EXACT, BATCH-INVARIANT AND FAST TEST-TIME QUERY OPTIMIZATION FOR LATE-INTERACTION RETRIEVAL

Anonymous authors
Paper under double-blind review

ABSTRACT

Test-time query optimization (TTO) refines a query embedding by gradient steps against a frozen candidate pool and lets compact multi-vector retrievers rival much larger ones. With late interaction, every step rescans every vector of every candidate, and how these scans are executed changes the result: TF32 tensor cores change maxima and selected vectors, and library matrix products change their bits with the batch a request is served in. We present ExactTTO, which makes TTO exact, that is, bitwise identical to a fixed FP32 reference, batch-invariant and fast by letting approximation decide only which products are computed, never their values, and fixing the order of all other arithmetic. Certified working sets skip rescans that provably cannot change the result. Tensor cores only filter candidates, whose FP32 maxima and selected vectors are recomputed exactly. A fused loop applies each step's gradient and Adam update in one kernel with a fixed operation order, so a query's trajectory does not depend on the requests it is batched with. Under the official protocol of Guided Query Refinement (GQR) on fourteen ViDoRe splits and four BEIR datasets, we recover GQR's gains. In batches of 64 requests, ExactTTO processes $3.05\times$ as many queries per second as GQR's FP32 scorer in our fused loop and $3.95\times$ as many as FLASH-MaxSim in FP32, and gives every query the bits it gets when served alone, which GQR's scorer does for 80.4% of the queries; a single ColNomic refinement runs $21\times$ faster than GQR's released eager loop, mostly by removing framework overhead. ExactTTO also accelerates a second objective, distilling a cross-encoder into ColBERTv2 queries, which raises nDCG@10 on BEIR from 37.1 to 41.5.

1 INTRODUCTION

Adapting a model to each input at inference time is now a standard tool (Sun et al., 2020; Wang et al., 2021a; Snell et al., 2025). In retrieval, test-time optimization (TTO) takes a few gradient steps on the embedding of a query so that its scores over a frozen candidate pool agree with a stronger or complementary signal (Sung et al., 2023; Reddy et al., 2023). Guided Query Refinement (GQR) applies this to multimodal retrieval: a text retriever guides the multi-vector query of a vision retriever, which then matches models with much larger representations at a fraction of their cost (Uzan et al., 2026). TTO touches only the query; documents and the index stay fixed.

With late-interaction retrievers (Khattab & Zaharia, 2020; Santhanam et al., 2022b; Faysse et al., 2025), every TTO step rescans every vector of every candidate (about a thousand per page image) for every query token, and **how these scans are executed changes the result** (Fig. 1a). Tensor-core kernels make a scan cheaper (Pony et al., 2026), but in TF32 they change maxima and sometimes the selected vector, which the gradient of MaxSim follows; at GQR's own operating points the final top-5 changes for 0.9% of the audited test queries. Even FP32 is not one computation: library matrix products choose their kernel, and with it their summation order, by problem shape, and the shape includes the batch a request is served in. In batches of 64, GQR's own FP32 scorer changes the final scores of 19.6% of our test queries, though not their top-5. A TTO result is thus defined only together with its implementation and traffic, the batch-invariance problem raised for LLM inference (He & Thinking Machines Lab, 2025). **Can bitwise-exact, batch-invariant TTO be fast on tensor cores?**

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

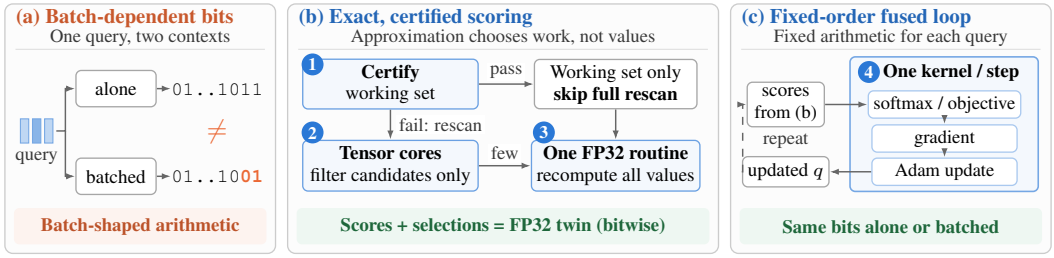

Figure 1: **Exact, batch-invariant TTO** (schematic). (a) Served alone or in a batch, the same request can end with different bits: library matrix products choose their kernel and summation order by problem shape, and TF32 changes maxima and selected vectors. (b) ExactTTO lets approximation decide only *which* products are computed: certificates skip rescans that cannot change the result, and tensor cores pick candidates that one fixed-order FP32 routine recomputes. (c) After scoring, one kernel per step applies each query's gradient and Adam update in a fixed order.

We answer yes with ExactTTO, built on one principle: *approximation may decide which products are computed, never their values*, and every operation that follows runs in a fixed order. A TTO run is then *exact*, i.e., bitwise identical at every step to a full FP32 computation with a fixed summation order, and *batch-invariant*; three components make it fast.

- **Certified working sets** (Section 3). For each query-token–document pair we keep the best few vectors of its last full scan and certify, from the query's displacement, that no other vector can overtake them; the test is FP32-rigorous (Theorem 2), and the number of rescans is bounded by the path length of the optimization (Theorem 3).
- **Exact maxima from tensor cores** (Section 4). A two-term BF16 split of the query, a rigorous accumulation bound and an FP32 recomputation of the surviving candidates return the FP32 maximum and its index bitwise (Theorem 4).
- **A fused, batch-invariant loop over document-major kernels** (Section 5). Skipped interactions become skipped memory traffic, and one kernel per step computes every query's gradient and Adam update in a fixed order, so a query's trajectory does not depend on its batch (Proposition 2); the loop alone makes GQR's own scorer $2.73\times$ faster than with autograd.

Results. Under GQR's released protocol on fourteen ViDoRe splits and four BEIR datasets (Section 6), we recover GQR's gains, and a ColNomic refinement takes 1.38 ms, $21\times$ faster than GQR's released eager loop; a single request leaves the GPU mostly idle, and there GQR's own scorer in our loop is faster. In batches of 64, ExactTTO processes $3.05\times$ as many queries per second as GQR's FP32 scorer in the same loop and $3.95\times$ as many as FLASH-MaxSim in FP32, and gives every query the bits it gets alone; FLASH-MaxSim's TF32 path, which is not exact, is $1.50\times$ faster still. Distilling a cross-encoder into ColBERTv2 queries as in ReFIT (Reddy et al., 2023) raises nDCG@10 on BEIR from 37.1 to 41.5 (cross-encoder reranking: 41.1).

2 Test-time query optimization over late interaction

Late interaction. A query is a matrix $q = (q_i)_{i=1}^{N}$ of token embeddings $q_i \in \mathbb{R}^h$, and a document d is a set of vectors $\{v_{dj}\}_{j=1}^{S}$ (patches of a page image, or tokens of a passage). MaxSim scores the document as

$$s_d(q) = \frac{1}{N}\sum_{i=1}^{N} \max_{1\le j\le S} \langle q_i, v_{dj}\rangle, \qquad \hat{\jmath}_{di}(q) = \min \arg\max_{j} \langle q_i, v_{dj}\rangle, \tag{1}$$

where $\hat{\jmath}_{di}$ indexes the *selected* vector of the token–document pair (d, i) (ties go to the smallest index). Almost everywhere, $\nabla_{q_i} s_d(q) = v_{d\hat{\jmath}_{di}}/N$, the gradient that automatic differentiation propagates.

Guided query refinement. GQR (Uzan et al., 2026) refines the query of a primary late-interaction retriever with the help of a complementary retriever, in our experiments a single-vector text embed-

der with cosine scores s^{fb}. We follow its released pool and tuning protocol. The candidate pool $\mathcal{C}$ is the union of the two retrievers' top-10 documents over the whole corpus ($10 \le C \le 20$). The target distribution over the pool is fixed at the start,

$$p = (1-\alpha)\,\mathrm{softmax}\big(s(q^{(0)})/\tau\big) + \alpha\,\mathrm{softmax}\big(s^{\mathrm{fb}}/\tau\big), \qquad \alpha = \tfrac{1}{2},\ \tau = 1, \tag{2}$$

and Adam (Kingma & Ba, 2015) minimizes $\mathrm{KL}\big(p \,\|\, \mathrm{softmax}(s(q)/\tau)\big)$ over the query tokens for T steps with step size η; the pool is then re-ranked by the refined primary scores $s(q^{(T)})$. For every dataset, (η, T) is selected on a random 10% development split of the queries from $\eta \in \{10^{-5}, 5{\times}10^{-5}, 10^{-4}, 5{\times}10^{-4}, 10^{-3}, 5{\times}10^{-3}\}$ and $T \in \{5, 10, \ldots, 100\}$, and nDCG@5 is reported on the remaining 90%. We call the selected pair the dataset's *operating point*. Our guarantees need only that the optimizer is a deterministic function of the gradients it has seen, which covers Adam and SGD with momentum.

Exactness. A scoring operator is *exact* if, at every call, it returns the same scores $s_d(q)$ and selected indices $\hat{\jmath}_{di}(q)$ as a full scan; it then leaves the gradients, the optimizer state and the whole trajectory $q^{(0)}, \ldots, q^{(T)}$ unchanged (Theorem 1). In floating point, the reference is a *dense twin* that computes every product in IEEE binary32 (FP32) with one fixed routine, and our operators must reproduce its scores and selections bit for bit. Unlike library matrix products, whose summation order depends on the GPU, the library version and the problem shape, including the batch, a fixed-order routine returns the same bits for the same inputs on any device with binary32 fused multiply–adds and round-to-nearest; tensor-core kernels in TF32 or BF16 (Pony et al., 2026) do not (Section 6.4). A TTO run makes $T{+}1$ calls of $CNSh$ multiply–adds each for a query that barely moves; Section 3 skips most of these scans exactly, and Section 4 runs the rest on tensor cores.

3 Certified working sets

3.1 Skipping scans without changing the trajectory

We cache the best vectors of each token–document pair and rescan the pair only when a vector outside the cache could overtake them. For a pair (d, i), at its last full scan, performed at the reference query q_i^{ref}, we store the *working set* H_{di}: the indices of the $r{+}1$ largest products $\langle q_i^{\mathrm{ref}}, v_{dj}\rangle$ (ties by index), its first element $j^\star$, and the *gap*

$$m_{di} = \langle q_i^{\mathrm{ref}}, v_{dj^\star}\rangle - \max_{j \notin H_{di}} \langle q_i^{\mathrm{ref}}, v_{dj}\rangle, \tag{3}$$

with $m_{di} = \infty$ if H_{di} covers the document. Let $R_{dj} = \max_{j'} \|v_{dj'} - v_{dj}\|_2$ be the radius of the document around vector j, computed once per corpus, and let $\Delta q_i = q_i - q_i^{\mathrm{ref}}$ be the displacement since the last scan.

Proposition 1 (Working-set certificate)**.** *If $m_{di} > R_{dj^\star}\|\Delta q_i\|_2$, every maximizer of $j \mapsto \langle q_i, v_{dj}\rangle$ lies in H_{di}. Consequently $\max_{j \in H_{di}} \langle q_i, v_{dj}\rangle = \max_j \langle q_i, v_{dj}\rangle$, and the smallest maximizing index within H_{di} is $\hat{\jmath}_{di}(q)$.*

The test depends on the query only through $\|\Delta q_i\|_2$ and costs h multiply–adds. Unlike the drift bounds of exact k-means (Elkan, 2003; Hamerly, 2010), it does not track the maximizer: while the test passes, the maximizer may change within H_{di} without any refresh. The *certified operator* evaluates a pair on H_{di} when the test passes; otherwise the certificate *fails* and the pair is *refreshed*: the operator scans the document and replaces $(q_i^{\mathrm{ref}}, H_{di}, j^\star, m_{di})$. Each pair keeps its own reference, so many small steps cannot silently exhaust an old gap.

Theorem 1 (Exact trajectories)**.** *Run TTO with any optimizer whose update is a deterministic function of the gradients seen so far. In real arithmetic, replacing the full scan by the certified operator leaves every score, selected vector, gradient, optimizer state and iterate $q^{(t)}$ unchanged, for all t.*

The proof (Appendix A) is an induction on t: Proposition 1 makes each certified call return the full-scan values and selections, which determine the gradient $\nabla_{q_i}\mathcal{L} = \sum_d \partial\mathcal{L}/\partial s_d \cdot v_{d\hat{\jmath}_{di}}/N$. The operator's internal state (which pairs were refreshed when) depends on the history, but its outputs never do. The radius $m_{di}/R_{dj^\star}$ is a cheap lower bound on the best radius that a test observing only $\|\Delta q_i\|_2$ can certify for this anchor (Proposition 3), and along GQR trajectories it certifies nearly as many interactions as that test (Appendix G).

3.2 EXACTNESS IN FLOATING POINT

Proposition 1 is a statement about real numbers; in FP32 the stored gap, the current products and the test are all rounded. Let $u = 2^{-24}$, $\gamma_n = nu/(1-nu)$, and let Q and V bound the norms of all query tokens along the trajectory and of all vectors of the document.

Theorem 2 (FP32 certificate)**.** *Assume IEEE binary32 arithmetic with round-to-nearest, no underflow or overflow, and that every product $\langle q_i, v_{dj}\rangle$ is evaluated as an h-term dot product in some order (possibly different at the reference and at the current call). Let $\hat{m}$ be the gap computed in FP32 from computed products, $\hat{t} = \mathrm{fl}\big(\sqrt{\mathrm{fl}(\sum_k \mathrm{fl}(q_{ik} - q_{ik}^{\mathrm{ref}})^2)}\big)$ with correctly rounded square root, and $\tilde{R} \geq \lambda_h R_{dj^\star}$ with $\lambda_h = (1+u)/\big((1-u)^4\sqrt{1-\gamma_h}\big)$. If the FP32 test $\hat{m} > \mathrm{fl}(\mathrm{fl}(\tilde{R}\,\hat{t}) + \varepsilon)$ passes with $\varepsilon \geq 4\gamma_h QV(1+u)/(1-u)$, then (i) the conclusion of Proposition 1 holds exactly, and (ii) every computed tail product is strictly smaller than the computed product of $j^\star$. Hence a working-set evaluation that computes each product with the same arithmetic as the full scan returns the full scan's FP32 maximum and selected index bitwise.*

For $h = 128$, $\lambda_h - 1 = 4.11\times10^{-6}$ and $\varepsilon = 3.05\times10^{-5}\,QV$: the certificate enlarges the stored radii by a few parts per million and adds a guard of about 256 units in the last place of a unit-norm product. Q follows from the Adam bound of Lemma 1, which Lemma 2 extends to binary32 (Appendix B). Part (ii) makes bitwise identity with the dense twin a matter of construction (Section 5).

Refreshes are paid per unit of path length. A refresh happens only after the query has moved at least the certified radius, so refreshes follow the distance traveled, not the number of steps.

Theorem 3 (Refreshes and path length)**.** *Let a pair be refreshed at calls $t_1 < t_2 < \cdots < t_K$ and let $\rho^{(k)} = m^{(k)}/R^{(k)}$ be the certified radius set at refresh k. Then $\sum_{t=t_k}^{t_{k+1}-1} \|q_i^{(t+1)} - q_i^{(t)}\|_2 \geq \rho^{(k)}$, and therefore $K \leq 1 + L_i/\rho_{\min}$ whenever $\rho_{\min} > 0$, where L_i is the path length of token i and $\rho_{\min}$ the smallest certified radius set along the run.*

With Adam, every coordinate moves by at most $\rho_t\eta$ at step t ($\rho_1 = 1$, $\rho_{25} = 1.228$; Lemma 1), so $L_i \leq \eta\sqrt{h}\sum_{t\leq T}\rho_t$: pairs whose certified radius (in FP32, the guarded one) stays away from zero obey the same bound however finely the path is cut, while pairs whose radius stays below one step are refreshed at every call, as zero-initialized tokens often are (Section 6.4).

