Speeding up Log-Sum-Exp: Kernel Fusion at the Memory Wall, Integer Arithmetic at the Compute Wall

Lingyun Yao *
Aalto University
Espoo, Finland

Martin Andraud
UC Louvain
Louvain-la-Neuve, Belgium

Niki Loppi
NVIDIA
Helsinki, Finland

Andrea Pilzer
NVIDIA
Bologna, Italy

Anji Liu
National University of Singapore
Singapore

Guy Van den Broeck
University of California
Los Angeles, USA

Martin Trapp
KTH Royal Institute of Technology
Stockholm, Sweden

Abstract

While the Log-Sum-Exp (LSE) function is a foundational numerical primitive for stable log-domain computation in modern machine learning, its fast and efficient computation on generic platforms (*i.e.*, CPUs and GPUs) has been underexplored, potentially creating execution bottlenecks for large workloads. On GPU, the de facto implementation `torch.logsumexp` dispatches a chain of nine sub-kernels per call, incurring redundant High Bandwidth Memory (HBM) traffic and launch overhead. On CPU, `exp` and `log` operations are hardware-hungry, limiting the computing efficiency. To address these bottlenecks, we propose two kernels to significantly accelerate LSE computation on both GPUs and CPUs: **FuseLSE**, a single-pass fused kernel that evaluates LSE's `exp` and `log` functions on the GPU's Special Function Unit (SFU); and **IntLSE**, which further replaces the LSE computation with approximated integer arithmetic for CPU computation. Our experiments, validated by benchmarks across workload sizes and hardware platforms, show that both kernels are $\sim 10-11\times$ faster than PyTorch at small workloads and $\sim 6\times$ faster on larger ones on GPUs, where the speed-up is attributed to reduced launch overhead and HBM traffic. On CPU, where no SFU is available, LSE computation dominates the cost, allowing **IntLSE** to gain $\sim 3-6\times$ execution speed over **FuseLSE**.

1 Introduction

The Log-Sum-Exp (LSE) function is one of the most widespread numerical primitives in modern machine learning, whenever numerically stable logarithmic computation is required. For example, LSE is used to compute per-token log-probabilities of Large Language Models (LLMs) during reinforcement learning, where every token's forward pass calls one LSE over the vocabulary size [37]; to normalize particle weights at every time step of a particle filter computation [2, 16, 7]; to enable log-domain probabilistic inference, for instance in probabilistic circuits [23, 5, 19]; or to stabilize the Sinkhorn algorithm for entropy-regularized optimal transport [10, 29].

*Correspondence to: `lingyun.yao@aalto.fi`. Code: `https://github.com/lingyunyao/fuselse`

40th Conference on Neural Information Processing Systems (NeurIPS 2026).

Despite this ubiquity, LSE itself is rarely an optimization target on generic CPUs and GPUs, where efforts focused on dense matrix multiplications and standard reduction primitives. To illustrate this gap, PyTorch's `torch.logsumexp` function can be $\sim 6\times$ slower than the analogous `torch.sum` reduction over the same tensor on a given GPU (1.95 ms vs. 0.31 ms in Fig. 1). On GPUs, although log computation libraries exist, they either support only element-wise operations [3] based on Mitchell's approximation [21], or are not fully optimized for LSE [27, 35, 20, 8]. On CPUs, ML applications requiring LSE rely on generic libraries (`scipy`,`numpy`, `torch`) (Section 5.3), all of which implement LSE via the same multi-step primitives. However, LSE optimization has shown promising gains to push the efficiency of specialized hardware for AI inference, including efficient FPGA implementations [38], integer approximations on 16 nm CMOS [41, 42], or alternative arithmetic in place of log-domain math on dedicated hardware [31, 40, 24].

Considering the growing model sizes and interest around probabilistic and symbolic models that perform extensive log-space computation [11], LSE plays an ever-increasing role in optimizing their training and inference time. Hence, this work targets the optimization of LSE on both CPU and GPU by exploiting two key opportunities. First, current baseline implementations dispatch a chain of sub-kernels per call, each paying a fixed launch overhead. Second, four of these sub-kernels touch the full input tensor and round-trip it through High-Bandwidth Memory (HBM) roughly $6\times$ per call (Section 2.1). We instantiate two acceleration strategies as kernel replacements: **FuseLSE**, which fuses the sub-kernel chain to eliminate the launch overhead and reduce HBM traffic, and **IntLSE**, which further replaces the `exp` and `log` functions with integer approximations [41, 21].

In terms of results, our experiments show that both kernels achieve a $\sim 10-11\times$ speed-up over `torch.logsumexp` at small workloads and $\sim 6\times$ at large workloads, confirmed by single-kernel benchmarks and three ML applications. We trace this gain through a time-breakdown analysis across workload sizes, GPUs with varying HBM bandwidths, and input distributions. Our results show that the choice of arithmetic is a second-order factor on GPU: **FuseLSE** and **IntLSE** achieve nearly identical speed, and both outpace HBM transfers. On CPU, where no Special Function Unit (SFU) is available, the choice can no longer be ignored: **IntLSE**'s integer pipeline beats **FuseLSE** by 3–6$\times$ on the microbenchmark and `torch.logsumexp` by 2.8–5.2$\times$ for particle filters.

2 Preliminaries

The LSE function is defined as:

$$\mathrm{LSE}(x_1, \ldots, x_N) \triangleq c + \log\left(\sum_{i=1}^{N} e^{x_i - c}\right), \qquad \text{with} \quad c = \max_i x_i. \tag{1}$$

LSE arises whenever probabilities or scores must be aggregated in the logarithmic domain for numerical stability. We highlight three application domains where LSE is a potential computational bottleneck, each evaluated experimentally in Sections 5.1 to 5.3.

RLHF log-probability computation. Reinforcement learning from human feedback (RLHF) [18] methods such as PPO [30], and RLOO [1] require per-token log-probabilities $\log p(y_t \mid y_{<t}, x) = z_{y_t} - \mathrm{LSE}(z_1, \ldots, z_V)$ at every training step for both the policy and reference models. The vocabulary dimension V ranges from $32k$ (Llama-2) to $256k$ (Gemma), making the LSE reduction over V expensive. The HuggingFace TRL library [37] implements this via `selective_log_softmax`, in which `torch.logsumexp` accounts for a significant share of runtime.

Probabilistic circuits. Probabilistic circuits (PCs) [5] are a family of tractable probabilistic models that represent probability distributions as directed acyclic computation graphs with sum ($\oplus$) and product ($\otimes$) nodes. When computed in the log domain to avoid numerical underflow, product nodes become additions and sum nodes require LSE operations [23]. A single inference pass through a deep PC may invoke millions of LSE evaluations [19]. Here we use PyJuice [19], a GPU-accelerated library for PC inference and learning, as both inference and learning heavily rely on LSE.

Particle filtering. Particle filters [16, 7] maintain a weighted set of particles to approximate the posterior, for example, of Bayesian state-space models. Standard implementations require computing normalized log weights, which are obtained via $\log \tilde{w}_i = \log w_i - \mathrm{LSE}(\log w_1, \ldots, \log w_N)$, making one LSE reduction per time step a fixed cost of the filter loop. In this work, we focus on the bootstrap particle filter [12] (BPF), a widely used SMC instance, implemented in the particles library [6].

2.1 Baseline: PyTorch's `torch.logsumexp`

PyTorch's LSE implementation [26] follows the formulation of Eq. (1), *i.e.*, (i) compute $c = \max(x)$; (ii) subtract the max, $x - c$; (iii) exponentiate; (iv) sum along the reduction dimension; (v) take the log; (vi) add the max back. To check how this composes on the GPU, we trace a single call of `torch.logsumexp(x, dim=-1)` at $(B, N) = (512,\ 128256)$ on a V100 with `torch.profiler` (20 timed iterations, 50-step warm-up). The six-step recipe expands into a chain of nine CUDA sub-kernels shown in Fig. 1. Extra `abs`, `unary`, and `mask_fill` kernels handle the $\pm\infty$ corner cases of step (i) and are only visible as thin lines in the profile due to their negligible contributions. Nsight Compute shows that the following four sub-kernels dominate the runtime (`amax`, `sub`, `exp`, and `sum`, at 312, 655, 642, and 317 μs respectively), and these same four account for six (B, N)-sized HBM transfers: `amax` $1\times$ (read), `sub` $2\times$ (read+write), `exp` $2\times$ (read+write), and `sum` $1\times$ (read).

