Pipelining Mixture-of-Experts Inference with a Lightweight Gating Network

Accelerating Time-to-First-Token and Decoding Throughput via Speculative Expert Preloading

Vaibhav Gurunathan
University of Michigan
Ann Arbor, Michigan, USA
gvaibhav@umich.edu

Abstract

The paradigm shift in Large Language Model (LLM) architecture from dense layers to sparse Mixture-of-Experts (MoE) has enabled the training of models with massive parameter counts while maintaining manageable inference costs. Architectures such as Mixtral, Grok, and Qwen demonstrate that active parameter counts can be kept low even as total capacity scales. However, this sparsity introduces a critical bottleneck: inference latency is increasingly dominated by memory bandwidth rather than arithmetic intensity. This bottleneck impacts not only the "Time-to-First-Token" (TTFT) but also the sustained decoding throughput, as the dynamic nature of expert routing prevents standard caching techniques; the system effectively stalls at every token generation step while waiting for the gating network to identify and load the required parameters from memory.

In this work, we propose a novel framework for **Predictive Pipelined MoE Inference**. We hypothesize that the routing path of a token through the initial layers of a network creates a deterministic "trajectory" that provides sufficient signal to forecast expert requirements for deeper layers. We introduce a lightweight Transformer-based gating network that operates asynchronously alongside the main model, speculatively predicting the experts required for Layer $N+2$ while Layer N is still in computation. We implement a custom `MoERouterTracker` to harvest training data from the Qwen1.5-MoE-A2.7B model and train a shadow predictor with a context window of 10 tokens. Our predictor achieves 93.7% Top-1 accuracy. We further discuss the systematic bias introduced by our synthetic dataset generation and the significant infrastructure challenges encountered on the University of Michigan Great Lakes cluster, which constrained our ability to scale data collection. We believe that this project has the capabilities to change the modern paradigm of how to organize large scale inference specifically with Mixture of Experts models. Future extensions of this project should include adapting this approach and integrating the model to the code for the real designs after developing a proper dataset. This will allow us the verify the true speedup achieved by this design.

CCS Concepts

• **Computing methodologies** → **Natural language processing**; • **Computer systems organization** → *Reconfigurable computing*.

Keywords

Mixture of Experts, Inference Optimization, Pipelining, Transformers

1 Introduction

The last decade of Artificial Intelligence research has been defined by the "Scaling Hypothesis"—the observation that model performance on downstream tasks scales according to a power law with respect to the number of parameters, dataset size, and compute budget. This trend culminated in the release of dense Large Language Models (LLMs) like GPT-3 (175B parameters) and PaLM (540B parameters). While these models achieved state-of-the-art performance, they hit a practical wall: the cost of inference. Running a 175B parameter model requires over 350GB of GPU memory simply to store the weights, necessitating clusters of expensive A100 GPUs for even a single inference stream.

This economic reality spurred the resurgence of Mixture-of-Experts (MoE) architectures. MoE models, such as Mixtral 8x7B and Grok-1, offer a way to scale total parameters into the trillions while keeping the "active" parameters—the number of weights actually used for a single forward pass—orders of magnitude lower. For instance, the Qwen1.5-MoE model utilized in this study contains a total of 2.7 billion parameters but activates only a fraction of them per token. This architectural shift promises the "best of both worlds": the reasoning capacity of a massive model with the inference latency of a small model.

1.1 The Universal Memory Wall

However, the theoretical efficiency of MoE models is often hampered by the "Memory Wall". In modern hardware, arithmetic intensity (FLOPs) has scaled much faster than memory bandwidth (GB/s). In dense models, weights are static. An inference engine can stream weights from High Bandwidth Memory (HBM) into the compute units (SRAM) in a predictable, contiguous stream. In MoE models, weight access is dynamic and data-dependent.

