

A Hardened, Memory-Bounded C++20 Engine for High-Throughput Omics

Christopher Alexander Perez

Abstract

Genomics preprocessing and epigenomic analysis run on tool chains that decompress, materialize and re-read an intermediate file at nearly every stage boundary, and on interpreted wrappers whose failure modes are inherited from the C libraries beneath them rather than audited. I present *pto-core*, five independent C++20 engines: a streaming Poisson peak caller, a memory-bounded fragment-QC toolkit, a CUT&Tag/CUT&RUN signal profiler, a lock-free FASTQ streaming engine, and a cache-aligned sparse-matrix and approximate-*k*NN library for single-cell RNA-seq. All five are built on three non-negotiable properties: bounded heap execution independent of input size, fail-closed parsing (a corrupt record is a refusal, never a plausible default), and correctness demonstrated under adversarial sanitizer verification rather than assumed from a green test suite.

Across 7 workloads measured against the tools they replace (samtools, bedtools, deeptools, fastp), *pto-core* is $4.1\times$ to $56.2\times$ faster (median $35.1\times$) at 0.4% to 101.2% of the reference tool’s peak resident memory, and every measured answer agrees with the reference either exactly or within a stated, physically motivated tolerance (Table 3, Table 4). Seven adversarial verification passes found and fixed 74 confirmed defects across the five modules (Table 2). Most of those were not crashes; they were silent-wrong-answer and denial-of-service classes, each closed with a regression test and a sanitizer-clean re-verification.

I report this work with the same discipline I built into it, which means stating two results as split decisions rather than rounding them into successes. The streaming peak caller (*pto-peaks*) reproduces MACS3’s summit locations to a 1 bp median offset and its peak-strength ranking at Pearson $r = 0.990$, but it calls systematically narrower intervals (median 96 bp against MACS3’s 275 bp), so the pre-registered 0.98 interval-overlap parity gate is **not met** (mean IoU 0.458). That gap is documented here as a boundary-convention difference; it is not presented as a passing regression test. The sparse-matrix engine’s HNSW backend processes 1.3 million cells in 37.1 s at 1.01 GB, closing a feasibility gap against an extrapolated 1.6-hour exact computation, but the comparative claims against `scanpy.pp.neighbors()` that motivated the module remain explicitly unmeasured. Source, build instructions and every report cited here are available under the MIT license at <https://github.com/chrisaperez/pto-core>.

1 Introduction

1.1 Motivation

I built *pto-core* out of a specific frustration, and I state it plainly because it shaped every architectural decision that follows rather than serving as a preface to them. A large fraction of production bioinformatics software is a thin interpreted orchestration layer (Python or R) wrapped around a C or C++ core that the orchestration layer’s author did not write and frequently has not read, invoked through a shell pipeline that materializes an intermediate file at every stage boundary. The observable properties that decide whether such a tool survives contact with real data, namely its memory footprint, its behavior on malformed input and its thread scaling, are therefore not designed at all. They are inherited from whichever library happened to be linked. This means that the most important performance and correctness characteristics of the software are accidental properties of its dependency graph. That is not a tuning problem: it is an architectural one.

Three consequences of that regime recur, and this project set out to refuse each of them structurally rather than patch them case by case. The first is memory bloat that scales with input size when nothing in the algorithm requires it, typically because an intermediate file is fully materialized where a stream would do. The second is parsers that fail *open*: a corrupted field becomes a plausible default, a truncated file becomes a short but valid one, and a partially read stream is treated as a clean end-of-file. The third is pipelines that fail *mid-run* and non-deterministically, in a way a retry masks rather than explains. The second of these is the one that should alarm a working scientist most, because its output is not an error. Its output is a number, and the number is wrong.

I want to be careful about the scope of that critique. None of it is a claim about any specific named tool’s internals, and I make no such claim without a reproduced artifact to back it; that is the same standard this document holds its own numbers to (Section 5). The claim is about a failure *class*, and the best evidence I can offer for its reality is adversarial evidence against my own work. Building five engines *deliberately* against these three patterns still produced 74 confirmed instances of them before sustained sanitizer-driven verification found and closed each one (Section 4). The lesson I draw is not that any particular library is unusually bad. It is that this class of defect is the default outcome of writing performance-path C++ without an adversarial verification discipline applied to it continuously, and that the discipline, not the raw throughput, is the harder and more transferable half of this work.

1.2 A convergence of interests

The position I take here sits at an intersection whose components are individually common and jointly rare: hardware-aware systems programming (cache-line layout, runtime SIMD dispatch across NEON, AVX2 and AVX-512BW, lock-free single-producer/single-consumer queues); a verification discipline closer to formal-methods practice than to conventional unit testing (sanitizer-driven fuzzing against explicit fail-closed invariants, differential testing against arbitrary-precision references); and computational biology’s own domain constraints (BGZF block framing, 0-based half-open coordinates, and the FRiP and Benjamini–Hochberg statistics by which a result is judged). Each of those disciplines has a mature literature. Very little software sits in all three at once.

I propose that the correct response to the status quo is to stop treating bioinformatics tooling as disposable scripting and start treating it as safety-critical systems engineering. The justification is not aesthetic. A peak caller’s output becomes a figure, the figure becomes a claim, and the claim outlives every detail of the pipeline that produced it. Software whose silent failure mode is a plausible wrong number carries the same class of risk as an unaudited control system; the difference is only how long it takes for the failure to surface. Each of the five modules below is small enough to read end to end, and that is a deliberate design constraint rather than a happy accident: a correctness claim about a sequencing-scale reduction is only as credible as the code making the claim is auditable.

1.3 Design ethos and contributions

Three properties hold across all five modules, not as aspirations but as invariants with regression tests behind them:

- **Bounded heap execution.** Peak resident memory is a property of the pipeline topology (buffer-pool size, ring capacity, batch size) and not of input size. `fastq_stream`’s peak RSS is 10 MB independent of file size (§3.4); `genomic_toolkit` processes fragments in 4096-record, 16-byte-per-record batches with no sort and no intermediate file (§3.2).
- **Fail-closed parsing.** A malformed record is a refusal, not a default. The rule is repository-wide: a corrupt MAPQ column must not silently pass a quality filter, and a truncated BGZF block must not silently report a shorter library as a complete one. It is also the single most frequently rediscovered defect class in the audit trail (Section 4); the same failure to check a parse result was found and fixed independently in *four* different modules across three separate review passes.
- **Verification over assertion.** Every module ships a dependency-free test suite (no GoogleTest, no Catch2, because these tools must build on a bare cluster that has a compiler and nothing else) and has been driven under AddressSanitizer (Serebryany *et al.*, 2012) and Undefined-BehaviorSanitizer with `-fno-sanitize-recover=all`, under mutation fuzzing, and, where concurrency is

Table 1: The five `pto-core` modules. Each is a self-contained CMake project with its own tests and install rules; there is no shared runtime or code layer between them by design.