2.2 Existing accelerating methods

The `torch.compile` operation (TorchInductor [27]) fuses the `torch.logsumexp` operation chain at runtime into Triton [35] kernels. On the workload of Fig. 1, `torch.compile(torch.logsumexp)` delivers a $\sim 3\times$ speedup and cuts HBM traffic from $\sim 6\times$ to $\sim 2\times$ the input size by eliminating inter-operator HBM traffic. However, it inherits PyTorch's two-pass max-subtract semantics and pays recompilation cost on every new input shape. TensorFlow's `tf.math.reduce_logsumexp` [34] and JAX's `jax.nn.logsumexp` [14] define the same two-pass structure, which the XLA compiler [22] preserves when compiling them. Online softmax [20] computes LSE in a single streaming pass by carrying two scalars (max and exp-sum); FlashAttention [8] exploits this to fuse softmax with the matmuls of self-attention, CCE [39] fuses LSE with the LM-head matmul, PyJuice [19] performs the same fusion for dense PCs. We benchmarked against this matmul-LSE-fused method in Section 5.1, the LM-head setting where they apply. PyJuice's sparse mode [19], EinsumNetworks (einsum-based PC framework) [23], and particle filters [6] fall back to the two-pass max-subtract recipe in the absence of an upstream matmul; these settings require an optimized standalone LSE primitive.

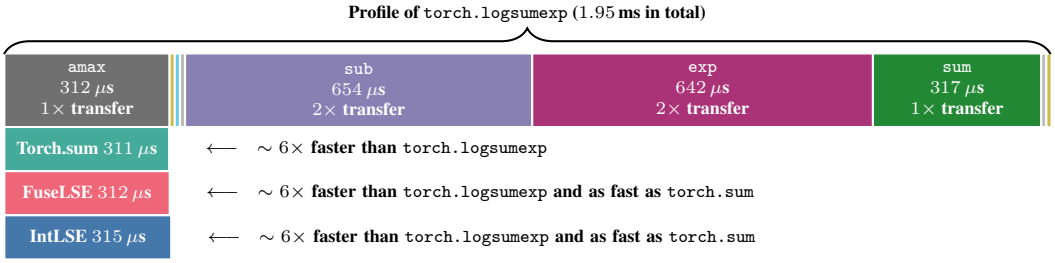


Figure 1: Per-call profile at $(B, N) = (512, 128256)$, on V100-PCIe-32 GB. Top row: `torch.logsumexp` with nine CUDA sub-kernels; the four heavy ops (`amax`, `sub`, `exp`, `sum`) are labeled with kernel name, GPU time, and HBM traffic as a multiple of the input volume, and the five narrow ops (`abs`, `unary`, `mask_fill`, `log`, `add`) are shown as thin coloured stripes. Bottom three rows: `torch.sum` for reference, **FuseLSE**, and **IntLSE** using only one kernel. Bar widths are linear in time. Per-launch times from `torch.profiler`; HBM traffic data from Nsight Compute.

3 Implementation of the proposed LSE acceleration kernels

The profile in Fig. 1 suggests two design opportunities: (1) fusing the nine kernel launches into one removes eight launches and reduces HBM traffic from $\sim 6\times$ to $\sim 1\times$ the input size, and (2) replacing `exp` and `log` by cheaper approximate operations, e.g., integer arithmetic [41]. These motivate our two kernels, **FuseLSE** (Section 3.1) and **IntLSE** (Section 3.2), whose details are outlined below.

3.1 FuseLSE: Single-Pass Fused Kernel

We base **FuseLSE** on the pair-wise form of log-sum-exp. Setting $N = 2$ in Eq. (1) with inputs x, y and $c = \max(x, y)$ gives

$$\mathrm{LSE}(x, y) = \max(x, y) + \log\big(1 + \exp(-|x-y|)\big), \tag{2}$$

since the two exponents become 0 and $-|x-y|$. As LSE is associative and commutative, applying Eq. (2) recursively in any order recovers the N-ary LSE. Note that each evaluation carries its own local max operation, with no upfront pass over the data to compute a global max. In exchange, **FuseLSE** evaluates $N-1$ `log` calls per row instead of one.

FuseLSE takes a float32 matrix $X \in \mathbb{R}^{B\times N}$ of base-e log-data (B rows, each of length N) and returns the B row-wise scalars in a single HBM pass. The kernel launches B blocks, each with 256 threads that cooperatively reduce one N-element row to a single scalar. Each thread keeps one running accumulator in a register throughout, initialized to -10^{30} and assigned the first valid input directly to avoid the NaN corner. Within a block, the 256 threads sweep the N-element row through `float4` (4 of float32 data) loads: each load delivers 4 float32 per thread, and a warp of 32 threads covers 128 float32 per step. With this, the kernel approaches peak HBM bandwidth. Each thread visits about $N/256$ floats and folds each into its register-resident accumulator via Eq. (2), evaluated through the SFU's of the GPU intrinsics `__expf` and `__logf`.

The 256 partials are reduced to one scalar via two stages of warp-shuffle butterflies: $32 \to 1$ within each warp, then $8 \to 1$ across warps through a 32-byte shared-memory buffer. Thread 0 writes the final scalar to global memory. Overall, the kernel reads each input element once and writes B output scalars, for a total HBM traffic of roughly $1\times$ the input size, against 6 times for `torch.logsumexp` (Section 2.1).

3.2 IntLSE: Approximate integer LSE

IntLSE preserves the structure of **FuseLSE** and replaces only the pair function in Eq. (2) with integer arithmetic, following Yao et al. [41]. Assuming $x \geq y$ and letting $z = y - x \leq 0$, the pair function becomes $\mathrm{LSE}(x,y) = x + \log_2(1+2^z)$. Define $f(z) \triangleq \log_2(1+2^z)$. Applying Mitchell's approximations [21], $2^v \approx 1+v$ and $\log_2(1+v) \approx v$ for $v \in [0,1)$, $f(z)$ collapses to a single addition and a right-shift,

$$\hat{f}(z) \triangleq (1+F_z) \gg (-I_z), \tag{3}$$

where $I_z \leq 0$ and $F_z \in [0,1)$ are the integer and fractional parts of z. The pair function reduces to $\mathrm{LSE}(x,y) \approx x + \hat{f}(z)$, computed with only integer adds and shifts. The error $\varepsilon(z) = f(z) - \hat{f}(z)$ is non-negative and periodic, peaking at each integer endpoint $z = -n$ with value $\log_2(1+2^{-n}) - 2^{-n}$ (Fig. 2 a,b); the global maximum ≈ 0.085 occurs at $n = 1$. Both $\hat{f}(z)$ and $\varepsilon(z)$ decay exponentially as $z \to -\infty$ (Fig. 2c); distant elements contribute negligibly. Assuming z is sampled with unit density on $(-\infty, 0]$, the expected accumulated error is $\int_{-\infty}^{0} \varepsilon(z)\, dz \approx 0.212$, independent of sample size (Section A.1), restricting the error accumulated over a chain of LSE operations. The actual error accumulated over a row depends on the input distribution, and is measured empirically in Table 4. A 64-entry constant-memory correction look-up table (LUT), stored in CUDA constant memory, indexed by the top 6 bits of $\hat{f}(z)$, further reduces the per-step peak error. The LUT lowers the maximum pointwise error from 0.085 to 0.0068 (Fig. 2 b).

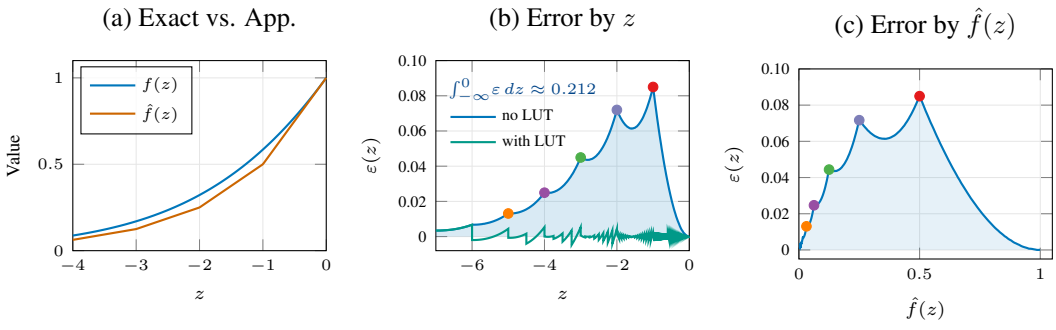


Figure 2: **(a)** Exact $f(z) = \log_2(1+2^z)$ and approximation $\hat{f}(z)$. **(b)** Per-step error $\varepsilon(z)$: colored markers show peak values at integer endpoints $z = -n$; shaded area equals $\int_{-\infty}^{0} \varepsilon\, dz \approx 0.212$. As $z \to -\infty$, $\varepsilon(z) \to 0$. **(c)** Parametric plot of $(\hat{f}(z), \varepsilon(z))$ as z sweeps from $-\infty$ to 0; each arc is one integer period, and matching colored markers identify the same $z = -n$ peaks as in (b).

