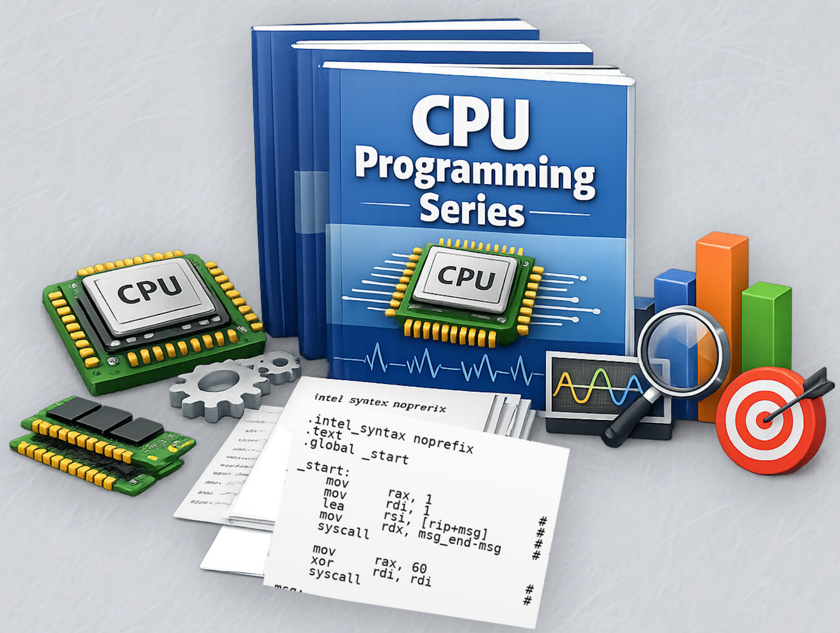
https://simplifycpp.org

CPU Programming Series

NEON on AArch64

SIMD the ARM Way



```
.intel_syntax noprefix
.text
.global _start
_start:
    mov      rax, 1
    mov      rdi, 1
    lea      rsi, [rip+msg]
    mov      rdx, msg_end-msg
    syscall
    mov      rax, 60
    xor      rdi, rdi
    syscall
```

21

Prepared by Ayman Alheraki

CPU Programming Series

NEON on AArch64

SIMD the ARM Way

Prepared by Ayman Alheraki

simplifycpp.org

February 2026

Contents

Preface

Scope and Design Philosophy

This booklet is a **precise, architecture-level study of NEON on AArch64**. Its scope is intentionally constrained to the **Advanced SIMD (NEON) execution engine** as defined and implemented in modern 64-bit ARM processors.
The design philosophy follows clear and non-negotiable principles:

- **Architecture first**: NEON is explained in terms of registers, lanes, instructions, and memory behavior.
- **Fixed-width reality**: NEON is treated strictly as a 128-bit SIMD engine, not as a scalable or abstract vector model.
- **No abstraction leakage**: concepts from SVE, GPUs, or high-level SIMD libraries are intentionally excluded.
- **Performance with correctness**: every optimization discussion is grounded in architectural guarantees, not folklore.

This booklet prioritizes **determinism, predictability, and measurable behavior**. It avoids API-driven explanations and instead builds a mental model aligned with what the CPU actually executes.

What This Booklet Covers (and What It Does Not)

This booklet covers:

- The NEON register model on AArch64 (V0–V31, lanes, aliases)
- Integer and floating-point vector data types
- Lane-based execution semantics
- Load/store behavior and data layout constraints
- Arithmetic, logical, permutation, and reduction instructions
- NEON usage via assembly and compiler intrinsics
- Performance characteristics and common pitfalls

This booklet explicitly does not cover:

- Scalable Vector Extension (SVE / SVE2)
- SME, matrix engines, or AI accelerators
- GPU or heterogeneous compute models
- High-level SIMD libraries or auto-generated kernels
- 32-bit ARM (AArch32) NEON specifics

The intent is to achieve **clarity through isolation**: by focusing solely on NEON, the reader gains a reliable and transferable understanding of SIMD execution on classic AArch64 systems.

Intended Audience

This booklet is written for readers who work close to the hardware and require **precise control over performance and data layout**.
It is intended for:

- Systems and low-level programmers
- Embedded and mobile developers targeting ARM64 platforms
- Compiler, runtime, and toolchain engineers
- Performance engineers optimizing numeric and data-parallel code
- Readers progressing through the *CPU Programming Series*

The reader is expected to be comfortable with:

- Assembly-level reasoning
- Registers, memory hierarchy, and calling conventions
- Basic performance concepts such as latency, throughput, and cache effects

No prior NEON experience is required, but this booklet is **not** an introduction to programming or computer architecture.

Illustrative Example: Scalar vs NEON Thinking

```
/* Scalar-style execution: one value per instruction */
add x0, x0, x1
add x0, x0, x2
add x0, x0, x3
```

```
/* NEON-style execution: multiple values per instruction */
add v0.4s, v1.4s, v2.4s
```

The purpose of this booklet is to teach the reader how and **when** to think in the second form—correctly, safely, and efficiently—based on the actual capabilities and constraints of NEON on AArch64.

Chapter 1

SIMD and NEON Fundamentals

1.1 SIMD Execution Model

SIMD (Single Instruction, Multiple Data) applies one operation to multiple independent data elements packed into a vector register. On AArch64, NEON (Advanced SIMD) uses **fixed 128-bit vector registers** `V0--V31`. Each instruction selects a **lane view**: 8/16/32/64-bit elements, producing 16/8/4/2 lanes respectively. Most NEON arithmetic is **lane-wise**: each lane is computed independently.

Lane Views and Work Per Instruction (Fixed Width)

- `vN.16b`: 16 lanes $\times$ 8-bit
- `vN.8h`: 8 lanes $\times$ 16-bit
- `vN.4s`: 4 lanes $\times$ 32-bit
- `vN.2d`: 2 lanes $\times$ 64-bit

Example: One Instruction, Multiple Independent Adds

```
/* AArch64 GAS syntax */
/* v0.4s = v1.4s + v2.4s (4 x 32-bit lanes) */
add v0.4s, v1.4s, v2.4s

/* v3.16b = v4.16b XOR v5.16b (16 x 8-bit lanes) */
eor v3.16b, v4.16b, v5.16b
```

Data-Parallel Loop Shape: Vector Main Loop + Scalar Tail

A correct SIMD loop has:

- a vectorized main loop that processes full vectors,
- a scalar tail for remaining elements.

```
#include <arm_neon.h>
#include <cstddef>
#include <cstdint>

/* dst[i] = a[i] + b[i] */
void add_u32_neon(std::uint32_t* dst,
                  const std::uint32_t* a,
                  const std::uint32_t* b,
                  std::size_t n)
{
    std::size_t i = 0;

    for (; i + 4 <= n; i += 4) {
        uint32x4_t va = vld1q_u32(a + i);
        uint32x4_t vb = vld1q_u32(b + i);
        uint32x4_t vc = vaddq_u32(va, vb);
        vst1q_u32(dst + i, vc);
    }

    for (; i < n; ++i) {
        dst[i] = a[i] + b[i];
    }
}
```

Explicit Cross-Lane Work Requires Explicit Instructions

Lane-wise arithmetic does not move data between lanes. Cross-lane behavior is explicit: **permute**, **zip/unzip**, **transpose**, **extract/insert**, **horizontal reduce**.

```
/* Pairwise add (horizontal-style operation), reduces within the
 ↪  vector */
addp v0.4s, v1.4s, v2.4s
```

1.2 Why NEON Exists in ARM

NEON exists to provide **high-throughput, energy-efficient data-parallel execution** for workloads dominated by uniform operations over arrays. It reduces the number of retired instructions per unit of work and increases arithmetic intensity when the data layout is favorable.

Typical acceleration targets:

- Multimedia kernels: image, audio, video, pixel/codec primitives
- DSP-like loops: FIR/IIR fragments, transforms, correlations
- Numeric kernels: vector add/mul/FMA, dot-products, reductions
- Bitwise-heavy kernels: masks, comparisons, packing/unpacking

Example: Absolute Difference on 16 Bytes at Once

```
/* v0.16b = abs(v1.16b - v2.16b) */
uabd v0.16b, v1.16b, v2.16b
```

Example: Saturating Arithmetic (Common in Multimedia)

```
/* Unsigned saturating add */
uqadd v0.16b, v1.16b, v2.16b

/* Signed saturating add */
sqadd v3.8h,  v4.8h,  v5.8h
```

Example: Multiply-Accumulate (Kernel Building Block)

```
/* v0.4s = v0.4s + (v1.4s * v2.4s) */
mla v0.4s, v1.4s, v2.4s
```

1.3 NEON vs Scalar Code

Scalar code processes one element per instruction (or per narrow datapath). NEON code processes multiple elements per instruction using the vector datapath. However, NEON only wins when the program can feed the vector engine efficiently.

When NEON Commonly Wins

- contiguous memory access (unit stride),
- few branches inside the loop body,
- enough iterations to amortize setup and tail handling,

- enough independent work to hide latency.

When Scalar Can Win (or NEON Gains Are Small)

- pointer chasing, gathers/scatters, irregular access,
- heavy control flow or data-dependent branches,
- very small problem sizes,
- memory-bound loops where bandwidth is saturated already.

Example: Scalar vs NEON Dot Product (Correct Tail Handling)

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

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

float dot_neon(const float* a, const float* b, std::size_t n)
{
    std::size_t i = 0;
    float32x4_t vacc = vdupq_n_f32(0.0f);

    for (; i + 4 <= n; i += 4) {
        float32x4_t va = vld1q_f32(a + i);
        float32x4_t vb = vld1q_f32(b + i);
        vacc = vfmaq_f32(vacc, va, vb); /* vacc += va * vb */
    }

    float32x2_t sum2 = vadd_f32(vget_low_f32(vacc), vget_high_f32(vacc));
    float acc = vget_lane_f32(vpadd_f32(sum2, sum2), 0);

    for (; i < n; ++i)
        acc += a[i] * b[i];

    return acc;
}
```

Important Practical Point: SIMD Does Not Remove Memory Costs

If the loop is limited by memory bandwidth, SIMD may not speed it up proportionally. In such cases, the best NEON strategy is often to:

- improve locality and alignment,
- reduce passes over memory,
- fuse loops to increase work per byte loaded.

1.4 Where NEON Fits in AArch64

In AArch64, NEON is integrated as **Advanced SIMD** using the same physical vector register file used for floating-point operations. Programmatically, NEON fits in the software stack through three main usage routes:

- **NEON intrinsics**: type-checked, compiler-schedulable, often the best balance.
- **Inline assembly**: maximum control for micro-kernels, higher maintenance cost.

- **Auto-vectorization**: simplest to write, but requires inspection and benchmarking.

Example: Same Operation in Intrinsics and Assembly

Intrinsics:

```
#include <arm_neon.h>

static inline uint32x4_t add4_u32(uint32x4_t a, uint32x4_t b)
{
    return vaddq_u32(a, b);
}
```

Assembly:

```
/* v0.4s = v1.4s + v2.4s */
add v0.4s, v1.4s, v2.4s
```

Example: Minimal AArch64 NEON Loop Skeleton (Assembly)

This skeleton shows the essential structure: load, compute, store, iterate.

```
/* x0 = dst, x1 = a, x2 = b, x3 = n_bytes (multiple of 16 for main
 ↪  loop) */
1:
    /* load 16 bytes from each input */
    ld1 {v0.16b}, [x1], #16
    ld1 {v1.16b}, [x2], #16

    /* compute */
    add v0.16b, v0.16b, v1.16b
    /* store 16 bytes */
    st1 {v0.16b}, [x0], #16

    /* decrement and loop */
    subs x3, x3, #16
    b.ne 1b
```

Key Takeaways for the Rest of the Booklet

- NEON is **fixed-width 128-bit SIMD**: you always know how much data each instruction processes.
- Most arithmetic is **lane-wise**; cross-lane work requires explicit permutes/reductions.
- NEON performance depends as much on **memory layout and loop structure** as on instruction selection.

Chapter 2

AArch64 NEON Register Model

2.1 V Registers (V0–V31)

AArch64 exposes **32 architectural SIMD/FP registers** named `V0` through `V31`. Each `V` register is **128 bits** wide and is used by both:

- NEON (Advanced SIMD) integer/vector instructions, and
- floating-point/SIMD instructions that share the same register file.

The register naming in assembly uses `vN` with a **lane view suffix** that defines how the 128 bits are interpreted (lane count + element size).

Example: Same Register Index, Different Operations

```
/* AArch64 GAS syntax */

/* Integer add on 4 lanes of 32-bit */
add  v0.4s, v1.4s, v2.4s
```

```
/* Floating add on 4 lanes of 32-bit */
fadd v3.4s, v4.4s, v5.4s

/* Bitwise XOR on 16 lanes of bytes */
eor  v6.16b, v6.16b, v7.16b
```

Practical rule

Treat `V0--V31` as **architectural state**: in mixed C/C++ and assembly, your function must obey the platform ABI rules about which vector registers must be preserved across calls.

2.2 Lane Structure and Vector Width

NEON vectors are **fixed-width 128-bit**. A lane is one element within that 128-bit container. Lane count depends only on element size:

$$\text{lanes} = \frac{128}{\text{element\_bits}}$$

Lane counts for 128-bit vectors

- 8-bit elements: 16 lanes (`.16b`)
- 16-bit elements: 8 lanes (`.8h`)
- 32-bit elements: 4 lanes (`.4s`)
- 64-bit elements: 2 lanes (`.2d`)

Lane-wise semantics (default)

Most NEON arithmetic is **lane-wise**: each lane is computed independently, with no implicit lane-to-lane mixing.

