arXiv:2611.19800v1 [cs.LG] 15 Nov 2026

Multi-Branch Transformer

Gaotang Li[1,*,†], **Lingjie Chen**[1,†], **Chunyuan Deng**[2,†], **Zhiwei Xu**[3,†], **Hongzheng Chen**[4,†], **Christopher De Sa**[4], **Jingrui He**[1], **Yingheng Wang**[4,†,§], **Hanghang Tong**[1,§]

[1] University of Illinois Urbana-Champaign [2] Rice University [3] University of Michigan [4] Cornell University

[*] Project lead [†] Core contributors [§] Last authors

Abstract

Transformer language models are commonly scaled by making their blocks wider or stacking them deeper. In this work, we explore a complementary path: **branching**, the parallel composition of full Transformer blocks. We introduce the **Multi-Branch Transformer** (**MBT**), which processes the input through shared initial layers, executes independently parameterized branches of sequential blocks in parallel, and fuses their outputs before passing the result through shared final layers. At comparable training compute relative to a standard transformer, we study three MBT variants: Wide-Shallow preserves block width, Narrow-Deep preserves sequential depth, and Balanced interpolates between these designs. Across controlled pretraining experiments at four mixture-of-experts model scales, ranging from 0.8B to 2.1B total parameters, we show that 2-branch Wide-Shallow MBTs consistently outperform sequential Transformers under matched parameter and training-compute budgets. Balanced MBTs further improve modeling performance over their Wide-Shallow counterparts. These models retain comparable quality as the number of branches increases from 2 to 3 or 4. To further improve inference efficiency at long contexts, we introduce **Joint-KV**, which constructs a shared KV cache from branch representations at each depth. Our final model recipe combines Balanced MBTs with Joint-KV, supported by an inference engine that exploits branch grouping and shared-cache reuse. Fitted scaling curves indicate that standard Transformers require up to **1.26**× as much training body compute to match our final models' validation loss. Across evaluated configurations, these models reduce full-model KV-cache storage by up to **34%** and deliver up to **51.4%** higher decoding throughput on a single H200 GPU. Extensive ablation studies show that branching yields gains beyond changes in width and depth, enables an effective KV-sharing strategy (Joint-KV), remains robust across fusion functions, and performs best in the middle layers. Together, these findings demonstrate that branching offers a practical path for scaling language models toward both better modeling quality and faster inference.

Code: https://github.com/GaotangLi/Multi-Branch-Transformer

Hugging Face: https://huggingface.co/collections/gaotang/multi-branch-transformer

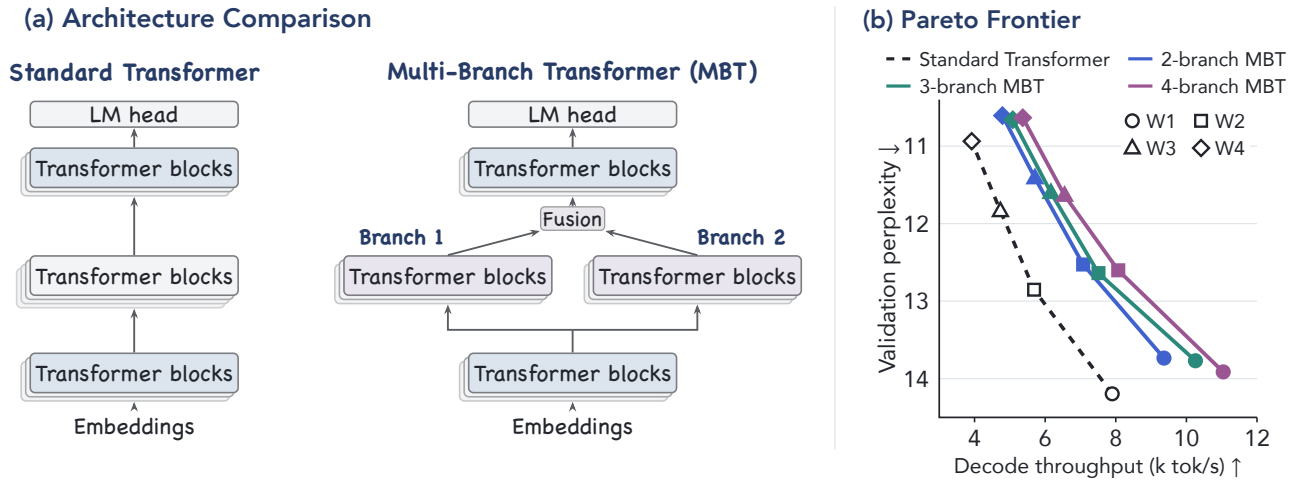

Figure 1. (a) Architecture comparison between a standard transformer and a Multi-Branch Transformer (MBT): MBT composes full blocks in parallel branches. (b) Validation perplexity versus decoding throughput for the Standard Transformer and 2-, 3-, and 4-branch MBTs on a single H200 GPU (16K input tokens, 64 requests). The y-axis is reversed, with the upper right indicating better quality and higher throughput. W1–W4 denote the four model scales in our scaling ladder.

gaotang3@illinois.edu; lingjie7@illinois.edu; chunyuan.deng@rice.edu; zhiweixu@umich.edu; hc676@cornell.edu; cmd353@cornell.edu; jingrui@illinois.edu; yingheng@cs.cornell.edu; htong@illinois.edu

1. Introduction

Scaling Transformer [1] language models involves deciding both how much parameter capacity to allocate and how to organize it [2, 3]. Depth and width are two conventional architectural choices: stacking more blocks in sequence or enlarging the transformations within each block [4–6]. Recent advances in LLM architectures have substantially expanded the design space within this template through innovations in attention mechanisms [7, 8], sparse expert computation [9], and residual connections [10–12], while the *sequential composition* of full blocks remains a common foundation. In this work, we explore **branching**, the parallel composition of full Transformer blocks, as a complementary way to organize and scale model parameters, and ask:

Can Transformers scale more effectively and efficiently through parallel compositions of full blocks?

To this end, we introduce the **Multi-Branch Transformer** (**MBT**), which enables parallel computation at the granularity of full Transformer blocks (Figure 1(a)). An MBT processes the input through shared initial layers (the `prelude`[1]) , splits computation across K independently parameterized branches that execute concurrently, fuses their outputs, and passes the result through shared final layers (the `coda`). The key idea is to preserve sequential composition within each branch while allowing subsequent shared layers to build on the results of multiple transformation paths. In comparison, related work on parallel Transformer architectures explores different design axes (Figure 2): concurrency within individual blocks [14], allocation of width and depth [4–6], or alternative parallel computation paths [15]. Systems approaches such as tensor and pipeline parallelism [16] are orthogonal to and compatible with this architectural design.

Relative to a standard Transformer, an MBT can be defined in three ways at comparable compute: Wide-Shallow MBTs preserve the baseline's block width and distribute its blocks across branches; Narrow-Deep MBTs preserve the baseline's depth along each branch using narrower blocks; and Balanced MBTs interpolate between these choices by jointly adjusting branch width and depth.

To further exploit the parallel branch structure of MBTs, we propose **Joint-KV**, which constructs a shared key-value (KV) cache from all branch representations at each depth while retaining separate queries for each branch (Figure 4). For K branches, this reduces cache storage in the branch region to $1/K$ of Independent-KV at fixed branch depth and KV dimensions. We combine Joint-KV with Balanced MBTs as our final model recipes (Figure 1(b)). To translate our architectural design into practical inference gains, we develop an inference engine based on nano-vLLM [17, 18] that exploits the parallel branches and shared KV caches through **branch grouping** and head folding, reducing sequential execution steps and redundant KV reads during decoding.

Extensive experiments validate the effectiveness and efficiency of MBT. First, branching improves **language modeling performance**. Across a scaling ladder of mixture-of-experts models (0.8B-2.1B total parameters), our scaling laws first indicate that the best 2-branch Wide-Shallow MBT matches a standard Transformer trained with **1.23**× as much compute, while allocating approximately 79% of blocks to parallel branches yields performance on par with the standard Transformer. Balanced MBTs achieve the strongest performance among the studied allocations and maintain comparable quality as the number of branches increases. Second, Joint-KV MBTs, together with our inference engine, substantially improve **inference speed** while retaining strong modeling performance. In particular, 2-, 3-, and 4-branch Joint-KV MBTs match standard Transformers trained with **1.16–1.26**× as much training body compute, while reducing KV cache size by **19–34%**. On a single H200 GPU, our models achieve up to **51.4%** higher decoding throughput than the standard Transformer with 64K-token inputs and 8 concurrent requests.

Extensive ablation studies spanning five scaling sweeps further demonstrate that the benefits of branching extend beyond conventional adjustments to model width and depth. At matched KV-cache capacity, sharing KV across

[1]We use the terminology `prelude` and `coda` following Geiping et al. [13].

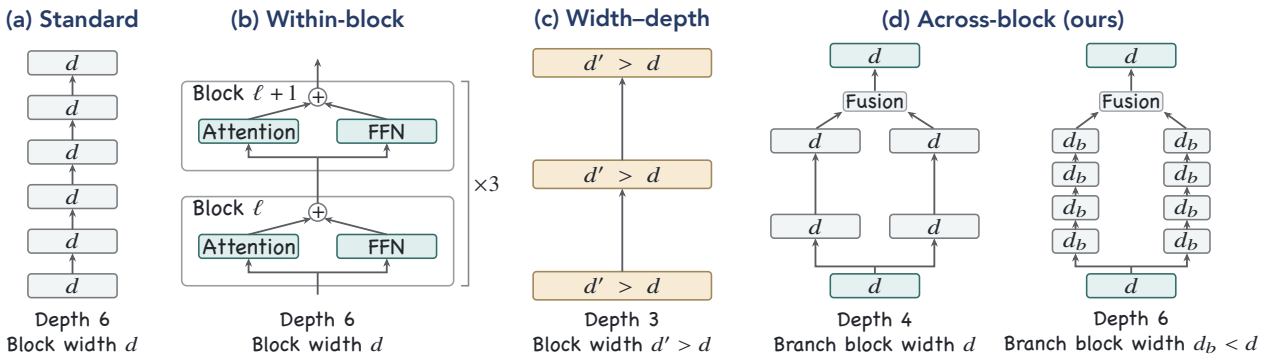

Figure 2. Comparison of Transformer computation structures. (a) Standard sequential blocks; (b) within-block parallel attention and FFN, as in PaLM [14]; (c) wider, shallower stacks; and (d) MBTs with concurrent, independently parameterized branches of full Transformer blocks. Depth denotes the number of sequential blocks along a computation path, excluding fusion. MBTs explore an orthogonal axis of parallelism at the granularity of full Transformer blocks.

branches (Joint-KV) achieves better language modeling performance than keeping a private KV cache in each branch, highlighting the benefit of exploiting MBT's branch structure for improving long-context inference efficiency. We also find that MBTs are robust to the choice of fusion function and perform best when branches are placed in the middle layers. Together, these findings highlight the organization of full Transformer blocks as a promising architectural dimension for improving both language modeling quality and inference efficiency.

2. Multi-Branch Transformer

An MBT replaces part of a serial Transformer stack with independently parameterized branches (Figure 1(a)). Shared blocks precede the branches and follow their fusion; we call these stacks the `prelude` and `coda`, respectively.

2.1 Architecture

Let $x = (x_1, \ldots, x_T)$ be an input token sequence and $H \in \mathbb{R}^{T \times d}$ its hidden states at a block's input, where d is the residual width. Each block $\mathcal{T}_\ell$ applies causal attention followed by a mixture-of-experts (MoE) feed-forward module [19]:

$$H' = H + \mathrm{Attn}_\ell(H), \qquad \mathcal{T}_\ell(H) = H' + \mathrm{MoE}_\ell(H'). \tag{1}$$

Here H' is the hidden state after the attention residual update; the blocks' pre-normalization is omitted for clarity. A standard Transformer embeds the tokens and composes L blocks before decoding:

$$X = \mathrm{Emb}(x), \qquad \widehat{Y} = \mathrm{Dec}\big((\mathcal{T}_L \circ \cdots \circ \mathcal{T}_1)(X)\big). \tag{2}$$

The decoder Dec applies final normalization, the LM head, and softmax; $\widehat{Y}_t$ is the predicted distribution of x_{t+1}.

An MBT replaces the serial middle stack with $K \geq 2$ independently parameterized branch stacks $\mathcal{G}_i$. The `prelude` $\mathcal{P}$, fusion $\mathcal{F}$, and `coda` $\mathcal{C}$ connect them:

$$H = \mathcal{P}(X), \quad Z_i = \mathcal{G}_i(H), \quad \widehat{Y} = \mathrm{Dec}\big(\mathcal{C}(\mathcal{F}(Z_1, \ldots, Z_K))\big). \tag{3}$$

Here H is the shared `prelude` output, and $\mathcal{G}_i$ is a stack of L_i blocks ($i = 1, \ldots, K$). When widths differ, $\mathcal{P}$ includes a shared input projection to the branch width.

Fusion. At each token position, we normalize the branch outputs, concatenate them along the hidden dimension, and project the result back to the shared width d:

$$\mathcal{F}(Z_1, \ldots, Z_K) = \mathrm{Concat}\big(\mathcal{N}_1(Z_1), \ldots, \mathcal{N}_K(Z_K)\big) W_F. \tag{4}$$

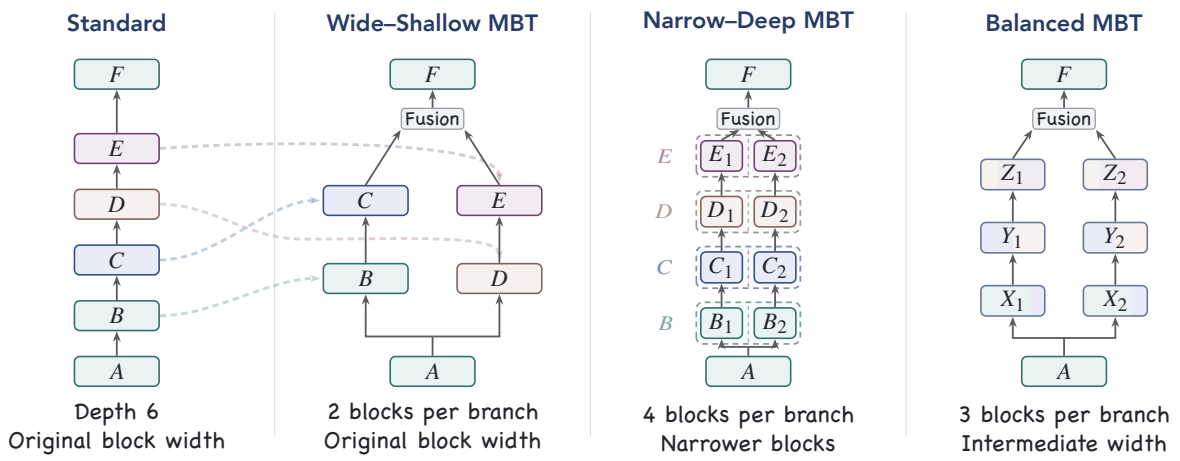

Figure 3. Three width–depth allocations for MBTs. Relative to a standard Transformer under matched compute, (1) Wide-Shallow MBT preserves block width and redistributes the middle blocks across branches; (2) Narrow-Deep MBT preserves the depth along each branch using narrower blocks; and (3) Balanced MBT uses intermediate width and depth.

