FUSE: Mixture of Experts as Block Sparse Attention

Chenwei Cui[*]
Arizona State University
Tempe, AZ
ccui17@asu.edu

Benjamin Joseph Herrera[*]
Arizona State University
Tempe, AZ
b10@asu.edu

Hannah Kerner
Arizona State University
Tempe, AZ
hkerner@asu.edu

Abstract

Large language models (LLM) are increasingly using Sparse Mixture-of-Experts (SMoE) to improve computational efficiency. In a resource-constrained setting, it is ideal to minimize the expert size while increasing the overall number of experts. However, existing SMoE frameworks struggle to handle a large number of small experts, often resulting in expert overload, under-use, or token dropping. In this paper, we propose FUSE: **F**lexible and **U**nified **S**parse mixture of **E**xperts. FUSE treats MoE computation as block sparse attention and uses FlexAttention for training and inference. This approach allows us to pre-train models of up to 576 experts per layer or 1.4 billion parameters with only one H100 GPU using pure PyTorch. Without requiring a custom CUDA kernel, we expand the activation function support from softmax to GeLU and ReLU through a reversal trick, resulting in a big performance increase. Moreover, expressing Sparse MoE as a form of attention allows us to fuse feedforward and self-attention layer in a single CUDA kernel call. This variant, termed Unified MoE (UMoE), enables higher GPU occupancy and faster autoregressive decoding. Experiments show, as we increase the number of dormant parameters, SMoE and UMoE under the FUSE framework consistently improves in FineWeb-EDU perplexity and zero-shot downstream performance, all while having negligible wall clock time increase. When compared to a dense model, FUSE reaches the same model performance with 75% H100 hours of pre-training, without expert parallelism. This makes FUSE single-GPU friendly. Our approach enables billion-parameter-scale LLM pre-training and inference under resource-constrained environments. This would speed up the ideation and experimentation from researchers with limited compute. All our billion-scale SMoE experiments will be open-sourced and can be replicated on a single H100 in under 24 hours.

1 Introduction

Sparse Mixture-of-Experts (SMoEs) have been proposed to increase the computational efficiency of Large Language Models (LLMs), pushing the scale and capability of frontier models [1–4]. SMoEs also have the potential to make larger-scale ML research more accessible, speeding up the ideation and exploration of many researchers under resource-constrained or edge-level settings.

However, existing SMoE frameworks are yet to realize this ideal. In a perfect scenario, we want to minimize the expert size, reduce the number of active experts, and drastically increase the number of dormant experts. Mainstream SMoE libraries [5] relies on multi-GPU expert parallelism and risks token dropping when the number of experts is high. Frameworks that introduces flexibility such as MegaBlocks [6] require custom CUDA kernels — a software lottery [7], which hinders further customization. These challenges make it difficult to utilize SMoEs in low-resource settings.

[*]Equal contribution. Correspondence to Hannah Kerner <hkerner@asu.edu>

On the other side of architecture research, optimization for attention modules have matured considerably, both in theory and application. Due to the higher interest from the community and industry, advanced algorithms such as FlashAttenion-1/2/3 [8–10] are quickly integrated into PyTorch [11]. Notably, the recent FlexAttention library [7] offers a way to combat the software lottery, offering a pure-PyTorch interface to perform block sparse attention operations. Prior research has identified the equivalence between attention and MLP layers [12], therefore, optimizing (sparse) feedforward layers through attention is a promising route. Currently, there are research focusing on interpreting attention as a type of fast MLP [13], as well as interpreting MLPs in attention terms [14, 15]. We present the first work on interpreting and scaling up SMoE from the perspective of block sparse attention.

In this paper we introduce **F**lexible and **U**nified **S**parse mixture of **E**xperts (FUSE), a framework that reformulates SMoE as block-sparse attention and employs the highly optimized FlexAttention to perform the actual computation [7]. FUSE avoids the need of implementing custom CUDA kernels, eliminates token dropping, and is highly efficient even when pre-training on a single GPU. To expand support beyond softmax activation, we propose the reversal trick, unlocking GeLU and ReLU activations without altering the underlying CUDA kernel. As part of hardware-software co-design, instead of the vanilla MLP-based experts, we implement multi-head experts that map naturally to FlexAttention's SRAM budget, allowing for reduced head size. A surprising benefit of expressing MoE in block sparse attention terms is the ability to seamlessly fuse feedforward and self-attention layer in a single CUDA kernel call. This model variant, termed Unified MoE (UMoE), shows faster autoregressive decoding when compared with its SMoE counterpart.

These optimizations allow us to aggressively scale up the number of dormant experts on a single H100 GPU, up to 576 experts. Our experiments show sustained model performance increases in terms of language modeling perplexity and downstream task performance. This is alongside as we increase the number of dormant experts, while fixing the number of active experts. On FineWeb-EDU [16] pre-training, SMoE with FUSE matches dense-model perplexity with just 75 % of the H100 hours and requires consistently lower FLOPs at the same downstream performance.

Our work lowers the hardware and software barriers for billion-scale language model pre-training. We believe FUSE will enable broader and diverse exploration within the research community.

2 Background

2.1 Notation

We use bold lowercase letters for vectors and bold uppercase letters for matrices. Vectors inside matrices are placed at a row-by-row format. To describe the size of a neural network, we use D_e for embedding size and D_h for the number of hidden neurons. N represents the sequence length and N_E represents the total number of experts. θ represents the set of parameters in a model.

2.2 Multi-Layer Perceptron

Multi-Layer Perceptron (MLP) [17] is a classical neural network module that can fit complex mappings. Its single-hidden-layer variant is the key building block for modern deep learning [18].

Definition 2.1 (MLP Module)**.** Consider a non-linear activation function ψ. For a sequence of feature vectors $\mathbf{X} = \{\mathbf{x}_i\}_{i=1}^N \in \mathbb{R}^{N \times D_e}$ and parameters $\mathbf{W}_{in} \in \mathbb{R}^{D_h \times D_e}$ and $\mathbf{W}_{out} \in \mathbb{R}^{D_h \times D_e}$, an MLP module is defined as

$$f_\theta(\mathbf{X}) = \psi(\mathbf{X}\mathbf{W}_{in}^T)\mathbf{W}_{out} \tag{1}$$

2.3 Attention

The attention module [19] first computes the scaled dot product between every query and key vector to form an attention score matrix. A row-wise softmax then converts these scores into attention weights, which are used to retrieve from a set of value vectors. This mechanism, called Scaled Dot-Product Attention (SDPA), powers the Transformer architecture [20].

Definition 2.2 (SDPA Module)**.** Consider sequences of vectors $\mathbf{Q} = \{\mathbf{q}_i\}_{i=1}^N \in \mathbb{R}^{N \times D_e}$, $\mathbf{K} = \{\mathbf{k}_j\}_{j=1}^{D_h} \in \mathbb{R}^{D_h \times D_e}$, and $\mathbf{V} = \{\mathbf{v}_j\}_{j=1}^{D_h} \in \mathbb{R}^{D_h \times D_e}$. Let $\mathbf{c} \in \mathbb{R}$ be a constant scaling factor. The attention score matrix $\mathbf{A}$ is defined as $\mathbf{A} = \frac{\mathbf{Q}\mathbf{K}^T}{\mathbf{c}}$. Applying the softmax function row-wise, we get