Module	Role
<code>pto-peaks</code>	Streaming Poisson peak calling
<code>genomic_toolkit</code>	Fragment size/dup/FRiP QC
<code>cuttag_profiler</code>	CUT&Tag/CUT&RUN signal matrices
<code>fastq_stream</code>	In-memory FASTQ QC and trimming
<code>scrna_matrix</code>	Sparse CSR + exact/HNSW k -NN

load-bearing, under ThreadSanitizer (Serebryany and Iskhodzhayev, 2009).

The contributions, each backed by a citable artifact in the repository, are as follows:

1. Five independently buildable C++20 engines covering streaming peak calling, fragment QC, chromatin signal profiling, FASTQ preprocessing and single-cell sparse linear algebra, unified by a deployment constraint (run on the host that already holds the data) rather than by a shared code layer.
2. A cross-tool benchmark harness that reports speed, peak memory *and* answer agreement together, so that a speedup is never reported over a different answer (Section 5).
3. An adversarial verification record covering 74 confirmed defects across seven passes (Table 2), maintained as a living document rather than a one-time disclosure. It includes four cases in which an already-fixed defect class recurred in a sibling code path, which I analyze as a methodological finding in its own right (Section 4).
4. A quantified concordance study against the field’s incumbent peak caller that reports a pre-registered parity gate as **not met**, with the measured shape of the disagreement, rather than redefining the gate after the fact in order to pass it (Section 5.4).

To see how these principles interact, we begin by defining the five engines and the deployment constraint that unifies them, then work through the mechanism of each in turn (Section 3). Next, we examine the verification methodology that produced the defect record and the invariants that now prevent its recurrence (Section 4). We then evaluate the engines against the tools they replace, including the one concordance study whose pre-registered gate was missed (Section 5). Finally, I set out what this system does not yet establish (Section 7).

2 System Overview

All five modules are C++20, built with CMake, and independently buildable with every other module deleted. What unifies them is a deployment constraint rather than a shared

abstraction: each tool is designed to run on the machine that already holds the sequencing data, with no service dependency and no network egress required for the computation itself. An earlier iteration introduced a shared top-level `include/python` layer across modules. It was reverted, and the repository’s build documentation records the absence of such a layer as a deliberate non-goal rather than an oversight. Table 1 summarizes the five; Section 3 takes each in turn.

3 Architecture and Systems Principles

3.1 pto-peaks: streaming Poisson peak calling

`pto-peaks` calls narrow peaks from a coordinate-sorted fragment BED or BAM in a single pass, with no sort and no intermediate `bedGraph`. Three components do the work: a per-base pileup feeding three *centred* sliding background windows (1 kb, 5 kb and 10 kb, held as fixed-capacity rings rather than resized buffers), a SIMD-vectorized Poisson tail-probability kernel, and an `IDLE/IN_PEAK/SUMMIT_SEARCH/CLOSING` state machine that consumes the scored bases and applies Benjamini–Hochberg FDR control across all candidates.

Two properties are architecturally load-bearing rather than incidental optimizations. The first is that the background windows are centred and truncated at contig ends. A trailing window would judge a peak against its own rising edge and clip every call at its 3’ end; an untruncated centred window at a contig boundary would divide by background that does not exist. The second concerns where vectorization is possible at all. The state machine cannot be vectorized, because the state at base $i+1$ depends on the state at base i , but the Poisson evaluation feeding it can: bases are queued and scored in a batch through one call into a SIMD kernel (scalar, NEON, AVX2 or AVX-512BW, selected once at startup by runtime CPUID dispatch rather than by a compile-time `-march` flag that would SIGILL on any machine other than the one that built the binary). Bases that provably cannot clear the significance cutoff never enter the queue, admitted by a closed-form bound on the Poisson tail’s median. In other words, the sequential part of the algorithm stays sequential and only the arithmetic goes wide.

The kernel contains no `libm` call anywhere, and that is deliberate: `log`, `exp` and `log1p` are open-coded from their series expansions so that all four ISA variants have something to vectorize, since no standard math library portably exposes a vector transcendental entry point. Numerical degeneracy fails closed. A zero, negative, NaN or infinite background λ returns $p = 1$ rather than the naive evaluation’s $p \rightarrow 0$. The direction of that choice is the entire point: the naive form reports every centromere, assembly gap and unmeasured window as the most significant peak in the genome. A caller that ranks assembly gaps above real biology is not slightly miscalibrated. It is inverted.

BAM input is decoded directly from BGZF and alignment records by byte-shift arithmetic rather than `reinterpret_cast`, because BAM packs `int32` fields at whatever offset the previous record ended on, which leaves

them routinely misaligned; the cast is undefined behavior and `-fsanitize=alignment` reports it as such. The per-block CRC32 check is retained rather than skipped for speed. A corrupt block that happens to inflate into plausible alignment records is precisely the silent-wrong-answer failure this project exists to refuse.

3.2 genomic_toolkit: memory-bounded fragment QC

`genomic_toolkit` computes fragment size distributions, duplicate rate and FRiP (Fraction of Reads in Peaks) in one streaming pass over a coordinate-sorted BAM, BEDPE or fragment BED, replacing a shell pipeline that would otherwise sort, spill to disk and make several full passes over the data. Everything downstream of the reader speaks a 16-byte, coordinate-only `Fragment` struct, handed to consumers in 4096-record batches (64 KiB, and therefore L2-resident) that are invalidated as soon as the callback returns. Nothing is ever materialized as text.

Duplicate detection needs no global state on coordinate-sorted input. Two fragments are duplicates when their 5’ ends agree, and on sorted input every fragment sharing a start position arrives contiguously, so the only state required is a set keyed on (end, orientation) that lives for exactly one start position. This means that memory tracks the depth of the deepest pile-up and never the size of the file: a 200 GB BAM and a 200 MB BAM have the same resident footprint here. FRiP is then computed as a reduction against peaks merged into a disjoint ascending cover at load time and queried by binary search. Merging is not a performance trick. It is the definition of the statistic: an unmerged peak file queried by naive interval intersection double-counts a fragment spanning overlapping peaks, and can report a FRiP above 1.

