How Much Parallelism Is "Free"? A Principle of Near-Free Parallelism for Parallel Decoding

Minghua He [*1,2], **Lingzhe Zhang**[2], **Yuan Liu**[1], **Xiao Zhou**[1], **Aiwei Liu**[†1]
[1]WeChat AI, Tencent, [2]Peking University
hemh2120@stu.pku.edu.cn, coveliu@tencent.com

Abstract

Parallel decoding improves generation efficiency by processing multiple decode positions within a single decode forward, but reported speedups conflate algorithmic token utilization with the system cost of executing multiple positions. We isolate the system side by introducing Near-Free Parallelism (NFP), the maximum number of positions executable at near-free latency. Analyzing Dense FFNs, MoE FFNs, and Attention against an idle-compute baseline, we find that NFP is shaped not by memory-bound resource slack alone, but also by implementation-induced kernel-granularity slack. Based on these mechanisms, we establish a Near-Free Parallelism principle that predicts the NFP boundary from hardware balance and implementation granularity. Validation on representative Dense and MoE models—spanning both diffusion and autoregressive decoding—shows that the principle accurately predicts practical NFP boundaries, revealing that the standard idle-compute intuition over-predicts them by 2.4–23× in over half of the evaluated deployment settings—offering a system-side budget for parallelism selection and model-system co-design.

1 Introduction

Autoregressive (AR) decoding is the dominant generation paradigm for large language models (Brown et al., 2020; Touvron et al., 2023), but its token-by-token nature has long been a fundamental bottleneck for inference efficiency. As models scale, each decode step typically produces only a single new token while accessing the full model parameters and KV cache, leaving modern accelerators underutilized in memory-bound regimes (Williams et al., 2009; Pope et al., 2023). This bottleneck has motivated a growing body of work on decoding architectures that introduce parallelism during generation.

Representative approaches include speculative decoding (Leviathan et al., 2023; Chen et al., 2023), multi-token prediction (MTP) (Gloeckle et al., 2024; Liu et al., 2024), and diffusion language models (DLLMs) (Sahoo et al., 2024; Nie et al., 2025; Ye et al., 2025a). Despite differing in algorithmic design, they share a common systems structure: processing multiple decode positions within a single decode forward. Existing work typically reports aggregate end-to-end speedups, but these numbers entangle two distinct factors: how many useful tokens an algorithm extracts from processed positions, and how cheaply the model-system stack can process those positions. Naively, increasing the number of positions N should increase computation and data movement. Yet in practical inference systems, latency often remains nearly flat over a non-trivial range of N. This raises a hidden systems question: **how many decode positions in total can the model-system stack process before latency becomes expensive?**

We call this system-side capacity boundary **Near-Free Parallelism (NFP)**: the maximum number of decode positions that a decode forward can process while latency remains within a prescribed tolerance. By isolating system-side capacity from algorithmic token utilization, NFP provides a common basis for comparing decoding methods, selecting parallelism levels, and co-designing model architectures with inference systems.

Despite its utility, the NFP boundary remains poorly understood. A common intuition attributes it to idle compute in memory-bound decoding (Williams et al., 2009; Ivanov et al., 2021)—underutilized compute absorbs extra positions at near-zero cost—but this explains only why a near-free regime may arise, not how far it extends or what governs its boundary. The central question is therefore: **what determines the NFP boundary, and can it be predicted for a given setting?**

To study this boundary, we abstract parallel

[*]This work was done during the internship at WeChat AI.
[†]Corresponding author.

Findings of the Association for Computational Linguistics: EMNLP 2026, pages 43520–43578
October 24-29, 2026 ©2026 Association for Computational Linguistics

decoding methods as a *multi-position decode forward* and formalize the idle-compute intuition—the prediction of classical roofline analysis—as a baseline. We then examine when this baseline succeeds or fails.

We perform a modular study of Dense FFNs, Attention, and MoE FFNs across multiple inference frameworks and hardware platforms. For each module, we ask: Does a near-free regime exist? Is its boundary predicted by idle compute alone? If not, what additional mechanisms govern it?

Our analysis shows that the NFP boundary is not determined by idle compute alone. Instead, it is shaped by two interacting sources of slack: memory-bound resource slack, which roofline analysis captures, and kernel-granularity slack introduced by implementation details such as padding and tiling, which roofline analysis cannot express and which can shift the realized boundary beyond the pure compute-to-memory prediction. Based on this joint mechanism, we establish a **Near-Free Parallelism principle** that predicts the NFP boundary as the minimum of a roofline-style resource bound and software-granularity bounds, computed from hardware and execution configurations.

We validate the principle on representative Dense and MoE models—covering both diffusion and autoregressive decoding paradigms—and show that module-level mechanisms compose into an accurate full-model predictor across hardware platforms. In over half of the evaluated deployment settings, the idle-compute baseline over-predicts the realized boundary by 2.4–23×. We further use the principle to reason about future hardware trends: as compute grows faster than memory bandwidth, the opportunity space for parallel decoding may enlarge.

In summary, our contributions are as follows:

- **A system-side capacity for parallel decoding.** We formulate NFP as the maximum number of decode positions processable at near-free latency, isolating system-side capacity from algorithmic token utilization.

- **A joint mechanism beyond roofline.** We show that NFP is not determined by the roofline idle-compute bound alone: implementation-induced slack, especially padding slack in kernel implementations, is absent from the roofline model and can substantially shift the realized boundary.

- **A predictive NFP principle.** We establish a unified min-of-bounds principle: roofline analysis supplies the resource-balance bound, kernel implementations supply the granularity bounds, and the NFP boundary is predicted by their minimum.

- **Cross-stack validation.** We validate the principle across representative modules and models under diverse inference frameworks, kernel implementations, and hardware platforms, and analyze future hardware implications.

2 Near-Free Parallelism: Abstraction and Baseline

2.1 Parallel Decoding as Multi-Position Decode Forward

We abstract parallel decoding methods by ignoring how positions are generated, verified, or updated, and focus only on the number of positions processed by one decode forward.

In standard autoregressive decoding, each decode forward processes one position and produces the output for one new token:

$$\ell_t = F_\theta(z_t), \qquad N = 1. \tag{1}$$

Here z_t denotes the input representation at the current decode position, ℓ_t denotes its output logits, and N is the number of decode positions processed in this forward pass. Thus, autoregressive decoding is a single-position decode forward.

In contrast, parallel decoding processes multiple decode positions within the same decode forward:

$$\boldsymbol{\ell}_{t:t+N-1} = F_\theta(Z_{t:t+N-1}), \qquad N > 1. \tag{2}$$

Here $Z_{t:t+N-1}$ denotes the inputs placed at N decode positions, and $\boldsymbol{\ell}_{t:t+N-1}$ denotes their corresponding outputs. The shared system-side question is then: **how does decode latency change as N increases?**

2.2 Near-Free Parallelism

In practice, decode latency often remains nearly flat as the number of positions increases over a non-trivial range. We formalize the boundary of this range as **Near-Free Parallelism (NFP)**.

Let $T(N)$ denote the latency of one decode forward when it processes N decode positions, where $N = 1$ corresponds to standard autoregressive decoding. Because latency is rarely exactly constant, we define near-free execution

with respect to a tolerance ϵ. A forward with N positions is near-free if its latency remains within $(1+\epsilon)$ of the single-position AR baseline:

$$T(N) \leq (1+\epsilon)T(1). \tag{3}$$

The near-free boundary is then the largest number of decode positions that satisfies this condition:

$$N_{\max}(\epsilon) = \max\{N : T(N) \leq (1+\epsilon)T(1)\}. \tag{4}$$

This definition is algorithm-independent: it measures only the system-side capacity to execute positions at negligible additional latency cost.

2.3 Idle-Compute Intuition as a Roofline Baseline

Prior work has suggested idle compute in memory-bound decoding as one explanation for NFP (Kim et al., 2023; Xia et al., 2024). We use this intuition as a simple baseline. In a standard AR decode step, the model processes only one position while still moving model weights and KV-cache data, leading to latency dominated by memory traffic. In this regime, additional computation can be absorbed by otherwise idle compute units, so increasing the number of decoded positions may have little latency effect.

When N positions are processed in one forward, let $C(N)$ and $B(N)$ denote the resulting computation and data movement, and define arithmetic intensity (AI). We count one fused multiply-add as two floating-point operations, consistent with hardware-reported TFLOP/s and profiler-reported FLOPs.

$$AI(N) = \frac{C(N)}{B(N)}, \qquad \rho = \frac{\phi}{\beta}, \tag{5}$$

where ϕ and β are hardware-dependent peak compute throughput and peak memory bandwidth, respectively, and ρ is the corresponding hardware balance point. The idle-compute baseline predicts that parallel positions remain near-free until the decode forward reaches this balance point:

$$AI(N_{\text{idle}}) = \rho. \tag{6}$$

Thus, N_{idle} is the roofline prediction of the near-free boundary: a first-order estimate of $N_{\max}$ that captures only the parallelism explained by memory-bound idle compute, excluding implementation effects such as kernel granularity and padding.

3 Module-Level Analysis of Near-Free Parallelism

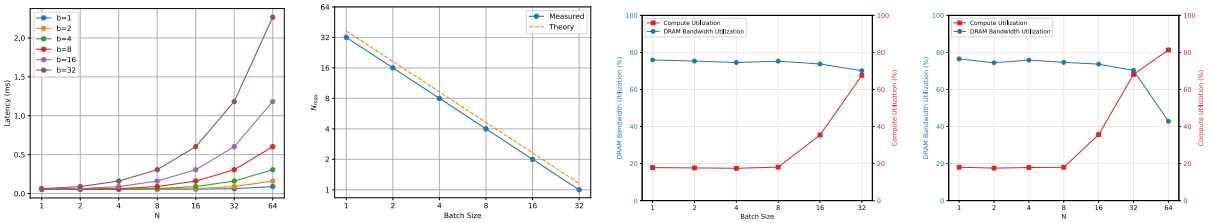
3.1 Study Design

Study Subject. In this section, we study NFP at the module level. Since a full decode forward involves multiple modules with distinct execution characteristics, its overall NFP boundary may conflate different mechanisms. We therefore isolate three representative module classes in modern LLMs (Vaswani et al., 2017): Dense FFNs, MoE FFNs, and Attention. Dense FFNs capture regular dense matrix computation, MoE FFNs introduce routed expert computation with token grouping and padding effects, and Attention captures KV-cache-dominated memory access.

Study Method. To enable a systematic comparison across modules, we analyze each module using the same set of research questions (RQ):

- **RQ1**: Does a near-free regime exist as the number of decode positions N increases?
- **RQ2**: Is the observed boundary predicted by the idle-compute baseline?
- **RQ3**: What factors govern the observed boundary?

For each module, we apply a common analysis procedure that connects the theoretical baseline to the measured latency behavior. We first instantiate the idle-compute baseline by deriving the arithmetic intensity $AI(N)$, which gives the predicted boundary N_{idle}. We then measure the latency curve $T(N)$ and define the observed NFP boundary $N_{\max} = N_{\max}(0.2)$ as the largest N whose latency remains within a 20% increase over the baseline. The 20% value reduces sensitivity to GPU timing noise and does not drive the conclusions: across a sweep of $\epsilon \in [0.05, 0.30]$, the MoE FFN and Attention boundaries are nearly invariant (Tables 19–24), and the Dense FFN boundary shifts by at most one measured N step while preserving its scaling trend (Table 18). Any $\epsilon \in [0.10, 0.30]$ yields consistent boundaries across all evaluated modules and backends (Appendix I). Except for the load-balanced MoE case, where the baseline is the smallest N that activates all experts, the baseline is $T(1)$. Finally, we compare $N_{\max}$ with N_{idle}: consistency between the two suggests that memory-bound idle compute is the dominant mechanism, whereas a mismatch indicates additional mechanism effects.



(a) Latency vs. N (b) $N_{\max}$ vs. b (c) $N=1$ utilization vs. b (d) $b=1$

Figure 1: Dense FFN evaluation results on **NVIDIA H20 GPU**.

Detailed experimental configurations, hardware platforms, and implementations for all module experiments are provided in Appendix C.

3.2 Dense FFN Layers

For the Dense FFN analysis, we model a standard feed-forward layer as two GEMM operators, corresponding to the up-projection and down-projection, respectively. We omit the activation function in order to isolate the dense matrix-computation behavior that dominates the module-level cost. Full results are provided in Appendix D.1.

3.2.1 Theoretical Baseline Prediction

We instantiate the idle-compute baseline for Dense FFNs. Let b denote the batch size, N the number of decode positions per request, d_{model} the hidden dimension, d_{ff} the FFN intermediate dimension, and s the number of bytes per element. Under a weight-traffic-dominated abstraction, a Dense FFN processes bN positions using two GEMM operators, whose computation and dominant memory traffic are approximated as

$$\begin{aligned} C_{\text{dense}}(N) &\approx 4bNd_{\text{model}}d_{\text{ff}}, \\ B_{\text{dense}}(N) &\approx 2d_{\text{model}}d_{\text{ff}}s. \end{aligned} \tag{7}$$

The key asymmetry is that computation scales with the total number of processed positions bN, while the dominant weight traffic is amortized across positions. The AI is therefore

$$AI_{\text{dense}}(N) = \frac{C_{\text{dense}}(N)}{B_{\text{dense}}(N)} \approx \frac{2bN}{s}. \tag{8}$$

Substituting this intensity into the idle-compute baseline condition yields

$$N_{\text{idle}}^{\text{dense}} \approx \frac{\rho s}{2b}, \tag{9}$$

This result predicts that the NFP boundary of Dense FFNs decreases inversely with batch size. Intuitively, increasing batch size consumes the same compute slack as increasing N, thereby reducing the near-free range.

3.2.2 RQ1: NFP Existence

To examine whether Dense FFNs exhibit module-level NFP, we sweep the number of decode positions N under different batch sizes b and measure the corresponding latency $T(N)$. As shown in Figure 1a, $T(N)$ remains nearly flat over an initial range of N and then increases as more positions are processed. This near-free region is larger at smaller batch sizes and shrinks as b increases.

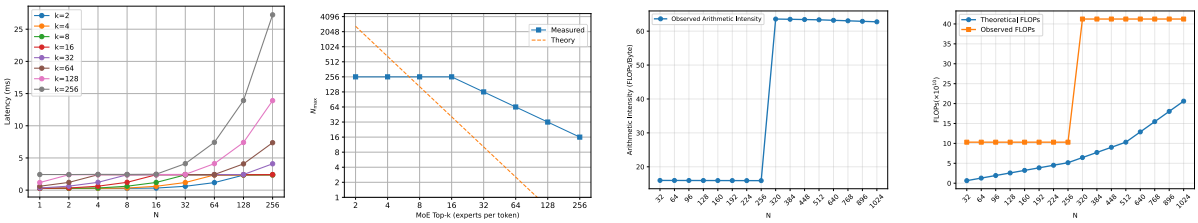
3.2.3 RQ2: Idle-Compute Prediction

To evaluate the idle-compute prediction for Dense FFNs, we compare the measured boundary $N_{\max}$ with the theoretical estimate $N_{\text{idle}}^{\text{dense}}$. As shown in Figure 1b, the measured boundary closely follows the predicted trend: as b increases, $N_{\max}$ decreases approximately proportionally to $1/b$. This agreement indicates that, for Dense FFNs, the observed NFP boundary is well explained by the idle-compute baseline.

3.2.4 RQ3: Boundary Mechanism

To verify the mechanism behind the Dense FFN boundary, we profile compute utilization and DRAM bandwidth utilization at $N=1$ and while sweeping N. As shown in Figure 1c–1d, Dense FFN execution at $N=1$ is bandwidth-dominated: DRAM bandwidth utilization is high, whereas compute utilization remains low. Across batch sizes, larger batches increase compute utilization at $N=1$, thereby reducing the available compute slack. As N increases, compute utilization rises while DRAM bandwidth utilization remains nearly unchanged, indicating that increasing N consumes idle compute until execution reaches the compute-bandwidth balance point predicted by the idle-compute baseline.

> Finding 1: Dense FFN NFP is governed by idle-compute slack. It scales inversely with batch size.



(a) Latency vs. N (b) $N_{\max}$ vs. k (c) AI, $k=8$ (d) FLOPs, $k=8$

Figure 2: MoE FFN evaluation for load-balanced routing (upper-bound case) with **vLLM** on **NVIDIA H20**. Load-skewed (lower-bound) results in Figure 14.

3.3 MoE FFN Layers

For the MoE FFN analysis, we instantiate a MoE layer (Shazeer et al., 2017; Jiang et al., 2024) as a routed expert-execution pipeline consisting of token dispatch, expert FFN computation, and weighted combine. Each expert is implemented as a standard two-GEMM FFN, and the full layer applies these expert computations to the tokens assigned to each expert through dispatch and combine. To capture practical implementation behavior, we use the fused MoE operators from vLLM (Kwon et al., 2023) and SGLang (Zheng et al., 2024), which jointly implement these stages.

Since real MoE routing is input-dependent and yields nondeterministic expert-load distributions, we analyze two controlled routing cases that expose distinct expert-load behaviors. The **load-balanced** case evenly distributes tokens across experts, whereas the **load-skewed** case concentrates all tokens on the same k experts. Under load-balanced routing, increasing N from 1 also activates additional experts, adding weight traffic unrelated to parallelism. To isolate the parallelism effect, we define the load-balanced baseline as the smallest N that activates all experts, so that further increases in N only add decode positions without changing the active expert set. Full results are provided in Appendix D.2.

3.3.1 Theoretical Baseline Prediction

We instantiate the idle-compute baseline for MoE FFNs (full derivation in Appendix B.1). Let k denote the number of selected experts per token and E_{act} the number of distinct experts activated by a routing pattern. Unlike Dense FFNs, MoE memory traffic includes both active expert weights and routing-induced activation movement. The idle-compute baseline condition then yields

$$N_{\text{idle}}^{\text{moe}} \approx \frac{2\rho s E_{\text{act}} d_{\text{ff}}}{b(4kd_{\text{ff}} - \rho s(1+3k+\eta k))}, \tag{10}$$

when $4kd_{\text{ff}} > \rho s(1 + 3k + \eta k)$; otherwise, the execution remains memory-bound under this abstraction.[1] This result predicts that the MoE NFP boundary decreases inversely with batch size, and is primarily controlled by k and E_{act}: larger k reduces the boundary, whereas larger E_{act} increases it. Thus, load-balanced routing serves as the upper-bound case and load-skewed routing as the lower-bound case.

3.3.2 RQ1: NFP Existence

We sweep N under both routing cases ($b = 1$); load-balanced (upper-bound) results are shown in Figure 2, with load-skewed (lower-bound) results in Figure 14. As shown in Figure 2a, $T(N)$ remains nearly flat over an initial range of N and then increases in both cases, confirming module-level NFP in MoE FFNs.

3.3.3 RQ2: Idle-Compute Prediction

We compare the measured boundary $N_{\max}$ with the theoretical estimate $N_{\text{idle}}^{\text{moe}}$ under both routing cases. As shown in Figure 2b, the measured boundaries do not consistently follow the idle-compute prediction in either case. This mismatch indicates that, unlike Dense FFNs, the MoE boundary requires mechanisms beyond idle compute alone.

3.3.4 RQ3: Boundary Mechanism

To identify the mechanisms missing from the idle-compute baseline, we profile the achieved arithmetic intensity while sweeping N. As shown in Figures 2c–2d, the measured arithmetic intensity changes in discrete steps rather than smoothly with N, suggesting that the MoE FFN execution is affected by implementation-level kernel granularity.

To further test this hypothesis, we compare the theoretical FLOPs from the baseline model with the actual FLOPs executed at runtime. The runtime

[1] $\eta = 2$ counts per-expert activation accesses in the combine stage.

FLOPs often exceed the theoretical FLOPs and exhibit a similar staircase pattern, indicating that the fused MoE kernels perform additional padded computation beyond the logical expert workload. We confirm this by inspecting the vLLM and SGLang fused MoE kernels in Appendix E, where tokens assigned to each expert are padded according to framework-defined block or tile sizes before expert GEMM execution. These results show that MoE FFN NFP arises primarily from implementation-induced kernel-granularity slack: larger N remains near-free within existing padded kernel regions until crossing a granularity boundary.

> Finding 2: MoE FFN NFP is governed by expert-token padding granularity. Balanced routing gives the upper bound on NFP; skewed routing gives the lower bound.

3.4 Attention Layers

For the Attention analysis, we model a decode causal MHA layer over a KV cache, processing N query positions against a cached sequence of length L, instantiated using FlashAttention (Dao et al., 2022) and FlashInfer (Ye et al., 2025b) to capture practical backend behavior. We set $b = 1$ and omit QKV and output projections to isolate KV-cache-dominated module-level cost (full results in Appendix D.3).

3.4.1 Theoretical Baseline Prediction

We instantiate the idle-compute baseline for Attention layers (full derivation in Appendix B.2). Let L denote the cached sequence length. The dominant computation scales with query-cache interactions NL, while the dominant KV-cache traffic is primarily controlled by L. The idle-compute baseline condition then yields

$$N_{\text{idle}}^{\text{attn}} \approx \begin{cases} \frac{\rho s L}{2L-\rho s}, & 2L > \rho s, \\ +\infty, & 2L \leq \rho s. \end{cases} \tag{11}$$

This predicts that the Attention NFP boundary is primarily governed by L: when L is small, attention remains memory-bound over a wide range of N; when L is large, the boundary approaches a finite value.

3.4.2 RQ1: NFP Existence

We sweep N under different cached sequence lengths L. As shown in Figure 3a, $T(N)$ exhibits a clear staircase pattern, with latency remaining nearly constant over several consecutive values of N before jumping to the next level, confirming module-level NFP in Attention layers.

3.4.3 RQ2: Idle-Compute Prediction

To evaluate the idle-compute prediction for Attention layers, we compare the measured boundary $N_{\max}$ with the theoretical estimate $N_{\text{idle}}^{\text{attn}}$ under different cached sequence lengths L. As shown in Figure 3b, the measured boundary remains nearly constant across L, whereas the idle-compute baseline predicts a length-dependent boundary. This mismatch indicates that the Attention NFP boundary is not fully explained by KV-cache memory-bound idle compute alone. This motivates examining implementation-level effects in RQ3.

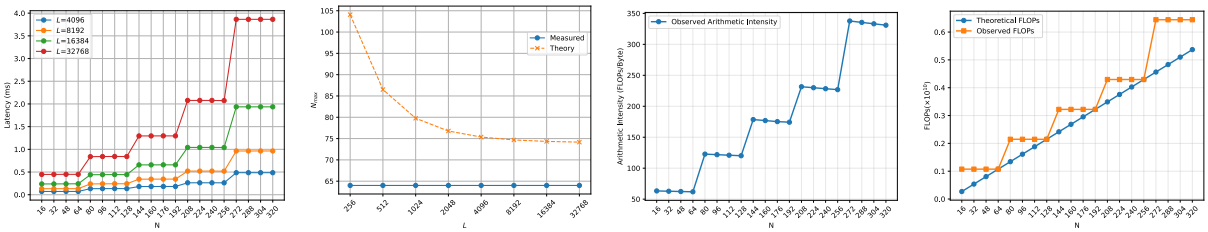
3.4.4 RQ3: Boundary Mechanism

We apply the same profiling protocol as for MoE FFNs (Figure 3c–3d). The results show the same pattern: arithmetic intensity and runtime FLOPs change in discrete steps due to query-position padding in FlashAttention and FlashInfer (Appendix F). Thus, Attention NFP is also governed by kernel-granularity slack. Unlike MoE, the query-tile boundary is reached before idle compute is exhausted, making granularity rather than resource balance the effective constraint on the Attention NFP boundary.

> Finding 3: Attention NFP is governed by backend query-tile granularity. It is largely independent of cached sequence length.

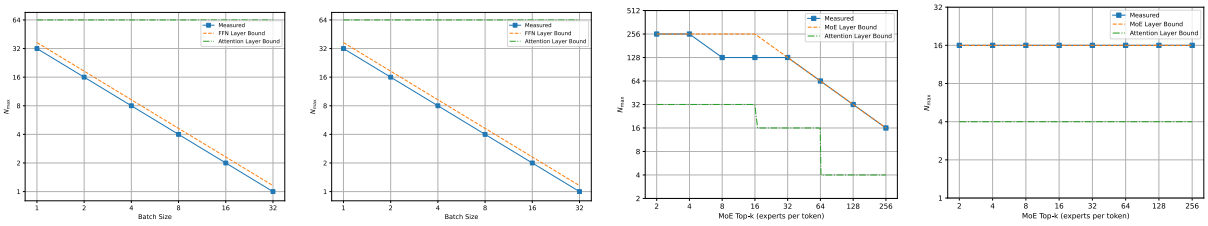
4 Near-Free Parallelism Principle

The module-level analysis identifies two NFP mechanisms: memory-bound idle compute for Dense FFNs, and implementation-induced kernel-granularity slack for MoE FFNs and Attention. At the model level, we use a first-exiting-module approximation: since modules within each layer execute sequentially without inter-layer overlap in standard inference, a full-model forward remains near-free only while all major modules remain within their module-level near-free regimes. Thus, the model-level NFP boundary is the minimum of module-level boundaries under a fixed backend and execution configuration. This min-of-bounds structure distinguishes the principle from roofline analysis: roofline supplies only the resource-balance term, while the remaining terms are kernel-granularity bounds with no roofline



(a) Latency vs. N (b) $N_{\max}$ vs. L (c) AI, $L=1024$ (d) FLOPs, $L=1024$

Figure 3: Attention evaluation results with **FlashAttention** on **NVIDIA H20**.



