FLASHSAMPLING: FAST AND MEMORY-EFFICIENT EXACT SAMPLING

Anonymous authors
Paper under double-blind review

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. In tensor-parallel decoding, FlashSampling replaces the all-gather of logits with streaming peer-to-peer writes: This overlaps GPU-to-GPU communication with computation and HBM loads across up to 8 GPUs, with near-ideal scaling at large batch sizes. Our kernel is exact because argmax decomposes over partitions; grouped variants for online and tensor-parallel settings are exact by hierarchical factorization of the categorical distribution. FlashSampling demonstrates kernel-level speedups on decode workloads across 4 different datacenter GPUs (H100, H200, B200, B300), and in end-to-end vLLM experiments, it reduces time per output token by up to 10% on the models we test. These results show that exact sampling, with no approximation, can be integrated into the matmul itself, consolidating the bandwidth-bound sampling step in an efficient epilogue.

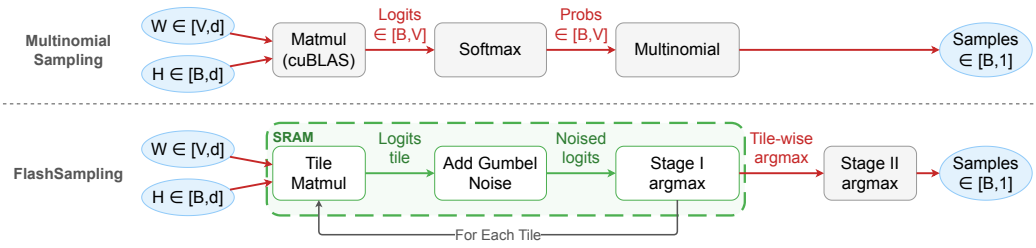

Figure 1: Conventional multinomial sampling (left) materializes the full $[B, V]$ logits tensor in HBM between the matmul and the sampler. FlashSampling (right) fuses sampling into the matmul epilogue, followed by a lightweight reduction over vocabulary tiles. Logits are computed tile-by-tile in on-chip 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.

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, on vocabularies with tens or hundreds of thousands of categories (Kwon et al., 2023; Ye et al., 2025; Maddison et al., 2014; Huijben et al., 2022). Recent measurements confirm the cost: sampling can account for over 10% of token generation time even on a single GPU (Key et al., 2024), and 20–38% in tensor-parallel settings where logits must be gathered across ranks (Zhao et al., 2025). The bottleneck is usually not arithmetic, but the chain of separate sampling kernels that materialize, normalize, and scan the logits tensor.

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.

At decode time, the LM-head projection already streams a large $[V, D]$ weight matrix from HBM (V=vocabulary size, D=hidden dimension). When the active batch is small, this projection is memory-bandwidth bound. Materializing the resulting $[B, V]$ logits tensor (B=batch size), launching extra kernels to normalize and sample from it, adds extra memory traffic but no useful model computation, since the logits are immediately discarded after a single sample is drawn. In this regime, the separate sampler is pure overhead (Dao et al., 2022; Wijmans et al., 2025). Furthermore, in the multi-GPU tensor-parallel setting, decoding must additionally synchronize and communicate the logits across ranks: Sampling becomes a memory and communication problem when full logits are materialized. Exact sampling is often described as "compute softmax, then sample", which obscures the fact that exact sampling does not require forming probabilities at all.

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; the method is exact and introduces no approximation. 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 can be summarized as follows:

1. We introduce a two-stage design that computes logits and samples tile-by-tile within the LM-head epilogue, instead of materializing the full logits tensor to HBM. In multi-GPU decoding, FlashSampling overlaps cross-GPU communication with the matmul compute and HBM loads and scales near-ideally on up to 8 GPUs.
2. 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. We demonstrate consistent speedups in the decode regime across four different NVIDIA datacenter GPUs in microbenchmarks and in end-to-end vLLM evaluations. We provide a simple I/O cost model to predict speedups, show that real speedups exceed these predictions, and explain why.

2 Background

Notation. Let $[V] := \{1, \dots, V\}$. Let $\tilde{\boldsymbol{\ell}} \in (\mathbb{R} \cup \{-\infty\})^V$ 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 $p(i) = \frac{\exp(\tilde{\ell}_i)}{\sum_{j=1}^{V} \exp(\tilde{\ell}_j)}$. Raw logits $\boldsymbol{\ell}$ are the special case $\tilde{\boldsymbol{\ell}} = \boldsymbol{\ell}$. We denote i.i.d. standard Gumbel variables by $g_i \sim \mathrm{Gumbel}(0, 1)$.

2.1 Why Sampling Is Expensive at Scale

A common LLM sampling pipeline first computes logits with a GEMM (LM-head projection), and then transforms them, normalizes them to probabilities, and finally samples from them. An example is softmax followed by inverse-CDF sampling. Algorithm A.1 in the appendix summarizes this pattern. An alternative to softmax-based sampling is described below in Section 2.2. Nevertheless, all samplers that materialize the logits to HBM must pay the price of at least one logits write, at least one logits re-read, and the price of the sampling work. This adds avoidable HBM round-trips in the critical path of the memory-bound decode loop.

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

Distributed logits. In the multi-GPU setting, the LM-head weights $\boldsymbol{W} \in \mathbb{R}^{V \times D}$ are sharded across n GPUs along the vocabulary dimension V (*column-parallel* sharding of $\boldsymbol{W}^\top$ (Shoeybi et al., 2020)). As a result, each GPU computes only a shard of the full logits $\boldsymbol{Y}$. Before sampling, all GPUs must communicate their logit shards via an all-gather collective, incurring per-GPU communication of order $B \cdot V$ and aggregate costs proportional to the number of GPUs n. This overhead is most pronounced in deployments using high tensor-parallelism, often aimed at minimizing latency.

2.2 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). *Given transformed logits $\tilde{\boldsymbol{\ell}} \in (\mathbb{R} \cup \{-\infty\})^V$, exact sampling from $\mathrm{Cat}(\mathrm{softmax}(\tilde{\boldsymbol{\ell}}))$ is: $i^\star = \mathrm{argmax}_{i \in [V]} \left(\tilde{\ell}_i + g_i\right), g_i \sim \mathrm{Gumbel}(0,1)$ i.i.d.*

This classical result goes back to Gumbel (1954) and is widely used in machine learning (Maddison et al., 2014; Huijben et al., 2022). The trick extends to sampling *without* replacement via the Gumbel-Top-k method (Kool et al., 2019), though not to the common sampling *with* replacement in LLMs. 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 and introduces no approximation: maintain the largest perturbed score seen so far and its index.

