Low-Level Programming on Linux (x86-64)

Performance Optimization and SIMD

redhat

debian

9

Prepared by Ayman Alheraki

https://simplifycpp.org

Low-Level Programming on Linux (x86-64) :

Performance Optimization and SIMD

Prepared by Ayman Alheraki

simplifycpp.org

November 2025

Contents

Author's Introduction

Performance on modern x86–64 Linux systems is not determined by individual instructions in isolation, but by the coordinated behavior of entire microarchitectural subsystems. Speculative execution, deep out-of-order pipelines, multi-level cache hierarchies, branch predictors, memory disambiguation logic, SIMD execution units, and high-resolution timing facilities collectively define the performance envelope of real programs. Writing fast software today requires understanding these components not as abstract concepts, but as concrete architectural forces that dictate whether an algorithm sustains peak throughput or stalls behind memory latency. This booklet examines performance from that perspective—through the lens of the hardware itself.
The primary goal of this book is to guide the reader beyond surface-level optimization techniques and into the fundamental principles that govern high-performance execution on Linux. Rather than focusing exclusively on compiler behavior or algorithmic complexity, we treat the processor as what it truly is: a parallel, speculative machine capable of issuing multiple instructions per cycle and executing wide SIMD operations concurrently. We examine how these capabilities degrade when memory access patterns become irregular, when branches are difficult to predict, or when loops fail to expose sufficient independent work to saturate execution resources.
This booklet introduces several essential concepts: the distinction between latency and throughput, the role of instruction-level parallelism, the mechanisms behind branch prediction, the importance of data locality, and the effective use of YMM and ZMM

registers for wide SIMD computation. These ideas are not presented abstractly. Instead, they are demonstrated through complete, hand-written assembly examples that expose the real cost of instructions and the observable behavior of the microarchitecture under controlled conditions. Kernels such as the introductory vector addition, the AVX2 reduction loop, and the final eight-element matrix-multiplication microkernel illustrate how performance depends on register residency, load–store patterns, loop structure, and the elimination of unnecessary control flow.

Throughout this book, performance is measured precisely using serialized rdtsc, dependency-chain microbenchmarks, and carefully controlled loop unrolling. These tools allow us to quantify execution behavior rather than approximate it. Small changes in instruction ordering, memory layout, or unrolling strategy frequently produce measurable differences in cycles per element, reinforcing the central principle of performance engineering: the correct optimization is the one validated by hardware measurements, not the one that merely appears optimal in theory.

This booklet also aims to demystify the internal design of high-performance libraries. The techniques explored here—tiling, register blocking, vectorized arithmetic, fused multiply–add instructions, and branch-free loops—form the foundation of optimized numerical kernels such as matrix multiplication and convolution. By exposing the microarchitectural strategies behind these implementations, the reader gains insight into how production-grade math libraries achieve their performance and how similar techniques can be applied to custom software.

Book 9 concludes the SIMD-focused portion of the series by integrating concepts from processor architecture, instruction scheduling, memory hierarchy, and vector execution. The objective is not merely to present fast assembly routines, but to provide a transferable methodology for analyzing and improving performance on future hardware platforms. As Linux and x86–64 systems evolve, the core principles presented here remain constant: measure precisely, understand the pipeline, reduce memory

traffic, exploit parallelism, and align algorithms with the structure of the processor. The techniques demonstrated in this book form the foundation upon which subsequent volumes will build. Whether designing memory allocators, cryptographic primitives, runtime components, or specialized numerical kernels, a solid understanding of SIMD execution and low-level performance engineering is indispensable for anyone who intends to write software that does not merely execute correctly, but executes at the full capability of the hardware.

Stay Connected

For more discussions and valuable content about Performance Optimization and SIMD, I invite you to follow me on LinkedIn:

https://linkedin.com/in/aymanalheraki

You can also visit my personal website:

https://simplifycpp.org

Ayman Alheraki

Chapter 1

Performance Bottlenecks

1.1 CPU vs Memory

Modern x86–64 processors expose execution resources that far exceed the ability of the memory subsystem to deliver data at comparable speed. The primary cause of most performance bottlenecks is not arithmetic throughput, but the latency and bandwidth gap between computation units and the hierarchical memory system. Understanding this gap at the level of pipeline behavior, cache residency, and data movement is the foundation of all serious performance engineering on Linux.

1.1.1 The Microarchitectural Gap

Post–2020 CPUs routinely sustain multiple fused-domain µops per cycle, schedule hundreds of in-flight instructions out of order, and expose wide SIMD execution ports capable of processing 256- and 512-bit vectors. Arithmetic throughput is rarely the limiting factor; the dominant constraint is the ability to feed the execution units. Memory, by contrast, remains fundamentally slow. Even with DDR5 and aggressive

hardware prefetching, a full DRAM round trip often costs hundreds of cycles. A single load that misses all cache levels introduces latency that cannot be completely hidden, even by deep out-of-order windows.

In real Linux workloads—especially those involving irregular access patterns, pointer chasing, or sparse data structures—the CPU frequently operates at a fraction of its theoretical throughput because memory cannot supply operands fast enough.

1.1.2 Cache Hierarchy as the Primary Battlefield

Performance on x86–64 Linux systems depends far more on cache residency than on the nominal cost of arithmetic instructions. Cache misses dominate execution time. Small changes in layout—alignment, stride, or access order—often determine whether the processor remains busy or stalls waiting for memory.

L1 cache latency is measured in a handful of cycles. L2 is several times slower. L3 is an order of magnitude slower. DRAM adds two additional orders of magnitude. Effective performance engineering minimizes movement toward the bottom of this hierarchy.

The fundamental rule is simple: the CPU is fast only when it operates on data that is already close to it.

1.1.3 Memory-Level Parallelism and Pipeline Exposure

Modern microarchitectures can maintain multiple outstanding cache misses, allowing useful work to continue while memory requests are pending. However, memory-level parallelism saturates quickly when access patterns are dependent, unaligned, pointer-based, or otherwise serialized.

When each load depends on the result of the previous one, memory latency becomes sequential. Even the largest reorder buffers cannot hide this behavior. This explains why linked lists, trees, and hash-based data structures often suffer order-of-magnitude

slowdowns compared to flat, contiguous arrays.
The pipeline executes aggressively, but it cannot violate the physical limits of memory movement.

1.1.4 Example: ALU-Bound vs Memory-Bound Loops

The following minimal examples illustrate the difference. No libc is used; each loop terminates by reaching zero.

1. ALU-Bound Loop (Fast)

   All data remains in registers. Throughput is limited only by execution ports.

```
.intel_syntax noprefix
.text
.global alu_bound

alu_bound:
    mov rcx, 1000000        # loop count
    mov rax, 1
.loop:
    imul rax, 13
    xor  rax, 7
    dec  rcx
    jnz  .loop
    ret
```

   Modern CPUs retire several of these µops per cycle. The loop executes at near-ideal throughput.

2. Memory-Bound Loop (Slow)

   Each iteration touches a distant memory location, forcing repeated cache misses.

```
.intel_syntax noprefix
.bss
.align 64
buffer:
    .skip 8000000          # 8 MB region

.text
.global memory_bound
memory_bound:
    mov rcx, 1000000
    lea rbx, buffer

.loop:
    mov rax, [rbx]         # potential DRAM access
    add rbx, 64            # skip to next cache line
    dec rcx
    jnz .loop
    ret
```

Once the working set exceeds cache capacity, each iteration may incur hundreds of cycles of latency. SIMD provides no benefit when loads miss all cache levels.

1.1.5 CPU Optimization Without Memory Awareness Fails

Optimizing arithmetic, reordering instructions, or applying SIMD does not yield improvement when the working set does not fit in cache. Bandwidth and latency dominate. SIMD units idle while waiting for scalar loads, and random access patterns negate theoretical speedups.

The most effective optimizations on Linux reorganize data, not code. Linearization, blocking, cache-aligned unrolling, and prefetch-friendly traversals typically outperform instruction-level tricks.

1.1.6 Prefetching and Its Limits on Linux

Hardware prefetchers detect linear and near-linear patterns, but they cannot predict:

- pointer-based traversals,
- hash probes,
- tree navigation,
- unpredictable strides.

Linux user space can issue explicit prefetch instructions, but they help only when future accesses are predictable and sufficiently far ahead.

```
.intel_syntax noprefix
.text
.global prefetch_loop

prefetch_loop:
    mov rcx, 1000000
    lea rbx, buffer

.loop:
    prefetcht0 [rbx + 256]  # prefetch future line
    mov rax, [rbx]
    add rbx, 64
    dec rcx
    jnz .loop
    ret
```

Prefetching becomes counterproductive when access patterns are dependent or irregular.

1.1.7 The Dominant Principle: Move Less Data

Memory movement is more expensive than arithmetic. Transferring a single cache line often costs as many cycles as hundreds of integer operations. The memory hierarchy, not the ALUs, defines real performance limits.
Optimizing Linux software requires treating memory as an active constraint that must be minimized, aligned, and predicted.