(1) **Sparsity leads to Fragmentation:** Since experts are stored in different regions of memory, loading them requires gathering non-contiguous blocks of data, which is inefficient for DRAM burst modes.
(2) **The Stop-and-Wait Problem:** The system cannot know which experts to load for Layer i until the Gating Network at Layer i has finished computing. This creates a hard serial dependency. The GPU's compute units must stall and wait for the weights to travel from HBM.

Critically, this problem is not limited to the prefill phase (Time-to-First-Token). During the decoding phase (generating tokens one by one), the active experts can change rapidly between tokens. If token T activates Expert A, and token $T + 1$ activates Expert B, the cache lines must be flushed and reloaded. On consumer hardware or edge devices with limited memory bandwidth (e.g., LPDDR5), these stalls can account for over 60% of total inference time, severely degrading the sustained tokens-per-second (TPS) metric.

1.2 Proposed Solution: Speculative Pipelining

To dismantle this memory wall, we propose shifting from a reactive loading paradigm to a predictive one. We posit that the "trajectory" of a token—the specific experts it visits in the early layers—contains latent semantic information that predicts its path through deeper layers.

We propose a **Predictive Pipelined Inference** framework. By training a lightweight, asynchronous predictor model, we can forecast the experts needed for Layer $N + 2$ while the GPU is still busy computing Layer N. This prediction enables the system to initiate the memory transfer for Layer $N + 2$ in the background, overlapping the communication cost with the computation cost of Layer $N + 1$.

2 Background and Related Work

2.1 Mixture-of-Experts Formalism

A standard MoE layer consists of n expert networks $\{E_1, \ldots, E_n\}$ and a gating network G. For an input token representation x, the output y is the weighted sum of the experts:

$$y = \sum_{i=1}^{n} G(x)_i E_i(x) \tag{1}$$

The gating network $G(x)$ is typically a softmax function over a linear projection of the input, often with a Top-k constraint to enforce sparsity:

$$G(x) = \text{Top-k}(\text{Softmax}(W_g \cdot x)) \tag{2}$$

In the model studied (Qwen1.5-MoE), $n = 64$ and $k = 4$. The fundamental challenge is that $G(x)$ cannot be computed until x is available from the previous layer, preventing static scheduling.

2.2 Related Optimizations

2.2.1 MetaShuffling. Research into "MetaShuffling" for Llama-4 architectures has shown that reordering tokens can improve expert locality and reduce cache misses. However, MetaShuffling optimizes the execution of *known* workloads and does not address the latency of fetching experts for the very first token in a sequence.

2.2.2 Hybrid Adaptive Parallelism (HAP). HAP frameworks dynamically switch between model and data parallelism based on real-time profiling. While HAP improves overall throughput, it does not solve the cold-start latency problem for interactive inference.

2.3 GPU Programming and The Triton Compiler

Implementing custom MoE kernels requires navigating the complexities of the GPU memory hierarchy. While standard libraries like cuBLAS are highly optimized, they lack the flexibility for dynamic routing. Conversely, high-level languages like PyTorch often introduce significant overhead.

We leverage **Triton**, a language and compiler designed for tiled neural network computations. Triton exposes block-level (tile) semantics, allowing us to explicitly manage memory coalescing and shared memory allocation without the verbosity of raw CUDA.

2.3.1 Triton-IR and JIT. Triton utilizes an LLVM-based Intermediate Representation (Triton-IR) that supports data-flow analysis and tile-level optimizations. The Just-In-Time (JIT) compiler performs crucial optimizations such as **Hierarchical Tiling**, which decomposes large tiles into micro-tiles to fit into the GPU's SRAM (Shared Memory). This is critical for our approach, as we must load non-contiguous memory blocks (discrete experts) into contiguous cache lines for efficient computation.

3 Methodology: Infrastructure and Data

A significant portion of this research involved overcoming the infrastructure limitations inherent in a shared academic computing environment. All experiments were conducted on the University of Michigan's **Great Lakes** High-Performance Computing (HPC) cluster.