The module states its divergences rather than absorbing them. Duplicate identity uses the *aligned* fragment span where Picard `MarkDuplicates` uses the unclipped 5’ position, so a pair whose two copies were soft-clipped differently is one duplicate to Picard and two distinct fragments here; there is no optical-duplicate distinction and no UMI awareness, which makes `estimate_library_size()` a lower bound on library complexity rather than a point estimate. A breaking change is likewise stated rather than hidden: following an audit fix, column 5 of a fragment BED is read as a 10x-style read-pair count only under an explicit `--with-counts` flag. In a BED5 or BED6 that column is a score (`bedtools bamtobed` (Quinlan and Hall, 2010) writes MAPQ there), and reading it unconditionally had silently multiplied every reported statistic by the score column’s value. Finally, all 64-bit accumulators (fragment-base totals, size sums) are checked additions rather than raw arithmetic. That guard was added after a crafted input, reachable from as few as 4,612 lines under `--max-length 0`, was measured driving the signed accumulator past 2^{63} (Section 4).

3.3 cuttag_profiler: reference-point signal profiling

cuttag_profiler computes CUT&Tag and CUT&RUN (Kaya-Okur *et al.*, 2019) reference-point signal matrices (loci $\times$ bins) directly from an indexed BAM in one pass. It replaces the two-stage deeptools pipeline (Ramírez *et al.*, 2016), in which bamCoverage converts the BAM into a genome-wide bigWig and computeMatrix then windows that track, materializing an intermediate at genome scale even when the loci of interest cover a small fraction of the genome. Work is distributed one locus per task across a worker pool, block partitioned so that each worker writes a contiguous stripe of the output matrix and the write path needs no synchronization at all. The htstlib (Danecek *et al.*, 2021) index and header are immutable once loaded and are shared read-only, while each worker leases its own htstFile handle from a pool, because decompression state is mutable per handle.

One deliberate anti-pattern is worth recording, since it looks like a missed optimization. htstlib offers a thread pool that parallelizes BGZF block inflation, and the intuitive configuration is one pool sized to the machine. Here that is actively harmful: with per-region parallelism already saturating the cores, routing every block through a single shared pool serializes inflation behind that pool’s queue and adds a hand-off per block. Rather than merely adding threads wherever a library offers them, the profiler places parallelism at the level where the work actually divides, which is one locus per task, and lets each handle inflate on the thread that owns it.

Two correctness properties were closed by the 2026-09-11 audit (§4) and are architecturally central rather than incidental bug fixes. First, regions on a contig absent from the BAM are now excluded from the column mean, from BPM’s summed signal and from scaling, while remaining in the output matrix and carrying their names. Previously they were averaged in as a missing value of zero, which diluted the meta-profile in direct proportion to how many regions were skipped. The resulting curve was smooth, plausible and wrong by a constant factor the user had no way to see. Off-contig bins inside a *kept* window are still treated as missing, which matches deeptools’ own `--missingDataAsZero` semantics. Second, bin geometry (`width  $\times$  bin_index`) is bounded so that every product fits in `int64_t`: the window ceiling is `INT64_MAX` divided by the 10,000,000-bin limit, approximately 922 Gbp, and it is enforced identically at the CLI and at the `/api/profile` HTTP handler. That bound exists because an unbounded window was found to overflow signed 64-bit arithmetic and write wrong bin offsets in a Release build with no diagnostic whatsoever.

The interactive dashboard is an embedded HTTP server, bound to 127.0.0.1 by default, serving assets compiled into the binary under a strict Content-Security-Policy. It is structurally excluded from the multi-tenant cloud deployment of this codebase: a `-DPTO_CLOUD_BUILD=ON` build compiles the server out of the binary entirely, verified by the absence of `socket`, `bind` and `listen` symbols in the resulting image. The reasoning is that a per-run session token bound to one op-

erator’s browser is the wrong isolation primitive for a service whose isolation boundary is one ephemeral compute task per job. In other words, the safest way to secure a listening port in that deployment is to ensure it cannot be compiled into existence.

3.4 fastq_stream: lock-free in-memory FASTQ streaming

fastq_stream performs single-pass FASTQ quality control, adapter removal and quality trimming entirely in memory, streaming clean reads directly into an aligner through a named pipe. It eliminates the intermediate compressed FASTQ that a conventional `zcat`, `trim` and `gzip` pipeline writes and then reads back, whether the trimming stage is Trimmomatic (Bolger *et al.*, 2014), `fastp` (Chen *et al.*, 2018) or anything else. The engine is a five-rank pipeline (reader, an optional pool of inflaters, a record assembler, a pool of quality-control workers, and a writer) connected by genuinely single-producer/single-consumer lock-free ring buffers. The word “genuinely” is carrying weight. An SPSC ring is correct only with exactly one producer thread and one consumer thread, and a shared queue drained by a worker pool is neither; so the topology is arranged such that no ring is ever touched by more than two threads. In other words, the queue is fast because the topology guarantees its precondition, not because a weaker algorithm was tuned until it looked fast.

Head and tail indices are separated by `std::hardware_destructive_interference_size` so that the producer’s write and the consumer’s read never contend for the same cache line, and each side caches the opposite index in thread-private storage so a steady-state operation never reads the other core’s line at all. Measured inter-stage hand-off latency is 14 ns at 70 M items per second (module README; see §7 on the provenance of these single-host figures). Chunk order is preserved without a reorder buffer: chunk k is dispatched to lane $k \bmod D$, each lane preserves order internally, and a round-robin collector recovers stream order for free. This holds because a worker emits exactly one output buffer per input chunk, which trimming guarantees since it only ever shrinks records.

The most consequential architectural finding in this module is that *input framing, not the QC arithmetic, determines whether decompression can be parallelized at all*. `libdeflate` (Biggers, 2016) exposes no streaming interface; it inflates a complete gzip member into a caller-sized buffer, which is the source of its speed rather than an API oversight. A single-member plain gzip stream, which is what `bc12fastq`, `DRAGEN` and `gzip` itself produce, has exactly one member spanning the whole file, so no interior DEFLATE block boundary can be located without inflating everything before it. Parallel decompression of such a stream is therefore structurally impossible, for any tool, no matter how many cores it is given. BGZF, the SAM/BAM block-gzip framing, is instead a sequence of independent members that each declare their own compressed size and expand to at most 64 KiB, so members can be located by header arithmetic and inflated concurrently.

Measured on this engine, BGZF sustains 914 MB/s of compressed throughput against 372 MB/s for plain gzip on identical content. That $2.5\times$ difference costs nothing to obtain: it follows from running bgzip rather than gzip once, at the moment the FASTQ is generated, and it then benefits every downstream consumer of that file permanently. The bottleneck was never the arithmetic. It was the container.

