HarMoEny: Efficient multi-GPU inference of Mixture-of-Experts models by scheduling experts across accelerators

Zachary Doucet, School of Computer Science
McGill University, Montreal
May, 2025

A thesis submitted to McGill University in partial fulfillment of the requirements of the degree of Master of Computer Science

Abstract

Mixture-of-Experts (MoE) models enhance computational efficiency during inference by selectively activating specialized experts for each request. This approach facilitates efficient model scaling on multi-GPU systems leveraging expert parallelism, without compromising performance. However, uneven load distribution across experts often creates imbalances across GPUs, increasing latency due to idle waiting times. To address this, we propose HarMoEny, a novel solution that employs two simple techniques: *(i)* dynamic token redistribution to underutilized GPUs and *(ii)* asynchronous prefetching of experts from system to GPU memory. These methods achieve near-perfect load balancing across GPUs and amortize the cost of executing experts not resident in GPU memory, thereby reducing waiting times and latency. We implement HarMoEny and evaluate its performance against four state-of-the-art baselines using real-world and artificially imbalanced datasets. Under heavy load imbalance, HarMoEny improves throughput by 66.7%-68.6% and reduces time to first token by 31.8%-37.2%, compared to the next-best baseline. Furthermore, our ablation study demonstrates that HarMoEny's scheduling policy reduces the GPU idle time by up to 99% relative to the baseline policies.

Abrégé

Les modèles Mixture-of-Experts (MoE) améliorent l'efficacité computationnelle lors de l'inférence en activant sélectivement des experts spécialisés pour chaque requête. Cette approche permet une montée en échelle efficace des modèles sur des systèmes multi-GPU grâce au parallélisme d'experts, sans compromettre les performances. Cependant, une répartition inégale de la charge entre les experts engendre souvent des déséquilibres entre les GPU, augmentant ainsi la latence à cause des temps d'attente inactifs. Pour remédier à cela, nous proposons HarMoEny, une solution novatrice qui repose sur deux techniques simples : *(i)* la redistribution dynamique des jetons vers les GPU sous-utilisés et *(ii)* le préchargement asynchrone des experts de la mémoire système vers la mémoire GPU. Ces méthodes permettent un équilibrage de charge quasi-parfait entre les GPU et amortissent le coût d'exécution des experts non présents en mémoire GPU, réduisant ainsi les temps d'attente et la latence. Nous avons implémenté HarMoEny et évalué ses performances face à quatre références de pointe, en utilisant des jeux de données réels et artificiellement déséquilibrés. En cas de fort déséquilibre de charge, HarMoEny améliore le débit de 66,7 % à 68,6 % et réduit le temps jusqu'au premier jeton de 31,8 % à 37,2 %, par rapport à la meilleure référence. De plus, notre étude d'ablation montre que la politique d'ordonnancement de HarMoEny réduit le temps d'inactivité des GPU jusqu'à 99 % par rapport aux politiques de référence.

Contents

Acknowledgements

This endeavour is not endurable without the help of an amazing group of people. These people have written this thesis, figuratively, just as much as I have. I would like to firstly thank my supervisor **Prof. Oana Balmau** who has been my buoy for the past 3½ years of my life. This time has been filled with growth mostly thanks to her attention, work-ethic, and flexibility. Thanks for the constant push for me to achieve and for going through this thesis enumerable times to improve it to its current state. The lessons learned here will serve for decades to come. I would like to then thank **Prof. Anne-Marie Kermarrec** and the entire lab at **SaCS** for hosting me and helping to spawn the work of this thesis. Your work has been monumental, but more importantly has opened my mind to all that life has to offer. So, thank you **Dr. Rafael Pires**, **Dr. Martijn Abraham de Vos**, **Rishi Sharma** and of course **Dr. Sayan Biswas** (you know why). You have all been pivotal to this work. Next, I would like to thank the **DISCS** lab for its friendly collaborative atmosphere and to all the wonderful students, thank you all. I would like to thank the **DGX1** machine and also to its competitive time sharing, also thank you **Madeleine Robert** for giving me access to the EPFL cluster and managing to outspend everyone by an order of magnitude within 2 months.

Evidently, my **parents** played a quintessential role, and their continual strong support of me and my growth has been the biggest blessing of my life that I appreciate day in,

day out. Thanks **mom** for all the hot chocolates too. Also, thanks to the three little dogs of the apocalypse: **Rockie**, **Daisy** and **Bella**—who through some undiscovered forces can whisk away stress at a blink of an eye.

And to all who have not been listed here, *thank you*, it takes a city to write a thesis.

List of Figures

List of Tables

List of Algorithms

List of Equations

List of Abbreviations

- 2DH: Two-Dimensional Hierarchical
- CDF: Cumulative Distribution Function
- DRAM: Dynamic Random Access Memory
- E2E: End-to-End
- FFN: Feed-Forward Network
- GI: Gini-Index
- GQA: Grouped Query Attention
- ILP: Integer Linear Programming
- LOC: Lines of Code
- LSTM: Long Short Term Memory
- MLP: Multi-Layer Perceptron
- MMLU: Measuring Massive Multitask Language Understanding
- MoE: Mixture-of-Expert

- NLP: Natural Language Processing
- NN: Neural Network
- OOM: Out-of-Memory
- PR-MoE: Pyramid-Residual MoE
- RGB: Red Green Blue
- RLHF: Reinforcement Learning from Human Feedback
- RNN: Recurrent Neural Network
- RoPE: Rotary Positional Embeddings
- SLA: Service Level Agreement
- SOTA: State of the Art
- VRAM: Video Random Access Memory

Chapter 1

Introduction

The growth of machine learning models is intentional, driven by the observation that larger models consistently achieve higher accuracy [1]. In this pursuit, Mixture-of-Experts models have gained prominence as an efficient means of scaling parameter counts without a proportional increase in computation cost [2]–[5]. Mixture-of-Experts models partition a neural network into multiple specialized sub-networks, or experts, each tailored to process specific types of inputs, such as linguistic patterns or tasks. A learned routing mechanism directs each input token to only a few experts, enabling MoEs to scale to trillions of parameters while maintaining computational efficiency compared to dense models. They are widely used in natural language processing tasks like translation, text generation, and chatbots, offering superior performance in models like Google's Switch Transformer and Alibaba's Qwen1.5-MoE-A2.7B. Their importance lies in achieving high accuracy with reduced inference costs, critical for large-scale AI deployment.

Given their scale, MoE models require distribution across multiple GPUs, even those with substantial Video Random Access Memory (VRAM), to manage memory and computation. A natural approach, known as expert parallelism, assigns each GPU a subset

of experts, processing only the tokens routed to them by the model's gating mechanism. However, this dynamic routing often unevenly distributes workloads, as certain experts—favored for specific inputs—receive disproportionately more tokens, leaving some GPUs overloaded while others idle, thus degrading throughput. This master's thesis identifies expert token imbalance as a critical bottleneck in expert parallelism and proposes HarMoEny, a practical solution to mitigate this issue. HarMoEny leverages dynamic token redistribution and asynchronous expert prefetching to achieve near-perfect load balancing, engineered to perform at least as well as existing baselines under all conditions and offering an orthogonal enhancement that integrates seamlessly into any MoE inference framework.

1.1 Thesis Overview

- Chapter 2: Background.

  A brief history of machine learning is presented, tracing the emergence of the transformer model and its evolution into the Mixture-of-Experts architecture. This chapter also examines prevalent multi-GPU distribution strategies and their limitations.

- Chapter 3: Related Work.

  HarMoEny is contextualized alongside other efforts to optimize expert parallelism in Mixture-of-Experts models.

- Chapter 4: Methodology.

  This chapter analyzes expert parallelism, exploring the bottleneck of uneven token distribution. It introduces HarMoEny, detailing its two novel components that address this bottleneck. Additionally, it discusses HarMoEny's q hyperpa-

rameter, including its derivation, and describes a novel methodology for creating configurable, artificially imbalanced workloads to test the system.

- Chapter 5: Experimental setup.

  This section outlines the selected systems, models, datasets, hardware, and metrics, preceded by a brief overview of HarMoEny 's implementation details.

- Chapter 6: Results.

  Experimental results are presented and analyzed, highlighting significant achievements and unexpected findings. This chapter demonstrates HarMoEny 's effectiveness in addressing load imbalance challenges.

- Chapter 7: Conclusion.

  A summary of findings is presented to conclude the thesis.

- Chapter 8: Future Work.

  Potential avenues for extending this thesis are provided.

Chapter 2

Background

2.1 Neural networks and the transformer architecture

2.1.1 Brief history

The foundations of neural networks trace back to 1943, when McCulloch and Pitts proposed a mathematical model of the biological neuron [6]. This work introduced a simplified computational unit capable of performing logical operations, laying the groundwork for artificial neural systems. In 1958, Rosenblatt advanced this concept by developing the perceptron, a model accompanied by the first machine designed to learn from data [7]. The perceptron predicts the class of an input by computing a weighted sum of features, $w \cdot x + b$, and applying the Heaviside step function, which outputs 1 if the result exceeds 0 and 0 otherwise:

$$y = H(w \cdot x + b)$$

When a prediction is incorrect, the perceptron adjusts its decision boundary iteratively using the update rule:

$$w \leftarrow w + \alpha(y_{true} - y_{pred})x$$

Here, α represents the learning rate, enabling the perceptron to learn a linear separation between classes over successive data points. While the perceptron demonstrated significant potential, its limitations were exposed in 1969 by Minsky and Papert [8]. Their seminal work highlighted cases where the perceptron fails to learn accurate mappings, most notably the XOR problem. As shown in Table 2.1, the XOR function, which outputs 1 when inputs differ and 0 when they are the same, cannot be linearly separated with a single perceptron:

x_1	x_2	y
0	0	0
1	0	1
0	1	1
1	1	0

Table 2.1: XOR Dataset

This limitation arises because XOR requires a non-linear decision boundary, which a single-layer perceptron cannot capture. This challenge was overcome by stacking multiple perceptrons into a multi-layer perceptron (MLP), a breakthrough that enabled the approximation of any continuous function, as later proven by Hornik [9]. The MLP marked a pivotal shift in neural network design, establishing the foundation for modern architectures.

2.1.2 Natural language processing

Neural networks are widely used to address text-based tasks, such as text classification, text generation, sentiment analysis, and sequence transduction. Unlike images, which are naturally represented by numerical RGB values, text lacks an inherent numerical form, posing a challenge for neural network processing. To overcome this, text must undergo preprocessing, beginning with tokenization. Tokenization splits text into discrete units called tokens, commonly words, though subword or character-level tokenization is also employed in modern approaches. Each token is then mapped to a numerical value using a predefined vocabulary; tokens absent from this vocabulary are assigned a default value, such as an "unknown" token identifier.

Although tokenization yields a numerical representation, these values impose an unintended ordinal relationship—e.g., "wolf" and "cucumber" might be assigned numbers suggesting a ranking, despite lacking intrinsic order. This issue is addressed through embedding, which transforms each token into a dense vector in a latent space. For a neural network with a hidden dimension of d_{model}, each token is represented as a vector of d_{model} floating-point numbers. For instance, in a model with $d_{\text{model}} = 10$, "wolf" becomes a 10-dimensional vector. Unlike a fixed mapping, embeddings are typically learned, often via a neural network such as an MLP, either pre-trained or optimized jointly with the downstream task. Together, tokenization and embedding constitute the essential preprocessing steps for training neural networks on natural language processing (NLP) tasks.

2.1.3 Transformer

In 2017, Vaswani et al. introduced the transformer to enhance neural network performance on transduction tasks, such as machine translation, while reducing the compu-

tational burden of then state-of-the-art recurrent neural networks (RNNs) [9]. Unlike RNNs, which process sequences sequentially and struggle with long-range dependencies, transformers leverage an encoder-decoder architecture reliant entirely on attention mechanisms. The encoder transforms an input sequence (e.g., a source sentence) into a sequence of continuous representations, which the decoder uses to autoregressively generate an output sequence (e.g., a target translation), incorporating previously generated tokens. By eliminating RNNs, transformers enable parallel processing of tokens, significantly improving efficiency. Figure 2.1 illustrates the transformer architecture, highlighting its encoder-decoder structure designed for sequence-to-sequence tasks such as machine translation. The left side of the figure depicts the encoder, which processes an input sequence of tokens (e.g., a source sentence) through a stack of identical layers, each containing two primary sub-components: a multi-head self-attention mechanism and a feed-forward network (FFN). The right side shows the decoder, which generates the output sequence autoregressively, incorporating both self-attention (to attend to previously generated tokens) and encoder-decoder attention (to focus on the encoder's output). Both encoder and decoder layers include residual connections and layer normalization to stabilize training, as indicated by the "Add & Norm" blocks. Positional encodings, added to token embeddings at the input, preserve sequence order, addressing the absence of recurrence in the architecture.

The transformer's design rests on three key innovations: positional encoding, scaled dot-product attention, and multi-head attention. First, since RNNs inherently capture token order, their removal necessitated a mechanism to preserve positional information. Vaswani et al. proposed positional encodings added to token embeddings, defined as sinusoidal functions:

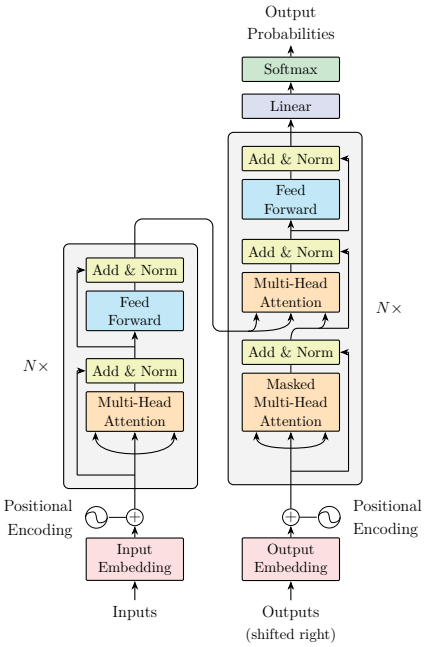

Figure 2.1: Transformer Architecture. Reproduction of [9].

$$PE_{(pos,2i)} = sin(pos/10000^{2i/d_{model}})$$

$$PE_{(pos,2i+1)} = cos(pos/10000^{2i/d_{model}})$$

These encodings, summed element-wise with embeddings (see Figure 2.1, the $\sim$ icon), were tested as both fixed and learned functions, with comparable performance [9]. The sinusoidal form ensures positional information persists across varying sequence lengths.

Second, the transformer employs a scaled dot-product attention mechanism:

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

Here, Q, K, and V represent query, key, and value matrices, respectively. A critical

insight was that unscaled dot products grow large with sequence length, producing small softmax gradients. Scaling by $\sqrt{d_k}$—the dimension of keys and queries—mitigates this, stabilizing training [9].

Third, multi-head attention extends this mechanism by computing attention across h parallel heads, each attending to distinct representation subspaces:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, ..., \text{head}_h)W^o$$
$$\text{where head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

Each head projects Q, K, and V into dimensions d_k, d_k, and d_v, respectively, using learned matrices W_i^Q, W_i^K, and W_i^V. The concatenated outputs are projected via W^o to d_{model}, enabling the model to capture diverse linguistic relationships [9].

These innovations yielded state-of-the-art results on transduction benchmarks. On WMT 2014 English-to-German, the transformer achieved a BLEU score of 28.4, a 2-point improvement over prior best results, and on English-to-French, it scored 41.0, a 0.5-point gain—all with one-quarter the training computation of RNN-based models [9]. This breakthrough catalyzed a surge in transformer-based research since 2017, with most subsequent models building on this foundational architecture.

2.1.4 The T5 transformer model

The transformer architecture introduced by Vaswani et al. [9] has inspired numerous models tailored to specific tasks, with the Text-to-Text Transfer Transformer (T5) standing out as a versatile example. Developed by Raffel et al. at Google, T5 reframes all natural language processing (NLP) tasks as text-to-text problems, taking text input and producing text output regardless of the task—be it classification, generation, or translation

Parameter	Value
dim	768
n_layers	12
head_dim	64
hidden_dim	3072
n_heads	12
vocab_size	32000

Table 2.2: T5-Base Model Architecture. Information aggregated from [10].

[10]. This unified approach enables a single model to handle diverse NLP applications, streamlining training and deployment. T5 builds on the encoder-decoder structure of the original transformer, with its Base variant detailed in Table 2.2

Here, d_{model} denotes the hidden dimension, d_k the attention head dimension, and d_{ff} the feed-forward layer size. Pre-trained on the Colossal Clean Crawled Corpus (C4), T5 demonstrates strong generalization across tasks, setting a foundation for subsequent models like the Switch Transformer, which adapts T5 for sparse computation (see Section 2.2.2). Its flexibility and performance have made T5 a cornerstone in transformer-based NLP research.

2.2 Mixture of Experts

2.2.1 Architecture

The Mixture of Experts (MoE) concept emerged in 1991 when Jacobs et al. proposed a model to address interference in datasets by dividing a task into specialized sub-tasks, each handled by a dedicated expert [11]. Rather than training a single neural network to manage a complex task holistically, Mixture of Experts posits that decomposing it into distinct sub-tasks—each processed by a separate expert—enhances performance by mit-

igating conflicts in learning patterns. For instance, consider training a model to classify word pairs as synonyms or antonyms: a single model might struggle to reconcile these opposing objectives, as features useful for identifying synonyms could clash with those for antonyms. In the MoE framework, each expert, implemented as a neural network, competes to process inputs assigned by a learned gating function. This competition is optimized using a novel loss function:

$$E^c = -\log \sum_i p_i^c e^{-\frac{1}{2}||d^c - o_i^c||^2}$$

Here, c represents a specific input case, i indexes an expert, p_i^c is the gating function's probability of selecting expert i, d^c is the target output, and o_i^c is the expert's prediction. This loss encourages specialization by rewarding experts that closely match their assigned inputs while penalizing poor performers, weighted by the gating probabilities. The approach yielded promising early results, demonstrating improved accuracy on multi-faceted tasks, yet it remained under-explored for nearly two decades as computational resources and interest in scalable architectures lagged.

Renewed interest emerged as model scaling became a priority. Kaplan et al. demonstrated that neural network performance scales with compute, dataset size, and parameter count [1]. However, increasing parameters typically raises computational cost proportionally. Sparse computation, as in MoE, addresses this by activating only a subset of the model, enabling larger architectures without commensurate compute increases. Shazeer et al. advanced this idea by replacing the FFN in a stacked Long Short Term Memory (LSTM) with thousands of experts, each a neural network matching the FFN's input-

output dimensions [12]. The MoE layer's output is computed as:

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

where $E_i(x)$ is expert i's output, and $G(x)_i$ is the gating function's weight for that expert. Typically a single-layer neural network with weights W_g, the gating function incorporates sparsity and noise:

$$G(x) = \text{Softmax}(\text{KeepTopK}(H(x), k))$$

$$H(x)_i = (x \cdot W_g)_i + \text{StandardNormal}() \cdot \text{Softplus}\left((x \cdot W_{\text{noise}})_i\right)$$

$$\text{KeepTopK}(v, k)_i = \begin{cases} v_i & \text{if } v_i \text{ is in the top } k \text{ elements of } v, \\ -\infty & \text{otherwise.} \end{cases}$$

The noise term in $H(x)_i$ enhances load balancing by preventing over-reliance on a few experts, while KeepTopK retains only the top k scores, setting others to zero post-softmax [12].