4 EXACT SCORING ON TENSOR CORES

Certificates remove most scans at small step sizes, but a refreshed pair, and every pair of a dense run, still needs the maximum of S products. On an A100, tensor cores multiply BF16 matrices $16\times$ faster than the FP32 units that the dense twin uses, but they round the inputs and accumulate with truncation. We use them only to *filter*: a tensor-core product with a rigorous error bound decides which vectors could attain the FP32 maximum, and those few are recomputed with the dense twin's own FP32 routine, as in filter-and-refine exact search (Weber et al., 1998) but with the approximation error bounded by rounding analysis.

Two-term BF16 queries. All document vectors in our experiments are stored in BF16 (ColNomic and ColPali emit BF16, and we store ColBERTv2's vectors in BF16), so converting them for the tensor cores is exact, and the dense twin reads the same values in FP32. A query token is split as $q = q^{\mathrm{hi}} + q^{\mathrm{lo}} + r$ with $q^{\mathrm{hi}} = \mathrm{bf16}(q)$ and $q^{\mathrm{lo}} = \mathrm{bf16}(q - q^{\mathrm{hi}})$, rounding to nearest, so that $|r_k| \leq 2^{-16}|q_k|$ (Ootomo & Yokota, 2022). Products of two BF16 numbers are exact in FP32, and the approximate score $\tilde{s}_j = \langle q^{\mathrm{hi}}, v_{dj}\rangle + \langle q^{\mathrm{lo}}, v_{dj}\rangle$ costs two tensor-core products.

Theorem 4 (Exact maxima from tensor cores)**.** *Assume no underflow or overflow, let the vectors of document d be BF16 numbers with $\|v_{dj}\|_2 \leq V_d$, and let $\tilde{s}_j$ be the sum of the $2h$ exact products $q_k^{\mathrm{hi}} v_{djk}$ and $q_k^{\mathrm{lo}} v_{djk}$, accumulated in binary32 with an error of at most γ' times the sum of their magnitudes. Let H be the indices of the R largest $\tilde{s}_j$ (ties by index), $U = \max_{j\notin H}\tilde{s}_j + c_h\|q\|_2 V_d$ with*

$$c_h = (1 + 2^{-7} + 2^{-16})\,\gamma' + 2^{-16} + \gamma_h, \tag{4}$$

and let m and w be the largest value and the smallest maximizing index of the dense twin's FP32 products $\hat{s}_j$ over $j \in H$. If $m > U$, then $m = \max_j \hat{s}_j$ and w is the smallest index attaining it, i.e.,

the dense twin's value and selection. Moreover $m - U$ is a lower bound on the FP32 gap between w and every vector outside H.

The proof (Appendix C) bounds $|\tilde{s}_j - \hat{s}_j| \le c_h \|q\|_2 V_d$ for every j by combining the split error, the accumulation error and the FP32 error of the dense twin (Higham, 2002); ties and near-ties can only make the test fail, never make it wrong. We take $\gamma' = \gamma'_{2h} = 2hu'/(1-2hu')$ with $u' = 2^{-22}$ (Higham, 2002), which covers the measured behavior of NVIDIA tensor cores (Fasi et al., 2021); the bit-accurate A100 model of Khattak & Mikaitis (2026) gives $\gamma' \le 1.9\times10^{-5}$, $3.2\times$ below our $\gamma'_{256} = 6.1\times10^{-5}$ (Appendix C). We use $2c_h = 1.69\times10^{-4}$ as the coefficient and round U up and $m - U$ down; lockstep audits compare every call with the dense twin.

Two uses. With $R = 1$, Theorem 4 yields an *exact dense* operator: per token, the tensor-core winner is recomputed in FP32 and accepted when it exceeds the bound U built from the runner-up; rare rejected tokens fall back to an FP32 scan. With $R = r{+}1$, it yields an exact *refresh* for the certified operator: the working set, its FP32 values and a gap $\mathrm{fl}_\downarrow(m - U)$ no larger than the FP32 gap, so Theorem 2 continues to hold.

5 Executing exact TTO on a GPU

Document-major execution. A skipped interaction saves time only if its memory is not read, and a pair-parallel kernel reads a document once per failing token (Appendix D). We therefore organize each call by document. One thread block owns a document d of one query and all N of its tokens and proceeds in three phases (Algorithm 1 in Appendix D): *certify* applies the FP32 test of Theorem 2 to every token and evaluates the working sets of the certified ones; *compact* appends the failing tokens to a per-document list; and *refresh* streams the document once for all of them, maintaining per-token top-$(r{+}2)$ lists, split over several thread blocks when the pools are small.

Tensor-core kernels. The refresh kernel reads documents in BF16, which halves the traffic and is exact (Section 4), keeps per-token lists of $r{+}2$ candidates by their tensor-core scores, recomputes the $r{+}1$ head candidates in FP32 and applies Theorem 4. Our exact dense operator is a Triton kernel derived from FLASH-MaxSim (Pony et al., 2026) that splits 32 query tokens into two BF16 terms in registers and tracks the maximum, its index and the runner-up of a document segment; a second kernel recomputes each winner in FP32 and applies Theorem 4 with $R = 1$, with an FP32 fallback.

Bitwise identity by construction. Every FP32 value that leaves our operators is computed by one routine: a single lane accumulates $\sum_k q_{ik} v_{djk}$ with fused multiply–adds in the order $k = 0, \ldots, h-1$, starting from zero, without fast-math. The working-set phase, the FP32 refresh, the verification of tensor-core candidates and the dense twin all use it, so the same product has the same bits wherever it is computed; tensor cores only decide *which* products are computed. By Theorem 2(ii) and Theorem 4, every operator returns the twin's values and selections, and since the deterministic backward pass needs only the selected vectors, gradients and iterates match bitwise.

A fused, batch-invariant loop. After scoring, a step needs the document scores, the softmax and the gradient of the loss with respect to them, the gradient of every query token (a sum over documents of selected vectors weighted by these document gradients), and the Adam update. Automatic differentiation launches many small kernels for this, and at GQR's pool sizes they can take longer than the scoring itself. We compute all of it in one kernel per step. Each block of its grid owns one query and 128 of its coordinates: it recomputes the query's document scores, softmax and document gradients with a reduction tree that depends only on the thread layout, accumulates each coordinate's gradient over the pool in document order with fused multiply–adds, and applies Adam in binary32 with explicitly rounded operations, to which Lemma 2 applies. The kernel takes any scorer that returns per-token maxima (or their sums) and the selected vectors (or their indices), so GQR's scorer and FLASH-MaxSim run in it unchanged. No kernel synchronizes with the host, so an entire run (initial call, T steps and final scoring) is captured once in a CUDA graph and replayed. A batch pads short queries and small pools; padding is masked out of every sum and never updated.

Proposition 2 (Batch invariance)**.** *Let every query of a batch occupy the leading positions of its rows, with padded tokens and documents after them. If the scorer's outputs for a query's real tokens*

and documents (maxima and selected vectors, or sums over its real tokens) depend only on those tokens and documents, then on a given GPU the fused loop returns every query's iterates, optimizer state and final scores bit for bit as when the query runs alone, whatever the other queries, their order and the amount of padding.

Proof. A block reads only its own query's inputs. Document-level reductions run over the 128 threads of a block in a tree fixed by the thread layout, in which masked and out-of-range threads contribute the same identities ($+0$ to sums, $-\infty$ to maxima); sums over tokens and documents run sequentially in token and pool order and skip padded entries. Every operation of a query thus receives the same operands in the same order as alone; induction over steps concludes. □

Our exact operators meet the hypothesis; a library matrix product does not, as its kernel and summation order follow the batch shape (He & Thinking Machines Lab, 2025).

6 EXPERIMENTS

We ask four questions: do we reproduce GQR's gains under its protocol (Section 6.2), how fast is ExactTTO for single requests and for batches (Section 6.3), is it batch-invariant where other scorers are not (Section 6.4), and does it carry over to another TTO objective (Section 6.5)?

6.1 PROTOCOL

Benchmarks and retrievers. We use the ten ViDoRe v1 and four ViDoRe v2 splits (Faysse et al., 2025; Macé et al., 2025) with ColNomic-7B (Nomic Team, 2025) (747–779 vectors per page) and ColPali-v1.3 (Faysse et al., 2025) (1,030), and four BEIR datasets (Thakur et al., 2021) (SciFact, NFCorpus, FiQA, SciDocs) with ColBERTv2 (Santhanam et al., 2022b) (up to 180 vectors per passage), all with $h = 128$. The complementary retrievers are Qwen3-Embedding-4B (Zhang et al., 2025) and, on ViDoRe, Linq-Embed-Mistral (Choi et al., 2024) on OCR text (Appendix E). GQR's pipeline pads ColNomic's queries with zero vectors (23.4% of all query tokens) that TTO moves like real tokens (Appendix F); we run ColNomic both *official*, as released, and *clean*, without them.

Loops, baselines and timing. Every scorer runs in two graph-captured loops with the same objective and optimizer but different floating-point trajectories (Section 6.4): GQR's *autograd loop* (PyTorch operators and automatic differentiation) and our fused loop (Section 5). The baselines are GQR's released loop, run eagerly, GQR's FP32 scorer (`einsum` and `max`), and FLASH-MaxSim (Pony et al., 2026), differentiated for the query only, with IEEE FP32 products or its released, inexact TF32 path. ExactTTO runs one of our exact operators, dense or certified ($r{+}1 = 9$), in FP32 or on tensor cores, chosen by timing each dataset's first two development queries alone: among all of them for single requests, between the two tensor-core operators for batches. We time the first twelve test queries alone ($B = 1$) and serve the first 64 as one batch or as length-grouped batches of 16 (Appendix E); timings cover the TTO loop with the pool given, as medians of five warm graph replays (three eager runs) on one A100-SXM4-80GB, and exclude encoding, first-stage retrieval and queueing.

6.2 QUALITY UNDER GQR'S PROTOCOL

We recover GQR's gains where it reports numbers (Table 1): on ViDoRe v2, ColNomic alone scores 60.4 nDCG@5 (60.3 reported), and GQR adds $+1.0$ points with Qwen3 ($+0.7$ reported) and $+4.9$ with Linq ($+2.5$ reported). On BEIR, GQR improves ColBERTv2 by $+10.2$ points but stays below Qwen3 alone (49.2). The top-50 of each retriever raise the pool recall on ViDoRe v2 from 74.7 to 91.9% but change nDCG@5 by only -0.4 points (Appendix E), so GQR's requests stay small; more work comes from batching requests. Table 1 reports GQR; Section 6.4 ties our operators to it.

6.3 SPEED: SINGLE REQUESTS AND BATCHES

Single requests. Table 2 times one TTO run. The released loop is dominated by framework overhead; a CUDA graph removes the host, and the fused loop replaces the many small kernels of the

Table 1: **GQR under its released protocol** (clean ColNomic queries; test nDCG@5, %, macro-averaged over datasets, number in parentheses). Primary and Guide score the whole corpus alone; GQR runs at the operating point selected on the development split. Δ: GQR minus Primary (points) with a 95% stratified bootstrap interval, green or orange when it excludes zero; ↑/↓: datasets with a significant paired change ($p < 0.05$). Table 10 adds zero-padded queries and fusion.

	Retrievers		Alone					
Benchmark	Primary	Guide	Primary	Guide	GQR	Δ	95% CI	↑/↓
ViDoRe v1 (10)	ColNomic	Qwen3	89.4	74.1	89.3	0.0	[−0.2, +0.1]	1/0
	ColNomic	Linq	89.4	71.5	89.3	−0.1	[−0.2, 0.0]	0/2
	ColPali (9)	Qwen3	86.4	77.7	87.8	+1.4	[+0.8, +1.9]	3/0
	ColPali (9)	Linq	86.4	75.3	87.3	+0.9	[+0.2, +1.6]	0/0
ViDoRe v2 (4)	ColNomic	Qwen3	60.4	50.3	61.4	+1.0	[−0.5, +2.6]	0/0
	ColNomic	Linq	60.4	58.8	65.3	+4.9	[+2.7, +7.0]	3/0
	ColPali (3)	Qwen3	54.4	48.6	55.2	+0.8	[−0.6, +2.1]	1/0
	ColPali (3)	Linq	54.4	56.2	59.9	+5.5	[+3.9, +7.1]	3/0
BEIR (4)	ColBERTv2	Qwen3	36.2	49.2	46.4	+10.2	[+9.2, +11.2]	4/0

Table 2: Latency (ms) of one whole TTO run ($B = 1$, pool given) at each dataset's operating point (guide: Qwen3; geometric means over datasets of medians over twelve test queries; ColBERTv2 on BEIR, the others on ViDoRe). Released: GQR's loop, eager; Autograd: GQR's scorer in the autograd loop, captured; fused loop: ours, with GQR's scorer, FLASH-MaxSim (FP32 or †TF32, not exact) or ExactTTO (the exact operator selected on development queries). Right: speedups of ExactTTO over the same-named columns. ColNomic*: official zero-padded queries.

			Fused loop (ours)				ExactTTO: speedup over		
Primary	Released	Autograd	GQR	FLASH FP32	FLASH TF32†	ExactTTO	Released	Autograd	GQR
ColNomic	29.1	2.13	0.78	4.88	0.88	1.38	21	1.54	0.56
ColNomic*	30.3	2.53	0.96	5.75	1.03	2.15	14	1.17	0.45
ColPali	24.6	1.99	0.80	5.51	0.91	1.53	16	1.30	0.52
ColBERTv2	16.8	0.98	0.32	0.81	0.28	0.56	30	1.74	0.58

loss, backward pass and optimizer with one per step, which makes GQR's scorer 2.73× and ExactTTO 1.73× faster than in the captured autograd loop (ColNomic). A ColNomic refinement with ExactTTO takes 1.38 ms, against 29.1 ms for the released eager loop and 0.78 ms for GQR's scorer in our fused loop. A single request uses a small part of the GPU and every call is latency-bound, so ExactTTO trails GQR's scorer here: certification and FP32 verification add dependent steps.

Batches. We benchmark warm batches of TTO requests, not an end-to-end serving system (Fig. 2; Tables 15 and 16). With more work per call, tensor cores pay off: at $B = 64$, ExactTTO processes 3.05× as many queries per second as GQR's FP32 scorer in the same fused loop and 3.95× as many as FLASH-MaxSim in FP32 (geometric means over 44 combinations; 3.04× and 3.87× on random queries), and the inexact TF32 path is 1.50× faster still. With top-50 pools, TF32's lead nearly vanishes on ColNomic: ExactTTO serves 4.84× as many queries as GQR's scorer, TF32 only 1.03× as many as ExactTTO, and ExactTTO is faster on 7 of 14 datasets (Table 16). Development queries pick the certified operator for 15 combinations and the exact dense one elsewhere, serving 1.27× (ColNomic) and 1.22× (ColPali) as many queries as dense scoring; hindsight would add 2.5% on average, and timing the development queries as one batch cuts this to 1.4% (Appendix E).