Each $\mathcal{N}_i$ is an RMSNorm, and W_F is a bias-free linear map. The projected output goes directly to the `coda`, with no separate branch residual. We explore more expressive fusion in Section 4.3, but find Eq. 4 to be sufficient.

2.2 Width–Depth Allocation

Let P and C denote the numbers of `prelude` and `coda` blocks (L_{in} and L_{out} in Figure 1(a)), and L_i denote the number of blocks in branch i. The total block count L_{physical} and depth D of an MBT are

$$L_{\text{physical}} = P + \sum_{i=1}^{K} L_i + C, \qquad D = P + \max_{i=1,\ldots,K} L_i + C. \tag{5}$$

With the shared `prelude` and `coda` fixed, we compare three branch width–depth allocations under matched training compute:

- **Wide–Shallow:** preserve the original block width and redistribute the middle blocks across parallel branches.
- **Narrow–Deep:** preserve the original sequential depth along each branch while reducing block width.
- **Balanced:** use intermediate branch width and depth between these two endpoints.

For Narrow–Deep and Balanced, we adjust the residual width and MoE dimensions within each branch to match the training FLOPs of the Standard reference. All other configurations (*e.g.,* total expert, top-k) remain unchanged.

Figure 3 illustrates these variants using a six-block Transformer, $A \to B \to C \to D \to E \to F$. Here, A and F serve as the shared `prelude` and `coda`, respectively, while the middle stack is reorganized into two parallel branches. Wide–Shallow places two original-width blocks in each branch; Narrow–Deep uses four narrower blocks per branch; and Balanced uses three blocks per branch with an intermediate width.

2.3 KV-Cache Sharing and Infrastructure Design

In the default configuration, which we call Independent-KV, each block in each branch constructs keys and values from its own hidden states and maintains a separate KV cache, so more branches result in more caches. To improve the long-context efficiency of MBTs, we introduce Joint-KV, which shares the KV cache across branches per depth, and an inference engine that turns the parallel branches and the shared caches into faster decoding. Joint-KV changes the model and is used in training. The engine changes only how the model is executed.

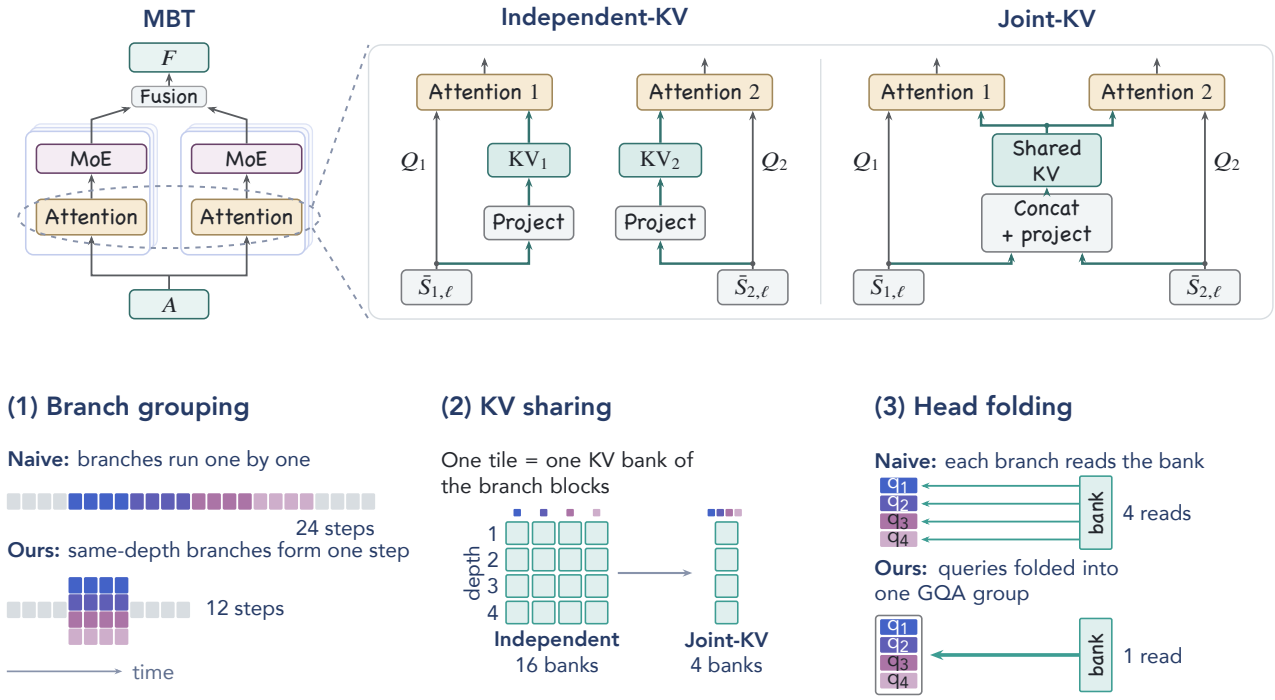

Figure 4. KV-cache sharing and serving of MBTs. Top: KV-cache construction at one branch depth. Independent-KV constructs a separate cache from each branch input. Joint-KV projects the concatenated branch inputs into one shared cache, while retaining branch-local queries. Bottom: naive versus optimized serving for MBTs (Example in figure: 4-branch MBT at scale W3). (1) Branch grouping executes same-depth blocks in one stage. (2) We call the KV cache of one KV projection a *KV bank*. Independent-KV stores one bank per branch block, and Joint-KV stores one bank per branch depth. (3) All branches read a shared KV bank in one attention call. More details are in Figure 12 in Appendix B.1.

Joint-KV. In Joint-KV, all branches share one KV cache at each depth, constructed jointly from their pre-attention inputs, while queries remain local to each branch. Formally, let $\bar{S}_{i,\ell} \in \mathbb{R}^{T \times d_b}$ denote the RMS-normalized input to branch i at depth ℓ. The two KV-cache constructions are

$$\mathrm{KV}^{\mathrm{ind}}_{i,\ell} = \bar{S}_{i,\ell} W^{\mathrm{ind}}_{i,\ell}, \qquad \mathrm{KV}^{\mathrm{shared}}_{\ell} = \frac{1}{\sqrt{K}} [\bar{S}_{1,\ell}; \ldots; \bar{S}_{K,\ell}] W^{\mathrm{shared}}_{\ell}. \tag{6}$$

Let d_b denote the branch width, h_{KV} the number of KV heads, and d_h the head dimension. At each depth, Independent-KV uses K learned projection matrices $W^{\mathrm{ind}}_{i,\ell} \in \mathbb{R}^{d_b \times (2h_{\mathrm{KV}} d_h)}$, whereas Joint-KV uses a single $W^{\mathrm{shared}}_{\ell} \in \mathbb{R}^{(Kd_b) \times (2h_{\mathrm{KV}} d_h)}$. The factor 2 accounts for keys and values. Both constructions therefore have the same KV-projection parameter count and matrix-multiplication FLOPs. $[\cdot;\cdot]$ concatenates along the hidden dimension. Subsequent key normalization and RoPE are omitted for clarity. Projection weights and caches remain depth-specific: Joint-KV shares only across branches at the same depth, requiring synchronization before attention.

Cache Size. At fixed branch depth and KV dimensions, the Joint-KV cache in the branch region is $1/K$ that of Independent-KV, so the savings increase with the branch count.

Serving. Our inference engine, built on nano-vLLM [18], uses the same underlying kernel backend and cache manager for MBTs and Standard. A decode step processes the `prelude`, branch region, and `coda` in order. We use two additional execution mechanisms, branch grouping and head folding, to run the branches together and reduce repeated KV reads (Figure 4).

Branch Grouping. Our engine groups blocks at the same depth across branches into a batched execution stage. With Joint-KV, the engine first constructs the shared KV cache from the branch input representations,

then computes the branch-specific attention and MoE outputs together. Compared with executing every block sequentially, branch grouping reduces the number of sequential block stages from L_{physical} to $D = P + \max_i L_i + C$ (Eq. 5). For example, a 4-branch W3 Balanced MBT contains 24 physical blocks but requires only 12 sequential block stages with branch grouping, compared with 16 for Standard.

Head Folding. Sharing KV caches reduces storage, but separate attention calls for different branches can still reload the same KV data. *Head folding* folds the branch dimension into the query-head dimension, grouping queries by their shared KV head so that all branches use a single grouped-query attention (GQA) call [20]. This allows the attention kernel to reuse shared KV data across branches, reducing redundant memory traffic. Each query head still computes its own softmax over the same keys and values, preserving the attention function itself.

Distributed Execution. MBTs can be distributed through tensor parallelism or branch parallelism. With Joint-KV, these layouts differ in shared KV-cache construction. With tensor parallelism, devices replicate branch hidden states while sharding projection weights and attention heads. Each device constructs its assigned shared KV heads locally, without an additional collective. Head folding also remains local when the corresponding query heads from all branches are placed on the same device. Alternatively, branches can run on different devices. Each device computes its branch's contribution to the shared KV projection, and an all-reduce combines these contributions before attention at each depth. The all-reduce operates on a tensor of width $2h_{\text{KV}}d_h$, avoiding an all-gather of branch representations of total width Kd_b. At fixed KV dimensions, its per-token tensor size is independent of the branch count, although communication cost depends on the group size and topology. This layout distributes branch weights and computation across devices but requires synchronization at every branch depth. This layout may be useful when branch sharding is needed to fit the model in memory and fast interconnects are available.

3. Experiments

3.1 Scaling Ladder Setup

We pretrain standard Transformers and MBTs at four scales, W1–W4, with standard models ranging from 0.80B to 2.07B total parameters (139M–394M active-body parameters). All models use FineWeb-Edu [21], the Llama 3 tokenizer [22], and a context length of 4,096 tokens. Our scaling ladder is comparable in scale to that of Attention Residuals [12], a recent industrial technical report. Our training recipes, experimental protocols, and backbone components also draw on recent technical reports [23–26]. At each scale, Standard and MBT models share a training-token budget of approximately 200 tokens per parameter (TPP) [23], using active-body parameter count and excluding embeddings and the LM head. A brief overview of our ladder is provided in Table 1.

Our backbone uses pre-RMSNorm [27] blocks with full causal GQA [20], a 2:1 query-to-KV head ratio, and a head dimension of 64. We apply headwise Q/K normalization, RoPE [28], and sigmoid gating of attention outputs [29]. Each block contains a latent MoE [25] with 64 SwiGLU experts and top-8 routing. Quantile Balancing [26] balances expert loads without an auxiliary balancing loss.

We train in BF16 using AdamW [30], with $\beta = (0.9, 0.95)$, weight decay of 0.1, and gradient clipping at 1.0. The learning rate warms up linearly over the first 1% of training, then follows a cosine schedule to one tenth of its peak. Our primary peak-learning-rate candidates are $\{1, 3, 5, 7, 10, 30\} \times 10^{-4}$. We extensively tune the learning rate for the standard Transformer at each scale to establish competitive baselines. For MBTs, we tune the learning rate on a single configuration and reuse the selected value across the remaining settings. Global batches contain 128 sequences for W1/W2 and 256 for W3/W4. Table 7 in Appendix A.1 provides the full experimental details.

3.2 Scaling Transformers with Parallel Branches

We first study how allocating blocks, width, and depth across branches affects language-model quality. In this subsection, the experiments use independent-KV caches in each branch.

Table 1. Scaling ladder and best Wide–Shallow 2-branch MBTs from Figure 5. Standard and MBT share the backbone dimensions, physical block count, and training tokens at each scale.

Shared backbone				Tokens	Selected MBT	Parameters (B / M)		Loss ↓	
Scale	L	d_{model}	d_{ffn}	(B)	Allocation	Standard	MBT	Standard	MBT
W1	12	1024	512	27.845	$4/(2+2)/4$	0.799 / 139.2	0.801 / 141.3	2.653	**2.622**
W2	14	1152	576	41.088	$5/(2+2)/5$	1.134 / 205.4	1.136 / 208.1	2.554	**2.525**
W3	16	1280	640	57.942	$5/(3+3)/5$	1.555 / 289.7	1.558 / 293.0	2.471	**2.444**
W4	18	1408	704	78.841	$6/(3+3)/6$	2.074 / 394.2	2.077 / 398.2	2.392	**2.367**

L: physical attention–MoE blocks; $P/(A+B)/C$: `prelude`/branches/`coda`. Parameter pairs give total (B) / active-body (M), including/excluding the tied embedding/head, respectively. The MoE latent width equals d_{ffn}. Bold marks lower validation loss.

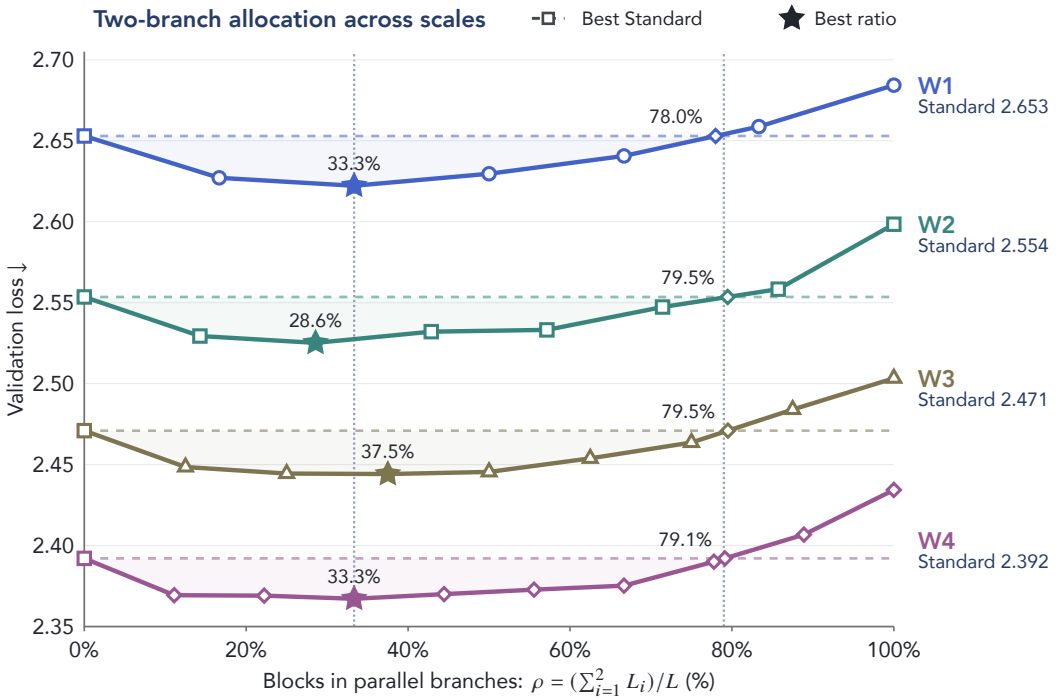

Figure 5. Scaling the parallel block ratio. All four scales exhibit a consistent U-shaped trend: allocating approximately $1/3$ of blocks to parallel branches reduces loss by 0.025–0.031 relative to Standard, while increasing this fraction to approximately 79% maintains performance on par with Standard.

