

When Does Manual SIMD Beat Compiler Auto-Vectorization?

A Kernel-Level Dot-Product Case Study for CPU Inference

Omer Isildak

ASELSAN

omerisildak@aselsan.com

Abstract—Dot products are a basic reduction in CPU inference. We compare four builds of the same float dot-product loop (scalar, compiler-optimized, compiler-reassociated, and hand-written AVX2) on a dedicated x86-64 server (AMD EPYC 8024P; boost disabled, performance governor, single pinned core) under GCC 15.2 and Clang 21, plus a separate two-variant ARM record (scalar and compiler-optimized, Apple Clang 21). On x86 we compare two ISA conditions. With `-march=native` (up to AVX-512), the reassociation-permitting builds reach the manual kernel’s median speed within a few percent at most sizes (GCC $0.96\text{--}1.07\times$, Clang $0.89\text{--}1.05\times$); with an equal-ISA AVX2 build, Clang still matches the manual kernel, while GCC trails it by up to about $2\times$ at mid sizes. The default-order builds remain about $8.8\text{--}17.4\times$ slower under both conditions. The trade-off is between the portability cost of intrinsics and the tolerance for reordered floating-point sums. These kernel measurements do not establish end-to-end inference speedups or model accuracy; flags, hardware, and methodology are recorded for reproducibility.

Index Terms—auto-vectorization, SIMD, AVX2, dot product, reproducible benchmarking, CPU inference

I. INTRODUCTION

Matrix-vector and matrix-matrix operations in CPU inference contain inner-product reductions. Systems such as llama.cpp [1] and oneDNN [2] motivate studying how these computations map to CPU instructions. We study the float dot product, a basic reduction kernel, because it shows how floating-point summation order and SIMD code generation interact, without the complexity of a full model.

In practice the question is whether a kernel needs architecture-specific intrinsics or only a compiler flag change. The default build alone cannot answer it, because preserving summation order can prevent a parallel reduction; the reassociated build isolates that policy constraint from code generation. The distinction matters even for a short loop: exact arithmetic is order-insensitive, but floating-point rounding makes summation order observable. Using several accumulators changes both the instruction structure and the numerical policy, so comparing only against the default build would conflate the two.

We ask when a hand-written AVX2 kernel outperforms a compiler-optimized build of the *same* loop, and why. Four variants use recorded flags and identical synthetic inputs,

following standard benchmarking practice [3]; the experiment measures kernel latency only, and transfer to an inference workload needs separate validation.

We make the following contributions:

- a reproducible benchmark and correctness suite (raw CSV, environment and CI records; the data will be released with a public repository);
- a methodology for kernels with different summation orders (dataset-specific tolerance, tail sweeps, sanitizers);
- a compiler-behavior result on x86 (GCC 15.2 and Clang 21): without FP reassociation the accumulation stays a serial chain; permitting reassociation lets the compiler vectorize it;
- assembly-level evidence and measurements that tie code generation to the dependency-chain and ILP analysis.

II. RELATED WORK

Vectorizer evaluation. Maleki et al. examined compiler vectorization coverage and its dependence on loop structure [4]; Nuzman and Zaks studied outer-loop vectorization for short-SIMD architectures [5]. Permission to vectorize does not ensure that the accumulation itself becomes parallel. We extend these studies by fixing the source loop and varying only the permitted FP transformations.

Reductions and SLP. LLVM documents FP reduction reassociation as an opt-in [6]. Look-Ahead SLP improves operand pairing for commutative chains of independent operations [7]. Our reassociated loop instead permits regrouping a sum, which changes rounding; commutation alone does not grant that permission. With binary32 arithmetic and round-to-nearest-even, $(2^{24} + 1) - 2^{24}$ evaluates to zero, while $2^{24} + (1 - 2^{24})$ evaluates to one; a compiler therefore cannot generally replace an ordered sum with a tree reduction without changing results. LLVM also documents target-specific ordered reductions, so this is not a universal prohibition [6].

Floating-point semantics. Because summation order changes results, FP contraction and reassociation require explicit policies [8]–[10]. We disable contraction throughout to separate reordered summation from FMA.

Benchmark methodology. Performance results are sensitive to setup: configuration can change which implementation looks faster [11] and noise makes single runs unreliable [12], so a benchmark is scientific only when the full setup and raw data are reported [13].

Kernel engineering. Classic matrix-kernel work informs the kernel structure: register blocking and multiple accumulators shorten dependency chains [14]; latency/throughput tables [15] and vendor reference material [16] provide the underlying model.