Each input float x_i is converted to base-2 log space in Q16.16 fixed-point via $\mathrm{round}(x_i \cdot \log_2 e \cdot 2^{16})$. Pair-LSE then runs entirely in INT32: Eq. (3) is one add plus one shift, the LUT lookup is a single constant-memory load, and the warp- and block-level reductions are identical to **FuseLSE**. Thread 0 converts the final integer back to float32 base-e and writes it to global memory. The Q16.16 signed-integer representation bounds inputs to $|x| < 32{,}768$ in log-base-2, which comfortably covers typical ML log-values.

4 Individual kernel experiments and benchmarking

This section evaluates **FuseLSE** and **IntLSE** as standalone kernels in three versions: CUDA kernels for GPU, CPU implementations (multithreaded C++ for **FuseLSE**, hand-vectorized AVX2 C for **IntLSE**) that show when reducing arithmetic pays off on each platform, and reduced-precision CUDA variants. Table 2 lists all kernel variants and where they are used, and Table 3 gives the supported shapes, dtypes, edge cases, backward pass, and fallback rules of the robust versions.

4.1 Kernel benchmark

Setup. We set $B = 512$, N swept on a log grid from 2 to 128,256, inputs drawn from $\mathrm{U}(-1, 0)$, $\mathrm{U}(-100, 0)$, and $\mathrm{U}(-500, 0)$. We compare `torch.logsumexp`, **FuseLSE**, and **IntLSE** on GPU: V100-PCIe-32 GB (200 iters, 20 warm-ups) and CPU: Xeon Gold 6134 (50/30/20/10 iters scaled with N, 5 warm-ups). Each point is the median over 20 random seeds. Error is reported against the FP64 baseline, with a no-LUT **IntLSE** ablation in the error panels.

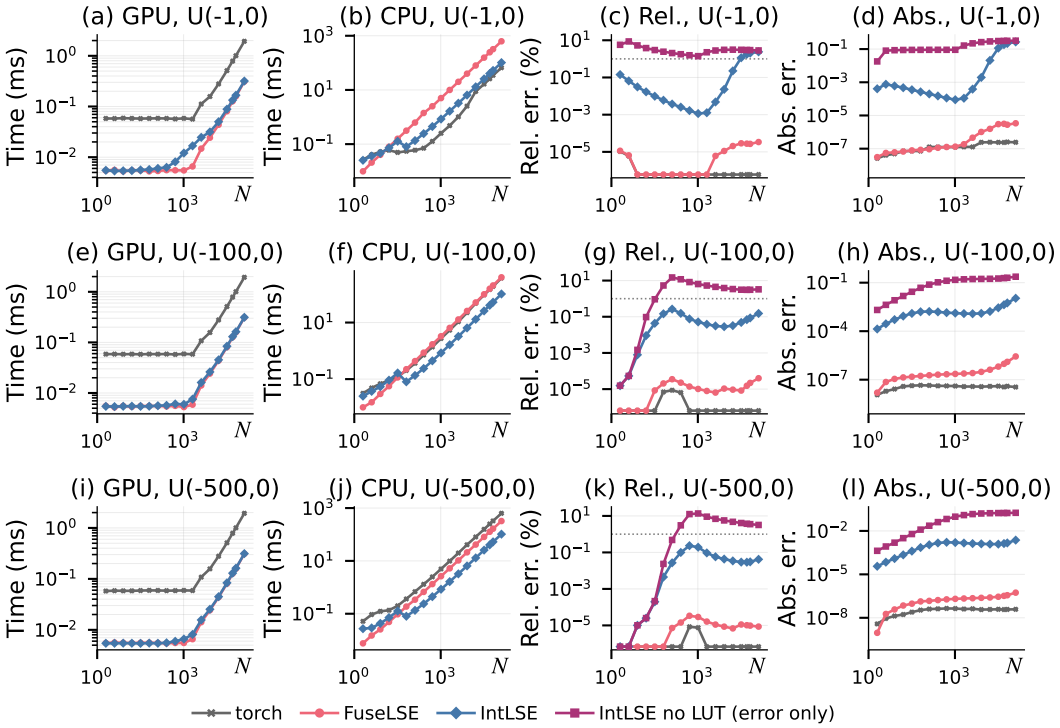


Figure 3: Batched LSE microbenchmark, $B = 512$, N swept 2 to 128,256. Rows: input ranges $\mathrm{U}(-1, 0)$, $\mathrm{U}(-100, 0)$, $\mathrm{U}(-500, 0)$. Columns: GPU per-call time (V100), CPU per-call time (Xeon Gold 6134), median relative error (%), and mean absolute error vs. an FP64 inline reference; the no-LUT **IntLSE** ablation is shown only in the error columns.

Results. On GPU (Fig. 3, col. 1) **FuseLSE** and **IntLSE** speed are within 1% of each other and beat `torch.logsumexp` by $\sim 10-11\times$ for $N \leq 1024$ and $\sim 6\times$ for $N \geq 8192$, settling at $6.24\times$ /$6.18\times$ at $N = 128{,}256$ (1.94 ms vs. 0.310/0.313 ms). On CPU (col. 2) **IntLSE** speed beats **FuseLSE** by $3.1-6.1\times$; against torch the gap is range-dependent: $1.54\times$ slower on $\mathrm{U}(-1, 0)$, $3.66\times$ faster on $\mathrm{U}(-100, 0)$, and $6.15\times$ faster on $\mathrm{U}(-500, 0)$ at $N = 128{,}256$. **FuseLSE** matches torch at

float precision (median rel. err. $<10^{-4}\%$). **IntLSE** with the LUT stays below 0.2% on the wider ranges (0.15% and 0.04% at $N = 128{,}256$ for $\mathrm{U}(-100, 0)$ and $\mathrm{U}(-500, 0)$) and grows to 2.38% on $\mathrm{U}(-1, 0)$; the no-LUT ablation saturates around ~ 0.2 in absolute error at large N, and $\sim 20-80\times$ above the LUT-corrected variant on the wider ranges.

Discussion. The two GPU regimes match the profile in Section 2.1: $\sim 10-11\times$ at small N comes from collapsing nine kernel launches into one, and $\sim 6\times$ at large N comes from cutting torch's HBM traffic from $6\times$ the input volume to $1\times$. **FuseLSE** and **IntLSE** coincide on GPUs because in both regimes the arithmetic cost of the pair function is hidden, so replacing `exp`/`log` with integer ops has no effect on runtime. On CPU, there is no SFU and the kernel becomes compute-bound, so `exp`/`log` loses to **IntLSE**'s integer arithmetic. The range-dependent gap against torch reflects how torch's Single Instruction Multiple Data (SIMD) pipeline responds to different input values, whereas the runtime of **IntLSE**'s integer arithmetic is independent of the input values. On all but the narrowest range, **IntLSE** is faster than torch even though it uses a single thread against torch's four.

Table 4 extends this comparison to additional input distributions, including extreme values: **FuseLSE**'s p95 error stays within 4.5×10^{-5}, the same order as torch's own FP32 error, and **IntLSE**'s within 4.2×10^{-3}. With matched single threads (Table 5), **IntLSE** runs in a constant 27.6 ms on every finite input, $1.2-1.3\times$ faster than torch when no input underflows and $10.7-22.0\times$ faster otherwise.

4.2 Cross-GPU time decomposition

Setup. To verify the regime analysis of Section 4.1 and check how the speedup generalizes across GPUs, we decompose where each kernel's GPU time goes on V100-PCIe-32 GB (900 GB/s peak HBM) and H100-SXM5-80 GB (3350 GB/s) using NVIDIA Nsight Compute. We profile the three kernels at $B = 512$, across $N \in \{256, 1024, 4096, 16384, 65536, 128256\}$, with 10 launches per kernel, and report (Fig. 4) total GPU time, HBM lower bound (`dram__bytes.sum` / peak HBM bandwidth), Streaming Multiprocessor (SM)-active compute time ($\sum_k \mathrm{gpu_time}_k \cdot \mathrm{sm_pct}_k$), and a launch reference (total GPU time at the smallest profiled $N = 256$). Inputs use $\mathrm{U}(-100, 0)$.