3.2.1 Scaling Laws of 2-branch MBTs

Research Question. What fraction of a Transformer's blocks can be placed in two parallel branches while preserving or improving model quality?

Setup. Using Wide–Shallow MBTs (where total training FLOPs and parameters remain almost identical to the standard reference model), we vary the fraction of physical blocks in two equal-depth branches, $\rho = (\sum_{i=1}^{2} L_i)/L$, from zero (Standard) to one (fully branched). Within each scale, we fix the backbone width, total physical block count, and training tokens, and split the remaining blocks equally between the `prelude` and `coda` ($P = C$). Branch outputs are fused by normalized concatenation followed by a linear projection. We compare final validation loss across allocations (Figure 5), then select the lowest-loss allocation at each scale to study body-compute scaling and zero-shot downstream performance (Figure 6). Table 1 lists these selected allocations.

Result and Interpretation. All four scales exhibit a highly consistent U-shaped relationship between parallel block ratio and validation loss. The best tested allocations cluster around $1/3$ of the blocks (28.6–37.5%), reducing

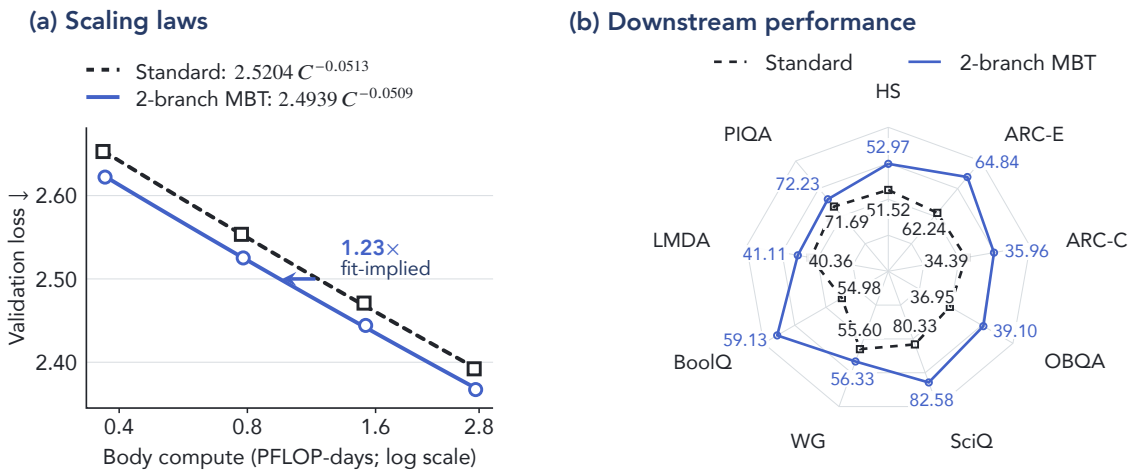

Figure 6. Compute scaling and downstream performance. Best Wide-Shallow two-branch MBTs versus Standard across W1–W4. (a) Validation loss versus training body compute, with power-law fits. (b) Zero-shot task scores averaged over W1–W4. MBTs improve compute efficiency and mean accuracy on all nine tasks.

loss by 0.025–0.031 relative to Standard. Larger parallel regions also retain Standard-level quality: the largest measured ratios no worse than Standard are 66.7%, 71.4%, 75.0%, and 77.8% for W1–W4.

The sweep adds one block to each branch at a time, giving a discrete ratio increment of $2/L$; exact equality with Standard need not fall on a sampled allocation. We therefore linearly interpolate between neighboring observations to estimate the equality points. All four crossings fall within 78–80%. The consistent curve shape, minima near $1/3$, and tightly clustered crossings suggest a stable block-allocation pattern across model scales.

The fitted scaling curves indicate an approximately **1.23**× body-compute advantage for the selected two-branch MBTs over Standard at matched validation loss (Figure 6(a)). The downstream gains are also consistent across tasks. Averaged over W1–W4, MBT outperforms Standard on *all nine* benchmarks, raising overall mean zero-shot accuracy from 54.23% to 56.03% (Figure 6(b)).

3.2.2 Scaling Laws of multi-branch MBTs

Research Question. Do the language modeling gains persist beyond two branches, and how do they depend on the allocation of width and depth?

Setup. We compare MBTs with 2, 3, and 4 branches using the Wide–Shallow and Balanced variants defined in Figure 3, matching training compute at each scale.[2] Relative to the standard model at each scale, we retain **half of the blocks** in the shared `prelude` and `coda`, keeping their block configurations unchanged, and reorganize the middle region into parallel branches. Wide–Shallow models place half of the original blocks in this region, preserving their configurations and distributing them as evenly as possible across branches. Branch depths may differ when this block count is not divisible by the number of branches. Balanced models instead adjust branch widths and depths to match the reference training body compute; their total physical block count and fraction of blocks in branches can therefore differ. Table 2 illustrates an examplar W3 settings. Figure 7 plots final loss against body compute across W1–W4 with 7 fitted scaling curves; Table 9 lists all configurations.

Result and Interpretation. Increasing the branch count from 2 to 3 or 4 yields a mean absolute change in validation loss below 0.006 across both families and all four scales, while every configuration outperforms Standard. MBTs

[2]Our 2-branch Narrow–Deep results suggest benefits from retaining depth as models scale (Section 4.1). With 3 or 4 branches, however, preserving full depth at fixed compute requires much smaller widths and is accompanied by poorer quality, especially with 4 branches (Appendix A.4.1). We therefore study Balanced MBTs, trading some depth for greater width within the same compute budget.

Table 2. W3 width–depth allocations at comparable compute. Full configurations appear in Table 9.

Model	K (# branches)	Allocation	d	m	f	L/D	Body	Loss ↓
Standard	—	16 (serial)	1280	640	640	16/16	1.5032	2.471
Wide–Shallow	2	$4/(4+4)/4$	1280	640	640	16/12	1.5164	2.445
Wide–Shallow	3	$4/(3+3+2)/4$	1280	640	640	16/11	1.5230	2.457
Wide–Shallow	4	$4/(2+2+2+2)/4$	1280	640	640	16/10	1.5296	2.451
Balanced	2	$4/(6+6)/4$	1024	512	512	20/14	1.5301	2.436
Balanced	3	$4/(5+5+5)/4$	896	432	448	23/13	1.5160	2.446
Balanced	4	$4/(4+4+4+4)/4$	896	400	400	24/12	1.5148	2.449

d, m, f: residual, MoE latent, and expert FFN widths; L/D: physical blocks / depth. Body compute: PFLOP-days.

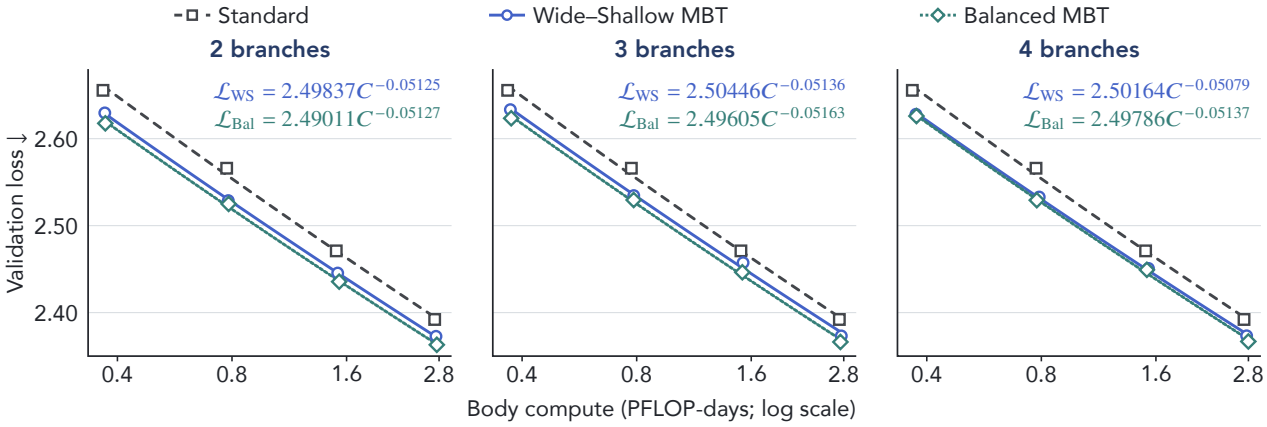

Figure 7. Scaling to more parallel branches. Across W1–W4, increasing from 2 to 4 branches largely preserves performance. Balanced MBTs improve the performance through joint width–depth tuning.

can therefore expose more concurrent branch executions while maintaining their quality advantage. Balanced MBTs achieve lower loss than their Wide–Shallow counterparts at every scale and branch count. For all 3 branch counts, the fitted law $L = AC^{-\alpha}$ yields both a lower A and a slightly larger α for Balanced models (Figure 7). These results suggest that MBTs offer a rich design space: joint width–depth tuning can improve performance beyond fixed-width branching within a given compute budget.

3.3 Scaling MBTs with Joint-KV

Research Question. Can our final recipe, Balanced MBTs with Joint-KV, simultaneously improve both language-model quality and inference speed over standard Transformers?

Setup. We reuse the Balanced MBTs from Section 3.2.2 as Independent-KV baselines, covering 2, 3, and 4 branches across W1–W4. For each configuration, we train a Joint-KV counterpart with the same block allocation, widths, and training settings to isolate the effect of KV sharing. Table 3 shows the W4 setup as an example. Full configurations and results appear in Appendix A.5. We report the theoretical full-model KV-cache savings relative to Standard at the same scale, context length, and cache precision.

Language Modeling. Figure 8 compares final validation loss against each variant's body compute. Joint-KV closely tracks Independent-KV across scales and branch counts. All 12 Joint-KV models achieve lower validation loss than Standard, retaining the gains from branching with significantly smaller KV cache size. The gains also extend to downstream evaluation: averaged equally over W1–W4 and the nine zero-shot tasks, 2-, 3-, and 4-branch Joint-KV MBTs score 55.92%, 55.56%, and 55.47%, respectively, compared with 54.23% for Standard. Appendix Table 12 records the full quantitative results.

Inference Efficiency. We benchmark decoding on one NVIDIA H200 GPU with the engine of Section 2.3, which

Table 3. W4 Balanced MBTs with Independent-KV and Joint-KV. Cache capacity and savings are for Joint-KV relative to Standard; full W1–W4 results appear in Table 11.

Architecture		Block widths			Loss ↓		KV cache	
Model	Allocation	d	m	f	Independent	Joint	KiB/token	Saved
Standard	18 (serial)	1408	704	704	2.392	–	49.50	–
$K = 2$	$4/(7+7)/4$	1152	592	592	2.363	**2.361**	37.75	23.74%
$K = 3$	$4/(6+6+6)/5$	896	496	496	2.366	2.366	35.25	28.79%
$K = 4$	$4/(6+6+6+6)/5$	768	432	432	2.367	**2.364**	33.75	31.82%

K: number of branches. Widths apply to Standard blocks or MBT branches; shared outer blocks use Standard widths. Cache capacity uses BF16.

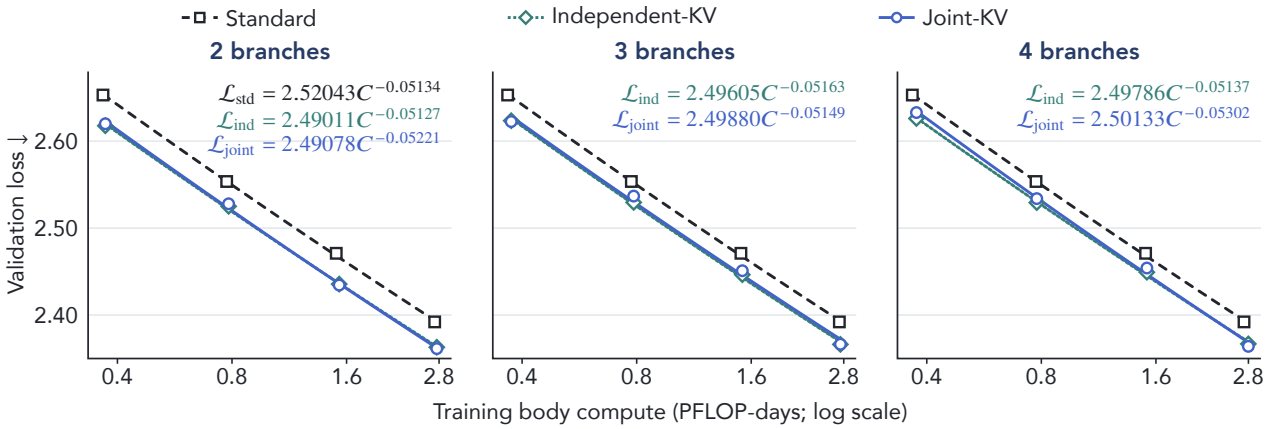

Figure 8. Joint-KV preserves strong scaling performance. Joint-KV closely tracks Independent-KV across branch counts and improves on Standard while reducing cache capacity.

serves Standard and MBTs identically. We test the Balanced Joint-KV MBTs of Table 11 with two and four branches at all four scales. Each request has a 4K-token prompt from FineWeb-Edu [21] or a 16K-, 64K-, or 128K-token prompt from RULER [31], and generates 1K tokens with temperature 1.0 and top-p 0.95. The engine keeps N requests in flight, with $N \in \{1, 8, 64\}$ ($\{1, 8, 32\}$ for 64K-token and $\{1, 8, 16\}$ for 128K-token prompts). We define *decode speedup* as the decode throughput of an MBT, *i.e.*, generated tokens per second excluding prefill, divided by that of Standard. Each MBT runs on the same GPU as its Standard baseline. We measure Standard before and after the MBTs and use the mean of the two Standard runs. On average, the two Standard measurements differ by 2.0% at $N = 1$ and 0.6% for $N \geq 8$. With the 1K generated tokens, every sequence exceeds the 4K training context, so these runs measure speed only.