1.1.8 Summary

Modern CPUs execute billions of instructions per second, but memory delivers data orders of magnitude more slowly. This asymmetry is the single largest source of performance bottlenecks on x86–64 Linux systems. Real optimization begins when computation is recognized as cheap and data movement as expensive.
Later chapters apply SIMD techniques not merely for arithmetic density, but for reducing memory traffic, maximizing locality, and ensuring data remains resident in high-speed caches.

1.2 Latency vs Throughput

Performance on modern x86–64 Linux systems is governed by two distinct dimensions: latency and throughput. Although related, they describe fundamentally different execution properties. Optimizing one does not imply optimizing the other.

1.2.1 Understanding Latency

Latency is the time required for a single operation to complete from start to finish.
It reflects the real delay introduced by instructions, memory accesses, or dependency chains.
Examples of high-latency events include:

- cache misses reaching DRAM,
- long dependency chains,
- serialized instructions.

Even aggressive out-of-order execution cannot hide latency when instructions depend on previous results.

```
.intel_syntax noprefix
.text
.global latency_chain

latency_chain:
    mov rcx, 20000000
    mov rax, 3
.loop:
    imul rax, 7          # dependent chain
    add  rax, 5
```

```
    sub  rax, 2
    dec  rcx
    jnz  .loop
    ret
```

Each iteration exposes full instruction latency.

1.2.2 Understanding Throughput

Throughput is the rate at which independent operations execute. Modern CPUs issue multiple µops per cycle when instructions are independent and data is resident in cache.

```
.intel_syntax noprefix
.text
.global throughput_loop

throughput_loop:
    mov rcx, 20000000
    xor rax, rax
    xor rbx, rbx
    xor rdx, rdx

.loop:
    add rax, 3
    add rbx, 5
    add rdx, 7
    dec rcx
    jnz .loop
    ret
```

Independent instructions execute in parallel, maximizing throughput.

1.2.3 The Principle

Latency limits depth; throughput limits width. Effective optimization begins by identifying which constraint dominates and applying the appropriate strategy.

1.2.4 Summary

Latency governs serial progress. Throughput governs parallel capacity. Modern Linux performance engineering depends on recognizing which dimension constrains a workload and optimizing accordingly.

1.3 Cache Miss Behavior

Cache misses are architectural hazards that define real performance limits. As execution width increases, memory latency becomes the dominant bottleneck. Understanding misses requires analyzing access patterns, pipeline reaction, and the limits of latency hiding.

1.3.1 Sequential Access (Cache-Friendly)

```
.intel_syntax noprefix
.text
.global seq_walk

seq_walk:
    mov rcx, rsi            # element count
.loop:
    mov rax, [rdi]
    add rdi, 8
    dec rcx
    jnz .loop
    ret
```

Prefetchers reduce miss rates dramatically for contiguous data.

1.3.2 Strided Access (Disruptive)

```
.intel_syntax noprefix
.text
.global stride_walk

stride_walk:
.loop:
```

```
    mov rax, [rdi]
    add rdi, rdx          # stride
    dec rcx
    jnz .loop
    ret
```

Large strides defeat prefetching and increase cache and TLB misses.

1.3.3 Pointer Chasing (Worst Case)

```
.intel_syntax noprefix
.text
.global pointer_chase

pointer_chase:
.loop:
    mov rdi, [rdi]        # dependent load
    test rdi, rdi
    jnz .loop
    ret
```

Each miss stalls the entire chain.

1.3.4 Summary

Cache behavior determines performance far more than instruction choice. Sequential access maximizes throughput. Strides disrupt locality. Pointer chasing serializes latency. Cache-aware design is the first requirement for meaningful optimization.

Chapter 2

Branch Optimization

2.1 Branch Prediction

Modern x86–64 processors rely heavily on branch prediction to sustain high throughput. Without accurate prediction, pipelines stall, instruction retirement collapses, and execution degrades to a fraction of theoretical performance. Branch prediction is not an optional enhancement; it is fundamental to maintaining instruction-level parallelism in Linux workloads.

2.1.1 The Pipeline's Dependence on Prediction

Contemporary CPUs are deeply pipelined, superscalar, and aggressively speculative. To maintain throughput, the frontend must fetch and decode instructions far ahead of current execution. Conditional branches disrupt this continuity. When a conditional jump is encountered, the processor must predict the outcome before the condition is resolved.

Correct prediction allows uninterrupted execution. A misprediction forces the

pipeline to flush speculative work and restart from the correct path. On post–2020 microarchitectures, this penalty typically costs dozens of cycles.
Linux user-space code that appears trivial at the source level frequently becomes dominated by branch behavior after compilation, particularly when conditions depend on runtime data.

2.1.2 Types of Branch Predictors

Modern x86–64 processors employ hybrid prediction engines composed of:

- local history predictors,
- global history predictors,
- TAGE-like pattern tables,
- loop predictors,
- indirect target predictors,
- return address stack (RAS).

Linux applications do not control these predictors directly, but their behavior is shaped by code structure and data regularity. Predictors excel when patterns are stable and repetitive. They fail when conditions are random, irregular, or frequently inverted.

2.1.3 Example: Predictable Branch (Fast Path)

```
.intel_syntax noprefix
.text
.global predictable_branch
```

```
predictable_branch:
    mov rcx, 50000000
.loop:
    cmp rcx, 0
    jne .continue        # strongly biased
.continue:
    dec rcx
    jnz .loop
    ret
```

Because the branch outcome is consistent across iterations, prediction accuracy approaches 100%. The pipeline remains saturated and execution is limited by throughput rather than control flow.

2.1.4 Example: Unpredictable Branch (Misprediction Storm)

```
.intel_syntax noprefix
.text
.global unpredictable_branch

# rdi = pointer to array
# rcx = element count
unpredictable_branch:
.loop:
    mov eax, [rdi]
    test eax, eax
    jz .zero
    add rdi, 4
    dec rcx
    jnz .loop
    ret
.zero:
    add rdi, 4
```

```
dec rcx
jnz .loop
ret
```

When data distribution is irregular, the predictor repeatedly mispredicts. Each misprediction triggers a pipeline flush that costs tens of cycles. The penalty grows as speculation depth increases.

2.1.5 Misprediction Cost and Pipeline Flush

When a misprediction occurs:

1. speculative instructions are invalidated,
2. architectural state is rolled back,
3. the frontend fetches from the correct path,
4. the pipeline refills before execution resumes.

On modern cores, this penalty can exceed the latency of a DRAM access. Eliminating unpredictable branches often yields dramatic performance improvements.

2.1.6 Avoiding Branch Penalties

Linux performance optimization frequently focuses on removing unpredictable branches entirely. Common techniques include:

- conditional moves (cmov),
- control-flow to data-flow transformation,
- arithmetic masking,

- SIMD blending,
- data layout transformations.

2.1.7 Replacing a Branch with cmov

```
.intel_syntax noprefix
.text
.global cmov_replace

# rdi = pointer
# rcx = count
cmov_replace:
    xor rbx, rbx
.loop:
    mov eax, [rdi]
    mov edx, 7
    test eax, eax
    cmovnz edx, eax
    add rbx, rdx
    add rdi, 4
    dec rcx
    jnz .loop
    mov rax, rbx
    ret
```

The unpredictable branch is replaced by a conditional move. No pipeline flush occurs, and execution remains linear.

2.1.8 Branch Elimination with SIMD Masking

```
.intel_syntax noprefix
.text
```

```
.global simd_branchless

# rdi = pointer to 8 integers
simd_branchless:
    vmovdqu ymm0, [rdi]
    vpxor   ymm1, ymm1, ymm1
    vpcmpgtd ymm2, ymm0, ymm1
    vpblendvb ymm3, ymm1, ymm0, ymm2
    vmovdqu [rdi], ymm3
    ret
```

Masked SIMD operations eliminate control flow entirely.

2.1.9 Indirect Branches and the BTB

```
.intel_syntax noprefix
.text
.global indirect_jump

# rdi = pointer to target
indirect_jump:
    jmp [rdi]
```

Indirect branches stress the branch target buffer. Reducing target variability improves prediction accuracy.

2.1.10 Return Prediction and the RAS

```
.intel_syntax noprefix
call label
label:
    ret
```

Balanced call/return behavior preserves RAS accuracy. Irregular control flow disrupts it.

2.1.11 Summary

Predictable branches sustain throughput. Unpredictable branches trigger pipeline flushes that dominate execution time. Low-level optimization on Linux often begins by reducing branch unpredictability through conditional moves, branch elimination, and SIMD masking.

2.2 Using cmov

Conditional moves provide value selection without altering control flow. They replace unpredictable branches with data dependencies, preserving pipeline continuity.

2.2.1 Basic Mechanics

```
.intel_syntax noprefix
cmp eax, ebx
cmovg ecx, edx
```

No jump occurs; selection is resolved inside the execution units.

2.2.2 Branch vs Branchless Selection

```
.intel_syntax noprefix
.text
.global cmov_select

cmov_select:
    mov rcx, 20000000
    xor rbx, rbx
.loop:
    mov eax, [rdi]
    mov edx, 7
    test eax, eax
    cmovnz edx, eax
    add rbx, edx
    add rdi, 4
    dec rcx
    jnz .loop
    mov rax, rbx
```

```
    ret
```

This version avoids misprediction entirely.