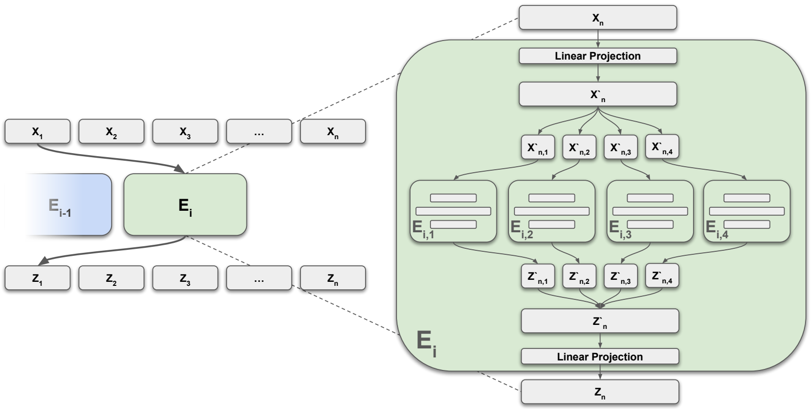


Figure 1: **Multi-Head Experts.** This figure illustrates how an input feature vector $\mathbf{X}_n$ is transformed into the output $\mathbf{Z}_n$. The figure assumes four heads.

the attention weight matrix $\mathbf{S} = \text{Softmax}(\frac{\mathbf{QK}^T}{\mathbf{c}})$. The output of the SDPA module is

$$\mathbf{O} = \mathbf{SV} = \text{Softmax}(\frac{\mathbf{QK}^T}{\mathbf{c}})\mathbf{V}. \tag{2}$$

There are known similarities between attention and MLP modules [12]. Specifically, the input to an attention module corresponds to $\mathbf{Q}$, and its activation function is row-wise softmax. The matrices $\mathbf{K}$ and $\mathbf{V}$ in attention are analogous to $\mathbf{W}_{in}$ and $\mathbf{W}_{out}$ in an MLP.

In practice, one caveat is the activation function. Common SDPA implementations in mainstream libraries implements softmax. However, there is a significant performance gap between softmax and other activation functions such as GeLU (see Section 4.2). Therefore, in Section 3.4 we propose a strategy to bypass this limit and implement different activation functions.

3 Methodology

3.1 Sparse Mixture-of-Experts with Multi-Head Experts

The expert networks in a Sparse Mixture-of-Experts (SMoEs) [21, 22] architecture are typically implemented as MLPs. From a hardware-software co-design standpoint, instead of using standard MLP-based experts, we introduce multi-head experts with smaller head sizes, that fits well with the SRAM limits of GPUs.

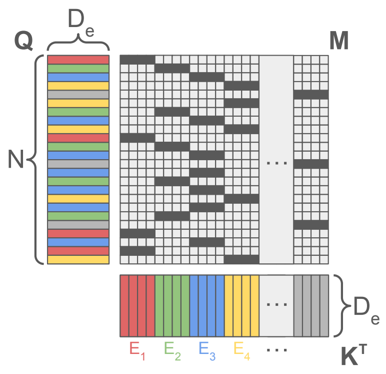
As shown in Figure 1, a multi-head expert is a neural network that contains multiple sub-experts. When a token $\mathbf{X}_n$ is passed to a multi-head expert E_i, it is first linearly projected to a vector $\mathbf{X}'_n$ of the same size. It is then divided into a number of sub-tokens and passed to its respective sub-expert. Each sub-expert is a smaller MLP module. The output of all sub-experts are later combined and linearly projected into the output token $\mathbf{Z}_n$.

Although we introduce an expert type that can be fitted into limited SRAM space, the gating function can face imbalance issues [23, 5] where router selects certain experts more than other experts. We use the standard load balancing loss to mitigate this (See Appendix A).

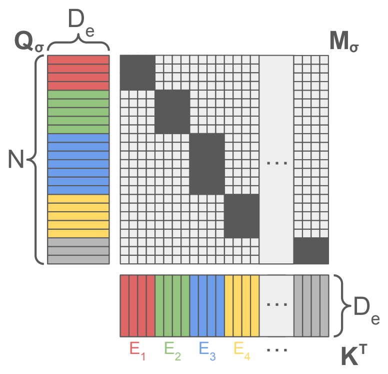
3.2 Formulating MoEs as Block Sparse Attention

MoEs assign experts to tokens and aggregate the outputs from each expert per token. A way to apply this is via block sparse attention. For simplicity, we explain the formulation assuming a single activated expert, as multiple activated experts can always be converted into a single expert case.

Proposition 3.1 (MoE and Attention Equivalence)**.** *An MoE is equivalent to an attention module with the addition of a expert assignment mask* $\mathbf{M} \in \{1, -\infty\}^{N \times N_E}$*. This mask is implemented to*



(a) Before Packing



(b) After Packing

Figure 2: **Packing Demonstration**. This illustration demonstrates the permutation with σ. For brevity, this illustration omits the computation of the score matrix to $\mathbf{V}$ in f_{MA}. An assignment mask $\mathbf{M}$ is represented with dark gray, and is illustrated over the scoring matrix.

the attention module f_{MA} via a hadamard product to $\mathbf{S}$ *as*

$$f_{MA}(\mathbf{Q}, \mathbf{K}, \mathbf{V}, \mathbf{c}; \mathbf{M}) = \mathit{Softmax}(\frac{\mathbf{Q}\mathbf{K}^T}{\mathbf{c}} \odot \mathbf{M})\mathbf{V} \tag{3}$$

Proof. We utilize some input matrix $\mathbf{X}$ of the MoE module as $\mathbf{Q}$. A set of experts E can be defined with the key matrix $\mathbf{K}$. In other words, each row of the $\mathbf{K}$ represents the $\mathbf{W}_{in}$ of its respective MLP expert in $\mathbf{E}$. This makes the shape of $\mathbf{K}$ into $\mathbb{R}^{N_E \times D_e}$. The resulting projection of $\mathbf{Q}$ onto $\mathbf{K}^T$ has a shape of $\mathbb{R}^{N \times N_E}$. To produce some expert assignment mask $\mathbf{M}$, we utilize the Top-K routing function. This in turn introduces another parameter to the attention module, $\mathbf{W}_G$. From the Top-K indexing function, we utilize the resulting expert assignment matrix $\mathbf{K}^{'}$ to define each element m_{ij} of $\mathbf{M}$ as

$$m_{ij} = \begin{cases} 1 & \text{if } j \text{ is in } \mathbf{K}_i^{'} \\ -\infty & \text{otherwise} \end{cases} \tag{4}$$

where $\mathbf{K}_i^{'} \in \mathbb{Z}^k$. The assignment expert mask $\mathbf{M}$ is element-wise multiped to the attention score matrix, $\mathbf{A}$, in which a Softmax is applied afterwards. The resulting output from the Softmax is then projected onto $\mathbf{V} \in \mathbb{R}^{N_E \times D_e}$, in which each row of $\mathbf{V}$ represents the $\mathbf{W}_{out}$ of the respective expert. □

3.3 Improving Access Pattern with Packing and Unpacking

As shown in Figure 2, the initial block masks can be scattered. To improve the access pattern of GPUs, expert assignments needs be grouped together. To achieve this, we propose a packing and unpacking algorithm. Figure 2 supplements this definition with a visual conception of the approach.

Definition 3.2 (Packing and Unpacking Algorithm)**.** Using the assignment matrix $\mathbf{K}$ returned by the SMoE gate, we can represent $\mathbf{K}^{'} \in \mathbb{R}^N$ as the top t-th expert assignments for $\mathbf{X}$ (or $\mathbf{Q}$ when referenced inside f_{MA}). To group assignments together, an argsort is applied to $\mathbf{K}$ to get a sorted mapping $\sigma : \{1, \ldots, N\} \to \{1, \ldots, N\}$. To apply this sorting, the inputs $\mathbf{X}$ and mask $\mathbf{M}$ are permuted with σ in a cross-row manner to get $\mathbf{X}_\sigma$ and $\mathbf{M}_\sigma$, respectively. When passing $\mathbf{X}_\sigma$ and $\mathbf{M}_\sigma$ to f_{MA}, we get some output $\mathbf{T}_\sigma$. To reverse the effects of the permutation, we apply an inverse mapping σ^{-1} to T_σ to get an original output $\mathbf{T}$.