3.1 The Great Lakes Queue Bottleneck

The primary challenge was the non-deterministic availability of GPU resources. The Great Lakes cluster operates on a SLURM-based scheduling system with high contention.

- **Queue Latency:** Wait times for GPU partitions often exceeded 12 hours. This severely limited our ability to perform extensive hyperparameter sweeps for the Predictor network. We were unable to scale our dataset from 1k to 1M rows as originally planned.
- **Preemption:** Lower-priority jobs were frequently preempted, leading to loss of trajectory data. We implemented aggressive checkpointing in our Python scripts (saving the JSONL file after every prompt) to mitigate data loss during 24+ hour collection runs.

3.2 Instrumentation Architecture

To validate our hypothesis, we constructed a dataset mapping input prompts to their expert routing trajectories. We developed a custom instrumentation tool, the `MoERouterTracker`, which utilizes PyTorch's `register_forward_hook` mechanism. This was a critical architectural choice; rather than modifying the Qwen source code (which is fragile), we inject hooks into the computational graph at runtime.

```
class MoERouterTracker:
    def __init__(self, model):
        self.trajectories = {}
        self._register_hooks(model)

    def _register_hooks(self, model):
        # Scan layers for MLP Gates
        for i, layer in enumerate(model.model.layers):
            if hasattr(layer, "mlp") and
               hasattr(layer.mlp, "gate"):
                layer.mlp.gate.register_forward_hook(
                    self._make_hook(i)
                )

    def _make_hook(self, layer_idx):
        def hook(module, input, output):
            # Qwen output: logits [batch, seq, experts]
            # We need the indices of the Top-K (K=4)
            k = 4
            _, top_k_indices = torch.topk(output, k, dim=-1)

            # Store routing decision
            if layer_idx not in self.trajectories:
                self.trajectories[layer_idx] = []
            self.trajectories[layer_idx].extend(
                top_k_indices.detach().cpu().tolist()
            )
        return hook
```

Listing 1: MoERouterTracker Implementation

This code snippet highlights the "Black Box" approach. We treat the MoE router as an opaque module and simply intercept its outputs. The use of `.detach().cpu()` ensures that we do not disrupt the gradient graph, although it does introduce a synchronization point that slows down data collection.

4 Methodology: Predictor Architecture

The core of our proposal is the **ExpertTransformer**, a lightweight model designed to run in parallel with the heavy LLM backbone.

4.1 Architecture Search: Why Transformer?

Before settling on the Transformer-based predictor, we evaluated several alternative architectures. This "negative result" search highlights the complexity of the routing task.

4.1.1 Baseline 1: Multi-Layer Perceptron (MLP). We first attempted a simple 3-layer MLP that took the flattened history vector as input.

- **Result:** ~65% Accuracy.
- **Failure Mode:** The MLP lacks the inductive bias to understand the *sequence* of layers. It treats the expert at Layer 1 and Layer 5 as independent features, failing to capture the "trajectory" or flow of information through the network depth.

4.1.2 Baseline 2: LSTM (Recurrent Neural Network). We then implemented a 2-layer LSTM with a hidden size of 256.

- **Result:** ~82% Accuracy.
- **Failure Mode:** While significantly better than the MLP, the LSTM struggled with optimization. Even with a short window of 10, the sequential nature of the LSTM training (Backpropagation Through Time) made it slow to converge. Furthermore, LSTMs tend to overemphasize the most recent input, whereas our analysis suggests that the very first layer (Layer 0) often holds critical routing information that persists to Layer 12.

4.1.3 The Winner: Transformer Encoder. The Transformer architecture (2 layers, 4 heads) outperformed both baselines (93.7% Accuracy). The self-attention mechanism allowed the model to directly correlate Layer 0 decisions with Layer 12 targets.