6.4 Exactness and batch invariance

Batch invariance. Batching changes the bits of GQR's scorer, a library matrix product whose kernel and summation order depend on the batch shape: at $B = 64$, 19.6% of the queries end with final scores that differ from those computed alone (none with another top-5), from 0.0% (official ColNomic) to 100.0% (ColBERTv2) per retriever (Fig. 2b). In its autograd loop, whose backward pass

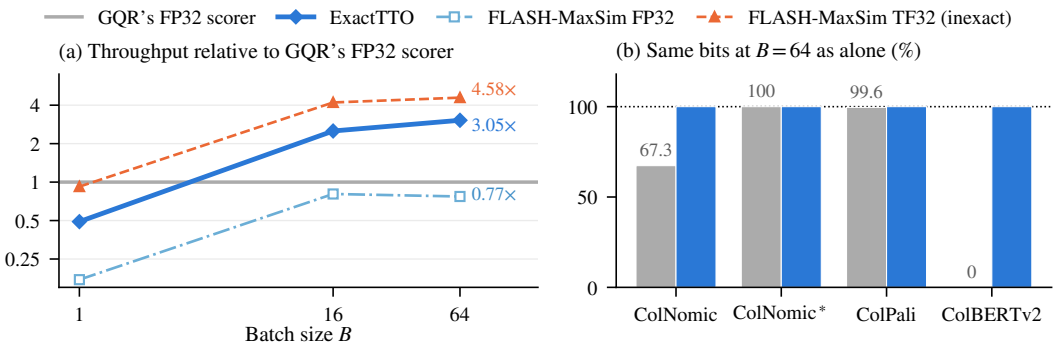

Figure 2: **Serving TTO in batches** (first 64 test queries of 44 dataset–retriever combinations, one fused loop; ExactTTO: tensor-core operators). (a) Throughput over GQR's scorer at equal B (geometric mean). (b) Queries scored bit-identically at $B = 64$ and alone (ExactTTO: both operators).

is another such product, all but 6 of 192 queries of controlled batches of 16 change their final scores (Appendix D). ExactTTO returns every query's bits unchanged (Proposition 2), also in random mixed-length and reversed batches and under extra padding (960 comparisons each, Appendix D), as do both FLASH-MaxSim kernels, which score each query–document pair in one program.

Audits. The guarantees hold by construction under the assumptions of Theorems 2 and 4; no bit differed between any exact operator and the dense twin over 57.6 million token–document maxima in lockstep, 216 comparisons of final queries and scores in the fused loop (Appendix D), 8,346 query checks of Section 6.3 and 9.1 million comparisons under injected ties and near-ties (Appendix C). On the first 96 test queries of every dataset (7,688 queries), the twin's top-5 equals that of GQR's released loop for every query, so on them our exact operators, run in the autograd loop, give the rankings behind Table 1. Other scorers change results: TF32 selects a different vector than FP32 for 0.13% of the maxima and changes the top-5 of 0.9% of the audited queries, and FLASH-MaxSim in FP32 for 0.4%. Neither changes average nDCG@5 significantly ($p \geq 0.17$); TF32 moves the nDCG@5 of 17 zero-padded queries, 9 up and 8 down, by up to 38.7 points. Even two FP32 loops differ: our fused loop and GQR's autograd loop end with queries up to 12.1% apart (relative max-norm). This rarely changes quality: in the fused loop we time, ExactTTO gives 2,778 of the 2,782 served test queries the nDCG@5 they have under GQR's released loop; the other four, all official ColNomic, move their group's mean by less than 0.01 points.

What limits the speedups. A document is read whenever a query token fails on it, and with FP32 operators a certified run takes $0.16 + 0.86\,\pi$ of a dense run's time, where π is the fraction of document–call pairs with a failure ($R^2 = 0.97$, Fig. 3). With 100-document pools, the certified operator refreshes 3.8% of the documents per call at $\eta = 10^{-5}$ and runs 9.19× faster than FLASH-MaxSim in FP32, but refreshes 84.5% at $\eta = 10^{-3}$ (Table 7); on GQR's pools at $\eta = 10^{-4}$, it refreshes 80.6% with official ColNomic's zero tokens and 11.9% without them (Tables 7 and 8).

6.5 A SECOND OBJECTIVE: DISTILLING A RERANKER

ExactTTO does not depend on GQR's target. ReFIT (Reddy et al., 2023) distills a reranker's scores over the first-stage top-100 into the query and retrieves again; we apply it to ColBERTv2 on BEIR with bge-reranker-v2-m3 as the teacher ($\alpha = 1$) and select (η, T) on development queries, $T = 0$ allowed (Appendix E). Refinement raises nDCG@10 of full-corpus retrieval significantly on all four datasets, from 37.1 to 41.5 on average, where reranking the top-100 with the teacher gives 41.1; Recall@100 rises on all four, significantly on two (Table 3). With 100-document pools, at $B = 64$, ExactTTO (with the exact dense operator on every dataset, whose selected steps are large) runs 3.16× faster than GQR's scorer and 3.36× faster than FLASH-MaxSim in FP32 (its TF32 path is 1.59× faster still), with final scores identical to the dense twin's in all 568 checks of 264 queries.

Table 3: Distilling bge-reranker-v2-m3 into ColBERTv2 queries over the top-100, then retrieving from the whole corpus. Teacher: the cross-encoder's order of the top-100; η, T: selected step size and steps. $^*p < 0.05$ against ColBERTv2 (paired permutation test). Right: TTO-loop throughput of ExactTTO at $B = 64$ over GQR's scorer and FLASH-MaxSim FP32 (average: geometric mean).

	nDCG@10 (%)			Recall@100 (%)				$B = 64$ speedup	
Dataset	ColBERTv2	+TTO	Teacher	ColBERTv2	+TTO	η	T	vs GQR	vs FP32
SciFact	64.5	71.2*	71.3	89.0	92.2*	0.5	10	3.08	3.30
NFCorpus	33.5	35.4*	34.2	27.9	28.9	0.1	10	3.14	3.37
FiQA	34.7	41.0*	41.2	63.3	64.7	0.2	50	3.25	3.40
SciDocs	15.7	18.5*	17.7	36.2	40.5*	0.01	5	3.18	3.36
Average	37.1	41.5	41.1	54.1	56.6			3.16	3.36

7 Related work

Test-time optimization for retrieval. Query refinement from feedback goes back to Rocchio (Rocchio, 1971) and has been adapted to dense and multi-vector retrievers (Yu et al., 2021; Wang et al., 2021b). TOUR, ReFIT and ranking-aware reward maximization optimize a query embedding against a cross-encoder, a reranker or a ranking reward (Sung et al., 2023; Reddy et al., 2023; Chen & Wu, 2026), and GQR guides a vision retriever with a text retriever (Uzan et al., 2026). These methods share the loop that ExactTTO executes; we evaluate GQR and reranker distillation.

Efficient late interaction. Compression, centroid pruning and filtering (Khattab & Zaharia, 2020; Santhanam et al., 2022b;a; Scheerer et al., 2025; Nardini et al., 2024) and reductions to token or single-vector retrieval (Lee et al., 2023; Dhulipala et al., 2024) accelerate a first-stage call, mostly approximately; FLASH-MaxSim fuses a call into an IO-aware tensor-core kernel (Pony et al., 2026; Dao et al., 2022). We exploit the structure *between* the calls of an optimization loop.

Skipping work exactly. Exact inner-product search (Ram & Gray, 2012; Teflioudi et al., 2015; Li et al., 2017), filter-and-refine search (Weber et al., 1998), exact k-means with drifting centers (Elkan, 2003; Hamerly, 2010; Ding et al., 2015) and safe screening (El Ghaoui et al., 2012; Fercoq et al., 2015; Ndiaye et al., 2017) skip provably useless work. Our certificates hold inside a nonconvex objective whose gradient flows through the selected vector, and in finite precision; a concurrent pair-parallel certified kernel (anonymous) is a baseline in Appendix D.

Numerics and reproducibility. Tensor-core rounding has been modeled bit-accurately (Fasi et al., 2021; Khattak & Mikaitis, 2026), and multi-term splits recover single-precision accuracy (Ootomo & Yokota, 2022) but not bitwise reproducibility. Batch-invariant kernels remove the main source of irreproducible LLM inference (He & Thinking Machines Lab, 2025); ExactTTO obtains batch invariance from exactness and a fixed-order loop without giving up tensor cores.

8 Limitations and conclusion

Limitations. ExactTTO makes a TTO run exact, not better, and exactness is relative to one FP32 routine: other FP32 implementations need not agree (Section 6.4). We measured one GPU type ($h = 128$, up to 96 query tokens); the accumulation bound of Theorem 4 must be re-established for others. With GQR's pools, single requests run faster with GQR's scorer; our speedups need batches or larger pools, large steps and zero-vector tokens reduce what certification saves, and FLASH-MaxSim's inexact TF32 path remains faster on average.

Conclusion. ExactTTO accelerates batched TTO while matching a fixed FP32 reference bit for bit. Letting approximation decide only which products are computed, it skips rescans that cannot change the result, takes FP32 maxima exactly from tensor cores and runs a fixed-order fused loop, so batches are served several times faster than with GQR's FP32 scorer and every result is independent of its batch.

REPRODUCIBILITY STATEMENT

The proof of Proposition 2 is in Section 5; all other proofs are in Appendix A, with the floating-point analysis of Theorem 2 in Appendix B and that of Theorem 4, including the tensor-core accumulation model, in Appendix C. Appendix D specifies the kernels, their state, the fused loop and the bitwise-identity construction. Appendix E describes the data, the embeddings and text extraction, our implementation of GQR's released protocol and its check against the released code, the selected operating points, the timing and serving protocols, the reranker-distillation protocol, the lockstep audit and the test-set audit. Every number in the paper is generated by a script from logged measurements, and the code, the scripts that prepare embeddings, the logs and the generators will be released.

ETHICS STATEMENT

This work accelerates an existing retrieval procedure while matching a fixed FP32 reference; it introduces no new data, models or human-subject studies. We use public benchmarks and released model checkpoints under their licenses.

AI USE STATEMENT

LLM-based assistants helped polish the writing and helped modify and implement the code of our kernels and experiments.

REFERENCES

Jianlyu Chen, Shitao Xiao, Peitian Zhang, Kun Luo, Defu Lian, and Zheng Liu. M3-Embedding: Multi-linguality, multi-functionality, multi-granularity text embeddings through self-knowledge distillation. In *Findings of the Association for Computational Linguistics: ACL 2024*, pp. 2318–2335, 2024.

Tianyu Chen and Jiaxing Wu. Test-time optimization of query embeddings with ranking aware reward maximization. *arXiv preprint arXiv:2608.12569*, 2026.

Chanyeol Choi, Junseong Kim, Seolhwa Lee, Jihoon Kwon, Sangmo Gu, Yejin Kim, Minkyung Cho, and Jy-yong Sohn. Linq-Embed-Mistral technical report. *arXiv preprint arXiv:2412.03223*, 2024.

Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. FlashAttention: Fast and memory-efficient exact attention with IO-awareness. In *Advances in Neural Information Processing Systems*, 2022.

Laxman Dhulipala, Majid Hadian, Rajesh Jayaram, Jason Lee, and Vahab Mirrokni. MUVERA: Multi-vector retrieval via fixed dimensional encodings. In *Advances in Neural Information Processing Systems*, 2024.

Yufei Ding, Yue Zhao, Xipeng Shen, Madanlal Musuvathi, and Todd Mytkowicz. Yinyang K-means: A drop-in replacement of the classic K-means with consistent speedup. In *International Conference on Machine Learning*, pp. 579–587, 2015.

Laurent El Ghaoui, Vivian Viallon, and Tarek Rabbani. Safe feature elimination for the Lasso and sparse supervised learning. *Pacific Journal of Optimization*, 8(4):667–698, 2012.

Charles Elkan. Using the triangle inequality to accelerate k-means. In *International Conference on Machine Learning*, 2003.

Massimiliano Fasi, Nicholas J. Higham, Mantas Mikaitis, and Srikara Pranesh. Numerical behavior of NVIDIA tensor cores. *PeerJ Computer Science*, 7:e330, 2021.

Manuel Faysse, Hugues Sibille, Tony Wu, Bilel Omrani, Gautier Viaud, Céline Hudelot, and Pierre Colombo. ColPali: Efficient document retrieval with vision language models. In *International Conference on Learning Representations*, 2025.

Olivier Fercoq, Alexandre Gramfort, and Joseph Salmon. Mind the duality gap: Safer rules for the Lasso. In *International Conference on Machine Learning*, 2015.

Greg Hamerly. Making k-means even faster. In *Proceedings of the SIAM International Conference on Data Mining*, pp. 130–140, 2010.

Horace He and Thinking Machines Lab. Defeating nondeterminism in LLM inference. Thinking Machines Lab: Connectionism, 2025. `https://thinkingmachines.ai/blog/defeating-nondeterminism-in-llm-inference/`.

Nicholas J. Higham. *Accuracy and Stability of Numerical Algorithms*. SIAM, 2nd edition, 2002.

Omar Khattab and Matei Zaharia. ColBERT: Efficient and effective passage search via contextualized late interaction over BERT. In *Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval*, pp. 39–48, 2020.

Faizan A. Khattak and Mantas Mikaitis. Accurate models of NVIDIA tensor cores. *ACM Transactions on Architecture and Code Optimization*, 23(3):105, 2026.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *International Conference on Learning Representations*, 2015.

Jinhyuk Lee, Zhuyun Dai, Sai Meher Karthik Duddu, Tao Lei, Iftekhar Naim, Ming-Wei Chang, and Vincent Y. Zhao. Rethinking the role of token retrieval in multi-vector retrieval. In *Advances in Neural Information Processing Systems*, 2023.

Hui Li, Tsz Nam Chan, Man Lung Yiu, and Nikos Mamoulis. FEXIPRO: Fast and exact inner product retrieval in recommender systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2017.

Quentin Macé, António Loison, and Manuel Faysse. ViDoRe benchmark V2: Raising the bar for visual retrieval. *arXiv preprint arXiv:2505.17166*, 2025.

Franco Maria Nardini, Cosimo Rulli, and Rossano Venturini. Efficient multi-vector dense retrieval with bit vectors. In *Advances in Information Retrieval: 46th European Conference on Information Retrieval*, pp. 3–17, 2024.

Eugene Ndiaye, Olivier Fercoq, Alexandre Gramfort, and Joseph Salmon. Gap safe screening rules for sparsity enforcing penalties. *Journal of Machine Learning Research*, 18(128):1–33, 2017.

Nomic Team. Nomic Embed Multimodal: Interleaved text, image, and screenshots for visual document retrieval. `https://www.nomic.ai/news/nomic-embed-multimodal`, 2025. Model nomic-ai/colnomic-embed-multimodal-7b.

Hiroyuki Ootomo and Rio Yokota. Recovering single precision accuracy from tensor cores while surpassing the FP32 theoretical peak performance. *The International Journal of High Performance Computing Applications*, 36(4):475–491, 2022.

Roi Pony, Adi Raz Goldfarb, Idan Friedman, Daniel Ezer, and Udi Barzelay. FLASH-MAXSIM: IO-aware fused kernels for late-interaction retrieval. *arXiv preprint arXiv:2605.29517*, 2026.

Parikshit Ram and Alexander G. Gray. Maximum inner-product search using cone trees. In *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 931–939, 2012.

Revanth Gangi Reddy, Pradeep Dasigi, Md Arafat Sultan, Arman Cohan, Avirup Sil, Heng Ji, and Hannaneh Hajishirzi. ReFIT: Relevance feedback from a reranker during inference. *arXiv preprint arXiv:2305.11744*, 2023.

J. J. Rocchio. Relevance feedback in information retrieval. In Gerard Salton (ed.), *The SMART Retrieval System: Experiments in Automatic Document Processing*, pp. 313–323. Prentice-Hall, 1971.

Keshav Santhanam, Omar Khattab, Christopher Potts, and Matei Zaharia. PLAID: An efficient engine for late interaction retrieval. In *Proceedings of the 31st ACM International Conference on Information and Knowledge Management*, pp. 1747–1756, 2022a.