3.1 EXACT SAMPLING VIA ONLINE GUMBEL-MAX

Algorithm. Generate i.i.d. Gumbels, compute $s_i = \tilde{\ell}_i + g_i$, 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, analogous to the online normalizer calculation for softmax (Milakov & Gimelshein, 2018). No softmax, no normalization constant, and no prefix sum are required (see Algorithm I.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 ALGORITHM

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.

Why the two-stage design is simple. The fused stage does all the 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.

Algorithm 1 FlashSampling (two-stages)

Input: 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
Output: Samples $\boldsymbol{i}^\star \in \{1, \dots, V\}^B$

Stage 1 (Fused Kernel): for each batch tile $\mathcal{B}$ and vocabulary tile $\mathcal{V}_t$ in parallel
$\boldsymbol{Y}^{(t)}_{b,i} \leftarrow \sum_{d=1}^{D} \boldsymbol{H}_{b,d}\, \boldsymbol{W}_{i,d}$ for $(b,i) \in \mathcal{B} \times \mathcal{V}_t$ ▷ tiled matmul over D, kept on chip
for each output element $(b,i) \in \mathcal{B} \times \mathcal{V}_t$ **do**
  $\tilde{y}_{b,i} \leftarrow \mathrm{transform}\Big(\boldsymbol{Y}^{(t)}_{b,i}\Big)$ ▷ apply temperature, bias, mask
  Draw $u_{b,i} \in (0,1)$ and set $g_{b,i} \leftarrow -\log\big(-\log u_{b,i}\big)$
  $s_{b,i} \leftarrow \tilde{y}_{b,i} + g_{b,i}$
end for
for each row $b \in \mathcal{B}$ **do**
  $(m^{(t)}_b, \mathrm{idx}^{(t)}_b) \leftarrow \max_{i \in \mathcal{V}_t} s_{b,i}$ ▷ idx = global vocabulary index
  Write $(m^{(t)}_b, \mathrm{idx}^{(t)}_b)$ to HBM
  if multi-GPU **then**
    Fan-out $(m^{(t)}_b, \mathrm{idx}^{(t)}_b)$ to peer ranks via P2P
  end if
end for
if multi-GPU **then**
  Cross-rank barrier ▷ P2P writes are not collectives; sync before Stage 2
end if
Stage 2 (Reduction):
for each row $b \in \{1, \dots, B\}$ **do**
  $t^\star \leftarrow \mathrm{argmax}_t\, m^{(t)}_b$
  $i^\star_b \leftarrow \mathrm{idx}^{(t^\star)}_b$
end for
return $\boldsymbol{i}^\star$

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. For further discussion, please refer to C.

Multi-GPU Communication. A naive sampler must wait until the end of the GEMM to assemble the full logits $\boldsymbol{Y}$ using an all-gather collective. FlashSampling instead issues per-tile point-to-point writes (*fan-out* pattern) from inside the matmul epilogue. This broadcasts each tile-local candidate to the other GPUs via P2P (NVLink) as it is produced. Because this is not a collective, an explicit cross-rank barrier follows the kernel before Stage 2. The benefit is that GPU-to-GPU communication overlaps with the matmul's compute and HBM loads, instead of being deferred until after the GEMM.

The complete algorithm is provided in Algorithm 1, and a more detailed analysis and proof of its correctness can be found in Sections C, D.

3.3 IO Cost Model and Predicted Speedup

We outline a simple model to reason about speedups. It consists of comparing the theoretical minimal data movement of FlashSampling to the baseline, and inferring speedups from the ratio.

Baseline Cost: The baseline runs two steps: (1) computing logits with a GEMM, and (2) sampling from the logits. The GEMM must read the weights $\boldsymbol{W}$ and hidden states $\boldsymbol{H}$ and write the logits $\boldsymbol{Y}$ once. An (idealized) sampling step reads the logits $\boldsymbol{Y}$ once, and writes the sampled index $i^\star$ once. In practice, the baseline uses multiple kernels for sampling, not one. The total data movement M_{baseline}

from HBM is therefore:

$$M_{\text{baseline}} = \overbrace{\underbrace{VD}_{\text{read } \boldsymbol{W}} + \underbrace{DB}_{\text{read } \boldsymbol{H}} + \underbrace{VB}_{\text{write } \boldsymbol{Y}}}^{\text{GEMM}} + \overbrace{\underbrace{VB}_{\text{read } \boldsymbol{Y}} + \underbrace{B}_{\text{write } i^\star}}^{\text{sampling}}$$

FlashSampling Cost: Sampling is fused into the GEMM epilogue, so the logits write and read are eliminated. By avoiding the $\boldsymbol{Y}$ round-trip, fusion reduces data movement and improves arithmetic intensity.

$$M_{\text{fused}} = \overbrace{\underbrace{VD}_{\text{read } \boldsymbol{W}} + \underbrace{DB}_{\text{read } \boldsymbol{H}} + \underbrace{B}_{\text{write } i^\star}}^{\text{fused GEMM + sampling}}$$

Predicted speedup: The IO-model speedup is the ratio of the two costs M_{baseline} and M_{fused}. Since both $1/V \approx 0$ and $D/V \approx 0$ for current LLMs, the speedup equation simplifies to:

$$\frac{M_{\text{baseline}}}{M_{\text{fused}}} = \frac{VD + DB + 2VB + B}{VD + DB + B} = 1 + \frac{2}{D/B + D/V + 1/V} \approx 1 + \frac{2B}{D}.$$

The predicted speedup increases with batch size B and decreases with model hidden size D, i.e. smaller models experience larger speedups. We verify the IO model empirically by running FlashSampling with an optional flag that stores the computed logits back to HBM, isolating the logits data movement overhead with no other changes to the kernel. The measured overhead tracks the predicted $2B/D$ ratio (Appendix K, Table 9). Despite the IO model being accurate, FlashSampling achieves much larger speedups over the baseline than the IO model predicts, which we review in Section 4.

4 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 (Kwon et al., 2023) integration that measures autoregressive decode latency. We implement FlashSampling using Triton (Tillet et al., 2019); the implementation source code and experiment scripts are provided in the anonymized supplementary zip archive.

4.1 Setup

Hardware. Kernel microbenchmarks are run on four NVIDIA datacenter GPUs spanning two architecture generations (Hopper and Blackwell). Table 3 summarizes their specifications. All GPUs are provisioned via Modal cloud with at least 16 CPU cores.

Software. PyTorch 2.11.0 (Paszke et al., 2019), CUDA 13.0, Triton 3.6, and FlashInfer 0.6.9 (Ye et al., 2025). To measure kernel runtime, we use NVIDIA CUDA Profiling Tools Interface (CUPTI) in the single-GPU setting, and CUDA events in the multi-GPU setting, for compatibility. All kernels are warmed up for 25 iterations before timing. Inputs and weights are in BF16.