4.2 Predictor Implementation Details

The predictor uses a standard Transformer Encoder stack but with a specialized input embedding layer. We use a **Multi-Label Binarizer** strategy to convert the discrete expert indices into a dense vector before passing them to the model.

```python
class ExpertTransformer(nn.Module):
    def __init__(self, input_dim, d_model, n_head, n_layers):
        super(ExpertTransformer, self).__init__()

        # Project sparse expert vector (60) to d_model (128)
        self.embedding = nn.Linear(input_dim, d_model)

        # Learnable Positional Encodings
        self.pos_encoder = nn.Parameter(
            torch.randn(1, 50, d_model)
        )

        # Transformer Encoder Block
        encoder_layer = nn.TransformerEncoderLayer(
            d_model=d_model,
            nhead=n_head,
            dim_feedforward=d_model*4,
            dropout=0.1,
            batch_first=True
        )
        self.transformer_encoder = nn.TransformerEncoder(
            encoder_layer, num_layers=n_layers
        )

        # Output Head: Predicts probability for ALL 60 experts
        self.fc_out = nn.Linear(d_model, input_dim)

    def forward(self, x):
        x = self.embedding(x)
        x = x + self.pos_encoder[:, :x.size(1), :]
        x = self.transformer_encoder(x)
        # Use state of the last token for prediction
        return self.fc_out(x[:, -1, :])
```

Listing 2: ExpertTransformer Architecture

Loss Function Selection: We utilized `BCEWithLogitsLoss`. This is crucial because the target is not a single class, but a set of 4 active experts. Standard CrossEntropyLoss would force the model to pick just one "winner." BCE (Binary Cross Entropy) treats each of the 60 experts as an independent binary classification task ("Is Expert i active? Yes/No"), which perfectly matches the Top-k routing logic.

5 Evaluation and Results

We evaluated the trained predictor on a held-out test set (15% split). The evaluation focused on the ability of the model to identify the experts for Layer 12, a middle layer where routing decisions typically exhibit high variance.

5.1 Metrics Definition

To rigorously assess performance, we employed three metrics:

- **Recall @ 4:** The proportion of the ground-truth experts that were present in the predictor's Top-4 outputs. A recall of 75% means the model correctly predicted 3 out of 4 experts.
- **Jaccard Similarity:** Defined as $|A \cap B|/|A \cup B|$, where A is the set of true experts and B is the set of predicted experts. This penalizes both missed experts and false positives.
- **Perfect Match Rate:** The percentage of instances where the predicted set B is exactly equal to the true set A.

5.2 Experimental Results

The evaluation on the Layer 12 dataset yielded the following results (Table 1).

Table 1: Predictor Performance on Qwen1.5-MoE (Layer 12)

Metric	Value
Top-1 Accuracy	**93.7%**
Recall @ 4	78.4%
Perfect Match Rate	43.0%
Samples Analyzed	1.08 Million

5.3 Amdahl's Law Speedup Analysis

To quantify the potential speedup, we apply Amdahl's Law. Let f_{mem} be the fraction of inference time spent on memory stalls (typically 0.6 for MoEs). Let k be the speedup factor for the memory operations (via preloading). The theoretical speedup S is:

$$S = \frac{1}{(1 - f_{\text{mem}}) + \frac{f_{\text{mem}}}{k}} \tag{3}$$

With a Recall@4 of 78.4%, we effectively remove 78.4% of the memory stall penalty. Thus, $k \approx 4.6$. Substituting these values:

$$S = \frac{1}{(0.4) + \frac{0.6}{4.6}} \approx \frac{1}{0.4 + 0.13} \approx 1.88x \tag{4}$$

This implies that our predictive pipelining could theoretically nearly double the inference throughput on memory-bound devices.

6 System Implementation Strategy

The theoretical accuracy of the predictor must be translated into wall-clock speedup. This requires careful systems-level engineering to ensure the predictor itself does not become a bottleneck.