Figure 9 shows the results. Averaged over the four scales, the 4-branch Joint-KV MBT decodes faster than Standard in every setting. Its speedup ranges from 1.12× with a single 4K-token request to 1.50× with 16 concurrent 128K-token requests. It is at least 1.08× faster at every scale and setting. Averaged over scales, two branches range from 1.01× to 1.32×. Four branches outperform two at every scale and in every setting, since they need fewer serial steps and read fewer KV bytes. These speedups come without a quality cost: at every scale, both models reach lower validation loss than Standard (Table 11).

Sources of the Speedup. The time of a decode step has two parts. The first is a fixed cost for executing the serial blocks. It depends on neither the number of requests nor the input length. Branch grouping runs the blocks of all branches at the same depth as one step, so an MBT pays this cost for fewer blocks. The depth of Standard divided by that of the MBT is the depth ratio r_d (Table 15). The second part is the time to load data from memory and compute on it, which grows with the concurrency and the context length. For the models we test, this part is bound by memory traffic. With Joint-KV and head folding, each depth of the branch region reads its shared KV cache

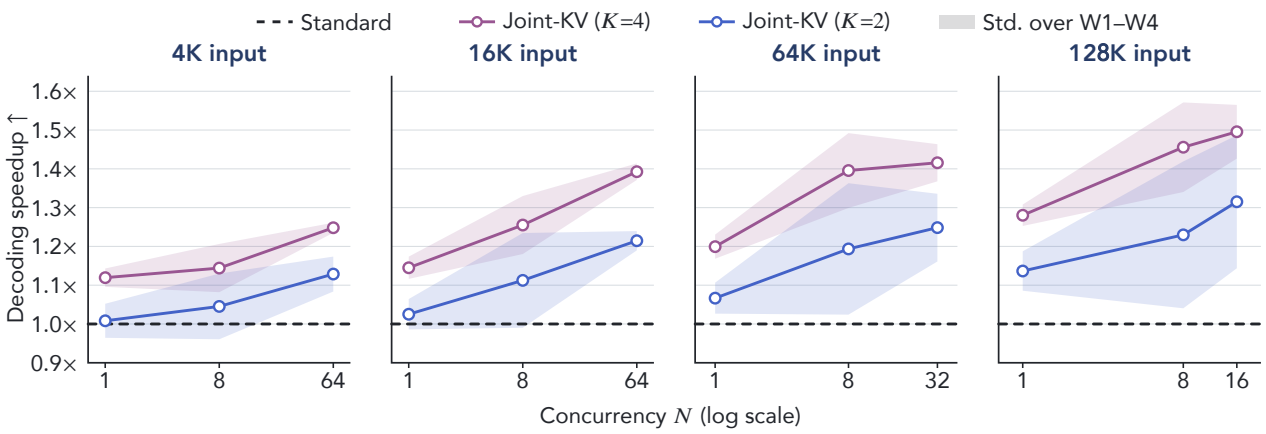

Figure 9. Decode speedup over Standard. Decode throughput of Balanced Joint-KV MBTs relative to Standard on one H200 GPU, averaged over scales W1–W4, as a function of the number of concurrent requests N, for 4K-, 16K-, 64K-, and 128K-token inputs that each generate 1K tokens. Shaded bands show one standard deviation across scales.

once. An MBT therefore reads only $1 - s$ of the KV bytes of Standard, where s is the KV saving of Table 11. We denote by F the fixed cost and by M the KV-read time of Standard. The speedup is then approximately

$$S \approx \frac{F + M}{F/r_{\mathrm{d}} + (1 - s)\, M}. \tag{7}$$

Our settings span 4K to 128K tokens and 1 to 64 requests. With short inputs and low concurrency, M is small compared with F, and the speedup comes mainly from the depth ratio. As the context length grows, KV reads take most of each step, and the speedup approaches the bound $1/(1 - s)$. Appendix B.3 gives the profiles and a more detailed analysis. Appendix B.4 measures the contribution of each engine mechanism.

4. Ablation Studies

4.1 Width and Depth Ablations

Ablation Goal. We test whether the gains of branching can be explained by a shallower, wider network, and whether they persist when Standard's depth is preserved.

Setup. We retain Standard and 2-branch Wide–Shallow MBT as fixed references at each scale, with roughly half the blocks in parallel branches in the latter. We add three controls. *Global wide–shallow* widens the entire sequential stack and matches Wide–Shallow MBT's depth, following the conventional width–depth trade-off [6]. *Local wide-core* is our localized control: it replaces the two branches with one wider core of the same depth, preserving the `prelude` and `coda`. Both controls test whether fewer, wider sequential blocks can recover the gains of Wide–Shallow MBT at comparable compute. Narrow–Deep MBT instead keeps the same `prelude` and `coda`, together roughly half of Standard's blocks, and gives each of two narrower branches the full depth of Standard's middle stack. It thus serves as a depth-matched ablation against Standard models. Table 4 gives a W1 example. Full configurations appear in Table 13.

Result and Interpretation. Figure 10 shows that Wide–Shallow MBT outperforms both widening baselines (Global Wide–Shallow and Local Wide–Core) at all four scales, with a consistently better-fitted scaling curve. Narrow–Deep MBT also outperforms Standard across all scales and exhibits the steepest scaling exponent ($\alpha = 0.0548$, versus 0.0515 for Wide–Shallow), suggesting that branching improves modeling quality even when the original sequential depth is preserved. Together, these results demonstrate benefits beyond sequential widening at matched depth, while indicating that the best MBT can be tuned by jointly considering width and depth.

Table 4. Example of the ablation setting at W1 with fixed data and approximately matched body compute.

Family	Allocation	Blocks / depth	Outer or all-layer $d/m/f$	Core or branch $d/m/f$	Body compute (PFLOP-days)
Standard	12	12/12	1024/512/512	—	0.3665
Global wide–shallow	9	9/9	1280/640/512	—	0.3721
Local wide-core	3/3/3	9/9	1024/512/512	1664/640/640	0.3636
Wide–Shallow MBT	3/(3 + 3)/3	12/9	1024/512/512	1024/512/512	0.3706
Narrow–Deep MBT	3/(6 + 6)/3	18/12	1024/512/512	640/320/416	0.3705

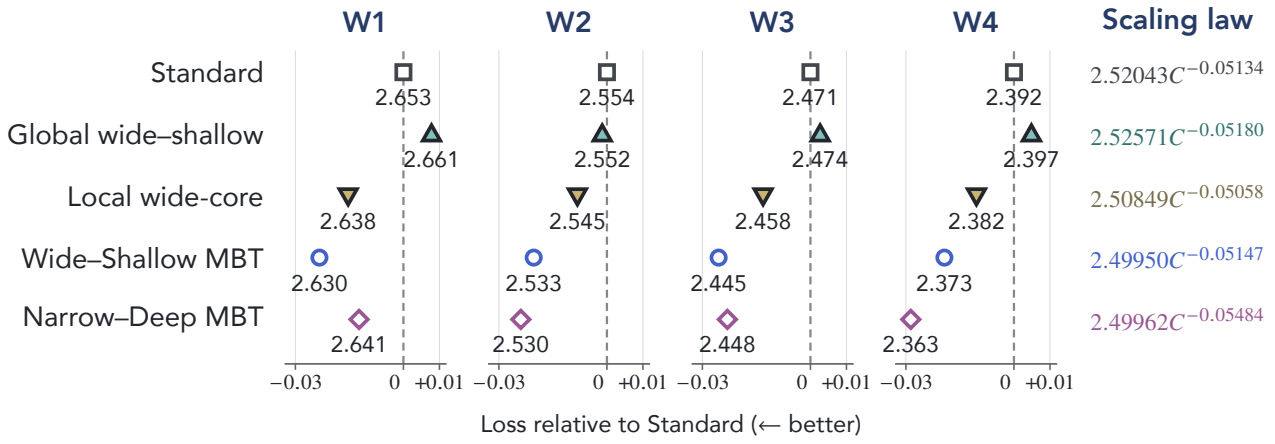

Figure 10. Ablation studies on branching. Positions show loss differences from Standard; labels give absolute loss. The right column fits the scaling laws. We find branching is the key to performance improvement.

4.2 KV Sharing Across Branches

Ablation Goal. At the same KV-cache size and training budget, does constructing shared keys and values jointly across branches (**Joint-KV**) improve modeling quality over keeping smaller, independent caches in each branch?

Setup. We start from the 2-branch Balanced MBTs at W1–W4. On the main ladder, the shared KV heads cannot always be split evenly between two branches (e.g., W2 and W4), so an exactly cache-matched Independent-KV control does not exist at every scale. We therefore keep block allocations and widths and re-derive the head counts so that every pair can be matched exactly. In doing so, we also raise the GQA ratio from the 2:1 of the main recipe to 8:1 or higher (Table 5), since how KV heads are allocated across branches matters most when few are available. We compare two constructions with **identical KV-cache size** (2 KV heads per branch depth in total): **Independent-KV** gives each branch 1 private KV head; **Joint-KV** constructs 2 KV heads per branch depth, shared by both branches. All other settings, including training tokens and LR candidates, are identical within each pair (Appendix A.7).

Table 5. Joint-KV versus Independent-KV across W1–W4 at matched KV-cache capacity. Each pair shares block allocations, widths, query heads, and training tokens. Training body compute is in PFLOP-days; full configurations in Table 14.

		Q/KV heads				Body compute		Validation loss ↓		
Scale	Allocation	Outer (ratio)	Independent-KV	Joint-KV	KV/token (KiB)	Independent-KV	Joint-KV	Independent-KV	Joint-KV	Δ
W1	3/(5 + 5)/3	16/2 (8:1)	16/1	16/2	5.5	0.3848	0.3867	2.657	**2.646**	−0.011
W2	3/(6 + 6)/3	18/2 (9:1)	16/1	16/2	6.0	0.7723	0.7763	2.568	**2.557**	−0.011
W3	4/(6 + 6)/4	20/2 (10:1)	16/1	16/2	7.0	1.4436	1.4499	2.499	**2.487**	−0.012
W4	4/(7 + 7)/4	24/3 (8:1)	18/1	18/2	9.5	2.6551	2.6664	2.392	**2.380**	−0.012

Joint-KV's KV heads are shared by both branches; Independent-KV heads are per branch, so both store 2 KV heads per branch depth. Independent-KV here keeps 1 KV head per branch to match Joint-KV's cache, unlike the Independent-KV models in Table 3, which use the main 2:1 recipe.

Result and Interpretation. Joint-KV achieves lower validation loss at all four scales, by 0.011–0.012 (Table 5). Its larger KV projection adds less than 0.6% body compute, far too little to explain the gap. This result shows that, at a fixed cache size, sharing KV across branches is more effective than keeping a private KV cache in each branch.

4.3 Fusion Mechanisms

Ablation Goal and Setup. We test whether fusion needs a learned projection and whether more expressive operators improve MBT. We fix the setting to be Wide-Shallow 2-branch MBT of the configuration $4/(4+4)/4$ at scale W3 (half of the parameters in parallel branches), changing only the fusion operator. Since fusing multiple vectors is extensively used in multimodal research, we adapt gating, joint-representation, and bilinear mechanisms from highly-cited existing papers [32–34], and also test joint SwiGLU and tokenwise softmax aggregation. We additionally test a normalized sum, $(h_1 + h_2)/\sqrt{2}$, whose aggregation introduces no learned parameters beyond the existing branch RMSNorms. Table 6 gives the formulations and results.

Table 6. Fusion of normalized branch vectors h_1, h_2 in W3 $4/(4+4)/4$. $W = [W_1\ W_2]$, A_i, B_i, and P are learned weight matrices; $a_i = A_i W_i h_i$, $b_i = B_i W_i h_i$, and ϕ = SiLU. g is a channelwise sigmoid gate and β is a tokenwise softmax over branches; $\odot$ denotes elementwise multiplication. The last row tests parameter-free aggregation.

Fusion	Fused output z	Loss ↓
Concat–projection [35]	$W[h_1; h_2]$	2.445
Channel competition [32]	$2[g \odot W_1 h_1 + (1-g) \odot W_2 h_2]$	2.446
Joint MLP [33]	$W[h_1; h_2] + P\phi(a_1 + a_2)$	2.445
Low-rank bilinear [34]	$W[h_1; h_2] + P(a_1 \odot a_2)$	2.446
Joint SwiGLU [36]	$W[h_1; h_2] + P[\phi(a_1 + a_2) \odot (b_1 + b_2)]$	2.444
Token softmax [15]	$2(\beta_1 W_1 h_1 + \beta_2 W_2 h_2)$	2.448
Normalized sum (no projection)	$(h_1 + h_2)/\sqrt{2}$	2.458

Result and Interpretation. Across the six learned fusion operators, the final loss varies by less than 0.005. The projection-free normalized sum reaches 2.458, compared with 2.445 for concat–projection. We therefore retain concat–projection as a simple, effective default: removing the projection hurts quality in this setting, while more elaborate learned fusion offers little benefit.

4.4 Branch Placement

Ablation Goal and Setup. We test *where to branch* by placing the parallel blocks in the early, middle (ours), or late layers. We use a 2-branch Wide–Shallow MBT at scale W3, with 16 blocks in total, 8 of which are distributed across the parallel branches.

Result and Interpretation. As shown in Figure 11, middle placement achieves the lowest loss (2.445), outperforming both early (2.458) and late (2.464) branching. The advantage of early over late branching suggests that retaining a shared `coda` is more beneficial than retaining a shared `prelude` in this setting. Overall, these results favor placing the parallel region in the middle layers.

