MoEBlaze: Breaking the Memory Wall for Efficient MoE Training on Modern GPUs

Jiyuan Zhang [1] **Yining Liu** [1] **Siqi Yan** [1] **Lisen Deng** [1] **Jennifer Cao** [1] **Shuqi Yang** [1] **Min Ni** [1] **Bi Xue** [2] **Shen Li** [1]

Abstract

The pervasive "memory wall" bottleneck is significantly amplified in modern large-scale Mixture-of-Experts (MoE) architectures. MoE's inherent architectural sparsity leads to sparse arithmetic compute and also introduces substantial activation memory overheads—driven by large token routing buffers and the need to materialize and buffer intermediate tensors. This memory pressure limits the maximum batch size and sequence length that can fit on GPUs, and also results in excessive data movements that hinders performance and efficient model scaling. We present MoEBlaze, a memory-efficient MoE training framework that addresses these issues through a co-designed system approach: (i) an end-to-end token dispatch and MoE training method with optimized data structures to eliminate intermediate buffers and activation materializing, and (ii) co-designed kernels with smart activation checkpoint to mitigate memory footprint while simultaneously achieving better performance. We demonstrate that MoEBlaze can achieve over $4\times$ speedups and over 50% memory savings compared to existing MoE frameworks.

1 Introduction

Over the past several decades, processor throughput has advanced much faster than memory bandwidth and latency, creating a persistent "memory wall" that widens the gap between compute and data movement (Wulf & McKee, 1995). In practice, this disparity means that even with ample arithmetic units, end-to-end throughput is often limited by how quickly parameters and activations can be read, written, and exchanged (Williams et al., 2009).

Mixture-of-Experts (MoE) architectures have reshaped large-scale deep learning by enabling trillion-parameter models at manageable training cost through sparse activation (Shazeer et al., 2017; Kaplan et al., 2020; Hoffmann et al., 2022). However, the very sparsity that delivers these gains simultaneously lowers compute density because only a subset of experts is active per token. This architectural sparsity, when combined with the scale of distributed training in Large Language Models (LLM), significantly exacerbates memory pressure in modern systems. As models exceed single-device High-Bandwidth Memory (HBM) capacity, training must be distributed across more GPUs and nodes, increasing pressure on device memory bandwidth and interconnect throughput. With the continuous growth in sequence lengths and batch sizes, performance rapidly becomes bounded by the system's memory and communication subsystems rather than raw FLOPs. In light of this, directly reducing the memory footprint and improving effective bandwidth utilization end-to-end has become critical to break the memory wall for MoE training and achieve efficient model scalings.

While parameter storage often gets the spotlight, activation memory is an equally significant driver of the memory wall during training. In state-of-the-art LLM training at trillion-token scale (Brown et al., 2020; Touvron et al., 2023; Team et al., 2024), the combination of longer sequences, larger batches, and more complex routing mechanisms leads to a dramatic expansion of the memory buffers required to compact, reorder, and stage intermediate tensors. Consequently, these activation buffers consume a significant portion of GPU memory footprint and bandwidth, directly limiting the maximum batch size and sequence length a system can handle, and thereby capping the model's scalability and training efficiency.

To address this system bottleneck, earlier methods relied on heuristics like token dropping or padding to cap and manage activation buffers (Samuel, 1959; Fedus et al., 2022), which, however, often came at the cost of model stability. More recent systems are focused on optimizing computation and communication complexity with regards to sparse expert computations (Gale et al., 2023; Aminabadi et al., 2025). Nevertheless, the auxiliary activation buffers needed for token dispatch and the requirement to pad or materialize intermediate results still contribute a major portion of the overall model memory footprint.

To address these limitations, we present MoEBlaze, a memory-efficient MoE training framework that drastically

[1]Meta [2]Thinking Machine Labs. Correspondence to: Jiyuan Zhang <jiyuanz@meta.com>, Shen Li <shenli@meta.com>.

Proceedings of the 9^{th} MLSys Conference (Industry Track), Bellevue, WA, USA, 2026.

improve MoE training memory efficiency without comprising accuracy, while simultaneously achieving better training throughput. Concretely, we target two principal sources of activation memory bottleneck: (i) token routing, where conventional implementations allocate large auxiliary per-expert buffers to compact and store activations; and (ii) intermediate activation storage amplified by modern non-linearities such as SiLU and SwiGLU (Shazeer, 2020; Ramachandran et al., 2017; Elfwing et al., 2017). MoEBlaze is designed to effectively break through the memory wall and maximize the utility of modern GPU architectures to better throughput. Our contributions are:

- We introduce an efficient end-to-end token dispatch and training method that significantly reduces the intermediate activation buffers for token routing and activation materializing. Our approach avoids both padding and token dropping, reducing memory usage and data movement without sacrificing accuracy while simultaneously achieving better compute efficiency.
- We introduce efficient data structure and algorithm for above memory-efficient compute scheme that can efficiently leverage GPU's massive parallelism and high bandwidth and avoids complex multi-kernel pipelines.
- We co-design training kernels with smart activation checkpoint schemes, which can further mitigate the substantial memory footprint associated with modern complex activation functions while achieving better compute efficiency on GPU.
- Overall our method can achieve over $4\times$ speedups and over 50% compared memory savings to other state-of-the-art MoE training frameworks across various MoE benchmarks.

2 Background and Motivations

In this section, we provide a review of the background of MoE training, along with a identification of the key system bottlenecks that currently limit its performance.

We first define the notations that will be used throughout the rest of the paper. We mainly focus on the token-choice MoE as it has been adopted extensively in production. The MoE computation begins with the token input, which is represented as a vector $\mathbf{x} \in \mathbb{R}^{L\times d}$, where we denote by L the number of routed token instances in a step (e.g., batch size × sequence length), K the number of selected experts per token, E the number of experts, d the model dimension.

2.1 Gating Network and Token Routing

The gating network determines the routing of each input token to the most relevant experts. The network is typically a linear transformation mapping the input dimension d to the number of experts E, thereby generating a score for each expert per token. This is followed by a Top-K selection, where expert-ids corresponding to the highest gating scores are collected for each token. The gating output for input $\mathbf{x}$ is defined as:

$$\text{topk\_experts} = \text{TopK}(\text{softmax}(W_g\mathbf{x}))$$

where $W_g \in \mathbb{R}^{E\times d}$ are the gating network parameters and K is the number of selected experts per token. The result *topk_experts* is the list of selected expert-ids by each token.

Following Top-k selection, tokens must be physically routed to their corresponding expert's execution buffer. In conventional implementations, this routing process requires substantial auxiliary memory and extra processing to compact and store the dispatched tokens, which constitutes a critical memory bottleneck.

Earlier work such as Switch Transformers and GShard (Fedus et al., 2022; Lepikhin et al., 2021) adopts capacity-limited routing (token-dropping) mechanism to manage token dispatch. Tokens are sorted by their gate score and packed into expert e's buffer; any tokens exceeding C are either dropped or routed to a residual path. A typical choice for capacity is:

$$C \approx \gamma \cdot \frac{Bk}{E}.$$

where γ is the user-defined capacity factor. Capacity-limited routing is amenable to system implements due to fixed-size buffers but comes at the cost of reduced model quality.

More recent literature focuses on dropless routing mechanisms (Rajbhandari et al., 2022; He et al., 2021), which generally yields better model quality. This method ensures every token is processed by an expert, allowing for better model quality and eliminating the need for capacity factor tuning. However, since the number of tokens assigned to each expert is variable, the underlying system must efficiently manage dynamic compute and memory needs. Consequently, recent literature primarily focuses on optimizing the computation with these dynamic and varying-length workloads (Gale et al., 2023; Aminabadi et al., 2025).

Nevertheless, a fundamental challenge persists across both token-dropping and dropless routing schemes: current implementations require storing the indices and compacted token data, resulting in memory footprint proportional to $L \times K \times d$. In modern LLM training with longer sequence lengths and higher batch sizes, this leads to a dramatic expansion of the memory buffers.

