3 -march=native -fno-omit-frame-pointer -S -masm=intel your_file.cpp

# Build an AVX2-targeted variant (when you explicitly want that codegen)
g++ -O3 -mavx2 -mfma -fno-omit-frame-pointer your_file.cpp -o app_avx2
```

Assembly syntax note: whenever assembly is shown directly, it uses GAS with Intel syntax enabled. Comments use `/* ... */` as requested.

```
.intel_syntax noprefix
/* Example skeleton only: not a full function. */
vmovups ymm0, ymmword ptr [rdi]   /* load 8 floats */
vmulps  ymm0, ymm0, ymm1          /* multiply */
vmovups ymmword ptr [rsi], ymm0   /* store */
.att_syntax prefix
```

Correctness assumptions:

- For floating point, you must define what "same result" means (bitwise identical vs within tolerance).
- For reductions, you must accept that reassociation may change rounding unless you enforce strict FP rules.

How to Read and Apply the Patterns

Each pattern in this booklet is meant to be **copied into a real codebase** with minimal adaptation. Apply them in this order:

1) Start with a scalar reference and a measurable baseline

Write the simplest correct version first, and ensure you can measure it.

```
/* Baseline: correctness first. */
int count_gt_scalar(const float* x, int n, float t) {
    int cnt = 0;
    for (int i = 0; i < n; ++i) cnt += (x[i] > t);
    return cnt;
}
```

2) Make data layout and loop shape SIMD-friendly

Before intrinsics, remove blockers:

- avoid pointer aliasing surprises (separate input/output buffers),
- prefer contiguous memory and predictable strides,
- keep hot loops small and straight-line when possible,
- handle tails cleanly (scalar tail is acceptable and often best).

3) Apply the SIMD pattern with a clear tail strategy

Example: vectorized count using AVX comparisons and bitmask extraction.

```
#include <immintrin.h>

int count_gt_avx(const float* x, int n, float t) {
    const __m256 vt = _mm256_set1_ps(t);

    int cnt = 0;
    int i = 0;
```

```
    for (; i + 8 <= n; i += 8) {
        __m256 vx = _mm256_loadu_ps(x + i);
        __m256 m  = _mm256_cmp_ps(vx, vt, _CMP_GT_OQ);

        /* Convert mask to bits, count set lanes. */
        int bits = _mm256_movemask_ps(m);
#if defined(__GNUG__)  defined(__clang__)
        cnt += __builtin_popcount((unsigned)bits);
#else
        /* Portable fallback popcount (simple): */
        bits = bits - ((bits >> 1) & 0x55);
        bits = (bits & 0x33) + ((bits >> 2) & 0x33);
        cnt += (((bits + (bits >> 4)) & 0x0F) * 0x01);
#endif
    }

    for (; i < n; ++i) cnt += (x[i] > t);
    return cnt;
}
```

4) Validate correctness with edge cases

Always test:

- **small sizes:** 0..(vector width + 2),
- **unaligned pointers:** offsets 1..(alignment-1),
- **non-multiple lengths:** tail coverage,
- **special FP values:** NaNs, infinities, denormals (if relevant).

5) Measure the right thing, the right way

Guidelines:

- warm up caches and branch predictors,
- avoid timing allocations and I/O,
- run enough iterations to reduce noise,
- compare scalar vs SIMD on the same input.

```cpp
#include <chrono>
#include <cstdint>

template <class F>
std::uint64_t time_ns(F&& f, int iters) {
    auto t0 = std::chrono::high_resolution_clock::now();
    for (int i = 0; i < iters; ++i) f();
    auto t1 = std::chrono::high_resolution_clock::now();
    return
     ↪  (std::uint64_t)std::chrono::duration_cast<std::chrono::nanoseconds>(t1
     ↪  - t0).count();
}
```

6) Keep maintainability: isolate SIMD and allow multiple backends

Recommended structure:

- one clean API,
- scalar reference implementation,
- one or more SIMD implementations,

- optional runtime dispatch if your distribution requires it.

```c
/* Skeleton: select implementation strategy (compile-time or runtime). */

float dot(const float* a, const float* b, int n) {
#if defined(__AVX2__)
    return dot_scalar(a, b, n); /* replace with AVX2 variant later */
#else
    return dot_scalar(a, b, n);
#endif
}
```

Final message of this booklet: SIMD performance is not magic. It is disciplined engineering: **data layout + loop shape + correct tails + correct measurement + maintainable structure**.

Chapter 1

SIMD Fundamentals on x86

1.1 What SIMD Really Means at the Microarchitectural Level

At the instruction-set level, **SIMD** (Single Instruction, Multiple Data) means one instruction operates on multiple "lanes" of data packed inside a vector register. At the microarchitectural level, SIMD matters because it changes:

- **work per decoded instruction** (one instruction produces multiple results),
- **pressure on the front-end** (fewer instructions to fetch/decode/rename),
- **pressure on execution ports** (vector ALUs and vector load/store paths are used),
- **dependency structure** (fewer loop-control dependencies, different reduction shapes),
- **register pressure and scheduling** (wider values, fewer independent live vectors may fit).

A key reality: SIMD does *not* guarantee speedup. It increases *potential throughput* if your loop is structured so the CPU can feed vector units efficiently and memory can deliver data.

1.1.1 The canonical SIMD loop shape

Most high-performance SIMD loops are variations of:

- **Load** a vector (often unaligned),
- **Compute** with 1–3 vector ops,
- **Store** a vector (or accumulate for reduction),
- handle the **tail** (remaining elements).

```
#include <immintrin.h>

/* y[i] = a*x[i] + y[i]  (AXPY), AVX form */
void axpy_avx(float* y, const float* x, float a, int n) {
    const __m256 va = _mm256_set1_ps(a);

    int i = 0;
    for (; i + 8 <= n; i += 8) {
        __m256 vx = _mm256_loadu_ps(x + i);
        __m256 vy = _mm256_loadu_ps(y + i);
        vy = _mm256_fmadd_ps(va, vx, vy);       /* vy = a*vx + vy */
        _mm256_storeu_ps(y + i, vy);
    }
    for (; i < n; ++i) y[i] = a * x[i] + y[i];
}
```

1.1.2 What the core “sees”

Even without exact port tables, you should reason in this order:

1. Is the loop **memory-bound** (loads/stores dominate)?
2. Is it **compute-bound** (vector ALU throughput dominates)?
3. Is it **front-end/branch-bound** (too many instructions or unpredictable control flow)?

SIMD helps most when the loop is compute-bound or front-end-bound; for memory-bound loops, SIMD can still help by reducing overhead, but the speedup is limited by bandwidth.

1.2 Vector Registers vs Scalar Execution

Vector registers are architectural containers (XMM/YMM/ZMM) that hold multiple elements. Scalar registers hold one element. On x86, many scalar floating-point operations also use XMM registers, but with **only lane 0** as meaningful input/output.

1.2.1 Lane semantics

- **Scalar:** 1 element computed per instruction.
- **Vector:** W elements computed per instruction (W = lanes).

Example (conceptual): element-wise add.

```
void add_scalar(float* c, const float* a, const float* b, int n) {
    for (int i = 0; i < n; ++i) c[i] = a[i] + b[i];
}

#include <immintrin.h>
void add_avx(float* c, const float* a, const float* b, int n) {
    int i = 0;
    for (; i + 8 <= n; i += 8) {
        __m256 va = _mm256_loadu_ps(a + i);
```

```
        __m256 vb = _mm256_loadu_ps(b + i);
        __m256 vc = _mm256_add_ps(va, vb);
        _mm256_storeu_ps(c + i, vc);
    }
    for (; i < n; ++i) c[i] = a[i] + b[i];
}
```

1.2.2 Architectural width vs physical execution

Architectural vector width does not always map to one “single” physical operation internally. Depending on CPU generation, a wide vector op might:

- execute as one wide op,
- or be handled as multiple internal chunks (still often efficient),
- or contend for fewer vector execution resources than you expect.

Therefore, measure on target machines and prefer patterns that are robust under these differences.

1.3 Data Parallelism vs Instruction-Level Parallelism

1.3.1 Data Parallelism (DLP)

DLP means doing the same operation on many independent data elements. This is exactly what SIMD exploits. Good DLP loops have:

- contiguous or predictable memory access,
- few loop-carried dependencies,
- limited divergence (branchless or uniform branches).

1.3.2 Instruction-Level Parallelism (ILP)

ILP is the CPU running multiple independent instructions in parallel (out-of-order execution). ILP can speed up scalar code *without* SIMD if you expose independent work.

1.3.3 They complement each other

SIMD increases work per instruction (DLP), while unrolling / multiple accumulators increase independent instruction streams (ILP). A classic example is a **reduction**: naive code is dependency-bound.

```
/* Dependency chain: each iteration depends on previous sum */
float sum_scalar(const float* x, int n) {
    float s = 0.0f;
    for (int i = 0; i < n; ++i) s += x[i];
    return s;
}
```

Improve ILP using multiple accumulators (still scalar):

```
float sum_scalar_4acc(const float* x, int n) {
    float s0 = 0.0f, s1 = 0.0f, s2 = 0.0f, s3 = 0.0f;
    int i = 0;
    for (; i + 4 <= n; i += 4) {
        s0 += x[i + 0];
        s1 += x[i + 1];
        s2 += x[i + 2];
        s3 += x[i + 3];
    }
    float s = (s0 + s1) + (s2 + s3);
    for (; i < n; ++i) s += x[i];
    return s;
}
```

Then combine with SIMD + ILP (multiple vector accumulators):

```
#include <immintrin.h>

float sum_avx_2acc(const float* x, int n) {
    __m256 a0 = _mm256_setzero_ps();
    __m256 a1 = _mm256_setzero_ps();

    int i = 0;
    for (; i + 16 <= n; i += 16) {
        __m256 v0 = _mm256_loadu_ps(x + i);
        __m256 v1 = _mm256_loadu_ps(x + i + 8);
        a0 = _mm256_add_ps(a0, v0);
        a1 = _mm256_add_ps(a1, v1);
    }

    __m256 acc = _mm256_add_ps(a0, a1);

    /* Horizontal reduction */
    __m128 lo = _mm256_castps256_ps128(acc);
    __m128 hi = _mm256_extractf128_ps(acc, 1);
    __m128 s  = _mm_add_ps(lo, hi);
    s = _mm_hadd_ps(s, s);
    s = _mm_hadd_ps(s, s);

    float result = _mm_cvtss_f32(s);
    for (; i < n; ++i) result += x[i];
    return result;
}
```

Rule: reductions are often limited by dependency chains; use ILP (multiple accumulators) and SIMD together.

1.4 SIMD Width Evolution: 128 → 256 → 512

On x86, the mainstream progression is:

- **SSE:** 128-bit vectors (XMM) — widely available baseline.
- **AVX/AVX2:** 256-bit vectors (YMM) — more lanes, VEX encoding, better throughput patterns.
- **AVX-512:** 512-bit vectors (ZMM) + opmask registers — more lanes and masking features.

1.4.1 Practical consequences of widening

- Wider vectors increase **lanes** (e.g., float: 4 lanes at 128-bit, 8 at 256-bit, 16 at 512-bit).
- Wider vectors can increase **register pressure** (fewer independent live vectors fit).
- Wider vectors can expose **frequency/thermal behavior** differences on some CPUs (important for performance predictability).
- Wider vectors improve **tail amortization** (fewer loop iterations), but tail-handling complexity may grow if you insist on fully-vector tails.

1.4.2 A robust width strategy

A production-safe strategy often looks like:

- baseline implementation (scalar or SSE2),
- optimized implementation (AVX2),

- optional higher-tier implementation (AVX-512) only when you control deployment hardware and have validated power/frequency behavior.

```
/* Example: compile-time selection (runtime dispatch is a later chapter
↪  topic). */

float sum_portable(const float* x, int n) {
#if defined(__AVX2__)
    return sum_avx_2acc(x, n);
#else
    return sum_scalar_4acc(x, n);
#endif
}
```

1.4.3 Intel-syntax assembly intuition (GAS with Intel syntax)

Below is an illustrative snippet of a vector load/add/store pattern (not a full function).

```
.intel_syntax noprefix
/* ymm0 = [rdi], ymm1 = [rsi], ymm0 += ymm1, store to [rdx] */
vmovups ymm0, ymmword ptr [rdi]
vmovups ymm1, ymmword ptr [rsi]
vaddps  ymm0, ymm0, ymm1
vmovups ymmword ptr [rdx], ymm0
.att_syntax prefix
```

1.5 Costs and Benefits of Vectorization

1.5.1 Benefits

- **Higher throughput:** more elements processed per instruction.

- **Lower loop overhead:** fewer branches and index updates per element.
- **Less front-end pressure:** fewer instructions to fetch/decode/rename.
- **Better arithmetic intensity:** often improves the compute-to-overhead ratio.
- **Branch removal:** many per-element branches can be converted to compare+mask+blend.

1.5.2 Costs

- **Register pressure:** vector temporaries can reduce ILP and increase spills.
- **Shuffles/permutations:** rearranging lanes can be expensive if overused.
- **Tail handling:** remainder elements must be handled safely and correctly.
- **Alignment and aliasing constraints:** poor layout can erase SIMD gains.
- **Numerical differences:** vectorized reductions may reorder operations and change rounding.
- **Deployment variability:** ISA availability and CPU behavior vary across machines.

1.5.3 A realistic "win/lose" checklist

Vectorization is likely to win when:

- the loop is hot and dominates runtime,
- memory access is contiguous or predictable,
- there are few loop-carried dependencies,

- per-element work is non-trivial (or the loop is front-end-limited),
- tails are handled simply (scalar tail is acceptable),
- you can validate correctness and measure properly.

Vectorization is likely to disappoint when:

- the loop is strongly memory-bandwidth-bound (already saturating bandwidth),
- the loop requires frequent shuffles/gathers with poor locality,
- branch divergence is high and cannot be masked cheaply,
- register spills dominate due to high live ranges.

1.5.4 Extensive example: turning branches into masks

Branchy scalar code:

```
void clamp_scalar(float* x, int n, float lo, float hi) {
    for (int i = 0; i < n; ++i) {
        if (x[i] < lo) x[i] = lo;
        else if (x[i] > hi) x[i] = hi;
    }
}
```

Branchless SIMD clamp (AVX): `min(max(x, lo), hi)`.

```
#include <immintrin.h>

void clamp_avx(float* x, int n, float lo, float hi) {
    const __m256 vlo = _mm256_set1_ps(lo);
    const __m256 vhi = _mm256_set1_ps(hi);
```

```
    int i = 0;
    for (; i + 8 <= n; i += 8) {
        __m256 v = _mm256_loadu_ps(x + i);
        v = _mm256_max_ps(v, vlo);
        v = _mm256_min_ps(v, vhi);
        _mm256_storeu_ps(x + i, v);
    }
    for (; i < n; ++i) {
        float v = x[i];
        if (v < lo) v = lo;
        if (v > hi) v = hi;
        x[i] = v;
    }
}
```

1.5.5 Extensive example: layout impact (AoS vs SoA)

AoS (harder to vectorize when you only need one field):

```
struct Particle { float x, y, z, w; }; /* Example payload */

void scale_x_aos(Particle* p, int n, float s) {
    for (int i = 0; i < n; ++i) p[i].x *= s; /* x is strided within structs
    ↪  */
}
```

SoA (friendly contiguous loads):

```
struct ParticlesSOA { float* x; float* y; float* z; float* w; };

#include <immintrin.h>
void scale_x_soa_avx(ParticlesSOA p, int n, float s) {
    const __m256 vs = _mm256_set1_ps(s);
    int i = 0;
```

```
    for (; i + 8 <= n; i += 8) {
        __m256 vx = _mm256_loadu_ps(p.x + i);
        vx = _mm256_mul_ps(vx, vs);
        _mm256_storeu_ps(p.x + i, vx);
    }
    for (; i < n; ++i) p.x[i] *= s;
}
```

Bottom line: SIMD is an architectural contract between your data layout, your loop structure, and the CPU's ability to sustain vector loads/ops/stores. You do not "sprinkle intrinsics" to get performance; you design for vectorization, then validate with measurement.

Chapter 2

x86 SIMD Execution Model

2.1 SIMD Pipelines and Execution Units

On modern x86 cores, SIMD instructions are executed by **vector-capable execution units** (vector ALUs, FP units, load/store units) connected through an out-of-order backend. The important practical model is:

- The front-end fetches and decodes instructions into micro-operations.
- The backend schedules these micro-operations onto **execution ports** that provide specific vector capabilities.
- Loads/stores move data between L1 cache and registers; vector ALUs perform packed integer and packed floating-point math.

2.1.1 What you should care about (portable mental model)

Without relying on CPU-specific port numbers, you can reason effectively using:

- **Load bandwidth:** how many bytes/loads per cycle can be sustained from L1.
- **Store bandwidth:** how many bytes/stores per cycle can be sustained to L1.
- **Vector ALU throughput:** how many vector adds/muls/fmas per cycle are possible if data is ready.
- **Shuffles/permutations:** often use distinct resources and may become bottlenecks.

2.1.2 A practical bottleneck example: load–compute–store

A simple vector transform often becomes limited by memory movement (loads/stores) rather than ALU throughput.

```
#include <immintrin.h>

/* y[i] = x[i] + c  -- very light compute; often memory-bound for large
↪  arrays */
void addc_avx(float* y, const float* x, float c, int n) {
    const __m256 vc = _mm256_set1_ps(c);
    int i = 0;
    for (; i + 8 <= n; i += 8) {
        __m256 vx = _mm256_loadu_ps(x + i);    /* load */
        __m256 vy = _mm256_add_ps(vx, vc);     /* compute */
        _mm256_storeu_ps(y + i, vy);           /* store */
    }
    for (; i < n; ++i) y[i] = x[i] + c;
}
```

If this is memory-bound, adding more arithmetic (e.g., extra mul/add) may have nearly zero cost until you saturate vector ALUs. This is why measuring matters.

2.2 Register Files and Rename Behavior

2.2.1 Architectural vs physical registers

x86 exposes architectural registers (XMM/YMM/ZMM), but the CPU implements a larger set of **physical registers**. Register renaming maps architectural names to physical storage to eliminate false dependencies.

2.2.2 Why renaming matters for SIMD

Renaming helps SIMD code by:

- removing **write-after-read** and **write-after-write** hazards,
- enabling aggressive out-of-order scheduling of independent vector ops,
- allowing unrolled loops with multiple accumulators to run efficiently.

However, SIMD code can still suffer from **register pressure**:

- Too many live vectors may exceed physical register availability, causing spills or scheduling limits.
- Heavy shuffle code often increases live ranges and pressure.

2.2.3 Example: register pressure from “too many temporaries”

The following style can easily inflate live vectors and restrict scheduling.

```
#include <immintrin.h>

/* Illustrative: excessive temporaries increase register pressure */
void bad_style(float* out, const float* a, const float* b, int n) {
```

```
    int i = 0;
    for (; i + 8 <= n; i += 8) {
        __m256 v0 = _mm256_loadu_ps(a + i);
        __m256 v1 = _mm256_loadu_ps(b + i);
        __m256 v2 = _mm256_mul_ps(v0, v1);
        __m256 v3 = _mm256_add_ps(v2, v0);
        __m256 v4 = _mm256_add_ps(v3, v1);
        __m256 v5 = _mm256_sub_ps(v4, v0);
        _mm256_storeu_ps(out + i, v5);
    }
    for (; i < n; ++i) out[i] = (a[i] * b[i] + a[i] + b[i]) - a[i];
}
```

A better style is to **reuse** registers and shorten live ranges.

```
#include <immintrin.h>

void better_style(float* out, const float* a, const float* b, int n) {
    int i = 0;
    for (; i + 8 <= n; i += 8) {
        __m256 v = _mm256_loadu_ps(a + i);
        __m256 w = _mm256_loadu_ps(b + i);
        v = _mm256_mul_ps(v, w);
        v = _mm256_add_ps(v, w);
        _mm256_storeu_ps(out + i, v);
    }
    for (; i < n; ++i) out[i] = a[i] * b[i] + b[i];
}
```

2.3 Throughput vs Latency for Vector Instructions

2.3.1 Definitions that matter in practice

- **Latency:** how many cycles until the result of an instruction is available to a dependent instruction.
- **Throughput:** how often the core can start a new instance of that instruction (e.g., one per cycle).

Throughput is what determines peak performance in **independent** streams (unrolled loops, multiple accumulators). Latency dominates in **dependency chains** (reductions, recurrences).

2.3.2 Example: latency-bound reduction

A naive scalar reduction is dominated by latency of the add dependency chain.

```
float sum_chain(const float* x, int n) {
    float s = 0.0f;
    for (int i = 0; i < n; ++i) s += x[i]; /* dependent chain */
    return s;
}
```

Break the chain with multiple accumulators (improves ILP, uses throughput).