Remark 3.3. Note that the algorithm laid out in Definition 3.2 is applicable if a single attention-based MoE utilizes only one active expert. In cases where $k > 1$ for a single attention-based MoE, this packing-and-unpacking algorithm becomes non-trivial to organize expert assignments. Still, multiple activated experts can always be converted into a single expert case by duplicating the input sequences. We are able to experiment on up to 4 activated experts without noticing overhead.

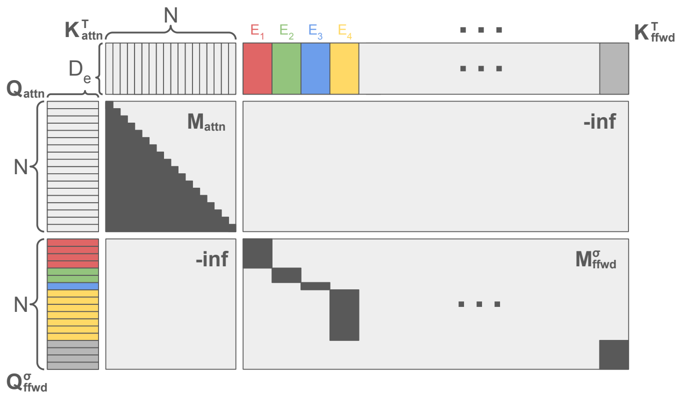


Figure 3: **Unified attention architecture.** Two sets of query and key matrixes are represented: attention queries ($\mathbf{Q}_{\text{attn}}$) and keys ($\mathbf{K}_{\text{attn}}$), and feed-forward queries ($\mathbf{Q}_{\text{ffwd}}$) and keys ($\mathbf{K}_{\text{ffwd}}$). The queries and keys are combined to one another.

3.4 The Reversal Trick to Support Arbitrary Activation Functions

Mainstream attention libraries implement softmax as the activation function. While this is the standard for self-attention, softmax function is not a performant activation function for MLP or MoE [15] (See 4.2). We propose a reversal trick to allow for other activation functions.

Consider Definition 2.2 and its definition of each element $\mathbf{s}_{ij}$ in the normalized scoring matrix $\mathbf{S}$. To revert the effect of softmax and apply arbitrary activation function ψ, it is ideal to get $\mathbf{s}_{ij} = \sigma(\mathbf{a}_{ij})$. We can define a vector $\mathbf{s}_{SE} \in \mathbb{R}^N$ that contains the summation of each exponentiated element within $\mathbf{S}_i$. This correspond to the `return_lse` function of FlexAttention. A vector $\mathbf{s}'_{SE}$ is a vector where all elements inside it are inverted. We can then define the output of the normalized scoring matrix as $\mathbf{S} = \text{diag}(\mathbf{s}'_{SE}) \cdot \exp(\mathbf{A})$. By extension, the output of an attention model can be defined as $\mathbf{O} = (\text{diag}(\mathbf{s}'_{SE}) \cdot \exp(\mathbf{A})) \cdot \mathbf{V}$. To cancel out the effects of $\text{diag}(\mathbf{s}'_{SE})$, we can multiply $\text{diag}(\mathbf{s}_{SE})$ into $\mathbf{O}$ like such:

$$\begin{aligned} \mathbf{O} &= \text{diag}(\mathbf{s}_{SE}) \cdot (\text{diag}(\mathbf{s}'_{SE}) \cdot \exp(\mathbf{A})) \cdot \mathbf{V} \\ &= (\text{diag}(\mathbf{s}_{SE}) \cdot \text{diag}(\mathbf{s}'_{SE})) \cdot (\exp(\mathbf{A}) \cdot \mathbf{V}) \\ &= (\exp(\mathbf{A}) \cdot \mathbf{V}). \end{aligned} \tag{5}$$

In turn each element inside $\mathbf{S}$ is now defined as $s_{ij} = e^{(a_{ij})}$. To remove the exponentiation, we can redefine a_{ij} as $a'_{ij} = \log(\beta + \psi(a_{ij}))$, resulting $e^{(a'_{ij})} = \beta + \psi(a_{ij})$. This can be achieved using the `score_mod` function of FlexAttention [7]. The introduction of β alongside the activation function ψ is meant to ensure positive values inside the log function. This means that $\beta = \min(\psi(x))$, allowing for further application of activation functions. Should $\psi(x) \to -\infty$, β can be set to be any arbitrary positive value and $-\beta$ can be used as the empirical minimum of all outputs $\psi(x)$. With this in mind, the output of the attention module is now $(\psi(A) + \mathbf{B}) \cdot V$, where $\mathbf{B} = \beta \cdot \mathbf{1} \in \mathbb{R}^{N \times D_h}$. We can then subtract the output by $(\mathbf{B} \cdot \mathbf{V})$ to get attention based output with any arbitrary activation function as:

$$\begin{aligned} \mathbf{O} &= (\exp(\mathbf{A}) \cdot \mathbf{V}) - (\mathbf{B} \cdot \mathbf{V}) \\ &= ((\psi(\mathbf{A}) + \mathbf{B}) \cdot \mathbf{V}) - (\mathbf{B} \cdot \mathbf{V}) \\ &= (\psi(\mathbf{A}) \cdot \mathbf{V}) + (\mathbf{B} \cdot \mathbf{V}) - (\mathbf{B} \cdot \mathbf{V}) \\ &= (\psi(\mathbf{A}) \cdot \mathbf{V}). \end{aligned} \tag{6}$$

3.5 Unified Computation

A decoder transformer block contains two main components, an attention module and a feed forward module. Under the FUSE framework, these main components of a decoder layer can be combined into one module, similar to PaLM-1 [24],

$$y = x + \text{Attention}(x) + \text{FeedForward}(x). \tag{7}$$

This allow for kernel fusing and improved GPU occupancy during implementation. For simplicity, we explain the kernel fusing strategy assuming one active expert.

We first define one combined query matrix $\mathbf{Q} \in \mathbb{R}^{2N \times D_e}$ derived by concatenating two duplicates of inputs $\mathbf{X} \in \mathbb{R}^{N \times D_e}$, sequentially. These duplicates can be referenced as $\mathbf{Q}_{\text{attn}}$ and $\mathbf{Q}_{\text{ffwd}}$, both of shape $\mathbb{R}^{N \times D_e}$. The same can be applied for the combined keys matrix $\mathbf{K} \in \mathbb{R}^{(N+N_E) \times D_e}$ of the attention ($\mathbf{K}_{\text{attn}} \in \mathbb{R}^{N \times D_e}$) and feed-forward ($\mathbf{K}_{\text{ffwd}} \in \mathbb{R}^{N \times D_e}$) modules. We introduce a unified mask $\mathbf{M} \in \mathbb{R}^{2N \times (N+N_E)}$ which contains the different mask strategies for the combined attention and feed forward modules. This means that if the feed forward module is a MoE with a packing and unpacking algorithm, then $\mathbf{Q}_{\text{ffwd}}$ and $\mathbf{M}_{\text{ffwd}} \in \mathbb{R}^{N \times N_E}$ will be permuted with σ, turning into $\mathbf{Q}^{\sigma}_{\text{ffwd}}$ and $\mathbf{M}^{\sigma}_{\text{ffwd}}$, respectively. When calculating the scoring matrix $\mathbf{S} \in \mathbb{R}^{2N \times (N+N_E)}$ with some unified mask $\mathbf{M}$, we can define each quadrant of the scoring matrix as

