
Masked Gated Linear Unit

Yukito Tajima¹ Nakamasa Inoue¹ Yusuke Sekikawa² Ikuro Sato^{1,2} Rio Yokota¹

¹Institute of Science Tokyo, Japan ²Denso IT Laboratory, Japan

yukito@rio.scrs.iir.isct.ac.jp

Abstract

Gated Linear Units (GLUs) have become essential components in the feed-forward networks of state-of-the-art Large Language Models (LLMs). However, they require twice as many memory reads compared to feed-forward layers without gating, due to the use of separate weight matrices for the gate and value streams. To address this bottleneck, we introduce Masked Gated Linear Units (MGLUs), a novel family of GLUs with an efficient kernel implementation. The core contribution of MGLUs include: (1) the Mixture of Element-wise Gating (MoEG) architecture that learns multiple binary masks, each determining gate or value assignments at the element level on a single shared weight matrix resulting in reduced memory transfer, and (2) FlashMGLU, a hardware-friendly kernel that yields up to a $19.7\times$ inference-time speed-up over a naïve PyTorch MGLU and is 47% more memory-efficient and 34% faster than standard GLUs despite added architectural complexity on an RTX5090 GPU. In LLM experiments, the Swish-activated variant SwiMGLU preserves its memory advantages while matching—or even surpassing—the downstream accuracy of the SwiGLU baseline.

1 Introduction

Transformers (Vaswani et al., 2017) have revolutionized deep learning and given rise to large-scale language models (LLMs) that achieve remarkable results across a wide spectrum of natural language processing tasks (Brown et al., 2020; Wei et al., 2022; Ouyang et al., 2022; Grattafiori et al., 2024; OpenAI, 2024; Gemma Team, 2025). Yet the explosive growth in parameter count and inference scaling translates directly into substantial inference cost and latency. The decode latency is fundamentally dominated by communication between high-bandwidth memory (HBM) and SRAM due to transferring model weights for computation, therefore memory reduction being crucial in real-world applications (Gholami et al., 2024).

Modern LLMs predominantly adopt a decoder-only architecture (Radford et al., 2018), with each decoder layer typically consisting of two modules: a self-attention module and a feed-forward network (FFN). Recent successful optimizations for attention mechanisms, such as FlashAttention (Dao et al., 2022; Dao, 2024; Shah et al., 2024) and alternative approaches like the Mamba (Gu and Dao, 2025; Dao and Gu, 2024), have significantly improved the computational efficiency by reducing global memory reads/writes. Nevertheless, kernel-level optimization of the FFN remains particularly challenging, as its simple architecture provides fewer opportunities for efficiency improvements compared to attention.

The activation function within the FFN plays a crucial role in determining the decoder output. Traditional nonlinearities such as ReLU (Nair and Hinton, 2010) and GELU (Hendrycks and Gimpel, 2016) laid the groundwork for deep learning, but the Gated Linear Unit (GLU) variants (Dauphin et al., 2017a; Shazeer, 2020) introduce multiplicative interactions that markedly boost expressivity. For example, SwiGLU, which swaps the sigmoid gate in the original GLU for the smoother Swish function (Ramachandran et al., 2017), has become a primary choice in modern LLMs because it

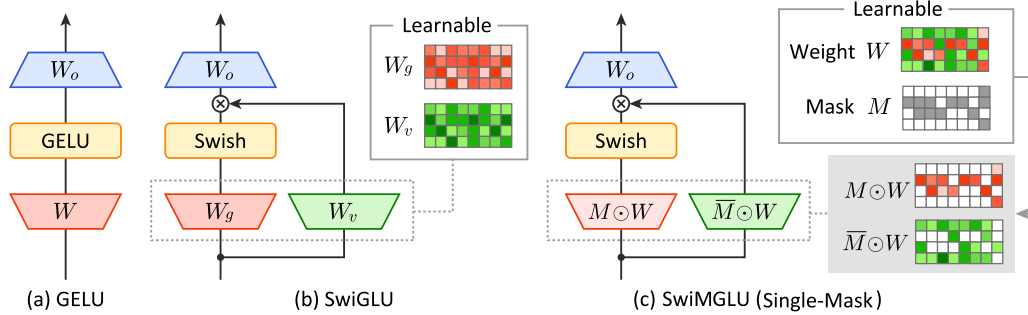


Figure 1: Comparison of FFNs. (a) Two-layer FFN using GELU. (b) SwiGLU FFN with a gating mechanism that requires two separate weights: W_g and W_v . (c) Single-mask variant of SwiMGLU FFN (ours), which introduces a learnable binary mask M to decompose a single weight W into two complementary projections, reducing the required memory load during inference.

further enhances convergence and downstream accuracy. However, the explicit gating operations involving two separate projections and their associated memory overhead at inference time can erode throughput gains, especially in latency-sensitive scenarios.

In this paper, we propose **Masked Gated Linear Units (MGLUs)**, a novel family of GLUs, along with an efficient CUDA kernel implementation. Rather than maintaining two separate projections W_g, W_v for the gate and value streams, our MGLUs adopt the Mixture of Element-wise Gating (MoEG) architecture, which applies learnable binary masks M_i to a single projection W , reproducing GLU’s hallmark gating interactions. In Figure 1, we illustrate a single-mask variant, where a single weight W performs both gate and value projections via a mask M .

The core mechanism enabling efficient kernel implementation, which we call **FlashMGLU**, lies in using complementary masks $\bar{M}_i = 1 - M_i$. Given an input vector x , FlashMGLU simultaneously computes the gate $x(M_i \odot W)$ and the value $x(\bar{M}_i \odot W)$ by leveraging their complementarity. With four masks and a Swish activation applied, our SwiMGLU variant achieves downstream performance comparable to or even better than the widely adopted SwiGLU, while reducing computational cost of the projection layers by 29.1% and memory usage by 37.5% during inference. It is worth noting that FlashMGLU is $12.51\times$ faster than a naïve PyTorch implementation of MGLU, opening new opportunities for more efficient and effective implementations of FFNs.

Our primary contributions are summarized as follows:

- We propose **MGLUs**, a novel family of GLUs with the MoEG architecture. We effectively mimic and enhance gating mechanism without incurring the computational and memory overhead associated with two separate full-rank projections.
- We introduce **FlashMGLU**, an efficient CUDA kernel implementation of MGLUs, reducing memory bandwidth requirements and enabling faster inference with minimal code modification.
- We conduct extensive experiments on a variety of downstream NLP tasks, demonstrating that SwiMGLU achieves comparable or superior downstream accuracy to SwiGLU while notably improving inference throughput and memory efficiency, validating its practical effectiveness for resource-constrained LLM deployments.

2 Related Work

Efficient Inference via Sparse Masks. Model pruning methods seek to reduce the inference memory load by eliminating redundant weights. Early unstructured pruning methods removes individual parameters to achieve high sparsity (Frankle and Carbin, 2019; Xia et al., 2022). While unstructured pruning yields irregular patterns that impede efficient execution, structured pruning excises entire components for straightforward speedups at the cost of coarse-grained weight removal (Hou et al., 2025; Sandri et al., 2025; Ashkboos et al., 2024). Semi-structured approaches blend these two extremes by enforcing regular N:M sparsity blocks, combining fine-grained flexibility with hardware-friendly patterns (Fang et al., 2024; Sun et al., 2024; Frantar and Alistarh, 2023). Learnable N:M domain specific specialized masks optimized on calibration data to attain up to 50% sparsity with minimal perplexity degradation and marked improvements in inference throughput. However, the

weights that are masked out still occupy storage but take no part in computation, leaving a reservoir of unused capacity. If we can reactivate or repurpose these dormant parameters in a controlled way, we may further boost accuracy without increasing the deployed model’s memory footprint.

Efficient Inference via Activation Sparsity. Whereas weight sparsity reduces model size, activation sparsity directly lowers runtime FLOPs at inference (Zhang et al., 2025; HAZIZA et al., 2025; Mirzadeh et al., 2024). DeJaVu shows that, for each input, only a small, input-dependent subset of heads and MLP channels is required; a lightweight predictor selects them on-the-fly, halving latency without pre-training or quality loss (Liu et al., 2023). Complementary to this contextual approach, TEAL applies magnitude-based pruning to the hidden states themselves, achieving 40–50% uniform sparsity without any retraining and realizing decoding speed-ups on modern architectures (Liu et al., 2025). Activation sparsity methods reduce computation and memory load by skipping unneeded activations during inference. In contrast we compress weight matrices directly by adding arithmetic complexity in order to generate multiple streams of outputs.

Choices of Activation Functions in Large Language Models. Transformer (Vaswani et al., 2017) MLP/FFN layers initially adopted simple piecewise-linear rectifiers such as ReLU (Nair and Hinton, 2010), and Gaussian Error Linear Units (GELUs) (Hendrycks and Gimpel, 2016) which weight inputs by the Gaussian CDF to produce smooth, non-saturating transitions. Later, non-monotonic self-gating activations like Swish ($x \cdot \text{sigmoid}(x)$) were discovered via automated search, offering improved deep network performance through enhanced gradient propagation (Ramachandran et al., 2017). More expressive gated linear units (GLUs) introduce multiplicative interactions by splitting a projection into value and gate streams modulated via sigmoid (Dauphin et al., 2017b; Shazeer, 2020); variants such as GEGLU (using GELU) and SwiGLU (using Swish) have been incorporated into Llama series (Grattafiori et al., 2024) to boost convergence and downstream accuracy (Shazeer, 2020).

3 Method

This section presents the **Masked Gated Linear Units (MGLUs)**, a novel family of GLUs that reduces memory access by emulating the gate and value streams using a single shared weight matrix. We first review the GLU variants and discuss the challenges associated with sharing their two separate projections. We then introduce MGLUs, which replace explicit gating with a mixture of element-wise gating (MoEG), sculpting distinct subspaces from the single weight matrix through learnable binary masks. Although our MoEG increases the architectural complexity, it is designed to enable efficient CUDA kernel implementation, eliminating extra matrix multiplies and memory accesses.

3.1 Preliminary

GLU Variants. The GLU variants for FFNs (Dauphin et al., 2017a; Shazeer, 2020) augment a standard two-layer FFN by splitting the intermediate projection into the gate and value streams. Specifically, given an input vector $\mathbf{x} \in \mathbb{R}^h$ and two learnable weight matrices $W_g, W_v \in \mathbb{R}^{h \times d}$, the generalized GLU layer¹ is defined as:

$$\text{GLU}(\mathbf{x}) = g(\mathbf{x}W_g) \odot (\mathbf{x}W_v), \quad (1)$$

where g is a gating function, $\odot$ denotes Hadamard product, h is the hidden size, and d is the intermediate size. The resulting intermediate representation is then mapped back to the hidden size by an output projection $W_o \in \mathbb{R}^{d \times h}$. Figures 1(a) and 1(b) illustrate the architectures of the standard two-layer Linear Unit (LU) FFN using GELU (Hendrycks and Gimpel, 2016) and the SwiGLU FFN using Swish (Ramachandran et al., 2017) for g , respectively.