Example: To illustrate this token dispatch associated activation footprint, we use the example of a real-world MoE model (e.g DeepSeek) for a quantitative study here. For a typical MoE layer in DeepSeek model, it has $L \approx 2$ million

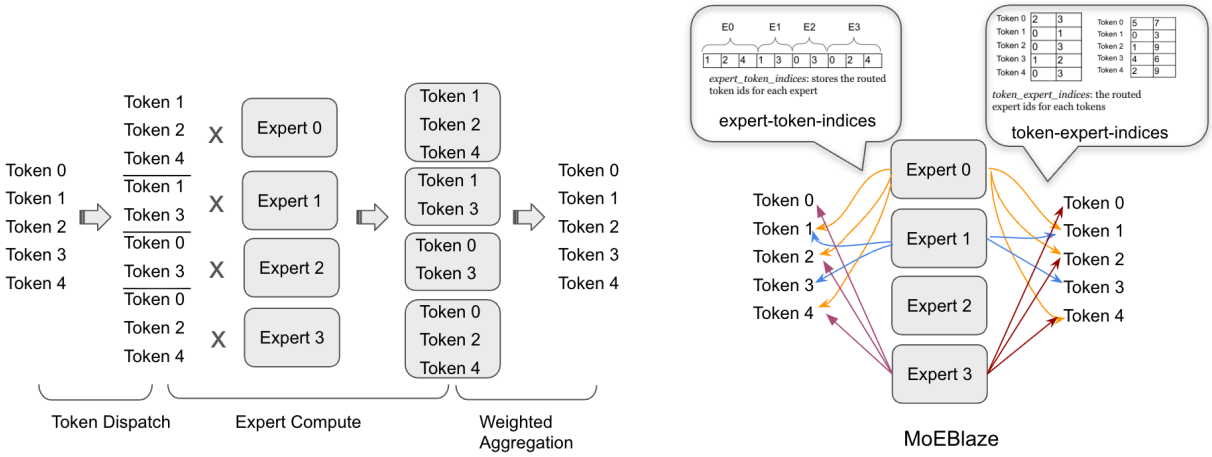

Figure 1. MoE in conventional approach vs. MoEBlaze. Left illustrates the conventional MoE computation, comprising token dispatch, expert computation, and weighted aggregation (details in section 2). Right presents the proposed MoEBlaze framework, which introduces memory-efficient token-routing and expert computation (details in section 3).

tokens, active experts $K = 4$, model dimension $d = 6144$, and using a 2 bytes per element (bfloat16) for the routed token buffer, the memory footprint is:

$$\text{Mem}_{routing} = L \times d \times k \times 2bytes \approx 94\text{GB}$$

We can see a single MoE layer can consume almost a hundred gigabytes of memory for one routing buffer alone.

2.2 Expert Feed-Forward Networks (FFNs)

Following the token dispatch is the Feed-Forward Networks (FFN) computation across experts. Each FFN computation is typically realized as a two-layer Multi-Layer Perceptron (MLP). The first layer projects the input from dimension d to a higher-dimensional hidden space h, and the second layer projects back to the output dimension d' (assuming $d' = d$). The total memory required for the parameters across all E experts is $\mathcal{O}(E \times (h \times d + d \times h))$, but the conditional computation paradigm ensures that only k experts are active for each token, maintaining a low computational cost per forward pass. The FFN computation within expert E_i for a given input $\mathbf{x}$ is defined as:

$$E_i(\mathbf{x}) = W_{2,i} \cdot \sigma(W_{1,i}\mathbf{x})$$

where $W_{1,i} \in \mathbb{R}^{h \times d}$ and $W_{2,i} \in \mathbb{R}^{d \times h}$, and σ is the non-linear activation function (e.g., ReLU, GELU, SwiGLU).

The second principal memory bottleneck stems from intermediate activation storage during the FFN computation. For an active expert, the first linear transformation $W_{1,i}\mathbf{x}$ generates an intermediate activation of size $L_i \times h$, where L_i is the number of tokens routed to expert i. While the number of active experts is small, the aggregated memory for these activations across all E experts is $\mathcal{O}(L \times h)$ during the forward pass and can be much higher during backpropagation due to the need to store intermediate values for gradient calculation. The choice of activation function (e.g., SwiGLU) will further exacerbate the memory pressure.

Example: We use deepseek's configuration as an example to illustrate the significance of the activation footprint created by FFN computation. We have $L \approx 2$ million, FFN hidden dimension $d = 24576$, and using 2 bytes per element (bfloat16), we can get the activation footprint for the intermediate :

$$\text{Mem}_{act} = 2L \times h \approx 98\text{GB}$$

2.3 Output Aggregation

The final stage is output aggregation, where the outputs of the selected experts are combined using a weighted summation to produce the final output for each token. The weights are derived from the gating network's scores. The MoE output $\mathbf{y}$ for an input token $\mathbf{x}$ is:

$$\mathbf{y} = \sum_{i=1}^{E} g_i(\mathbf{x}) \cdot E_i(\mathbf{x})$$

where $g_i(\mathbf{x})$ is the gating score for expert i, and only the top-k experts have non-zero scores. The memory required for the aggregated outputs is $\mathcal{O}(L \times d)$. Computationally, this involves $\mathcal{O}(L \times k \times d)$ operations, which is generally efficient given the sparsity ($k \ll E$).

3 Memory-Efficient Token Routing and Training Algorithm

As detailed in Section 2.1, in token-choice MoE, a gating network assigns each input token to one or more experts. To

facilitate efficient token indexing and organization during expert computation, conventional systems compact these tokens into per-expert buffers. This compaction step, followed by the execution of per-expert Multi-Layer Perceptron (MLP) blocks, creates intermediate results kept at the compacted token length before being summed and reduced to the original token length at the output. Crucially, this separation and intermediate storage introduces significant activation buffers throughout the entire MoE training. In this section, we present our memory-efficient routing and expert computation algorithm that substantially reduces the auxiliary activation footprint while also allowing for efficient MoE training.

Given the input as an activation tensor of shape (L, d), the core idea of our algorithm is to leverage auxiliary index lists, generated during the token dispatch step, to track routing decisions and perform on-the-fly token accessing and result reduction throughout the Mixture-of-Experts (MoE) computation. Concretely, our fused kernel operates as follows: 1. it consumes the gating decisions and builds the expert-token index lists and other associated indexing structures, 2. it performs the expert MLP computations using on-the-fly gathers from the original, unpermuted activation tensor, guided by the expert-token index list, and 3. The expert summation then uses the token-expert index list to directly sum and reduce the MLP results into the final output tensor. By directly accessing the input and storing only the final result, we eliminate many intermediate activations that are typically required for materialized token routing in other papers. The token-expert index list, which only stores token and expert IDs, is extremely lightweight. Moreover, this approach allows us to tightly fuse the token/expert indexing with computation, opening possibilities for overlapped memory access and computations. This is particularly advantageous on modern hardware like the latest H100 GPUs, achieving better resource utilization and faster speed.

Below, we detail the forward and backward passes of our proposed method. Data structures and the methods for efficiently building them will be explained in next section.

3.1 Forward Pass

Token Dispatch: In the token dispatch step, we do not create dedicated buffer for routed tokens. Instead we generate several lightweight indexing data structures based on the gating scores produced in the preceding gating stage. These structures include: the per-expert token list, which tracks the token-IDs assigned to each expert; and the per-token expert list, which stores the expert-IDs chosen for each token. No memory is allocated or preserved for materialized routed token activations at this stage.

Expert Computation: We perform the expert computation MLPs with on-the-fly gathers from the original unpermuted activation tensor utilizing the indices recorded in the per-expert token list. To maximize memory efficiency, only the intermediate result between the two back-to-back MLPs (i.e., the output of the first MLP) is buffered for the backward pass.

Output Aggregation: The final results from the experts are aggregated to produce the final (L, d) output. As we do not store the activation buffer for the materialized token dispatch result, this summation is tightly fused with the 2nd MLP computation and we directly leverage the per-token expert list to perform on-the-fly reduction into output tensor.

3.2 Backward Pass

The backward pass takes the gradient of the (L, d) tokens and propagates it back through the inverse of the forward steps. The conventional backward process for expert summation relies on the routed token activation buffer to perform an "expansion" or materialization of the (L, d) gradients to the $(L \times k, d)$ "routed gradient tokens" before backpropagation through the MLP experts. However, our proposed approach avoids this intermediate expansion step by using the same reverse mapping indices.

