FlashSampling: Fast and Memory-Efficient Exact Sampling

FlashSampling Authors

FlashSampling Project

March 7, 2026

Abstract

Sampling from a categorical distribution is mathematically simple, but in large-vocabulary decoding it often triggers extra memory traffic and extra kernels after the LM head. We present **FlashSampling**, an exact sampling primitive that fuses sampling into the LM-head matmul and never materializes the logits tensor in HBM. The method is simple: compute logits tile-by-tile on chip, add Gumbel noise, keep only one maximizer per row and per vocabulary tile, and finish with a small reduction over tiles. The fused tiled kernel is exact because argmax decomposes over a partition; grouped variants for online and tensor-parallel settings are exact by hierarchical factorization of the categorical distribution. Across H100, H200, B200, and B300 GPUs, FlashSampling speeds up kernel-level decode workloads, and in end-to-end vLLM experiments it reduces time per output token by up to 19% on the models we test. These results show that exact sampling can be integrated into the matmul itself, turning a bandwidth-bound postprocessing step into a lightweight epilogue.

Project Page: https://github.com/FlashSampling/FlashSampling

1 Introduction

Sampling from a categorical distribution is a small mathematical operation, but in large-categorical systems it can become an expensive inner-loop primitive. Modern LLM serving stacks invoke sampling repeatedly during autoregressive decoding, often on outputs with tens or hundreds of thousands of categories (Kwon et al., 2023; Ye et al., 2025; Maddison et al., 2014; Huijben et al., 2022). The bottleneck is usually not arithmetic. It is the systems decomposition used after the LM head.

At decode time, the LM-head projection already streams a large $[V, D]$ weight matrix from HBM. When the active batch is small, this projection is typically memory-bandwidth bound. Materializing the resulting $[B, V]$ logits tensor, launching extra kernels to normalize and sample from it, and then discarding it adds extra memory traffic and synchronization but no useful model computation. In this regime, the separate sampler is pure overhead (Dao et al., 2022; Wijmans et al., 2025). Throughout, B denotes batch size and V denotes the number of categories, such as vocabulary size.

Standard pipelines write logits to HBM and read them back for sampling, even though logits are immediately discarded after one sample is drawn. Exact sampling is often described as "compute softmax, then sample", which obscures the fact that exact sampling does not require forming

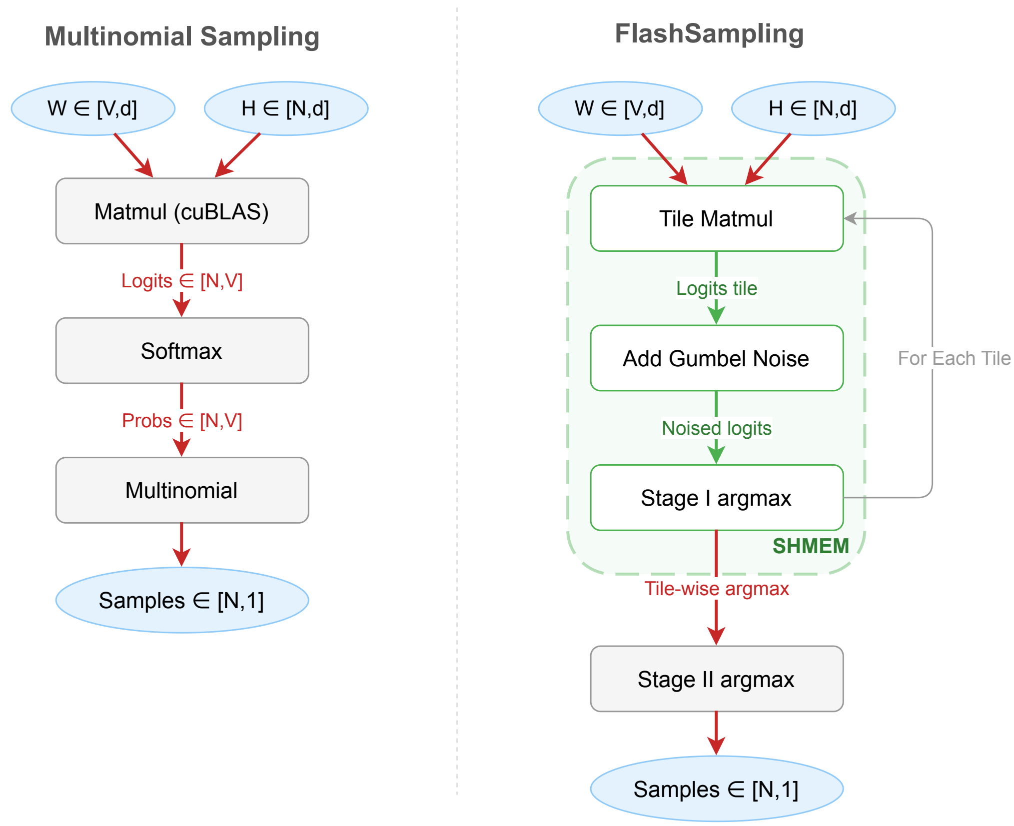

Figure 1 Conventional multinomial sampling (left) materializes the full $[B, V]$ logits tensor in HBM between the matmul and the sampler. FlashSampling (right) fuses sampling into the matmul epilogue, followed by a lightweight reduction over vocabulary tiles. Logits are computed tile-by-tile in on-chip memory, perturbed with Gumbel noise, and reduced without ever writing the full logits tensor to HBM. Red arrows denote HBM traffic; green arrows denote on-chip data movement.

probabilities at all. For large vocabularies, streaming and tensor-parallel settings turn sampling into a memory and communication problem if full logits must be materialized or gathered.

In this work, we introduce FlashSampling, which computes logits tile-by-tile on chip and writes only one candidate per row and per vocabulary tile, followed by a lightweight reduction. Exact sampling needs only the index of the largest perturbed logit, so there is no need to form a softmax, a prefix sum, or normalized probabilities. A simple hierarchical factorization yields exact online and distributed variants that keep only small summaries in flight and communicate only small summaries across ranks.

Our contributions could can be summarized below:

1. **FlashSampling, a simple fused exact sampler.** We introduce a two-stage design that computes logits tile-by-tile in the LM-head epilogue, adds Gumbel noise on chip, and stores only one candidate per row and per vocabulary tile instead of materializing the full $[B, V]$ logits tensor.

2. **A clean exactness argument.** We separate the two ingredients used in the paper: the fused tiled kernel is exact pathwise by argmax decomposition over vocabulary tiles, while grouped, online, and distributed variants are exact in distribution by hierarchical factorization through group log-masses.

3. **A systems analysis and evaluation.** We show why raw logits-byte savings alone are too small to explain the measured speedups, and we demonstrate consistent gains in the memory-bandwidth-bound decode regime across four NVIDIA GPUs and in end-to-end vLLM evaluation.

2 Background

Notation. Let $[V] := \{1, \ldots, V\}$. Let $\widetilde{\boldsymbol{\ell}} \in (\mathbb{R} \cup \{-\infty\})^V$ denote *transformed logits* after any deterministic operations such as additive bias, temperature scaling, or masking. We assume that each row has at least one finite entry; otherwise, the target categorical distribution is undefined. The target distribution is

$$p(i) = \frac{\exp(\widetilde{\ell}_i)}{\sum_{j=1}^{V} \exp(\widetilde{\ell}_j)}.$$

Raw logits $\boldsymbol{\ell}$ are the special case $\widetilde{\boldsymbol{\ell}} = \boldsymbol{\ell}$. We denote i.i.d. standard Gumbel variables by $g_i \sim \mathrm{Gumbel}(0, 1)$. Because the Gumbel law is continuous, ties occur with probability zero, so argmax is unique almost surely.

2.1 Why sampling is expensive at scale

A common materialized-logits pipeline first computes transformed logits, then forms probabilities, and finally samples from those probabilities. One representative example is softmax followed by prefix-sum sampling:

$$\text{GEMM(produce logits)} \;\rightarrow\; \text{write logits to HBM} \;\rightarrow\; \text{read logits for sampling.}$$

Algorithm 1 summarizes this pattern.

Not every implementation uses exactly these kernels, but any materialized-logits baseline pays the same structural costs: at least one logits write, at least one logits reread, and extra sampling work after the GEMM.