```
float sum_4acc(const float* x, int n) {
    float s0 = 0.0f, s1 = 0.0f, s2 = 0.0f, s3 = 0.0f;
    int i = 0;
    for (; i + 4 <= n; i += 4) {
        s0 += x[i + 0];
        s1 += x[i + 1];
        s2 += x[i + 2];
        s3 += x[i + 3];
    }
    float s = (s0 + s1) + (s2 + s3);
```

```
    for (; i < n; ++i) s += x[i];
    return s;
}
```

Now combine with SIMD + multiple accumulators (uses both DLP and throughput).

```
#include <immintrin.h>

float sum_avx_2acc(const float* x, int n) {
    __m256 a0 = _mm256_setzero_ps();
    __m256 a1 = _mm256_setzero_ps();

    int i = 0;
    for (; i + 16 <= n; i += 16) {
        __m256 v0 = _mm256_loadu_ps(x + i);
        __m256 v1 = _mm256_loadu_ps(x + i + 8);
        a0 = _mm256_add_ps(a0, v0);
        a1 = _mm256_add_ps(a1, v1);
    }

    __m256 acc = _mm256_add_ps(a0, a1);

    __m128 lo = _mm256_castps256_ps128(acc);
    __m128 hi = _mm256_extractf128_ps(acc, 1);
    __m128 s  = _mm_add_ps(lo, hi);
    s = _mm_hadd_ps(s, s);
    s = _mm_hadd_ps(s, s);

    float result = _mm_cvtss_f32(s);
    for (; i < n; ++i) result += x[i];
    return result;
}
```

2.4 Vector Dependency Chains

A **dependency chain** occurs when each iteration depends on the previous iteration's result. Common forms:

- **reductions** (sum, min, max),
- **prefix sums / scans**,
- **recurrences** (state machines, filters where output feeds next input).

2.4.1 Recognizing dependency-limited SIMD

Even in SIMD, a reduction can remain latency-limited if you use only one accumulator:

```c
#include <immintrin.h>

/* One vector accumulator: still a dependency chain across iterations. */
float sum_avx_1acc(const float* x, int n) {
    __m256 acc = _mm256_setzero_ps();
    int i = 0;
    for (; i + 8 <= n; i += 8) {
        __m256 v = _mm256_loadu_ps(x + i);
        acc = _mm256_add_ps(acc, v); /* chain: acc depends on previous acc
        ↪  */
    }

    __m128 lo = _mm256_castps256_ps128(acc);
    __m128 hi = _mm256_extractf128_ps(acc, 1);
    __m128 s  = _mm_add_ps(lo, hi);
    s = _mm_hadd_ps(s, s);
    s = _mm_hadd_ps(s, s);
```

```
    float result = _mm_cvtss_f32(s);
    for (; i < n; ++i) result += x[i];
    return result;
}
```

2.4.2 Breaking vector chains: the multi-accumulator rule

Use 2–4 independent accumulators to increase ILP and approach throughput limits.

- Too few accumulators → latency-bound.
- Too many accumulators → register pressure and spills.

```
#include <immintrin.h>

float sum_avx_4acc(const float* x, int n) {
    __m256 a0 = _mm256_setzero_ps();
    __m256 a1 = _mm256_setzero_ps();
    __m256 a2 = _mm256_setzero_ps();
    __m256 a3 = _mm256_setzero_ps();

    int i = 0;
    for (; i + 32 <= n; i += 32) {
        a0 = _mm256_add_ps(a0, _mm256_loadu_ps(x + i +  0));
        a1 = _mm256_add_ps(a1, _mm256_loadu_ps(x + i +  8));
        a2 = _mm256_add_ps(a2, _mm256_loadu_ps(x + i + 16));
        a3 = _mm256_add_ps(a3, _mm256_loadu_ps(x + i + 24));
    }

    __m256 acc = _mm256_add_ps(_mm256_add_ps(a0, a1), _mm256_add_ps(a2,
    ↪  a3));

    __m128 lo = _mm256_castps256_ps128(acc);
    __m128 hi = _mm256_extractf128_ps(acc, 1);
```

```
    __m128 s  = _mm_add_ps(lo, hi);
    s = _mm_hadd_ps(s, s);
    s = _mm_hadd_ps(s, s);

    float result = _mm_cvtss_f32(s);
    for (; i < n; ++i) result += x[i];
    return result;
}
```

2.5 SIMD and Out-of-Order Execution

Out-of-order (OoO) cores execute instructions when their inputs are ready, not strictly in program order. SIMD code benefits strongly from OoO execution when you provide:

- enough independent work (unrolling, multiple accumulators),
- predictable memory access (so loads can be overlapped),
- limited control-flow divergence.

2.5.1 OoO-friendly SIMD pattern: software pipelining

The idea: overlap future loads with current computes and previous stores. Compilers may do some of this automatically, but manual structure helps.

```
#include <immintrin.h>

/* Overlap pattern: load next while computing current (conceptual). */
void axpy_avx_pipelined(float* y, const float* x, float a, int n) {
    const __m256 va = _mm256_set1_ps(a);

    int i = 0;
```

```
    if (n >= 16) {
        __m256 vx0 = _mm256_loadu_ps(x + 0);
        __m256 vy0 = _mm256_loadu_ps(y + 0);

        for (i = 8; i + 8 <= n; i += 8) {
            __m256 vx1 = _mm256_loadu_ps(x + i);
            __m256 vy1 = _mm256_loadu_ps(y + i);

            vy0 = _mm256_fmadd_ps(va, vx0, vy0);
            _mm256_storeu_ps(y + (i - 8), vy0);

            vx0 = vx1;
            vy0 = vy1;
        }

        vy0 = _mm256_fmadd_ps(va, vx0, vy0);
        _mm256_storeu_ps(y + (i - 8), vy0);
    }

    for (; i < n; ++i) y[i] = a * x[i] + y[i];
}
```

2.5.2 When OoO cannot save you

OoO execution has limits. SIMD performance will stall when:

- the loop is **bandwidth-bound** and already saturates memory,
- there is a **hard dependency chain** with insufficient ILP,
- there are frequent **cache misses** that exceed the core's ability to hide latency,
- the loop performs excessive **shuffles/gathers** with poor locality.

2.5.3 Assembly intuition (GAS, Intel syntax)

The following illustrates independent vector ops that OoO can overlap if dependencies allow (not a full function).

```
.intel_syntax noprefix
/* Two independent streams: ymm0 and ymm2 can progress in parallel if
 ↪  inputs are ready. */
vmovups ymm0, ymmword ptr [rdi]         /* load stream A */
vmovups ymm2, ymmword ptr [rsi]         /* load stream B */
vaddps  ymm1, ymm0, ymm3               /* compute A */
vaddps  ymm4, ymm2, ymm5               /* compute B */
vmovups ymmword ptr [rdx], ymm1         /* store A */
vmovups ymmword ptr [rcx], ymm4         /* store B */
.att_syntax prefix
```

Practical summary:

- SIMD increases work per instruction, but the core must still schedule loads, stores, and vector ALU ops.
- To approach peak throughput, break dependency chains with multiple accumulators and unrolling.
- Keep register pressure under control; too many live vectors can reduce OoO effectiveness.
- Measure on target hardware: the balance of load/store/shuffle/ALU resources differs across x86 generations.

Chapter 3

Data Layout for SIMD

3.1 Array of Structures vs Structure of Arrays

3.1.1 AoS vs SoA: what changes for SIMD

Array of Structures (AoS) stores records contiguously:

```
struct Particle { float x, y, z, w; };
Particle p[N]; /* AoS */
```

Structure of Arrays (SoA) stores each field contiguously:

```
struct ParticlesSOA { float* x; float* y; float* z; float* w; };
ParticlesSOA p; /* SoA */
```

SIMD prefers **contiguous lanes**. If you usually operate on one or two fields at a time (common in physics, graphics, signal processing, analytics), SoA typically vectorizes better:

- SoA enables **unit-stride loads/stores** for each field.
- AoS often turns field access into **strided** patterns that may require shuffles, gathers, or scalar code.

3.1.2 Example: scaling only `x`

AoS access for `x` is a stride inside each record; SoA access is contiguous.

```
/* AoS: x values are interleaved with y,z,w */
struct Particle { float x, y, z, w; };

void scale_x_aos(Particle* p, int n, float s) {
    for (int i = 0; i < n; ++i) p[i].x *= s;
}
```

```
/* SoA: x values are contiguous */
struct ParticlesSOA { float* x; float* y; float* z; float* w; };

#include <immintrin.h>
void scale_x_soa_avx(ParticlesSOA p, int n, float s) {
    const __m256 vs = _mm256_set1_ps(s);

    int i = 0;
    for (; i + 8 <= n; i += 8) {
        __m256 vx = _mm256_loadu_ps(p.x + i);
        vx = _mm256_mul_ps(vx, vs);
        _mm256_storeu_ps(p.x + i, vx);
    }
    for (; i < n; ++i) p.x[i] *= s;
}
```

3.1.3 When AoS is acceptable

AoS can still be fine when:

- you almost always use **all fields together** (full-struct computation),
- the struct is small and naturally aligned,

- the code is not hot, or auto-vectorization already succeeds.

3.1.4 Hybrid layouts: AoSoA (practical compromise)

A common compromise is **Array of Structures of Arrays (AoSoA)**: blocks of SoA packed inside cache-friendly tiles. You get contiguous lanes within a block while keeping record locality across fields.

```
/* AoSoA: block size 8 matches AVX float lanes */
struct ParticlesAoSoA8 {
    alignas(32) float x[8];
    alignas(32) float y[8];
    alignas(32) float z[8];
    alignas(32) float w[8];
};
/* Then store as array of blocks: ParticlesAoSoA8 blocks[M]; */
```

3.2 Alignment Requirements and Penalties

3.2.1 Alignment is a performance contract, not just correctness

Most modern x86 supports unaligned vector loads/stores, so correctness usually does not depend on alignment. Performance does.
Key facts you must design around:

- L1 data cache line size is typically **64 bytes**.
- A vector load that crosses a cache-line boundary may require **two cache-line accesses**.
- Misalignment can reduce sustained load/store bandwidth and increase micro-ops.

3.2.2 Practical alignment targets

- SSE (128-bit): prefer **16-byte** alignment.
- AVX/AVX2 (256-bit): prefer **32-byte** alignment.
- AVX-512 (512-bit): prefer **64-byte** alignment.

3.2.3 Correct allocation for aligned SIMD buffers

Prefer aligned allocation when you control the buffer lifetime.

```
#include <cstddef>
#include <cstdlib>
#include <new>

float* alloc_aligned_f32(std::size_t n) {
    void* p = nullptr;
#if defined(_MSC_VER)
    p = _aligned_malloc(n * sizeof(float), 32);
    if (!p) throw std::bad_alloc();
#else
    if (posix_memalign(&p, 32, n * sizeof(float)) != 0) throw
    ↪  std::bad_alloc();
#endif
    return static_cast<float*>(p);
}

void free_aligned(void* p) {
#if defined(_MSC_VER)
    _aligned_free(p);
#else
    free(p);
#endif
```

```
}
```

3.2.4 Aligned vs unaligned intrinsics

Use aligned loads/stores only when you can guarantee alignment. Otherwise use unaligned loads/stores; modern CPUs handle them well *when they do not cross cache lines*.

```
#include <immintrin.h>

void add_avx(float* y, const float* x, int n) {
    int i = 0;
    for (; i + 8 <= n; i += 8) {
        __m256 vx = _mm256_loadu_ps(x + i);   /* safe for any alignment */
        __m256 vy = _mm256_loadu_ps(y + i);
        vy = _mm256_add_ps(vy, vx);
        _mm256_storeu_ps(y + i, vy);
    }
    for (; i < n; ++i) y[i] += x[i];
}
```

3.3 Padding, Stride, and Cache-Line Interaction

3.3.1 Stride determines whether SIMD feeds efficiently

A stride of 1 element (unit stride) is ideal:

- predictable prefetching,
- full cache-line utilization,
- easy vector loads/stores.

A larger stride causes:

- more cache lines touched per useful element,
- wasted bandwidth,
- higher chance of crossing cache lines on vector loads.

3.3.2 Example: strided access is hostile to SIMD

```
/* Reads every k-th element: poor locality when k is large. */
float sum_strided(const float* x, int n, int k) {
    float s = 0.0f;
    for (int i = 0; i < n; i += k) s += x[i];
    return s;
}
```

3.3.3 Padding to avoid cache-line splits and simplify tails

Padding is often a net win when it:

- ensures vector loops can run to a rounded length,
- reduces conditional tail handling in hot paths,
- avoids out-of-bounds loads while keeping correctness clear.

Safe pattern: allocate padded length, but compute only on logical length.

```
#include <algorithm>
#include <immintrin.h>

static inline int round_up8(int n) { return (n + 7) & ~7; }

void add_padded_avx(float* y, const float* x, int n) {
    int n8 = round_up8(n);
```

```
    /* Requirement: x and y have at least n8 elements allocated.
       Elements n..n8-1 must be valid (often zero-padded). */

    int i = 0;
    for (; i < n8; i += 8) {
        __m256 vx = _mm256_loadu_ps(x + i);
        __m256 vy = _mm256_loadu_ps(y + i);
        vy = _mm256_add_ps(vy, vx);
        _mm256_storeu_ps(y + i, vy);
    }

    /* If padding is zero and you only care about y[0..n-1], you are done.
     ↪  */
}
```

3.3.4 Cache-line interaction rule of thumb

When possible:

- align arrays so the hot starting pointer is cache-line aligned,
- keep the inner loop unit-stride,
- avoid mixing unrelated streams in the same tight loop if it increases cache pressure.

3.4 False Sharing and Vector Loads

3.4.1 What false sharing is

False sharing occurs when two threads write to different variables that happen to reside on the same cache line. The cache-coherence protocol then forces expensive invalidations even though the variables are logically independent.

SIMD interacts with this because vector stores typically write 16/32/64 bytes at a time, increasing the chance that your stores touch shared cache lines if your per-thread partitions are not cache-line separated.

3.4.2 Bad pattern: per-thread counters in the same cache line

```
#include <cstdint>

struct CountersBad {
    std::uint64_t c[8]; /* likely fits in one 64B cache line */
};

/* Different threads updating c[tid] can fight over the same cache line. */
```

3.4.3 Good pattern: pad each thread's writable data to a cache line

```
#include <cstddef>
#include <cstdint>

struct alignas(64) CounterPadded {
    std::uint64_t value;
    std::uint8_t  pad[64 - sizeof(std::uint64_t)];
};

struct CountersGood {
    CounterPadded c[64]; /* up to 64 threads without sharing lines */
};
```

3.4.4 SIMD partitioning rule

If multiple threads process a large array:

- split the array into **non-overlapping, cache-line-aligned chunks**,

- ensure each chunk begins on a cache-line boundary (64B) when possible,
- avoid having two threads write to different elements within the same cache line near chunk boundaries.

3.5 Designing SIMD-Friendly Memory Layouts

3.5.1 The design checklist

A SIMD-friendly layout typically satisfies:

- **unit-stride** access in the hottest loops,
- **predictable alignment** (at least vector-width alignment when feasible),
- minimal **pointer aliasing** between inputs/outputs,
- clear handling for **tails** (scalar tail or padded buffers),
- minimal **gathers/scatters** and minimal shuffling.

3.5.2 Alias control: separate input and output buffers

When inputs and outputs may alias, compilers and humans must assume worst-case overlap, which can block vectorization and safe reordering.

```
/* Good: separate buffers clearly describe non-alias intent. */
void saxpy(float* y, const float* x, float a, int n) {
    for (int i = 0; i < n; ++i) y[i] = a * x[i] + y[i];
}
```

3.5.3 SoA + alignment + tail: a production-quality pattern

```
#include <immintrin.h>

struct VecF32 {
    float* p;
    int n;
};

void mul_add_soa_avx(VecF32 out, VecF32 a, VecF32 b, float c) {
    const __m256 vc = _mm256_set1_ps(c);

    int i = 0;
    for (; i + 8 <= out.n; i += 8) {
        __m256 va = _mm256_loadu_ps(a.p + i);
        __m256 vb = _mm256_loadu_ps(b.p + i);
        __m256 r  = _mm256_add_ps(_mm256_mul_ps(va, vb), vc);
        _mm256_storeu_ps(out.p + i, r);
    }
    for (; i < out.n; ++i) out.p[i] = a.p[i] * b.p[i] + c;
}
```

3.5.4 AoSoA block layout: maximizing locality and SIMD lanes

AoSoA is useful when you frequently process multiple fields together, but still want contiguous lanes.

```
#include <immintrin.h>

/* Block of 8 (AVX float lanes). */
struct Block8 {
    alignas(32) float x[8];
    alignas(32) float y[8];
```

```
};

void add_blocks_avx(Block8* out, const Block8* a, const Block8* b, int
↪  blocks) {
    for (int i = 0; i < blocks; ++i) {
        __m256 ax = _mm256_load_ps(a[i].x);
        __m256 ay = _mm256_load_ps(a[i].y);
        __m256 bx = _mm256_load_ps(b[i].x);
        __m256 by = _mm256_load_ps(b[i].y);

        _mm256_store_ps(out[i].x, _mm256_add_ps(ax, bx));
        _mm256_store_ps(out[i].y, _mm256_add_ps(ay, by));
    }
}
```

3.5.5 Assembly intuition: cache-line crossing matters

Unaligned loads are usually fine, but crossing cache lines can cost extra work. This snippet illustrates a unit-stride vector load/store pattern (not a full function).

```
.intel_syntax noprefix
/* Unit-stride stream: best case for caches and prefetchers. */
vmovups ymm0, ymmword ptr [rdi]         /* load 32B */
vaddps  ymm0, ymm0, ymm1               /* compute */
vmovups ymmword ptr [rsi], ymm0         /* store 32B */
.att_syntax prefix
```

Practical summary:

- Prefer **SoA** (or **AoSoA**) when you operate on subsets of fields.
- Design for **unit-stride** access and **cache-line-friendly** starts.

- Use alignment to reduce cache-line splits; unaligned intrinsics are safe, but layout still matters.
- Avoid false sharing by padding thread-writable data to **cache lines** and partitioning arrays carefully.
- Treat padding and tails as correctness-first engineering: the fastest SIMD is the SIMD that stays correct.

Chapter 4

SSE Programming Model

4.1 XMM Registers and Instruction Classes

4.1.1 XMM registers: the 128-bit SIMD baseline

SSE introduces **XMM** registers: 128-bit wide architectural registers used for packed floating-point and (later) packed integer operations. In practice:

- You treat an XMM register as a vector of lanes: e.g., `4x float`, `2x double`, `16x int8`, `8x int16`, `4x int32`, `2x int64`.
- Many scalar floating-point ops also use XMM (lane 0), but SIMD uses all lanes.
- SSE is a stable baseline on x86-64 (SSE2 is effectively universal there).

4.1.2 Instruction classes (practical grouping)

SSE operations fall into classes you must recognize:

- **Load/Store:** move data between memory and XMM.

- **Arithmetic:** add/sub/mul/div for packed FP; add/sub/mul variants for integer.
- **Compare/Test:** produce masks for selection.
- **Logical:** and/or/xor, bitwise operations.
- **Shift:** packed shifts for integer lanes.
- **Shuffle/Unpack:** rearrange lanes (often the real cost center).
- **Convert:** int ↔ float, float precision changes.

```
#include <xmmintrin.h>   /* SSE */
#include <emmintrin.h>   /* SSE2 (integers, double) */
```

4.2 Packed Integer vs Packed Floating-Point

4.2.1 Two different universes

Packed floating-point and packed integer SIMD share registers but differ in:

- **semantics** (rounding, NaNs/Inf, denormals for FP),
- **available operations** (e.g., integer multiply widths, saturating arithmetic),
- **comparison behavior** (ordered/unordered FP comparisons vs exact integer compares),
- **common idioms** (bitmask extraction for compares differs).

4.2.2 Example: packed float add (4 floats)

```
#include <xmmintrin.h>

void add4_ps(float* c, const float* a, const float* b) {
```

```
    __m128 va = _mm_loadu_ps(a);
    __m128 vb = _mm_loadu_ps(b);
    __m128 vc = _mm_add_ps(va, vb);
    _mm_storeu_ps(c, vc);
}
```

4.2.3 Example: packed int32 add (4 int32)

```
#include <emmintrin.h>

void add4_epi32(int* c, const int* a, const int* b) {
    __m128i va = _mm_loadu_si128((const __m128i*)a);
    __m128i vb = _mm_loadu_si128((const __m128i*)b);
    __m128i vc = _mm_add_epi32(va, vb);
    _mm_storeu_si128((__m128i*)c, vc);
}
```

4.2.4 FP comparisons produce masks, not booleans

FP compare yields a vector mask: each lane becomes all-ones or all-zeros (bitwise), enabling blend/select via bitwise logic.