(a) Dense, $L=128$ (b) Dense, $L=256$ (c) Load-Balanced, $L=256$ (d) Load-Skewed, $L=256$

Figure 4: Full-model NFP principle validation results on **NVIDIA H20**.

counterpart. All terms follow from the backend and execution configuration, so the binding bound is identified before measurement.

4.1 Principle for Dense Models

For dense models, NFP is jointly constrained by Dense FFNs and Attention. The Dense FFN constraint comes from the idle-compute boundary, while the Attention constraint comes from backend-specific query granularity. Combining these two constraints gives

$$N_{\max}^{\text{dense}} \approx \min\left(\frac{\rho s}{2b}, M_{\text{attn}}\right). \tag{12}$$

where M_{attn} denotes the maximum number of query positions absorbable within one backend-specific Attention execution granularity.

4.2 Principle for MoE Models

For MoE models, NFP is jointly constrained by MoE FFNs and Attention. Let E be the number of experts, k the number of selected experts per token, M_{moe} the expert-token padding granularity of the MoE implementation, M_{attn} the query granularity of the Attention backend, and τ the token-count threshold beyond which the MoE backend switches kernel configuration.

In the load-balanced case, expert-token padding slack is distributed across experts. The MoE side can absorb roughly $M_{\text{moe}}E$ padded expert-token slots, while each decode position consumes k expert-token slots. This gives the backend-aware upper-bound boundary

$$N_{\max}^{\text{moe,bal}} \approx \min\left(\frac{M_{\text{moe}}E}{k}, \tau, M_{\text{attn}}\right). \tag{13}$$

The first term captures aggregate MoE padding capacity, the second term enforces the validity range of the selected MoE backend branch, and the third term captures the Attention backend constraint. For the evaluated vLLM and SGLang backend configurations, $\tau = E$, yielding the specialized form used in our experiments.

In the load-skewed case, tokens concentrate on the same k experts. The MoE boundary is therefore fixed by a single expert-token padding granularity rather than the aggregate slack across all experts:

$$N_{\max}^{\text{moe,skew}} \approx \min(M_{\text{moe}}, M_{\text{attn}}). \tag{14}$$

Both M_{moe} and M_{attn} are implementation-dependent and are determined from the backend configurations analyzed in Appendices E.3 and F.3.

5 Model-Level Validation of Near-Free Parallelism Principle

We validate the principle on full-model inference using WeDLM-8B (Liu et al., 2025) and Qwen3-8B (Yang et al., 2025) (Dense, nano-vLLM + FlashAttention) and LLaDA-2.1-mini (Bie et al., 2026) and Ling-2.0-mini (Team et al., 2025) (MoE, SGLang + FlashInfer), covering DLLM and AR

Model type	Representative setting	Idle-compute baseline	NFP Principle	Baseline discrepancy
Dense	H20, $b=1$	~ 37	~ 37	—
Dense	H20, $b=4$	~ 9	~ 9	—
Dense	A800, $b=1$	~ 153	~ 64	$2.4\times$ over (Attn. tile)
Dense	H800, $b=1$	~ 295	~ 64	$4.6\times$ over (Attn. tile)
Dense	H800, $b=8$	~ 37	~ 37	—
MoE, balanced	$E=256, k=8$	$\sim 1453^{\dagger}$	~ 64	$23\times$ over (granularity)
MoE, balanced	$E=256, k=32$	$\sim 362^{\dagger}$	~ 64	$5.7\times$ over (granularity)
MoE, balanced	$E=256, k=64$	$\sim 181^{\dagger}$	~ 64	$2.8\times$ over (granularity)
MoE, skewed	any k	$\sim 45^{\dagger}$	~ 16	$2.8\times$ over (MoE padding)

Table 1: Deployment lookup comparing the idle-compute baseline with the NFP principle under representative settings ($M_{\text{attn}}=64$). †MoE baseline values use H20 ($\rho=37$) as a conservative lower bound. Details in Appendix J.2.

settings. We retain the 20% latency tolerance and controlled routing patterns from the module-level study. Detailed configurations are in Appendix G.

5.1 Dense Model Validation

As shown in Figures 4a–4b and 29, the measured boundaries closely match the principle-predicted boundaries across both DLLM and AR settings. Full cross-platform results are provided in Appendix H.1. This confirms that, for dense models, the module-level Dense FFN and Attention constraints compose into an accurate full-model NFP predictor regardless of the decoding paradigm.

5.2 MoE Model Validation

As shown in Figures 4c–4d and 33–37, the measured boundaries follow the predicted trend across both DLLM and AR settings. Full cross-platform and sequence-length results are provided in Appendix H.2. At shorter sequence lengths, the boundary is primarily limited by the MoE FFN granularity, whereas at longer sequence lengths the limiting factor shifts to the Attention backend. These results show that the NFP principle captures not only the boundary value but also the dominant limiting module in full MoE model inference, consistent across decoding paradigms.

6 Implications

The NFP principle turns near-free parallelism from an empirical observation into a predictive system-side budget for parallel decoding: given a deployment configuration, it estimates how many decode positions can be executed at near-free latency—without predicting end-to-end speedup of any specific algorithm. First, this budget directly guides parallelism selection—speculative verification length, MTP prediction length, or diffusion-style block size. The analytical minimum determines this budget: instantiating each module-level bound of Section 4 from the model, hardware, and backend configuration and taking the smallest yields the binding bound, which caps the practical decoding length. Each bound is a closed-form function of the configuration, so the budget—and the module that binds it—is known before deployment, without profiling. Table 1 instantiates this procedure as a deployment lookup: in over half of the evaluated settings, the idle-compute intuition over-predicts the realized boundary by 2.4–23×. Second, NFP enables capacity-normalized evaluation. End-to-end speedup conflates system-side capacity with algorithm-side utilization; measuring NFP separates these factors, distinguishing whether limited speedup stems from insufficient system capacity or underutilized positions (additional implications in Appendix J.2).

7 Related Work

7.1 Parallel Decoding

Speculative decoding generates candidate tokens with a draft mechanism and verifies them in parallel using the target model (Stern et al., 2018; Leviathan et al., 2023; Chen et al., 2023; Cai et al., 2024; Li et al., 2024b,a; Miao et al., 2024). Multi-token prediction methods predict multiple positions from one forward pass (Gloeckle et al., 2024; Liu et al., 2024; Fu et al., 2024; Kou et al., 2024). Diffusion language models and masked iterative generation expose parallelism by updating multiple positions simultaneously (Austin et al., 2021; Lou et al., 2023; Sahoo et al., 2024; Gat et al., 2024; Nie et al., 2025; Ye et al., 2025a; Cheng et al., 2025). Despite differing mechanisms, all share the same system-level structure: a single decode forward processes multiple decode positions. This work addresses the shared system-side

question—how many positions can a decode forward process at near-free latency—independent of algorithm-specific token acceptance.

7.2 Performance Modeling for LLM Inference

Prior work analyzes LLM inference performance at multiple levels, including end-to-end latency and throughput (Pope et al., 2023; Aminabadi et al., 2022; Sheng et al., 2023; Patel et al., 2024), batching and scheduling effects (Yu et al., 2022; Kwon et al., 2023; Zhong et al., 2024; Agrawal et al., 2024), memory traffic such as model weights and KV-cache access (Ivanov et al., 2021), and operator- or kernel-level execution behavior (Dao et al., 2022; Dao, 2024; Ye et al., 2025b). Among these, roofline-style resource-balance analysis (Williams et al., 2009; Yuan et al., 2024) is commonly used to explain whether a workload is limited by compute or memory.

Our work adopts this modeling perspective: increasing N can be near-free when it consumes idle compute in memory-bound execution. We further show that this resource-balance view is incomplete: implementation-level granularity, such as padding and tiling, can also determine the realized boundary. The NFP principle therefore treats the roofline bound as one term of a unified min-of-bounds predictor, alongside these granularity bounds.

8 Conclusion

This paper introduced Near-Free Parallelism (NFP) as a system-side capacity boundary for parallel decoding. Rather than treating the latency behavior of multi-position execution as an empirical artifact, we showed that this boundary can be explained and predicted from the interaction between model structure, hardware balance, and implementation granularity. Our analysis reveals that NFP is governed by two sources of slack: memory-bound idle compute and kernel-granularity slack introduced by practical implementations. Building on these mechanisms, we established a Near-Free Parallelism principle and validated that it predicts full-model NFP boundaries across representative Dense and MoE DLLMs, serving stacks, and hardware platforms.

Beyond explaining existing behavior, the NFP principle provides a design signal for future parallel decoding systems. It can guide how many positions algorithms should expose, how runtimes and kernels should absorb them efficiently, and where future hardware scaling can expand low-latency parallel capacity. Overall, NFP offers a unified systems lens for co-designing decoding algorithms, inference implementations, and hardware around the parallelism that is actually near-free.

Limitations

Experimental conditions. The module-level analysis uses single-GPU execution, small batch sizes, and controlled routing patterns. These choices isolate the target mechanisms but limit direct coverage: the Dense FFN boundary scales as $1/b$ under larger batches (validated in our sweep), real MoE routing falls between the analyzed balanced and skewed bounds, and multi-GPU settings introduce communication overlap that requires separate modeling.

Threshold and implementation versions. The NFP boundary is extracted using a fixed 20% latency tolerance, and the granularity parameters are tied to the evaluated framework versions and BF16 precision. These choices affect the extracted boundary values but not the mechanism identification; a sensitivity sweep across $\epsilon \in [0.05, 0.30]$ confirms this robustness (Appendix I). Updating to newer backends or precisions requires only re-querying kernel configuration rules.

Execution model. The current analysis assumes sequential module execution without inter-layer overlap, and does not account for potential effects of kernel fusion or cross-layer pipelining.

References

Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav Gulavani, Alexey Tumanov, and Ramachandran Ramjee. 2024. Taming {Throughput-Latency} tradeoff in {LLM} inference with {Sarathi-Serve}. In *18th USENIX symposium on operating systems design and implementation (OSDI 24)*, pages 117–134.

Reza Yazdani Aminabadi, Samyam Rajbhandari, Ammar Ahmad Awan, Cheng Li, Du Li, Elton Zheng, Olatunji Ruwase, Shaden Smith, Minjia Zhang, Jeff Rasley, and 1 others. 2022. Deepspeed-inference: enabling efficient inference of transformer models at unprecedented scale. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15. IEEE.

Jacob Austin, Daniel D Johnson, Jonathan Ho, Daniel Tarlow, and Rianne Van Den Berg. 2021. Structured denoising diffusion models in discrete state-spaces. *Advances in neural information processing systems*, 34:17981–17993.

Tiwei Bie, Maosong Cao, Xiang Cao, Bingsen Chen, Fuyuan Chen, Kun Chen, Lun Du, Daozhuo Feng,

Haibo Feng, Mingliang Gong, and 1 others. 2026. Llada2. 1: Speeding up text diffusion via token editing. *arXiv preprint arXiv:2602.08676*.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, and 1 others. 2020. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901.

Tianle Cai, Yuhong Li, Zhengyang Geng, Hongwu Peng, Jason D Lee, Deming Chen, and Tri Dao. 2024. Medusa: Simple llm inference acceleration framework with multiple decoding heads. *arXiv preprint arXiv:2401.10774*.

Charlie Chen, Sebastian Borgeaud, Geoffrey Irving, Jean-Baptiste Lespiau, Laurent Sifre, and John Jumper. 2023. Accelerating large language model decoding with speculative sampling. *arXiv preprint arXiv:2302.01318*.

Shuang Cheng, Yihan Bian, Dawei Liu, Linfeng Zhang, Qian Yao, Zhongbo Tian, Wenhai Wang, Qipeng Guo, Kai Chen, Biqing Qi, and 1 others. 2025. Sdar: A synergistic diffusion-autoregression paradigm for scalable sequence generation. *arXiv preprint arXiv:2510.06303*.

Tri Dao. 2024. Flashattention-2: Faster attention with better parallelism and work partitioning. In *International Conference on Learning Representations*, volume 2024, pages 35549–35562.

Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in neural information processing systems*, 35:16344–16359.

Yichao Fu, Peter Bailis, Ion Stoica, and Hao Zhang. 2024. Break the sequential dependency of llm inference using lookahead decoding. *arXiv preprint arXiv:2402.02057*.

Itai Gat, Tal Remez, Neta Shaul, Felix Kreuk, Ricky TQ Chen, Gabriel Synnaeve, Yossi Adi, and Yaron Lipman. 2024. Discrete flow matching. *Advances in Neural Information Processing Systems*, 37:133345–133385.

Fabian Gloeckle, Badr Youbi Idrissi, Baptiste Rozière, David Lopez-Paz, and Gabriel Synnaeve. 2024. Better & faster large language models via multi-token prediction. *arXiv preprint arXiv:2404.19737*.

Andrei Ivanov, Nikoli Dryden, Tal Ben-Nun, Shigang Li, and Torsten Hoefler. 2021. Data movement is all you need: A case study on optimizing transformers. *Proceedings of Machine Learning and Systems*, 3:711–732.

Albert Q Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Emma Bou Hanna, Florian Bressand, and 1 others. 2024. Mixtral of experts. *arXiv preprint arXiv:2401.04088*.

Sehoon Kim, Coleman Hooper, Thanakul Wattanawong, Minwoo Kang, Ruohan Yan, Hasan Genc, Grace Dinh, Qijing Huang, Kurt Keutzer, Michael W Mahoney, and 1 others. 2023. Full stack optimization of transformer inference: a survey. *arXiv preprint arXiv:2302.14017*.

Siqi Kou, Lanxiang Hu, Zhezhi He, Zhijie Deng, and Hao Zhang. 2024. Cllms: Consistency large language models. In *Forty-first International Conference on Machine Learning*.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th symposium on operating systems principles*, pages 611–626.

Yaniv Leviathan, Matan Kalman, and Yossi Matias. 2023. Fast inference from transformers via speculative decoding. In *International Conference on Machine Learning*, pages 19274–19286. PMLR.

Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. 2024a. Eagle-2: Faster inference of language models with dynamic draft trees. In *Proceedings of the 2024 conference on empirical methods in natural language processing*, pages 7421–7432.

Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. 2024b. Eagle: Speculative sampling requires rethinking feature uncertainty. *arXiv preprint arXiv:2401.15077*.

Aiwei Liu, Minghua He, Shaoxun Zeng, Sijun Zhang, Linhao Zhang, Chuhan Wu, Wei Jia, Yuan Liu, Xiao Zhou, and Jie Zhou. 2025. Wedlm: Reconciling diffusion language models with standard causal attention for fast inference. *arXiv preprint arXiv:2512.22737*.

Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, and 1 others. 2024. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*.

Aaron Lou, Chenlin Meng, and Stefano Ermon. 2023. Discrete diffusion modeling by estimating the ratios of the data distribution. *arXiv preprint arXiv:2310.16834*.

Xupeng Miao, Gabriele Oliaro, Zhihao Zhang, Xinhao Cheng, Zeyu Wang, Zhengxin Zhang, Rae Ying Yee Wong, Alan Zhu, Lijie Yang, Xiaoxiang Shi, and 1 others. 2024. Specinfer: Accelerating large language model serving with tree-based speculative inference and verification. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 932–949.

Shen Nie, Fengqi Zhu, Zebin You, Xiaolu Zhang, Jingyang Ou, Jun Hu, JUN ZHOU, Yankai Lin, Ji-Rong Wen, and Chongxuan Li. 2025. Large language diffusion models. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*.

Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. 2024. Splitwise: Efficient generative llm inference using phase splitting. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, pages 118–132. IEEE.

Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. 2023. Efficiently scaling transformer inference. *Proceedings of machine learning and systems*, 5:606–624.

Subham S Sahoo, Marianne Arriola, Yair Schiff, Aaron Gokaslan, Edgar Marroquin, Justin T Chiu, Alexander Rush, and Volodymyr Kuleshov. 2024. Simple and effective masked diffusion language models. *Advances in Neural Information Processing Systems*, 37:130136–130184.

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

Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. 2023. Flexgen: High-throughput generative inference of large language models with a single gpu. In *International Conference on Machine Learning*, pages 31094–31116. PMLR.

Mitchell Stern, Noam Shazeer, and Jakob Uszkoreit. 2018. Blockwise parallel decoding for deep autoregressive models. *Advances in Neural Information Processing Systems*, 31.

Ling Team, Ang Li, Ben Liu, Binbin Hu, Bing Li, Bingwei Zeng, Borui Ye, Caizhi Tang, Changxin Tian, Chao Huang, and 1 others. 2025. Every activation boosted: Scaling general reasoner to 1 trillion open language foundation. *arXiv preprint arXiv:2510.22115*.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, and 1 others. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*.

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

Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76.

Heming Xia, Zhe Yang, Qingxiu Dong, Peiyi Wang, Yongqi Li, Tao Ge, Tianyu Liu, Wenjie Li, and Zhifang Sui. 2024. Unlocking efficiency in large language model inference: A comprehensive survey of speculative decoding. *Findings of the Association for Computational Linguistics: ACL 2024*, pages 7655–7671.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, and 1 others. 2025. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*.

Jiacheng Ye, Zhihui Xie, Lin Zheng, Jiahui Gao, Zirui Wu, Xin Jiang, Zhenguo Li, and Lingpeng Kong. 2025a. Dream 7b: Diffusion large language models. *arXiv preprint arXiv:2508.15487*.

Zihao Ye, Lequn Chen, Ruihang Lai, Wuwei Lin, Yineng Zhang, Stephanie Wang, Tianqi Chen, Baris Kasikci, Vinod Grover, Arvind Krishnamurthy, and 1 others. 2025b. Flashinfer: Efficient and customizable attention engine for llm inference serving. *Proceedings of Machine Learning and Systems*, 7.

Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. 2022. Orca: A distributed serving system for {Transformer-Based} generative models. In *16th USENIX symposium on operating systems design and implementation (OSDI 22)*, pages 521–538.

Zhihang Yuan, Yuzhang Shang, Yang Zhou, Zhen Dong, Zhe Zhou, Chenhao Xue, Bingzhe Wu, Zhikai Li, Qingyi Gu, Yong Jae Lee, and 1 others. 2024. Llm inference unveiled: Survey and roofline model insights. *arXiv preprint arXiv:2402.16363*.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody H Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, and 1 others. 2024. Sglang: Efficient execution of structured language model programs. *Advances in neural information processing systems*, 37:62557–62583.

Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. 2024. {DistServe}: Disaggregating prefill and decoding for goodput-optimized large language model serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 193–210.

Appendix

Contents

A Notation and Definitions **13**

B Theoretical Baseline Derivations **15**
- B.1 MoE FFN Idle-Compute Baseline Derivation 15
- B.2 Attention Idle-Compute Baseline Derivation 16

C Module-Level Analysis Implementation Details **17**
- C.1 Overall Settings 17
- C.2 Dense FFN Layers Implementation Details 19
- C.3 MoE FFN Layers Implementation Details 19
- C.4 Attention Layers Implementation Details 20

D Module-Level Analysis Results **21**
- D.1 Dense FFN Layers Results 21
- D.2 MoE FFN Layers Results 23
- D.3 Attention Layers Results 29

E Implementation Evidence for MoE Expert-Token Padding **33**
- E.1 vLLM Fused-MoE Expert-Token Alignment 33
- E.2 SGLang Fused-MoE Expert-Token Alignment 35
- E.3 From MoE Expert-Token Alignment to MoE NFP Boundaries 37

F Implementation Evidence for Attention Query-Tile Granularity **39**
- F.1 FlashAttention Query-Tile Scheduling 39
- F.2 FlashInfer Query-Tile Scheduling 41
- F.3 From Query-Tile Granularity to Attention NFP Boundaries 43

G Model-Level Validation Implementation Details **46**
- G.1 Overall Validation Protocol 46
- G.2 Dense Model Validation Implementation Details 46
- G.3 MoE Model Validation Implementation Details 47

H Model-Level Validation Results **48**
- H.1 Dense Model Results 48
- H.2 MoE Model Results 50

I Sensitivity to Tolerance Threshold **54**

J Discussion and Implications **57**
- J.1 Revisiting the Idle-Compute Baseline 57
- J.2 Implications 58

A Notation and Definitions

Symbol	Definition
	Parallel Decoding Notation
t	Decode position index.
θ	Model parameters.
$F_\theta(\cdot)$	Model forward function parameterized by θ.
z_t	Input representation at decode position t.
ℓ_t	Output logits at decode position t.
$Z_{t:t+N-1}$	Inputs placed at the N decode positions from t to $t+N-1$.
$\boldsymbol{\ell}_{t:t+N-1}$	Output logits for the N decode positions from t to $t+N-1$.
N	Number of decode positions processed in one forward pass. $N=1$ corresponds to standard autoregressive decoding.
$T(N)$	Latency of one decode forward that processes N decode positions.
$T(1)$	Latency of the single-position autoregressive baseline.
ϵ	Latency tolerance used to define near-free execution.
$N_{\max}(\epsilon)$	Maximum number of decode positions whose latency remains within $(1+\epsilon)T(1)$.
$N_{\max}$	Observed near-free parallelism boundary. In the experiments, $N_{\max}=N_{\max}(0.2)$.
	Shared Notation and Idle-Compute Baseline Notation
b	Batch size.
d_{model}	Model hidden dimension.
d_{ff}	FFN intermediate dimension.
s	Number of bytes per element.
$C(N)$	Computation cost of a decode forward with N positions.
$B(N)$	Data movement, or memory traffic, of a decode forward with N positions.
$AI(N)$	Arithmetic intensity, defined as $AI(N)=C(N)/B(N)$.
ϕ	peak hardware compute throughput.
β	peak hardware memory bandwidth.
ρ	Hardware balance point, defined as $\rho=\phi/\beta$.
N_{idle}	Near-free boundary predicted by the idle-compute baseline, obtained from $AI(N_{\text{idle}})=\rho$.
	Dense FFN Notation
$C_{\text{dense}}(N)$	Computation cost of a Dense FFN layer with N decode positions.
$B_{\text{dense}}(N)$	Dominant memory traffic of a Dense FFN layer with N decode positions.
$AI_{\text{dense}}(N)$	Arithmetic intensity of a Dense FFN layer.
$N_{\text{idle}}^{\text{dense}}$	Idle-compute-predicted near-free boundary for Dense FFN layers.
$N_{\max}^{\text{dense}}$	Principle-predicted model-level near-free boundary for dense models.

Symbol	Definition
	MoE FFN Notation
k	Number of selected experts per token in top-k routing.
E	Total number of experts.
E_{act}	Number of distinct experts activated by a routing pattern.
η	Implementation-dependent factor capturing additional read-modify-write overhead in the MoE combine stage.
$C_{\mathrm{moe}}(N)$	Computation cost of a MoE FFN layer with N decode positions.
$B_{\mathrm{moe},W}(N)$	Memory traffic from loading active expert weights.
$B_{\mathrm{moe},A}(N)$	Routing-induced activation memory traffic from dispatch and combine.
$B_{\mathrm{moe}}(N)$	Total dominant memory traffic of a MoE FFN layer.
$AI_{\mathrm{moe}}(N)$	Arithmetic intensity of a MoE FFN layer.
$N_{\mathrm{idle}}^{\mathrm{moe}}$	Idle-compute-predicted near-free boundary for MoE FFN layers.
$N_{\mathrm{max}}^{\mathrm{moe,bal}}$	Principle-predicted model-level near-free boundary for MoE models under load-balanced routing.
$N_{\mathrm{max}}^{\mathrm{moe,skew}}$	Principle-predicted model-level near-free boundary for MoE models under load-skewed routing.
M_{moe}	Expert-token padding granularity of the MoE implementation.
τ	Validity bound of the backend branch that selects M_{moe}.
	Attention Notation
L	Cached sequence length in attention.
d_{kv}	KV hidden dimension. For standard multi-head attention, $d_{\mathrm{kv}} = d_{\mathrm{model}}$.
$C_{\mathrm{attn}}(N)$	Computation cost of an attention layer with N query positions.
$B_{\mathrm{attn}}(N)$	KV-cache-dominated memory traffic of an attention layer with N query positions.
$AI_{\mathrm{attn}}(N)$	Arithmetic intensity of an attention layer.
$N_{\mathrm{idle}}^{\mathrm{attn}}$	Idle-compute-predicted near-free boundary for attention layers.
M_{attn}	Maximum number of query positions that can be absorbed within one backend-specific attention execution granularity.

Table 2: Notation and definitions used throughout the paper.