Algorithm 1 One common materialized-logits sampling pipeline

Require: Hidden state $\boldsymbol{h} \in \mathbb{R}^D$, LM-head weights $\boldsymbol{W} \in \mathbb{R}^{V \times D}$, optional deterministic transforms
Ensure: Sampled index $i^\star \in \{1, \ldots, V\}$

$\boldsymbol{\ell} \leftarrow \boldsymbol{W}\boldsymbol{h}$ ▷ GEMM: compute logits and write to HBM
$\widetilde{\boldsymbol{\ell}} \leftarrow \text{transform}(\boldsymbol{\ell})$ ▷ temperature, bias, mask; read/write HBM
$m \leftarrow \max_i \widetilde{\ell}_i$ ▷ pass 1 over transformed logits
$Z \leftarrow \sum_{i=1}^{V} \exp(\widetilde{\ell}_i - m)$ ▷ pass 2 over transformed logits
$p_i \leftarrow \exp(\widetilde{\ell}_i - m)/Z$ for all i ▷ write probabilities
$c_i \leftarrow \sum_{j=1}^{i} p_j$ for all i ▷ prefix sum
Draw $u \sim \text{Unif}(0, 1)$
$i^\star \leftarrow \min\{i : c_i \geq u\}$ ▷ search
return $i^\star$

Decode regime. In autoregressive decoding, B is typically small. The LM-head projection is then often memory-bandwidth bound because it repeatedly streams the large $[V, D]$ weight matrix from HBM. Materializing $[B, V]$ logits and reading them back for sampling adds multiple avoidable HBM round-trips in the most latency-sensitive part of the decode loop (Kwon et al., 2023; Ye et al., 2025).

2.2 GPU Memory Hierarchy

Table 1 summarizes the GPU memory hierarchy. On-chip memory (registers, SRAM) is orders of magnitude faster than HBM but far smaller. FlashSampling exploits this gap by keeping logits in registers/SRAM and never writing the full logits tensor to HBM.

Table 1 GPU memory hierarchy (H100 SXM) (NVIDIA, 2022, 2024).

Level	Capacity	Bandwidth
Registers/SRAM	256 KB / SM	> 100 TB/s
L2 cache	50 MB	∼ 12 TB/s
HBM3	80 GB	3.35 TB/s

2.3 The Gumbel-Max trick

The classical Gumbel-Max trick states that exact categorical sampling can be performed by adding i.i.d. Gumbel noise and taking an argmax:

Theorem 2.1 (Gumbel-Max)**.** Let $\widetilde{\boldsymbol{\ell}} \in (\mathbb{R} \cup \{-\infty\})^V$ have at least one finite entry, and let $\{g_i\}_{i=1}^V$ be i.i.d. Gumbel$(0, 1)$. Then

$$i^\star = \operatorname*{argmax}_{i \in [V]} \left(\widetilde{\ell}_i + g_i\right) \quad \Longrightarrow \quad \mathbb{P}(i^\star = i) = \frac{e^{\widetilde{\ell}_i}}{\sum_{j=1}^{V} e^{\widetilde{\ell}_j}}.$$

This classical result goes back to Gumbel (1954) and is widely used in machine learning (Maddison et al., 2014; Huijben et al., 2022). The key point for this paper is simple: *exact sampling does not require an explicit softmax.* It only requires the index of the largest perturbed logit.

3 FlashSampling

We now describe FlashSampling from simplest to most practical form. The core algorithm is intentionally simple: maintain the largest perturbed score seen so far and its index.

3.1 Exact sampling via online Gumbel-Max

Given transformed logits $\widetilde{\boldsymbol{\ell}} \in (\mathbb{R} \cup \{-\infty\})^V$, exact sampling from $\mathrm{Cat}(\mathrm{softmax}(\widetilde{\boldsymbol{\ell}}))$ is:

$$i^\star = \underset{i \in [V]}{\mathrm{argmax}} \left(\widetilde{\ell}_i + g_i \right), \qquad g_i \sim \mathrm{Gumbel}(0, 1) \text{ i.i.d.}$$

Algorithm. Generate i.i.d. Gumbels, compute $s_i = \widetilde{\ell}_i + g_i$, and return $i^\star = \mathrm{argmax}_i\, s_i$. The computation can be performed online in a single pass that maintains only the current best score and its index. No softmax, no normalization constant, and no prefix sum are required (see Algorithm B.1 in the Appendix).

Systems implication. Sampling reduces to a single reduction over perturbed logits. This naturally fits GPU reductions and removes the extra normalization and prefix-sum work used by common softmax-based pipelines.

Simplicity. The online algorithm keeps only two running state variables per row: the current best perturbed score and the corresponding index. This simplicity is what makes fusion with the LM-head epilogue practical.

GPU parallelization. Each threadblock can process one contiguous vocabulary chunk, or *vocabulary tile*. The block computes perturbed scores for that chunk, keeps only the tile-local maximizer, and a small second-stage reduction selects the global maximizer across vocabulary tiles.

3.2 FlashSampling for LM-head sampling

We now consider the common case where logits are produced by GEMM:

$$\boldsymbol{Y} = \boldsymbol{H}\boldsymbol{W}^\top \in \mathbb{R}^{B \times V},$$

where $\boldsymbol{H} \in \mathbb{R}^{B \times D}$ are hidden states and $\boldsymbol{W} \in \mathbb{R}^{V \times D}$ are LM-head weights. We wish to sample one index per row from $\mathrm{Cat}(\mathrm{softmax}(\boldsymbol{Y}_{b,:}))$, possibly after deterministic transforms such as temperature scaling, additive bias, or masking.

Goal: avoid materializing $\boldsymbol{Y}$. FlashSampling performs sampling inside the matmul kernel and writes only one candidate per row and per vocabulary tile, never the full $[B, V]$ logits tensor:

- **Stage 1 (fused kernel):** compute one batch tile and one vocabulary tile on chip, apply deterministic transforms, add Gumbel noise, and keep the tile-local maximizer for each row.
- **Stage 2 (reduction):** reduce over vocabulary-tile candidates to obtain one global sample per row.

Algorithm 2 FlashSampling fused matmul-sample (two-stage): one candidate per row and per vocabulary tile, followed by reduction

Require: Hidden states $\boldsymbol{H} \in \mathbb{R}^{B\times D}$, LM-head weights $\boldsymbol{W} \in \mathbb{R}^{V\times D}$, temperature $\tau > 0$, optional mask/bias, RNG key
Ensure: Samples $\boldsymbol{i}^\star \in \{1,\ldots,V\}^B$

Stage 1 (fused kernel): for each batch tile $\mathcal{B}$ and vocabulary tile $\mathcal{T}_t$ in parallel

```
Initialize accumulator A^(t) ∈ R^{|B|×|T_t|} ← 0
for d_0 = 1, 1 + K_tile, ..., D do
    Load H_{B, d_0:d_0+K_tile−1} and W_{T_t, d_0:d_0+K_tile−1} into on-chip memory
    A^(t) ← A^(t) + H_{B, d_0:d_0+K_tile−1} (W_{T_t, d_0:d_0+K_tile−1})^⊤
end for
for each output element (b, i) ∈ B × T_t do
    ỹ_{b,i} ← transform(A^(t)_{b,i})                ▷ temperature, bias, mask
    Draw u_{b,i} ∈ (0, 1) and set g_{b,i} ← −log(−log u_{b,i})
    s_{b,i} ← ỹ_{b,i} + g_{b,i}
end for
for each row b ∈ B do
    (m_b^(t), j_b^(t)) ← argmax_{i∈T_t} s_{b,i}
    idx_b^(t) ← global vocabulary index corresponding to j_b^(t)
    Write (m_b^(t), idx_b^(t)) to HBM
end for
```

Stage 2 (reduction): for each row b

```
t* ← argmax_t m_b^(t)
i*_b ← idx_b^(t*)
return i*
```

In math notation:

$\boldsymbol{A}^{(t)} \in \mathbb{R}^{|\mathcal{B}|\times|\mathcal{T}_t|} \leftarrow 0$; $d_0 = 1, 1+K_{\text{tile}}, \ldots, D$; $\boldsymbol{H}_{\mathcal{B},\, d_0:d_0+K_{\text{tile}}-1}$ and $\boldsymbol{W}_{\mathcal{T}_t,\, d_0:d_0+K_{\text{tile}}-1}$; $\boldsymbol{A}^{(t)} \leftarrow \boldsymbol{A}^{(t)} + \boldsymbol{H}_{\mathcal{B},\, d_0:d_0+K_{\text{tile}}-1}\big(\boldsymbol{W}_{\mathcal{T}_t,\, d_0:d_0+K_{\text{tile}}-1}\big)^\top$; $(b,i) \in \mathcal{B}\times\mathcal{T}_t$; $\widetilde{y}_{b,i} \leftarrow \text{transform}\Big(A^{(t)}_{b,i}\Big)$; $u_{b,i} \in (0,1)$, $g_{b,i} \leftarrow -\log\big(-\log u_{b,i}\big)$; $s_{b,i} \leftarrow \widetilde{y}_{b,i} + g_{b,i}$; $b \in \mathcal{B}$; $(m_b^{(t)}, j_b^{(t)}) \leftarrow \text{argmax}_{i\in\mathcal{T}_t}\, s_{b,i}$; $\text{idx}_b^{(t)}$, $j_b^{(t)}$; $(m_b^{(t)}, \text{idx}_b^{(t)})$; $t^\star \leftarrow \text{argmax}_t\, m_b^{(t)}$; $i_b^\star \leftarrow \text{idx}_b^{(t^\star)}$; **return** $\boldsymbol{i}^\star$

Why the two-stage design is simple. The fused stage does all expensive work in the matmul epilogue. The second stage is only an argmax over a small candidate buffer of shape roughly $[B, \#\text{vocab tiles}]$. This design is easy to implement and already captures most of the benefit in the decode regime.