2.2.3 Absolute Value Using cmov

```
.intel_syntax noprefix
.text
.global abs_cmov

abs_cmov:
    mov eax, edi
    mov edx, eax
    neg edx
    test eax, eax
    cmovs eax, edx
    ret
```

2.2.4 Min and Max Using cmov

```
.intel_syntax noprefix
.text
.global min_cmov

min_cmov:
    cmp eax, edx
    cmovg eax, edx
    ret
```

```
.intel_syntax noprefix
.text
.global max_cmov
```

```
max_cmov:
    cmp eax, edx
    cmovl eax, edx
    ret
```

2.2.5 Summary

cmov trades unpredictable control flow for predictable data flow. Its cost is negligible compared to branch misprediction penalties and it enables stable execution and vectorization.

2.3 Removing Conditionals

Removing conditionals transforms divergent control flow into uniform execution, enabling steady pipeline utilization and SIMD-friendly loops.

2.3.1 Arithmetic Masking

```
.intel_syntax noprefix
mov edx, eax
sar edx, 31
and edx, eax
add rbx, edx
```

2.3.2 Branchless Loop Transformation

```
.intel_syntax noprefix
loop:
    mov eax, [rdi]
    mov edx, eax
    sar edx, 31
    and edx, eax
    add rbx, edx
    add rdi, 4
    dec rcx
    jnz loop
```

2.3.3 SIMD Conditional Elimination

```
.intel_syntax noprefix
vmovdqu ymm0, [rdi]
vpxor   ymm1, ymm1, ymm1
vpcmpgtd ymm2, ymm0, ymm1
```

```
vpblendvb ymm3, ymm1, ymm0, ymm2
vmovdqu [rdi], ymm3
ret
```

2.3.4 Summary

Removing conditionals eliminates misprediction penalties, stabilizes pipeline behavior, and enables vectorization. Branchless execution is a cornerstone of modern high-performance Linux software on x86–64.

Chapter 3

Loop Optimization

3.1 Loop Unrolling

Loop unrolling is one of the oldest and most effective low-level optimization techniques, and its importance has increased rather than diminished on modern x86–64 systems. As CPUs grow deeper pipelines, wider execution ports, and more aggressive out-of-order engines, loop control overhead becomes increasingly visible. Unrolling reduces this overhead by increasing the amount of useful work per iteration, improving instruction-level parallelism (ILP), reducing branch pressure, and enabling more effective SIMD vectorization.
When applied correctly, unrolling produces consistently higher throughput on Linux, particularly in performance-critical kernels where each cycle matters.

3.1.1 Why Unrolling Improves Performance

Loop control introduces overhead in the form of:

- conditional branches,

- loop-counter dependencies,
- pointer increments,
- compare operations.

When these operations dominate or interfere with useful work, they limit execution throughput.
Unrolling increases the ratio of useful instructions to loop-control instructions:

1. fewer branch evaluations,
2. more independent instructions,
3. better alignment with execution ports,
4. larger straight-line instruction sequences,
5. increased opportunity for SIMD vectorization.

Array processing, numerical kernels, cryptographic routines, and data transforms on Linux often benefit heavily from carefully tuned unrolling factors.

3.1.2 Baseline Scalar Loop (Not Unrolled)

```
.intel_syntax noprefix
.text
.global baseline_loop

# rdi = pointer
# rcx = element count
baseline_loop:
.loop:
```

```
    mov eax, [rdi]
    add eax, 3
    mov [rdi], eax

    add rdi, 4
    dec rcx
    jnz .loop
    ret
```

Each iteration performs one load, one arithmetic operation, one store, and several loop-control instructions. The control overhead represents a significant fraction of total execution.

3.1.3 Simple 2× Unrolling

```
.intel_syntax noprefix
.text
.global unroll2

# rdi = pointer
# rcx = count (even)
unroll2:
.loop:
    mov eax, [rdi]
    add eax, 3
    mov [rdi], eax

    mov eax, [rdi + 4]
    add eax, 3
    mov [rdi + 4], eax

    add rdi, 8
    sub rcx, 2
```

```
    jnz .loop
    ret
```

Branch frequency is halved, and the CPU observes more independent work per iteration.

3.1.4 4× Unrolling — A Practical Sweet Spot

```
.intel_syntax noprefix
.text
.global unroll4

# rdi = pointer
# rcx = count (multiple of 4)
unroll4:
.loop:
    mov eax, [rdi]
    add eax, 3
    mov [rdi], eax

    mov eax, [rdi + 4]
    add eax, 3
    mov [rdi + 4], eax

    mov eax, [rdi + 8]
    add eax, 3
    mov [rdi + 8], eax

    mov eax, [rdi + 12]
    add eax, 3
    mov [rdi + 12], eax

    add rdi, 16
```

```
    sub rcx, 4
    jnz .loop
    ret
```

Unrolling by four typically balances ILP gains against instruction-cache pressure on modern Linux systems.

3.1.5 Unrolling Combined with SIMD

```
.intel_syntax noprefix
.text
.global simd_unroll4

# rdi = pointer
# rcx = count (multiple of 32 bytes)
simd_unroll4:
    vpbroadcastd ymm7, DWORD PTR [rip + addval]

.loop:
    vmovdqu ymm0, [rdi]
    vmovdqu ymm1, [rdi + 32]
    vmovdqu ymm2, [rdi + 64]
    vmovdqu ymm3, [rdi + 96]

    vpaddd ymm0, ymm0, ymm7
    vpaddd ymm1, ymm1, ymm7
    vpaddd ymm2, ymm2, ymm7
    vpaddd ymm3, ymm3, ymm7

    vmovdqu [rdi], ymm0
    vmovdqu [rdi + 32], ymm1
    vmovdqu [rdi + 64], ymm2
    vmovdqu [rdi + 96], ymm3
```

```
    add rdi, 128
    sub rcx, 32
    jnz .loop
    ret

.data
addval:
    .long 3
```

SIMD combined with unrolling maximizes arithmetic density while minimizing branch overhead.

3.1.6 Handling Remainders

```
.intel_syntax noprefix
.text
.global unroll2_with_tail

# rdi = pointer
# rcx = total count
unroll2_with_tail:
    mov rdx, rcx
    shr rcx, 1
    jz .tail

.loop:
    mov eax, [rdi]
    add eax, 3
    mov [rdi], eax

    mov eax, [rdi + 4]
    add eax, 3
```

```
    mov [rdi + 4], eax

    add rdi, 8
    dec rcx
    jnz .loop

.tail:
    test rdx, 1
    jz .done
    mov eax, [rdi]
    add eax, 3
    mov [rdi], eax

.done:
    ret
```

3.1.7 Summary

Loop unrolling increases work density, reduces branch frequency, improves ILP, and enhances SIMD effectiveness. On modern Linux systems, well-tuned unrolling is among the highest-impact optimizations available.

3.2 Register Blocking

Register blocking improves loop performance by keeping active data in registers across multiple operations, reducing memory traffic and dependency chains.

3.2.1 Baseline Loop

```
.intel_syntax noprefix
.text
.global baseline_scalar

baseline_scalar:
.loop:
    mov eax, [rdi]
    add eax, 5
    mov [rdi], eax

    add rdi, 4
    dec rcx
    jnz .loop
    ret
```

3.2.2 Two-Register Blocking

```
.intel_syntax noprefix
.text
.global regblock2

regblock2:
.loop:
    mov eax, [rdi]
    mov edx, [rdi + 4]
```

```
    add eax, 5
    add edx, 5

    mov [rdi], eax
    mov [rdi + 4], edx

    add rdi, 8
    sub rcx, 2
    jnz .loop
    ret
```

3.2.3 Four-Register Blocking

```
.intel_syntax noprefix
.text
.global regblock4

regblock4:
.loop:
    mov eax, [rdi]
    mov edx, [rdi + 4]
    mov r8d, [rdi + 8]
    mov r9d, [rdi + 12]

    add eax, 5
    add edx, 5
    add r8d, 5
    add r9d, 5

    mov [rdi], eax
    mov [rdi + 4], edx
    mov [rdi + 8], r8d
```

```
    mov [rdi + 12], r9d

    add rdi, 16
    sub rcx, 4
    jnz .loop
    ret
```

3.2.4 Blocked Reduction

```
.intel_syntax noprefix
.text
.global reduce_block2

reduce_block2:
    xor rax, rax
    xor rdx, rdx

.loop:
    add rax, [rdi]
    add rdx, [rdi + 4]

    add rdi, 8
    sub rcx, 2
    jnz .loop

    add rax, rdx
    ret
```

3.2.5 SIMD Register Blocking

```
.intel_syntax noprefix
.text
.global simd_blocked
```

```
simd_blocked:
    vpbroadcastd ymm7, DWORD PTR [rip + addval]

.loop:
    vmovdqu ymm0, [rdi]
    vmovdqu ymm1, [rdi + 32]

    vpaddd ymm0, ymm0, ymm7
    vpaddd ymm1, ymm1, ymm7

    vmovdqu [rdi], ymm0
    vmovdqu [rdi + 32], ymm1

    add rdi, 64
    sub rcx, 16
    jnz .loop
    ret

.data
addval:
    .long 5
```

3.2.6 Summary

Register blocking reduces memory traffic, breaks dependency chains, and maximizes ILP. It complements unrolling and is foundational to high-performance SIMD kernels on Linux.