```
/* Each lane i: v0.s[i] = v1.s[i] + v2.s[i] */
add v0.4s, v1.4s, v2.4s
```

Cross-lane work is explicit

Lane mixing requires explicit permute/rearrange/reduction instructions (e.g., zip/unzip, ext, tbl, addp).

```
/* Pairwise add (horizontal-style behavior) */
addp v0.4s, v1.4s, v2.4s
```

2.3 Element Sizes and Views

The suffix on a `vN` operand selects the element size and lane count:

- `.b` = 8-bit elements
- `.h` = 16-bit elements
- `.s` = 32-bit elements
- `.d` = 64-bit elements

The **same physical 128 bits** can be interpreted under different views. The view is not a cast; it is part of the instruction encoding and determines the operation.

Example: Same Register Bits, Different Views

```
/* 16 lanes of 8-bit add */
add v0.16b, v1.16b, v2.16b

/* 8 lanes of 16-bit add */
add v0.8h,  v1.8h,  v2.8h

/* 4 lanes of 32-bit add */
add v0.4s,  v1.4s,  v2.4s

/* 2 lanes of 64-bit add */
add v0.2d,  v1.2d,  v2.2d
```

Example: Integer vs Floating Point Uses Different Mnemonics

```
/* Integer add (modular arithmetic) */
add  v0.4s, v1.4s, v2.4s

/* Floating-point add (IEEE-style) */
fadd v0.4s, v1.4s, v2.4s
```

Intrinsic types reflect the same structure

```
#include <arm_neon.h>

/* 128-bit vector types */
uint8x16_t  u8x16;
int16x8_t   s16x8;
uint32x4_t  u32x4;
int64x2_t   s64x2;
```

```
float32x4_t f32x4;
float64x2_t f64x2;
```

2.4 Register Aliasing Rules

AArch64 provides architectural subviews (aliases) of the same SIMD register:

- `Vn`: full 128-bit register (canonical in AArch64)
- `Qn`: full 128-bit view (historical naming, widely used in documentation)
- `Dn`: low 64-bit view of `Vn`

Core aliasing rule

- Writing `Vn` (or `Qn`) overwrites all 128 bits.
- Writing `Dn` overwrites only the low 64 bits; the upper 64 bits remain unchanged until overwritten.

Example: Partial Write Bug Pattern

If you initialize only the low half and later use the full vector, upper lanes can contain stale data.

```
/* Initialize full vector to zero */
movi v1.16b, #0

/* Overwrite only low 64 bits (d1) */
movi d1, #0xffffffffffffffff
```

```
/* Full-width use reads both halves:
   low = 0xff..., high = 0x00... */
add v0.2d, v1.2d, v2.2d
```

Correctness rule of thumb

- If a value will be used as a **full vector**, initialize it as a **full vector** (`movi vN.16b, #0`, `dup`, etc.).
- Avoid mixing `dN` writes with later `vN` computation unless you intentionally preserve upper lanes.

Example: Safe Full Initialization Patterns

```
/* Full 128-bit zero */
movi v0.16b, #0

/* Full 128-bit broadcast of a 32-bit scalar (lane fill) */
dup  v1.4s, w0
```

Intrinsic equivalents (avoid accidental partial state)

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

static inline uint8x16_t  vzero_u8(void)    { return vdupq_n_u8(0); }
static inline uint32x4_t  vbcast_u32(uint32_t x) { return vdupq_n_u32(x); }
static inline float32x4_t vbcast_f32(float x)    { return vdupq_n_f32(x); }
```

Chapter 3

NEON Data Types and Layout

3.1 Integer Vector Types

NEON uses a fixed 128-bit `V` register and interprets it as packed integer lanes. The view selects both **element width** and **lane count**. Integer instructions then define how those bits are interpreted (signed/unsigned, saturating/widening/narrowing).

Core 128-bit Integer Views

- `vN.16b`: 16 lanes of 8-bit integers
- `vN.8h`: 8 lanes of 16-bit integers
- `vN.4s`: 4 lanes of 32-bit integers
- `vN.2d`: 2 lanes of 64-bit integers

Signed vs Unsigned Is an Instruction Property

The same bit-pattern can be treated as signed or unsigned depending on the instruction. Saturating variants are especially sensitive to this distinction.

```
/* Unsigned saturating add on bytes */
uqadd v0.16b, v1.16b, v2.16b

/* Signed saturating add on bytes */
sqadd v3.16b, v4.16b, v5.16b
```

Widening and Narrowing: Avoid Overflow and Control Range

Widening multiply is a standard pattern: multiply small lanes into larger lanes.

```
/* Unsigned widening multiply: 8-bit lanes -> 16-bit results */
umull v0.8h, v1.8b, v2.8b
```

Narrowing is used for packing after intermediate computation.

```
/* Saturating narrow: 16-bit -> 8-bit (useful in image/audio
 ↪  pipelines) */
sqxtn v0.8b, v1.8h
```

Intrinsic Type Mapping

```
#include <arm_neon.h>

/* Integer vector types (128-bit) */
uint8x16_t  u8x16;
int8x16_t   s8x16;
uint16x8_t  u16x8;
```

```
int16x8_t   s16x8;
uint32x4_t  u32x4;
int32x4_t   s32x4;
uint64x2_t  u64x2;
int64x2_t   s64x2;
```

Example: Byte-wise Add Kernel

```
#include <arm_neon.h>
#include <cstddef>
#include <cstdint>

/* dst[i] = a[i] + b[i] */
void add_u8_neon(uint8_t* dst, const uint8_t* a, const uint8_t* b,
 ↪  std::size_t n)
{
    std::size_t i = 0;
    for (; i + 16 <= n; i += 16) {
        uint8x16_t va = vld1q_u8(a + i);
        uint8x16_t vb = vld1q_u8(b + i);
        vst1q_u8(dst + i, vaddq_u8(va, vb));
    }
    for (; i < n; ++i) dst[i] = (uint8_t)(a[i] + b[i]);
}
```

3.2 Floating-Point Vector Types

NEON uses the same `V` registers for floating-point lanes. Floating-point operations are selected via `f*` mnemonics and corresponding intrinsic functions. Common floating-point lane widths are:

- **FP32**: `vN.4s` (4 lanes of 32-bit float)

- **FP64**: `vN.2d` (2 lanes of 64-bit double)

Example: FP32 Add and FMA

```
/* v0.4s = v1.4s + v2.4s */
fadd v0.4s, v1.4s, v2.4s

/* v3.4s = v3.4s + (v4.4s * v5.4s) */
fmla v3.4s, v4.4s, v5.4s
```

Intrinsic Floating Types

```
#include <arm_neon.h>

float32x4_t f32x4;
float64x2_t f64x2;
```

Example: AXPY Kernel (y = a*x + y) Using FMA

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

void axpy_f32(float* y, const float* x, float a, std::size_t n)
{
    std::size_t i = 0;
    float32x4_t va = vdupq_n_f32(a);

    for (; i + 4 <= n; i += 4) {
        float32x4_t vx = vld1q_f32(x + i);
        float32x4_t vy = vld1q_f32(y + i);
        vy = vfmaq_f32(vy, vx, va); /* vy += vx * va */
        vst1q_f32(y + i, vy);
    }

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

Floating-Point Correctness Note (Practical)

Vectorization changes evaluation order (more parallel accumulation), which may change last-bit rounding compared to a scalar loop. When numerical reproducibility matters, use stable reduction strategies (pairwise sums, controlled reduction order) and validate error bounds.

3.3 Lane Indexing Semantics

A lane index is the position of an element inside a vector. Lane indexing is **zero-based** and depends on the chosen view:

- `.16b`: lanes 0..15
- `.8h`: lanes 0..7
- `.4s`: lanes 0..3
- `.2d`: lanes 0..1

Lane indices are used for:

- extract/insert operations,
- lane-based multiply/add forms,
- explicit rearrangement and table lookup instructions,
- reductions and horizontal operations.

Example: Extract and Insert (32-bit Lane)

```
/* Move lane 2 (32-bit) from v1 into w0 */
umov w0, v1.s[2]

/* Insert w0 into lane 0 of v2 (32-bit) */
ins v2.s[0], w0
```

Example: Vector Multiply by a Selected Lane (Broadcast Lane Form)

A common pattern is multiplying by one lane from another register.

```
/* v0.4s = v1.4s * v2.s[1] */
mul v0.4s, v1.4s, v2.s[1]
```

Intrinsic Extract/Insert Equivalents

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

static inline uint32_t lane2_u32(uint32x4_t v) { return vgetq_lane_u32(v,
↪  2); }
static inline uint32x4_t set0_u32(uint32x4_t v, uint32_t x) { return
↪  vsetq_lane_u32(x, v, 0); }
```

Lane Order and Memory Layout

For contiguous arrays on common little-endian AArch64 systems, lane 0 corresponds to the lowest-addressed element in a vector loaded by `ld1`/`vld1q`. This makes lane indexing match array indexing for unit-stride loads.

3.4 Alignment Constraints

Alignment impacts two different properties:

- **Legality**: whether an access is permitted by the architecture and OS mapping rules.
- **Performance**: whether an access is efficient (extra cycles, split transactions, cache-line crossing).

In practice, modern AArch64 systems typically permit unaligned loads/stores for normal memory, but performance can still degrade when:

- a vector load crosses a cache-line boundary,
- the address is frequently misaligned in a tight loop,
- the data layout triggers extra micro-operations in the load/store pipeline.

Example: Canonical 16-Byte Vector Load/Store

```
/* Load 16 bytes into v0 and store back */
ld1 {v0.16b}, [x0]
st1 {v0.16b}, [x1]
```

Example: Align-Aware Loop Strategy

Common strategy:

- short scalar prologue until a chosen pointer is aligned,
- pure vector main loop,
- scalar tail.

```
#include <arm_neon.h>
#include <cstddef>
#include <cstdint>

void add_u8_align_aware(uint8_t* dst, const uint8_t* a, const uint8_t* b,
 ↪  std::size_t n)
{
    std::size_t i = 0;

    /* Prologue: align dst to 16 bytes when possible */
    while (i < n && ((reinterpret_cast<std::uintptr_t>(dst + i) & 15u) !=
     ↪  0u)) {
        dst[i] = (uint8_t)(a[i] + b[i]);
        ++i;
    }

    /* Main vector loop */
    for (; i + 16 <= n; i += 16) {
        uint8x16_t va = vld1q_u8(a + i);
        uint8x16_t vb = vld1q_u8(b + i);
        vst1q_u8(dst + i, vaddq_u8(va, vb));
    }

    /* Tail */
    for (; i < n; ++i) {
        dst[i] = (uint8_t)(a[i] + b[i]);
    }
}
```

Example: Data Declaration with 16-Byte Alignment

```
#include <cstdint>
```

```
#if defined(_MSC_VER)
#  define AL16 __declspec(align(16))
#else
#  define AL16 __attribute__((aligned(16)))
#endif

AL16 std::uint8_t buf[1024];
```

Practical Rules of Thumb

- Prefer **16-byte alignment** for buffers processed heavily with 128-bit vectors.
- Measure bandwidth: if the loop is memory-bound, reducing cache-line splits can matter more than instruction tweaks.
- Use vector loads/stores in the main loop and keep the tail correct and simple.

Chapter 4

Load and Store Instructions

4.1 Basic Vector Loads and Stores

The fundamental NEON memory operations on AArch64 are the **single-structure** load/store forms: `ld1` and `st1`. These move a contiguous block of bytes between memory and a vector register view.

Load/Store One 128-bit Vector

```
/* Load 16 bytes into v0 */
ld1 {v0.16b}, [x0]

/* Store 16 bytes from v0 */
st1 {v0.16b}, [x1]
```

Post-indexed Address Update

Post-indexing is the standard way to advance pointers in a tight loop.