Position of this study. We isolate one reduction and compare default FP ordering, explicit reassociation, and manual AVX2 on identical inputs. The contribution is a reproducible case study that links timing, numerical checks, and assembly evidence, rather than a new vectorizer or matrix kernel. It examines a known FP constraint under two recorded ISA policies; end-to-end inference is out of scope.

III. METHOD

All four variants implement the same float dot product $\sum_{i=0}^{n-1} a_i b_i$ from one source tree with fixed, recorded flags: no global `-ffast-math` (except the dedicated reassociated variant) and no LTO; kernel calls live in separate translation units so the compiler cannot fuse across variants.

A. Variants

- **scalar:** the reference loop with auto-vectorization and FP contraction disabled at compile time (exact flags in Table I).
- **compiler-optimized:** the identical loop under default Release optimization: vectorization permitted, reassociation not forced.
- **compiler-reassociated:** the identical loop, compiled with FP reassociation allowed (flags in Table I, contraction still off); Clang’s `-ffast-math` adds further FP relaxations [17].
- **manual AVX2:** an 8-lane kernel with four independent accumulators, unaligned vector loads, and a scalar tail; FMA is deliberately excluded; built only on x86-64 and gated on runtime AVX2 (flags in Table I).

The manual kernel processes 32 elements per main-loop iteration using four independent 8-lane accumulators. It then handles any remaining 8-element blocks, combines the accumulators, sums their lanes, and handles at most seven scalar tail elements. This reduces dependence between successive additions, at the cost of a different grouping of products. The final lane reduction is included in the timing; the benchmark does not time only the vectorized main loop.

The compiler targets (scalar, compiler-optimized, and compiler-reassociated) are compiled with the native CPU target (`-march=native` on x86, `-mcpu=native` on ARM) unless `DOT_PRODUCT_NATIVE=OFF`; the manual kernel keeps a fixed `-mavx2` and binaries are not portable across CPUs. On the EPYC server the native flag permits AVX-512 (the Clang reassociated listing uses `zmm` registers), while the manual kernel stays 256-bit AVX2; we therefore also build an

equal-ISA configuration (`DOT_PRODUCT_ISA_AVX2=ON`) that compiles the compiler targets with `-mavx2` instead of `-march=native`, and report both conditions. Exact flags are exported per build (`compile_commands.json`); the verification record archives them with object disassembly and a source checksum (Section IV).

The two compilers receive different policy flags: GCC gets the explicit associative-math options in Table I; Clang gets the broader `-ffast-math`, with `-ffp-contract=off` re-applied (which does not undo other fast-math assumptions [17]). The Clang result therefore mixes several policy changes at once; the reduction structure is still what we study.

B. Benchmark procedure

The harness runs each variant on identical, deterministically generated inputs, verifies each result against a double-precision reference, then applies five warm-up calls and calibrates the iteration count by doubling until one timed block reaches 10 ms (cap 2^{24}); seven samples are recorded per size. Results pass a compiler barrier; input generation, allocation, and CSV output stay outside the timed region. Sizes: 256, 1024, 4096, 16384, 65536, 2^{20} ; buffers are allocated once and reused across timed blocks, so every timed call operates on resident data (the two arrays reach 8 MiB at the largest size). All variants run through one function-pointer table (equal call overhead), in a fixed order, a known limitation.

Inputs are periodic synthetic sequences, generated from the element index with periods 29 and 17 and scaled to contain positive and negative values. They provide a repeatable arithmetic workload, not a sample of model weights or activations. Correctness tail sweeps exercise sizes absent from the timing grid: the six measured lengths are all multiples of 32, so the performance record does not characterize irregular tails.

Each CSV row records a timed block and its iteration count; `ns_per_call` is elapsed nanoseconds divided by that count. For each compiler, we pool seven samples from each of two runs and take the median of the resulting 14 values per variant and size. Ratios use these medians, not a mean of per-sample speedups. The plotted min–max ranges describe observed spread, not confidence intervals; repeated blocks in two runs are not 14 independent machine-level replications.

