ONE VOCABULARY PASS, MANY REQUESTS: A STREAM-ONCE OUTPUT-HEAD KERNEL FOR CONCURRENT LLM DECODING

Anonymous authors
Paper under double-blind review

ABSTRACT

In greedy decoding, the output head of a language model must find, for each hidden state, the vocabulary row with the largest inner product. For small models with large vocabularies this step is costly: in Qwen2.5-0.5B, the output head holds 28% of the weights used for every generated token. When a server decodes many requests at once, existing exact methods either compute every score at full precision or run a separate search for each request. We present FUSEHEAD, a CPU kernel that rules out almost every row with cheap 8-bit upper bounds and computes exact FP32 scores only for the few rows that could win. One pass over an 8-bit copy of the vocabulary serves the whole batch, and because the final choice uses the original FP32 scores, the chosen tokens are the same as those of a full FP32 head. On a 24-request WikiText-2 trace with a model based on Qwen2.5-0.5B, running on eight CPU cores, FUSEHEAD keeps a median of two rows per request, returns the full FP32 result for all 288 decisions, and is **2.03**× [1.18, 4.10] faster than an exact oracle that picks the faster of batched matrix multiplication and MAXIMUS for each batch. It is **6.56**× [6.06, 6.99] faster than the exact batched search LEMP-LI. On held-out hidden states, the speedup over the exact oracle is **1.95**× [1.13, 3.46] and **1.41**× [1.02, 2.05] for two ways of grouping requests into batches, again with every output equal to the full FP32 result.

1 INTRODUCTION

At every decoding step, a language model turns its last hidden vector into one score per vocabulary entry. With an output matrix $W \in \mathbb{R}^{V \times d}$ that has no bias term and a hidden state $h \in \mathbb{R}^d$, greedy decoding returns

$$\arg\max_{i \in \{0,\dots,V-1\}} w_i^\top h. \tag{1}$$

This is a maximum inner product search (MIPS). For the Qwen2.5-0.5B model used in this paper, $d = 896$ and $V = 151{,}936$ (Yang et al., 2025; Qwen Team, 2024), so the FP32 head takes 544.5 MB, even though each step needs only one token ID. For small models like this one, the head is a large part of the model: it holds 136.1 million of the 494.0 million weights that are used for every generated token (27.6%), and every decoding step must search it. This paper makes that search cheaper on CPUs.

Serving systems decode many requests together. Schedulers such as Orca move all active requests forward by one token at the same time (Yu et al., 2022), so the output head sees a *batch* of hidden states at every step. Only one row wins for each request, yet existing exact methods do far more work than that. Dense matrix multiplication computes all $B \times V$ scores at full FP32 precision. Exact MIPS indexes such as LEMP (Teflioudi & Gemulla, 2017), FEXIPRO (Li et al., 2017), and MAXIMUS (Abuzaid et al., 2019) skip rows using upper bounds, but each query still runs its own search. Approximate indexes such as HNSW are faster (Loretz & Hochreiter, 2026), but they can miss the true maximum and change the output; in our trace, a strong HNSW setting picks the wrong token in two of 288 decisions.

Our idea is to make the exact work tiny. Since only one row per request wins, almost every row can be ruled out by a cheap bound. FUSEHEAD (Figure 1) stores each FP32 row as 8-bit codes plus

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.

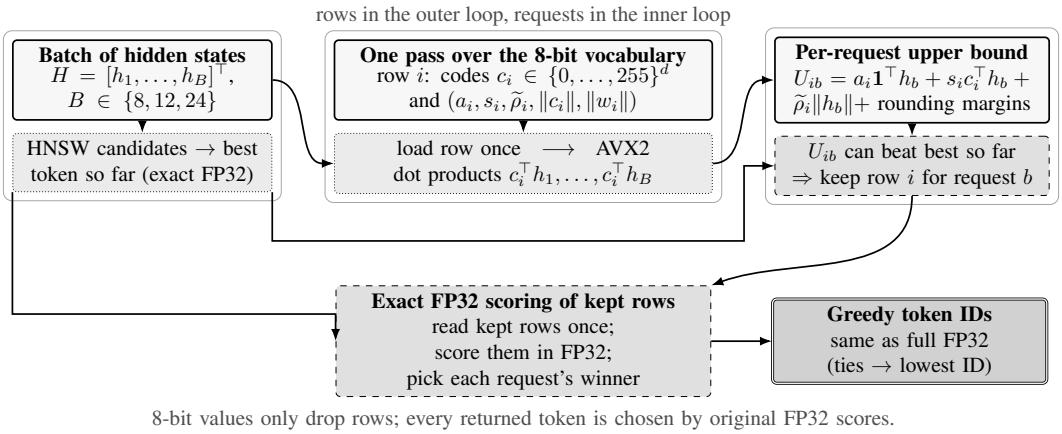


Figure 1: **Overview of FUSEHEAD.** Instead of searching the vocabulary once per request, FUSEHEAD reads each 8-bit row once for the whole batch, computes an upper bound on the row's score for every request, and keeps a row only if it could beat that request's best token so far. The kept rows of all requests are read once and scored in the original FP32 format. On the main trace, each batch reads a 142.2 MB compact copy of the vocabulary and scores a median of 101 rows exactly (0.066% of the vocabulary).

a few numbers per row, and for each row and each request computes an upper bound on the row's true score. A row is kept for a request only if this bound could beat the best token found so far; in our trace this leaves a median of two rows per request. Because all requests in a batch use the same 8-bit rows, the kernel reads each row once and updates the bounds of all requests while the row is in cache. The kept rows from all requests are then read once from the original FP32 head and scored exactly. The 8-bit values are used only to drop rows; every returned token is chosen by FP32 scores computed in the same way as the full FP32 reference.