1. **Expert Summation Backward:** Using the token-mapping structure derived from the dispatch metadata, the (L, d) gradient tensor is mapped back to the $(L \times k, d)$ routed gradient tokens. This is done via an efficient operation that 'scatters' the output gradient to the corresponding locations in the materialized intermediate MLP result tensor.

2. **Expert Computation Backward:** Next, the gradients flow backward through the MLPs. The previously checkpointed intermediate result between the two back-to-back MLPs will be used here when computing the weight gradients.

3. **Token Gradient Accumulation:** Finally, the gradients with respect to the input tokens are accumulated from all experts. This step sums the contributions from the k experts each token was routed to, producing the final (L, d) gradient tensor for the input activations. As we do not have the activation storage the materialized routed token result, we also leverage the token index data structure to perform on-the-fly reductions.

4 Efficient and Parallelizable Dispatch and Data Structures

4.1 Data Structures

We define the key data structures needed for the memory efficient MoE training algorithm we mentioned above.

- expert_token_indices: A compact tensor storing the indices of tokens assigned to each expert, concatenated across all experts. In the token-choice MoE training, each token chooses k experts, thus the *expert_token_indices* has size $L \times k$. This list is fundamental for the experts to retrieve their designated input tokens.
- expert_token_offsets: An array of length $E + 1$ storing the exclusive prefix sums of token counts per expert. For expert i, the indices of its assigned tokens reside from expert_token_offsets$[i]$ up to expert_token_offsets$[i{+}1] - 1$.
- token_expert_indices: token_expert_indices is basically the inverse mapping of expert_token_indices. It stores the routed expert-ids for each token which are ordered by the token IDs. Its shape is also $L \times k$. This list is need for coalesced indexing into the intermediate materialized results (e.g., between two back-to-back MLPs) when processing tokens per expert.
- token_index_map: A $L \times k$ compact tensor that stores the routed token positions in the expert_token_indices list. It is logically grouped by the original token ID $i \in L$, allowing a token to efficiently find and gather its k expert outputs from the intermediate buffer for the final combination step.

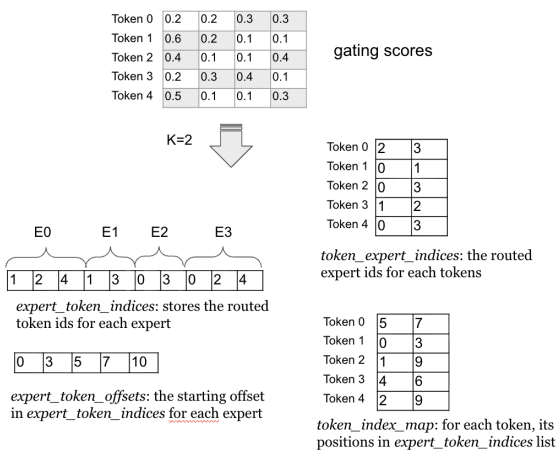

Figure 2. Data structures for the memory-efficient MoE training.

Example. Figure 2 demonstrates the data structure for an example of L=6 tokens, E=4 experts, and $k = 2$ activated experts. From the gating score matrix, we obtain per-token's assignment as:

Token 0: expert$\{2, 3\}$; Token 1: expert$\{0, 1\}$; Token 2: expert$\{0, 3\}$; Token 3: expert$\{1, 2\}$; Token 4: expert$\{0, 3\}$. Concatenate the tokens' assignment together, we will get the token_expert_indices as:

$$\text{token\_expert\_indices} = [2, 3, 0, 1, 1, 2, 0, 3],$$

Similarly we can get for each expert the routed tokens: Expert 0: token$\{1, 2, 4\}$; Expert 1: token$\{1, 3\}$; Expert 2: token$\{0, 3\}$; Expert 3: token$\{0, 2, 4\}$. Concatenate them together, we get the expert_token_indices and expert_token_offsets

$$\text{expert\_token\_indices} = [1, 2, 4, 1, 3, 0, 3, 0, 2, 4],$$
$$\text{expert\_token\_offsets} = [0, 3, 5, 7, 10]$$

The token_index_map stores the positions of each token within the concatenated experts' token list. For example, token_index_map$[0] = \{5, 7\}$ as token_0 is routed to 2 experts ($k = 2$) and placed in the 5th and 7th position of the expert_token_indices.

4.2 Efficient Dispatch Data Structure Construction

We now detail the methods to efficiently construct the aforementioned data structures. The construction process presents a challenge: the inherent design of the expert-centric data structures requires a many-to-one mapping where multiple tokens are assigned to the same expert. Utilizing a naive approach would result in severe thread-level write contention on the GPU architecture, thereby compromising performance.

One solution is to rely on a sorting-based approach for building the token dispatch. This method flattens all tokens' top-k choice results (topk_experts) into a 1D array of length Lk containing (expert_id, token_id) tuples. The array is then globally sorted by expert_id to group tokens, followed by index recovery to reconstruct token order and compute per-expert ranges.

This sorting procedure, while conceptually simple, introduces severe performance bottlenecks at scale. Sorting is implemented as multi-pass radix sort on GPUs, which requires several global-memory passes proportional to key width, forcing frequent global-memory passes and moving $O(Lk)$ data multiple times. This results in a actual high complexity and poor resource utilization on GPUs. Furthermore, this global ordering step limits fine-grained parallelism, forces a multi-kernel dispatch pipeline (multi-pass sorts, segmented scans, index recoveries etc) with high

kernel launch latencies. These limitations motivate a more efficient, GPU-friendly approach.

To this end, we introduce an efficient method that replaces complex global sortings and organizations with parallelizable builds upon local index construction that map well to GPUs. The method is a simple 3-step process with each step designed to be atomic-free and parallelized on GPU which can minimize expensive global-memory passes and avoids complex multi-kernel pipelines. Below we will go over the details of the three steps.

Build Dense Token-Expert Map In the first step, we construct a dense bitmap denoted as $\mathrm{dense\_token\_map}$ to encode the top-k token-to-expert routing. For each token i, we consider its top-k assigned experts $\{e_{i,0}, \ldots, e_{i,k-1}\}$. For each gate slot, we set $\mathrm{dense\_token\_map}[i, e_{i,k}]$ to i. All other entries remain unset.

The construction of the encoding map is highly parallelizable on the GPU. We initiate the process by allocating an $\mathbf{L} \times \mathbf{E}$ dense map and launching the kernel over the CTA grid. The parallelism is managed by assigning each warp a disjoint tile of token rows (i) from which it loads the $\mathbf{top\_k}$ expert results. Each (i, e) pair is written out at most once because expert IDs per token are unique; This guarantees no intra-warp collisions.

While a dense $L \times E$ map might initially appear memory-intensive at scale, the footprint is highly manageable in practice. For the motivating large-scale regime (e.g., $L \approx$ 2M, $E = 100$), using 1 byte per element requires only $\sim$200MB. Furthermore, this allocation is strictly temporary; it is discarded immediately after the dispatch construction is complete and contributes nothing to the training memory lifecycle.

Compute Expert Lengths Leveraging the constructed $\mathrm{dense\_token\_map}$, the next step is to efficiently compute the lengths and offsets for the sparse token-ID list for each expert. We launch a custom kernel with the CTA grid mapped across the columns (experts) of $\mathrm{dense\_token\_map}$. Each CTA is dedicated to a single expert e_i and counts the non-zero entries (token-to-expert assignments) within that column. The use of warp-level reductions aggregates the row-wise sums within the CTA, producing the $\mathrm{expert\_lengths}$ array. The value $\mathrm{expert\_lengths}[e_i]$ represents the final number of tokens routed to expert e_i. Following the length computation, the $\mathrm{expert\_offsets}$ are derived by applying a prefix sum over the $\mathrm{expert\_lengths}$ array outside the initial counting kernel.