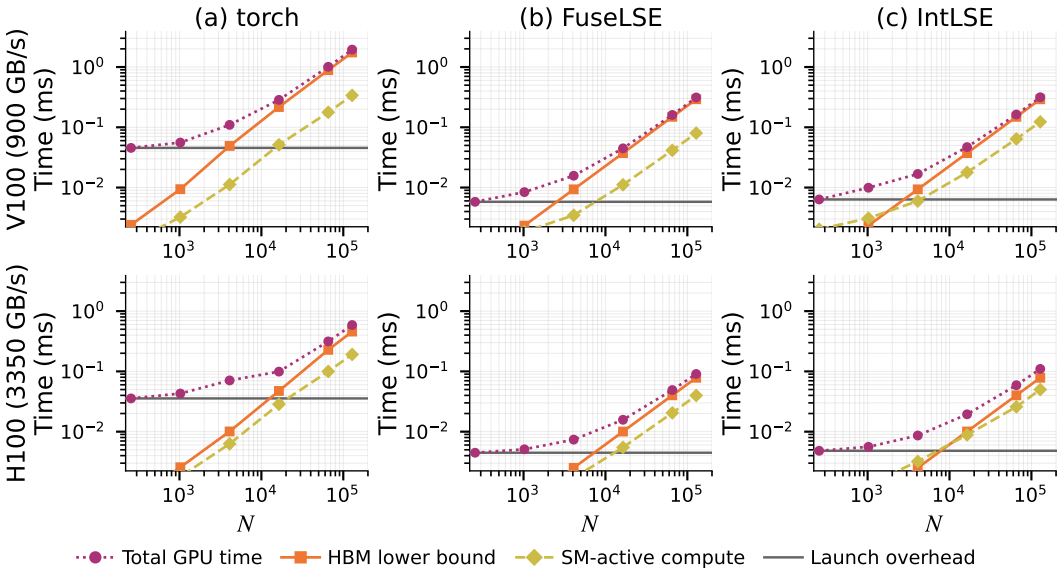


Figure 4: Per-call time decomposition of `torch.logsumexp`, **FuseLSE**, and **IntLSE** (columns (a)/(b)/(c)) on V100 (top) and H100 (bottom). All four lines come from a single ncu run: total GPU time, HBM lower bound, SM-active compute, and a launch reference (total time at the smallest $N = 256$). Total time approaches the HBM lower bound at large N, while SM-compute stays below it across all kernels.

Results. At small N, the total GPU time of all three kernels sits well above both the HBM lower bound and the SM-compute line, tracking the launch reference. At large N it converges to each kernel's HBM lower bound on both V100 and H100, with SM-compute staying well below; `torch.logsumexp`

moves $\sim 6\times$ more bytes than both kernels. **IntLSE**'s integer pair costs $\sim 50\%$ more SM-compute than **FuseLSE** on V100 (123 vs. 80 μs at $N = 128{,}256$) and $\sim 26\%$ more on H100 (50 vs. 40 μs).

Discussion. For small N, the gap to the launch reference is due to kernel fusion; for large N, the $\sim 6\times$ speedup comes from the amount of bytes moved, since all three kernels saturate HBM well above their SM-compute lines. **IntLSE**'s extra SM-compute over **FuseLSE** reflects the absence of an integer advantage on GPUs. Jia et al. [15] report V100 INT32 and FP32 share latency of 4-cycle. Even on older Pascal GPUs, SFU instructions only (`__expf`, `__logf`) sit at ~ 14 cycles, while **IntLSE** adds ALU cost on top. On CPUs, FP32 ADD latency is around 4× that of INT32 ADD of 1 cycle [9], single `log` can cost up to 40 cycles [32]. **FuseLSE** is therefore the right choice on modern GPUs; **IntLSE**'s value is restricted to CPU and platforms without a fast SFU.

We further compare against compiler-fused baselines, PyTorch 2.8 `torch.compile` and JAX 0.7.1 (XLA), on V100, H100, H200, and the newest B300, excluding compilation time (Table 6). The speedup persists on newer GPUs: **FuseLSE** is $6.3-6.9\times$ faster than eager PyTorch on all four GPUs and $1.8-2.1\times$ faster than both compilers; on B300, Triton and XLA cannot yet target `sm_103`. Nsight Compute explains the gap (Table 7). Both compilers keep the two-pass structure and move $1.9-2.0\times$ the input volume per call, versus $\sim 6\times$ for eager PyTorch and $1.01\times$ for **FuseLSE**. The extra `log` calls do not offset this saving: the pair-wise `exp`/`log` work and the input traffic both scale as $O(N)$, so **FuseLSE**'s arithmetic intensity is constant in N, and although its SFU utilization rises from 28% (V100) to 46% (H100) and 66% (H200) as HBM bandwidth grows, at most 0.21% of cycles stall waiting for a math pipe.

4.3 Reduced-bit LSE

As seen from Section 4.1 and 4.2, we address the launch overhead by using a single kernel launch in **FuseLSE**, yet the HBM transfer bottleneck remains. To address this, we reduce the input bit-width and halve the bytes that must cross HBM. We evaluate two 16-bit variants alongside the 32-bit baselines: FuseLSE16 (FP16 storage, FP32 internal accumulation) and IntLSE-Q10.6 (INT16 Q10.6 storage, INT32 internal accumulation; with 10 integer and 6 fractional bits).

Setup. We set $B = 512$, N from 2 to 128,256, inputs $\mathrm{U}(-100, 0)$. We compare the time and error of `torch.logsumexp`, **FuseLSE**, FuseLSE16, and IntLSE-Q10.6 on V100-PCIe-32 GB, Times are from CUDA events averaged over 1000 iterations after 100 warm-ups, accuracy is averaged over 20 seeds against FP64 reference, as seen in Fig. 5.

Results. For small N (≤ 1024) all three kernels are launch-overhead-bound and tie at $\sim 10-11\times$. At $N = 128{,}256$ the three kernels reach $6.25\times$ (**FuseLSE**), $12.06\times$ (FuseLSE16), and $10.73\times$ (IntLSE-Q10.6) over `torch.logsumexp`. The two 16-bit variants split on accuracy. FuseLSE16's mean absolute error stays below 3×10^{-3} and decreases with N, tracking `torch.logsumexp` above $N \approx 10^4$. IntLSE-Q10.6's error instead grows with N, reaching $\sim 1\%$ at $N = 128{,}256$, four orders of magnitude above FuseLSE16.

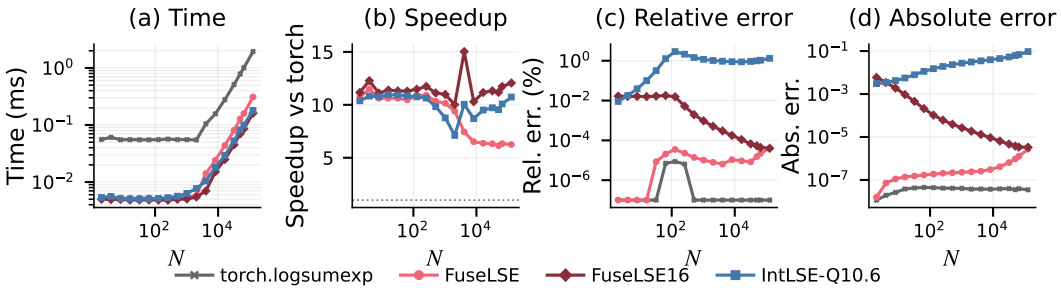


Figure 5: Reduced-bit `logsumexp` on V100-PCIe ($B = 512$, $x \sim \mathrm{U}(-100, 0)$). **(a)** Per-call time. **(b)** Speedup of the three fused variants vs. `torch.logsumexp`. **(c)** Median relative error against FP64. **(d)** Mean absolute error against FP64.

Discussion. On this V100 workload, FuseLSE16 dominates: same HBM saving, near FP32 accuracy. IntLSE-Q10.6 captures the bandwidth speedup but is preferable only on hardware without a fast SFU path, where the integer pipeline becomes the actual compute floor. We do not pursue either variant in the downstream applications, since surrounding operations run in FP32.

5 Applications

We benchmark our kernels in three applications against the baselines used in practice. RLHF log-probability computation (Section 5.1, GPU) replaces `torch.logsumexp` on dense (B, V) vocabulary reductions. Probabilistic-circuit inference (Section 5.2, GPU) replaces PyJuice's default and sparse-Triton sum-layer kernels with sparse variants of **FuseLSE**/**IntLSE**. In both RLHF and Probabilistic-circuit, **FuseLSE** is our recommended kernel on GPU. Particle filtering (Section 5.3, CPU) is commonly deployed on CPUs, where `exp`/`log` are expensive without an SFU. We replace the NumPy LSE with the CPU version of **IntLSE**.

5.1 RLHF log-probability computation on GPU

Reinforcement-learning [18] of LLMs computes per-token log-probabilities $z_{y_t} - \mathrm{LSE}(z_1, \ldots, z_V)$ over a vocabulary of size V, so each token costs one V-element LSE. The HuggingFace TRL library [37] packages this as `selective_log_softmax` (SLS), the operation we benchmark here.