Workload configuration. The main text focuses on the decode-centric configuration ($D = 4{,}096; V = 151{,}936$), which matches models such as Qwen3-8B and Qwen3-235B-A22B MoE (Yang et al., 2025). We sweep batch sizes $B \in \{1, 2, 4, 8, 16, 32, 64, 128, 256\}$. Additional results for a larger configuration $D = 8{,}192; V = 128{,}256$ show the same qualitative trends (Appendix E).

We have the following baselines to be compared:

1. **Multinomial Sampling.** This baseline computes the logits using a matmul (cuBLAS), saves materialized logits to HBM, and samples with softmax and multinomial. We apply `torch.compile` to it, which improves speed by 11% on average over PyTorch eager (range: 5–17% across GPUs and batch sizes). Unless explicitly stated, all references to Multinomial Sampling refer to the compiled version.
2. **FI1 (FlashInfer top-k/top-p).** `top_k_top_p_sampling_from_logits`[1], a sampling kernel used by vLLM for top-k/top-p decode. Logits are also computed using cuBLAS and material-

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

ized on HBM. For fair comparison, we deactivate top-k and top-p to prevent unnecessary work (`top_k=-1, top_p=1.0`).

3. **FI2 (FlashInfer Gumbel-Max).** `sampling_from_logits`[1], FlashInfer's exact Gumbel-Max sampler on materialized logits (skipping softmax normalization). Logits computed using cuBLAS, and materialized on HBM.

Multi-GPU setting. We compare FlashSampling against the 3 baselines in the multi-GPU distributed setting (tensor parallel) with 2, 4, and 8 GPUs. For these experiments, we use a larger hidden size ($V = 128{,}256$; $D = 8{,}192$) to mirror the size of real models run in tensor parallel mode, like Llama3 70B (Grattafiori et al., 2024) and DeepSeek V3. The baselines are `torch.compiled` with full graphs, including the all-gather collective operation[2].

4.2 Microbenchmarking Results

Table 4 and Figure 2 report FlashSampling speedups relative to the three baselines. A relative speedup of *e.g.* $2\times$ means that FlashSampling takes half the runtime as the baseline. We observe that FlashSampling is faster than all baselines across all batch sizes on the B200 GPU. The complete data for all four datacenter GPUs is in Appendix E.

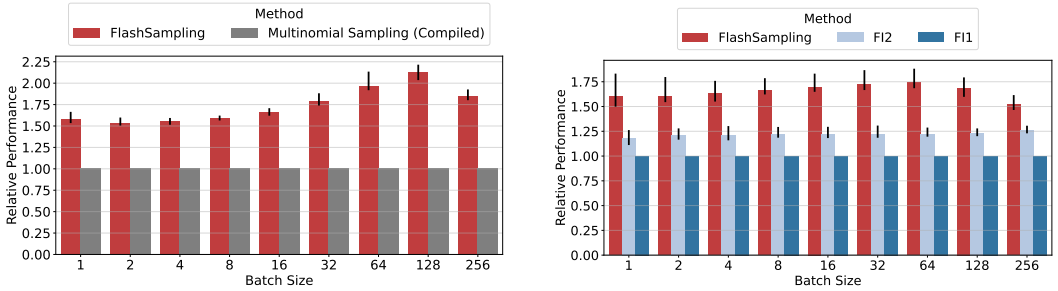

Figure 2: Relative performance on B200. Left: Versus Multinomial Sampling. Right: Versus FlashInfer FI1 and FI2. The error bars indicate min-max relative speedups across 10 runs of 100 iterations each.

We have the following observations:

1. For $B \leq 64$, FlashSampling is faster than all baselines on all GPUs. The peak speedup vs. Multinomial Sampling is $2.23\times$ (B300) and the peak speedup vs. FI1 is $1.74\times$ (B200).
2. Speedups over FI2 are smaller than speedups over Multinomial Sampling or FI1 because FI2 also uses Gumbel-Max sampling. The remaining gain therefore comes mainly from eliminating logits materialization and sampling overhead (Section 4.4).
3. As batch size grows to $B = 256$, GEMM efficiency matters more, and the workload becomes less dominated by memory-bound sampling. Appendix E, Table 5 shows the same qualitative trend for the larger hidden dimension $D = 8192$ with the crossover occurring earlier.

4.3 Multi-GPU Results with Tensor Parallelism

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. In practice, this is implemented as per-tile P2P writes from inside the matmul epilogue (Section 3.2), with a cross-rank barrier before the reduction stage.

We measure the lowest runtime across 5 runs of 100 iterations (the minimum is more robust to one-sided benchmarking noise (Chen & Revels, 2016), which we observed in multi-GPU benchmarks)[3].

[2]Implemented with the compile-friendly ops in `torch.distributed.functional_collectives`.

[3]We also observed a large runtime difference between different hardware nodes on Modal cloud, even when GPUs were connected via NVLink.

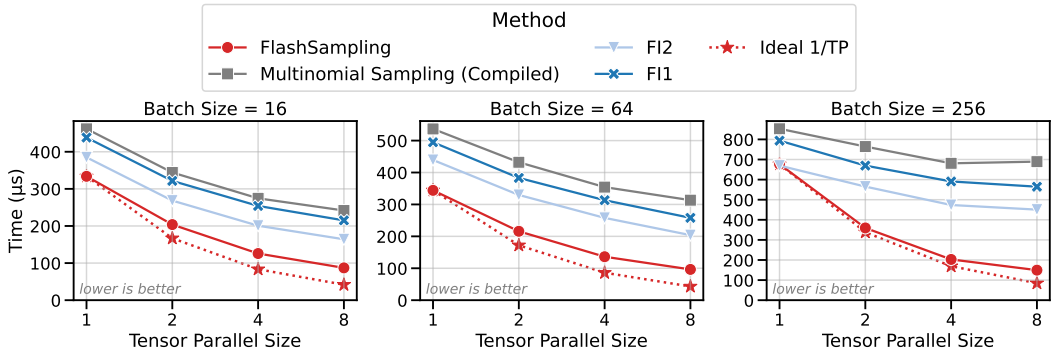

Figure 3: Runtime (lower is better) of FlashSampling and the three baselines at TP=1, 2, 4, 8. The batch sizes (16, 64, 256) are selected to include compute- and memory-bound decode regimes. The multi-GPU microbenchmarks took under 5 hours total to run. The data is tabulated in Table 6 (Appendix F).