In transformer architectures (see Figure 2.1), FFNs—typically MLPs—form key building blocks with no inherent limit on their number per layer. These can be scaled horizontally into experts, paired with a gating function (henceforth termed a router), to increase parameters without inflating compute costs–as seen in Figure 2.2. A load-balancing loss is added during training to encourage even token distribution across experts. Hyperparameters like the number of experts and top-k value allow customization. Top-k, the number of experts per token, has been shown to work with a value of 1 with minimal impact [2]. The number of experts in a layer boosts downstream performance positively—though

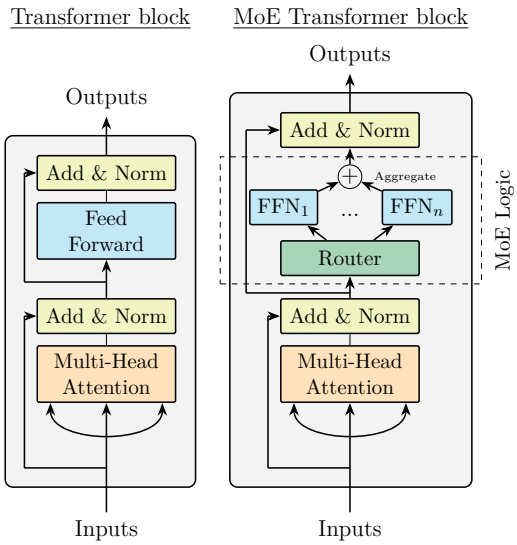

Figure 2.2: MoE Architecture (block view).

with diminishing returns, as evidenced by multiple studies [2], [13], [14]. This trade-off highlights a critical design consideration: beyond a certain point, additional experts yield marginal gains.

A key challenge in MoE training is achieving balanced expert utilization, which is critical for efficient parallelized inference (see Section 4.1). Without intervention, a feedback loop emerges: early-selected experts, exposed to more data initially, optimize their weights faster, making them more likely to be chosen by the router in subsequent iterations. This self-reinforcing cycle skews the gating function toward a few dominant experts, leaving others underutilized. To address this, Eigen et al. [15] applied hard constraints at the start of training, forcing uniform expert selection. In contrast, Bengio et al. [16] used soft constraints, adjusting batch-wise gate averages to encourage balance. Modern approaches favor a soft constraint, incorporating a loss term to penalize imbalance, enabling autograd [17] to optimize the gating function through backpropagation. This sparse, scalable design forms the foundation of transformer-based MoE models, as

detailed in later sections.

2.2.2 Models

Switch Transformer

Building on the sparse computation principles of MoE models, the Switch Transformer, developed by Fedus et al. at Google, adapts the T5 architecture (see Table 2.3) to enhance efficiency and performance [2]. This model modifies T5 by replacing the FFN in every second encoder and decoder layer with an MoE layer, where each FFN is expanded into multiple expert networks—typically 32 in the base configuration. This transformation leverages the scalability of MoE, increasing parameter count without proportional compute cost, as outlined in Section 2.2.1.

The Switch Transformer introduces two distinctive innovations. First, its router—rebranded as a "switch"—selects a single expert per token, choosing the expert with the highest softmax output rather than the top-k approach common in prior MoE designs. Earlier work suggested that routing to multiple experts was necessary for effective learning by comparing their outputs [12], yet Fedus et al. demonstrated that a single-expert switch can train effectively, simplifying token allocation [2]. Second, training is streamlined by retaining only the load-balancing loss, which ensures even expert utilization, while discarding the expert importance loss previously used to weight expert contributions.

These adaptations yield significant gains. Pre-training converges seven times faster than the T5-Base model, and downstream performance improves markedly, with a 6.7-point increase over T5's SuperGLUE score (from 74.6 to 81.3) [2]. By optimizing sparse computation, the Switch Transformer not only advances T5's text-to-text framework

Parameter	Value
vocab_size	32128
dim	768
n_layers	24
n_decode_layers	12
n_moe_layers	12
hidden_dim	3072
n_heads	12
d_kv	64
num_experts	128
expert_size (MB)	18
model_size (MB)	28286.81

Table 2.3: Switch Transformer Model Architecture with 128 experts

but also sets a precedent for efficient, high-capacity models, influencing subsequent MoE designs like those explored in this thesis.

Qwen1.5-MoE-A2.7B

Qwen1.5-MoE-A2.7B, developed by Alibaba's Qwen Team, exemplifies the power of MoE architectures in delivering high-performance language models with minimal computational overhead [18]–[20]. Upcycled from the dense Qwen-1.8B model, this transformer-based decoder-only MoE model comprises 14.3 billion total parameters, with only 2.7 billion activated per token during runtime. Built on the Qwen1.5 Transformer architecture, it incorporates SwiGLU activation [21] for enhanced non-linearity, attention QKV bias for improved attention stability, grouped query attention (GQA) [22] for efficient KV caching, Rotary Positional Embeddings (RoPE) [23] for robust position encoding, sliding window attention, and an adaptive tokenizer optimized for multiple natural languages and code [18]. Supporting a context length of up to 32,768 tokens, Qwen1.5-MoE-A2.7B achieves performance comparable to the denser Qwen1.5-7B while requiring just 25% of the training resources and offering 1.74 times faster inference speed, making it a corner-

Parameter	Value
vocab_size	151936
dim	2048
n_layers	24
n_decode_layers	24
n_moe_layers	24
hidden_dim	1408
n_heads	16
d_kv	128
num_experts	60
expert_size (MB)	33
model_size (MB)	54610.38

Table 2.4: Qwen1.5-MoE-A2.7B architecture

stone of sparse computation strategies explored in this thesis.

Two key innovations define Qwen1.5-MoE-A2.7B's approach to efficient scaling. First, its upcycling process transforms the dense Qwen-1.8B model by replacing FFN layers with MoE layers, each containing multiple specialized experts that are selectively activated via learned routing. This method retains the linguistic knowledge of the dense model while enabling sparse computation, significantly reducing training and inference costs compared to traditional MoE training from scratch. Second, the model benefits from an improved tokenizer, adaptive to diverse natural languages and programming code, which enhances its multilingual and coding capabilities without requiring architectural modifications. These advancements allow Qwen1.5-MoE-A2.7B to support a wide range of tasks, with frameworks like LLaMA-Factory [24] and Axolotl enabling optional post-training via supervised fine-tuning (SFT) and reinforment learning from humand feedback (RLHF) [25] for tailored alignment.

2.2.3 Parallelism

The scalability of MoE models, where parameter count grows linearly with the number of experts, necessitates parallelism to manage computational and memory demands across multiple GPUs. This is particularly critical as models like the Switch Transformer (Section 2.2.2) exceed single-GPU capacity. Four established strategies—data parallelism, model parallelism, tensor parallelism, and expert parallelism—are explored, each tailored to specific scaling challenges.

Data Parallelism

This strategy scales data processing by replicating the model across multiple GPUs, each handling a distinct subset of training data or inference requests [26]. During training, each GPU computes gradients independently, which are then averaged and synchronized across devices to update model weights. In inference, no inter-GPU communication is required, making it a straightforward approach to leverage additional GPUs. Its effectiveness lies in maximizing throughput without altering model architecture or loss functions.

Model Parallelism

When a model exceeds a single GPU's memory capacity, model parallelism distributes its components—such as attention mechanisms, FFNs, or entire layers—across multiple devices [27]. Processing a request requires inter-GPU communication, as outputs from one component (e.g., a layer on GPU 0) become inputs to another (e.g., a layer on GPU 1). This can be refined into pipeline parallelism, where consecutive layers are assigned to distinct GPUs, creating a sequential dependency chain [28]. For instance, a six-layer model on six GPUs assigns one layer per device, reducing memory load but introducing

latency from data transfers.

Tensor Parallelism

Addressing large models differently, tensor parallelism splits individual tensors within layers across GPUs, rather than dividing architectural components [29]. For example, an FFN's weight matrix might be partitioned into four submatrices for a tensor group size of four. Each GPU computes its portion, and results are recombined as if from a single operation. This approach demands frequent communication—more than model parallelism—due to tensor dependencies, but it integrates seamlessly with data or model parallelism for hybrid scaling.

Expert Parallelism

Designed specifically for MoE models, expert parallelism reduces per-GPU memory footprints by distributing experts across devices [30]. As a specialized form of model parallelism, it shards experts while duplicating non-expert components (e.g., attention layers) or splitting them via other parallelism types. During a forward pass, each GPU receives input data—often via data parallelism—and the router assigns tokens to experts (see Figure 2.3). Tokens assigned to experts on remote GPUs, such as token t_1 routed from GPU 0 to expert e_4 on GPU 2, require an all-to-all communication step, implemented as an optimized scatter-gather primitive [31]. Post-execution, results return via a second all-to-all exchange. This process, illustrated in Figure 2.3, balances memory use but incurs latency from cross-device transfers.

Expert placement strategies remain underexplored. Early works assumed one expert per GPU [30], yet with fewer GPUs than experts, co-location becomes essential. A common practice, adopted by systems like DeepSpeed and FastMoE, employs round-

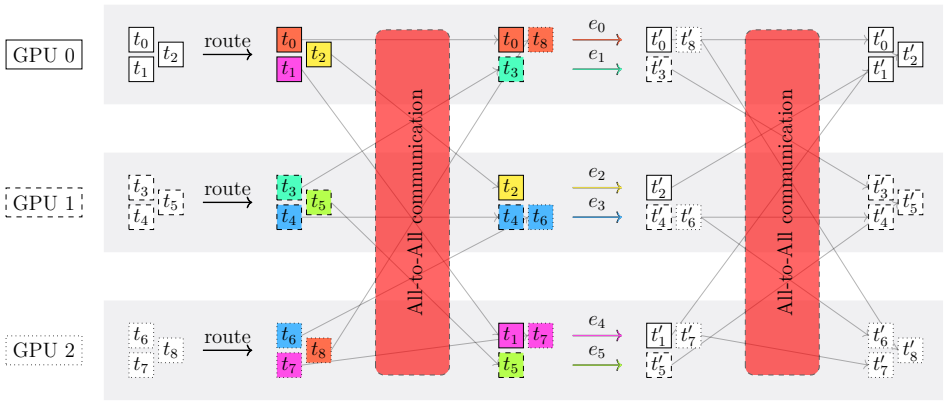

Figure 2.3: Example of Expert Parallelism data movement.

robin assignment [32]–[36]. This router-agnostic approach, while simple, overlooks token distribution imbalances, leading to inefficiencies detailed in Section 4.1. These limitations underscore the need for adaptive scheduling, as addressed by HarMoEny.

Chapter 3

Related Work

The emergence of trillion-parameter MoE models has catalyzed the development of specialized frameworks to manage their complexities: dynamic token routing, load imbalance across experts, and high communication costs in distributed environments. This section examines seven prominent systems—FastMoE, FasterMoE, DeepSpeed, Tutel, ExFlow, SmartMoE, and MegaBlocks—detailing their architectures, optimizations, and performance characteristics. Sections and Figures are cited from the respective paper for ease of access.

3.1 FastMoE: A flexible foundation for MoE execution

FastMoE [33], introduced in a arXiv preprint from Tsinghua University, is one of the first open-source PyTorch framework designed for distributed training of MoE models on GPU clusters. It replaces Transformer feedforward layers with sparse MoE layers, integrating seamlessly with Megatron-LM via a plugin-style module to leverage its DP

and TP (Section 3.1). FastMoE employs EP by distributing experts across GPUs, using NCCL all-to-all communication to dispatch tokens to assigned experts and gather results (Section 3.2). A lightweight gate module, based on a top-k routing strategy (e.g., top-2), dynamically assigns tokens to experts, supported by custom CUDA kernels to optimize sparse computation (Section 4).

FastMoE enhances GPU utilization through a batched execution model, grouping tokens for each expert to improve matrix multiplication efficiency (Section 4, Figure 4). Evaluated on a single NVIDIA V100 GPU, it outperforms a naive PyTorch baseline for an MoE layer with 64 experts, maintaining stable latency as the number of experts increases (Figure 5). On an 8-GPU cluster, FastMoE scales sub-linearly, achieving a throughput increase from 10 TFLOPs to 25 TFLOPs (Figure 6).

3.2 FasterMoE: Optimizing distributed MoE training

FasterMoE [34], presented at PPoPP'22, is a distributed training system from Tsinghua University that extends FastMoE to optimize trillion-scale MoE models on GPU clusters. Its architecture introduces a DDL-Roofline model (Section 3.3) to analyze performance bottlenecks across different distribution strategies, dynamically balancing computation and communication [34]. FasterMoE employs dynamic expert shadowing, replicating high-demand experts across nodes to reduce all-to-all communication costs (Figure 4.1, Figure 6), and an asynchronous scheduling engine that overlaps expert computation with token dispatches using a fine-grained timeline (Section 4.2). The routing mechanism features a congestion-aware gate, adjusting token assignments based on net-

work latency and expert load via a topology-aware selection strategy (Section 4.3).

Built on PyTorch with NCCL, FasterMoE supports hybrid parallelism by partitioning experts across GPUs and nodes, enabling scalable expert counts with hardware resources (Section 2.3). Evaluated on 64 NVIDIA V100 GPUs, FasterMoE achieves a 17.87× end-to-end training speedup over ZeRO stage 3 (Section 5.4). Dynamic shadowing and smart scheduling contribute significantly, achieving up to 1.95× and 1.40× speedups, respectively, over baseline FastMoE (Section 5.5), demonstrating substantial efficiency gains for large-scale MoE training.

3.3 DeepSpeed MoE: Bridging training and inference for scalable MoE

DeepSpeed MoE [32] introduced at ICML 2022, is a comprehensive framework within the DeepSpeed ecosystem, designed to optimize both training and inference for MoE models at trillion-parameter scales. Its architecture leverages a multi-dimensional parallelism strategy, integrating DP, TP, ZeRO [37], and EP to distribute computation across GPU clusters (Section 5.1). A key innovation is the Pyramid-Residual MoE (PR-MoE) structure, which increases the number of experts as you go deeper into the network and passes each token to not just a single expert but to the chosen expert and an MLP, a form of residual connection. For scalability, DeepSpeed MoE employs a hierarchical all-to-all communication approach, splitting token dispatches into smaller, GPU-group-level operations, and fuses computation kernels (e.g., gating function) to minimize overhead (Section 5.1). It also introduces Mixture-of-Students (MoS), distilling MoE knowledge into smaller, inference-ready models, achieving up to 12.5% parameter reduction while

retaining 99.3% the performance of the teacher.

Evaluated on 256 NVIDIA A100 GPUs, it achieves a 7.3× latency improvement over PyTorch baselines [38], serving a 1T-parameter model under 25ms latency.

3.4 Tutel: Adaptive parallelism and pipelining for dynamic MoE

Tutel [35], presented at MLSys 2023, is a full-stack MoE system designed to handle dynamic workloads, such as the 4.38× workload variation observed in SwinV2-MoE–an MoE version of [39]–training (Figure 1). Built atop PyTorch and integrated with Fairseq [40] and DeepSpeed [32], its architecture combines adaptive parallelism and pipelining to optimize both training and inference across distributed GPU clusters. Tutel's adaptive parallelism dynamically switches between DP and a hybrid of EP, DP, and MP, controlled by a tunable parameter r (Section 3).

To optimize communication, Tutel employs adaptive pipelining, overlapping All-to-All operations with expert computation using two algorithms: Linear All-to-All for smaller scales and Two-Dimensional Hierarchical (2DH) All-to-All for larger scales (Section 3.2). These strategies are precomputed in a dictionary for efficient runtime scheduling (Section 3.3), yielding 1% to 107% average throughput improvement over static pipelining for Linear and 11% to 106% for 2DH (Table 6a). Tutel's 2DH All-to-All improves bandwidth utilization and scales better than NCCL's Linear All-to-All at large scales by aggregating small messages, reducing latency overhead compared to point-to-point communication (Appendix A). Fast encode/decode kernels optimize token-to-expert routing, saving 20% to 90% GPU memory (Appendix B). Flexible All-to-All ensures consistent

expert computation efficiency across scales (Figure 11).

Evaluated on Azure clusters with up to 2,048 A100 GPUs, TUTEL achieves 4.96× and 5.75× speedup for a single MoE layer on 16 and 2,048 GPUs, respectively, over Fairseq/DEEPSPEED baselines (Section 5). For SwinV2-MoE, TUTEL delivers 1.55× training and 2.11× inference speedup over Fairseq (Section 5).

3.5 ExFlow: Exploiting inter-layer expert affinity for inference acceleration

EXFLOW [41], developed at The Ohio State University, is a system designed to accelerate GPT-based MoE inference on distributed GPU clusters. Its architecture exploits inter-layer expert affinity (Figure 2), a statistical observation that tokens routed to a specific expert in one MoE layer are likely to select the same expert in subsequent layers. This reduces cross-GPU token movement, a bottleneck in traditional expert parallelism. EXFLOW introduces context-coherent expert parallelism, replacing the conventional all-to-all communication with an AllGather operation to maintain token contexts across GPUs. By grouping tokens with shared expert assignments into context-coherent batches (Figure 5a), it halves the number of all-to-all operations per layer (Figure 5b)—from two (dispatch and gather) to one—while preserving full expert utilization. To optimize expert placement, EXFLOW employs an offline Integer Linear Programming (ILP) solver (Equations 8-12), minimizing cross-GPU routing costs by assigning affinity-correlated experts to the same device. The ILP formulation balances computation load (Equation 9) and communication overhead (Equation 10), solved using Gurobi (Section V.A).

Evaluated on the Wilkes3 cluster, a Tier-2 cluster of A100 GPUs at Cambridge,

ExFlow reduces cross-GPU routing latency by up to 67% and achieves 2.2× higher inference throughput over DeepSpeed MoE for GPT MoE models with 8 to 64 experts (Section V.C). Its affinity-aware design shines in multi-node settings—where communication costs dominate (Figure 9), though AllGather's scaling with GPU count introduces overhead beyond 16 GPUs (Section V.B).

3.6 SmartMoE: Two-Stage hybrid parallelism for sparse MoE training

SmartMoE [42], presented at USENIX ATC 2023, is an automatic parallelization system developed by Tsinghua University to optimize training of MoE models. Its architecture tackles the dynamic, data-sensitive nature of MoE workloads (e.g., varying expert loads in Figure 3) through a two-stage design: offline pool construction and online adaptive parallelization. In the offline stage, SmartMoE constructs a pool of execution plans using a workload-aware performance model that estimates expert selection distributions—such as GShard gate capacity factors ranging from 1.2 to infinity (Figure 6)—without requiring training data. This model predicts computation and communication costs across a hybrid parallelism space, combining DP, TP, PP, EP via an expert slot abstraction (Table 1). The online stage dynamically selects and refines these plans using two algorithms: a lightweight greedy approach (Algorithm 1, $O(NE)$ complexity) for rapid adjustments, and a hybrid dynamic programming method (Algorithm 2, $O(ME + N \times 4^M)$) for more precise optimization. These adjustments occur every 10 iterations (Figure 12), minimizing load imbalance (Equation 1) with a switching overhead of 20ms (Table 4).