```
#include <xmmintrin.h>

/* out[i] = (x[i] > t) ? x[i] : 0 */
void keep_gt_ps(float* out, const float* x, float t) {
    __m128 vx = _mm_loadu_ps(x);
    __m128 vt = _mm_set1_ps(t);
    __m128 m  = _mm_cmpgt_ps(vx, vt);
    __m128 r  = _mm_and_ps(vx, m);
    _mm_storeu_ps(out, r);
}
```

4.3 Load / Store Semantics

4.3.1 Aligned vs unaligned

SSE provides both aligned and unaligned load/store forms. Correctness is straightforward:

- **aligned** loads/stores require aligned addresses (typically 16B for XMM),
- **unaligned** forms work for any address.

Performance depends on whether accesses cross cache-line boundaries and on microarchitecture; therefore, prefer aligned allocation when you control buffers, but use unaligned loads for general pointers.

4.3.2 Typical intrinsics

- `_mm_load_ps` / `_mm_store_ps`: aligned float loads/stores.
- `_mm_loadu_ps` / `_mm_storeu_ps`: unaligned float loads/stores.
- `_mm_loadu_si128` / `_mm_storeu_si128`: unaligned integer loads/stores (SSE2).

```
#include <xmmintrin.h>
#include <emmintrin.h>

void copy16_bytes(void* dst, const void* src) {
    __m128i v = _mm_loadu_si128((const __m128i*)src);
    _mm_storeu_si128((__m128i*)dst, v);
}
```

4.3.3 Streaming stores (when appropriate)

Non-temporal (streaming) stores can reduce cache pollution for large write-only outputs, but they require careful use:

- best when writing large buffers once,
- avoid when data will be read soon (it would be a cache miss),
- ensure alignment and store size rules for effectiveness.

```c
#include <emmintrin.h>

/* Example: streaming store of 16B (SSE2) */
void store_stream_si128(__m128i* dst, __m128i v) {
    _mm_stream_si128(dst, v);
}
```

4.4 Shuffle, Permute, and Blend Operations

4.4.1 Why shuffles matter

Shuffles/permutes are the primary cost center in many SIMD algorithms. Excessive lane rearrangement can dominate runtime even when arithmetic is cheap. Therefore:

- prefer algorithms that keep lanes aligned with natural data order,
- treat shuffles as “expensive glue”—use only when they reduce total work.

4.4.2 Core SSE lane rearrangement tools

- **Unpack:** interleave lanes (good for transpose-like operations).
- **Shuffle:** reorder lanes within a vector (float shuffle).
- **Shuffle bytes:** rearrange bytes within 128-bit (SSSE3), powerful but easy to abuse.
- **Blend:** select lanes from two vectors (SSE4.1); without it use mask logic.

4.4.3 Example: unpack interleave (transpose building block)

```
#include <xmmintrin.h>

/* Interleave low halves: (a0,b0,a1,b1) */
static inline __m128 interleave_lo(__m128 a, __m128 b) {
    return _mm_unpacklo_ps(a, b);
}

/* Interleave high halves: (a2,b2,a3,b3) */
static inline __m128 interleave_hi(__m128 a, __m128 b) {
    return _mm_unpackhi_ps(a, b);
}
```

4.4.4 Example: shuffle to broadcast a lane

```
#include <xmmintrin.h>

/* Broadcast lane 0 across all lanes */
static inline __m128 splat0(__m128 v) {
    return _mm_shuffle_ps(v, v, _MM_SHUFFLE(0,0,0,0));
}
```

4.4.5 Example: blend using mask logic (portable across SSE)

If you do not rely on SSE4.1 blend, use compare mask + and/or.

```
#include <xmmintrin.h>

/* select: m ? a : b   (m is all-ones/all-zeros per lane) */
static inline __m128 select_ps(__m128 m, __m128 a, __m128 b) {
    return _mm_or_ps(_mm_and_ps(m, a), _mm_andnot_ps(m, b));
}
```

4.4.6 Example: byte shuffle (SSSE3) for packed data

This is powerful for parsing and rearranging bytes (e.g., RGBA transforms), but keep masks constant and avoid chaining many shuffles.

```
#include <tmmintrin.h> /* SSSE3 */

static inline __m128i reverse_bytes_16(__m128i v) {
    const __m128i m = _mm_setr_epi8(
        15,14,13,12,11,10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0
    );
    return _mm_shuffle_epi8(v, m);
}
```

4.5 Common SSE Idioms

4.5.1 Idiom 1: “Load–Compute–Store” with scalar tail

This is the baseline for almost everything.

```
#include <xmmintrin.h>

void add_sse(float* y, const float* x, int n) {
    int i = 0;
    for (; i + 4 <= n; i += 4) {
        __m128 vx = _mm_loadu_ps(x + i);
        __m128 vy = _mm_loadu_ps(y + i);
        vy = _mm_add_ps(vy, vx);
        _mm_storeu_ps(y + i, vy);
    }
    for (; i < n; ++i) y[i] += x[i];
}
```

4.5.2 Idiom 2: reduction with horizontal add (SSE3) or manual

Horizontal reductions collapse lanes. SSE3 provides `hadd`, but a manual method is often clearer and avoids overusing specialized ops.

```c
#include <xmmintrin.h>

static inline float hsum_ps_sse(__m128 v) {
    /* v = [a b c d] */
    __m128 shuf = _mm_movehdup_ps(v);        /* [b b d d] (SSE3) */
    __m128 sums = _mm_add_ps(v, shuf);       /* [a+b b+b c+d d+d] */
    shuf = _mm_movehl_ps(shuf, sums);        /* [c+d d+d ? ?] */
    sums = _mm_add_ss(sums, shuf);           /* lane0 = (a+b) + (c+d) */
    return _mm_cvtss_f32(sums);
}
```

If SSE3 is not available, use shuffle + add patterns. In this booklet, SSE2 and above are assumed as a practical baseline on x86-64.

4.5.3 Idiom 3: compare to mask, then use bitwise select

A standard branchless selection idiom.

```c
#include <xmmintrin.h>

/* out[i] = (a[i] < b[i]) ? a[i] : b[i]   (min) */
void min_sse(float* out, const float* a, const float* b) {
    __m128 va = _mm_loadu_ps(a);
    __m128 vb = _mm_loadu_ps(b);
    __m128 m  = _mm_cmplt_ps(va, vb);
    __m128 r  = _mm_or_ps(_mm_and_ps(m, va), _mm_andnot_ps(m, vb));
    _mm_storeu_ps(out, r);
}
```

4.5.4 Idiom 4: integer lane widening and packing

Packed integer work often needs widening (avoid overflow) and then packing.

```
#include <emmintrin.h>

/* Example: widen 8-bit unsigned to 16-bit, then add */
void add_u8_as_u16(const unsigned char* a, const unsigned char* b, unsigned
↪  short* out) {
    __m128i va = _mm_loadu_si128((const __m128i*)a);
    __m128i vb = _mm_loadu_si128((const __m128i*)b);

    __m128i z  = _mm_setzero_si128();
    __m128i a0 = _mm_unpacklo_epi8(va, z); /* 8 x u16 */
    __m128i b0 = _mm_unpacklo_epi8(vb, z);
    __m128i s0 = _mm_add_epi16(a0, b0);

    _mm_storeu_si128((__m128i*)out, s0);
}
```

4.5.5 Idiom 5: avoid slow paths by keeping data “lane-aligned”

If your algorithm forces frequent cross-lane rearrangement, consider redesigning the data layout (SoA/AoSoA) or changing the loop nest. Shuffles are valuable, but the best SSE code uses them as little as possible.

4.5.6 Assembly intuition (GAS, Intel syntax)

Illustrative SSE load/add/store and compare-mask-select patterns (not full functions).

```
.intel_syntax noprefix
/* add 4 floats: xmm0 = [rdi], xmm1 = [rsi], xmm0 += xmm1, store */
```

```
movups xmm0, xmmword ptr [rdi]
movups xmm1, xmmword ptr [rsi]
addps  xmm0, xmm0
movups xmmword ptr [rdx], xmm0
.att_syntax prefix
```

```
.intel_syntax noprefix
/* compare + mask select: r = (a < b) ? a : b, using and/andn/or */
movups xmm0, xmmword ptr [rdi]      /* a */
movups xmm1, xmmword ptr [rsi]      /* b */
cmpltps xmm2, xmm0, xmm1            /* mask in xmm2 (conceptual form)
 ↪  */
andps   xmm0, xmm2                 /* a & mask */
andnps  xmm2, xmm1                 /* (~mask) & b */
orps    xmm0, xmm2                 /* combine -> result in xmm0 */
movups  xmmword ptr [rdx], xmm0
.att_syntax prefix
```

Practical summary:

- XMM is the 128-bit SIMD baseline; SSE2 makes packed integer and double-precision SIMD practical.
- Packed FP and packed integer code have different pitfalls; treat them as different problem spaces.
- Prefer unaligned loads/stores unless you can guarantee alignment; performance depends on cache-line splits.
- Shuffles/unpacks/blends are powerful but often the bottleneck; redesign layout/loop shape to reduce them.

- Common SSE idioms revolve around mask-based selection, multi-accumulator reductions, and minimal lane movement.

Chapter 5

AVX and AVX2 Extensions

5.1 YMM Registers and 256-bit Execution

5.1.1 From XMM (128) to YMM (256)

AVX extends the SIMD register file by defining **YMM** registers as 256-bit wide. Conceptually:

- **XMM** is the low 128 bits.
- **YMM** adds an upper 128-bit half.

This increases lane count:

- `float`: 8 lanes per YMM (vs 4 in XMM),
- `double`: 4 lanes per YMM (vs 2 in XMM),
- packed integers: twice as many lanes as SSE2.

5.1.2 Practical consequences

- Wider vectors reduce loop overhead (half as many iterations for same work).
- Wider vectors increase **register bandwidth** demands and can raise register pressure.
- Many workloads remain **memory-bound**; the speedup is limited by load/store bandwidth and cache behavior.

```
#include <immintrin.h>

/* y[i] = x[i] + c, AVX: 8 floats per iteration */
void addc_avx(float* y, const float* x, float c, int n) {
    const __m256 vc = _mm256_set1_ps(c);
    int i = 0;
    for (; i + 8 <= n; i += 8) {
        __m256 vx = _mm256_loadu_ps(x + i);
        __m256 vy = _mm256_add_ps(vx, vc);
        _mm256_storeu_ps(y + i, vy);
    }
    for (; i < n; ++i) y[i] = x[i] + c;
}
```

5.2 VEX Encoding and Its Implications

5.2.1 What VEX changes (the programmer-visible effects)

AVX introduces **VEX-encoded** instructions. The practical implications:

- **Three-operand form**: `dst = src1 op src2` without destroying a source.
- **Non-destructive semantics**: enables better scheduling and reduces register moves.
- **Unified scalar+vector forms**: many scalar FP ops become VEX-encoded too.

5.2.2 Why three-operand matters

SSE arithmetic is often two-operand and overwrites one input:

```
#include <xmmintrin.h>
/* SSE: must overwrite a register holding one operand */
__m128 sse_add(__m128 a, __m128 b) { return _mm_add_ps(a, b); }
```

AVX form can keep inputs intact (conceptually), improving code generation and reducing register shuffles:

```
#include <immintrin.h>
__m256 avx_add(__m256 a, __m256 b) { return _mm256_add_ps(a, b); }
```

5.2.3 Practical coding guideline

When writing intrinsics, prefer AVX forms consistently within a hot region to avoid mixing legacy SSE encodings with VEX forms. This supports clean register-lifetime behavior and reduces transition pitfalls (see later section).

5.3 Fused Multiply-Add (FMA)

5.3.1 What FMA does

FMA performs **multiply and add as one operation**:

$$r = a \times b + c$$

The fused form performs a single rounding step for floating-point, improving both:

- **performance** (one instruction instead of mul+add),
- **numerical behavior** (single rounding can reduce error in some cases).

5.3.2 AXPY and dot-product building blocks

```
#include <immintrin.h>

/* y[i] = a*x[i] + y[i] */
void axpy_fma_avx(float* y, const float* x, float a, int n) {
    const __m256 va = _mm256_set1_ps(a);
    int i = 0;
    for (; i + 8 <= n; i += 8) {
        __m256 vx = _mm256_loadu_ps(x + i);
        __m256 vy = _mm256_loadu_ps(y + i);
        vy = _mm256_fmadd_ps(va, vx, vy);
        _mm256_storeu_ps(y + i, vy);
    }
    for (; i < n; ++i) y[i] = a * x[i] + y[i];
}
```

5.3.3 FMA and “semantic compatibility”

Be explicit: **FMA may not produce bitwise-identical results** compared to separate multiply and add, because the fused operation rounds once instead of twice. For numerically sensitive code:

- define an acceptable tolerance policy, or
- enforce strict FP rules when required (at a performance cost).

5.3.4 Dot product with FMA and multiple accumulators

```
#include <immintrin.h>

float dot_fma_avx2(const float* a, const float* b, int n) {
    __m256 s0 = _mm256_setzero_ps();
```

```
    __m256 s1 = _mm256_setzero_ps();

    int i = 0;
    for (; i + 16 <= n; i += 16) {
        __m256 a0 = _mm256_loadu_ps(a + i);
        __m256 b0 = _mm256_loadu_ps(b + i);
        __m256 a1 = _mm256_loadu_ps(a + i + 8);
        __m256 b1 = _mm256_loadu_ps(b + i + 8);

        s0 = _mm256_fmadd_ps(a0, b0, s0);
        s1 = _mm256_fmadd_ps(a1, b1, s1);
    }

    __m256 acc = _mm256_add_ps(s0, s1);

    __m128 lo = _mm256_castps256_ps128(acc);
    __m128 hi = _mm256_extractf128_ps(acc, 1);
    __m128 s  = _mm_add_ps(lo, hi);
    s = _mm_hadd_ps(s, s);
    s = _mm_hadd_ps(s, s);

    float result = _mm_cvtss_f32(s);
    for (; i < n; ++i) result += a[i] * b[i];
    return result;
}
```

5.4 Integer SIMD with AVX2

5.4.1 What AVX2 adds

AVX (original) is mostly about 256-bit vectors for floating-point and some integer moves. **AVX2** extends 256-bit SIMD to **integer arithmetic, shifts, compares, blends, and permutes**

broadly, enabling high-throughput integer vector code without falling back to 128-bit lanes.

5.4.2 Example: 32-bit integer add

```
#include <immintrin.h>

void add_i32_avx2(int* c, const int* a, const int* b, int n) {
    int i = 0;
    for (; i + 8 <= n; i += 8) {
        __m256i va = _mm256_loadu_si256((const __m256i*)(a + i));
        __m256i vb = _mm256_loadu_si256((const __m256i*)(b + i));
        __m256i vc = _mm256_add_epi32(va, vb);
        _mm256_storeu_si256((__m256i*)(c + i), vc);
    }
    for (; i < n; ++i) c[i] = a[i] + b[i];
}
```

5.4.3 Example: byte-wise absolute difference (useful in image/audio)

A common integer SIMD task is per-byte operations.

```
#include <immintrin.h>

/* SAD-like primitive: sum of absolute differences for 8-bit lanes.
   Here: compute abs(a-b) for 32 bytes, store result bytes (not summed). */
void absdiff_u8_avx2(unsigned char* out,
                     const unsigned char* a,
                     const unsigned char* b) {
    __m256i va = _mm256_loadu_si256((const __m256i*)a);
    __m256i vb = _mm256_loadu_si256((const __m256i*)b);

    __m256i d1 = _mm256_subs_epu8(va, vb); /* saturating subtract */
    __m256i d2 = _mm256_subs_epu8(vb, va);
```

```
    __m256i r  = _mm256_or_si256(d1, d2);  /* abs diff for u8 */

    _mm256_storeu_si256((__m256i*)out, r);
}
```

5.4.4 Example: compare + mask extraction for filtering

AVX2 comparisons yield vector masks; you often extract a bitmask to count or to drive compaction logic.

```
#include <immintrin.h>

/* Count elements > t in int32 array */
int count_gt_i32_avx2(const int* x, int n, int t) {
    const __m256i vt = _mm256_set1_epi32(t);
    int cnt = 0;

    int i = 0;
    for (; i + 8 <= n; i += 8) {
        __m256i vx = _mm256_loadu_si256((const __m256i*)(x + i));
        __m256i m  = _mm256_cmpgt_epi32(vx, vt);

        /* movemask extracts sign bits of bytes; use it carefully.
           A reliable trick: reinterpret as bytes and use movemask, then
           ↪ popcount.
           Here we use movemask on 32 bytes: each 32-bit lane all-ones ->
           ↪ bytes all 0xFF. */
        int bits = _mm256_movemask_epi8(m);

#if defined(__GNUG__)  defined(__clang__)
        /* Each int32 lane contributes 4 bytes -> 4 bits set if lane true.
        ↪ */
        cnt += __builtin_popcount((unsigned)bits) / 4;
```

```
#else
        /* Portable popcount (simple) */
        unsigned u = (unsigned)bits;
        u = u - ((u >> 1) & 0x55555555u);
        u = (u & 0x33333333u) + ((u >> 2) & 0x33333333u);
        cnt += (int)((((u + (u >> 4)) & 0x0F0F0F0Fu) * 0x01010101u) >> 24) /
         ↪  4;
#endif
    }

    for (; i < n; ++i) cnt += (x[i] > t);
    return cnt;
}
```

5.5 Transition Costs Between SSE and AVX

5.5.1 The core issue: mixing legacy SSE with AVX in hot code

A well-known performance pitfall is mixing:

- **legacy SSE-encoded** instructions that write XMM registers, and
- **VEX-encoded** AVX instructions that use YMM registers,

in a way that creates penalties on some microarchitectures.
In practice, two safe rules cover most cases:

1. **Do not mix SSE and AVX inside the same tight loop.** Keep a region purely AVX (VEX) or purely SSE.
2. **Before returning to SSE code after AVX, zero the upper halves of YMM.**

5.5.2 The standard fix: `vzeroupper`

`vzeroupper` clears the upper 128 bits of all YMM registers, avoiding transition penalties when executing legacy SSE instructions afterward.

```
#include <immintrin.h>

void avx_region_then_sse_region(float* y, const float* x, int n) {
    /* AVX region */
    const __m256 one = _mm256_set1_ps(1.0f);
    int i = 0;
    for (; i + 8 <= n; i += 8) {
        __m256 v = _mm256_loadu_ps(x + i);
        v = _mm256_add_ps(v, one);
        _mm256_storeu_ps(y + i, v);
    }

    /* Prevent AVX->SSE transition penalty for subsequent legacy SSE code.
     ↪  */
    _mm256_zeroupper();

    /* SSE/scalar tail region (or other SSE-heavy code) */
    for (; i < n; ++i) y[i] = x[i] + 1.0f;
}
```

5.5.3 Assembly form (GAS, Intel syntax)

```
.intel_syntax noprefix
/* ... AVX work ... */
vzeroupper                          /* clear upper halves of ymm regs */
/* ... legacy SSE work may follow safely ... */
.att_syntax prefix
```

5.5.4 A disciplined region policy

For production codebases:

- Compile AVX code so it is consistently VEX-encoded.
- Use clear boundaries between SSE-only and AVX-only regions.
- Insert `vzeroupper` at boundaries that return to legacy SSE code paths.
- Prefer one ISA per translation unit or per hot function when possible.

Practical summary:

- YMM doubles lane count vs XMM, but memory bandwidth and register pressure still bound speed.
- VEX enables non-destructive 3-operand forms; it improves scheduling and reduces moves.
- FMA improves throughput and can change floating-point rounding; define correctness policy.
- AVX2 makes 256-bit integer SIMD broadly practical (arith/shift/compare/permute).
- Mixing SSE and AVX can cost; use `vzeroupper` and keep hot loops ISA-consistent.

Chapter 6

Masking and Control Flow in SIMD

6.1 Branchless Programming Fundamentals

6.1.1 Why branches are a problem for SIMD

SIMD prefers **uniform control flow**: one instruction stream applied to many elements. A per-element `if/else` introduces divergence, and CPUs pay when:

- the branch predictor mispredicts (pipeline flush and lost cycles),
- both paths do meaningful work (wasted execution),
- vectorization is blocked because the compiler cannot prove safety.

The SIMD replacement is **predication by masks**:

- compute a mask from comparisons,
- use bitwise select or blend to choose results without branching,
- keep loop structure straight-line and predictable.

6.1.2 Core identity: select by mask

For float vectors, masks are typically all-ones/all-zeros per lane.

$$\text{select}(m, a, b) = (m \wedge a) \vee (\neg m \wedge b)$$

This is the most important branchless building block.