Route Indices to Gates This 3rd step involves generating the per-expert token id list $\mathrm{expert\_token\_indices}$, which serves as the input for subsequent MLP computations. To achieve a compact, per-expert concatenation of indices in a contention-free manner on the GPU, we employ a two-phase process centered around generating a location map. This map specifies the final destination position-ID for every non-zero entry in the $\mathrm{dense\_token\_map}$ within the $\mathrm{expert\_token\_indices}$ list. Once the location map is built, a simple parallel kernel reads elements from $\mathrm{dense\_token\_map}$ and writes them directly to their calculated, corresponding positions in $\mathrm{expert\_token\_indices}$, guaranteeing full parallelism without atomics.

The construction of the location map can be challenging. We utilize a two-step strategy to ensure its atomic-free construction: (i). tile-level scan: We launch one CTA per expert. Threads within the same CTA process contiguous tokens assigned to that expert in $\mathrm{dense\_token\_map}$. They first compute the tile-level counts within shared memory, followed by an exclusive scan operation (prefix sum) performed locally inside the CTA. (ii). The resulting CTA-local exclusive scan counts then add with the expert's pre-computed global $\mathrm{expert\_offsets}$. This addition yields the correct, final position-ID in the concatenated indices array.

5 Training–Kernel Co-Design for End-to-End Efficiency

This section details our approach to jointly optimize the Mixture-of-Experts training kernels and smart activation checkpointing method to address the memory issues associated with some advanced activation methods.

5.1 SwiGLU MoE and the Memory Bottleneck

Modern MoE training has increasingly adopted advanced non-linear activations such as *SiLU* and *SwiGLU* in place of ReLU/GELU. Prior work shows these activations provide smoother nonlinearity, which can improve optimization stability and lead to better empirical accuracy on large-scale language tasks. While numerically favorable, these activations introduce more complex compute and larger memory footprint during training. We take the SwiGLU activation as an example. The SwiGLU activation is defined as:

$$\mathrm{SwiGLU}(\mathbf{x}; W_1, W_2) \;=\; \mathrm{SiLU}(\mathbf{x}W_1) \;\cdot\; (\mathbf{x}W_2),$$

where $\mathrm{SiLU}(u) = u \cdot \sigma(u)$ and σ is the *sigmoid* operation. For an MoE layer with E experts, each implementing a SwiGLU Feed-Forward Network (FFN), a routed batch of tokens $x \in \mathbb{R}^{L \times d}$ for a single expert induces two projections: $a = xW_1 \in \mathbb{R}^{L \times h}$ and $b = xW_2 \in \mathbb{R}^{L \times h}$. This is followed by the element-wise operations $\mathrm{SiLU}(a)$ and the final product $\mathrm{SiLU}(a) \odot b$.

5.2 Activation Checkpoint and Kernel Codesign

In conventional kernels, the forward pipeline necessitates materializing multiple intermediates in order to accommodate the backward computation (e.g., in the SwiGLU example, it includes the two GEMM outputs a and b, the sigmoid $\sigma(a)$, SiLU(a), and the final product). These intermediate results are written to and subsequently read from global memory, which incurs non-negligible overheads. As models and batch sizes scale, this incurs significant memory traffic and storage costs, which quickly becomes a non-negligible bottleneck.

To mitigate the observed memory pressure, we present a joint optimization of the MoE training flow and its underlying GPU kernels that reduces the activation memory footprint and memory traffic without sacrificing performance. Our optimization is based on below observations:

- Computation of activation functions is generally memory bandwidth bound on modern GPUs. For example, on an NVIDIA H100 GPU, saturating vector compute units requires an arithmetic intensity of roughly 40 FLOPs/byte. In contrast, point-wise operations like SiLU in `bfloat16` (reading 2 bytes, writing 2 bytes, and performing $\sim$5 operations) yield an arithmetic intensity of just $\sim$1.25 FLOPs/byte. Even at a peak HBM3 bandwidth of 3.35 TB/s, the achievable compute throughput is bounded under 5 TFLOPS—less than 1% of the device's peak capacity. Consequently, in LLM training where sequence lengths far exceed embedding dimensions ($L \gg d$), materializing these intermediate activations operates well below peak compute utilization and remains strictly memory-bound.
- While activation computation is computationally light, its memory footprint is surprisingly significant. This is particularly true for complex, modern activation functions, which requires materialization and saving many intermediate, point-wise results for the backward pass. The resulting memory allocation is substantial, scaling linearly with the batch size, sequence length, and FFN dimensions. In the context of today's trillion-token training environments, the memory required to store these activations can be prohibitive.

Based on this observations, we propose the joint activation checkpoint and kernel fusion approach. Our approach fuses the two first-layer projections in SwiGLU with the activation epilogue, and applies activation checkpoint in the specialized path to "break the memory wall" arising from intermediate activation storage.

To reduce activation traffic and kernel launch overhead, we fuse both first-layer projections and the SwiGLU epilogue into a single kernel. The kernel consumes non-materialized routed tokens, loads the input x only once, streams it through both (W_1, W_2) GEMMs simultaneously, computes SiLU(a) in register/shared memory, and immediately performs the multiplication with b, writing only the final output to global memory.

This "epilogue fusion" eliminates global reads for elementwise operations, effectively moving computation from the memory-bound domain to the compute-bound domain where possible. It also halves the input reads of x compared to separate kernels for each projection.

During backward, fusing the two first-layer projections implies that gradients w.r.t. the shared input x from both paths must be aggregated. Rather than allocating two separate activation buffers and stitching them, our implementation computes the two branches' activation derivatives in a fused fashion and aggregates gradients in-place via tiled reductions—completely eliminating temporary global buffers.

On top of the fusion, we further applied the activation checkpoint strategy – where we skip saving the SwiGLU intermediate result (SiLU) during forward. Instead, we adopt a recomputation strategy during the backward pass, leveraging the fact that the SiLU function is computationally inexpensive (e.g elementwise operation), and are heavily memory bandwidth bounded on modern GPUs.

5.3 Putting It Together: E2E Training on Swiglu MoE

Algorithm 1 summarizes the end-to-end training process for an MoE model utilizing the SwiGLU activation function. The pseudo-code specifically demonstrates the integration of activation checkpoint and kernel fusion detailed in section 5. While the low-level implementation of the fused kernels is omitted for brevity, the high-level methodology is derived from the memory-efficient token dispatch explained in section 3.

6 Experiments

In this section, we benchmark MoEBlaze against the current state-of-the-art sparse training system, Megablocks (Gale et al., 2023), demonstrating significant improvements in both training speed and memory efficiency across a range of representative Mixture-of-Experts (MoE) configurations.

6.1 Experiment Setups

We conducted all experiments on a single NVIDIA H100 Tensor Core GPU. The software stack utilizes PyTorch 2.0.1 and CUDA 12.1. We measure end-to-end training time for a single MoE layer, focusing on the Sparse-to-Sparse computation phase. We evaluate performance with two different activation functions: ReLU (Rectified Linear Unit) and SwiGLU (Swish-Gated Linear Unit).

Algorithm 1 Fused SwiGLU MoE Training

Input: Input Tokens $X \in \mathbb{R}^{L \times d}$, Projection Weights $W_1, W_2 \in \mathbb{R}^{d \times h}, W_3 \in \mathbb{R}^{h \times d}$,
Output: Output $Y_{\text{out}} \in \mathbb{R}^{L \times d}$, Gradients $\nabla W_1, \nabla W_2, \nabla W_3, \nabla Z$

// Forward module for Swiglu MoE training
Procedure FusedForward(X, W_1, W_2, W_3)
// Load input tokens once
Load X
// 1st MLP projection:
// Compute A and B;
// SiLU(A) and Y_{swi} computed in-kernel
// SiLU(A) is transient
$(A, B), Y_{\text{swi}} \leftarrow \text{Fused\_SwiGLU}(Z, W_1, W_2)$
$Y_{\text{out}} \leftarrow Y_{\text{swi}} W_3$
Store A, B, Y_{swi}
// Store activations and SwiGLU output for backward
Return Y_{out}

Procedure FusedBackward
$(Y_{\text{out}}, \nabla Y_{\text{out}}, W_1, W_2, W_3, A, B, Y_{\text{swi}})$
// Gradient for final projection
$\nabla W_3 \leftarrow Y_{\text{swi}}^T \nabla Y_{\text{out}}$
// Backpropagate gradient to SwiGLU output
$\nabla Y_{\text{swi}} \leftarrow \nabla Y_{\text{out}} W_3^T$
// Load stored activations
Load A, B
// Recomputes SiLU(A) to save memory
$S_{\text{recomp}} \leftarrow \text{SiLU}(A)$
// Derivative w.r.t A
$\nabla A \leftarrow \nabla Y_{\text{swi}} \odot B \odot \nabla \text{SiLU}(A)$
// Derivative w.r.t B
$\nabla B \leftarrow \nabla Y_{\text{swi}} \odot S_{\text{recomp}}$
$\nabla W_1, \nabla W_2 \leftarrow \text{FusedBwdW}(X, \nabla A, \nabla B)$
$\nabla X \leftarrow \text{FusedBwdX}(\nabla A W_1^T, \nabla B W_2^T)$
Return $\nabla W_1, \nabla W_2, \nabla W_3, \nabla Z$

We selected a set of seven representative MoE configurations that explore varied dimensions for the input hidden size (d), the number of experts (E), the top-k tokens routed to experts, and common training parameters (sequence length L and batch size B). The specific configurations, which mimic common settings in large language models, are detailed in Table 1.