$$\mathbf{A} = \mathbf{Q}\mathbf{K}^T \odot \mathbf{M} = \mathbf{Q}\mathbf{K}^T \odot \begin{bmatrix} \mathbf{M}_{\text{attn}} & -\infty \\ -\infty & \mathbf{M}^{\sigma}_{\text{ffwd}} \end{bmatrix} = \begin{bmatrix} \mathbf{A}_{\text{attn}} & -\infty \\ -\infty & \mathbf{A}^{\sigma}_{\text{ffwd}} \end{bmatrix} \tag{8}$$

where $-\infty$ represents the quadrant of the scoring matrix filled with $-\infty$. Additionally, $\mathbf{M}_{\text{attn}} \in \mathbb{R}^{N \times N}$, and $\mathbf{A}_{\text{attn}} \in \mathbb{R}^{N \times N}$ and $\mathbf{A}^{\sigma}_{\text{ffwd}} \in \mathbb{R}^{N \times N_E}$ are the scoring quadrant matrices for the attention and the MoE components, respectively. For further reference, Figure 3 illustrates the structure of computing the scoring matrix. In practice, FlexAttention [7] handles the sparsity highly efficiently, in line with the theoretical savings.

4 Experiments

4.1 Experimental Setup

Model Definitions To analyze the methodologies introduced above, we configure a wide range of models. Given a Model ID, the corresponding hyperparameters can be found in Appendix E. All MoE models are trained and evaluated with only one activated expert, unless specified. Notably, we take two type of approaches to scale up the number of parameters of a model: width scaling and optimal scaling. The former scales up only the number of hidden neurons, while the latter scales up the depth as well as other aspects. The former is in line with the scaling pattern of SMoEs.

Training and Evaluation Settings All models are pre-trained on 10 billion tokens of the FineWeb-EDU [16] dataset, a subset of the larger FineWeb corpus. A 100 million token subset is withheld from training for validation. We use 4 H100s to pre-train the models and 1 H100 to evaluate all models analyzed in this section. All experiments can be performed on a single H100 GPU.

Section Structure Section 4.2 conducts ablation study to show the efficacy of different approaches mentioned in Section 3. Section 4.3 presents the performance of the configured models on downstream tasks. Section 4.4 provides a joint evaluation of model performance and computational efficiency.

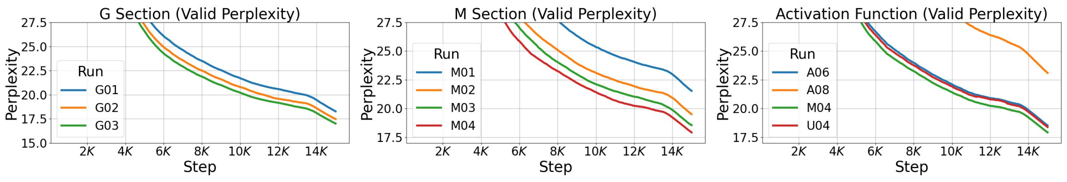
4.2 Ablation Study

With various hyperparameters and model configurations, we show three significant considerations when setting SMoE models.

Granularity Part (a) of Figure 4 represents SMoEs of granularities of 1, 2, and 4 from top to bottom, respectively. The results show that as the granularity increases, the validation perplexity decreases. This falls in line with standard behaviors of MoE models.

Total Experts In Figure 4 (b), we fix the number of activated experts while increasing the number of total experts N_E up to 576. The validation perplexity drops quickly as the size of the model increases. It is worth noting that as we increase N_E, the total H100 hours required to pre-train the model stays largely constant, as shown in Figure 5.

Activation In Figure 4, we notice that softmax performs poorly compared to ReLU and GeLU. The perplexity curve for a SMoE of softmax activation (A08) is significantly higher than its ReLU and GeLU counterparts (A06, M04, and U04). This supports the use of our proposed reversal trick. We also notice that ReLU performs very similarly to GeLU activate experts.



(a) G Series (b) M Series (c) Activation Function

Figure 4: **Validation Perplexity Across Different Runs.** The above graphs represent the validation perplexity as models trained. (a) Represents the validation perplexities as the granularity of a 1.3B model increases. (b) Shows the validation perplexity as we increase the number of total experts while keeping the number of active experts as 1. (c) Demonstrates perplexity performance across Softmax, GeLU, and ReLU activated experts.

4.3 Model Performance

Number of Dormant Experts A key part of the work proposed is the ability to significantly increase the amount of dormant/inactive experts. As Discussed for part (b) of Figure 4, the validation perplexity decreases as the size of the model increases. In terms of actual compute shown in Figure 5, the amount of H100 hours needed to train such larger dormant models remains the same to it's smallest configuration. Combined with the results presented in Figure 6, the proposed work attain considerable performance advantages given a limited compute budget.

Model Scaling Behavior Model scaling is presented by the amount of FLOPs and H100-hours needed to train one model. We show that as the size of the model increases, the amount of FLOPs and H100-hours to train increases for MLP models, with optimal or width scaling. However, UMoEs and SMoEs remain largely constant in terms of the amount of compute needed. Combined with the findings in the theoretical and empirical efficiency analysis in Section 4.4, these results show that performance gains can be achieved with the same FLOPs and H100 time on training any sized SMoE and UMoE model, bounded only by the VRAM of GPUs (see Figure 5).

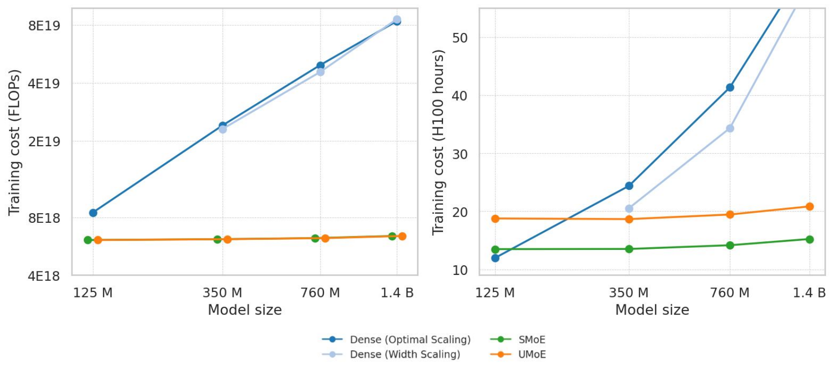


Figure 5: **Scaling Behavior of Training Cost.** The left side of the plot represents the FLOPs cost to train a single model, while the right shows the amount of H100-hours needed for training.

Downstream Tasks Several models are presented with varying configurations in terms of d_e, model size, granularity, and activation function. Downstream task evaluations include: Hellaswag [25], PiQA [26], Lambada (OpenAI) [27], and ARC Easy and Challenge [28]. Benchmark runs are reported on the raw accuracy and with zero context. In addition to evaluation scores, each table reports a validation perplexity score to show pretraining performance over the validation set. All models presented below are accompanied with their respective ID, which can be crossreferenced in Appendix E for further information about the evaluated models.