Keshav Santhanam, Omar Khattab, Jon Saad-Falcon, Christopher Potts, and Matei Zaharia. ColBERTv2: Effective and efficient retrieval via lightweight late interaction. In *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pp. 3715–3734, 2022b.

Jan Luca Scheerer, Matei Zaharia, Christopher Potts, Gustavo Alonso, and Omar Khattab. WARP: An efficient engine for multi-vector retrieval. In *Proceedings of the 48th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pp. 2504–2512, 2025.

Charlie Victor Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. Scaling LLM test-time compute optimally can be more effective than scaling parameters for reasoning. In *International Conference on Learning Representations*, 2025.

Yu Sun, Xiaolong Wang, Zhuang Liu, John Miller, Alexei A. Efros, and Moritz Hardt. Test-time training with self-supervision for generalization under distribution shifts. In *International Conference on Machine Learning*, 2020.

Mujeen Sung, Jungsoo Park, Jaewoo Kang, Danqi Chen, and Jinhyuk Lee. Optimizing test-time query representations for dense retrieval. In *Findings of the Association for Computational Linguistics: ACL 2023*, pp. 5731–5746, 2023.

Christina Teflioudi, Rainer Gemulla, and Olga Mykytiuk. LEMP: Fast retrieval of large entries in a matrix product. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2015.

Nandan Thakur, Nils Reimers, Andreas Rücklé, Abhishek Srivastava, and Iryna Gurevych. BEIR: A heterogeneous benchmark for zero-shot evaluation of information retrieval models. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*, 2021.

Omri Uzan, Asaf Yehudai, Roi Pony, Eyal Shnarch, and Ariel Gera. Guided query refinement: Multimodal hybrid retrieval with test-time optimization. In *International Conference on Learning Representations*, 2026. arXiv:2510.05038.

Dequan Wang, Evan Shelhamer, Shaoteng Liu, Bruno Olshausen, and Trevor Darrell. Tent: Fully test-time adaptation by entropy minimization. In *International Conference on Learning Representations*, 2021a.

Xiao Wang, Craig Macdonald, Nicola Tonellotto, and Iadh Ounis. Pseudo-relevance feedback for multiple representation dense retrieval. In *Proceedings of the ACM SIGIR International Conference on Theory of Information Retrieval*, 2021b.

Roger Weber, Hans-Jörg Schek, and Stephen Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *International Conference on Very Large Data Bases*, pp. 194–205, 1998.

HongChien Yu, Chenyan Xiong, and Jamie Callan. Improving query representations for dense retrieval with pseudo relevance feedback. In *Proceedings of the 30th ACM International Conference on Information and Knowledge Management*, 2021.

Yanzhao Zhang, Mingxin Li, Dingkun Long, Xin Zhang, Huan Lin, Baosong Yang, Pengjun Xie, An Yang, Dayiheng Liu, Junyang Lin, Fei Huang, and Jingren Zhou. Qwen3 embedding: Advancing text embedding and reranking through foundation models. *arXiv preprint arXiv:2506.05176*, 2025.

A PROOFS

Proof of Proposition 1. Let $j \notin H_{di}$. By the definition of the gap and the Cauchy–Schwarz inequality,

$$\begin{aligned}\langle q_i, v_{dj^\star} - v_{dj}\rangle &= \langle q_i^{\mathrm{ref}}, v_{dj^\star} - v_{dj}\rangle + \langle \Delta q_i, v_{dj^\star} - v_{dj}\rangle \\ &\geq m_{di} - \|\Delta q_i\|_2 \, \|v_{dj^\star} - v_{dj}\|_2 \;\geq\; m_{di} - R_{dj^\star} \|\Delta q_i\|_2 \;>\; 0.\end{aligned}$$

Hence every tail product is strictly below the product of $j^\star \in H_{di}$, so all maximizers lie in H_{di}, the maximum over H_{di} equals the maximum over the document, and the smallest maximizing index over the document is found in H_{di}. □

Proof of Theorem 1. Index the scoring calls of a TTO run by $c = 0, 1, \ldots$ and let $q^{(c)}$ and $\tilde{q}^{(c)}$ be the queries of the full-scan run and the certified run. We show by induction that $\tilde{q}^{(c)} = q^{(c)}$ and that the optimizer states coincide. The base case holds because both runs start from $q^{(0)}$ with the same fresh optimizer. Assume the claim for c. For each pair, the certified operator either scans the document, or evaluates H_{di} when the test passes, in which case Proposition 1 gives the same maximum and the same smallest maximizing index as a full scan. Hence both runs obtain identical scores $s(q^{(c)})$ and selected indices $\hat{\jmath}_{di}(q^{(c)})$. The loss depends on the scores only (in GQR also on the target p, computed from the scores of the initial call, which coincide by the same argument), and its gradient with respect to q_i is $\sum_d \partial\mathcal{L}/\partial s_d \cdot v_{d\hat{\jmath}_{di}}/N$, which is therefore identical. A deterministic optimizer fed identical gradients from identical states produces identical updates, so $\tilde{q}^{(c+1)} = q^{(c+1)}$. The certified operator's internal state (references, working sets, gaps) never enters the loss. □

Proposition 3 (Optimal norm-based certificate)**.** *Let $g_j = \langle q_i^{\mathrm{ref}}, v_{dj^\star} - v_{dj}\rangle$ and $\delta_j = \|v_{dj^\star} - v_{dj}\|_2$ for $j \notin H_{di}$, and $\rho_{di} = \min_{j \notin H_{di}} g_j/\delta_j$. Then (i) $\|\Delta q_i\|_2 < \rho_{di}$ implies that $v_{dj^\star}$ strictly beats every tail vector at q_i; (ii) no larger radius has this property: for every $\rho > \rho_{di}$ there is a Δq_i with $\|\Delta q_i\|_2 < \rho$ at which some tail vector ties or beats $v_{dj^\star}$; and (iii) $m_{di}/R_{dj^\star} \leq \rho_{di}$.*

Thus ρ_{di} is the radius of the largest ball around q_i^{ref} inside the polyhedral cone where $j^\star$ beats the tail, the best that any test of the form "$\|\Delta q_i\|_2 < \tau$" can certify for this anchor; the gap–radius test is its cheap lower bound, which needs one stored scalar instead of the distances from $j^\star$ to the whole tail.

Proof of Proposition 3. (i) For $j \notin H_{di}$, $\langle q_i, v_{dj^\star} - v_{dj}\rangle \geq g_j - \|\Delta q_i\|_2 \delta_j > g_j - \rho_{di}\delta_j \geq 0$ whenever $\delta_j > 0$; if $\delta_j = 0$ the two vectors coincide, which the convention $g_j/\delta_j = +\infty$ for $g_j > 0$ and 0 for $g_j = 0$ handles. (ii) If the tail is empty, $\rho_{di} = +\infty$ and there is nothing to show. Otherwise let k attain the minimum. If $\delta_k = 0$, then $v_{dk} = v_{dj^\star}$, the two vectors tie at every query and $\Delta q_i = 0$ serves. Otherwise set $\Delta q_i = -\rho_{di}(v_{dj^\star} - v_{dk})/\delta_k$; then $\|\Delta q_i\|_2 = \rho_{di} < \rho$ and $\langle q_i^{\mathrm{ref}} + \Delta q_i, v_{dj^\star} - v_{dk}\rangle = g_k - \rho_{di}\delta_k = 0$, a tie. (iii) For every $j \notin H_{di}$, $g_j \geq m_{di}$ by equation 3 and $\delta_j \leq R_{dj^\star}$ by the definition of the radius, so $g_j/\delta_j \geq m_{di}/R_{dj^\star}$. □

Lemma 1 (Displacement of bias-corrected Adam)**.** *For Adam with $\beta_1, \beta_2 \in (0, 1)$, zero-initialized moments and bias correction, every coordinate moves by at most $\rho_t\,\eta$ at step t for any gradient sequence, with $\rho_t = \frac{1-\beta_1}{1-\beta_1^t}\sqrt{\frac{1-\beta_2^t}{1-\beta_2}}\big(\sum_{k=0}^{t-1}(\beta_1^2/\beta_2)^k\big)^{1/2}$; for the default $(0.9, 0.999)$, $\rho_1 = 1$, $\rho_{25} = 1.228$ and $\rho_{100} = 2.243$, so $L_i \leq \eta\sqrt{h}\sum_{t\leq T}\rho_t$.*

Proof of Lemma 1. With zero-initialized moments, $m_t = \sum_{s=1}^t a_s g_s$ and $v_t = \sum_{s=1}^t b_s g_s^2$ coordinatewise, where $a_s = (1-\beta_1)\beta_1^{t-s}$ and $b_s = (1-\beta_2)\beta_2^{t-s}$. By Cauchy–Schwarz, $|m_t| \leq \sum_s (a_s/\sqrt{b_s})\sqrt{b_s}|g_s| \leq \big(\sum_s a_s^2/b_s\big)^{1/2}\sqrt{v_t}$, and $\sum_s a_s^2/b_s = \frac{(1-\beta_1)^2}{1-\beta_2}\sum_{k=0}^{t-1}(\beta_1^2/\beta_2)^k$. The bias-corrected step is $\eta\,\hat{m}_t/(\sqrt{\hat{v}_t} + \epsilon)$ with $\hat{m}_t = m_t/(1-\beta_1^t)$ and $\hat{v}_t = v_t/(1-\beta_2^t)$; dropping $\epsilon \geq 0$ gives $|\eta\,\hat{m}_t/(\sqrt{\hat{v}_t} + \epsilon)| \leq \eta\,\frac{\sqrt{1-\beta_2^t}}{1-\beta_1^t}\,\frac{|m_t|}{\sqrt{v_t}} \leq \eta\rho_t$. If $v_t = 0$, all $g_s = 0$ and the step is zero. □

Proof of Theorem 3. The test fails at call t_{k+1}, so $m^{(k)} \leq R^{(k)}\|q_i^{(t_{k+1})} - q_i^{(t_k)}\|_2$, i.e., $\|q_i^{(t_{k+1})} - q_i^{(t_k)}\|_2 \geq \rho^{(k)}$, and the triangle inequality bounds this distance by the path length between the two

calls. Summing over the $K-1$ refreshes after the first gives $(K-1)\rho_{\min} \le L_i$. In binary32, the test fails at t_{k+1} only if $\hat{m}^{(k)} \le \mathrm{fl}(\mathrm{fl}(\tilde{R}^{(k)}\hat{t}) + \varepsilon) \le (1+u)^2\tilde{R}^{(k)}\hat{t} + (1+u)\varepsilon$, and the computed displacement satisfies $\hat{t} \le (1+u)^3\sqrt{1+\gamma_h}\,\|\Delta q_i\|_2$ by the argument of Appendix B; the same bound therefore holds with the certified radius $\rho^{(k)} = \big(\hat{m}^{(k)} - (1+u)\varepsilon\big)/\big((1+u)^5\sqrt{1+\gamma_h}\,\tilde{R}^{(k)}\big)$ whenever $\tilde{R}^{(k)} > 0$ and $\rho^{(k)} > 0$. □

B FLOATING-POINT ANALYSIS

We prove Theorem 2. Write $p_j = \langle q_i, v_{dj}\rangle$ and $p_j^{\mathrm{ref}} = \langle q_i^{\mathrm{ref}}, v_{dj}\rangle$ for exact products and $\hat{p}_j, \hat{p}_j^{\mathrm{ref}}$ for their FP32 values. For an h-term dot product in any order, with or without fused multiply-adds, $|\hat{p}_j - p_j| \le \gamma_h \sum_k |q_{ik}||v_{djk}| \le \gamma_h \|q_i\|_2\|v_{dj}\|_2 \le \gamma_h QV =: \delta$ (Higham, 2002, Eq. (3.5)), and likewise at the reference.

Stored gap. Let $a = \hat{p}_{j^\star}^{\mathrm{ref}}$ and $b = \max_{j\notin H_{di}} \hat{p}_j^{\mathrm{ref}} \le a$. Then $\hat{m} = \mathrm{fl}(a-b) \le (1+u)(a-b)$, and the exact reference gap satisfies $m^\star := p_{j^\star}^{\mathrm{ref}} - \max_{j\notin H_{di}} p_j^{\mathrm{ref}} \ge (a-\delta) - (b+\delta) \ge \hat{m}/(1+u) - 2\delta$.

Displacement. Each difference is rounded once, $\mathrm{fl}(q_{ik} - q_{ik}^{\mathrm{ref}}) = \Delta_k(1+\theta_k)$ with $|\theta_k| \le u$, so the squared differences are at least $(1-u)^2\Delta_k^2$. Their computed sum s of h nonnegative terms satisfies $s \ge (1-\gamma_h)\sum_k \mathrm{fl}(\Delta_k)^2$, and the correctly rounded square root gives $\sqrt{s} \le \hat{t}/(1-u)$. Hence $\|\Delta q_i\|_2 \le \kappa\,\hat{t}$ with $\kappa = 1/\big((1-u)^2\sqrt{1-\gamma_h}\big)$.

Test. With two roundings, $\mathrm{fl}(\mathrm{fl}(\tilde{R}\hat{t}) + \varepsilon) \ge (1-u)^2\tilde{R}\hat{t} + (1-u)\varepsilon$; a fused multiply-add only increases the right-hand side. If the test passes and $\tilde{R} \ge \lambda_h R_{dj^\star}$ with $\lambda_h = \kappa(1+u)/(1-u)^2$, then

$$
\begin{aligned}
m^\star - R_{dj^\star}\|\Delta q_i\|_2 &\ge \frac{\hat{m}}{1+u} - 2\delta - R_{dj^\star}\kappa\hat{t} \;>\; \frac{(1-u)^2\tilde{R}\hat{t} + (1-u)\varepsilon}{1+u} - 2\delta - R_{dj^\star}\kappa\hat{t} \\
&\ge \frac{1-u}{1+u}\,\varepsilon - 2\delta \;\ge\; 2\delta,
\end{aligned}
$$

using $\varepsilon \ge 4\delta(1+u)/(1-u)$. (i) Since $m^\star - R_{dj^\star}\|\Delta q_i\|_2 > 0$, Proposition 1 applies at q_i. (ii) For every $j \notin H_{di}$, as in the proof of Proposition 1, $p_{j^\star} - p_j \ge m^\star - R_{dj^\star}\|\Delta q_i\|_2 > 2\delta$, so $\hat{p}_{j^\star} - \hat{p}_j \ge (p_{j^\star} - \delta) - (p_j + \delta) > 0$. Thus the computed full-scan maximum is attained only inside H_{di}; if the working-set evaluation computes each product exactly as the full scan does, it sees the same values on H_{di} and returns the same maximum and the same smallest maximizing index. □

Implementation of the parameters. Radii are computed once per corpus in FP64 from the Gram identity $\|v_j - v_{j'}\|_2^2 = \|v_j\|_2^2 + \|v_{j'}\|_2^2 - 2\langle v_j, v_{j'}\rangle$. The FP64 error of this *squared* distance is at most $4\gamma_{h+3}^{(64)}V^2$; we add this bound to the squared radius (itself enlarged by $1+2^{-50}$) before taking the square root, multiply the result by $\lambda_h(1+10^{-12})$ and round it up to FP32. V is the largest vector norm of the pool computed in FP64 and multiplied by $1+10^{-12}$, which dominates its rounding error. Q is the largest initial token norm, likewise inflated, plus $3.2\,\eta\sqrt{h}\,T$. Here $\sum_{t\le T}\rho_t \le 3.2\,T/1.4$ for all $T \le 200$ (the ratio is tightest at $T = 200$, where $\rho_{200} = 3.096$), and the following lemma shows that this Q bounds every token norm of the FP32 trajectory. ε is rounded up to FP32, and the kernel is unchanged.