```
/* x0 points to src, x1 points to dst */
ld1 {v0.16b}, [x0], #16
st1 {v0.16b}, [x1], #16
```

Loop Skeleton: Load, Compute, Store

```
/* x0=dst, x1=src, x2=n_bytes (multiple of 16) */
1:
    ld1 {v0.16b}, [x1], #16
    /* compute: example XOR with a mask vector v1 */
    eor v0.16b, v0.16b, v1.16b
    st1 {v0.16b}, [x0], #16
    subs x2, x2, #16
    b.ne 1b
```

Intrinsic Equivalents (Portable and Inspectable)

```
#include <arm_neon.h>
#include <cstddef>
#include <cstdint>

void xor_u8(uint8_t* dst, const uint8_t* src, uint8x16_t mask, std::size_t
 ↪  n)
{
    std::size_t i = 0;
    for (; i + 16 <= n; i += 16) {
        uint8x16_t v = vld1q_u8(src + i);
        v = veorq_u8(v, mask);
        vst1q_u8(dst + i, v);
    }
    for (; i < n; ++i) dst[i] = (uint8_t)(src[i] ^ vgetq_lane_u8(mask, 0));
}
```

4.2 Aligned vs Unaligned Access

On AArch64, unaligned accesses to normal memory are typically **architecturally permitted** for regular loads/stores, including `ld1`/`st1`. However:

- **Performance** can degrade when loads frequently cross cache-line boundaries.
- Some **microarchitectures** handle misalignment with extra internal work.
- Some **mappings** (device memory, special regions) may impose stricter constraints.

Practical Alignment Target

For NEON 128-bit vectors, the practical target is **16-byte alignment** for the hottest buffers.

Example: Alignment-Aware Loop (Scalar Prologue + Vector Body)

```
#include <arm_neon.h>
#include <cstddef>
#include <cstdint>

void add_u32_align_aware(uint32_t* dst, const uint32_t* a, const uint32_t*
 ↪ b, std::size_t n)
{
    std::size_t i = 0;

    /* Align dst to 16 bytes (4 x u32 per vector) */
    while (i < n && ((reinterpret_cast<std::uintptr_t>(dst + i) & 15u) !=
     ↪ 0u)) {
        dst[i] = a[i] + b[i];
        ++i;
    }
    for (; i + 4 <= n; i += 4) {
        uint32x4_t va = vld1q_u32(a + i);
        uint32x4_t vb = vld1q_u32(b + i);
        vst1q_u32(dst + i, vaddq_u32(va, vb));
    }

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

Cache-Line Crossing Mental Model (Why Misalignment Can Hurt)

If a 16-byte load starts near the end of a cache line, it may require two line fetches internally. That can increase latency and consume extra bandwidth in tight streaming loops. The cost is workload- and core-dependent, so treat alignment as a **measurable optimization**, not a guaranteed win.

4.3 Structure Loads and Stores

NEON provides structure load/store forms that move **multiple registers** using **interleaved** memory layouts. These are designed for “array-of-structures” (AoS) data or packed pixel formats.
The families are:

- `ld1/st1` with multiple registers in a list
- `ld2/st2`, `ld3/st3`, `ld4/st4` for 2/3/4-way structures

Example: Load 2-Vector Structure (De-interleave Two Streams)

If memory is laid out as: `a0 b0 a1 b1 a2 b2 ...` (interleaved), then: `ld2` splits it into two vectors (one for all `a`’s and one for all `b`’s).

```
/* x0 points to interleaved bytes: a0 b0 a1 b1 ... */
ld2 {v0.16b, v1.16b}, [x0]
/* v0 = [a0 a1 a2 ...], v1 = [b0 b1 b2 ...] */
```

Example: Store 2-Vector Structure (Interleave Two Streams)

```
/* Store interleaved: a0 b0 a1 b1 ... */
st2 {v0.16b, v1.16b}, [x0]
```

Pixel-Style Example: RGBA (4-way Interleave)

For RGBA bytes in memory: `R0 G0 B0 A0 R1 G1 B1 A1 ...`

```
/* Load and de-interleave RGBA into 4 vectors */
ld4 {v0.16b, v1.16b, v2.16b, v3.16b}, [x0]
/* v0=R, v1=G, v2=B, v3=A */
```

Intrinsic Equivalents

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

/* De-interleave two streams of bytes */
uint8x16x2_t deint2_u8(const uint8_t* p) { return vld2q_u8(p); }

/* De-interleave RGBA bytes */
uint8x16x4_t deint4_u8(const uint8_t* p) { return vld4q_u8(p); }
```

4.4 Interleaving and De-interleaving

Interleaving transforms **structure-of-arrays** (SoA) into **array-of-structures** (AoS), and de-interleaving does the reverse. NEON supports both directions efficiently:

- **Memory-side** using `ld2/ld3/ld4` and `st2/st3/st4`
- **Register-side** using permutation instructions (zip/unzip, transpose) when data is already in registers

De-interleave in Memory (Recommended When Source Is Interleaved)

Example: split interleaved stereo audio (L,R) bytes into two streams:

```
/* x0 = interleaved LRLR... */
ld2 {v0.16b, v1.16b}, [x0], #32
/* v0 = L samples, v1 = R samples */
```

Interleave in Memory (Recommended When Destination Is Interleaved)

```
/* v0 = L samples, v1 = R samples */
st2 {v0.16b, v1.16b}, [x0], #32
```

Register-Side Interleave (Zip) and De-interleave (Unzip)

When data is already in registers, you can reorganize lanes:

```
/* Interleave lanes: zip1/zip2 produce low/high interleavings */
zip1 v0.16b, v1.16b, v2.16b
zip2 v3.16b, v1.16b, v2.16b
```

```
/* De-interleave lanes: uzp1/uzp2 extract even/odd lanes */
uzp1 v4.16b, v0.16b, v3.16b
uzp2 v5.16b, v0.16b, v3.16b
```

Concrete Example: Convert AoS RGBA to SoA R,G,B,A

If memory is RGBA interleaved, `ld4` directly produces SoA vectors:

```
/* x0 = RGBA RGBA ... */
ld4 {v0.16b, v1.16b, v2.16b, v3.16b}, [x0]
/* v0=R, v1=G, v2=B, v3=A */
```

Concrete Example: Convert SoA Back to AoS

```
/* v0=R, v1=G, v2=B, v3=A */
st4 {v0.16b, v1.16b, v2.16b, v3.16b}, [x0]
/* memory becomes RGBA RGBA ... */
```

Practical Rules of Thumb

- Use `ld2/ld3/ld4` and `st2/st3/st4` when your **memory layout is interleaved**.
- Use zip/unzip/transpose when you must **repack data already loaded into registers**.
- Prefer contiguous `ld1/st1` for SoA layouts to maximize streaming efficiency.
- Always benchmark: memory layout choices can dominate the performance of SIMD kernels.

Chapter 5

Arithmetic and Logical Operations

5.1 Integer Arithmetic

NEON integer arithmetic operates lane-wise on packed elements. The element size is selected by the view suffix (`.16b/.8h/.4s/.2d`). Integer operations are modular by default (wrap on overflow) unless a saturating form is used.

Core Integer Operations

- Add/Sub: `add`, `sub`
- Multiply: `mul` (lane-wise)
- Multiply-Accumulate: `mla`, `mls`
- Min/Max (signed/unsigned): `smin/smax`, `umin/umax`
- Absolute difference: `abd`, `uabd` (and widening variants)

Example: Add/Sub on Different Element Widths

```
/* byte add: v0[i] = v1[i] + v2[i] */
add v0.16b, v1.16b, v2.16b

/* 16-bit subtract */
sub v3.8h,  v4.8h,  v5.8h

/* 32-bit add */
add v6.4s,  v7.4s,  v8.4s
```

Example: Multiply and Multiply-Accumulate (Integer)

```
/* v0.4s = v1.4s * v2.4s */
mul v0.4s, v1.4s, v2.4s

/* v3.4s = v3.4s + (v4.4s * v5.4s) */
mla v3.4s, v4.4s, v5.4s

/* v6.4s = v6.4s - (v7.4s * v8.4s) */
mls v6.4s, v7.4s, v8.4s
```

Example: Min/Max (Signed vs Unsigned)

```
/* signed min/max */
smin v0.8h, v1.8h, v2.8h
smax v3.8h, v4.8h, v5.8h

/* unsigned min/max */
umin v6.16b, v7.16b, v8.16b
```

```
umax v9.16b, v10.16b, v11.16b
```

Intrinsic Example: Vector Add and Multiply

```
#include <arm_neon.h>

static inline uint32x4_t add_u32(uint32x4_t a, uint32x4_t b) { return
↪  vaddq_u32(a, b); }
static inline int16x8_t  mul_s16(int16x8_t a, int16x8_t b)   { return
↪  vmulq_s16(a, b); }
```

5.2 Floating-Point Arithmetic

Floating-point SIMD uses `f*` mnemonics and follows IEEE-style semantics for the supported element types. Common vector shapes are:

- `vN.4s`: 4 lanes of FP32
- `vN.2d`: 2 lanes of FP64

Core FP Operations

- Add/Sub/Mul: `fadd`, `fsub`, `fmul`
- Fused multiply-add: `fmla` (and lane forms)
- Min/Max: `fmin`, `fmax` (and variants)
- Reciprocal/sqrt approximations: often exist as separate instructions on many cores; treat as performance tools requiring error analysis

Example: FP32 Add/Mul/FMA

```
/* v0.4s = v1.4s + v2.4s */
fadd v0.4s, v1.4s, v2.4s

/* v3.4s = v4.4s * v5.4s */
fmul v3.4s, v4.4s, v5.4s

/* v6.4s = v6.4s + (v7.4s * v8.4s) */
fmla v6.4s, v7.4s, v8.4s
```

Example: FMA-Based Dot Fragment (Accumulate 4 Lanes)

```
#include <arm_neon.h>

static inline float32x4_t dot_step(float32x4_t acc, const float* a, const
↪   float* b)
{
    float32x4_t va = vld1q_f32(a);
    float32x4_t vb = vld1q_f32(b);
    return vfmaq_f32(acc, va, vb);
}
```

Floating-Point Note: Reduction Order

Vectorization changes the accumulation order compared to scalar loops and may change last-bit results. When numerical reproducibility matters, use controlled reductions and validate acceptable error bounds.

5.3 Saturating and Widening Instructions

Saturating and widening instructions exist because many real kernels naturally overflow narrow types. NEON provides systematic families for:

- **Saturating add/sub**: clamp on overflow instead of wrapping
- **Widening multiply/accumulate**: compute into a larger element type
- **Narrowing with saturation**: pack results back to smaller types safely

Saturating Add/Sub

```
/* Unsigned saturating add */
uqadd v0.16b, v1.16b, v2.16b

/* Signed saturating add */
sqadd v3.8h,  v4.8h,  v5.8h

/* Unsigned saturating subtract */
uqsub v6.8h,  v7.8h,  v8.8h

/* Signed saturating subtract */
sqsub v9.4s,  v10.4s, v11.4s
```

Widening Multiply: 8-bit → 16-bit

```
/* Unsigned widening multiply:
   v0.8h[i] = (uint16_t)v1.8b[i] * (uint16_t)v2.8b[i] */
umull v0.8h, v1.8b, v2.8b
```

```
/* Signed widening multiply */
smull v3.8h, v4.8b, v5.8b
```

Widening Multiply-Accumulate (Common in Filters)

```
/* Unsigned widening multiply-accumulate:
   v0.8h += v1.8b * v2.8b */
umlal v0.8h, v1.8b, v2.8b
```

Narrowing With Saturation: 16-bit → 8-bit

```
/* Signed saturating narrow:
   v0.8b = sat_s8(v1.8h) */
sqxtn v0.8b, v1.8h

/* Unsigned saturating narrow:
   v2.8b = sat_u8(v3.8h) */
uqxtn v2.8b, v3.8h
```

Intrinsic Example: Saturating Narrow After Accumulation

```
#include <arm_neon.h>

/* Pack 16-bit signed values into 8-bit signed with saturation */
static inline int8x8_t pack_sat_s8(int16x8_t x)
{
    return vqmovn_s16(x);
}
```

5.4 Logical and Bitwise Operations

Bitwise operations are among the most efficient NEON instructions and are essential for masks, blending, packing, and many crypto-like primitives. They are lane-wise on the chosen element view, but conceptually operate on the raw bits.

Core Bitwise Operations

- AND/OR/XOR: `and`, `orr`, `eor`
- Bit clear: `bic` (AND with NOT)
- Not: `mvn` (bitwise NOT)
- Shifts: `shl` (left), `sshr/ushr` (right arithmetic/logical)
- Select (bitwise blend): `bsl` (bitwise select using mask)

Example: AND/OR/XOR and NOT

```
/* v0 = v1 & v2 */
and v0.16b, v1.16b, v2.16b

/* v3 = v4 | v5 */
orr v3.16b, v4.16b, v5.16b

/* v6 = v7 ^ v8 */
eor v6.16b, v7.16b, v8.16b

/* v9 = ~v9 */
mvn v9.16b, v9.16b
```

Example: Clear Bits (v0 = v1 & ˜v2)