We test FUSEHEAD on a 24-request trace, which we call the main trace, with batches of 8, 12, and 24 requests (288 decisions in total) on eight CPU cores. It returns exactly the full FP32 token in all 288 decisions and needs 0.591 s of median head time. Our strongest exact baseline, the *exact oracle*, picks the faster of batched matrix multiplication and MAXIMUS separately for each batch and needs 2.542 s; the paired speedup is 2.03× [1.18, 4.10]. FUSEHEAD is also 6.56× [6.06, 6.99] faster than the official batched LEMP-LI and 2.94× [1.47, 5.56] faster than the fastest baseline that returns the same tokens. The speedup over the exact oracle also holds on held-out hidden states. Our contributions are:

- **Cheap bounds that remove almost all exact work.** 8-bit upper bounds drop all but a median of two rows per request, so exact FP32 scoring touches only 0.066% of the vocabulary per batch. One pass over the 8-bit rows serves the whole batch, and rows kept by several requests are read from the FP32 head once (Section 3).
- **Upper bounds that account for rounding.** The bound covers both the 8-bit encoding error and floating-point rounding, so the 8-bit codes can drop rows while final scores and tie breaking still use the original FP32 rows (Section 3.2).
- **A comparison with exact baselines.** On Qwen2.5-0.5B, FUSEHEAD beats the exact oracle on the main trace and on two held-out groupings, and beats the official batched LEMP-LI on every trace, while matching the full FP32 output on every decision (Section 4).

2 BACKGROUND

Greedy decoding is MIPS. A transformer applies a learned output matrix before choosing a token (Vaswani et al., 2017). The Qwen2 output head has no bias, so Equation 1 is exactly the greedy decoding rule, which picks the single top token.

Exact MIPS. Exact MIPS methods avoid computing every inner product by bounding scores. LEMP groups rows by their length and chooses a search method for each group (Teflioudi & Gemulla, 2017). FEXIPRO rotates the data, scans it in order, and uses partial products and error bounds (Li et al., 2017). MAXIMUS skips rows by angle and uses matrix multiplication on blocks of rows (Abuzaid et al., 2019). LEMP and MAXIMUS already share some work across a batch: LEMP keeps each group of rows in cache while it serves several queries, and MAXIMUS shares block products within clusters of similar queries. Each query, however, still runs its own search over the index.

What batching changes. Suppose a decoding step produces B hidden states. Only B rows can win, yet dense matrix multiplication computes all $B \times V$ scores at full precision. Running an indexed search B times skips some rows, but repeats the search for every request. We want something in between: rule out rows with a cheap bound that all B requests can share, and do full-precision work only for rows that might win. Doing this without changing any output requires four things. (1) The compact rows must only rule out rows; they must not define a different, quantized model. (2) Bounds must hold for the FP32 scores that the reference actually computes, including rounding. (3) The kernel must read the compact rows and their extra data once per batch. (4) The final FP32 step must break ties in the same way as the reference, while scoring only the rows that remain.

3 METHOD

FUSEHEAD handles each batch in three steps. First, it finds a good candidate token for each request and computes that token's exact FP32 score; we call this the *best token so far*. Second, it reads the 8-bit vocabulary once and, for every row and request, decides whether the row could beat the best token so far. Third, it collects every row that some request kept, scores those rows exactly, and returns the winner for each request. Throughout the paper, the *full FP32 reference* is the argmax of FP32 scores computed by one fixed dot-product routine, which adds the terms in coordinate order using fused multiply-add (FMA), with ties broken toward the lowest token ID. FUSEHEAD uses the same routine for every exact score, so it compares exactly the same numbers as the reference.

3.1 8-BIT ROWS AND A BOUND IN EXACT ARITHMETIC

For each FP32 row w_i, we store 8-bit codes together with the row's minimum value a_i and a step size s_i:

$$a_i = \min_k w_{ik}, \qquad s_i = \frac{\max_k w_{ik} - a_i}{255}, \qquad c_{ik} = \operatorname{round}\left(\frac{w_{ik} - a_i}{s_i}\right), \tag{2}$$

with $s_i = 0$ for a constant row. Besides the codes $c_i \in \{0, \dots, 255\}^d$, we store five FP64 numbers per row: a_i, s_i, $\|c_i\|_2$, $\|w_i\|_2$, and an error radius $\widetilde{\rho}_i = \operatorname{up}(\|\mathrm{fl}_{64}(w_i - \widetilde{w}_i)\|_2)$ that measures how far the 8-bit version $\widetilde{w}_i = \mathrm{fl}_{64}(a_i\mathbf{1} + s_i c_i)$ is from the original row, where up means rounding toward $+\infty$. This is the standard way to turn real numbers into 8-bit integers (Jacob et al., 2018), but the codes never replace the output head; they are used only to bound its scores, which meets requirement (1). Let $\rho_i^\star$ be the exact distance between w_i and $a_i\mathbf{1} + s_i c_i$. By the Cauchy–Schwarz inequality, for any request h_b,

$$w_i^\top h_b \le a_i \mathbf{1}^\top h_b + s_i c_i^\top h_b + \rho_i^\star \|h_b\|_2. \tag{3}$$

The right-hand side needs only the 8-bit codes, the numbers stored per row, and two numbers per request, $\mathbf{1}^\top h_b$ and $\|h_b\|_2$.

3.2 ACCOUNTING FOR FLOATING-POINT ROUNDING

Equation 3 holds in exact arithmetic, but the kernel computes it in floating point, and the score it must stay above is itself a rounded FP32 dot product. We therefore add an error term for each of the three floating-point computations involved, and we round every operation of the bound upward, so each partial result can only be too large, never too small. The kernel computes the code dot product $\widehat{z}_{ib}$ with AVX2 FP32 FMAs, and the sums $\widehat{q}_b$ of h_b and A_b of $|h_b|$ in FP64. Let C_i, W_i, and Q_b be stored upper bounds on $\|c_i\|_2$, $\|w_i\|_2$, and $\|h_b\|_2$, each rounded upward. With machine epsilon ε_p for precision p, let $\gamma_d^{(p)} = d\varepsilon_p/(1 - d\varepsilon_p)$, and let η_{32} be the smallest positive FP32 number (a

subnormal number). The three error terms and the final bound are

$$D_{ib} = \mathrm{up}\Big(\gamma_d^{(32)} C_i Q_b + (d+8)\eta_{32}\Big), \quad S_b = \mathrm{up}\Big(\gamma_d^{(64)} A_b\Big), \quad T_{ib} = \mathrm{up}\Big(\gamma_d^{(32)} W_i Q_b + d\eta_{32}\Big), \tag{4}$$

$$U_{ib} = \mathrm{up}\big(a_i \widehat{q}_b + s_i \widehat{z}_{ib} + |s_i| D_{ib} + |a_i| S_b + \widetilde{\rho}_i Q_b + T_{ib}\big). \tag{5}$$