Evaluated on clusters like inky (8× A100 GPUs) and blinky (64 GPUs), Smart-

MoE achieves up to 1.88× end-to-end training speedup over FasterMoE for GPT-MoE (4.5B–14B parameters) and Swin-MoE (0.54B–1B parameters) models (Figures 7-8). Its offline model's accuracy ($R^2 > 0.5$, Figure 10) ensures robust plan generation, while online adaptations deliver per-layer gains of up to 1.43× (Figure 11). Ablation studies show its enlarged parallelism space yields 2.67× speedup over static plans (Figure 9), highlighting its ability to capture diverse execution strategies ignored by prior systems. SmartMoE excels in training efficiency, but its reliance on offline preprocessing and periodic switching (20ms overhead, Table 4) limits its suitability for inference, where latency is critical.

3.7 MegaBlocks: Block-Sparse dropless MoE training

MegaBlocks [43], presented at MLSys 2023, is a GPU-optimized system that reformulates MoE training as block-sparse operations to eliminate token dropping and padding overheads inherent in prior frameworks like Tutel. Built atop Megatron-LM, its architecture constructs a large block-diagonal matrix per device (Figure 3), where experts are grouped into variable-sized blocks. This enables parallel expert computation using custom block-sparse GPU kernels—extending CUTLASS with a hybrid blocked-CSR-COO format (Figure 6)—achieving 98.6% of cuBLAS throughput (Figure 9). Unlike traditional approaches requiring serial execution or token dropping for batched matrix multiplication, MegaBlocks avoids both, supporting dynamic, load-imbalanced token assignments efficiently. Evaluated on Transformer MoEs with 64 experts (Table 2), it delivers up to 4.35× end-to-end training speedup over Tutel's padding-based dMoEs

and 2.4× over dense Megatron-LM Transformers on 8 A100 GPUs (Figure 7).

3.8 Comparative analysis and HarMoEny 's contribution

FastMoE offers foundational flexibility, FasterMoE optimizes training scalability, DeepSpeed MoE bridges training and inference, Tutel adapts dynamically, ExFlow enhances inference communication, SmartMoE refines training parallelism, and MegaBlocks eliminates token dropping. None fully address real-time inference needs. HarMoEny integrates their strengths—flexibility, modeling, parallelism, adaptivity, affinity, workload awareness—with dynamic scheduling and prefetching.

HarMoEny's approach to addressing load imbalance in MoE inference through dynamic token redistribution and asynchronous expert prefetching is largely orthogonal to several techniques discussed in this chapter, enabling potential integration for mutual performance benefits. For instance, systems like FastMoE (Section 3.1), FasterMoE (Section 3.2), and DeepSpeed MoE (Section 3.3) could incorporate HarMoEny's load-aware token scheduling and prefetching mechanisms to mitigate the straggler effects caused by uneven expert utilization, as these systems primarily focus on optimized kernels and static expert placement without dynamic rebalancing. Conversely, HarMoEny could benefit from adopting the specialized CUDA kernels employed by FastMoE and FasterMoE, which enhance computational efficiency, or DeepSpeed 's memory optimization strategies, potentially reducing HarMoEny's scheduling overhead, which currently accounts for 20-30% of layer latency (Section 6.2). Additionally, SmartMoE's two-stage hybrid parallelism (Section 3.6) could leverage HarMoEny's dynamic load

balancing to further improve its sparse model training efficiency under varying workloads. Similarly, MegaBlocks' block-sparse training approach (Section 3.7) is complementary, as HarMoEny could integrate its efficient sparse computation framework to enhance inference throughput, particularly in scenarios with high expert inequality. This orthogonality underscores HarMoEny's versatility, allowing it to augment existing MoE frameworks while also benefiting from their optimizations, fostering a synergistic improvement in inference performance.

Chapter 4

Methodology

4.1 Expert parallelism load-imbalance

Expert parallelism, as outlined in Section 2.2.3, distributes MoE experts across GPUs to manage memory constraints, yet its effectiveness hinges on balanced token allocation. Despite training regimens incorporating a load-balancing loss to promote even token distribution across experts [2], inference reveals persistent imbalances. A technical report by NVIDIA highlights this issue, noting that some experts consistently receive more tokens than others, even with diverse datasets [44]. Figure 4.1 illustrates this across MMLU topics, where the most popular expert garners 40–60% more tokens than the least popular, relative to a uniform distribution. Prompt specificity exacerbates this skew, as shown in Figure 4.2, where Layer 31's token distribution becomes markedly uneven, suggesting input sensitivity outstrips training regularization.

This phenomenon persists in advanced models like the Switch Transformer [2]. Analysis on the BookCorpus dataset (Figure 4.3) reveals Layer 7's most popular expert receiving 26.3 times more tokens than its least popular counterpart—an imbalance of

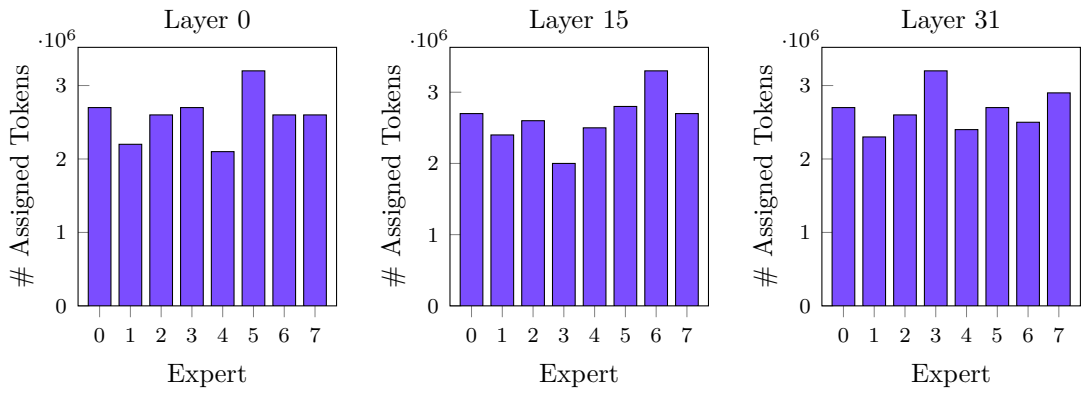

Figure 4.1: Expert token distribution over all Measuring Massive Multitask Language Understanding (MMLU) topics. Reproduction of [44].

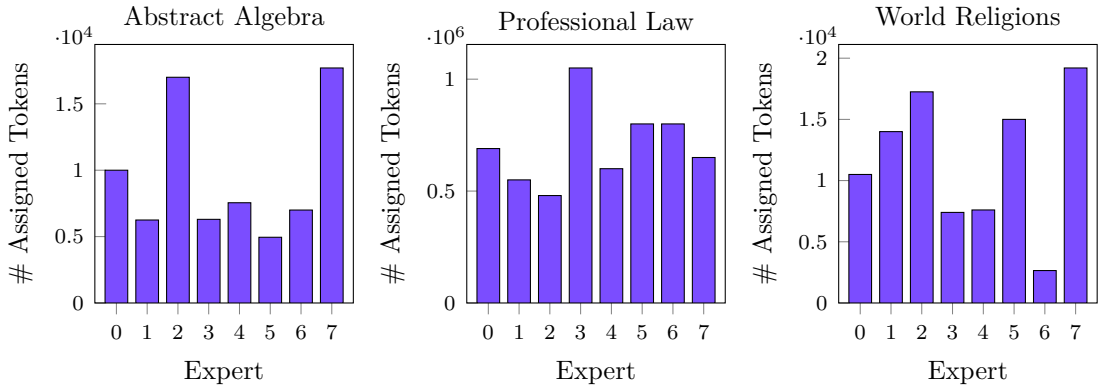

Figure 4.2: Expert token distribution at layer 31. Reproduction of [44].

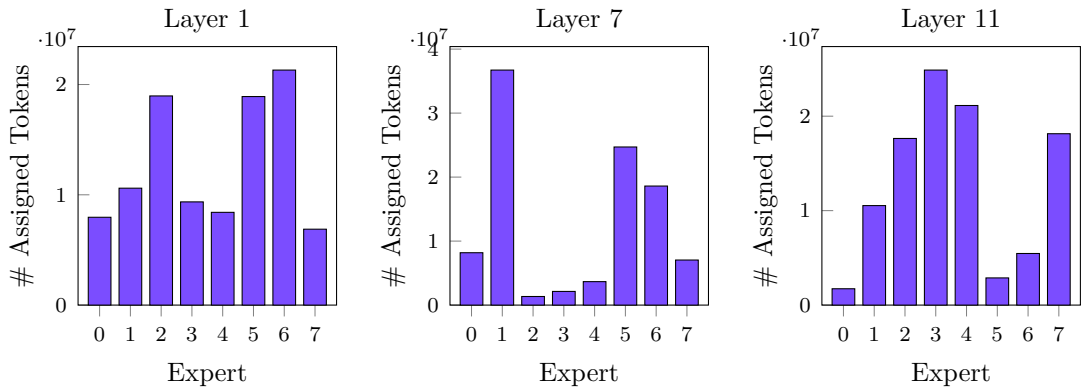

Figure 4.3: Expert token distribution of Switch Transformer with 8 experts on BookCorpus dataset.

2,634%. Scaling to 128 experts amplifies this disparity (Figure 4.4). Excluding Layers 7 and 11, where the least popular experts receive near-zero tokens, Layer 1's most popular expert processes 1.1 million times more tokens than its least popular peer, indicating that imbalance intensifies with expert count.

Similar trends emerge with Qwen1.5-MoE-A2.7B. Figure 4.5 shows Layer 0's expert 25 receiving 180,000 times more tokens than expert 5 on BookCorpus, with only the five most and least popular experts plotted for brevity (full data in Appendix A). Cumulative distribution functions (CDFs) further quantify this skew. For the Switch Transformer with 128 experts, Figure 4.6 reveals that four experts (3% of 128) handle over 50% of Layer 7's tokens. Likewise, Figure 4.7 shows two experts (3% of 60) in Qwen1.5-MoE-A2.7B's Layer 23 managing over 50%, underscoring a consistent concentration of workload.

Such expert-level imbalance translates to GPU-level disparities under expert parallelism. Round-robin expert placement, widely adopted by systems like DeepSpeed, FastMoE, and FasterMoE [32]–[34], assigns experts agnostically, ignoring token distribution. Figures 4.8 and 4.9 (CDFs of GPU token loads) demonstrate this consequence:

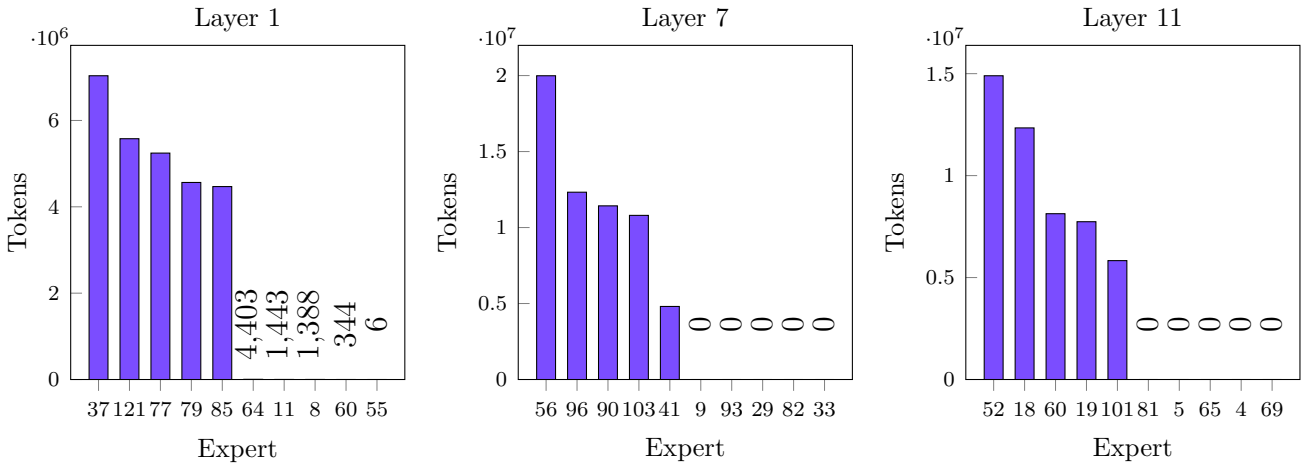

Figure 4.4: Expert token distribution of Switch Transformer with 128 experts on BookCorpus dataset. 5 most and 5 least popular experts shown, ordered by popularity.

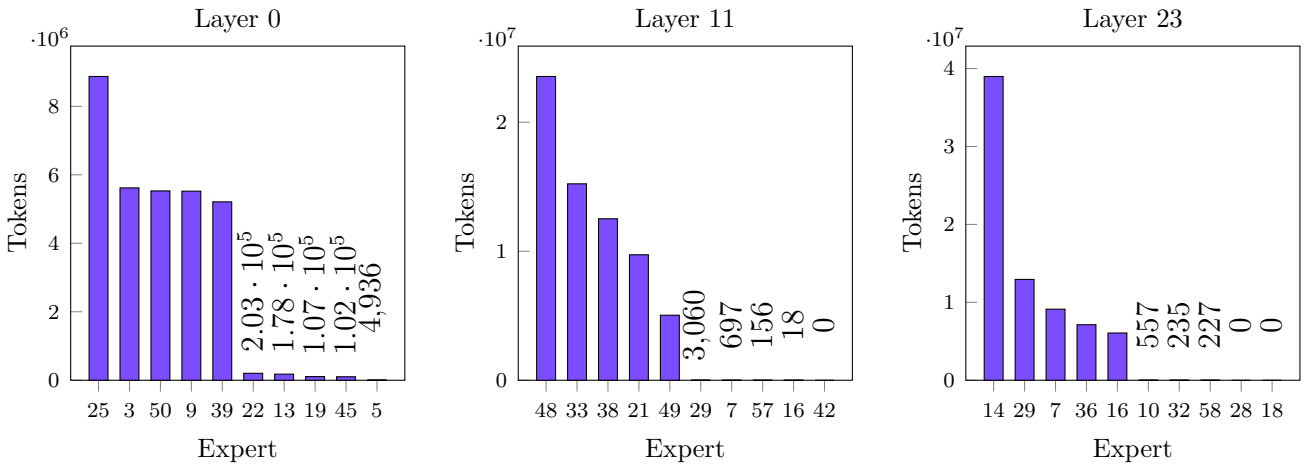

Figure 4.5: Expert token distribution of Qwen1.5-MoE-A2.7B on BookCorpus dataset. 5 most and 5 least popular experts shown, ordered by popularity.

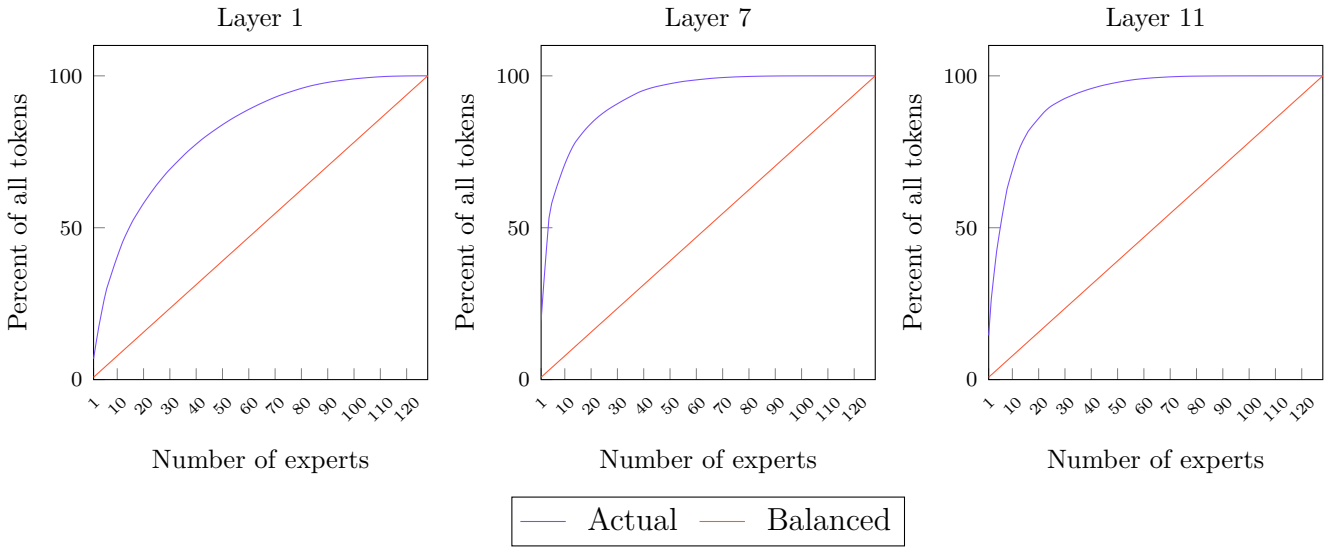

Figure 4.6: CDF of expert token distribution of Switch Transformer with 128 experts on BookCorpus dataset. Tokens increasingly converge in a select few experts at higher layers.

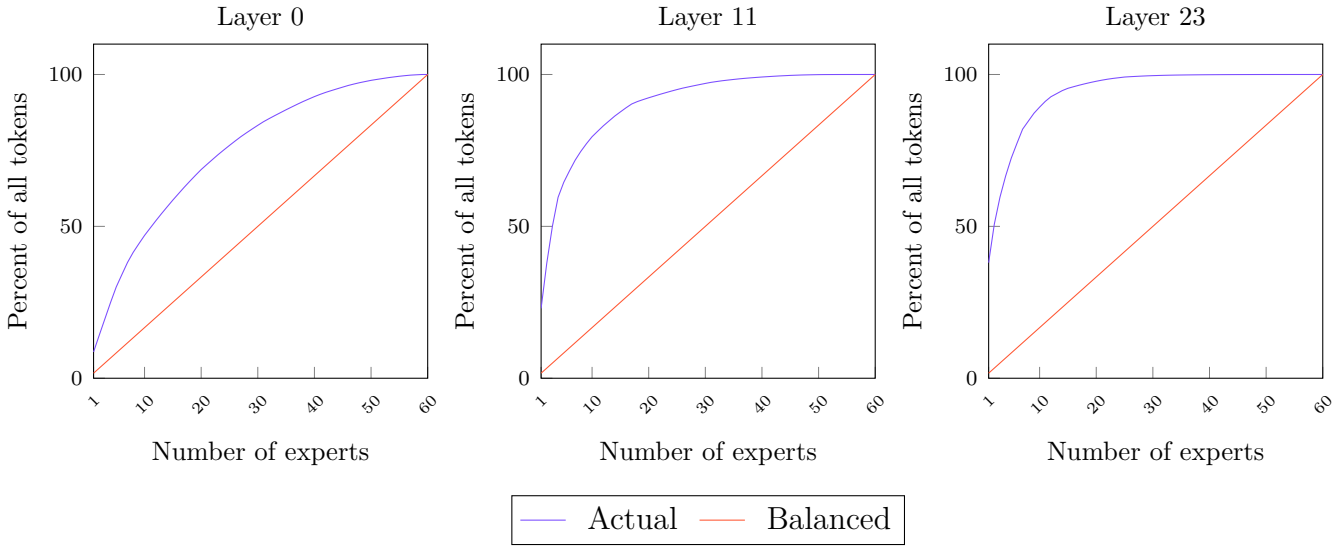

Figure 4.7: CDF of expert token distribution of Qwen1.5-MoE-A2.7B on BookCorpus dataset. Similarly, tokens increasingly converge in a select few experts at higher layers.