```
#include <immintrin.h>

/* AVX select: m ? a : b  (m is per-lane all-ones/all-zeros) */
static inline __m256 select_ps(__m256 m, __m256 a, __m256 b) {
    return _mm256_or_ps(_mm256_and_ps(m, a), _mm256_andnot_ps(m, b));
}
```

6.2 Comparison Instructions and Masks

6.2.1 Comparisons produce masks, not booleans

Vector comparisons produce a **mask vector**:

- float compare: `_mm256_cmp_ps` yields FP mask lanes,
- int compare: `_mm256_cmpgt_epi32` yields integer mask lanes (all-bits set/zero).

6.2.2 Example: clamp using compares indirectly (min/max)

A common branchless pattern uses **min/max** instead of explicit compares:

$$\text{clamp}(x, lo, hi) = \min(\max(x, lo), hi)$$

```
#include <immintrin.h>
```

```
void clamp_avx(float* x, int n, float lo, float hi) {
    const __m256 vlo = _mm256_set1_ps(lo);
    const __m256 vhi = _mm256_set1_ps(hi);

    int i = 0;
    for (; i + 8 <= n; i += 8) {
        __m256 v = _mm256_loadu_ps(x + i);
        v = _mm256_max_ps(v, vlo);
        v = _mm256_min_ps(v, vhi);
        _mm256_storeu_ps(x + i, v);
    }
    for (; i < n; ++i) {
        float v = x[i];
        if (v < lo) v = lo;
        if (v > hi) v = hi;
        x[i] = v;
    }
}
```

6.2.3 Example: explicit compare mask for “keep or zero”

```
#include <immintrin.h>

/* out[i] = (x[i] > t) ? x[i] : 0 */
void keep_gt_avx(float* out, const float* x, int n, float t) {
    const __m256 vt = _mm256_set1_ps(t);
    const __m256 vz = _mm256_setzero_ps();

    int i = 0;
    for (; i + 8 <= n; i += 8) {
        __m256 vx = _mm256_loadu_ps(x + i);
        __m256 m  = _mm256_cmp_ps(vx, vt, _CMP_GT_OQ);
        __m256 r  = select_ps(m, vx, vz);
```

```
        _mm256_storeu_ps(out + i, r);
    }
    for (; i < n; ++i) out[i] = (x[i] > t) ? x[i] : 0.0f;
}
```

6.2.4 Mask extraction: turning lane masks into bits

Sometimes you need scalar decisions (counting, early exit, compaction). For floats:

```
#include <immintrin.h>

/* Count lanes where x[i] > t, per 8-float chunk */
static inline int count_gt_lanes(__m256 vx, __m256 vt) {
    __m256 m = _mm256_cmp_ps(vx, vt, _CMP_GT_OQ);
    int bits = _mm256_movemask_ps(m); /* 8-bit mask */
#if defined(__GNUG__)  defined(__clang__)
    return __builtin_popcount((unsigned)bits);
#else
    bits = bits - ((bits >> 1) & 0x55);
    bits = (bits & 0x33) + ((bits >> 2) & 0x33);
    return (((bits + (bits >> 4)) & 0x0F) * 0x01);
#endif
}
```

6.3 Conditional Execution via Blends

6.3.1 Blend is selection, but mind the cost model

Conditional execution in SIMD is almost always **blend/select**. You can implement it using:

- bitwise select (`and/andnot/or`) – universally available,
- blend intrinsics (where available) – can be convenient, not always faster.

6.3.2 Example: piecewise function without branches

Scalar version:

```
float f_scalar(float x) {
    if (x < 0.0f) return -x;
    return x * x;
}
```

Vector version:

```
#include <immintrin.h>

void f_avx(float* out, const float* x, int n) {
    const __m256 zero = _mm256_setzero_ps();

    int i = 0;
    for (; i + 8 <= n; i += 8) {
        __m256 vx = _mm256_loadu_ps(x + i);

        __m256 neg = _mm256_sub_ps(zero, vx);        /* -x */
        __m256 sq  = _mm256_mul_ps(vx, vx);          /* x*x */

        __m256 m   = _mm256_cmp_ps(vx, zero, _CMP_LT_OQ);
        __m256 r   = select_ps(m, neg, sq);

        _mm256_storeu_ps(out + i, r);
    }

    for (; i < n; ++i) {
        float v = x[i];
        out[i] = (v < 0.0f) ? -v : (v * v);
    }
}
```

6.3.3 Example: saturating update (if condition then update else keep)

```
#include <immintrin.h>

/* y[i] = (x[i] > t) ? (y[i] + x[i]) : y[i] */
void add_if_gt_avx(float* y, const float* x, int n, float t) {
    const __m256 vt = _mm256_set1_ps(t);

    int i = 0;
    for (; i + 8 <= n; i += 8) {
        __m256 vx = _mm256_loadu_ps(x + i);
        __m256 vy = _mm256_loadu_ps(y + i);

        __m256 m  = _mm256_cmp_ps(vx, vt, _CMP_GT_OQ);
        __m256 upd = _mm256_add_ps(vy, vx);

        vy = select_ps(m, upd, vy);
        _mm256_storeu_ps(y + i, vy);
    }

    for (; i < n; ++i) if (x[i] > t) y[i] += x[i];
}
```

6.4 Vectorized Loops with Partial Tails

6.4.1 The tail problem

Vector loops handle elements in chunks of width W (4 for SSE float, 8 for AVX float). The remainder $n \bmod W$ is the **tail**. Correct tail handling is non-negotiable.
There are three robust strategies:

1. **Scalar tail:** simplest and often best overall.

2. **Padding:** allocate extra elements and pad with safe values.

3. **Masked tail:** use masked load/store (native in AVX-512; emulated in SSE/AVX).

6.4.2 Strategy 1: scalar tail (recommended default)

```
#include <immintrin.h>

void add_avx_tail_scalar(float* y, const float* x, int n) {
    int i = 0;
    for (; i + 8 <= n; i += 8) {
        __m256 vx = _mm256_loadu_ps(x + i);
        __m256 vy = _mm256_loadu_ps(y + i);
        _mm256_storeu_ps(y + i, _mm256_add_ps(vy, vx));
    }
    for (; i < n; ++i) y[i] += x[i];
}
```

6.4.3 Strategy 2: padding (fast when you control allocation)

```
static inline int round_up8(int n) { return (n + 7) & ~7; }

/* Requires x and y allocated at least round_up8(n) and padded safely. */
```

6.4.4 Strategy 3: masked tail for AVX (emulation)

AVX (without AVX-512 masks) can emulate masked tail by building a mask vector and using:

- safe loads from a temporary buffer, or
- load the full vector only when it does not read past allocation.

The safest general approach uses a small stack buffer.

```
#include <immintrin.h>
#include <cstring>

/* Safest masked tail: copy remaining elements into temp, operate, copy
 ↪  back. */
void add_avx_masked_tail(float* y, const float* x, int n) {
    int i = 0;
    for (; i + 8 <= n; i += 8) {
        __m256 vx = _mm256_loadu_ps(x + i);
        __m256 vy = _mm256_loadu_ps(y + i);
        _mm256_storeu_ps(y + i, _mm256_add_ps(vy, vx));
    }

    int r = n - i;
    if (r > 0) {
        alignas(32) float tx[8] = {0};
        alignas(32) float ty[8] = {0};

        std::memcpy(tx, x + i, (size_t)r * sizeof(float));
        std::memcpy(ty, y + i, (size_t)r * sizeof(float));

        __m256 vx = _mm256_load_ps(tx);
        __m256 vy = _mm256_load_ps(ty);
        __m256 vr = _mm256_add_ps(vy, vx);
        _mm256_store_ps(ty, vr);

        std::memcpy(y + i, ty, (size_t)r * sizeof(float));
    }
}
```

6.5 Eliminating Scalar Branches

6.5.1 Replace per-element branches with vector masks

The general transformation:

- compute mask from condition,
- compute both candidate values (or compute one + keep old),
- select with mask.

6.5.2 Example: threshold + update (branchy to branchless)

Branchy scalar:

```
void relu_scalar(float* y, const float* x, int n) {
    for (int i = 0; i < n; ++i) y[i] = (x[i] > 0.0f) ? x[i] : 0.0f;
}
```

Branchless SIMD:

```
#include <immintrin.h>

/* ReLU: max(x, 0) */
void relu_avx(float* y, const float* x, int n) {
    const __m256 zero = _mm256_setzero_ps();
    int i = 0;
    for (; i + 8 <= n; i += 8) {
        __m256 vx = _mm256_loadu_ps(x + i);
        __m256 r  = _mm256_max_ps(vx, zero);
        _mm256_storeu_ps(y + i, r);
    }
    for (; i < n; ++i) y[i] = (x[i] > 0.0f) ? x[i] : 0.0f;
}
```

6.5.3 Example: filtering without branches (count + mark)

A common pattern is to produce a mask and either:

- count matches (movemask + popcount),
- write a marker array (0/1),
- or later compact using a separate pass.

```
#include <immintrin.h>

/* mark[i] = (x[i] >= t) ? 1 : 0  (int32 markers) */
void mark_ge_avx2(int* mark, const float* x, int n, float t) {
    const __m256 vt = _mm256_set1_ps(t);
    const __m256 one = _mm256_set1_ps(1.0f);
    const __m256 zero = _mm256_setzero_ps();

    int i = 0;
    for (; i + 8 <= n; i += 8) {
        __m256 vx = _mm256_loadu_ps(x + i);
        __m256 m  = _mm256_cmp_ps(vx, vt, _CMP_GE_OQ);

        /* Produce 1.0f or 0.0f then convert to int32. */
        __m256 f  = select_ps(m, one, zero);
        __m256i mi = _mm256_cvttps_epi32(f);

        _mm256_storeu_si256((__m256i*)(mark + i), mi);
    }

    for (; i < n; ++i) mark[i] = (x[i] >= t) ? 1 : 0;
}
```

6.5.4 Assembly intuition (GAS, Intel syntax)

Mask-based control flow is typically “compare + bitwise select” or “compare + min/max”.

```
.intel_syntax noprefix
/* compare -> mask bits, then AND/ANDN/OR selection (conceptual) */
vmovups ymm0, ymmword ptr [rdi]         /* x */
vcmpps  ymm2, ymm0, ymm1, 14            /* /* example predicate id */
 ↪  */
vandps  ymm3, ymm2, ymm0                /* x & mask */
vandnps ymm2, ymm2, ymm4                /* (~mask) & alt */
vorps   ymm3, ymm3, ymm2                /* select */
vmovups ymmword ptr [rsi], ymm3
.att_syntax prefix
```

Practical summary:

- SIMD control flow is **masking** and **selection**, not branches.
- Comparisons produce masks; masks feed select/blend or min/max.
- Tail handling must be correct; scalar tail is the default robust choice.
- Eliminate per-element branching by computing candidates and selecting via masks.
- Measure: branchless can be faster when mispredicts are common, but can be slower if it forces expensive extra work.

Chapter 7

Practical SIMD Loop Patterns

7.1 Vectorized Reduction (Sum, Min, Max)

7.1.1 Reduction fundamentals

A reduction collapses many elements to one value. The main performance issue is the **dependency chain**. To improve throughput:

- use **multiple accumulators** (ILP),
- reduce vectors horizontally only once per chunk,
- keep the loop unit-stride and simple.

7.1.2 Sum reduction with AVX and two accumulators

```
#include <immintrin.h>

static inline float hsum256_ps(__m256 v) {
```

```
    __m128 lo = _mm256_castps256_ps128(v);
    __m128 hi = _mm256_extractf128_ps(v, 1);
    __m128 s  = _mm_add_ps(lo, hi);
    s = _mm_hadd_ps(s, s);
    s = _mm_hadd_ps(s, s);
    return _mm_cvtss_f32(s);
}

float sum_avx_2acc(const float* x, int n) {
    __m256 a0 = _mm256_setzero_ps();
    __m256 a1 = _mm256_setzero_ps();

    int i = 0;
    for (; i + 16 <= n; i += 16) {
        __m256 v0 = _mm256_loadu_ps(x + i);
        __m256 v1 = _mm256_loadu_ps(x + i + 8);
        a0 = _mm256_add_ps(a0, v0);
        a1 = _mm256_add_ps(a1, v1);
    }

    float result = hsum256_ps(_mm256_add_ps(a0, a1));
    for (; i < n; ++i) result += x[i];
    return result;
}
```

7.1.3 Min/Max reduction with AVX

```
#include <immintrin.h>

static inline float hmin256_ps(__m256 v) {
    __m128 lo = _mm256_castps256_ps128(v);
    __m128 hi = _mm256_extractf128_ps(v, 1);
    __m128 m  = _mm_min_ps(lo, hi);
```

```
    m = _mm_min_ps(m, _mm_movehl_ps(m, m));
    m = _mm_min_ss(m, _mm_shuffle_ps(m, m, _MM_SHUFFLE(1,1,1,1)));
    return _mm_cvtss_f32(m);
}

static inline float hmax256_ps(__m256 v) {
    __m128 lo = _mm256_castps256_ps128(v);
    __m128 hi = _mm256_extractf128_ps(v, 1);
    __m128 m  = _mm_max_ps(lo, hi);
    m = _mm_max_ps(m, _mm_movehl_ps(m, m));
    m = _mm_max_ss(m, _mm_shuffle_ps(m, m, _MM_SHUFFLE(1,1,1,1)));
    return _mm_cvtss_f32(m);
}

float min_avx(const float* x, int n) {
    if (n <= 0) return 0.0f;

    int i = 0;
    __m256 vmin = _mm256_loadu_ps(x);
    i = 8;

    for (; i + 8 <= n; i += 8) {
        __m256 v = _mm256_loadu_ps(x + i);
        vmin = _mm256_min_ps(vmin, v);
    }

    float result = hmin256_ps(vmin);
    for (; i < n; ++i) if (x[i] < result) result = x[i];
    return result;
}

float max_avx(const float* x, int n) {
    if (n <= 0) return 0.0f;
```

```
    int i = 0;
    __m256 vmax = _mm256_loadu_ps(x);
    i = 8;

    for (; i + 8 <= n; i += 8) {
        __m256 v = _mm256_loadu_ps(x + i);
        vmax = _mm256_max_ps(vmax, v);
    }

    float result = hmax256_ps(vmax);
    for (; i < n; ++i) if (x[i] > result) result = x[i];
    return result;
}
```

7.1.4 Reduction correctness note

FP reductions can change rounding due to reassociation. Define whether you require:

- bitwise-identical results (strict, slower), or
- tolerance-based equality (typical for numeric code).

7.2 Vectorized Search and Filtering

7.2.1 Search: find the first element matching a predicate

Approach:

- compare a vector chunk,
- convert compare result to a bitmask,

- if mask non-zero, locate first set bit and return index.

```
#include <immintrin.h>
#include <cstdint>

static inline int ctz32(unsigned x) {
#if defined(__GNUG__)  defined(__clang__)
    return __builtin_ctz(x);
#else
    int n = 0;
    while ((x & 1u) == 0u) { x >>= 1; ++n; }
    return n;
#endif
}

/* Find first x[i] == key, return index or -1 */
int find_eq_f32_avx(const float* x, int n, float key) {
    const __m256 vk = _mm256_set1_ps(key);

    int i = 0;
    for (; i + 8 <= n; i += 8) {
        __m256 vx = _mm256_loadu_ps(x + i);
        __m256 m  = _mm256_cmp_ps(vx, vk, _CMP_EQ_OQ);
        unsigned bits = (unsigned)_mm256_movemask_ps(m);
        if (bits) {
            int lane = ctz32(bits); /* 0..7 */
            return i + lane;
        }
    }
    for (; i < n; ++i) if (x[i] == key) return i;
    return -1;
}
```

7.2.2 Filtering: count matches fast (predicate density unknown)

If you only need counts, do not compact in the hot loop. Count masks.

```
#include <immintrin.h>

int count_gt_f32_avx(const float* x, int n, float t) {
    const __m256 vt = _mm256_set1_ps(t);
    int cnt = 0;

    int i = 0;
    for (; i + 8 <= n; i += 8) {
        __m256 vx = _mm256_loadu_ps(x + i);
        __m256 m  = _mm256_cmp_ps(vx, vt, _CMP_GT_OQ);
        int bits  = _mm256_movemask_ps(m);
#if defined(__GNUG__)  defined(__clang__)
        cnt += __builtin_popcount((unsigned)bits);
#else
        bits = bits - ((bits >> 1) & 0x55);
        bits = (bits & 0x33) + ((bits >> 2) & 0x33);
        cnt += (((bits + (bits >> 4)) & 0x0F) * 0x01);
#endif
    }

    for (; i < n; ++i) cnt += (x[i] > t);
    return cnt;
}
```

7.2.3 Filtering strategy note

- **Count-only** is cheap: compare + movemask + popcount.
- **Compaction** (writing matching elements contiguously) is harder and typically needs a second pass or specialized techniques; treat it as a separate pattern with clear costs.

7.3 Vectorized Copy and Transform Loops

7.3.1 Copy and scale: the bread-and-butter kernels

These kernels often become **memory-bandwidth bound**. The goal is:

- minimize extra instructions,
- keep unit-stride loads/stores,
- avoid unnecessary shuffles.

7.3.2 Vectorized copy (float) with AVX

```
#include <immintrin.h>

void copy_f32_avx(float* dst, const float* src, int n) {
    int i = 0;
    for (; i + 8 <= n; i += 8) {
        __m256 v = _mm256_loadu_ps(src + i);
        _mm256_storeu_ps(dst + i, v);
    }
    for (; i < n; ++i) dst[i] = src[i];
}
```

7.3.3 Transform: y = a*x + b (FMA when available)

```
#include <immintrin.h>

void linear_fma_avx(float* y, const float* x, int n, float a, float b) {
    const __m256 va = _mm256_set1_ps(a);
    const __m256 vb = _mm256_set1_ps(b);
```

```
    int i = 0;
    for (; i + 8 <= n; i += 8) {
        __m256 vx = _mm256_loadu_ps(x + i);
        __m256 vy = _mm256_fmadd_ps(va, vx, vb); /* a*x + b */
        _mm256_storeu_ps(y + i, vy);
    }
    for (; i < n; ++i) y[i] = a * x[i] + b;
}
```

7.3.4 Byte transforms with AVX2 (integer SIMD)

Example: saturating clamp to `[0, 255]` is natural for bytes; many image/audio kernels use this style.

```
#include <immintrin.h>

/* out[i] = min(in[i] + add, 255) for uint8 */
void add_sat_u8_avx2(unsigned char* out, const unsigned char* in, int n,
↪   unsigned char add) {
    const __m256i vadd = _mm256_set1_epi8((char)add);

    int i = 0;
    for (; i + 32 <= n; i += 32) {
        __m256i v = _mm256_loadu_si256((const __m256i*)(in + i));
        v = _mm256_adds_epu8(v, vadd); /* unsigned saturating add */
        _mm256_storeu_si256((__m256i*)(out + i), v);
    }
    for (; i < n; ++i) {
        unsigned t = (unsigned)in[i] + (unsigned)add;
        out[i] = (unsigned char)(t > 255u ? 255u : t);
    }
}
```

7.4 Horizontal vs Vertical Computation

7.4.1 Definitions

- **Vertical computation:** do the same operation independently per lane (map-like). This is the natural SIMD case.
- **Horizontal computation:** mix lanes within a vector (reductions, dot-products, horizontal sums/min/max).

Vertical is typically cheap (vector ALU ops). Horizontal requires shuffles/permutations and is often the bottleneck.

7.4.2 Example: vertical vs horizontal

Vertical (element-wise multiply):

```
#include <immintrin.h>
static inline __m256 mul8(__m256 a, __m256 b) { return _mm256_mul_ps(a, b);
↪  }
```

Horizontal (sum all lanes) needs lane movement:

```
#include <immintrin.h>
static inline float hsum256_ps(__m256 v) {
    __m128 lo = _mm256_castps256_ps128(v);
    __m128 hi = _mm256_extractf128_ps(v, 1);
    __m128 s  = _mm_add_ps(lo, hi);
    s = _mm_hadd_ps(s, s);
    s = _mm_hadd_ps(s, s);
    return _mm_cvtss_f32(s);
}
```

7.4.3 Design rule

Keep the hot loop mostly vertical. Push horizontal work to:

- the end of the loop,
- one reduction step per chunk,
- and use multiple accumulators to reduce horizontal frequency.

7.5 Loop Unrolling with SIMD

7.5.1 Why unroll SIMD loops

Unrolling increases **ILP** and helps hide:

- instruction latency (especially for dependent reductions),
- load latency (when data is in cache but still not free),
- throughput limits by feeding multiple independent operations.

7.5.2 Unroll patterns

- **Unroll-by-2** with two accumulators is a common sweet spot.
- **Unroll-by-4** can help reductions but may increase register pressure.

7.5.3 Example: unrolled map kernel (two independent vectors per iteration)

```
#include <immintrin.h>
```

```
/* y[i] = x[i] * c  (unroll by 2) */
void scale_avx_unroll2(float* y, const float* x, float c, int n) {
    const __m256 vc = _mm256_set1_ps(c);

    int i = 0;
    for (; i + 16 <= n; i += 16) {
        __m256 v0 = _mm256_loadu_ps(x + i);
        __m256 v1 = _mm256_loadu_ps(x + i + 8);

        v0 = _mm256_mul_ps(v0, vc);
        v1 = _mm256_mul_ps(v1, vc);

        _mm256_storeu_ps(y + i, v0);
        _mm256_storeu_ps(y + i + 8, v1);
    }

    for (; i + 8 <= n; i += 8) {
        __m256 v = _mm256_loadu_ps(x + i);
        v = _mm256_mul_ps(v, vc);
        _mm256_storeu_ps(y + i, v);
    }

    for (; i < n; ++i) y[i] = x[i] * c;
}
```

7.5.4 Example: unrolled reduction (four accumulators)

```
#include <immintrin.h>

static inline float hsum256_ps(__m256 v) {
    __m128 lo = _mm256_castps256_ps128(v);
    __m128 hi = _mm256_extractf128_ps(v, 1);
```

```
    __m128 s  = _mm_add_ps(lo, hi);
    s = _mm_hadd_ps(s, s);
    s = _mm_hadd_ps(s, s);
    return _mm_cvtss_f32(s);
}

float sum_avx_4acc(const float* x, int n) {
    __m256 a0 = _mm256_setzero_ps();
    __m256 a1 = _mm256_setzero_ps();
    __m256 a2 = _mm256_setzero_ps();
    __m256 a3 = _mm256_setzero_ps();

    int i = 0;
    for (; i + 32 <= n; i += 32) {
        a0 = _mm256_add_ps(a0, _mm256_loadu_ps(x + i +  0));
        a1 = _mm256_add_ps(a1, _mm256_loadu_ps(x + i +  8));
        a2 = _mm256_add_ps(a2, _mm256_loadu_ps(x + i + 16));
        a3 = _mm256_add_ps(a3, _mm256_loadu_ps(x + i + 24));
    }

    __m256 acc = _mm256_add_ps(_mm256_add_ps(a0, a1), _mm256_add_ps(a2,
    ↪  a3));
    float result = hsum256_ps(acc);
    for (; i < n; ++i) result += x[i];
    return result;
}
```

7.5.5 Unrolling limits

Stop unrolling when:

- register spills appear,
- code size harms instruction cache,

- the loop becomes memory-bandwidth bound and extra ILP no longer helps.

7.5.6 Assembly intuition (GAS, Intel syntax)

Two independent vector streams allow OoO overlap (not a full function).