D_{ib} covers the FP32 code dot product (including the final sum of its eight AVX2 lanes), S_b covers the FP64 sum over the request, and T_{ib} covers the FP32 dot product that the reference uses to score row i; in Equation 5, every multiplication and addition is rounded toward $+\infty$. Each thread switches the CPU to upward rounding, turns off the two modes that would silently replace very small numbers with zero, and restores the caller's settings when it finishes (Section 3.4). If any partial result overflows or becomes NaN, the bound is set to $+\infty$, so the row is kept. The error terms and upward rounding together meet requirement (2).

3.3 Dropping rows and exact scoring

Best token so far. For each request, an HNSW search with $ef = 16$ proposes eight candidate rows. We add a fixed list of special tokens and common single-token words, score all of these candidates exactly in FP32, and keep the best one: token j_b with score ℓ_b. HNSW only supplies this starting point; it never decides the output.

One pass over the 8-bit vocabulary. The rows are split into fixed, equal blocks, one per OpenMP thread. For each row, an AVX2 kernel loads the codes 16 bytes at a time, converts them once, updates one sum per request, and then computes U_{ib} for every request b. Row i is kept for request b if

$$U_{ib} > \ell_b \quad \text{or} \quad (U_{ib} \geq \ell_b \ \wedge \ i < j_b). \tag{6}$$

The second condition keeps a row with a lower ID that could tie the best token so far, so that ties can still go to the lowest ID. We compile a separate version of the kernel for each batch size $B \in \{8, 12, 24\}$ so that the per-request sums stay in CPU registers. Each 8-bit row and its extra numbers are thus read once per batch, which meets requirement (3).

Exact scoring of the kept rows. Let S_b be the rows kept for request b. FuseHead reads all rows in $S = \bigcup_b S_b$ from the FP32 head once and computes $HW_S^\top$; for each request, a mask ignores the rows that request did not keep. Every remaining score uses the same FP32 dot-product routine as the reference, and candidates are compared in increasing order of token ID, which meets requirement (4).

Why the output matches the reference. The returned token can differ from the reference only if a dropped row scores higher than the returned token, or ties with it and has a lower ID. Equation 6 rules this out whenever U_{ib} is at least the reference FP32 score of every dropped row. We check this inequality for every row and every request on all tested batches (Section 4.3): it always holds, with a smallest margin of 0.407 on the main trace and 0.419 on the held-out states.

3.4 Implementation and cost

FuseHead is a C++ kernel written with AVX2 instructions. It runs on several threads with OpenMP and is called from Python through a C interface. The 8-bit codes are stored row by row in one block of memory, and the extra numbers per row are stored in FP64. The per-request sums and upper bounds on the norms are computed once per batch. Each thread handles a block of consecutive rows and keeps its own per-request state, which is combined after the pass. The full FP32 reference, the exact scoring of HNSW candidates, the exact scoring of kept rows, and the row-by-row bound checks all call the same native FP32 dot-product routine.

Rounding control. Each OpenMP thread saves the caller's floating-point control register (MXCSR) and switches to upward rounding. It also turns off the flush-to-zero and denormals-are-zero modes, which would silently replace very small numbers with zero, and it restores the register when it finishes. Tests cover callers in all four IEEE rounding modes with one and four threads.

Cost. For each batch, the kernel reads Vd code bytes and $5V$ FP64 numbers, regardless of B, whereas B separate 8-bit searches would read them B times. Exact scoring reads $4d|\bigcup_b S_b|$ bytes of FP32 rows rather than $4d\sum_b |S_b|$, so a row needed by several requests is read once. The arithmetic and the $B \times V$ byte mask still grow with B. On the main trace, the compact input is 142.2 MB per batch, the total bytes read, including the candidate search, masks, and exact scoring, are 144.1–148.0 MB, and the median batch scores only 101 FP32 rows exactly.

4 EXPERIMENTS

We ask four questions. Is FUSEHEAD faster than exact baselines and baselines that give the same output (Section 4.2)? Does it always return the reference token, and how many rows does it drop (Section 4.3)? How do per-token latency and the rest of the model affect the result (Section 4.4)? Does the speedup hold on held-out hidden states (Section 4.5)?

4.1 SETUP

Model and data. We use an FP32 checkpoint based on Qwen2.5-0.5B (Yang et al., 2025; Qwen Team, 2024), with hidden size 896 and all 151,936 output rows. We select 24 prompts, without looking at model outputs, from evenly spaced parts of the WikiText-2 test split (Merity et al., 2017); each prompt has 24 tokens and each request generates 12 tokens greedily, which gives 288 decisions. These requests form the main trace used in Sections 4.2–4.4; Section 4.5 uses separate held-out hidden states. The test prompts share no rows or text with the data used to build or tune the method. Requests arrive in groups of 8, 4, and 12 at steps 0, 4, and 8 of a simulated schedule, and every active request is served once per step. When 8, 12, or 24 requests are active they form one batch; 16 active requests are split into two batches of eight. This gives 24 batches: 12 of size 8, eight of size 12, and four of size 24. A sanity check with synthetic hidden states follows the same schedule, includes an exact tie, drops 10,296 row–request pairs, and returns all 288 expected IDs.