```
bic v0.16b, v1.16b, v2.16b
```

Example: Shifts (Element-Width Aware)

```
/* Shift left by 1 bit within each 32-bit lane */
shl v0.4s, v1.4s, #1

/* Unsigned shift right by 3 bits within each 16-bit lane */
ushr v2.8h, v3.8h, #3

/* Signed arithmetic shift right by 2 bits within each 32-bit lane */
sshr v4.4s, v5.4s, #2
```

Example: Masked Select (Blend)

`bsl` selects bits from two vectors using a mask:

$$\text{dst} = (\text{mask}\&\text{a}) \mid (\sim \text{mask}\&\text{b})$$

```
/* v0 = (v0 & v1) | (~v0 & v2)   (mask in v0) */
bsl v0.16b, v1.16b, v2.16b
```

Intrinsic Example: Compare + Select

```
#include <arm_neon.h>

/* dst = (a > b) ? a : b  for signed 32-bit lanes */
static inline int32x4_t max_s32(int32x4_t a, int32x4_t b)
{
    uint32x4_t m = vcgtq_s32(a, b);     /* all-ones where true */
    return vbslq_s32(m, a, b);          /* select per-lane */
}
```

Practical Rules of Thumb

- Prefer **FMA** (`fmla`/`vfmaq`) in numeric kernels: it improves throughput and reduces rounding steps.
- Use **widening** for intermediate results, then **saturating narrow** for safe packing.
- Use bitwise ops + masks for branchless selection and packing/unpacking pipelines.

Chapter 6

Data Rearrangement and Permutation

6.1 Zip, Unzip, and Transpose

Rearrangement instructions are used when data is already loaded into registers but needs a different layout for efficient computation. NEON provides systematic families for:

- **Zip** (`zip1`, `zip2`): interleave lanes from two vectors
- **Unzip** (`uzp1`, `uzp2`): de-interleave lanes (extract even/odd lanes)
- **Transpose** (`trn1`, `trn2`): swap/interleave within element-size groups

These are building blocks for SoA↔AoS transforms, matrix transposes, channel shuffles, and bit-sliced layouts.

Zip: Interleave Lanes

```
/* zip1: low half interleave, zip2: high half interleave */
zip1 v0.16b, v1.16b, v2.16b
zip2 v3.16b, v1.16b, v2.16b
```

Unzip: Extract Even/Odd Lanes

```
/* uzp1: even lanes, uzp2: odd lanes */
uzp1 v4.16b, v1.16b, v2.16b
uzp2 v5.16b, v1.16b, v2.16b
```

Transpose: Pairwise Interleave at Element Granularity

```
/* transpose 32-bit lanes between two vectors */
trn1 v0.4s, v1.4s, v2.4s
trn2 v3.4s, v1.4s, v2.4s
```

Concrete Example: SoA $\rightarrow$ AoS for Two Streams

Assume: `v1 = [a0 a1 a2 ...]`, `v2 = [b0 b1 b2 ...]` (byte lanes). Goal: `[a0 b0 a1 b1 ...]`.

```
/* Interleave bytes from v1 and v2 */
zip1 v0.16b, v1.16b, v2.16b   /* a0 b0 a1 b1 ... up to low half */
zip2 v3.16b, v1.16b, v2.16b   /* a8 b8 a9 b9 ... up to high half */
```

Concrete Example: AoS $\rightarrow$ SoA for Two Streams

Assume: `v0 = [a0 b0 a1 b1 ...]`, `v3 = [a8 b8 a9 b9 ...]`. Recover `a` and `b` streams:

```
/* De-interleave back into separate streams */
uzp1 v1.16b, v0.16b, v3.16b   /* a0 a1 a2 ... */
uzp2 v2.16b, v0.16b, v3.16b   /* b0 b1 b2 ... */
```

Transpose Example: 4x4 Matrix of 32-bit Elements

Given rows in `v0..v3` (each `.4s`), transpose to columns using `trn` + `zip`.

```
/* Step 1: transpose within 32-bit lanes */
trn1 v4.4s, v0.4s, v1.4s
trn2 v5.4s, v0.4s, v1.4s
trn1 v6.4s, v2.4s, v3.4s
trn2 v7.4s, v2.4s, v3.4s

/* Step 2: zip 64-bit pairs to finish 4x4 transpose */
zip1 v0.2d, v4.2d, v6.2d
zip2 v2.2d, v4.2d, v6.2d
zip1 v1.2d, v5.2d, v7.2d
zip2 v3.2d, v5.2d, v7.2d
```

Intrinsic Mapping

```
#include <arm_neon.h>

/* Examples for byte lanes */
static inline uint8x16_t zip1_u8(uint8x16_t a, uint8x16_t b) { return
↪  vzip1q_u8(a, b); }
static inline uint8x16_t zip2_u8(uint8x16_t a, uint8x16_t b) { return
↪  vzip2q_u8(a, b); }
static inline uint8x16_t uzp1_u8(uint8x16_t a, uint8x16_t b) { return
↪  vuzp1q_u8(a, b); }
static inline uint8x16_t uzp2_u8(uint8x16_t a, uint8x16_t b) { return
↪  vuzp2q_u8(a, b); }
```

6.2 Extract and Insert

Extract/insert instructions move elements or byte-ranges between registers and between SIMD and general-purpose registers. They are essential for:

- handling boundaries between vector and scalar code,
- building sliding windows for filters,
- rotating/shifting vectors across register boundaries.

Extract a Lane to a GP Register

```
/* w0 = lane 2 of v1 (32-bit) */
umov w0, v1.s[2]

/* x1 = lane 1 of v2 (64-bit) */
umov x1, v2.d[1]
```

Insert a GP Register Value Into a Lane

```
/* insert w0 into lane 0 of v3 (32-bit view) */
ins v3.s[0], w0

/* insert x1 into lane 1 of v4 (64-bit view) */
ins v4.d[1], x1
```

Extract a Byte Window Across Two Registers (`ext`)

`ext` concatenates two vectors and extracts a 16-byte window starting at an immediate byte offset. This is a canonical primitive for sliding windows and byte rotations.

```
/* v0 = (v1  v2)[imm : imm+16]  where imm is byte offset 0..15 */
ext v0.16b, v1.16b, v2.16b, #4   /* shift by 4 bytes */
ext v3.16b, v1.16b, v2.16b, #12  /* shift by 12 bytes */
```

Concrete Example: Sliding Window by 1 Byte

If `v1` holds bytes 0..15 and `v2` holds bytes 16..31:

```
/* v0 becomes bytes 1..16 */
ext v0.16b, v1.16b, v2.16b, #1
```

Intrinsic Equivalents

```
#include <arm_neon.h>

static inline uint32_t get_lane2_u32(uint32x4_t v) { return
↪  vgetq_lane_u32(v, 2); }
static inline uint32x4_t set_lane0_u32(uint32x4_t v, uint32_t x) { return
↪  vsetq_lane_u32(x, v, 0); }

/* ext: byte offset within concatenation */
static inline uint8x16_t ext_u8(uint8x16_t a, uint8x16_t b, const int imm)
{
    return vextq_u8(a, b, imm);
}
```

6.3 Table Lookup Instructions

Table lookup (`tbl`) performs a byte-wise permutation using an index vector. It is a high-value instruction for:

- byte shuffles (e.g., endian swaps, packed-format conversions),

- small lookups (nibble tables, ASCII transforms),
- arbitrary reordering within one or more source vectors.

One-Vector Table Lookup

`tbl` uses indices 0..15 to select bytes from the source vector. Out-of-range indices typically yield zero.

```
/* v0[i] = v1[ index[i] ] for each byte lane i */
tbl v0.16b, {v1.16b}, v2.16b
```

Two-Vector Table Lookup (32-byte Table)

With two source vectors, the table is 32 bytes wide (v1 followed by v3). Indices 0..31 select across both vectors.

```
/* v0 = lookup from concatenated table {v1, v3} */
tbl v0.16b, {v1.16b, v3.16b}, v2.16b
```

Concrete Example: Reverse Bytes Within 16-byte Vector

Build an index vector `[15 14 ...  0]` and apply `tbl`.

```
/* v2 = indices 15..0 */
movi v2.16b, #0
/* Load indices from memory in real code, or build them once in a
↪  constant pool.
   Here is the lookup step: */
tbl v0.16b, {v1.16b}, v2.16b
```

Intrinsic Equivalent

```
#include <arm_neon.h>

static inline uint8x16_t tbl1_u8(uint8x16_t table, uint8x16_t idx)
{
    return vqtbl1q_u8(table, idx);
}
```

6.4 Shuffle Patterns

A “shuffle pattern” is a reusable rearrangement idiom. Efficient SIMD code often alternates:

- **load/de-interleave** (`ld2`/`ld4`) to get computation-friendly layout,
- **compute** with lane-wise arithmetic,
- **repack/interleave** (`zip`/`trn`/`st2`/`st4`) for output layout.

Pattern 1: Interleave Two Streams (Zip)

```
/* v0/v1 are input streams */
zip1 v2.16b, v0.16b, v1.16b
zip2 v3.16b, v0.16b, v1.16b
```

Pattern 2: De-interleave Two Streams (Unzip)

```
uzp1 v2.16b, v0.16b, v1.16b
uzp2 v3.16b, v0.16b, v1.16b
```

Pattern 3: Sliding Window for FIR-like Kernels (ext)

```
/* v0 = bytes 0..15, v1 = bytes 16..31 */
ext v2.16b, v0.16b, v1.16b, #1   /* window shifted by 1 byte */
ext v3.16b, v0.16b, v1.16b, #2   /* shifted by 2 bytes */
```

Pattern 4: Byte Permute by Index (tbl)

```
/* v0 = permute(v1) using per-lane indices in v2 */
tbl v0.16b, {v1.16b}, v2.16b
```

Pattern 5: Small Matrix Transpose (trn + zip)

```
/* 4x4 transpose for 32-bit lanes: see earlier section for full
 ↪  sequence */
trn1 v4.4s, v0.4s, v1.4s
trn2 v5.4s, v0.4s, v1.4s
trn1 v6.4s, v2.4s, v3.4s
trn2 v7.4s, v2.4s, v3.4s
zip1 v0.2d, v4.2d, v6.2d
zip2 v2.2d, v4.2d, v6.2d
zip1 v1.2d, v5.2d, v7.2d
zip2 v3.2d, v5.2d, v7.2d
```

Practical Rules of Thumb

- Prefer **memory de-interleave** (`ld2/ld4`) when the source is interleaved in memory.
- Prefer **zip/uzp/trn** when reshaping data already in registers.
- Use **ext** for sliding windows and byte rotations across vector boundaries.

- Use **tbl** when the shuffle is irregular or index-driven (but treat it as a more expensive primitive; measure).

Chapter 7

Comparisons and Vector Control

7.1 Vector Comparisons

Vector comparisons produce per-lane results. In NEON, a comparison typically yields a vector where each lane becomes:

- all-bits-ones (true), or
- all-bits-zero (false).

This representation is ideal for mask-based control without branches.

Integer Comparisons (Common Forms)

```
/* Equality / inequality-like building blocks */
cmeq v0.16b, v1.16b, v2.16b     /* mask = (v1 == v2) */
cmhi v3.8h,  v4.8h,  v5.8h      /* unsigned: (v4 > v5) */
cmhs v6.4s,  v7.4s,  v8.4s      /* unsigned: (v7 >= v8) */
cmgt v9.4s,  v10.4s, v11.4s     /* signed: (v10 > v11) */
```

```
cmge v12.2d, v13.2d, v14.2d     /* signed: (v13 >= v14) */
```

Floating-Point Comparisons

FP comparisons use `fcm*` mnemonics and produce the same all-ones/all-zeros mask style.

```
/* mask = (v1 > v2) for FP32 lanes */
fcmgt v0.4s, v1.4s, v2.4s

/* mask = (v3 == v4) for FP64 lanes */
fcmeq v5.2d, v3.2d, v4.2d
```

Intrinsic Equivalents

```
#include <arm_neon.h>

static inline uint32x4_t gt_s32(int32x4_t a, int32x4_t b) { return
↪  vcgtq_s32(a, b); }
static inline uint32x4_t ge_u32(uint32x4_t a, uint32x4_t b) { return
↪  vcgeq_u32(a, b); }
static inline uint32x4_t gt_f32(float32x4_t a, float32x4_t b) { return
↪  vcgtq_f32(a, b); }
```

7.2 Mask Generation

A “mask” in NEON is usually the result of a compare. You can also generate masks from:

- bitwise operations (and/or/xor/not),
- shifts to extract sign bits or ranges,
- arithmetic to create thresholds.

Mask From Compare Against Zero (Sign Test Pattern)

A common mask is "negative lanes" for signed integers.