Lemma 2 (Adam in binary32)**.** *Evaluate the update of Lemma 1 in binary32 as our loops do, with each multiply–add rounded once or twice: $m \leftarrow \beta_1 m + (1-\beta_1)g$, $v \leftarrow \beta_2 v + (1-\beta_2)g^2$ and $q \leftarrow q - a_t \cdot m/\big(\sqrt{v}/b_t + \epsilon\big)$, where β_1, β_2, $1-\beta_1$, $1-\beta_2$, $a_t = \eta/(1-\beta_1^t)$ and $b_t = \sqrt{1-\beta_2^t}$ are each rounded to binary32 once, $\epsilon > 0$, and no operation underflows (without flushing subnormals to zero) or overflows. Then for $1 \le t \le 200$, at step t every coordinate moves by at most $\rho_t\eta(1+\delta_t) + u'|q_{ik}^{(t)}|$, with $\delta_t = (5t+10)u$, $u' = u/(1-u)$ and $q^{(t)}$ the updated query, and therefore $\max_{t\le T}\|q_i^{(t)}\|_2 \le \big(\|q_i^{(0)}\|_2 + \eta\sqrt{h}\sum_{t\le T}\rho_t(1+\delta_t)\big)/(1-Tu')$ for $T \le 200$.*

Proof. Every term of the computed first moment is rounded at most twice when it enters and twice per later step (once when the multiply–add is fused), and every term of the computed second moment at most three times when it enters and twice per later step, so the computed moments are $\sum_s \hat{a}_s g_s(1+\theta_s)$ and $\sum_s \hat{b}_s g_s^2(1+\phi_s)$ with $|\theta_s| \le \gamma_{2t}$ and $|\phi_s| \le \gamma_{2t+1}$, where $\hat{a}_s$ and $\hat{b}_s$ are

the weights of Lemma 1 formed with the rounded constants. The Cauchy–Schwarz argument of Lemma 1 bounds the first computed moment by $(1+\gamma_{2t})/\sqrt{1-\gamma_{2t+1}}$ times the bound of Lemma 1 with these weights times the square root of the second, and replacing the exact constants by their binary32 values changes ρ_t by a factor of at most $1+(1.6t+2)u$. The square root, the two divisions, the scaling by a_t and the addition of $\epsilon > 0$ contribute at most five further roundings, so the computed step s satisfies $|s| \le \rho_t\eta\,(1+\gamma_{2t})\big(1+(1.6t+2)u\big)/\big(\sqrt{1-\gamma_{2t+1}}\,(1-u)^5\big) \le \rho_t\eta(1+\delta_t)$ for $t \le 200$. The update $\mathrm{fl}(q-s)$ moves the coordinate by at most $|s| + u'|\mathrm{fl}(q-s)|$. Summing over coordinates and steps, $\|q_i^{(t)}\|_2 \le \|q_i^{(0)}\|_2 + \eta\sqrt{h}\sum_{s\le t}\rho_s(1+\delta_s) + tu'\max_{s\le t}\|q_i^{(s)}\|_2$, which gives the bound. □

For $T \le 200$, $\delta_t \le 6.1\times10^{-5}$ and $Tu' \le 1.2\times10^{-5}$, so for $h = 128$ our Q dominates the bound of Lemma 2 whenever $\eta \ge 6\times10^{-9}\max_i\|q_i^{(0)}\|_2$; in all our experiments the initial token norms are at most 1.005 and $\eta \ge 10^{-5}$.

C Exact maxima from tensor cores

Proof of Theorem 4. Fix j and write $a = \sum_k |q_k||v_{djk}| \le \|q\|_2 V_d$ (Cauchy–Schwarz). Rounding to nearest gives $|q_k - q_k^{\mathrm{hi}}| \le 2^{-8}|q_k|$; the difference $q_k - q_k^{\mathrm{hi}}$ is exact in binary32, and q_k^{lo} rounds it with $|r_k| = |q_k - q_k^{\mathrm{hi}} - q_k^{\mathrm{lo}}| \le 2^{-8}|q_k - q_k^{\mathrm{hi}}| \le 2^{-16}|q_k|$. Hence $|q_k^{\mathrm{hi}}| + |q_k^{\mathrm{lo}}| \le (1+2^{-8})|q_k| + 2^{-8}(1+2^{-8})|q_k| \le (1+2^{-7}+2^{-16})|q_k|$. Products of two BF16 numbers are exact in binary32, so by assumption

$$\Big|\tilde{s}_j - \sum_k (q_k - r_k)v_{djk}\Big| \le \gamma' \sum_k \big(|q_k^{\mathrm{hi}}| + |q_k^{\mathrm{lo}}|\big)|v_{djk}| \le (1+2^{-7}+2^{-16})\,\gamma' a,$$

$$\Big|\sum_k r_k v_{djk}\Big| \le 2^{-16} a,$$

and the dense twin's routine satisfies $|\hat{s}_j - \langle q, v_{dj}\rangle| \le \gamma_h a$ (Higham, 2002, Eq. (3.5)). Together, $|\tilde{s}_j - \hat{s}_j| \le c_h a \le c_h\|q\|_2 V_d$ for every j. For $j \notin H$, therefore, $\hat{s}_j \le \tilde{s}_j + c_h\|q\|_2 V_d \le U < m$: no vector outside H attains or exceeds m, the FP32 maximum over the document is m, all its maximizers lie in H, and the smallest one is w. The same inequality gives $m - \hat{s}_j \ge m - U$ for every $j \notin H$. If a stored gap $\tilde{m} \le m - \max_{j\notin H}\hat{s}_j$ replaces the computed gap $\hat{m}$ of Theorem 2, the proof of that theorem (Appendix B) goes through unchanged, since it uses only $\hat{m} \le (1+u)(a-b)$ for the computed products a of $j^\star$ and b of the best tail vector. □

The A100 accumulation model. Khattak & Mikaitis (2026) model the A100's BF16 multiply–accumulate with FP32 output bit-accurately: blocks of eight exact products and the running accumulator are aligned to the largest exponent among them in a 26-bit window (two integer and 24 fractional bits, i.e., one bit beyond the output precision), truncated, summed exactly, and the result is truncated to binary32. If the largest of the nine terms lies in $[2^E, 2^{E+1})$, alignment loses less than 2^{E-24} on each of the at most eight other terms, and the final truncation less than 2^{-23} of the result, so a block errs by less than $(2^{-21}+2^{-23})$ times the sum of the magnitudes of its terms. In a chain of $2h/8$ blocks, the accumulator entering a block is bounded by the magnitudes of all earlier products, so the total error is at most $(2h/8)(2^{-21}+2^{-23})$ times the sum of the magnitudes of the $2h$ products, i.e., $\gamma' \le 1.9\times10^{-5}$ for $h = 128$. Computing the two terms as separate chains and adding them in FP32, as our kernels do, halves the chain length and only lowers this bound.

Measured accumulation error. We also measured the accumulation error of our filter directly: a kernel with the filter's tile computation writes every approximate score $\tilde{s}_j$, which we compare with the exact sum of its $2h$ products computed in FP64. Over 1.2 billion token–vector pairs of ColNomic, ColPali and ColBERTv2, and on synthetic vectors whose coordinates spread over many binades, the largest error was 1.7×10^{-6} and 2.5×10^{-6} times the sum of the product magnitudes, at least $7\times$ below the bound of the A100 model and $24\times$ below our γ'. A measurement does not prove the assumption, but it shows the margin by which it holds on our data.

Implementing the bound. The token norm is computed in FP32 from the sum of squares, multiplied by $1+10^{-5} \ge 1+\gamma_h$ and square-rooted with upward rounding; V_d is computed per document

in FP64 and rounded up to FP32; the error term $2c_h\|q\|_2 V_d$ and U are formed with upward rounding and the stored gap $m - U$ with downward rounding. Triton converts FP32 to BF16 by rounding to nearest even, which the analysis requires, and at start-up we check that the split computed inside the filter kernel matches PyTorch's bit for bit. In the exact dense operator ($R = 1$) the segments of a document are merged by taking the largest maximum (the earliest segment on ties) as the candidate and the largest of all other approximate scores, including the other segments' maxima, as the runner-up, so a tie across segments makes the test fail rather than select the wrong vector.

Adversarial ties and near-ties. Real trajectories rarely put two candidates within the error bound, so we also test the decision boundary directly. We use random BF16 pools with either one query of 32 tokens against 10 documents of 768 or 1,030 vectors, or four queries against 32 documents. We copy one token's winner in every document to a position about half a document away, before or after it (in another filter segment for the small pools), and keep the copy (an exact tie), change it at a coordinate where the token is zero (a tie in FP32 between different vectors, so the selection is observable), or move 1, 4, 16 or 64 of its coordinates by one BF16 ulp (near-ties). In 48 further runs we place twelve copies, exact or one ulp apart, across the document, more candidates than a working set holds ($r{+}1 = 9$). Over all 192 five-step TTO runs with $\eta \in \{10^{-5}, 10^{-3}\}$, all five exact variants agree bit for bit with the dense twin in all 9.1 million comparisons, each of the 1,184 observable ties goes to the smaller index, and in the single-copy runs the exact dense filter ($R = 1$) falls back on 1.8–3.9% of the token–document pairs, against the 3.1% that carry an injected pair, so both the accepting and the falling-back branch are exercised. With twelve copies, the certified operators refresh 83.4–100.0% of the documents per call (39.7% with one copy), and their tensor-core refresh falls back to FP32 on 2.4–3.1% of the token–document pairs.

D EXECUTION DETAILS

Algorithm 1 One certified call, document-major. The certify kernel performs phases 1 and 2 of Section 5; the three kernels run without host synchronization.

Require: query tokens $q_{1:N}$; documents v_d; per-pair state $(q_i^{\mathrm{ref}}, H_{di}, j^\star, m_{di}, \tilde{R}_{di})$; guard ε; split factor P

Certify (one thread block per query–document pair d)

load $q_{1:N}$ into shared memory; $F_d \leftarrow \emptyset$
for each token i **in parallel over warps do**
  $t \leftarrow \|q_i - q_i^{\mathrm{ref}}\|_2$ ▷ warp reduction
  if $m_{di} > \tilde{R}_{di}\, t + \varepsilon$ **then** ▷ certificate, Theorem 2
    $(s_{di}, \hat{\jmath}_{di}) \leftarrow \max_{j \in H_{di}} \textsc{Dot}(q_i, v_{dj})$ ▷ $r{+}1$ lanes, smallest index on ties
  else append i to the compact list F_d
  end if
end for

Refresh (P thread blocks per document with $F_d \neq \emptyset$; block p reads tiles $p, p{+}P, \dots$)

for each tile $\mathcal{T}$ of 32 vectors of the block **do** ▷ each document vector read once per call
  stage $v_{d\mathcal{T}}$ in shared memory; lane j computes $\textsc{Dot}(q_i, v_{dj})$ for the tokens $i \in F_d$ of its warp
  merge into per-token top-$(r{+}2)$ lists by ballot on the current threshold
end for

Merge (one thread block per document with $F_d \neq \emptyset$)

for $i \in F_d$ **do**: merge the P lists; write $\hat{\jmath}_{di}$, s_{di} and the new $(q_i^{\mathrm{ref}}, H_{di}, j^\star, m_{di}, \tilde{R}_{di})$
end for

State. For every token–document pair the certified operator stores the reference query (h floats), the $r{+}1$ working-set indices in increasing index order (int32), the gap, the inflated radius and the winner, i.e., $4(h + r + 4)$ bytes; for $h = 128$ and $r = 8$ this is 560 bytes per pair, or about 0.2 MB for a query with $N = 37$ tokens and $C = 10$ documents. Working-set vectors are *not* cached: the working-set phase reads them from the document by index, so the state is independent of the document length. The radius index holds one FP32 value per vector and is built once per corpus.

Kernel configuration. The FP32 document-major kernels use thread blocks of 256 threads (8 warps), $N \leq 64$ tokens and $h = 128$. Working-set products are computed by $\lfloor 32/(r{+}1) \rfloor$ tokens per warp and round, one lane per (token, working-set slot), reading the vector's row with 128-bit loads. The refresh kernel holds the failing tokens ($|F_d|h$ floats), one tile of 32 vectors with rows padded to $h{+}4$ floats, so that rows are 16-byte aligned and 128-bit shared-memory loads are free of bank conflicts, and per-token candidate lists of capacity $r{+}2$ (value and index), i.e., at most 56.5 KB. It assigns failing tokens round-robin to warps (at most eight per warp) and one vector of the current tile to each lane, which reads four coordinates of its vector and of each of its tokens per shared-memory access. The split factor is $P = \min\{8, \lceil 4\, n_{\mathrm{SM}}/(BC) \rceil, \lceil S/32 \rceil\}$ for B queries with C documents each, so that 100-document pools use one refresh block per page once $B \geq 5$, and 10-page pools up to eight.

One product, one routine. Every product is computed by a single lane as $a \leftarrow 0$; $a \leftarrow \mathrm{fma}(q_{ik}, v_{djk}, a)$ for $k = 0, \ldots, h-1$. The working-set phase and the refresh phase unroll this loop differently (128-bit global loads versus 128-bit loads from shared-memory tiles) but perform the same sequence of fused multiply-adds on the same operands, so they produce identical bits, however the tiles of a document are split over blocks; the code is compiled without fast-math. Ties are broken everywhere by the total order "larger value first, then smaller index".

Backward pass and capture. Both document-major kernels write the selected vector of every pair; the backward pass computes $\nabla_{q_i} = \sum_d g_d\, v_{d\hat{\jmath}_{di}}$ by a sequential sum over pages in a separate kernel shared by the certified and dense operators, which makes their gradients bitwise identical whenever their selections and scores are. No kernel synchronizes with the host, and the shared-memory attribute is set during warm-up, so the whole TTO loop can be captured in one CUDA graph.

The fused loop. The kernel of Section 5 runs in three modes: an initial mode forms the target p from the initial scores and the complementary scores, a step mode performs one TTO step, and a final mode writes the pool scores. In step mode, a query with n tokens occupies $\lceil nh/128 \rceil$ blocks of 128 threads. Thread d of every block sums the n maxima of document d in token order (or reads their sum, which FLASH-MaxSim returns), so every block holds the same document scores; the maximum, the sum of exponentials and $\langle \sigma, r \rangle$ with $r_d = -p_d/(\sigma_d + 10^{-8})$ are reduced by a fixed tree (five warp shuffles, then the four warp results in a fixed order), which gives the document gradients $g_d = \sigma_d(r_d - \langle \sigma, r \rangle)/(\tau n)$. Each thread then owns one coordinate (i, k): it accumulates $\sum_d g_d v_{d\hat{\jmath}_{di}k}$ over the pool in document order with fused multiply–adds, reading the selected vector from the scorer's output or, for scorers that return indices, from the document, and applies $m \leftarrow \mathrm{fma}(\beta_1, m, (1{-}\beta_1)g)$, $v \leftarrow \mathrm{fma}(\beta_2, v, ((1{-}\beta_2)g)g)$ and $q \leftarrow q - a_t\,(m/(\sqrt{v}/b_t + \epsilon))$, every other operation rounded once, with a_t and b_t formed in double precision and rounded to binary32 once. Padded tokens are neither summed nor updated, and padded documents are masked out of every sum. On 39 controlled configurations (the batches of Table 7 and nine GQR pools from ArxivQA and ESG-human with official and clean queries, $\eta \in \{10^{-5}, 10^{-4}, 10^{-3}\}$), the kernel's target and first-step moments agree with a float64 evaluation of the same step to 1.7×10^{-7} and 1.6×10^{-5} relative error; the fused loop reproduces the final top-5 of the autograd loop for all 219 queries; and batches of 16 return every query's bits unchanged for our operators and FLASH-MaxSim (192 of 192 queries), against 96 for GQR's scorer in the fused loop and 6 in the autograd loop. A stress test of Proposition 2 on 4 datasets (192 test queries at their operating points) draws batches of 16 at random without grouping by length, reverses each batch, and pads single queries to 64 tokens and 40 documents: of 960 comparisons per method, the final query had the same bits as alone in 960 for our exact dense and 960 for our certified operator, 960 and 960 for FLASH-MaxSim in FP32 and TF32, and 550 for GQR's scorer.