Hardware and procedure. All runs use an AMD EPYC 7742 and are restricted to CPUs 16–23, one hardware thread on each of eight physical cores; the second hardware thread of each core is left unused. Software versions are PyTorch 2.14.0, Transformers 5.16.1, hnswlib 0.8.0, and oneDNN v3.12. Every method gets the same hidden-state tensors. We run two warm-up traces and 28 measured repetitions. In each repetition, one method processes all 24 batches before the next method starts, and we rotate the order of the methods so that each method runs right before and right after every other method equally often. Python garbage collection is turned off while timing. Each batch timer starts when the hidden states are ready and stops when the token IDs are returned; a method's trace time is the sum over its 24 batches. For FUSEHEAD, the timer includes the HNSW search, the 8-bit pass, the masks, exact FP32 scoring, and tie breaking.

How we report speedups. For each comparison we report two numbers. The *ratio of medians* divides the baseline's median trace time by FUSEHEAD's. The *paired speedup* divides the baseline's total time over all 28 repetitions by FUSEHEAD's total time; the repetitions are paired because both methods run in each of them. Its 95% confidence interval comes from resampling the repetition pairs 10,000 times. We use the paired speedup as the main number because it uses every repetition, including slow ones.

Baselines.

- **FP32 SGEMM**: `torch.mm` followed by an argmax that returns the first largest index.
- **Single-query 8-bit search**: the same 8-bit data and bounds as FUSEHEAD, but each request is searched separately with eight threads.
- **Query-parallel 8-bit search**: the same per-request searches run side by side, one thread each, after a shared HNSW step, never using more than eight threads in total.
- **Official batched LEMP-LI** (Teflioudi & Gemulla, 2017): our wrapper calls `Lemp::initialize` once and then `Lemp::runTopK` with $k = 1$ on each batch. We widen one 8-bit type in its source so that it can represent $d = 896$; the search logic is unchanged. Both official SIMD options are on. We try several cache sizes on the first batch of

Table 1: **Head time on the main trace.** All methods process the same 288 hidden states on eight CPU cores. Head time is the median over 28 repetitions; the last column divides it by the matching FUSEHEAD median. LEMP-LI was timed in a separate run in which FUSEHEAD's median is 0.675 s; all other rows come from one shared run, where it is 0.591 s.

Method	Computation	Same output	Head (s)	Relative time
FUSEHEAD (ours)	8-bit bounds + exact FP32	**288/288**	**0.591**	1.00×
HNSW ($ef = 400$, top-50)	approximate	286/288	1.221	2.07×
Query-parallel 8-bit search	8-bit bounds + exact FP32	288/288	2.462	4.17×
Single-query 8-bit search	8-bit bounds + exact FP32	288/288	2.376	4.02×
Native BMM	exact MIPS	288/288	2.542	4.30×
Exact oracle (BMM or MAXIMUS)	exact MIPS	288/288	2.542	4.30×
FP32 SGEMM	full FP32 head	288/288	2.692	4.56×
Official batched LEMP-LI	exact MIPS	288/288	4.454	6.60×
oneDNN int8 + error bound	int8 bounds + exact FP32	288/288	9.387	15.89×
Complete native MAXIMUS	exact MIPS	288/288	34.975	59.22×

24 in the main trace and use the best one everywhere else. LEMP-LI is timed in a separate run against FUSEHEAD alone, and its one-time setup cost is reported separately.

- **Exact oracle**: for each batch, the faster of two exact methods from the official MAXIMUS code (Abuzaid et al., 2019): batched matrix multiplication (CBLAS DGEMM plus `computeTopK`, called BMM below) and complete MAXIMUS (query clustering, the angle-based index, `computeTopKForCluster`, exact products on blocks of rows, and top-1 selection). The number of clusters and the block size are chosen on the first batch of 24 in the main trace. Taking the faster of the two for each batch makes this baseline stronger than either method alone.
- **oneDNN int8 with error bounds**: oneDNN v3.12 (oneDNN Contributors, 2026) with signed 8-bit weights (one scale per row), signed 8-bit inputs (one scale per request), and 32-bit integer sums, followed by error bounds and FP32 scoring of the kept rows. In our runs, no integer sum overflows or saturates, and every token it returns matches the reference.
- **HNSW**: inner-product search with $ef = 400$, followed by exact re-scoring of the top 50 candidates; this baseline is approximate.

4.2 IS FUSEHEAD FASTER?

Table 1 gives the main result. FUSEHEAD's median head time is 0.591 s, compared with 2.542 s for the exact oracle and 2.376 s for single-query 8-bit search, the fastest baseline that returns the same tokens. These are ratios of medians of 4.30× and 4.02×; the paired speedups are 2.03× [1.18, 4.10] and 2.94× [1.47, 5.56]. In the oracle, BMM is faster than complete MAXIMUS in all 672 batch choices: complete MAXIMUS visits all 151,936 rows for every request and takes 34.975 s in total (1.020 s clustering, 10.960 s building the angle-based index, 20.202 s search, and 2.822 s selection). Against query-parallel 8-bit search, the paired speedup is 2.03× [1.16, 4.03]. The approximate HNSW baseline is also slower and changes two tokens.

FUSEHEAD has the lowest median batch time at every batch size (Figure 2a): 15.6, 25.9, and 47.2 ms for $B = 8$, 12, and 24, against 65.8, 98.8, and 107.5 ms for the baseline with the same output and the lowest total time at each size (single-query 8-bit search for $B = 8$ and 12, the exact oracle for $B = 24$). The paired speedups are 2.41× [1.21, 4.69], 2.95× [1.61, 5.06], and 1.31× [0.73, 2.36]. The gain is smallest at $B = 24$ because the fastest baseline changes there: batched matrix multiplication spreads one full pass over the head across all requests in the batch, so its time per request falls as the batch grows (Figure 2b): per request, the oracle's median falls from 12.1 ms at $B = 8$ to 8.9 and 4.5 ms at $B = 12$ and 24, while FUSEHEAD's stays between 1.95 and 2.15 ms, because its arithmetic, masks, and candidate search grow with B (Section 3.4). Single-query 8-bit search stays at 8.2 ms per request at every size, since each request repeats the full 8-bit pass.