5. Related Work

Parallelism in large language models. Architectural work on language models explores concurrency within individual blocks, as in PaLM [14], width–depth trade-offs in sequential stacks [4–6], and parallel passes through a shared backbone with learned prefixes, as in ParScale [15]. From a systems perspective, parallelism supports efficient execution across devices: tensor and pipeline parallelism distribute model computation [16], and track

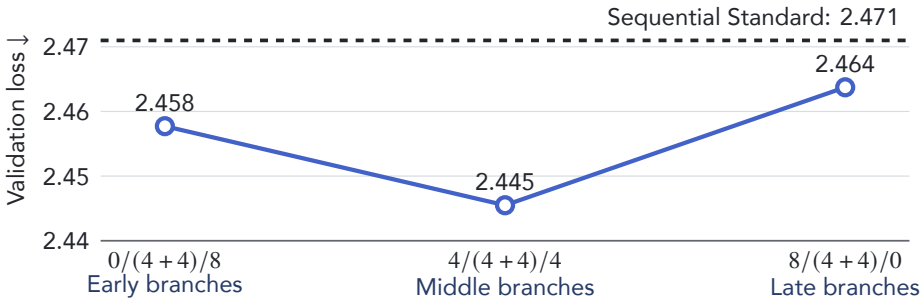

Figure 11. Branching in the middle works best. At W3, we place 8 shared blocks around two 4-block branches while holding all other settings fixed. Shared processing before and after the branches gives the lowest loss.

parallelism uses narrower stacks with periodic synchronization to reduce communication overhead [37]. Our approach explores an architectural design axis orthogonal to those studied in prior work on parallel language models, while remaining compatible with existing systems approaches. More broadly, multi-branch architectures have a long history in computer vision, with prominent examples including Inception [38], ResNeXt [39], FractalNet [40], and HRNet [41]. Earlier NLP work (Pre-LLM era) explored parallel branches within attention modules [42, 43] and encoder backbones [44]. However, parallel composition of full Transformer blocks remains underexplored as a means of scaling modern autoregressive language models.

Information flow and KV-cache sharing. Recent work improves information flow through expanded residual streams and selective aggregation of representations across depth [10–12, 45, 46]. These approaches enrich interactions within a sequential block stack, although maintaining and mixing additional representations can introduce extra memory traffic. MBT takes a different approach by reorganizing full Transformer blocks into parallel branches and fusing their outputs before subsequent shared layers. Prior work shares keys and values across attention heads [20, 47] or across decoder layers, as in Cross-Layer Attention and YOCO [48, 49]. Joint-KV shares along a different axis that exists only in multi-branch models: across parallel branches at the same depth, while retaining branch-specific queries. This axis is orthogonal to head and layer sharing (our models already combine it with GQA) and enables head folding to reduce redundant KV reads during long-context decoding. Our results show that the resulting cache savings and inference gains coexist with stronger language modeling performance than standard Transformers.

6. Conclusion

We introduced MBTs, which organize full Transformer blocks into parallel branches between shared initial and final layers. Across four MoE model scales, branching improves language-model quality and fitted scaling at comparable training compute. Balanced branch allocations, Joint-KV, and an inference engine that exploits branch structure together enable better modeling quality and faster decoding. These results establish branching as a practical architectural choice alongside width and depth. These findings open several directions for future work:

- **Joint Scaling Laws with Branching.** Existing scaling studies guide model and training configurations under a given compute budget [3, 50]. Extending this framework to jointly account for branch count, branch depth, and hidden dimensions could identify how best to allocate parameters within that budget. Incorporating inference cost would further guide configurations that balance modeling quality and serving efficiency.
- **MBT + YOCO + Cross-Layer Attention.** YOCO enables aggressive cache reduction and early-exit prefill [49], while Cross-Layer Attention shares KV across neighboring layers [48]. Recent work such as HySparse2 combines selective sharing across layers with sparse attention [51]. Combining these approaches with MBT

would enable joint design of KV sharing across branches and layers, potentially improving the trade-off between modeling quality, prefill efficiency, and decoding speed.

- **MBTs for Multimodality.** Multimodal Transformers such as MM-DiT already use distinct parameters for different modalities [52]. MBTs could extend this idea to branches that serve as text, image, or audio backbones, with widths, depths, and block types tailored to each modality. Studying where these branches should interact and fuse could improve the balance between modality specialization and cross-modal information exchange.

References

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

[2] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020.

[3] Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, et al. Training compute-optimal large language models. *arXiv preprint arXiv:2203.15556*, 2022.

[4] Yi Tay, Mostafa Dehghani, Jinfeng Rao, William Fedus, Samira Abnar, Hyung Won Chung, Sharan Narang, Dani Yogatama, Ashish Vaswani, and Donald Metzler. Scale efficiently: Insights from pretraining and finetuning transformers. In *International Conference on Learning Representations*, 2022. URL https://openreview.net/forum?id=f2OYVDyfIB.

[5] Sean Michael McLeish, John Kirchenbauer, David Yu Miller, Siddharth Singh, Abhinav Bhatele, Micah Goldblum, Ashwinee Panda, and Tom Goldstein. Gemstones: A model suite for multi-faceted scaling laws. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025. URL https://openreview.net/forum?id=iZk78dZ1Ap.

[6] Song Bian, Minghao Yan, and Shivaram Venkataraman. Scaling inference-efficient language models. In *Forty-second International Conference on Machine Learning*, 2025. URL https://openreview.net/forum?id=wslytZ28Ms.

[7] Kimi Team, Yu Zhang, Zongyu Lin, Xingcheng Yao, Jiaxi Hu, Fanqing Meng, Chengyin Liu, Xin Men, Songlin Yang, Zhiyuan Li, et al. Kimi Linear: An expressive, efficient attention architecture. *arXiv preprint arXiv:2510.26692*, 2025.

[8] Gemma Team, Aishwarya Kamath, Johan Ferret, Shreya Pathak, Nino Vieillard, Ramona Merhej, Sarah Perrin, Tatiana Matejovicova, Alexandre Ramé, Morgane Rivière, et al. Gemma 3 technical report. *arXiv preprint arXiv:2503.19786*, 2025.

[9] DeepSeek-AI, Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. DeepSeek-V3 technical report. *arXiv preprint arXiv:2412.19437*, 2024.

[10] Defa Zhu, Hongzhi Huang, Zihao Huang, Yutao Zeng, Yunyao Mao, Banggu Wu, Qiyang Min, and Xun Zhou. Hyper-connections. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=9FqARW7dwB.

[11] Zhenda Xie, Yixuan Wei, Huanqi Cao, Chenggang Zhao, Chengqi Deng, Jiashi Li, Damai Dai, Huazuo Gao, Jiang Chang, Kuai Yu, et al. mHC: Manifold-constrained hyper-connections. *arXiv preprint arXiv:2512.24880*, 2025.

[12] Kimi Team, Guangyu Chen, Yu Zhang, Jianlin Su, Weixin Xu, Siyuan Pan, Yaoyu Wang, Yucheng Wang, Guanduo Chen, Bohong Yin, et al. Attention residuals. *arXiv preprint arXiv:2603.15031*, 2026.

[13] Jonas Geiping, Sean McLeish, Neel Jain, John Kirchenbauer, Siddharth Singh, Brian Bartoldson, Bhavya Kailkhura, Abhinav Bhatele, and Tom Goldstein. Scaling up test-time compute with latent reasoning: A recurrent depth approach. In D. Belgrave, C. Zhang, H. Lin, R. Pascanu, P. Koniusz, M. Ghassemi, and N. Chen, editors, *Advances in Neural Information Processing Systems*, volume 38, Main Conference, pages 41340–41391. Curran Associates, Inc., 2025. doi: 10.52202/085713-1380. URL https://proceedings.neurips.cc/paper_files/paper/2025/file/3b01972cf31e6fa0fe29e4b8b5c2a0a1-Paper-Conference.pdf.

[14] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. PaLM: Scaling language modeling with Pathways. *Journal of machine learning research*, 24(240):1–113, 2023.

[15] Mouxiang Chen, Binyuan Hui, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Jianling Sun, Junyang Lin, and Zhongxin Liu. Parallel scaling law for language models. *Advances in Neural Information Processing Systems*, 38:118958–118998, 2025.

[16] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, et al. Efficient large-scale language model training

on GPU clusters using Megatron-LM. In *Proceedings of the international conference for high performance computing, networking, storage and analysis*, pages 1–15, 2021.

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

[18] Xingkai Yu. Nano-vLLM. GitHub repository, 2025. URL https://github.com/GeeeekExplorer/nano-vllm.

[19] William Fedus, Barret Zoph, and Noam Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *Journal of Machine Learning Research*, 23(120):1–39, 2022.

[20] Joshua Ainslie, James Lee-Thorp, Michiel De Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. GQA: Training generalized multi-query transformer models from multi-head checkpoints. In *Proceedings of the 2023 conference on empirical methods in natural language processing*, pages 4895–4901, 2023.

[21] Guilherme Penedo, Hynek Kydlíček, Loubna Ben allal, Anton Lozhkov, Margaret Mitchell, Colin Raffel, Leandro Von Werra, and Thomas Wolf. The FineWeb datasets: Decanting the web for the finest text data at scale. *Advances in Neural Information Processing Systems*, 37:30811–30849, 2024.

[22] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. The Llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

[23] The Microsoft AI Team. MAI-Thinking-1: Building a hill-climbing machine. Technical report, Microsoft AI, 2026. URL https://microsoft.ai/pdf/mai-thinking-1.pdf.

[24] Larry Dial. Improving our LLM pretraining efficiency. https://www.openathena.ai/blog/pretraining-speedup/, June 2026. Open Athena Blog.

[25] Venmugil Elango, Nidhi Bhatia, Roger Waleffe, Rasoul Shafipour, Tomer Asida, Abhinav Khattar, Nave Assaf, Maximilian Golub, Joey Guman, Tiyasa Mitra, et al. LatentMoE: Toward optimal accuracy per FLOP and parameter in mixture of experts. *arXiv preprint arXiv:2601.18089*, 2026.

[26] Kimi Team, Tongtong Bai, Yifan Bai, Yiping Bao, M. C., Jianfeng Cai, Xinyuan Cai, Peizhou Cao, Yuxuan Cao, Ziwei Chai, Y Charles, et al. Kimi K3: Open frontier intelligence. *arXiv preprint arXiv:2607.24653*, 2026.

[27] Biao Zhang and Rico Sennrich. Root mean square layer normalization. *Advances in neural information processing systems*, 32, 2019.

[28] Jianlin Su, Murtadha Ahmed, Yu Lu, Shengfeng Pan, Wen Bo, and Yunfeng Liu. RoFormer: Enhanced transformer with rotary position embedding. *Neurocomputing*, 568:127063, 2024.

[29] Zihan Qiu, Zekun Wang, Bo Zheng, Zeyu Huang, Kaiyue Wen, Songlin Yang, Rui Men, Le Yu, Fei Huang, Suozhi Huang, Dayiheng Liu, Jingren Zhou, and Junyang Lin. Gated attention for large language models: Non-linearity, sparsity, and attention-sink-free. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025. URL https://openreview.net/forum?id=1b7whO4SfY.

[30] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. In *International Conference on Learning Representations*, 2019. URL https://openreview.net/forum?id=Bkg6RiCqY7.

[31] Cheng-Ping Hsieh, Simeng Sun, Samuel Kriman, Shantanu Acharya, Dima Rekesh, Fei Jia, and Boris Ginsburg. RULER: What's the real context size of your long-context language models? In *First Conference on Language Modeling*, 2024. URL https://openreview.net/forum?id=kIoBbc76Sy.

[32] John Arevalo, Thamar Solorio, Manuel Montes-y Gómez, and Fabio A González. Gated multimodal units for information fusion. In *International Conference on Learning Representations, Workshop Track*, 2017. URL https://openreview.net/forum?id=S12_nquOe.

[33] Jiquan Ngiam, Aditya Khosla, Mingyu Kim, Juhan Nam, Honglak Lee, and Andrew Y Ng. Multimodal deep learning. In *Proceedings of the 28th International Conference on Machine Learning*, pages 689–696, 2011.

[34] Jin-Hwa Kim, Kyoung-Woon On, Woosang Lim, Jeonghee Kim, Jung-Woo Ha, and Byoung-Tak Zhang. Hadamard

product for low-rank bilinear pooling. In *International Conference on Learning Representations*, 2017. URL https://openreview.net/forum?id=r1rhWnZkg.

[35] Christoph Feichtenhofer, Axel Pinz, and Andrew Zisserman. Convolutional two-stream network fusion for video action recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1933–1941, 2016.

[36] Noam Shazeer. GLU variants improve transformer. *arXiv preprint arXiv:2002.05202*, 2020.

[37] Chong Wang, Nan Du, Tom Gunter, Tao Lei, Kulin Seth, Senyu Tong, Jianyu Wang, Guoli Yin, Xiyou Zhou, Kelvin Zou, et al. Parallel track transformers: Enabling fast GPU inference with reduced synchronization. *arXiv preprint arXiv:2602.07306*, 2026.

[38] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.

[39] Saining Xie, Ross Girshick, Piotr Dollár, Zhuowen Tu, and Kaiming He. Aggregated residual transformations for deep neural networks. In *2017 IEEE conference on computer vision and pattern recognition (CVPR)*, pages 5987–5995. IEEE, 2017.

[40] Gustav Larsson, Michael Maire, and Gregory Shakhnarovich. FractalNet: Ultra-deep neural networks without residuals. In *International Conference on Learning Representations*, 2017. URL https://openreview.net/forum?id=S1VaB4cex.

[41] Ke Sun, Bin Xiao, Dong Liu, and Jingdong Wang. Deep high-resolution representation learning for human pose estimation. In *2019 IEEE/CVF conference on computer vision and pattern recognition (CVPR)*, pages 5686–5696. IEEE, 2019.

[42] Karim Ahmed, Nitish Shirish Keskar, and Richard Socher. Weighted transformer network for machine translation. *arXiv preprint arXiv:1711.02132*, 2017.

[43] Yang Fan, Shufang Xie, Yingce Xia, Lijun Wu, Tao Qin, Xiang-Yang Li, and Tie-Yan Liu. Multi-branch attentive transformer. *arXiv preprint arXiv:2006.10270*, 2020.

[44] Mikhail Burtsev and Anna Rumshisky. Multi-stream transformers. *arXiv preprint arXiv:2107.10342*, 2021.

[45] Boyi Zeng, Yiqin Hao, Zitong Wang, Shixiang Song, He Li, Feichen Song, Yifan Liu, Ziwei He, Xinbing Wang, and Zhouhan Lin. Depth-attention: Cross-layer value mixing for language models. In Hung-Yi Lee, Sunipa Dev, Junyi Jessy Li, and Desmond Elliott, editors, *Proceedings of the 2026 Conference on Empirical Methods in Natural Language Processing*, pages 47493–47514, Budapest, Hungary, October 2026. Association for Computational Linguistics. ISBN 979-8-89176-459-0. doi: 10.18653/2026.emnlp-main.2085. URL https://aclanthology.org/2026.emnlp-main.2085/.

[46] Lianghui Zhu, Yuxin Fang, Bencheng Liao, Shijie Wang, Tianheng Cheng, Zilong Huang, Chen Chen, Lai Wei, Yutao Zeng, Ya Wang, et al. Mixture-of-depths attention. *arXiv preprint arXiv:2603.15619*, 2026.