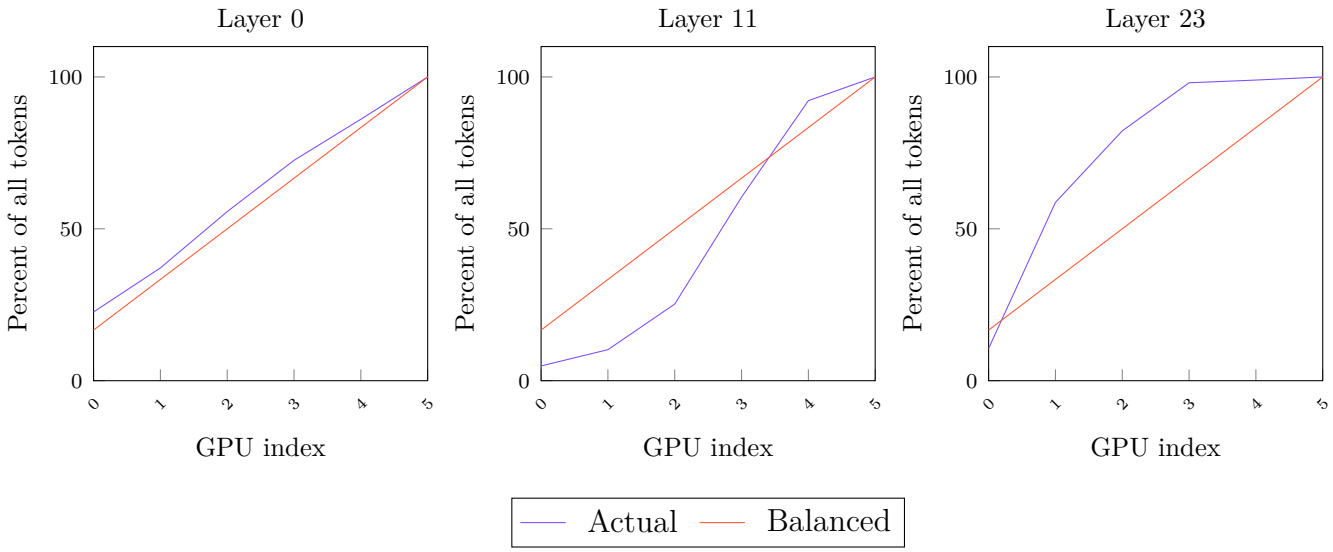

Figure 4.8: CDF of GPU token distribution of Switch Transformer with 128 experts on BookCorpus dataset.

GPUs hosting popular experts face disproportionate workloads, while others idle. Figure 4.10 further illustrates that static placement causes throughput fluctuations and prolonged waiting times under imbalanced requests. This inefficiency, rooted in router skew and naive allocation, motivates HarMoEny 's dynamic scheduling approach (Section 4.3).

4.2 HarMoEny

The GPU-level load imbalances identified in Section 4.1, as illustrated by Figures 4.8, 4.9, 4.10, degrade MoE inference throughput due to prolonged idle times under static expert placement. To address this, HarMoEny is introduced—an inference system optimizing expert parallelism across multi-GPU setups. HarMoEny integrates two key techniques: *(i)* a load-aware scheduler redistributing tokens to balance GPU workloads,

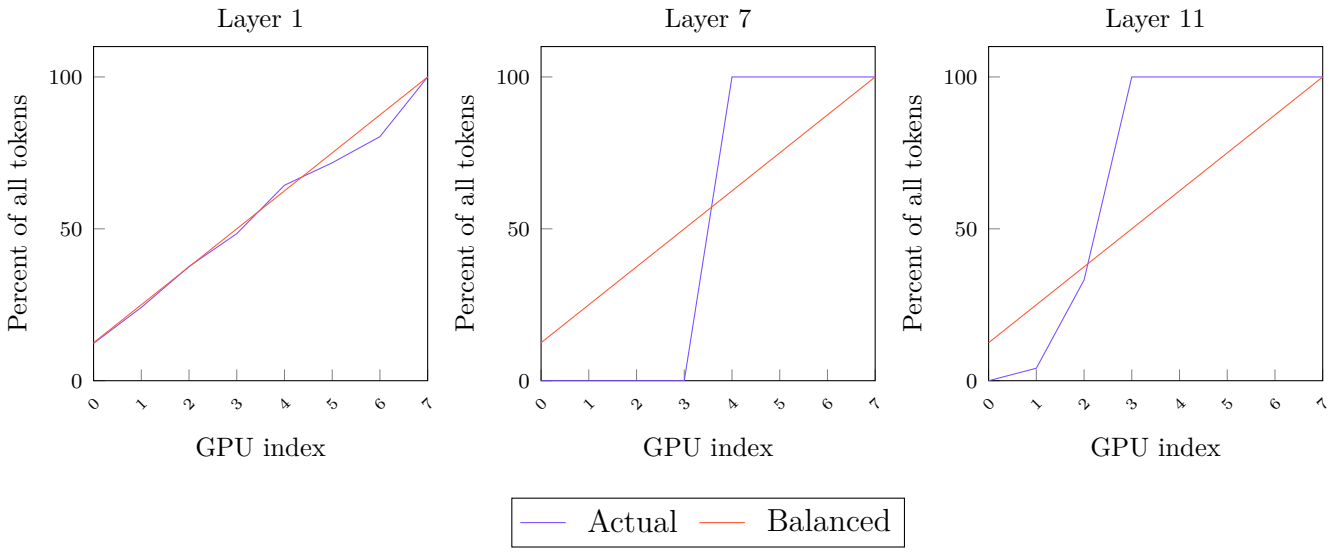

Figure 4.9: CDF of GPU token distribution of Qwen1.5-MoE-A2.7B on BookCorpus dataset.

and *(ii)* an asynchronous expert prefetching protocol transferring experts from DRAM to GPU memory with minimal overhead. These mechanisms mitigate idle time without requiring online profiling, enabling adaptation to dynamic workload shifts while preserving throughput, as validated in Chapter 6.

A forward pass through HarMoEny 's MoE layer is detailed in Algorithm 1, executed on each GPU with input tensors x of shape [$batch\_size$, $sequence\_length$, $hidden\_size$]. These inputs, preprocessed by attention layers in a transformer-based MoE architecture (Section 2.2.1), are processed as follows:

Step 1: Token Routing. The router assigns each token in x to an expert, producing m_{expert}, a tensor mapping tokens to their designated experts (Line 4). This mirrors standard MoE routing (Section 2.2.1), setting the stage for load-aware adjustments.

Step 2: Metadata Exchange. Each GPU computes an expert frequency distribution from m_{expert}, counting token assignments per expert. This metadata, broadcast to all

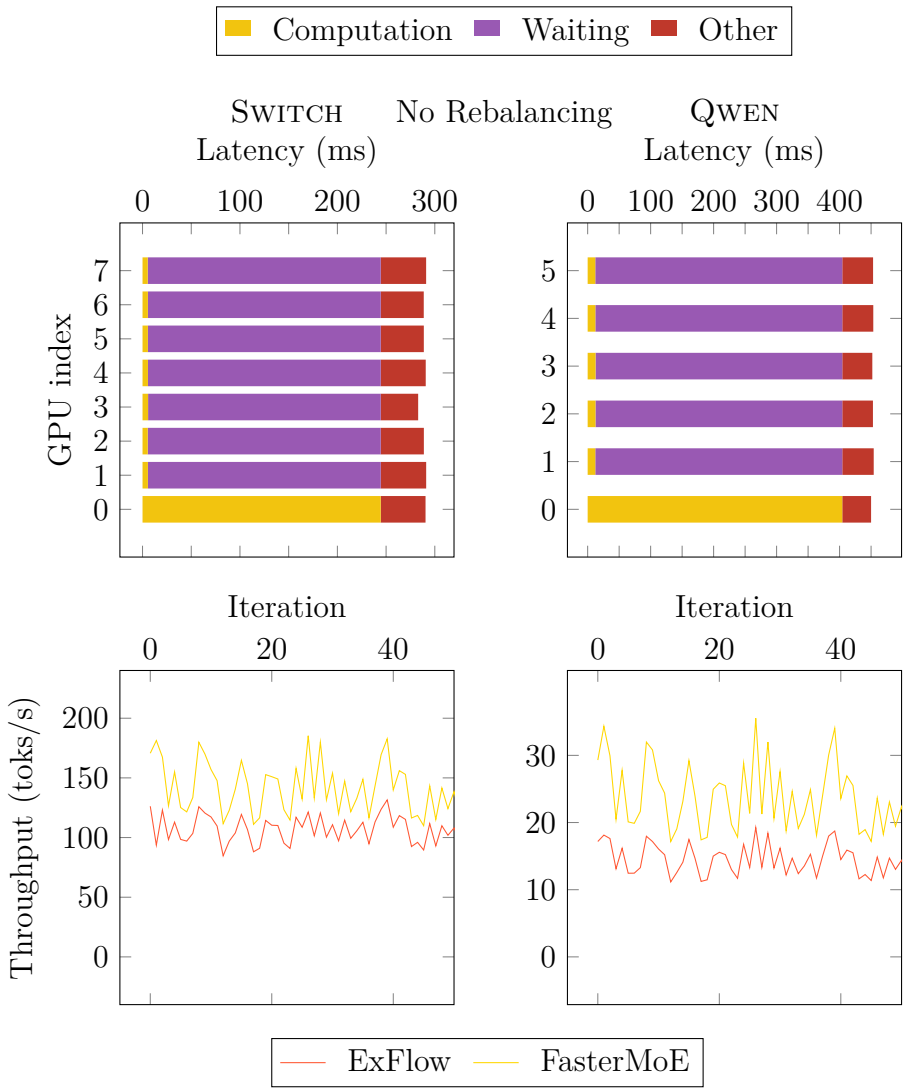

Figure 4.10: *(top)* Fixed expert placement causes long waiting due to load imbalance. *(bottom)* Throughput fluctuates given static placement given nature of requests.

Algorithm 1 HarMoEny MoE layer

require: G: set of GPUs
function FORWARD(x)
    // **Step 1: token routing**
    $m_{expert} \leftarrow$ ROUTER(x)

    // **Step 2: metadata exchange**
    sendMetadataToGPUs(count(m_{expert}))
    receive $m_{all}[i]$ from each GPU $i \in G$

    // **Step 3: token scheduling**
    $S_{initial} \leftarrow$ initialAssign(m_{all})
    $S \leftarrow$ REBALANCE($S_{initial}$)

    // **Step 4: scatter and gather tokens**
    sendTokensToGPUs(x, m_{expert}, S)
    receive $x'[i]$ from each GPU $i \in G$

    // **Step 5: expert processing**
    $x'' \leftarrow$ EXPERTS(x')

    // **Step 6: gather tokens**
    sendTokensBackToGPUs(x'', S)
    receive $y[i]$ from each GPU $i \in G$
    $x \leftarrow$ RECONSTRUCT(y, m_{expert}, S)
end function

GPUs via an all-to-all primitive, totals $4 \times |E| \times |G|$ bytes (assuming 32-bit integers), where $|E|$ is the number of experts and $|G|$ is the number of GPUs. The resulting global distribution, m_{all} , aggregates frequencies across GPUs. Unlike training-focused systems (e.g., DeepSpeed [32]), this step enables global scheduling, with negligible communication overhead offset by efficiency gains (Section 6.2).

Step 3: Token Scheduling. Using m_{all}, an initial schedule S_{initial} assigns tokens to their resident GPUs (Line 11). Represented as a tensor of shape $[|G|, |E|, |G|]$; $S[i, e, j]$ denotes the number of tokens sent from GPU i to GPU j for expert e. Given Section 4.1's imbalance findings, S_{initial} is rebalanced via Algorithm 2 (Section 4.3), redistributing tokens from overutilized to underutilized GPUs (Line 12).

Step 4: Scatter and Gather Tokens. The balanced schedule S governs token distribution, with GPU i sending tokens to GPU j per $S[i, e, j]$ for all experts e, guided by m_{expert} (Line 15). An all-to-all communication aggregates received tokens into x' on each GPU (Line 16), preparing them for processing.

Step 5: Expert Processing. Tokens in x' are executed by their assigned experts, yielding x'' (Line 19). Rebalancing may require loading non-resident experts from DRAM, addressed by asynchronous prefetching (Section 4.4), which overlaps transfers with computation to minimize stalls.

Step 6: Gather Tokens. Processed tokens x'' are returned to their originating GPUs via a second all-to-all communication, stored as $y[i]$ per GPU i. Reconstruction leverages S and m_{expert} to restore the output x (Line 24), completing the pass.

HarMoEny 's scheduler (Section 4.3) and prefetching protocol (Section 4.4) synergistically address load imbalance and latency, respectively, outperforming static SOTA approaches (e.g., FastMoE [33]) under skewed workloads, as evidenced in Chapter 6.

4.3 Load-aware token scheduler

HarMoEny 's load-aware token scheduler is a core component designed to mitigate the GPU-level load imbalances identified in Section 4.1, where uneven token distributions lead to prolonged idle times under static expert placement. Operating in polynomial time, this scheduler dynamically redistributes tokens from overutilized to underutilized GPUs, minimizing idle time and enhancing inference throughput in multi-GPU MoE systems. Unlike static round-robin approaches (e.g., DeepSpeed [32], FastMoE [33]), which ignore runtime token skew, HarMoEny adapts to workload variations, leveraging global metadata from Algorithm 1 (Section 4.2) to achieve near-optimal balance.

The scheduler refines the initial schedule S_{initial} , a tensor of shape $[|G|, |E|, |G|]$ where $S[i, e, j]$ denotes tokens sent from GPU i to GPU j for expert e (Section 4.2). It begins by computing t_{avg}, the average token load per GPU, and t_g, the current load per GPU, summing over sending GPUs and experts. Iterations continue while any GPU's load exceeds t_{avg} (Line 6), redistributing tokens as follows:

1. **Identify Overload**: The most utilized GPU, $g_{\max}$, is found via the maximum of t_g. The GPU sending the most tokens to $g_{\max}$, g_{from} , and its most active expert, $e_{\max}$, are determined by successive summations and maxima (Lines 8, 9). The number of movable tokens, t_{move} , is extracted from S.

2. **Check Threshold**: If t_{move} falls below q, rebalancing halts to avoid inefficient transfers (Line 12), with q's derivation detailed in Section 4.5.

3. **Select Target**: The least utilized GPU, $g_{\min}$, is identified. Rebalancing stops if $g_{\min} = g_{\max}$ or adding q tokens to $g_{\min}$ exceeds t_{avg} , ensuring feasibility.

4. **Redistribute Tokens**: The number of tokens to move, t_s, is the minimum of t_{move}

Algorithm 2 HarMoEny token rebalancing

require: G: set of GPU, E: set of experts, q: token transfer threshold
function REBALANCE($S_{initial}$)
    $S \leftarrow S_{initial}$
    $t_{avg} \leftarrow \lfloor S.\text{SUM}()/|G| \rfloor$ ▷ Avg tokens per GPU
    $t_g \leftarrow S.\text{SUM}(\text{dim}=(0,1))$ ▷ Token count per GPU for processing
    while ANY($t_g > t_{avg}$) **do**
        $g_{max} \leftarrow \text{ARGMAX}(t_g)$
        $g_{from} \leftarrow \text{ARGMAX}(\text{SUM}(S[:,:,g_{max}],\text{dim}=1))$
        $e_{max} \leftarrow \text{ARGMAX}(S[g_{from},:,g_{max}])$

        $t_{move} \leftarrow S[g_{from},e_{max},g_{max}]$
        if $t_{move} < q$ **then**
            return S ▷ Insufficient tokens to move
        end if

        $g_{min} \leftarrow \text{ARGMIN}(t_g)$ ▷ Find least utilized GPU
        if $g_{min} = g_{max}$ **or** $t_g[g_{min}] + q > t_{avg}$ **then**
            return S ▷ No feasible transfer possible
        end if

        $t_s \leftarrow \min(t_{move}, t_{avg} - t_g[g_{min}])$
        $S[g_{from},e_{max},g_{max}]$ -= t_s
        $S[g_{from},e_{max},g_{min}]$ += t_s
        $t_g[g_{max}]$ -= t_s
        $t_g[g_{min}]$ += t_s
    end while
end function

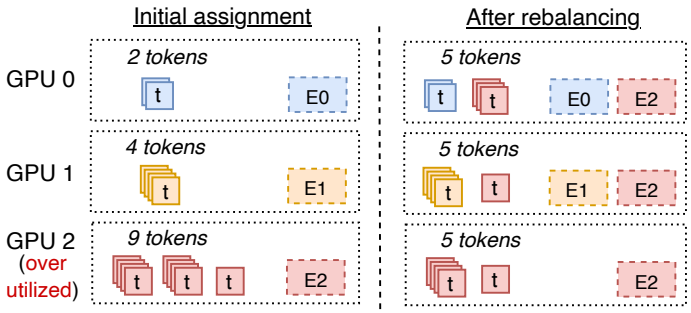

Figure 4.11: Scheduling example with offloading

and the capacity of $g_{\min}$ relative to t_{avg} . S and t_g are updated by subtracting t_s from $g_{\max}$'s load and adding it to $g_{\min}$'s for expert $e_{\max}$.

Figure 4.11 exemplifies HarMoEny 's rebalancing efforts within its load-aware token scheduler. Following token routing and metadata exchange across GPUs (Section 4.2), the initial schedule S_{initial} maps each token to the GPU hosting its assigned expert, as determined by the router. In this scenario, visualized in Figure 4.11, tokens are distributed unevenly: GPU 2 is assigned 9 tokens for expert e_2, exceeding the average of 5 tokens per GPU (computed as $t_{\text{avg}} = \lfloor S.\text{SUM}()/|G| \rfloor$), while GPU 0 and GPU 1 hold 2 and 4 tokens, respectively, for distinct experts. To address this disparity, Algorithm 2 identifies GPU 0 as the least utilized ($g_{\min}$, with $t_g[g_{\min}] = 2$) and calculates its capacity to accept 3 additional tokens without surpassing t_{avg}. Assuming a token transfer threshold $q = 0$, permitting even minimal reassignments, 3 tokens from GPU 2's e_2 workload are transferred to GPU 0. The while loop then reevaluates t_g: GPU 2 retains 6 tokens, still above t_{avg}, prompting a second iteration. GPU 1, now the least utilized with 4 tokens, can accommodate 1 more; thus, 1 token moves from GPU 2 to GPU 1. A final assessment confirms balance—each GPU processes 5 tokens—terminating the loop. Consequently, GPUs 0 and 1 must execute e_2 alongside their resident experts. This extra expert, e_2 must be fetched from DRAM and this loading process, critical to minimizing idle time

post-rebalancing, is detailed in Section 4.4.