```
.intel_syntax noprefix
/* unroll-by-2 idea: two loads, two computes, two stores */
vmovups ymm0, ymmword ptr [rdi]          /* stream 0 */
vmovups ymm1, ymmword ptr [rdi + 32]     /* stream 1 */
vmulps  ymm0, ymm0, ymm2
vmulps  ymm1, ymm1, ymm2
vmovups ymmword ptr [rsi], ymm0
vmovups ymmword ptr [rsi + 32], ymm1
.att_syntax prefix
```

Practical summary:

- Reductions need multiple accumulators; minimize horizontal work frequency.
- Searches and filters use compare + movemask; compaction is a separate, heavier problem.
- Copy/transform loops are often bandwidth-bound; keep them straight-line and unit-stride.
- Vertical computation is cheap; horizontal computation costs shuffles.
- Unroll until you hit register pressure or instruction-cache limits, then stop.

Chapter 8

SIMD and Memory Bandwidth

8.1 Load/Store Bottlenecks

8.1.1 The bandwidth reality

Many SIMD kernels become **memory-bound**: performance is limited by moving bytes, not by arithmetic. Typical symptoms:

- adding more math does not change runtime (until you add a lot),
- throughput scales with cache level (L1 fast, L2 slower, DRAM much slower),
- the loop approaches a stable "GB/s" ceiling independent of SIMD width.

8.1.2 Roofline-style reasoning (practical)

Define arithmetic intensity:

$$I = \frac{\text{useful ops}}{\text{bytes moved}}$$

If I is low, the kernel is likely memory-bound. SIMD can still help by reducing instruction overhead, but it cannot exceed the memory system's sustained bandwidth.

8.1.3 Example: SAXPY is often bandwidth-bound for large arrays

```
#include <immintrin.h>

/* y[i] = a*x[i] + y[i]  : 1 load x, 1 load y, 1 store y per element */
void saxpy_avx(float* y, const float* x, float a, int n) {
    const __m256 va = _mm256_set1_ps(a);
    int i = 0;
    for (; i + 8 <= n; i += 8) {
        __m256 vx = _mm256_loadu_ps(x + i);
        __m256 vy = _mm256_loadu_ps(y + i);
        vy = _mm256_fmadd_ps(va, vx, vy);
        _mm256_storeu_ps(y + i, vy);
    }
    for (; i < n; ++i) y[i] = a * x[i] + y[i];
}
```

8.1.4 Practical bottleneck checklist

A SIMD loop is often load/store limited when:

- it performs 1–2 arithmetic ops per element,
- it streams large arrays (working set exceeds caches),
- it has unit-stride access (good) but still saturates bandwidth.

In such cases, focus on:

- avoiding extra passes over memory,

- reducing stores and store traffic,
- choosing appropriate store policy (temporal vs non-temporal),
- correct measurement (see final section).

8.2 Streaming Loads and Stores

8.2.1 Temporal vs non-temporal (streaming) stores

A **temporal** store goes through caches normally. A **non-temporal** (streaming) store aims to avoid polluting caches for large write-only outputs. Use streaming stores when:

- you write a large output once,
- the output will not be read soon,
- you want to preserve cache for other hot data.

8.2.2 AVX non-temporal stores

AVX provides streaming store intrinsics (example for floats):

```
#include <immintrin.h>

/* Write-only destination: prefer non-temporal stores when dst is not
 ↪  reused soon. */
void copy_stream_avx(float* dst, const float* src, int n) {
    int i = 0;
    for (; i + 8 <= n; i += 8) {
        __m256 v = _mm256_loadu_ps(src + i);      /* normal load */
        _mm256_stream_ps(dst + i, v);            /* non-temporal store
         ↪  (requires 32B alignment for best behavior) */
```

```
    }
    for (; i < n; ++i) dst[i] = src[i];

    /* Ensure streaming stores are globally visible before subsequent
     ↪  dependent work. */
    _mm_sfence();
}
```

8.2.3 Streaming loads

On x86, streaming loads are less commonly needed than streaming stores. Hardware prefetchers are effective for unit-stride streams. Streaming-load hints exist (prefetch hints), but correctness does not depend on them; treat them as performance hints only.

8.2.4 When streaming stores hurt

Avoid non-temporal stores when:

- the written data will be read soon (it would cause cache misses),
- the store size is small (overhead outweighs benefits),
- your destination is not aligned/structured for efficient streaming,
- you interleave streaming stores with many loads that already saturate memory.

8.3 Prefetching Strategies

8.3.1 Hardware prefetch usually wins for unit stride

For simple streaming loops with unit stride, modern hardware prefetchers usually provide near-optimal behavior. Manual prefetch can help when:

- the access pattern is regular but not perfectly unit stride,
- the working set is large and misses are frequent,
- the loop has enough independent work to overlap latency.

8.3.2 Software prefetch basics

- Prefetch is a **hint**; it does not change correctness.
- Prefetch too close → useless.
- Prefetch too far → cache pollution.
- Prefetch distance must be tuned per CPU and per loop.

8.3.3 Example: prefetch ahead in a streaming transform

```
#include <immintrin.h>

void add_prefetch_avx(float* y, const float* x, float c, int n) {
    const __m256 vc = _mm256_set1_ps(c);

    int i = 0;
    const int PFD = 256; /* bytes ahead: starting point, must be tuned */

    for (; i + 8 <= n; i += 8) {
        const char* pf = (const char*)(x + i) + PFD;
        _mm_prefetch(pf, _MM_HINT_T0);

        __m256 vx = _mm256_loadu_ps(x + i);
        __m256 vy = _mm256_add_ps(vx, vc);
        _mm256_storeu_ps(y + i, vy);
    }
```

```
    for (; i < n; ++i) y[i] = x[i] + c;
}
```

8.3.4 Prefetch for two-stream kernels

Two-input kernels (e.g., add, dot, saxpy) may benefit from prefetching both streams if misses dominate:

```
#include <immintrin.h>

float dot_prefetch_fma(const float* a, const float* b, int n) {
    __m256 s0 = _mm256_setzero_ps();
    __m256 s1 = _mm256_setzero_ps();

    int i = 0;
    const int PFD = 256;

    for (; i + 16 <= n; i += 16) {
        _mm_prefetch((const char*)(a + i) + PFD, _MM_HINT_T0);
        _mm_prefetch((const char*)(b + i) + PFD, _MM_HINT_T0);

        __m256 a0 = _mm256_loadu_ps(a + i);
        __m256 b0 = _mm256_loadu_ps(b + i);
        __m256 a1 = _mm256_loadu_ps(a + i + 8);
        __m256 b1 = _mm256_loadu_ps(b + i + 8);

        s0 = _mm256_fmadd_ps(a0, b0, s0);
        s1 = _mm256_fmadd_ps(a1, b1, s1);
    }

    __m256 acc = _mm256_add_ps(s0, s1);
    __m128 lo = _mm256_castps256_ps128(acc);
```

```
    __m128 hi = _mm256_extractf128_ps(acc, 1);
    __m128 s  = _mm_add_ps(lo, hi);
    s = _mm_hadd_ps(s, s);
    s = _mm_hadd_ps(s, s);

    float result = _mm_cvtss_f32(s);
    for (; i < n; ++i) result += a[i] * b[i];
    return result;
}
```

8.4 Write Combining and Store Forwarding

8.4.1 Write combining (practical meaning)

Write combining is a mechanism where multiple adjacent stores are accumulated and written as larger bursts. You benefit when:

- you write sequentially,
- stores are naturally aligned and contiguous,
- you avoid mixing many small scattered stores.

Non-temporal stores often rely on write-combining behavior to achieve high bandwidth.

8.4.2 Store forwarding (practical meaning)

Store forwarding is the CPU's ability to satisfy a load from a recent store in the store buffer without waiting for it to commit to cache. Problems arise when:

- store size and subsequent load size mismatch,
- addresses are misaligned or cross boundaries,

- you do partial writes then read a larger chunk (read-modify-write patterns).

8.4.3 Avoid store-forwarding stalls: do full-width stores

Prefer full-width stores matching your later loads. Avoid patterns like "write 4 bytes then load 16/32 bytes from the same region immediately" in tight loops.

```
/* Bad idea (conceptual): partial stores then wide loads from same region
↪  in hot loop.
   Better: compute into a vector and store full width once. */
```

8.4.4 Example: structured full-width store

```
#include <immintrin.h>

/* Compute then store full vector width: friendlier for store buffers and
↪  forwarding. */
void square_store_avx(float* y, const float* x, int n) {
    int i = 0;
    for (; i + 8 <= n; i += 8) {
        __m256 v = _mm256_loadu_ps(x + i);
        v = _mm256_mul_ps(v, v);
        _mm256_storeu_ps(y + i, v);
    }
    for (; i < n; ++i) y[i] = x[i] * x[i];
}
```

8.5 Measuring Memory-Bound SIMD Code

8.5.1 Measure bandwidth, not just time

For memory-bound kernels, report:

- **bytes processed** per iteration,
- **time**, and
- derived **GB/s**.

8.5.2 Avoid common benchmark traps

- warming: first iteration includes cold-cache misses,
- dead-code elimination: compiler removes unused work,
- tiny sizes: fits in cache and does not represent DRAM behavior,
- allocator noise: measuring allocations and page faults.

8.5.3 A minimal, robust bandwidth benchmark harness

```
#include <chrono>
#include <cstdint>

static inline std::uint64_t now_ns() {
    return
    ↪  (std::uint64_t)std::chrono::duration_cast<std::chrono::nanoseconds>(
        std::chrono::high_resolution_clock::now().time_since_epoch()
    ).count();
}

template <class F>
double bench_gbps(F&& f, std::size_t bytes_per_iter, int iters) {
    /* Warm-up */
    for (int i = 0; i < 10; ++i) f();

    std::uint64_t t0 = now_ns();
```

```
    for (int i = 0; i < iters; ++i) f();
    std::uint64_t t1 = now_ns();

    double sec = (double)(t1 - t0) * 1e-9;
    double gb  = (double)bytes_per_iter * (double)iters / 1e9;
    return gb / sec;
}
```

8.5.4 Example: measuring a copy kernel

Copy reads `n*sizeof(T)` and writes `n*sizeof(T)`. So bytes per iteration $\approx 2 \times n \times sizeof(T)$ (ignoring overhead).

```
#include <immintrin.h>
#include <cstddef>

void copy_avx(float* dst, const float* src, int n) {
    int i = 0;
    for (; i + 8 <= n; i += 8) {
        __m256 v = _mm256_loadu_ps(src + i);
        _mm256_storeu_ps(dst + i, v);
    }
    for (; i < n; ++i) dst[i] = src[i];
}

double measure_copy_gbps(float* dst, const float* src, int n, int iters) {
    std::size_t bytes = (std::size_t)2 * (std::size_t)n * sizeof(float);
    volatile float sink = 0.0f; /* prevent DCE in simplistic setups */

    auto fn = [&]() {
        copy_avx(dst, src, n);
        sink += dst[0];
    };
```

```
    return bench_gbps(fn, bytes, iters);
}
```

8.5.5 Assembly intuition: a bandwidth kernel is mostly moves

```
.intel_syntax noprefix
/* copy-like loop core: load + store dominate */
vmovups ymm0, ymmword ptr [rsi]      /* load 32B */
vmovups ymmword ptr [rdi], ymm0      /* store 32B */
.att_syntax prefix
```

Practical summary:

- Many SIMD loops are limited by load/store bandwidth; do not expect linear speedup with wider vectors.
- Streaming (non-temporal) stores can help for large write-only outputs, but hurt if data is reused soon.
- Prefetch is a hint: start with hardware prefetch, then tune cautiously only when misses dominate.
- Write combining rewards contiguous aligned stores; avoid partial stores + wide loads patterns that trigger forwarding stalls.
- Measure memory-bound code in GB/s with warm-up, large sizes, and DCE-resistant harnesses.

Chapter 9

Compiler Vectorization vs Manual SIMD

9.1 Auto-Vectorization Capabilities

9.1.1 What modern compilers can do well

Current optimizing compilers can auto-vectorize many loops when they can prove:

- **no harmful aliasing** between pointers (or that aliasing is irrelevant),
- **unit-stride** or predictable memory access,
- **no loop-carried dependencies** that prevent reordering,
- **a vector-friendly control flow** (or predicate/mask conversion is possible),
- **safe bounds** for loads/stores (tail handling is correct).

They also commonly perform:

- loop unrolling,

- strength reduction,
- constant hoisting,
- partial vectorization (vector main loop + scalar remainder),
- vector reductions with multiple accumulators in favorable cases.

9.1.2 Baseline example: a vectorizable transform loop

```
void linear_scalar(float* y, const float* x, int n, float a, float b) {
    for (int i = 0; i < n; ++i) y[i] = a * x[i] + b;
}
```

When vectorized, the compiler typically generates:

- vector loads from `x`,
- vector multiply/add (or FMA),
- vector stores to `y`,
- scalar tail.

9.1.3 Make the compiler's job easier: noalias + alignment contracts

Even without nonstandard keywords, you can improve vectorization by writing code that clearly separates inputs/outputs.

```
/* Friendly: separate read-only input and distinct output. */
void linear_out(float* y, const float* x, int n, float a, float b) {
    for (int i = 0; i < n; ++i) y[i] = a * x[i] + b;
}
```

9.2 When Compilers Fail to Vectorize

Auto-vectorization commonly fails (or produces weak code) due to:

9.2.1 1) Pointer aliasing uncertainty

If the compiler cannot prove that `x` and `y` do not overlap, it may avoid vectorization or generate conservative code.

```
/* Potential alias: y may overlap x, especially if caller passes same
↪  pointer. */
void add_inplace(float* y, const float* x, int n) {
    for (int i = 0; i < n; ++i) y[i] += x[i];
}
```

If aliasing must be forbidden by design, reflect it in API rules and enforce with tests or higher-level wrappers.

9.2.2 2) Non-unit stride / irregular access

```
/* Stride breaks vector-friendly contiguous loads. */
void strided(float* y, const float* x, int n, int k) {
    for (int i = 0; i < n; i += k) y[i] = x[i] * 2.0f;
}
```

9.2.3 3) Loop-carried dependencies

```
/* Dependency: y[i] depends on y[i-1] */
void prefix_like(float* y, const float* x, int n) {
    if (n <= 0) return;
    y[0] = x[0];
    for (int i = 1; i < n; ++i) y[i] = y[i-1] + x[i];
}
```

9.2.4 4) Hard-to-predicate branches

```
/* Divergent control flow may block vectorization */
void branchy(float* y, const float* x, int n, float t) {
    for (int i = 0; i < n; ++i) {
        if (x[i] > t) y[i] = x[i] * 2.0f;
        else          y[i] = x[i] + 1.0f;
    }
}
```

Compilers can sometimes transform this into compare+select, but not always (especially if the branches contain complex work or function calls).

9.2.5 5) Function calls inside the loop

Calls are barriers unless inlined and proven pure. Example:

```
float g(float x);

void with_call(float* y, const float* x, int n) {
    for (int i = 0; i < n; ++i) y[i] = g(x[i]); /* often blocks
    ↪  vectorization */
}
```

9.3 Pragmas, Hints, and Assumptions

9.3.1 General principle

Hints should be used to express **truths you can guarantee**. If a hint is wrong, behavior can become incorrect or undefined depending on the mechanism.

9.3.2 Common categories of hints

- **Assume alignment:** tell the compiler a pointer is aligned.
- **Assume noalias:** tell the compiler pointers do not overlap.
- **Vectorization enablement:** request vectorization/unrolling.
- **Reduction recognition:** help the compiler see safe reassociation.

9.3.3 Portable alignment declaration with C++

Prefer expressing alignment through allocation and types:

```
#include <cstddef>
#include <new>

struct alignas(32) Aligned32F32 { float v; }; /* type-level alignment
↪  example */
```

When you need to convey runtime alignment to the compiler, use compiler-specific builtins *only when you can guarantee it*:

```
float* assume_aligned_32(float* p) {
#if defined(__clang__)  defined(__GNUG__)
    return (float*)__builtin_assume_aligned(p, 32);
#else
    return p;
#endif
}
```

9.3.4 OpenMP SIMD pragma (widely supported idea)

If you use OpenMP, `#pragma omp simd` can encourage vectorization for simple loops, provided dependencies are valid.

```
void scale_omp_simd(float* y, const float* x, int n, float c) {
#pragma omp simd
    for (int i = 0; i < n; ++i) y[i] = x[i] * c;
}
```

9.3.5 “Assume” rules of thumb

- Only assume alignment if you control allocation and can prove it.
- Only assume non-alias if the API contract forbids overlap.
- Avoid piling hints on unclear code: rewrite loop shape first.

9.4 Manual Intrinsics vs Compiler Output

9.4.1 When intrinsics are justified

Manual SIMD via intrinsics is usually justified when:

- the loop is performance-critical and stable,
- auto-vectorization fails or produces suboptimal code,
- you need specific instructions (FMA, shuffles, permutes) in a known pattern,
- you want predictable ISA usage and a controlled hot path.

9.4.2 The cost of intrinsics

- higher maintenance burden,
- more complicated debugging,

- portability and dispatch complexity,
- risk of over-optimizing for one microarchitecture.

9.4.3 Side-by-side example: branchless clamp (compiler vs intrinsics)

Scalar loop many compilers can auto-vectorize:

```
void clamp_scalar(float* x, int n, float lo, float hi) {
    for (int i = 0; i < n; ++i) {
        float v = x[i];
        if (v < lo) v = lo;
        if (v > hi) v = hi;
        x[i] = v;
    }
}
```

Manual intrinsics guarantee a vector min/max pattern:

```
#include <immintrin.h>

void clamp_avx(float* x, int n, float lo, float hi) {
    const __m256 vlo = _mm256_set1_ps(lo);
    const __m256 vhi = _mm256_set1_ps(hi);

    int i = 0;
    for (; i + 8 <= n; i += 8) {
        __m256 v = _mm256_loadu_ps(x + i);
        v = _mm256_max_ps(v, vlo);
        v = _mm256_min_ps(v, vhi);
        _mm256_storeu_ps(x + i, v);
    }
    for (; i < n; ++i) {
        float v = x[i];
```

```
        if (v < lo) v = lo;
        if (v > hi) v = hi;
        x[i] = v;
    }
}
```

9.4.4 Correctness and FP semantics

If the compiler uses FMA or reassociation, results may differ in last bits. Manual intrinsics also can change semantics (FMA). Define correctness policy for numeric kernels.

9.5 Reading Compiler-Generated Assembly

9.5.1 What to look for (a deterministic checklist)

When you inspect assembly:

- Identify the **vector width**: XMM (128) vs YMM (256).
- Identify the **main loop**: repeated `vmovups/vmovdqu` + arithmetic + store.
- Look for **tail handling**: a scalar loop after vector loop, or a remainder block.
- Check for **shuffles/permutations**: many shuffles often indicates layout mismatch or a difficult reduction.
- Check for **unrolling**: multiple loads/computes per iteration.
- Check for **transition hygiene**: `vzeroupper` before returning to legacy SSE regions.

9.5.2 Example: expected AVX pattern in assembly (conceptual)

A simple vectorized transform resembles:

- load YMM from `[x+i]`,
- multiply/add (or FMA),
- store to `[y+i]`,
- increment pointer/index by 32 bytes.