6.2 Baselines and Metrics

Our primary baseline for comparison is Megablocks, a system that optimizes MoE training through custom kernels and efficient token dispatch, serving as the industry standard for high-performance sparse training.

We evaluate performance using two key metrics: (1) Training Speed: measured as the speedup factor of MoEBlaze relative to Megablocks in an end-to-end single training pass. The training time includes both forward and backward passes, but we exclude optimizer updates as optimizer is irrelavant to both approach designs. Higher values indicate better performance. (2) Activation Memory Consumption: measured as the total memory allocated to save the intermediate activation tensors for given inputs. To measure the activation memory, we utilize PyTorch's saved tensor hooks to trace and calculate the exact activation space allocated during model training with the given input configuration.

Table 1. MoE configurations used in experiments. The FFN hidden dim is set to be four times the input dimension (ffn_hidden_size $=$ $4 \times$ input_d).

	INPUT_D	EXPERTS #	K	BATCH	SEQ_LEN
CONF1	512	4	1	32	2048
CONF2	1024	8	2	32	2048
CONF3	1024	16	4	32	2048
CONF4	512	16	4	32	1024
CONF5	1024	16	4	16	1024
CONF6	2048	8	4	16	512

6.3 Memory Efficiency in MoE Training using SiLU

As shown in Figure 3, MoEBlaze consistently and significantly reduces activation memory consumption compared to the Megablocks baseline across all tested configurations.

The memory savings achieved by MoEBlaze are especially significant in configurations characterized by large input dimensions and high expert counts, such as conf4. Specifically, MoEBlaze requires only $6,100$ MB of memory, achieving a nearly $3.6\times$ reduction compared to the $22,000$ MB consumed by Megablocks. For smaller configurations (e.g., conf1), the activation memory saving is less pronounced, which is expected since the savings scale proportionally with the sequence length L and the number of activated experts k, both of which are small in conf1 ($k = 1$). This substantial reduction in peak activation memory is a direct outcome of two core system innovations: (1) a more memory-efficient token dispatch mechanism that minimizes intermediate buffer allocations, and (2) the adoption of smart recomputation within our custom activation checkpoint scheme.

6.4 Training Speed in SiLU-based MoEs

Figure 4 illustrates the training speedup of MoEBlaze over Megablocks over the given configurations. MoEBlaze achieves notable performance gains, showing a speedup factor of $1.4\times$ to $3.7\times$.

The maximum speedup is achieved at conf4 ($D_{input} =$

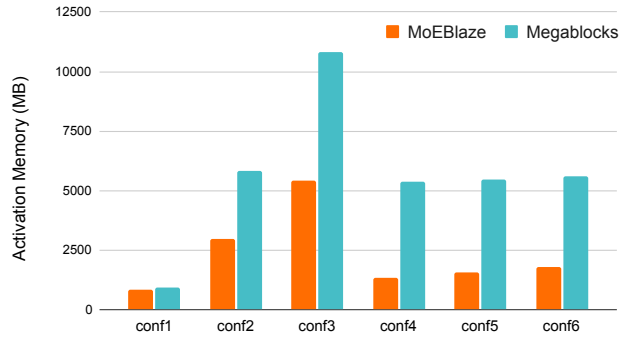

Figure 3. Activation memory footprint comparison between MoEBlaze and Megablocks across the set of MoE configs defined in Table 1 using SiLU activation function.

2048, $E = 16$, $L = 1024$, $B = 32$), demonstrating that MoEBlaze scales particularly well with larger model dimensions. This training speeds are attributed to three factors: (1) our highly optimized token dispatch implementation, which reduces the latency overheads associated with expensive token dispatch and permute operations; (2) the efficient data dispatch construction kernels, which is very light-weight and runs rapidly on GPUs, avoiding the expensive multiple-passes kernel in other sorting-based approaches and greatly eliminating the CPU-side bottlenecks. (3) the fused kernel for the batched-GEMM computations that effectively leverages H100's latest hardware acceleration features such as warp-group matrix multiplication, tensor memory accelerator, etc.

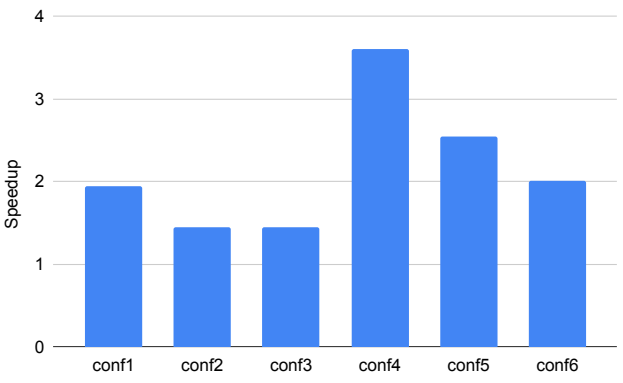

Figure 4. Speedups of MoEBlaze w.r.t to Megablocks on the set of configurations in Table 1 using SiLU as the activation function.

6.5 Memory Efficiency in MoE Training with SwiGLU

The SwiGLU activation function inherently requires higher memory usage due to the additional gating and element-wise multiplication operations. Figure 5 shows the memory consumption comparison under the SwiGLU setting. MoEBlaze maintains a substantial memory advantage over Megablocks, with peak activation memory often less than half of the baseline's usage. For instance, in configuration conf3, Megablocks requires over $40,000$ MB, while MoEBlaze is contained to approximately $10,000$ MB. This consistent $4\times$ reduction in memory pressure confirms that our memory-efficient dispatch and smart recomputation schemes are highly effective even for more complex activation functions.

6.6 Training Speed in SwiGLU-based MoEs

Figure 4 presents the speedup of MoEBlaze relative to Megablocks when using the SwiGLU activation. Compared to the ReLU results, the speedup factors are generally higher and more consistent, ranging from $2\times$ to $6.2\times$. The increased relative speed is a result of two factors: (1) The more complex computation in SwiGLU exposes greater opportunities for MoEBlaze's highly fused kernels to outperform the baseline; and (2) the memory-bandwidth savings from our activation optimization are more critical in the SwiGLU case, where intermediate activation sizes are larger and more compound, thereby reducing the excessive global memory accesses through smart kernel fusion and recomputation allows MoEBlaze to execute the whole kernel faster.

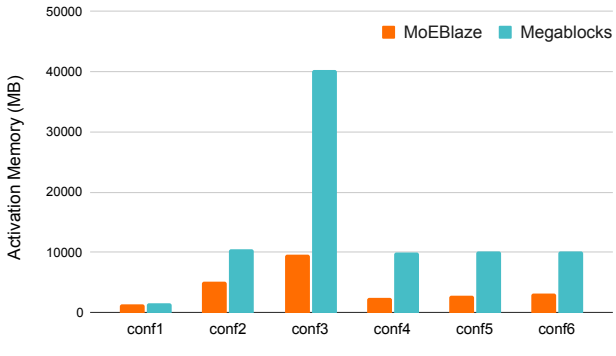

Figure 5. Activation memory footprint comparison between MoEBlaze and Megablocks across the set of MoE configs defined in Table 1 using SwiGLU activation function.