Peak resident memory is structurally bounded by $(\text{lanes} \times \text{slots} + \text{in-flight}) \times 256 \text{ KiB}$ and is independent of input size, measured at 10 MB on a 200k-read run. Quality kernels (per-cycle sums, $Q \geq 20$ and $Q \geq 30$ fractions, adapter seed-and-extend) are runtime-dispatched across scalar, NEON, AVX2 and AVX-512BW paths, and each is asserted bit-identical to a scalar double-precision reference at every read length from 0 to 300. That range is chosen deliberately to cover the sub-vector tails, which is where such kernels typically diverge from their scalar reference. A maximum record size of 64 KiB (roughly 32 kbp) is a stated capability limit rather than an oversight; raising it means raising per-lane buffer memory, which is a capacity decision and not a bug fix.

3.5 `scrna_matrix`: cache-aligned sparse matrices and approximate k -NN

`scrna_matrix` is a header-only C++20 library built around `Block_CSR<T>`, a 64-byte (cache-line) aligned compressed sparse-row/column representation with zero-copy pybind11 interoperability. A NumPy buffer that is already 64-byte aligned, C-contiguous and (critically, after a defect closed in review) provably immutable can be *adopted* without a copy. The immutability requirement is not fastidiousness: the module’s AVX2 and AVX-512 gather kernels index a dense scratch buffer with the adopted column indices using *unchecked* hardware gathers. This means that the bounds validation performed once at construction is the only thing standing between the inner loop and an arbitrary memory write, which is exactly why a buffer the caller can still mutate afterward may not be adopted at all.

The module offers two k -nearest-neighbor backends with genuinely different operating regimes, rather than one algorithm tuned to look good on a headline benchmark: an exact $O(n^2)$ brute-force path that operates natively on the sparse representation, and an approximate path over vendored `hnswlib` (Malkov and Yashunin, 2018) that must densify its input first, since HNSW indexes only dense fixed-dimension vectors. That densification is why the approximate backend is reported as *losing* to exact brute force by $0.05\times$ in raw 1,000-dimensional gene space: each HNSW distance touches 1,000 dense dimensions where the exact path touches roughly 60 non-zeros, so a smaller number of comparisons does not pay for their individual cost. On a 50-dimensional PCA embedding with real cluster structure the ordering reverses, and HNSW wins by $11.2\times$ at 50,000 cells at a recall of 0.997. In other words, the representation and not the algorithm decides which backend wins, and the module documents that as a rule of thumb rather than leaving each user to rediscover it by benchmarking both.

Table 2: Confirmed defects found and fixed across seven adversarial passes, 2026-08-15 to 2026-09-11, by module. “High/Crit.” is the subset rated High or Critical severity in the source document.

Module	Confirmed defects	High/Crit.
<code>fastq_stream</code>	27	9
<code>genomic_toolkit</code>	12	4
<code>cuttag_profiler</code>	9	5
<code>scrna_matrix</code>	18	4
<code>peaks (pto-peaks)</code>	8	2
Total	74	24

Every confirmed defect carries a regression test added in the same commit that fixed it. Verification after each pass included, per module: a sanitizer-clean ASan+UBSan run under `-fno-sanitize-recover=all`, the pre-existing suite re-run to confirm no regression, and where the defect was concurrency-related, a clean ThreadSanitizer run. `fastq_stream`’s 2026-09-11 pass additionally drove a 900-mutant corpus (bit flips, byte replacement, deletion, insertion, duplication and truncation across raw, gzip and BGZF framing) through the sanitizer build at 1, 3 and 8 threads with no crash, hang or sanitizer report.

Reductions accumulate in a widened type (`WideAcc<T>`, which is `double` for `float` input) and narrow exactly once through a checked cast, a rule enforced at build time by a source-level scan rather than left to code review. The reason is worth stating precisely, because the intuition about floating-point error is wrong here. A bare `float` accumulator does not gradually lose precision on large input; it saturates to $\pm\infty$, and the downstream guard that maps a non-finite similarity to zero then reports two *identical* rows as each other’s worst possible match. The input was finite, no exception was raised, no diagnostic was printed, and the neighbor graph is wrong. This exact defect class was independently found and fixed at four separate sites across three review passes (§4) before the source-scan test existed to prevent a fifth.

Concurrent index mutation is serialized behind a mutex that covers the label-count check and the insertion together, closing a data race that had silently discarded half the rows of two concurrently-added sets while raising no exception and leaving a perfectly valid index behind. Queries deliberately do *not* take that lock: `hnswlib`’s search is safe against a concurrent insertion, and serializing queries would forfeit exactly the parallel query throughput that releasing the Python GIL during a call exists to provide.

4 Formal Invariants and Adversarial Verification

4.1 Methodology

Verification here means driving hostile input at the shipped binary or the public header under a sanitizer build. It does not mean writing unit tests for behavior already believed correct. The protocol is common to all five modules and was repeated across seven distinct passes between 2026-08-15 and 2026-09-

11 (Table 2). First, establish a sanitizer-clean baseline on the pre-existing suite under AddressSanitizer and UndefinedBehaviorSanitizer with `-fno-sanitize-recover=all`, so that any later finding is attributable to the probe rather than to a pre-existing gap in the harness. Second, reproduce every finding against the pre-fix binary or header *before* patching it, and record the reproduction so it can be re-run. Third, fix the defect and add a regression test in the same change. Fourth, re-verify the full sanitizer suite, adding ThreadSanitizer wherever the defect was concurrency-related. Nothing is recorded as a finding on suspicion alone.

Negative results are recorded with the same care. Where a suspected defect could not be reproduced through the public API, the record says so explicitly. One example is a latent finite-arithmetic overflow in a cosine-similarity helper that Block_CSR’s own input validation makes unreachable in practice; it was fixed anyway, as defence in depth for a direct caller of the public header, and documented as unreachable rather than written up as an exploit that was never demonstrated.

Two purpose-built instruments recur across modules. The text-format readers in `genomic_toolkit`, `fastq_stream` and elsewhere are driven by seeded mutation fuzzers (bit flips, byte substitution, deletion, insertion, duplication and truncation) rather than by a coverage-guided engine. That choice is practical rather than principled: `libFuzzer`’s runtime is not shipped with the Apple Clang toolchain used for local development, and a harness that runs only where a coverage-guided engine happens to be installed does not run in this project’s own CI. A fixed seed also makes every failure reproduce exactly. The Poisson kernel in `peaks` is instead checked against arbitrary-precision (80-digit `mpmath`) golden values rather than against itself, at grid points chosen to stress the kernel’s own stated accuracy model, which is a difference of three terms each reaching $O(k \ln k)$ whose relative error is machine epsilon times the largest term. The tolerances are derived from that model; they are not observed values written down after a passing run.

4.2 Representative findings

The full record, with a reproduction, patch and regression test for each finding, lives in the repository’s AUDIT, TORTURE and REVIEW documents. Five are worth stating here in the detail that explains *why* they were findings, because each one instantiates a general failure mode rather than a one-off typo.