4.4 Asynchronous expert prefetching

The GPU idle times induced by load imbalances (Section 4.1) are exacerbated under static expert placement, where tokens must be routed to GPUs hosting their assigned experts (Section 2.2.1). HarMoEny's load-aware scheduler (Section 4.3) inverts this by redistributing tokens across GPUs via Algorithm 2, necessitating the dynamic transfer of expert weights from DRAM to GPU memory at inference time. Given expert sizes—18 MB Switch Transformer and 33 MB Qwen1.5-MoE-A2.7B—synchronous fetching would stall computation, as PCIe bandwidth limits transfer rates. To address this, HarMoEny introduces an asynchronous expert prefetching protocol, overlapping weight transfers with token execution to minimize idle time, bringing total idle time to near 0% (Section 6.2).

This protocol exploits the availability of resident experts—those already in GPU memory—to mask transfer latency. For each MoE layer, every GPU executes a four-step process guided by the balanced schedule S from Algorithm 2, a tensor of shape $[|G|, |E|, |G|]$ where $S[i, e, j]$ denotes tokens sent from GPU i to GPU j for expert e:

1. **Identify**. Post-rebalancing, GPU i inspects $S[:, :, i]$, the column representing tokens it must process, and identifies required experts. These are compared against the GPU's current expert cache. Missing experts are queued in a fetch list, ensuring only necessary transfers are initiated. Other experts are added to the execution list with the fetch list concatenated at the end.

2. **Fetch**. When an expert finishes executing on the execution list e_e, an asynchronous transfer from DRAM to GPU memory is launched using CUDA streams to bring

in the weights of the next expert in the fetch list, e_{fetch}, enabling non-blocking operation. Concurrently, the next expert is executed, e_{e+1}, while e_{fetch} is retrieved from DRAM, leveraging the GPU's parallel compute capabilities to overlap data movement with execution.

3. **Execute**. Before executing e_e, the GPU verifies if e_e's weights are already loaded, and if not then waits. Once loaded, the expert executes the tokens corresponding to $S[:, e, i]$.

4. **Cleanup**. Once finished executing, all resident experts that were evicted are loaded back into the GPU memory for the next iteration.

To manage GPU memory constraints (e.g., 32 GB per V100, Section 5.5), HarMoEny maintains a per-GPU expert cache of capacity c, a tunable hyperparameter. When the cache is full, the most recently used (MRU) expert is overwritten with the new expert weights. Overwriting reduces expert loading latency by 5.5x based on a microbenchmark that reduced expert loading from 11 ms to 2 ms. Figure 4.12 illustrates how the asynchronous fetching reduces computational stalling. It is necessary for $c >= 2$ otherwise the protocol executes serially.

The protocol synergizes with Algorithm 2 by enabling token redistribution without stalling computation, amortizing DRAM latency (quantified in Section 6.2). As shown in Section 6.2, this reduces layer latency by 9–17% over synchronous fetching, scaling with expert size and imbalance severity, thus enhancing HarMoEny 's throughput under dynamic workloads.

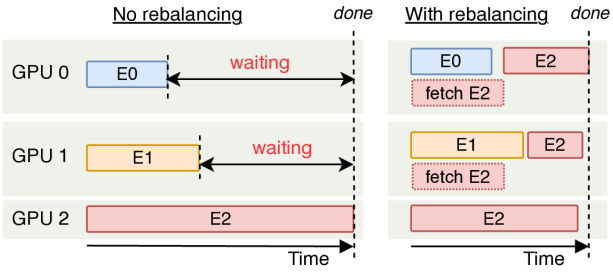

Figure 4.12: Execution timeline with (left) no token distribution and (right) token distribution with asynchronous fetching

4.5 Determining token threshold q

HarMoEny 's load-aware scheduler (Section 4.3) relies on the token threshold q to determine when to redistribute tokens across GPUs, as per Algorithm 2, when fetching an expert from DRAM rather than routing tokens to the expert's resident GPU is justified. This threshold, a critical hyperparameter, ensures that the asynchronous prefetching protocol (Section 4.4) overlaps computation efficiently, balancing the cost of expert transfers against load rebalancing benefits. A low q triggers excessive fetching, where transfer latency exceeds execution time, while a high q restricts rebalancing opportunities, preserving imbalances identified in Section 4.1. To establish a practical lower bound, q is derived based on expert size, GPU compute capacity, and interconnect bandwidth, with full details in Appendix B.

The design principle is to ensure that processing q tokens on a fetched expert takes at least as long as transferring its weights from DRAM to GPU memory, enabling effective overlap in Step 2 of Section 4.4. Experts are modeled as two-layer MLPs—consistent with feed-forward networks in transformer-based MoE models like Switch Transformer (Section 2.2.2, hidden_dim $= 3072$)—with layers of sizes $m \times p$ and $p \times m$, computed as xW^1W^2 for input x of shape $[q, m]$. The derivation compares FLOPs for q tokens against

transfer time, yielding Equation 4.1:

$$q > \frac{\phi \cdot d_{type}}{2\beta} \tag{4.1}$$

Here, ϕ is GPU FLOPS (e.g., 14 TFLOPS for V100, Section 5.5), β is PCIe bandwidth (e.g., 16 GB/s for DGX-1), and d_{type} is bytes per parameter (e.g., 4 for FP32). The FLOPs term, $qp(2m-1) + qm(2p-1)$, accounts for matrix multiplications: xW^1 $(q \times m \times p)$ and $(xW^1)W^2$ $(q \times p \times m)$, approximated as $4qpm$ by dropping lower-order terms (e.g., $-q(p+m)$). Expert size, $(mp + pm)d_{\text{type}}$, reflects weights of both layers, simplified to $2pmd_{\text{type}}$. For DGX-1 with FP32 parameters:

$$q > \frac{14 \times 10^{12} \cdot 4}{2 \cdot 16 \times 10^{9}} = 1750 \text{ tokens}$$

4.6 Artificial imbalance generation

Real-world MoE inference exhibits significant expert imbalance (Section 3.1), with CDFs showing 3% of experts handling over 50% of tokens in models like Switch Transformer (Figure 4.6). To evaluate HarMoEny under controlled conditions without relying on the natural imbalance of the router, an artificial imbalance generation method is introduced, leveraging the Gini Index [45]. Originally a measure of wealth inequality, the Gini Index is repurposed to quantify token distribution skew across experts within an MoE layer. A Gini Index (G) of 0 denotes perfect equality (each expert receives identical token counts), while $G = 1$ indicates perfect inequality (one expert processes all tokens). By tuning G, this method generates controlled, variable imbalances to assess HarMoEny 's

load-aware scheduler (Section 4.3) and asynchronous prefetching protocol (Section 4.4), complementing real-world dataset tests (Section 5.4).

The method designates ρ experts as overloaded, each assigned $\hat{N}$ tokens, while the remaining $\epsilon - \rho$ experts receive N tokens, where $\hat{N} > N$ and ϵ is the total expert count. The total tokens processed by the layer, χ, is:

$$\chi = \rho\hat{N} + (\epsilon - \rho)N$$

For a target Gini Index G, token assignments are derived from the Gini formula (Appendix C), yielding Equation 4.2:

$$\begin{aligned} \hat{N} &= \frac{\epsilon\chi G + \rho\chi}{\rho(\epsilon - 2\rho)} \\ N &= \frac{\chi - \rho\hat{N}}{\epsilon - \rho} \end{aligned} \tag{4.2}$$

Here, ρ (where $0 < \rho < \frac{\epsilon}{2}$) controls the number of overloaded experts, ensuring the denominator $\epsilon - 2\rho$ remains positive and skew is feasible. For example, with $\epsilon = 128$, $\rho = 10$, $\chi = 10,000$, and $G = 0.5$, $\hat{N} \approx 685$ and $N \approx 27$, amplifying imbalance predictably. This distribution is enforced by overriding the router's learned output $G(x)$ (Section 2.2.1) during testing, assigning synthetic weights to achieve the desired $\hat{N}$ and N, as trained routers do not yield predictable consistent skew for each input request.

Chapter 5

Experimental setup

5.1 HarMoEny

HarMoEny is implemented in 1,115 lines of Python code within the PyTorch framework. The load-aware scheduler operates entirely in Python without custom kernels, while expert fetching leverages a dedicated NVIDIA CUDA stream for GPU-host communication. Computation occurs on the main CUDA stream, synchronized using CUDA events to track two critical milestones: (1) completion of expert i's execution and (2) completion of expert i's loading into GPU memory. These events prevent race conditions in the independent streams, ensuring experts are neither overwritten prematurely nor executed with incorrect weights. The MoE layer is encapsulated as a PyTorch `nn.Module`, optimized for expert parallelism. Designed for modularity, HarMoEny integrates seamlessly with any PyTorch-based MoE model via an injector, simplifying its adoption (see Appendix D for setup details).

5.2 MoE system baselines

HarMoEny is evaluated against four SOTA baselines: DeepSpeed, FastMoE, FasterMoE, and ExFlow.

DeepSpeed is a framework for distributed training and inference of machine learning models [32]. It integrates Tutel [35] to enhance inference performance. For a fair comparison, token dropping is disabled, as its implementation is faulty and causes program hangs [46]. Consequently, the evaluation capacity factor is set to accommodate the worst-case imbalance scenario and adjusted per dataset. Expert parallelism is enabled with a static round-robin expert placement. Modern variants like DeepSpeed-MII and vLLM, which lack support for the Volta architecture (e.g., V100 GPUs) and expert parallelism [47], are excluded from this study.

FastMoE pioneered expert parallelism for MoE models, supporting multiple experts per GPU [33]. This flexible system accommodates any model with appropriate configuration, allowing users to define gating and expert networks. Built with custom kernels, FastMoE optimizes decision-making and data movement, assigning experts to GPUs via a round-robin strategy.

FasterMoE extends FastMoE with load-balancing enhancements, including dynamic shadowing and fine-grained scheduling [34]. For overutilized experts, it broadcasts expert weights to all GPUs for local execution, avoiding token transfers to a single GPU. It also introduces a topology-aware gating function to prioritize local experts, reducing network congestion. A congestion-avoiding expert selection, designed for training, is omitted here due to its adverse impact on inference performance.

ExFlow mitigates load imbalance by leveraging inter-layer expert affinity [41]. By tracing token patterns across consecutive MoE layers, it optimizes expert placement us-

ing offline integer programming. This process, while adaptable to online updates (e.g., every few minutes), incurs significant overhead: 8.5 minutes per batch for Switch Transformer and 45 minutes for Qwen1.5-MoE-A2.7B on a CPU. Given this cost, we implement ExFlow within HarMoEny using its provided code, performing the affinity-based mapping offline using a trace of BookCorpus.

5.3 MoE models

HarMoEny is evaluated on two distinct MoE models to demonstrate its versatility: Switch Transformer [2] and Qwen1.5-MoE-A2.7B [18]. The Switch Transformer, built upon the T5 architecture [48], replaces each feed-forward network (FFN) with an MoE layer by replicating the FFN across multiple experts and incorporating a learned router. It comprises 12 encoder and 12 decoder blocks, with every second block featuring an MoE layer, totaling 12 MoE-enabled transformer blocks. In this study, each MoE layer contains 128 experts; we denote this configuration as Switch. Variants with 8, 16, 32, or 64 experts exist but are not evaluated here. Qwen, developed by Alibaba Cloud, is a series of transformer-based language models for diverse tasks. We use the Qwen1.5-MoE-A2.7B variant, which includes 24 transformer blocks, each an MoE layer with 60 experts, referred to as Qwen.

5.4 Datasets

Beyond the artificial Gini-Index workload (Section 4.6), three established datasets are employed to reflect real-world scenarios: (1) BookCorpus, a large-scale collection of up to 7,185 unique books sourced from smashwords.com [49]; (2) Wikitext, comprising

over 100 million tokens extracted from Wikipedia's *Good* and *Featured* articles [50]; and (3) WMT19, a dataset of translation pairs, with the German-to-English subset (34.8 million samples) selected for this study [51].

Additionally, a synthetic dataset, RANDOM, is introduced to model uniform token distribution. It is generated online by concatenating random token sequences of a fixed length, with a consistent seed ensuring reproducibility across systems.

5.5 Hardware

All experiments are conducted on an NVIDIA DGX-1 system equipped with eight V100 GPUs (each with 32 GB of memory), interconnected via NVLINK. The system includes 16 DDR4 32 GB Micron memory modules rated at 2,133 MT/s and 2 20-core Intel Xeon E5-2698 V4 CPUs.

5.6 Metrics

Two metrics evaluate HARMOENY against competing systems: throughput and Mean Time to First Token (MTTFT). Throughput, measured in tokens per second, is computed as the total number of tokens generated during an experiment divided by its duration in seconds. MTTFT, expressed in milliseconds, represents the average time to process a batch. For MTTFT, all systems use identical batch sizes to ensure comparability, whereas throughput allows variable batch sizes to optimize performance.

Chapter 6

Results

6.1 Comparison to SOTA systems

The performance of HarMoEny is compared to the aforementioned systems as listed in Section 5.2. For the comparison, two differing environments are used: (1) an artificially skewed environment created with the Gini-Index as described in Section 4.6, and (2) more organic datasets as described in section 5.4. Each system will be given the exact same amount of identical data that will each be processed to only the first token. After the first token generations these sequences are considered complete. The generation phase is not considered as the KV cache largely discounts the need to do any form of rebalancing. The total number of requests gives the number of tokens which have been generated and which is divided by the running time of the experiment to yield the throughput.

It is strongly visible that HarMoEny shines under workloads that feature high expert inequality. Figure 6.1 bottom row show the differing systems compared at different Gini-Indexes, it can be seen that HarMoEny improves favourably as the level of inequality increases. At a GI of 0.5 HarMoEny attains a 1.36x improved throughput over

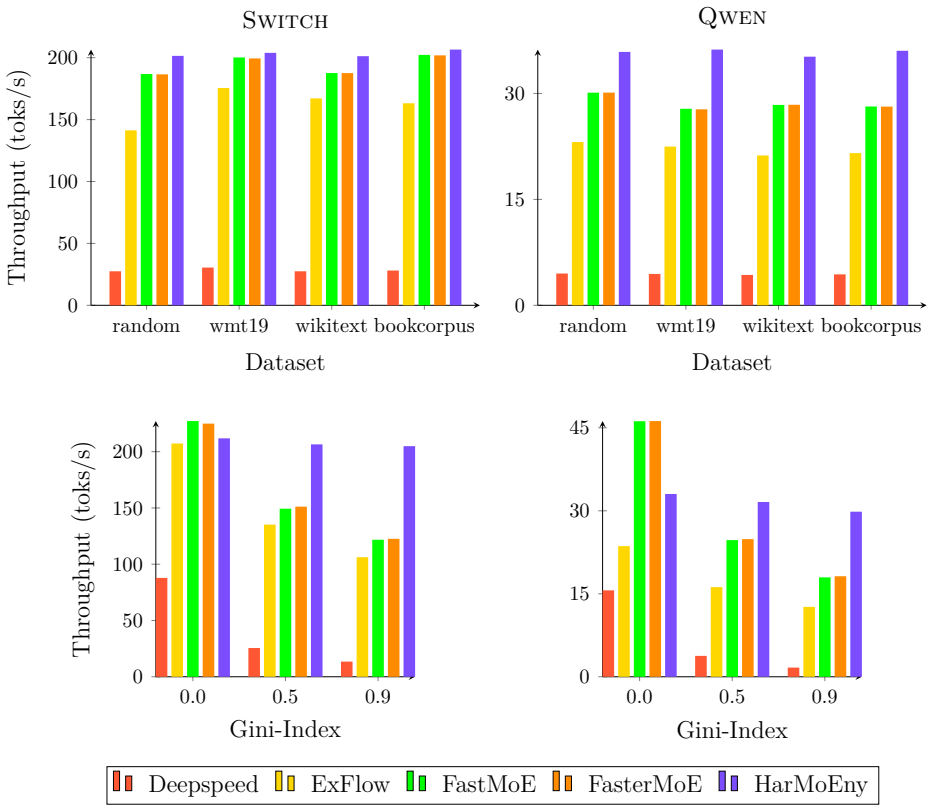

Figure 6.1: HarMoEny provides consitent performance regardless of input data, unlike competitors

FasterMoe, 1.38x over FastMoE, 1.53x over ExFlow, and 8.09x over DeepSpeed. These values improve more at the GI level of 0.9 getting an improved throughput of 1.69x over FastMoE and 15.21x over DeepSpeed. This trend is replicated for Qwen, where increasing expert inequality leads to larger performance improvements on HarMoEny compared to the competitors. At the GI of 0.9, HarMoEny attains a throughput gain of 1.66x over FastMoE, 2.36x over ExFlow, and 17.98x over DeepSpeed. The larger gains with Qwen can be attributed to the larger experts that take longer to process. Given the same inequality, larger experts would take longer, given there are more weights to multiply against.

There is an ever slight decline in performance for HarMoEny as the GI increases and that can be attributed to the fact that as the GI continuously increases, the number of tokens that each non-overloaded expert receives is less. This can create a situation where a given expert has so few tokens that it takes longer to load the overloaded expert than to execute the precedent expert, leading to GPU idling and thus a performance drop. q is set to keep the pipeline working at 100% but the start, where experts that are already loaded are executed first, may be a source of uncontrollable idling.

Furthermore, it can be seen and expected that with a GI of 0.0 each system should, in theory, perform similarly. HarMoEny has a slight performance drop over FastMoE and FasterMoE due to the principal fact that these two systems employ special optimized performance kernels which HarMoEny does not implement. It is these other optimizations that help bolster its throughput over HarMoEny at low expert inequality. If these other optimizations were to be removed to uniquely compare token sharing strategy, then it would be expected for HarMoEny to perform even better, and that is shown in Section 6.3.

Figure 6.1 also features the datasets from section 5.4 including the artificial Random

dataset in the first row with SWITCH on the left and QWEN on the right. It can be seen that HARMOENY retains the highest throughput across all datasets with FasterMoE and FastMoE trailing behind followed by ExFlow and, trailing significantly behind is, DeepSpeed. On average across all datasets HARMOENY's throughput is 1.047x more than FastMoE, 1.05x FasterMoE, 1.26x ExFlow, and 7.17x more than Deepspeed. Furthermore the throughput of HARMOENY varies the least with a variance of 6 $toks^2/s^2$ compared to FasterMoe at 62.7, FastMoe at 66.3, ExFlow at 212, and finally Deepspeed at 2.03. In terms of raw throughput and variance Deepspeed is an outlier. Deepspeed is an outlier because of a current bug in its implementation to avoid dropping tokens. Given that functionality is incomplete and prone to program hanging it is necessary to set the eval capacity factor to a worse case value. Setting to this value means that each GPU will allocate that amount over the average to execute on each expert, regardless of how many actual tokens it receives. This is a padding approach of MoE and although it removes a communication round, it is extremly inneficient, as is visible. The time is entirely predicated on the *eval capacity factor* and therefore it is expected that the running time across datasets for Deepspeed with the same value for *eval capacity factor* to be close to identical. Given that Deepspeed does not factor in imbalance, it is expected for the variance to be close to 0.