Why this avoids softmax. The algorithm never forms probabilities and never computes an explicit softmax. Exactness follows because it computes the same maximizer of the perturbed logits that a full Gumbel-Max pass would compute.

Tensor-parallel fusion. When the vocabulary is sharded across ranks, each rank can run the fused kernel on its local shard and return only small summaries rather than all local logits. In the grouped formulation below, these summaries are a local sample and a local log-mass. No $O(V)$ all-gather of logits is required.

RNG determinism. For reproducibility, RNG streams are indexed by the logical output position (b, i) using a counter-based RNG (e.g. Philox), so each random number is a deterministic function of a key and a counter. Uniform variates are mapped to the open interval $(0, 1)$ to avoid infinities in the Gumbel transform $g = -\log(-\log u)$.

Numerical precision. GEMM accumulation and perturbed scores are computed in FP32 for stability, even when inputs are FP16 or BF16. Gumbel noise is likewise generated in FP32 to avoid numerical error in the logarithms. The overhead is minor compared with the GEMM itself.

4 Theoretical Analysis of FlashSampling

This section separates the two exactness arguments used in the paper. The fused tiled kernel is exact *pathwise*: once perturbed scores are formed, the global maximizer is exactly the maximizer of the tile-local maxima. Grouped, online, and distributed variants are exact *in distribution*: they rely on hierarchical factorization through group log-masses.

4.1 Group-Gumbel-Max: hierarchical exact sampling

Partition $[V]$ into m disjoint groups $\{\mathcal{G}_k\}_{k=0}^{m-1}$; the groups need not have equal size. For any group with at least one finite transformed logit, define

$$L_k \;=\; \log \sum_{i \in \mathcal{G}_k} \exp(\widetilde{\ell}_i) \;=\; \operatorname{logsumexp}(\widetilde{\boldsymbol{\ell}}_{\mathcal{G}_k}).$$

If a group contains no finite transformed logit, then $L_k = -\infty$, the group has zero probability mass, and it can be skipped.

After discarding zero-mass groups, the categorical distribution factorizes as

$$\underbrace{\mathbb{P}(K = k)}_{\text{choose group}} \propto \exp(L_k), \qquad \underbrace{\mathbb{P}(I = i \mid K = k)}_{\text{choose within group}} \propto \exp(\widetilde{\ell}_i) \quad \text{for } i \in \mathcal{G}_k.$$

Thus exact sampling from the full categorical can be implemented by first choosing a group using the logits $\{L_k\}$ and then sampling within the chosen group.

Parallel FlashSampling. Suppose logits arise from a linear projection $\boldsymbol{y} = \boldsymbol{W}\boldsymbol{x}$, where $\boldsymbol{W} \in \mathbb{R}^{V \times D}$ and $\boldsymbol{x} \in \mathbb{R}^D$. Let $\boldsymbol{W}_{\mathcal{G}_k} \in \mathbb{R}^{|\mathcal{G}_k| \times D}$ be the block of rows indexed by group $\mathcal{G}_k$, so $\boldsymbol{y}_k = \boldsymbol{W}_{\mathcal{G}_k}\boldsymbol{x} \in \mathbb{R}^{|\mathcal{G}_k|}$ are the group logits. Parallel FlashSampling computes groups independently: each group with nonzero mass computes (i) an exact local sample $z_k \sim \operatorname{Cat}(\operatorname{softmax}(\boldsymbol{y}_k))$ and (ii) its group log-mass $L_k = \operatorname{logsumexp}(\boldsymbol{y}_k)$. The algorithm then samples $K \sim \operatorname{Cat}(\operatorname{softmax}(\boldsymbol{L}))$ and returns z_K mapped to its global index. This is exact by direct factorization.

Online FlashSampling. When memory is the primary constraint, FlashSampling can stream groups one at a time and maintain only a running log-mass and a running sample. Suppose the current running state is (L_{run}, z) and the next nonzero-mass group has log-mass L_k and exact local sample z_k. Define

$$L_{\text{new}} = \log\left(e^{L_{\text{run}}} + e^{L_k}\right).$$

Then replace z by z_k with probability

$$\frac{e^{L_k}}{e^{L_{\text{run}}} + e^{L_k}} = e^{L_k - L_{\text{new}}} = \frac{1}{1 + e^{L_{\text{run}} - L_k}},$$

and otherwise keep z. Section 4.4 proves that this binary merge rule preserves exactness by induction.

4.2 Distributed FlashSampling for tensor-parallel vocabularies

In tensor-parallel LM heads, the vocabulary dimension is sharded across n GPUs. Naively, each GPU computes local logits and then an all-gather concatenates the full V logits before sampling, incurring communication proportional to the vocabulary size per row. FlashSampling treats shards as groups: each rank returns (i) a local exact sample from its shard, if its shard has nonzero mass for that row, and (ii) the shard log-mass L_k. A final exact categorical sample over the shard log-masses chooses which rank provides the global sample. Communication therefore scales with the number of shards, not the number of vocabulary entries.

4.3 A unifying view: max-stability of grouped Gumbel perturbations

Group-Gumbel-Max and FlashSampling both rely on the same structural fact: *max* decomposes over partitions. For grouped variants we additionally use the max-stability of Gumbel perturbations.

Lemma 4.1 (Gumbel max-stability under grouping)**.** Let $\{g_i\}_{i=1}^{V}$ be i.i.d. Gumbel$(0,1)$ and let $\{\mathcal{G}_k\}_{k=0}^{m-1}$ be a partition of $[V]$. Assume each group under discussion contains at least one finite transformed logit. Define

$$M_k \;=\; \max_{i\in\mathcal{G}_k}(\widetilde{\ell}_i + g_i), \qquad I_k \;=\; \operatorname*{argmax}_{i\in\mathcal{G}_k}(\widetilde{\ell}_i + g_i), \qquad L_k \;=\; \log\sum_{i\in\mathcal{G}_k} e^{\widetilde{\ell}_i}.$$

Then:

1. $M_k \sim \text{Gumbel}(L_k, 1)$,
2. $\{M_k\}$ are independent across disjoint groups,
3. $\mathbb{P}(I_k = i) = e^{\widetilde{\ell}_i} / \sum_{j\in\mathcal{G}_k} e^{\widetilde{\ell}_j}$ for $i \in \mathcal{G}_k$.

Proof. For any real t,

$$\mathbb{P}(M_k \le t) = \prod_{i\in\mathcal{G}_k} \mathbb{P}(g_i \le t - \widetilde{\ell}_i) = \prod_{i\in\mathcal{G}_k} \exp\big(-e^{-(t-\widetilde{\ell}_i)}\big) = \exp\Big(-e^{-(t-L_k)}\Big),$$

which is the CDF of Gumbel$(L_k, 1)$. Independence follows because the groups are disjoint and the underlying Gumbels are independent. The within-group argmax probabilities are exactly the Gumbel-Max trick applied to the restricted transformed logits. □

Consequence. For grouped variants, selecting a group by $\operatorname{argmax}_k M_k$ is equivalent in distribution to applying Gumbel-Max directly to the group logits $\{L_k\}$. The outer group sample may therefore use fresh independent Gumbels, or it may reuse explicitly computed group maxima. For the fused two-stage kernel in Algorithm 2, exactness does *not* rely on max-stability: once the perturbed scores $x_i = \widetilde{\ell}_i + g_i$ have been formed, exactness is simply the deterministic identity

$$\max_i x_i = \max_t \max_{i\in\mathcal{T}_t} x_i.$$

4.4 Exactness of Group-Gumbel-Max

The correctness of grouped FlashSampling rests on two facts: exact group factorization, and the binary merge rule used by the online variant.

Lemma 4.2 (Exact group factorization)**.** Let $[V]$ be partitioned into groups $\{\mathcal{G}_k\}_{k=0}^{m-1}$, and discard any zero-mass groups. Define $L_k = \log \sum_{i \in \mathcal{G}_k} \exp(\widetilde{\ell}_i)$. If we sample $K \sim \mathrm{Cat}(\mathrm{softmax}(\boldsymbol{L}))$ and then sample $I \mid (K = k) \sim \mathrm{Cat}(\mathrm{softmax}(\widetilde{\boldsymbol{\ell}}_{\mathcal{G}_k}))$, the marginal distribution of I equals $\mathrm{Cat}(\mathrm{softmax}(\widetilde{\boldsymbol{\ell}}))$.