7 Extended Evaluation

To further validate the performance, scalability, and practical impact of MoEBlaze, we present extended evaluations

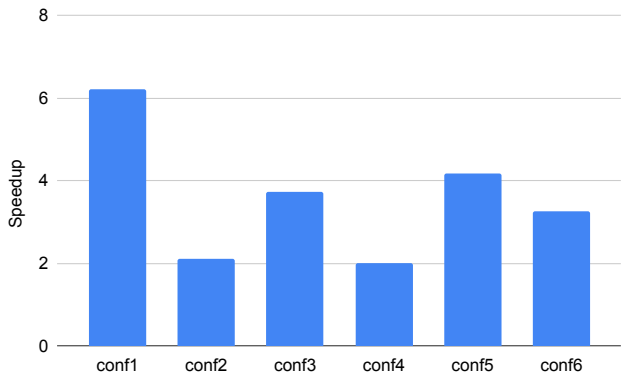

Figure 6. Speedups of MoEBlaze w.r.t to Megablocks on the set of configurations in Table 1 using SwiGLU as the activation function.

covering large-scale motivating regimes, production-grade model configurations, and comprehensive ablation studies.

7.1 Large-Scale Evaluation

A primary motivation for MoEBlaze is to address the severe memory bottlenecks encountered in extreme-scale MoE training. Table 2 demonstrates MoEBlaze's performance at configurations matching this motivating regime (up to $L =$ 2M tokens and $E = 128$ experts). MoEBlaze maintains consistent speedups and massive memory savings compared to Megablocks. Furthermore, as sequence lengths scale beyond 512k tokens, Megablocks reliably fails due to Out-Of-Memory (OOM) errors, whereas MoEBlaze operates stably up to 2 million tokens.

Table 2. Execution time and memory footprint of MoEBlaze and Megablocks across different input sizes ($K = 4$, $E = 128$, $d = d' = 256$). Megablocks runs out-of-memory (OOM) beyond 512k tokens, while MoEBlaze scales to 2048k tokens.

		Time (ms)		Speedup	Mem. (MB)		Save
Act.	L	Blaze	Block		Blaze	Block	(%)
SwiGLU	8k	1.0	1.8	1.9	155	495	68.6
SwiGLU	32k	2.3	5.9	2.6	477	1931	75.3
SwiGLU	128k	7.7	22.5	2.9	1766	7675	77.0
SwiGLU	512k	28.7	OOM	–	6918	OOM	–
SwiGLU	2m	115.2	OOM	–	27528	OOM	–
SiLU	8k	0.7	1.4	1.9	108	303	64.5
SiLU	32k	1.8	4.4	2.4	334	1163	71.3
SiLU	128k	6.3	16.1	2.4	1238	4603	73.1
SiLU	512k	24.0	OOM	–	4854	OOM	–
SiLU	2m	95.3	OOM	–	19320	OOM	–

7.2 Real-World Model Configurations

To confirm that these benefits translate to modern architectural trends, we evaluate our proposed hash-based dispatch mechanism against a standard sorting-based baseline across a variety of production-scale MoE model settings (e.g., Qwen3, DeepSeek V3, DBRX). As shown in Table 3, for million-scale token counts, our dispatch method consistently achieves a $> 2\times$ speedup over the sorting approach.

Table 3. Sorting-dispatch vs. hash-dispatch execution time for processing 1M tokens across production MoE models. Speedup is computed as sorting time divided by hash time.

Model	E/K	Time (ms)		Speedup
		Sorting	Hash	
Qwen3	128/8	5.1	2.1	2.4
DeepSeek V3	256/8	6.8	3.0	2.2
DBRX	16/4	1.7	0.9	1.8
Mixtral 8×22B	8/2	1.9	1.4	1.3
gpt-oss-120b	128/4	3.6	1.9	1.9
Granite	40/8	3.8	1.6	2.3

7.3 Ablation Studies

We conduct a component-wise breakdown to isolate the contributions of individual optimizations. Memory savings in our approach stem primarily from reduced activation buffers (scaling linearly with token count, activated experts, and model dimension) and intermediate activation savings. Compute speedups are driven by our lightweight dispatch function and fused expert computation.

Tables 4 and 5 illustrate how the baseline PyTorch implementation improves sequentially as we apply Triton kernel optimizations, Smart Activation Checkpointing (SAC), and our Hash dispatch method across various configurations.

Additionally, we isolate the specific overhead of our on-the-fly gather operations. Microbenchmarks reveal that the on-the-fly gather introduces an overhead of approximately 30% compared to standalone, contiguous GEMM kernels (evaluated on a problem size of roughly 400k tokens, $d = 1024$, $E = 8$, and $K = 2$). However, this kernel-level overhead is substantially outweighed by the end-to-end memory and latency benefits achieved by avoiding the materialization of intermediate token buffers.

Table 4. Ablation study on FastMoE MLP with SwiGLU at $L =$ 512k ($d = d' = 256$, $E = 128$, $K = 4$).

Implementation	Dispatch	Time (ms)	Act. Mem (MB)
PyTorch	sort	654	11 750
Triton	sort	333	7958
Triton + SAC	sort	290	6934
Triton + SAC	hash	287	6918

Table 5. Ablation study of FastMoE MLP with SwiGLU at $L = 512$k. Each column group corresponds to a different MoE configuration (d, E, K).

		d=256, E=512, K=4		d=256, E=128, K=4		d=512, E=32, K=4	
Implementation	Dispatch	Time (ms)	Act. Mem (MB)	Time (ms)	Act. Mem (MB)	Time (ms)	Act. Mem (MB)
PyTorch	sort	1076	8166	654	11 750	887	22 278
Triton	sort	352	6294	324	7958	561	14 646
Triton + SAC	sort	340	5782	290	6934	511	12 598
Triton + SAC	hash	336	5766	287	6918	509	12 582

8 Scalability and Distributed Integration

The applicability of MoEBlaze is not limited to single-device execution. Our methodology is fundamentally orthogonal to distributed training strategies and integrates seamlessly with existing parallelism schemes. We utilize a collective-communication model (AllGather, ReduceScatter, All-to-All) rather than point-to-point push/pull semantics.

Industrial MoE training relies on diverse paradigms, primarily Fully Sharded Data Parallelism (FSDP), Tensor Parallelism (TP), Pipeline Parallelism (PP), and Expert Parallelism (EP). MoEBlaze supports these through two distinct interactions:

FSDP, TP, and PP: MoEBlaze integrates natively with FSDP, TP, and PP without any modification. These parallel schemes do not shard token activations across devices; therefore, the per-device MoE computation remains mathematically identical to the single-device formulation. As demonstrated in Figure 7, when applying FSDP across multiple GPUs, MoEBlaze maintains near-linear scalability with minimal overhead as the cluster size increases. Furthermore, because communication in FSDP, TP, and PP happens at the module, bucket, or micro-batch level, it remains entirely orthogonal to our kernel-level optimizations. This preserves all existing flexibility for compute-communication overlap.

Expert Parallelism (EP): MoEBlaze is highly compatible with EP. In EP, after a cross-device All-to-All token shuffle, each device hosts a specific subset of experts. The subsequent computation on each device reduces to a local MoE problem—precisely the scenario MoEBlaze is designed to optimize. While EP naturally interleaves communication with compute, modern networking techniques (such as NVSHMEM) allow for fine-grained fusion of communication with compute kernels, preserving overlap opportunities. The specific fusion of cross-node token shuffling with our single-device routing is a promising avenue for complementary future optimization.

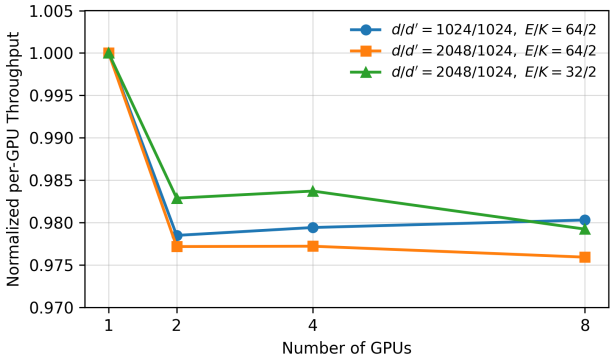

Figure 7. Normalized per-GPU throughput scaling under FSDP across different MoE configurations up to 8 GPUs. Normalized throughput is defined as the per-GPU throughput at N GPUs divided by the per-GPU throughput of the 1-GPU baseline.