LEMP-LI, the most relevant earlier method because it also shares cached data across a batch of queries, takes 4.454 s against 0.675 s for FUSEHEAD in the same run, a paired speedup of 6.557× [6.062, 6.993], with both methods returning all 288 reference tokens.

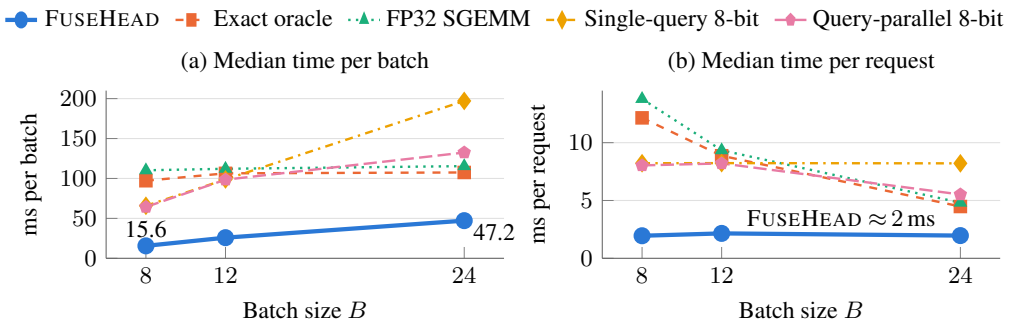


Figure 2: **Head time by batch size on the main trace.** Medians over all batches of each size and all 28 repetitions; (b) divides (a) by B. Only methods that return all 288 reference tokens are shown (oneDNN int8, 313–468 ms per batch, is off scale). Batched matrix multiplication shares one pass over the head, so its gap to FUSEHEAD narrows as B grows.

4.3 DOES FUSEHEAD RETURN THE REFERENCE OUTPUT?

Matching tokens alone could hide a wrong bound that happened not to matter, so we check the bound itself. All 288 outputs on the main trace equal the reference, and FUSEHEAD never falls back to dense computation. On a replayed batch of 24 requests from the main trace, we compare all $24 \times 151{,}936$ bounds with the reference FP32 scores: every bound is larger, with a smallest margin of 0.407. The same check on every held-out request, against every row, gives a smallest margin of 0.419. Further tests target the arithmetic and the tie rule directly: callers in each rounding mode, large values that cancel each other, values that differ only in the last bit, very large and very small values, random inputs, near ties, and exact ties. Almost all rows are dropped: the median request keeps a fraction 1.316×10^{-5} of the rows (two rows for this vocabulary), and no request keeps more than 2.83×10^{-4}. Because different requests keep different rows, the set of kept rows per batch is larger, 57–186 rows with a median of 101, which is still tiny compared with the 8-bit pass.

4.4 LATENCY AND THE REST OF THE MODEL

The median per-token head latency is 26.0 ms for FUSEHEAD, compared with 103.8 ms for the exact oracle and 99.1 ms for single-query 8-bit search. In a full model, the head is only one part of each decoding step. We therefore also time rerunning the full Qwen model on each prefix without a KV cache, together with building inputs and collecting the final hidden states; the recomputed states match the reference within 3.44×10^{-4} maximum absolute error and give the same winning tokens. Adding this model time to each measured head time gives 19.158 s for FUSEHEAD and 20.841 s for single-query 8-bit search, or 1.088× its throughput.

4.5 HELD-OUT HIDDEN STATES

The held-out set contains 36 hidden states taken from three WikiText-2 rows that do not appear in the main trace. We group the same 36 states in two ways fixed in advance: three batches of 12 ($12+12+12$), and one batch of 24 plus one of 12 ($24+12$). In both groupings, every exact method and every 8-bit method returns all 36 reference tokens, every FUSEHEAD bound is checked against every row, and no dense fallback occurs. Table 2 summarizes the speedup over the exact oracle. For $12+12+12$, median head times are 83.5 ms for FUSEHEAD and 323.4 ms for the oracle; for $24+12$, they are 73.3 and 205.8 ms. As on the main trace, the gain is smaller when the grouping includes a batch of 24. Against query-parallel 8-bit search, FUSEHEAD is 1.879× [1.034, 3.304] and 1.492× [1.086, 2.152] faster, and against LEMP-LI it is 6.518× [5.646, 7.118] and 6.158× [6.030, 6.313] faster, again with all 36 tokens matching.

Table 2: **Speedup over the exact oracle on the main trace and on both held-out groupings.** Each row uses 28 paired repetitions, and every output matches the reference.

Trace	Paired speedup	95% interval
Main trace	**2.029**×	**[1.180, 4.098]**
Held-out 12 + 12 + 12	**1.952**×	**[1.131, 3.461]**
Held-out 24 + 12	**1.407**×	**[1.022, 2.052]**

5 Related Work

Exact and approximate MIPS. LEMP keeps each group of rows in cache while serving a batch of queries (Teflioudi & Gemulla, 2017), and our LEMP-LI baseline runs its official batch loop. FEXIPRO applies SVD and integer upper bounds before exact scoring of one query at a time (Li et al., 2017), and MAXIMUS shares block products within clusters of users (Abuzaid et al., 2019). Several methods first filter with low-precision bounds and then compute full-precision scores, but for one query at a time: EAHR's PVS uses int8 intervals for each vector before float32 scoring (Zhang, 2026); RaBitQ uses error bounds that hold with high probability to decide which original vectors to re-score (Gao & Long, 2024); and ETAR scans part of an int8 copy by a fixed rule and then re-scores a fixed number of candidates in FP32 (Rasyidi & Alhussein, 2026). Faiss PQ FastScan scans compressed codes for several queries at once and can re-score candidates with the original vectors (André et al., 2015; Faiss Contributors, 2026), but its scores are approximate rather than upper bounds, and it neither guarantees the true top-1 nor shares one re-scoring set across queries. Shi et al. (2018) merge approximate LSH candidate lists across beam-search states and score the merged list with one batched multiplication; this shares candidates across queries, but it cannot rule out words it never retrieved. Among recent work, NanoSpec (arXiv:2605.26444) reuses one small draft vocabulary across decoding steps and checks the result with the full head; SpecVocab (arXiv:2602.13836) learns a vocabulary subset for each state without bounds; and the Certified Inference code (DOI:10.5281/zenodo.21893073) proves the exact top token in real arithmetic for short prompts rather than selecting tokens in FP32 during decoding. Approximate graph search, including graphs for inner products (Morozov & Babenko, 2018), gives up some accuracy for speed, and recent work applies HNSW directly to LLM output heads (Loretz & Hochreiter, 2026). FuseHead uses HNSW only to find a starting candidate. We know of no earlier design that combines one shared 8-bit pass over the vocabulary, a bound for each request, one shared FP32 read of the kept rows, and the reference's rounding and tie rules; we claim novelty for this combination, not for any single part.