B Theoretical Baseline Derivations

The Dense FFN idle-compute baseline derivation is presented in the main text. This section provides the full derivations for MoE FFNs and Attention layers.

B.1 MoE FFN Idle-Compute Baseline Derivation

We instantiate the idle-compute baseline for MoE FFNs. Let k denote the number of selected experts per token, E_{act} the number of distinct experts activated by a routing pattern, and $\eta = 2$ the number of per-expert activation accesses in the combine stage.[2] A MoE FFN processes bN positions, each routed to k experts. Since each expert is implemented as a two-GEMM FFN, the total computation is approximated as

$$C_{\text{moe}}(N) \approx 4bNkd_{\text{model}}d_{\text{ff}}. \tag{15}$$

Unlike Dense FFNs, the dominant memory traffic in MoE FFNs consists of both active expert weights and routing-induced activation movement. The weight traffic is proportional to the number of activated experts,

$$B_{\text{moe},W}(N) \approx 2E_{\text{act}}d_{\text{model}}d_{\text{ff}}s, \tag{16}$$

while dispatch and combine move activations between tokens and experts. We approximate this routing-induced activation traffic as

$$B_{\text{moe},A}(N) \approx bNd_{\text{model}}s(1+3k+\eta k). \tag{17}$$

Combining weight and activation traffic yields the arithmetic intensity

$$AI_{\text{moe}}(N) \approx \frac{4bNkd_{\text{ff}}}{s(2E_{\text{act}}d_{\text{ff}}+bN(1+3k+\eta k))}. \tag{18}$$

Substituting this intensity into the idle-compute baseline condition yields

$$N_{\text{idle}}^{\text{moe}} \approx \frac{2\rho sE_{\text{act}}d_{\text{ff}}}{b(4kd_{\text{ff}}-\rho s(1+3k+\eta k))}, \tag{19}$$

when $4kd_{\text{ff}} > \rho s(1+3k+\eta k)$; otherwise, the execution remains memory-bound under this abstraction.

[2] One read of each expert output and one write for weighted accumulation.

B.2 Attention Idle-Compute Baseline Derivation

We instantiate the idle-compute baseline for Attention layers. Let L denote the cached sequence length, N the number of decode positions per request, d_{kv} the KV hidden dimension, and s the number of bytes per element. For standard MHA, $d_{\mathrm{kv}} = d_{\mathrm{model}}$. Under a KV-cache-dominated abstraction, a decode attention layer processes bN query positions over a cache of length L. The dominant computation comes from the attention score computation and value aggregation, while the dominant memory traffic comes from accessing the KV cache:

$$\begin{aligned} C_{\mathrm{attn}}(N) &\approx 4bNLd_{\mathrm{kv}}, \\ B_{\mathrm{attn}}(N) &\approx 2b(L+N)d_{\mathrm{kv}}s. \end{aligned} \tag{20}$$

The key asymmetry is that computation scales with the number of query-cache interactions NL, while the dominant KV-cache traffic is primarily controlled by the cached sequence length. The AI is therefore

$$AI_{\mathrm{attn}}(N) = \frac{C_{\mathrm{attn}}(N)}{B_{\mathrm{attn}}(N)} \approx \frac{2NL}{(L+N)s}. \tag{21}$$

Substituting this intensity into the idle-compute baseline condition yields

$$N_{\mathrm{idle}}^{\mathrm{attn}} \approx \begin{cases} \frac{\rho s L}{2L-\rho s}, & 2L > \rho s, \\ +\infty, & 2L \leq \rho s. \end{cases} \tag{22}$$

This result predicts that the NFP boundary of Attention layers is primarily governed by the cached sequence length L. When L is small, the attention core remains memory-bound over a wide range of N. When L is sufficiently large, the boundary approaches a finite value determined by the compute-to-memory balance, and increasing N is near-free only until it exhausts the idle compute exposed by KV-cache-dominated execution.

C Module-Level Analysis Implementation Details

C.1 Overall Settings

C.1.1 Experimental Goal and Scope

The goal is not to benchmark complete models, but to measure the NFP boundary of individual modules under controlled configurations. This module-level analysis serves as the first step toward the model-level NFP principle: by isolating the boundary of each major module, we can later identify the limiting module and compose module-level constraints into full-model predictions. We consider the same three module classes as in the main analysis: Dense FFNs, MoE FFNs, and Attention. The scope of the module-level experiments is summarized as follows:

- **Dense FFN layers**: isolated two-GEMM FFN operators used to study resource slack and the idle-compute baseline.
- **MoE FFN layers**: fused MoE operators with controlled routing patterns used to study expert-token padding and kernel-granularity slack.
- **Attention layers**: decode attention kernels over a pre-allocated KV cache used to study backend-specific query granularity.

C.1.2 Common NFP Measurement Protocol

For each module and configuration, we vary only the number of decode positions N and keep all other factors fixed, including the module shape, hardware platform, inference backend, and numerical precision. We denote by $T(N)$ the measured latency of one module forward that processes N decode positions under this fixed configuration. The single-position case $N=1$ is used as the autoregressive baseline $T(1)$.

Following the definition in the main text, we define a configuration with N decode positions to be near-free if its latency remains within a 20% increase over the single-position baseline:

$$T(N) \leq 1.2T(1), \tag{23}$$

The observed NFP boundary is then extracted as

$$N_{\max} = N_{\max}(0.2) = \max\{N \in \mathcal{N} : T(N) \leq 1.2T(1)\}. \tag{24}$$

where N denotes the set of sampled decode-position counts in the sweep.

This protocol turns the visual notion of a flat latency region into a uniform measurement rule. The 20% tolerance captures a practical low-overhead regime rather than exact latency equality, while reducing sensitivity to GPU timing noise, kernel launch jitter, and minor backend-dependent latency discontinuities. We use this threshold only as an operational criterion for extracting a boundary from each latency curve; the mechanism analysis further relies on the latency trend, profiling metrics, and comparison with the idle-compute baseline. We apply the same threshold and extraction rule to all modules, hardware platforms, and inference backends, so that the measured boundaries can be compared consistently with the idle-compute baseline and the model-level NFP principle.

C.1.3 Latency Measurement Procedure

For each sampled value of N, we measure the steady-state latency of the corresponding module forward. Before timing, all input tensors, output buffers, routing tensors, and KV-cache buffers required by the module are pre-allocated on the GPU, so that the reported latency excludes memory allocation, data movement from the host, and other setup overheads. Each configuration is first warmed up for 50 iterations to trigger CUDA context initialization, kernel loading or compilation, backend autotuning, and cache stabilization.

After warm-up, we perform 10 independent measurement rounds. Each round contains 200 timed iterations of the same module forward under the fixed configuration. Latency is measured using CUDA events with explicit device synchronization, which avoids artifacts from asynchronous GPU execution. For each configuration, we compute the median latency within each measurement round and report the median across the 10 rounds as $T(N)$. This median-based procedure provides a stable estimate of the steady-state module latency and reduces the influence of occasional timing spikes when extracting the NFP boundary.

C.1.4 Hardware Platforms

We evaluate all module-level experiments on three GPU platforms: NVIDIA A800, H800, and H20. These platforms cover different architectures and compute-to-memory balance points, allowing us to test whether the measured NFP boundary follows the hardware-dependent resource-balance prediction or is instead dominated by backend-specific kernel granularity. The same set of hardware platforms is used for Dense FFN, MoE FFN, and Attention experiments to ensure that the observed trends are not artifacts of a single GPU generation or hardware configuration. All experiments are conducted on a single GPU to avoid potential confounding effects from inter-GPU communication.

Table 3 summarizes the hardware platforms. We use ϕ to denote the peak FP16 compute throughput and β to denote the peak memory bandwidth. The hardware balance point is computed as $\rho = \phi/\beta$. All module-level experiments are executed in BF16 precision. We use the same numerical precision across modules and platforms to avoid precision-dependent differences in kernel selection and throughput, while matching a common precision setting for LLM inference.

GPU	Release Date	Architecture	ϕ (FP16/BF16 TFLOP/s)	β (TB/s)	$\rho = \phi/\beta$
H20	2024	Hopper	148	4.0	37.0
A800	2022	Ampere	312	2.039	153.0
H800	2023	Hopper	989	3.35	295.2

Table 3: Hardware platforms used in the module-level experiments. ϕ denotes peak FP16/BF16 compute throughput, β denotes peak memory bandwidth, and $\rho = \phi/\beta$ is the compute-to-memory balance point.

C.1.5 Profiling Details

We profile the measured forward region using NVIDIA Nsight Compute in roofline mode. The profiled region is delimited by CUDA profiler start and stop markers, and includes all GPU kernels launched by the corresponding module forward. Kernel-level counters are aggregated into forward-level metrics using kernel execution time as the weight.

For compute-side profiling, we use dtype-matched tensor-core throughput counters according to the numerical precision of the experiment, i.e., BF16 or FP16. The reported compute utilization is obtained by normalizing the time-weighted achieved tensor-core throughput by the corresponding peak tensor-core throughput. We also derive the observed runtime FLOPs from the achieved tensor-core throughput and kernel execution time, normalized by the number of profiled forward executions.

For memory-side profiling, we use DRAM throughput counters. The observed DRAM traffic per forward is computed from the measured DRAM byte throughput and kernel execution time, again normalized by the number of profiled forward executions. When the profiler directly reports percentage-of-peak DRAM throughput, we report the time-weighted average as DRAM bandwidth utilization. The observed arithmetic intensity is computed as the ratio between observed runtime FLOPs and observed DRAM traffic. These metrics are used to interpret whether the measured NFP boundary is dominated by resource slack or by implementation-induced kernel-granularity effects.

C.1.6 Reproducibility Controls

We use a fixed random seed, `torch.manual_seed(0)`. All measurements are conducted on pre-allocated GPU tensors and focus only on the measured module forward. Data loading, CPU preprocessing, graph construction, and other host-side setup operations are excluded from the measured latency. First-run initialization and compilation overheads are also excluded through the warm-up procedure described above. These controls ensure that the reported latency and profiling metrics reflect the steady-state execution behavior of the module rather than input-generation or setup overheads.

C.2 Dense FFN Layers Implementation Details

We implement the Dense FFN layer as an isolated two-GEMM operator, matching the module abstraction used in the main analysis. One forward consists of an up-projection followed by a down-projection. We omit activation functions, bias terms, and gating branches, so that the measured latency reflects the dense matrix-computation behavior of the FFN rather than additional elementwise or routing operations. We set $d_{\text{model}} = 4096$ and $d_{\text{ff}} = 9216$, following the hidden size and intermediate size of LLaDA-2.1-Flash, a representative existing DLLM, to make the microbenchmark reflect realistic model dimensions.

The two GEMMs are executed as separate `torch.matmul` operations, using the underlying cuBLAS/cuBLASLt backend. We do not use fused MLP kernels, compiler fusion, Triton kernels, or custom CUTLASS kernels in this benchmark. This design keeps the Dense FFN experiment aligned with the idle-compute baseline and avoids introducing additional implementation-specific granularity effects. All tensors are stored in BF16 precision. The benchmark measures a single FFN layer in isolation, without attention, normalization, residual connections, or layer stacking.

C.3 MoE FFN Layers Implementation Details

We instantiate MoE FFN layers using the fused MoE operators from vLLM v0.9.1 and SGLang v0.5.9. Each MoE forward consists of token dispatch, expert FFN computation, and weighted combine, following the execution path of practical MoE inference backends. Each expert is implemented as a two-GEMM FFN, and the benchmark isolates a single MoE FFN layer without attention, normalization, residual connections, or layer stacking.

We use controlled top-k routing instead of input-dependent gating, so that the expert-load distribution is deterministic and comparable across N, hardware platforms, and backends. We use $E = 256$ experts and sweep k from 2 to 256 selected experts per token. Each expert uses $d_{\text{model}} = 4096$ and $d_{\text{ff}} = 1024$, following the hidden size and MoE intermediate size of LLaDA-2.1-Flash. All MoE experiments are executed in BF16 precision using the same logical workload across vLLM and SGLang. We evaluate both backends on the same set of GPU platforms to obtain cross-stack evidence and to avoid attributing an implementation- or hardware-specific artifact to the general MoE NFP mechanism.

C.3.1 Controlled Routing Patterns

We evaluate two controlled routing patterns. In the load-balanced case, tokens are assigned to experts in a deterministic round-robin manner:

$$\{(i \cdot k + j) \bmod E\}_{j=0}^{k-1} \tag{25}$$

for token i. With $T = bN$ total tokens, this gives $E_{\text{act}} = \min(E, Tk)$. Since $b = 1$ in the MoE experiments, $E_{\text{act}} = \min(E, Nk)$. This case spreads tokens across as many experts as possible.

For the load-balanced case, we do not use $N = 1$ as the latency baseline, because the number of activated experts would still change as N increases. Instead, for each top-k setting, we first increase N until all experts are activated and use this smallest saturated point as the baseline:

$$N_{\text{bal},0}(k) = \left\lceil \frac{E}{bk} \right\rceil. \tag{26}$$

In our experiments, $E = 256$ and $b = 1$, so $N_{\text{bal},0}(k) = \lceil 256/k \rceil$ as k is swept from 2 to 256. The load-balanced NFP boundary is extracted relative to $T(N_{\text{bal},0}(k))$, so that the measured near-free region reflects increasing N under a fixed fully activated expert set rather than the cost of activating more experts.

In the load-skewed case, all tokens are routed to the same k experts, so $E_{\text{act}} = k$ regardless of N. We therefore use the standard single-position baseline for this case. These two patterns represent the upper- and lower-bound expert-load cases used to study how expert-token distribution and fused-kernel padding affect the MoE NFP boundary.

C.4 Attention Layers Implementation Details

We benchmark the decode attention core over a pre-allocated KV cache. Each forward processes N query positions against a cached sequence of length L under causal masking. We omit the QKV projections and output projection, so that the measured latency reflects the KV-cache attention backend rather than additional dense GEMM operations.

For simplicity, we instantiate the attention module as standard multi-head attention (MHA). We use $n_{\text{heads}} = 32$ and $d_{\text{head}} = 128$, giving $d_{\text{model}} = n_{\text{heads}} d_{\text{head}} = 4096$. Since the benchmark uses MHA, the key-value hidden dimension is $d_{\text{kv}} = d_{\text{model}} = 4096$. The same formulation also applies to GQA or MQA settings, where only the KV width changes to $d_{\text{kv}} = n_{\text{kv_heads}} d_{\text{head}}$.

We evaluate two practical decode attention backends: FlashAttention-2 and FlashInfer. The same logical attention workload is used across both backends and all hardware platforms. This allows us to attribute differences in the measured NFP boundary to backend-specific query granularity and kernel implementation behavior rather than changes in model shape or attention semantics. All attention experiments are executed in BF16 precision and isolate a single attention layer without normalization, residual connections, or layer stacking.

D Module-Level Analysis Results

D.1 Dense FFN Layers Results

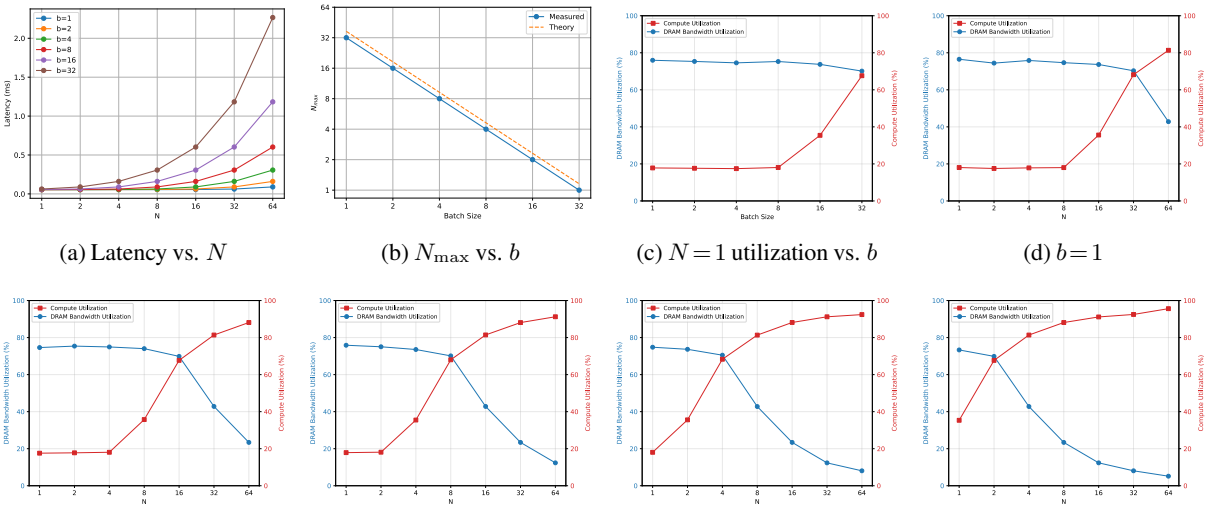
Figures 5–7 report the complete Dense FFN module-level results from single-GPU experiments on NVIDIA H20, A800, and H800, respectively. These experiments complement the main-text analysis by validating the Dense FFN idle-compute mechanism on GPU platforms with different compute-to-memory balance points. In each figure, the first subfigure shows the latency curve as the number of decode positions N increases, the second reports the extracted NFP boundary $N_{\max}$ as a function of batch size b, the third shows the $N=1$ resource utilization across batch sizes, and the remaining subfigures show the utilization trends when sweeping N for each fixed batch size.

On each evaluated GPU, Dense FFNs exhibit a clear near-free region. For each batch size, latency remains nearly flat over an initial range of N and increases only after the available slack is exhausted. The size of this region is batch-size dependent: smaller batches support a larger near-free range, while larger batches reduce the maximum near-free N. This behavior is consistent across H20, A800, and H800.

The extracted boundaries further match the idle-compute prediction. As shown in Figures 5b, 6b, and 7b, $N_{\max}$ decreases approximately inversely with b, following the predicted trend $N_{\text{idle}}^{\text{dense}} \approx \rho s/2b$. The absolute boundary differs across GPU platforms because the amount of available compute slack depends on the hardware balance point ρ. A GPU with larger ρ exposes more compute capacity relative to memory bandwidth and therefore supports a larger Dense FFN near-free region under the same workload.

The utilization profiles explain this trend. At $N=1$, Dense FFN execution is bandwidth-dominated: DRAM bandwidth utilization is already high, while tensor-core utilization remains comparatively low. As N increases, tensor-core utilization rises, indicating that increasing N consumes otherwise idle compute. In contrast, DRAM bandwidth utilization changes more slowly because the dominant weight traffic is already incurred by the single-position baseline. Increasing the batch size raises the initial compute utilization at $N=1$, leaving less remaining compute slack and therefore reducing $N_{\max}$.

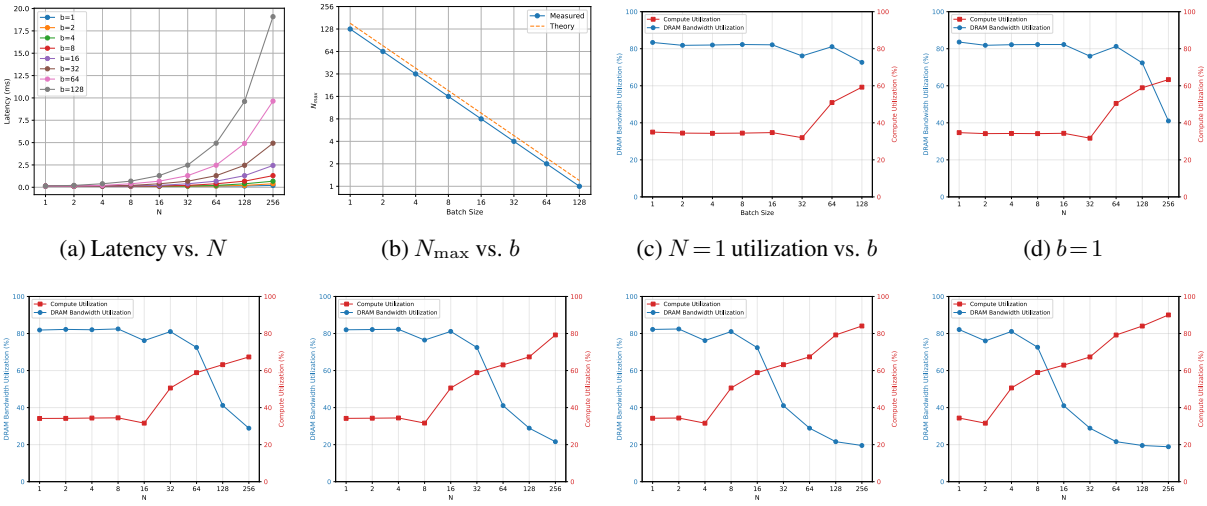
Overall, these single-GPU results confirm that the Dense FFN NFP boundary is primarily governed by memory-bound resource slack rather than implementation-induced kernel granularity. Dense FFNs therefore contribute the hardware-sensitive term $\rho s/2b$ to the model-level NFP principle.



(a) Latency vs. N (b) $N_{\max}$ vs. b (c) $N=1$ utilization vs. b (d) $b=1$

(e) $b=2$ (f) $b=4$ (g) $b=8$ (h) $b=16$

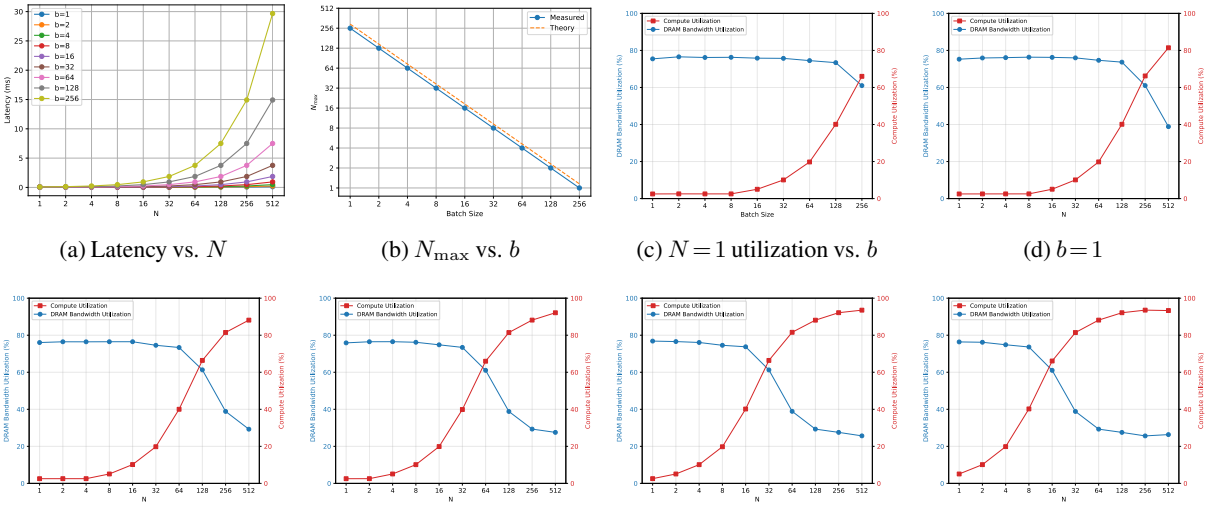
Figure 5: Dense FFN evaluation results on **NVIDIA H20 GPU**.



(a) Latency vs. N (b) $N_{\max}$ vs. b (c) $N=1$ utilization vs. b (d) $b=1$

(e) $b=2$ (f) $b=4$ (g) $b=8$ (h) $b=16$

Figure 6: Dense FFN evaluation results on **NVIDIA A800 GPU**.



(a) Latency vs. N (b) $N_{\max}$ vs. b (c) $N=1$ utilization vs. b (d) $b=1$

(e) $b=2$ (f) $b=4$ (g) $b=8$ (h) $b=16$

Figure 7: Dense FFN evaluation results on **NVIDIA H800 GPU**.

D.2 MoE FFN Layers Results

This section reports the complete MoE FFN module-level results. These experiments complement the main-text analysis by validating the implementation-induced kernel-granularity mechanism across routing patterns, fused MoE backends, and single-GPU platforms. We consider two controlled routing cases: load-balanced routing, which distributes tokens across experts and exposes the upper-bound MoE NFP behavior, and load-skewed routing, which routes all tokens to the same selected experts and exposes the lower-bound behavior. For each case, we evaluate both vLLM and SGLang fused MoE operators on NVIDIA H20, A800, and H800.