Figure 3 compares the runtime (lower is better) of FlashSampling and the three baselines over tensor parallel sizes $TP \in \{1, 2, 4, 8\}$. The dotted line shows the *ideal speedup*, which is the runtime for TP=1 divided by the TP size. We observe that FlashSampling is faster than all baselines in low and medium batch sizes (16 and 64, memory-bound regime). At higher batch sizes (256, close to the compute-bound regime), FlashSampling closely follows the ideal speedup, scaling optimally across multiple GPUs. The baselines (FI1, FI2, Multinomial) wait for cuBLAS to finish, then issue an all-gather collective. FlashSampling instead emits per-tile P2P writes, which run concurrently with the matmul, effectively hiding the GPU-to-GPU communication.

Table 1: Sampling as a percentage of total kernel time. A high fraction spent on sampling rather than matmul is an 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		*Multinomial Sampling*		*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**

4.4 Interpreting the Batch-Size Trend

The cost model in Section 3.3 showed that HBM savings from avoiding the logits write and reread alone are small ($\leq$6% of traffic). Figure 4 reveals a larger effect: the baselines' separate sampling kernels are expensive, and their runtime grows steeply with batch size, while FlashSampling absorbs sampling into the matmul at negligible cost (Table 1: 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. Note that Triton is platform-agnostic (AMD, Intel GPUs, etc.), so the cuBLAS gap is a trade-off for portability. Profiling was performed on an RTX 3090 using Nsight Compute and Proton.

4.5 End-to-End vLLM Evaluation

In this section, we demonstrate the end-to-end speedups achieved by FlashSampling on LLM inference. We integrate FlashSampling into vLLM (Kwon et al., 2023) by replacing the LM-head projection and the sampling step. We benchmark TPOT (Time Per Output Token) using problems

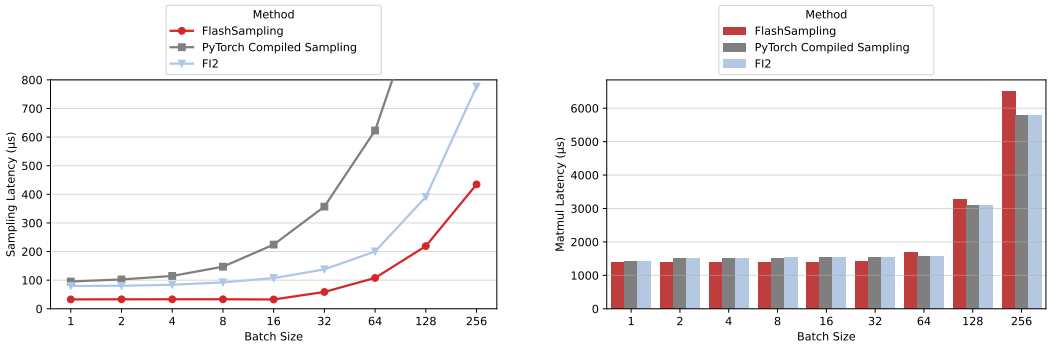

Figure 4: Sampling runtime (left) and matmul runtime (right) in μs vs. batch size. Lower is better.

from the AIME22-24 dataset[4]. vLLM uses continuous batching, so the effective batch size varies dynamically during serving. We use `vllm bench sweep serve` with `-max-concurrency`=B to implement the batch size, and set `-request-rate`=B for requests to follow a Poisson process at B requests per second. We rerun the benchmark 5 times for each batch size, compare TPOT between baseline and FlashSampling, and report the median TPOT reduction across the 5 runs. Experiments run on B200 GPUs across four models spanning a range of sizes: Qwen3-1.7B and Qwen3-8B on a single GPU (TP1), and Qwen3-32B and Llama-3.3-70B on two GPUs (TP2). Table 7, 8, together with Figure 5, present the experimental results.

Key observations. The speedups are proportional to the decoding time spent on the LM head compared to attention and FFN. This explains the highest speedups on Qwen3-1.7B and 8B, which see up to 10.2% and 8.7% TPOT reduction, respectively. For Qwen3-32B and Llama-3.3-70B, attention and FFN layers dominate decode time, so the speedups are smaller, but consistent (peaks of 2.9% and 2.7%, respectively).

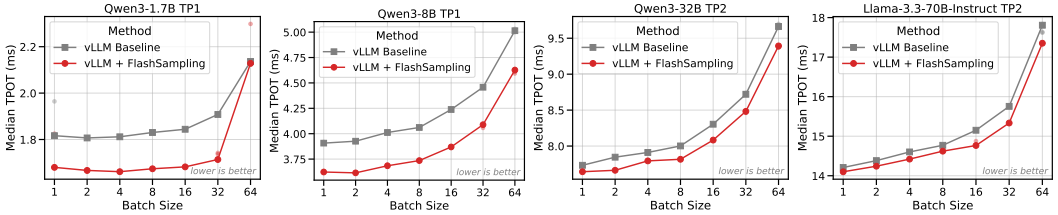

Figure 5: TPOT vs. concurrency on B200 across four model sizes. Qwen3-1.7B (TP1) and Qwen3-8B (TP1) see the largest reductions (up to 10% and 7–9%, respectively). Qwen3-32B (TP2) and Llama-3.3-70B (TP2) achieve smaller but consistent gains (peaks of 2.9% and 2.7%), since attention and FFN dominate decode time at these scales.

4.6 Empirical Correctness Verification

Kernel Level. To verify sampling correctness, we compare samples from FlashSampling to the reference PyTorch implementation using a chi-squared goodness-of-fit test. We use a vocabulary size $V = 512$ and draw 10,000 samples, giving sufficiently large expected counts per category for the test, and find no statistically significant difference.

End-to-end Level. We run FlashSampling on 1,319 questions from the GSM8K dataset (Cobbe et al., 2021) using Qwen3-1.7B and check the answers with a LLM judge. FlashSampling achieves 89.4% accuracy versus 89.6% for the baseline. This difference is not statistically significant (p=0.776), according to a paired bootstrap test. This is consistent with exact sampling. One cannot use greedy sampling here, since it would disable FlashSampling.

[4] https://huggingface.co/datasets/AI-MO/aimo-validation-aime

5 Related Work

Gumbel-Max and Extensions. The Gumbel-Max trick for exact categorical sampling dates to Gumbel (1954) and was formalized by Maddison et al. (2014). Jang et al. (2017) introduced the Gumbel-Softmax relaxation for differentiable discrete sampling, which complements our focus on exact sampling. Huijben et al. (2022) surveys the broader Gumbel-Max literature. Kool et al. (2019) extend the trick to top-k sampling without replacement, and Qi et al. (2020) study fast Gumbel variate generation. Ahmed & Singh (2026) modify the sampling distribution via entropy-aware reweighting and use Gumbel-Max as a subroutine. FlashSampling contributes a systems-oriented hierarchical decomposition for exact online and distributed sampling in LLM inference, preserving the original distribution exactly.