Proof. For any $i \in \mathcal{G}_k$,

$$\mathbb{P}(I = i) = \mathbb{P}(K = k)\,\mathbb{P}(I = i \mid K = k) = \frac{e^{L_k}}{\sum_s e^{L_s}} \cdot \frac{e^{\widetilde{\ell}_i}}{\sum_{j \in \mathcal{G}_k} e^{\widetilde{\ell}_j}} = \frac{e^{\widetilde{\ell}_i}}{\sum_{j=1}^{V} e^{\widetilde{\ell}_j}}.$$

□

Lemma 4.3 (Binary merge rule)**.** Let $A, B \subseteq [V]$ be disjoint and suppose both have nonzero mass. Define

$$L_A = \log \sum_{i \in A} e^{\widetilde{\ell}_i}, \qquad L_B = \log \sum_{i \in B} e^{\widetilde{\ell}_i}.$$

Suppose $Z_A \sim \mathrm{Cat}(\mathrm{softmax}(\widetilde{\boldsymbol{\ell}}_A))$, $Z_B \sim \mathrm{Cat}(\mathrm{softmax}(\widetilde{\boldsymbol{\ell}}_B))$, and an independent Bernoulli choice selects B with probability $e^{L_B}/(e^{L_A} + e^{L_B})$. Returning Z_B when B is selected and Z_A otherwise yields an exact sample from $\mathrm{Cat}(\mathrm{softmax}(\widetilde{\boldsymbol{\ell}}_{A \cup B}))$.

Proof. For any $i \in A$,

$$\mathbb{P}(Z = i) = \mathbb{P}(\text{choose } A)\,\mathbb{P}(Z_A = i) = \frac{e^{L_A}}{e^{L_A} + e^{L_B}} \cdot \frac{e^{\widetilde{\ell}_i}}{\sum_{j \in A} e^{\widetilde{\ell}_j}} = \frac{e^{\widetilde{\ell}_i}}{\sum_{j \in A \cup B} e^{\widetilde{\ell}_j}}.$$

The same calculation for $i \in B$ gives

$$\mathbb{P}(Z = i) = \frac{e^{L_B}}{e^{L_A} + e^{L_B}} \cdot \frac{e^{\widetilde{\ell}_i}}{\sum_{j \in B} e^{\widetilde{\ell}_j}} = \frac{e^{\widetilde{\ell}_i}}{\sum_{j \in A \cup B} e^{\widetilde{\ell}_j}}.$$

Hence $Z \sim \mathrm{Cat}(\mathrm{softmax}(\widetilde{\boldsymbol{\ell}}_{A \cup B}))$. □

Theorem 4.4 (Exactness of hierarchical FlashSampling)**.** Algorithms B.2, B.3, and B.4 return an exact sample from $\mathrm{Cat}(\mathrm{softmax}(\widetilde{\boldsymbol{\ell}}))$.

Proof. For the parallel and distributed variants, Lemma 4.2 shows that it suffices to sample the group or shard index from logits $\{L_k\}$ and then sample within the chosen group; both steps are exact.

For the online variant, initialize with an exact sample from the first nonzero-mass group. Each subsequent update merges the current union with the next nonzero-mass group using Lemma 4.3. An induction over the streamed groups therefore yields an exact sample from the full categorical distribution. □

4.5 Exactness of tile-wise FlashSampling reduction

FlashSampling also relies on a simpler structural lemma: the global maximum equals the maximum of the tile-local maxima.

Lemma 4.5 (Max over vocabulary tiles)**.** Let $\{x_i\}_{i=1}^{V}$ be real numbers and let $\{\mathcal{T}_s\}_{s=0}^{n_{\text{tile}}-1}$ be a partition of $[V]$ into vocabulary tiles. For each tile, define

$$m_s = \max_{i \in \mathcal{T}_s} x_i, \qquad \widehat{\imath}_s \in \operatorname*{argmax}_{i \in \mathcal{T}_s} x_i,$$

where $\widehat{\imath}_s$ is a global index in $\mathcal{T}_s$. Then

$$\max_{i \in [V]} x_i = \max_s m_s.$$

Moreover, for any $s^\star \in \operatorname{argmax}_s m_s$, the chosen index $\widehat{\imath}_{s^\star}$ is a global maximizer. Conversely, every global maximizer lies in some tile $s^\star \in \operatorname{argmax}_s m_s$.

Proof. The identity for the maximum value is immediate:

$$\max_{i \in [V]} x_i = \max_s \max_{i \in \mathcal{T}_s} x_i = \max_s m_s.$$

If $s^\star \in \operatorname{argmax}_s m_s$, then $x_{\widehat{\imath}_{s^\star}} = m_{s^\star} = \max_i x_i$, so $\widehat{\imath}_{s^\star}$ is a global maximizer. Conversely, if $i^\star$ is any global maximizer, then its tile $s^\star$ satisfies $m_{s^\star} = x_{i^\star} = \max_i x_i$, hence $s^\star \in \operatorname{argmax}_s m_s$. $\square$

Applying Lemma 4.5 to $x_i = \widetilde{\ell}_i + g_i$ justifies the two-stage fused design in Algorithm 2. Because the Gumbel variables are continuous, the global maximizer is unique almost surely, so the tile-wise reduction returns exactly the same index as a full row-wise argmax with probability one.

4.6 Top-k, Nucleus Sampling, and Masking

Practical decoding often uses truncated supports, and the tiled structure of FlashSampling naturally accommodates most of them.

- **Top-k:** The Group-Gumbel-Max decomposition extends directly to top-k. Each tile computes top-k candidates locally (logits and indices), and a second stage reduces all per-tile candidates into a global top-k. Sampling from the final k candidates can be done with multinomial or Gumbel-Max sampling.
- **Top-p (nucleus):** Unlike top-k, nucleus sampling requires a global softmax followed by a sorted cumulative sum, neither of which decomposes into independent tile-local work. However, top-p can be applied *after* top-k on the reduced candidate set of only k elements, where softmax, sorting, and cumulative summation are negligible. This sequential top-k-then-top-p strategy is used in practice by vLLM[1,2], FlashInfer[3], and other SOTA top-k top-p algorithms (Park et al., 2026).

[1] https://github.com/vllm-project/vllm/blob/v0.16.0/vllm/v1/sample/ops/topk_topp_sampler.py#L264-L279

[2] https://github.com/vllm-project/vllm/blob/v0.16.1rc0/vllm/v1/sample/ops/topk_topp_triton.py#L956

[3] https://github.com/flashinfer-ai/flashinfer/blob/v0.6.3/flashinfer/sampling.py#L1069-L1072

- **Masking:** Forbidden indices (e.g. banned tokens, grammar constraints) are supported by setting their logits to $-\infty$ before perturbation, which preserves exactness over the restricted support.

While the FlashSampling theory allows integrating these sampling strategies, we leave the implementation to future work.

4.7 Cost model: bandwidth, kernels, and overhead

We outline a simple model to reason about speedups.

Materialized baseline (lower bound). For a BF16 baseline that materializes logits, the GEMM must at least read $\boldsymbol{W}$ and $\boldsymbol{H}$ and write $\boldsymbol{Y}$ once; sampling must then read $\boldsymbol{Y}$ at least once again. An optimistic lower bound on arithmetic intensity is therefore

$$I_{\mathrm{mat}}(B) \approx \frac{2BVD}{2(VD+BD+2BV)} = \frac{BVD}{VD+BD+2BV} \qquad \text{FLOP/byte},$$

where the denominator counts mandatory BF16 traffic only. Real softmax-based samplers usually make more than one pass over the materialized logits, so the true baseline intensity is lower.

Fused matmul + sampling. If sampling is fused into the GEMM epilogue so that the logits write and reread are removed, then, up to lower-order terms from the small candidate buffer,

$$I_{\mathrm{fused}}(B) \approx \frac{2BVD}{2(VD+BD)} = \frac{BV}{V+B} \qquad \text{FLOP/byte}.$$

Thus fusion raises the effective arithmetic intensity.

Incremental traffic saved by fusion. Relative to a fused kernel, any materialized baseline incurs at least one write and one reread of the $[B,V]$ logits tensor. In BF16 this minimal extra traffic is $4BV$ bytes. Compared with the mandatory LM-head weight read of $2VD$ bytes, the extra fraction is

$$\frac{4BV}{2VD} = \frac{2B}{D}.$$

For the small configuration ($D = 4096$), this ratio is 0.049% at $B = 1$, 3.125% at $B = 64$, and 6.25% at $B = 128$. Thus raw logits-byte savings alone are too small to explain the largest measured speedups. The main gains come from eliminating extra sampling kernels, global-memory round-trips through those kernels, and their launch and synchronization overhead. In the memory-bandwidth-bound decode regime, these extra kernels are pure overhead.

At $B = 1$ on the small configuration, the minimal avoided logits round-trip is

$$4BV = 4 \cdot 1 \cdot 151{,}936 = 607{,}744 \text{ bytes} \approx 0.608 \text{ MB}.$$

At 8 TB/s, this corresponds to only 7.6×10^{-5} ms. The observed latency gap therefore cannot be explained by raw HBM bandwidth alone.

5 Experiments

We evaluate FlashSampling at two levels: kernel-level microbenchmarks that isolate fused matmul-plus-sample across four GPU architectures, and end-to-end vLLM integration that measures autoregressive decode latency. All benchmarks use the open-source FlashSampling Triton implementation (Ruiz, 2026).