The performance of HARMOENY is not significantly better than Fast- and FasterMoE due to their having specialized kernels and SWITCH having smaller experts. For a more significant performance improvement, the top right quadrant of Figure 6.1 can be viewed where QWEN is executed on the various systems across the datasets. On average HARMOENY gets a throughput improvement of 1.25x over FastMoE, 1.62x over ExFlow, and 8.13x over Deepspeed. This improvements over FastMoE is 20% more in the QWEN case when compared to SWITCH.

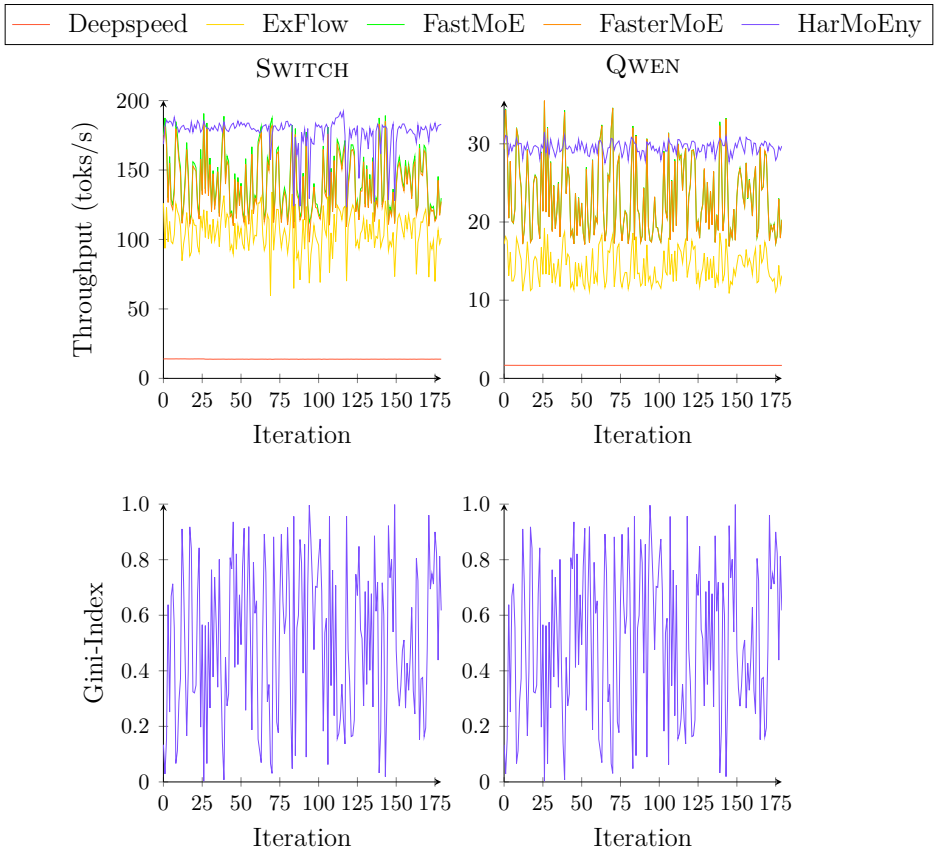

Figure 6.2: HarMoEny maintains consistent throughput regardless of input distribution

To display the adaptability of a system a timeline plot is provided, Figure 6.2, which at each iteration randomly selects a Gini index between 0 and 1 to artificially skew the batch for that iteration by that amount. The bottom row of Figure 6.2 feature a plot displaying the Gini-Index at each iteration; the left and right plots are identical. The top row more interestingly shows the throughput as a factor of the different systems over time affected by the Gini-Index for both the SWITCH and QWEN models. It is visually apparent for both SWITCH and QWEN that the variance between batches of same size but different Gini-Index is significantly less. HARMOENY has a variance of 152 $toks^2/s^2$ compared to ExFlow's 206 $toks^2/s^2$, FasterMoE's 447 $toks^2/s^2$, and FastMoE's 477 $toks^2/s^2$. Deepspeed has the lowest variance but that is not a fair comparison given the fact that Deepspeed pads the input to the longest possible worse case, which in this case is a Gini-Index of 1.0, meaning each GPU processes the same number of tokens albeit an excessive number that negatively impacts throughput significantly with an average of 13 $toks/s$ compared to HARMOENY's 176 $toks/s$. HARMOENY does suffer some performance drops of unknown origin but these happen infrequently compared to the opposing systems, multiple runs were taken and each time there were drops albeit at different locations but ever present. Of which, such drops did not occur for QWEN. Looking at QWEN HARMOENY achieves a larger variance gap with a variance of 0.59 $toks^2/s^2$ compared to ExFlow's 4.58 $toks^2/s^2$ (7.76x), FasterMoE's 22.77 $toks^2/s^2$ (38.59x), and FastMoE's 22.90 $toks^2/s^2$ (38.8x). For QWEN the variance drop for HARMOENY is higher with a maximum of 38.8x compared to SWITCH's maximum drop of 3.14x.

Another common metric measured is MTTFT, or the mean time to first token. This experiment, Figure 6.3, is conducted by fixing the batch size for all datasets and skews and systems unlike Figure 6.1 which has a varying batch size that is optimized for performance. As spoken earlier, the Deepspeed inference engine has an issue which forces the

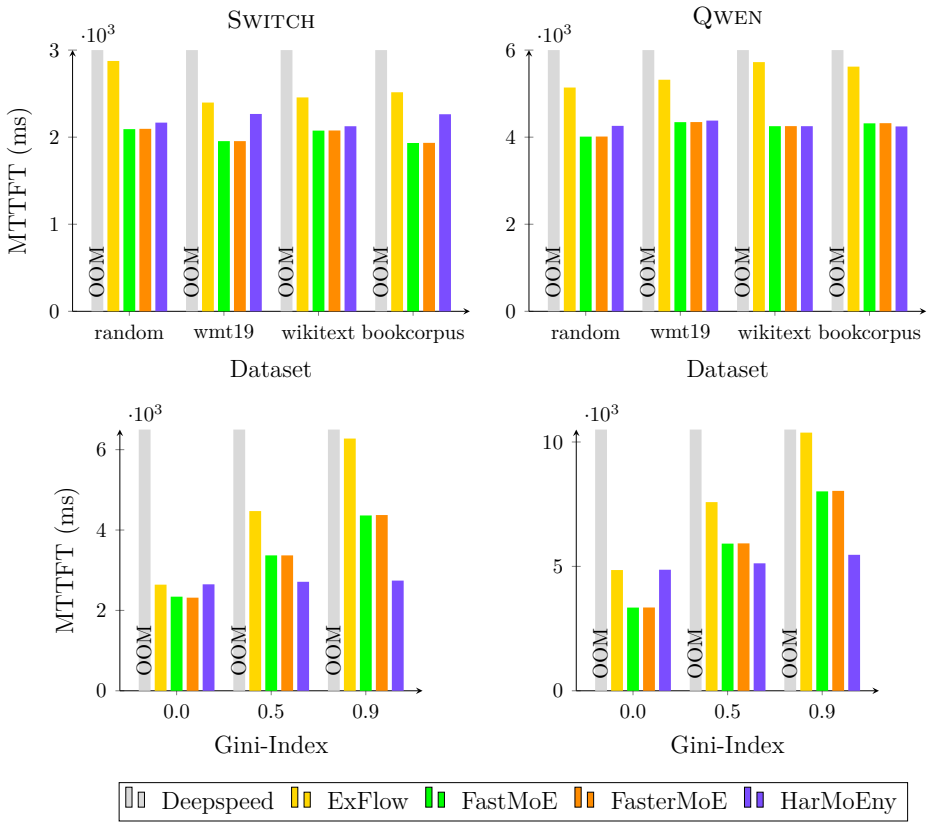

Figure 6.3: HarMoEny achieves similar MTTFT to other systems

use of unrealistically small batch sizes which would not procure interesting results, therefore Deepspeed inference is gray on the plot and marked with Out-of-Memory (OOM). A batch size is set such that it does not cause OOM exceptions for any system for any run. This is easily done by finding the largest batch size that works for each system with a Gini-Index of 0.9 for at least 1 batch, and taking the smallest value. With small levels of imbalance HarMoEny compares to Fast and FasterMoE and at very low levels of imbalance actually performs worse with a MTTFT increase of 13% over FastMoE for Switch and 45% over FastMoE for Qwen. However, with increasing levels of the Gini-Index HarMoEny maintains a steady MTTFT while other systems begin to steeply increase in latency, with HarMoEny getting a latency improvement of 62% over FastMoE for Switch and 68% over FastMoE for Qwen. These results do no directly reciprocally match Figure 6.1, HarMoEny gets better performances in Figure 6.1 because batch sizes are allowed to vary and with a more balanced system it is possible to use larger batch sizes which directly impacts throughput.

6.2 HarMoEny time breakdown

To present insights into which parts of the forward pass is taking the most time, a time breakdown of the different operations on a forward pass for an MoE layer serviced by HarMoEny is presented. To demonstrate significant performance, an ideal environment is created that is similar to the Gini-Index workload except it is modified such that 10 experts get 90% of the tokens (or 9% each) with the remaining tokens allocated uniformly across the leftover experts. Furthermore, the experts are chosen such that all overloaded experts are present on a single GPU. To obtain precise time measurements, NVIDIA CUDA Events are used before and after each LOC responsible for that operation of the

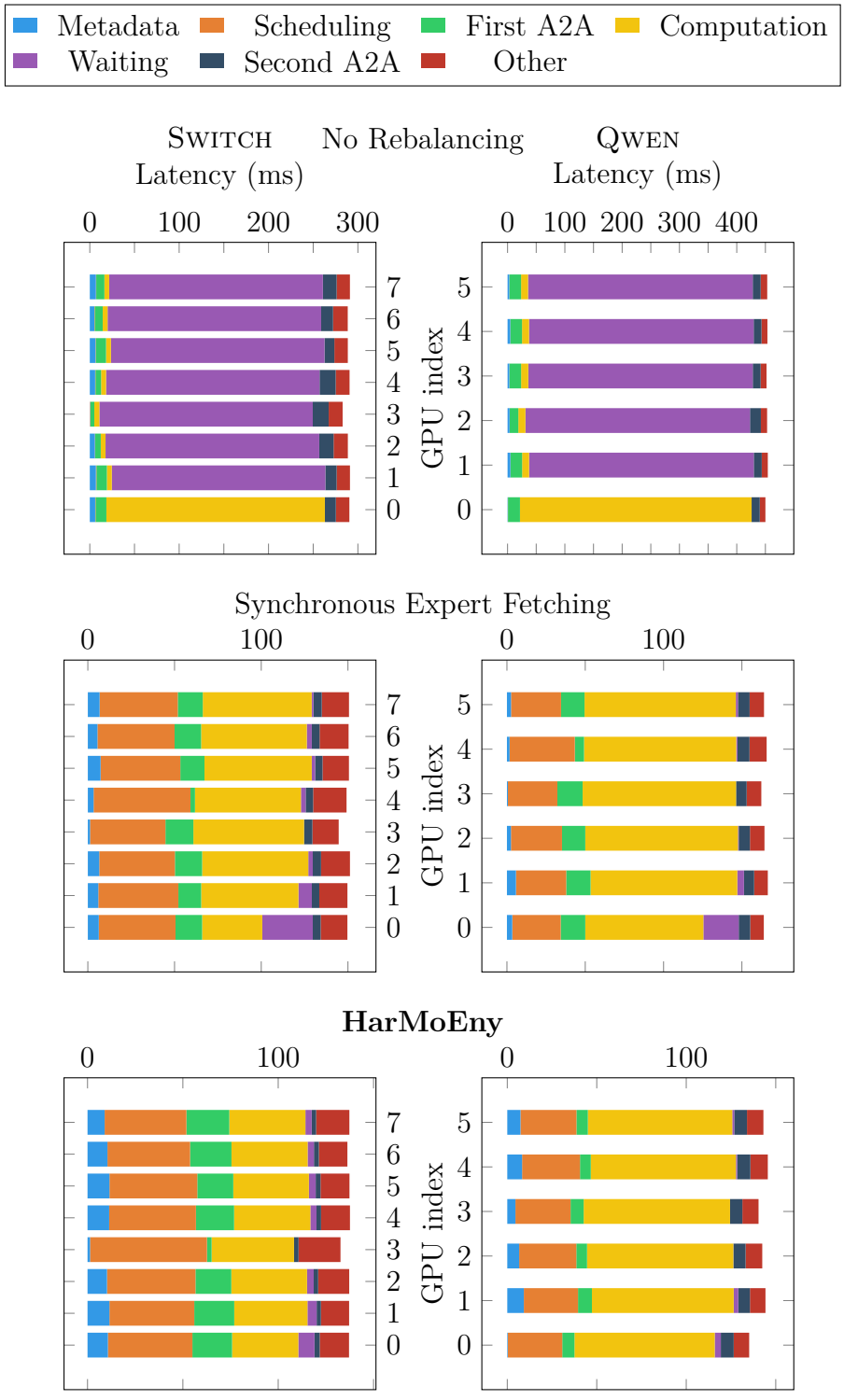

Figure 6.4: HarMoEny's performance is dependent on the use of its two principal components

forward pass. To prevent any overhead the time between events, to measure the elapsed time, is only computed after the forward pass completes. Figure 6.4 shows the duration of different operations in HarMoEny without any rebalancing (top), with rebalancing but without asynchronous expert fetching (middle), and HarMoEny with both components enabled (bottom). Given that the router statically assigns tokens to experts, each layer will perform identically and to save space only layer 0 is presented here.

With all 10 overloaded experts present on GPU 0, without any rebalancing (see Algorithm 2), all other GPUs are destined to wait a long time for all overloaded experts to be finished executing. Specifically the other GPUs spend on average 82.7% (Switch) and 86.5% (Qwen) of their MoE layer time waiting for GPU 0 to finish executing the overloaded experts.

The token rebalancing algorithm reduces the waiting time across GPUs. The mean waiting time goes from 82.7% to 3.99% (Switch) and 86.5% to 2.98% (Qwen) as seen in the middle row of Figure 2. This reduction in waiting time significantly reduces the overall time going from 289 ms to 149 ms (Switch) a speedup of 1.94x and 452 ms to 164 ms (Qwen) a speedup of 2.76x. Qwen gets a larger time saving and that can be attributed to Qwen having larger experts, which take longer to execute, making imbalance more costly for larger experts. There is the added cost of scheduling which takes 30% (Switch) and 20% (Qwen) of the total layer latency, but this added cost helps to significantly reduce the overall added cost of imbalance. However, looking closely at the middle row of Figure 2 it can be seen that GPU 0 is waiting a significant portion (19.5% for Switch and 13.8% for Qwen) instead of the previous situation of the other GPUs waiting. This is due to the added cost of the other GPUs taking time to load experts not already present in GPU memory.

This waiting time can be reduced by using asynchronous expert fetching and by

looking at the bottom of Figure 2 it can be seen that HarMoEny with both components enabled reduces the time further over synchronous going from 149 ms to 136.6 ms (Switch) a speedup of 1.1x and 164 ms to 141.8ms (Qwen) a speedup of 1.16x. Furthermore the time GPU 0 spends waiting is 6% (Switch) and 2% (Qwen) which is a drastic reduction from the previous 19.5% and 13.8% respectively. Therefore, it can be viewed that a combination of token rescheduling and asynchronous expert fetching in HarMoEny minimizes the amount of time spent idling on each GPU which further reduces the overall layer latency of a forward pass.

6.3 Load balancing policies

To include a more fair comparison the following ablation study on token routing policy is included as Figure 6.5, similarly to 6.1 the top row is the datasets while the bottom row is the artificial Gini-Index. Round-Robin encompasses the techniques of Deepspeed and FastMoE, Expert Shadowing is not implemented given that FasterMoE saw only marginal improvements over FastMoE. Even Split is also included which is a policy whereby each expert's tokens are evenly distributed across each GPU such that each GPU executes every single expert. ExFlow and HarMoEny both present the same values from 6.1.

The Gini-Index performance is helpful to validate the system against a myriad of imbalance possibilities. It can be seen that in Figure 6.5 bottom row that HarMoEny outperforms on all occasions. The performance of HarMoEny remains relatively stable as imbalance increases while the other policies that do not do any form of load balancing struggle with larger imbalanced workloads (minus Even Split which load balances perfectly). Now looking at Gini-Index of 0.0, Round-Robin is no longer peforming marginally better thanks to the removal of all other optimizations and specialized kernels. Looking

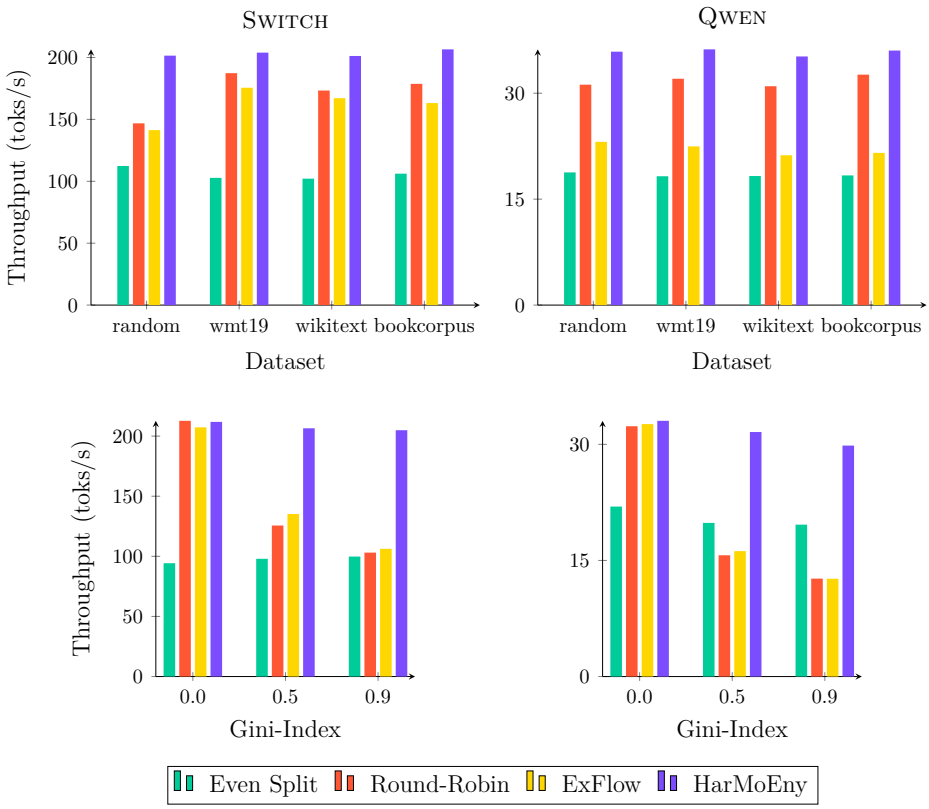

Figure 6.5: Focusing on how tokens are distributed yields HARMOENY outperforming all other techniques consistently

at the highest GI HarMoEny gets a 1.49x higher throughput over ExFlow, 1.54x over Round-Robin, and 2.13x over Even Split. The performance gains continue with Qwen where HarMoEny gets a 1.9x higher throughput over ExFlow, 1.92 over Deepspeed, and 1.54x over Even Split. Even Split is perfectly balanced yet performs relatively poorly, this is since it has to load and execute each expert. The running time is more dependent on the size of each expert and the number of experts. The loading induced by Switch with 128 experts will perform worse compared to Qwen with 60 experts. It is possible in certain situations for Even Split to more viable such as only 8 experts, or each expert is very small, or the PCIe bandwidth is very large. However, HarMoEny remains adaptive and can work in any environment regardless of number of experts, expert size, or hardware performance, as long as q is tuned correctly.