IO-Aware Kernel Fusion. FlashAttention (Dao et al., 2022) showed that avoiding materialization of the attention matrix can substantially reduce HBM traffic, with subsequent work improving parallelism (Dao, 2024) and exploiting hardware asynchrony (Shah et al., 2024). Cut Your Losses (Wijmans et al., 2025), Liger Kernel (Hsu et al., 2025), and Dong & Chang (2025) apply the same idea to training-time cross-entropy by fusing the LM-head matmul with the loss computation. The same matmul-plus-epilogue fusion pattern appears in MLP layers (Zhang et al., 2026), RNNs (Pöppel et al., 2025), and whole-model inference (Nrusimha et al., 2025). At the compiler level, EVT (Chen et al., 2024) auto-generates fused GEMM epilogues via CUTLASS, and Samaga et al. (2025) fuses approximate top-k selection into the matmul on TPUs. FlashSampling applies this methodology to a different domain: inference-time sampling, exploiting domain-specific structure (Gumbel-Max decomposability), and achieving exactness (no approximations).

Efficient LLM Sampling. FlashInfer (Ye et al., 2025) provides optimized GPU kernels for attention and sampling in LLM serving, including sorting-free rejection sampling for top-k/top-p. Qrita (Park et al., 2026) speeds up top-k/top-p truncation, which is orthogonal to FlashSampling and theoretically composable with it. Min-p sampling (Minh et al., 2025) proposes a dynamic truncation method that, like top-p, requires probability computation before truncation. SIMPLE (Zhao et al., 2025) offloads sampling to the CPU, motivated by the same bottleneck FlashSampling addresses. Sampled softmax (Rawat et al., 2019) reduces large-vocabulary cost by computing the loss over a random subset, trading exactness for speed. All these methods operate on pre-materialized logits, while FlashSampling avoids materializing them entirely and introduces no approximation.

6 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. The method introduces no approximation: it produces exact samples from the target categorical distribution. Empirically, FlashSampling is most effective in the memory-bound decode regime, where it absorbs the sampling kernel into the matmul. In multi-GPU settings, it overlaps cross-GPU communication with logit computation and HBM loads, scaling near-ideally to 8 GPUs.

AI use statement

We used generative AI tools to help implement the FlashSampling kernels, benchmarks, and tests, and to help design the experiments. We did not use them to develop the theory, formulate or prove the mathematical results, propose hypotheses, or interpret the results; generating synthetic data, cleaning or reformatting datasets, qualitative data analysis, and translation do not apply to this work. We also used generative AI tools to edit the paper for readability and to migrate it to the ICLR template. We reviewed all AI-assisted work: an author reviewed every AI-assisted code change and all AI-edited text, and unit tests and the correctness checks in Section 4.6 verify the sampling distribution of the kernels. We take responsibility for the final content of this work, including text, claims, and artifacts produced with the aid of generative AI.

Reproducibility statement

The anonymized supplementary material contains the FlashSampling implementation and the experiment scripts. Section 4.1 gives the GPUs, software versions, timing protocol, workload shapes, batch sizes, and baselines, and Table 3 lists the GPU specifications. Algorithm 1 and Appendix I give pseudocode for every FlashSampling variant, and Appendix C describes the RNG and numerical precision. Section 4.5 describes the vLLM integration, models, dataset, and benchmark commands. Appendix D states the exactness results with complete proofs. Appendices E, F, and G tabulate the kernel speedups, multi-GPU runtimes, and vLLM TPOT results.

References

Kareem Ahmed and Sameer Singh. Entropy-aligned decoding of lms for better writing and reasoning, 2026. URL https://arxiv.org/abs/2601.01714.

Jiahao Chen and Jarrett Revels. Robust benchmarking in noisy environments. *arXiv preprint arXiv:1608.04295*, 2016.

Zhaodong Chen, Andrew Kerr, Richard Cai, Jack Kosaian, Haicheng Wu, Yufei Ding, and Yuan Xie. Evt: Accelerating deep learning training with epilogue visitor tree. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS '24, pp. 301–316, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400703867. doi: 10.1145/3620666.3651369. URL https://doi.org/10.1145/3620666.3651369.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, et al. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

Tri Dao. Flashattention-2: Faster attention with better parallelism and work partitioning. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=mZn2Xyh9Ec.

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.

Jianbing Dong and Jianbin Chang. From projection to prediction: Beyond logits for scalable language models, 2025. URL https://arxiv.org/abs/2511.17599.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

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

Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. The curious case of neural text degeneration. *arXiv preprint arXiv:1904.09751*, 2019.

Pin-Lun Hsu, Yun Dai, Vignesh Kothapalli, Qingquan Song, Shao Tang, Siyu Zhu, Steven Shimizu, Shivam Sahni, Haowen Ning, Yanning Chen, and Zhipeng Wang. Liger-kernel: Efficient triton kernels for LLM training. In *Championing Open-source DEvelopment in ML Workshop @ ICML25*, 2025. URL https://openreview.net/forum?id=36SjAIT42G.

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.

Oscar Key, Luka Ribar, Alberto Cattaneo, Luke Hudlass-Galley, and Douglas Orr. Approximate top-k for increased parallelism. *arXiv preprint arXiv:2412.04358*, 2024.

Wouter Kool, Herke Van Hoof, and Max Welling. Stochastic beams and where to find them: The gumbel-top-k trick for sampling sequences without replacement. In *International conference on machine learning*, pp. 3499–3508. PMLR, 2019.

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*, pp. 611–626, 2023.

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

Maxim Milakov and Natalia Gimelshein. Online normalizer calculation for softmax. *arXiv preprint arXiv:1805.02867*, 2018.

Nguyen Nhat Minh, Andrew Baker, Clement Neo, Allen G Roush, Andreas Kirsch, and Ravid Shwartz-Ziv. Turning up the heat: Min-p sampling for creative and coherent LLM outputs. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=FBkpCyujtS.

Aniruddha Nrusimha, William Brandon, Mayank Mishra, Yikang Shen, Rameswar Panda, Jonathan Ragan-Kelley, and Yoon Kim. Flashformer: Whole-model kernels for efficient low-batch inference, 2025. URL https://arxiv.org/abs/2505.22758.