```
.intel_syntax noprefix
/* conceptual inner loop for y = a*x + b (AVX) */
vmovups ymm0, ymmword ptr [rsi]      /* load x */
vmulps  ymm0, ymm0, ymm2             /* x * a  (or vfmadd* with b) */
vaddps  ymm0, ymm0, ymm3             /* + b */
vmovups ymmword ptr [rdi], ymm0      /* store y */
add     rsi, 32
add     rdi, 32
/* loop ... */
.att_syntax prefix
```

9.5.3 Example: recognizing reduction structure

A good vector reduction typically shows:

- multiple vector accumulators updated independently,
- a final horizontal collapse at the end,
- a scalar tail add.

```
.intel_syntax noprefix
/* conceptual: a0 += [x], a1 += [x+32] */
vaddps  ymm4, ymm4, ymm0
vaddps  ymm5, ymm5, ymm1
/* later: acc = ymm4 + ymm5, then horizontal reduce */
.att_syntax prefix
```

9.5.4 Practical workflow

- First: rewrite loop in the simplest vectorizable shape (unit stride, no calls).
- Second: compile with optimizations and inspect assembly output.
- Third: only if needed, introduce hints or intrinsics.
- Finally: benchmark end-to-end with realistic sizes; do not optimize micro-benchmarks only.

Practical summary:

- Auto-vectorization is strong for simple unit-stride loops with clear alias and dependency rules.
- Vectorization fails mainly due to alias uncertainty, irregular access, loop-carried dependencies, branches, and calls.
- Use hints only to express truths; wrong assumptions can break correctness.
- Intrinsics provide control and predictability but add maintenance cost.
- Reading assembly is about recognizing vector width, loop shape, tail handling, shuffles, unrolling, and ISA transitions.

Chapter 10

SIMD Precision, Accuracy, and Pitfalls

10.1 Floating-Point Reordering Effects

10.1.1 Why SIMD changes results

Vectorization often changes the **order of operations**. Floating-point addition/multiplication is not associative:

$$(a + b) + c \neq a + (b + c)$$

Therefore:

- vector reductions (sum/min/max with NaNs) may differ from scalar,
- unrolling creates different partial sums,
- FMA fuses rounding steps and can change last bits.

10.1.2 Example: scalar sum vs vector sum

```
float sum_scalar(const float* x, int n) {
```

```
    float s = 0.0f;
    for (int i = 0; i < n; ++i) s += x[i];
    return s;
}
```

A vectorized reduction (multiple accumulators) changes the grouping:

```
#include <immintrin.h>

static inline float hsum256_ps(__m256 v) {
    __m128 lo = _mm256_castps256_ps128(v);
    __m128 hi = _mm256_extractf128_ps(v, 1);
    __m128 s  = _mm_add_ps(lo, hi);
    s = _mm_hadd_ps(s, s);
    s = _mm_hadd_ps(s, s);
    return _mm_cvtss_f32(s);
}

float sum_avx(const float* x, int n) {
    __m256 a0 = _mm256_setzero_ps();
    __m256 a1 = _mm256_setzero_ps();

    int i = 0;
    for (; i + 16 <= n; i += 16) {
        a0 = _mm256_add_ps(a0, _mm256_loadu_ps(x + i));
        a1 = _mm256_add_ps(a1, _mm256_loadu_ps(x + i + 8));
    }

    float s = hsum256_ps(_mm256_add_ps(a0, a1));
    for (; i < n; ++i) s += x[i];
    return s;
}
```

10.1.3 Engineering rule

Decide per-kernel:

- **Bitwise reproducibility required?** Avoid reassociation/FMA or implement a stable summation strategy.
- **Numerical tolerance acceptable?** SIMD reductions are usually fine; verify with error bounds.

10.2 Denormals and Flush-to-Zero

10.2.1 What denormals do to performance

Denormals (subnormal floats) represent very small magnitudes. On many x86 implementations, denormal handling can be **much slower** than normal FP operations. SIMD-heavy code can suddenly collapse in throughput when values drift into the subnormal range.

10.2.2 FTZ and DAZ

Two common controls:

- **FTZ (Flush-To-Zero):** results that would be denormal are flushed to 0.
- **DAZ (Denormals-Are-Zero):** denormal inputs are treated as 0.

These improve performance but change numerical behavior for tiny magnitudes.

10.2.3 Controlling FTZ/DAZ via MXCSR

```
#include <xmmintrin.h>
```

```c
/* Enable FTZ and DAZ for the current thread (SSE MXCSR). */
void enable_ftz_daz() {
    unsigned mxcsr = _mm_getcsr();
    mxcsr |= (1u << 15); /* FTZ */
    mxcsr |= (1u << 6);  /* DAZ */
    _mm_setcsr(mxcsr);
}

/* Restore a previous MXCSR (recommended pattern). */
unsigned save_mxcsr() { return _mm_getcsr(); }
void restore_mxcsr(unsigned v) { _mm_setcsr(v); }
```

10.2.4 Safe usage pattern

```c
void run_kernel_with_ftz_daz(void (*kernel)()) {
    unsigned old = save_mxcsr();
    enable_ftz_daz();
    kernel();
    restore_mxcsr(old);
}
```

10.2.5 Rule

Only enable FTZ/DAZ when your numeric requirements allow it. Treat it as part of the kernel contract and document it.

10.3 Precision Loss in Vectorized Math

10.3.1 Where precision issues come from

- **reassociation** and changed reduction trees,

- **FMA** changing rounding behavior (single rounding),
- **approximate math** from compiler fast-math settings,
- **catastrophic cancellation** (subtracting nearly equal numbers),
- **limited mantissa** in `float` (24-bit significand).

10.3.2 Example: accumulating many floats into float vs double

Accumulate in `double` to reduce error even if the input is `float`.

```
double sum_f32_to_f64(const float* x, int n) {
    double s = 0.0;
    for (int i = 0; i < n; ++i) s += (double)x[i];
    return s;
}
```

10.3.3 Vector pattern: accumulate float lanes, reduce into double

```
#include <immintrin.h>

/* Still uses float vectors for throughput, but final accumulation is
↪  double. */
double sum_avx_to_double(const float* x, int n) {
    __m256 a0 = _mm256_setzero_ps();
    __m256 a1 = _mm256_setzero_ps();

    int i = 0;
    for (; i + 16 <= n; i += 16) {
        a0 = _mm256_add_ps(a0, _mm256_loadu_ps(x + i));
        a1 = _mm256_add_ps(a1, _mm256_loadu_ps(x + i + 8));
    }
```

```
    __m256 acc = _mm256_add_ps(a0, a1);

    __m128 lo = _mm256_castps256_ps128(acc);
    __m128 hi = _mm256_extractf128_ps(acc, 1);
    __m128 s  = _mm_add_ps(lo, hi); /* 4 floats now */

    alignas(16) float tmp[4];
    _mm_store_ps(tmp, s);

    double result = (double)tmp[0] + (double)tmp[1] + (double)tmp[2] +
     ↪  (double)tmp[3];
    for (; i < n; ++i) result += (double)x[i];
    return result;
}
```

10.3.4 Rule of thumb

- For sums over large `n`, consider **double accumulation** or a stable summation algorithm.
- Avoid enabling aggressive fast-math unless you explicitly accept changed semantics.

10.4 Integer Overflow and Saturation

10.4.1 Overflow is a correctness problem, not a performance problem

Packed integer operations in SIMD follow the same fundamental rules:

- for unsigned: arithmetic is modulo 2^N unless you use saturating instructions,
- for signed: overflow is a semantic risk at the language level; treat it carefully in C/C++.

10.4.2 Use saturating arithmetic when the domain requires it

SSE2/AVX2 provide saturating add/sub for 8-bit and 16-bit lanes.

```
#include <immintrin.h>

/* Saturating add for unsigned bytes: out = min(a+b, 255) */
void add_sat_u8_avx2(unsigned char* out,
                     const unsigned char* a,
                     const unsigned char* b,
                     int n) {
    int i = 0;
    for (; i + 32 <= n; i += 32) {
        __m256i va = _mm256_loadu_si256((const __m256i*)(a + i));
        __m256i vb = _mm256_loadu_si256((const __m256i*)(b + i));
        __m256i vc = _mm256_adds_epu8(va, vb);
        _mm256_storeu_si256((__m256i*)(out + i), vc);
    }
    for (; i < n; ++i) {
        unsigned t = (unsigned)a[i] + (unsigned)b[i];
        out[i] = (unsigned char)(t > 255u ? 255u : t);
    }
}
```

10.4.3 Widen–compute–narrow (avoid overflow)

When you must multiply or add with headroom, widen first:

```
#include <immintrin.h>

/* Add u8 as u16 to avoid overflow, store u16 results */
void add_u8_widen_u16(const unsigned char* a,
                      const unsigned char* b,
                      unsigned short* out) {
```

```
    __m128i va = _mm_loadu_si128((const __m128i*)a);
    __m128i vb = _mm_loadu_si128((const __m128i*)b);
    __m128i z  = _mm_setzero_si128();

    __m128i a0 = _mm_unpacklo_epi8(va, z);
    __m128i b0 = _mm_unpacklo_epi8(vb, z);
    __m128i s0 = _mm_add_epi16(a0, b0);

    _mm_storeu_si128((__m128i*)out, s0);
}
```

10.5 Debugging SIMD Bugs

10.5.1 The common bug categories

- **out-of-bounds loads/stores** due to tail mistakes,
- **alignment assumptions** violated at runtime,
- **aliasing violations** (input/output overlap),
- **incorrect masks** (signed vs unsigned compares, wrong predicate),
- **lane order errors** (shuffle/permutation mistakes),
- **FP environment differences** (FTZ/DAZ, rounding modes),
- **SSE/AVX transition issues** (missing `vzeroupper` in mixed code).

10.5.2 Debug strategy: scalar reference + randomized tests

Always keep a scalar reference implementation and compare results.

```
#include <cmath>
#include <cstdlib>

static inline int almost_equal(float a, float b, float eps) {
    float da = std::fabs(a - b);
    float ma = std::fabs(a) > std::fabs(b) ? std::fabs(a) : std::fabs(b);
    return da <= eps * (ma > 1.0f ? ma : 1.0f);
}

void check_kernel(const float* x, int n,
                  void (*scalar)(float*, const float*, int),
                  void (*simd)(float*, const float*, int),
                  float eps) {
    float* a = (float*)std::malloc((size_t)n * sizeof(float));
    float* b = (float*)std::malloc((size_t)n * sizeof(float));

    for (int i = 0; i < n; ++i) {
        a[i] = x[i];
        b[i] = x[i];
    }

    scalar(a, x, n);
    simd(b, x, n);

    for (int i = 0; i < n; ++i) {
        if (!almost_equal(a[i], b[i], eps)) {
            /* In real code: print i, a[i], b[i] and abort. */
            break;
        }
    }

    std::free(a);
    std::free(b);
```

```
}
```

10.5.3 Debug strategy: isolate tails and boundaries

Force tests where:

- `n` is just below/at/above vector width (e.g., 7,8,9 for AVX float),
- pointers start at different alignments (offset by 1..63 bytes),
- input contains extreme FP values (NaNs, Infs, denormals if not flushed),
- integer lanes hit overflow boundaries.

10.5.4 Debug strategy: inspect intermediate vectors

For intrinsics, store intermediate values to arrays and print them in debug builds.

```
#include <immintrin.h>

void dump_vec8(__m256 v, float out[8]) { _mm256_storeu_ps(out, v); }
```

10.5.5 Assembly intuition: silent bugs are often tails or masks

```
.intel_syntax noprefix
/* Typical bug pattern: vector loop correct, tail missing or wrong
 ↪  predicate id in compare. */
vcmpps ymm2, ymm0, ymm1, 14   /* predicate must match intended
 ↪  compare */
.att_syntax prefix
```

Practical summary:

- SIMD often changes FP results by reordering and by enabling FMA; decide reproducibility vs tolerance.
- Denormals can destroy throughput; FTZ/DAZ may help but changes semantics.
- Precision loss comes from limited mantissa and cancellation; use double accumulation or stable summation when needed.
- Integer SIMD must handle overflow intentionally: saturate or widen–compute–narrow.
- Debug SIMD with scalar references, randomized tests, boundary sizes, alignment offsets, and intermediate dumps.

Chapter 11

Performance Measurement and Analysis

11.1 Benchmarking SIMD Code Correctly

11.1.1 What “correct” means for SIMD benchmarks

A useful SIMD benchmark must satisfy:

- **correctness validated** against a reference,
- **dead-code elimination prevented**,
- **steady-state timing** (warm caches / stable frequency),
- **representative problem sizes** (in-cache and out-of-cache),
- **repeatability** (multiple runs, stable statistics).

11.1.2 A minimal, robust timing harness

```
#include <chrono>
```

```
#include <cstdint>

static inline std::uint64_t now_ns() {
    return
     ↪  (std::uint64_t)std::chrono::duration_cast<std::chrono::nanoseconds>(
        std::chrono::high_resolution_clock::now().time_since_epoch()
    ).count();
}

template <class F>
double bench_ns_per_iter(F&& f, int iters) {
    for (int i = 0; i < 10; ++i) f(); /* warm-up */

    std::uint64_t t0 = now_ns();
    for (int i = 0; i < iters; ++i) f();
    std::uint64_t t1 = now_ns();

    return (double)(t1 - t0) / (double)iters;
}
```

11.1.3 Preventing dead-code elimination

If the benchmark result is unused, the compiler may remove the work. Use a volatile sink or accumulate a value that escapes.

```
#include <cstddef>

volatile float g_sink_f32;

template <class F>
void prevent_dce(F&& f) {
    g_sink_f32 += f();
}
```

11.1.4 Example: benchmark a dot product kernel

```
#include <immintrin.h>
#include <cstddef>

static inline float hsum256_ps(__m256 v) {
    __m128 lo = _mm256_castps256_ps128(v);
    __m128 hi = _mm256_extractf128_ps(v, 1);
    __m128 s  = _mm_add_ps(lo, hi);
    s = _mm_hadd_ps(s, s);
    s = _mm_hadd_ps(s, s);
    return _mm_cvtss_f32(s);
}

float dot_avx_fma(const float* a, const float* b, int n) {
    __m256 s0 = _mm256_setzero_ps();
    __m256 s1 = _mm256_setzero_ps();

    int i = 0;
    for (; i + 16 <= n; i += 16) {
        __m256 a0 = _mm256_loadu_ps(a + i);
        __m256 b0 = _mm256_loadu_ps(b + i);
        __m256 a1 = _mm256_loadu_ps(a + i + 8);
        __m256 b1 = _mm256_loadu_ps(b + i + 8);

        s0 = _mm256_fmadd_ps(a0, b0, s0);
        s1 = _mm256_fmadd_ps(a1, b1, s1);
    }

    float r = hsum256_ps(_mm256_add_ps(s0, s1));
    for (; i < n; ++i) r += a[i] * b[i];
    return r;
}
```

```
double bench_dot(const float* a, const float* b, int n, int iters) {
    auto fn = [&]() -> float { return dot_avx_fma(a, b, n); };
    double ns = bench_ns_per_iter([&]() { prevent_dce(fn); }, iters);
    return ns;
}
```

11.2 Cycle Counting and Hardware Counters

11.2.1 Cycle counting: what you can and cannot assume

Cycle counting is valuable but subtle:

- modern CPUs change frequency (turbo, power states),
- out-of-order execution overlaps work; one instruction is not one cycle,
- OS scheduling noise can dominate short runs.

Therefore, treat cycles as a **relative** metric in controlled conditions.

11.2.2 RDTSC for quick, local measurement (x86)

Use serialized measurement to reduce reordering effects. This pattern is for **microbench insight**, not absolute truth.

```
#include <cstdint>

#if defined(_MSC_VER)
#include <intrin.h>
static inline std::uint64_t rdtsc() { return __rdtsc(); }
static inline void cpuid_barrier() { int cpuInfo[4]; __cpuid(cpuInfo, 0); }
#else
```

```
static inline std::uint64_t rdtsc() {
    unsigned hi, lo;
    __asm__ __volatile__("rdtsc" : "=a"(lo), "=d"(hi));
    return ((std::uint64_t)hi << 32) | lo;
}
static inline void cpuid_barrier() {
    unsigned a, b, c, d;
    __asm__ __volatile__("cpuid" : "=a"(a), "=b"(b), "=c"(c), "=d"(d) :
    ↪ "a"(0));
}
#endif

template <class F>
std::uint64_t bench_cycles(F&& f) {
    cpuid_barrier();
    std::uint64_t t0 = rdtsc();
    f();
    cpuid_barrier();
    std::uint64_t t1 = rdtsc();
    return t1 - t0;
}
```

11.2.3 Hardware performance counters (conceptual workflow)

Counters tell you *why* performance behaves as it does:

- instruction and uop throughput,
- cache misses and bandwidth,
- branch misses (even in SIMD code, outer loops),
- stalls due to memory ordering or resource conflicts.

Use counters to validate hypotheses (memory-bound, front-end bound, back-end bound).

11.3 Identifying Compute-Bound vs Memory-Bound Code

11.3.1 Two quick tests

Test A (scale computation): Add extra arithmetic per element without changing memory traffic.

- If runtime barely changes, you were memory-bound.
- If runtime increases proportionally, you were compute-bound.

Test B (scale memory traffic): Add an extra pass over memory (or extra stream).

- If runtime increases strongly, memory bandwidth/latency dominates.

11.3.2 Example: adding math to detect memory-bound behavior

Baseline:

```
#include <immintrin.h>

void scale_avx(float* y, const float* x, int n, float c) {
    const __m256 vc = _mm256_set1_ps(c);
    int i = 0;
    for (; i + 8 <= n; i += 8) {
        __m256 v = _mm256_loadu_ps(x + i);
        v = _mm256_mul_ps(v, vc);
        _mm256_storeu_ps(y + i, v);
    }
    for (; i < n; ++i) y[i] = x[i] * c;
}
```

Heavier compute, same traffic:

```
#include <immintrin.h>

/* same loads/stores, more ALU work */
void scale_poly_avx(float* y, const float* x, int n, float c) {
    const __m256 vc = _mm256_set1_ps(c);
    const __m256 one = _mm256_set1_ps(1.0f);

    int i = 0;
    for (; i + 8 <= n; i += 8) {
        __m256 v = _mm256_loadu_ps(x + i);
        __m256 r = _mm256_mul_ps(v, vc);
        r = _mm256_fmadd_ps(r, r, one);
        r = _mm256_fmadd_ps(r, v, one);
        _mm256_storeu_ps(y + i, r);
    }

    for (; i < n; ++i) {
        float v = x[i];
        float r = v * c;
        r = r * r + 1.0f;
        r = r * v + 1.0f;
        y[i] = r;
    }
}
```

11.3.3 Bytes and arithmetic intensity

For a kernel, estimate:

- bytes per element (loads + stores),
- operations per element,
- $I = ops/bytes$.

Low I suggests memory-bound behavior on large arrays.

11.4 Scaling Behavior with Vector Width

11.4.1 What scaling you should expect

Moving SSE $\rightarrow$ AVX $\rightarrow$ AVX2 doubles lane width (128 $\rightarrow$ 256). The theoretical max speedup is often $\approx 2\times$, but real scaling is bounded by:

- memory bandwidth (dominant in streaming kernels),
- load/store throughput limits,
- instruction mix (shuffles, permutes, gathers),
- frequency behavior (some CPUs reduce clocks under heavy wide-vector usage),
- register pressure and front-end limits.

11.4.2 A controlled scaling experiment

Benchmark the **same algorithm** with:

- scalar baseline,
- SSE (128),
- AVX (256),
- optionally AVX2/FMA variants.

Keep:

- same data layout,

- same memory traffic,
- same tail handling policy.

11.4.3 Example: same kernel in SSE vs AVX

```
#include <xmmintrin.h>
#include <immintrin.h>

void add_sse(float* y, const float* x, int n) {
    int i = 0;
    for (; i + 4 <= n; i += 4) {
        __m128 vx = _mm_loadu_ps(x + i);
        __m128 vy = _mm_loadu_ps(y + i);
        _mm_storeu_ps(y + i, _mm_add_ps(vy, vx));
    }
    for (; i < n; ++i) y[i] += x[i];
}

void add_avx(float* y, const float* x, int n) {
    int i = 0;
    for (; i + 8 <= n; i += 8) {
        __m256 vx = _mm256_loadu_ps(x + i);
        __m256 vy = _mm256_loadu_ps(y + i);
        _mm256_storeu_ps(y + i, _mm256_add_ps(vy, vx));
    }
    for (; i < n; ++i) y[i] += x[i];
}
```

11.4.4 Interpretation

If `add_avx` is not close to `2x` faster than `add_sse` for large arrays, you are almost certainly limited by memory bandwidth or load/store throughput.

11.5 Avoiding Misleading Benchmarks

11.5.1 The classic benchmark traps

- **Too small inputs:** everything stays in L1/L2 and looks unrealistically fast.
- **Cold vs warm confusion:** first run includes page faults and cold misses.
- **Measuring the timer:** too few iterations makes overhead dominate.
- **Compiler deletes work:** unused results or constant inputs get optimized away.
- **Non-representative alignment:** testing only aligned or only unaligned hides worst-case.
- **Frequency instability:** turbo and thermal limits distort results.
- **Comparing different algorithms:** an "optimized" version changes work, not just ISA.

11.5.2 A disciplined checklist for SIMD benchmarks

- Validate outputs vs a known-correct reference.
- Use multiple sizes: fits in L1, fits in L2, exceeds LLC, exceeds RAM caches.
- Report both time and derived metrics (ns/elem, GB/s, FLOP/s).
- Run enough iterations for stable results; repeat and take median.
- Control alignment and test multiple alignments/offsets.
- Separate setup/allocation from timed region.
- Use counter evidence to confirm the bottleneck (cache misses, bandwidth, uops).

11.5.3 Assembly intuition: measure the loop you think you measure

When you inspect the compiled output, confirm:

- the loop was vectorized (XMM/YMM instructions present),
- there is no unexpected scalar fallback in the hot region,
- tail handling is outside the measured steady-state core (or included intentionally),
- ISA transition overhead is not contaminating the loop (e.g., missing `vzeroupper` around mixed code).