Setup. We benchmark across four vocabulary sizes spanning the modern open-model landscape: Llama-2 [36] ($V = 32$K), GPT-2 [28] ($V = 50$K), Llama-3 [13] ($V = 128$K), and Gemma [33] ($V = 256$K). Since SLS's cost depends only on the batch size, sequence length, and vocabulary size, not on the backbone's parameter count, we use a single GPT-2 (124 M) backbone for the forward pass and swap only the LM head to match each target vocabulary size. For $V = 50$K we keep the pretrained head; for the other sizes we use a random projection of matching shapes. The GPT-2 backbone runs a forward pass on Anthropic HH-RLHF text [4] tokenized to fixed length $T = 1024$, producing logits of shape $(B, T-1, V)$; we then time TRL's `selective_log_softmax`, which reduces over the vocabulary axis V at every position. **FuseLSE** and **IntLSE** are plugged into TRL by reassigning `torch.logsumexp` at runtime to a wrapper that forwards to our kernel and falls back to PyTorch otherwise; TRL itself is unmodified. Batch sizes $B = (8, 8, 4, 2)$ scale with V to fit GPU memory. We run 50 timed iterations after 3 warm-ups on V100-PCIe-32 GB.

Table 1: Per-call timings of the GPT-2 LM-head + SLS stage (V100, B=8, T=1024, V=50K).

	`torch`	`torch.compile`	FuseLSE	IntLSE	`cce`	`cce` (`torch.compile`)
LM-head + SLS (ms)	59.12	51.64	49.20	49.33	133.11	**48.74**
SLS (ms)	12.34	*fused*	**2.01**	2.07	*fused*	*fused*
Cold compile (s)	–	4.56	–	–	0.97	1.74

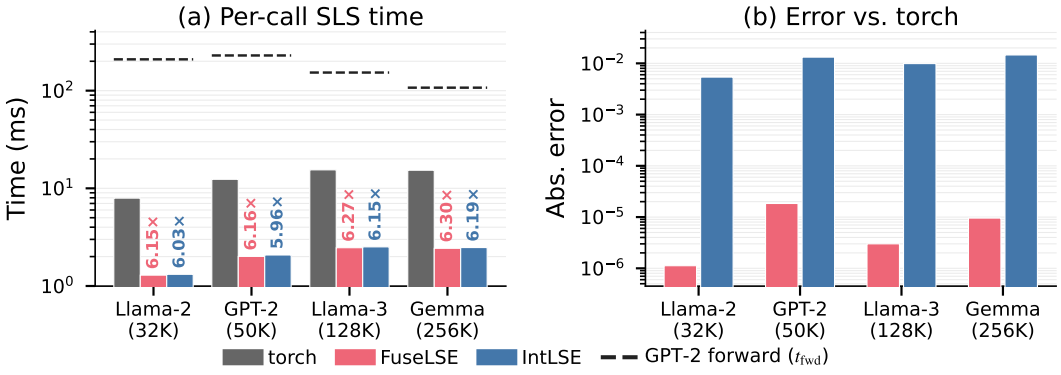


Figure 6: RLHF SLS benchmark on V100-PCIe, GPT-2 backbone, $T = 1024$, batch sizes $B = (8, 8, 4, 2)$. **(a)** Per-call SLS time (log-scale y-axis): each call processes the full $(B, T-1, V)$ logits tensor and produces $B \cdot (T-1)$ per-token log-probabilities. Speedup over `torch.logsumexp` annotated above each **FuseLSE** / **IntLSE** bar in the matching colour; the dashed reference is the full GPT-2 forward time at each V. **(b)** Absolute reduction error vs. `torch.logsumexp`.

Results. Both **FuseLSE** and **IntLSE** achieve 5.96–6.30× speedup over `torch.logsumexp` across all four vocabularies (Fig. 6, panel a), with the two kernels within ∼3.3% of each other in every column. The dashed reference is the GPT-2 forward time at the same shape; SLS contributes to

a significant fraction of time ($\sim$4%–14%) and grows with V, so the kernel speedup gets more important as open-model vocabularies expand. On accuracy (Fig. 6, panel b), **FuseLSE** matches `torch.logsumexp` to 1.1×10^{-6}–1.9×10^{-5} absolute error in the logsumexp output; **IntLSE** stays within 5.4×10^{-3}–1.5×10^{-2}. RLHF is the only setting in this paper where prior matmul-LSE-fused methods [39] apply. On the GPT-2 LM-head + SLS benchmark (Table 1), **FuseLSE** and **IntLSE** deliver a $\sim6\times$ speedup on the SLS step and $1.20\times$ on the LM-head + SLS stage, capped by the shared 47-ms matmul. They pay no recompile cost, in contrast to `torch.compile` and `cce(torch.compile)` (`torch.compile` based implementation), which require $4.56/1.74$ s of cold compile on every new input shape.

5.2 Probabilistic circuit inference on GPU

We evaluate **FuseLSE** and **IntLSE** for probabilistic circuits, where each sum layer computes LSE over weighted child log-likelihoods.

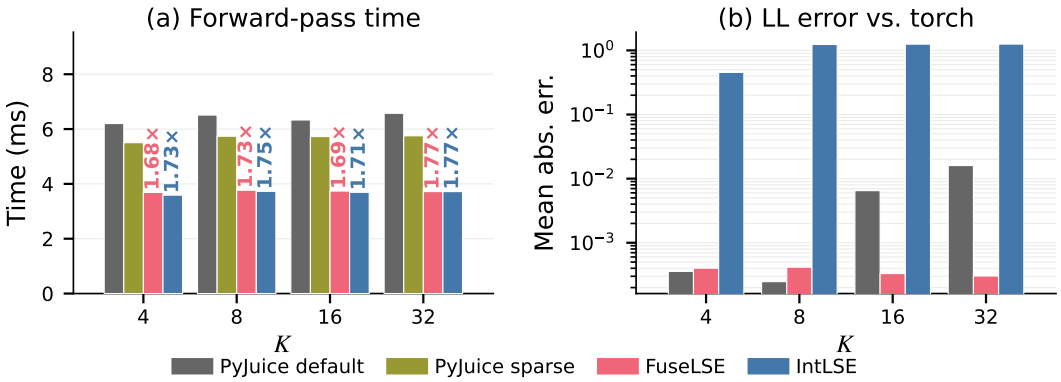


Figure 7: PyJuice HCLT inference on MNIST, batch size 512, V100-PCIe-32 GB. **(a)** Forward-pass time per batch vs. number of latents K for four implementations: PyJuice default (mixed-mode), PyJuice sparse (forced sparse), **FuseLSE**, and **IntLSE**. Speedup over PyJuice default is annotated above each **FuseLSE** / **IntLSE** bar in matching colour. **(b)** Per-sample mean absolute log-likelihood error against a `torch.logsumexp` reference (log axis), computed on the full MNIST validation set.

Setup. We build Hidden Chow–Liu Tree (HCLT) circuits on MNIST (28×28 pixels, 784 features) with PyJuice [19]. We sweep the latent-state count $K\in\{4,8,16,32\}$, producing circuits of increasing width. In PyJuice's sparse mode, each sum layer dispatches a chain of torch operations that performs a two-pass max-shift logsumexp ($\max$, then $\exp$, $\sum$, $\log$). We replace this with our fused kernel by reusing the **IntLSE** and **FuseLSE** pair primitives, but bundle a sparse gather and log-weight add inside one kernel pass. All layers are forced into sparse mode for our kernels, so the replacement covers the full circuit. Forward-pass time is averaged over 20 batches at batch size 512 after a 5-batch warm-up on Tesla V100-PCIe-32 GB. For accuracy, we report the per-sample mean absolute log-likelihood error against a `torch.logsumexp` reference, computed on the full MNIST validation set ($n=9{,}728$ samples).

Results. For throughput, **FuseLSE** and **IntLSE** achieve a $\sim1.7\times$ forward-pass speedup over the PyJuice default mode for all four K (Fig. 7, panel a), with the two kernels within $\sim3\%$ of each other in every column. Separate LSE speedup is invisible since the call is integrated sparsely inside the forward pass. For accuracy (Fig. 7, panel b), **FuseLSE** stays within 3.0–4.2×10^{-4} absolute log-likelihood error, while **IntLSE** sits at 0.45–1.26 nats, under the per-sample LL of ≈-4500 nats. PyJuice's default LL relative to torch is shown for context, comparable to our **FuseLSE** at small K but growing to 1.6×10^{-2} at $K=32$.

5.3 Particle filter on CPU

We evaluate **IntLSE** for the particle filter [7, 6] on CPU, where particle filters are commonly deployed. Each time step issues one length N LSE call to normalise particle weights, so an N-particle, T-step pass executes T such calls.