3.3 Software Prefetching

Software prefetching overlaps memory latency with computation by explicitly requesting future cache lines before they are accessed.

3.3.1 Baseline Without Prefetch

```
.intel_syntax noprefix
.text
.global linear_walk

linear_walk:
.loop:
    mov eax, [rdi]
    add eax, 1
    mov [rdi], eax

    add rdi, 4
    dec rcx
    jnz .loop
    ret
```

3.3.2 Prefetching Ahead

```
.intel_syntax noprefix
.text
.global prefetch_walk

prefetch_walk:
.loop:
    prefetcht0 [rdi + 64]
```

```
    mov eax, [rdi]
    add eax, 1
    mov [rdi], eax

    add rdi, 4
    dec rcx
    jnz .loop
    ret
```

3.3.3 Blocked Prefetching

```
.intel_syntax noprefix
.text
.global prefetch_blocked

prefetch_blocked:
.loop:
    prefetcht0 [rdi + 256]

    mov eax, [rdi]
    mov edx, [rdi + 4]
    mov r8d, [rdi + 8]
    mov r9d, [rdi + 12]

    add eax, 1
    add edx, 1
    add r8d, 1
    add r9d, 1

    mov [rdi], eax
    mov [rdi + 4], edx
    mov [rdi + 8], r8d
    mov [rdi + 12], r9d
```

```
    add rdi, 16
    sub rcx, 4
    jnz .loop
    ret
```

3.3.4 SIMD Prefetching

```
.intel_syntax noprefix
.text
.global simd_prefetch

simd_prefetch:
.loop:
    prefetcht0 [rdi + 256]

    vmovdqu ymm0, [rdi]
    vpaddd ymm0, ymm0, [rip + addvec]
    vmovdqu [rdi], ymm0

    add rdi, 32
    dec rcx
    jnz .loop
    ret

.data
addvec:
    .long 1,1,1,1,1,1,1,1
```

3.3.5 Summary

Software prefetching allows latency to be overlapped with computation in loops whose access patterns defeat hardware prefetchers. When combined with unrolling, register

blocking, and SIMD execution, it is a powerful tool for sustaining throughput in memory-intensive Linux workloads.

Chapter 4

SIMD and AVX

4.1 XMM Registers

XMM registers form the foundation of SIMD computation on x86-64 Linux systems. Introduced with SSE and extended through SSE4.2, the XMM register file contains sixteen 128-bit registers (xmm0–xmm15) available in 64-bit mode. Although later AVX extensions expand SIMD width to 256-bit and 512-bit registers, XMM remains a critical execution domain because many operations still operate exclusively on 128-bit lanes and because all wider operations conceptually remain compatible with XMM semantics. Understanding XMM registers is essential before advancing to AVX2 and AVX-512 because XMM operations define the fundamental SIMD execution model: packed arithmetic, data parallelism, register-to-register operations, and the avoidance of control-flow divergence.

4.1.1 XMM Registers as 128-Bit Data Containers

Each XMM register is 128 bits wide and can represent:

- sixteen 8-bit integers,
- eight 16-bit integers,
- four 32-bit integers or floats,
- two 64-bit integers or doubles.

The interpretation depends on the instruction. The hardware treats the register as a raw 128-bit vector; only the instruction semantics define how the bits are grouped. For example:

```
; xmm0 used as four 32-bit signed integers
paddd xmm0, xmm1

; xmm0 used as two 64-bit floats
addpd xmm0, xmm1
```

The physical register is the same; the instruction's type determines its meaning.

4.1.2 SIMD Parallelism Through XMM

XMM operations perform parallel arithmetic on multiple lanes simultaneously. This is the fundamental advantage of SIMD: executing multiple operations per instruction. Example: increment four 32-bit integers at once.

```
.intel_syntax noprefix
.text
.global add_four

# rdi = pointer
```

```
add_four:
    movdqu xmm0, [rdi]
    paddd  xmm0, [rel incvec]
    movdqu [rdi], xmm0
    ret

.data
incvec: .long 1,1,1,1
```

One instruction performs four additions. This scales far better than scalar loops on Linux when data is well-aligned and contiguous.

4.1.3 XMM Loads and Stores

Loading and storing are central to SIMD code performance because memory bandwidth, not arithmetic throughput, is often the limiting factor.

- Unaligned Load/Store

  ```
  movdqu xmm0, [rdi]     ; load 128 bits unaligned
  movdqu [rdi], xmm0     ; store 128 bits unaligned
  ```

- Aligned Load/Store

  ```
  movdqa xmm0, [rdi]
  movdqa [rdi], xmm0
  ```

  Aligned accesses are faster on some microarchitectures, but unaligned operations are allowed and generally safe. Alignment remains beneficial for sustained throughput in tight loops.

4.1.4 XMM Arithmetic and Logic Operations

XMM arithmetic instructions operate on all lanes simultaneously. These include:

- packed integer adds/subs (padd*, psub*),
- logical bitwise operations (por, pand, pxor),
- shifting/rotating packed integers (pslld, psrld, etc.),
- saturated arithmetic for specific element sizes.

Example: multiply four integers by a constant and add another vector.

```
.intel_syntax noprefix
.text
.global muladd4

muladd4:
    movdqu xmm0, [rdi]        ; load input
    pmulld xmm0, [rsi]        ; multiply element-wise
    paddd xmm0, [rdx]         ; add element-wise
    movdqu [rdi], xmm0
    ret
```

Such fused multiply-add sequences form the basis of vectorized processing loops.

4.1.5 XMM Register Use in Loops

High-performance SIMD loops depend on keeping XMM registers live across iterations. This minimizes memory transactions and sustains high throughput.

Basic XMM Loop (4-element increments)

```
.intel_syntax noprefix
.text
.global xmm_loop

# rdi = pointer
# rcx = iterations (each iteration processes 4 ints)
xmm_loop:
    movdqu xmm1, [rel add4]

.loop:
    movdqu xmm0, [rdi]
    paddd  xmm0, xmm1
    movdqu [rdi], xmm0

    add rdi, 16
    dec rcx
    jnz .loop
    ret

.data
add4: .long 1,1,1,1
```

This code removes nearly all scalar overhead and saturates the XMM execution ports.

4.1.6 Interaction Between XMM and General-Purpose Registers

XMM registers do not overlap with general-purpose registers; the CPU can use both domains concurrently. This allows:

- pointer arithmetic in GPRs,

- computation in XMM registers,
- loop counters in GPRs,
- independent instruction scheduling across domains.

Post-2020 superscalar cores can issue multiple SIMD instructions and multiple scalar instructions per cycle, provided dependencies are minimized.

4.1.7 XMM and the Register File Constraints

While XMM registers provide parallelism, they also introduce constraints:

- a loop that uses too many XMM registers may pressure the register allocator,
- spills to the stack degrade performance,
- mixing XMM and YMM instructions without VEX prefixes forces transitions known as "domain crossings," which hurt performance.

Correct use of VEX-encoded instructions (vaddps, vmovdqu) is recommended even in pure 128-bit workloads to avoid these penalties.
Example:

```
vaddps xmm0, xmm0, xmm1      ; VEX-encoded, avoids SSE AVX penalty
```

4.1.8 Using XMM for Branchless SIMD Logic

XMM registers enable masking and blending operations that eliminate branches.

Example: XMM Branchless Select

```
.intel_syntax noprefix
.text
.global xmm_select

# rdi = pointer
xmm_select:
    movdqu xmm0, [rdi]       ; input vector
    pxor   xmm1, xmm1        ; zero vector
    pcmpgtd xmm2, xmm0, xmm1 ; mask: >0 elements
    pand   xmm3, xmm0, xmm2  ; positive values
    ret
```

This produces a vector containing only positive elements with no branching.

4.1.9 Memory Alignment for XMM Performance

Although modern CPUs tolerate unaligned loads, optimal performance requires the data to be aligned to at least 16 bytes. Alignment reduces microarchitectural penalties, avoids split-line loads, and improves prefetcher accuracy.
Linux memory allocators often return sufficiently aligned blocks, but low-level developers must enforce alignment when designing custom allocators or SIMD-heavy data structures.

4.1.10 Summary

XMM registers provide the foundation of SIMD programming on x86-64 Linux systems. They enable processing multiple data elements per instruction, minimizing memory traffic and maximizing arithmetic throughput. Understanding XMM behavior—its operations, alignment requirements, masking capabilities, and interaction with the

pipeline—is essential before advancing to wider AVX2 and AVX-512 execution domains. Proper use of XMM registers is the first step toward writing high-performance, vectorized kernels that fully exploit the hardware.

Following sections will build on this foundation by exploring YMM and ZMM registers, vector-width scaling, and advanced SIMD optimization strategies.

4.2 YMM Registers

YMM registers extend SIMD computation from the 128-bit XMM domain to the 256-bit AVX/AVX2 domain. Each YMM register (ymm0–ymm15) is 256 bits wide and conceptually composed of two 128-bit lanes. AVX instructions operate across lanes in parallel, enabling double the data throughput compared to XMM-based SIMD. This expansion dramatically increases arithmetic density, memory bandwidth demands, and the importance of alignment and data layout for peak performance on Linux systems. Working effectively with YMM registers is essential for exploiting the full computational potential of modern x86-64 processors, especially in data-parallel workloads such as image processing, audio/video pipelines, cryptography, numerical simulations, and machine-learning primitives.