Can two matrices be shared in SwiGLU? While SwiGLU is a primary choice in state-of-the-art LLMs (Grattafiori et al., 2024), the separate value and gate projection matrices incur $2 \times$ the memory reads compared to a single linear layer. One might ask whether a single shared weight W can serve both streams, for example by using distinct channel-wise transformations or learned offsets. However, naively sharing W typically collapses the expressivity of the multiplicative interaction: independent full-rank projections are required

FFN	Parameters	PPL
GELU	W	25.6
SwiGLU	$\{W_g, W_v\}$	23.6
SwiGLU (shared)	$W_g = W_v$	27.0

Table 1: Perplexity comparison.

¹Throughout this paper, we refer to the generalized GLU layer simply as the GLU layer unless ambiguity arises. Following recent practice in LLMs, we omit bias vectors, but incorporating them is straightforward.

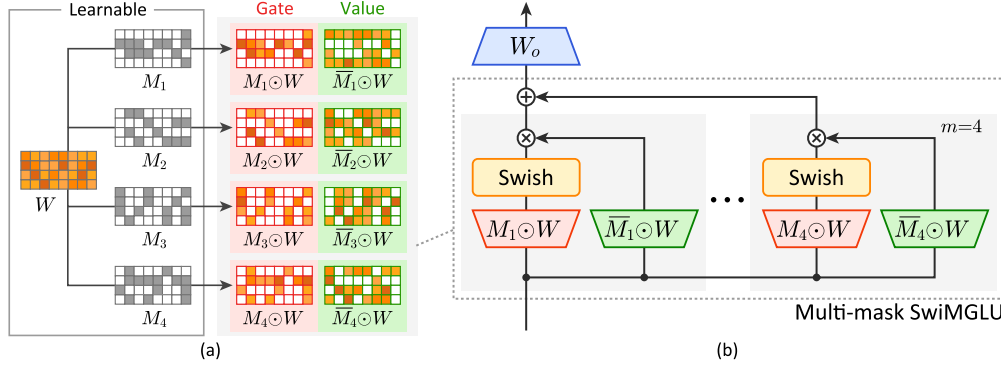


Figure 2: **Mixture of Element-wise Gating (MoEG)**. (a) All gate and value projection matrices are computed from the shared weight matrix W . (b) The MoEG-based SwiMGLU architecture with m routes, each of which leverages element-wise gating.

to learn disentangled feature gating and value transformations as shown in Table 1. This could be a limitation of explicit gating, motivating us to propose MGLUs with element-wise gating.

3.2 MGLU Layer

Our goal is to retain the expressive multiplicative effect of the GLU layer while reducing the number of its full-rank projections. Instead of the two separate projection matrices $\{W_g, W_v\}$, our MGLU layer introduces a small set of binary masks $\mathcal{M} = \{M_i\}_{i=1}^{n_m}$, each of which sculpts different subspaces of a single weight matrix W .

Single-Mask Variant. We start by describing the single-mask variant of the MGLU layer. Let $W \in \mathbb{R}^{h \times d}$ be a shared weight matrix and $M \in \{0, 1\}^{h \times d}$ be a binary mask matrix. We define the MGLU layer as

$$\text{MGLU}_1(x) = g(x(M \odot W)) \odot (x(\bar{M} \odot W)), \quad (2)$$

where x is an input vector, g is a gating function, and $\bar{M} = \mathbf{1} - M$ is the complementary mask. As shown in Figure 1(c), the MGLU layer recovers the multiplicative gate-value interaction using only one full-rank weight matrix W . Through element-wise gating with two binary matrices $\{M, \bar{M}\}$, W is effectively partitioned into two complementary subspaces, resulting in greater parameter efficiency compared to the GLU layer. During training, M is optimized jointly with W via the straight-through estimator (Bengio et al., 2013) on the binarization.

We utilize the complementary mask rather than two separate masks because it enables a more efficient CUDA kernel implementation. FlashMGLU in Section 4 computes the gate projection $x(M \odot W)$ and the value projection $x(\bar{M} \odot W)$ simultaneously by leveraging their complementarity.

MoEG Variant. To capture diverse gating patterns across channels, the MoEG architecture allows multiple complementary masks and lets the model learn which subspace of W serves as gate versus value. Specifically, we define the MoEG-based MGLU as follows:

$$\text{MGLU}_{n_m}(x) = \sum_{i=1}^{n_m} \left[g(x(M_i \odot W)) \odot (x(\bar{M}_i \odot W)) \right], \quad (3)$$

where n_m is the number of mixtures, and $\mathcal{M} = \{M_i\}_{i=1}^{n_m}$ is a set of binary mask matrices.

Figure 2 shows the overall architecture. As shown, all gate projection matrices $M_i \odot W$ and value projection matrices $\bar{M}_i \odot W$ are derived from the shared full-rank weight matrix W (Figure 2(a)), and their resulting representations are subsequently aggregated (Figure 2(b)). This architecture, featuring m parallel routes, can be interpreted as a variant of the mixture of expert architecture (Jacobs et al., 1991; Jordan and Jacobs, 1994), thereby improving representational capacity. During training, the masks $\{M_i\}_{i=1}^{n_m}$ are optimized jointly with W . At inference, all masks are fixed, and the fused masked projections execute with a single kernel.

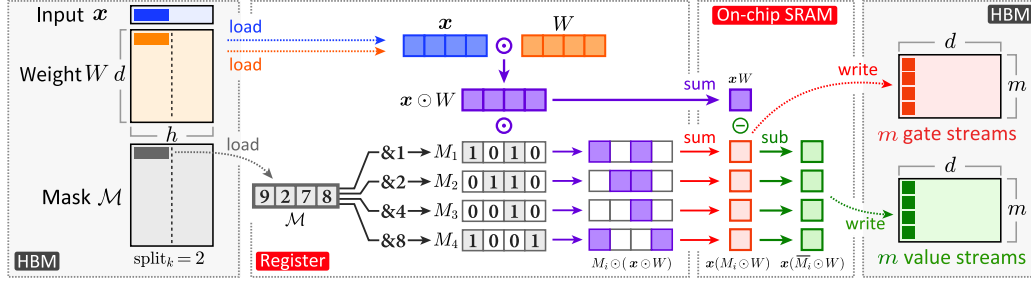


Figure 3: Diagram of how **FlashMGLU** forward pass is performed during generative inference. When the weight W is split into two blocks in the K dimension ($\text{split}_k = 2$), the input vector x is also partitioned into two. We pre-compute the unmasked matrix-vector operation, then selectively add-up the sum according to the mask values avoiding excessive memory reads of weight matrices.

Algorithm 1 FlashMGLU forward pass: Split-K Matrix Vector Product with Packed n_m Masks.

Require: $A \in \mathbb{R}^{M \times N}$, $x \in \mathbb{R}^N$, $\text{mask} \in \{0, \dots, 2^{n_m} - 1\}^{M \times N}$ on HBM, split_k

- 1: Initialize accumulators $z \in \mathbb{R}^{2^{n_m} \times M} \leftarrow 0$
- 2: **for** row = 0 **to** $M - 1$ **do**
- 3: **for** chunk = 0 **to** $\text{split}_k - 1$ **do**
- 4: $s \leftarrow \lceil (N + \text{split}_k - 1) / \text{split}_k \rceil$, $\text{start} \leftarrow \text{chunk} \times s$, $\text{end} \leftarrow \min(\text{start} + s, N)$
- 5: Each thread sets $t \leftarrow 0$ and $s_i \leftarrow 0$ for $i = 1, \dots, n_m$
- 6: **for** each $k = \text{start}, \dots, \text{end} - 1$ (stride = $\text{blockDim}.x$) **do**
- 7: Load $A[\text{row}, k]$, $x[k]$, $\text{mask}[\text{row}, k]$ from HBM to register.
- 8: On chip, compute $v = A[\text{row}, k] \times x[k]$
- 9: On chip, compute $t = t + v$
- 10: **for** $i = 1, \dots, n_m$ **do**
- 11: On chip, compute if $\text{mask}[\text{row}, k] \wedge (1 \ll (i - 1)) \neq 0$ then $s_i = s_i + v$
- 12: **end for**
- 13: **end for**
- 14: Reduce t and all s_i across threads on SRAM.
- 15: **for** $i = 1, \dots, n_m$ **do**
- 16: $\text{atomicAdd}(z[i, \text{row}], s_i)$, $\text{atomicAdd}(z[n_m + i, \text{row}], t - s_i)$ to HBM.
- 17: **end for**
- 18: **end for**
- 19: **end for**
- 20: **return** z

4 FlashMGLU: Efficient Kernel Implementation of MGLUs

Hardware-friendly functions such as attention has widespread application. Here we aim to make MGLUs efficient on modern hardware accelerators (GPUs) as well. In general, low batch size settings (for simplicity, we consider batch size 1) of LLMs’ generative inference require matrix-vector products. Such operations are almost always memory-bound, as the cost of reading the weight matrix dominates the compute time. A naïve PyTorch implementation of MGLUs would issue n_m separate matrix-vector multiplies (and corresponding element-wise Swish and multiplies), resulting in $n_m \times 2 + 1$ full-precision weight memory reads per token—an extremely inefficient pattern.

To address this, we implement a fused CUDA kernel and a simple triton (Tillet et al., 2019) kernel that:

- **Packs mask bits:** Combine the n_m binary masks M_i for each weight entry into a single 8-bit integer, so that one load fetches all mask information for a block of weights.
- **Shared loading:** For each thread block, load a tile of the shared weight matrix W and the corresponding packed mask words in one coalesced transaction for all $n_m \times 2$ output vectors.

This design reduces the number of global memory reads from $n_m \times 2$ to $1 (W \text{ load}) + 1 (\text{mask load})$ per tile, while performing all Swish and element-wise multiplications in fast on-chip memory. As a result, our kernel achieves up to $19.66 \times$ speedup over naïve PyTorch implementation on RTX5090 with $n_m = 8$. Algorithm 1 presents the detailed procedure, and Figure 3 illustrates the forward

pass. The core idea is to fetch all necessary weight and packed-mask blocks for each output element in a single coalesced read and complete every arithmetic operation entirely in registers, thereby eliminating redundant global-memory traffic.

4.1 Discussion

Memory Load During Inference. Table 2 represents the parameter count and required memory load at inference time. Inference in modern transformer FFNs is typically bound by memory bandwidth rather than raw compute. In SwiGLU, the gating and value projections each require reading an FP16 weight matrix of size $h \times d$ for a total of two matrices per token (memory load $16 \times 2hd$ bits), whereas a normal GELU activation only requires one up-projection. SwiMGLU instead folds the two intermediate projections into a single FP16 matrix $W \in \mathbb{R}^{h \times d}$ and applies n_m binary masks $M_i \in \{0, 1\}^{h \times d}$ at inference to recover gating. The resulting per-token memory load is one FP16 matrix ($16 \times 2hd$ bits for up-projection) plus n_m mask bits ($n_m hd$ bits). For $n_m = 1$, the relative reduction is

$$\frac{16 \cdot 2hd - (16 \cdot hd + hd)}{16 \cdot 2hd} = 0.46875,$$

i.e., up to 47% fewer bits transferred, directly translating to faster inference on memory-bound hardware. Mixtures with $n_m \leq 16$ therefore still reduces the parameter count of a standard SwiGLU FFN at 16bit inference time, yet retain SwiGLU-level expressivity during optimization.²

Number of Parameters. SwiMGLU replaces two full-rank SwiGLU weight matrices ($2hd$ FP16 values) with one matrix plus n_m masks, yielding a footprint of $16hd + n_m hd$ bits at inference (masks stored in 1-bit form). During training, masks are kept as FP16 logits, adding $n_m hd$ FP16 parameters.