[47] Noam Shazeer. Fast transformer decoding: One write-head is all you need. *arXiv preprint arXiv:1911.02150*, 2019.

[48] William Brandon, Mayank Mishra, Aniruddha Nrusimha, Rameswar Panda, and Jonathan Ragan-Kelley. Reducing transformer key-value cache size with cross-layer attention. *Advances in Neural Information Processing Systems*, 37: 86927–86957, 2024.

[49] Yutao Sun, Li Dong, Yi Zhu, Shaohan Huang, Wenhui Wang, Shuming Ma, Quanlu Zhang, Jianyong Wang, and Furu Wei. You only cache once: Decoder-decoder architectures for language models. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024. URL https://openreview.net/forum?id=25Ioxw576r.

[50] Will Held. Scaling laws that extrapolate 300× past the fit. https://www.openathena.ai/blog/delphi/, May 2026. Open Athena Blog.

[51] Jianyu Wei, Yizhao Gao, Qihao Zhang, Shimao Chen, Zhengju Tang, Yu Cheng, Shengjie Zhou, Zihan Jiang, Yifan Song, Hailin Zhang, et al. HySparse2: Hybrid sparse attention with two-level KV sharing. *arXiv preprint arXiv:2609.26368*, 2026.

[52] Patrick Esser, Sumith Kulal, Andreas Blattmann, Rahim Entezari, Jonas Müller, Harry Saini, Yam Levi, Dominik Lorenz, Axel Sauer, Frederic Boesel, Dustin Podell, Tim Dockhorn, Zion English, and Robin Rombach. Scaling

rectified flow transformers for high-resolution image synthesis. In *Forty-first International Conference on Machine Learning*, 2024. URL https://openreview.net/forum?id=FPnUhsQJ5B.

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

[54] Zihao Ye, Lequn Chen, Ruihang Lai, Wuwei Lin, Yineng Zhang, Stephanie Wang, Tianqi Chen, Baris Kasikci, Vinod Grover, Arvind Krishnamurthy, et al. FlashInfer: Efficient and customizable attention engine for LLM inference serving. *Proceedings of Machine Learning and Systems*, 7, 2025.

A. Additional Experimental Details

We first give general training and evaluation settings, then expand the corresponding main-text experiments and analyses in the same order.

A.1 Architecture and pretraining setup

General settings. This subsection expands the common training recipe in Section 3.1 and the ladder in Table 1. Table 7 lists the Standard backbone at each scale. Wide–Shallow MBTs retain these block dimensions; Narrow–Deep and Balanced MBTs change the branch widths and depths, as listed with their results. All variants retain the training-token budget of their Standard reference. Here m is the shared MoE latent width and f is each expert's SwiGLU hidden width; neither is the residual width d.

Table 7. Standard backbone and pretraining configuration for W1–W4. MBT variants use the same per-scale data and optimization recipe unless their experiment specifies a change.

Setting	W1	W2	W3	W4
Backbone				
Blocks L	12	14	16	18
Residual width d	1,024	1,152	1,280	1,408
MoE latent width m	512	576	640	704
Expert hidden width f	512	576	640	704
Query / KV heads	16 / 8	18 / 9	20 / 10	22 / 11
Attention head dimension	64	64	64	64
Experts / active experts per token	64 / 8	64 / 8	64 / 8	64 / 8
Total parameters (B)	0.7990	1.1335	1.5549	2.0735
Active-body parameters (M)	139.226	205.441	289.713	394.204
Training schedule				
Training tokens (B)	27.845	41.088	57.942	78.841
Context length	4,096	4,096	4,096	4,096
Global batch (sequences)	128	128	256	256
Optimizer updates T	53,110	78,370	55,258	75,189
Warmup updates	531	783	552	751
Shared settings				
Optimizer	AdamW; $\beta = (0.9, 0.95)$, $\epsilon = 10^{-8}$			
Weight decay / gradient clipping	0.1 / 1.0			
Learning-rate schedule	Linear warmup; cosine decay to $0.1\eta_{\max}$			
Normalization	Pre-RMSNorm and Q/K RMSNorm; $\epsilon = 10^{-5}$			
Position embedding	Full-channel RoPE; base 500,000			
Vocabulary / embedding	128,256; tied input embedding and LM head			
Precision / seed	BF16 / 1234			
Attention / hidden dropout	0 / 0			

Total parameters include the tied embedding/head once; active-body parameters exclude it and count the selected experts. Token budgets are approximately 200× Standard active-body parameters, rounded to a complete optimizer update. Widths and depths of modified branches are specified in their experiment tables.

Attention and experts. Attention uses learned headwise RMSNorm on queries and keys before RoPE, and a channelwise sigmoid gate on the attention output before its output projection [29]. The MoE router reads the normalized d-dimensional residual state. Shared projections map $d \to m$ before expert computation and $m \to d$ after expert combination. Each expert has two $m \to f$ SwiGLU input maps and one $f \to m$ output map. Quantile Balancing [26] updates expert-selection biases once per optimizer step using statistics accumulated over the global batch with 10,000 histogram bins. Combination weights use the original, unbiased sigmoid scores of the selected

experts, renormalized to sum to 2.5. Routing arithmetic is FP32; no tokens are dropped, and there are no shared experts, auxiliary balancing loss, or router z-loss.

Data and tokenization. We use the same pretokenized FineWeb-Edu release across comparisons, with 64 equal-weight training shards in a fixed order and the Llama 3 vocabulary of 128,256 tokens. Documents end with an end-of-document token and are packed into 4,096-token training sequences with next-token labels. Attention and positions are not reset at document boundaries, and end-of-document tokens contribute to the loss. Within each scale, comparisons retain the same data prefix and sample-order construction. Validation uses a separate, fixed prefix of 12,160 sequences (49,807,360 prediction positions), excluded from the training split.

Optimization and initialization. Training uses Megatron-LM [16] with BF16 weights and FP32 optimizer state. AdamW excludes normalization scales and biases from weight decay but applies it to the tied embedding matrix. Base weights use a normal initialization with standard deviation 0.02; normalization gains start at one. In the reference backbone, attention output, expert output, and shared latent up-projection weights use $0.02/\sqrt{2L}$. The input embedding and LM head share weights. We keep the global batch fixed when changing the microbatch size or number of gradient-accumulation steps.

Learning-rate selection. Our primary peak-learning-rate candidates are $\{1, 3, 5, 7, 10, 30\}\times10^{-4}$, as in Section 3.1. We select checkpoints by final validation loss. The selected Standard reference uses 5×10^{-4} at all four scales (larger-scaled model has larger batch sizes). Architecture sweeps use 7×10^{-4} for W1/W2 and 5×10^{-4} for W3/W4. Warmup starts at zero and lasts $\lfloor 0.01T \rfloor$ optimizer updates, followed by cosine decay to $0.1\eta_{\max}$ at the full horizon T. All runs use seed 1234.

A.2 Evaluation and scaling fits

This subsection details the downstream evaluation in Sections 3.2.1 and 3.3 (Figure 6 and Table 12) and the compute-accounting and fitting conventions used throughout Sections 3 and 4.

Downstream evaluation. We use lm-evaluation-harness v0.4.12 with frozen task definitions and dataset revisions. All nine tasks are zero-shot and use the native BF16 model with a 4,096-token context limit. Requests of equal length are batched without padding; batch shapes that fail numerical equivalence checks use single-request evaluation. When a request exceeds the context limit, only the left context is truncated, preserving the scored continuation. The nine tasks are HellaSwag (HS); the Easy and Challenge subsets of the AI2 Reasoning Challenge (ARC-E and ARC-C); OpenBookQA (OBQA); SciQ; WinoGrande (WG); Boolean Questions (BoolQ); LAMBADA (LMDA); and Physical Interaction: Question Answering (PIQA). We report length-normalized accuracy (`acc_norm`) for HS, ARC-E/C, OBQA, SciQ, and PIQA, and accuracy (`acc`) for WG, BoolQ, and LMDA. LMDA uses the `lambada_openai` task. Tables 8 and 12 list the per-scale scores. Figure 6(b) averages each task equally over W1–W4; Average weights the nine tasks equally. All averages are calculated before rounding.

Compute and scaling fits. Body compute excludes the tied LM head and includes attention, MoE, width bridges, and fusion projections. We count two FLOPs per multiply–add and approximate training matrix work by three times the forward work, including causal attention at sequence length 4,096. Elementwise operations, activation recomputation, optimizer updates, communication, and evaluation are excluded. One PFLOP-day equals 8.64×10^{19} FLOPs. For each model family, we fit $\mathcal{L}(C) = AC^{-\alpha}$ by unweighted least squares in log compute and log loss using the four final W1–W4 checkpoints. Curves are drawn over their measured compute ranges. The 1.23× comparison in Figure 6(a) is obtained by inverting the two fits at loss 2.500, within their common measured range.

Radar axes. Each spoke in Figure 6(b) spans 8 percentage points, with rings spaced 2 points apart. Clockwise from HS, the lower bounds are 47, 58, 30, 33, 76, 51, 52, 36, and 67 for HS, ARC-E, ARC-C, OBQA, SciQ, WG, BoolQ, LMDA, and PIQA, respectively. Point labels show the original percentage scores.

A.3 Two-branch downstream results

For Section 3.2.1, Table 8 reports the per-scale results underlying Figure 6(b), using the selected allocations in Table 1.

Table 8. Per-scale zero-shot results for the Standard and selected 2-branch MBT checkpoints in Figure 6. Scores are percentages (↑); the higher score in each pair is bold.

	W1		W2		W3		W4	
Task	Standard	MBT	Standard	MBT	Standard	MBT	Standard	MBT
HellaSwag	45.15	**46.44**	50.29	**52.15**	53.30	**54.65**	57.34	**58.63**
ARC-E	57.37	**60.82**	62.46	**63.72**	63.47	**67.17**	65.66	**67.63**
ARC-C	31.06	**31.40**	33.45	**35.15**	35.58	**38.91**	37.46	**38.40**
OBQA	34.40	**36.20**	35.20	**39.20**	37.60	**39.20**	40.60	**41.80**
SciQ	75.20	**77.00**	80.50	**82.50**	84.50	**84.70**	81.10	**86.10**
WG	**54.85**	54.38	53.28	**55.33**	**57.85**	57.62	56.43	**58.01**
BoolQ	56.36	**58.65**	47.22	**58.07**	57.55	**58.84**	58.78	**60.95**
LMDA	**37.42**	37.05	38.95	**39.47**	40.56	**41.84**	44.52	**46.07**
PIQA	**69.80**	69.53	71.00	**71.65**	**72.36**	72.25	73.61	**75.46**
Average	51.29	**52.39**	52.48	**55.25**	55.86	**57.24**	57.28	**59.23**

We report normalized accuracy for HellaSwag, ARC-E/C, OBQA, SciQ, and PIQA; accuracy for WG, BoolQ, and LMDA (LAMBADA). Average weights the nine tasks equally. All metrics use lm-eval 0.4.12.

A.4 Multi-branch configurations

For Section 3.2.2 and Figure 7, Table 9 expands the W3 example in Table 2 to all four scales. Shared-layer dimensions and training tokens follow Table 7.

A.4.1 Preserving depth with narrow branches

To complement the branch-count study in Section 3.2.2, Table 10 holds W1 depth at 12 with 6 blocks per branch and 3 shared blocks on each side. Increasing branch count requires narrower blocks at fixed compute and raises loss. Balanced MBTs instead trade some branch depth for greater width.

A.5 KV-cache sharing configurations and cache accounting

For Section 3.3, Table 11 expands the W4 example in Table 3 to all scales, combining configurations, final losses, cache savings, and training compute. Within each configuration, only KV construction changes.

Cache capacity. Let Standard have L_0 blocks of width d_o. An MBT has P `prelude` blocks, C `coda` blocks, and K branches of q blocks at width d_b. With our 2:1 GQA ratio, BF16 keys and values occupy $2d$ bytes per token per layer. Full-model capacities per token are

$$
\begin{aligned}
M_{\text{Standard}} &= 2L_0 d_o, \\
M_{\text{Independent}} &= 2\big[(P+C)d_o + Kqd_b\big], \\
M_{\text{Shared}} &= 2\big[(P+C)d_o + qd_b\big].
\end{aligned}
$$

Savings are $1 - M/M_{\text{Standard}}$ at fixed batch size, context length, and precision. Standard occupies 24, 31.5, 40, and 49.5 KiB/token for W1–W4. These capacities count stored keys and values, excluding runtime overhead and cross-device replication.

Table 9. Branch-count configurations for Figure 7. L/D denotes physical block count / depth; $d/m/f$ gives branch residual / MoE latent / expert FFN widths. Compute is body / total PFLOP-days.

Scale	MBT family	K	$P/(\text{branches})/C$	$d/m/f$	L/D	Body / Total	Loss ↓
W1	Wide–Shallow	2	$3/(3+3)/3$	1024/512/512	12/9	0.3706 / 0.6245	2.630
W1	Wide–Shallow	3	$3/(2+2+2)/3$	1024/512/512	12/8	0.3726 / 0.6266	2.633
W1	Wide–Shallow	4	$3/(2+2+1+1)/3$	1024/512/512	12/8	0.3746 / 0.6286	2.628
W1	Balanced	2	$3/(5+5)/3$	768/368/384	16/11	0.3717 / 0.6257	2.618
W1	Balanced	3	$3/(4+4+4)/3$	640/368/368	18/10	0.3745 / 0.6284	2.623
W1	Balanced	4	$3/(3+3+3+3)/3$	640/368/368	18/9	0.3758 / 0.6297	2.626
W2	Wide–Shallow	2	$3/(4+3)/4$	1152/576/576	14/11	0.7822 / 1.2038	2.529
W2	Wide–Shallow	3	$3/(2+3+2)/4$	1152/576/576	14/10	0.7860 / 1.2076	2.535
W2	Wide–Shallow	4	$3/(2+2+2+1)/4$	1152/576/576	14/9	0.7898 / 1.2114	2.533
W2	Balanced	2	$3/(6+6)/3$	896/464/464	18/12	0.7838 / 1.2054	2.525
W2	Balanced	3	$3/(5+5+5)/4$	768/352/352	22/12	0.7856 / 1.2072	2.529
W2	Balanced	4	$3/(4+4+4+4)/4$	640/384/384	23/11	0.7782 / 1.1998	2.529
W3	Wide–Shallow	2	$4/(4+4)/4$	1280/640/640	16/12	1.5164 / 2.1769	2.445
W3	Wide–Shallow	3	$4/(3+3+2)/4$	1280/640/640	16/11	1.5230 / 2.1835	2.457
W3	Wide–Shallow	4	$4/(2+2+2+2)/4$	1280/640/640	16/10	1.5296 / 2.1901	2.451
W3	Balanced	2	$4/(6+6)/4$	1024/512/512	20/14	1.5301 / 2.1907	2.436
W3	Balanced	3	$4/(5+5+5)/4$	896/432/448	23/13	1.5160 / 2.1765	2.446
W3	Balanced	4	$4/(4+4+4+4)/4$	896/400/400	24/12	1.5148 / 2.1754	2.449
W4	Wide–Shallow	2	$4/(5+4)/5$	1408/704/704	18/14	2.7482 / 3.7369	2.373
W4	Wide–Shallow	3	$4/(3+3+3)/5$	1408/704/704	18/12	2.7591 / 3.7478	2.373
W4	Wide–Shallow	4	$4/(3+2+2+2)/5$	1408/704/704	18/12	2.7699 / 3.7586	2.373
W4	Balanced	2	$4/(7+7)/4$	1152/592/592	22/15	2.7620 / 3.7507	2.363
W4	Balanced	3	$4/(6+6+6)/5$	896/496/496	27/15	2.7443 / 3.7330	2.366
W4	Balanced	4	$4/(6+6+6+6)/5$	768/432/432	33/15	2.7985 / 3.7872	2.367

Table 10. W1 full-depth Narrow–Deep controls, reporting the best final loss for each configuration. Compute is body / total PFLOP-days; L/D denotes physical block count / depth.