Silent format misdetection. `fastq_stream` had no code path that knew any quality encoding but Phred+33. The identical 2,000 reads, encoded as Phred+64 and read as Phred+33, produced a mean quality of 50.85 against a correct value of 19.85, and a Q_{30} rate of 1.000 against a correct 0.265. Every read passed quality trimming untouched and the process exited 0. The fix is a histogram-based heuristic evaluated at no per-base cost, keyed on three properties that hold together only for Phred+64 misread as Phred+33: nothing below Q_{31} , nothing above Q_{72} , and at least 1% of bases at Q_{61} or higher. An explicit `--phred-offset` override is available, and undeclared Phred+64 input now exits with the repository-wide code meaning “the input’s shape made the answer wrong” instead of

continuing silently.

Unchecked multiplication sizing a write. Four of the five HNSW graph producers in `scrna_matrix` sized their result buffer as $n \times k$ with no overflow check, while the fifth (brute-force k -NN) had been patched for exactly this in an earlier review. With $n = 4$ rows and $k = 2^{62}$, that product wraps to exactly 0: both allocations succeed against empty vectors, the search proceeds normally, and the result write lands on a null pointer, confirmed as a store-to-null under UBSan. The fix is a single checked helper called by every producer including brute force, so that the definition exists exactly once. Notably, the same audit reports two *other* findings of the same shape, in which an already-fixed defect class survived in a sibling path the earlier fix did not walk.

Truncation that decodes cleanly. A BGZF stream (a BAM, or a `.gz`) cut exactly at a block boundary, which is what an interrupted `aws s3 cp` or `cp` routinely produces, decodes without error. The CRC of every remaining block is intact and only the trailing 28-byte end-of-file marker is missing, so there is nothing for a record-level parser to trip on. Both `genomic_toolkit` and `fastq_stream` reported a plausible partial statistic for such a file at exit 0 before this was closed. The fix checks for the EOF marker immediately after the header and before any record is read, which is the rule `samtools quickcheck` and `pto-peaks`’ own BAM reader already enforced; it was applied to the two modules that lacked it.

A validation gap at the second entry point. `cuttag_profiler`’s window and bin-count guards were added to the CLI in `main.cpp` when the defect was first found. But `ProfileOptions` has a second constructor in effect: the `POST /api/profile` handler, which built the same struct from untrusted JSON while applying only a subset of the checks. An unauthenticated request with a sufficiently large upstream/downstream pair sized a 640 MB-per-region allocation and had the kernel SIGKILL the server with no diagnostic at all, which is the exact scenario the CLI-side fix believed it had already closed. The durable repair moved every guard onto `ProfileOptions::validate()` itself, called by both entry points and again by the allocating function as defence in depth. This means a future third caller cannot reintroduce the gap by construction, rather than merely being expected to remember the check.

A cliff, not a slope, in a float accumulator. `scrna_matrix`’s sparse dot-product kernel accumulates in float. Below roughly 10^{17} in row magnitude, the float path is exact enough to change nothing at all: a differential test against an exact double reference found zero neighbor-set changes across every ranked position. Above a measured threshold the dot product saturates to $+\infty$, the guard mapping non-finite similarities to zero fires, and two *identical* rows are reported as maximally dissimilar. In one measured sweep, 300 of 300 rows had a different and wrong neighbor set once magnitudes crossed the threshold. The behavior is not a gradual precision slope; it is a cliff, and everything past the edge is wrong at once. The chosen fix rejects the input at the boundary, with `Block_CSR::validate()` bounding squared

row norm by Cauchy–Schwarz four orders of magnitude below `FLT_MAX`, rather than widening the hot $O(n^2)$ kernel. That is an explicit performance trade, taken deliberately rather than as a drive-by change to the module’s headline benchmark.

4.3 What did not break

Reporting only findings would misstate the balance of evidence, so what survived hostile testing is recorded with the same care as what did not. The SPSC ring-buffer memory ordering (release-store of the tail index before publication, acquire-load before consumption) held clean under ThreadSanitizer across every adversarial input tried. `cuttag_profiler` produced bit-identical output across thread counts from 1 to 96. Zero-norm input vectors, which is what an empty cell looks like, were measured rather than merely argued to produce no NaN anywhere in `scrna_matrix`: the all-zero row scores a defined similarity of 0 against everything, which is the correct degradation for a cell with no counts, and it can never displace a real neighbor in anyone else’s result.

One reported ThreadSanitizer race deserves its own note, because the temptation to dispose of it in either direction was real. The race appeared in `genomic_toolkit`’s OpenMP FRiP reduction, and it was investigated to ground rather than reflexively suppressed or accepted. It reproduced only against an uninstrumented `libomp` runtime; it vanished when OpenMP was disabled entirely and when the same workload ran single-threaded; and 55 runs across 11 thread counts produced exactly one distinct result, which a genuinely broken reduction would not do. No suppression was added. The investigation and its four independent pieces of evidence were recorded instead, so that a future genuine race in that file is not waved through by an overbroad filename-based rule.

4.4 Recurrence as a methodological finding

The most useful observation to come out of this record is not any individual defect but a pattern across them, and it was the second review pass that made it explicit: of five findings in that pass, four were the *same defect class as an already-closed finding*, surviving in a sibling function, a sibling code path, or a second entry point the earlier pass did not walk. I hypothesized from that pass that the recurrence was structural rather than coincidental, and the subsequent audits bore it out. A fix for an unsigned-integer underflow in one quality statistic left a character-for-character identical expression unfixed 50 lines below, in a function that gates a filtering decision rather than a printed report. A float-square overflow fix applied to one norm computation left three hand-rolled sibling loops untouched. The HTTP validation gap described above is the same pattern applied to an entry point rather than to a function.

The cause is visible once named: the audit discipline that produces good findings, namely reproduce before fixing, also focuses the fix on the single line where the reproduction landed. In other words, the very rigor that makes each finding trustworthy is what narrows the repair. Three process

changes follow, and they are recorded alongside the findings themselves. After any fix, `grep` the tree for the defective *construct* rather than the defective line. Validate on the type rather than at any one entry point, so that a new caller cannot bypass a check by construction. Enumerate every untrusted-input entry point explicitly instead of re-deriving the list from memory during each review. I regard those three rules as at least as transferable a contribution as any individual throughput number in Section 5.