Computational Cost. At inference time (next token prediction), only the forward pass is executed. SwiMGLU incurs $2(1 + n_m)hd$ multiply-add operations per token, versus $6hd$ for SwiGLU. On memory-bound hardware the extra FLOPs are negligible: memory bandwidth dominates, and SwiMGLU’s reduction in FP16 reads directly lowers inference latency. Training on the other hand is compute bound with the backward costing roughly twice the FLOPs of the forward. Masks are stored as FP16 logits and use straight-through estimation, adding $n_m hd$ FP16 parameters and one extra element-wise multiply-add per mask in each pass. Consequently, the total training cost becomes $(6 + 8n_m)hd$, compared to $18hd$ per token in SwiGLU, introducing a runtime overhead according to the mask size.

Table 2: Parameter count and memory-access cost per token for the intermediate layers of FFN variants during FP16 inference. h and d denote the hidden and intermediate sizes, respectively.

Layer type	#Params (FP16)	#Params (Binary)	Memory Load (bits)
LU	hd	0	$16hd$
GLU	$2hd$	0	$32hd$
MGLU	hd	$n_m hd$	$(16 + n_m)hd$

5 Experiments

We demonstrate the effectiveness and efficiency of SwiMGLU. We selected the Llama 3 architecture (Grattafiori et al., 2024) with SwiGLU as the baseline and compare the different FFN layers.

5.1 Setup

Model Configuration. Our baseline model follows the Llama family design (Grattafiori et al., 2024). We train models at two scales: a 159M *small* (12 layers, $h = 768$, $d = 3072$) and a 1.08B *large* (16 layers, $h = 2048$, $d = 8192$). For SwiMGLU, we substitute each SwiGLU layer with the SwiMGLU layer, keeping all other architectural components (e.g. attention, normalization, embeddings) identical. Note that SwiMGLU would have a smaller model size as the number of projection weights are reduced.

Hyperparameters. All model are trained using the AdamW optimizer ($\beta_1 = 0.9$, $\beta_2 = 0.99$, $\epsilon = 1 \times 10^{-8}$) with a learning rate of 3×10^{-4} and 1×10^{-4} for *small* and *large* models respectively, weight decay of 0.1, linear warmup for the first 10% steps, followed by cosine decay. A full list of hyperparameters and training settings are listed in Appendix A.

²For Llama-1B ($h = 2048$, $d = 8192$) the FFN weights per layer shrink from 96MB to 64MB, while the binary mask costs only 2MB in boolean form.

Table 3: Zero-shot accuracy (%) on downstream tasks and validation perplexity. #weights represent the number of weight parameters excluding masks bits.

	n_m	#weights	PPL↓	ArcE↑	ArcC↑	HS↑	PiQA↑	SciQ↑	WG↑	Avg ↑
GELU	–	113M	25.8	47.47	19.45	28.09	60.61	64.80	52.41	45.47
SwiGLU	–	141M	23.7	48.15	20.05	28.53	61.43	67.90	51.14	46.20
SwiMGLU	1	113M	25.0	48.91	19.28	28.25	60.72	69.00	50.20	46.06
SwiMGLU	2	113M	24.5	49.12	19.97	28.49	60.01	70.60	51.53	46.62
SwiMGLU	4	113M	23.9	48.99	20.56	28.49	61.70	69.10	50.04	46.48
SwiMGLU	8	113M	23.5	48.65	20.56	28.63	61.53	68.00	51.54	46.49
SwiGLU	–	1.08B	12.3	64.94	28.92	37.20	69.15	84.50	51.30	56.00
SwiMGLU	1	808M	13.0	63.72	27.73	36.20	68.61	83.00	54.30	55.59
SwiMGLU	2	808M	12.7	62.08	26.71	36.52	68.44	84.20	51.85	54.97
SwiMGLU	4	808M	12.4	65.78	28.92	37.69	69.26	84.20	55.25	56.85

Table 4: Two-shot accuracy (%) on downstream tasks and validation perplexity. #weights represent the number of weight parameters excluding masks bits.

	n_m	#weights	PPL↓	ArcE↑	ArcC↑	HS↑	PiQA↑	SciQ↑	WG↑	Avg ↑
GELU	–	113M	25.8	48.57	19.45	27.87	60.77	61.10	52.88	45.11
SwiGLU	–	141M	23.7	48.65	20.14	28.64	61.70	62.30	51.70	45.52
SwiMGLU	1	113M	25.0	48.48	19.03	28.16	60.45	61.40	50.75	44.71
SwiMGLU	2	113M	24.5	49.62	21.42	28.55	60.39	62.50	51.14	45.60
SwiMGLU	4	113M	23.9	49.41	21.16	28.61	62.51	66.10	50.59	46.40
SwiMGLU	8	113M	23.5	52.02	19.62	28.54	62.08	66.80	51.38	46.74
SwiGLU	–	1.08B	12.3	66.16	31.48	37.25	69.21	88.30	51.78	57.36
SwiMGLU	1	808M	13.0	65.57	30.20	36.25	69.59	88.50	53.83	57.32
SwiMGLU	2	808M	12.7	65.07	31.66	36.32	68.77	88.70	52.17	57.11
SwiMGLU	4	808M	12.4	67.13	30.55	37.53	69.48	88.60	53.91	57.87

Datasets and Metrics. We pre-train both baseline and SwiMGLU models on the FineWeb-Edu 100B dataset (Penedo et al., 2024) with *small* models being trained on a 10B token subset. For downstream evaluation, we report zero-shot and two-shot accuracy on six standard benchmarks: ARC Easy (ArcE) (Clark et al., 2018), ARC Challenge (ArcC) (Clark et al., 2018), HellaSwag (HS) (Zellers et al., 2019), PiQA Bisk et al. (2020), SciQ (Welbl et al., 2017), and Winogrande (WG) (Sakaguchi et al., 2021). We utilize the LM Evaluation Harness (Gao et al., 2024) for standardized performance evaluation.

5.2 Downstream Performance

Downstream Accuracy. Tables 3 and 4 present the zero-shot and two-shot performance respectively. We observe that SwiMGLU generally matches or outperforms SwiGLU when 4 or more masks are employed. For example $m = 4$ showcases an average of 56.85% compared to SwiGLU’s 56.00%. These results highlight the superior performance of model using the SwiMGLU activation function. Increasing the mask count generally improves the model performance.

Latency. Figure 4 shows the wall-clock latency of a single projection layer in a single batch setting measured on an RTX 5090 at FP16 precision. We evaluate three implementations: our fused CUDA kernel (FlashMGLU), a Triton re-implementation, and the naïve PyTorch GLU nn.Linear baseline.

For the *large* setting ($h = 2048$, $d = 8192$) with $n_m = 8$, FlashMGLU cuts the matrix-vector time from **0.5210 ms to 0.0834 ms**, delivering a **19.66×** speed-up over the PyTorch baseline and a **6.25×** acceleration even on our triton prototype. Importantly, these gains persist at much larger scales: with

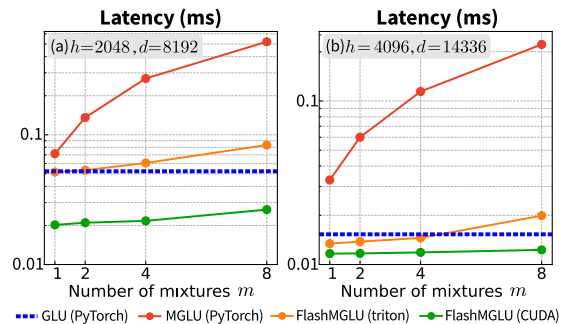


Figure 4: Latency comparison

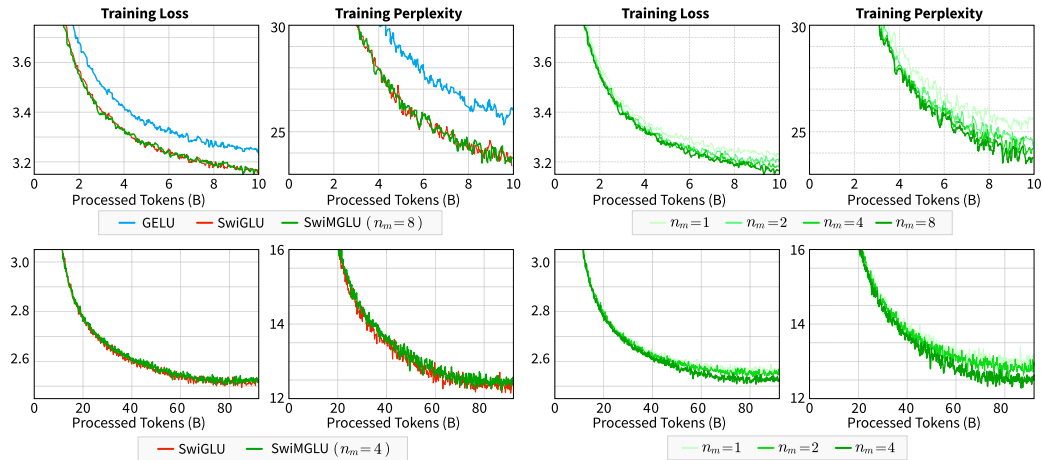


Figure 5: **Comparison of learning curves for different FFN architectures.** The top and bottom rows illustrate the changes in training loss / training perplexity of *small* and *large* models respectively. The left columns compare existing methods against SwiMGLU, and the right columns compare the number of masks.

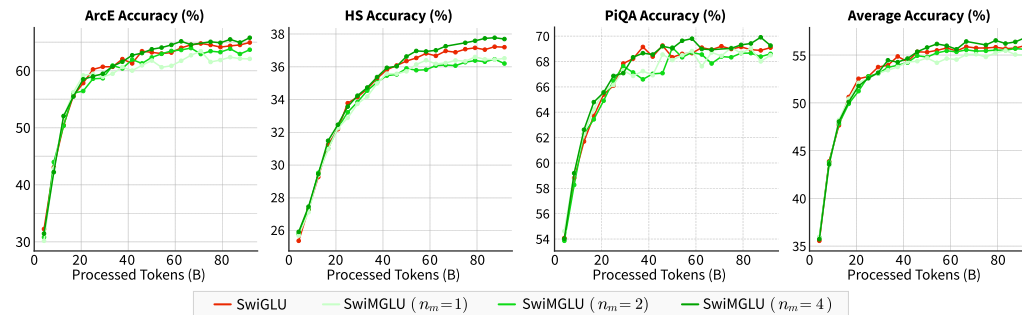


Figure 6: **Comparison of downstream task scores across different FFN architectures of large models.** In all metrics, the proposed method, SwiMGLU $n_m = 4$ achieves the best performance.

larger intermediate sizes ($h = 4096$, $d = 14336$) used in larger Llama-3.2 8B models, our optimized CUDA kernel still outpaces the PyTorch baseline by an impressive $18.0\times$, and a $11.1\times$ speed-up in our triton prototype.