```
/* mask = (v1 < 0) for signed 32-bit lanes */
cmgt v0.4s, vzr.4s, v1.4s
```

Mask From Compare Against Immediate Zero via Duplicate

When you need a non-zero constant, build it in a vector first.

```
/* v2 = broadcast(127) as bytes */
movi v2.16b, #127

/* mask = (v1 > 127) unsigned bytes */
cmhi v0.16b, v1.16b, v2.16b
```

Mask Normal Form (All-Ones / All-Zeros)

It is often useful to keep masks in an all-ones/all-zeros form and use `bsl` or bitwise blends. Avoid converting masks to 0/1 scalars unless required for final scalar decisions.

Reducing a Mask to a Scalar Decision (Any/All)

To answer questions like "any lane matches?", you must reduce. One practical approach is to reinterpret the mask as bytes and horizontally reduce.

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

/* Return 1 if any byte lane is true (non-zero mask), else 0 */
static inline int any_true_u8(uint8x16_t m)
```

```
{
    /* vmaxvq_u8 reduces across lanes to the maximum byte value */
    return (vmaxvq_u8(m) != 0);
}
```

7.3 Select and Blend Operations

Once you have a mask, you can perform conditional behavior without branches using **select/blend**. NEON provides:

- `bsl` for bitwise selection,
- compare + `bif/bit` style bit-insert patterns,
- intrinsic-level `vbslq_*` select operations.

Bitwise Select (`bsl`)

`bsl` treats the first operand as a mask:

$$\text{mask} = v0, \quad v0 = (mask\&v1) \mid (\sim mask\&v2)$$

```
/* v0 is mask; select from v1 (true) else v2 (false); result
 ↪  overwrites v0 */
bsl v0.16b, v1.16b, v2.16b
```

Example: Per-Lane Max Without Branches (Signed 32-bit)

```
/* mask = (a > b) */
cmgt v2.4s, v0.4s, v1.4s

/* select: dst = (a > b) ? a : b */
/* bsl overwrites mask register, so use v2 as mask carrier */
bsl v2.16b, v0.16b, v1.16b
/* v2 now holds the selected result */
```

Intrinsic Equivalent: Max via Compare + Select

```
#include <arm_neon.h>

static inline int32x4_t max_s32(int32x4_t a, int32x4_t b)
{
    uint32x4_t m = vcgtq_s32(a, b);      /* all-ones where a>b */
    return vbslq_s32(m, a, b);
}
```

Blend With Precomputed Masks (Common in Clamp)

Clamp is built from two comparisons and two selects:

```
#include <arm_neon.h>

/* clamp x to [lo, hi] for signed 16-bit lanes */
static inline int16x8_t clamp_s16(int16x8_t x, int16x8_t lo, int16x8_t hi)
{
    uint16x8_t m_lo = vcltq_s16(x, lo);
    x = vbslq_s16(m_lo, lo, x);

    uint16x8_t m_hi = vcgtq_s16(x, hi);
    x = vbslq_s16(m_hi, hi, x);

    return x;
}
```

7.4 Conditional SIMD Patterns

NEON does not use predicate registers like SVE. Conditional behavior is achieved via:

- **mask + select** (branchless),
- **mask + bitwise ops** (merge/clear/keep),
- **scalar control around vector loops** (prologue/tail),
- **mask reduction to scalar** when you truly need a branch.

Pattern 1: Branchless "if" Using Masked Select

Compute both candidate results and select per lane.

```
#include <arm_neon.h>

/* if (a > b) out = a - b else out = b - a  (abs diff) */
static inline uint32x4_t absdiff_u32(uint32x4_t a, uint32x4_t b)
{
    uint32x4_t gt = vcgtq_u32(a, b);
    uint32x4_t x1 = vsubq_u32(a, b);
    uint32x4_t x2 = vsubq_u32(b, a);
    return vbslq_u32(gt, x1, x2);
}
```

Pattern 2: Apply Operation Only Where Mask Is True

Example: add constant to lanes matching a condition.

```
#include <arm_neon.h>

/* x += delta where x > threshold (unsigned 16-bit) */
```

```
static inline uint16x8_t add_if_gt_u16(uint16x8_t x, uint16x8_t threshold,
 ↪  uint16x8_t delta)
{
    uint16x8_t m = vcgtq_u16(x, threshold);
    uint16x8_t y = vaddq_u16(x, delta);
    return vbslq_u16(m, y, x);
}
```

Pattern 3: Masked Merge Using AND/OR

When the transformation is simple (set/clear bits), use bitwise forms.

```
/* v0 = (v0 & mask_keep) | mask_set */
and v0.16b, v0.16b, v1.16b
orr v0.16b, v0.16b, v2.16b
```

Pattern 4: Decide to Exit Early (Reduce Mask to Scalar)

When an algorithm truly needs a scalar branch (e.g., “stop when any mismatch”), reduce the mask:

```
#include <arm_neon.h>
#include <cstddef>
#include <cstdint>

/* Return 1 if any byte differs between a and b (first 16 bytes) */
static inline int any_diff_16(const uint8_t* a, const uint8_t* b)
{
    uint8x16_t va = vld1q_u8(a);
    uint8x16_t vb = vld1q_u8(b);
    uint8x16_t m  = vceqq_u8(va, vb);   /* 0xFF where equal */
    m = vmvnq_u8(m);                    /* 0xFF where different */
    return (vmaxvq_u8(m) != 0);
}
```

Practical Rules of Thumb

- Prefer **mask + select** over per-element branching.
- Keep masks in **all-ones/all-zeros** form; it composes cleanly with bitwise logic.
- Reduce masks to scalars only when required for control flow.
- Validate performance: some patterns compute both sides of a conditional and may be more expensive than a predictable branch on certain workloads.

Chapter 8

Horizontal and Reduction Operations

8.1 Pairwise Operations

Pairwise operations combine adjacent lanes, producing partial horizontal work while keeping results in vector form. They are used to:

- reduce a vector into fewer lanes step-by-step,
- build tree-like sums/min/max,
- restructure data for later stages.

Pairwise Add (`addp`)

`addp` performs pairwise addition across lanes. It is commonly used as a reduction building block.

```
/* Pairwise add on 32-bit lanes:
   result lanes are sums of adjacent pairs */
addp v0.4s, v1.4s, v2.4s
```

Pairwise Add Within One Vector (Common Reduction Idiom)

A practical idiom is to fold a vector by repeatedly pairwise-adding it with itself.

```
/* v0.4s contains [a0 a1 a2 a3] */
addp v0.4s, v0.4s, v0.4s  /* becomes [a0+a1, a2+a3, a0+a1, a2+a3]
 ↪  (lane replication behavior is instruction-defined) */
```

In many kernels, it is clearer and safer to do explicit reductions using well-known sequences (intrinsics or well-tested assembly macros), rather than relying on compact but less readable self-pairing forms.

Pairwise Min/Max (Conceptual Pattern)

Pairwise min/max can be constructed by combining compare/select or using min/max instructions after permutation steps. The key idea: reduce candidates by halves until one lane remains.

8.2 Reductions

A reduction collapses multiple lanes into a single scalar (sum/min/max/any/all/etc.). NEON reductions are usually built from:

- lane-wise arithmetic,
- pairwise operations,
- explicit extracts, or dedicated horizontal reduction intrinsics where available.

Example: Sum of 4 FP32 Lanes (Intrinsic Reduction)

```
#include <arm_neon.h>
```

```
/* Horizontal sum of float32x4_t */
static inline float hsum_f32(float32x4_t v)
{
    float32x2_t s = vadd_f32(vget_low_f32(v), vget_high_f32(v));
    s = vpadd_f32(s, s);
    return vget_lane_f32(s, 0);
}
```

Example: Sum of 4 FP32 Lanes (Assembly Sequence)

```
/* Input: v0.4s = [a0 a1 a2 a3]
   Output: s0 = a0+a1+a2+a3 (one lane extracted) */

/* pairwise add: v1.s[0]=a0+a1, v1.s[1]=a2+a3 (other lanes may be
↪  replicated) */
addp v1.4s, v0.4s, v0.4s

/* sum the two partials: v1.s[0] = (a0+a1) + (a2+a3) */
fadd s0, s1, s1  /* NOTE: shown conceptually; prefer intrinsic
↪  reduction or explicit lane ops for portability */
```

Practical note: horizontal reductions in handwritten assembly should be implemented using a tested, explicit sequence for your chosen lane width. Intrinsics are often preferable because the compiler selects the best available sequence for the target.

Example: Max of 16 Bytes (Any/All Style via Max-Reduce)

For many boolean-like reductions, reducing via max/min is efficient:

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

```
/* Return 1 if any lane is non-zero */
static inline int any_nonzero_u8(uint8x16_t v)
{
    return (vmaxvq_u8(v) != 0);
}

/* Return 1 if all lanes are non-zero */
static inline int all_nonzero_u8(uint8x16_t v)
{
    return (vminvq_u8(v) != 0);
}
```

Example: Dot Product with Vector Accumulator + Final Reduction

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

float dot_f32_neon(const float* a, const float* b, std::size_t n)
{
    std::size_t i = 0;
    float32x4_t acc = vdupq_n_f32(0.0f);

    for (; i + 4 <= n; i += 4) {
        float32x4_t va = vld1q_f32(a + i);
        float32x4_t vb = vld1q_f32(b + i);
        acc = vfmaq_f32(acc, va, vb);
    }

    float sum = hsum_f32(acc);

    for (; i < n; ++i) sum += a[i] * b[i];
    return sum;
}
```

8.3 Accumulation Patterns

Accumulation is the most common reduction-adjacent workload. The main challenge is keeping the pipeline busy while avoiding dependency chains.

Pattern 1: Multiple Accumulators (Unroll to Hide Latency)

Using multiple independent accumulators reduces the dependency depth and increases instruction-level parallelism.

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

float dot_f32_2acc(const float* a, const float* b, std::size_t n)
{
    std::size_t i = 0;
    float32x4_t acc0 = vdupq_n_f32(0.0f);
    float32x4_t acc1 = vdupq_n_f32(0.0f);

    for (; i + 8 <= n; i += 8) {
        float32x4_t a0 = vld1q_f32(a + i);
        float32x4_t b0 = vld1q_f32(b + i);
        float32x4_t a1 = vld1q_f32(a + i + 4);
        float32x4_t b1 = vld1q_f32(b + i + 4);

        acc0 = vfmaq_f32(acc0, a0, b0);
        acc1 = vfmaq_f32(acc1, a1, b1);
    }

    float32x4_t acc = vaddq_f32(acc0, acc1);
    float sum = hsum_f32(acc);

    for (; i < n; ++i) sum += a[i] * b[i];
    return sum;
}
```

Pattern 2: Widen-Accumulate for Narrow Types

For 8-bit/16-bit inputs, accumulate in a wider type to prevent overflow and preserve precision.

```
/* Example concept: widen multiply then accumulate into 16-bit lanes
 ↪  */
umlal v0.8h, v1.8b, v2.8b
```

Pattern 3: Tree Reduction vs Linear Reduction

- **Linear** (single accumulator): simplest, but dependency chain can limit throughput.
- **Tree** (pairwise reduction): better for precision and can reduce dependency depth.

8.4 Precision Considerations

SIMD acceleration changes how floating-point sums are formed, and it often changes results in the last bits due to different association order. This is not a NEON quirk; it is a property of floating-point arithmetic.

What Changes Under Vectorization

- **Association order**: scalar loop is typically left-to-right; SIMD reduction is often pairwise.

- **Use of FMA**: fused multiply-add performs one rounding instead of two.
- **Unrolling/multiple accumulators**: changes the reduction tree and rounding points.

Practical Approaches

- For performance-centric kernels: prefer **FMA** and **multiple accumulators**, validate error bounds.
- For reproducibility: use a **fixed reduction strategy** (explicit tree), or accumulate in higher precision when acceptable.
- For integer reductions: widen early (e.g., 8-bit → 16/32-bit accumulators) to prevent overflow.