C. Correctness and tolerance

Each variant is checked against a double-precision reference on the same dataset; because summation order differs, bitwise equality is not expected, and a result is accepted when its absolute deviation stays below a dataset-specific threshold $10^{-5} \cdot \max(1, \sum_i |a_i b_i|)$. This threshold is an acceptance policy, not a derived error bound: at $n = 2^{20}$ it is about 2.25, loose enough that even an all-zero result would pass, while the observed deviations at that size are at most 4.1×10^{-3} (scalar and default compiler builds), 2.2×10^{-4} (reassociated), and 1.6×10^{-4} (manual AVX2), against a reference value of about -0.126 . Because the magnitude-scaled threshold says little about accuracy under near-total cancellation, checks also require finite results, exact-size buffers, and tail sweeps

TABLE I
BUILD CONFIGURATION OF THE FOUR VARIANTS.

Variant	Key flags	Availability
scalar	-O3; vectorization disabled (clang -fno-vectorize -fno-slp-vectorize, gcc -fno-tree-vectorize); FP contraction off (-ffp-contract=off)	all
compiler-optimized	-O3; vectorization permitted, no FP reassociation, -ffp-contract=off	all
compiler-reassociated	-O3; FP reassociation allowed (GCC -fassociative-math -fno-signed-zeros -fno-trapping-math; Clang -ffast-math); -ffp-contract=off	all
manual AVX2	-mavx2 -mno-fma -ffp-contract=off	x86-64 with runtime AVX2

($n = 0..65$, offsets 0/1) under ASan/UBSan, and an additional well-conditioned fixture (all products positive, no cancellation) is checked against a relative bound $10^{-4} \cdot \sum_i |a_i b_i|$ (0.01% of the magnitude sum) at $n = 2^{16}$ and $n = 2^{20}$; all four variants pass. For this fixture the magnitude sum equals the reference magnitude, so the check enforces a meaningful relative error threshold and rejects an all-zero result. Exceeding it would call for investigation but would not by itself prove an implementation defect. The threshold is empirical, and the fixtures do not establish accuracy for arbitrary inputs.

D. Committed records

Raw CSVs, environment notes, and regeneration scripts live under `results/`; CI verifies correctness across GCC/Clang on x86-64 (AVX2 required) and arm64. The records are a two-variant ARM run (scalar and compiler-optimized, Apple Clang) and four-variant x86 sessions under GCC 15.2 and Clang 21 on two dedicated servers of the same model (AMD EPYC 8024P). The first session (2026-09-07) recorded the native build only; the second (2026-09-08), analyzed below, recorded the native and the equal-ISA AVX2 conditions with two pooled runs per compiler and condition. The first session is kept as an independent replication (its native per-cell medians agree with the second within at most about 6%, most cells within 1%) and the sessions are never pooled. A VM record from the same compiler series is archived for the environment comparison in the Discussion.

IV. RESULTS

We report the second session (2026-09-08): four-variant x86 measurements under GCC 15.2 and Clang 21 in two ISA conditions on the dedicated server described above. Table II, Table III, and Fig. 1 use the native condition; Table IV uses the equal-ISA AVX2 condition of the same session. Per-run medians agreed within a few percent at every cell (worst case about 5.3% under GCC and 1.8% under Clang across both conditions).

We draw three observations. First, compiler-optimized matches scalar at every size under *both* tested compilers on the pinned core: median runtimes differ by at most about 6% (GCC at $n = 256$, in the default build’s favor) and by under 2% elsewhere. Archived assembly listings of the same loop under the recorded flags show why: without FP reassociation the accumulation is never

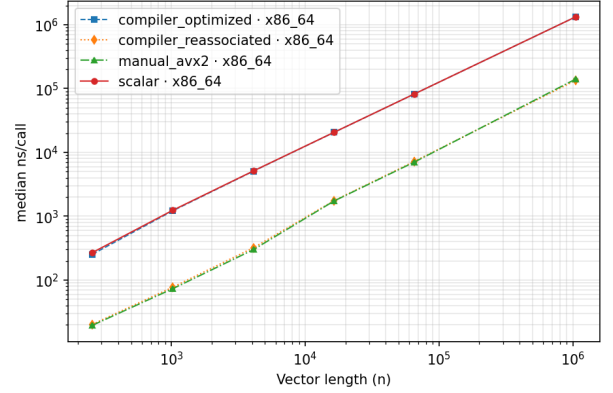


Fig. 1. Median ns/call per input size (GCC 15.2, native condition of the second dedicated-server session; two runs pooled; bands span the sample min and max). compiler-optimized overlaps scalar (single serial add chain); compiler-reassociated and manual AVX2 separate from this pair and differ by at most about 7%.