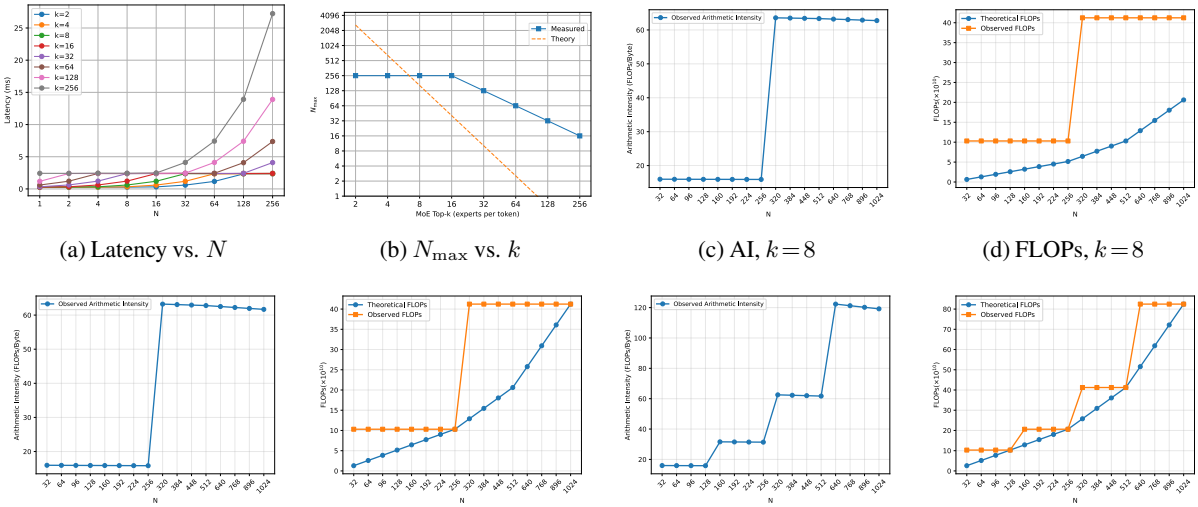
D.2.1 Load-balanced Routing as Upper Bound

Figures 8, 9, and 10 report the load-balanced MoE FFN results using the vLLM fused MoE backend. In this setting, tokens are distributed across experts, so the padding slack of fused expert GEMMs is aggregated over many activated experts. Across the evaluated GPU platforms, the latency curves exhibit clear near-free regions followed by discrete jumps, indicating that larger N remains within existing padded expert-token blocks until the next kernel-granularity boundary is crossed.

The extracted boundaries follow the expected upper-bound trend. As shown in Figures 8b, 9b, and 10b, $N_{\max}$ increases as routing becomes more sparse. This is consistent with the granularity-based upper-bound prediction: smaller k spreads the workload over more available expert-token padding capacity, while larger k consumes more expert computation per token and reduces the maximum near-free N. The arithmetic-intensity and runtime-FLOPs profiles further show staircase-like behavior, confirming that the measured boundary is governed by padded kernel execution rather than a smooth idle-compute transition.

Figures 11, 12, and 13 report the corresponding load-balanced results using the SGLang fused MoE backend. The same qualitative behavior appears under SGLang. The latency curves show near-free plateaus, the extracted $N_{\max}$ values follow the sparsity-dependent upper-bound trend, and the arithmetic-intensity and FLOPs profiles change in discrete steps as N crosses backend granularity boundaries. As shown in Figures 11b, 12b, and 13b, this trend is consistent across all evaluated GPU platforms.

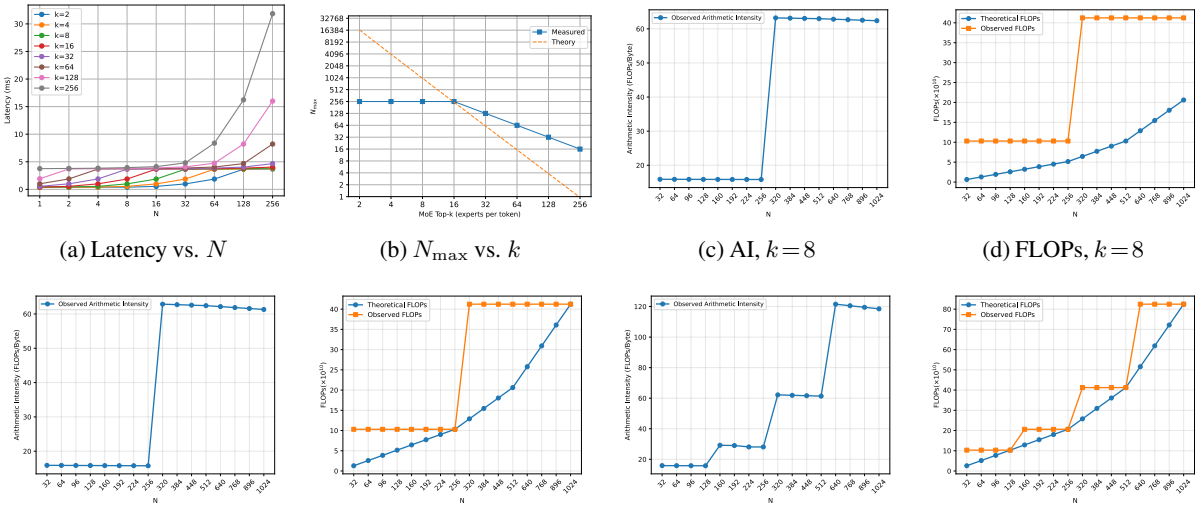
Overall, the load-balanced results show that MoE FFN NFP in the upper-bound case is primarily determined by aggregate expert-token padding slack. The agreement between vLLM and SGLang indicates that this behavior is not specific to one backend, but arises from the common granularity structure of fused MoE execution.



(a) Latency vs. N (b) $N_{\max}$ vs. k (c) AI, $k=8$ (d) FLOPs, $k=8$

(e) AI, $k=16$ (f) FLOPs, $k=16$ (g) AI, $k=32$ (h) FLOPs, $k=32$

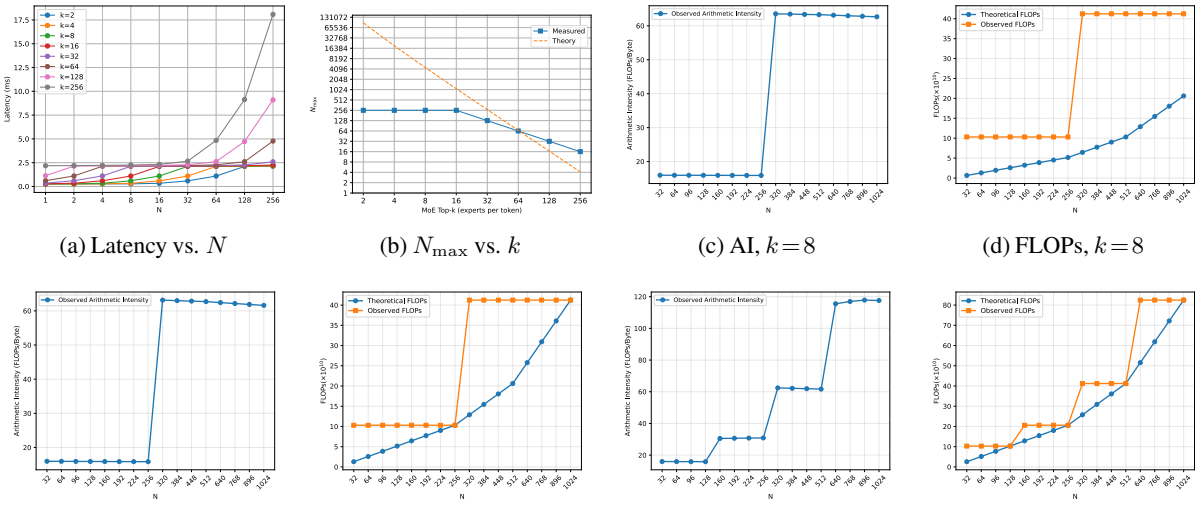
Figure 8: MoE FFN evaluation for load-balanced routing, the upper-bound case, with **vLLM** on **NVIDIA H20**.



(a) Latency vs. N (b) $N_{\max}$ vs. k (c) AI, $k=8$ (d) FLOPs, $k=8$

(e) AI, $k=16$ (f) FLOPs, $k=16$ (g) AI, $k=32$ (h) FLOPs, $k=32$

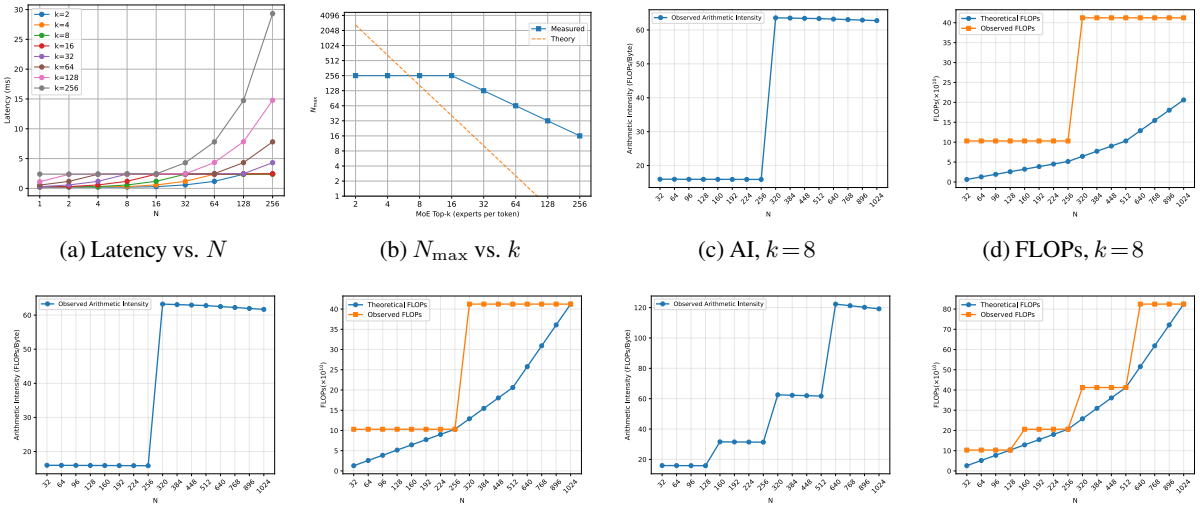
Figure 9: MoE FFN evaluation for load-balanced routing, the upper-bound case, with **vLLM** on **NVIDIA A800**.



(a) Latency vs. N (b) $N_{\max}$ vs. k (c) AI, $k=8$ (d) FLOPs, $k=8$

(e) AI, $k=16$ (f) FLOPs, $k=16$ (g) AI, $k=32$ (h) FLOPs, $k=32$

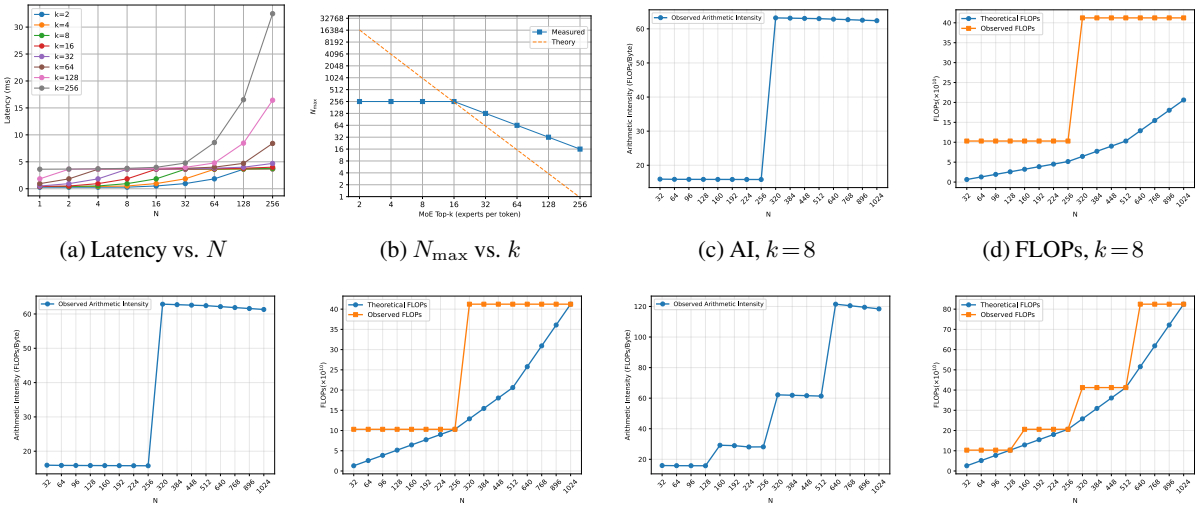
Figure 10: MoE FFN evaluation for load-balanced routing, the upper-bound case, with **vLLM** on **NVIDIA H800**.



(a) Latency vs. N (b) $N_{\max}$ vs. k (c) AI, $k=8$ (d) FLOPs, $k=8$

(e) AI, $k=16$ (f) FLOPs, $k=16$ (g) AI, $k=32$ (h) FLOPs, $k=32$

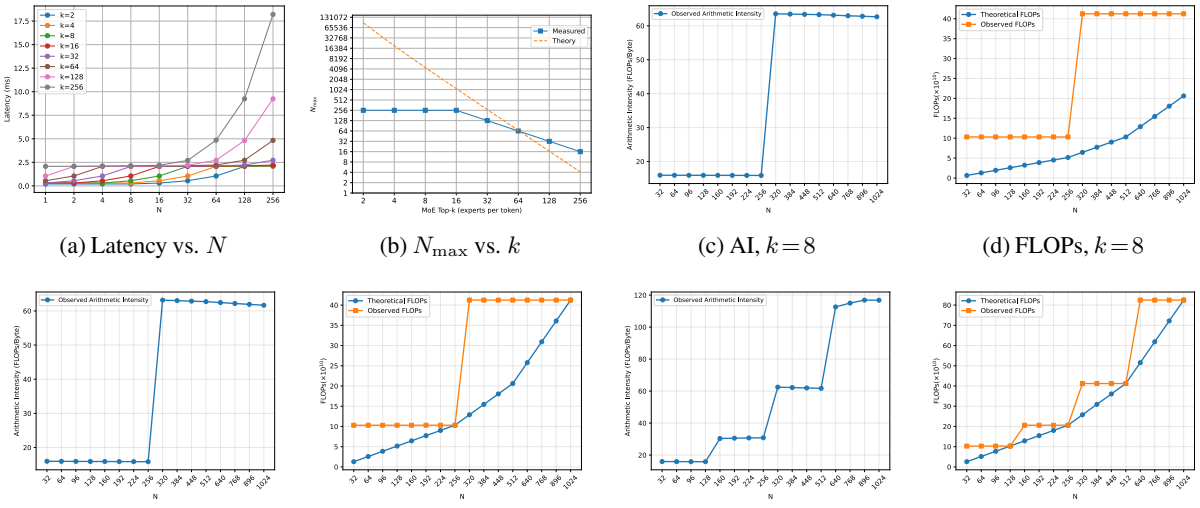
Figure 11: MoE FFN evaluation for load-balanced routing, the upper-bound case, with **SGLang** on **NVIDIA H20**.



(a) Latency vs. N (b) $N_{\max}$ vs. k (c) AI, $k=8$ (d) FLOPs, $k=8$

(e) AI, $k=16$ (f) FLOPs, $k=16$ (g) AI, $k=32$ (h) FLOPs, $k=32$

Figure 12: MoE FFN evaluation for load-balanced routing, the upper-bound case, with **SGLang** on **NVIDIA A800**.



(a) Latency vs. N (b) $N_{\max}$ vs. k (c) AI, $k=8$ (d) FLOPs, $k=8$

(e) AI, $k=16$ (f) FLOPs, $k=16$ (g) AI, $k=32$ (h) FLOPs, $k=32$

Figure 13: MoE FFN evaluation for load-balanced routing, the upper-bound case, with **SGLang** on **NVIDIA H800**.

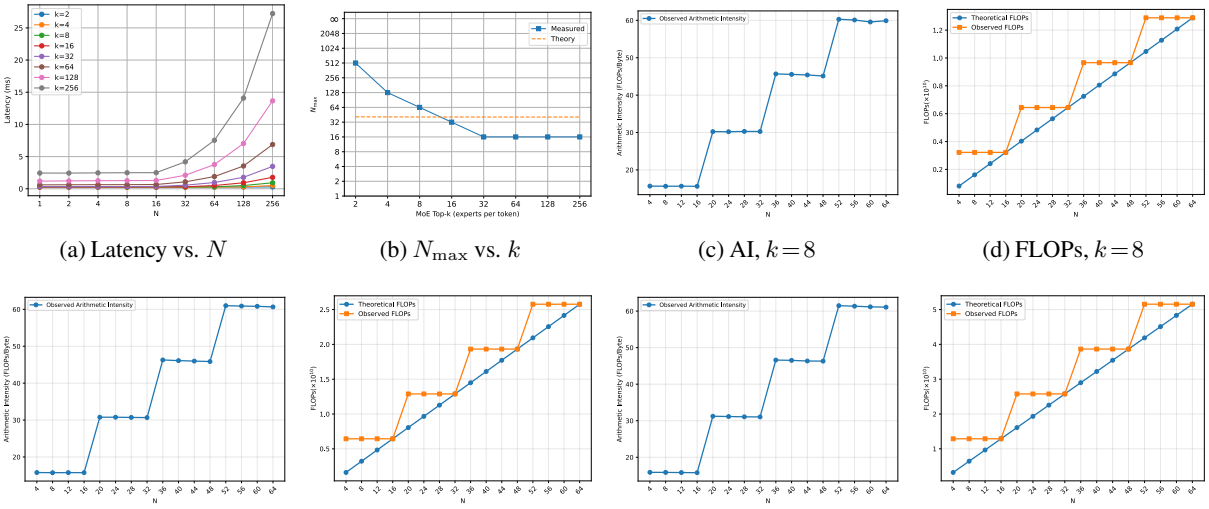
D.2.2 Load-skewed Routing as Lower Bound

Figures 14, 15, and 16 report the load-skewed MoE FFN results using the vLLM fused MoE backend. In this setting, all tokens are routed to the same selected experts, so larger N can only exploit padding slack within a fixed set of expert-token blocks. The latency curves still exhibit near-free regions with staircase-like transitions, but the boundary is much smaller than in the load-balanced case. This is because load-skewed routing does not create aggregate slack across many activated experts.

The extracted boundaries confirm the lower-bound behavior. Unlike the load-balanced case, increasing sparsity does not expose additional expert-distributed padding capacity. Instead, $N_{\max}$ is largely fixed by the backend's expert-token padding granularity. The arithmetic-intensity and runtime-FLOPs profiles show discrete changes as N crosses granularity boundaries, again indicating that the observed near-free region is produced by padded fused-kernel execution rather than by the idle-compute baseline.

Figures 17, 18, and 19 report the corresponding load-skewed results using the SGLang fused MoE backend. The same lower-bound behavior appears under SGLang: latency remains nearly flat within a granularity block and increases after the block is exceeded, while the extracted boundary remains primarily determined by the single expert-token padding granularity. This cross-backend consistency confirms that load-skewed routing exposes the granularity-fixed lower bound of MoE FFN NFP.

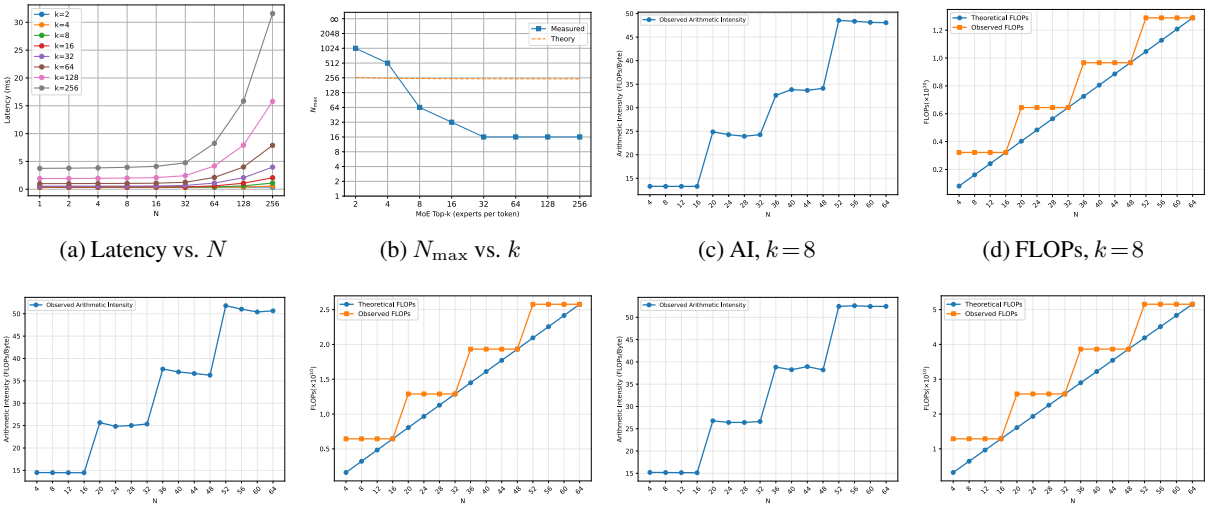
Taken together, the MoE FFN results support the mechanism identified in the main text. Across routing patterns, fused MoE backends, and evaluated single-GPU platforms, the MoE NFP boundary is governed primarily by implementation-induced kernel-granularity slack. Load-balanced routing exposes an aggregate upper-bound boundary that grows with sparsity, whereas load-skewed routing exposes a backend-granularity-fixed lower-bound boundary.



(a) Latency vs. N (b) $N_{\max}$ vs. k (c) AI, $k=8$ (d) FLOPs, $k=8$

(e) AI, $k=16$ (f) FLOPs, $k=16$ (g) AI, $k=32$ (h) FLOPs, $k=32$

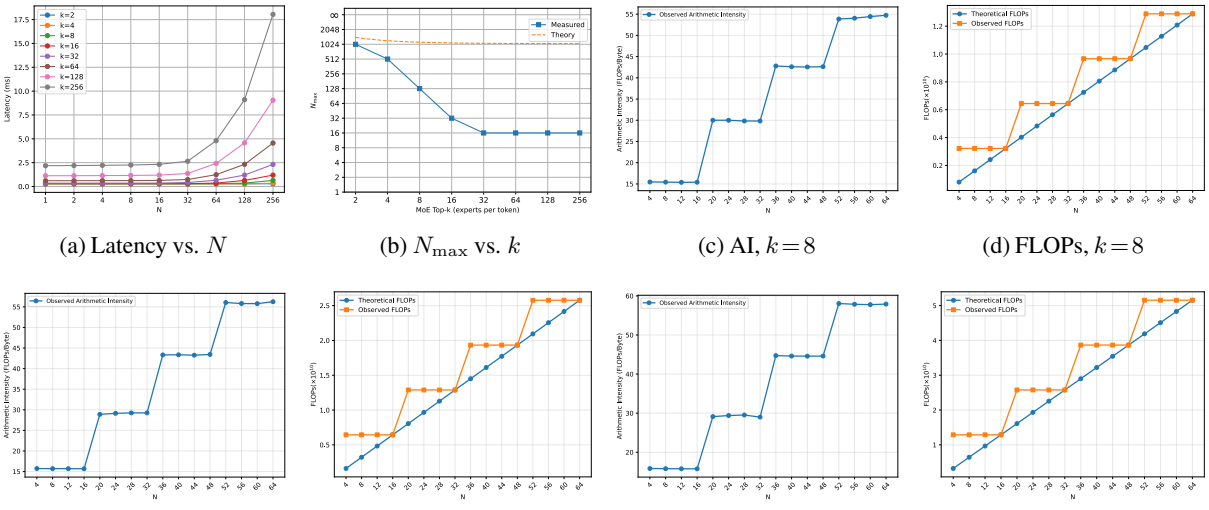
Figure 14: MoE FFN evaluation for load-skewed routing, the lower-bound case, with **vLLM** on **NVIDIA H20**.



(a) Latency vs. N (b) $N_{\max}$ vs. k (c) AI, $k=8$ (d) FLOPs, $k=8$

(e) AI, $k=16$ (f) FLOPs, $k=16$ (g) AI, $k=32$ (h) FLOPs, $k=32$

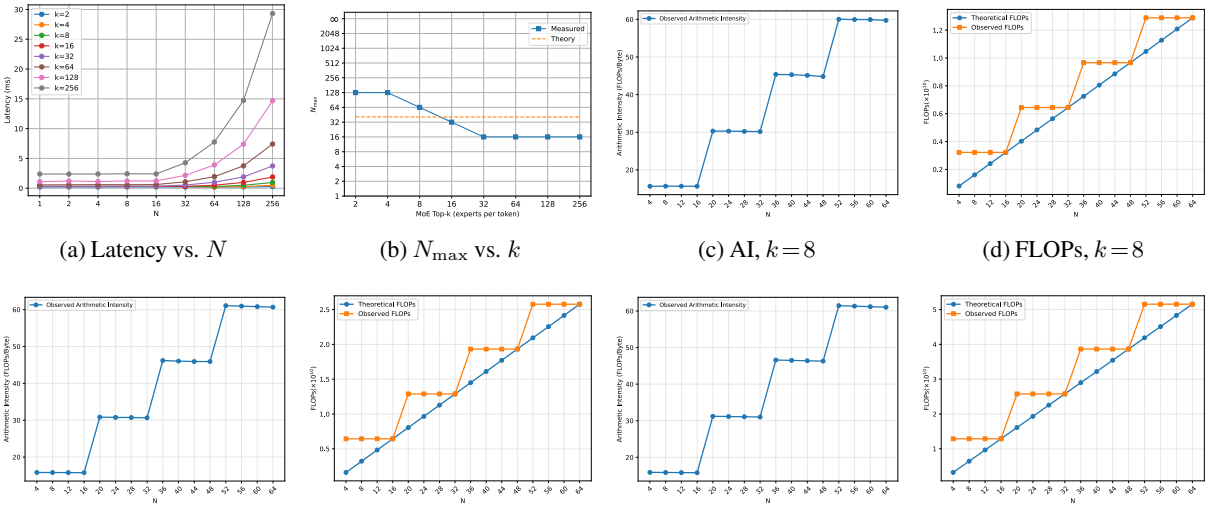
Figure 15: MoE FFN evaluation for load-skewed routing, the lower-bound case, with **vLLM** on **NVIDIA A800**.