4.2.1 Structure and Execution Model of YMM Registers

A YMM register is:

- 256 bits wide,
- split into two 128-bit lanes internally,
- accessible via VEX-encoded AVX/AVX2 instructions.

All AVX arithmetic operates in parallel on the 256-bit register, but operations do not mix elements across lanes. Understanding this dual-lane structure is essential when designing algorithms that depend on vertical reductions or cross-lane communication.
Example:
Operating on eight 32-bit integers in parallel:

```
vpaddd ymm0, ymm1, ymm2    ; add 8 int32s simultaneously
```

The CPU handles four integers per 128-bit lane, two lanes at a time.

4.2.2 Advantages Over XMM Execution

Using YMM registers provides:

1. Double data width → 2× elements per instruction.
2. Higher arithmetic throughput → more work per µop.
3. More efficient memory bandwidth usage → fewer loads/stores for same work.
4. Better compatibility with modern execution ports → optimized for AVX2 flows.

These benefits come with increased pressure on L1 and L2 caches, making memory layout and alignment more critical.

4.2.3 YMM Load/Store Operations

YMM loads/stores move 256 bits (32 bytes) at a time.

- Unaligned Load/Store

  ```
  vmovdqu ymm0, [rdi]
  vmovdqu [rdi], ymm0
  ```

- Aligned Load/Store

  ```
  vmovdqa ymm0, [rdi]
  vmovdqa [rdi], ymm0
  ```

  Unaligned loads are allowed but may incur penalties on some microarchitectures when crossing cache-line boundaries. Alignment improves throughput in tight loops.

4.2.4 Basic YMM Arithmetic Example

Increment eight 32-bit integers at once.

```
.intel_syntax noprefix
.text
.global add_eight

# rdi = pointer
add_eight:
    vmovdqu ymm0, [rdi]
    vpaddd  ymm0, ymm0, [rel eight_add]
    vmovdqu [rdi], ymm0
    ret

.data
eight_add: .long 1,1,1,1,1,1,1,1
```

This doubles the throughput of equivalent XMM-based code.

4.2.5 Loop-Based YMM SIMD Processing

A typical YMM loop processes large arrays efficiently by loading/storing full vectors per iteration.

```
.intel_syntax noprefix
.text
.global ymm_loop

# rdi = pointer
```

```
# rcx = element count (multiple of 8)
ymm_loop:
    vmovdqu ymm1, [rel add8]

.loop:
    vmovdqu ymm0, [rdi]
    vpaddd  ymm0, ymm0, ymm1
    vmovdqu [rdi], ymm0

    add rdi, 32
    dec rcx
    jnz .loop
    ret

.data
add8: .long 1,1,1,1,1,1,1,1
```

Each iteration updates eight integers with minimal scalar overhead.

4.2.6 Using YMM Registers for Branchless Vector Logic

Masking and comparison operations enable branchless SIMD logic.

Example: Zero-Out Negative Elements

```
.intel_syntax noprefix
.text
.global zero_neg

# rdi = pointer
```

```
zero_neg:
    vmovdqu ymm0, [rdi]
    vpxor   ymm1, ymm1, ymm1
    vpcmpgtd ymm2, ymm0, ymm1     ; mask: element > 0
    vpblendvb ymm3, ymm1, ymm0, ymm2
    vmovdqu [rdi], ymm3
    ret
```

This eliminates branches entirely and processes eight integers in parallel.

4.2.7 Register Blocking with YMM

YMM registers excel when combined with register blocking. Wider registers allow more data to remain live across iterations.

Blocked Example

```
vmovdqu ymm0, [rdi]
vmovdqu ymm1, [rdi + 32]

vpaddd ymm0, ymm0, ymm7
vpaddd ymm1, ymm1, ymm7

vmovdqu [rdi], ymm0
vmovdqu [rdi + 32], ymm1
```

This processes sixteen integers before issuing the next branch, increasing ILP and data reuse.

4.2.8 Understanding AVX "Domain Crossings"

YMM instructions must use VEX prefixes. Mixing legacy SSE instructions (movdqu xmm0, ...) with VEX-encoded AVX instructions triggers a costly transition penalty. To avoid this:

- Always use VEX-encoded forms (vmovdqu, vaddps, etc.).
- Avoid using plain SSE instructions in AVX-heavy loops.

Example of correct usage:

```
vmovdqu xmm0, [rdi]
```

Even though the width is 128 bits, VEX encoding prevents domain penalties.

4.2.9 Memory Layout and Alignment Considerations

Because YMM loads fetch 32 bytes:

- Data should be aligned to 32 bytes for optimal performance.
- Structure-of-arrays (SoA) layouts perform better than array-of-structures (AoS).
- Large sequential arrays benefit from prefetching.
- Strided patterns may require manual prefetch instructions.

Linux allocators typically provide alignment sufficient for YMM usage, but high-performance kernels should enforce alignment explicitly.

4.2.10 Example: AVX2 Multiply-Add Kernel

A common pattern in numerical code is elementwise multiply-add processing.

```
.intel_syntax noprefix
.text
.global muladd_ymm

# rdi = ptr a
# rsi = ptr b
# rdx = ptr c (result)
# rcx = count (multiple of 8)
muladd_ymm:
.loop:
    vmovdqu ymm0, [rdi]
    vmovdqu ymm1, [rsi]
    vpmulld ymm2, ymm0, ymm1
    vpaddd  ymm2, ymm2, [rdx]    ; cumulative/other vector
    vmovdqu [rdx], ymm2

    add rdi, 32
    add rsi, 32
    add rdx, 32
    dec rcx
    jnz .loop
    ret
```

This kernel uses YMM registers exclusively to compute eight multiply-add operations per iteration.

4.2.11 Summary

YMM registers expand SIMD computation to 256 bits, doubling parallel throughput relative to XMM operations. They enable high-performance vector processing on Linux through wide loads, packed arithmetic, and branchless logic. Correct alignment, VEX encoding, domain consistency, and careful memory structuring are essential to achieve full performance. YMM registers form the core of AVX and AVX2 workloads and serve as a stepping stone toward even larger ZMM-based kernels.
Next sections will introduce ZMM registers, AVX-512 execution, and full-width vectorization strategies.

4.3 Vectorized Arithmetic

Vectorized arithmetic is the core functionality of SIMD execution on modern x86-64 Linux systems. By performing multiple arithmetic operations per instruction, vectorized computation dramatically increases throughput and reduces instruction count. With AVX/AVX2, arithmetic can be performed on 128-bit (XMM) and 256-bit (YMM) vectors, processing 4, 8, 16, or 32 elements per iteration depending on element size. Modern Linux workloads—such as cryptography, numerical kernels, compression, filtering, audio/video processing, and machine-learning primitives—derive substantial performance gains from vectorized arithmetic. Effective SIMD programming requires understanding the execution characteristics of packed arithmetic, lane behavior, instruction types, and memory-bandwidth limitations.

4.3.1 SIMD Arithmetic Concepts

Vectorized arithmetic computes the same operation on multiple data elements packed into one vector. Examples include:

- elementwise addition
- elementwise subtraction
- elementwise multiplication
- saturated arithmetic
- fused multiply-add
- horizontal operations (reductions)

These operations are executed in parallel across all lanes in the vector.

For instance, an AVX instruction processing eight 32-bit integers operates like eight scalar adds occurring simultaneously.

```
vpaddd ymm0, ymm1, ymm2    ; add eight 32-bit integers in parallel
```

4.3.2 Core Vector Arithmetic Instructions

Common SIMD arithmetic operations include:

- Integer arithmetic: vpaddd, vpsubd, vpmulld
- Bitwise logic: vpand, vpor, vpxor
- Floating-point arithmetic: vaddps, vmulps, vsubps, vdivps
- Fused operations: vfmadd132ps, vfmadd231ps, etc.
- Shift operations: vpslld, vpsrld

These instructions operate per lane, maintaining data parallelism across all elements.

4.3.3 Basic Example: Vectorized Add (AVX2)

```
.intel_syntax noprefix
.text
.global vec_add

# rdi = pointer to array
# rcx = number of blocks (each block = 8 integers)
vec_add:
    vmovdqu ymm1, [rel add8]      ; broadcast vector of 1's
```

```
.loop:
    vmovdqu ymm0, [rdi]          ; load 8 ints
    vpaddd  ymm0, ymm0, ymm1     ; add 1 to all elements
    vmovdqu [rdi], ymm0          ; store result

    add rdi, 32                  ; advance pointer
    dec rcx
    jnz .loop
    ret

.data
add8: .long 1,1,1,1,1,1,1,1
```

This loop performs eight increments per iteration with only one arithmetic instruction.