While the standard GLU implementation must fetch two full-precision weight matrices from global memory, FlashMGLU accesses each weight exactly once and performs all mask operations directly within on-chip registers. By fusing weight loading with mask evaluation, FlashMGLU significantly increases arithmetic intensity, effectively saturating the GPU’s Streaming Multiprocessor pipeline and reducing latency by up to $1.51\times$ under the single-mask variant. As a result, our native CUDA implementation of FlashMGLU consistently outperforms both the naïve MGLU and the standard GLU baselines. Although Triton incurs slightly higher overhead on very small matrices, this gap narrows as the hidden dimensions increase. Moreover, since CUDA allows developers to explicitly control memory access patterns, native CUDA code achieves higher performance overall. A full head-to-head comparison between FlashMGLU and a highly tuned *standard* GLU kernel is presented in Appendix E.

Scaling with the number of masks. Under a naïve PyTorch MGLU kernel, runtime scales nearly linearly for $n_m \leq 8$, doubling the number of mask mixtures almost doubles the latency, driven by proportional increases in memory-traffic. In contrast, FlashMGLU’s unified weight-and-mask loading strategy decouples execution time from mask count, yielding only marginal slowdowns as n_m increases. Even under a computationally intense eight mask variant, we still observe a $1.15\times$ speed-up on small models and a $1.24\times$ acceleration on larger variants. Once $n_m = 8$, register-file pressure begins to spill into local memory, and the performance advantage over the standard GLU kernel gradually erodes, which is an effect that is especially noticeable in our Triton prototype.

5.3 Ablations and Analysis

Training Dynamics. Figure 5 plots token-level cross-entropy (left axis) and perplexity (right axis) over the first 150k optimization steps for both *small* and *large* models trained with GELU, SwiGLU, and our SwiMGLU variants. Models using SwiMGLU with $n_m = 4$ consistently show comparable or lower losses compared to those using SwiGLU. Figure 6 illustrates the downstream performance on ARC-Easy, HellaSwag, PiQA, and the average accuracy across all downstream tasks. SwiMGLU with $n_m = 4$ outperforms SwiGLU on all tasks, with notable improvements, demonstrating strong generalization capabilities of MoGE despite its reduced parameter count.

Training Cost. While SwiMGLU significantly reduces inference-time memory access and latency, it incurs a higher computational cost during pretraining. As shown in Appendix A (Table 6), achieving comparable downstream performance to SwiGLU requires approximately twice the wall-clock training time due to the additional optimization of mask parameters. This trade-off reflects a deliberate design choice: we prioritize lower inference cost and memory footprint at the expense of moderately higher training time. We believe this is a reasonable compromise, as real-world deployments of LLMs are typically dominated by inference workloads rather than pretraining.

Training Stability. At default learning rates, both MGLU and SwiGLU converge with comparable smoothness. However, under higher learning rates, where SwiGLU experiences pronounced loss spikes, the masked variants effectively dampen these excursions and recover more rapidly. The four-mask configuration, in particular, proves exceptionally robust. Detailed loss curves for all learning-rate settings are provided in Appendix B.

Number of Masks n_m . For the *small* model size, increasing the number of masks from $n_m = 1$ to $n_m = 8$ results in a strictly monotonic reduction in training loss. Concurrently, this also improves the two-shot average accuracy, which rises from 44.71% to 46.74%. Nevertheless, the marginal benefit diminishes once $n_m = 8$, while memory and compute overhead continue to increase. Empirically, the $n_m = 4$ setting achieves the optimal balance between quality and resource usage. These findings suggest that a compact collection of complementary masks can fully recover, and occasionally surpass, the expressive power of a dense SwiGLU gating mechanism.

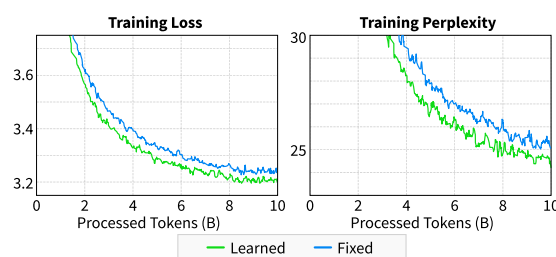


Figure 7: Training loss and perplexity of learned vs. fixed masks in *small* configuration and $n_m = 2$.

Learned vs. Fixed Masks. Figure 7 contrasts models in which the masks are co-optimized with the network parameters against models that rely on randomly sampled but *fixed* masks. Allowing the masks to learn consistently tracks a lower training loss throughout and improves the final perplexity by roughly 0.5. By comparison, freezing the masks essentially collapses the learning curve to that of the single-mask ($n_m = 1$) baseline, underlining that it is *mask-combination learning*—not sparsity alone—that drives the gain in expressivity. Complete results are provided in Appendix B.

Mask Distribution. Inspection of the learned binary masks shows that their activation ratio is *not* forcibly balanced: depending on the layer, the proportion of ones can skew either above or below 50% (typically ranging from roughly 45% to 55%). Full statistics for every layer and model size are provided in Appendix C. This flexibility lets the network *learn* how much capacity to devote to the gate versus the value pathway, effectively tuning the gate-value trade-off on a per-layer basis. Because a dense SwiGLU block lacks such adaptive capacity allocation, this mechanism offers a plausible explanation for the cases in which MGLU surpasses SwiGLU in accuracy.

6 Conclusion and Future Work

In this work, we introduced Masked Gated Linear Units (MGLUs), a novel family of feed-forward activations that recover the expressivity of traditional GLUs using a single shared weight matrix sculpted by learnable binary masks. Our Mixture of Element-wise Gating (MoEG) design not only matches or exceeds the performance of the commonly used SwiGLU activation on a variety of language-understanding benchmarks, but also delivers substantial efficiency gains at inference time.

In terms of limitations, MGLUs are more computationally expensive than other GLU variants, since they increase arithmetic complexity in exchange for a reduced memory footprint. However, by developing FlashMGLU, an optimized kernel that fuses mask unpacking with the core matrix-vector operations, we achieve up to a $19.66\times$ speedup over a naïve PyTorch MGLU implementation on modern GPUs, while reducing memory-bandwidth requirements and latency compared to standard GLU variants. These advances pave the way for richer gating mechanisms to be deployed in latency- and memory-constrained settings without compromising model quality.

7 Acknowledgement

This work was supported by DENSO IT LAB Recognition, Control, and Learning Algorithm Collaborative Research Chair (Science Tokyo).

This work used computational resources TSUBAME4.0 supercomputer provided by Institute of Science Tokyo through the HPCI System Research Project (Project ID: hp240170).

References

- Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Proc. Annual Conference on Neural Information Processing Systems (NeurIPS)*, 2017.
- Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, and Amanda et al. Askell. Language models are few-shot learners. In *Proc. Annual Conference on Neural Information Processing Systems (NeurIPS)*, 2020.
- Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, brian ichter, Fei Xia, Ed H. Chi, Quoc V Le, and Denny Zhou. Chain of thought prompting elicits reasoning in large language models. In *Proc. Annual Conference on Neural Information Processing Systems (NeurIPS)*, 2022.
- Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, and Alex Gray et al. Training language models to follow instructions with human feedback. In *Proc. Annual Conference on Neural Information Processing Systems (NeurIPS)*, 2022.
- 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.
- OpenAI. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2024.
- Gemma Team. Gemma 3 technical report. *arXiv preprint arXiv:arXiv:2503.19786*, 2025.
- Amir Gholami, Zhewei Yao, Sehoon Kim, Coleman Hooper, Michael W. Mahoney, and Kurt Keutzer. Ai and memory wall. *arXiv preprint arXiv:2403.14123*, 2024.
- Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving language understanding by generative pre-training. *OpenAI Technical Report*, 2018.
- Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher R’e. FlashAttention: Fast and memory-efficient exact attention with io-awareness. In *Proc. Annual Conference on Neural Information Processing Systems (NeurIPS)*, 2022.
- Tri Dao. FlashAttention-2: Faster attention with better parallelism and work partitioning. In *Proc. International Conference on Learning Representations (ICLR)*, 2024.
- Jay Shah, Ganesh Bikshandi, Ying Zhang, Vijay Thakkar, Pradeep Ramani, and Tri Dao. FlashAttention-3: Fast and accurate attention with asynchrony and low-precision. In *Proc. Annual Conference on Neural Information Processing Systems (NeurIPS)*, 2024.
- Albert Gu and Tri Dao. Mamba: Linear-time sequence modeling with selective state spaces. In *Proc. Conference on Language Modeling (COLM)*, 2025.