(a) Latency vs. N (b) $N_{\max}$ vs. k (c) AI, $k=8$ (d) FLOPs, $k=8$

(e) AI, $k=16$ (f) FLOPs, $k=16$ (g) AI, $k=32$ (h) FLOPs, $k=32$

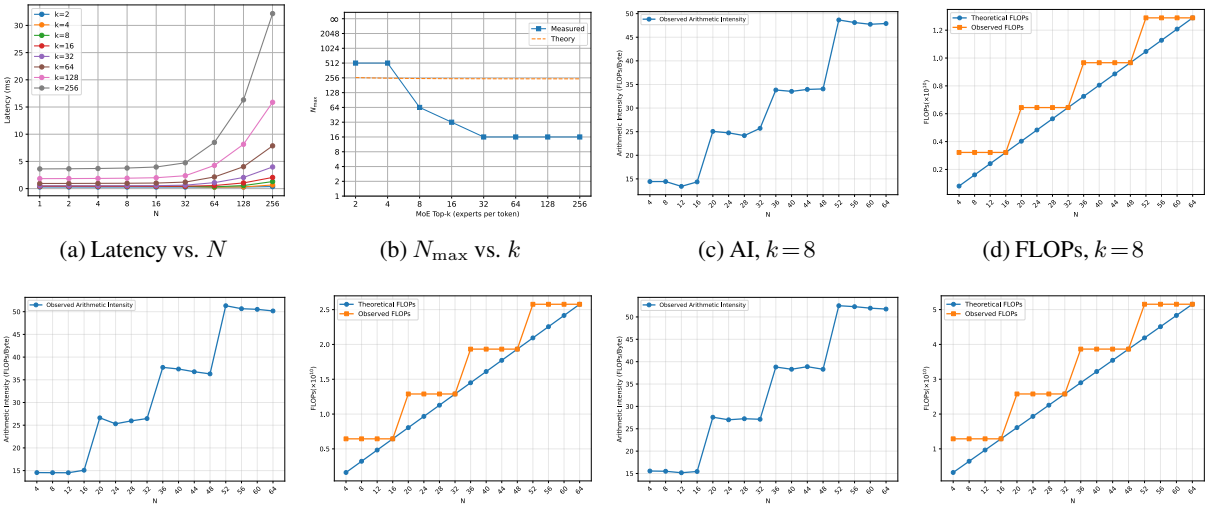
Figure 16: MoE FFN evaluation for load-skewed routing, the lower-bound case, with **vLLM** on **NVIDIA H800**.



(a) Latency vs. N (b) $N_{\max}$ vs. k (c) AI, $k=8$ (d) FLOPs, $k=8$

(e) AI, $k=16$ (f) FLOPs, $k=16$ (g) AI, $k=32$ (h) FLOPs, $k=32$

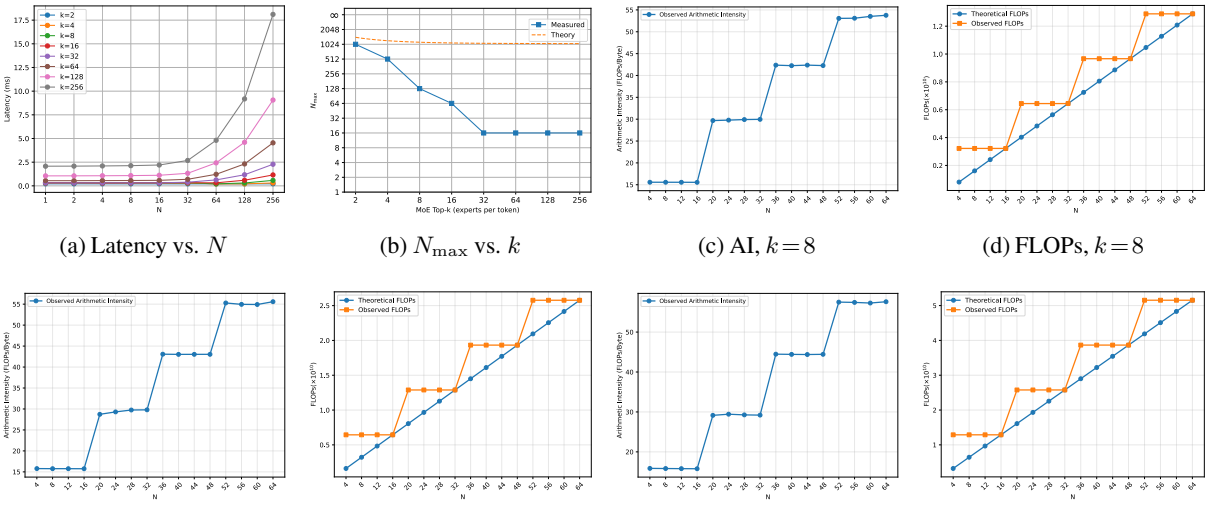
Figure 17: MoE FFN evaluation for load-skewed routing, the lower-bound case, with **SGLang** on **NVIDIA H20**.



(a) Latency vs. N (b) $N_{\max}$ vs. k (c) AI, $k=8$ (d) FLOPs, $k=8$

(e) AI, $k=16$ (f) FLOPs, $k=16$ (g) AI, $k=32$ (h) FLOPs, $k=32$

Figure 18: MoE FFN evaluation for load-skewed routing, the lower-bound case, with **SGLang** on **NVIDIA A800**.



(a) Latency vs. N (b) $N_{\max}$ vs. k (c) AI, $k=8$ (d) FLOPs, $k=8$

(e) AI, $k=16$ (f) FLOPs, $k=16$ (g) AI, $k=32$ (h) FLOPs, $k=32$

Figure 19: MoE FFN evaluation for load-skewed routing, the lower-bound case, with **SGLang** on **NVIDIA H800**.

D.3 Attention Layers Results

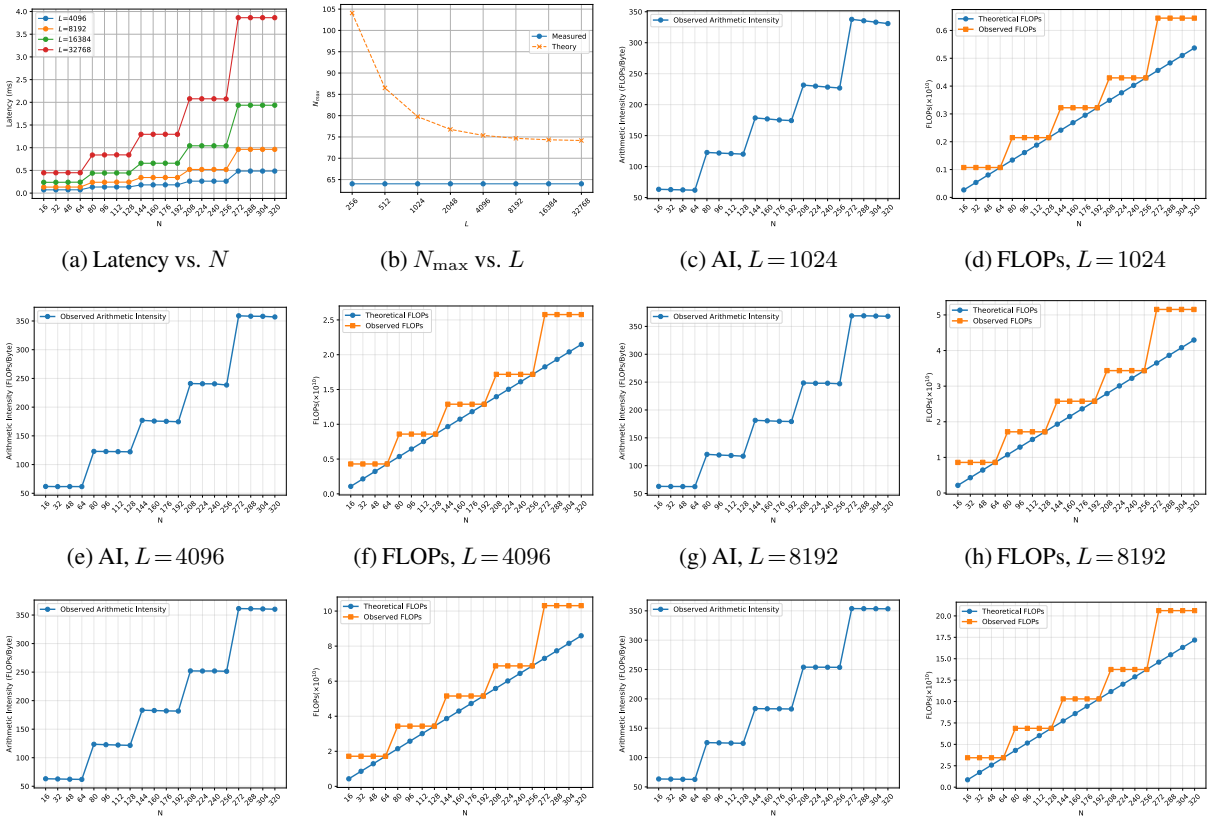
This section reports the complete Attention module-level results. These experiments complement the main-text analysis by validating the backend-specific query-granularity mechanism across attention backends and single-GPU platforms. We evaluate FlashAttention and FlashInfer on NVIDIA H20, A800, and H800. For each backend and GPU platform, we sweep the number of decode positions N under multiple cached sequence lengths L and extract $N_{\max}$ using the same 20% latency tolerance. In each figure, the latency subfigure shows how latency changes as N increases, the $N_{\max}$ subfigure compares the extracted boundary with the idle-compute prediction across sequence lengths, and the remaining subfigures report the arithmetic-intensity and runtime-FLOPs profiles for representative values of L.

D.3.1 FlashAttention Backend

Figures 20, 21, and 22 report the Attention results using the FlashAttention backend. Across all evaluated GPU platforms, the latency curves exhibit clear staircase behavior: latency remains nearly flat over several consecutive values of N and increases only when N crosses a backend granularity boundary. This confirms the existence of module-level NFP in Attention.

The extracted boundaries do not follow the idle-compute prediction. As shown in Figures 20b, 21b, and 22b, the measured $N_{\max}$ remains largely stable across cached sequence lengths L, whereas the KV-cache idle-compute baseline predicts an L-dependent boundary. This mismatch indicates that the Attention NFP boundary is not primarily governed by the compute-to-memory balance induced by KV-cache access.

The profiling results explain this mismatch. The achieved arithmetic intensity and runtime FLOPs change in discrete steps rather than smoothly with N. This indicates that FlashAttention executes padded or tiled query blocks, so larger N can remain within an existing backend granularity until the next boundary is crossed. Thus, the FlashAttention boundary is determined mainly by backend-specific query padding and tiling granularity.

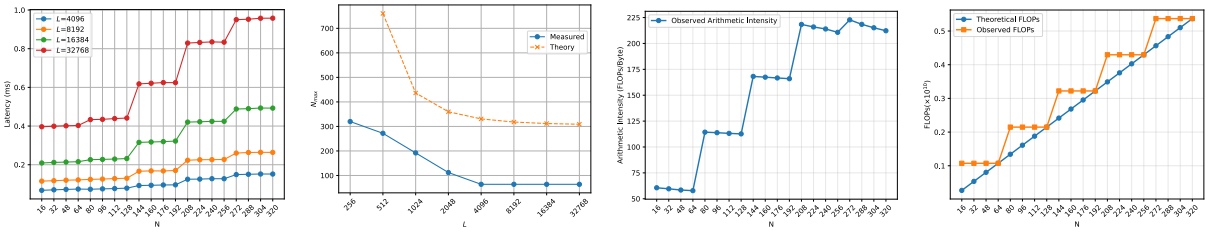


(a) Latency vs. N (b) $N_{\max}$ vs. L (c) AI, $L=1024$ (d) FLOPs, $L=1024$

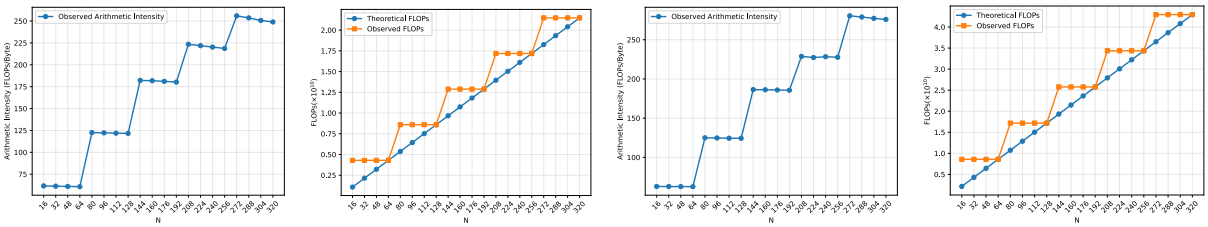
(e) AI, $L=4096$ (f) FLOPs, $L=4096$ (g) AI, $L=8192$ (h) FLOPs, $L=8192$

(i) AI, $L=16384$ (j) FLOPs, $L=16384$ (k) AI, $L=32768$ (l) FLOPs, $L=32768$

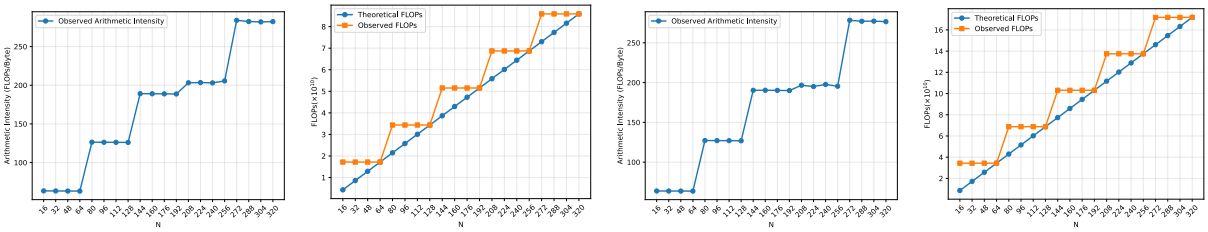
Figure 20: Attention evaluation results with **FlashAttention** on **NVIDIA H20**.



(a) Latency vs. N (b) $N_{\max}$ vs. L (c) AI, $L=1024$ (d) FLOPs, $L=1024$

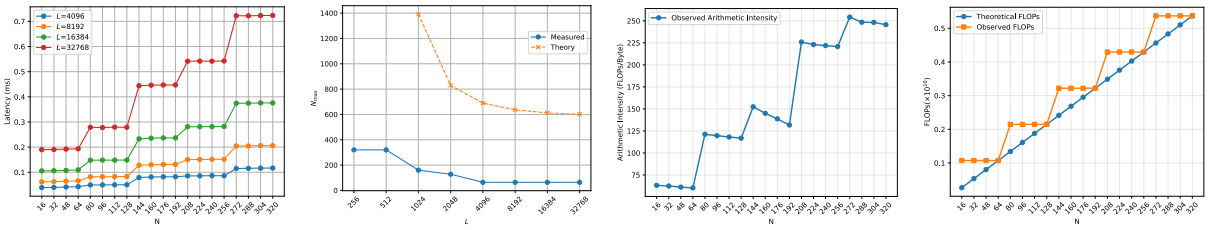


(e) AI, $L=4096$ (f) FLOPs, $L=4096$ (g) AI, $L=8192$ (h) FLOPs, $L=8192$

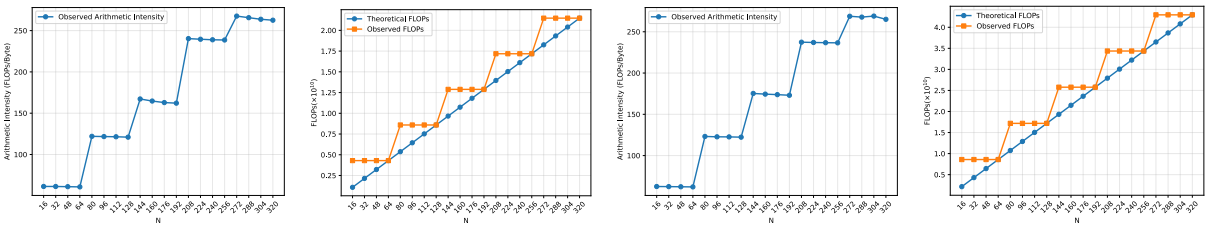


(i) AI, $L=16384$ (j) FLOPs, $L=16384$ (k) AI, $L=32768$ (l) FLOPs, $L=32768$

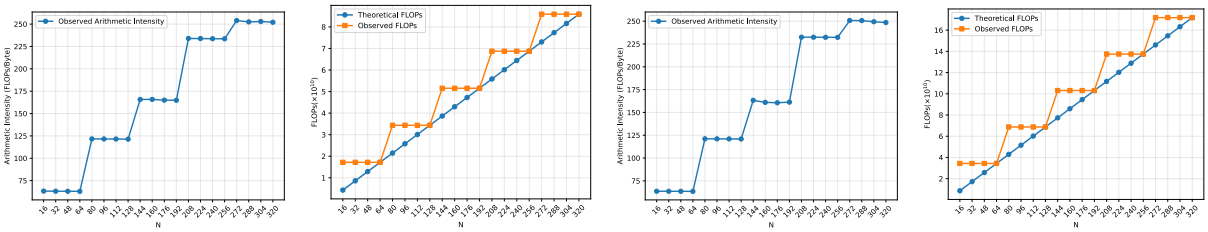
Figure 21: Attention evaluation results with **FlashAttention** on **NVIDIA A800**.



(a) Latency vs. N (b) $N_{\max}$ vs. L (c) AI, $L=1024$ (d) FLOPs, $L=1024$



(e) AI, $L=4096$ (f) FLOPs, $L=4096$ (g) AI, $L=8192$ (h) FLOPs, $L=8192$



(i) AI, $L=16384$ (j) FLOPs, $L=16384$ (k) AI, $L=32768$ (l) FLOPs, $L=32768$

Figure 22: Attention evaluation results with **FlashAttention** on **NVIDIA H800**.

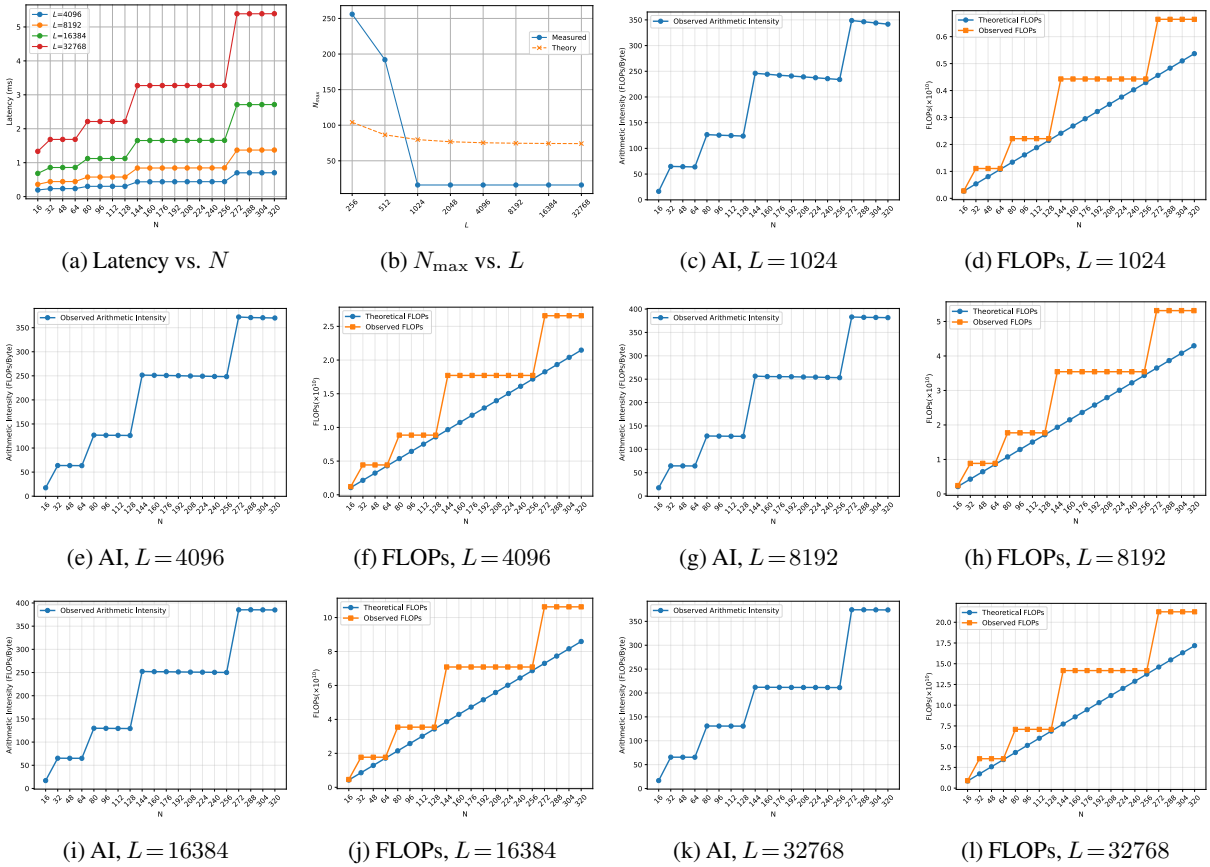
D.3.2 FlashInfer Backend

Figures 23, 24, and 25 report the corresponding Attention results using the FlashInfer backend. The same qualitative behavior appears under FlashInfer. The latency curves show near-free plateaus followed by discrete jumps, indicating that larger N remains near-free within a backend granularity block and becomes expensive only after crossing the next boundary.

The extracted boundaries again remain largely independent of cached sequence length. As shown in Figures 23b, 24b, and 25b, the measured $N_{\max}$ does not track the L-dependent idle-compute prediction. Instead, it is primarily determined by the query granularity of the FlashInfer backend. The arithmetic-intensity and FLOPs profiles further show discrete changes as N increases, confirming that the observed near-free region is produced by padded backend execution rather than a smooth resource-balance transition.

The agreement between FlashAttention and FlashInfer confirms that the Attention NFP mechanism is not specific to one backend implementation. At the same time, the actual boundary value is backend-specific, since different attention kernels use different query tiling and padding rules. Therefore, Attention contributes an implementation-dependent granularity term M_{attn} to the model-level NFP principle.

Overall, the Attention results confirm the mechanism identified in the main text. Across attention backends and evaluated single-GPU platforms, the Attention NFP boundary is governed primarily by backend-specific query padding and tiling rather than by the KV-cache idle-compute baseline. This explains why the measured boundary is largely independent of cached sequence length L and supports the use of M_{attn} in the model-level NFP principle.