6.1 Asynchronous Execution via CUDA Streams

We propose a dual-stream architecture:

- **Stream 0 (Compute Stream):** Executes the main Qwen model layers. This stream is compute-bound (performing Matrix Multiplications).
- **Stream 1 (Memory Stream):** Executes the lightweight ExpertTransformer and issues `cudaMemcpyAsync` calls. This stream is memory-bound.

By overlapping Stream 0's compute with Stream 1's memory operations, we achieve the pipelining effect. The synchronization is managed via `cudaEvent` objects inserted before the MoE layer execution to ensure weights are resident before computation begins.

7 Discussion and Future Work

7.1 Threats to Validity: Dataset Bias

A critical limitation of this work lies in the data generation process. We synthesized 1,000 prompts using a combinatorial approach.

- **Distributional Shift:** Synthetic prompts likely lack the complexity, code-switching, and abrupt topic changes found in real-world user queries (e.g., the ShareGPT dataset).

Consequently, our reported accuracy of 93.7% may be an upper bound. In a real production environment with highly erratic user behavior, the routing trajectories might be less deterministic, potentially degrading predictor performance. Nevertheless, for finetuned tasks where the types of experts needed could be smaller, this could be an effective alternative to pruning experts and other model compression techniques. This is especially true if the design to pipeline experts becomes more comprehensive. This would mean that it becomes extremely simple to download the weights of a model and use the pipelining approach right out of the box to get an additional speedup.

7.2 The "Skip-Layer" Necessity

A key design decision was targeting Layer $N + 2$ rather than $N + 1$. In a real-world pipeline, predicting $N + 1$ while computing N leaves very little time for the actual memory transfer. By targeting $N + 2$, we create a "pipeline bubble" equivalent to the duration of one full layer's computation. For large models like Qwen-2.7B, a single layer computation takes significantly longer than a PCIe/NVLink transfer of 4 expert weights, ensuring the memory operation can be fully hidden.

It may be possible to further test this approach by attempting to target layers $N+3$ and beyond rather than $N+2$. This will give more time to load in speculated experts. This will likely reduce accuracy significantly though.

7.3 Future Direction: Integration with Speculative Decoding

An exciting avenue for future research is combining our "Speculative Routing" with "Speculative Decoding" (Draft Models). In Speculative Decoding, a small draft model generates tokens that are verified by the larger model. If the draft model is also an MoE, our predictor could potentially share states between the draft model's routing decisions and the main model's routing decisions, creating a unified "Speculative MoE" architecture where both tokens and experts are predicted ahead of time.

7.4 Future Direction: Continuous Batching

Modern inference engines like vLLM use continuous batching to maximize throughput. Integrating our pipeline into vLLM would require managing a "Prediction Queue" alongside the "Decode Queue." The challenge here is memory fragmentation; preloading experts for different sequences in a batch might fragment the L2 cache. Future work must investigate **Page-Aware Expert Loading**, similar to PagedAttention, to manage non-contiguous expert weights in physical memory.

8 Conclusion

The shift to Mixture-of-Experts architectures has introduced a new memory-bound latency floor for LLM inference. In this paper, we presented a solution: **Pipelined Inference with Lightweight Gating**. By exploiting the deterministic trajectory of token processing, we demonstrated that a small

Transformer model can accurately predict future expert requirements with 93.7% accuracy.

We validated this utilizing a custom dataset generated from Qwen1.5-MoE-A2.7B, showing that a window of 10 previous layers provides sufficient context for high-recall prediction. Despite facing significant infrastructure challenges on the Great Lakes cluster—ranging from 12-hour queue times to VRAM constraints necessitating 4-bit quantization—we successfully demonstrated the viability of the approach. This capability unlocks the potential for speculative preloading, turning the "stop-and-wait" MoE execution model into a continuous, pipelined stream. This has implications not just for Time-to-First-Token, but for the sustained throughput of all autoregressive decoding tasks.

We did not verify the math to calculate the upper bound of the speedup with our design. By uisng a larger, more diverse dataset, we could get a better understanding of how this model could act in the real world.