Tensor-core kernels. The kernels of Section 4 support up to 96 query tokens. Documents are kept in FP32 for the dense twin and in a BF16 copy, which is exact because the stored vectors are BF16 numbers (checked when the copy is made). The refresh kernel stages tiles of 32 BF16 document vectors in shared memory with asynchronous 16-byte copies into two buffers, so that the next tile loads while the current one is processed; each warp owns up to 4, 8 or 12 failing tokens (a template parameter chosen from the number of failures), whose two BF16 terms are staged once per call, and computes its tokens' approximate scores for the tile with 16×16 BF16 tensor-core fragments;

per-token candidate lists of $r+2$ entries are updated by warp ballots against the current threshold. The merge kernel combines the lists of the blocks that share a document, recomputes the $r+1$ head candidates with the FP32 routine, applies Theorem 4 with upward-rounded bounds, and sends failing tokens to the FP32 refresh of the same kernel family. The exact dense operator uses the Triton kernel of Section 5 with blocks of 32 tokens and 64 vectors; its grid has one program per document, token block and segment, with as many segments as needed to launch about two programs per streaming multiprocessor. The verification kernel assigns one thread per token–document pair, merges the segments, recomputes the winner with the FP32 routine and applies Theorem 4 with $R = 1$; the rejected pairs are refreshed in FP32 by the refresh kernel above. The counters of rejected pairs are reset by the filter kernel itself, so a call launches four kernels (filter, verification, and the refresh and merge of rejected pairs) and no memory initialization.

Design ablation. Table 4 adds the design steps one at a time. With 16 queries and 100-document pools, tensor-core filtering is what makes the exact dense operator competitive, and a FLASH-style streaming kernel doubles the gain of a WMMA filter; BF16 storage with double buffering and tensor-core refreshes speed up the certified operator most at large steps, where refreshes dominate. With a single request and a GQR pool, there is too little work per call for tensor cores to pay off, and the FP32 document-major kernels are as fast or faster; this is why the executor is chosen per dataset (Section 6.1). Fusing the FP32 verification into the Triton filter kernel is also exact (64 configurations, no difference in lockstep) but runs at only 0.51–$0.70\times$ the speed of the separate verification in every setting of Tables 7 and 8; we attribute this to the sequential 128-term FMA chain of each token, whose operands the filter program loads with strided accesses, and keep the verification in a separate CUDA kernel.

Table 4: Kernel design ablation: speedup over FLASH-MaxSim with IEEE FP32 products for whole TTO runs ($T = 25$) under CUDA graphs. Every row is exact and passed the lockstep check.

	$B = 16$, 100 docs		$B = 1$, GQR pools	
Operator, η	10^{-4}	10^{-3}	10^{-4}	10^{-3}
Dense				
FP32 document-major	1.30	1.30	2.12	2.12
+ tensor-core filter, WMMA	2.85	2.84	1.52	1.52
+ FLASH-style Triton filter	5.28	5.29	1.95	1.96
Certified				
FP32 document-major	4.74	1.48	2.20	1.54
+ BF16 storage, double-buffered FP32 refresh	5.99	2.47	2.24	1.60
+ tensor-core refresh	6.90	3.30	2.20	1.60

Refresh ablation. Table 5 isolates the two refresh optimizations—128-bit shared-memory loads and the split over blocks—by comparing the kernels above with an otherwise identical variant that refreshes each page in a single block with scalar loads. The two variants compute every product with the same routine and are bitwise identical to each other and to the dense twin in all calls; the optimized refresh is $1.53\times$ faster for the certified kernel and $1.35\times$ for the dense twin (geometric means over 36 configurations).

Table 5: Refresh ablation under whole-loop CUDA graphs on the same batches (padded queries of the released pipeline; geometric means): the split refresh with 128-bit shared-memory loads used throughout the paper against a single-block refresh with scalar loads. Both are bitwise identical to each other and to the dense twin in every call (36 configurations, 10.00 million token–page calls).

			Split over single-block		Certified vs. FLASH	
Batches	n	Refresh (%)	Dense	Certified	Single-block	Split
100-page pools	4	54.0	1.38	1.25	1.62	2.02
10-page pools	4	47.7	1.34	1.63	0.87	1.42
Table 6 batches, $\eta \leq 10^{-4}$	21	58.6	1.36	1.65	0.72	1.20
Table 6 batches, $\eta > 10^{-4}$	7	97.8	1.29	1.30	0.61	0.79

Table 6: Where the time goes, summed over 28 batches with 10-page pools from the released GQR pipeline (queries keep their padding tokens; step sizes up to 10^{-3}; A100, milliseconds). Eager: PyTorch execution with CUDA events at phase boundaries (median of five runs per batch); forward, backward and optimizer columns include host launch gaps. Graph: the whole run (initial call and 25 steps) captured in one CUDA graph and replayed. Ratios are relative to query-only FLASH-MaxSim in the same mode.

	Eager					CUDA graph	
Scoring operator	Fwd	Bwd	Adam	Total	Ratio	Total	Ratio
GQR scorer (Uzan et al., 2026)	158	1053	75	1317	1.85	1208	4.55
FLASH-MaxSim (Pony et al., 2026)	343	220	168	766	1.08	311	1.17
FLASH-MaxSim, query-only	340	167	167	711	1.00	265	1.00
Pair-parallel dense	482	61	117	702	0.99	521	1.97
Pair-parallel certified	321	117	150	636	0.90	334	1.26
Document-major dense twin (ours)	220	153	162	567	0.80	233	0.88
Document-major certified (ours)	238	157	155	602	0.85	249	0.94

Cost model. Figure 3a plots the whole-loop time with the FP32 certified operator relative to that with its dense twin against the document refresh rate for every 100-document configuration of the controlled workloads of Appendix F; the least-squares line is the cost model of Section 6.4. Larger pools amortize the fixed cost of certification (Fig. 3b), and more, smaller steps at a fixed ηT raise the certified fraction, as Theorem 3 suggests (Fig. 3c).

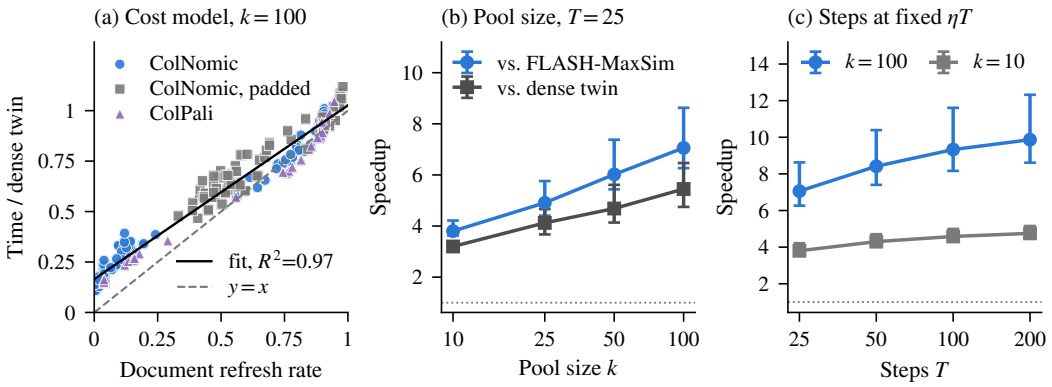

Figure 3: Controlled workloads (primary-retriever pools, FP32 document-major kernels, whole-loop CUDA graphs). (a) Whole-loop time with the certified operator relative to the dense twin against the document refresh rate, for all 100-document configurations, with the least-squares line. (b) Speedup over FLASH-MaxSim in FP32 and over the dense twin as the pool grows, on four ColNomic batches from TAT-DQA and Biomedical-v2 ($\eta = 10^{-5}$, $T = 25$; geometric mean, bars show the range). (c) Speedup over FLASH-MaxSim for runs with more, smaller steps (ηT fixed).

Pair-parallel versus fused execution. Table 6 compares execution schemes on 28 batches of GQR's padded ColNomic queries with 10-page pools, $T = 25$ and step sizes up to 10^{-3}, where tokens fail often. Under whole-loop CUDA graphs, the pair-parallel certified kernel, which reads a document once for every failing token, takes $1.26\times$ the time of FLASH-MaxSim in FP32, and document-major execution removes most of this penalty.

Step size and workload. Table 7 separates step size, pool size and batch size on fixed batches outside the development-set selection, in the autograd loop with $T = 25$. With 16 queries and 100-document pools, GQR's scorer, which materializes every product before taking the maxima, runs at only $0.16\times$ the speed of FLASH-MaxSim in FP32; the exact dense operator on tensor cores runs $5.30\times$ faster than FLASH-MaxSim in FP32 and close to its TF32 path ($6.01\times$), and certification adds the savings of skipped scans: at $\eta = 10^{-5}$ the certified operator refreshes 3.8% of the docu-

Table 7: Speedup over FLASH-MaxSim with IEEE FP32 products for whole TTO runs ($T = 25$) under CUDA graphs, by step size and workload (geometric means over batches or queries). The document refresh rate is that of the certified operator. †Not exact.

	$B = 16$, 100-document pools				$B = 1$, GQR pools, clean queries			
η	10^{-5}	10^{-4}	$5{\times}10^{-4}$	10^{-3}	10^{-5}	10^{-4}	$5{\times}10^{-4}$	10^{-3}
Document refresh rate (%)	3.8	13.1	68.4	84.5	3.8	11.9	71.7	88.0
GQR scorer (FP32)	0.16	0.16	0.16	0.16	2.49	2.49	2.49	2.49
FLASH TF32†	6.01	6.02	6.02	6.02	3.16	3.16	3.16	3.16
Dense twin (FP32)	1.30	1.30	1.30	1.30	2.13	2.12	2.12	2.12
Certified (FP32)	7.75	4.74	1.87	1.48	2.98	2.20	1.72	1.54
Exact dense (tensor cores)	5.30	5.28	5.28	5.29	1.99	1.95	1.96	1.96
Certified (tensor cores)	9.19	6.90	3.99	3.30	2.83	2.20	1.78	1.60

ments per call and runs $9.19\times$ faster, and at 10^{-4} it still leads ($6.90\times$). As the step grows, token amplification takes over, and at $\eta = 10^{-3}$ the exact dense operator is the better choice. Table 8 complements Table 7 with 10-document pools and with the official, zero-padded ColNomic queries.

Table 8: As Table 7, for 10-document pools ($B = 16$) and GQR pools with the official, zero-padded ColNomic queries ($B = 1$).

	$B = 16$, 10-document pools				$B = 1$, GQR pools, official queries			
η	10^{-5}	10^{-4}	$5{\times}10^{-4}$	10^{-3}	10^{-5}	10^{-4}	$5{\times}10^{-4}$	10^{-3}
Document refresh rate (%)	3.8	12.1	61.7	77.4	80.4	80.6	97.6	99.6
GQR scorer (FP32)	0.24	0.24	0.24	0.24	2.70	2.71	2.71	2.71
FLASH TF32†	4.00	4.01	4.01	4.01	3.69	3.69	3.69	3.69
Dense twin (FP32)	1.21	1.21	1.21	1.21	1.97	1.97	1.96	1.96
Certified (FP32)	3.97	2.35	1.39	1.19	1.28	1.28	1.20	1.12
Exact dense (tensor cores)	2.52	2.51	2.52	2.51	2.40	2.37	2.36	2.36
Certified (tensor cores)	4.14	2.77	1.96	1.67	1.36	1.36	1.26	1.18

E EXPERIMENTAL DETAILS

Data and embeddings. On ViDoRe we use the query and page embeddings of ColNomic-7B and ColPali-v1.3 produced by GQR's released embedding pipeline (Table 9); ColPali embeddings exist for twelve of the fourteen splits. ViDoRe v2 ESG-human has graded relevance labels, the other v2 splits a single grade and v1 binary labels; nDCG uses linear gains, as in `pytrec_eval`. The complementary retrievers read the text of each page, which we extract with RapidOCR; GQR used Docling, and page text is the one known difference between our pipeline and the released one. On BEIR we use the test splits of SciFact, NFCorpus, FiQA and SciDocs; ArguAna is left out because its queries are corpus documents that the standard evaluation must exclude from their own rankings. ColBERTv2 runs the released checkpoint with its query and document encoders re-implemented in `transformers`: queries carry the `[Q]` marker and are augmented with `[MASK]` tokens to 32 positions, documents carry the `[D]` marker, are truncated to 180 tokens and lose punctuation tokens, and all vectors are 128-dimensional and L2-normalized, then stored in BF16. Documents shorter than the longest are padded with copies of their own vector nearest to their centroid. A duplicate never exceeds the maximum, and ties go to the smaller, real index, so every maximum and every selected vector equals that of the unpadded document (we verified this bitwise against masked scoring); Qwen3-Embedding-4B encodes BEIR with a maximum length of 512 tokens.

Protocol. Pools are the union of the two retrievers' full-corpus top-10, with duplicates removed in index order as in the released code; targets use $\alpha = \frac{1}{2}$ and $\tau = 1$, the KL divergence uses the released stabilizer 10^{-8}, and Adam uses its defaults. Every configuration records nDCG@5 and nDCG@10 of the re-ranked pool every five steps up to $T = 100$. The development split is the first 10% of a

Table 9: ViDoRe splits. Vectors: patch vectors per page (padded); padding: share of ColNomic query tokens that are zero vectors from batch padding (Appendix F); batches: number of batches of 16 queries with equal token count, for ColNomic queries without and with padding tokens (in parentheses: the largest batch used where there is none of 16).

			ColNomic-7B				ColPali-v1.3	
				Padding	Batches			
Split	Queries	Pages	Vectors	(%)	clean	padded	Vectors	Batches
ArxivQA	500	500	779	29.0	17	22	1030	18
DocVQA	451	500	779	24.4	21	22	1030	22
InfoVQA	494	500	779	28.3	20	23	1030	21
TabFQuAD	280	70	771	22.8	5	10	1030	3
TAT-DQA	1646	277	755	24.0	85	95	–	–
ShiftProject	100	1000	747	21.4	0 (11)	2	1030	0
AI	100	968	779	18.9	0 (11)	4	1030	0
Energy	100	977	770	28.3	0 (12)	2	1030	0
Government	100	972	770	20.7	1	3	1030	2
Healthcare	100	965	779	22.0	0 (10)	4	1030	0
Biomedical (v2)	640	1016	779	17.6	25	30	1030	26
Economics (v2)	232	452	770	19.4	2	10	1030	4
ESG (v2)	228	1538	767	19.4	0 (13)	10	1030	2
ESG-human (v2)	52	1538	767	26.3	0 (8)	1	–	–

random permutation with seed 42, and (η, T) is selected in the released order (step sizes ascending, then step counts ascending, replacing the incumbent only on strict improvement), which keeps $(\eta, T) = (10^{-5}, 5)$ when no configuration improves on the development split. Queries of a dataset are optimized in batches of up to 64 with token and pool masks, which is mathematically equivalent to per-query optimization because Adam acts coordinatewise and the losses are summed; on a test set of queries our batched implementation matches the released `optimize_queries_union` to within 2×10^{-6} in every score. Table 11 lists the selected operating points.