Setup. We use the univariate stochastic-volatility model [17] (default parameters from [25]) as packaged by [7], with $T = 200$ observations. We sweep $N \in \{50, 100, 200, 500, 1000, 2000\}$ on Intel Xeon Gold 6134, compared across: `scipy.special.logsumexp`, the NumPy (actual LSE path in [7]), `torch.logsumexp`, and **IntLSE**. For accuracy, we report the mean absolute error against the FP64 LSE baseline, averaged over T. For timing, we report the per-call median wall-clock time, 100 warm-ups followed by 500 timed calls.

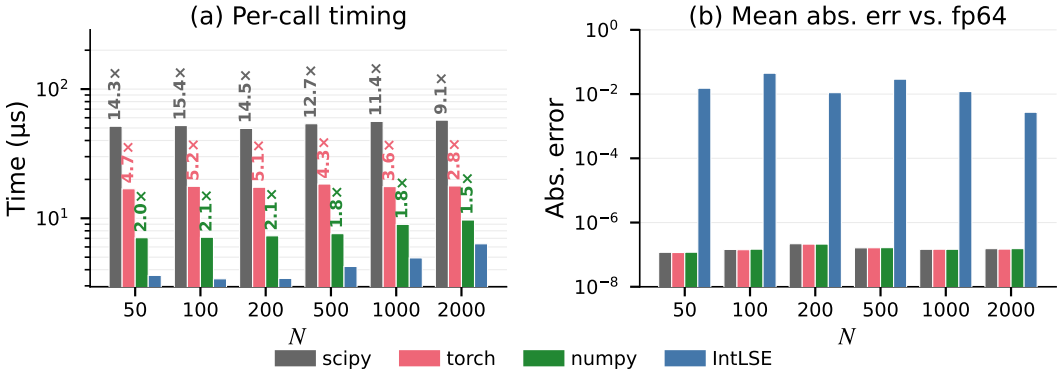


Figure 8: Particle-filter LSE primitive for the stochastic-volatility model of [25] with T=200 observations, evaluated on Intel Xeon Gold 6134 with one thread for all methods. Inputs are the FP32 log weights encountered along the filter trajectory. **(a)** Median latency of one LSE call versus the number of particles N; labels report the speedup of **IntLSE** over each baseline. **(b)** Mean absolute LSE error of SciPy, PyTorch, NumPy, and **IntLSE** relative to an FP64 reference, averaged over all T calls. The figure reports primitive-level latency and accuracy, not complete-filter runtime.

Results. **IntLSE** is the fastest primitive at every N (Fig. 8, panel a): 9.1–15.4× faster than scipy, 1.5–2.1× faster than NumPy, and 2.8–5.2× faster than torch. Its absolute error stays $\sim 10^{-2}$ across the N, making **IntLSE** a viable LSE primitive for CPU particle filters.

6 Conclusion and discussion

We presented **FuseLSE** and **IntLSE**, two kernels that serve as replacements for `torch.logsumexp`, delivering $\sim 6-12\times$ speedups across microbenchmarks and three ML applications. Our experiments show that for memory-bound reductions like LSE and softmax on GPU, kernel fusion is the critical lever, cutting redundant HBM traffic that arithmetic optimization cannot reach. On platforms without a fast SFU (CPUs, microcontrollers, FPGAs), computation becomes the bottleneck and integer arithmetic wins.

Broader impact. For ML systems, redundant HBM traffic is the dominant cost of memory-bound reductions, and the analyses and kernels we present generalize to similar operations. For hardware designers, the lesson is that HBM bandwidth matters more than peak FLOPs for GPUs, while fast integer pipelines should still be exposed, not only on SFU-less platforms but also on GPUs. More fundamentally, modern GPUs are heavily optimized for dense matrix operations, a choice that suits today's LLMs but may not extend to future architectures.

Limitations. **IntLSE**'s integer approximation incurs a cumulative error that may be unsuitable for the most accuracy-critical workloads. Our reduced-precision variants (FuseLSE16, IntLSE-Q10.6) capture additional bandwidth savings but are not yet integrated into the application benchmarks.

Acknowledgements

This work has been funded by the Research Council of Finland through the FerrARI project (Grant Number 359046). We acknowledge the computational resources provided by the Aalto Science-IT project. This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.

References

[1] Arash Ahmadian, Chris Cremer, Matthias Gallé, Marzieh Fadaee, Julia Kreutzer, Olivier Pietquin, Ahmet Üstün, and Sara Hooker. Back to basics: Revisiting reinforce-style optimization for learning from human feedback in llms. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 12248–12267, 2024.

[2] Christophe Andrieu, Arnaud Doucet, and Roman Holenstein. Particle Markov chain Monte Carlo methods. *Journal of the Royal Statistical Society Series B: Statistical Methodology*, 72 (3):269–342, 2010.

[3] Mark Arnold. XLNSCUDA: A CUDA package for Logarithmic Number System experimentation. Available at `https://github.com/xlnsresearch/xlnscuda`, 2025.

[4] Yuntao Bai, Andy Jones, Kamal Ndousse, Amanda Askell, Anna Chen, Nova DasSarma, Dawn Drain, Stanislav Fort, Deep Ganguli, Tom Henighan, et al. Training a helpful and harmless assistant with reinforcement learning from human feedback. *arXiv preprint arXiv:2204.05862*, 2022.

[5] YooJung Choi, Antonio Vergari, and Guy Van den Broeck. Probabilistic circuits: A unifying framework for tractable probabilistic models. Technical report, UCLA, 2020. Available at `http://starai.cs.ucla.edu/papers/ProbCirc20.pdf`.

[6] Nicolas Chopin. Sequential Monte Carlo in python. `https://github.com/nchopin/particles`.

[7] Nicolas Chopin and Omiros Papaspiliopoulos. *An introduction to sequential Monte Carlo*, volume 4. Springer, 2020.

[8] Tri Dao, Dan 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 (NeurIPS)*, 2022.

[9] Agner Fog. *Instruction tables: lists of instruction latencies, throughputs and micro-operation breakdowns for Intel*, 2014. Available at `http://www.agner.org/optimize/instruction_tables.pdf`.

[10] Peyré Gabriel and Cuturi Marco. Computational optimal transport with applications to data sciences. *Foundations and Trends in Machine Learning*, 11(5-6):355–607, 2019.

[11] Artur d'Avila Garcez and Luis C Lamb. Neurosymbolic ai: The 3 rd wave. *Artificial Intelligence Review*, 56(11):12387–12406, 2023.

[12] Neil J Gordon, David J Salmond, and Adrian FM Smith. Novel approach to nonlinear/non-Gaussian Bayesian state estimation. *IEE Proceedings F - Radar and Signal Processing*, 140(2): 107–113, 1993.

[13] 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.

[14] JAX Developers. JAX documentation: `jax.nn.logsumexp`. Available at `https://docs.jax.dev/en/latest/_autosummary/jax.nn.logsumexp.html`, 2025.

[15] Zhe Jia, Marco Maggioni, Benjamin Staiger, and Daniele P Scarpazza. Dissecting the NVIDIA volta gpu architecture via microbenchmarking. Technical report, Citadel, 2018. Available at `https://arxiv.org/pdf/1804.06826`.

[16] Adam Johansen. A tutorial on particle filtering and smoothing: Fifteen years later. 2009.

[17] Sangjoon Kim, Neil Shephard, and Siddhartha Chib. Stochastic volatility: Likelihood inference and comparison with ARCH models. *The Review of Economic Studies*, 65(3):361–393, 1998. doi: 10.1111/1467-937X.00050.

[18] Nathan Lambert. Reinforcement learning from human feedback. *arXiv preprint arXiv:2504.12501*, 2025.

[19] Anji Liu, Kareem Ahmed, and Guy Van den Broeck. Scaling tractable probabilistic circuits: A systems perspective. In *Proceedings of the 41th International Conference on Machine Learning (ICML)*, 2024.

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

[21] John N Mitchell. Computer multiplication and division using binary logarithms. *IRE Transactions on Electronic Computers*, EC-11(4):512–517, 1962.

[22] OpenXLA Project. XLA: An open-source machine learning compiler. Available at `https://openxla.org/xla`, 2025.

[23] Robert Peharz, Steven Lang, Antonio Vergari, Karl Stelzner, Alejandro Molina, Martin Trapp, Guy Van den Broeck, Kristian Kersting, and Zoubin Ghahramani. Einsum networks: Fast and scalable learning of tractable probabilistic circuits. In *International Conference on Machine Learning*, pages 7563–7574. PMLR, 2020.

[24] Karthekeyan Periasamy, Jelin Leslin, Aleksi Korsman, Lingyun Yao, and Martin Andraud. Autopc: an open-source framework for efficient probabilistic reasoning on fpga hardware. In *2024 22nd IEEE Interregional NEWCAS Conference (NEWCAS)*, pages 21–25. IEEE, 2024.