5 Results

To evaluate these claims, we begin by defining how every figure below was obtained. All wall-clock and memory measurements come from a single host (darwin arm64, 12 logical CPUs) through a harness that runs the reference tool and `pto-core` in separate processes, so that `RUSAGE_CHILDREN`’s monotonic high-water mark cannot leak one tool’s peak into the other’s measurement. Peak RSS is read from `wait4/ru_maxrss` rather than from a sampler that could miss a spike between polls. For a multi-stage reference pipeline, wall-clock time is summed across stages while peak RSS is taken as the *maximum* across stages rather than the sum. That asymmetry is deliberate, and it runs against this project’s interest: summing RSS would inflate the baseline and flatter `pto-core`.

5.1 Cross-tool Pareto benchmarks

Table 3 reports every case in which `pto-core` and the established tool it replaces were confirmed to agree on the answer, with the agreement evidence itself in Table 4 below. A speedup is shown only where that agreement holds, which is the same discipline applied to the peak-concordance study in Section 5.4. The two `gtk/sizes` rows are the cases where `pto-core` does not win on memory (RAM ratio $\approx 1\times$), and the reason is instructive rather than disappointing: both `genomic_toolkit sizes` and `samtools stream` the input and neither buffers it, so there is no intermediate materialization for a streaming architecture to remove. The `fastq/qc [small]` case is listed as skipped rather than quietly omitted, following the harness’s own convention that a report showing only the cases that ran is indistinguishable from a report of a run in which everything ran.

5.2 Correctness under speed

A speedup over a wrong answer is not a result, so Table 4 reports the same run’s answer-agreement alongside the timings above. Three of the four cases require exact, bit-identical counts: `fastq/qc`’s read and base totals, `gtk/frip`’s fragment and in-peak counts, and `gtk/sizes`’ median length together with the Kolmogorov–Smirnov statistic against the reference size distribution. `cuttag/profile`’s meta-profile is compared by Pearson correlation under a stated tolerance instead, because `deeptools` and `cuttag_profiler` normal-

Table 3: Wall-clock and peak-RSS comparison against the established tool each workload replaces, over the full case $\times$ tier matrix that produced an agreed answer (7 of 7 cases; concordance is Table 4). MB is 10^6 bytes throughout.

Workload	Tier	pto-core (s)	Reference (s)	Reference tool	pto RSS (MB)	Ref. RSS (MB)	Speedup	RAM ratio
cuttag/profile	medium	2.30	87.72	deeptools	89.3	447.1	38.18 $\times$	5.01 $\times$
cuttag/profile	small	0.08	4.39	deeptools	10.0	83.6	56.27 $\times$	8.36 $\times$
fastq/qc	medium	0.0947	0.7088	fastp	4.7	1,163	7.48 $\times$	247.43 $\times$
gtk/frip	medium	0.78	30.24	samtools+bedtools	23.5	1,681.2	38.80 $\times$	71.54 $\times$
gtk/frip	small	0.08	2.80	samtools+bedtools	8.7	198.4	35.11 $\times$	22.80 $\times$
gtk/sizes	medium	0.64	2.64	samtools	6.8	6.7	4.14 $\times$	0.99 $\times$
gtk/sizes	small	0.07	0.30	samtools	6.8	6.7	4.29 $\times$	0.99 $\times$

fastq/qc [small] is omitted: the tier stages no reads_r1 input and the case reports skip, not a measurement – listed as an omission in the underlying report rather than left out of this table silently. Across the 7 cases shown, peak RSS ranges from 0.4% to 101.2% of the reference tool’s (the gtk/sizes rows, where both tools stream and neither buffers, are the two cases pto-core does not win on memory), and the reference pipelines wrote 2,090.9 MB of intermediate files across these cases that pto-core never materialises at all.

Table 4: Correctness under the speedups of Table 3. All deltas are on the medium tier unless noted.

Case	pto-core	Reference	Agrees
cuttag/profile: Pearson r	0.999977	1.0	tol. 0.01
fastq/qc: reads	1,168,663	1,168,663	exact
fastq/qc: bases	29,216,575	29,216,575	exact
gtk/frip: fragments	3,999,989	3,999,989	exact
gtk/frip: in peaks	1,629,113	1,629,113	exact
gtk/frip: FRiP	0.407279	0.4072794	tol. 10^{-6}
gtk/sizes: median len. (bp)	174	174	exact
gtk/sizes: KS statistic	0.0	0.0	exact

ize against different denominators by construction (a bigWig-derived coverage track versus fragment counts read directly from the BAM). This means the property under test there is the *shape* of the profile, not numerical identity, and the tolerance says so rather than hiding a mismatch.

5.3 Thread scaling and the cores-busy caveat

On the cuttag/profile [medium] workload (80 bins), wall-clock time falls from 16.86 s at one thread to 2.91 s at four (5.79 $\times$) and 2.30 s at eight (7.34 $\times$, 92% parallel efficiency). That figure needs a caveat the harness itself surfaces: the nominal one-thread run already averaged 4.32 cores busy, because htlib decompresses BGZF on its own internal threads regardless of the `--threads` flag passed to the profiler. In other words, the baseline was never serial, so the efficiency denominator is wrong in this project’s favor, and any figure that reads as superlinear scaling is a property of that baseline rather than of the algorithm. I report it this way instead of quoting the more flattering ratio a naive one-thread-to-many-thread comparison would produce.

5.4 Peak-calling concordance: a split decision

pto-peaks was benchmarked against MACS3 (Zhang *et al.*, 2008) on real ENCODE (ENCODE Project Consortium, 2012) K562 ATAC-seq data (ENCSTR217QAB, ENCFF121ZQX.bam, 2,889,130 fragments), which is the reference result this project’s own implementation plan called for. The outcome is a genuine split decision and is reported as one.

Table 5: pto-peaks vs. MACS3 on real ENCODE K562 ATAC-seq. The pre-registered parity bar (0.98 on both gated metrics) is stated alongside the result it was measured against, not omitted.

Metric	Value	Parity gate	Met?
Total MACS3 peaks	31,607	—	—
Total pto-peaks peaks	31,835	—	—
IoU-matched pairs	18,434	—	—
Summit within 50 bp	0.9616	≥ 0.98	No
Spearman r , $-\log_{10} p$	0.8568	≥ 0.98	No
Mean IoU	0.458	not gated	—

Caller	Median width (bp)	p10	p90
pto-peaks (defaults)		96	57
MACS3		275	169
ENCODE (replicated, pooled)		336	—

On the 7,442 pairs at $\geq 50\%$ reciprocal overlap	
Median summit distance	1 bp
Summit within 50 bp / 100 bp	0.976 / 0.997
Pearson r on peak strength	0.990
Spearman r on peak strength	0.908

On summit location and relative peak strength, the two callers are effectively interchangeable. Over the 7,442 peak pairs at $\geq 50\%$ reciprocal overlap, median summit distance is 1 bp, 0.976 of summits fall within 50 bp, and peak-strength Pearson correlation is 0.990. On interval width they are not. pto-peaks calls systematically narrower intervals, with a median of 96 bp against MACS3’s 275 bp, roughly 0.35 $\times$ the width, and this single structural difference is the direct cause of every failing summary metric. Mean IoU of 0.458 sits close to the theoretical ceiling for a 96 bp call nested inside a 275 bp call, and the harness’s IoU-greedy 1:1 matcher pairs only 23.5% of peaks at $\geq 50\%$ reciprocal overlap. This means that much of the apparent 42% “unmatched” rate measures the matcher’s inability to bridge a width gap, not the two callers finding different genomic regions. Peak splitting is minor and one-directional: 523 MACS3 peaks (1.7%) are covered by two or more pto-peaks calls, while zero pto-peaks calls are covered by two or more MACS3 peaks. That asymmetry is consistent with pto-peaks occasionally fragmenting one

wide MACS3 peak and never merging two.