ID	Model	Size	FLOPs $\times 10^{18}$	H100-Hours	FineWeb ppl ↓	Hellaswag acc ↑	PiQA acc ↑	LAMBADA acc ↑	Arc E. acc ↑	Arc C. acc ↑	Average acc ↑
B01	MLP$_o$	125M	8.46	11.96	20.14	29.33%	62.51%	26.41%	53.24%	21.33%	38.56%
B02	MLP$_o$	350M	24.00	24.38	16.95	31.97%	66.27%	29.21%	57.62%	22.95%	41.60%
B03	MLP$_o$	760M	49.40	41.33	15.23	33.79%	67.57%	33.53%	61.03%	25.26%	44.24%
B04	MLP$_o$	1.3B	83.80	62.98	14.49	34.43%	66.92%	34.41%	61.32%	25.94%	44.60%
P01	MLP$_p$	360M	23.00	20.53	17.32	31.48%	63.93%	27.07%	56.10%	23.04%	40.32%
P02	MLP$_p$	760M	45.50	34.31	16.29	32.66%	67.57%	29.40%	58.80%	24.74%	42.64%
P03	MLP$_p$	1.4B	85.80	59.45	15.51	33.97%	67.25%	30.16%	59.81%	26.28%	43.49%
M01	SMoE	125M	6.09	13.48	21.16	29.19%	61.92%	25.40%	50.13%	20.99%	37.52%
M02	SMoE	360M	6.15	13.51	19.19	30.18%	63.60%	26.53%	52.74%	21.84%	38.98%
M03	SMoE	760M	6.23	14.15	18.27	30.93%	64.69%	26.12%	55.26%	21.76%	39.75%
M04	SMoE	1.4B	6.39	15.22	17.62	31.81%	64.74%	29.52%	54.97%	24.23%	41.05%
U01	UMoE	125M	6.09	18.76	21.71	28.82%	62.35%	25.03%	51.35%	21.08%	37.73%
U02	UMoE	360M	6.15	18.65	19.74	30.22%	63.55%	26.97%	52.44%	20.05%	38.65%
U03	UMoE	760M	6.23	19.44	18.82	30.43%	63.60%	25.36%	54.25%	23.46%	39.42%
U04	UMoE	1.4B	6.39	20.83	18.08	31.55%	65.51%	26.94%	53.87%	22.18%	40.01%

Table 1: **Model Performance on Downstream Tasks With the same** d_e. The models listed above have the same d_e at 768 and are grouped by similar model size. MLP$_p$ represents MLP models with the same d_e (width scaling), while MLP$_o$ represents models of increasing d_e (optimal scaling). The latter is presented in this table for additional comparisons.

Table 1 presents compute costs, validation perplexity, and benchmark evaluations as the model size increases with the same d_e. It is expected that MLP models will outperform the MoE models presented, both in validation perplexity and downstream tasks. However, the MoE models continue to scale in performance as the their size increases, showing that such models do learn in larger size configurations. In certain downstream tasks, the MoE models perform slightly similar to their MLPs counterparts. In addition to the presented fixed d_e models are the B-series models (MLP$_o$) that have increasing d_e sizes. These MLP$_o$ models achieve better perplexity and downstream task performance compared to everything else. As shown in Section 4.4, the amount of activated parameters is significantly higher in such MLP models, while the MoEs and SMoEs have consistent active parameters. Thus, the MoE performances listed above are comparable given the current resource environment that they are deployed under.

ID	Model	Granularity	σ	Size	FineWeb ppl ↓	Hellaswag acc ↑	PiQA acc ↑	LAMBADA acc ↑	Arc E. acc ↑	Arc C. acc ↑	Average acc ↑
G01	SMoE	1	ReLU	1.3B	17.85	31.56%	64.42%	28.10%	54.55%	22.87%	40.30%
G02	SMoE	2	ReLU	1.3B	17.11	31.86%	65.23%	30.14%	56.86%	23.55%	41.53%
G03	SMoE	4	ReLU	1.3B	16.62	32.25%	65.29%	30.78%	57.20%	24.06%	41.92%
P03	MLP	1	GeLU	1.4B	15.51	33.97%	67.25%	30.16%	59.81%	26.28%	43.49%
M04	SMoE	1	GeLU	1.4B	17.62	31.81%	64.74%	29.52%	54.97%	24.23%	41.05%
U04	UMoE	1	GeLU	1.4B	18.08	31.55%	65.51%	26.94%	53.87%	22.18%	40.01%

Table 2: **SMoE Performance on Downstream Tasks With Increasing Granularity**. Similar in structure to Figures 1, the models listed above have roughly the same model size and d_e at 768. The first three rows report the models with increasing granularity and with ReLU as the activation function. The last three have a granularity of one and GeLU for activation function.

Table 2 demonstrates benchmark scores with increasing granularity and ReLU as the activation function with a model size of 1.3 billion parameters. Other models of the same size, but with different granularity and activation function are presented alongside for easier comparison. As the granularity increases, the number of activated parameter also increases. This in turn enables SMoEs of the same model size to attain better performance in such increases, leading to an average 1.62% in benchmark performance. The multi-activated expert SMOE models (G series) attain better performance than the GeLU-based MoEs. Although it underperforms on benchmarks compared to its MLP counterpart, the number of activate experts is still significantly less.

4.4 Computational Efficiency

Theoretical vs Empirical Efficiency In Figure 6 we compare FUSE-based SMoE models with dense MLP baselines. Theoretically, the sparse models introduce negligible additional FLOPs as they scale, yet validation perplexity improves steadily. Empirically, although total H100-hours rise with model size, the SMoE points remain on the Pareto front: for any target perplexity they require far fewer hardware hours than dense models. The high-granularity models push perplexity even lower while fitting the same VRAM budget. These results demonstrate that FUSE-based SMoEs scale more efficiently than dense models under tight compute budgets.

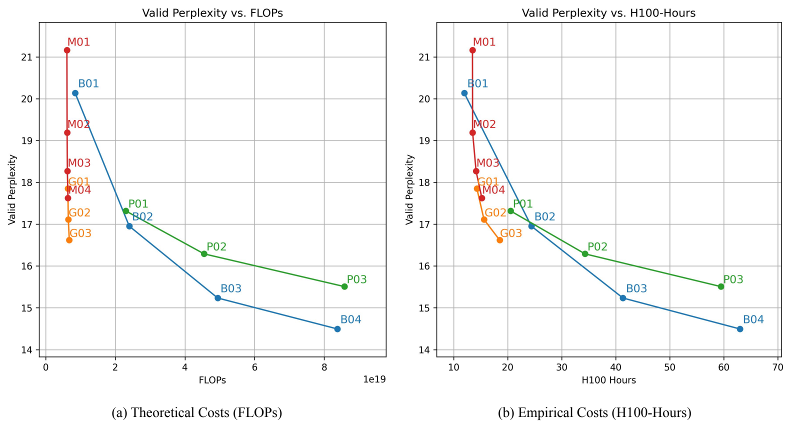


Figure 6: **Model performance versus computational cost.** (a) Perplexity versus theoretical FLOPs for four model families: dense MLPs with optimal scaling (B), dense MLPs with width-only scaling (P), SMoE models with FUSE (M), and higher-granularity SMoE models with FUSE (G). (b) Likewise, perplexity over H100-Hours is plotted for the same families of models.

Throughput Throughput analysis is done by testing a UMoE and SMoE model's inference speed over some set of random tokens. Shown in Figure 7, UMoE's consistently outperformed SMoE as the model size increased. This falls in line with the expectations of the unified approach fuses the sparse feedforward and self-attention layers in one CUDA kernel call.

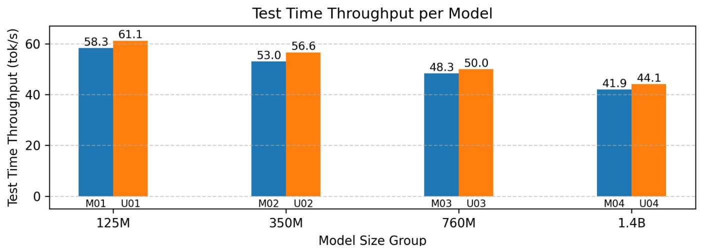


Figure 7: **Test Time Throughput.** Test time throughput is compared between SMoEs and UMoEs. Throughput measurements are grouped by model size from 125 million to 1.4 billion parameters.