[25] Michael K. Pitt and Neil Shephard. Filtering via simulation: Auxiliary particle filters. *Journal of the American Statistical Association*, 94(446):590–599, 1999.

[26] PyTorch Foundation. PyTorch documentation: `torch.logsumexp`. Available at `https://docs.pytorch.org/docs/stable/generated/torch.logsumexp.html`, 2024.

[27] PyTorch Team. TorchInductor: A PyTorch-native compiler with define-by-run IR. Available at `https://pytorch.org/docs/stable/torch.compiler.html`, 2023.

[28] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[29] Bernhard Schmitzer. Stabilized sparse scaling algorithms for entropy regularized transport problems. *SIAM Journal on Scientific Computing*, 41(3):A1443–A1481, 2019.

[30] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[31] Nimish Shah, Laura Isabel Galindez Olascoaga, Shirui Zhao, Wannes Meert, and Marian Verhelst. Dpu: Dag processing unit for irregular graphs with precision-scalable posit arithmetic in 28 nm. *IEEE Journal of Solid-State Circuits*, 57(8):2586–2596, 2021.

[32] Naoki Shibata and Francesco Petrogalli. Sleef: A portable vectorized library of c standard mathematical functions. *IEEE Transactions on Parallel and Distributed Systems*, 31(6):1316–1327, 2019.

[33] Gemma Team, Thomas Mesnard, Cassidy Hardin, Robert Dadashi, Surya Bhupatiraju, Shreya Pathak, Laurent Sifre, Morgane Rivière, Mihir Sanjay Kale, Juliette Love, et al. Gemma: Open models based on Gemini research and technology. *arXiv preprint arXiv:2403.08295*, 2024.

[34] TensorFlow Developers. TensorFlow documentation: `tf.math.reduce_logsumexp`. Available at `https://www.tensorflow.org/api_docs/python/tf/math/reduce_logsumexp`, 2025.

[35] 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*, pages 10–19, 2019.

[36] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

[37] Leandro von Werra, Younes Belkada, Lewis Tunstall, Edward Beeching, Tristan Thrush, Nathan Lambert, Shengyi Huang, Kashif Rasul, and Quentin Gallouédec. TRL: Transformers Reinforcement Learning, 2020. URL `https://github.com/huggingface/trl`.

[38] Lukas Weber, Lukas Sommer, Jens Oppermann, Alejandro Molina, Karl Kersting, and Andreas Koch. Resource-efficient logarithmic number scale arithmetic for spn inference. In *International Conference on Field-Programmable Technology*, pages 251–254. IEEE, 2019.

[39] Erik Wijmans, Brody Huval, Alexander Hertzberg, Vladlen Koltun, and Philipp Krähenbühl. Cut your losses in large-vocabulary language models. *arXiv preprint arXiv:2411.09009*, 2024.

[40] Lingyun Yao, Martin Trapp, Jelin Leslin, Gaurav Singh, Peng Zhang, Karthekeyan Periasamy, and Martin Andraud. On hardware-efficient inference in probabilistic circuits. In *Proceedings of the Fortieth Conference on Uncertainty in Artificial Intelligence*, volume 244 of *Proceedings of Machine Learning Research*, pages 3979–3996. PMLR, 15–19 Jul 2024.

[41] Lingyun Yao, Shirui Zhao, Martin Trapp, Jelin Leslin, Marian Verhelst, and Martin Andraud. LogSumExp: Efficient approximate logarithm acceleration for embedded tractable probabilistic reasoning. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 2025.

[42] Lingyun Yao, Shirui Zhao, Marian Verhelst, and Martin Andraud. Einchip: A 4.6/1.5 tops/w 6/24b log-compute einsum-based accelerator for neuro symbolic ai. In *2026 IEEE Custom Integrated Circuits Conference (CICC)*, pages 1–4. IEEE, 2026.

A Technical appendices and supplementary material

A.1 IntLSE Approximation-error analysis

We split the expected accumulated error $\int_{-\infty}^{0} \varepsilon(z)\,dz$ in Section 3.2, with $\varepsilon(z) = f(z) - \hat{f}(z)$, into the exact part (I), $f(z) = \log_2(1+2^z)$ and the approximation (II), $\hat{f}(z) = (1+F_z)\,2^{I_z}$:

$$\int_{-\infty}^{0} \varepsilon(z)\,dz = \underbrace{\int_{-\infty}^{0} \log_2(1+2^z)\,dz}_{\text{(I)}} - \underbrace{\int_{-\infty}^{0} (1+F_z)\,2^{I_z}\,dz}_{\text{(II)}}. \tag{4}$$

For (II), summing over periods with $t = z + n \in [0, 1)$,

$$\text{(II)} = \sum_{n=1}^{\infty} 2^{-n} \int_0^1 (1+t)\,dt = \tfrac{3}{2} \sum_{n=1}^{\infty} 2^{-n} = \tfrac{3}{2}. \tag{5}$$

For (I), substituting $u = 2^z$ and using $\int_0^1 \ln(1+u)/u\,du = \pi^2/12$,

$$\text{(I)} = \frac{1}{(\ln 2)^2} \int_0^1 \frac{\ln(1+u)}{u}\,du = \frac{\pi^2}{12(\ln 2)^2}.$$

Subtracting (II) from (I) yields

$$\int_{-\infty}^{0} \varepsilon(z)\,dz \;=\; \frac{\pi^2}{12(\ln 2)^2} - \frac{3}{2} \;\approx\; 0.212. \tag{6}$$

A.2 Additional experimental results

Table 2: All kernels, organised by variant and arithmetic, with the sections where they are used.

	FuseLSE (float `exp/log`)		IntLSE (integer + LUT)	
Variant	Kernel	Used in	Kernel	Used in
GPU, FP32	FuseLSE	§4.1–§4.3, §5.1	IntLSE	§4.1–§4.2, §5.1
GPU, 16-bit	FuseLSE16	§4.3	IntLSE-Q10.6	§4.3
GPU, sparse PC	FuseLSE-sparse	§5.2	IntLSE-sparse	§5.2
CPU	FuseLSE (CPU)	§4.1	IntLSE (CPU)	§4.1, §5.3
Robust API	FuseLSE-robust (GPU)	Table 3	IntLSE-robust (CPU)	Table 3

Table 3: Edge-case handling, backward support, and fallback rules of the robust kernels. copy: contiguous copy; →ref: redirected to the reference implementation (PyTorch or NumPy); ×: unsupported. Overhead vs. the non-robust kernel at N=128,256 (GPU: V100, B=512; CPU: Xeon Gold 6148, one thread, B=128).

	Axis		Layout		Dtype	Output for			Backward	Overhead
Kernel	Other	Multi	Non-contig.	$N \le 1$	Others	All $-\infty$	$+\infty$	NaN		
FuseLSE-robust	→ref	×	copy	→ref	→ref	$-\infty$	$+\infty$	NaN	✓	+3.2%
IntLSE-robust	→ref	×	copy	→ref	→ref	$-\infty$	→ref	→ref	→ref	+7.4%

Table 4: Accuracy across input distributions: p95 absolute error vs. FP64 at N=128,256, B=128, 3 seeds. **FuseLSE** on V100, **IntLSE** on CPU; torch FP32 shown for reference. †Robust kernels.

Input	torch (FP32)	FuseLSE	IntLSE
$\mathrm{U}(-1, 0)$	7.0e-7	4.1e-6	1.4e-4
$\mathrm{U}(-100, 0)$	2.6e-7	3.0e-6	1.4e-3
$\mathrm{U}(-500, 0)$	2.5e-7	8.7e-7	1.7e-3
$\mathcal{N}(0, 1)$	6.9e-7	8.4e-6	6.5e-4
$\mathcal{N}(0, 20^2)$	3.7e-6	7.7e-6	4.2e-3
$\mathrm{U}(-1000, 1000)$	2.9e-5	4.5e-5	2.1e-3
Near ties, $\mathcal{N}(0, 10^{-8})$	7.9e-7	4.1e-6	4.8e-4
One $+1000$ entry	0	0	6.1e-5
50% $-\infty$ mask†	7.0e-7	4.2e-6	3.0e-4

Table 5: CPU time across input distributions at N=128,256, B=128, on Xeon Gold 6148 with one thread for both methods. Underflow: share of $x_i - \max_j x_j < -87.3$, where FP32 `exp` is subnormal or zero. †Robust **IntLSE** kernel.