NVIDIA. NVIDIA H100 tensor core GPU architecture. Technical report, NVIDIA, 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, NVIDIA, 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.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

Korbinian Pöppel, Maximilian Beck, and Sepp Hochreiter. FlashRNN: I/o-aware optimization of traditional RNNs on modern hardware. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=l0ZzTvPfTw.

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 2020*, pp. 796–807, 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.

Yashas Samaga, Varun Yerram, Spandana Raj Babbula, Prateek Jain, and Praneeth Netrapalli. Faster approx. top-k: Harnessing the full power of two stages, 2025. URL https://arxiv.org/abs/2506.04165.

Jay Shah, Ganesh Bikshandi, Ying Zhang, Vijay Thakkar, Pradeep Ramani, and Tri Dao. Flashattention-3: Fast and accurate attention with asynchrony and low-precision. *Advances in Neural Information Processing Systems*, 37:68658–68685, 2024.

Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism, 2020. URL https://arxiv.org/abs/1909.08053.

Philippe Tillet, Hsiang-Tsung Kung, and David Cox. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pp. 10–19, 2019.

Erik Wijmans, Brody Huval, Alexander Hertzberg, Vladlen Koltun, and Philipp Krähenbühl. Cut your losses in large-vocabulary language models. In *International Conference on Learning Representations*, volume 2025, pp. 68174–68193, 2025.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 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.

Zixi Zhang, Zhiwen Mo, Yiren Zhao, and Robert Mullins. Deep kernel fusion for transformers, 2026. URL https://arxiv.org/abs/2602.11808.

Bohan Zhao, Zane Cao, and Yongchao He. Simple: Disaggregating sampling from gpu inference into a decision plane for faster distributed llm serving, 2025. URL https://arxiv.org/abs/2512.00719.

A Baseline Algorithm

Algorithm A.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 $i^\star \in \{1, \dots, V\}$

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

B GPU Configuration

B.1 GPU Memory Hierarchy

Table 2 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 2: GPU memory hierarchy (H100 SXM) (NVIDIA, 2022; 2024).

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

Table 3: GPU specifications. Peak BF16 TFLOP/s are dense (without structured sparsity), since the LM-head matmul is a dense GEMM. The ops:byte ratio (peak compute / bandwidth) 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

C Further details of FlashSampling

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.

D 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.

D.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

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

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

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

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 $z_k \sim \mathrm{Cat}(\mathrm{softmax}(\boldsymbol{y}_k))$ and (ii) its group log-mass $L_k = \mathrm{logsumexp}(\boldsymbol{y}_k)$. The algorithm then samples $K \sim \mathrm{Cat}(\mathrm{softmax}(\boldsymbol{L}))$ 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

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

Then replace z by z_k with probability

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

and otherwise keep z. Section D.4 proves that this binary merge rule preserves exactness by induction.

D.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. Mechanically, each rank fan-outs its per-tile candidates to peers via NVLink P2P from within the matmul epilogue, so this communication overlaps with the GEMM rather than executing as a separate post-GEMM collective.

D.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 D.1 (Gumbel max-stability under grouping). *Let $\{g_i\}_{i=1}^V$ be i.i.d.* $\mathrm{Gumbel}(0,1)$ *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*

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

Then:

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

Proof. For any real t,

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

which is the CDF of $\mathrm{Gumbel}(L_k, 1)$. 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 $\mathrm{argmax}_k\, M_k$ 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 1, exactness does *not* rely on max-stability: once the perturbed scores $x_i = \tilde{\ell}_i + g_i$ have been formed, exactness is simply the deterministic identity

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

D.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 D.2 (Exact group factorization). *Let $[V]$ be partitioned into groups $\{\mathcal{G}_k\}_{k=0}^{m-1}$, and discard any zero-mass groups. Define $L_k = \log\sum_{i\in\mathcal{G}_k}\exp(\tilde{\ell}_i)$. If we sample $K \sim \mathrm{Cat}(\mathrm{softmax}(\boldsymbol{L}))$ and then sample $I \mid (K = k) \sim \mathrm{Cat}(\mathrm{softmax}(\tilde{\boldsymbol{\ell}}_{\mathcal{G}_k}))$, the marginal distribution of I equals $\mathrm{Cat}(\mathrm{softmax}(\tilde{\boldsymbol{\ell}}))$.*

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

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

□

Lemma D.3 (Binary merge rule). *Let $A, B \subseteq [V]$ be disjoint and suppose both have nonzero mass. Define*

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

Suppose $Z_A \sim \mathrm{Cat}(\mathrm{softmax}(\tilde{\boldsymbol{\ell}}_A))$, $Z_B \sim \mathrm{Cat}(\mathrm{softmax}(\tilde{\boldsymbol{\ell}}_B))$, 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 $\mathrm{Cat}(\mathrm{softmax}(\tilde{\boldsymbol{\ell}}_{A\cup B}))$.

Proof. For any $i \in A$,

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

The same calculation for $i \in B$ gives

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

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

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

Proof. For the parallel and distributed variants, Lemma D.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 D.3. An induction over the streamed groups therefore yields an exact sample from the full categorical distribution. □

D.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 D.5 (Max over vocabulary tiles)**.** *Let* $\{x_i\}_{i=1}^{V}$ *be real numbers and let* $\{\mathcal{V}_s\}_{s=0}^{n_{\text{tile}}-1}$ *be a partition of* $[V]$ *into vocabulary tiles. For each tile, define*

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

where $\hat{\imath}_s$ *is a global index in* $\mathcal{V}_s$*. 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* $\hat{\imath}_{s^\star}$ *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:

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

If $s^\star \in \operatorname{argmax}_s m_s$, then $x_{\hat{\imath}_{s^\star}} = m_{s^\star} = \max_i x_i$, so $\hat{\imath}_{s^\star}$ 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 $s^\star \in \operatorname{argmax}_s m_s$. □

Applying Lemma D.5 to $x_i = \tilde{\ell}_i + g_i$ justifies the two-stage fused design in Algorithm 1. 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.

D.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 via the Gumbel-Top-k trick (Kool et al., 2019). 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 (Holtzman et al., 2019) 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[5,6], FlashInfer[7], and other SOTA top-k top-p algorithms (Park et al., 2026).
- **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.

E Kernel Microbenchmark Data

For completeness, Table 4 reports the smaller-configuration kernel results deferred from the main text, and Table 5 reports the larger configuration. The same qualitative pattern appears in both: FlashSampling is strongest in the small-batch decode regime, while the advantage narrows once the workload becomes more GEMM-efficiency dominated.