Example: More Stable Summation via Pairwise Strategy

```
#include <arm_neon.h>

/* Pairwise reduction: sum lanes with a fixed tree */
static inline float reduce_pairwise_f32(float32x4_t v)
{
    float32x2_t lo = vget_low_f32(v);
    float32x2_t hi = vget_high_f32(v);
    float32x2_t s  = vadd_f32(lo, hi);      /* [a0+a2, a1+a3] */
    s = vpadd_f32(s, s);                    /* [(a0+a2)+(a1+a3), ...] */
    return vget_lane_f32(s, 0);
}
```

Example: Integer Sum Requires Widening

Summing many bytes into an 8-bit accumulator overflows immediately. Widen first:

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

/* Sum 16 bytes into a 32-bit scalar (widen + reduce) */
static inline uint32_t sum_u8_16(uint8x16_t v)
{
    uint16x8_t lo = vpaddlq_u8(v);          /* widen + pairwise add: 8 lanes
     ↪  of u16 */
    uint32x4_t lo2 = vpaddlq_u16(lo);       /* widen + pairwise add: 4 lanes
     ↪  of u32 */
    return vaddvq_u32(lo2);                /* horizontal add u32 lanes */
}
```

Rules of Thumb

- Prefer **multiple accumulators** for throughput; reduce at the end.
- Prefer **pairwise reductions** for better numerical stability in FP.
- Use **widening** for integer accumulation to avoid overflow.
- Treat reduction choice as part of correctness: specify acceptable error bounds and test.

Chapter 9

Programming NEON

9.1 Inline Assembly Model

Inline assembly is the lowest-level way to use NEON inside C/C++ translation units. It gives maximum control over instruction selection and scheduling, but it also introduces risk:

- ABI and calling-convention requirements (register preservation rules),
- correctness under optimization (clobbers, memory ordering, aliasing),
- portability across compilers and build flags,
- maintenance cost (harder to review and evolve).

Two Practical Styles

- **Standalone `.S` files (recommended for serious assembly)**: assembled by GAS, clear ABI boundaries.
- **Inline asm blocks**: best kept small and local (micro-kernels), carefully constrained.

Minimal NEON Inline Assembly Example (GCC/Clang Style)

This example performs `dst[i..i+15] = a[i..i+15] + b[i..i+15]` for one 16-byte block.

```
#include <cstdint>

static inline void add16_u8_asm(uint8_t* dst, const uint8_t* a, const
↪   uint8_t* b)
{
    __asm__ volatile(
        "ld1 {v0.16b}, [%[pa]]\n\t"
        "ld1 {v1.16b}, [%[pb]]\n\t"
        "add v0.16b, v0.16b, v1.16b\n\t"
        "st1 {v0.16b}, [%[pd]]\n\t"
        :
        : [pd] "r"(dst), [pa] "r"(a), [pb] "r"(b)
        : "v0", "v1", "memory"
    );
}
```

Key Correctness Rules for Inline asm

- Declare every vector register you modify in the clobber list.
- Use `"memory"` clobber when the asm reads/writes memory not otherwise visible to the compiler.
- Do not assume anything about register allocation; treat your asm as a black box to the compiler.
- Prefer a separate `.S` file when the asm exceeds a few lines or requires careful ABI control.

Standalone GAS Example: A Tiny NEON Kernel Skeleton

```
/* AArch64 GAS syntax */
/* void add_u8_16(uint8_t* dst, const uint8_t* a, const uint8_t* b);
 ↪  */
/* x0=dst, x1=a, x2=b */
.global add_u8_16
add_u8_16:
    ld1 {v0.16b}, [x1]
    ld1 {v1.16b}, [x2]
    add v0.16b, v0.16b, v1.16b
    st1 {v0.16b}, [x0]
    ret
```

9.2 NEON Intrinsics

NEON intrinsics expose SIMD operations through C/C++ functions that map closely to instructions. They provide a strong practical balance:

- type-checked vector types (fewer silent mistakes),
- compiler-managed register allocation and scheduling,
- easier portability across AArch64 compilers,
- predictable mapping when inspected via generated assembly.

Basic Intrinsics Example: Load, Compute, Store

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

```
#include <cstdint>

void add_u32_intrin(uint32_t* dst, const uint32_t* a, const uint32_t* b,
↪  std::size_t n)
{
    std::size_t i = 0;
    for (; i + 4 <= n; i += 4) {
        uint32x4_t va = vld1q_u32(a + i);
        uint32x4_t vb = vld1q_u32(b + i);
        uint32x4_t vc = vaddq_u32(va, vb);
        vst1q_u32(dst + i, vc);
    }
    for (; i < n; ++i) dst[i] = a[i] + b[i];
}
```

Intrinsics for Shuffles and Reductions

Intrinsics cover zip/unzip/transpose/extract as well as reductions.

```
#include <arm_neon.h>

/* zip (interleave) */
static inline uint8x16_t zip_lo_u8(uint8x16_t a, uint8x16_t b) { return
↪  vzip1q_u8(a, b); }

/* extract a lane */
static inline uint32_t lane2_u32(uint32x4_t v) { return vgetq_lane_u32(v,
↪  2); }

/* reduce: sum of u32 lanes */
static inline uint32_t sum_u32(uint32x4_t v) { return vaddvq_u32(v); }
```

Intrinsic Hygiene Rules

- Keep vector widths explicit in the type names (`...x4`, `...x16`).
- Avoid needless lane extracts inside the hot loop; reduce at the end.
- Keep memory access patterns simple (unit stride) unless you know why you deviate.

9.3 Compiler Auto-Vectorization

Auto-vectorization means the compiler recognizes a loop as data-parallel and emits NEON instructions automatically. It can be very effective for:

- simple arithmetic loops,
- reductions with recognized idioms,
- contiguous memory patterns,
- predictable control flow.

However, it is not guaranteed. The compiler needs proof that the transformation preserves semantics:

- no hidden aliasing problems,
- no loop-carried dependencies,
- no undefined behavior,
- predictable memory alignment and bounds (or safe fallback handling).

Vectorizable Loop Example (Often Auto-Vectorizes)

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

/* dst[i] = a[i] + b[i] */
void add_auto(uint32_t* dst, const uint32_t* a, const uint32_t* b,
 ↪  std::size_t n)
{
    for (std::size_t i = 0; i < n; ++i)
        dst[i] = a[i] + b[i];
}
```

De-vectorization Traps

Common reasons vectorization fails:

- potential pointer aliasing between `dst`, `a`, `b`,
- data-dependent branches inside the loop,
- function calls inside the loop that the compiler cannot analyze,
- mixed integer sizes with implicit promotions or overflow semantics that must be preserved.

Help the Compiler (Without Lying)

When valid for your program:

- use `restrict`-style non-aliasing (or equivalent) where supported,
- keep loop bounds and indexing simple,

- separate the scalar tail explicitly when helpful,
- keep data aligned and contiguous.

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

#if defined(__GNUC__)  defined(__clang__)
#  define RESTRICT __restrict__
#else
#  define RESTRICT
#endif

void add_auto_restrict(uint32_t* RESTRICT dst,
                       const uint32_t* RESTRICT a,
                       const uint32_t* RESTRICT b,
                       std::size_t n)
{
    for (std::size_t i = 0; i < n; ++i)
        dst[i] = a[i] + b[i];
}
```

Verification Requirement

Never assume auto-vectorization happened. Verify by:

- inspecting compiler output (assembly),
- using compiler vectorization reports when available,
- benchmarking on the target core.

9.4 Choosing the Right Approach

You have three viable programming models:

- **Auto-vectorization**: lowest effort, best for simple loops, must be verified.
- **Intrinsics**: best general choice for performance-critical code with maintainability.
- **Handwritten assembly**: maximum control, best for tight micro-kernels and special instruction sequences.

Decision Table (Practical)

Choose **auto-vectorization** when:

- the loop is simple and regular,
- performance is important but not at micro-kernel level,
- portability and maintainability dominate.

Choose **intrinsics** when:

- you need predictable SIMD behavior and layout control,
- you need specific shuffles/reductions,
- you want performance with readable, reviewable code.

Choose **assembly** when:

- you must control instruction scheduling precisely,
- you rely on specific instruction sequences not reliably emitted by compilers,
- you are writing a small kernel used everywhere (amortize maintenance).

Extensive Example: Same Kernel in Three Models

(1) Auto-vectorization candidate

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

void add_u8_auto(uint8_t* dst, const uint8_t* a, const uint8_t* b,
 ↪  std::size_t n)
{
    for (std::size_t i = 0; i < n; ++i)
        dst[i] = (uint8_t)(a[i] + b[i]);
}
```

(2) Intrinsics

```
#include <arm_neon.h>
#include <cstddef>
#include <cstdint>

void add_u8_intrin(uint8_t* dst, const uint8_t* a, const uint8_t* b,
 ↪  std::size_t n)
{
    std::size_t i = 0;
    for (; i + 16 <= n; i += 16) {
        uint8x16_t va = vld1q_u8(a + i);
        uint8x16_t vb = vld1q_u8(b + i);
        vst1q_u8(dst + i, vaddq_u8(va, vb));
    }
    for (; i < n; ++i) dst[i] = (uint8_t)(a[i] + b[i]);
}
```

(3) Assembly micro-kernel (one block)

```
/* x0=dst, x1=a, x2=b */
ld1 {v0.16b}, [x1]
ld1 {v1.16b}, [x2]
add v0.16b, v0.16b, v1.16b
st1 {v0.16b}, [x0]
```

Rule of Thumb for This Booklet

Start with intrinsics for correctness and clarity. Use assembly only when:

- profiling proves a hotspot,
- you can isolate the kernel behind a stable interface,
- you can validate ABI correctness and maintain tests.

Chapter 10

Performance and Optimization

10.1 Throughput vs Latency

Performance tuning starts with separating two limits:

- **Latency-bound**: each iteration depends on the previous result (tight dependency chain). Speed is limited by the latency of the critical instructions.
- **Throughput-bound**: there is enough independent work to keep execution units busy. Speed is limited by how many operations can be issued/retired per cycle.

NEON kernels often become throughput-bound when:

- the loop body contains independent vector ops (multiple accumulators, multiple streams),
- loads/stores are regular and can be overlapped with compute.

They become latency-bound when:

- there is a single accumulator updated every iteration,
- the kernel uses expensive shuffles in the critical path,
- the kernel is dominated by dependent load-to-use chains.

Example: Latency-Bound Pattern (Single Accumulator)

```
/* Each iteration depends on previous v0 result */
1:
    fmla v0.4s, v1.4s, v2.4s    /* v0 = v0 + v1*v2 */
    subs x3, x3, #1
    b.ne 1b
```

Example: Throughput-Oriented Pattern (Multiple Accumulators)

Independent accumulators reduce dependency depth and increase ILP.

```
/* v0 and v3 are independent accumulators */
fmla v0.4s, v1.4s, v2.4s
fmla v3.4s, v4.4s, v5.4s
```

Concrete Example: Dot Product With Two Accumulators

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

static inline float hsum_f32(float32x4_t v)
{
    float32x2_t s = vadd_f32(vget_low_f32(v), vget_high_f32(v));
    s = vpadd_f32(s, s);
    return vget_lane_f32(s, 0);
}

float dot_f32_2acc(const float* a, const float* b, std::size_t n)
{
    std::size_t i = 0;
    float32x4_t acc0 = vdupq_n_f32(0.0f);
    float32x4_t acc1 = vdupq_n_f32(0.0f);

    for (; i + 8 <= n; i += 8) {
        float32x4_t a0 = vld1q_f32(a + i);
        float32x4_t b0 = vld1q_f32(b + i);
        float32x4_t a1 = vld1q_f32(a + i + 4);
        float32x4_t b1 = vld1q_f32(b + i + 4);

        acc0 = vfmaq_f32(acc0, a0, b0);
        acc1 = vfmaq_f32(acc1, a1, b1);
    }

    float sum = hsum_f32(vaddq_f32(acc0, acc1));

    for (; i < n; ++i) sum += a[i] * b[i];
    return sum;
}
```

10.2 Instruction Scheduling

Instruction scheduling is about arranging operations so the core can:

- overlap load latency with arithmetic,
- avoid long dependent chains,
- keep pipelines full (especially FMA/ALU and load/store).

Rule: Hide Load-to-Use Latency

Start loads early, do independent work, then consume loaded data.

Example: Software Pipelining Skeleton

```
/* x0=dst, x1=a, x2=b, x3=n_bytes (multiple of 16) */
/* Preload first block */
ld1 {v0.16b}, [x1], #16
ld1 {v1.16b}, [x2], #16

1:
    /* Preload next block (early) */
    ld1 {v2.16b}, [x1], #16
    ld1 {v3.16b}, [x2], #16

    /* Compute on previous block */
    add v0.16b, v0.16b, v1.16b
    st1 {v0.16b}, [x0], #16

    /* Rotate registers for next iteration */
    mov v0.16b, v2.16b
    mov v1.16b, v3.16b

    subs x3, x3, #16
    b.ne 1b
```

Note on Register-to-Register Moves

The `mov vX.16b, vY.16b` shuffle shown above is a conceptual “rename”. In practice, you can often avoid explicit moves by using a different register assignment strategy in the loop, or by unrolling.

Rule: Put Shuffles Off the Critical Path

Shuffles (`tbl`, heavy zip/trn chains, `ext`) can be expensive. When possible:

- do shuffles once per many arithmetic ops,
- combine shuffles with memory layout transforms (`ld2/ld4`) instead of register shuffles.

10.3 Register Pressure

NEON provides 32 vector registers, but register pressure still matters because:

- unrolling increases live values,
- AoS$\leftrightarrow$SoA transforms require temporary registers,
- multiple accumulators increase live state,
- compilers may spill vectors to the stack if live ranges are large.

Symptoms of Excess Register Pressure

- unexpected stack traffic (vector spills/reloads),
- reduced unrolling by the compiler,
- worse performance even though the algorithm is “more SIMD”.

Example: High Register Pressure Pattern (Too Many Live Vectors)

```
#include <arm_neon.h>

/* Illustrative only: many live temporaries can trigger spills */
static inline uint8x16_t heavy_shuffle(uint8x16_t a, uint8x16_t b,
↪  uint8x16_t c, uint8x16_t d)
{
    uint8x16_t t0 = vzip1q_u8(a, b);
    uint8x16_t t1 = vzip2q_u8(a, b);
    uint8x16_t t2 = vzip1q_u8(c, d);
    uint8x16_t t3 = vzip2q_u8(c, d);
    uint8x16_t t4 = vuzp1q_u8(t0, t2);
    uint8x16_t t5 = vuzp2q_u8(t1, t3);
    return veorq_u8(t4, t5);
}
```

Mitigations

- reduce unroll factor if spills appear,
- shorten live ranges (compute and store sooner),
- reuse registers intentionally (overwrite temporaries once no longer needed),
- prefer memory-side interleave/de-interleave (`ld2/ld4`) over long shuffle chains.

10.4 Common NEON Pitfalls

Pitfall 1: Assuming NEON Always Speeds Things Up

If the loop is memory-bandwidth limited, SIMD may not scale. Fixes:

- fuse passes over memory,
- improve locality (blocking),
- reduce loads/stores, avoid redundant reads.

Pitfall 2: Partial Register Writes Then Full-Width Use

Writing only dN and later using vN can leak stale upper lanes.