On the datasets, top row of Figure 6.5, it can be seen that HarMoEny exhibits the highest throughput constantly by a relatively strong margin with random performing the best. Furthermore HarMoEny features consistency across all datasets similar to Even Split, getting a variance of only 6 $toks^2/s^2$ compared to Even Split's 21.9, ExFlow's 212, and Round-Robin's 305. When it comes to establishing SLAs, HarMoEny provides confidence that regardless of input data, performance will be consistent. Qwen features a similar story with HarMoEny having the best throughput across all datasets. However, to the benefit of the other strategies, larger experts decrease the variability: 0.19 HarMoEny, 0.06 Even Split, 0.75 ExFlow, 0.57 Round-Robin (305 for Switch).

Figure 6.6 is similar to 6.2 but focuses on the token routing policies. The same run for HarMoEny and ExFlow from Figure 6.2 is used here. It is clearly apparent in Figure 6.6 that HarMoEny achieves significantly better performance to the competitors compared to Figure 6.2. HarMoEny has a consistent lead of 71 $toks/s$ over the next best vs Figure 6.2's 34 $toks/s$ lead over the next best for Switch. The same can be said for Qwen

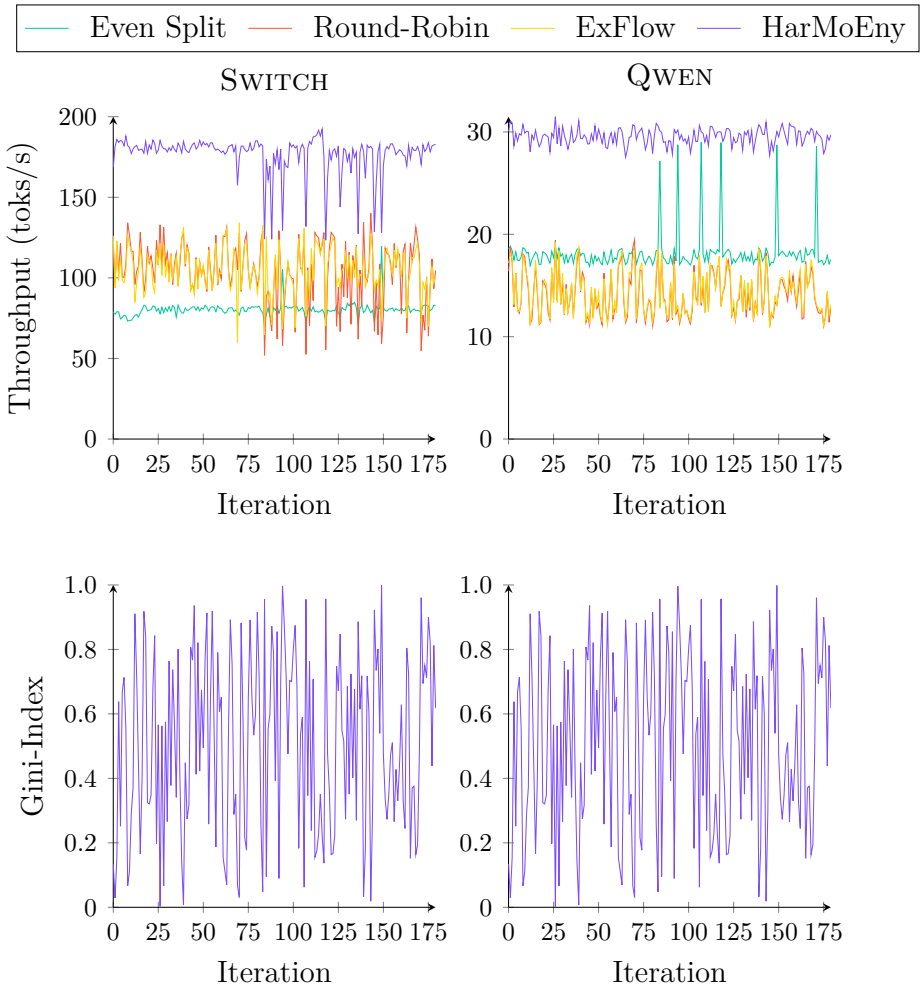

Figure 6.6: Focusing on how tokens are distributed yields HarMoEny obtaining a larger performance improvement over all other techniques consistently

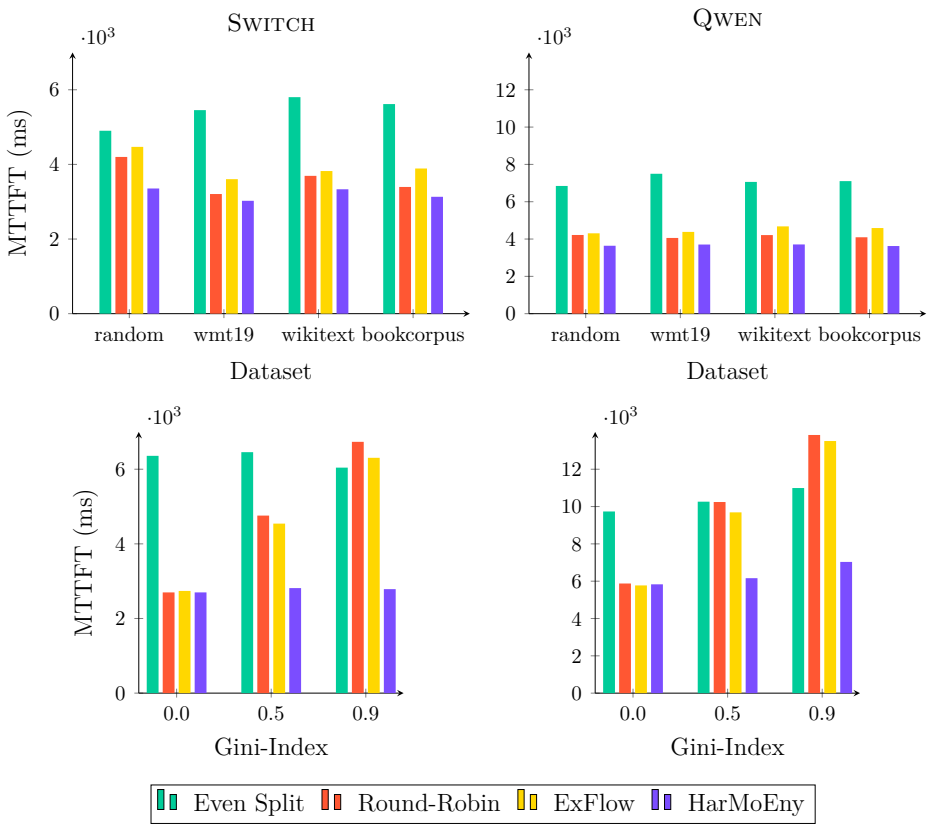

Figure 6.7: Focusing on how tokens are distributed yields HARMOENY maintaining a steady reduction in latency to first token

with 15 vs 6 $toks/s$. The performance reduction in the competitors is due to removing all other optimizations and focusing solely on the strategy of token placement. Interestingly Even Split goes from being worse perfoming in SWITCH to best performing for QWEN. This is due to having less experts in QWEN reducing the cost of loading every expert. Both Round-Robin and ExFlow perform identically as given the situation of skewing a single expert to achieve a target Gini-Index, it does not matter how experts are placed across GPUs, it will never amortize the imbalance.

Similarly to Figure 6.3 a similar experiment is performed fixing the batch size for all policies and finding the MTTFT. Figure 6.7 present HARMOENY and the different

policies on a level playing field without any other optimizations and show HarMoEny consistently outperforming across the board in all situation and mirroring performance when the level of imbalance is low. Of note, is that HarMoEny keeps the MTTFT consistent regardless of input imbalance: HarMoEny MTTFT increased, for Switch, by 4.2% for GI of 0.5 and 3.2% for GI of 0.9; Round-Robin increased by 76% for GI of 0.5 and 150% for GI of 0.9; ExFlow increased by 65.8% for GI of 0.5 and 130% for GI of 0.9. It is clear that HarMoEny outmatches other policies when it comes to adaptability and consistent performance. Similarly for Qwen we have 5.6% and 20% for HarMoEny, 74.3% and 135% for Round-Robin, and 67.8% and 134% for ExFlow.

Chapter 7

Conclusion

This work presents a thorough investigation into a critical aspect of machine learning's evolution: Mixture-of-Experts models. Despite training strategies that employ a balancing loss on the router to distribute tokens evenly across experts, inference reveals persistent imbalances. Specifically, Google's Switch Transformer and Alibaba's Qwen1.5-MoE-A2.7B exhibit routers that disproportionately favor certain experts, as demonstrated in this study. In expert parallelism, this imbalance leads to straggler effects, where GPUs idle while awaiting a single overloaded GPU, wasting cycles and degrading E2E performance. These slowdowns, varying with input data, can severely impact inference throughput.

To address this, HarMoEny introduces selective token load balancing—redistributing tokens to underutilized GPUs and prefetching experts with minimal overhead—yielding significant performance gains. Throughput improvements reach up to 1.69× in highly imbalanced scenarios, with gains amplifying as imbalance intensifies. Moreover, HarMoEny maintains stable throughput across diverse inputs, offering service-level consistency. Crucially, this approach complements existing techniques like specialized kernels,

enabling seamless integration with systems like FastMoE and DeepSpeed. In balanced scenarios, HarMoEny incurs no overhead, ensuring no performance degradation relative to the base system. As datacenters prioritize GPU utilization to enhance efficiency and reduce costs, HarMoEny offers a versatile, low-effort solution adaptable to any MoE inference framework.

Chapter 8

Future Work

HarMoEny was developed as a standalone inference engine to expedite prototyping, independent of existing frameworks. Integrating its core components—the *Scheduling Algorithm* and *Asynchronous Expert Prefetching Protocol*—into widely adopted systems like DeepSpeed would validate their speedup and orthogonality. This would leverage pre-existing optimizations in these frameworks, offering a clearer comparison of performance with and without HarMoEny's enhancements. An initial ablation study (Section 6.3) explores this, but a full integration with DeepSpeed would provide more compelling evidence.

A natural extension involves scaling HarMoEny to multi-server environments, such as single racks or datacenters. Higher data transfer latencies in these settings would necessitate adaptations to the current techniques, potentially unlocking new optimizations to mitigate expert imbalance at scale.

Furthermore, HarMoEny assumes that all accelerators have the same performance when scheduling. It would be interesting to also study the effects of expert scheduling with a heterogeneous setup featuring different GPUs or a homogeneous one with straggler

GPUs.

While the Gini-Index effectively stress-tests HarMoEny, real-world inference traces from user prompts—unlike the kitchen-sink training datasets (Section 5.4)—would better capture the unpredictability of chatbot interactions. Such traces could deepen insights into MoE imbalance limitations and refine load-balancing strategies.

Finally, converting the PyTorch-based *Scheduling Algorithm* and *Asynchronous Expert Prefetching Protocol* into specialized CUDA kernels could reduce overhead. Scheduling currently consumes 20–30% of an MoE layer's forward-pass latency; optimized kernels could minimize this cost, further boosting performance.

Bibliography

[1] J. Kaplan, S. McCandlish, T. Henighan, *et al.*, *Scaling laws for neural language models*, 2020. arXiv: 2001.08361 [cs.LG]. [Online]. Available: https://arxiv.org/abs/2001.08361.

[2] W. Fedus, B. Zoph, and N. Shazeer, "Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity," *Journal of Machine Learning Research*, vol. 23, no. 120, pp. 1–39, 2022. [Online]. Available: http://jmlr.org/papers/v23/21-0998.html.

[3] Qwen, : A. Yang, *et al.*, *Qwen2.5 technical report*, 2025. arXiv: 2412.15115 [cs.CL]. [Online]. Available: https://arxiv.org/abs/2412.15115.

[4] A. Q. Jiang, A. Sablayrolles, A. Roux, *et al.*, *Mixtral of experts*, 2024. arXiv: 2401.04088 [cs.LG]. [Online]. Available: https://arxiv.org/abs/2401.04088.

[5] DeepSeek-AI, D. Guo, D. Yang, *et al.*, *Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning*, 2025. arXiv: 2501.12948 [cs.CL]. [Online]. Available: https://arxiv.org/abs/2501.12948.

[6] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *The bulletin of mathematical biophysics*, vol. 5, pp. 115–133, 1943.

[7] F. Rosenblatt, "The perceptron: A probabilistic model for information storage and organization in the brain.," *Psychological review*, vol. 65, no. 6, p. 386, 1958.

[8] M. Minsky and S. Papert, "An introduction to computational geometry," *Cambridge tiass., HIT*, vol. 479, no. 480, p. 104, 1969.

[9] A. Vaswani, N. Shazeer, N. Parmar, *et al.*, "Attention is all you need," in *Advances in Neural Information Processing Systems*, I. Guyon, U. V. Luxburg, S. Bengio, *et al.*, Eds., vol. 30, Curran Associates, Inc., 2017. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.

[10] C. Raffel, N. Shazeer, A. Roberts, *et al.*, "Exploring the limits of transfer learning with a unified text-to-text transformer," *Journal of Machine Learning Research*, vol. 21, no. 140, pp. 1–67, 2020. [Online]. Available: http://jmlr.org/papers/v21/20-074.html.

[11] R. A. Jacobs, M. I. Jordan, S. J. Nowlan, and G. E. Hinton, "Adaptive mixtures of local experts," *Neural Computation*, vol. 3, no. 1, pp. 79–87, 1991. DOI: 10.1162/neco.1991.3.1.79.

[12] N. Shazeer, A. Mirhoseini, K. Maziarz, *et al.*, *Outrageously large neural networks: The sparsely-gated mixture-of-experts layer*, 2017. arXiv: 1701.06538 [cs.LG]. [Online]. Available: https://arxiv.org/abs/1701.06538.

[13] E. Frantar, C. Riquelme, N. Houlsby, D. Alistarh, and U. Evci, "Scaling laws for sparsely-connected foundation models," *arXiv preprint arXiv:2309.08520*, 2023.

[14] J. Krajewski, J. Ludziejewski, K. Adamczewski, *et al.*, "Scaling laws for fine-grained mixture of experts," *arXiv preprint arXiv:2402.07871*, 2024.

[15] D. Eigen, M. Ranzato, and I. Sutskever, *Learning factored representations in a deep mixture of experts*, 2014. arXiv: `1312.4314 [cs.LG]`. [Online]. Available: `https://arxiv.org/abs/1312.4314`.

[16] E. Bengio, P.-L. Bacon, J. Pineau, and D. Precup, *Conditional computation in neural networks for faster models*, 2016. arXiv: `1511.06297 [cs.LG]`. [Online]. Available: `https://arxiv.org/abs/1511.06297`.

[17] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *nature*, vol. 323, no. 6088, pp. 533–536, 1986.

[18] Q. Team, *Introducing qwen1.5*, Feb. 2024. [Online]. Available: `https://qwenlm.github.io/blog/qwen1.5/`.

[19] J. Bai, S. Bai, Y. Chu, *et al.*, "Qwen technical report," *arXiv preprint arXiv:2309.16609*, 2023.

[20] hiyouga, *Qwen1.5: An Open-Source Language Model*. [Online]. Available: `https://github.com/hiyouga/Qwen1.5`.

[21] Y. N. Dauphin, A. Fan, M. Auli, and D. Grangier, "Language modeling with gated convolutional networks," in *International conference on machine learning*, PMLR, 2017, pp. 933–941.

[22] J. Ainslie, J. Lee-Thorp, M. De Jong, Y. Zemlyanskiy, F. Lebrón, and S. Sanghai, "Gqa: Training generalized multi-query transformer models from multi-head checkpoints," *arXiv preprint arXiv:2305.13245*, 2023.

[23] J. Su, M. Ahmed, Y. Lu, S. Pan, W. Bo, and Y. Liu, "Roformer: Enhanced transformer with rotary position embedding," *Neurocomputing*, vol. 568, p. 127 063, 2024.

[24] Y. Zheng, R. Zhang, J. Zhang, *et al.*, "Llamafactory: Unified efficient fine-tuning of 100+ language models," in *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations)*, Bangkok, Thailand: Association for Computational Linguistics, 2024. [Online]. Available: http://arxiv.org/abs/2403.13372.

[25] L. Ouyang, J. Wu, X. Jiang, *et al.*, "Training language models to follow instructions with human feedback," *Advances in neural information processing systems*, vol. 35, pp. 27 730–27 744, 2022.

[26] C. J. Shallue, J. Lee, J. Antognini, J. Sohl-Dickstein, R. Frostig, and G. E. Dahl, "Measuring the effects of data parallelism on neural network training," *Journal of Machine Learning Research*, vol. 20, no. 112, pp. 1–49, 2019.

[27] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Advances in neural information processing systems*, vol. 25, 2012.

[28] Y. Huang, Y. Cheng, A. Bapna, *et al.*, "Gpipe: Efficient training of giant neural networks using pipeline parallelism," *Advances in neural information processing systems*, vol. 32, 2019.

[29] B. Wang, Q. Xu, Z. Bian, and Y. You, "Tesseract: Parallelize the tensor parallelism efficiently," in *Proceedings of the 51st International Conference on Parallel Processing*, 2022, pp. 1–11.

[30] D. Lepikhin, H. Lee, Y. Xu, *et al.*, "Gshard: Scaling giant models with conditional computation and automatic sharding," *arXiv preprint arXiv:2006.16668*, 2020.

[31] N. Corporation, *Point-to-point communication*, 2020. [Online]. Available: https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/usage/p2p.html#all-to-all.

[32] S. Rajbhandari, C. Li, Z. Yao, *et al.*, "DeepSpeed-MoE: Advancing mixture-of-experts inference and training to power next-generation AI scale," in *Proceedings of the 39th International Conference on Machine Learning*, K. Chaudhuri, S. Jegelka, L. Song, C. Szepesvari, G. Niu, and S. Sabato, Eds., ser. Proceedings of Machine Learning Research, vol. 162, PMLR, Jul. 2022, pp. 18 332–18 346. [Online]. Available: https://proceedings.mlr.press/v162/rajbhandari22a.html.

[33] J. He, J. Qiu, A. Zeng, Z. Yang, J. Zhai, and J. Tang, *Fastmoe: A fast mixture-of-expert training system*, 2021. arXiv: 2103.13262 [cs.LG]. [Online]. Available: https://arxiv.org/abs/2103.13262.

[34] J. He, J. Zhai, T. Antunes, *et al.*, "Fastermoe: Modeling and optimizing training of large-scale dynamic pre-trained models," in *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '22, Seoul, Republic of Korea: Association for Computing Machinery, 2022, pp. 120–134, ISBN: 9781450392044. DOI: 10.1145/3503221.3508418. [Online]. Available: https://doi.org/10.1145/3503221.3508418.