Table 4: FlashSampling relative speedup vs. three baselines on the smaller configuration (D=4096, V=151k). Values > 1 indicate FlashSampling is faster; bold marks the peak per GPU within each baseline. The numbers are medians over 100 iterations. Each column takes under 10 minutes to run.

	vs. Multinomial Sampling				*vs. FI1* (top-k/top-p)				*vs. FI2* (Gumbel-Max)			
B	H100	H200	B200	B300	H100	H200	B200	B300	H100	H200	B200	B300
1	1.29	1.35	1.58	1.55	1.34	1.39	1.65	1.57	1.20	1.24	1.35	1.33
2	1.28	1.34	1.52	1.50	1.32	1.44	1.58	1.56	1.19	1.22	1.32	1.29
4	1.28	1.35	1.58	1.54	1.32	1.42	1.62	1.61	1.19	1.23	1.37	1.32
8	1.31	1.38	1.58	1.55	1.35	1.48	1.63	1.60	1.19	1.24	1.37	1.33
16	1.35	1.44	1.66	1.63	1.37	1.49	1.70	1.65	1.20	1.25	1.39	1.35
32	1.44	1.53	1.79	1.74	1.39	**1.50**	1.70	1.67	1.22	**1.27**	1.42	1.37
64	1.64	**1.67**	1.96	1.92	**1.47**	1.49	**1.74**	1.69	**1.27**	1.25	**1.43**	1.39
128	**1.67**	1.43	**2.14**	**2.23**	1.36	1.15	1.68	**1.74**	1.14	0.94	1.39	**1.44**
256	1.30	1.22	1.84	2.03	1.03	1.00	1.54	1.65	0.81	0.79	1.19	1.30

F Multi-GPU Runtime Values

Table 6 reports the minimum kernel runtime (μs) underlying Figure 3, taken as the lowest of 5 runs of 100 iterations each. The configuration is the larger one (D=8192, V=128k). FlashSampling attains the lowest runtime in every cell except (B=256, TP=1), where FI2 is marginally faster.

G Absolute/Relative TPOT Results for vLLM Evaluation

Table 7 reports the median TPOT (ms) for baseline and FlashSampling on B200, complementing the relative speedups in Table 8. Smaller models (Qwen3-1.7B, Qwen3-8B) run on a single GPU (TP1); larger models (Qwen3-32B, Llama-3.3-70B) run on two GPUs (TP2).

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

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

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

Table 5: 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. Multinomial Sampling				*vs. FI1* (top-k/top-p)				*vs. FI2* (Gumbel-Max)			
B	H100	H200	B200	B300	H100	H200	B200	B300	H100	H200	B200	B300
1	1.23	1.26	1.45	1.41	1.20	1.25	1.36	1.30	1.14	1.13	1.21	1.19
2	1.22	1.23	1.41	1.39	1.21	1.21	1.33	1.29	1.13	1.11	1.21	1.18
4	1.22	1.24	1.35	1.34	1.20	1.21	1.31	1.28	1.13	1.12	1.15	1.13
8	1.23	1.25	1.37	1.36	1.21	1.23	1.30	1.29	1.13	1.12	1.14	1.13
16	1.24	1.27	1.40	1.39	1.21	1.25	1.32	1.30	1.14	1.13	1.15	1.15
32	1.27	1.34	1.47	1.45	1.23	1.30	1.36	1.35	1.15	1.18	1.20	1.19
64	**1.36**	**1.43**	**1.61**	1.58	**1.28**	**1.33**	**1.46**	**1.44**	**1.19**	**1.20**	**1.30**	**1.28**
128	1.30	1.01	1.59	**1.61**	1.16	0.88	1.34	1.39	1.05	0.78	1.19	1.23
256	0.88	0.82	1.27	1.32	0.80	0.74	1.12	1.19	0.69	0.65	0.97	1.02

Table 6: Minimum kernel runtime (μs, lower is better) across tensor-parallel sizes TP $\in \{1, 2, 4, 8\}$ and batch sizes $B \in \{16, 64, 256\}$ for input shape (D=8192, V=128k). Bold marks the fastest method per (B, TP) cell.

B	Method	TP=1	TP=2	TP=4	TP=8
16	FlashSampling	**333.8**	**203.8**	**125.8**	**87.1**
	FI1	438.3	321.5	254.1	215.1
	FI2	385.0	268.4	200.7	164.2
	Multinomial	461.8	344.1	274.5	241.7
64	FlashSampling	**344.1**	**215.9**	**136.2**	**96.4**
	FI1	494.8	383.1	313.3	258.0
	FI2	439.2	329.7	257.9	203.8
	Multinomial	536.6	432.1	354.3	313.3
256	FlashSampling	676.8	**359.5**	**202.7**	**149.5**
	FI1	793.7	669.7	590.8	564.7
	FI2	**669.7**	565.3	473.6	450.6
	Multinomial	852.9	764.5	681.0	689.4

Table 7: Median TPOT (ms) over 5 runs on B200 for baseline and FlashSampling. Lower is better. B is the batch size.

	Qwen3-1.7B (TP1)		Qwen3-8B (TP1)		Qwen3-32B (TP2)		Llama-3.3-70B (TP2)	
B	Base	FlashSampling	Base	FlashSampling	Base	FlashSampling	Base	FlashSampling
1	1.82	1.68	3.91	3.62	7.73	7.64	14.21	14.10
2	1.81	1.67	3.93	3.61	7.85	7.66	14.38	14.24
4	1.81	1.66	4.01	3.68	7.91	7.79	14.60	14.42
8	1.83	1.67	4.06	3.74	8.00	7.82	14.77	14.62
16	1.84	1.68	4.24	3.87	8.30	8.08	15.15	14.76
32	1.91	1.71	4.46	4.09	8.72	8.48	15.75	15.33
64	2.14	2.13	5.01	4.63	9.66	9.39	17.81	17.35

Table 8: TPOT speedup in % (larger is better) computed as $(1 - \text{FlashSampling}/\text{baseline})$, and standard deviation across 5 runs. B is the batch size. Bold marks the peak per model. Absolute TPOT values are in Appendix G. Each column takes under 2 hours to run.