- Tri Dao and Albert Gu. Transformers are SSMS: Generalized models and efficient algorithms through structured state space duality. In *Proc. International Conference on Machine Learning (ICML)*, 2024.
- Vinod Nair and Geoffrey E. Hinton. Rectified linear units improve restricted boltzmann machines. In *Proc. International Conference on Machine Learning (ICML)*, pages 807–814, 2010.
- Dan Hendrycks and Kevin Gimpel. Gaussian error linear units (gelus). *arXiv preprint arXiv:1606.08415*, 2016.
- Yann N. Dauphin, Angela Fan, Michael Auli, and David Grangier. Language modeling with gated convolutional networks. In *Proc. International Conference on Machine Learning (ICML)*, pages 933–941, 2017a.
- Noam Shazeer. Glu variants improve transformer. *arXiv preprint arXiv:2002.05202*, 2020.
- Prajit Ramachandran, Barret Zoph, and Quoc V. Le. Searching for activation functions. *arXiv preprint arXiv:1710.05941*, 2017.
- Jonathan Frankle and Michael Carbin. The lottery ticket hypothesis: Finding sparse, trainable neural networks. In *Proc. International Conference on Learning Representations (ICLR)*, 2019.
- Mengzhou Xia, Zexuan Zhong, and Danqi Chen. Structured pruning learns compact and accurate models. In *Proc. Annual Meeting of the Association for Computational Linguistics (ACL)*, 2022.
- Bairu Hou, Qibin Chen, Jianyu Wang, Guoli Yin, Chong Wang, Nan Du, Ruoming Pang, Shiyu Chang, and Tao Lei. Instruction-following pruning for large language models. In *Proc. International Conference on Machine Learning (ICML)*, 2025.
- Fabrizio Sandri, Elia Cunegatti, and Giovanni Iacca. 2SSP: A two-stage framework for structured pruning of llms. *arXiv preprint arXiv:2501.17771*, 2025.
- Saleh Ashkboos, Maximilian L. Croci, Marcelo Gennari do Nascimento, Torsten Hoeffler, and James Hensman. SliceGPT: Compress large language models by deleting rows and columns. In *Proc. International Conference on Learning Representations (ICLR)*, 2024.
- Gongfan Fang, Hongxu Yin, Saurav Muralidharan, Greg Heinrich, Jeff Pool, Jan Kautz, Pavlo Molchanov, and Xinchao Wang. MaskLLM: Learnable semi-structured sparsity for large language models. In *Proc. Annual Conference on Neural Information Processing Systems (NeurIPS)*, 2024.
- Mingjie Sun, Zhuang Liu, Anna Bair, and J. Zico Kolter. A simple and effective pruning approach for large language models. In *Proc. International Conference on Learning Representations (ICLR)*, 2024.
- Elias Frantar and Dan Alistarh. SparseGPT: massive language models can be accurately pruned in one-shot. In *Proc. International Conference on Machine Learning (ICML)*, 2023.
- Zhenyu Zhang, Zechun Liu, Yuandong Tian, Harshit Khaitan, Zhangyang Wang, and Steven Li. R-sparse: Rank-aware activation sparsity for efficient LLM inference. In *Proc. International Conference on Learning Representations (ICLR)*, 2025.
- Daniel HAZIZA, Timothy Chou, Dhruv Choudhary, Jesse Cai, Luca Wehrstedt, Francisco Massa, Jiecao Yu, Geonhwa Jeong, Supriya Rao, and Patrick Labatut. Accelerating transformer inference and training with 2:4 activation sparsity. In *Sparsity in LLMs (SLLM): Deep Dive into Mixture of Experts, Quantization, Hardware, and Inference*, 2025.
- Seyed Iman Mirzadeh, Keivan Alizadeh-Vahid, Sachin Mehta, Carlo C del Mundo, Oncel Tuzel, Golnoosh Samei, Mohammad Rastegari, and Mehrdad Farajtabar. ReLU strikes back: Exploiting activation sparsity in large language models. In *Proc. International Conference on Learning Representations (ICLR)*, 2024.
- Zichang Liu, Jue Wang, Tri Dao, Tianyi Zhou, Binhang Yuan, Zhao Song, Anshumali Shrivastava, Ce Zhang, Yuandong Tian, Christopher Re, and Beidi Chen. Deja vu: Contextual sparsity for efficient LLMs at inference time. In *Proc. International Conference on Machine Learning (ICML)*, 2023.

- James Liu, Pragaash Ponnusamy, Tianle Cai, Han Guo, Yoon Kim, and Ben Athiwaratkun. Training-free activation sparsity in large language models. In *Proc. International Conference on Learning Representations (ICLR)*, 2025.
- Yann N. Dauphin, Angela Fan, Michael Auli, and David Grangier. Language modeling with gated convolutional networks. In *Proc. International Conference on Machine Learning (ICML)*, 2017b.
- Yoshua Bengio, Nicholas L'eonard, and Aaron Courville. Estimating or propagating gradients through stochastic neurons for conditional computation. *arXiv preprint arXiv:1308.3432*, 2013.
- Robert A. Jacobs, Michael I. Jordan, Steven J. Nowlan, and Geoffrey E. Hinton. Adaptive mixtures of local experts. *Neural Comput.*, 3(1):79–87, 1991.
- Michael I. Jordan and Robert A. Jacobs. Hierarchical mixtures of experts and the EM algorithm. *Neural Comput.*, 6(2):181–214, 1994.
- Philippe Tillet, H. T. Kung, and David Cox. Triton: an intermediate language and compiler for tiled neural network computations. In *Proc. ACM SIGPLAN International Workshop on Machine Learning and Programming Languages (MAPL)*, page 10–19, 2019.
- Guilherme Penedo, Hynek Kydlíček, Loubna Ben allal, Anton Lozhkov, Margaret Mitchell, Colin Raffel, Leandro Von Werra, and Thomas Wolf. The fineweb datasets: Decanting the web for the finest text data at scale. In *Proc. Annual Conference on Neural Information Processing Systems (NeurIPS)*, 2024.
- Peter Clark, Isaac Cowhey, Oren Etzioni, Tushar Khot, Ashish Sabharwal, Carissa Schoenick, and Oyvind Tafjord. Think you have solved question answering? try arc, the ai2 reasoning challenge. *arXiv preprint arXiv:1803.05457*, 2018.
- Rowan Zellers, Ari Holtzman, Yonatan Bisk, Ali Farhadi, and Yejin Choi. HellaSwag: Can a machine really finish your sentence? In *Proc. Annual Meeting of the Association for Computational Linguistics (ACL)*, 2019.
- Yonatan Bisk, Rowan Zellers, Ronan Le Bras, Jianfeng Gao, and Yejin Choi. Piqa: Reasoning about physical commonsense in natural language. In *Proc. AAAI Conference on Artificial Intelligence*, 2020.
- Johannes Welbl, Nelson F. Liu, and Matt Gardner. Crowdsourcing multiple choice science questions. In *Proc. Workshop on Noisy User-generated Text*, 2017.
- Keisuke Sakaguchi, Ronan Le Bras, Chandra Bhagavatula, and Yejin Choi. Winogrande: An adversarial winograd schema challenge at scale. *Communications of the ACM*, 64(9):99–106, 2021.
- Leo Gao, Jonathan Tow, Baber Abbasi, Stella Biderman, Sid Black, Anthony DiPofi, Charles Foster, Laurence Golding, Jeffrey Hsu, Alain Le Noac'h, Haonan Li, Kyle McDonell, Niklas Muennighoff, Chris Ociepa, Jason Phang, Laria Reynolds, Hailey Schoelkopf, Aviya Skowron, Lintang Sutawika, Eric Tang, Anish Thite, Ben Wang, Kevin Wang, and Andy Zou. The language model evaluation harness. 2024. doi: 10.5281/zenodo.12608602.
- Kazuki Fujii, Taishi Nakamura, and Rio Yokota. llm-recipes, May 2024. URL <https://github.com/okoge-kaz/llm-recipes>.
- William Fedus, Barret Zoph, and Noam M. Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *J. Mach. Learn. Res.*, 23:120:1–120:39, 2021.

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 [NA].
- [NA] 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 NA).

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 be asked to also 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 "[NA]" 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: This submission introduces a novel MGLU FFN architecture in Large Language Models, including a fast FlashMGLU kernel for inference.

Guidelines:

- The answer NA 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 NA 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: Section 6 explicitly discusses limitations.

Guidelines:

- The answer NA 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: [NA]

Justification: No theoretical results in this submission.

Guidelines:

- The answer NA 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 training settings and hyperparameters necessary for reproducing the main results are available in Appendix A.

Guidelines:

- The answer NA 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: [No]

Justification: The code will be released with a permissive license in the near future.

Guidelines:

- The answer NA means that paper does not include experiments requiring code.
- Please see the NeurIPS code and data submission guidelines (<https://nips.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://nips.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, hyper-parameters, how they were chosen, type of optimizer, etc.) necessary to understand the results?

Answer: [Yes]

Justification: We discuss datasets and hyper-parameters used for training and testing in the appendix.

Guidelines:

- The answer NA 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: [No]

Justification: Error bars are not available in this submission.

Guidelines:

- The answer NA 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 report the number of GPUs and GPU hours in appendix.

Guidelines:

- The answer NA 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: This submission follows the NeurIPS Code of Ethics.

Guidelines:

- The answer NA 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: [Yes]

Justification: This submission discusses potential societal impacts in the appendix.

Guidelines:

- The answer NA means that there is no societal impact of the work performed.
- If the authors answer NA 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., pretrained language models, image generators, or scraped datasets)?

Answer: [NA]

Justification: This paper does not include publicly released models nor datasets.

Guidelines:

- The answer NA 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: All assets are properly credited.

Guidelines:

- The answer NA 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: [\[NA\]](#)

Justification: The paper does not include new assets.

Guidelines:

- The answer NA 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: [\[NA\]](#)

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

Guidelines:

- The answer NA 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: [NA]

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

Guidelines:

- The answer NA 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 rigorousness, or originality of the research, declaration is not required.

Answer: [NA]

Justification: LLMs do not involve in core method development in the paper.

Guidelines:

- The answer NA 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 (<https://neurips.cc/Conferences/2025/LLM>) for what should or should not be described.

A Experimental Details

Here we provide more details about the model architecture, training configurations and resources used in our experiments.

A.1 Model Architecture and Training Configuration

Table 5 summarizes the architectural configurations shared across all experiments. Table 6 lists the number of weight and mask parameters, along with the corresponding estimated storage sizes at inference time for each model variant.

All experiments reported in this paper are implemented based on the `llm-recipes` framework (Fujii et al., 2024). All models are trained on TSUBAME supercomputer with NVIDIA H100 GPUs (94GB), with *small* models trained on 4GPUs and *large* models trained on 16GPUs with Fully Sharded Data Parallel (FSDP). Training GPU hours are shown in Table 6.

Table 5: Model architecture of *small* and *large* variants.

Model Size	h	d	Context Length	#Heads	#Layers
<i>small</i>	768	3072	1024	24	12
<i>large</i>	2048	8192	4096	32	16

Table 6: Number of weight and mask parameters, estimated storage size, and training GPU hours for each model configuration.

Scale	Model	n_m	#Weights	#Masks	Size (MB)	GPU Hours
<i>small</i>	GELU	–	113M	0	215	18
	SwiGLU	–	141M	0	269	22
	SwiMGLU	1	113M	28.3M	219	20
	SwiMGLU	2	113M	56.6M	222	24
	SwiMGLU	4	113M	113M	229	33
	SwiMGLU	8	113M	226M	242	52
<i>large</i>	SwiGLU	–	1.08B	0	2052	768
	SwiMGLU	1	808M	268M	1573	752
	SwiMGLU	2	808M	537M	1605	1008
	SwiMGLU	4	808M	1.07B	1669	1376

A.2 Hyperparameters

Table 7 lists the hyperparameters that we use by default at training time for all our experiments.

Table 7: Pretraining hyperparameters for *small* and *large* models.

	<i>small</i>	<i>large</i>
Optimizer	AdamW	AdamW
Learning Rate (LR)	3E-4	1E-4
Minimum LR	3E-5	1E-5
LR Schedule	cosine	cosine
Weight Decay	0.1	0.1
β_1	0.9	0.9
β_2	0.99	0.99
ϵ	1E-8	1E-8
Gradient Clipping	1	1
Global Batch Size	512	512
Warmup Steps	1000	4800

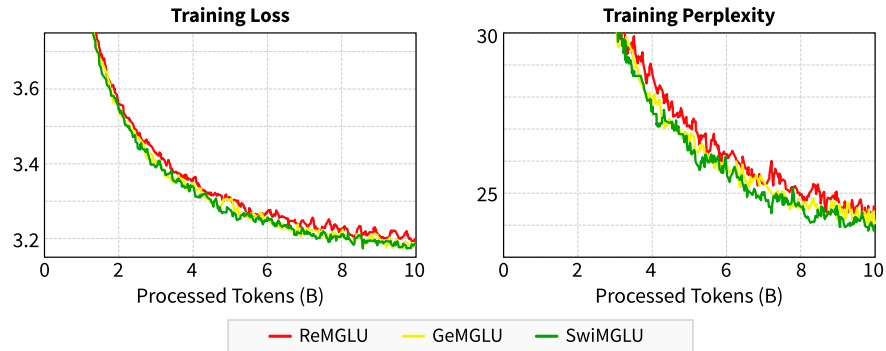


Figure 8: **Trainig Curves of *small* MGLU variants with different activation functions.** Left: training loss; right: validation perplexity.