Efficient output layers and quantization. Adaptive softmax changes the trained output layer to reduce the expected work (Grave et al., 2017); FuseHead instead works on an existing dense head without retraining. SVD-Softmax first scores the vocabulary with a low-rank approximation and then recomputes selected entries at full width, leaving the rest approximate (Shim et al., 2017). Learning to Screen predicts a word subset for each context and reports high but not perfect agreement with the full vocabulary (Chen et al., 2019). Fast Graph Decoder and vector-index output embeddings apply approximate graph search to output layers (Zhang et al., 2018; Loretz & Hochreiter, 2026). FlashHead groups the vocabulary into clusters and scores them with partly quantized values, without retraining (Tranheden et al., 2026); its fast path supports only batch size one, raises an error for batch sizes 2–10, and falls back to dense computation above 10, and it does not guarantee that the true winner is among the tokens it scores. The concurrent `topk-head` scans FP16/FP32 blocks with a bound for each query, without exact FP32 scoring of a shared candidate set or a fixed tie rule (anishesg, 2026). ARCHead compresses the output head into a quantized low-rank part plus a correction term, and the compressed head defines its outputs (Kocabay et al., 2026). FBGEMM (Khudia et al., 2021) and SmoothQuant (Xiao et al., 2023) show the value of low-precision matrix multiplication for accurate quantized inference. In FuseHead, by contrast, the 8-bit rows only bound the scores of an unchanged FP32 head, and any row that might win is scored in FP32. FlashSampling v2 combines the dense GPU output-head multiplication with Gumbel-Max sampling: it computes blocks of the vocabulary on chip, keeps the best entry of each block, and combines them without writing all scores to GPU memory (Ruiz et al., 2026b;a). It keeps the sampling distribution unchanged but still computes every dense block product; FuseHead targets greedy decoding on CPUs and avoids most FP32 products.

LLM serving and CPU execution. Schedulers that advance all requests step by step create the changing batches that FUSEHEAD uses (Yu et al., 2022). CPU-aware inference systems decide where and how to run each part of the model based on the available hardware (Park & Egger, 2024; Na et al., 2025), and specialized engines combine serving kernels with scheduling (Ye et al., 2025). These systems optimize the whole serving pipeline; FUSEHEAD is a component that such systems could use for exact greedy token selection on CPUs.

6 LIMITATIONS

Scope of the evaluation. All results come from one checkpoint, one output head with 151,936 rows, one eight-core AMD EPYC 7742 machine, and batch sizes 8, 12, and 24. The checkpoint is the public Qwen2.5-0.5B model after 120 further training steps of our own; we do not report results for the original public model. The main trace uses 24 WikiText-2 prompts and a simulated arrival pattern rather than real traffic, and we test greedy decoding only; sampling from the full softmax needs every score and is outside the method. Other model families, hidden sizes, CPUs, and accelerators are untested.

Scope of the correctness guarantee. The error radius $\widetilde{\rho}_i$ is computed from an ordinary FP64 reconstruction and subtraction, so it is not guaranteed to be at least the exact error $\rho_i^\star$. Equation 5 is therefore not proven for every input; it is shown to hold by the comparisons against every row on the main trace and on both held-out groupings (Section 4.3). Computing an enclosure of $\rho_i^\star$ with directed rounding would close this gap at a one-time offline cost, but we have not implemented it. Output token IDs are checked for one timed repetition per method; the other 27 repetitions are timed only.

Baselines, batch size, and variance. Several baselines were not tuned beyond the settings in Section 4.1, so the reported speedups may be larger than those against fully optimized versions; for example, the oracle's BMM top-1 selection takes 33.8 ms per batch of 24 in the main run but 9.4 ms in separate runs, and LEMP-LI with its default small-batch settings drops no rows (Appendix A). The advantage over the exact oracle shrinks as batches grow (Figure 2), the $B = 24$ interval includes one, and batches larger than 24 are untested. Three of the 28 main-trace FUSEHEAD repetitions take 4.0–11.7 s against a median of 0.59 s, which explains the gap between the paired speedup (2.03×) and the ratio of medians (4.30×), and FUSEHEAD's 95th-percentile per-token head latency (229.0 ms) is higher than the exact oracle's (145.4 ms).

Source of the speedup and end-to-end gains. We do not separate how much of the speedup comes from dropping rows and how much from reading each 8-bit row once per batch: under a controlled memory load on the batch of 24, the speedup over the exact oracle is 0.873× [0.521, 1.652] (Appendix A), so we make no claim that the design reduces DRAM traffic. The 1.088× full-model result adds separately measured model and head times and is not a measurement of a serving system.

7 CONCLUSION

FUSEHEAD uses cheap 8-bit upper bounds, computed in one pass over the vocabulary for a whole batch of requests, to rule out almost every row, and scores the few remaining rows exactly in FP32. On the tested CPU workload, it keeps a median of two rows per request, returns the full FP32 token for every decision, and beats the exact oracle by 2.03× [1.18, 4.10] on the main trace and by 1.95× [1.13, 3.46] and 1.41× [1.02, 2.05] on two held-out groupings. It beats the official batched LEMP-LI by 6.56× [6.06, 6.99] on the main trace and by at least 6.16× on both held-out groupings. Exact greedy token selection therefore needs neither a full FP32 pass over the head nor a separate index search for each request. Extending this result to other models, hardware, and serving setups is the natural next step.