9 Related Work

MoE architectures and scaling. Scaling laws and empirical studies established that model performance improves predictably with compute, dataset size, and parameter count (Kaplan et al., 2020; Hoffmann et al., 2022). GPT-3 demonstrated few-shot generalization at 175B parameters (Brown et al., 2020), while follow-on models explored scaling in parameters, training data, and context length (e.g., Gopher at 280B (Rae et al., 2022); PaLM at 540B (Chowdhery et al., 2023)). Open foundation models such as LLaMA (Touvron et al., 2023) accelerated progress by enabling reproducibility and broad evaluation. More recently, proprietary systems improved multimodal integration and long-context capabilities (e.g., GPT-4 (OpenAI et al., 2024), Gemini (Team et al., 2023)), while open releases distilled insights from such systems (Gemma (Team et al., 2024)). These trends increase pressure on both throughput and memory, particularly under longer contexts and larger hidden dimensions.

Mixture-of-Experts (MoE) was popularized for scaling neural networks via sparse conditional computation, initially in recurrent architectures with the Sparsely-Gated MoE formulation (Shazeer et al., 2017). Subsequent work demon-

strated large-scale Transformer-based MoEs with automatic sharding and conditional computation in production-scale systems (e.g., GShard) (Samuel, 1959). Switch Transformers replaced top-k expert selection with top-1 to simplify routing and improve throughput (Fedus et al., 2022). GLaM explored large-scale MoE training with expert sparsity and strong efficiency–quality tradeoffs (Du et al., 2021). In the open-source ecosystem, Mixtral 8×7B employs top-2 routing with strong performance at 32k context (Jiang et al., 2024), while DeepSeek-V3 reports a 671B-parameter MoE with 37B active parameters per token and efficient large-scale training (DeepSeek-AI et al., 2025).

Systems for MoE training and routing. Early GPU-first stacks such as FastMoE offered a PyTorch-based distributed MoE system with practical acceleration and multi-node expert placement (He et al., 2021). Tutel proposed Flex, a design enabling runtime-adaptive parallelism and pipelining to handle routing-induced workload variability, showing large speedups across scales (Hwang et al., 2023). DeepSpeed-MoE introduced both training and inference optimizations to support next-generation MoE at scale (Rajbhandari et al., 2022). MegaBlocks reformulated MoE computation as block-sparse operations to avoid padding or token dropping and map well to GPUs (Gale et al., 2023). TurboMoE argued the gating path is a core bottleneck and introduced fused, metadata-driven kernels and data-layout transformations that reduce sparse-compute overheads and improve large-scale throughput (Aminabadi et al., 2025).

Routing policies, load balancing, and capacity. MoE quality and efficiency depend on the router. The auxiliary load-balancing loss in Sparsely-Gated MoE mitigates expert imbalance (Shazeer et al., 2017); Switch Transformers adopted top-1 routing plus capacity constraints to reduce compute and simplify gather/scatter (Fedus et al., 2022). GShard explored routing and tensor-sharding policies at massive scale (Lepikhin et al., 2021). Open MoE models such as Mixtral use top-2 routing and capacity factors tuned for stability and throughput (Jiang et al., 2024). DeepSeek-V3 further reports an auxiliary-loss-free strategy for load balancing while scaling training to very large regimes (DeepSeek-AI et al., 2025). Across these designs, routing capacity and token dropping vs. padding interact with both throughput and memory pressure, particularly at long contexts.

Kernel and performance optimization. GPU performance for MoE hinges on data movement minimization and on-chip residency. MegaBlocks leverages block-sparse kernels to avoid wasteful dense padding (Gale et al., 2023), and TurboMoE fuses gating, scatter/gather, and expert combination with tailored kernels that avoid expensive sparse MMs (Aminabadi et al., 2025). Complementary to sparse mapping and orchestration, architecture-conscious fusion can shorten activation lifetimes (e.g., fusing non-linearities such as SiLU/SwiGLU with expert GEMMs) and reduce read/write traffic. Our work (MoEBlaze) advances this line by eliminating per-expert routed activation buffers via compact metadata and co-fusing routing and expert compute in microarchitecture-optimized kernels for current-generation GPUs.

10 Conclusion and Future Work

We present MoEBlaze, a fast and memory efficient system for MoE training on GPU. MoEBlaze eliminating the need for materializing large per-expert activation buffers with fused token dispatch and compute kernel designs. Furthermore, MoEBlaze consolidates the MoE computation and activation pipelines to minimize read/write traffic for better memory bandwidth savings and footprint reduction. Our experimental shows that MoEBlaze provides a highly efficient and scalable solution across a range of configurations with over 4× speedups and over 50% activation memory savings.

While this paper primarily focuses on single-device performance, we note that the core mechanisms of MoEBlaze are also applicable to distributed settings. As furture work, we plan to extend MoEBlaze to distributed training frameworks and study the optimizations for multi-node, multi-GPU MoE training.

11 Acknowledgments

We gratefully acknowledge Carole-Jean Wu for insightful discussions and consultation. We are also grateful to Hongtao Yu for his expertise and support on the Triton language..

References

Aminabadi, R. Y., Holmes, C., Rajbhandari, S., Yao, Z., and He, Y. Turbomoe: Enhancing moe model training with smart kernel-fusion and data transformation. 2025.

Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D. M., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., and Amodei, D. Language models are few-shot learners. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*, NIPS '20, Red Hook, NY, USA, 2020. Curran Associates Inc. ISBN 9781713829546.

Chowdhery, A., Narang, S., Devlin, J., Bosma, M., Mishra, G., Roberts, A., Barham, P., Chung, H. W., Sutton,

C., Gehrmann, S., Schuh, P., Shi, K., Tsvyashchenko, S., Maynez, J., Rao, A., Barnes, P., Tay, Y., Shazeer, N., Prabhakaran, V., Reif, E., Du, N., Hutchinson, B., Pope, R., Bradbury, J., Austin, J., Isard, M., Gur-Ari, G., Yin, P., Duke, T., Levskaya, A., Ghemawat, S., Dev, S., Michalewski, H., Garcia, X., Misra, V., Robinson, K., Fedus, L., Zhou, D., Ippolito, D., Luan, D., Lim, H., Zoph, B., Spiridonov, A., Sepassi, R., Dohan, D., Agrawal, S., Omernick, M., Dai, A. M., Pillai, T. S., Pellat, M., Lewkowycz, A., Moreira, E., Child, R., Polozov, O., Lee, K., Zhou, Z., Wang, X., Saeta, B., Diaz, M., Firat, O., Catasta, M., Wei, J., Meier-Hellstern, K., Eck, D., Dean, J., Petrov, S., and Fiedel, N. Palm: scaling language modeling with pathways. *J. Mach. Learn. Res.*, 24(1), January 2023. ISSN 1532-4435.

DeepSeek-AI, Liu, A., Feng, B., Xue, B., Wang, B., Wu, B., Lu, C., Zhao, C., Deng, C., Zhang, C., Ruan, C., Dai, D., Guo, D., Yang, D., Chen, D., Ji, D., Li, E., Lin, F., Dai, F., Luo, F., et al. Deepseek-v3 technical report, 2025. URL https://arxiv.org/abs/2412.19437.

Du, N., Huang, Y., Dai, A. M., Tong, S., Lepikhin, D., Xu, Y., Krikun, M., Zhou, Y., Yu, A. W., Firat, O., Zoph, B., Fedus, L., Bosma, M., Zhou, Z., Wang, T., Wang, Y. E., Webster, K., Pellat, M., Robinson, K., Meier-Hellstern, K. S., Duke, T., Dixon, L., Zhang, K., Le, Q. V., Wu, Y., Chen, Z., and Cui, C. Glam: Efficient scaling of language models with mixture-of-experts. In *International Conference on Machine Learning*, 2021. URL https://api.semanticscholar.org/CorpusID:245124124.