4.3.4 Vectorized Multiply-Add (Integer)

A frequent SIMD pattern combines multiplication and addition.

```
.intel_syntax noprefix
.text
.global vec_muladd

# rdi = ptr a
# rsi = ptr b
# rdx = ptr c
# rcx = blocks
vec_muladd:
.loop:
```

```
    vmovdqu ymm0, [rdi]
    vmovdqu ymm1, [rsi]
    vpmulld ymm2, ymm0, ymm1      ; elementwise multiplication
    vpaddd  ymm2, ymm2, [rdx]     ; addition with third vector
    vmovdqu [rdx], ymm2

    add rdi, 32
    add rsi, 32
    add rdx, 32
    dec rcx
    jnz .loop
    ret
```

The loop performs sixteen integer operations per iteration (8 multiplies + 8 adds).

4.3.5 Floating-Point Vectorized Arithmetic

Floating-point AVX instructions handle 4-wide XMM and 8-wide YMM vectors.

Vector Multiply (vaddps / vmulps)

```
.intel_syntax noprefix
.text
.global vec_fmul

# rdi = pointer
# rcx = number of blocks (each block = 8 floats)
vec_fmul:
    vbroadcastss ymm7, DWORD PTR [rel factor]
```

```
.loop:
    vmovups ymm0, [rdi]
    vmulps  ymm0, ymm0, ymm7     ; multiply 8 floats by constant
    vmovups [rdi], ymm0

    add rdi, 32
    dec rcx
    jnz .loop
    ret

.data
factor: .long 0x3f800000       ; float 1.0f
```

Floating-point operations take more cycles than integer arithmetic, but benefit significantly from SIMD parallelism.

4.3.6 Fused Multiply-Add (FMA) Operations

Post-2020 CPUs support efficient fused operations that combine multiplication and addition in one instruction, reducing latency and increasing throughput.

Float FMA Example

```
vfmadd231ps ymm0, ymm1, ymm2   ; ymm0 = ymm0 + (ymm1 * ymm2)
```

This eliminates an extra instruction and reduces rounding error.

4.3.7 Horizontal Vectorized Arithmetic (Reductions)

Reductions must combine elements across lanes. Without AVX-512 horizontal instructions, reductions are performed by:

1. operating structurally on lanes,

2. reducing within lanes,

3. shuffling or extracting halves,

4. combining final elements.

Example: 8-element Sum Reduction

```
.intel_syntax noprefix
.text
.global vec_reduce_sum

# rdi = pointer to array of 8 ints
vec_reduce_sum:
    vmovdqu ymm0, [rdi]           ; load 8 integers
    vextracti128 xmm1, ymm0, 1    ; extract high 128 bits

    vpaddd xmm0, xmm0, xmm1       ; combine lanes
    movhlps xmm1, xmm0            ; move hi to lo
    paddd xmm0, xmm1
    pshufd xmm1, xmm0, 0x1
    paddd xmm0, xmm1
    movd eax, xmm0                ; result
    ret
```

This performs a full 8-integer reduction using lane manipulation and packed adds.

4.3.8 Data Parallel Branchless Arithmetic

SIMD excels at eliminating branches through mask and blend operations.

Example: Clamp Values to Zero

```
.intel_syntax noprefix
.text
.global vec_clamp_zero

# rdi = pointer
vec_clamp_zero:
    vpxor ymm1, ymm1, ymm1          ; zero vector
    vmovdqu ymm0, [rdi]
    vpcmpgtd ymm2, ymm0, ymm1       ; mask = (x > 0)
    vpblendvb ymm3, ymm1, ymm0, ymm2
    vmovdqu [rdi], ymm3
    ret
```

Every lane is selected via blending rather than branching, increasing predictability and pipeline utilization.

4.3.9 Memory Bandwidth and Arithmetic Throughput

Vectorized arithmetic often becomes limited by memory bandwidth rather than execution throughput. Each 256-bit YMM load fetches 32 bytes, stretching L1 and L2 bandwidth.
To maintain performance:

- ensure data locality,

- prefetch upcoming cache lines,
- use streaming stores when appropriate,
- avoid scattered loads/stores.

Arithmetic instructions retire quickly; memory movement defines the bottleneck.

4.3.10 Summary

Vectorized arithmetic is the centerpiece of SIMD acceleration on Linux. By performing multiple arithmetic operations per instruction, SIMD kernels achieve significant improvements in throughput. Mastery of packed integer and floating-point arithmetic, understanding lane structure, leveraging fused operations, and designing memory-efficient loops are all essential. These concepts form the basis for advanced vectorization in AVX2 and AVX-512 domains and underpin high-performance numeric and data-processing software.
The next sections will examine masking, vector control-flow elimination, and AVX-512 expansion.

Chapter 5

Performance Measurement

5.1 rdtsc Usage

Accurate performance measurement at the instruction level requires a high-resolution, low-overhead timing source. On modern x86–64 Linux systems, the Time Stamp Counter (TSC) provides a 64-bit cycle counter that increments at a constant rate on processors supporting Invariant TSC. The rdtsc instruction exposes this counter directly to user space without a privileged transition, making it indispensable for profiling tight loops, evaluating SIMD kernels, and analyzing microarchitectural behavior.
However, rdtsc must be used correctly. Because it is not inherently serializing, interaction with out-of-order execution and speculative pipelines can produce incorrect results unless explicit ordering barriers are used.

5.1.1 Purpose of rdtsc

rdtsc returns a 64-bit cycle counter split across edx:eax. The instruction itself does not enforce ordering, so the CPU may execute surrounding instructions before or after it.

Accurate benchmarking therefore requires serialization barriers such as lfence.
rdtsc is most effective for:

- measuring tight loops and arithmetic kernels,
- evaluating effects of unrolling and SIMD,
- comparing scalar and vector throughput,
- quantifying latency of loads, stores, branches, and reductions.

Linux user space can invoke rdtsc directly with minimal overhead.

5.1.2 Incorrect Usage: Unserialized rdtsc

```
rdtsc                # not serialized
mov rbx, rax
```

Without serialization, the CPU may speculatively reorder instructions, producing unstable or artificially low timing results.

5.1.3 Correct Serialization with lfence

The recommended barrier on modern x86–64 CPUs is lfence, which ensures all prior instructions retire before rdtsc executes.

- Start Timestamp

```
.intel_syntax noprefix
lfence
rdtsc
shl rdx, 32
or  rax, rdx
mov r8, rax
```

- End Timestamp

```
.intel_syntax noprefix
lfence
rdtsc
shl rdx, 32
or  rax, rdx
mov r9, rax
```

lfence brackets the measured region precisely.

5.1.4 Minimal Benchmark Template

```
.intel_syntax noprefix
.text
.global benchmark_kernel

benchmark_kernel:
    lfence
    rdtsc
    shl rdx, 32
    or  rax, rdx
    mov r8, rax          # start

    # ------------------------------------------------
    # code under test
    # ------------------------------------------------

    lfence
    rdtsc
    shl rdx, 32
    or  rax, rdx
    sub rax, r8          # elapsed cycles
```

```
    ret
```

This structure guarantees that only the intended code region is timed.

5.1.5 Measuring a Scalar Loop

```
.intel_syntax noprefix
.text
.global measure_add_loop

# rdi = iteration count
measure_add_loop:
    lfence
    rdtsc
    shl rdx, 32
    or  rax, rdx
    mov r8, rax

    xor rcx, rcx
.loop:
    add rcx, 1
    dec rdi
    jnz .loop

    lfence
    rdtsc
    shl rdx, 32
    or  rax, rdx
    sub rax, r8
    ret
```

This establishes a scalar baseline for comparison.

5.1.6 Measuring a SIMD Kernel

```
.intel_syntax noprefix
.text
.global measure_simd_loop

# rdi = pointer
# rcx = block count
measure_simd_loop:
    vmovdqu ymm1, [rip + addvec]

    lfence
    rdtsc
    shl rdx, 32
    or  rax, rdx
    mov r8, rax

.loop:
    vmovdqu ymm0, [rdi]
    vpaddd  ymm0, ymm0, ymm1
    vmovdqu [rdi], ymm0
    add rdi, 32
    dec rcx
    jnz .loop

    lfence
    rdtsc
    shl rdx, 32
    or  rax, rdx
    sub rax, r8
    ret

.data
```

```
addvec:
    .long 1,1,1,1,1,1,1,1
```

This isolates SIMD arithmetic and memory behavior.

5.1.7 Common Measurement Pitfalls

Even with serialization, errors arise from:

- dynamic frequency scaling,
- cold caches on first iteration,
- page faults and TLB misses,
- branch predictor warm-up,
- instruction-cache alignment effects.

Stable measurements require warm-up runs and long iteration counts.

5.1.8 When rdtsc Is Insufficient

rdtscp partially serializes execution but still requires lfence for strict bracketing. Hardware counters (e.g., perf) provide deeper insight, but rdtsc remains the lowest-overhead mechanism for cycle-accurate timing in Linux user space.

5.1.9 Summary

When used with proper serialization, rdtsc provides precise, cycle-accurate timing for low-level benchmarking. Correct methodology is essential; misuse invalidates results. Throughout this book, rdtsc forms the foundation of performance analysis.

5.2 Cycles per Element (CPE)

Cycles per Element (CPE) normalizes performance by expressing how many CPU cycles are consumed per unit of work. It enables meaningful comparison across implementations, SIMD widths, and architectures.

5.2.1 Definition

```
CPE = total_cycles / elements_processed
```

CPE reflects computational density, memory behavior, and pipeline efficiency.