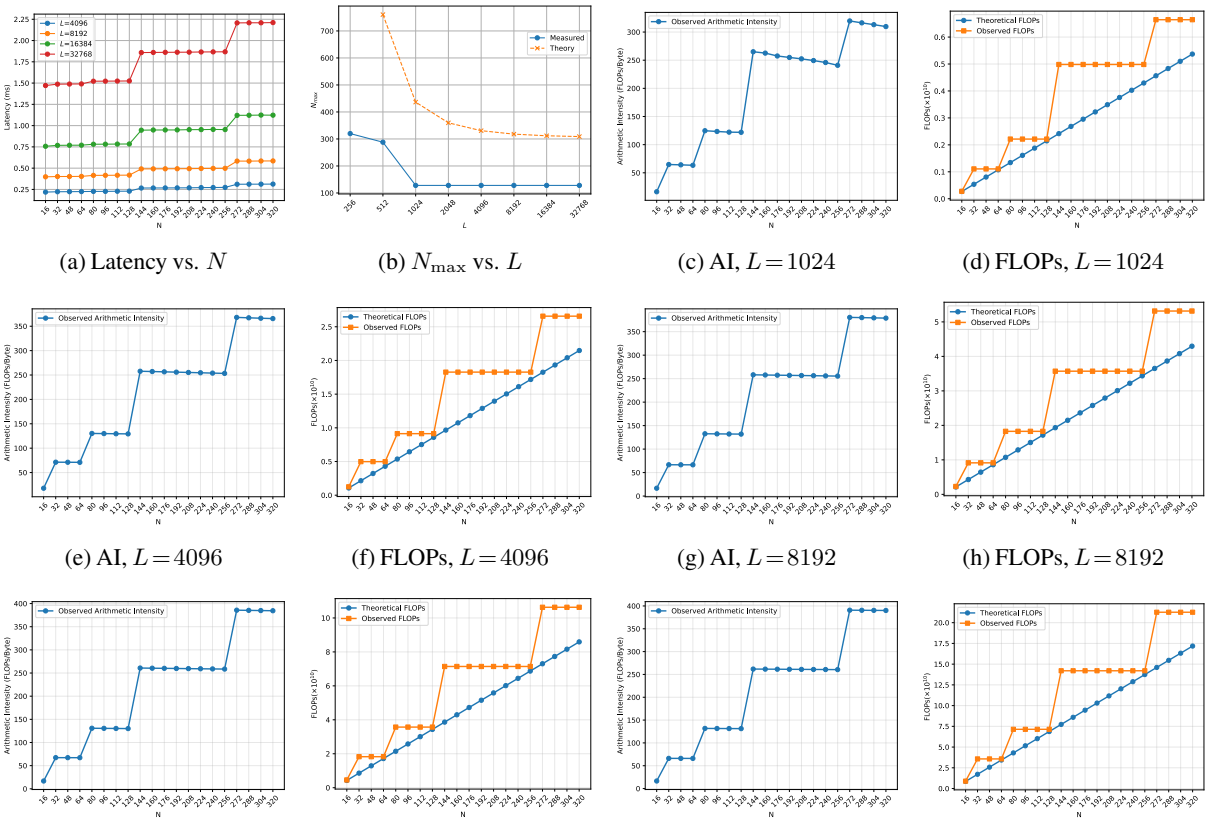


(a) Latency vs. N (b) $N_{\max}$ vs. L (c) AI, $L=1024$ (d) FLOPs, $L=1024$

(e) AI, $L=4096$ (f) FLOPs, $L=4096$ (g) AI, $L=8192$ (h) FLOPs, $L=8192$

(i) AI, $L=16384$ (j) FLOPs, $L=16384$ (k) AI, $L=32768$ (l) FLOPs, $L=32768$

Figure 23: Attention evaluation results with **FlashInfer** on **NVIDIA H20**.

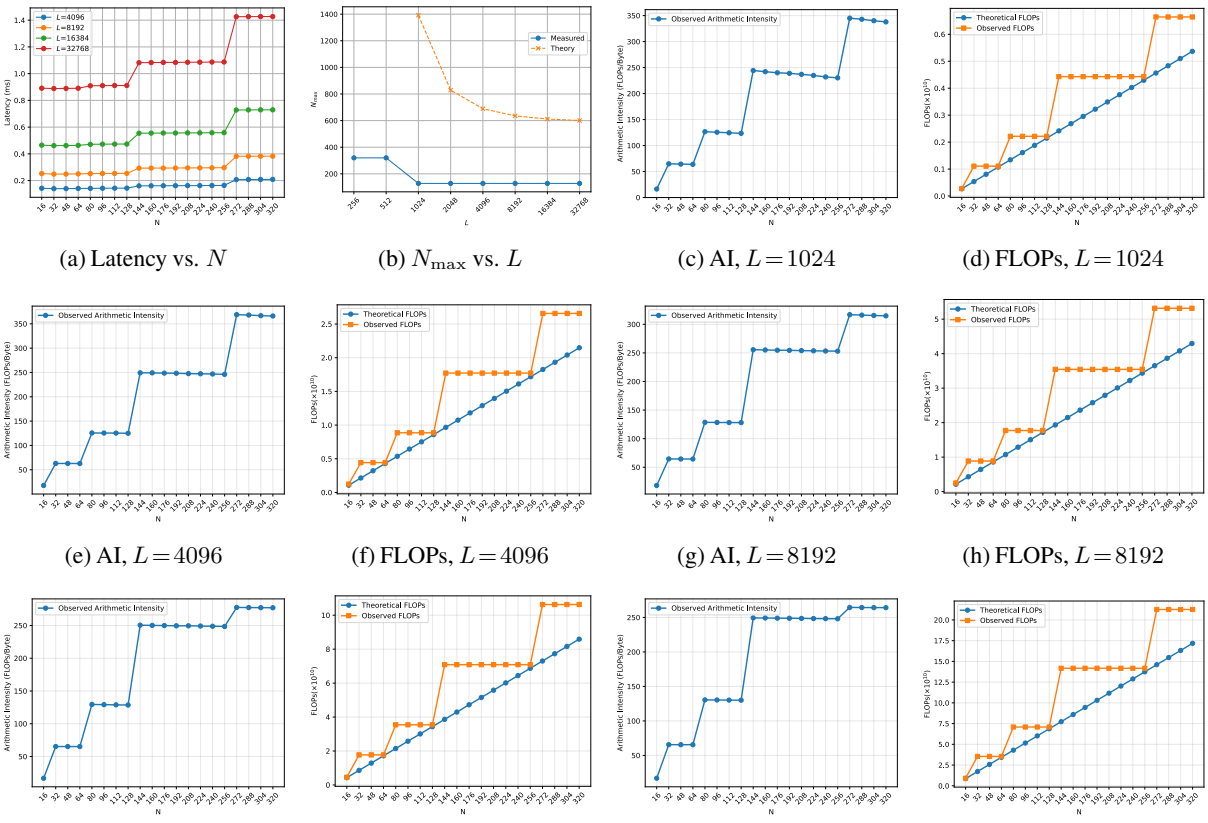


(a) Latency vs. N (b) $N_{\max}$ vs. L (c) AI, $L=1024$ (d) FLOPs, $L=1024$

(e) AI, $L=4096$ (f) FLOPs, $L=4096$ (g) AI, $L=8192$ (h) FLOPs, $L=8192$

(i) AI, $L=16384$ (j) FLOPs, $L=16384$ (k) AI, $L=32768$ (l) FLOPs, $L=32768$

Figure 24: Attention evaluation results with **FlashInfer** on **NVIDIA A800**.



(a) Latency vs. N (b) $N_{\max}$ vs. L (c) AI, $L=1024$ (d) FLOPs, $L=1024$

(e) AI, $L=4096$ (f) FLOPs, $L=4096$ (g) AI, $L=8192$ (h) FLOPs, $L=8192$

(i) AI, $L=16384$ (j) FLOPs, $L=16384$ (k) AI, $L=32768$ (l) FLOPs, $L=32768$

Figure 25: Attention evaluation results with **FlashInfer** on **NVIDIA H800**.

E Implementation Evidence for MoE Expert-Token Padding

The MoE FFN results in Appendix D.2 show staircase latency, arithmetic intensity, and runtime FLOPs, indicating that fused MoE execution performs padded computation beyond the logical routed-token workload. This appendix provides source-level evidence for this mechanism. We inspect the fused MoE implementations in vLLM and SGLang and show that both backends align expert-token workloads to implementation-defined block sizes before expert GEMM execution. This alignment creates kernel-granularity slack: larger N can remain within existing padded expert-token blocks until the next block boundary is crossed.

E.1 vLLM Fused-MoE Expert-Token Alignment

Tables 4 and 5 show the relevant vLLM fused MoE execution paths. In both the standard and chunked paths, vLLM first selects a kernel configuration from the MoE problem size and obtains a matrix-multiplication row-block parameter `BLOCK_SIZE_M`. Before launching the expert GEMMs, the backend passes this block size to `moe_align_block_size`, which constructs the expert-token layout used by the fused MoE kernel.

Table 6 shows the implementation of this alignment helper. The function explicitly aligns the routed token distribution across experts to the matrix-multiplication block size, allocates padded token and expert-block buffers, invokes the backend alignment operator, and returns `num_tokens_post_pad`. Thus, the expert GEMMs operate on padded expert-token blocks rather than exactly the logical routed-token counts.

This implementation explains the staircase behavior observed in the MoE profiling results. When increasing N increases the logical routed-token count but remain within the same `BLOCK_SIZE_M`-aligned expert-token block, the executed GEMM shape does not change. Once the logical workload crosses a block boundary, the padded shape increases discretely, producing the observed jumps in latency and runtime FLOPs.

```
for chunk in range((num_tokens // CHUNK_SIZE) + 1):
    begin_chunk_idx, end_chunk_idx = (
        chunk * CHUNK_SIZE,
        min((chunk + 1) * CHUNK_SIZE, num_tokens),
    )

    curr_hidden_states = hidden_states[begin_chunk_idx:end_chunk_idx]
    tokens_in_chunk, _ = curr_hidden_states.size()

    if tokens_in_chunk == 0:
        break

    if tokens_in_chunk < CHUNK_SIZE and chunk > 0:
        config = get_config_func(tokens_in_chunk)

    curr_topk_ids = topk_ids[begin_chunk_idx:end_chunk_idx]
    curr_topk_weights = topk_weights[begin_chunk_idx:end_chunk_idx]

    # ...

    sorted_token_ids, expert_ids, num_tokens_post_padded = (
        moe_align_block_size(curr_topk_ids, config['BLOCK_SIZE_M'],
                             global_num_experts, expert_map))
```

Table 4: Source-code evidence for expert-token padding in the **vLLM chunked fused MoE** path. The backend slices tokens and routing metadata into chunks, optionally reselects the kernel configuration for a tail chunk, and applies `moe_align_block_size` with `BLOCK_SIZE_M` to each chunk before expert GEMM execution.

```
E, num_tokens, N, K, top_k_num = mk._moe_problem_size(
    hidden_states, w1, w2, topk_ids)

if global_num_experts == -1:
    global_num_experts = E

config = try_get_optimal_moe_config(
    w1.size(),
    w2.size(),
    top_k_num,
    self.quant_config.config_name(hidden_states.dtype),
    num_tokens,
    block_shape=self.block_shape,
)

# ...

sorted_token_ids, expert_ids, num_tokens_post_padded = (
    moe_align_block_size(topk_ids, config['BLOCK_SIZE_M'],
                         global_num_experts, expert_map))
```

Table 5: Source-code evidence for expert-token padding in the standard **vLLM fused MoE** path. The backend selects a kernel configuration from the MoE problem size and passes `BLOCK_SIZE_M` to `moe_align_block_size`, which produces padded expert-token assignments before expert GEMM execution.

```
def moe_align_block_size(
    topk_ids: torch.Tensor,
    block_size: int,
    num_experts: int,
    expert_map: Optional[torch.Tensor] = None,
    pad_sorted_ids: bool = False
) -> tuple[torch.Tensor, torch.Tensor, torch.Tensor]:
    """
    Aligns the token distribution across experts to be compatible with block
    size for matrix multiplication.
    """
    max_num_tokens_padded = topk_ids.numel() + num_experts * (block_size - 1)
    if pad_sorted_ids:
        max_num_tokens_padded = round_up(max_num_tokens_padded, block_size)
    sorted_ids = torch.empty((max_num_tokens_padded, ),
                             dtype=torch.int32,
                             device=topk_ids.device)
    max_num_m_blocks = triton.cdiv(max_num_tokens_padded, block_size)
    expert_ids = torch.empty((max_num_m_blocks, ),
                             dtype=torch.int32,
                             device=topk_ids.device)
    num_tokens_post_pad = torch.empty((1),
                                      dtype=torch.int32,
                                      device=topk_ids.device)
    ops.moe_align_block_size(topk_ids, num_experts, block_size, sorted_ids,
                             expert_ids, num_tokens_post_pad)
    if expert_map is not None:
        expert_ids = expert_map[expert_ids]

    return sorted_ids, expert_ids, num_tokens_post_pad
```

Table 6: Implementation of expert-token block alignment in **vLLM fused MoE**. The helper function `moe_align_block_size` explicitly aligns routed tokens to the matrix-multiplication block size, allocates padded token buffers, invokes the backend alignment operator, and returns `num_tokens_post_pad`, providing direct evidence that fused MoE execution pads expert-token workloads to a block granularity.

E.2 SGLang Fused-MoE Expert-Token Alignment

SGLang follows the same expert-token alignment pattern. As shown in Table 7, the fused MoE backend derives the top-k routing tensors, selects a Triton MoE kernel configuration containing `BLOCK_SIZE_M`, slices the routing metadata for the current token chunk, and invokes `moe_align_block_size` before expert GEMM execution. The returned `num_tokens_post_padded` indicates that the routed expert-token workload has been expanded to a padded block-granular representation.

Table 8 provides the corresponding helper implementation. Similar to vLLM, the function aligns routed tokens to the matrix-multiplication block size, allocates padded token and expert buffers according to `max_num_tokens_padded`, invokes the backend alignment operator, and returns `num_tokens_post_pad`. Therefore, SGLang also executes fused MoE computation at expert-token block granularity.

The agreement between vLLM and SGLang shows that the MoE NFP mechanism is not an artifact of a single backend. Instead, both practical fused MoE implementations use the same structural pattern: routed tokens are grouped by expert and padded to block-size multiples before expert GEMM execution.

```
topk_weights, topk_ids, _ = topk_output

...

get_config_func = functools.partial(
    try_get_optimal_moe_config,
    w1.shape,
    (w2.shape[0], w2.shape[1], w2.shape[2] - padded_size),
    topk_ids.shape[1],
    config_dtype,
    block_shape=block_shape,
    per_channel_quant=per_channel_quant,
    return_down_config=True,
)

config, (down_config, max_block_m) = get_config_func(M)

...

curr_topk_ids = topk_ids[begin_chunk_idx:end_chunk_idx]

curr_topk_weights = topk_weights[begin_chunk_idx:end_chunk_idx]

...

sorted_token_ids, expert_ids, num_tokens_post_padded = moe_align_block_size(
    curr_topk_ids, config["BLOCK_SIZE_M"], E
)
```

Table 7: Source-code evidence for expert-token padding in **SGLang fused MoE**. The backend derives the top-k routing tensors, selects a Triton MoE kernel configuration that includes `BLOCK_SIZE_M`, slices the routing metadata for the current token chunk, and applies `moe_align_block_size` before expert GEMM execution. The returned `num_tokens_post_padded` shows that SGLang also pads routed expert-token workloads to a block granularity.

```python
def moe_align_block_size(
    topk_ids: torch.Tensor, block_size: int, num_experts: int
) -> Tuple[torch.Tensor, torch.Tensor, torch.Tensor]:
    """
    Aligns the token distribution across experts to be compatible with block
    size for matrix multiplication.
    """

    if topk_ids.numel() < num_experts + 1:
        max_num_tokens_padded = topk_ids.numel() * block_size
    else:
        max_num_tokens_padded = topk_ids.numel() + (num_experts + 1) * (block_size - 1)

    sorted_ids = torch.empty(
        (max_num_tokens_padded,), dtype=torch.int32, device=topk_ids.device
    )

    max_num_m_blocks = triton.cdiv(max_num_tokens_padded, block_size)

    expert_ids = torch.empty(
        (max_num_m_blocks,), dtype=torch.int32, device=topk_ids.device
    )

    num_tokens_post_pad = torch.empty((1), dtype=torch.int32, device=topk_ids.device)

    # In EP, expert_ids for filtered experts are -1. We have num_experts + 1 ids in total.
    cumsum_buffer = torch.empty(
        (num_experts + 2,), dtype=torch.int32, device=topk_ids.device
    )

    sgl_moe_align_block_size(
        topk_ids,
        num_experts + 1,
        block_size,
        sorted_ids,
        expert_ids,
        num_tokens_post_pad,
        cumsum_buffer,
        True,
    )

    return sorted_ids, expert_ids, num_tokens_post_pad
```

Table 8: Implementation of expert-token block alignment in **SGLang fused MoE**. The helper function `moe_align_block_size` explicitly aligns routed tokens to the matrix-multiplication block size, allocates padded token and expert buffers based on `max_num_tokens_padded`, invokes the backend alignment operator, and returns `num_tokens_post_pad`. This provides direct evidence that SGLang pads expert-token workloads to a block granularity before fused MoE execution.

E.3 From MoE Expert-Token Alignment to MoE NFP Boundaries

This section maps the backend kernel-selection rules to the concrete parameter values used in the NFP principle: the padding granularity M_{moe} and the branch-validity bound τ. The source-code evidence above shows that fused MoE kernels pad routed expert-token workloads to a row-block granularity before expert GEMM execution. We identify the backend-selected row-block size with the MoE padding granularity used in the NFP principle:

$$M_{\text{moe}} := \texttt{BLOCK_SIZE_M}. \tag{27}$$

For an expert e with $m_e(N)$ logical routed tokens, the fused MoE kernel executes the padded count

$$\widetilde{m}_e(N) = \left\lceil \frac{m_e(N)}{M_{\text{moe}}} \right\rceil M_{\text{moe}}. \tag{28}$$

Thus, increasing N remains near-free as long as it does not increase the padded expert-token counts $\widetilde{m}_e(N)$ or trigger a new kernel configuration.

To instantiate M_{moe} in practice, we inspect the backend rules that select `BLOCK_SIZE_M`. Tables 9 and 10 summarize these source-derived rules for vLLM and SGLang fused MoE fallback kernels. In these tables, M denotes the token dimension seen by the kernel configuration, while M_{moe} denotes the resulting expert-token padding granularity. For a given backend, version, numerical format or quantization mode, and baseline token shape, the corresponding table determines both the current M_{moe} and the range of M over which this value remains valid.

This distinction is important for deriving the load-balanced upper bound. Suppose the baseline workload selects M_{moe} and remains in the same kernel-configuration branch up to token dimension τ. Under load-balanced routing, padding slack is aggregated across activated experts. With E experts and top-k routing, the padding-capacity bound is approximately $M_{\text{moe}}E/k$. However, this bound is valid only while the same `BLOCK_SIZE_M` branch is used. Therefore, the backend-aware MoE upper bound is

$$N_{\max}^{\text{moe,bal}} \approx \min\left(\frac{M_{\text{moe}}E}{k}, \tau, M_{\text{attn}}\right). \tag{29}$$

For the evaluated vLLM v0.9.1 and SGLang BF16/FP16 fallback configurations, the small-token branch uses $M_{\text{moe}} = 16$ and remains valid while $M \leq E$. Hence $\tau = E$, giving the specialized form used in our experiments:

$$N_{\max}^{\text{moe,bal}} \approx \min\left(\frac{M_{\text{moe}}E}{k}, E, M_{\text{attn}}\right). \tag{30}$$

For load-skewed routing, the activated expert set is fixed from the single-position baseline. Larger N can only use the slack within the same selected expert-token blocks, so the MoE-side boundary is fixed by the local padding granularity:

$$N_{\max}^{\text{moe,skew}} \approx \min(M_{\text{moe}}, M_{\text{attn}}). \tag{31}$$

In this case, the kernel-configuration branch limit is typically non-binding in the small-N regime.

Overall, Tables 9 and 10 provide the practical rule for instantiating M_{moe}. The padding equation determines the available expert-token slack, while the branch-validity bound τ ensures that the principle is applied within the backend branch that selected the current M_{moe}.

vLLM version	Condition	BLOCK_SIZE_M
v0.9.0–v0.16.0	M <= E	16
	otherwise	64
v0.17.0–v0.20.2	M <= 32	16
	32 < M <= 96	32
	96 < M <= 512	64
	M > 512	128

Table 9: Source-derived BLOCK_SIZE_M selection rules for vLLM fused MoE fallback kernels. Here M denotes the token dimension used by the kernel configuration, and E denotes the number of experts. For a branch selected at the baseline workload, the corresponding BLOCK_SIZE_M instantiates the MoE padding granularity M_{moe}, while the branch condition determines the validity range τ over which this M_{moe} remains applicable. In the evaluated vLLM v0.9.1 configuration, the small-token branch uses $M_{\mathrm{moe}} = 16$ for $M \leq E$.

Quantization	Condition	BLOCK_SIZE_M
bf16/fp16	M <= E	16
	M > E	64
per-tensor FP8	M <= E	64
	M > E	128
block-wise FP8	any M	64

Table 10: Source-derived BLOCK_SIZE_M selection rules for SGLang Triton fused MoE fallback kernels. Here M denotes the token dimension used by the kernel configuration, and E denotes the number of experts. For a branch selected at the baseline workload, the corresponding BLOCK_SIZE_M instantiates the MoE padding granularity M_{moe}, while the branch condition determines the validity range τ over which this M_{moe} remains applicable. The selected branch depends on the quantization mode; in the evaluated SGLang BF16/FP16 configuration, the small-token branch uses $M_{\mathrm{moe}} = 16$ for $M \leq E$.

F Implementation Evidence for Attention Query-Tile Granularity

The Attention results in Appendix D.3 show staircase latency, arithmetic intensity, and runtime FLOPs, while the measured NFP boundary remains largely independent of the cached sequence length L. This appendix provides source-level evidence for this behavior. We show that practical attention backends execute query positions in backend-defined query tiles. This query-tile granularity allows larger N to remain within an existing tile until the next tile boundary is crossed.

F.1 FlashAttention Query-Tile Scheduling

Tables 11 and 12 show how FlashAttention implements query-tile execution. On the kernel side, Table 11 shows that the device function receives a query-tile index `m_block`, obtains `kBlockM` from `Kernel_traits`, and uses it to tile the query tensor with shape `(kBlockM, kHeadDim)`. This indicates that each FlashAttention query tile covers up to `kBlockM` query positions.

On the launch side, Table 12 shows that the backend computes the number of query tiles by applying ceiling division to `seqlen_q` with respect to `kBlockM`. The resulting tile count is used as a kernel grid dimension, and the launch path also checks whether `seqlen_q` is aligned to `kBlockM`. Thus, the amount of launched attention work changes at query-tile boundaries.

Together, these two code paths show that increasing the number of decode positions does not continuously change the executed FlashAttention workload. Larger N remains within the same query tile until the query length crosses the next `kBlockM`-aligned boundary, at which point an additional query tile is launched. Therefore, `kBlockM` is the FlashAttention instantiation of the backend query-tile granularity M_{attn}.

```
template<typename Kernel_traits, bool Is_dropout, bool Is_causal, bool Is_local,
     bool Has_alibi, bool Is_even_MN, bool Is_even_K, bool Is_softcap, bool
    Return_softmax, typename Params>
inline __device__ void compute_attn_1rowblock(const Params &params, const int
    bidb, const int bidh, const int m_block) {
   using Element = typename Kernel_traits::Element;
   using ElementAccum = typename Kernel_traits::ElementAccum;
   using index_t = typename Kernel_traits::index_t;

   // Shared memory.
   extern __shared__ char smem_[];

   // The thread index.
   const int tidx = threadIdx.x;

   constexpr int kBlockM = Kernel_traits::kBlockM;
   constexpr int kBlockN = Kernel_traits::kBlockN;
   constexpr int kHeadDim = Kernel_traits::kHeadDim;

   ...

   Tensor mQ = make_tensor(make_gmem_ptr(reinterpret_cast<Element*>(params.
    q_ptr)
                     + binfo.q_offset(params.q_batch_stride, params.q_row_stride,
     bidb)),
                       make_shape(binfo.actual_seqlen_q, params.h, params.d),
                       make_stride(params.q_row_stride, params.q_head_stride, _1
    {}));

   Tensor gQ = local_tile(mQ(_, bidh, _), Shape<Int<kBlockM>, Int<kHeadDim>>{},
                      make_coord(m_block, 0));  // (kBlockM, kHeadDim)
```

Table 11: FlashAttention kernel-side query-tile execution. The device function receives the query-tile index `m_block`, loads the query-tile height `kBlockM` from `Kernel_traits`, and tiles the query tensor into `(kBlockM, kHeadDim)` regions. This shows that each query tile covers up to `kBlockM` query positions.

```
const int num_m_block = (params.seqlen_q + Kernel_traits::kBlockM - 1) /
    Kernel_traits::kBlockM;

dim3 grid(num_m_block, params.b, params.h);

const bool is_even_MN = params.cu_seqlens_q == nullptr && params.cu_seqlens_k ==
    nullptr && params.seqlen_k % Kernel_traits::kBlockN == 0 && params.seqlen_q
    % Kernel_traits::kBlockM == 0;
```

Table 12: FlashAttention launch-side query-tile scheduling. The launch template computes the number of query tiles by applying ceiling division to `seqlen_q` with respect to `kBlockM`, uses the resulting `num_m_block` as a grid dimension, and checks whether the query length is aligned to `kBlockM`.

F.2 FlashInfer Query-Tile Scheduling

Tables 13 and 14 show how FlashInfer implements query-tile execution. On the kernel side, Table 13 shows that `CTA_TILE_Q` is part of the kernel traits and is used to size the shared-memory buffers for query, output, and intermediate attention states. This indicates that `CTA_TILE_Q` defines the number of query rows covered by one FlashInfer query tile.

On the scheduler side, Table 14 shows that FlashInfer selects a concrete `cta_tile_q`, computes the number of query tiles by applying ceiling division to `packed_qo_len`, and then enumerates `q_tile_idx` for execution. Thus, the scheduled attention work changes at query-tile boundaries rather than continuously with the logical query length.