B	Qwen3-1.7B (TP1)	Qwen3-8B (TP1)	Qwen3-32B (TP2)	Llama-3.3-70B (TP2)
1	$7.5 \pm 0.2\,\%$	$7.3 \pm 0.2\,\%$	$1.2 \pm 0.0\,\%$	$0.8 \pm 0.0\,\%$
2	$7.7 \pm 0.1\,\%$	$7.9 \pm 0.2\,\%$	$2.3 \pm 0.1\,\%$	$1.0 \pm 0.0\,\%$
4	$8.3 \pm 0.0\,\%$	$8.1 \pm 0.1\,\%$	$1.5 \pm 0.0\,\%$	$1.2 \pm 0.0\,\%$
8	$8.5 \pm 0.0\,\%$	$8.1 \pm 0.1\,\%$	$2.3 \pm 0.0\,\%$	$1.0 \pm 0.0\,\%$
16	$8.8 \pm 0.1\,\%$	$\mathbf{8.7} \pm 0.3\,\%$	$2.7 \pm 0.0\,\%$	$2.5 \pm 0.3\,\%$
32	$\mathbf{10.2} \pm 0.8\,\%$	$8.5 \pm 0.3\,\%$	$\mathbf{2.9} \pm 0.2\,\%$	$\mathbf{2.7} \pm 0.1\,\%$
64	$0.4 \pm 3.6\,\%$	$8.0 \pm 0.5\,\%$	$2.8 \pm 0.3\,\%$	$2.5 \pm 0.5\,\%$

H Roofline Analysis and Bandwidth Utilization

The LM-head projection is memory-bandwidth-bound at small batch sizes because arithmetic intensity equals B (the weight matrix dominates traffic). Figure 6 confirms this on B200.

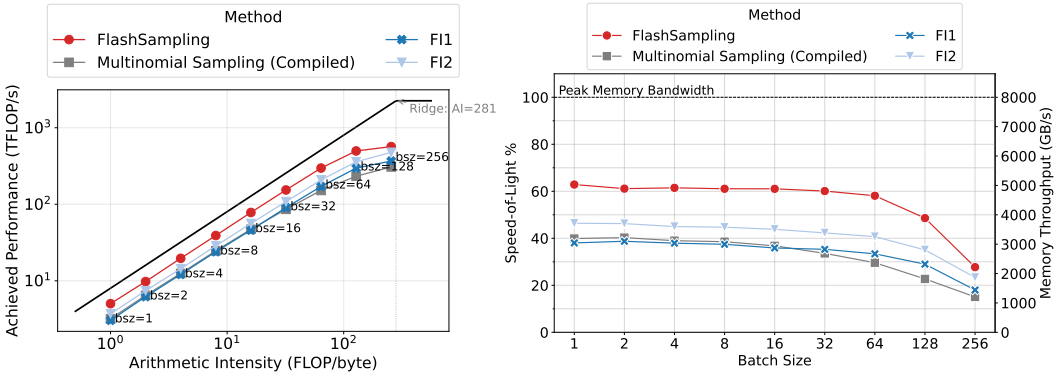

Figure 6: Roofline (left) and HBM bandwidth utilization (right) on B200 (higher is better). Left: all methods track the memory-bound slope for $B \leq 64$; FlashSampling sits above baselines because it avoids the logits round-trip. Close to the ridge point ($\text{AI} \approx 281$), performance flattens below the compute ceiling, where cuBLAS outperforms Triton. Right: FlashSampling achieves higher bandwidth utilization than all baselines in the decode regime, confirming that fusion removes overhead rather than shifting it.

I FlashSampling Algorithm Pseudocode

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

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

Algorithm I.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 $g = -\log(-\log u), u \sim \text{Unif}(0, 1)$
  $s \leftarrow \ell_i + g$
  if $s > m$ **then**
    $m \leftarrow s, i^\star \leftarrow i$
  end if
end for
return $i^\star$

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

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

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

Algorithm I.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 $z \in \{1, \dots, V\}$ and optional log-normalizer $\ell_Z = \mathrm{logsumexp}(\boldsymbol{y})$

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

Algorithm I.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 $z \in \{1, \dots, V\}$ and optional log-normalizer ℓ_Z

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

Algorithm I.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)

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 $L_k \leftarrow \mathrm{logsumexp}(\boldsymbol{y}^{(k)})$
  sample local index $\tilde{z}_k \sim \mathrm{Cat}(\mathrm{softmax}(\boldsymbol{y}^{(k)}))$ ▷ e.g., via Gumbel-Max / Group-Gumbel-Max / fused kernel
All-gather $\{(L_k, \tilde{z}_k)\}_{k=0}^{n-1}$ to a coordinator (or perform an equivalent reduction)
Sample winning rank $k^\star \leftarrow \mathrm{argmax}_{k \in [n]} \left(L_k - \log(-\log \bar{u}_k)\right)$ ▷ $\bar{u}_k \sim \mathrm{Unif}(0,1)$
$z \leftarrow k^\star \cdot (V/n) + \tilde{z}_{k^\star}$ ▷ convert rank-local index to global
Optionally $\ell_Z \leftarrow \mathrm{logsumexp}([L_0, \dots, L_{n-1}])$
return z (and ℓ_Z)

J Numerically Stable and Fast Gumbel Generation

Gumbel noise can be generated as $g = -\log(-\log u)$ with $u \sim \mathrm{Unif}(0, 1)$. 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 $r \in \{0, \ldots, 2^{32} - 1\}$, 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.

K Logits-Store Ablation

Table 9 reports the overhead of storing the computed $[B, V]$ logits tensor (FP32) back to HBM from within the fused kernel, compared with the predicted overhead of $2B/D$ from the cost model in Section 3.3. The ablation toggles a single flag in the kernel and changes nothing else. The measured overhead was slightly larger than predicted, but tracked the trend closely.

Table 9: Predicted vs. measured overhead of storing logits to HBM. Predicted: $2B/D$. Measured: relative slowdown of the fused kernel with the logits store enabled, averaged over 5 runs (B200 GPU).

	D=8192, V=128k		D=4096, V=152k	
B	Predicted	Measured	Predicted	Measured
1	0.02%	0.5%	0.05%	0.9%
4	0.10%	0.7%	0.20%	1.2%
16	0.39%	1.6%	0.78%	2.1%
64	1.56%	3.6%	3.13%	4.8%
128	3.13%	5.4%	6.25%	8.9%
256	6.25%	8.1%	12.50%	15.2%

L Returning Log-Normalizers or Max Values

Some applications need $\log Z = \log \sum_j e^{\tilde{\ell}_j}$, 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.

M Licenses of Existing Assets

Table 10 lists the third-party software, models, and datasets used in this paper, with citation and license. All assets are used in accordance with their respective licenses.

Table 10: Licenses of existing assets used in this paper.

Asset	Reference	License
PyTorch	Paszke et al. (2019)	BSD-3-Clause
Triton	Tillet et al. (2019)	MIT
vLLM	Kwon et al. (2023)	Apache-2.0
FlashInfer	Ye et al. (2025)	Apache-2.0
Qwen3 (1.7B, 8B, 32B)	Yang et al. (2025)	Apache-2.0
Llama-3.3-70B	Grattafiori et al. (2024)	Llama 3.3 Community License
GSM8K	Cobbe et al. (2021)	MIT