vectorized but stays one serial add chain of n add latencies (LLVM documents order-preserving reduction vectorization only on some targets [6]). The listings are compiled from the same sources with the measured toolchains (GCC 15.2 and Clang 21.1.8) under the recorded flags; the verification record also archives object-level disassembly, but listings and disassembly remain supporting evidence rather than a disassembly of the measured binary. Second, when summation may be reordered, both compilers vectorize the accumulation. Under the native (up to AVX-512) build the reassociated variant reaches the manual kernel within a few percent at most sizes: median reassociated/manual (R/M) ratios are $0.96\text{--}1.07\times$ under GCC and $0.89\text{--}1.05\times$ under Clang (the one outlier is Clang at $n = 2^{20}$, R/M $0.89\times$: the compiler build is about 12% faster). Under the equal-ISA AVX2 build the outcome splits by compiler (Table IV): Clang’s reassociated build still matches the manual kernel ($0.97\text{--}1.02\times$), but GCC’s trails it by $1.15\text{--}2.01\times$, with the largest gap at $n = 4096$ and a gap above $1.4\times$ from $n = 1024$ through $n = 65536$. The reassociated builds remain much faster than scalar in every condition: median scalar/reassociated ratios are $9.9\text{--}16.0\times$ (GCC) and $9.9\text{--}16.6\times$ (Clang) under the native build, and $8.0\text{--}11.7\times$ (GCC) and $8.8\text{--}17.1\times$ (Clang) under the equal-ISA AVX2 build. Third, without reassociation the

TABLE II
MEDIAN NS/CALL PER INPUT SIZE (SECOND DEDICATED-SERVER SESSION, NATIVE CONDITION, AMD EPYC 8024P, GCC 15.2; TWO RUNS POOLED: FOURTEEN SAMPLES PER CELL). SPEEDUP IS $\text{SCALAR}/\text{MANUAL AVX2}$.

n	scalar	compiler-opt.	compiler-reassoc.	manual AVX2	speedup
256	270	254	20	20	13.8×
1024	1243	1220	78	74	16.9×
4096	5117	5081	321	300	17.1×
16384	20571	20532	1751	1728	11.9×
65536	82400	82387	7242	7070	11.7×
1048576	1322439	1322369	134203	140285	9.4×

TABLE III
COMPILER/SCALAR RATIO OF THE `COMPILER-OPTIMIZED` VARIANT PER ENVIRONMENT (MEDIAN OF POOLED RUNS; X86 COLUMNS USE THE NATIVE CONDITION OF THE SECOND DEDICATED-SERVER SESSION). RATIOS NEAR 1 INDICATE SIMILAR RUNTIMES; WHETHER THE REDUCTION WAS VECTORIZED IS ESTABLISHED BY THE ASSEMBLY LISTINGS, NOT BY TIMING.

n	GCC 15.2 (x86, bare-metal)	Clang 21 (x86, bare-metal)	Apple Clang 21 (arm64, Mac)
256		0.941×	1.000×
1024		0.982×	1.000×
4096		0.993×	0.999×
16384		0.998×	1.000×
65536		1.000×	1.000×
1048576		1.000×	1.001×

scalar and default compiler builds remain about 8.8–17.4× slower than manual AVX2: the manual speedup peaks at about 17× ($n = 4096$) and is still about 9× at $n = 2^{20}$. Total time grows with n as expected; per-element cost is roughly constant across the two largest sizes for the serial-chain builds (about 1.26 ns per element at $n = 2^{20}$ under both compilers) and, at $n = 2^{20}$, about 0.13–0.14 ns per element for the vectorized builds. We make no statistical-equivalence claim; cache and memory bandwidth were not measured.

The ARM column is a separate within-machine comparison: its default compiler/scalar ratios of approximately 0.50–0.77 differ from the near-unity x86 ratios. It includes neither a manual AVX2 variant nor a reassociated timing record, so it shows target sensitivity rather than a ranking of ARM against x86.

V. DISCUSSION

The measurements align with the assembly analysis (Section IV): the archived default-order x86 accumulation stays a serial add chain; reassociation or manual ILP both break it, at the price of results that differ from the scalar reference within the tested tolerance. Whether the shrinking speedup at large n reflects a memory-bound regime is untested.