5.2.2 CPE Measurement Template

```
.intel_syntax noprefix
.text
.global measure_cpe_kernel

measure_cpe_kernel:
    lfence
    rdtsc
    shl rdx, 32
    or  rax, rdx
    mov r8, rax

.loop:
    mov eax, [rdi]
    add eax, 1
    mov [rdi], eax
    add rdi, 4
    dec rcx
    jnz .loop
```

```
    lfence
    rdtsc
    shl rdx, 32
    or  rax, rdx
    sub rax, r8
    ret
```

5.2.3 SIMD CPE Example

```
.intel_syntax noprefix
.text
.global measure_cpe_simd

measure_cpe_simd:
    vmovdqu ymm1, [rip + adds]

    lfence
    rdtsc
    shl rdx, 32
    or  rax, rdx
    mov r8, rax

.loop:
    vmovdqu ymm0, [rdi]
    vpaddd  ymm0, ymm0, ymm1
    vmovdqu [rdi], ymm0
    add rdi, 32
    dec rcx
    jnz .loop

    lfence
    rdtsc
```

```
    shl rdx, 32
    or  rax, rdx
    sub rax, r8
    ret

.data
adds:
    .long 1,1,1,1,1,1,1,1
```

5.2.4 Interpreting CPE

- Low CPE indicates compute-bound execution.
- Medium CPE indicates balanced compute/memory behavior.
- High CPE indicates memory-bound or branch-limited code.

CPE serves as a diagnostic signal, not merely a speed metric.

5.2.5 Summary

CPE is the central metric for evaluating low-level performance. Combined with serialized rdtsc, it reveals the true limiting factor of a kernel and guides optimization strategy.

5.3 Microbenchmark Techniques

Microbenchmarks isolate small instruction sequences to measure latency, throughput, and pipeline behavior. On Linux, they are essential for validating that optimizations behave as expected.

5.3.1 Core Principles

Effective microbenchmarks require:

1. precise timing boundaries,
2. warmed cache and predictor state,
3. large iteration counts,
4. minimal extraneous instructions,
5. prevention of dead-code elimination.

5.3.2 Repeated-Block Structure

```
kernel:
    # instructions under test
```

Repeated:

```
loop:
    kernel
    kernel
    kernel
    dec rcx
    jnz loop
```

5.3.3 Throughput Benchmark Example

```
.intel_syntax noprefix
.text
.global bench_add

bench_add:
    lfence
    rdtsc
    shl rdx, 32
    or  rax, rdx
    mov r8, rax

.loop:
    add rax, 1
    add rax, 1
    add rax, 1
    add rax, 1
    dec rdi
    jnz .loop

    lfence
    rdtsc
    shl rdx, 32
    or  rax, rdx
    sub rax, r8
    ret
```

5.3.4 Latency Benchmark Example

```
.intel_syntax noprefix
.text
.global bench_add_latency
```

```
bench_add_latency:
    lfence
    rdtsc
    shl rdx, 32
    or  rax, rdx
    mov r8, rax

.loop:
    add rax, 1
    dec rdi
    jnz .loop

    lfence
    rdtsc
    shl rdx, 32
    or  rax, rdx
    sub rax, r8
    ret
```

5.3.5 Summary

Microbenchmarking reveals true instruction behavior by isolating execution under controlled conditions. Correctly designed benchmarks expose throughput, latency, memory penalties, and SIMD scaling, forming the foundation for all performance conclusions in this book.

Chapter 6

Final Project

6.1 8× Faster Matrix Multiplication

Matrix multiplication is a canonical performance benchmark because it stresses nearly every component of a modern CPU: execution units, SIMD lanes, load/store bandwidth, register renaming, cache hierarchy, memory latency, and prefetch behavior. While a naïve implementation is easy to write, achieving even a fraction of peak floating-point throughput requires careful alignment between algorithmic structure and microarchitectural capabilities.

This section presents a hand-designed AVX2 microkernel for single-precision matrix multiplication (SGEMM). The objective is a performance level at least 8× faster than a naïve scalar triple-loop, achieved through register blocking, SIMD broadcasting, contiguous memory access, and fused multiply–add (FMA) instructions. The implementation is written in pure assembly using Intel syntax for Linux x86–64 and does not rely on the C standard library.

6.1.1 The Computational Problem

Matrix multiplication computes:

$$C[i][j] \mathrel{+}= A[i][k] \times B[k][j]$$

For three $N \times N$ matrices:

- Iteration count: N^3
- Floating-point operations: $2 \times N^3$ (one multiply and one add)
- Memory references: $O(N^3)$ unless optimized

This leads to a fundamental insight:

> Arithmetic is cheap; memory traffic is expensive.

The primary goal of SIMD optimization is therefore to maximize arithmetic work per byte loaded by reusing data in registers and caches as long as possible.

6.1.2 Why the Naïve Scalar Kernel Performs Poorly

A naïve scalar implementation typically:

- reloads elements of A and B repeatedly,
- updates C one element at a time,
- exhibits poor cache locality,
- fails to exploit SIMD width or ILP.

As a result, execution becomes memory-bound and dominated by stalls rather than computation. Scalar code cannot saturate SIMD execution units or FMA pipelines.

6.1.3 SIMD Strategy for an 8× Speedup

The AVX2 microkernel follows these principles:

1. Register Blocking
   Eight columns of C are accumulated in YMM registers.

2. Broadcasting Scalars from A
   Each scalar $A[i][k]$ is broadcast across a YMM register using vbroadcastss.

3. Contiguous Loads from B
   Eight contiguous floats from B are loaded per iteration.

4. Fused Multiply–Add
   Eight multiply–add operations are executed in a single instruction.

5. Minimal Memory Traffic
   C is loaded once and stored once per tile.

6. Pointer Arithmetic in GPRs
   General-purpose registers handle address updates efficiently.

7. Loop Unrolling
   Reduces branch overhead and improves ILP.

6.1.4 The 8-Element AVX2 Microkernel

The following kernel computes:

$$C[i][j \dots j+7]$$

for one row of A and eight columns of B.

```
.intel_syntax noprefix
.text
.global mmul8x_kernel

# rdi = &A[i][0]
# rsi = &B[0][j]
# rdx = &C[i][j]
# rcx = K

mmul8x_kernel:
    vmovups ymm0, [rdx]        # load C accumulator

.loop_k:
    vbroadcastss ymm1, DWORD PTR [rdi]  # broadcast A[i][k]
    vmovups      ymm2, [rsi]            # load B[k][j..j+7]

    vfmadd231ps ymm0, ymm1, ymm2        # C += A * B

    add rdi, 4                          # next A element
    add rsi, 32                         # next B row block

    dec rcx
    jnz .loop_k

    vmovups [rdx], ymm0                 # store C block
    ret
```

6.1.5 Why This Achieves an 8× Speedup

Each loop iteration performs:

- 8 floating-point multiplies,
- 8 floating-point adds,

- in a single FMA instruction.

A scalar loop performs only one multiply and one add per iteration. In ideal conditions, the SIMD kernel therefore approaches an 8× speedup. Actual results typically fall between 5× and 8× depending on memory bandwidth.

6.1.6 The Role of FMA

The instruction:

$$\text{vfmadd231ps dst, a, b}$$

computes:

$$\text{dst} = \text{dst} + (a \times b)$$

in one instruction. On modern CPUs:

- one-cycle throughput,
- eight lanes per YMM register,
- sixteen floating-point operations per cycle.

This enables near-peak FLOP throughput when the pipeline is fully fed.

6.1.7 Expanding to Larger Tiles

To build a full matrix multiplication, the microkernel is expanded into larger tiles such as 2×8, 4×8, or 8×8.

6.1.7.1 Example: 2×8 Tile

```
vmovups ymm0, [rdx]        # C[i][j..j+7]
vmovups ymm3, [rdx+32]     # C[i+1][j..j+7]

.loop_k:
    vbroadcastss ymm1, DWORD PTR [rdi]
    vbroadcastss ymm4, DWORD PTR [rdi+4]

    vmovups ymm2, [rsi]

    vfmadd231ps ymm0, ymm1, ymm2
    vfmadd231ps ymm3, ymm4, ymm2

    add rdi, 8
    add rsi, 32
    dec rcx
    jnz .loop_k

vmovups [rdx], ymm0
vmovups [rdx+32], ymm3
ret
```

Larger tiles amortize load costs, reduce branch frequency, and improve cache reuse.

6.1.8 Prefetching Strategy

To reduce memory latency:

```
prefetcht0 [rsi + 256]
prefetcht0 [rdi + 128]
```

Prefetch distances depend on CPU generation and loop structure and must be tuned empirically.

6.1.9 Memory Alignment and Layout

For best performance:

1. Matrices are 32-byte aligned.
2. A and C use row-major layout.
3. B is preferably block-transposed.
4. Tiles fit in L1 or L2 cache.

Misalignment increases load penalties and AGU pressure.

6.1.10 Loop Unrolling

Unrolling the inner k loop reduces branch overhead and dependency chains.

```
vbroadcastss ymm1, [rdi]
vmovups      ymm2, [rsi]
vfmadd231ps ymm0, ymm1, ymm2

vbroadcastss ymm1, [rdi+4]
vmovups      ymm2, [rsi+32]
vfmadd231ps ymm0, ymm1, ymm2
```

Greater unrolling improves throughput at the cost of code size.