The pre-registered parity gate for this project was 0.98 on both summit overlap and Spearman correlation of peak strength. Both are **not met** (Table 5). Following the 2026-09-10 result, the harness’s exit code was re-baselined onto two *regression floors*, 0.95 and 0.85, chosen just under the measured 0.9616 and 0.8568. I want to be precise about what that re-baselining does and does not mean. It changes the question the exit code answers, from “does this meet parity” to “has this drifted from its first real-data result,” and nothing more. The original 0.98 gates are still computed, printed and written to the metrics artifact unconditionally, and they are still reported as not met; an exit code of 0 from the regression harness is documented, both in the repository and here, as not a parity claim.

A compatibility mode exists and does not resolve the question either. `--extend-peaks` pads every reported interval symmetrically without altering summit position or significance, and at `--extend-peaks 90` the median width moves from 96 bp to approximately 276 bp, landing on MACS3’s own median. But the harness measures the default, unextended configuration, and whether the *default* boundary convention should change is recorded as an open decision. Adding a flag is not the same as answering the question.

5.5 `scrna_matrix` at scale: feasibility without a baseline

On a synthetic 1.3 million-cell dataset (1,000 genes, 6% density), naive extrapolation of the exact $O(n^2)$ brute-force path’s measured scaling law, approximately 3.5 ns per n^2 operation and holding within measurement noise across a $16\times$ range of n , projects 1.6 hours of wall-clock time. The HNSW backend on the same dataset is a real measured run rather than an extrapolation: 37.1 s wall-clock at 1.01 GB peak RSS, including index construction and all 1.3M queries, for a $155\times$ reduction against the exact path’s extrapolated cost.

I report that as closing a *feasibility* gap, in which an intractable exact computation becomes a tractable approximate one, and explicitly not as a validated performance claim against any competing tool. The comparative targets that motivated this module, a $20\times$ wall-clock speedup and a 60% peak-RSS reduction against `scanpy.pp.neighbors()` (Wolf *et al.*, 2018), have not been measured at this scale, and no number for either appears anywhere in this manuscript (Section 7). Recall of the HNSW approximation is likewise not reported above 50,000 cells, for a reason worth stating rather than eliding: the exact baseline needed to measure recall is precisely the computation that is infeasible at that size.

5.6 Live deployment validation

Beyond the single-host measurements above, the full pipeline was exercised end to end against a deployed AWS stack rather than only against mocked AWS services. That path comprises API submission, an SQS FIFO queue, a Lambda dispatcher, one ephemeral Fargate task per job, and a callback consumer

that restores job state. One real job (`genomic_toolkit frip` on an 8.5 MB input) completed queue-to-terminal in 27.2 s. This is evidence that the architecture in Section 3 runs as a deployed system; it is not a performance claim. A single job on a small input is not a load test, and no throughput or concurrency figure is claimed from it.

6 Discussion

The throughput results in Section 5 are the least interesting part of this work, and I would rather say so directly than let them carry the argument. Most of the speedups have a structural explanation rather than a micro-optimization one: a streaming reduction does less work than a pipeline that materializes an intermediate at genome scale, and binary search over a merged cover does less work than a general interval intersection. Anyone rebuilding these tools with the same architectural commitments should expect to land in the same neighborhood. The numbers are a consequence of the design; they are not the contribution.

What I think does generalize is the verification record and the shape of what it caught. Of the 74 confirmed defects, the ones that mattered most were almost never crashes. They were a truncated file reporting half a library at exit 0, a quality encoding misread into a plausible mean, a skipped region diluting a meta-profile by a constant the user could not see, and a saturating accumulator turning two identical cells into each other’s worst match. Each of those produces output that passes every downstream sanity check a biologist would think to apply, because it is well-formed, finite and in range. This is what makes the fail-open failure mode categorically worse than a segfault: a crash costs an afternoon, while a plausible wrong number can cost a paper. Fail-closed parsing is therefore not defensive programming in the ordinary sense. It is the recognition that in this domain, refusing to answer is a better outcome than answering incorrectly.

The recurrence pattern in §4 sharpens the point. Four separate instances of an already-closed defect class surviving in a sibling path is not a story about carelessness; it is a story about how a reproduce-then-fix discipline naturally narrows a repair to the line where the reproduction landed. The corrective is to move the guard onto the type, so that the invariant holds for callers who have not been written yet. Rather than merely fixing the defect that was found, the goal is to make the class of defect unrepresentable at the boundary where untrusted data enters. That is an old idea in systems and formal-methods practice. It is not yet a default expectation in bioinformatics tooling, and I would like it to become one.

This also frames what `pto-core` is not. It is not a claim that existing tools are badly built by people who should have known better; most of them were written under constraints that made different trade-offs reasonable, and several are cited throughout this manuscript precisely because they are the correct reference for their task. The claim is narrower and, I think, harder to argue with: the economics have changed. A sequencing core now generates data faster than the analysis

layer’s memory ceiling scales, HPC allocations are spent re-reading files that never needed to be written, and the cost of a silent wrong answer rises with every downstream analysis built on it. Treating these engines as safety-critical systems software, with bounded memory, runtime-dispatched vector kernels, fail-closed invariants and a continuous adversarial audit, is not gold-plating under those conditions. It is the minimum standard the data volume now demands.

7 Limitations and Honest Residuals

I want to be transparent about the limits of this system, because each of the following bears directly on what a reader should and should not conclude from the numbers above.