5.1 Setup

Hardware. Kernel microbenchmarks are run on four NVIDIA GPUs spanning two architecture generations. Table 2 summarizes their specifications. All GPUs are provisioned via Modal cloud.

Table 2 GPU specifications. Peak BF16 TFLOP/s are dense (without structured sparsity), since the LM-head matmul is a dense GEMM. The ops:byte ratio contextualizes the crossover between bandwidth- and compute-limited regimes, although the exact crossover is kernel-dependent.

	H100	**H200**	**B200**	**B300**
Architecture	Hopper	Hopper	Blackwell	Blackwell
HBM capacity (GB)	80	141	192	288
HBM bandwidth (TB/s)	3.35	4.8	8.0	8.0
Peak BF16 dense (TFLOP/s)	989	989	2,250	2,250
Ops:byte ratio	295	206	281	281

Software. PyTorch 2.10.0, CUDA 13.0, Triton 3.6, and FlashInfer 0.6.3. All kernels are warmed up for 25 iterations before timing.

Workload configuration. The main text focuses on the decode-centric configuration

$$D = 4{,}096, \qquad V = 151{,}936,$$

which matches models such as Qwen3-8B and Qwen3-235B-A22B MoE. We sweep batch sizes $B \in \{1, 2, 4, 8, 16, 32, 64, 128, 256\}$. Additional results for a larger configuration show the same qualitative trends (Appendix A).

Baselines.

1. **PyTorch Pipeline.** This baseline materializes the logits using a matmul (cuBLAS), followed by sampling with softmax and multinomial. We apply `torch.compile` to it, improving speed by 5% and 15% compared to PyTorch eager.
2. **FI1 (FlashInfer top-k/top-p).** `top_k_top_p_sampling_from_logits`[4], a sampling kernel used by vLLM for top-k/top-p decode (Ye et al., 2025). Logits are also materialized using cuBLAS.
3. **FI2 (FlashInfer Gumbel-Max).** `sampling_from_logits`, FlashInfer's exact Gumbel-Max sampler on pre-materialized logits (Ye et al., 2025). Logits materialized using cuBLAS.

[4] `https://docs.flashinfer.ai/api/sampling.html`

5.2 Standalone logits sampling

Standalone FlashSampling applies Gumbel-Max to pre-materialized logits. This is algorithmically close to FI2, which also uses Gumbel-Max on materialized logits. We therefore focus on the fused setting, which is the primary systems contribution: FlashSampling's advantage comes from eliminating the logits materialization and the sampling pass.

5.3 Fused matmul and sampling

Table 3 reports FlashSampling speedups relative to the three baselines on the main configuration (D=4096, V=151k). All numbers are median latency over 100 timed iterations.

Table 3 FlashSampling speedup vs. three baselines on the main configuration (D=4096, V=151k). Values > 1 indicate FlashSampling is faster; bold marks the peak per GPU within each baseline. FI1: FlashInfer top-k/top-p kernel. FI2: FlashInfer Gumbel-Max kernel.

	vs. PyTorch pipeline				*vs. FI1* (top-k/top-p)				*vs. FI2* (Gumbel-Max)			
B	H100	H200	B200	B300	H100	H200	B200	B300	H100	H200	B200	B300
1	1.22	1.24	1.36	1.40	**1.30**	1.29	1.37	2.06	**1.15**	**1.17**	**1.25**	**1.25**
2	1.23	1.25	1.38	1.38	1.25	**1.37**	1.46	2.05	**1.15**	**1.17**	1.24	1.23
4	1.21	1.22	1.37	1.39	**1.30**	1.35	**1.48**	1.94	1.14	1.14	1.24	1.24
8	1.22	1.25	1.41	1.35	**1.30**	**1.37**	1.47	2.24	1.13	1.14	1.22	1.23
16	1.25	1.28	1.41	1.43	1.29	1.36	**1.48**	**2.30**	1.13	1.13	1.22	1.23
32	1.29	1.30	1.39	1.39	1.27	1.32	1.36	2.10	1.11	1.08	1.14	1.14
64	1.40	**1.33**	1.36	1.38	1.29	1.23	1.23	1.90	1.10	1.01	1.03	1.04
128	**1.52**	1.26	**1.48**	**1.48**	1.25	1.04	1.17	1.72	1.03	0.84	0.96	1.00
256	1.23	1.06	1.31	1.33	1.00	0.89	1.06	1.63	0.77	0.70	0.85	0.89

Key observations.

1. **FlashSampling is consistently faster in the decode regime.** For $B \leq 64$, FlashSampling is faster than all three baselines on all four GPUs. In this regime, the peak speedup vs. the PyTorch pipeline is 1.43× and the peak speedup vs. FI1 is 2.30×.
2. **The gain is primarily from fusion.** Speedups over FI2 are smaller than speedups over the PyTorch pipeline or FI1 because FI2 already uses Gumbel-Max. The remaining gain therefore comes mainly from eliminating logits materialization and sampling overhead (Section 5.4).
3. **The advantage narrows at larger batch sizes.** As batch size grows, GEMM efficiency matters more and the workload becomes less dominated by memory-bandwidth-bound postprocessing. The larger-configuration appendix shows the same qualitative trend, with the crossover occurring earlier.

5.4 Interpreting the batch-size trend

The cost model in Section 4.7 showed that HBM savings from avoiding the logits write and reread alone are small (≤6% of traffic). Figure 4 reveals a larger effect: the baselines' separate sampling

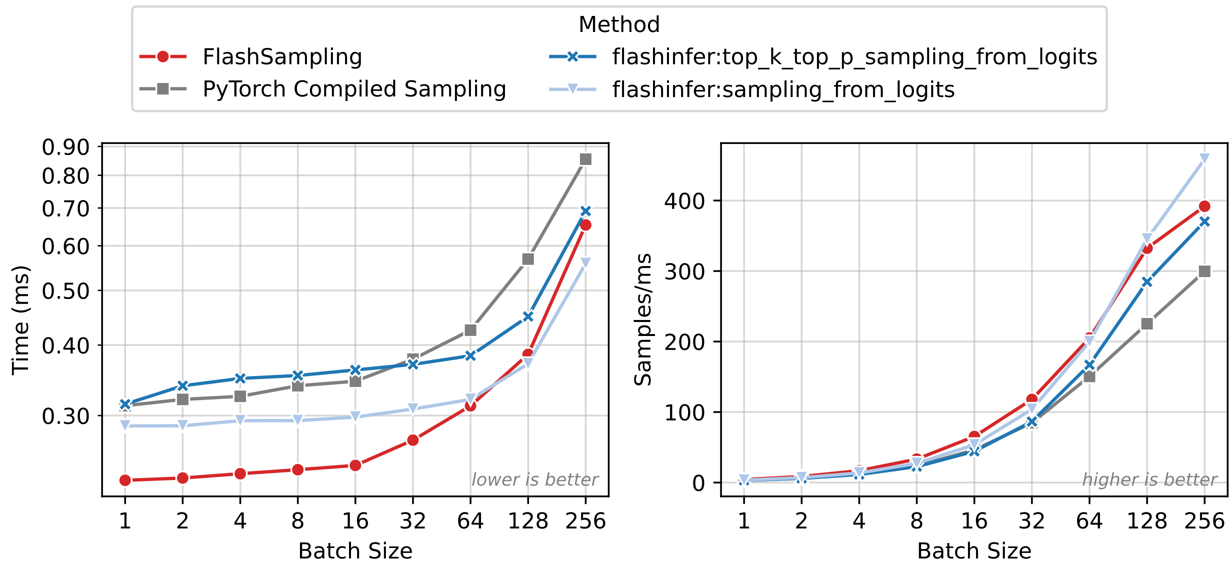

Figure 2 Absolute latency (ms) vs. batch size on B200 for the main configuration. FlashSampling is faster than all baselines for $B \leq 64$. At larger batch sizes, the advantage narrows as GEMM efficiency becomes more important.