Table 10: GQR under its official protocol: test nDCG@5 (%), macro-averaged over datasets (number in parentheses). ColNomic* uses the official, zero-padded queries; Guide is the complementary retriever. Primary and Guide score the whole corpus alone, Fusion ranks the pool by the target of Eq. (2), and GQR runs at the operating point selected on the development split. Δ: GQR minus Primary with a 95% stratified bootstrap interval; ↑/↓: datasets with a significant paired change ($p < 0.05$, randomization test). GQR's reported numbers use Docling text (Uzan et al., 2026).

	Retrievers		Without TTO						
Benchmark	Primary	Guide	Primary	Guide	Fusion	GQR	Δ	95% CI	↑/↓
ViDoRe v1 (10)	ColNomic	Qwen3	89.4	74.1	84.9	89.3	0.0	[−0.2, +0.1]	1/0
	ColNomic	Linq	89.4	71.5	86.8	89.3	−0.1	[−0.2, 0.0]	0/2
	ColNomic*	Qwen3	89.4	74.1	81.0	89.3	−0.1	[−0.2, 0.0]	0/0
	ColNomic*	Linq	89.4	71.5	83.5	89.3	−0.1	[−0.1, 0.0]	0/0
	ColPali (9)	Qwen3	86.4	77.7	85.6	87.8	+1.4	[+0.8, +1.9]	3/0
	ColPali (9)	Linq	86.4	75.3	86.0	87.3	+0.9	[+0.2, +1.6]	0/0
ViDoRe v2 (4)	ColNomic	Qwen3	60.4	50.3	58.4	61.4	+1.0	[−0.5, +2.6]	0/0
	ColNomic	Linq	60.4	58.8	65.1	65.3	+4.9	[+2.7, +7.0]	3/0
	ColNomic*	Qwen3	60.4	50.3	57.2	61.4	+0.9	[−0.1, +2.1]	1/0
	ColNomic*	Linq	60.4	58.8	64.6	65.8	+5.4	[+3.2, +7.6]	3/0
	ColPali (3)	Qwen3	54.4	48.6	55.1	55.2	+0.8	[−0.6, +2.1]	1/0
	ColPali (3)	Linq	54.4	56.2	60.0	59.9	+5.5	[+3.9, +7.1]	3/0
GQR paper	ColNomic	Qwen3	60.3			61.0	+0.7		
	ColNomic	Linq	60.3			62.8	+2.5		
BEIR (4)	ColBERTv2	Qwen3	36.2	49.2	45.5	46.4	+10.2	[+9.2, +11.2]	4/0

Table 11: Operating points $\eta^\star/T^\star$ selected on the development split for every primary retriever (top row), complementary retriever and dataset (ViDoRe v1, ViDoRe v2 and BEIR, separated by rules); ColNomic* uses the official, zero-padded queries.

	ColNomic		ColNomic*		ColPali		ColBERTv2
Dataset	Qwen3	Linq	Qwen3	Linq	Qwen3	Linq	Qwen3
ArxivQA	5×10^{-5}/65	5×10^{-5}/25	10^{-5}/70	10^{-5}/65	5×10^{-3}/5	5×10^{-4}/70	–
DocVQA	5×10^{-5}/45	5×10^{-5}/30	5×10^{-5}/25	10^{-5}/75	5×10^{-5}/70	5×10^{-5}/30	–
InfoVQA	5×10^{-5}/90	10^{-5}/5	5×10^{-5}/75	10^{-5}/5	5×10^{-4}/60	5×10^{-4}/75	–
TabFQuAD	5×10^{-4}/30	10^{-5}/5	10^{-4}/60	10^{-5}/5	5×10^{-3}/10	10^{-5}/75	–
TAT-DQA	10^{-4}/60	5×10^{-4}/15	5×10^{-5}/100	5×10^{-5}/75	–	–	–
ShiftProject	10^{-5}/5	10^{-5}/5	10^{-5}/5	10^{-5}/5	10^{-4}/100	5×10^{-3}/10	–
AI	10^{-5}/5	10^{-5}/5	10^{-5}/5	10^{-5}/5	5×10^{-4}/25	5×10^{-4}/25	–
Energy	10^{-5}/5	10^{-5}/5	10^{-5}/5	10^{-5}/5	10^{-5}/5	10^{-5}/5	–
Government	10^{-5}/5	10^{-5}/5	10^{-5}/5	10^{-5}/5	10^{-5}/5	10^{-5}/5	–
Healthcare	10^{-5}/5	10^{-5}/5	10^{-5}/5	10^{-5}/5	10^{-5}/5	5×10^{-4}/30	–
Biomedical	5×10^{-4}/45	10^{-3}/40	5×10^{-4}/40	5×10^{-4}/25	10^{-3}/10	10^{-3}/40	–
Economics	10^{-5}/55	5×10^{-4}/30	10^{-5}/45	5×10^{-4}/30	5×10^{-4}/30	5×10^{-3}/5	–
ESG	5×10^{-4}/35	5×10^{-3}/5	5×10^{-4}/35	5×10^{-3}/5	5×10^{-4}/60	10^{-3}/35	–
ESG-human	5×10^{-4}/40	5×10^{-3}/40	10^{-4}/55	5×10^{-4}/50	–	–	–
SciFact	–	–	–	–	–	–	5×10^{-3}/10
NFCorpus	–	–	–	–	–	–	5×10^{-3}/15
FiQA	–	–	–	–	–	–	5×10^{-3}/15
SciDocs	–	–	–	–	–	–	5×10^{-3}/10

Larger pools. GQR's official pools hold at most twenty documents. To test whether our conclusions depend on this choice, we rerun the grid with the union of the two retrievers' top-50 for ColNomic (clean queries) with both complementary retrievers and for ColBERTv2 with Qwen3, select (η, T) again on the same development split (Table 12), and time the selected operating points on six test queries per dataset (Table 14). With these pools and the autograd loop, our exact operator runs at $1.22\times$ the speed of GQR's FP32 scorer with ColNomic and $0.85\times$ with ColBERTv2, whose short passages and large selected steps leave little to certify, and whole-loop capture gains more over the released loop than with GQR's pools. Served in batches of 64 in the fused loop (first 64 test queries per dataset), our exact operator selected on development queries processes $4.84\times$ (ColNomic) and $3.12\times$ (ColBERTv2) as many queries per second as GQR's scorer with these pools, against $3.87\times$ and $2.87\times$ with GQR's pools; FLASH-MaxSim's TF32 path is $1.03\times$ and $1.21\times$ as fast as ours, and GQR's scorer changes the final bits of 0.7% of the queries (Table 16).

Table 12: Larger union pools. GQR's pool is the union of the two retrievers' top-K over the whole corpus; the official protocol uses $K = 10$. For $K = 50$ we select (η, T) again on the same development split. Pool size is the mean number of documents, pool recall the share of relevant documents in the pool (%), GQR the test nDCG@5 (%), macro-averaged over datasets, and Δ the paired difference of test nDCG@5 ($K = 50$ minus $K = 10$) with a 95% stratified bootstrap interval, colored when it excludes zero.

			Pool size		Pool recall		GQR			
Benchmark	Primary	Guide	K=10	K=50	K=10	K=50	K=10	K=50	Δ	95% CI
ViDoRe v1	ColNomic	Qwen3	15.5	75.8	96.0	98.1	89.3	89.3	0.0	[−0.2, +0.1]
ViDoRe v2	ColNomic	Qwen3	14.8	71.1	74.7	91.9	61.4	61.0	−0.4	[−1.6, +0.7]
ViDoRe v1	ColNomic	Linq	15.7	75.9	95.7	98.1	89.3	89.3	+0.1	[0.0, +0.1]
ViDoRe v2	ColNomic	Linq	14.9	70.5	77.9	93.4	65.3	65.1	−0.2	[−1.5, +1.1]
BEIR	ColBERTv2	Qwen3	16.2	82.1	53.0	66.4	46.4	44.7	−1.7	[−2.2, −1.3]

Timing. For every dataset we time its first two development queries, which select our exact operator, and its first twelve test queries, which are reported. The autograd loop is a functional TTO

Table 13: Median latency (ms) of one request ($B = 1$) at each dataset's selected operating point (guide: Qwen3; ColBERTv2 on BEIR, the others on ViDoRe) and speedups of ExactTTO (geometric means over test queries). GQR scorer: the released loop run eagerly, and its scorer in our whole-loop CUDA graph; FLASH-MaxSim: IEEE FP32 products or its tensor-core path (†TF32, not exact); ExactTTO: the exact operator selected on development queries. Last column: timed queries (%) whose final top-5 under TF32 differs from the exact one.

	GQR scorer		FLASH-MaxSim			ExactTTO: speedup over			TF32
Primary	eager	graph	FP32	TF32†	ExactTTO	eager	graph	FP32	top-5 ≠
ColNomic	50.2	3.70	8.37	2.80	4.04	13	0.90	2.27	0.6
ColNomic*	51.1	4.06	9.53	2.97	4.71	11	0.82	2.04	1.8
ColPali	24.7	1.81	5.55	1.49	2.25	12	0.85	2.54	0.7
ColBERTv2	19.2	1.00	1.28	0.75	1.15	16	0.88	1.12	0.0

Table 14: As Table 13, with union pools of the two retrievers' top-50 (Table 12), operating points re-selected on the development split and six test queries per dataset.

	GQR scorer		FLASH-MaxSim			ExactTTO: speedup over			TF32
Primary	eager	graph	FP32	TF32†	ExactTTO	eager	graph	FP32	top-5 ≠
ColNomic	128.4	7.32	10.4	3.49	6.04	24	1.22	1.85	0.0
ColBERTv2	89.0	2.45	2.61	1.61	3.01	30	0.85	0.88	0.0

loop whose Adam update uses the formulas of `torch.optim.Adam`, and all its methods are captured in the same way as the fused loop; we run it twice outside capture, capture one run, replay it twice, and report the median of five timed replays. The released loop runs eagerly with `torch.optim.Adam` and one optimizer per query (the release shares one optimizer across all queries, whose per-step cost grows with their number); we report the median wall-clock time of three runs after one warm-up. All methods of a query run in the same process on the same inputs. Replays include every cost of the captured loop, including the reset of the certified operator's state. Building the operator, two warm-up runs and the capture take 161 ms for the certified tensor-core operator (median), the time of 32 replays. A graph could serve later requests of the same shape (N, C) once their inputs, pool-dependent bounds and certificate state are copied into its buffers; we do not evaluate such reuse.

Table 13 reports the same single requests with every scorer in the autograd loop (measured in an earlier run with the same operating points and queries).

Serving. Each dataset serves its first 64 test queries (all of them for smaller test sets) at its operating point. Queries are sorted by token count and cut into batches of 16 or 64, so a batch of 64 holds all served queries of a dataset, whatever their length; a batch pads its pools to the largest one by repeating a pool's first document and masks the padding, and pads short queries with copies of their first token, which the fused loop neither sums nor updates and FLASH-MaxSim masks through its query lengths (zero vectors would tie with every document vector and send the tensor-core filter to its FP32 fallback). Every method runs the same batches. These masks exclude only padding added for serving; the zero tokens of official ColNomic queries remain part of the optimized query. Each batch runs twice eagerly, which checks run-to-run determinism, and once captured (two warm-ups, capture, two replays, five timed replays); throughput is the number of queries divided by the sum of the batch medians. Invariance compares every query's final pool scores in a batch with those of its single request, and the exactness check compares the final scores of all our exact operators. On the timed queries, GQR's scorer reaches the same final top-5 in the fused and autograd loops for 524 of 528 queries, and repeated autograd-loop runs returned the same bits for 528, 528 and 528 of them with GQR's scorer and FLASH-MaxSim in FP32 and TF32. A second run serves 64 test queries drawn at random from each test set (seed 1; all of them for smaller test sets) with the same operating points and development queries: at $B = 64$, ExactTTO serves $3.04\times$ as many queries as GQR's scorer and $3.87\times$ as many as FLASH-MaxSim in FP32 (first 64: $3.05\times$ and $3.95\times$), TF32 is $1.53\times$ faster still, and GQR's scorer changes the final scores of 17.8% of the queries (none with another

Table 15: Serving TTO requests in batches at the selected operating points (44 dataset–retriever combinations, up to 64 test queries each, queries bucketed by length). Throughput relative to GQR's FP32 scorer in the same fused loop and batch size (geometric means over combinations); the ExactTTO row selected on dev uses, at every batch size, the tensor-core operator chosen on development queries (Table 2 chooses among all exact operators). Right: test queries (%) whose final scores in a batch of 64 have the same bits as when served alone, and whose top-5 differs. †Not exact.

	Throughput over GQR scorer			$B = 64$ vs. alone (%)	
Scorer in the fused loop	$B = 1$	$B = 16$	$B = 64$	same bits	top-5 $\neq$
GQR scorer (FP32)	1.00	1.00	1.00	80.4	0.0
FLASH-MaxSim FP32	0.17	0.81	0.77	100.0	0.0
FLASH-MaxSim TF32†	0.92	4.20	4.58	100.0	0.0
ExactTTO: exact dense	0.45	2.20	2.68	100.0	0.0
ExactTTO: certified	0.34	1.76	2.38	100.0	0.0
ExactTTO: selected on dev	0.49	2.51	3.05	as selected	

top-5), our exact operators of 0.0% and 0.0%. Per combination, the random sample's speedup over GQR's scorer is 0.72–1.31 times that of the first 64 (within 10% for 41 of 44), ExactTTO is faster than GQR's scorer on all 44 combinations (at least $1.77\times$), and choosing the operator in hindsight would add 2.3%; the two samples share 41.0% of their queries. Alone, the two development queries of a combination would choose different operators for 4 of 44 combinations; for all four of them the choice from both is the faster operator at $B = 64$, whereas the other query alone would lose up to $2.03\times$. Summing the single-request times of every pair of the first 5–16 development queries of each combination (3,935 pairs), the rule picks the faster operator at $B = 64$ for 82.1% of the pairs and is 3.4% slower than the faster one on average (geometric mean; at most $2.03\times$). Single requests predict the faster batch operator for 34 of 44 combinations; the largest losses arise on ColPali on DocVQA, where the certified operator takes $1.03\times$ the dense one's time alone but serves $2.03\times$ as many queries at $B = 64$, so single-request and batch rankings disagree. Timing all 5–16 development queries of a combination as one batch instead picks the faster operator at $B = 64$ for 38 of 44 combinations and loses 1.4% on average (at most 26.2%); the same queries timed alone pick it for only 34 (4.1%, at most $2.03\times$) and the first two alone for 35 (2.5%, at most 31.0%), so the gain comes from timing a batch, not from timing more queries. On the random test sample the three rules pick it for 39, 35 and 36 combinations. Batches of 64 hold queries of all lengths, and 19.5% of their query tokens are padding; random instead of length-grouped batches of 16 raise the padded share from 5.2% to 14.7%, and ExactTTO then serves $2.34\times$ as many queries as GQR's scorer ($2.51\times$ grouped), TF32 is $1.77\times$ faster ($1.67\times$), and GQR's scorer changes the final scores of 19.6% of the queries, our exact operators of 0.0% and 0.0%.

Table 16: Serving at $B = 64$ by primary retriever and pool size (K: top-K of each retriever in the union). GQR's scorer in the fused loop: queries per second (geometric mean over datasets) and test queries whose final scores have the same bits as when served alone; FLASH-MaxSim and ExactTTO: throughput relative to GQR's scorer (selected: the tensor-core exact operator chosen on development queries). †Not exact.