```
/* BUG: only low 64 bits initialized */
movi d0, #0

/* Later uses full vector lanes (upper half may be stale) */
add v1.2d, v0.2d, v2.2d

/* FIX: initialize full 128 bits */
movi v0.16b, #0
add  v1.2d,  v0.2d,  v2.2d
```

Pitfall 3: Incorrect Tail Handling

Vector loops must not read/write past the end. Always handle tails explicitly.

```
#include <arm_neon.h>
#include <cstddef>
#include <cstdint>

void add_u32_tail_safe(uint32_t* dst, const uint32_t* a, const uint32_t* b,
↪   std::size_t n)
{
    std::size_t i = 0;
    for (; i + 4 <= n; i += 4) {
        vst1q_u32(dst + i, vaddq_u32(vld1q_u32(a + i), vld1q_u32(b + i)));
    }
    for (; i < n; ++i) dst[i] = a[i] + b[i];
}
```

Pitfall 4: Expensive Shuffles in Hot Loops

Frequent `tbl` or long zip/trn chains can dominate cost. Fixes:

- change data layout (SoA),
- use `ld2/ld4` to de-interleave at load time,
- move shuffles out of the inner loop, or amortize them.

Pitfall 5: Misunderstanding Signed vs Unsigned Variants

`cmgt` vs `cmhi`, `smin` vs `umin`, saturating signed vs unsigned forms. A wrong variant can silently change results.

```
/* unsigned compare (a > b) */
cmhi v0.8h, v1.8h, v2.8h

/* signed compare (a > b) */
cmgt v3.8h, v4.8h, v5.8h
```

Pitfall 6: Assuming Alignment Is Irrelevant

Unaligned loads are often legal, but can be slower due to cache-line splits. Prefer 16-byte alignment for hot buffers when practical.

Pitfall 7: Not Verifying What the Compiler Did

Auto-vectorization and intrinsic mapping must be verified by inspecting generated assembly and measuring on target hardware.

Performance Checklist (Concise)

- Identify whether the kernel is **latency-bound**, **throughput-bound**, or **memory-bound**.
- Use **multiple accumulators** to break dependency chains in reductions.
- Schedule loads early and keep shuffles off the critical path.
- Watch **register pressure** and avoid spills.
- Keep tails correct and measure on the real target core.

Appendices

Appendix A — NEON Instruction Classification

This appendix classifies commonly used AArch64 NEON (Advanced SIMD) instructions by **role in SIMD kernels**. The goal is practical: when reading or writing NEON, you should instantly recognize whether an instruction is moving data, computing, rearranging, or reducing.

Load/Store

Purpose: Move contiguous or interleaved data between memory and vector registers. These instructions define the kernel's memory behavior and often dominate performance in bandwidth-bound loops.

- **Contiguous loads/stores:** `ld1`, `st1`
- **Structured interleaved loads/stores:** `ld2/ld3/ld4`, `st2/st3/st4`

Canonical Forms

```
/* Contiguous 16-byte load/store */
ld1 {v0.16b}, [x0]
st1 {v0.16b}, [x1]
```

```
/* Post-indexed streaming */
ld1 {v1.4s}, [x2], #16
st1 {v1.4s}, [x3], #16

/* De-interleave / interleave memory layouts */
ld2 {v2.16b, v3.16b}, [x4]                  /* a0 b0 a1 b1 ... ->
 ↪  v2=a*, v3=b* */
st2 {v2.16b, v3.16b}, [x5]                  /* v2/v3 -> interleaved
 ↪  output */

ld4 {v4.16b, v5.16b, v6.16b, v7.16b}, [x6] /* RGBA... -> R,G,B,A
 ↪  vectors */
st4 {v4.16b, v5.16b, v6.16b, v7.16b}, [x7] /* R,G,B,A -> RGBA... */
```

Kernel Snippet: Streaming Add

```
/* x0=dst, x1=a, x2=b, x3=n_bytes (multiple of 16) */
1:
    ld1 {v0.16b}, [x1], #16
    ld1 {v1.16b}, [x2], #16
    add v0.16b, v0.16b, v1.16b
    st1 {v0.16b}, [x0], #16
    subs x3, x3, #16
    b.ne 1b
```

Arithmetic

Purpose: Lane-wise computation on packed elements (integer or floating-point). Arithmetic instructions are the “work” in the work-per-byte ratio.

- **Integer add/sub/mul:** `add`, `sub`, `mul`, `mla`, `mls`
- **FP add/mul/FMA:** `fadd`, `fmul`, `fmla`
- **Min/max/absdiff:** `smin/smax`, `umin/umax`, `abd/uabd`
- **Saturating/widening/narrowing:** `sqadd/uqadd`, `smull/umull`, `sqxtn/uqxtn`

Representative Arithmetic Forms

```
/* Integer arithmetic */
add v0.4s,  v1.4s,  v2.4s
sub v3.8h,  v4.8h,  v5.8h
mul v6.4s,  v7.4s,  v8.4s
mla v9.4s,  v10.4s, v11.4s

/* Floating-point arithmetic */
fadd v0.4s, v1.4s, v2.4s
fmul v3.4s, v4.4s, v5.4s
fmla v6.4s, v7.4s, v8.4s

/* Saturating and widening */
uqadd v0.16b, v1.16b, v2.16b
sqadd v3.8h,  v4.8h,  v5.8h
umull v6.8h,  v7.8b,  v8.8b
sqxtn v9.8b,  v10.8h
```

Intrinsic Mapping Example

```
#include <arm_neon.h>
```

```
static inline uint32x4_t add_u32(uint32x4_t a, uint32x4_t b) { return
↪  vaddq_u32(a, b); }
static inline float32x4_t fma_f32(float32x4_t acc, float32x4_t a,
↪  float32x4_t b) { return vfmaq_f32(acc, a, b); }
static inline uint8x8_t sat_narrow_s16_to_s8(int16x8_t x) { return
↪  vqmovn_s16(x); }
```

Permutation

Purpose: Rearrange lanes/bytes for layout conversion, sliding windows, transpose, and irregular shuffles. Permutation often determines whether arithmetic can run at full speed.

- **Interleave/de-interleave lanes:** `zip1/zip2`, `uzp1/uzp2`
- **Transpose within element groups:** `trn1/trn2`
- **Extract window across vectors:** `ext`
- **Byte table lookup:** `tbl` (index-driven shuffle)
- **Lane move:** `ins`, `umov`

Representative Permutation Forms

```
/* Zip/unzip */
zip1 v0.16b, v1.16b, v2.16b
zip2 v3.16b, v1.16b, v2.16b
uzp1 v4.16b, v1.16b, v2.16b
uzp2 v5.16b, v1.16b, v2.16b

/* Transpose (32-bit lanes) */
trn1 v6.4s, v7.4s, v8.4s
trn2 v9.4s, v7.4s, v8.4s
```

```
/* Sliding window / rotation */
ext v10.16b, v11.16b, v12.16b, #1

/* Table lookup shuffle */
tbl v13.16b, {v14.16b}, v15.16b

/* Lane extract/insert */
umov w0, v0.s[2]
ins  v1.s[0], w0
```

Canonical Shuffle Pattern: AoS ↔ SoA

```
/* Memory de-interleave (preferred when input is interleaved) */
ld2 {v0.16b, v1.16b}, [x0]     /* LRLR... -> v0=L, v1=R */

/* Register interleave (when already in regs) */
zip1 v2.16b, v0.16b, v1.16b
zip2 v3.16b, v0.16b, v1.16b
```

Reduction

Purpose: Collapse multiple lanes into fewer lanes or into a scalar result. Reductions appear in dot products, sums, norms, min/max, any/all checks, and histogram-like kernels.

- **Pairwise operations:** `addp` (and FP pairwise variants)
- **Widen+pairwise add for integer reduction:** `uaddlp/saddlp` families via intrinsics
- **Horizontal reductions (intrinsics):** `vaddvq`, `vmaxvq`, `vminvq`

Representative Reduction Forms

```
/* Pairwise add (reduction building block) */
addp v0.4s, v1.4s, v2.4s
```

Reduction: Sum of 4 FP32 Lanes (Intrinsic, Fixed Tree)

```
#include <arm_neon.h>

static inline float hsum_f32(float32x4_t v)
{
    float32x2_t s = vadd_f32(vget_low_f32(v), vget_high_f32(v));
    s = vpadd_f32(s, s);
    return vget_lane_f32(s, 0);
}
```

Reduction: Any/All Style Checks on Bytes

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

static inline int any_true_u8(uint8x16_t m) { return (vmaxvq_u8(m) != 0); }
static inline int all_true_u8(uint8x16_t m) { return (vminvq_u8(m) != 0); }
```

Reduction: Safe Integer Sum Requires Widening

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

/* Sum 16 unsigned bytes into a 32-bit scalar */
static inline uint32_t sum_u8_16(uint8x16_t v)
{
    uint16x8_t w1 = vpaddlq_u8(v);       /* widen + pairwise add -> u16
     ↪  lanes */
    uint32x4_t w2 = vpaddlq_u16(w1);     /* widen + pairwise add -> u32
     ↪  lanes */
    return vaddvq_u32(w2);              /* horizontal sum -> scalar */
}
```

Classification Rule of Thumb

- If it touches **memory**: Load/Store.
- If it changes **values without moving lanes**: Arithmetic/Logical.
- If it changes **lane order or grouping**: Permutation.
- If it shrinks **many lanes to fewer**: Reduction.

Appendix B — Lane and Element Reference

This appendix is a compact reference for how a 128-bit NEON register is viewed as lanes, and which operation families are valid (and common) for each element type.

Element Sizes

AArch64 NEON uses fixed-width 128-bit vectors. The element size determines the lane view:

- `.b`: 8-bit element (byte)
- `.h`: 16-bit element (halfword)
- `.s`: 32-bit element (word / single-precision FP lane)
- `.d`: 64-bit element (doubleword / double-precision FP lane)

The View Is Part of the Instruction

The same physical `Vn` bits can be interpreted under different views. The mnemonic + suffix controls what the hardware does.