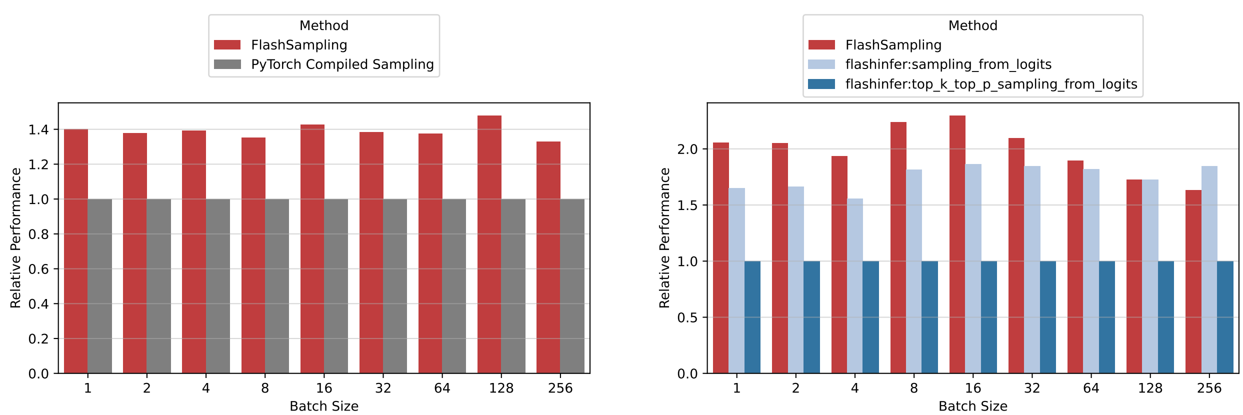

Figure 3 Relative performance on B300 for the main configuration. Left: FlashSampling vs. the PyTorch pipeline (baseline = 1). Right: FlashSampling vs. FlashInfer FI1 and FI2 (baseline = 1). FlashSampling is faster than the PyTorch pipeline across all shown batch sizes, faster than FI1 throughout, and faster than FI2 in the decode regime.

kernels are expensive and their runtime grows steeply with batch size, while FlashSampling absorbs sampling into the matmul at negligible cost (Table 4: 2–6% of kernel time). Eliminating these separate kernels is the primary source of speedup. The advantage narrows at large batch sizes because FlashSampling's Triton matmul becomes less efficient than cuBLAS (right panel), partially offsetting the sampling savings. Profiling was performed on an RTX 3090 using Nsight Compute and Proton.

Table 4 Sampling as a percentage of total kernel time. A high fraction spent on sampling rather than matmul is a indicator of inefficient sampling implementation. FlashSampling's sampling fraction stays low because it is fused into the matmul epilogue; the baselines' fraction grows with batch size B. Bold marks the highest sampling fraction for each method.

	FlashSampling		*PyTorch pipeline*		*FI2* (Gumbel-Max)	
B	matmul (%)	sampl. (%)	matmul (%)	sampl. (%)	matmul (%)	sampl. (%)
1	97.7	2.3	93.7	6.3	94.7	5.3
16	97.7	2.3	87.1	12.9	93.4	6.6
64	93.6	6.0	71.3	**28.7**	88.6	11.4
256	93.4	**6.2**	73.1	26.9	88.2	**11.8**

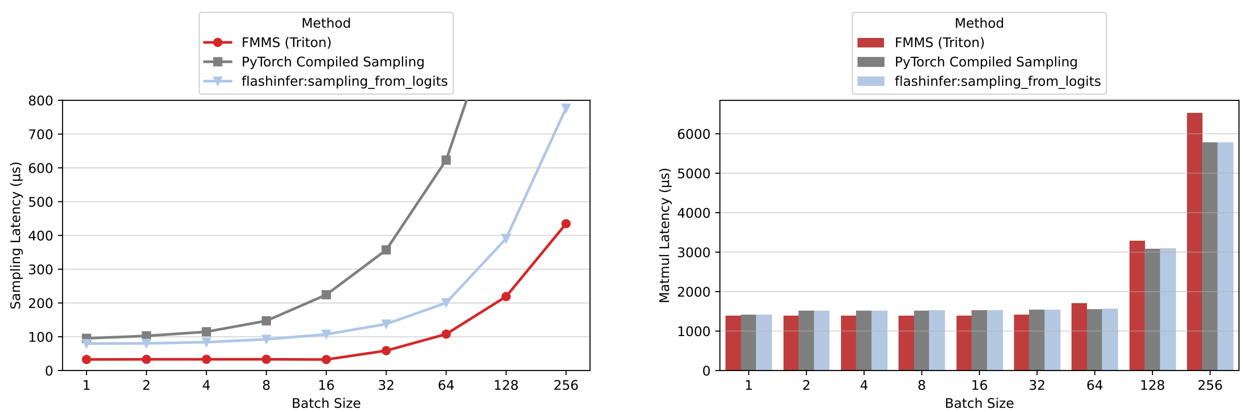

Figure 4 Sampling runtime (left) and matmul runtime (right) in μs vs. batch size. Lower is better.

5.5 End-to-end vLLM evaluation

We integrate FlashSampling into vLLM (Kwon et al., 2023) by replacing the decode-time LM-head projection and sampler with a single `fused_mm_sample_triton` call. Prefill uses the standard path because logits are needed there for prompt log-probabilities. Experiments run on a single B200 GPU with four models spanning a range of sizes and architectures.

For each model and concurrency level, we run 5 trials of 256 requests from the ShareGPT dataset, measuring time per output token (TPOT). We pair baseline and FlashSampling runs by trial number and report the median TPOT across trials for robustness against outliers.

Key observations.

Table 5 TPOT speedup (%) on B200 with vLLM: $1 - \text{FlashSampling/baseline}$ (median $\pm$ std over 5 paired trials of 256 requests). B is the maximum number of concurrent requests. Bold marks the peak per model. FlashSampling gains decrease with model size as attention and FFN dominate decode time.

B	Qwen3-1.7B	Qwen3-8B	Qwen3-32B	gpt-oss-120b
1	$10.8 \pm 0.3\,\%$	$2.8 \pm 0.9\,\%$	$\mathbf{1.9} \pm 1.4\,\%$	$0.3 \pm 0.8\,\%$
2	$12.8 \pm 0.6\,\%$	$\mathbf{6.9} \pm 1.7\,\%$	$-1.8 \pm 0.5\,\%$	$1.7 \pm 0.5\,\%$
4	$14.5 \pm 0.4\,\%$	$3.7 \pm 0.1\,\%$	$1.6 \pm 0.0\,\%$	$\mathbf{2.4} \pm 0.3\,\%$
8	$15.2 \pm 0.4\,\%$	$3.6 \pm 0.4\,\%$	$1.1 \pm 0.1\,\%$	$2.1 \pm 0.1\,\%$
16	$17.7 \pm 9.0\,\%$	$3.0 \pm 0.4\,\%$	$1.1 \pm 0.1\,\%$	$1.8 \pm 0.4\,\%$
32	$9.7 \pm 5.1\,\%$	$4.4 \pm 1.7\,\%$	$1.3 \pm 0.3\,\%$	$1.5 \pm 2.0\,\%$
64	$\mathbf{18.7} \pm 6.8\,\%$	$5.2 \pm 2.2\,\%$	$1.2 \pm 0.3\,\%$	$1.6 \pm 0.8\,\%$