5 Conclusion

In this paper, we propose FUSE, a new framework for SMoEs that allow for drastically increased number of experts and significant computational savings. Our results demonstrate advantages in computational efficiency. With leveled FLOP usage and near constant H100-hours, FUSE allows pre-training billion-parameter-scale language models under resource-constrained settings. We notice that the scaling of current approach is bounded not by the FLOPs but by the amount of VRAM a GPU has. Therefore, one interesting future direction is to explore memory-efficient techniques such as parameter offloading.

References

[1] Shiyi Cao, Shu Liu, Tyler Griggs, Peter Schafhalter, Xiaoxuan Liu, Ying Sheng, Joseph E. Gonzalez, Matei Zaharia, and Ion Stoica. Moe-lightning: High-throughput moe inference on memory-constrained gpus. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ASPLOS '25, page 715–730, New York, NY, USA, 2025. Association for Computing Machinery. ISBN 9798400706981. doi: 10.1145/3669940.3707267. URL `https://doi.org/10.1145/3669940.3707267`.

[2] Haiyang Huang, Newsha Ardalani, Anna Sun, Liu Ke, Shruti Bhosale, Hsien-Hsin Lee, Carole-Jean Wu, and Benjamin Lee. Toward efficient inference for mixture of experts. *Advances in Neural Information Processing Systems*, 37:84033–84059, 2024.

[3] Changho Hwang, Wei Cui, Yifan Xiong, Ziyue Yang, Ze Liu, Han Hu, Zilong Wang, Rafael Salas, Jithin Jose, Prabhat Ram, et al. Tutel: Adaptive mixture-of-experts at scale. *Proceedings of Machine Learning and Systems*, 5:269–287, 2023.

[4] Nan Du, Yanping Huang, Andrew M Dai, Simon Tong, Dmitry Lepikhin, Yuanzhong Xu, Maxim Krikun, Yanqi Zhou, Adams Wei Yu, Orhan Firat, et al. Glam: Efficient scaling of language models with mixture-of-experts. In *International conference on machine learning*, pages 5547–5569. PMLR, 2022.

[5] Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. Gshard: Scaling giant models with conditional computation and automatic sharding. *arXiv preprint arXiv:2006.16668*, 2020.

[6] Trevor Gale, Deepak Narayanan, Cliff Young, and Matei Zaharia. Megablocks: Efficient sparse training with mixture-of-experts. *Proceedings of Machine Learning and Systems*, 5:288–304, 2023.

[7] Juechu Dong, Boyuan Feng, Driss Guessous, Yanbo Liang, and Horace He. Flex attention: A programming model for generating optimized attention kernels, 2024. URL `https://arxiv.org/abs/2412.05496`.

[8] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness, 2022. URL `https://arxiv.org/abs/2205.14135`.

[9] Tri Dao. Flashattention-2: Faster attention with better parallelism and work partitioning. *arXiv preprint arXiv:2307.08691*, 2023.

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

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

[12] Mor Geva, Roei Schuster, Jonathan Berant, and Omer Levy. Transformer feed-forward layers are key-value memories, 2021. URL `https://arxiv.org/abs/2012.14913`.

[13] Imanol Schlag, Kazuki Irie, and Jürgen Schmidhuber. Linear transformers are secretly fast weight programmers. In *International conference on machine learning*, pages 9355–9366. PMLR, 2021.

[14] Mor Geva, Roei Schuster, Jonathan Berant, and Omer Levy. Transformer feed-forward layers are key-value memories. *arXiv preprint arXiv:2012.14913*, 2020.

[15] Haiyang Wang, Yue Fan, Muhammad Ferjad Naeem, Yongqin Xian, Jan Eric Lenssen, Liwei Wang, Federico Tombari, and Bernt Schiele. Tokenformer: Rethinking transformer scaling with tokenized model parameters. *arXiv preprint arXiv:2410.23168*, 2024.

[16] 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, 2024. URL `https://arxiv.org/abs/2406.17557`.

[17] Frank Rosenbaltt. The perceptron–a perciving and recognizing automation. *Cornell Aeronautical Laboratory*, 1957.

[18] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017. URL `https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf`.

[19] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate, 2016. URL `https://arxiv.org/abs/1409.0473`.

[20] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[21] Robert A Jacobs, Michael I Jordan, Steven J Nowlan, and Geoffrey E Hinton. Adaptive mixtures of local experts. *Neural computation*, 3(1):79–87, 1991.

[22] Michael I Jordan and Robert A Jacobs. Hierarchical mixtures of experts and the em algorithm. *Neural computation*, 6(2):181–214, 1994.

[23] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *arXiv preprint arXiv:1701.06538*, 2017.

[24] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. Palm: Scaling language modeling with pathways. *Journal of Machine Learning Research*, 24(240): 1–113, 2023.

[25] Rowan Zellers, Ari Holtzman, Yonatan Bisk, Ali Farhadi, and Yejin Choi. Hellaswag: Can a machine really finish your sentence? *arXiv preprint arXiv:1905.07830*, 2019.

[26] Yonatan Bisk, Rowan Zellers, Ronan Le Bras, Jianfeng Gao, and Yejin Choi. Piqa: Reasoning about physical commonsense in natural language. In *Thirty-Fourth AAAI Conference on Artificial Intelligence*, 2020.

[27] Denis Paperno, Germán Kruszewski, Angeliki Lazaridou, Quan Ngoc Pham, Raffaella Bernardi, Sandro Pezzelle, Marco Baroni, Gemma Boleda, and Raquel Fernández. The lambada dataset: Word prediction requiring a broad discourse context. *arXiv preprint arXiv:1606.06031*, 2016.

[28] 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*, abs/1803.05457, 2018.

[29] Xun Wu, Shaohan Huang, Wenhui Wang, and Furu Wei. Multi-head mixture-of-experts, 2024. URL `https://arxiv.org/abs/2404.15045`.

[30] Vincent-Pierre Berges, Barlas Oğuz, Daniel Haziza, Wen-tau Yih, Luke Zettlemoyer, and Gargi Ghosh. Memory layers at scale. *arXiv preprint arXiv:2412.09764*, 2024.

[31] Sainbayar Sukhbaatar, Edouard Grave, Guillaume Lample, Herve Jegou, and Armand Joulin. Augmenting self-attention with persistent memory. *arXiv preprint arXiv:1907.01470*, 2019.

[32] 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, 07 2024. URL `https://zenodo.org/records/12608602`.

A Load Balancing Loss for MoE Training

As discussed near the end of Section 3.1, we define the load balancing loss below.

Definition A.1 (Load Balancing Loss)**.** Let $\mathbf{e}_{count} \in \mathbb{Z}^{N_E}$ be a vector that contains the count of assigned experts, based on the assignment matrix $\mathbf{K}^{'}$. In other words, each element within $\mathbf{e}_{\text{count}}$ epresents the number of tokens assigned to some expert i. Let a vector $\mathbf{p} \in \mathbb{R}^{N_E}$ be the column-wise mean of Softmax$(\mathbf{G})$, where $\mathbf{G} \in \mathbb{R}^{N \times N_E}$ is the logit values given by the router layers. We introduce a scalar $\alpha \in \mathbb{R}$ to be a hyperparameter that controls the weight of the loss defined below. A load balance loss $\mathcal{L}_{balance}$ is defined as

$$\mathcal{L}_{balance} = \alpha N_E \langle \frac{1}{kN} \mathbf{e}_{count}, \mathbf{p} \rangle, \tag{9}$$

where k is the number of activated experts and N is the number of tokens in a single sample. Following standard practice [23], we set α to be 0.01. It is worth noting that even though our router is activated by sigmoid, we still use the softmax function for the load balancing loss.