Table 8: Zero-shot accuracy (%) on downstream tasks and validation perplexity across different activation functions. #weights represent the number of weight parameters excluding masks bits.

	n_m	#weights	PPL↓	ArcE↑	ArcC↑	HS↑	PiQA↑	SciQ↑	WG↑	Avg ↑
SwiGLU	–	141M	23.7	48.15	<u>20.05</u>	28.53	61.43	67.90	51.14	46.20
ReMGLU	4	113M	24.3	49.66	19.71	28.45	61.53	<u>69.10</u>	49.80	46.38
GeMGLU	4	113M	24.0	48.65	19.54	<u>28.50</u>	62.30	69.30	52.17	46.74
SwiMGLU	4	113M	<u>23.9</u>	<u>48.99</u>	20.56	28.49	<u>61.70</u>	<u>69.10</u>	50.04	<u>46.48</u>

Table 9: Two-shot accuracy (%) on downstream tasks and validation perplexity across different activation functions. #weights represent the number of weight parameters excluding masks bits.

	n_m	#weights	PPL↓	ArcE↑	ArcC↑	HS↑	PiQA↑	SciQ↑	WG↑	Avg ↑
SwiGLU	–	141M	23.7	48.65	20.14	<u>28.64</u>	61.70	62.30	<u>51.70</u>	45.52
ReMGLU	4	113M	24.3	<u>49.28</u>	19.71	28.25	61.32	61.80	51.22	45.26
GeMGLU	4	113M	24.0	<u>49.28</u>	20.56	28.74	<u>62.13</u>	66.40	52.25	46.56
SwiMGLU	4	113M	<u>23.9</u>	49.41	21.16	28.61	62.51	<u>66.10</u>	50.59	<u>46.40</u>

B Additional Experiments

B.1 Other Activation Functions.

In the main paper we concentrated on SwiMGLU, whose gating function relies on the Swish non-linearity used in standard SwiGLU. To verify that the advantages of our mask-based design are not tied to a single activation, this appendix evaluates several alternatives that are widely adopted in large-scale language models—GELU, ReLU, and SiLU. For each activation we replace the gating function $g(\cdot)$ in Eq. (3) while keeping all other architectural choices and training hyper-parameters identical to the baseline. We report both pre-training perplexity and downstream accuracy of *small* models so that we can directly compare the impact of each non-linearity under the same experimental conditions.

Training Curve. Figure 8 plots training loss and perplexity for the various activations. ReMGLU converges more slowly than both GeMGLU and SwiMGLU, with SwiMGLU achieving the fastest convergence and slightly outperforming GeMGLU.

Downstream Evaluation. Table 8 and Table 9 show the downstream evaluation scores on different activation functions in MGLU. The masked design improves or matches the SwiGLU baseline across all activations while using fewer trained weights, confirming that the capacity unlocked by mask pairs is activation-agnostic. Smoother nonlinearities (GELU and SiLU) benefit most: GeMGLU achieves the best average zero-shot accuracy (46.74%) and ties for the best two-shot average (46.56%), whereas SwiMGLU attains the lowest validation perplexity (23.9). ReMGLU, though slightly weaker than its smoother counterparts, and comparable to the SwiGLU baseline on average accuracy.

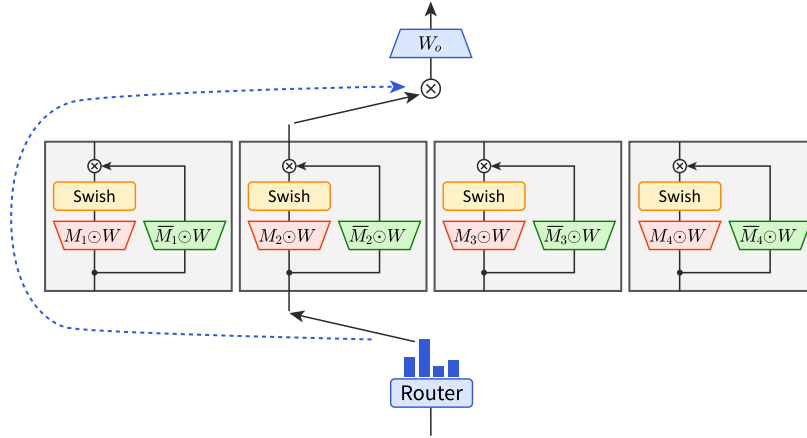


Figure 9: **Diagram of a Top-1 SwiMGLU block.** We illustrate one token being routed across four mask experts, where the router independently routes each token. The Top-1 SwiMGLU layer returns the output of the selected experts multiplied by the router gate value.

B.2 Top-K Routing

Summation over every masking route maximizes capacity but may be wasteful for tokens that require only a subset of specialized subspaces. Inspired by routing techniques in switch transformers (Fedus et al., 2021), we therefore explore a *Top-K routing* variant of MGLU that, at inference time, activates only the K masks whose logits yield the highest gate magnitudes. The subsection first details the routing algorithm and its lightweight implementation—requiring a single additional linear router over the input features, then evaluates downstream scores as K varies from 1 (fully sparse) to n_m (fully dense). We show that, on *small* configuration, routing with $K = 2$ retains most of the accuracy gains of full MGLU with $n_m = 4$, while reducing computational FLOPs by $2\times$, offering a practical knob for deployments with stringent latency budgets.

Top-K routed MGLU. Given an input token representation $\mathbf{x} \in \mathbb{R}^h$, a lightweight “router” computes a vector of logits

$$\ell = \mathbf{x}W_r \in \mathbb{R}^{n_m}, \quad (4)$$

where W_r is a learned $h \times n_m$ matrix. Following Fedus et al. (2021), we retain only the K largest logits. Applying a softmax over this truncated vector yields sparse gating weights

$$G(\mathbf{x}) = \text{Softmax}(\text{TopK}(\ell)) \in \mathbb{R}^{n_m}, \quad (5)$$

The Top-K routed MGLU is denoted by

$$\text{MGLU}_{\text{Top-K}}(\mathbf{x}) = \sum_{i=1}^{n_m} G(\mathbf{x})_i g(\mathbf{x}(M_i \odot W)) \odot \mathbf{x}(\bar{M}_i \odot W). \quad (6)$$

Because $G(\mathbf{x})$ contains at most K non-zero terms, only those K masked projections need be evaluated—reducing memory traffic and latency compared with summing over all n_m routes while preserving most of the accuracy gains of full MGLU. Figure 9 illustrates the overall architecture.

Training Curves. Figure 10 presents the training loss and perplexity across different routing configurations. We observe that both Top-4 and Top-2 routed SwiMGLU closely track the loss trajectory of the non-routed SwiMGLU, indicating that sparse routing with multiple active masks retains most of the model’s learning capacity. In contrast, Top-1 routing introduces a noticeable performance degradation, suggesting that activating only a single expert per token limits representational power.

Downstream Evaluation. Table 10 and Table 11 summarize validation perplexity and task performance for each routing strategy. The **Top-4** router ($K = 4$) consistently dominates, delivering the lowest perplexity in both settings, and the highest average accuracy. The **Top-2** variant ($K = 2$) offers a favorable trade-off: it matches or surpasses the unrouted SwiMGLU ($n_m=4$) on most metrics, boosts average zero-shot accuracy to 46.64 %, and retains a competitive two-shot score of 45.79 %.

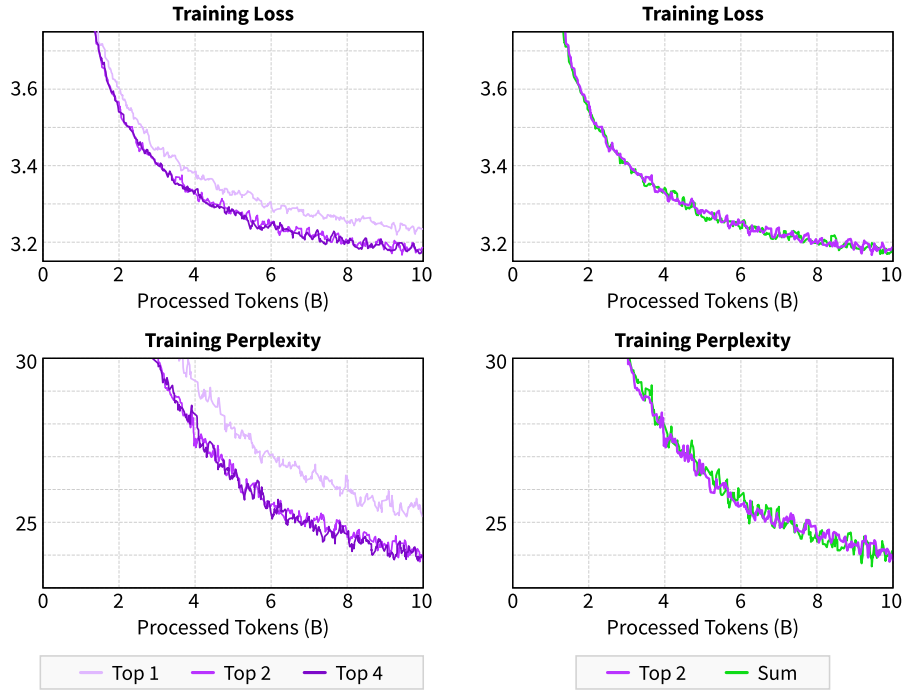


Figure 10: **Learning curves of *small* SwiMGLU models under different Top- K routing strategies.** Left: training loss and validation perplexity for $K \in \{1, 2, 4\}$; right: Top-2 routing compared with the non-routed SwiMGLU baseline.