For this dot product, serial-chain builds remain about 8.8–17.4× slower than manual AVX2. Whether reassociated compiler builds close that gap depends on the ISA and the compiler: Clang matches the manual kernel under both the native and the equal-ISA AVX2 build, whereas GCC matches only under the native (AVX-512) build and trails by up to about 2× at mid sizes when limited to AVX2. The equal-ISA listings suggest a mechanism: GCC keeps a single vector accumulator (one `vaddps` per 32-byte chunk), so halving the vector width from the native build’s 16-lane registers to 8-lane AVX2

doubles its serial accumulate chain, while the manual kernel and Clang’s build use four independent accumulators. This is consistent with GCC’s reassociated/manual ratio moving from about 1.0 (native) to about 2.0 (AVX2) at mid sizes, but it is a dependency-structure reading of the listings, not a counter-verified attribution; `-march=native` also enables native scheduling, so the switch changes more than vector width alone. These results suggest checking FP policy and the targeted ISA before writing intrinsics for an inference reduction. Transfer to a full model is untested: operator fusion, data layout, precision, and other operators can change the benefit, and kernel tolerance does not establish model accuracy. In practice the numerical policy should be fixed first: compare compiler builds under that policy, then inspect the remaining bottleneck before adding intrinsics. If reordered sums are unacceptable, the manual kernel is not a drop-in alternative to the ordered loop; if they are acceptable, the close timings favor testing the compiler-generated build before committing to a separate ISA-specific code path. Neither outcome makes manual SIMD clearly better or clearly unnecessary.

A. Environment sensitivity (VM vs. dedicated server)

An earlier four-variant record produced with the same harness and compiler series on a rented VM is archived under `results/`; its qualitative pattern matched, but quantities differed: the worst-case spread between pooled runs was about 10.6%/9.1% (GCC/Clang) on the VM versus 2.3%/3.8% on the first dedicated-server session; scalar/manual at $n = 2^{20}$ was 4.37×/4.62× versus 8.9×/8.6×; and the small-size GCC reassociated/manual ratio was 1.54–1.82× (manual ahead) versus at most about 1.10× on the first dedicated-server session; the “manual leads at small n under GCC” observation does not reproduce on the dedicated server. Because the host microar-

TABLE IV

EQUAL-ISA AVX2 CONDITION OF THE SECOND DEDICATED-SERVER SESSION (COMPILER TARGETS BUILT WITH `-mavx2`, THE MANUAL KERNEL’S WIDTH; TWO RUNS POOLED, FOURTEEN SAMPLES PER CELL). R/M = MEDIAN REASSOCIATED/MANUAL RATIO; S/M = MEDIAN SCALAR/MANUAL RATIO.

n	GCC R/M	Clang R/M	GCC S/M	Clang S/M
256	1.15×	0.97×	13.4×	14.1×
1024	1.56×	0.99×	17.0×	17.0×
4096	2.01×	1.02×	17.1×	17.3×
16384	1.46×	1.01×	11.8×	11.9×
65536	1.47×	1.00×	11.8×	11.7×
1048576	1.18×	1.00×	9.4×	8.8×

chitecture changed with the environment (the VM reported a QEMU “Haswell” guest model), these differences are not attributable to virtualization alone; absolute VM numbers are not portable. The native ISA also changed: the VM exposed a Haswell model, whereas the dedicated servers permit AVX-512. This comparison therefore cannot separate virtualization, microarchitecture, and vector-width effects.

VI. CONCLUSION

We studied the float dot product, a basic reduction in CPU inference. With reassociation enabled, Clang matches the manual kernel under both native and equal-ISA AVX2 builds, while GCC matches only under the native (AVX-512) build and trails up to about $2\times$ at mid sizes under AVX2; serial-chain builds remain $8.8\text{--}17.4\times$ slower. The results suggest checking FP policy before investing in intrinsics, subject to the limitations below. End-to-end inference benefit and model accuracy remain unmeasured.

A. Future Work

The next step is to test whether the observed FP-policy trade-off transfers to a CPU inference operator. We plan to extend the benchmark to a small GEMV-based dense layer, where each output $y_j = \sum_i W_{ji}x_i + b_j$ contains a dot-product reduction. The comparison would retain explicit FP policies, add randomized measurement order, and report full operator latency and output error against a higher-precision reference. A later small-model experiment would assess end-to-end latency and task-level accuracy on a stated dataset. Whether kernel gains survive data reuse and surrounding operators, and whether reordered sums meet accuracy requirements, does not follow from the present measurements.