Together, these two code paths show that larger N remains within the same FlashInfer query tile until the query workload crosses the next `CTA_TILE_Q`-aligned boundary. At that point, an additional query tile is scheduled. Therefore, `CTA_TILE_Q` is the FlashInfer instantiation of the backend query-tile granularity M_{attn}.

```
template <uint32_t NUM_WARPS_KV, uint32_t CTA_TILE_Q, uint32_t CTA_TILE_KV, uint32_t HEAD_DIM_QK,
        uint32_t HEAD_DIM_VO, typename DTypeQ, typename DTypeKV, typename DTypeO>
struct SharedStorageQKVO {
  union {
    struct {
      alignas(16) DTypeQ q_smem[CTA_TILE_Q * HEAD_DIM_QK];
      ...
    };
    struct {  // NOTE(Zihao): synchronize attention states across warps
      alignas(
        16) std::conditional_t<NUM_WARPS_KV == 1, float[1],
                               float[NUM_WARPS_KV * CTA_TILE_Q * HEAD_DIM_VO]> cta_sync_o_smem;
      alignas(16) std::conditional_t<NUM_WARPS_KV == 1, float2[1],
                                     float2[NUM_WARPS_KV * CTA_TILE_Q]> cta_sync_md_smem;
    };
    alignas(16) DTypeO smem_o[CTA_TILE_Q * HEAD_DIM_VO];
  };

  ...
};

template <MaskMode MASK_MODE_, uint32_t CTA_TILE_Q_, uint32_t NUM_MMA_Q_, uint32_t NUM_MMA_KV_,
        uint32_t NUM_MMA_D_QK_, uint32_t NUM_MMA_D_VO_, uint32_t NUM_WARPS_Q_,
        uint32_t NUM_WARPS_KV_, PosEncodingMode POS_ENCODING_MODE_, typename DTypeQ_,
        typename DTypeKV_, typename DTypeO_, typename DTypeQKAccum_, typename IdType_,
        typename AttentionVariant_>
struct KernelTraits {
  ...

  static constexpr uint32_t CTA_TILE_Q = CTA_TILE_Q_;
  ...

  using SharedStorage = SharedStorageQKVO<NUM_WARPS_KV, CTA_TILE_Q, CTA_TILE_KV, HEAD_DIM_QK,
                                          HEAD_DIM_VO, DTypeQ, DTypeKV, DTypeO>;

  ...
};
```

Table 13: FlashInfer kernel-side query-tile definition. The kernel traits expose `CTA_TILE_Q` as the query-tile size, and the shared-memory layout allocates query, output, and synchronization buffers with dimensions proportional to `CTA_TILE_Q`. This shows that each FlashInfer query tile covers up to `CTA_TILE_Q` query rows.

```
template <typename IdType>
inline auto PrefillSplitQOKVIndptr(IdType* qo_indptr_h, IdType* kv_indptr_h,
                                   uint32_t total_num_rows, uint32_t batch_size,
                                   uint32_t num_qo_heads, uint32_t num_kv_heads,
    uint32_t head_dim,
                                   uint32_t page_size, uint32_t
    max_batch_size_if_split,
                                   bool enable_cuda_graph, int32_t window_left,
                                   int32_t fixed_split_size, bool disable_split_kv) {
  ...
  const uint32_t gqa_group_size = num_qo_heads / num_kv_heads;

  std::vector<int64_t> packed_qo_len_arr(batch_size), kv_len_arr(batch_size);
  for (uint32_t i = 0; i < batch_size; ++i) {
    packed_qo_len_arr[i] = int64_t(qo_indptr_h[i + 1] - qo_indptr_h[i]) * int64_t
      (gqa_group_size);

    ...
  }

  ...

  uint32_t cta_tile_q;
  ...
  cta_tile_q = FA2DetermineCtaTileQ(avg_packed_qo_len, head_dim);

  ...

  for (uint32_t request_idx = 0; request_idx < batch_size; ++request_idx) {
    const int64_t packed_qo_len = packed_qo_len_arr[request_idx];
    const int64_t num_tiles_q = ceil_div(packed_qo_len, cta_tile_q);

    ...

    for (uint32_t q_tile_idx = 0; q_tile_idx < num_tiles_q; ++q_tile_idx) {
      ...
      qo_tile_indices.push_back(q_tile_idx);
      ...
    }

    ...
  }

  ...
}
```

Table 14: FlashInfer scheduler-side query-tile scheduling. The scheduler selects a concrete `cta_tile_q`, computes the number of query tiles by applying ceiling division to `packed_qo_len`, and enumerates query-tile indices `q_tile_idx` for execution.

F.3 From Query-Tile Granularity to Attention NFP Boundaries

The source-code evidence in Appendix F.1 and Appendix F.2 shows that practical attention backends execute query positions in backend-defined query tiles. We use the query-tile size selected by the backend as the concrete implementation of the Attention granularity in the NFP principle:

$$M_{\text{attn}} := \begin{cases} \texttt{kBlockM}, & \text{FlashAttention}, \\ \texttt{CTA_TILE_Q}, & \text{FlashInfer}. \end{cases} \tag{32}$$

Thus, M_{attn} is not a model-intrinsic constant. It is determined by the attention backend, kernel path, architecture, head dimension, and other kernel-configuration conditions.

For a decode forward with N query positions, the backend executes query tiles rather than individual query positions. The number of executed query tiles is

$$n_{\text{tile}}(N) = \left\lceil \frac{N}{M_{\text{attn}}} \right\rceil, \tag{33}$$

and the corresponding tiled query capacity is

$$\widetilde{N} = n_{\text{tile}}(N) M_{\text{attn}} = \left\lceil \frac{N}{M_{\text{attn}}} \right\rceil M_{\text{attn}}. \tag{34}$$

Therefore, runtime work changes at query-tile boundaries rather than continuously with the logical query count. Increasing N remains near-free as long as it does not increase $n_{\text{tile}}(N)$; once N crosses the next M_{attn}-aligned boundary, an additional query tile is executed.

To instantiate M_{attn} in practice, we inspect the backend rules that select the query-tile size. Tables 15, 16, and 17 summarize the source-derived rules for FlashAttention-2, FlashAttention-3, and FlashInfer, respectively. These rules specify which `kBlockM` or `CTA_TILE_Q` is selected under a given backend configuration, and therefore determine the M_{attn} used by the NFP principle.

These rules provide a practical procedure for determining the Attention-side NFP boundary. Given a backend and kernel configuration, we first identify the selected query-tile size and instantiate M_{attn}. We then determine the largest N that can be used before the query workload crosses the next tile boundary. Starting from the single-position baseline, the Attention-side boundary is therefore approximated by

$$N_{\max}^{\text{attn}} \approx M_{\text{attn}}. \tag{35}$$

This rule explains why the measured Attention boundary is largely independent of the cached sequence length L. Unlike the KV-cache idle-compute baseline, which predicts an L-dependent boundary, the realized boundary is controlled by backend query-tile granularity. Consequently, Attention contributes the implementation-dependent term M_{attn} to the model-level NFP principle, constraining both Dense and MoE models whenever the attention module is the first component to leave the near-free regime.

F.3.1 FlashAttention-2 Query-Tile Rules

Kernel path	Condition	kBlockM
Non-split-KV forward	headdim = 32	128
	headdim = 64	128
	headdim = 96, SM8x and causal	64
	headdim = 96, otherwise	128
	headdim = 128, no dropout, SM8x and causal	64
	headdim = 128, otherwise	128
	headdim = 192, no dropout	128
	headdim = 192, with dropout	64
	headdim = 256, high shared-memory satisfied	128
	headdim = 256, otherwise	64
Split-KV forward	Main split-KV attention kernel	64

Table 15: Source-derived kBlockM selection rules for FlashAttention-2 forward kernels. Here kBlockM denotes the query-tile size used by the selected kernel path and therefore instantiates the Attention granularity M_{attn} in the NFP principle. The selected value depends on the forward kernel path, head dimension, architecture, causal mode, dropout, and shared-memory constraints.

F.3.2 FlashAttention-3 Query-Tile Rules

Kernel path	Condition	kBlockM
SM90, BF16/FP16	headdim $\leq$ 64 and headdim_v = 512	64
	headdim $\leq$ 64 and headdim_v = 256	128
	headdim $\leq$ 64 and other headdim_v	192
	64 < headdim $\leq$ 96	192
	96 < headdim $\leq$ 128	128
	128 < headdim $\leq$ 192	128
	headdim > 192	128
SM90, FP8 / non-16-bit	headdim $\leq$ 64	192
	64 < headdim $\leq$ 96	192
	96 < headdim $\leq$ 128	128
	128 < headdim $\leq$ 192	128
	headdim > 192	128
SM8x, BF16/FP16	All listed headdim ranges	128
SM8x, FP8 / non-16-bit	All listed headdim ranges	128

Table 16: Source-derived kBlockM selection rules for FlashAttention-3 forward kernels. The selected kBlockM defines the query-tile size and instantiates M_{attn} for FlashAttention-3. The rule depends on the GPU architecture, numerical format, head dimension, value-head dimension, and kernel path.

F.3.3 FlashInfer Query-Tile Rules

FlashInfer determines the query-tile size through a scheduler-side selector. Let x denote the packed query/output workload used by the scheduler to select `CTA_TILE_Q`. Intuitively, x summarizes the effective query-side length seen by the scheduler after packing and grouped-query attention. The selected `CTA_TILE_Q` then defines the FlashInfer query-tile size and instantiates M_{attn} in the NFP principle. This connects the backend rule to NFP: as the number of decode positions N increases, the packed query/output workload seen by the scheduler also increases, which may change the selected query-tile size or the number of executed query tiles.

The array `qo_indptr` is the index-pointer array for the packed query/output tensor. The difference $\texttt{qo_indptr}_{i+1} - \texttt{qo_indptr}_i$ gives the number of query/output rows contributed by request i. In the uniform multi-position decode setting used in our analysis, each request contributes N query positions, so this difference is N.

FlashInfer uses different definitions of x depending on whether CUDA Graph execution is enabled.

- **CUDA Graph path.** The selector uses the maximum packed query/output length:

$$x = \texttt{max_qo_len} = (\texttt{total_num_rows} - b + 1) \cdot \texttt{gqa_group_size}. \tag{36}$$

  In our multi-position decode setting, `block_size` corresponds to the number of query positions per request, i.e., N. Therefore, when $\texttt{total_num_rows} = b \cdot N$, the selector input becomes

$$x = \big(b(N-1)+1\big) \cdot \texttt{gqa_group_size}. \tag{37}$$

- **Non-CUDA Graph path.** The selector uses the batch-average packed query/output length:

$$x = \texttt{avg_packed_qo_len} = \frac{1}{b}\sum_{i=0}^{b-1}\big(\texttt{qo_indptr}_{i+1} - \texttt{qo_indptr}_i\big) \cdot \texttt{gqa_group_size}. \tag{38}$$

  In our uniform multi-position decode setting, each request contributes N query positions, so $\texttt{qo_indptr}_{i+1} - \texttt{qo_indptr}_i = N$. After multiplication by `gqa_group_size`, the scheduler's packed query/output length scales linearly with N.

Condition	**`CTA_TILE_Q`**
$x > 64$ and $\texttt{head_dim} < 256$	128
Otherwise, Ampere or newer, and $x > 16$	64
Otherwise, Ampere or newer, and $x \leq 16$	16
Otherwise, Turing / pre-Ampere	64

Table 17: Source-derived `CTA_TILE_Q` selection rules for FlashInfer attention kernels. The scheduler computes a selector input x from the packed query/output workload and uses it to select `CTA_TILE_Q`, which defines the FlashInfer query-tile size and instantiates M_{attn} in the NFP principle.

G Model-Level Validation Implementation Details

G.1 Overall Validation Protocol

We validate the NFP principle at the full-model level by measuring the latency of a complete decode forward. For each model and serving configuration, a decode forward processes N decode positions over a pre-allocated KV cache. The measured latency includes all transformer layers and the corresponding serving backend kernels, but excludes prefill, tokenization, sampling, data loading, and CPU-side preprocessing. This scope matches the system-side quantity modeled by the NFP principle while avoiding unrelated request-level overheads.

For each configuration, we sweep the number of decode positions N and repeat the measurement 10 times. The measured boundary is extracted using the same criterion as in the module-level analysis:

$$N_{\max} = N_{\max}(0.2). \tag{39}$$

That is, $N_{\max}$ is the largest N whose decode-forward latency remains within a 20% increase over the baseline latency. Using the same tolerance keeps the full-model validation directly comparable with the module-level results and avoids introducing a separate boundary definition.

All full-model experiments are conducted on the same three single-GPU platforms used in the module-level analysis: NVIDIA H20, A800, and H800. These platforms cover different compute-to-memory balance points ρ, allowing us to test whether the full-model boundary follows the hardware-sensitive terms in the NFP principle while avoiding inter-GPU communication effects. Table 3 summarizes the hardware platforms. For each setting, the predicted boundary is computed directly from the NFP principle using the model configuration, hardware balance point ρ, and backend granularity parameters. No fitted parameter is introduced in the model-level validation.

In addition to DLLMs, we validate the NFP principle on autoregressive (AR) models. At the abstraction level modeled by the NFP principle, the verification forward in speculative decoding or multi-token prediction shares the same multi-position decode paradigm as a DLLM forward: both process multiple decode positions through the transformer stack in a single pass, and the NFP boundary is determined by the same module-level granularity and resource-balance constraints. The principle is therefore expected to apply regardless of how the candidate decode positions are generated. AR validation is conducted on NVIDIA H800. Cross-platform generalization has already been established in the DLLM setting across H20, A800, and H800; the AR experiments target paradigm-independence rather than repeating hardware-generalization evidence.

G.2 Dense Model Validation Implementation Details

For dense-model validation, we use WeDLM-8B, a representative dense DLLM. The model has 36 transformer layers, hidden size $d_{\text{model}} = 4096$, intermediate size $d_{\text{ff}} = 12288$, 32 attention heads, 8 key-value heads, and head dimension 128. The model is executed in BF16 precision.

We serve WeDLM-8B using a nano-vLLM-based serving stack with FlashAttention-2 as the attention backend. For each hardware platform and batch size, the dense-model prediction is instantiated as

$$N_{\max}^{\text{dense}} \approx \min\left(\frac{\rho s}{2b}, M_{\text{attn}}\right), \tag{40}$$

where $s = 2$ for BF16, b is the serving batch size, and M_{attn} is the query granularity of the FlashAttention-2 backend.

We evaluate sequence lengths $L \in \{128, 256, 384, 512\}$. These lengths are chosen to test the prediction across different cached-context sizes while keeping all experiments within a single-GPU setting. For dense models, the primary hardware-sensitive term is the Dense FFN term $\rho s/2b$; therefore, we focus on varying batch size and hardware balance rather than pushing to very long contexts. Longer contexts at larger batch sizes exceed the single-GPU memory budget for this model and would require multi-GPU execution, introducing communication overheads that are outside the scope of this validation.

For AR verification, we use Qwen3-8B with the same architecture and serving configuration as WeDLM-8B. The evaluation parameters (batch sizes, sequence lengths) are kept identical, and the experiment is conducted on NVIDIA H800.

G.3 MoE Model Validation Implementation Details

For MoE-model validation, we use LLaDA-2.1-mini, a representative MoE DLLM. The model has 20 transformer layers, hidden size $d_{\text{model}} = 2048$, intermediate size 5120, 16 attention heads, 4 key-value heads, and head dimension 128. The MoE layers contain $E = 256$ experts, select $k = 8$ experts per token, and use MoE intermediate size 512. The model is executed in BF16 precision.

We serve LLaDA-2.1-mini using SGLang with FlashInfer as the attention backend and the SGLang fused MoE backend for expert execution. This serving stack is used because the MoE NFP principle depends explicitly on both the fused-MoE expert-token padding granularity and the attention query granularity. For each setting, the predicted boundary is computed from the MoE NFP principle. For the load-balanced case, we use

$$N_{\max}^{\text{moe,bal}} \approx \min\left(\frac{M_{\text{moe}}E}{k}, E, M_{\text{attn}}\right). \tag{41}$$

For the load-skewed case, we use

$$N_{\max}^{\text{moe,skew}} \approx \min(M_{\text{moe}}, M_{\text{attn}}). \tag{42}$$

Here M_{moe} is the expert-token padding granularity of the fused MoE backend, and M_{attn} is the query granularity of the FlashInfer attention backend. For this evaluated backend configuration, the branch-validity bound satisfies $\tau = E$, so the backend-aware form in Section 4 reduces to the specialized formula used here.

As in the module-level MoE analysis, we use controlled routing patterns rather than input-dependent routing. The load-balanced pattern distributes tokens across experts to expose the aggregate expert-token padding slack, while the load-skewed pattern routes all tokens to the same selected experts to expose the lower-bound granularity-limited case. This keeps the expert-load distribution deterministic and makes the full-model boundary comparable with the module-level MoE principle.

We evaluate sequence lengths $L \in \{256, 4096, 16384, 32768\}$. These settings cover short, medium, long, and very long contexts for the MoE model. The purpose of this sweep is to test whether the predicted limiting module changes with context length: at shorter contexts the boundary is expected to be dominated by MoE FFN granularity, whereas at longer contexts the increased attention workload can make the attention backend granularity the limiting factor.

For AR verification, we use Ling-2.0-mini with the same architecture and serving configuration as LLaDA-2.1-mini. The evaluation parameters (top-k sweep, sequence lengths) are kept identical, and the experiment is conducted on NVIDIA H800.

H Model-Level Validation Results

This section reports the full-model validation results of the NFP principle. The goal is to test whether the module-level constraints identified in the main text compose into accurate full-model predictions, and whether the principle can identify the active limiting module across sequence lengths, routing patterns, and GPU platforms.

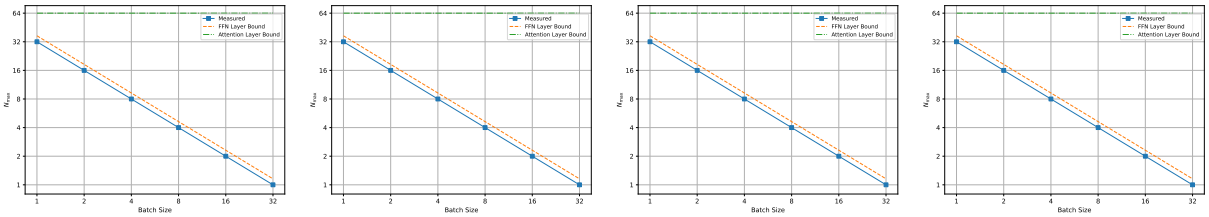
H.1 Dense Model Results

Figures 26–28 report the Dense model-level validation results on NVIDIA H20, A800, and H800. Across all evaluated GPU platforms, the measured NFP boundaries closely follow the principle-predicted boundaries. This agreement shows that the Dense FFN idle-compute constraint and the Attention query-granularity constraint compose into an accurate full-model predictor.

The same agreement holds across the evaluated sequence lengths. In this short-context validation range, changing the cached sequence length does not substantially change the measured boundary trend. The sequence-length robustness also supports the module-level Attention finding that the realized Attention boundary is determined mainly by backend query granularity.

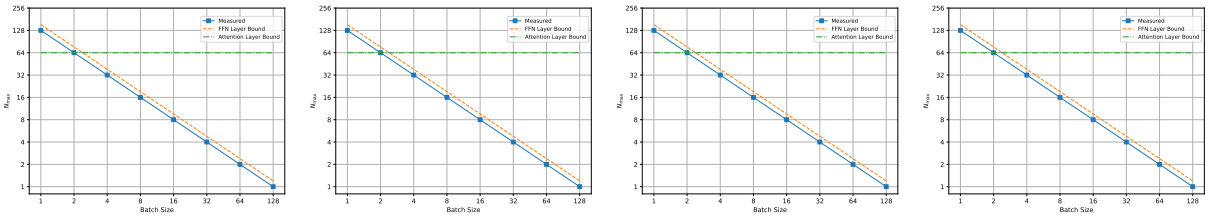
Across hardware platforms, the measured boundary shifts consistently with the hardware-sensitive Dense FFN term. At the same batch size, GPUs with larger compute-to-memory balance points expose larger near-free capacity, while increasing the batch size reduces the boundary. Overall, the Dense model results confirm that the proposed principle transfers from isolated modules to full-model inference and preserves the expected hardware and batch-size trends.

Figure 29 extends the validation to an AR model (Qwen3-8B) on H800. The measured boundary exhibits the same $1/b$ scaling governed by idle-compute slack and the same Attention-limited plateau, confirming that the Dense NFP mechanism applies equally to the AR verification forward.



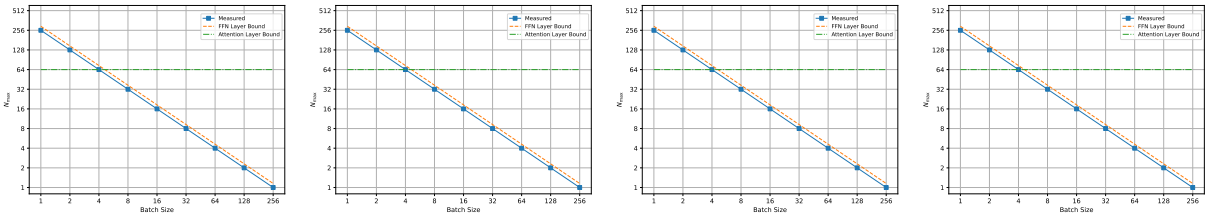
(a) $L=128$ (b) $L=256$ (c) $L=384$ (d) $L=512$

Figure 26: Dense model-level NFP principle validation results on **NVIDIA H20** (WeDLM-8B).



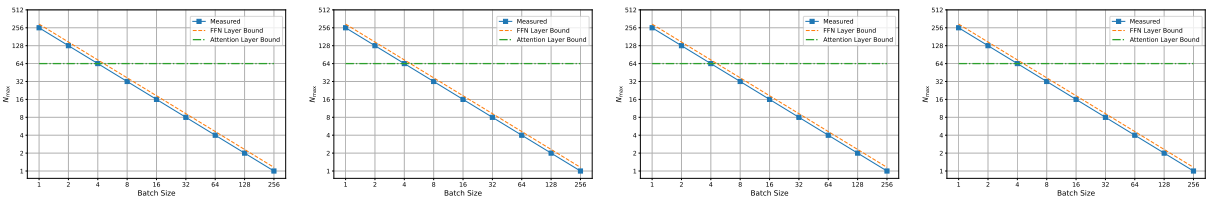
(a) $L=128$ (b) $L=256$ (c) $L=384$ (d) $L=512$

Figure 27: Dense model-level NFP principle validation results on **NVIDIA A800** (WeDLM-8B).



(a) $L=128$ (b) $L=256$ (c) $L=384$ (d) $L=512$

Figure 28: Dense model-level NFP principle validation results on **NVIDIA H800** (WeDLM-8B).



(a) $L=128$ (b) $L=256$ (c) $L=384$ (d) $L=512$

Figure 29: Dense model-level NFP principle validation results on **NVIDIA H800** (Qwen3-8B).

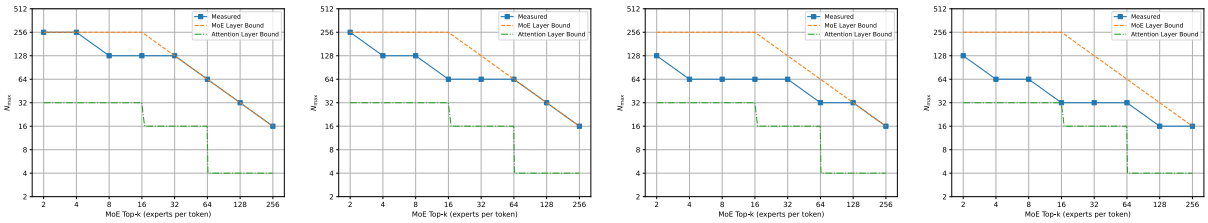
H.2 MoE Model Results

H.2.1 Load-balanced Routing as Upper Bound

Figures 30–32 report the MoE model-level results under load-balanced routing, which represents the upper-bound case. The measured boundaries follow the predicted trend across GPU platforms and sequence lengths: smaller k exposes more aggregate expert-token padding slack, while larger k reduces the maximum near-free N. This validates that the load-balanced MoE FFN granularity term remains predictive at the full-model level.

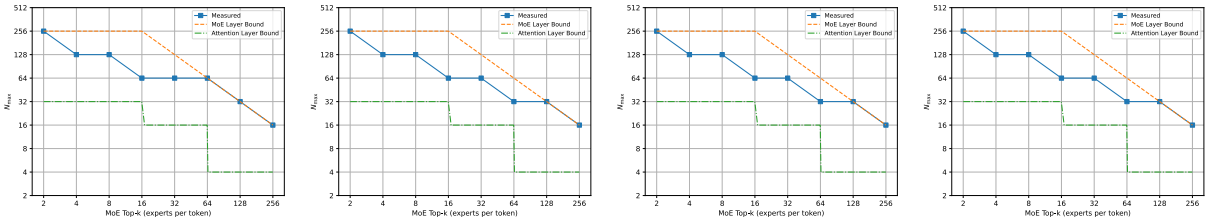
The results also show a clear limiting-module transition. At short sequence lengths, the measured boundary is primarily controlled by the MoE FFN granularity term, since Attention does not yet impose the active constraint. As the cached sequence length increases, the Attention workload becomes more significant, and the boundary approaches the Attention-side limit. Thus, the principle captures not only the boundary value, but also when the active bottleneck shifts from MoE FFN to Attention.