Table 10: Zero-shot accuracy (%) on downstream tasks and validation perplexity across different routing coefficient K . #weights represent the number of weight and router parameters excluding masks bits. The boldface and underline indicate, respectively, the best and second-best value per column.

	n_m	K	#weights	PPL↓	ArcE↑	ArcC↑	HS↑	PiQA↑	SciQ↑	WG↑	Avg ↑
SwiGLU	—	—	141M	23.7	48.15	20.05	28.53	61.43	67.90	51.14	46.20
SwiMGLU	4	—	113M	23.9	<u>48.99</u>	<u>20.56</u>	28.49	61.70	69.10	50.04	46.48
SwiMGLU	4	1	113M	25.2	48.11	19.80	27.89	61.04	68.90	50.67	46.07
SwiMGLU	4	2	113M	24.0	48.86	<u>20.56</u>	28.29	60.55	<u>69.50</u>	<u>52.09</u>	<u>46.64</u>
SwiMGLU	4	4	113M	<u>23.8</u>	49.20	21.84	28.70	61.37	71.80	53.43	47.72

Table 11: Two-shot accuracy (%) on downstream tasks and validation perplexity across different routing coefficient K . #weights represent the number of weight and router parameters excluding masks bits. The boldface and underline indicate, respectively, the best and second-best value per column.

	n_m	K	#weights	PPL↓	ArcE↑	ArcC↑	HS↑	PiQA↑	SciQ↑	WG↑	Avg ↑
SwiGLU	—	—	141M	23.7	48.65	20.14	28.64	61.70	62.30	51.70	45.52
SwiMGLU	4	—	113M	23.9	49.41	<u>21.16</u>	28.61	62.51	66.10	50.59	<u>46.40</u>
SwiMGLU	4	1	113M	25.2	49.03	20.31	27.95	60.50	59.90	<u>52.17</u>	44.98
SwiMGLU	4	2	113M	24.0	51.39	20.65	<u>28.71</u>	62.08	60.40	51.54	45.79
SwiMGLU	4	4	113M	<u>23.8</u>	<u>49.79</u>	22.27	28.82	<u>62.13</u>	<u>63.00</u>	52.57	46.43

In contrast, **Top-1** routing ($K = 1$) shows a significant drop in both perplexity and average accuracy, suggesting that allocating at least two experts per token is critical for maintaining representation power. Overall, these results indicate that $K = 2$ not only reduces compute but also enhances generalization across a diverse suite of downstream tasks.

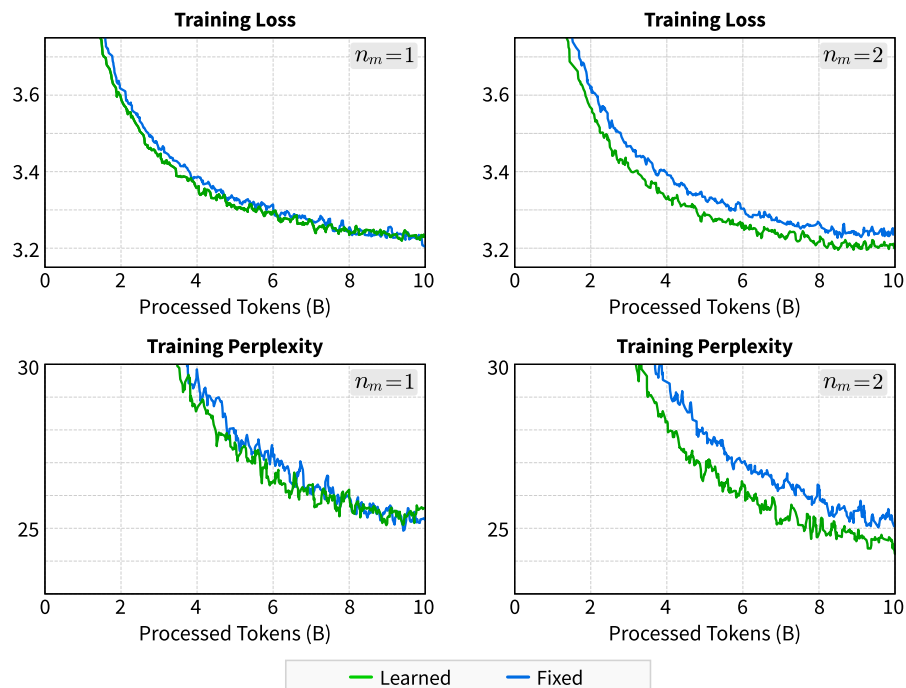


Figure 11: **Training curves of learned vs. fixed masks in *small* SwiMGLU models.** Lef: $n_m = 1$; Right: $n_m = 2$.

Table 12: Zero-shot accuracy (%) on downstream tasks and validation perplexity on *small* models.

	n_m	Mask Type	PPL↓	ArcE↑	ArcC↑	HS↑	PiQA↑	SciQ↑	WG↑	Avg ↑
SwiMGLU	1	Learned	25.0	48.91	19.28	28.25	60.72	69.00	50.20	46.06
SwiMGLU	1	Fixed	25.1	47.60	21.08	28.19	61.10	68.30	51.54	46.30
SwiMGLU	2	Learned	24.5	49.12	19.97	28.49	60.01	70.60	51.53	46.62
SwiMGLU	2	Fixed	25.1	48.32	19.03	28.23	61.81	67.30	50.36	45.84

Table 13: Two-shot accuracy (%) on downstream tasks and validation perplexity on *small* models.

	n_m	Mask Type	PPL↓	ArcE↑	ArcC↑	HS↑	PiQA↑	SciQ↑	WG↑	Avg ↑
SwiMGLU	1	Learned	25.0	48.48	19.03	28.16	60.45	61.40	50.75	44.71
SwiMGLU	1	Fixed	25.1	48.11	20.14	28.02	61.21	60.30	51.78	44.92
SwiMGLU	2	Learned	24.5	49.62	21.42	28.55	60.39	62.50	51.14	45.60
SwiMGLU	2	Fixed	25.1	47.14	20.31	28.24	61.43	61.40	51.93	45.07

B.3 Learned vs. Fixed Masks.

This section compares learned and fixed masks in *small* SwiMGLU models. While previous sections focus on the number and structure of masks, here we investigate the benefit of co-optimizing masks alongside model weights.

Training Curve. Figure 11 shows the training loss and perplexity when using learned versus fixed masks. While with $n_m = 2$, learned masks consistently outperform fixed ones, with $n_m = 1$ the difference is subtle.

Downstream Evaluation. Table 12 and Table 13 shows the downstream task scores across different mask configurations. The learned mask configuration with $n_m = 2$ also improves downstream accuracy compared to fixed masks, highlighting the importance of mask combination adaptation. These results suggest that expressivity is not solely due to sparsity, but also driven by mask learning with multiple masks.

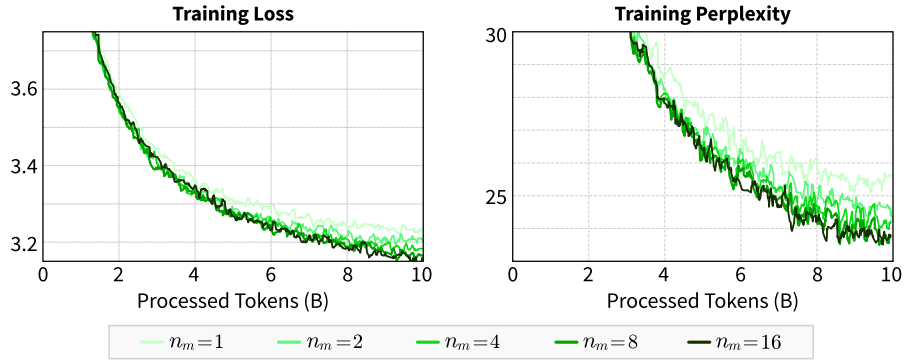


Figure 12: Training curves of *small* SwiMGLU models across different mask count.

Table 14: Zero-shot accuracy (%) on downstream tasks and validation perplexity. #weights represent the number of weight parameters excluding masks bits. The boldface and underline indicate, respectively, the best and second-best value per column.

	n_m	#weights	PPL↓	ArcE↑	ArcC↑	HS↑	PiQA↑	SciQ↑	WG↑	Avg ↑
SwiMGLU	1	113M	25.0	48.91	19.28	28.25	60.72	69.00	50.20	46.06
SwiMGLU	2	113M	24.5	49.12	19.97	28.49	60.01	70.60	51.53	46.62
SwiMGLU	4	113M	23.9	48.99	<u>20.56</u>	28.49	61.70	<u>69.10</u>	50.04	46.48
SwiMGLU	8	113M	23.5	48.65	<u>20.56</u>	28.63	61.53	68.00	51.54	<u>46.49</u>
SwiMGLU	16	113M	23.3	48.15	21.08	28.63	<u>61.59</u>	68.50	50.91	46.47

Table 15: Two-shot accuracy (%) on downstream tasks and validation perplexity. #weights represent the number of weight parameters excluding masks bits. The boldface and underline indicate, respectively, the best and second-best value per column.

	n_m	#weights	PPL↓	ArcE↑	ArcC↑	HS↑	PiQA↑	SciQ↑	WG↑	Avg ↑
SwiMGLU	1	113M	25.0	48.48	19.03	28.16	60.45	61.40	50.75	44.71
SwiMGLU	2	113M	24.5	49.62	21.42	28.55	60.39	62.50	51.14	45.60
SwiMGLU	4	113M	23.9	49.41	<u>21.16</u>	<u>28.61</u>	62.51	66.10	50.59	46.40
SwiMGLU	8	113M	23.5	52.02	19.62	28.54	<u>62.08</u>	<u>66.80</u>	<u>51.38</u>	<u>46.74</u>
SwiMGLU	16	113M	23.3	<u>50.76</u>	21.08	28.66	61.70	66.90	52.25	46.89

B.4 Scaling n_m to 16

We investigate the impact of increasing the number of masks n_m up to 16. As shown in Figure 12, scaling n_m leads to steady improvements in both training loss and perplexity. However, the benefit diminishes beyond $n_m = 4$, and the gains from $n_m = 8$ to $n_m = 16$ are marginal.

Downstream Evaluation. Tables 14 and 15 report zero-shot and two-shot accuracy across different values of n_m . The $n_m = 16$ setting achieves the best average accuracy, but the improvements over $n_m = 4$ and $n_m = 8$ are small. These results suggest that increasing the number of complementary masks helps, but overly large mask sets yield diminishing returns.

B.5 Partial Mask Ablation

In the standard single-mask MGLU layer (Eq. 2), the gate and value streams are computed from complementary subspaces of a shared weight matrix W defined by binary masks M and $\bar{M}$. To assess the necessity of these masks, we introduce three ablation variants that remove the masks from

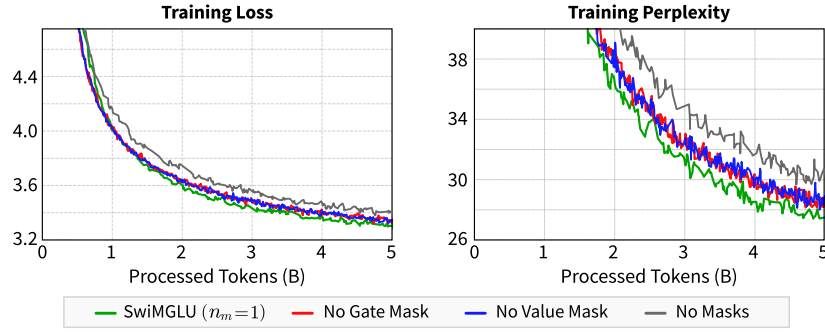


Figure 13: **Training loss and perplexity for mask-ablation variants.** All ablation variants—No Gate Mask, No Value Mask, and No Masks—converge to higher loss and perplexity compared to the fully masked MGLU baseline, highlighting the necessity of maintaining complementary mask-defined subspaces.

gate, value, or both streams:

$$\text{No Gate Mask (Dense Gate)} : h_{\text{NG}}(x) = g(xW) \odot (x(\overline{M} \odot W)), \quad (7)$$

$$\text{No Value Mask (Dense Value)} : h_{\text{NV}}(x) = g(x(M \odot W)) \odot (xW), \quad (8)$$

$$\text{No Masks (Fully Shared)} : h_{\text{NM}}(x) = g(xW) \odot (xW). \quad (9)$$

These variants maintain the multiplicative gate-value interaction characteristic of the MGLU architecture while systematically testing the significance of the mask-defined complementary subspaces.