Reproducibility Statement

The supplementary material provides code, configurations, fixed model and data versions, all repetitions, token IDs, bound values, byte counts, and held-out results. Runs fix random seeds, CPU assignment, thread counts, method order, request routing, and bootstrap units. Sections 3.4 and 4.1 describe the implementation, the setup, and every baseline.

LLM Usage Statement

In line with the ICLR policy on large language models (LLMs), we disclose that LLM-based assistants were used in this work in three main ways. **(1) Writing and polishing the paper.** LLMs helped draft and revise the text, improve clarity, wording, and grammar, and check that the terminology and numbers stay consistent across sections. The authors reviewed and edited all LLM-assisted text and verified the reported numbers against the logged raw results. **(2) Running experiments.** LLM-based coding agents helped launch the benchmark and verification scripts under our fixed configurations, monitor the runs, and collect the raw outputs (timings, token IDs, bound values, and byte counts) into result files. The experimental protocol, the pinned CPUs and thread counts, the method order, and the acceptance checks were fixed in configuration files before the timed runs. **(3) Writing code.** LLMs assisted in writing the C++/AVX2 kernel, the Python interface, the baseline wrappers, the analysis and plotting scripts, and the tests. The generated code was reviewed by the authors and checked by the tests described in Sections 3.4 and 4.3, including the rounding-mode tests, the exact-tie cases, and the comparison of every bound against the reference FP32 score of every row. The authors take full responsibility for all content of the paper, including any text or code produced with LLM assistance; LLMs are not authors of this work.

References

Firas Abuzaid, Geet Sethi, Peter Bailis, and Matei Zaharia. To index or not to index: Optimizing exact maximum inner product search. In *2019 IEEE 35th International Conference on Data Engineering*, pp. 1250–1261, 2019. doi: 10.1109/ICDE.2019.00114. URL https://doi.org/10.1109/ICDE.2019.00114.

Fabien André, Anne-Marie Kermarrec, and Nicolas Le Scouarnec. Cache locality is not enough: High-performance nearest neighbor search with product quantization fast scan. *Proceedings of the VLDB Endowment*, 9(4):288–299, 2015. doi: 10.14778/2856318.2856324. URL https://doi.org/10.14778/2856318.2856324.

anishesg. topk-head: Fused LM-head top-k decode. GitHub repository, 2026. URL https://github.com/anishesg/topk-head. Commit bf28f51e0ba8.

Patrick H. Chen, Si Si, Sanjiv Kumar, Yang Li, and Cho-Jui Hsieh. Learning to screen for fast softmax inference on large vocabulary neural networks. In *ICLR*, 2019.

Faiss Contributors. Faiss fast accumulation of PQ and AQ codes. Official software documentation, 2026. URL https://github.com/facebookresearch/faiss/wiki. Sections "Reranking" and "handling multiple queries", accessed 2026-09-04.

Jianyang Gao and Cheng Long. RaBitQ: Quantizing high-dimensional vectors with a theoretical error bound for approximate nearest neighbor search. *Proceedings of the ACM on Management of Data*, 2(3):1–27, 2024. DOI: 10.1145/3654970.

Edouard Grave, Armand Joulin, Moustapha Cisse, David Grangier, and Herve Jegou. Efficient softmax approximation for GPUs. In *Proceedings of the 34th International Conference on Machine Learning*, volume 70, pp. 1302–1310, 2017. URL https://proceedings.mlr.press/v70/grave17a.html.

Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 2704–2713, 2018. doi: 10.1109/CVPR.2018.00286. URL https://doi.org/10.1109/CVPR.2018.00286.

Daya Khudia, Jianyu Huang, Protonu Basu, Summer Deng, Haixin Liu, Jongsoo Park, and Mikhail Smelyanskiy. FBGEMM: Enabling high-performance low-precision deep learning inference. *arXiv preprint arXiv:2101.05615*, 2021.

Şuayp Talha Kocabay, Talha Rüzgar Akkuş, and Kamer Ali Yuksel. ARCHead: Activation-metric residual correction for large language model output heads. *arXiv preprint arXiv:2608.02703*, 2026.

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 2017 ACM International Conference on Management of Data*, pp. 835–850, 2017. doi: 10.1145/3035918.3064009. URL https://doi.org/10.1145/3035918.3064009.

Martin Loretz and Sepp Hochreiter. Accelerating LLM inference via vector index based output embeddings. *arXiv preprint arXiv:2608.27460*, 2026.

Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models. In *ICLR*, 2017.

Stanislav Morozov and Artem Babenko. Non-metric similarity graphs for maximum inner product search. In *Advances in Neural Information Processing Systems*, volume 31, pp. 4721–4730, 2018.

Seonjin Na, Geonhwa Jeong, Byung Hoon Ahn, Aaron Jezghani, Jeffrey Young, Christopher J. Hughes, Tushar Krishna, and Hyesoon Kim. FlexInfer: Flexible LLM inference with CPU computations. In *Proceedings of Machine Learning and Systems*, volume 7, 2025.

oneDNN Contributors. oneDNN v3.12 matmul primitive and quantization attributes. Software documentation and release, 2026. URL https://github.com/uxlfoundation/oneDNN/tree/v3.12.

Daon Park and Bernhard Egger. Improving throughput-oriented LLM inference with CPU computations. In *Proceedings of the 2024 International Conference on Parallel Architectures and Compilation Techniques*, pp. 233–245, 2024. DOI: 10.1145/3656019.3676949.

Qwen Team. Qwen2.5-0.5B model artifact. Hugging Face model repository, 2024. URL https://huggingface.co/Qwen/Qwen2.5-0.5B. Revision 060db6499f32.