			GQR's scorer		FLASH-MaxSim		ExactTTO		
Primary	K	Datasets	q/s	same bits (%)	FP32	TF32†	dense	certified	selected
ColNomic	10	14	1348	67.3	0.77	4.72	3.05	3.80	3.87
ColNomic*	10	14	1277	100.0	0.76	4.60	2.19	1.55	2.19
ColPali	10	12	1201	99.6	0.73	4.74	2.84	3.20	3.46
ColBERTv2	10	4	8533	0.0	0.98	3.66	2.87	0.87	2.87
ColNomic	50	14	262	99.1	0.83	4.96	3.60	4.39	4.84
ColBERTv2	50	4	978	100.0	0.94	3.79	3.12	1.08	3.12

Reranker distillation. Pools are ColBERTv2's full-corpus top-100 under GQR's scoring. The reranker bge-reranker-v2-m3 (Chen et al., 2024) scores every query–passage pair in FP16 with inputs truncated to 512 tokens, and the target is the softmax of its logits ($\tau = 1$, $\alpha = 1$). (η, T) is

selected from $\{1, 2, 5, 10, 20, 50, 100, 200, 500, 1000\} \times 10^{-3} \times \{5, 10, 15, 20, 25, 30, 40, 50\}$ and $T = 0$ on the development split (seed 42) by nDCG@10 of retrieval from the whole corpus with the refined query, ties going to fewer steps; FiQA selects the largest step count; the test split runs the selected point. Quality runs use our exact dense operator in batches of 64, which by Proposition 2 gives every query the result it gets alone. Timing and bitwise checks at the selected point use the first 64 test queries at $B \in \{16, 64\}$ and the first twelve at $B = 1$; the first two development queries at $B = 1$ select our exact operator, as in Section 6.1.

Lockstep and FP64 checks. Along the dense twin's trajectory, at each of the $T+1$ scoring calls, every exact operator is called on the same query with its own state; its values and selected vectors must equal the twin's bitwise, and the FP64 product of each selected vector must be within 10^{-6} of the FP64 maximum over the document. Queries longer than 64 tokens use the FP32 tensor-core-free path of our kernels as the reference.

Test-set audit. At each dataset's operating point, we run TTO on its first 96 test queries in index order (all of them for smaller test sets; 7,688 of 26,125 test queries overall) with the autograd loop, in batches of up to 64 queries of equal length, once with each scorer: the dense twin, the tensor-core exact dense and certified operators, GQR's einsum scorer, and FLASH-MaxSim in FP32 and TF32. We compare the final pool scores with the twin's bitwise and the rankings they induce, and we audit the premise of Theorem 2 by checking that the largest token norm along every trajectory stays below Q; at least 81% of the additive margin $3.2\,\eta\sqrt{h}\,T$ remained unused. We also rerun the released loop of the grid (`torch.optim.Adam`, the grid's mixed batches of all queries) up to the selected step count; on the audited queries its nDCG@5 equals the stored grid value for 100.0% of them, and its final top-5 equals the twin's for every audited query. Differences in average nDCG@5 are tested with paired permutation tests. The queries served in Section 6.3 are the first 64 of these, and Section 6.4 compares their nDCG@5 in the fused loop with the released loop's.

Certificate parameters. Radii, the BF16 copies and the norm bounds V_d depend only on the document and would be computed once per corpus; our experiments compute them for the documents of each pool before timing (298 µs per page and 231 µs per passage on the A100, including the guard). Radii are computed in FP64 by the Gram identity and inflated as described in Appendix B. V is the largest vector norm in the query's pool and Q the largest initial token norm plus $3.2\,\eta\sqrt{h}\,T$, which dominates the Adam bounds of Lemmas 1 and 2 for all $T \le 200$ ($\rho_{200} = 3.096$); the guard ε is rounded up to FP32.

Hardware and software. One NVIDIA A100-SXM4-80GB, PyTorch 2.8.0 with CUDA 12.6 and Triton 3.4.0; PyTorch's TF32 is disabled and matmul precision is "highest", so only FLASH-MaxSim's released tensor-core path uses TF32. Our CUDA kernels are compiled with `-O3` for `sm_80` and without fast-math.

F Padding tokens in the released pipeline

Setting of this and the next two appendices. These analyses study the certificates themselves on controlled workloads: pools of the primary retriever's top-k pages ($k = 10$ or 100), a fixed $T = 25$ and step sizes $\eta \in \{10^{-5}, 10^{-4}, 5\times10^{-4}, 10^{-3}\}$, with float64 replays where stated. They are independent of GQR's union pools and development-set selection, and their conclusions concern the certificates, not GQR's quality.

GQR's released code encodes the queries of a split in batches and stores every output row of the retriever, including the positions that pad a batch to its longest query, and pads the stored queries once more to a common length for scoring. ColNomic's encoder returns zero vectors at the padding positions, so these positions become query tokens that are exactly zero; our embeddings follow the same procedure with batches of eight queries. Over the fourteen splits, 23.4% of ColNomic's query tokens are such zeros, and 85.9% of the queries contain at least one (Table 9); ColPali's stored queries contain none. A zero token contributes $\max_j \langle 0, v_{dj} \rangle = 0$ to every page, so initial scores and rankings are those of the query without it, up to the factor $1/N$, which is larger without padding and sharpens the softmax target of GQR. Its gradient is a weighted sum of the selected patches of all pages, each the first one in index order since all its products tie, and Adam moves it by at most

$\eta\sqrt{h}$ in the first step. From then on it behaves like a free token that the optimization steers, so TTO on a padded query optimizes a different objective than on the query without padding. Such tokens resemble the query augmentation of ColBERT (Khattab & Zaharia, 2020), except that ColBERT's `[MASK]` tokens receive contextual embeddings from the encoder, whereas these start at zero and are shaped only by TTO.

The first-round nDCG@5 of every ColNomic query is the same with and without its padding tokens, as it must be, and the effect of TTO barely changes: under GQR's official protocol, with each format's own selected operating point, the test nDCG@5 of the clean queries differs from that of the official ones by $+0.05$ points with Qwen3 and -0.20 with Linq, paired over $4{,}523$ test queries (Section 6.2).

Table 17 splits the certificate failures of the float64 replay by token type. Padding tokens fail on most calls at every step size, while real tokens fail rarely at small steps; since a page is read whenever any of its tokens fails, the padding tokens alone set the page refresh rate of padded queries, and removing them lowers it to that of the real tokens. Table 18 repeats the certificate hierarchy of Table 19 on the padded queries.

Table 17: Padding tokens and token amplification (float64 replay, protocol of Table 19). Padded: the released pipeline's queries; failure rates of the gap–radius certificate on padding and on real tokens, the page refresh rate, the rate a page would be refreshed if only real tokens counted, and the oracle's rate. Clean: the same queries without padding tokens, re-optimized.

	Padded queries (%)					Clean queries (%)	
η	Pad fail	Real fail	Refresh	Real-only	Oracle	Refresh	Oracle
10^{-5}	77.1	3.8	77.6	3.9	15.1	3.9	3.8
10^{-4}	77.0	4.3	80.8	13.9	15.0	13.9	3.8
$5{\times}10^{-4}$	76.1	10.5	95.6	72.9	15.3	72.4	4.1
10^{-3}	73.4	16.0	97.2	86.0	16.4	85.3	5.2

Table 18: Certificate hierarchy for the queries of the released GQR pipeline, which keep their padding tokens (same protocol and queries as Table 19; reuse and page refresh rate in %).

	$\eta = 10^{-5}$		10^{-4}		$5{\times}10^{-4}$		10^{-3}	
Certificate	Reuse	Refresh	Reuse	Refresh	Reuse	Refresh	Reuse	Refresh
Winner only	72.5	95.3	59.4	99.4	37.8	99.5	29.4	99.5
Gap–radius (ours)	77.0	77.6	76.7	80.8	72.0	95.6	68.4	97.2
Optimal ball (Proposition 3)	78.5	71.8	78.5	72.5	76.1	90.1	73.9	94.0
Best anchor in working set	78.6	71.5	78.6	72.0	76.4	89.5	74.3	93.7
Clustered directions (32)	79.9	66.5	79.9	67.1	77.7	85.6	75.9	91.4
Low-rank directions (16)	78.9	70.5	78.9	70.9	76.3	89.6	74.0	94.2
Oracle	93.3	15.1	93.3	15.0	93.2	15.3	93.3	16.4

G CERTIFICATE HIERARCHY: PROTOCOL AND FULL RESULTS

Protocol. For each of TAT-DQA, Biomedical-v2 and Energy we draw 32 queries uniformly at random (fixed seed), remove their padding tokens (Appendix F), build top-10 pools by initial MaxSim, and run exact GQR in float64 (Adam, 25 steps, $\alpha = 0.5$, $\tau = 1$) at each step size. Every certificate is replayed along the same trajectory with its own per-pair state; a pair that fails is refreshed exactly, and every certified call is checked against the full float64 maximum, so all certificates below are exact by construction and differ only in how often they certify. The first call of each run is a full scan for every certificate. *Reuse* is the fraction of token–page calls that are certified; *Refresh* is the fraction of page–calls with at least one refreshed token. Table 19 averages over the three splits and Table 20 reports each split at $\eta = 5{\times}10^{-4}$.

Certificates. *Winner only*: the working set is the maximizer alone, with the gap to the runner-up and the page diameter as radius. *Gap–radius*: Proposition 1 with $r{+}1 = 9$. *Optimal ball*:

Table 19: How static is TTO? Float64 replay of exact GQR trajectories (32 random queries per split on TAT-DQA, Biomedical-v2 and Energy, top-10 pools, 25 Adam steps). For each certificate: certified token–page interactions (reuse, %) and page refresh rate (refresh, %), averaged over the three splits. All certificates are exact and each keeps its own references along the same trajectory; the oracle knows the true maximizer. Queries are used without padding tokens; the last two rows replay the same queries with the padding tokens of the released pipeline (Table 18).

	$\eta = 10^{-5}$		10^{-4}		$5{\times}10^{-4}$		10^{-3}	
Certificate	Reuse	Refresh	Reuse	Refresh	Reuse	Refresh	Reuse	Refresh
Winner only	93.3	56.7	76.5	94.3	49.0	96.1	38.4	96.1
Gap–radius (ours)	96.2	3.9	95.7	13.9	89.6	72.4	84.5	85.3
Optimal ball (Proposition 3)	96.2	3.8	96.1	5.6	92.9	55.3	89.5	74.6
Best anchor in working set	96.2	3.8	96.1	5.2	93.1	53.5	89.8	73.6
Clustered directions (32)	96.2	3.8	96.1	5.5	93.1	52.1	89.9	72.0
Low-rank directions (16)	96.2	3.8	96.1	5.3	92.7	56.5	89.1	75.8
Oracle	96.2	3.8	96.2	3.8	96.1	4.1	96.1	5.2
Gap–radius (ours), padded queries	77.0	77.6	76.7	80.8	72.0	95.6	68.4	97.2
Oracle, padded queries	93.3	15.1	93.3	15.0	93.2	15.3	93.3	16.4

Proposition 3. *Best anchor in working set*: a tail patch is excluded if some working-set patch a beats it with margin $\langle q_i^{\mathrm{ref}}, v_{da} - v_{dj}\rangle > \|\Delta q_i\|_2 \|v_{da} - v_{dj}\|_2$. *Clustered directions*: after a failed gap–radius test, each page's patches are grouped into 32 k-means clusters with centers μ_c and radii r_c, and cluster c is excluded if $G_c + \langle \Delta q_i, v_{dj^\star} - \mu_c\rangle - \|\Delta q_i\|_2 r_c > 0$, with G_c the smallest reference gap in the cluster. *Low-rank directions*: the tail is bounded through a 16-dimensional principal subspace of the page and the residual norms. *Oracle*: a call is certified whenever the true maximizers lie in the working set of the last refresh.

Per-patch exclusion. Applying Proposition 3 to each tail patch separately excludes patch j whenever $g_j > \|\Delta q_i\|_2 \delta_j$, with g_j its reference gap to $j^\star$ and $\delta_j = \|v_{dj^\star} - v_{dj}\|_2$. A token that fails the gap–radius test then rescans only H_{di} and the patches it cannot exclude, without moving its reference, or the whole page if more than half of the page remains. Counting a page read as the union of the rescanned rows plus the stored gaps and distance rows ($2S/h$ rows per failing token), the streamed data at $\eta = 10^{-4}$, $5{\times}10^{-4}$ and 10^{-3} is 5.2%, 36.2% and 69.3% of a dense run, against 13.9%, 72.4% and 85.3% for the gap–radius test and 3.8%, 4.1% and 5.2% for the oracle.

Table 20: Certificate hierarchy per split at $\eta = 5{\times}10^{-4}$ (queries without padding tokens; float64 replay; reuse and page refresh rate in %).

	TAT-DQA		Biomedical-v2		Energy	
Certificate	Reuse	Refresh	Reuse	Refresh	Reuse	Refresh
Winner only	53.1	96.1	42.2	96.2	51.8	96.1
Gap–radius (ours)	91.1	65.5	87.6	82.4	90.3	69.1
Optimal ball (Proposition 3)	93.4	48.2	92.4	64.3	92.8	53.4
Best anchor in working set	93.5	46.8	92.7	61.7	93.0	51.9
Clustered directions (32)	93.8	43.4	92.1	65.7	93.4	47.2
Low-rank directions (16)	93.0	52.1	92.3	63.5	92.7	54.0
Oracle	96.2	3.9	96.1	4.3	96.1	4.1

H THE PRICE OF EXACTNESS AND THE WORKING-SET SIZE

Non-exact reuse. To measure what exactness costs, we disable the certificate after the first call: the certified kernel then evaluates every pair on the working set of its first full scan for the rest of the run and never refreshes. This is the cheapest reuse our kernels can perform, but it is no longer exact. Table 21 compares it with the exact certified kernel and the dense twin on one batch per split (16

clean ColNomic queries of equal length, or the largest such batch) and compares its rankings after TTO with the exact ones.

Table 21: Non-exact working-set reuse: the certified kernel with the certificate disabled, so that after the first full scan no pair is ever refreshed (ColNomic queries without padding tokens, one batch per split, whole-loop CUDA graphs). Time ratios of the exact certified kernel and of the dense twin to this non-exact variant (geometric means over splits); splits whose final queries are bitwise identical to the exact run; queries whose top-5 ranking differs from the exact run; nDCG@5 after exact and non-exact TTO.

		Time over non-exact				nDCG@5	
k	η	Certified	Dense	Identical	Top-5 changed (%)	Exact	Non-exact
100	10^{-5}	1.00	5.51	13/13	0.0	71.3	71.3
100	10^{-4}	1.59	5.51	13/13	0.0	71.1	71.1
100	5×10^{-4}	3.90	5.51	0/13	0.6	70.4	70.4
100	10^{-3}	4.91	5.51	0/13	2.3	69.4	69.4
10	10^{-5}	1.00	3.05	14/14	0.0	74.0	74.0
10	10^{-4}	1.60	3.05	14/14	0.0	73.8	73.8
10	5×10^{-4}	2.64	3.05	0/14	0.0	71.3	71.3
10	10^{-3}	3.11	3.05	0/14	0.0	69.3	69.3

Working-set size. Table 22 varies the working set from the maximizer alone ($r{+}1 = 1$) to 15 patches. Larger working sets certify more interactions at the price of more products per certified pair; every size is exact and passed the lockstep check.

Table 22: Working-set size: the exact certified kernel with $r{+}1$ patches per working set (ColNomic queries without padding tokens, 100-page pools, one batch per split); reuse and page refresh rate in %, time relative to the dense twin under whole-loop CUDA graphs (geometric mean over splits). Every configuration passed the lockstep check.

	$\eta = 10^{-4}$			$\eta = 5\times10^{-4}$		
$r{+}1$	Reuse	Refresh	Time	Reuse	Refresh	Time
1	81.2	90.6	0.72	55.7	96.2	0.90
3	93.4	44.5	0.40	79.5	89.7	0.77
5	95.0	25.3	0.32	85.1	81.9	0.72
9	95.8	11.3	0.29	88.9	70.6	0.71
15	96.1	5.7	0.33	91.0	60.0	0.78