```
.intel_syntax noprefix
/* A vectorized inner loop is usually dominated by: load, compute,
 ↪  store */
vmovups ymm0, ymmword ptr [rsi]
vaddps  ymm0, ymm0, ymm1
vmovups ymmword ptr [rdi], ymm0
.att_syntax prefix
```

Practical summary:

- Benchmark correctness-first: prevent DCE, warm up, use representative sizes, repeat runs.
- Use cycle counts for relative insight; hardware counters explain bottlenecks.
- Distinguish compute-bound vs memory-bound by scaling ALU work vs scaling memory traffic.
- Expect imperfect scaling with width due to bandwidth, load/store limits, and frequency behavior.

- Avoid misleading benchmarks by validating compiled assembly and testing realistic workloads.

Chapter 12

Real-World SIMD Design Guidelines

12.1 Choosing the Right SIMD Width

12.1.1 Start from the bottleneck

Choose SIMD width by identifying what limits the kernel:

- **Memory-bound (streaming loads/stores):** wider vectors often do *not* scale linearly.
- **Compute-bound (ALU/FMA dominated):** wider vectors usually help more.
- **Shuffle/permute dominated:** width can increase rearrangement cost and register pressure.

12.1.2 Practical baseline policy on x86

A pragmatic tiered approach:

- **Baseline:** SSE2 (universal on x86-64) for correctness and broad compatibility.

- **Fast path:** AVX2+FMA when available for throughput-critical kernels.
- **Avoid over-committing:** do not assume a wider width is always faster.

12.1.3 Runtime dispatch skeleton (multi-versioning)

```
#include <immintrin.h>
#include <cstdint>

/* Example: feature detection (compiler/OS dependent; keep abstract here).
 ↪  */
static inline int cpu_has_avx2_fma() {
#if defined(__GNUC__)  defined(__clang__)
    /* Many environments support this builtin; if unavailable, return 0. */
    return __builtin_cpu_supports("avx2") && __builtin_cpu_supports("fma");
#else
    return 0;
#endif
}

void kernel_sse2(float* y, const float* x, int n);
void kernel_avx2(float* y, const float* x, int n);

void kernel_dispatch(float* y, const float* x, int n) {
    if (cpu_has_avx2_fma()) kernel_avx2(y, x, n);
    else                    kernel_sse2(y, x, n);
}
```

12.1.4 Design rule

Pick the **simplest width that meets the performance goal** on the target fleet. Then add a wider specialized path only when measurements show real benefit.

12.2 When SIMD Hurts Performance

12.2.1 Common reasons SIMD gets slower

- **Bandwidth ceiling:** the loop is already limited by memory throughput.
- **More shuffles:** vectorization forces expensive lane rearrangement.
- **Register pressure:** spills to stack defeat SIMD gains.
- **Misaligned or split loads:** frequent cache-line splits degrade throughput.
- **Tiny loops:** overhead of setting up vectors outweighs benefit.
- **Hot instruction footprint:** unrolling and intrinsics inflate code size and hurt I-cache.
- **Frequency behavior:** heavy wide-vector usage can reduce core frequency on some CPUs.

12.2.2 Example: shuffle-heavy "bad vectorization"

A common pitfall is forcing AoS data into vector operations that require repeated shuffles. Instead, change layout (SoA/AoSoA) or batch-transform into a vector-friendly buffer.

```
/* Conceptual: if every iteration needs shuffles to extract fields from
↪  AoS,
   vector width may increase shuffle count and register pressure. */
```

12.2.3 Example: too-small n

```
#include <immintrin.h>

/* For very small n, scalar can win due to lower overhead. */
```

```
void add_small(float* y, const float* x, int n) {
    if (n < 16) { /* policy threshold from measurement */
        for (int i = 0; i < n; ++i) y[i] += x[i];
        return;
    }
    int i = 0;
    for (; i + 8 <= n; i += 8) {
        __m256 vx = _mm256_loadu_ps(x + i);
        __m256 vy = _mm256_loadu_ps(y + i);
        _mm256_storeu_ps(y + i, _mm256_add_ps(vy, vx));
    }
    for (; i < n; ++i) y[i] += x[i];
}
```

12.2.4 Rule

Always benchmark with realistic sizes and call patterns. SIMD is not a universal “on/off” win.

12.3 Maintainability vs Raw Speed

12.3.1 Prefer stable abstractions

A maintainable SIMD strategy:

- keep a clear scalar reference,
- isolate SIMD code in a small number of translation units,
- provide clean APIs that hide ISA details,
- write tests that validate SIMD paths against scalar.

12.3.2 Example: single-purpose kernel interface

```
struct Kernel {
    void (*run)(float* out, const float* in, int n);
};

void run_kernel(const Kernel& k, float* out, const float* in, int n) {
    k.run(out, in, n);
}
```

12.3.3 Keep intrinsics local, not everywhere

Do not spread intrinsics across business logic. Put them behind:

- kernel functions,
- small “SIMD math” modules,
- or codegen-generated units if you use multiple ISAs.

12.3.4 Correctness-first rule

Any optimization that changes FP results must be justified by an explicit numeric policy (tolerances, reproducibility requirements, FTZ/DAZ policy).

12.4 SIMD in Large Codebases

12.4.1 Architecture patterns that scale

- **Hot kernels isolated:** limited entry points, easy to benchmark and tune.
- **Feature-gated dispatch:** runtime dispatch or build-time multi-versioning.

- **Consistent data layout:** SoA/AoSoA for hot paths; avoid per-kernel layout hacks.
- **Clear fallbacks:** scalar or SSE2 baseline always available.

12.4.2 Avoid ISA mixing hazards

If you mix SSE and AVX paths in the same call chain:

- keep regions ISA-consistent,
- insert `vzeroupper` at boundaries returning to legacy SSE code.

```
#include <immintrin.h>

void avx_region() { /* ... */ }

void call_chain_with_boundary() {
    avx_region();
    _mm256_zeroupper(); /* boundary hygiene */
    /* legacy SSE or scalar-heavy code can follow */
}
```

12.4.3 Testing strategy for large systems

- randomized tests for numeric kernels,
- boundary tests for tails (n near vector width),
- alignment-offset tests (different pointer alignments),
- FP edge tests (NaN/Inf/denormals depending on policy),
- A/B tests against scalar for correctness and error bounds.

12.5 Long-Term Portability Considerations

12.5.1 Portability is an architectural decision

Even “x86-only” systems evolve:

- ISA levels differ across machines (SSE2, AVX, AVX2, AVX-512),
- OS support affects what is usable (state saving, ABI rules),
- compilers differ in vectorization and intrinsic support behavior.

12.5.2 Design for multiple implementations

A robust long-term plan:

1. Keep a clean scalar reference for correctness and portability.
2. Provide an SSE2 baseline SIMD path.
3. Add AVX2/FMA as an optional fast path via dispatch.
4. Keep data layout decisions independent of the ISA as much as possible.

12.5.3 Avoid over-specialization

Over-specialization risks:

- locking into one microarchitecture’s sweet spots,
- fragile code that breaks with small changes,
- lost performance when real workloads change.

12.5.4 A practical “SIMD contract” for kernel APIs

Document and enforce:

- aliasing rules (whether in/out may overlap),
- alignment assumptions (if any),
- FP environment requirements (FTZ/DAZ, rounding),
- numeric tolerances or reproducibility requirements,
- supported ISA levels and dispatch rules.

12.5.5 Assembly intuition: portable performance comes from stable loop shapes

```
.intel_syntax noprefix
/* The most portable high-performance inner loop is still:
 ↪  unit-stride load, compute, store. */
vmovups ymm0, ymmword ptr [rsi]
vfmadd231ps ymm0, ymm1, ymm2          /* example: fused kernel */
vmovups ymmword ptr [rdi], ymm0
.att_syntax prefix
```

Practical summary:

- Choose SIMD width based on the bottleneck; wider is not automatically faster.
- SIMD can hurt due to bandwidth ceilings, shuffles, spills, code size, small n, and frequency behavior.
- Preserve maintainability: isolate intrinsics, keep scalar reference, and test thoroughly.

- Large codebases need dispatch, data-layout discipline, and ISA boundary hygiene (`vzeroupper`).
- Long-term portability requires multiple implementations and explicit kernel contracts.

Appendices

Appendix A — SIMD Instruction Reference Overview

Common SSE Instruction Groups

SSE instruction sets evolved in layers (SSE, SSE2, SSE3, SSSE3, SSE4.1/4.2). In practice, treat them as **families of operations** you repeatedly use in real kernels.

Loads and Stores (XMM, 128-bit)

- **Float loads/stores:** `movups/movaps` (`_mm_loadu_ps`, `_mm_load_ps`)
- **Integer loads/stores (SSE2):** `movdqu/movdqa` (`_mm_loadu_si128`, `_mm_load_si128`)
- **Scalar moves:** `movss/movsd` for lane0 scalar FP

Arithmetic (packed float / packed double / packed int)

- **Packed float:** `addps, subps, mulps, divps, sqrtps, maxps, minps`
- **Packed double (SSE2):** `addpd, subpd, mulpd, divpd, sqrtpd, maxpd, minpd`

- **Packed int (SSE2):** `paddb/w/d/q, psubb/w/d/q, pmullw, pmuludq`
- **Saturating int:** `paddusb/paddusw, paddsb/paddsw, psubusb/psubusw, psubsb/psubsw`

Compare, Test, and Mask Formation

- **FP compares:** `cmpps/cmppd` (produce all-ones/all-zeros per lane)
- **Integer compares (SSE2):** `pcmpeqb/w/d, pcmpgtb/w/d`
- **Bitmask extraction:** `movmskps/movmskpd, pmovmskb`

Logical and Bitwise

- **FP bitwise:** `andps, andnps, orps, xorps` (often used for select)
- **Integer bitwise:** `pand, pandn, por, pxor` (SSE2)

Shuffles, Unpacks, and Lane Rearrangement

- **Unpack/interleave:** `unpcklps/unpckhps`, `punpcklbw/...`
- **Shuffle:** `shufps` (lane reorder for floats)
- **Move/extract:** `movhlps, movlhps`
- **Horizontal ops (SSE3):** `haddps/hsubps`
- **Byte shuffle (SSSE3):** `pshufb` (powerful byte permutation)
- **Blend (SSE4.1):** `blendps/blendpd/pblendvb`

Convert and Pack/Unpack

- **FP ↔ int:** `cvttps2dq/cvtdq2ps`, `cvttpd2dq/cvtdq2pd`
- **Pack with saturation:** `packsswb/packuswb`, `packssdw`

Representative SSE patterns (assembly snippets)

```
.intel_syntax noprefix
/* load + compute + store (packed float) */
movups xmm0, xmmword ptr [rsi]
addps  xmm0, xmm1
movups xmmword ptr [rdi], xmm0
.att_syntax prefix
```

```
.intel_syntax noprefix
/* compare -> mask -> select with AND/ANDN/OR */
movups xmm0, xmmword ptr [rsi]          /* x */
cmpps  xmm2, xmm0, xmm1, 14              /* predicate id example */
andps  xmm3, xmm2                        /* x & mask (conceptual:
 ↪  arrange operands as needed) */
andnps xmm2, xmm4                        /* (~mask) & alt */
orps   xmm3, xmm2
.att_syntax prefix
```

Common AVX / AVX2 Instruction Groups

AVX introduces VEX encoding and YMM registers (256-bit), enabling **3-operand non-destructive** forms. AVX2 extends 256-bit support to most integer operations.

Loads and Stores (YMM, 256-bit)

- **Float/double moves:** `vmovups/vmovaps, vmovupd/vmovapd`
- **Integer moves:** `vmovdqu/vmovdqa` (AVX2)
- **Non-temporal stores:** `vmovntps/vmovntdq` (useful for write-only streams)
- **Upper-lane hygiene:** `vzeroupper` (avoid AVX→SSE transition costs)

Arithmetic (packed float/double) + FMA

- **Packed float:** `vaddps, vsubps, vmulps, vdivps, vsqrtps, vminps, vmaxps`
- **Packed double:** `vaddpd, vsubpd, vmulpd, vdivpd, vsqrtpd, vminpd, vmaxpd`
- **FMA:** `vfmadd*` forms (fused multiply-add, single rounding)

Integer SIMD (AVX2 core groups)

- **Add/sub:** `vpaddb/w/d/q, vpsubb/w/d/q`
- **Saturating:** `vpaddusb/vpaddusw, vpaddsb/vpaddsw` and sub variants
- **Multiply:** `vpmullw, vpmulld, vpmuludq` (and related)
- **Compare:** `vpcmpeqb/w/d, vpcmpgtb/w/d`
- **Min/max (some types):** `vpmin* / vpmax*` variants where available

Logical, Shifts, and Bit Manipulation

- **Bitwise:** `vandps/vandnps/vorps/vxorps`, `vpand/vpandn/vpor/vpxor`
- **Shifts:** `vpsllw/d/q, vpsrlw/d/q, vpsraw/psrad`
- **Mask extraction:** `vmovmskps, vpmovmskb`

Permutes, Shuffles, and Lane Crossings

- **Within-lane shuffles:** `vshufps`, `vpshufd`
- **Byte shuffle (AVX2):** `vpshufb` (per 128-bit lane)
- **Lane permute/extract:** `vperm2f128, vextractf128, vinsertf128`
- **Variable permute:** `vpermps` and related (use with care)

Representative AVX/AVX2 patterns (assembly snippets)

```
.intel_syntax noprefix
/* load + compute + store (AVX, 8 floats) */
vmovups ymm0, ymmword ptr [rsi]
vaddps  ymm0, ymm0, ymm1
vmovups ymmword ptr [rdi], ymm0
.att_syntax prefix
```

```
.intel_syntax noprefix
/* FMA: r = a*b + r */
vmovups ymm0, ymmword ptr [rsi]
vmovups ymm1, ymmword ptr [rdx]
vfmadd231ps ymm2, ymm0, ymm1          /* ymm2 = ymm2 + ymm0*ymm1 */
.att_syntax prefix
```

```
.intel_syntax noprefix
/* compare -> mask bits */
vcmpps  ymm2, ymm0, ymm1, 14          /* predicate id example */
vmovmskps eax, ymm2                   /* 8-bit mask */
.att_syntax prefix
```

Load, Compute, Shuffle, Store Categories

Category 1: Load

Goal: feed execution units efficiently.

- Prefer unit-stride contiguous loads (`movups`/`vmovups`).
- Use aligned loads when you can guarantee alignment.
- Avoid patterns that cross cache-line boundaries frequently.

Category 2: Compute

Goal: maximize throughput while controlling dependencies.

- Use multiple accumulators for reductions.
- Prefer FMA where numerically acceptable.
- Keep computations vertical (per-lane) as much as possible.

Category 3: Shuffle / Permute / Blend

Goal: rearrange lanes only when it reduces total work.

- Treat shuffles as expensive glue; reduce their count.

- Prefer SoA/AoSoA layouts to avoid repeated shuffles.
- Remember many AVX2 byte shuffles are lane-local (128-bit lanes).

Category 4: Store

Goal: retire results without overwhelming the memory subsystem.

- Use full-width stores (avoid partial writes in hot loops).
- Consider streaming stores for write-only large outputs.
- Keep stores contiguous to benefit from write combining.

A compact "kernel template" tying the categories together

```
#include <immintrin.h>

/* Template pattern: load -> compute -> store (+ tail) */
void kernel_template(float* y, const float* x, int n, float c) {
    const __m256 vc = _mm256_set1_ps(c);

    int i = 0;
    for (; i + 8 <= n; i += 8) {                    /* Load */
        __m256 vx = _mm256_loadu_ps(x + i);

        __m256 vy = _mm256_mul_ps(vx, vc);          /* Compute */

        _mm256_storeu_ps(y + i, vy);                /* Store */
    }

    for (; i < n; ++i) y[i] = x[i] * c;             /* Tail */
}
```

Appendix A takeaway:

- Learn SIMD instructions by **groups** (load/compute/shuffle/store), not by memorizing mnemonics.
- SSE groups cover 128-bit XMM operations; AVX/AVX2 extend width and add 3-operand forms and 256-bit integer power.
- Real performance depends less on "which instruction" and more on how you organize data movement, dependencies, and shuffles.

Appendix B — SIMD Performance Rules of Thumb

Alignment and Access Rules

Rule 1: Prefer unit-stride contiguous access

- Best-case SIMD loops load/store sequential addresses.
- Strided or gather-like patterns often turn compute-bound kernels into memory-latency-bound kernels.

Rule 2: Align when you can guarantee it; otherwise use unaligned safely

- If you **control allocation**, align arrays to 16B (SSE) or 32B (AVX) to reduce split-line effects.
- If you **cannot guarantee alignment**, always use unaligned loads/stores and keep correctness first.
- Avoid crossing cache-line boundaries frequently (e.g., repeated offsets that force split loads).

```
#include <immintrin.h>
#include <cstddef>

/* Safe default: unaligned loads/stores. */
void add_avx_unaligned(float* y, const float* x, int n) {
    int i = 0;
    for (; i + 8 <= n; i += 8) {
        __m256 vx = _mm256_loadu_ps(x + i);
        __m256 vy = _mm256_loadu_ps(y + i);
        _mm256_storeu_ps(y + i, _mm256_add_ps(vy, vx));
    }
    for (; i < n; ++i) y[i] += x[i];
}
```

Rule 3: Avoid out-of-bounds vector loads

- The simplest safe policy is **scalar tail**.
- If you use padding, make it explicit and validated.
- If you must do masked tails on AVX (no AVX-512), use a temporary buffer copy.

```
#include <immintrin.h>
#include <cstring>

void add_avx_safe_tail(float* y, const float* x, int n) {
    int i = 0;
    for (; i + 8 <= n; i += 8) {
        __m256 vx = _mm256_loadu_ps(x + i);
        __m256 vy = _mm256_loadu_ps(y + i);
        _mm256_storeu_ps(y + i, _mm256_add_ps(vy, vx));
    }

    int r = n - i;
```

```
    if (r) {
        alignas(32) float tx[8] = {0};
        alignas(32) float ty[8] = {0};
        std::memcpy(tx, x + i, (size_t)r * sizeof(float));
        std::memcpy(ty, y + i, (size_t)r * sizeof(float));

        __m256 vx = _mm256_load_ps(tx);
        __m256 vy = _mm256_load_ps(ty);
        __m256 vr = _mm256_add_ps(vy, vx);
        _mm256_store_ps(ty, vr);

        std::memcpy(y + i, ty, (size_t)r * sizeof(float));
    }
}
```

Rule 4: Prefer SoA/AoSoA when fields are processed independently

- AoS often forces repeated shuffles to extract fields.
- SoA/AoSoA makes loads contiguous and reduces shuffle overhead.

Instruction Selection Heuristics

Rule 1: Prefer min/max and bitwise select over branches

- For clamps and piecewise logic: `min/max` often beats compare+blend.
- For general conditional selection: use mask + `and/andnot/or`.

```
#include <immintrin.h>

static inline __m256 select_ps(__m256 m, __m256 a, __m256 b) {
    return _mm256_or_ps(_mm256_and_ps(m, a), _mm256_andnot_ps(m, b));
}
```

```
void relu_avx(float* y, const float* x, int n) {
    const __m256 zero = _mm256_setzero_ps();
    int i = 0;
    for (; i + 8 <= n; i += 8) {
        __m256 vx = _mm256_loadu_ps(x + i);
        __m256 r  = _mm256_max_ps(vx, zero); /* min/max idiom */
        _mm256_storeu_ps(y + i, r);
    }
    for (; i < n; ++i) y[i] = (x[i] > 0.0f) ? x[i] : 0.0f;
}
```

Rule 2: Use FMA when numerically acceptable

- FMA can improve throughput and reduce rounding steps.
- It can also change results vs separate mul+add; enforce numeric policy.

Rule 3: Avoid gathers unless you truly need them

- Irregular memory access tends to be latency-bound.
- If possible, rearrange data once (layout transform) and then run regular SIMD loops.

Rule 4: Shuffles are glue; minimize them

- Too many shuffles often means the layout is wrong.
- Many AVX2 shuffles operate per 128-bit lane; cross-lane shuffles require special permute ops.

Loop Structure Guidelines

Rule 1: Keep the hot loop straight-line

- Avoid function calls, unpredictable branches, and complex control flow.
- Compute masks and select values branchlessly.

Rule 2: Separate main vector loop from tail

- Vector loop: full-width iterations only.
- Tail: scalar remainder (default) or safe masked tail.

Rule 3: Use multiple accumulators for reductions

- Reductions are dependency-chain limited; ILP requires parallel accumulators.
- Reduce horizontally once, outside the main loop.