6.1.11 Handling Edge Cases

When dimensions are not multiples of eight:

- full blocks use YMM registers,

- remainder columns use XMM registers,
- final scalars use fallback loops.

This preserves correctness without sacrificing vector performance.

6.1.12 Why the Speedup Is Sustainable

The kernel sustains high throughput because:

1. arithmetic intensity is high,
2. memory traffic is minimized,
3. B is accessed linearly,
4. FMA saturates SIMD execution units,
5. ILP keeps pipelines full,
6. bandwidth limits are respected.

6.1.13 Summary

This final project presented a complete AVX2 SGEMM microkernel achieving an 8× speedup over a naïve scalar implementation by:

- exploiting 8-wide SIMD,
- using FMA for computational density,
- applying register blocking,
- structuring memory access for cache efficiency,

- unrolling loops for ILP,
- prefetching to hide latency,
- enforcing proper alignment.

This microkernel represents the foundational building block used by high-performance numerical libraries and machine-learning runtimes on modern x86–64 Linux systems.

Conclusion

Performance optimization on modern x86–64 Linux systems is not a matter of applying isolated techniques, but of mastering the interaction between computation, memory, caching, branch prediction, speculation, vector execution units, and compiler behavior. This volume has shown that high-performance software emerges only when the programmer develops a unified understanding of microarchitectural realities and maintains explicit control over execution at the instruction level. Techniques such as loop unrolling, register blocking, branch elimination, software prefetching, and SIMD vectorization are most effective not as standalone optimizations, but as coordinated mechanisms that reshape algorithms to match the expectations of the CPU pipeline. Throughout this book, SIMD has appeared not merely as an optional acceleration feature, but as an architectural requirement for any serious performance-oriented workload. XMM, YMM, and ZMM registers represent distinct execution domains, each with its own throughput characteristics, penalties, and alignment constraints. Effective use of these domains demands careful control of VEX-encoded instructions, elimination of costly domain crossings, and disciplined management of register pressure. When these factors are respected, instructions such as vfmadd231ps, vpaddd, vmovdqu, and prefetcht0 transform conventional scalar loops into highly parallel pipelines capable of sustaining arithmetic throughput that approaches the theoretical limits of the hardware.

Raw computational throughput, however, is meaningless if memory cannot keep pace.

This volume has repeatedly demonstrated that memory—not arithmetic—defines the ultimate performance ceiling. Load–store bandwidth, cache locality, TLB behavior, and access regularity determine whether SIMD execution achieves peak throughput or collapses behind DRAM latency. Techniques such as software prefetching and cache-aware tiling restructure data flow so that expensive memory transfers are amortized across substantial in-register computation. Hand-written microkernels, including the AVX2 matrix multiplication kernel presented in the final chapter, illustrate how increasing arithmetic intensity is essential for converting memory-bound workloads into compute-bound kernels.

Performance measurement completes the optimization cycle. Without precise clocks and repeatable conditions, optimization devolves into guesswork. The disciplined use of serialized rdtsc, dependency chains for latency measurement, and carefully designed microbenchmarks for throughput analysis forms the foundation of reliable performance engineering. Every optimized kernel presented in this book was guided by such measurements, reinforcing the principle that meaningful optimization is inseparable from empirical observation.

Ultimately, performance optimization on Linux is an engineering discipline rooted in understanding the CPU as a deeply parallel and speculative machine. The techniques explored throughout this volume—from branchless arithmetic and SIMD reductions to loop transformations and fully optimized AVX microkernels—demonstrate that peak performance is achieved not through unnecessary complexity, but through clarity: the clarity of understanding hardware behavior and reshaping algorithms to align with it.

This volume concludes the SIMD and performance-optimization segment of the series, but the principles developed here extend naturally into all subsequent topics. Whether designing custom allocators, low-latency execution pipelines, cryptographic primitives, or kernel-level components, the same foundations apply: maximize locality, reduce stalls, avoid unpredictable branches, minimize memory traffic, saturate vector units,

and measure everything.

Performance is not accidental—it is the result of deliberate, low-level design. The tools and techniques presented in this book provide a durable foundation for building software that is not only functionally correct, but architecturally aligned with the full capabilities of modern x86–64 Linux systems.

References

Architecture and Microarchitecture Manuals

- Intel
  - Intel® 64 and IA-32 Architectures Software Developer's Manual (Volumes 1–3)
  - Intel® Optimization Reference Manual
  - Intel® Intrinsics Guide
  - Intel® Memory Latency Checker Documentation
- AMD
  - AMD64 Architecture Programmer's Manual (Volumes 1–5)
  - AMD Optimizing CPU Libraries Guide
  - AMD Zen Architecture Optimization Manual

These documents define instruction semantics, pipeline behavior, SIMD execution characteristics, cache hierarchy, and optimization constraints for modern x86–64 processors.

Books on Performance, SIMD, and Low-Level Optimization

- Computer Organization and Design: The Hardware/Software Interface — Hennessy & Patterson
- Computer Architecture: A Quantitative Approach — Hennessy & Patterson
- Structured Computer Organization — Andrew S. Tanenbaum
- Optimizing Software in C++: An Optimization Guide for Windows, Linux, and Mac Platforms — Agner Fog
- Modern Processor Design: Fundamentals of Superscalar Processors — John Paul Shen & Mikko Lipasti
- Algorithms for Modern Hardware — Sergey Slotin
- Programming Massively Parallel Processors — Hwu & Kirk
- High Performance Python / High Performance Computing — for additional perspective on vectorization and memory behavior
- Performance Analysis and Tuning on Modern CPUs — Igor Ostrovsky

These works provide theoretical foundations and practical insight into instruction-level parallelism, SIMD execution, memory hierarchy, and modern performance analysis.

Low-Level Linux and System Programming References

- Linux Kernel Development — Robert Love
- Linux Systems Programming — Robert Love

- The Linux Programming Interface — Michael Kerrisk
- Linux man-pages — system-call semantics and kernel behavior relevant to benchmarking

While not SIMD-specific, these references clarify Linux memory models, process scheduling, and system-level effects that influence performance measurement and microbenchmark reliability.

Technical Magazines and Journals

- ACM / IEEE Publications
  - ACM Queue
  - Communications of the ACM
  - IEEE Micro
  - IEEE Computer Architecture Letters
  - Journal of Instruction-Level Parallelism (JILP)
- Specialized Industry Publications
  - Intel Software Developer Zone Articles
  - AMD Developer Central Technical Articles
  - ACM Transactions on Architecture and Code Optimization (TACO)
  - USENIX ;login:

These publications regularly cover advances in CPU design, SIMD execution, compiler optimization, and empirical performance evaluation.

Authoritative Resources on SIMD and Microarchitecture

- Agner Fog's Optimization Manuals — instruction timing, port usage, branch prediction, and SIMD behavior
- InstLatX64 — instruction latency and throughput tables for modern CPUs
- uops.info — microbenchmark-based execution data for x86 instructions
- Linux Kernel Documentation — memory ordering, barriers, and CPU isolation
- RealWorldTech — historical and architectural deep dives
- OpenBLAS, BLIS, and Eigen Documentation — production-grade SIMD tiling and microkernel design
- LLVM and GCC Developer Documentation — auto-vectorization, backend behavior, and register allocation

These resources are widely referenced in professional performance engineering and microarchitectural analysis.

SIMD ISA and Instruction-Set Documentation

- Intel AVX, AVX2, and AVX-512 Instruction Set Extensions Programmer's Guide
- Intel FMA Instruction Set Reference
- Intel SSE/AVX Transition Penalty Documentation
- AMD XOP, FMA4, and SSE5 Documentation
- x86 Instruction Set Architecture References

These documents define the SIMD and floating-point instructions used throughout the AVX microkernels presented in this book.

Microbenchmarking and Performance Measurement Tools

- Linux perf Documentation
- OProfile Documentation
- Intel VTune Profiler Guides
- Linux PerfEvent ABI Documentation

They describe performance counters, cycle measurement, and CPU isolation mechanisms essential for validating low-level optimizations.

Open-Source SIMD and HPC Implementations (Conceptual References)

The following projects are not copied or reproduced, but represent the state of the art in SIMD microkernel and high-performance numerical design:

- BLIS Microkernel Framework
- OpenBLAS SGEMM and DGEMM Kernels
- Eigen Tensor and Matrix Multiplication Kernels
- xsimd SIMD Library
- Highway SIMD (Google)

- Vc SIMD Library

Their design principles strongly influence modern performance engineering practice.

Summary

The references listed in this chapter collectively form the technical foundation of Book 9 — Performance Optimization and SIMD. They define:

- SIMD instruction semantics and execution behavior,
- microarchitectural constraints and opportunities,
- principles of vectorization, blocking, and tiling,
- cache hierarchy and memory-system effects,
- performance measurement and microbenchmarking methodology,
- validated optimization strategies used in production systems.

The content of this volume builds upon these authoritative sources, transforming architectural knowledge into practical, hand-optimized AVX2 assembly suitable for high-performance Linux systems.