```
/* Same register index, different element views */
add  v0.16b, v1.16b, v2.16b
add  v0.8h,  v1.8h,  v2.8h
add  v0.4s,  v1.4s,  v2.4s
add  v0.2d,  v1.2d,  v2.2d

/* Floating-point view uses FP mnemonics */
fadd v3.4s,  v4.4s,  v5.4s
fadd v6.2d,  v7.2d,  v8.2d
```

Lane Counts

For a 128-bit vector:

$$\text{lane\_count} = \frac{128}{\text{element\_bits}}$$

- 8-bit (`.16b`) : 16 lanes
- 16-bit (`.8h`) : 8 lanes
- 32-bit (`.4s`) : 4 lanes
- 64-bit (`.2d`) : 2 lanes

Lane Indexing Is Zero-Based

- `.16b`: lanes `[0..15]`
- `.8h`: lanes `[0..7]`

- `.4s`: lanes `[0..3]`
- `.2d`: lanes `[0..1]`

Lane Extract/Insert Examples

```
/* Extract lanes to GP regs */
umov w0, v1.s[2]
umov x1, v2.d[1]

/* Insert GP regs into lanes */
ins v3.s[0], w0
ins v4.d[1], x1
```

Valid Operations per Type

The tables below are **operation families** that are typically valid and useful for each lane type. Not every instruction exists for every type, but these families describe what NEON is designed to do.

Integer 8-bit / 16-bit / 32-bit / 64-bit (Signed and Unsigned)

Valid and common families:

- Lane-wise add/sub: `add`, `sub`
- Lane-wise mul (mainly 16/32-bit, and some widening forms from 8-bit): `mul`, `smull/umull`, `smlal/umlal`
- Saturating arithmetic: `sqadd/uqadd`, `sqsub/uqsub`
- Min/max: `smin/smax`, `umin/umax`

- Compare (mask results): `cmeq`, `cmgt`/`cmge`, `cmhi`/`cmhs`
- Shifts: `shl`, `sshr`, `ushr`
- Bitwise ops: `and`, `orr`, `eor`, `bic`, `mvn`

Representative Integer Examples

```
/* Add/sub */
add  v0.8h,  v1.8h,  v2.8h
sub  v3.4s,  v4.4s,  v5.4s

/* Signed/unsigned compare masks */
cmgt v6.4s,  v7.4s,  v8.4s     /* signed (>) */
cmhi v9.8h,  v10.8h, v11.8h    /* unsigned (>) */

/* Saturating add */
sqadd v12.16b, v13.16b, v14.16b
uqadd v15.8h,  v16.8h,  v17.8h

/* Widening multiply (8-bit -> 16-bit) */
umull v0.8h, v1.8b, v2.8b

/* Narrow with saturation (16-bit -> 8-bit) */
sqxtn v3.8b, v4.8h

/* Bitwise and shifts */
eor  v5.16b, v5.16b, v6.16b
ushr v7.8h,  v7.8h,  #3
```

Intrinsic Family Mapping (Integer)

```
#include <arm_neon.h>

static inline uint8x16_t  add_u8(uint8x16_t a, uint8x16_t b) { return
 ↪  vaddq_u8(a, b); }
static inline int32x4_t   max_s32(int32x4_t a, int32x4_t b)
{
    uint32x4_t m = vcgtq_s32(a, b);
    return vbslq_s32(m, a, b);
}
static inline uint8x8_t   pack_sat_s16_to_s8(int16x8_t x) { return
 ↪  vqmovn_s16(x); }
```

Floating-Point 32-bit (FP32) and 64-bit (FP64)

Valid and common families:

- FP add/sub/mul: `fadd`, `fsub`, `fmul`
- Fused multiply-add: `fmla` (and lane forms)
- FP compare (mask results): `fcmgt`/`fcmge`, `fcmeq`
- FP min/max: `fmin`, `fmax` (and variants)

Representative FP Examples

```
/* FP32 arithmetic */
fadd v0.4s, v1.4s, v2.4s
fmul v3.4s, v4.4s, v5.4s
fmla v6.4s, v7.4s, v8.4s

/* FP64 arithmetic */
```

```
fadd v9.2d, v10.2d, v11.2d
fmla v12.2d, v13.2d, v14.2d

/* FP compare masks */
fcmgt v15.4s, v0.4s, v1.4s
fcmeq v16.2d, v9.2d, v10.2d
```

Intrinsic Family Mapping (FP)

```
#include <arm_neon.h>

static inline float32x4_t fma_f32(float32x4_t acc, float32x4_t a,
↪  float32x4_t b)
{
    return vfmaq_f32(acc, a, b);
}

static inline uint32x4_t gt_f32(float32x4_t a, float32x4_t b)
{
    return vcgtq_f32(a, b);
}
```

Permutation and Lane-Movement (All Types)

Permutation instructions operate on the layout, mostly independent of “signed vs unsigned” meaning.

Common families:

- Interleave/de-interleave: `zip1/zip2`, `uzp1/uzp2`
- Transpose: `trn1/trn2`
- Extract window: `ext`

- Table lookup shuffle: `tbl`
- Lane extract/insert: `umov`, `ins`

```
zip1 v0.16b, v1.16b, v2.16b
uzp2 v3.8h,  v4.8h,  v5.8h
trn1 v6.4s,  v7.4s,  v8.4s
ext  v9.16b, v10.16b, v11.16b, #4
tbl  v12.16b, {v13.16b}, v14.16b
```

Reduction Families (Selected Patterns)

Reductions must widen for narrow integer types and must consider rounding/association for FP.

Integer reduction patterns:

- widen + pairwise add, then horizontal add
- max/min reductions for any/all style checks

FP reduction patterns:

- accumulate in vector, then fixed-tree horizontal sum
- multiple accumulators to reduce dependency chains

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

/* Sum 16 bytes -> u32 scalar (widen then reduce) */
static inline uint32_t sum_u8_16(uint8x16_t v)
{
    uint16x8_t w1 = vpaddlq_u8(v);
    uint32x4_t w2 = vpaddlq_u16(w1);
    return vaddvq_u32(w2);
}

/* Sum 4 floats -> scalar (fixed tree) */
static inline float sum_f32_4(float32x4_t v)
{
    float32x2_t s = vadd_f32(vget_low_f32(v), vget_high_f32(v));
    s = vpadd_f32(s, s);
    return vget_lane_f32(s, 0);
}
```

Final Rules of Thumb

- Choose the **view suffix** that matches your element type; do not mix unintentionally.
- Treat comparisons as producing **masks** (all-ones/all-zeros) and keep them in that form for blends.
- Widen early for integer accumulation; define overflow behavior explicitly (wrap vs saturate).
- For FP reductions, expect last-bit differences; define acceptable numerical tolerance and test.

Appendix C — Practical Optimization Rules

This appendix distills practical, architecture-grounded rules for deciding **when to use NEON**, **when not to**, and how to reason about the **memory vs compute balance**. The intent is operational guidance, not theory.

When NEON Helps

NEON helps when the workload exhibits **data-level parallelism** with predictable control flow and efficient memory access.

Clear Win Conditions

- Unit-stride, contiguous memory access (SoA-friendly layouts).
- Many independent operations per cache line (high arithmetic intensity).
- Regular loops with no loop-carried dependencies.
- Accumulation that can use multiple independent accumulators.
- Kernels that amortize shuffles over many arithmetic ops.

Canonical Examples That Scale Well

- Vector adds/mults over large arrays.
- Dot products, FIR filters, convolution-like kernels.
- Image/audio pipelines with widening, FMA, and saturation.

Example: Compute-Heavy Kernel (FMA-Dominated)

```
/* v0,v1 independent accumulators */
fmla v0.4s, v2.4s, v3.4s
fmla v1.4s, v4.4s, v5.4s
```

Rule

If you can keep **multiple accumulators** live and hide load latency with useful work, NEON will usually scale close to peak throughput.

When NEON Hurts

NEON can hurt when overheads dominate or when vectorization introduces extra work that the scalar code did not have.

Common Anti-Patterns

- Memory-bound loops with low arithmetic intensity.
- Irregular access patterns (gathers/scatters emulated via shuffles).
- Heavy per-iteration shuffles (`tbl`, long `zip/trn` chains).
- Tiny loops where setup cost exceeds useful work.
- Frequent scalar-vector transitions (lane extracts inside the hot loop).

Example: Shuffle-Dominated Inner Loop (Red Flag)

```
/* High shuffle pressure, little arithmetic */
tbl  v0.16b, {v1.16b}, v2.16b
zip1 v3.16b, v0.16b, v4.16b
trn1 v5.4s,  v3.4s,  v6.4s
```

Rule

If a kernel spends more instructions **rearranging data** than computing on it, NEON may slow things down. Change the data layout or the algorithm first.

Memory vs Compute Balance

Performance is bounded by the slower of:

$$\text{time} = \max(\text{memory time},\ \text{compute time})$$

Understanding which side dominates is the key optimization skill.

Memory-Bound Kernels

Symptoms:

- Performance barely improves with more unrolling or FMAs.
- Speed correlates with cache behavior and bandwidth.

Typical causes:

- One or two arithmetic ops per cache line.
- Multiple passes over the same data.

Example: Memory-Bound Add

```
/* One add per 16-byte load */
ld1 {v0.16b}, [x1], #16
ld1 {v1.16b}, [x2], #16
add v0.16b, v0.16b, v1.16b
st1 {v0.16b}, [x0], #16
```

What Helps

- Fuse loops (do more work per load).
- Improve locality (blocking, reuse).
- Reduce loads/stores and passes over memory.

Compute-Bound Kernels

Symptoms:

- Performance scales with unrolling and multiple accumulators.
- Reducing dependency chains improves throughput.

Typical causes:

- Many FMAs or integer ops per loaded byte.
- Data fits in cache and is reused.

Example: Compute-Bound Accumulation

```
/* Load once, compute many times */
fmla v0.4s, v2.4s, v3.4s
fmla v0.4s, v4.4s, v5.4s
fmla v0.4s, v6.4s, v7.4s
```

What Helps

- Multiple independent accumulators.
- Scheduling loads early; compute while waiting.
- Using FMA to reduce instruction count and rounding steps.

Practical Decision Checklist

Before vectorizing with NEON, answer these questions:

- Is the loop **memory-bound or compute-bound**?

- Can the data be arranged as **SoA** to minimize shuffles?
- Can I perform **multiple operations per load**?
- Can I use **multiple accumulators** to break dependencies?
- Are shuffles amortized or dominating?

Rules of Thumb (Condensed)

- NEON helps when arithmetic intensity is high and control flow is simple.
- NEON hurts when memory traffic or shuffles dominate.
- Fix data layout before adding more SIMD instructions.
- Measure on the target core; intuition alone is unreliable.
- Prefer intrinsics first; drop to assembly only for proven hotspots.

References

Architectural Specifications (Primary)

- Arm Architecture Reference Manual for A-profile architecture (AArch64), covering Advanced SIMD (NEON) and floating-point instruction semantics, register model, load/store forms, permutation, and exception behavior.

- Arm Architecture Reference Manual supplement and revisions relevant to the deployed AArch64 baseline, clarifying instruction encodings, corner-case semantics, and architectural constraints.

Instruction-Level References (Primary)

- Official AArch64 instruction listings and descriptions for Advanced SIMD (NEON) operations: arithmetic, logical, compare/mask behavior, permute/shuffle (`zip/uzp/trn/ext/tbl`), and structured loads/stores (`ld1/ld2/ld3/ld4`, `st1/st2/st3/st4`).

- Official descriptions of lane indexing rules and element view suffix rules (`.b/.h/.s/.d`) and how they constrain valid instruction forms.

ABI and Calling Convention (Primary)

- AArch64 Procedure Call Standard (AAPCS64): register roles, caller/callee-saved rules, and ABI constraints relevant to using NEON in functions, inline assembly, and standalone assembly modules.
- AArch64 ELF ABI notes relevant to object-level integration and assembly linkage conventions.

Compiler and Toolchain Documentation (Primary)

- GCC documentation for AArch64 and vector extensions: supported intrinsic headers, vector types, code generation behavior, auto-vectorization constraints, and inline assembly constraints/clobbers.
- Clang/LLVM documentation for AArch64: NEON intrinsic support, codegen notes, auto-vectorization model, and inline asm rules.
- GAS (GNU assembler) documentation for AArch64 syntax and directives as used in `.S` files, including operand forms for Advanced SIMD instructions and structured load/store syntax.

Intrinsic Interfaces (Primary)

- Arm NEON intrinsic API definitions provided by toolchain headers (e.g., `arm_neon.h`) describing intrinsic names, operand types, lane semantics, and mapping to instruction families.
- Toolchain-specific intrinsic guides and notes for AArch64 Advanced SIMD and floating-point intrinsic coverage.

Microarchitecture and Performance Guidance (Authoritative)

- Official Arm optimization guides describing throughput/latency concepts, dependency chains, scheduling principles, load-to-use latency hiding, and common performance pitfalls for Advanced SIMD pipelines.
- Official Arm microarchitecture guides and software optimization manuals for typical core families, emphasizing practical tuning heuristics (unrolling, multiple accumulators, register pressure management) and memory-system interactions (cache-line effects, bandwidth limits).

Academic and Systems Performance Literature (Supporting)

- Peer-reviewed literature and established systems-performance texts covering: arithmetic intensity, roofline-style reasoning (memory vs compute balance), reduction accuracy and floating-point associativity effects, and general SIMD optimization principles applicable to NEON.
- Verified benchmark methodology references emphasizing measurement on target hardware, counter-based profiling, and avoiding misleading microbenchmarks.

Scope Note for This References Chapter

- This booklet intentionally relies on **primary specifications** (architecture reference, ABI) for correctness and on **toolchain documentation** for compiler/intrinsic/assembler behavior.
- Microarchitectural tuning guidance is treated as **core-dependent**; the rules presented are constrained to widely applicable principles and always require measurement on the target core.