Model / allocation	Branch $d/m/f$	L/D	Loss ↓	Body / Total
Standard (12 blocks)	1024/512/512	12/12	2.653	0.366514 / 0.620471
Wide–Shallow $3/(3+3)/3$	1024/512/512	12/9	2.630	0.370569 / 0.624527
Narrow–Deep $3/(6+6)/3$	640/352/368	18/12	2.632	0.369463 / 0.623421
Narrow–Deep $3/(6+6+6)/3$	512/272/272	24/12	2.650	0.369458 / 0.623416
Narrow–Deep $3/(6+6+6+6)/3$	384/256/256	30/12	2.675	0.370701 / 0.624659

Training compute. Joint-KV preserves matrix FLOPs: projecting the concatenated K branch states once costs the same as K independent projections, and each branch still performs attention. Body and total costs in Table 11 follow Appendix A.2; total includes the tied LM head.

Downstream evaluation. For Section 3.3, Table 12 reports the nine zero-shot tasks for the final 2-, 3-, and 4-branch Balanced Joint-KV MBTs in Table 11. We use the same Standard checkpoints and evaluation protocol as Table 8 and Appendix A.2.

A.6 Width and depth controls

For Section 4.1 and Figure 10, Table 13 expands the W1 example in Table 4 to all four scales. The Standard and two-branch Wide–Shallow references are fixed while the three control families change the allocation of width and depth.

Table 11. Balanced MBTs with Independent-KV and Joint-KV. Losses are final validation losses; Joint-KV cache savings are relative to Standard. Compute is body / total PFLOP-days for both variants.

	Architecture		Loss ↓		Cache and compute	
K	Allocation	Branch $d/m/f$	Independent	Joint	Joint KV saved	Body / Total
W1: Standard loss 2.653; Standard KV 24 KiB/token						
2	$3/(5+5)/3$	768/368/384	2.618	2.620	18.75%	0.371744 / 0.625702
3	$3/(4+4+4)/3$	640/368/368	2.623	2.622	29.17%	0.374484 / 0.628442
4	$3/(3+3+3+3)/3$	640/368/368	2.626	2.633	34.38%	0.375752 / 0.629709
W2: Standard loss 2.554; Standard KV 31.5 KiB/token						
2	$3/(6+6)/3$	896/464/464	2.525	2.528	23.81%	0.783830 / 1.205417
3	$3/(5+5+5)/4$	768/352/352	2.529	2.537	26.19%	0.785588 / 1.207175
4	$3/(4+4+4+4)/4$	640/384/384	2.529	2.534	34.13%	0.778221 / 1.199809
W3: Standard loss 2.471; Standard KV 40 KiB/token						
2	$4/(6+6)/4$	1024/512/512	2.436	2.434	20.00%	1.530086 / 2.190659
3	$4/(5+5+5)/4$	896/432/448	2.446	2.451	28.13%	1.515967 / 2.176539
4	$4/(4+4+4+4)/4$	896/400/400	2.449	2.454	32.50%	1.514800 / 2.175372
W4: Standard loss 2.392; Standard KV 49.5 KiB/token						
2	$4/(7+7)/4$	1152/592/592	2.363	2.361	23.74%	2.762023 / 3.750740
3	$4/(6+6+6)/5$	896/496/496	2.366	2.366	28.79%	2.744269 / 3.732987
4	$4/(6+6+6+6)/5$	768/432/432	2.367	2.364	31.82%	2.798531 / 3.787248

Table 12. Zero-shot downstream performance of Balanced Joint-KV MBTs and Standard. Scores are percentages (↑); the best displayed score within each group is bold.

Scale	Model	HS	ARC-E	ARC-C	OBQA	SciQ	WG	BoolQ	LMDA	PIQA	Average
W1	Standard	45.15	57.37	31.06	34.40	75.20	**54.85**	56.36	37.42	**69.80**	51.29
	Joint-KV, $K=2$	**47.65**	56.40	**31.66**	**36.80**	**79.60**	54.30	**60.00**	38.29	69.70	**52.71**
	Joint-KV, $K=3$	47.33	57.74	31.06	34.20	76.90	53.75	56.64	**38.91**	69.53	51.78
	Joint-KV, $K=4$	47.07	**58.63**	31.31	35.00	75.80	53.20	58.17	36.29	68.55	51.56
W2	Standard	50.29	62.46	33.45	35.20	80.50	53.28	47.22	38.95	71.00	52.48
	Joint-KV, $K=2$	**52.29**	61.11	**35.67**	37.00	**82.70**	53.51	58.04	40.03	70.67	54.56
	Joint-KV, $K=3$	51.71	62.12	34.98	35.00	81.00	**56.99**	**61.28**	39.49	71.27	54.87
	Joint-KV, $K=4$	51.79	**64.65**	34.64	**38.00**	81.80	53.28	60.76	**40.36**	**71.71**	**55.22**
W3	Standard	53.30	63.47	35.58	37.60	84.50	**57.85**	57.55	40.56	72.36	55.86
	Joint-KV, $K=2$	**55.65**	65.24	36.35	38.00	**85.00**	57.54	56.48	**44.15**	72.14	**56.73**
	Joint-KV, $K=3$	54.76	**65.78**	**37.20**	38.80	83.40	54.93	58.99	43.20	71.82	56.54
	Joint-KV, $K=4$	54.54	63.97	36.09	**39.40**	82.20	55.72	**61.77**	42.11	**72.69**	56.50
W4	Standard	57.34	65.66	37.46	40.60	81.10	56.43	58.78	44.52	73.61	57.28
	Joint-KV, $K=2$	**59.17**	**68.86**	**40.61**	39.00	**86.80**	57.85	61.93	48.13	**74.86**	**59.69**
	Joint-KV, $K=3$	58.54	66.54	38.14	**41.00**	85.70	**58.09**	**62.17**	46.24	74.81	59.03
	Joint-KV, $K=4$	59.04	66.75	38.65	40.20	84.90	57.85	56.82	**48.52**	74.65	58.60
Mean	Standard	51.52	62.24	34.39	36.95	80.33	55.60	54.98	40.36	71.69	54.23
	Joint-KV, $K=2$	**53.69**	62.90	**36.07**	37.70	**83.53**	55.80	59.11	**42.65**	71.84	**55.92**
	Joint-KV, $K=3$	53.09	63.05	35.35	37.25	81.75	**55.94**	**59.77**	41.96	71.86	55.56
	Joint-KV, $K=4$	53.11	**63.50**	35.17	**38.15**	81.17	55.01	59.38	41.82	**71.90**	55.47

K: number of branches. Average weights the nine tasks equally; Mean averages each task over W1–W4. Task definitions and metrics follow Appendix A.2; averages use unrounded scores.

A.7 Cache-matched Joint-KV and Independent-KV comparison

For Section 4.2, Table 14 provides widths and body compute for the paired comparisons in Table 5. Within each pair, both models retain the same allocation, widths, query heads, outer blocks, and fusion. Joint-KV constructs 2

Table 13. All five families in Figure 10. Widths are residual / MoE latent / expert FFN ($d/m/f$). Shared widths follow Table 7; all configurations retain the per-scale training-token budget.

Family	Scale	Allocation	Physical blocks / depth	Shared or all-layer $d/m/f$	Core or branch $d/m/f$
Standard	W1	12	12 / 12	1024/512/512	—
	W2	14	14 / 14	1152/576/576	—
	W3	16	16 / 16	1280/640/640	—
	W4	18	18 / 18	1408/704/704	—
Global wide–shallow	W1	9	9 / 9	1280/640/512	—
(sequential)	W2	10	10 / 10	1408/704/704	—
	W3	12	12 / 12	1536/768/720	—
	W4	13	13 / 13	1792/896/704	—
Local wide-core	W1	3/3/3	9 / 9	1024/512/512	1664/640/640
(sequential)	W2	3/4/3	10 / 10	1152/576/576	1920/704/704
	W3	4/4/4	12 / 12	1280/640/640	2176/768/768
	W4	4/5/4	13 / 13	1408/704/704	2304/880/944
Wide–Shallow MBT	W1	$3/(3+3)/3$	12 / 9	1024/512/512	1024/512/512
(two branches)	W2	$3/(4+4)/3$	14 / 10	1152/576/576	1152/576/576
	W3	$4/(4+4)/4$	16 / 12	1280/640/640	1280/640/640
	W4	$4/(5+5)/4$	18 / 13	1408/704/704	1408/704/704
Narrow–Deep MBT	W1	$3/(6+6)/3$	18 / 12	1024/512/512	640/320/416
(two branches)	W2	$3/(8+8)/3$	22 / 14	1152/576/576	768/384/384
	W3	$4/(8+8)/4$	24 / 16	1280/640/640	896/400/416
	W4	$4/(10+10)/4$	28 / 18	1408/704/704	896/448/560

shared KV heads at each branch depth; Independent-KV constructs 1 private head per branch. All heads have dimension 64, and caches use BF16. Outer widths follow Table 7; query and KV head counts are listed in Table 14.

Table 14. W1–W4 configurations for Section 4.2. Cache size is identical within each pair. Body compute (PFLOP-days) and loss report Joint-KV / Independent-KV at the same training-token budget.

			Q/KV heads					
Scale	Allocation	Branch $d/m/f$	Outer	Joint-KV	Independent-KV	KV/token (KiB)	Body compute	Loss ↓
W1	$3/(5+5)/3$	768/368/384	16/2	16/2	16/1	5.5	0.3867/0.3848	2.646/2.657
W2	$3/(6+6)/3$	896/464/464	18/2	16/2	16/1	6.0	0.7763/0.7723	2.557/2.568
W3	$4/(6+6)/4$	1024/512/512	20/2	16/2	16/1	7.0	1.4499/1.4436	2.487/2.499
W4	$4/(7+7)/4$	1152/592/592	24/3	18/2	18/1	9.5	2.6664/2.6551	2.380/2.392

Both variants are trained from scratch on the same ordered data and token budget at each scale. Both follow the protocol in Appendix A.1, including the LR candidates and batch size. All reported losses are measured at the end of this token budget. Joint-KV's larger KV projection adds less than 2.1M parameters and 0.6% body compute; Table 14 records each model's body compute.

A.8 Fusion implementation

This subsection gives the implementation details for the fusion ablation in Section 4.3 and Table 6.

Parameterization. $h_1, h_2 \in \mathbb{R}^d$ are normalized branch outputs, with $d = 1280$ and auxiliary width $r = 128$. The weights have shapes $W_i \in \mathbb{R}^{d\times d}$, $A_i, B_i \in \mathbb{R}^{r\times d}$, and $P \in \mathbb{R}^{d\times r}$. Both gates take $x = [W_1 h_1; W_2 h_2]$:

$$g = \sigma(G_o\,\mathrm{SiLU}(G_i x))\,, \qquad \beta = \mathrm{softmax}(G_o\,\mathrm{SiLU}(G_i x))\,.$$

Here $G_i \in \mathbb{R}^{r\times 2d}$; G_o has d rows for channel competition and 2 rows for token softmax. Softmax operates over branches at each token, without uniform-weight smoothing. All projections are bias-free.

Initialization and computation. The learned branch RMSNorms start with unit gains and use $\epsilon = 10^{-5}$. The concat projection uses normal standard deviation $0.02/\sqrt{32}$; auxiliary inner matrices use 0.02. Zero-initialized P and G_o make corrections vanish and gates equal, recovering concat for the learned correction and gating variants. We retain the fused concat projection and add corrections to preserve the identical initial BF16 output. Gates and correction accumulation use FP32; inner projections use BF16. Auxiliary work adds 0.96–1.09% body compute. Other settings follow the W3 training recipe.

Projection-free control. The normalized sum uses $z = (h_1 + h_2)/\sqrt{2}$ without a fusion matrix. It retains the two learned RMSNorm gains ($2d$ = 2,560 parameters), so only the aggregation, not the whole module, is parameter-free. Removing the concat projection saves $2d^2$ = 3,276,800 parameters.

B. Inference

This appendix describes the inference engine of Section 2.3, the workloads and measurement protocol of Section 3.3, the decode speedups of Joint-KV MBTs at all four scales, and an ablation of the engine mechanisms.

B.1 Engine implementation

Shared infrastructure. The engine extends nano-vLLM [18]. Standard and MBTs share the same infrastructure: a paged KV cache [17], FlashAttention-3 [53] for attention, and FlashInfer [54] for sampling. Each decode step is captured once as a CUDA graph and replayed.

Branch grouping. At each depth of the branch region, the engine stacks the weights of the K branches into one tensor. It also stacks their hidden states into a $K \times T \times d_b$ tensor, where T is the number of tokens in the step and d_b is the branch width. Each operator of a block then runs once for all branches, including a single attention call (see head folding below). In the MoE layer, the expert indices selected by branch i are offset into a concatenated expert table of all branches. A single MoE kernel call thus serves the tokens of every branch.

Head folding. All models use two query heads per KV head. Head folding reshapes the queries of the K branches from $[KT, 2h_{\mathrm{KV}}, d_h]$ to $[T, 2Kh_{\mathrm{KV}}, d_h]$. The $2K$ query heads that share one KV head then form one group, so one attention call reads the shared bank once for all branches.