B Related Works

Mainstream frameworks such as GShard [5] and Tutel [3] leverage expert parallelism to scale up the parameter size and set a capacity limit to each expert. However, (1) researchers without access to many computational nodes benefit less from expert parallelism, (2) the frequent all-reduce operation demands advanced interconnect between GPUs, and (3) token-dropping when exceeding the capacity limit makes these frameworks unsuitable for having a large number of small experts. Our FUSE framework offers significant speed-up without relying on expert parallelism, making it single-GPU-friendly. Also, operating without frequent all-reduce operations allows for communication-free independent computation across multiple GPU nodes during the gradient accumulation steps.

Dropless frameworks such as MegaBlocks [6] use dense-sparse matrix multiplication to resolve the token-dropping problem. While this is similar to our FUSE framework in that they both take advantage of sparsity, it is worth noting that FUSE relies on block sparse attention algorithm, which fuses two sets of dense-dense matrix multiplication in a single kernel, without materializing the intermediate activations. Also, FUSE is based on FlexAttention [7], which can be implemented entirely in PyTorch [11]. This removes the need to write and maintain custom CUDA kernels.

Multi-Head MoE (MH-MoE) has been explored by [29]. MH-MoE [29] splits tokens into subtokens and assigning each subtoken to a unique MLP expert. While this approach improves expert utilization, in order for MH-MoE to maintain the same number of parameters as their vanilla counterpart, the number of hidden neurons of a single expert needs to be multiplied by the number of heads. This in turn increases the tensor size of intermediate activations. FUSE instead takes a vanilla approach, routing for each complete token. The multi-head mechanism is restricted within the expert level (see Figure 1). We leverage multi-head mechanism purely as a software-hardware co-design to conserve the SRAM limit.

Feedforward Layer as Attention Mechanism has been explored in [12, 30, 15], all of which are dense feedforward modules. For memory layers [30], the activation function within attention mechanism is still softmax, while FUSE implements GeLU through the reversal trick (see Section 3.4). For Tokenformer [15], the activation function is GeLU with L2 normalization. While in FUSE we further remove the normalization as part of the reversal trick. FUSE is the first work that interprets and scales up SMoE from the perspective of block sparse attention.

Parallelizing the attention and feedforward modules has been explored before [24], which usually involves implementing a custom kernel. FUSE is the first work that allows for parallel attention and feedforward computation in pure PyTorch code. FUSE also combines self-attention with Sparse MoE in a single kernel. A similar approach that combines self-attention with dense MLP was proposed in [31]. However, they do not support arbitrary activation functions and share the same query matrix for both self-attention component and the feedforward component.

C MoE Definition

Let $\mathbf{X} \in \mathbb{R}^{N \times D_e}$ be the input matrix into the MoE module, $N_E \in \mathbb{R}$ be the number of total experts, and $\mathbf{W}_G \in \mathbb{R}^{N_E \times D_e}$ be the gating weights for the module. Additionally, we introduce a set $E = \{E_i\}_{i=1}^{N_E}$ to be an ordered set of MLP experts. We obtain a matrix $\mathbf{G} = \mathbf{X}\mathbf{W}_G^T \in \mathbb{R}^{N \times N_E}$ as the logits for the given input $\mathbf{X}$ based on the set of experts E.

Let $k \leq N_E$ be the number of active experts in a MoE. From this, we can define a Top-K indexing function $h : \mathbb{R}^{N \times N_E} \to \mathbb{Z}^{N \times k} \times \mathbb{R}^{N \times k}$ that returns two matrices. First, a matrix $\mathbf{K}^{'} \in \mathbb{Z}^{N \times k}$ that defines the top-k expert indices for a given input $\mathbf{X}$. Secondly, a logits matrix $\mathbf{L} \in \mathbb{R}^{N \times k}$ respective to each assigned experts of $\mathbf{X}$. Here, we define another matrix $\mathbf{W}_E = \text{Sigmoid}(\mathbf{L}) \in \mathbb{R}^{N \times k}$ that represents the respective experts weights of $\mathbf{X}$. Let $\mathbf{O} \in \mathbb{R}^{N \times D_e}$ be the output matrix of a MoE, and let $\mathbf{w}_{ij}$ and $\mathbf{k}_{ij}$ be an element at the i-th row and j-th column of $\mathbf{W}_E$ and $\mathbf{K}$, respectively. We define each vector $\mathbf{o}_i \in \mathbb{R}^{D_e}$ within $\mathbf{O}$ as:

$$\mathbf{o}_i = \sum_{j=1}^{k} \mathbf{w}_{ij} E_{\mathbf{k}_{ij}}(\mathbf{X}_i), \quad \mathbf{w}_{ij} \in \mathbb{R}, \quad \mathbf{k}_{ij} \in \mathbb{Z}, \quad E_{\mathbf{k}_{ij}}(\mathbf{X}_i) \in \mathbb{R}^{D_e}. \tag{10}$$

Altogether, we construct a MoE module $f_{\text{MoE}} : \mathbb{R}^{N \times D_e} \to \mathbb{R}^{N \times D_e}$ with inputs $\mathbf{X}$, learnable parameters $\mathbf{W}_G$, and set of learnable MLP experts E that outputs some matrix $\mathbf{O}$.

D Unified Computation, Continued

Continuing from Section 3.5, we apply a Softmax function to $\mathbf{A}$ to get the normalized score matrix.

$$\mathbf{S} = \begin{bmatrix} \mathbf{S}_{\text{attn}} & \mathbf{0} \\ \mathbf{0} & \mathbf{S}^{\sigma}_{\text{ffwd}} \end{bmatrix} \in \mathbb{R}^{2N \times (N+N_E)} \tag{11}$$

where $\mathbf{0}$ represents a matrix of zeroes. Following the structure of the attention mechanism, we can define a combined values matrix $\mathbf{V} \in \mathbb{R}^{(N+N_E) \times D_e}$. To get the output $\mathbf{O}$ of the unified attention mechanism, we simply multiply $\mathbf{S}$ and $\mathbf{V}$ together to get an output matrix $\mathbf{O} = \mathbf{S}\mathbf{V}$. The resulting output will have the following form of

$$\mathbf{O} = \begin{bmatrix} \mathbf{O}_{\text{attn}} \\ \mathbf{O}^{\sigma}_{\text{ffwd}} \end{bmatrix} \in \mathbb{R}^{2N \times D_e} \tag{12}$$

where $\mathbf{O}_{\text{attn}} \in \mathbb{R}^{N \times D_e}$ is the output of the attention mechanism and $\mathbf{O}^{\sigma}_{\text{ffwd}} \in \mathbb{R}^{N \times D_e}$ is the permuted output of the attention-based MoE. To reverse the permutation of the MoE output, the inverse mapping σ^{-1} is applied to get $\mathbf{O}_{\text{ffwd}}$. From here, The output of the unified attention decoder layer can be calculated as

$$\mathbf{Y} = \mathbf{X} + \mathbf{O}_{\text{attn}} + \mathbf{O}_{\text{ffwd}} \in \mathbb{R}^{N \times D_e} \tag{13}$$

E Implementation Details and Hyperparameter Choice

Architecture Table 3 presents the hyperparemeter choice for every model reported in this work. All models are decoder-only Transformers that use rotary positional embeddings and RMS normalization. Bias terms are removed from feedforward layers and normalization layers. The dense baselines **B** (B01–B04) scale up by growing both depth or width. Series **P** (P01–P03) pins the embedding size but stretches the feedforward hidden dimension, allowing us to directly compare against the Sparse Mixture-of-Expert (SMoE) models. The **M** (M01-M04) group implements SMoE design under the FUSE framework, sending each token to one expert with 128 neurons. The **U** (U01-U04) group implements UMoE models, which keeps feedforward module sparse but fuses self-attention and SMoE computation into a single kernel. **G** variants change MoE granularity by activating up to four experts for each token, and **A** runs perform targeted ablations on expert activation functions.