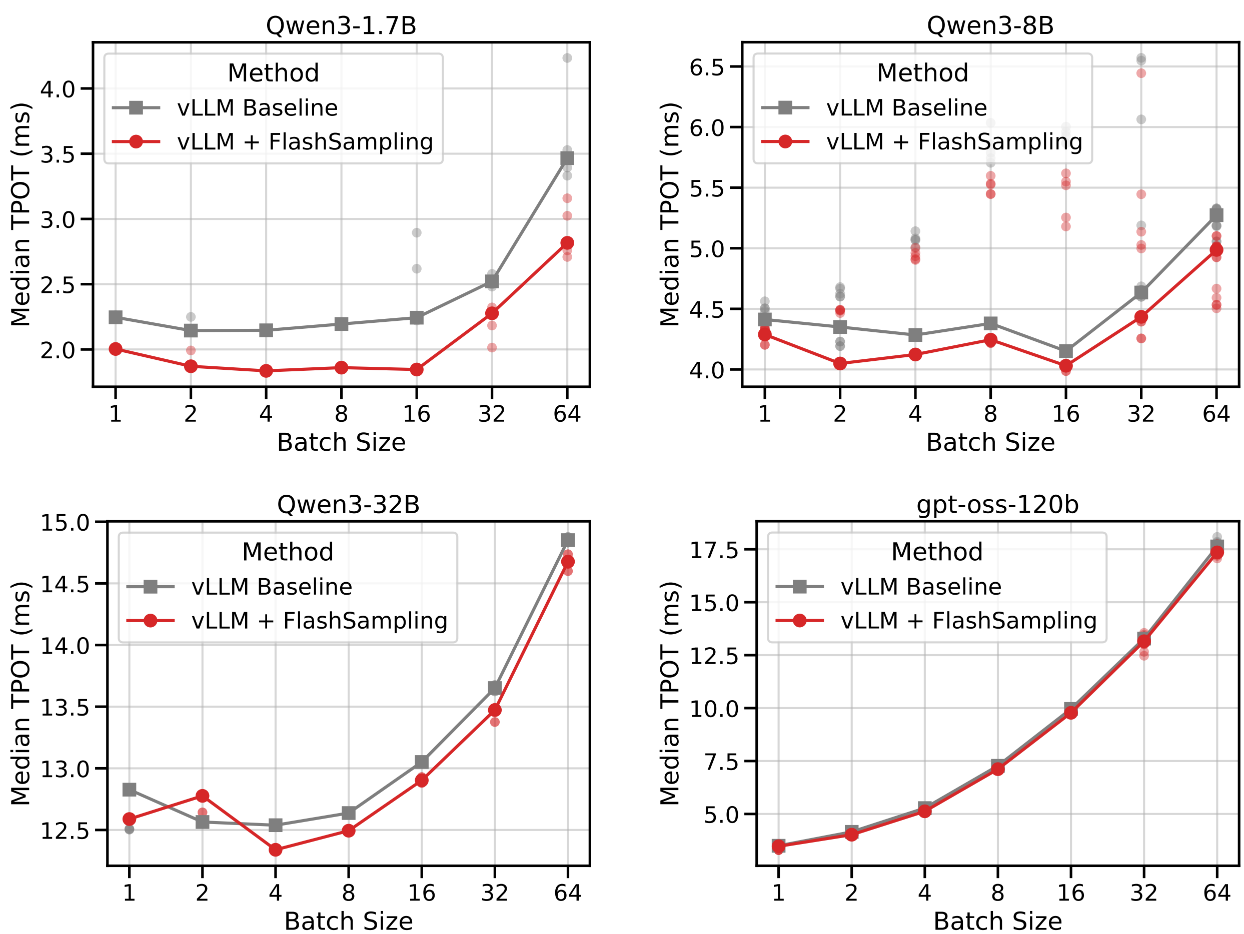

Figure 5 TPOT vs. concurrency on B200 for all four models. Top row: Qwen3-1.7B (up to 19% reduction) and Qwen3-8B (roughly 3–7%). Bottom row: Qwen3-32B and gpt-oss-120b, where gains are smaller because attention and FFN dominate decode time.

1. **Largest gains appear on the smallest model.** Qwen3-1.7B sees 11–19% TPOT reduction because its LM head is a larger fraction of total per-token decode cost.
2. **Gains decrease with model size.** For Qwen3-32B and gpt-oss-120b, attention and FFN layers dominate decode time, so overall TPOT improvements are smaller and can be noisy.
3. **No measurable quality difference on GSM8K.** On Qwen3-8B, FlashSampling achieves 89.4% accuracy versus 89.6% for the baseline on 500 GSM8K problems. A bootstrap test yields $p = 0.776$, so we do not detect a statistically significant difference. This is consistent with exact sampling.

6 Related work

Gumbel-Max and extensions. The Gumbel-Max trick for exact categorical sampling dates to Gumbel (1954) and was formalized in modern machine learning terms by Maddison et al. (2014). Jang et al. (2017) introduced the Gumbel-Softmax relaxation for differentiable discrete sampling, which is complementary to our focus on exact sampling. Huijben et al. (2022) review the broader Gumbel-Max literature, and Qi et al. (2020) study fast generation of large numbers of Gumbel variables. FlashSampling contributes a systems-oriented hierarchical decomposition for exact online and distributed sampling.

Kernel fusion for memory efficiency. FlashAttention (Dao et al., 2022) showed that avoiding materialization of the attention matrix can substantially reduce HBM traffic. Cut Your Losses (Wijmans et al., 2025) applies a similar idea to training-time cross-entropy by avoiding materialization of large logits tensors. FlashSampling extends this "flash" methodology from training-time loss computation to inference-time exact sampling.

LLM sampling implementations. FlashInfer (Ye et al., 2025) provides optimized GPU kernels for attention and sampling in LLM serving, including an exact Gumbel-Max sampler on materialized logits. vLLM (Kwon et al., 2023) uses FlashInfer's sampling kernels in its decode path. FlashSampling differs in the key systems choice: it fuses the LM-head matmul with sampling and therefore removes the logits round-trip to HBM.

Approximate and orthogonal methods. Approximate training-time methods such as sampled softmax reduce large-vocabulary cost by approximation (Rawat et al., 2019). In contrast, FlashSampling is exact. FlashSampling is also orthogonal to higher-level decode optimizations such as speculative decoding (Leviathan et al., 2023): those methods reduce the number of decode steps, while FlashSampling reduces the cost of each step when sampling is required.

7 Conclusion

We presented **FlashSampling**, a simple fused design for exact categorical sampling that avoids materializing the $[B, V]$ logits tensor in HBM. The key ideas are straightforward: exact sampling does not require an explicit softmax, the fused tiled kernel is exact by argmax decomposition over vocabulary tiles, and grouped log-masses yield exact online and distributed variants. Empirically, FlashSampling is most effective in the memory-bandwidth-bound decode regime, where it removes pure sampling overhead and turns sampling into a lightweight epilogue.

Acknowledgement

We sincerely thank Yongye Zhu, Zhuoqing Song, and Mayank Mishra for their helpful discussions and constructive feedback. We used large language models to assist in polishing the writing of this work.

References

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*, volume 35, 2022.

Emil Julius Gumbel. Statistical theory of extreme values and some practical applications. *National Bureau of Standards Applied Mathematics Series*, 33, 1954.

Iris AM Huijben, Wouter Kool, Max B Paulus, and Ruud JG Van Sloun. A review of the gumbel-max trick and its extensions for discrete stochasticity in machine learning. *IEEE transactions on pattern analysis and machine intelligence*, 45(2):1353–1371, 2022.

Eric Jang, Shixiang Gu, and Ben Poole. Categorical reparameterization with Gumbel-softmax. In *International Conference on Learning Representations*, 2017.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 611–626, 2023.

Yaniv Leviathan, Matan Kalman, and Yossi Matias. Fast inference from transformers via speculative decoding. In *International Conference on Machine Learning*, pages 19274–19286, 2023.

Chris J. Maddison, Daniel Tarlow, and Tom Minka. A* sampling. In *Advances in Neural Information Processing Systems*, volume 27, 2014.

NVIDIA. NVIDIA H100 tensor core GPU architecture. Technical report, 2022. URL `https://resources.nvidia.com/en-us-data-center-overview-mc/en-us-data-center-overview/gtc22-whitepaper-hopper`. Accessed: 2026-03-04.

NVIDIA. NVIDIA H100 tensor core GPU datasheet. Technical report, 2024. URL `https://www.megware.com/fileadmin/user_upload/LandingPage%20NVIDIA/nvidia-h100-datasheet.pdf`. Accessed: 2026-03-04.

Jongseok Park, Sunga Kim, Alvin Cheung, and Ion Stoica. Qrita: High-performance top-k and top-p algorithm for gpus using pivot-based truncation and selection, 2026. URL `https://arxiv.org/abs/2602.01518`.

Yiyan Qi, Pinghui Wang, Yuanming Zhang, Junzhou Zhao, Guangjian Tian, and Xiaohong Guan. Fast generating a large number of Gumbel-max variables. In *Proceedings of The Web Conference*, pages 2006–2012, 2020.

Ankit Singh Rawat, Jiecao Chen, Felix Xinnan X. Yu, Ananda Theertha Suresh, and Sanjiv Kumar. Sampled softmax with random Fourier features. In *Advances in Neural Information Processing Systems*, volume 32, 2019.

Tomas Ruiz. fmms-kernel: FMMS kernel: Fused matrix-multiplication + sampling. GitHub repository, 2026. URL `https://github.com/tomasruizt/fmms-kernel`. Triton implementation of fused matmul + Gumbel-Max sampling.

Erik Wijmans, Brody Koh, Roei Herzig, Jitendra Jain, Jianwei Zhu, Saurabh Kapoor, and Ross Girshick. Cut your losses in large-vocabulary language models. *arXiv preprint arXiv:2411.09009*, 2025.

Zihao Ye, Lequn Chen, Ruihang Lai, Wuwei Lin, Yineng Zhang, Stephanie Wang, Tianqi Chen, Baris Kasikci, Vinod Grover, Arvind Krishnamurthy, et al. Flashinfer: Efficient and customizable attention engine for llm inference serving. *Proceedings of Machine Learning and Systems*, 7, 2025.

Appendix

A Additional kernel results for the large configuration

For completeness, Table 6 reports the larger-configuration kernel results deferred from the main text. The same qualitative pattern appears: FlashSampling is strongest in the small-batch decode regime, while the advantage narrows once the workload becomes more GEMM-efficiency dominated.

Table 6 FlashSampling speedup vs. three baselines on the larger configuration (D=8192, V=128k). Values > 1 indicate FlashSampling is faster; bold marks the peak per GPU within each baseline. At $B \geq 128$ the advantage narrows and cuBLAS GEMM efficiency becomes increasingly important.

	vs. PyTorch pipeline				*vs. FI1* (top-k/top-p)				*vs. FI2* (Gumbel-Max)			
B	H100	H200	B200	B300	H100	H200	B200	B300	H100	H200	B200	B300
1	1.22	1.26	**1.35**	**1.34**	1.18	**1.25**	**1.27**	1.57	**1.13**	**1.13**	**1.15**	**1.14**
2	1.22	1.23	1.31	1.33	**1.22**	1.21	1.24	1.65	**1.13**	1.12	**1.15**	**1.14**
4	1.20	1.22	1.27	1.23	1.19	1.22	1.21	1.66	1.12	1.11	1.07	1.08
8	1.21	1.22	1.24	1.26	1.20	1.22	1.19	**1.67**	1.12	1.10	1.07	1.08
16	1.21	1.22	1.26	1.28	1.20	1.22	1.21	**1.67**	1.11	1.10	1.07	1.08
32	1.21	1.23	1.25	1.29	1.18	1.20	1.20	1.64	1.10	1.09	1.06	1.07
64	**1.28**	**1.28**	1.27	1.23	1.20	1.20	1.18	1.55	1.11	1.08	1.06	1.03
128	1.13	0.93	0.97	0.96	1.01	0.81	0.86	1.09	0.90	0.72	0.77	0.76
256	0.87	0.78	0.74	0.71	0.79	0.75	0.68	0.89	0.67	0.63	0.57	0.57

B FlashSampling Algorithm Pseudocode

This appendix collects detailed pseudocode for the FlashSampling variants described in the main text.