Elfwing, S., Uchibe, E., and Doya, K. Sigmoid-weighted linear units for neural network function approximation in reinforcement learning. *CoRR*, abs/1702.03118, 2017. URL http://arxiv.org/abs/1702.03118.

Fedus, W., Zoph, B., and Shazeer, N. Switch transformers: scaling to trillion parameter models with simple and efficient sparsity. *J. Mach. Learn. Res.*, 23(1), January 2022. ISSN 1532-4435.

Gale, T., Narayanan, D., Young, C., and Zaharia, M. Megablocks: Efficient sparse training with mixture-of-experts. In Song, D., Carbin, M., and Chen, T. (eds.), *Proceedings of Machine Learning and Systems*, volume 5, pp. 288–304. Curan, 2023.

He, J., Qiu, J., Zeng, A., Yang, Z., Zhai, J., and Tang, J. Fastmoe: A fast mixture-of-expert training system. *CoRR*, abs/2103.13262, 2021. URL https://arxiv.org/abs/2103.13262.

Hoffmann, J., Borgeaud, S., Mensch, A., Buchatskaya, E., Cai, T., Rutherford, E., de Las Casas, D., Hendricks, L. A., Welbl, J., Clark, A., Hennigan, T., Noland, E., Millican, K., van den Driessche, G., Damoc, B., Guy, A., Osindero, S., Simonyan, K., Elsen, E., Vinyals, O., Rae, J. W., and Sifre, L. Training compute-optimal large language models. In *Proceedings of the 36th International Conference on Neural Information Processing Systems*, NIPS '22, Red Hook, NY, USA, 2022. Curran Associates Inc. ISBN 9781713871088.

Hwang, C., Cui, W., Xiong, Y., Yang, Z., Liu, Z., Hu, H., Wang, Z., Salas, R., Jose, J., Ram, P., Chau, H., Cheng, P., Yang, F., Yang, M., and Xiong, Y. Tutel: Adaptive mixture-of-experts at scale. In Song, D., Carbin, M., and Chen, T. (eds.), *Proceedings of Machine Learning and Systems*, volume 5, pp. 269–287. Curan, 2023.

Jiang, A. Q., Sablayrolles, A., Roux, A., Mensch, A., Savary, B., Bamford, C., Chaplot, D. S., de Las Casas, D., Hanna, E. B., Bressand, F., Lengyel, G., Bour, G., Lample, G., Lavaud, L. R., Saulnier, L., Lachaux, M., Stock, P., Subramanian, S., Yang, S., Antoniak, S., Scao, T. L., Gervet, T., Lavril, T., Wang, T., Lacroix, T., and Sayed, W. E. Mixtral of experts. *CoRR*, abs/2401.04088, 2024. doi: 10.48550/ARXIV.2401.04088. URL https://doi.org/10.48550/arXiv.2401.04088.

Kaplan, J., McCandlish, S., Henighan, T., Brown, T. B., Chess, B., Child, R., Gray, S., Radford, A., Wu, J., and Amodei, D. Scaling laws for neural language models. *CoRR*, abs/2001.08361, 2020. URL https://arxiv.org/abs/2001.08361.

Lepikhin, D., Lee, H., Xu, Y., Chen, D., Firat, O., Huang, Y., Krikun, M., Shazeer, N., and Chen, Z. Gshard: Scaling giant models with conditional computation and automatic sharding. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021. URL https://openreview.net/forum?id=qrwe7XHTmYb.

OpenAI, Achiam, J., Adler, S., Agarwal, S., Ahmad, L., Akkaya, I., Aleman, F. L., Almeida, D., Altenschmidt, J., Altman, S., Anadkat, S., Avila, R., Babuschkin, I., Balaji, S., Balcom, V., Baltescu, P., Bao, H., Bavarian, M., Belgum, J., Bello, I., Berdine, J., Bernadett-Shapiro, G., Berner, C., Bogdonoff, L., Boiko, O., Boyd, M., Brakman, A.-L., Brockman, G., Brooks, T., Brundage, M., Button, K., Cai, T., Campbell, R., Cann, A., Carey, B., Carlson, C., Carmichael, R., Chan, B., Chang, C., Chantzis, F., Chen, D., Chen, S., Chen, R., Chen, J., Chen, M., Chess, B., Cho, C., Chu, C., Chung, H. W., Cummings, D., Currier, J., Dai, Y., Decareaux, C., Degry, T., et al. Gpt-4 technical report, 2024. URL https://arxiv.org/abs/2303.08774.

Rae, J. W., Borgeaud, S., Cai, T., Millican, K., Hoffmann, J., Song, F., Aslanides, J., Henderson, S., Ring,

R., Young, S., Rutherford, E., Hennigan, T., Menick, J., Cassirer, A., Powell, R., van den Driessche, G., Hendricks, L. A., Rauh, M., Huang, P.-S., Glaese, A., Welbl, J., Dathathri, S., Huang, S., Uesato, J., Mellor, J., Higgins, I., Creswell, A., McAleese, N., Wu, A., Elsen, E., Jayakumar, S., Buchatskaya, E., Budden, D., Sutherland, E., Simonyan, K., Paganini, M., Sifre, L., Martens, L., Li, X. L., Kuncoro, A., Nematzadeh, A., Gribovskaya, E., , et al. Scaling language models: Methods, analysis & insights from training gopher, 2022. URL https://arxiv.org/abs/2112.11446.

Rajbhandari, S., Li, C., Yao, Z., Zhang, M., Aminabadi, R. Y., Awan, A. A., Rasley, J., and He, Y. DeepSpeed-MoE: Advancing mixture-of-experts inference and training to power next-generation AI scale. In Chaudhuri, K., Jegelka, S., Song, L., Szepesvari, C., Niu, G., and Sabato, S. (eds.), *Proceedings of the 39th International Conference on Machine Learning*, volume 162 of *Proceedings of Machine Learning Research*, pp. 18332–18346. PMLR, 17–23 Jul 2022. URL https://proceedings.mlr.press/v162/rajbhandari22a.html.

Ramachandran, P., Zoph, B., and Le, Q. V. Searching for activation functions. *CoRR*, abs/1710.05941, 2017. URL http://arxiv.org/abs/1710.05941.

Samuel, A. L. Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 3(3):211–229, 1959.

Shazeer, N. GLU variants improve transformer. *CoRR*, abs/2002.05202, 2020. URL https://arxiv.org/abs/2002.05202.

Shazeer, N., Mirhoseini, A., Maziarz, K., Davis, A., Le, Q. V., Hinton, G. E., and Dean, J. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017. URL https://openreview.net/forum?id=B1ckMDqlg.

Team, G., Anil, R., Borgeaud, S., Alayrac, J.-B., Yu, J., Soricut, R., Schalkwyk, J., Dai, A. M., Hauth, A., Millican, K., et al. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805*, 2023.

Team, G., Mesnard, T., Hardin, C., Dadashi, R., Bhupatiraju, S., Pathak, S., Sifre, L., Rivière, M., Kale, M. S., Love, J., Tafti, P., Hussenot, L., Sessa, P. G., Chowdhery, A., Roberts, A., Barua, A., Botev, A., Castro-Ros, A., Slone, A., Héliou, A., Tacchetti, A., Bulanova, A., Paterson, A., Tsai, B., et al. Gemma: Open models based on gemini research and technology, 2024. URL https://arxiv.org/abs/2403.08295.

Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F., Rodriguez, A., Joulin, A., Grave, E., and Lample, G. Llama: Open and efficient foundation language models. *CoRR*, abs/2302.13971, 2023. doi: 10.48550/ARXIV.2302.13971. URL https://doi.org/10.48550/arXiv.2302.13971.

Williams, S., Waterman, A., and Patterson, D. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, April 2009. ISSN 0001-0782. doi: 10.1145/1498765.1498785. URL https://doi.org/10.1145/1498765.1498785.

Wulf, W. A. and McKee, S. A. Hitting the memory wall: implications of the obvious. *SIGARCH Comput. Archit. News*, 23(1):20–24, March 1995. ISSN 0163-5964. doi: 10.1145/216585.216588. URL https://doi.org/10.1145/216585.216588.