Figure 33 extends the load-balanced validation to an AR model (Ling-2.0-mini) on H800. The boundary follows the same staircase decay with increasing k and transitions to Attention-limited at long sequences, consistent with the DLLM results. This confirms that the aggregate expert-token padding slack mechanism governs the AR verification forward in the same manner.



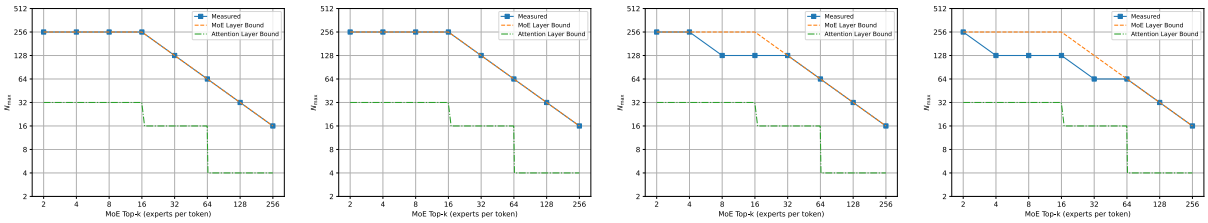
(a) $L = 256$ (b) $L = 4096$ (c) $L = 16384$ (d) $L = 32768$

Figure 30: MoE model-level NFP principle validation for load-balanced routing, the upper-bound case, on **NVIDIA H20** (LLaDA-2.1-mini).



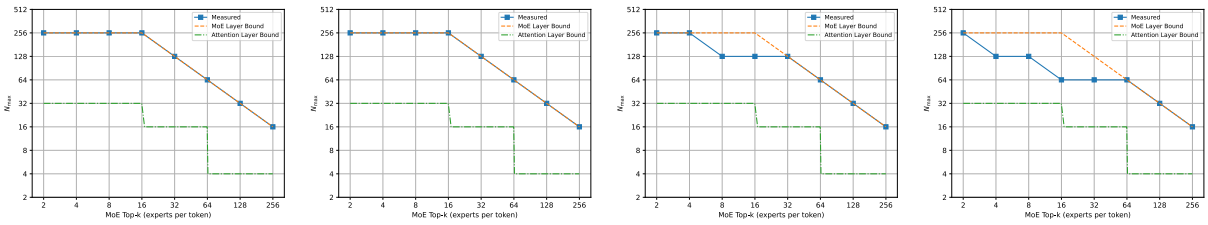
(a) $L = 256$ (b) $L = 4096$ (c) $L = 16384$ (d) $L = 32768$

Figure 31: MoE model-level NFP principle validation for load-balanced routing, the upper-bound case, on **NVIDIA A800** (LLaDA-2.1-mini).



(a) $L = 256$ (b) $L = 4096$ (c) $L = 16384$ (d) $L = 32768$

Figure 32: MoE model-level NFP principle validation for load-balanced routing, the upper-bound case, on **NVIDIA H800** (LLaDA-2.1-mini).



(a) $L = 256$ (b) $L = 4096$ (c) $L = 16384$ (d) $L = 32768$

Figure 33: MoE model-level NFP principle validation for load-balanced routing, the upper-bound case, on **NVIDIA H800** (Ling-2.0-mini).

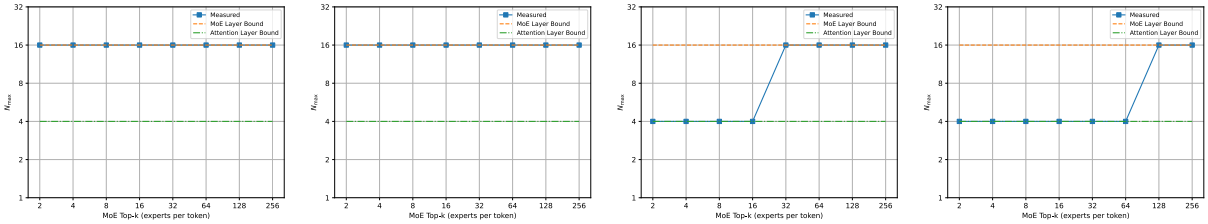
H.2.2 Load-skewed Routing as Lower Bound

Figures 34–36 report the corresponding results under load-skewed routing, which represents the lower-bound case. When Attention is not the active limiter, the measured boundary remains nearly fixed across k, matching the prediction that load-skewed routing exposes only a local expert-token padding granularity rather than aggregate slack across experts. The contrast with the load-balanced case confirms that expert-load distribution remains a first-order factor in full-model MoE NFP.

The hardware trends further support the principle. On higher-performance GPU platforms, the MoE-side boundary remains active over a wider range of sequence lengths, while lower-capacity platforms become Attention-limited earlier at long contexts. This does not imply that M_{attn} itself is hardware-derived; rather, it shows that the point at which the Attention constraint becomes the active full-model bottleneck depends on the interaction between backend granularity, attention workload, and platform performance.

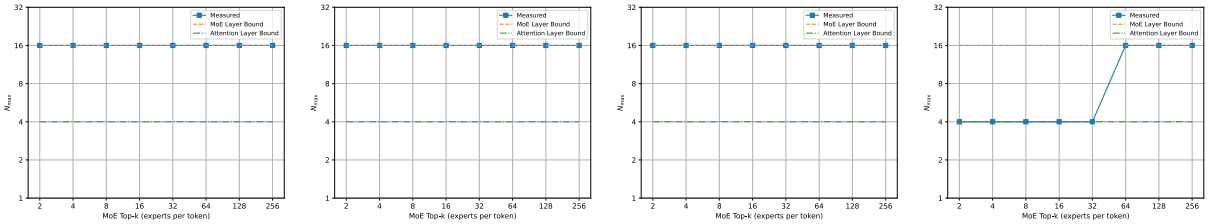
Figure 37 extends the load-skewed validation to an AR model (Ling-2.0-mini) on H800. The boundary remains flat across all k values, determined by local padding granularity, reproducing the DLLM observation. Together with the Dense and load-balanced results, these AR experiments confirm that the NFP principle is paradigm-independent: the system-level latency behavior is governed by the same granularity and resource-balance constraints regardless of whether decode positions originate from a diffusion schedule or a speculative draft.

Overall, the MoE model results confirm that the proposed NFP principle transfers to full-model routed inference. It predicts both upper- and lower-bound routing regimes, explains the transition from MoE-limited to Attention-limited behavior as sequence length increases, and identifies the active module constraint across hardware platforms.



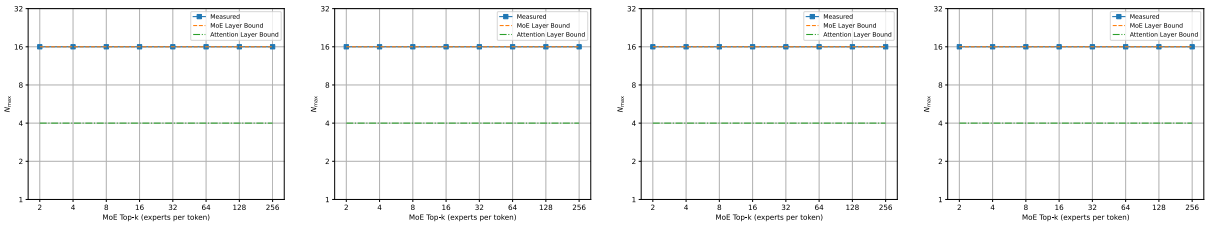
(a) $L = 256$ (b) $L = 4096$ (c) $L = 16384$ (d) $L = 32768$

Figure 34: MoE model-level NFP principle validation for load-skewed routing, the lower-bound case, on **NVIDIA H20** (LLaDA-2.1-mini).



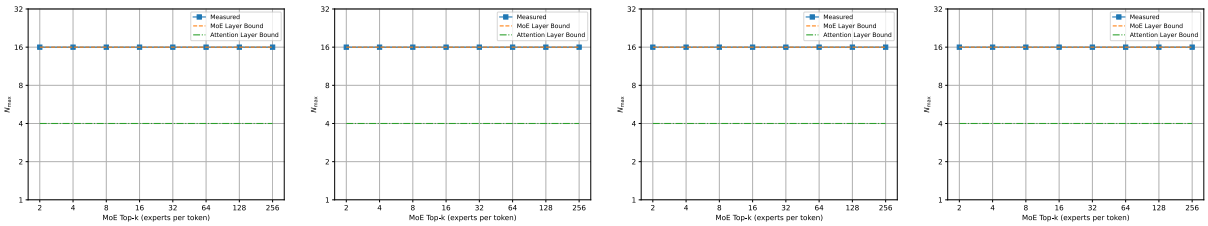
(a) $L = 256$ (b) $L = 4096$ (c) $L = 16384$ (d) $L = 32768$

Figure 35: MoE model-level NFP principle validation for load-skewed routing, the lower-bound case, on **NVIDIA A800** (LLaDA-2.1-mini).



(a) $L = 256$ (b) $L = 4096$ (c) $L = 16384$ (d) $L = 32768$

Figure 36: MoE model-level NFP principle validation for load-skewed routing, the lower-bound case, on **NVIDIA H800** (LLaDA-2.1-mini).



(a) $L = 256$ (b) $L = 4096$ (c) $L = 16384$ (d) $L = 32768$

Figure 37: MoE model-level NFP principle validation for load-skewed routing, the lower-bound case, on **NVIDIA H800** (Ling-2.0-mini).

I Sensitivity to Tolerance Threshold

The NFP boundary is extracted using a fixed 20% latency tolerance ($\epsilon = 0.2$). This section examines how the extracted boundary changes across tolerance values $\epsilon \in \{0.05, 0.10, 0.15, 0.20, 0.30\}$ on NVIDIA H20, covering all module classes and inference backends evaluated in this work.

For granularity-governed modules (MoE FFNs and Attention), $N_{\max}$ is nearly invariant across the evaluated tolerance range under both vLLM/SGLang and FlashAttention/FlashInfer backends (Tables 21–19). This robustness arises directly from the staircase latency pattern: within a kernel-granularity block, all N values produce nearly identical latency, while crossing a block boundary causes a discrete jump well above any reasonable tolerance. Any tolerance in $[0.05, 0.30]$ therefore yields the same extracted boundary for these modules at sufficiently long sequence lengths.

For Dense FFNs (Table 18), the boundary shifts by at most one discrete N step (e.g., from 16 to 32 at b=1) as ϵ varies, because the latency transition is smoother. The $1/b$ scaling trend is preserved regardless of threshold choice.

For the load-skewed MoE case (Tables 23–24), a tighter tolerance (ϵ=0.05) can yield a smaller boundary at some k values. This occurs because minor latency variation within a single padding block—well below the magnitude of cross-block jumps—may exceed a very tight threshold. However, the dominant staircase structure remains clearly identifiable regardless of ϵ: latency increases discretely at padding-block boundaries rather than smoothly with N. The mechanism conclusion is therefore independent of the threshold. For practical deployment guidance, $\epsilon \in [0.10, 0.30]$ produces consistent boundaries across all evaluated modules and backends.

We also note that at short cached sequence lengths, the absolute kernel latency is very small (sub-millisecond), making the percentage-based tolerance more susceptible to GPU timing noise. This is visible in the FlashInfer results at $L \leq 2048$ (Table 20), where the extracted boundary fluctuates at tight tolerances. At longer sequence lengths where latency is larger and the signal-to-noise ratio is higher, the boundary stabilizes and consistently reflects the backend query-tile granularity identified in Section 4.

Overall, the mechanism identification and scaling trends reported in this work are independent of the specific tolerance value.

b	ϵ=0.05	ϵ=0.10	ϵ=0.15	ϵ=0.20	ϵ=0.30
1	16	16	16	32	32
2	8	8	8	16	16
4	4	4	8	8	8
8	2	2	4	4	4
16	1	1	2	2	2
32	1	1	1	1	1

Table 18: Sensitivity of Dense FFN $N_{\max}$ to tolerance threshold ϵ (H20).

L	ϵ=0.05	ϵ=0.10	ϵ=0.15	ϵ=0.20	ϵ=0.30
256	64	64	64	64	256
512	16	48	64	64	64
1024	32	64	64	64	64
2048	64	64	64	64	64
4096	64	64	64	64	64
8192	64	64	64	64	64
16384	64	64	64	64	64
32768	64	64	64	64	64

Table 19: Sensitivity of Attention $N_{\max}$ to tolerance threshold ϵ (FlashAttention, H20).

L	ϵ=0.05	ϵ=0.10	ϵ=0.15	ϵ=0.20	ϵ=0.30
256†	240	240	256	256	256
512†	128	128	144	224	256
1024†	1	2	16	16	16
2048†	1	16	16	16	64
4096	16	16	16	16	64
8192	16	16	16	16	64
16384	16	16	16	16	64
32768	16	16	16	16	64

Table 20: Sensitivity of Attention $N_{\max}$ to tolerance threshold ϵ (FlashInfer, H20). †At short sequence lengths ($L \leq 2048$), kernel latency is very small (sub-ms), making the percentage-based tolerance susceptible to timing noise. Long-sequence results ($L \geq 4096$) reliably reflect the backend query-tile granularity.

k	ϵ=0.05	ϵ=0.10	ϵ=0.15	ϵ=0.20	ϵ=0.30
2	256	256	256	256	256
4	256	256	256	256	256
8	256	256	256	256	256
16	256	256	256	256	256
32	128	128	128	128	128
64	64	64	64	64	64
128	32	32	32	32	32
256	16	16	16	16	16

Table 21: Sensitivity of MoE FFN $N_{\max}$ to tolerance threshold ϵ (load-balanced, vLLM, H20). Baseline is the smallest N that activates all experts.

k	ϵ=0.05	ϵ=0.10	ϵ=0.15	ϵ=0.20	ϵ=0.30
2	256	256	256	256	256
4	256	256	256	256	256
8	256	256	256	256	256
16	256	256	256	256	256
32	128	128	128	128	128
64	64	64	64	64	64
128	32	32	32	32	32
256	16	16	16	16	16

Table 22: Sensitivity of MoE FFN $N_{\max}$ to tolerance threshold ϵ (load-balanced, SGLang, H20). Baseline is the smallest N that activates all experts.

k	ϵ=0.05	ϵ=0.10	ϵ=0.15	ϵ=0.20	ϵ=0.30
2	512	512	512	512	512
4	128	128	128	128	128
8	64	64	64	64	64
16	16	32	32	32	32
32	8	16	16	16	16
64	4	16	16	16	16
128	2	16	16	16	16
256	16	16	16	16	16

Table 23: Sensitivity of MoE FFN $N_{\max}$ to tolerance threshold ϵ (load-skewed, vLLM, H20).

k	ϵ=0.05	ϵ=0.10	ϵ=0.15	ϵ=0.20	ϵ=0.30
2	128	128	128	128	512
4	128	128	128	128	128
8	64	64	64	64	64
16	32	32	32	32	32
32	4	16	16	16	16
64	1	8	16	16	16
128	4	4	16	16	16
256	16	16	16	16	16

Table 24: Sensitivity of MoE FFN $N_{\max}$ to tolerance threshold ϵ (load-skewed, SGLang, H20).

J Discussion and Implications

J.1 Revisiting the Idle-Compute Baseline

This section provides a unified retrospective interpretation of the per-module mechanisms identified in Section 3, explaining why the idle-compute baseline succeeds for one module class but not the others. The key point is that the idle-compute baseline is not only a resource-balance argument; it also assumes an idealized execution model in which logical decode positions are mapped to physical kernel work approximately continuously.

Dense FFNs are close to this idealized setting. For a fixed set of weights, increasing the number of processed positions smoothly increases the GEMM workload, while the dominant weight traffic is largely amortized across positions. Thus, the physical work executed by the system closely follows the logical workload implied by the number of positions. In this regime, increasing N primarily consumes otherwise idle tensor-core capacity in the memory-bound single-position case, so the NFP boundary is well described by the resource-balance condition. This also explains why increasing batch size reduces Dense FFN NFP: batching and increasing N consume the same compute slack.

MoE FFNs and Attention violate this smooth-scaling assumption. In practical fused MoE kernels, routed tokens are grouped by expert and padded or aligned to backend-defined expert-token blocks. Similarly, practical attention backends execute query positions in backend-defined query tiles. As a result, physical work is a discretized, rounded version of the logical workload. Larger N is near-free when it falls inside already allocated expert-token blocks or query tiles, but becomes expensive when it crosses a granularity boundary.

Therefore, the failure of the idle-compute baseline for MoE FFNs and Attention does not mean that resource balance is irrelevant. Rather, implementation granularity changes the mapping from logical parallelism to physical work, and this granularity-induced slack dominates the observed NFP boundary. The relevant distinction is thus continuous versus granular realization of logical parallelism: Dense FFNs are mainly governed by resource slack, whereas MoE FFNs and Attention are mainly governed by padding, tiling, routing distribution, and backend-specific execution granularity.

J.2 Implications

The NFP principle turns near-free parallelism from an empirical latency observation into a system-side design signal. It does not predict the end-to-end speedup of a specific decoding algorithm. Instead, it estimates how much multi-position execution a given model-system stack can absorb at near-free latency, and which component limits this capacity.

J.2.1 Deployment-Time Parallelism Budget

For deployment, the NFP principle provides a latency-side budget for matching parallel decoding configurations to serving conditions. Given a model, hardware platform, batch size, and backend implementation, the principle predicts the range of decode positions that can be executed within the near-free regime. This is useful when the algorithmic parallelism level is fixed or only partially tunable. For example, an MTP model may expose a fixed prediction length, while a speculative decoding system may choose a verification length. The predicted NFP boundary indicates under which batch sizes, hardware platforms, and backend configurations these positions remain near-free, and when they begin to incur additional latency. Table 25 instantiates this budget for representative dense and MoE serving configurations, comparing the NFP principle prediction with the idle-compute baseline. The comparison shows that the standard resource-balance intuition over-predicts the realized boundary by 2.4–$23\times$ in over half of the evaluated settings, confirming that kernel-granularity constraints must be accounted for when selecting parallelism levels.

Model type	Representative setting	Idle-compute baseline	NFP Principle	Baseline discrepancy
Dense	H20, $b=1$	~ 37	~ 37	—
Dense	H20, $b=4$	~ 9	~ 9	—
Dense	A800, $b=1$	~ 153	~ 64	$2.4\times$ over (Attn. tile)
Dense	H800, $b=1$	~ 295	~ 64	$4.6\times$ over (Attn. tile)
Dense	H800, $b=8$	~ 37	~ 37	—
MoE, balanced	$E=256, k=8$	$\sim 1453^{\dagger}$	~ 64	$23\times$ over (granularity)
MoE, balanced	$E=256, k=32$	$\sim 362^{\dagger}$	~ 64	$5.7\times$ over (granularity)
MoE, balanced	$E=256, k=64$	$\sim 181^{\dagger}$	~ 64	$2.8\times$ over (granularity)
MoE, skewed	any k	$\sim 45^{\dagger}$	~ 16	$2.8\times$ over (MoE padding)

Table 25: Deployment lookup comparing the idle-compute baseline with the NFP principle under representative settings ($M_{\text{attn}}=64$). The idle-compute baseline predicts the NFP boundary from resource balance alone, ignoring implementation granularity. The baseline discrepancy column shows where and by how much this standard intuition over-predicts the realized boundary. In over half of the evaluated settings, the idle-compute baseline over-predicts by 2.4–$23\times$, demonstrating the practical importance of accounting for kernel-granularity constraints. †MoE idle-compute values use H20 ($\rho=37$) as a conservative lower bound; on higher-ρ hardware (A800, H800) the over-prediction is larger or unbounded. In the skewed case, k appears in both numerator and denominator of $N_{\text{idle}}^{\text{moe}}$, yielding a near-constant prediction (~ 45) across all k.

J.2.2 Parallel-Decoding-Aware Architecture Design

NFP also provides a system-side criterion for model and architecture design. An architecture optimized for single-position autoregressive decoding is not necessarily optimal for parallel decoding, because parallel decoding requires residual execution headroom for multi-position execution. If the single-position path already consumes the available compute budget, larger N quickly becomes expensive. In contrast, architectures that preserve memory-bound resource slack or expose favorable implementation granularity can provide a larger near-free region for parallel decoding.

This perspective helps interpret architectural choices such as MoE sparsity, routing balance, attention design, and auxiliary prediction modules. Our MoE analysis shows that higher sparsity and balanced routing can enlarge the available NFP region by distributing expert-token granularity slack across more experts, whereas load skew collapses this aggregate slack. More broadly, NFP suggests that parallel-decoding-friendly architectures should not only optimize the cost of the single-position path, but also consider how much near-free capacity remains for candidate, predicted, or updated positions.

J.2.3 Capacity-Normalized Evaluation

NFP provides a way to separate system capacity from algorithmic utilization when evaluating parallel decoding methods. End-to-end speedup alone conflates two factors: how many positions the system can execute cheaply, and how many of those positions become useful generated tokens. By measuring NFP, one can ask a more diagnostic question: given the same near-free execution budget, how effectively does an algorithm convert these positions into useful outputs?

This enables capacity-normalized evaluation. A method with low speedup may be limited because the underlying model-system stack exposes little near-free capacity, or because the algorithm fails to utilize the available capacity through acceptance, prediction accuracy, or convergence efficiency. NFP separates these cases by providing an explicit system-side budget against which algorithm-side utilization can be interpreted.

J.2.4 System and Backend Co-Design

Finally, the NFP principle acts as a bottleneck diagnostic for system optimization. Since the model-level boundary is determined by the minimum of module-level constraints, the limiting term identifies which component restricts near-free parallelism. If the Dense FFN term is limiting, the relevant factors are hardware balance, batch size, and dense weight traffic. If the Attention term is limiting, further gains require changing the attention backend or its query-tile granularity. If the MoE term is limiting, the relevant knobs include expert-token padding, routing distribution, top-k sparsity, and expert grouping.

Thus, the principle supports model-system co-design: it indicates not only how much parallelism is near-free, but also where implementation or architectural changes can expand that region. In this sense, padding and tiling are not merely low-level implementation details; they directly shape the amount of usable parallelism available to parallel decoding algorithms.

J.2.5 Implications for Future Hardware

The NFP principle links hardware evolution to dense-model parallel decoding capacity. For dense models, the hardware-sensitive boundary is the Dense FFN term $\rho s/2b$, while the Attention boundary M_{attn} is mainly fixed by backend-specific query granularity. As shown in Figure 38, ρ grows across GPU generations, leading to a larger Dense FFN boundary, with the Attention boundary shown as an implementation-dependent reference. This trend suggests that future compute-heavy hardware can expand the NFP opportunity in short-context dense-model inference, where larger N can be absorbed by the increasing Dense FFN-side near-free capacity.

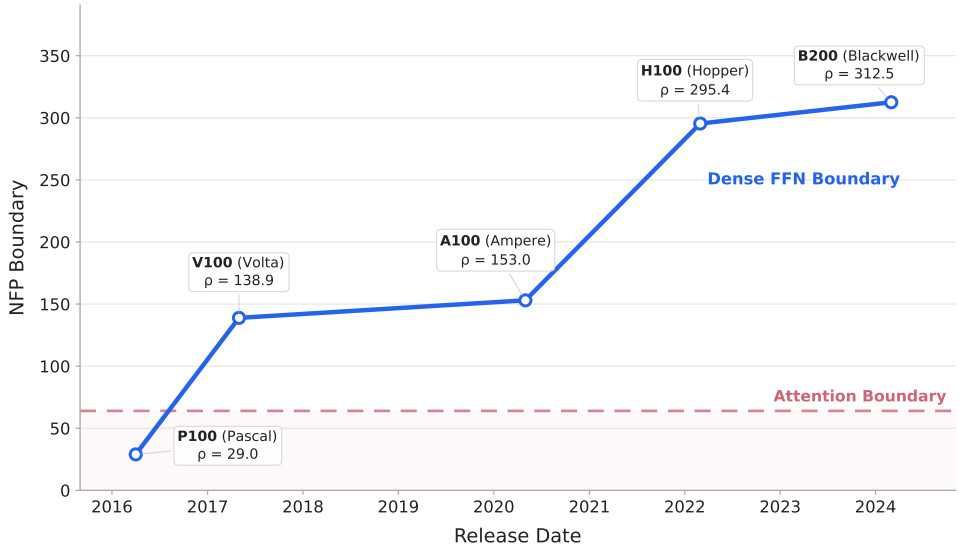


Figure 38: Hardware implication of the NFP principle for dense models. As the compute-to-memory balance point ρ increases across GPU generations, the Dense FFN boundary $\rho s/2b$ grows accordingly, while the Attention boundary M_{attn} remains an implementation-dependent reference set by backend-specific query granularity.