VII. LIMITATIONS

- Records cover one ARM machine (two variants) and two x86 dedicated servers of the same model (four variants per compiler). The analyzed session uses the second server alone; the first is an independent native replication. Results from different machines or sessions are not merged, and only within-record ratios are treated as evidence: the record is controlled but single-machine.
- Compiler targets are built with `-march=native` (up to AVX-512 on this server) or, in the separate equal-ISA build, with `-mavx2`. Because `-march=native` also

enables native scheduling, the comparison is a native-target vs. AVX2-target build comparison rather than a pure vector-width isolation; the two conditions are reported side by side and are not pooled.

- The reassociated variant relaxes IEEE-754 ordering (GCC `-fassociative-math -fno-signed-zeros -fno-trapping-math`; Clang `-ffast-math` [17], which enables further FP relaxations); its results stay within the tested tolerance but are not bitwise equal to the other variants.
- Variants are measured in a fixed order, so thermal and order bias are possible (randomized order is later work); the tolerance policy is dataset-specific and not a general accuracy guarantee, and NaN/Inf and signed-zero semantics are out of scope; FMA is deferred because FP contraction changes rounding.
- Native ARM results are not x86 claims, and CI timings are not used as results (shared runners are noisy).
- The performance grid covers one synthetic input family and six lengths divisible by 32. Irregular tails are checked for correctness, but not characterized for performance; no model-level validation, cache-counter study, or statistical equivalence test was performed.

REFERENCES

- [1] G. Gerganov, *llama.cpp: LLM inference in C/C++*, <https://github.com/ggml-org/llama.cpp>, accessed 2026.
- [2] Intel Corporation, *oneAPI Deep Neural Network Library (oneDNN)*, <https://github.com/uxlfoundation/oneDNN>, accessed 2026.
- [3] D. Bakhvalov, *Performance Analysis and Tuning on Modern CPUs*, 2nd ed., 2024, PDF freely available at <https://github.com/dendibakh/perf-book>, accessed 2026.
- [4] S. Maleki, Y. Gao, M. J. Garzaran, T. Wong, and D. A. Padua, “An evaluation of vectorizing compilers,” in *Proc. Int. Conf. on Parallel Architectures and Compilation Techniques (PACT)*, 2011, pp. 372–382.
- [5] D. Nuzman and A. Zaks, “Outer-loop vectorization: revisited for short SIMD architectures,” in *Proc. Int. Conf. on Parallel Architectures and Compilation Techniques (PACT)*, 2008, pp. 2–11.
- [6] “LLVM vectorization documentation (loop vectorizer, reductions),” <http://llvm.org/docs/Vectorizers.html>, accessed 2026.
- [7] V. Porpodas, R. C. O. Rocha, and L. F. W. Góes, “Look-ahead SLP: auto-vectorization in the presence of commutative operations,” in *Proc. Int. Symp. on Code Generation and Optimization (CGO)*, 2018, pp. 163–174.
- [8] D. Goldberg, “What every computer scientist should know about floating-point arithmetic,” *ACM Computing Surveys*, vol. 23, no. 1, 1991.
- [9] IEEE, *IEEE Standard for Floating-Point Arithmetic*, IEEE Std 754-2019 (revision of IEEE Std 754-2008), 2019.
- [10] N. J. Higham, *Accuracy and Stability of Numerical Algorithms*, 2nd ed., SIAM, Philadelphia, PA, 2002.

- [11] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney, “Producing wrong data without doing anything obviously wrong!,” in *Proc. Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2009, pp. 265–276.
- [12] C. Curtsinger and E. D. Berger, “STABILIZER: statistically sound performance evaluation,” in *Proc. Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013, pp. 219–228.
- [13] T. Hoefler and R. Belli, “Scientific benchmarking of parallel computing systems: twelve ways to tell the masses when reporting performance results,” in *Proc. Int. Conf. for High Performance Computing, Networking, Storage and Analysis (SC)*, 2015, pp. 73:1–73:12.
- [14] K. Goto and R. A. van de Geijn, “Anatomy of high-performance matrix multiplication,” *ACM Trans. Math. Softw.*, vol. 34, no. 3, 2008.
- [15] A. Fog, “Software optimization resources: optimizing software in C++ and instruction tables,” <https://www.agner.org/optimize/>, accessed 2026.
- [16] Intel Corporation, *Intel 64 and IA-32 Architectures Optimization Reference Manual*, vol. 1, doc. 248966, 2024, <https://www.intel.com/content/www/us/en/content-details/821612/intel-64-and-ia-32-architectures-optimization-reference-manual-volume-1.html>, accessed 2026.
- [17] “Clang compiler user’s manual: controlling floating-point behavior,” <https://clang.llvm.org/docs/UsersManual.html>, accessed 2026.