Mohammad Arif Rasyidi and Omar Alhussein. Index-free dynamic edge retrieval with energy-tail-aware partial scans. *arXiv preprint arXiv:2609.01820*, 2026. Accepted for presentation at the 2026 IEEE 12th World Forum on Internet of Things.

Tomas Ruiz, Zhen Qin, Yifan Zhang, Xuyang Shen, Yiran Zhong, and Mengdi Wang. FlashSampling: Official implementation. GitHub repository, 2026a. URL https://github.com/FlashSampling/FlashSampling. Commit 1e5d8fa3eb9a3fe84cb56a1f2b9949f2e1d16b50, accessed September 4, 2026.

Tomas Ruiz, Zhen Qin, Yifan Zhang, Xuyang Shen, Yiran Zhong, and Mengdi Wang. FlashSampling: Fast and memory-efficient exact sampling. *arXiv preprint arXiv:2603.15854*, 2026b.

Xing Shi, Shizhen Xu, and Kevin Knight. Fast locality sensitive hashing for beam search on GPU. *arXiv preprint arXiv:1806.00588*, 2018.

Kyuhong Shim, Minjae Lee, Iksoo Choi, Yoonho Boo, and Wonyong Sung. SVD-Softmax: Fast softmax approximation on large vocabulary neural networks. In *Advances in Neural Information Processing Systems*, volume 30, pp. 5463–5473, 2017.

Christina Teflioudi and Rainer Gemulla. Exact and approximate maximum inner product search with LEMP. *ACM Transactions on Database Systems*, 42(1):1–49, 2017. DOI: 10.1145/2996452.

Wilhelm Tranheden, Shahnawaz Ahmed, Devdatt Dubhashi, Jonna Matthiesen, and Hannes von Essen. FlashHead: Efficient drop-in replacement for the classification head in language model inference. *arXiv preprint arXiv:2603.14591*, 2026.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, volume 30, pp. 5998–6008, 2017.

Guangxuan Xiao, Ji Lin, Mickael Seznec, Hao Wu, Julien Demouth, and Song Han. SmoothQuant: Accurate and efficient post-training quantization for large language models. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202, pp. 38087–38099, 2023.

An Yang, Baosong Yang, Beichen Zhang, et al. Qwen2.5 technical report. *arXiv preprint arXiv:2412.15115*, 2025.

Zihao Ye, Lequn Chen, Ruihang Lai, Wuwei Lin, Yineng Zhang, Stephanie Wang, Tianqi Chen, Baris Kasikci, Vinod Grover, Arvind Krishnamurthy, and Luis Ceze. FlashInfer: Efficient and customizable attention engine for LLM inference serving. In *Proceedings of Machine Learning and Systems*, volume 7, 2025.

Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for transformer-based generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation*, pp. 521–538, 2022.

Chunran Zhang. Exact adaptive hybrid retrieval without fixed top-L cutoffs. *arXiv preprint arXiv:2608.07152*, 2026.

Minjia Zhang, Xiaodong Liu, Wenhan Wang, Jianfeng Gao, and Yuxiong He. Navigating with graph representations for fast and scalable decoding of neural language models. In *Advances in Neural Information Processing Systems*, volume 31, pp. 6308–6319, 2018.

A Further Details on the Limitations

This appendix gives the details behind the short Limitations section (Section 6).

Baseline implementations. In the exact oracle, BMM top-1 selection takes a median of 33.8 ms per batch of 24 in the one main repetition where we recorded the time of each step, while the same selection takes 9.4 ms per batch of 24 in our separate memory-pressure runs. In that repetition, selection takes about 0.77 s of the oracle's trace (2.54 s median), and we have not re-measured the oracle with the faster selection. FP32 SGEMM uses default `torch.mm` and takes about 112 ms per batch, slower than the oracle's FP64 matrix multiplication (about 70 ms). The timed oneDNN path includes our error-bound check, which runs on one thread. With the source's default settings for small batches, LEMP-LI computes all 43,757,568 exact dot products and drops nothing; we did not lower its 60-query tuning threshold. HNSW's 286/288 agreement describes one setting, not the full trade-off between accuracy and speed.

Timing variation and tail latency. All timings come from a single machine. Three of the 28 main-trace FuseHead repetitions take 4.0–11.7 s, while the median is 0.59 s; these slow runs widen the intervals and explain the gap between the paired speedup (2.03×) and the ratio of medians (4.30×). The interval for $B = 24$, 1.31× [0.73, 2.36], includes one. FuseHead's 95th-percentile per-token head latency, 229.0 ms, is lower than that of single-query 8-bit search (701.0 ms) but higher than those of the exact oracle (145.4 ms), FP32 SGEMM (130.8 ms), and HNSW (118.0 ms). In the held-out $24 + 12$ run, eight of the 28 times for the batch of 24 are more than twice the 48.2 ms median, because the HNSW search and the scoring of its candidates slow down (medians 149.7 and 92.6 ms instead of 11.2 and 2.00 ms), while the other steps change by at most 3.3 ms. An earlier $24 + 12$ run did not keep per-row results; the reported $24 + 12$ run, which adds timing for each step, was planned after that run, so it is a follow-up rather than an independent confirmation.

Source of the speedup. We measure how many rows are dropped, but we do not separate how much of the speedup comes from dropping rows and how much from reading each 8-bit row once per batch. Under a controlled memory load on the batch of 24 (four background threads streaming 66.2 GB/s), the speedup over the exact oracle is 0.873× [0.521, 1.652], and the change in speedup between high and zero memory load is 0.721 [0.422, 1.431] against the oracle and 0.675 [0.385,

1.347] against LEMP-LI; both intervals include one. Hardware counters for memory traffic were not available, so we make no claim that the design reduces cache misses, DRAM traffic, or memory bandwidth use.

End-to-end serving. The 1.088× full-model result adds separately measured model and head times; it is not a measurement of a full serving system. Rerunning the model on each prefix without a KV cache takes most of the time in this setup and reduces the combined gain to 8.8%; network, scheduling, and KV-cache effects are not included. Both native oracle paths exclude process startup and data transfer.