Input	Underflow (%)	torch (ms)	IntLSE (ms)	Speedup
$\mathrm{U}(-1, 0)$	0.0	33.0	27.6	1.2×
$\mathrm{U}(-100, 0)$	12.7	294.4	27.6	10.7×
$\mathrm{U}(-500, 0)$	82.5	389.1	27.6	14.1×
$\mathcal{N}(0, 1)$	0.0	34.2	27.6	1.2×
$\mathcal{N}(0, 20^2)$	50.2	606.9	27.6	22.0×
$\mathrm{U}(-1000, 1000)$	95.6	358.6	27.6	13.0×
Near ties, $\mathcal{N}(0, 10^{-8})$	0.0	36.3	27.6	1.3×
One $+1000$ entry	100.0	304.5	27.6	11.0×
50% $-\infty$ mask†	–	188.8	29.9	6.3×

Table 6: LSE latency (ms) at $(B, N) = (512, 128{,}256)$, excluding compilation; **FuseLSE** speedup in parentheses. –: toolchain does not support B300.

Implementation	V100	H100	H200	B300
Eager PyTorch	1.936 (6.25×)	0.584 (6.53×)	0.462 (6.86×)	0.366 (6.70×)
`torch.compile`	0.605 (1.95×)	0.163 (1.83×)	0.120 (1.78×)	–
JAX/XLA	0.651 (2.10×)	0.184 (2.06×)	0.134 (1.99×)	–
FuseLSE	**0.310**	**0.0894**	**0.0674**	**0.0546**

Table 7: Nsight Compute profile at $(B, N) = (512, 128{,}256)$. Below each GPU: peak HBM bandwidth and estimated peak SFU throughput (SMs × 16 ops/clock × boost clock). Traffic: DRAM bytes per call relative to the input size. DRAM, SFU: achieved throughput as % of peak. Pipe stall: % of cycles in which warps wait for a busy math pipe (including the SFU). Multi-kernel values are time-weighted.

GPU	Implementation	Traffic	Time (μs)	DRAM (%)	SFU (%)	Pipe stall (%)
V100 0.90 TB/s 1.8 Top/s SFU	Eager PyTorch	5.98×	1951.2	89.7	2.1	1.31
	`torch.compile`	1.98×	618.8	93.5	7.8	2.18
	JAX/XLA	2.02×	656.0	90.0	6.4	6.06
	FuseLSE	**1.01**×	**314.9**	92.0	28.0	0.20
H100 3.35 TB/s 4.2 Top/s SFU	Eager PyTorch	5.88×	594.4	77.5	3.3	0.64
	`torch.compile`	1.91×	169.6	88.2	12.2	0.38
	JAX/XLA	2.03×	187.2	84.8	11.3	0.25
	FuseLSE	**1.01**×	**94.2**	84.0	45.8	0.10
H200 4.80 TB/s 4.2 Top/s SFU	Eager PyTorch	5.85×	481.4	65.0	4.7	1.18
	`torch.compile`	1.89×	124.6	81.0	19.1	1.14
	JAX/XLA	2.03×	141.0	77.1	16.7	0.57
	FuseLSE	**1.01**×	**72.6**	74.3	65.9	0.21

NeurIPS Paper Checklist

The checklist is designed to encourage best practices for responsible machine learning research, addressing issues of reproducibility, transparency, research ethics, and societal impact. Do not remove the checklist: **The papers not including the checklist will be desk rejected.** The checklist should follow the references and follow the (optional) supplemental material. The checklist does NOT count towards the page limit.

Please read the checklist guidelines carefully for information on how to answer these questions. For each question in the checklist:

- You should answer [Yes], [No], or [N/A].
- [N/A] means either that the question is Not Applicable for that particular paper or the relevant information is Not Available.
- Please provide a short (1–2 sentence) justification right after your answer (even for [N/A]).

The checklist answers are an integral part of your paper submission. They are visible to the reviewers, area chairs, senior area chairs, and ethics reviewers. You will also be asked to include it (after eventual revisions) with the final version of your paper, and its final version will be published with the paper.

The reviewers of your paper will be asked to use the checklist as one of the factors in their evaluation. While [Yes] is generally preferable to [No], it is perfectly acceptable to answer [No] provided a proper justification is given (e.g., error bars are not reported because it would be too computationally expensive" or "we were unable to find the license for the dataset we used"). In general, answering [No] or [N/A] is not grounds for rejection. While the questions are phrased in a binary way, we acknowledge that the true answer is often more nuanced, so please just use your best judgment and write a justification to elaborate. All supporting evidence can appear either in the main paper or the supplemental material, provided in appendix. If you answer [Yes] to a question, in the justification please point to the section(s) where related material for the question can be found.

IMPORTANT, please:

- **Delete this instruction block, but keep the section heading "NeurIPS Paper Checklist"**,
- **Keep the checklist subsection headings, questions/answers and guidelines below.**
- **Do not modify the questions and only use the provided macros for your answers**.

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: The two Kernels claimed in the abstract and introduction, their design can be seen inSection 3, their speed up together with analysis in Section 4 and Section 5.

   Guidelines:

   - The answer [N/A] means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A [No] or [N/A] answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

Justification: Limitations subsection can be found in Section 6.

Guidelines:

- The answer [N/A] means that the paper has no limitation while the answer [No] means that the paper has limitations, but those are not discussed in the paper.
- The authors are encouraged to create a separate "Limitations" section in their paper.
- The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
- The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
- The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
- The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
- If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
- While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [Yes]

   Justification: The theoretical proof of **IntLSE** expected error is detailed in Section A.1, with assumption in Section 3.2.

   Guidelines:

   - The answer [N/A] means that the paper does not include theoretical results.
   - All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
   - All assumptions should be clearly stated or referenced in the statement of any theorems.
   - The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
   - Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
   - Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes]

   Justification: All information needed to reproduce the main experimental results of the paper can be seen in Section 4 and Section 5.

   Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- If the paper includes experiments, a [No] answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
- If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
- Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
- While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
  (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
  (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
  (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
  (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

   Answer: [Yes]

   Justification: We included the code URL in this paper.

   Guidelines:

   - The answer [N/A] means that paper does not include experiments requiring code.
   - Please see the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - While we encourage the release of code and data, we understand that this might not be possible, so [No] is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
   - The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
   - The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
   - At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).

- Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer) necessary to understand the results?

   Answer: [Yes]

   Justification: Our contributions are about computation speed up, not about training and testing, whose parameters are default from the original libraries; all details can be seen in Section 4 and Section 5.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [Yes]

   Justification: We report errors according to conventional methods; details can be seen in Section 4 and Section 5.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The authors should answer [Yes] if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
   - The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
   - The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
   - The assumptions made should be given (e.g., Normally distributed errors).
   - It should be clear whether the error bar is the standard deviation or the standard error of the mean.
   - It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
   - For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g., negative error rates).
   - If error bars are reported in tables or plots, the authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: We provide sufficient information on computer resources(CPU, GPU and memory) for each experiment, details can be seen in Section 4 and Section 5.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.

- The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
- The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
- The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: The research conducted in this paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`.

   Guidelines:

   - The answer [N/A] means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer [No], they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [N/A]

    Justification: This paper is not tied to particular applications or deployment; therefore, no societal impact.

    Guidelines:

    - The answer [N/A] means that there is no societal impact of the work performed.
    - If the authors answer [N/A] or [No], they should explain why their work has no societal impact or why the paper does not address societal impact.
    - Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
    - The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate Deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
    - The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
    - If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pre-trained language models, image generators, or scraped datasets)?

Answer: [N/A]

Justification: This paper does not introduce new data or models and poses no such risks.

Guidelines:

- The answer [N/A] means that the paper poses no such risks.
- Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
- Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
- We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

Answer: [Yes]

Justification: The original owners of the code this paper uses are cited.

Guidelines:

- The answer [N/A] means that the paper does not use existing assets.
- The authors should cite the original paper that produced the code package or dataset.
- The authors should state which version of the asset is used and, if possible, include a URL.
- The name of the license (e.g., CC-BY 4.0) should be included for each asset.
- For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
- If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
- For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

Answer: [Yes]

Justification: We included the code URL in this paper.

Guidelines:

- The answer [N/A] means that the paper does not release new assets.
- Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
- The paper should discuss whether and how consent was obtained from people whose asset is used.
- At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

Answer: [N/A]

Justification: This paper does not involve crowdsourcing nor research with human subjects.

Guidelines:

- The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
- Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
- According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

Answer: [N/A]

Justification: This paper does not involve crowdsourcing nor research with human subjects.

Guidelines:

- The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
- Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
- We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
- For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does *not* impact the core methodology, scientific rigor, or originality of the research, declaration is not required.

Answer: [N/A]

Justification: In this paper, LLM is used only for writing, and formatting purposes and does not impact the core methodology, scientific rigor, or originality of the research.

Guidelines:

- The answer [N/A] means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
- Please refer to our LLM policy in the NeurIPS handbook for what should or should not be described.