- **Peak-calling interval-width parity is an open problem, not a solved one.** Section 5.4’s 0.98 gates are not met, and the regression floors that replaced them as the harness’s exit criterion are explicitly a drift trip-wire set after the fact rather than a validation bar. Three remedies are documented as options rather than decided: widen the default boundary convention toward MACS3’s; change what the harness measures, weighting summit and strength agreement over full-interval IoU; or accept and document the divergence as a deliberate tighter-boundary design. None has been chosen.
- **scrna_matrix’s comparative performance claims against scanpy.pp.neighbors() are unmeasured at the scale that matters.** The 1.3M-cell HNSW run (Section 3.5) demonstrates feasibility, not superiority against a baseline. No like-for-like run against scanpy at that scale exists, and `sc.pp.neighbors` on a PCA representation is itself approximate and near-linear, which makes it a peer of the HNSW backend rather than a baseline it should categorically be expected to beat.
- **deeptools parity for cuttag_profiler was not re-measured after the skipped-region fix (§3.3),** because deeptools was not installed on the audit machine at the time of that fix. The fix changes the reported mean only when a region is skipped, and the benchmarked case in Table 3 had none, so “unchanged” for that table is inferred from the logic of the fix rather than independently observed.
- **AVX-512 is compiled but has never executed on hardware that natively supports it,** across every module that offers it (`fastq_stream`, `peaks`, `scrna_matrix`). The AVX2 kernels were executed on x86-64 only under Rosetta 2 emulation, which runs AVX2 instructions but does not report their availability in `CPUID`; runtime auto-selection of the AVX2 path was therefore forced rather than observed. No throughput claim is made for either ISA beyond compiling and being numerically correct against a scalar reference.
- **std::bad_alloc resilience is untested.** macOS does not enforce `RLIMIT_AS`, so no allocation failure could

be induced on the development host for any module. A Linux runner with a hard address-space limit, or an allocator shim, is the stated path to closing this.

- **The Python-binding sanitizer coverage is split across two hosts, for a specific and checkable reason.** macOS System Integrity Protection strips `DYLD_*` variables from a protected process before it starts, and those variables are exactly what preload-based sanitizer instrumentation of a dynamically loaded Python extension requires; this was verified from inside the running interpreter rather than assumed. `scrna_matrix`’s full C++ engine was verified locally under CI’s exact sanitizer flags, but its Python binding layer’s sanitizer coverage comes from Linux CI and not from this host.
- **Two parser-level residuals remain open.** Plain gzip and raw FASTQ carry no end-of-stream marker analogous to BGZF’s, so a file truncated exactly between records or between gzip members is, to any parser, a valid shorter input rather than a detectable truncation. Only an external checksum or manifest would catch it. Separately, no structure-aware fuzzer yet exists over `scrna_matrix`’s `from_scipy_csr` boundary, which is the entry point at which one of the eleven confirmed findings in that module’s audit (an unchecked narrowing conversion) was found by manual probing rather than by fuzzing.
- **CRAM truncation is unexercised.** `genomic_toolkit`’s BGZF end-of-file check does not extend to CRAM, which carries its own end-of-file container, and no CRAM fixture with a local reference was available to test against.

8 Availability

Source, build instructions, the full verification and benchmark record cited throughout this manuscript, and the license are available at <https://github.com/chrisaperez/pto-core> under the MIT license. All five modules build independently with CMake ≥ 3.20 and a C++20 compiler. `scrna_matrix` additionally requires OpenMP, enforced as a hard configure-time failure rather than a silent fallback to single-threaded execution. That hard failure exists because of an earlier incident: a soft OpenMP dependency once left `_OPENMP` undefined, every `#pragma omp` compiled to serial code, and the test suite reported 100% passed while never executing a single parallel iteration. A silently single-threaded engine that CI signs off on is worse than a build that stops. Every number in this manuscript resolves through `numbers.tex`, which is checked in beside `main.tex`, so that regenerating the underlying reports regenerates the manuscript.

9 Conclusion

I have presented five independent, hardware-aware C++20 engines for high-throughput omics, unified by a deployment

constraint and a verification discipline rather than by shared code. Measured against the tools they replace, they are consistently faster and lower in peak memory across 7 workloads without sacrificing answer correctness. But the result I consider most transferable came from building them adversarially rather than from the throughput figures: the same fail-open, unbounded-memory and unvalidated-second-entry-point defect classes this project explicitly set out to refuse were nonetheless found 74 times before a sustained sanitizer-driven practice closed them. Writing correct performance-path C++ for this domain is not a matter of intent. It is a matter of continuous, reproducible, adversarial verification.

Two results here do not resolve as cleanly as a headline speedup: peak-calling boundary-width parity, and single-cell k -NN performance at scale relative to an established baseline. I have reported both as open questions rather than rounding them into successes. The same numerical-integrity standard that makes the benchmark numbers in Section 5 worth trusting is the standard that requires it.

References

- Bolger, A.M., Lohse, M. and Usadel, B. (2014) Trimmomatic: a flexible trimmer for Illumina sequence data. *Bioinformatics*, **30**, 2114–2120.
- Chen, S., Zhou, Y., Chen, Y. and Gu, J. (2018) fastp: an ultra-fast all-in-one FASTQ preprocessor. *Bioinformatics*, **34**, i884–i890.
- Danecek, P. *et al.* (2021) Twelve years of SAMtools and BCFtools. *GigaScience*, **10**, giab008.
- Biggers, E. (2016) libdeflate: heavily optimized library for DEFLATE/zlib/gzip compression and decompression. <https://github.com/ebiggers/libdeflate>.
- The ENCODE Project Consortium (2012) An integrated encyclopedia of DNA elements in the human genome. *Nature*, **489**, 57–74.
- Kaya-Okur, H.S. *et al.* (2019) CUT&Tag for efficient epigenomic profiling of small samples and single cells. *Nat. Commun.*, **10**, 1930.
- Malkov, Y.A. and Yashunin, D.A. (2018) Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE Trans. Pattern Anal. Mach. Intell.*, **42**, 824–836.
- Quinlan, A.R. and Hall, I.M. (2010) BEDTools: a flexible suite of utilities for comparing genomic features. *Bioinformatics*, **26**, 841–842.
- Ramírez, F. *et al.* (2016) deepTools2: a next generation web server for deep-sequencing data analysis. *Nucleic Acids Res.*, **44**, W160–W165.
- Serebryany, K. and Iskhodzhanov, T. (2009) ThreadSanitizer: data race detection in practice. In *Proc. WBIA*, pp. 62–71.
- Serebryany, K., Bruening, D., Potapenko, A. and Vyukov, V. (2012) AddressSanitizer: a fast address sanity checker. In *Proc. USENIX ATC*, pp. 309–318.
- Wolf, F.A., Angerer, P. and Theis, F.J. (2018) SCANPY: large-scale single-cell gene expression data analysis. *Genome Biol.*, **19**, 15.
- Zhang, Y. *et al.* (2008) Model-based analysis of ChIP-Seq (MACS). *Genome Biol.*, **9**, R137.