Streaming Gumbel-Max (standalone logits). Algorithm B.1 presents the basic one-pass streaming Gumbel-Max sampler over pre-materialized logits.

Algorithm B.1 Gumbel-Max sampling (standalone logits): streaming argmax over perturbed logits

Require: Logits $\boldsymbol{\ell} \in \mathbb{R}^V$, RNG state
Ensure: Sample index $i^\star \in \{1, \ldots, V\}$

$m \leftarrow -\infty$, $i^\star \leftarrow 1$
for $i = 1$ **to** V **do**
  $g \leftarrow \textsc{Gumbel}(0, 1)$ ▷ via $g = -\log(-\log u)$, $u \sim \mathrm{Unif}(0, 1)$
  $s \leftarrow \ell_i + g$
  if $s > m$ **then**
    $m \leftarrow s$, $i^\star \leftarrow i$
  end if
end for
return $i^\star$

Parallel Group-Gumbel-Max. Algorithm B.2 extends streaming Gumbel-Max to a group-parallel setting where each group is processed by an independent threadblock.

Algorithm B.2 FlashSampling (parallel): Group-Gumbel-Max over groups

Require: Input $\boldsymbol{x} \in \mathbb{R}^d$, weight matrix $\boldsymbol{W} \in \mathbb{R}^{d \times V}$, group size g (so $V = mg$), RNG state
Ensure: Sample index $z \in \{1, \ldots, V\}$ and optional log-normalizer $\ell_Z = \text{logsumexp}(\boldsymbol{y})$

for $k = 0$ **to** $m - 1$ **in parallel do**
    $\boldsymbol{y}_k \leftarrow \boldsymbol{W}_k^\top \boldsymbol{x} \in \mathbb{R}^g$
    $z_k \leftarrow \text{argmax}_{j \in [g]} \left(y_{k,j} - \log(-\log u_{k,j}) \right)$ ▷ $u_{k,j} \sim \text{Unif}(0, 1)$
    $L_k \leftarrow \text{logsumexp}(\boldsymbol{y}_k)$
end for
$k^\star \leftarrow \text{argmax}_{k \in [m]} \left(L_k - \log(-\log \bar{u}_k) \right)$ ▷ $\bar{u}_k \sim \text{Unif}(0, 1)$
$z \leftarrow k^\star g + z_{k^\star}$ ▷ map group-local index to global vocabulary index
$\ell_Z \leftarrow \text{logsumexp}([L_0, \ldots, L_{m-1}])$ ▷ optional
return (z, ℓ_Z)

Sequential/online Group-Gumbel-Max. Algorithm B.3 provides a memory-efficient variant that streams groups one at a time.

Algorithm B.3 FlashSampling (sequential/online): streaming Group-Gumbel-Max with $O(g)$ working memory

Require: Input $\boldsymbol{x} \in \mathbb{R}^d$, weight matrix $\boldsymbol{W} \in \mathbb{R}^{d \times V}$, group size g (so $V = mg$), RNG state
Ensure: Sample index $z \in \{1, \ldots, V\}$ and optional log-normalizer ℓ_Z

Initialize with the first group.
$\boldsymbol{y}_0 \leftarrow \boldsymbol{W}_0^\top \boldsymbol{x} \in \mathbb{R}^g$
$L_0 \leftarrow \text{logsumexp}(\boldsymbol{y}_0)$
$z_0 \leftarrow \text{argmax}_{j \in [g]} \left(y_{0,j} - \log(-\log u_{0,j}) \right)$ ▷ $u_{0,j} \sim \text{Unif}(0, 1)$
$z \leftarrow z_0$, $\ell \leftarrow L_0$
for $k = 1$ **to** $m - 1$ **do**
    $\boldsymbol{y}_k \leftarrow \boldsymbol{W}_k^\top \boldsymbol{x} \in \mathbb{R}^g$
    $L_k \leftarrow \text{logsumexp}(\boldsymbol{y}_k)$
    $\ell_{\text{new}} \leftarrow \text{logsumexp}([\ell,\, L_k])$
    $p_{\text{replace}} \leftarrow \exp(L_k - \ell_{\text{new}})$ ▷ $= \frac{e^{L_k}}{e^{\ell} + e^{L_k}}$
    Draw $u \sim \text{Unif}(0, 1)$
    if $u < p_{\text{replace}}$ **then**
        $z_k \leftarrow \text{argmax}_{j \in [g]} \left(y_{k,j} - \log(-\log u_{k,j}) \right)$ ▷ sample within selected group
        $z \leftarrow kg + z_k$
    end if
    $\ell \leftarrow \ell_{\text{new}}$
end for
$\ell_Z \leftarrow \ell$ ▷ optional
return (z, ℓ_Z)

Distributed Group-Gumbel-Max. Algorithm B.4 extends FlashSampling to tensor-parallel vocabularies sharded across multiple GPUs.

Algorithm B.4 FlashSampling (distributed, tensor-parallel vocab): communicate $O(1)$ scalars per rank

---

Require: World size n. Rank $k \in \{0, \dots, n-1\}$ holds shard $\boldsymbol{W}^{(k)} \in \mathbb{R}^{d \times (V/n)}$ covering vocab indices $\{k \cdot V/n + 1, \dots, (k+1) \cdot V/n\}$. Input $\boldsymbol{x} \in \mathbb{R}^d$, RNG state.

Ensure: Global sample index $z \in \{1, \dots, V\}$ (and optional ℓ_Z)

1: On each rank k:
  compute local logits $\boldsymbol{y}^{(k)} \leftarrow (\boldsymbol{W}^{(k)})^\top \boldsymbol{x} \in \mathbb{R}^{V/n}$
  compute local log-mass $L_k \leftarrow \text{logsumexp}(\boldsymbol{y}^{(k)})$
  sample local index $\tilde{z}_k \sim \text{Cat}(\text{softmax}(\boldsymbol{y}^{(k)}))$ ▷ e.g., via Gumbel-Max / Group-Gumbel-Max / fused kernel

2: All-gather $\{(L_k, \tilde{z}_k)\}_{k=0}^{n-1}$ to a coordinator (or perform an equivalent reduction)

3: Sample winning rank $k^\star \leftarrow \text{argmax}_{k \in [n]} \left(L_k - \log(-\log \bar{u}_k)\right)$ ▷ $\bar{u}_k \sim \text{Unif}(0,1)$

4: $z \leftarrow k^\star \cdot (V/n) + \tilde{z}_{k^\star}$ ▷ convert rank-local index to global

5: Optionally $\ell_Z \leftarrow \text{logsumexp}([L_0, \dots, L_{n-1}])$

6: **return** z (and ℓ_Z)

C Numerically Stable and Fast Gumbel Generation

Gumbel noise can be generated as $g = -\log(-\log u)$ with $u \sim \text{Unif}(0,1)$. In GPU kernels, two issues matter:

- **Numerical stability:** avoid $u = 0$ or $u = 1$ which lead to infinities.
- **Throughput:** the cost of generating random numbers and computing logs should not dominate.

Practical recipe. Given a 32-bit RNG output $r \in \{0, \dots, 2^{32} - 1\}$, map to

$$u = \frac{r+1}{2^{32}+1} \in (0,1),$$

then compute $g = -\log(-\log u)$. Many GPU RNG libraries (e.g. Philox, XORWOW) support generating floats in $(0,1)$ directly; the above mapping is a safe fallback.

Approximate log options. If exactness in the distribution is required, the Gumbel generation must be statistically correct. However, using fast approximate log implementations can introduce small distortions. FlashSampling supports two modes:

- **Exact-math mode:** use standard log for high fidelity.
- **Fast-math mode:** use approximate logs for speed, with empirical validation that sampling bias remains negligible for target applications.

The sampling remains *algorithmically exact* with respect to the generated Gumbels; any bias comes from numeric approximations.

D Optional: returning log-normalizers or max values

Some applications need $\log Z = \log \sum_j e^{\widetilde{\ell}_j}$, for example to compute log-probabilities. The core FlashSampling sampler does not need $\log Z$, but it can be added as an optional mode by accumulating a numerically stable log-sum-exp alongside sampling. In fused settings this requires extra work in the epilogue, so we treat it as an optional feature rather than part of the core design.