Training Curves. Figure 13 compares training loss and validation perplexity for these mask-ablation variants in the *small* model setting. All ablation variants (No Gate Mask, No Value Mask, and No Masks) show slower convergence and higher final loss compared to the fully masked baseline. These results highlight the importance of distinct, complementary subspaces defined by masks for optimal MGLU performance.

C Mask Distribution

To understand how the model allocates capacity when multiple masks are learned, we measure the fraction of rows in the shared projection matrix W that each mask devotes to the *gate* pathway (higher values indicate more gate parameters; the remainder are routed to the value pathway). Figure 14 plots this layer-wise gate ratio for models trained with $n_m = 1, 2$, and 4.

All configurations exhibit a shallow U-shape: gate capacity is largest in the first layer, reaches a minimum around the middle of the network, and rises again toward the top. Because all masks share the same underlying weight matrix, their ability to partition rows adaptively provides a weight-efficient means of balancing gate and value capacity across depth—one that would be impractical with independently parameterized experts.

D PyTorch Implementation of MGLU

PyTorch implementations of the MGLU layer used for training are provided in Algorithm 2.

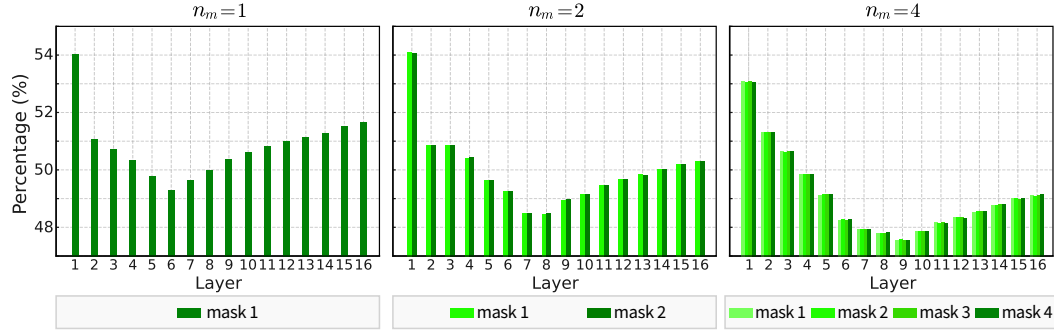


Figure 14: Layer-wise gate allocation for learned masks.

Algorithm 2 PyTorch-Style Implementation of MGLU ($n_m = 1$).

```

class MGLU(nn.Linear):
    def __init__(self, in_features, out_features):
        super(MGLU, self).__init__(in_features, out_features, False)
        self.register_parameter(
            "mask", nn.Parameter(0.01 * torch.randn(out_features, in_features), requires_grad=True)
        )

    # convert mask to binary by straight-through estimator
    def ste_mask(self, soft_mask):
        hard_mask = (soft_mask > 0).to(soft_mask.dtype)
        hard_mask = (hard_mask - soft_mask).detach() + soft_mask
        return hard_mask

    def forward(self, x):
        hard_mask = self.ste_mask(self.mask)
        # compute complementary output: e1, e2
        e1 = F.linear(x, self.weight * hard_mask)
        e2 = F.linear(x, self.weight * (1.0 - hard_mask))
        return e1, e2

```

E Efficient Kernel Implementation

A straightforward PyTorch implementation of a masked-GLU (MGLU) layer must (i) load the weight matrix from HBM multiple times—once for every mask bit—and (ii) launch several separate matmul + activation operations. Because each extra load traverses the bandwidth-limited HBM $\leftrightarrow$ SRAM link, such code quickly becomes *memory-bound*. By fusing the entire computation into a single CUDA kernel we touch each weight exactly *once*, keep partial results in registers, and remove almost all redundant global-memory traffic.

Implementation Details. Algorithm 3 shows the memory-access pattern and compute loop for the simplest MGLU case, $n_m = N_MASKS$. Each thread block is launched with coordinates (row, chunk) so that it processes one output row and one K -slice at a time. Two FP16 weights and activations are fetched together as a single `__half2` load, converted once to `float2`, and kept in registers for the entire multiply-accumulate step. The binary masks are packed eight bits per `int8`; testing the active bit yields a 0/1 scalar that is multiplied into the product. Both the ungated sum and each gated sum accumulate only in registers; after the loop a warp-level shuffle reduces these partials and a single atomic write per row sends the result to HBM.

Head-room on Hopper. Although the kernel already eliminates almost all redundant global-memory traffic, it deliberately avoids Hopper-specific optimisations such as `cp.async` and Tensor Memory Accelerator (TMA). Incorporating those features could conservatively deliver an additional $1.2\text{--}1.3\times$ speed-up on server-grade H100 GPUs—an avenue we leave to future work.

Latency Comparison. In Tables 16 and 17, we report the execution latency of our kernel versus a naïve Torch implementation on RTX 5090 and H100 GPUs, respectively; Tables 18 and 19 then compare our kernel against the standard GLU implementation on the same devices.

Algorithm 3 Simplified CUDA Implementation of MGLU ($n_m = N_MASKS$).

```
--global__ void mv_splitk_masks_kernel(  
    const __half* __restrict__ A, // [M x N], row-major  
    const __half* __restrict__ x, // [N]  
    const int8_t* __restrict__ mask, // [M x N],  
    float* __restrict__ y, // masked outputs  
    int M, int N, int split_k  
) {  
    int row = blockIdx.x; int chunk = blockIdx.y;  
    if (row >= M || chunk >= split_k) return;  
  
    // compute [start,end) of this K-chunk  
    int chunk_size = (N + split_k - 1) / split_k;  
    int start = chunk * chunk_size; int end = min(start + chunk_size, N);  
    int row_off = row * N;  
    int idx = start + threadIdx.x * 2; int stride = blockDim.x * 2;  
  
    float total = 0.0f; float msum[N_MASKS * 2] = {0.0f, 0.0f, ...};  
    for (; idx + 1 < end; idx += stride) {  
        __half2 a2 = *reinterpret_cast<const __half2*>(&A[row_off + idx]);  
        __half2 x2 = *reinterpret_cast<const __half2*>(&x[idx]);  
        float2 af = __half22float2(a2);  
        float2 xf = __half22float2(x2);  
        float2 prod = { af.x*xf.x, af.y*xf.y };  
        total += prod.x + prod.y;  
  
        int8_t m0 = __ldg(&mask[row_off + idx]);  
        int8_t m1 = __ldg(&mask[row_off + idx+1]);  
        unsigned int b0 = (unsigned int)m0;  
        unsigned int b1 = (unsigned int)m1;  
        #pragma unroll  
        for (int b = 0; b < N_MASKS; ++b) {  
            float mb0 = float((b0 >> b) & 1u); float mb1 = float((b1 >> b) & 1u);  
            if (b0 & (1u << b)) msum[b] += prod.x; if (b1 & (1u << b)) msum[b] += prod.y;  
        }  
    }  
    // reduce and write to HBM.  
}
```

Table 16: MGLU latency and speed-ups. Torch MGLU is the naïve `nn.Linear` implementation on an RTX 5090 GPU; higher ratios mean faster custom kernels.

n_m	h	d	CUDA (ms)	Triton (ms)	Torch (ms)	Torch/CUDA	Torch/Triton
1	8192	2048	0.0202	0.0516	0.0715	3.54×	1.39×
2	8192	2048	0.0210	0.0533	0.1358	6.47×	2.55×
4	8192	2048	0.0217	0.0606	0.2715	12.51×	4.48×
8	8192	2048	0.0265	0.0834	0.5210	19.66×	6.25×
1	14336	4096	0.1166	0.1342	0.3289	2.82×	2.45×
2	14336	4096	0.1169	0.1381	0.6001	5.13×	4.35×
4	14336	4096	0.1186	0.1454	1.1426	9.63×	7.86×
8	14336	4096	0.1229	0.1990	2.2172	18.04×	11.14×

Table 17: MGLU latency and speed-ups. Torch MGLU is the naïve `nn.Linear` implementation on a H100 GPU; higher ratios mean faster custom kernels.

n_m	h	d	CUDA (ms)	Triton (ms)	Torch (ms)	Torch/CUDA	Torch/Triton
1	8192	2048	0.0395	0.0639	0.1296	3.28×	2.03×
2	8192	2048	0.0409	0.0679	0.2213	5.41×	3.26×
4	8192	2048	0.0428	0.0810	0.4117	9.62×	5.08×
8	8192	2048	0.0483	0.1294	0.7910	16.38×	6.11×
1	14336	4096	0.1044	0.1191	0.3896	3.73×	3.27×
2	14336	4096	0.1067	0.1239	0.7080	6.64×	5.71×
4	14336	4096	0.1110	0.1608	1.3454	12.12×	8.37×
8	14336	4096	0.1215	0.3202	2.6354	21.68×	8.23×

Table 18: CUDA MGLU latency and speed-ups against the standard PyTorch GLU baseline (no masking) on an RTX 5090 GPU. Lower latency and higher speed-up are better.

n_m	h	d	CUDA MGLU (ms)	Torch GLU	GLU/CUDA
1	8192	2048	0.0202	0.0306	1.51 ×
2	8192	2048	0.0210	0.0306	1.46 ×
4	8192	2048	0.0217	0.0306	1.41 ×
8	8192	2048	0.0265	0.0306	1.15 ×
1	14336	4096	0.1166	0.1530	1.31 ×
2	14336	4096	0.1169	0.1530	1.31 ×
4	14336	4096	0.1186	0.1530	1.29 ×
8	14336	4096	0.1229	0.1530	1.24 ×

Table 19: CUDA MGLU latency and speed-ups against the standard PyTorch GLU baseline (no masking) on a H100 GPU. Lower latency and higher speed-up are better.

n_m	h	d	CUDA MGLU (ms)	Torch GLU	GLU/CUDA
1	8192	2048	0.0395	0.0485	1.23 ×
2	8192	2048	0.0409	0.0485	1.19 ×
4	8192	2048	0.0428	0.0485	1.13 ×
8	8192	2048	0.0483	0.0485	1.00 ×
1	14336	4096	0.1044	0.1215	1.16 ×
2	14336	4096	0.1067	0.1215	1.14 ×
4	14336	4096	0.1110	0.1215	1.09 ×
8	14336	4096	0.1215	0.1215	1.00 ×

F Broader Impacts

The method proposed in this paper will not lead to negative societal impact. By reducing memory-bandwidth pressure and per-token latency, it cuts the energy consumption of large-language-model inference and therefore lowers carbon emissions. These efficiency gains also make advanced language capabilities viable on commodity and edge hardware, broadening access while trimming operational costs.