```
#include <immintrin.h>

static inline float hsum256_ps(__m256 v) {
    __m128 lo = _mm256_castps256_ps128(v);
    __m128 hi = _mm256_extractf128_ps(v, 1);
    __m128 s  = _mm_add_ps(lo, hi);
    s = _mm_hadd_ps(s, s);
    s = _mm_hadd_ps(s, s);
    return _mm_cvtss_f32(s);
}

float sum_avx_2acc(const float* x, int n) {
    __m256 a0 = _mm256_setzero_ps();
    __m256 a1 = _mm256_setzero_ps();

    int i = 0;
```

```
    for (; i + 16 <= n; i += 16) {
        a0 = _mm256_add_ps(a0, _mm256_loadu_ps(x + i));
        a1 = _mm256_add_ps(a1, _mm256_loadu_ps(x + i + 8));
    }

    float s = hsum256_ps(_mm256_add_ps(a0, a1));
    for (; i < n; ++i) s += x[i];
    return s;
}
```

Rule 4: Unroll only until it helps

- Unroll improves ILP and hides latency.
- Stop when register spills appear or code size hurts I-cache.

```
.intel_syntax noprefix
/* unroll-by-2 concept: two independent streams */
vmovups ymm0, ymmword ptr [rsi]
vmovups ymm1, ymmword ptr [rsi + 32]
vaddps  ymm0, ymm0, ymm2
vaddps  ymm1, ymm1, ymm2
vmovups ymmword ptr [rdi], ymm0
vmovups ymmword ptr [rdi + 32], ymm1
.att_syntax prefix
```

Register Pressure Considerations

Why register pressure matters

When live values exceed the available registers, the compiler spills to stack, causing:

- extra loads/stores,

- cache pressure,
- hidden dependencies that reduce OoO overlap.

Rule 1: Count live vectors in the hot loop

A simple accounting method:

- inputs: how many vector temporaries are simultaneously needed?
- constants: broadcast vectors (`set1`) occupy registers too.
- accumulators: each independent accumulator consumes one register.

Rule 2: Reduce live ranges

- compute-and-consume: do not keep values alive longer than needed,
- split kernels into stages if necessary,
- store partial results if it removes large live sets (only if the store is not dominant).

Rule 3: Avoid needless temporaries

Prefer fused expressions and in-place updates when safe.

```
#include <immintrin.h>

/* Fewer temporaries: compute directly into destination variable */
void fused_avx(float* y, const float* x, int n, float a, float b) {
    const __m256 va = _mm256_set1_ps(a);
    const __m256 vb = _mm256_set1_ps(b);

    int i = 0;
    for (; i + 8 <= n; i += 8) {
```

```
        __m256 vx = _mm256_loadu_ps(x + i);
        __m256 vy = _mm256_fmadd_ps(va, vx, vb);
        _mm256_storeu_ps(y + i, vy);
    }
    for (; i < n; ++i) y[i] = a * x[i] + b;
}
```

Rule 4: Watch for spills indirectly

Even without tooling, spills often show up as:

- unexpected stack traffic in assembly,
- loss of scaling when unrolling further,
- performance regressions when adding “one more temporary”.

Appendix B takeaway:

- Favor unit-stride access and correct tails; alignment helps only when guaranteed.
- Choose instructions to reduce branches and shuffles; use FMA and saturation intentionally.
- Keep loops straight-line, separate tails, and use multiple accumulators for reductions.
- Manage register pressure: too many live vectors causes spills and destroys SIMD wins.

Appendix C — Common SIMD Anti-Patterns

Over-Vectorization

Anti-pattern: vectorizing work that is not hot

Symptoms:

- complicated SIMD code for cold paths,
- higher bug rate and maintenance cost with no measurable win,
- performance wins vanish in end-to-end profiling.

Fix: profile-first and keep a scalar fast path for tiny sizes

```
#include <immintrin.h>

void add_scalar(float* y, const float* x, int n) {
    for (int i = 0; i < n; ++i) y[i] += x[i];
}

void add_avx(float* y, const float* x, int n) {
    int i = 0;
    for (; i + 8 <= n; i += 8) {
        __m256 vx = _mm256_loadu_ps(x + i);
        __m256 vy = _mm256_loadu_ps(y + i);
        _mm256_storeu_ps(y + i, _mm256_add_ps(vy, vx));
    }
    for (; i < n; ++i) y[i] += x[i];
}

/* Policy: scalar for small n, SIMD for large n (threshold from
↪  measurement). */
void add_mixed(float* y, const float* x, int n) {
    if (n < 32) { add_scalar(y, x, n); return; }
    add_avx(y, x, n);
}
```

Rule

Vectorize only kernels that are:

- repeatedly executed,
- large enough to amortize overhead,
- measured to be bottlenecks.

Scalar-SIMD Transitions

Anti-pattern: frequent lane0 scalar extraction inside the loop

This breaks SIMD throughput by turning a vector loop into a scalar loop with extra moves.

```
/* Bad idea (conceptual): extract every iteration and branch on scalar */
```

Fix: keep decisions vector-wide, extract only occasionally

Use compare + movemask to decide if any lane triggers a slow path, then handle outside.

```
#include <immintrin.h>

/* Example: detect "any negative" in each 8-float block, handle separately
↪  */
int any_negative_blocks(const float* x, int n) {
    const __m256 zero = _mm256_setzero_ps();
    int blocks = 0;

    int i = 0;
    for (; i + 8 <= n; i += 8) {
        __m256 vx = _mm256_loadu_ps(x + i);
        __m256 m  = _mm256_cmp_ps(vx, zero, _CMP_LT_OQ);
        int bits  = _mm256_movemask_ps(m);
        blocks += (bits != 0);
    }
    for (; i < n; ++i) blocks += (x[i] < 0.0f);
```

```
    return blocks;
}
```

Anti-pattern: AVX region followed by SSE without hygiene

Mixing AVX code with legacy SSE code in a call chain can trigger transition penalties.

Fix: `vzeroupper` at ISA boundaries

```
#include <immintrin.h>

void avx_kernel(float* y, const float* x, int n) { (void)y; (void)x;
↪  (void)n; /* ... */ }

void call_with_boundary(float* y, const float* x, int n) {
    avx_kernel(y, x, n);
    _mm256_zeroupper(); /* boundary hygiene before legacy SSE-heavy code */
}
```

```
.intel_syntax noprefix
/* boundary hygiene */
vzeroupper
.att_syntax prefix
```

Excessive Shuffling

Anti-pattern: using SIMD as a format-conversion engine each iteration

Symptoms:

- many shuffles/permutations per few arithmetic ops,
- poor scaling from SSE to AVX,
- high register pressure and spills.

Example: AoS field extraction in the hot loop

If data is `struct {float x,y,z,w;}` and you repeatedly need all `x` values, AoS forces extraction/shuffles.

```
struct V4 { float x,y,z,w; };

/* Anti-pattern: repeated gathers/extractions from AoS in the hot loop
↪  (conceptual). */
```

Fix 1: change layout to SoA/AoSoA

```
struct SoA4 {
    float* x;
    float* y;
    float* z;
    float* w;
};
/* Now each field is unit-stride and loads cleanly into SIMD. */
```

Fix 2: pre-transform once, then run simple SIMD kernels

```
#include <cstddef>

/* One-time AoS -> SoA transform, then SIMD on SoA arrays. */
void aos_to_soa_x(float* out_x, const V4* in, int n) {
    for (int i = 0; i < n; ++i) out_x[i] = in[i].x;
}
```

Rule

If a kernel spends a large fraction of its uops on shuffles, the real fix is often **data layout**, not "more SIMD".

Memory-Dominated Vector Code

Anti-pattern: expecting width to solve a bandwidth ceiling

A streaming kernel (copy/add/saxpy) often saturates memory bandwidth. Doubling vector width may:

- reduce instruction count (some win),
- but not improve GB/s beyond the memory subsystem's limit.

Example: copy kernel is bandwidth-bound for large n

```
#include <immintrin.h>

void copy_avx(float* dst, const float* src, int n) {
    int i = 0;
    for (; i + 8 <= n; i += 8) {
        __m256 v = _mm256_loadu_ps(src + i);
        _mm256_storeu_ps(dst + i, v);
    }
    for (; i < n; ++i) dst[i] = src[i];
}
```

Fix: reduce traffic or improve store policy, not just compute width

Options:

- fuse passes (avoid multiple reads/writes of same arrays),
- avoid extra temporary arrays,
- use streaming stores for write-only large outputs,
- improve layout to keep accesses contiguous and cache-friendly.

Example: streaming store for write-only output

```c
#include <immintrin.h>

void copy_stream_avx(float* dst, const float* src, int n) {
    int i = 0;
    for (; i + 8 <= n; i += 8) {
        __m256 v = _mm256_loadu_ps(src + i);
        _mm256_stream_ps(dst + i, v);
    }
    for (; i < n; ++i) dst[i] = src[i];
    _mm_sfence();
}
```

```asm
.intel_syntax noprefix
/* memory-dominated inner loop is mostly moves */
vmovups ymm0, ymmword ptr [rsi]
vmovups ymmword ptr [rdi], ymm0
.att_syntax prefix
```

Rule

If the kernel is bandwidth-bound, focus on:

- fewer bytes moved per result,
- fewer passes,
- better store policy and layout,
- correct measurement in GB/s.

Appendix C takeaway:

- Over-vectorization increases complexity without end-to-end gains; profile-first.

- Avoid scalar extraction and frequent scalar-SIMD transitions; keep decisions vector-wide.
- Excessive shuffling is usually a layout problem; fix the data, not the mnemonics.
- Memory-dominated kernels hit bandwidth ceilings; optimize traffic and policy, not just SIMD width.

Appendix D — Tooling and Inspection Techniques

Compiler Flags for SIMD Visibility

Goals

When tuning SIMD, you need three kinds of visibility:

- **Vectorization diagnostics:** did the compiler vectorize, and why/why not?
- **Generated assembly:** what instructions and loop shapes were emitted?
- **Target ISA selection:** which SIMD features are enabled for codegen?

GCC / Clang flag families (practical)

- **Optimization level:** `-O2` or `-O3`
- **ISA selection:** `-msse2`, `-mavx`, `-mavx2`, `-mfma`
- **Emit assembly:** `-S` (and optionally `-fverbose-asm`)
- **Vectorization reports:** diagnostic flags (compiler-specific)
- **Prevent excessive "cleverness":** control fast-math / strict FP via explicit options

Example build command lines (conceptual templates)

```
# Baseline: generate assembly with optimization (GCC/Clang style)
c++ -O3 -S -fverbose-asm -masm=intel -o kernel.s kernel.cpp

# Force AVX2 + FMA codegen for a file
c++ -O3 -mavx2 -mfma -S -fverbose-asm -masm=intel -o kernel_avx2.s
↪  kernel.cpp

# Keep debug symbols while optimizing (useful for perf + disassembly
↪  correlation)
c++ -O3 -g -fno-omit-frame-pointer -mavx2 -mfma -o bench bench.cpp
```

MSVC flag families (practical)

- **Optimization:** `/O2`
- **ISA selection:** `/arch:AVX`, `/arch:AVX2`
- **Assembly listing:** `/FAs` (generates mixed source+asm listing)
- **Debug with optimization:** `/Zi` + `/O2` (with care)

```
cl /O2 /arch:AVX2 /FAs /c kernel.cpp
```

Rule of thumb

For inspection:

- compile twice: baseline (SSE2) and fast-path (AVX2/FMA),
- keep one TU per kernel when possible (reduces noise),
- do not inspect debug builds; inspect optimized builds.

Disassembly and Analysis Workflow

Workflow 1: trust-but-verify vectorization

1. Write a scalar reference + a clean loop version.
2. Compile with optimization and requested ISA.
3. Inspect the hot loop assembly:
    - vector width: XMM vs YMM
    - loop structure: unroll factor, pointer increments
    - tail handling: scalar remainder or special block
    - shuffles: count and placement
    - spills: unexpected stack loads/stores in the hot loop
4. Measure with representative sizes and validate correctness.

A practical disassembly pass checklist

Look for these instruction classes in the hot loop:

- **Loads/stores:** `movups/vmovups, movdqu/vmovdqu`
- **Compute:** `addps/mulps`, `vaddps/vmulps`, `vfmadd*`
- **Masking:** `cmpps/vcmpps` + `movmskps/vmovmskps`
- **Lane ops:** `shufps/vshufps`, `pshufb/vpshufb`, `vperm2f128`
- **Boundary hygiene:** `vzeroupper` before leaving AVX regions that may return to SSE code

Example: what a clean AVX loop looks like

```
.intel_syntax noprefix
/* expected shape: load -> compute -> store -> bump pointers */
vmovups ymm0, ymmword ptr [rsi]
vaddps  ymm0, ymm0, ymm1
vmovups ymmword ptr [rdi], ymm0
add     rsi, 32
add     rdi, 32
/* loop ... */
.att_syntax prefix
```

Example: red flags in disassembly

- many shuffles per iteration for simple math,
- stack traffic (spills) inside the hot loop,
- scalar loads/stores mixed into the vector main loop,
- frequent scalar extraction (`movss/cvtss2si`) in the main loop,
- missing `vzeroupper` around AVX/SSE boundaries in mixed codebases.

Workflow 2: compare manual intrinsics vs compiler

1. Implement intrinsics kernel.
2. Implement “clean scalar” kernel.
3. Compile both with the same ISA flags.
4. Compare:

- instruction mix (FMA usage, shuffle count),
- unroll factor and register usage,
- tail strategy,
- overall throughput from measurement.

Microbenchmark Framework Design

Design requirements

A SIMD microbenchmark must:

- isolate the kernel from setup/allocation,
- prevent dead-code elimination,
- run enough iterations for stable timing,
- support multiple sizes (L1/L2/LLC/DRAM regimes),
- report derived metrics (ns/elem, GB/s, FLOP/s when meaningful).

Minimal framework: time + metrics + DCE defense

```
#include <chrono>
#include <cstdint>
#include <cstddef>

static inline std::uint64_t now_ns() {
    return
     ↪  (std::uint64_t)std::chrono::duration_cast<std::chrono::nanoseconds>(
        std::chrono::high_resolution_clock::now().time_since_epoch()
    ).count();
}
```

```
volatile float g_sink_f32;

template <class F>
double bench_ns_per_iter(F&& f, int iters) {
    for (int i = 0; i < 10; ++i) f(); /* warm-up */
    std::uint64_t t0 = now_ns();
    for (int i = 0; i < iters; ++i) f();
    std::uint64_t t1 = now_ns();
    return (double)(t1 - t0) / (double)iters;
}

template <class F>
double bench_gbps(F&& f, std::size_t bytes_per_iter, int iters) {
    double ns = bench_ns_per_iter(f, iters);
    double sec = ns * 1e-9;
    double gb  = (double)bytes_per_iter / 1e9;
    return gb / sec;
}
```

Example: benchmark a bandwidth-style kernel

```
#include <immintrin.h>
#include <cstddef>

void copy_avx(float* dst, const float* src, int n) {
    int i = 0;
    for (; i + 8 <= n; i += 8) {
        __m256 v = _mm256_loadu_ps(src + i);
        _mm256_storeu_ps(dst + i, v);
    }
    for (; i < n; ++i) dst[i] = src[i];
}
```

```
double bench_copy(float* dst, const float* src, int n, int iters) {
    std::size_t bytes = (std::size_t)2 * (std::size_t)n * sizeof(float);

    auto fn = [&]() {
        copy_avx(dst, src, n);
        g_sink_f32 += dst[0]; /* DCE defense */
    };

    return bench_gbps(fn, bytes, iters);
}
```

Example: benchmark a compute-style kernel

For compute-bound kernels, GB/s is less meaningful; report ns/elem and optionally FLOP/s if you define the operation count.

```
#include <immintrin.h>

void fmadd_avx(float* y, const float* x, int n, float a, float b) {
    const __m256 va = _mm256_set1_ps(a);
    const __m256 vb = _mm256_set1_ps(b);

    int i = 0;
    for (; i + 8 <= n; i += 8) {
        __m256 vx = _mm256_loadu_ps(x + i);
        __m256 vy = _mm256_fmadd_ps(va, vx, vb);
        _mm256_storeu_ps(y + i, vy);
    }
    for (; i < n; ++i) y[i] = a * x[i] + b;
}
```

Avoiding common microbenchmark failure modes

- do not benchmark only tiny sizes (they exaggerate in-cache wins),
- separate allocation/initialization from timing,
- repeat runs and report median (avoid noise spikes),
- validate that the compiler emitted SIMD in the measured function,
- pinning and frequency control are environment concerns; interpret results accordingly.

Appendix D takeaway:

- Use the right flag families to control ISA and to generate readable assembly.
- Inspect assembly with a structured checklist: width, loop shape, tail, shuffles, spills, boundaries.
- Build microbenchmarks that prevent DCE, warm up, test multiple sizes, and report meaningful metrics.

References

x86 Architecture Manuals (Conceptual)

This booklet is grounded in the architectural model defined by the x86-64 ISA and its SIMD extensions as implemented in modern out-of-order superscalar processors.
Conceptual pillars derived from x86 architecture manuals include:

- the separation between **ISA semantics** and **microarchitectural execution**,
- the definition and evolution of SIMD register files (XMM, YMM),
- instruction encoding transitions (legacy SSE $\rightarrow$ VEX),
- memory ordering, alignment guarantees, and cache-line behavior,
- execution ports, pipelines, and instruction throughput vs latency.

The following conceptual domains are directly reflected throughout the booklet:

- SIMD instructions are **vectorized scalar operations**, not magic parallelism.
- Performance depends on **how instructions map to execution units**, not on instruction count alone.
- Load/store behavior and cache interaction dominate many SIMD kernels.

- Wider vectors increase register pressure and resource contention.

The treatment in this booklet intentionally avoids microarchitecture-specific tuning and instead focuses on **portable, architecture-informed reasoning** that remains valid across multiple x86 generations.

SIMD and Vectorization Literature

The conceptual foundation of SIMD programming and vectorization is drawn from long-standing academic and industrial literature on data-parallel execution.
Key themes consistently reinforced by the literature include:

- SIMD exploits **data parallelism**, not instruction-level parallelism.
- Effective vectorization requires **regular memory access patterns**.
- Reductions, scans, and conditionals require explicit restructuring.
- Control-flow-heavy code must be transformed into **mask-based execution**.

This booklet aligns with established findings that:

- Most SIMD speedups come from **loop restructuring**, not instruction tricks.
- Data layout (SoA vs AoS) is often more important than instruction selection.
- Over-vectorization degrades performance and maintainability.
- SIMD should be applied selectively to hot kernels.

The examples emphasize canonical vector patterns (map, reduce, filter, transform) rather than exotic instruction usage, reflecting best practices documented across decades of SIMD research.

Compiler Optimization Documentation

Modern compilers act as aggressive SIMD code generators and optimizers. The guidance in this booklet is aligned with documented compiler behavior and optimization models, including:

- auto-vectorization prerequisites and failure modes,
- alias analysis and its impact on vectorization,
- loop canonicalization and unrolling strategies,
- instruction selection and fusion (e.g., FMA),
- register allocation and spill behavior.

Core principles reflected throughout:

- Compilers vectorize **clean, well-structured loops** best.
- Ambiguous aliasing, complex control flow, and function calls inhibit SIMD.
- Manual intrinsics should be used to **express intent**, not to fight the compiler blindly.
- Inspection of generated assembly is essential for validation.

The booklet consistently treats the compiler as a **partner**, not an adversary, and emphasizes writing code that exposes vectorization opportunities clearly.

Performance Engineering Foundations

The performance methodology used in this booklet follows established performance engineering principles applicable far beyond SIMD.
These foundations include:

- measurement before optimization,
- separation of compute-bound and memory-bound analysis,
- steady-state benchmarking and warm-up effects,
- avoiding microbenchmark traps,
- understanding hardware limits (bandwidth, latency, throughput).

The SIMD-specific conclusions rest on general performance laws:

- Amdahl-style limits apply: SIMD accelerates only the vectorized fraction.
- Memory bandwidth sets hard ceilings for streaming kernels.
- Instruction throughput is secondary when memory dominates.
- Wider vectors do not guarantee linear scaling.

Throughout the booklet, performance claims are framed as:

- architecture-aware,
- measurement-driven,
- reproducible in principle,
- applicable across real-world systems rather than synthetic benchmarks.

Reference philosophy:
This booklet deliberately avoids treating references as a list of external documents. Instead, it distills **stable, widely accepted principles** from authoritative architecture manuals, compiler documentation, academic research, and performance engineering practice into a coherent, practical guide for SIMD on x86.
The goal is not memorization of sources, but mastery of the mental models that those sources collectively establish.