[35] C. Hwang, W. Cui, Y. Xiong, *et al.*, "Tutel: Adaptive mixture-of-experts at scale," in *Proceedings of Machine Learning and Systems*, D. Song, M. Carbin, and T. Chen, Eds., vol. 5, Curan, 2023, pp. 269–287. [Online]. Available: https://proceedings.mlsys.org/paper_files/paper/2023/file/5616d34cf8ff73942cfd5aa922842556-Paper-mlsys2023.pdf.

[36] J. Li, Y. Jiang, Y. Zhu, C. Wang, and H. Xu, "Accelerating distributed {moe} training and inference with lina," in *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, 2023, pp. 945–959.

[37] S. Rajbhandari, J. Rasley, O. Ruwase, and Y. He, "Zero: Memory optimizations toward training trillion parameter models," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2020, pp. 1–16. DOI: 10.1109/SC41405.2020.00024.

[38] Y. J. Kim, A. A. Awan, A. Muzio, *et al.*, "Scalable and efficient moe training for multitask multilingual models," *arXiv preprint arXiv:2109.10465*, 2021.

[39] Z. Liu, Y. Lin, Y. Cao, *et al.*, "Swin transformer: Hierarchical vision transformer using shifted windows," in *Proceedings of the IEEE/CVF international conference on computer vision*, 2021, pp. 10 012–10 022.

[40] M. Ott, S. Edunov, A. Baevski, *et al.*, "Fairseq: A fast, extensible toolkit for sequence modeling," *arXiv preprint arXiv:1904.01038*, 2019.

[41] J. Yao, Q. Anthony, A. Shafi, H. Subramoni, and D. K. DK Panda, "Exploiting inter-layer expert affinity for accelerating mixture-of-experts model inference," in *2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2024, pp. 915–925. DOI: 10.1109/IPDPS57955.2024.00086.

[42] M. Zhai, J. He, Z. Ma, Z. Zong, R. Zhang, and J. Zhai, "SmartMoE: Efficiently training Sparsely-Activated models through combining offline and online parallelization," in *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, Boston, MA: USENIX Association, Jul. 2023, pp. 961–975, ISBN: 978-1-939133-35-9. [Online]. Available: https://www.usenix.org/conference/atc23/presentation/zhai.

[43] T. Gale, D. Narayanan, C. Young, and M. Zaharia, "Megablocks: Efficient sparse training with mixture-of-experts," *Proceedings of Machine Learning and Systems*, vol. 5, pp. 288–304, 2023.

[44] K. Kranen and V. Nguyen, *Applying mixture of experts in llm architectures*, 2024. [Online]. Available: `https://developer.nvidia.com/blog/applying-mixture-of-experts-in-llm-architectures/`.

[45] C. Gini, "Variabilità e mutabilità (variability and mutability)," *Tipografia di Paolo Cuppini, Bologna, Italy*, vol. 156, 1912.

[46] Z. D. (Shamauk), *[bug] enabling drop_tokens in moe layer causes inference to hang*, GitHub, GitHub issue #6809, 2025. [Online]. Available: `https://github.com/deepspeedai/DeepSpeed/issues/6809`.

[47] imoneoi, *Feature request: Expert parallel for moe architectures*, Github, GitHub issue #2405, 2024. [Online]. Available: `https://github.com/vllm-project/vllm/issues/2405`.

[48] C. Raffel, N. Shazeer, A. Roberts, *et al.*, "Exploring the limits of transfer learning with a unified text-to-text transformer," *Journal of Machine Learning Research*, vol. 21, no. 140, pp. 1–67, 2020. [Online]. Available: `http://jmlr.org/papers/v21/20-074.html`.

[49] Y. Zhu, R. Kiros, R. Zemel, *et al.*, *Aligning books and movies: Towards story-like visual explanations by watching movies and reading books*, 2015. arXiv: `1506.06724 [cs.CV]`. [Online]. Available: `https://arxiv.org/abs/1506.06724`.

[50] S. Merity, C. Xiong, J. Bradbury, and R. Socher, *Pointer sentinel mixture models*, 2016. arXiv: 1609.07843 [cs.CL]. [Online]. Available: https://arxiv.org/abs/1609.07843.

[51] W. Foundation. "Acl 2019 fourth conference on machine translation (wmt19), shared task: Machine translation of news." (), [Online]. Available: http://www.statmt.org/wmt19/translation-task.html.

A Routing imbalance

A.1 Total tokens

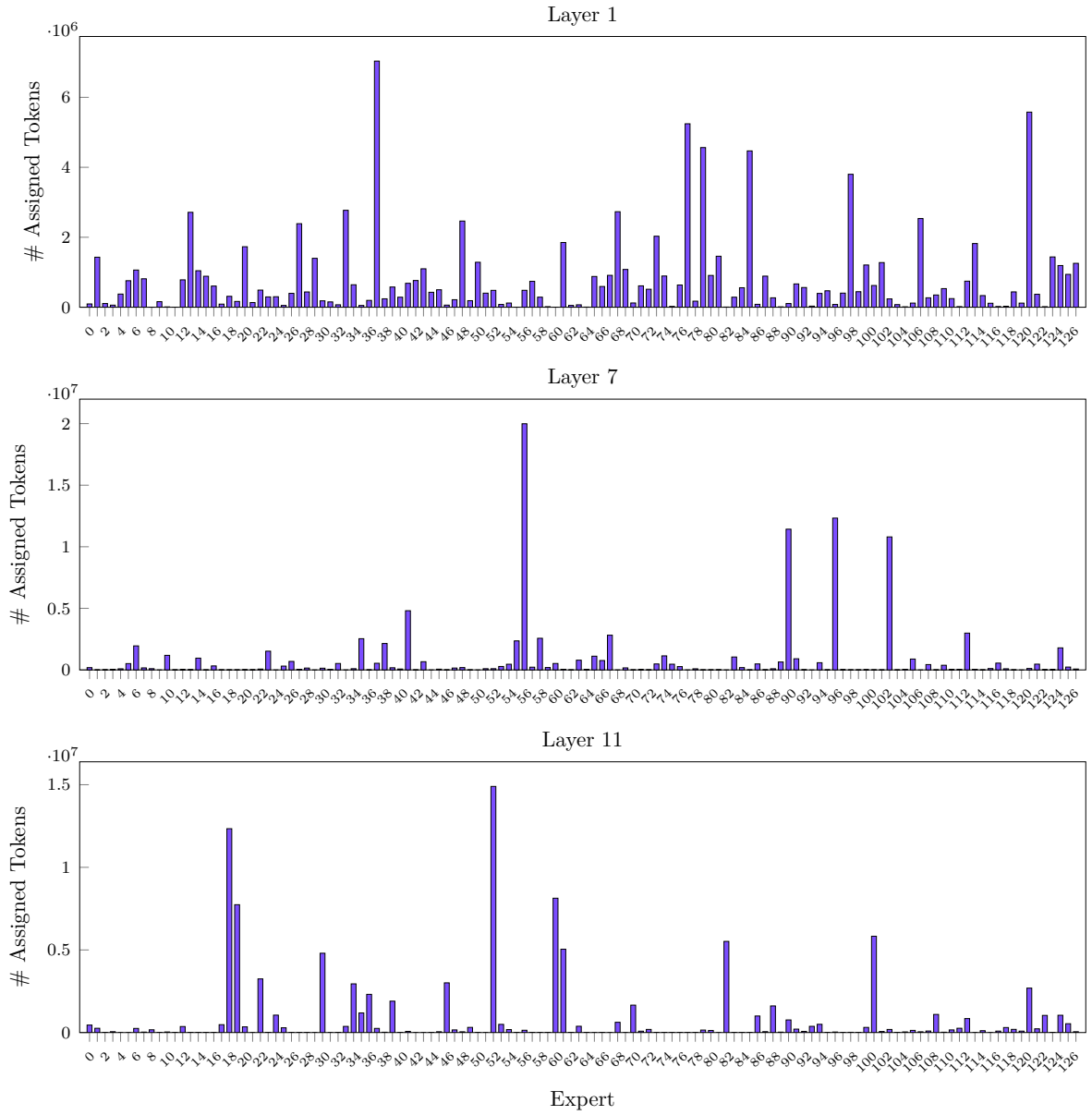

Figure 1: Token distribution on BookCorpus run across all Switch Transformer experts.

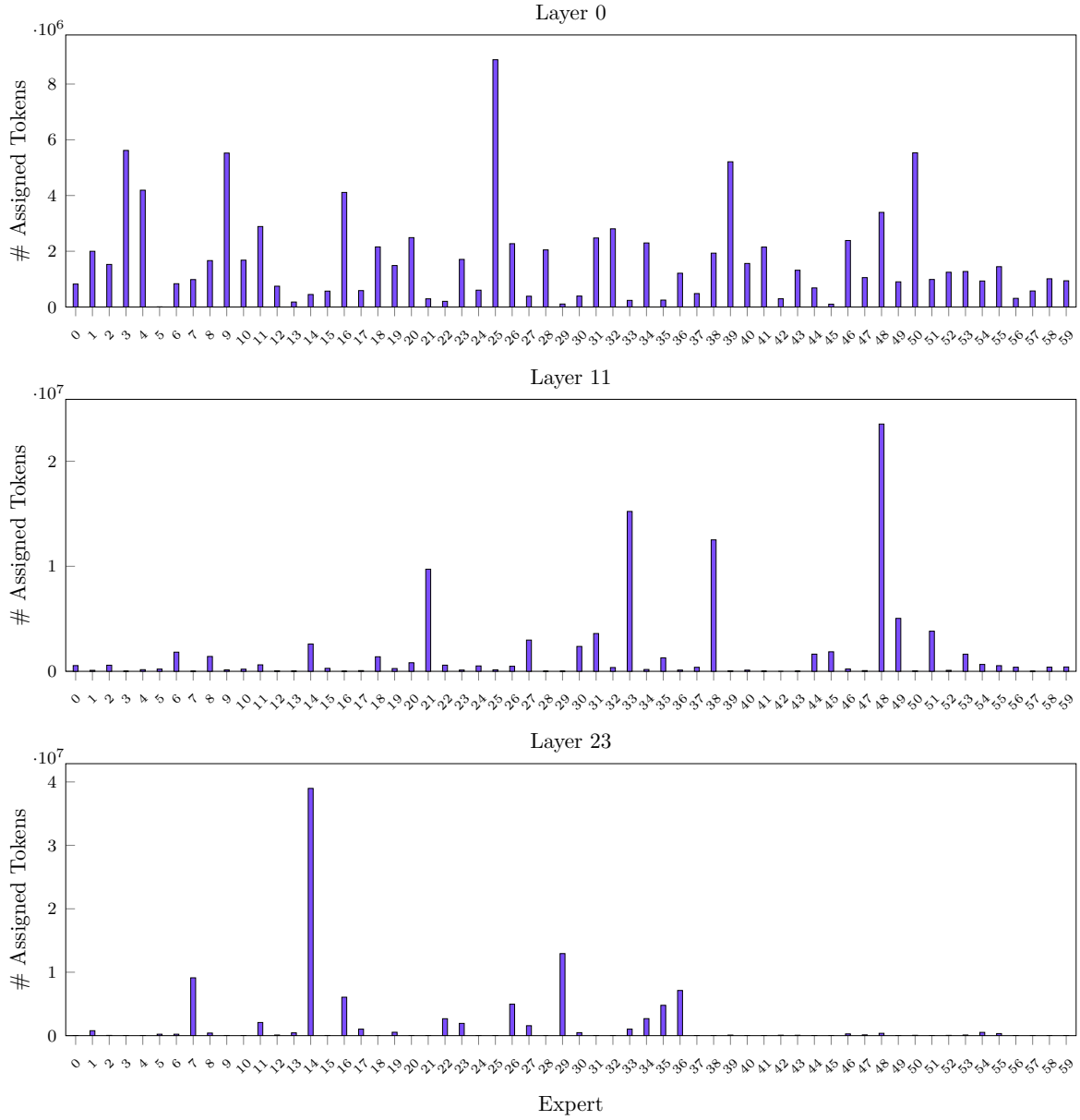

Figure 2: Token distribution on BookCorpus run across all Qwen1.5-MoE-A2.7B experts.

A.2 CDF

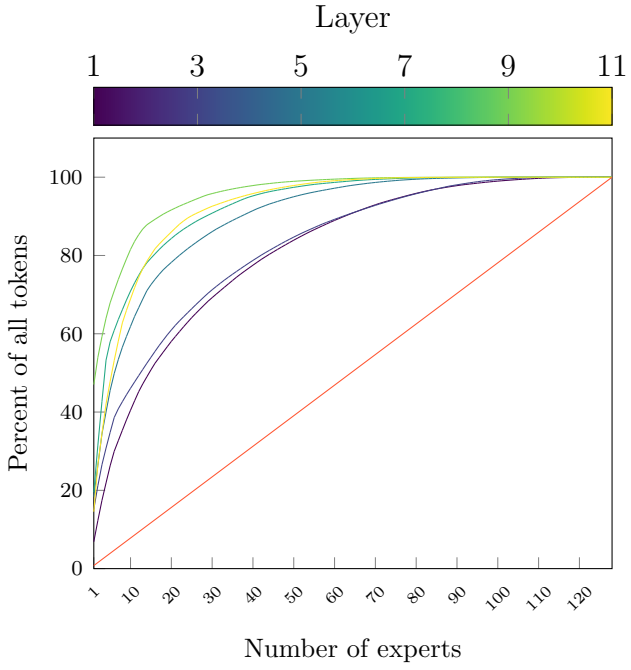

Figure 3: CDF of expert token distribution of Switch Transformer with 128 experts on BookCorpus dataset for all 12 MoE layers.

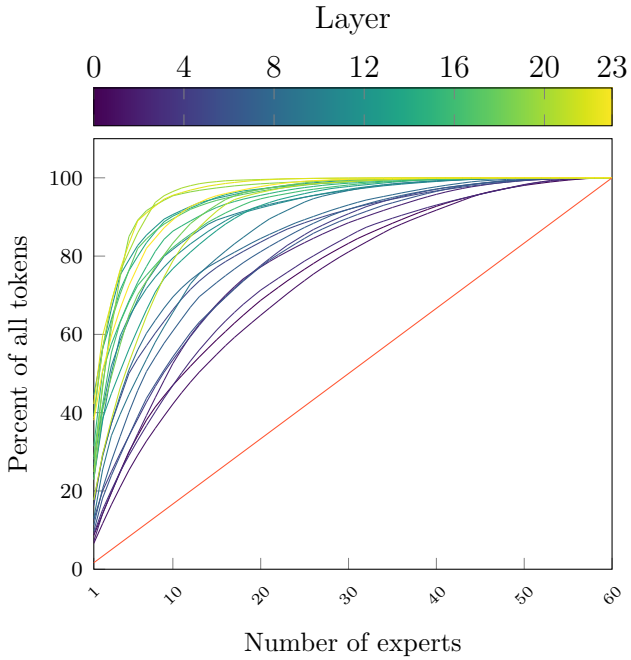

Figure 4: CDF of expert token distribution of Qwen1.5-MoE-A2.7B with 60 experts on BookCorpus dataset for all 24 MoE layers.

B q

Following is a step-by-step breakdown of getting a formula for estimating the minimal size for q given that the expert is a 2-layer MLP. q represents the number of tokens that are required so that its processing time is greater than the time it takes to load an expert. Let the expert have two linear layers with the first being of size $m \times p$, and the second being $p \times m$. The expert is evaluated as xW^1W^2. This can be represented as

$$\frac{\text{Number of Floating Point Operations}}{\text{GPU FLOPS}} > \frac{\text{Expert Size}}{\text{PCIe Bandwidth}} \tag{1}$$

$$\frac{|O|}{\phi} > \frac{|E|}{\beta} \tag{2}$$

$$\frac{qp(2m-1) + qm(2p-1)}{\phi} > \frac{(mp+pm)d_{type}}{\beta} \tag{3}$$

$$\tag{4}$$

On ignoring small terms and simplifying, we get

$$\frac{2qpm + 2qpm}{\phi} > \frac{2pm \cdot d_{type}}{\beta} \tag{5}$$

$$\frac{q \cdot 4pm}{\phi} > \frac{2pm \cdot d_{type}}{\beta} \tag{6}$$

$$q > \frac{\phi \cdot d_{type}}{2\beta} \tag{7}$$

C Gini Index

Taking Gini Index Formula

$$GINI\_INDEX = \frac{\sum_{i=1}^{\epsilon}\sum_{j=1}^{\epsilon}|N_i - N_j|}{2\epsilon\sum_{i=1}^{\epsilon}N_i} \tag{8}$$

$$= \frac{\rho(\rho 0 + (\epsilon-\rho)(\dot{N}-N)) + (\epsilon-\rho)(\rho(\dot{N}-N) + (\epsilon-\rho)0)}{2\epsilon\chi} \tag{9}$$

$$= \frac{\rho(\epsilon-\rho)(\dot{N}-N) + \rho(\epsilon-\rho)(\dot{N}-N)}{2\epsilon\chi} \tag{10}$$

$$= \frac{2\rho(\epsilon-\rho)(\dot{N}-N)}{2\epsilon\chi} \tag{11}$$

Total number of tokens sum of the amount on skewed experts plus amount on non-skewed

$$\chi = \rho\dot{N} + (\epsilon-\rho)N \tag{12}$$

$$\frac{\chi - \rho\dot{N}}{\epsilon-\rho} = N \tag{13}$$

Substituting the two together and isolating for $\dot{N}$

$$\epsilon\chi G = \rho(\epsilon - \rho)(\dot{N} - N) \tag{14}$$

$$\frac{\epsilon\chi G}{\rho(\epsilon - \rho)} = \dot{N} - N \tag{15}$$

$$\frac{\epsilon\chi G}{\rho(\epsilon - \rho)} = \dot{N} - \frac{\chi - \rho\dot{N}}{\epsilon - \rho} \tag{16}$$

$$\frac{\epsilon\chi G}{\rho} = \epsilon\dot{N} - \rho\dot{N} - \chi - \rho\dot{N} \tag{17}$$

$$\frac{\epsilon\chi G}{\rho} - \chi = \epsilon\dot{N} - \rho\dot{N} - \rho\dot{N} \tag{18}$$

$$\frac{\epsilon\chi G + \rho\chi}{\rho} = (\epsilon - 2\rho)\dot{N} \tag{19}$$

$$\frac{\epsilon\chi G + \rho\chi}{\rho(\epsilon - 2\rho)} = \dot{N} \tag{20}$$

D HarMoEny setup

The following Python code demonstrates how to add HarMoEny to an existing model. The `replace_moe_layer` function injects our MoE implementation based on user-specified parameters.

```
from harmonymoe.utils import replace_moe_layer
from harmonymoe.moe_layer import MoEConfig, MoELayer

model = create_pytorch_model() # Custom model

config = MoEConfig(
    rank,
    world_size,
    scheduling_policy,
    expert_cache_size,
    eq_tokens,
    d_model,
    num_experts,
)

replace_moe_layer(
    model,
    moe_parent_type,
    moe_type,
    path_to_experts,
    path_to_router_linear_layer,
```

```
    config,
)
```