Initialization Following standard practices, we removed all biases and initialized token embeddings and all weights randomly following $\mathcal{N}(0, 0.02)$. Output layers are further divided by the square root of the number of residual connections.

Optimization For optimization, we used Adam with $\beta_1 = 0.9$ and $\beta_2 = 0.999$. We linearly warmed up the learning rate for the first 5.24 billion tokens. The learning rate linearly decays to zero during the last 1,000 training steps. The weight decay strength is 0.1 for all non-bias parameters.

Training We trained all models on four NVIDIA H100 GPUs using a batch size of 0.65 million tokens for 10 billion tokens. All training runs can be completed on a single H100 GPU.

Evaluation At training time, we estimate the FineWebEDU perplexity value on a held-out set of 100 million tokens. Every 100 iterations, we calculate on a random subset of five million tokens and report the perplexity value. After training, we use the language model evaluation harness [32] to perform a comprehensive zero-shot downstream task evaluation.

ID	Model	Layers	d_{model}	$d_{ff, tot}$	$d_{ff, act}$	Act. experts	FF-heads	Attn-heads	Max-LR	Batch	Ctx	Actv.	Fuse?
B01	MLP_o	12	768	3 072	3 072	–	–	6	6×10^{-4}	320	2 048	GeLU	False
B02	MLP_o	24	1 024	4 096	4 096	–	–	6	3×10^{-4}	320	2 048	GeLU	False
B03	MLP_o	24	1 536	6 144	6 144	–	–	6	2.5×10^{-4}	320	2 048	GeLU	False
B04	MLP_o	24	2 048	8 192	8 192	–	–	6	2×10^{-4}	320	2 048	GeLU	False
P01	MLP_p	12	768	16 384	3 072	–	–	6	6×10^{-4}	320	2 048	GeLU	False
P02	MLP_p	12	768	36 864	3 072	–	–	6	6×10^{-4}	320	2 048	GeLU	False
P03	MLP_p	12	768	73 728	3 072	–	–	6	6×10^{-4}	320	2 048	GeLU	False
M01	SMoE	12	768	3 072	128	1	6	6	6×10^{-4}	320	2 048	GeLU	False
M02	SMoE	12	768	16 384	128	1	6	6	6×10^{-4}	320	2 048	GeLU	False
M03	SMoE	12	768	36 864	128	1	6	6	6×10^{-4}	320	2 048	GeLU	False
M04	SMoE	12	768	73 728	128	1	6	6	6×10^{-4}	320	2 048	GeLU	False
U01	UMoE	12	768	3 072	128	1	6	6	6×10^{-4}	320	2 048	GeLU	True
U02	UMoE	12	768	16 384	128	1	6	6	6×10^{-4}	320	2 048	GeLU	True
U03	UMoE	12	768	36 864	128	1	6	6	6×10^{-4}	320	2 048	GeLU	True
U04	UMoE	12	768	73 728	128	1	6	6	6×10^{-4}	320	2 048	GeLU	True
G01	SMoE	12	768	73 728	128	1	6	6	6×10^{-4}	320	2 048	ReLU	False
G02	SMoE	12	768	73 728	256	2	6	6	6×10^{-4}	320	2 048	ReLU	False
G03	SMoE	12	768	73 728	384	4	6	6	6×10^{-4}	320	2 048	ReLU	False
A06	UMoE	12	768	73 728	128	1	6	6	6×10^{-4}	320	2 048	ReLU	True
A08	SMoE	12	768	73 728	128	1	6	6	6×10^{-4}	320	2 048	Softmax	False

Table 3: Hyper-parameter summary of all models. $d_{ff,tot}$ is the total number of hidden neurons in the feedforward module, while $d_{ff,act}$ is the number of hidden neurons actually activated during computation. Act. experts reports the number of activated experts. FF-heads correspond to the number of heads in each individual multi-head expert. The "Fuse?" column indicates whether feedforward and attention operations are fused in a single kernel.

F Pseudocode for the Packing and Unpacking Algorithm

To ensure coalesced access pattern, we permute and organize the input tokens with the packing and unpacking algorithm. This makes the block sparsity mask clustered instead of scattered (See Figure 2). There are three stages to this algorithm.

Stage 1 (Algorithm 1, Preparation) For each expert, the preparation stage computes the minimum number of padding tokens required so that every group associated with an expert has length divisible by the block size. It then applies the padding, constructs a permutation mapping $\mathbf{u}$ that clusters tokens belonging to the same expert, and obtains the inverse mapping $\mathbf{u}^{-1}$.

Stage 2 (Algorithm 2, Packing) During packing, P zero vectors are concatenated to $\mathbf{X}$ to match the padded length. Next, the permutation mapping $\mathbf{u}$ is applied to the sequence of feature vectors $\mathbf{X}$. This stage clusters the features of tokens for each expert together in the desired format.

Stage 3 (Algorithm 3, Unpacking) After the block sparse attention computation (see Section 3.2), the unpacking algorithm restores the original token order using the inverse mapping $\mathbf{u}^{-1}$. It then removes the extra padding rows appended during the packing stage.

Algorithm 1 Preparation

```
Require: ExpertAssignment $\mathbf{a} \in \mathbb{Z}^N$, BlockSize $B$, NumOfExperts $N_E$
Ensure: Mapping $\mathbf{u}$, InverseMapping $\mathbf{u}^{-1}$, PaddingSize $P$
 1: Count $\leftarrow$ bincount($\mathbf{a}, N_E$)                 ▷ Bincount with $N_E$ categories
 2: PaddingNeeded[$i$] $\leftarrow (-$Count[$i$]$) \bmod B \quad \forall i$
 3: $P \leftarrow \sum_i$ PaddingNeeded[$i$]                 ▷ Total amount of padding vectors
 4: $\mathbf{a} \leftarrow$ Concat($\mathbf{a}$, PaddingNeeded)
 5: $\mathbf{u} \leftarrow$ argsort($\mathbf{a}$)
 6: $\mathbf{u}^{-1} \leftarrow [0, 1, \ldots, N+P-1][\mathbf{u}]$                 ▷ Inverse mapping
 7: return $\mathbf{u}$, $\mathbf{u}^{-1}$, $P$
```

Algorithm 2 Packing

```
Require: Input $\mathbf{X} \in \mathbb{R}^{N \times D_e}$, PaddingSize $P$, Mapping $\mathbf{u}$
Ensure: PackedInput $\mathbf{X}_{\text{pack}} \in \mathbb{R}^{(N+P) \times D_e}$
 1: $\mathbf{X}_{\text{pad}} \leftarrow$ Concat($\mathbf{X}, \mathbf{0}^{P \times D_e}$)                 ▷ Append zero paddings
 2: $\mathbf{X}_{\text{pack}} \leftarrow \mathbf{X}_{\text{pad}}[\,\mathbf{u}\,]$                 ▷ Permutation
 3: return $\mathbf{X}_{\text{pack}}$
```

Algorithm 3 Unpacking

```
Require: PackedOutput $\mathbf{Y}_{\text{pack}} \in \mathbb{R}^{(N+P) \times D_e}$, InverseMapping $\mathbf{u}^{-1}$
Ensure: output $\mathbf{Y} \in \mathbb{R}^{N \times D_e}$
 1: $\mathbf{Y}_{\text{perm}} \leftarrow \mathbf{Y}[\,\mathbf{u}^{-1}\,]$                 ▷ Restore order
 2: $\mathbf{Y} \leftarrow \mathbf{Y}_{\text{perm}}[0{:}N]$                 ▷ Drop the padding
 3: return $\mathbf{Y}$
```