Figure 12 shows what each mechanism changes for the 4-branch W3 Balanced MBT. Branch grouping cuts the serial steps per token from 24 to 12, against 16 for Standard. Joint-KV stores 0.675× the KV cache of Standard, against 1.2× for Independent-KV. Head folding cuts the KV read per decode step from 1.2× to 0.675× of Standard.

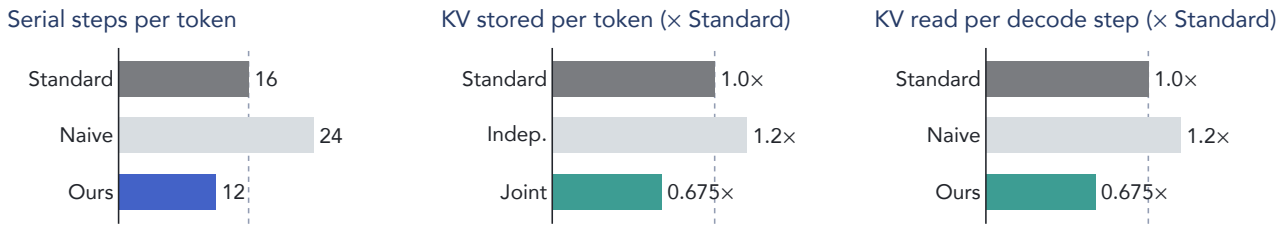

Figure 12. Quantities changed by the serving mechanisms of Figure 4 for the 4-branch W3 Balanced MBT, relative to Standard. All values are structural counts, not measurements. Left: serial block steps per token when the branches run one after another (Naive) and with branch grouping (Ours). Middle: KV cache stored per token. Right: KV read per decode step with Joint-KV, when each branch reads the shared bank separately (Naive) and with head folding (Ours).

Correctness. Before every timed run, we compare branch-grouped execution with head folding against sequential

execution, which runs the branch blocks one at a time with one attention call per branch. On two prompts, the generated tokens, the logits, and the selected experts are bitwise identical in every reported setting. For W1–W4, all hidden states and KV caches are identical as well.

B.2 Benchmark workloads and protocol

Prompts. All prompts are token sequences under the Llama 3 tokenizer [22] used in training. They are stored in fixed prompt files, so every model receives identical token ids. The 4K-token prompts are the first 256 consecutive 4096-token windows of the tokenized validation split of FineWeb-Edu [21], the corpus used in training. The prompts of 16K, 64K, and 128K tokens come from the official RULER generator [31]. We use seven of its tasks: five needle-in-a-haystack tasks (`niah_single_2`, `niah_single_3`, `niah_multikey_1`, `niah_multivalue`, and `niah_multiquery`) and two question-answering tasks (`qa_1` and `qa_2`). Each task has 40 samples per length for 16K and 64K and 20 for 128K, *i.e.*, 280 and 140 prompts per length. Each sample is truncated from the left to exactly the target length, which keeps the question at the end.

Requests. Requests run in rounds of N. In each round, N requests start together. Each request is prefilled and then decodes exactly 1024 tokens, because an end-of-sequence token does not stop generation. The next round starts when all N requests have finished. The rounds cycle through the first $4N$ prompts of the prompt file, with N prompts per round. Sampling uses no top-k truncation and a fixed seed.

Measurement. Timing starts after the model is loaded, the CUDA graphs are captured, and two warm-up rounds have run. Throughput is then measured in three windows, each lasting at least 60 s and ending at the next round boundary. CUDA events split each round into prefill, which includes the first generated token, and decode. Decode throughput is the number of tokens generated in the decode phase over all N requests, divided by the decode time. We report its mean over the three windows. Over the three windows, the decode throughput of every Joint-KV run in Figure 13 varies by at most 0.33% (coefficient of variation). For 4K-, 16K-, and 64K-token inputs, all models of a scale run on one GPU in the order Standard, Joint-KV with K=4, 3, and 2, and Standard again. For 128K-token inputs, each Joint-KV model runs on its own GPU between two Standard runs.

B.3 Decode speedup at all scales

Figure 13 and Table 16 report the decode speedup of the Joint-KV MBTs of Table 11 at all four scales. Over the 48 settings of each model, four branches reach 1.08× to 1.60× and three branches 1.03× to 1.48×. Two branches are slower than Standard in seven settings: six at W1 (0.93× to 0.99×) and one at W3 (0.99×, 4K input, N=1).

Table 15. Depth, *i.e.*, serial blocks per decode step, of Standard and of the Joint-KV MBTs in Figure 13 (Eq. 5).

Scale	Standard	K=4	K=3	K=2
W1	12	9	10	11
W2	14	11	12	12
W3	16	12	13	14
W4	18	15	15	15

Per-kernel profiles of decode steps on one H200, with the same engine, explain these results. The profiled speedups agree with Table 16 within 2.5%.

Short inputs at low concurrency. With a single 4K-token request, every MBT reaches 0.84 to 0.91 of the depth ratio (Table 15). Most of the step scales with depth: excluding the operations that only MBTs perform, a four-branch MBT takes the depth-scaled step time of Standard within 4%. The rest does not shrink with depth. The embedding, LM head, and sampling take 10 to 13% of a Standard step. The MBT-specific operations, namely the branch input map, the fusion, the bank construction, and head folding, add 100 to 160 μs per step with four

branches and 130 to 190 μs with two. Branch counts with equal depth, such as three and four branches at W4, reach similar speedups. At W1, two branches remove only one block. This saves less than the MBT-specific operations cost, so they are slower than Standard.

Short inputs at higher concurrency. Each layer routes a token to 8 of its 64 experts. More requests therefore make a step read more distinct experts, which accounts for most of the time added from N=1 to N=8 at 4K. This cost scales with the total expert parameters, which Balanced MBTs keep close to Standard. With four branches, the ratio is 1.02 at W1, 0.89 at W3, and 1.00 at W4. The measured growth of expert time matches these ratios (1.05, 0.89, and 1.04). The speedup thus dips at W1 and W4. At N=64, nearly all experts are read, KV reads grow, and the speedup rises again.

Long inputs at high concurrency. KV reads dominate these steps, so the speedup approaches the bound $1/(1-s)$. For the four-branch W3 MBT, this bound is 1.48. With 32 concurrent 64K-token requests, the MBT spends 0.668× the attention time of Standard and reaches 1.46×. With 16 concurrent 128K-token requests, each step reads the same number of KV tokens as with 32 concurrent 64K-token requests. The MBT then reaches 1.58×, which exceeds the bound. This happens because attention in the full-width layers runs slower at 128K and takes 27% longer than at 64K. Attention over the shared banks does not slow down. All of the KV cache of Standard sits in full-width layers, but only 74% of that of the MBT does. The MBT therefore reads KV at a higher average bandwidth (3.05 against 2.81 TB/s), so its time saving exceeds its byte saving. This comparison is fair. Both models use the same engine, GPU, paged KV cache, and attention kernel and settings. The full-width layers of the MBT slow down as much as those of Standard. The extra gain therefore comes from head folding.

Table 16. Decode speedup of Joint-KV MBTs over Standard at all scales on one H200 GPU. The speedup is the decode throughput of an MBT divided by that of Standard at the same input length and concurrency N. Drift (%) is the change of the Standard throughput between the runs before and after the Joint-KV models. At 128K, each model has its own pair of Standard runs, and the table lists the largest of the three drifts. For each scale and row, the fastest model is in **bold** and the second fastest is underlined.

		W1				W2				W3				W4			
Input	N	K=4	K=3	K=2	Drift	K=4	K=3	K=2	Drift	K=4	K=3	K=2	Drift	K=4	K=3	K=2	Drift
4K	1	**1.12**	1.04	0.96	−4.2	**1.12**	1.03	1.02	−3.0	**1.15**	1.07	0.99	+0.5	**1.09**	1.08	1.06	−1.8
	8	**1.10**	1.03	0.93	+6.2	**1.18**	1.09	1.11	+2.0	**1.21**	1.15	1.11	+1.6	1.08	**1.10**	1.03	+3.3
	64	**1.23**	1.15	1.08	+0.3	**1.26**	1.19	1.13	+0.7	**1.24**	1.17	1.12	−0.1	**1.26**	1.23	1.19	−0.2
16K	1	**1.15**	1.06	0.98	+3.1	**1.15**	1.05	1.05	+0.9	**1.17**	1.10	1.01	+2.8	**1.10**	1.09	1.06	+2.3
	8	**1.21**	1.13	0.95	+0.6	**1.33**	1.18	1.20	+1.1	**1.30**	1.23	1.22	−0.5	1.17	**1.18**	1.08	+0.3
	64	**1.40**	1.30	1.19	−0.2	**1.42**	1.32	1.24	−0.1	**1.38**	1.30	1.21	−0.3	**1.37**	1.30	1.22	−0.6
64K	1	**1.21**	1.11	1.01	+2.4	**1.20**	1.10	1.09	+0.7	**1.23**	1.15	1.06	+1.7	**1.16**	1.14	1.11	+2.0
	8	**1.37**	1.27	0.98	−0.5	**1.51**	1.27	1.30	+0.3	**1.41**	1.33	1.35	−0.2	1.28	**1.29**	1.14	−0.6
	32	**1.40**	1.27	1.15	+0.7	**1.45**	1.35	1.35	+0.2	**1.46**	1.39	1.28	+0.6	**1.36**	1.35	1.21	+1.0
128K	1	**1.27**	1.17	1.07	+4.0	**1.28**	1.17	1.16	+1.1	**1.32**	1.24	1.13	−0.6	**1.25**	1.23	1.19	−0.9
	8	**1.45**	1.34	0.99	+0.2	**1.60**	1.31	1.36	+0.7	**1.46**	1.37	1.40	−0.5	1.32	**1.33**	1.16	−0.8
	16	**1.42**	1.28	1.16	+0.4	**1.52**	1.45	**1.52**	−0.2	**1.58**	1.48	1.40	−0.5	**1.46**	1.43	1.19	−0.2

B.4 Engine ablation

Setup. We measure how much each engine mechanism of Section 2.3 contributes to the decode speedup. We fix the 4-branch W3 Joint-KV MBT and compare three engines on one GPU, with 4K-, 16K-, and 64K-token inputs. The naive engine runs the branch blocks one at a time, each with its own attention call. The second engine adds branch grouping. The third engine also adds head folding and is the full engine.

Results. Figure 14 shows the results. The naive engine is slower than Standard in every setting, because it runs more serial blocks and reads the shared KV cache once per branch. Branch grouping shortens the serial path, so it helps most with short inputs and low concurrency. Head folding removes the repeated KV reads, so its gain grows

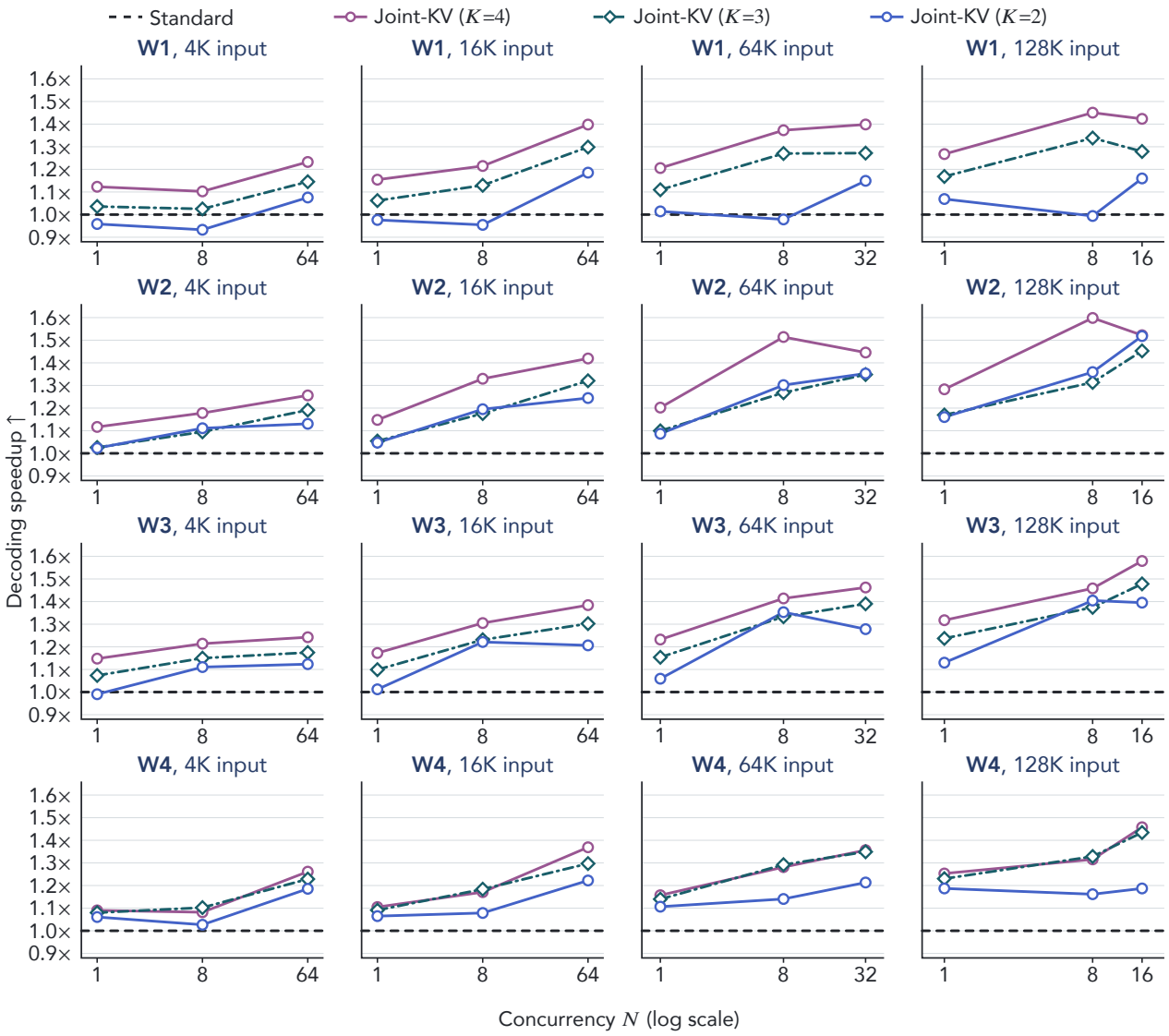

Figure 13. Decode speedup of Joint-KV MBTs over Standard at all scales. Decode throughput relative to Standard on one H200 GPU, as a function of the number of concurrent requests N, for 4K-, 16K-, 64K-, and 128K-token inputs that each generate 1K tokens. Rows are scales W1–W4, and all panels share the vertical axis.

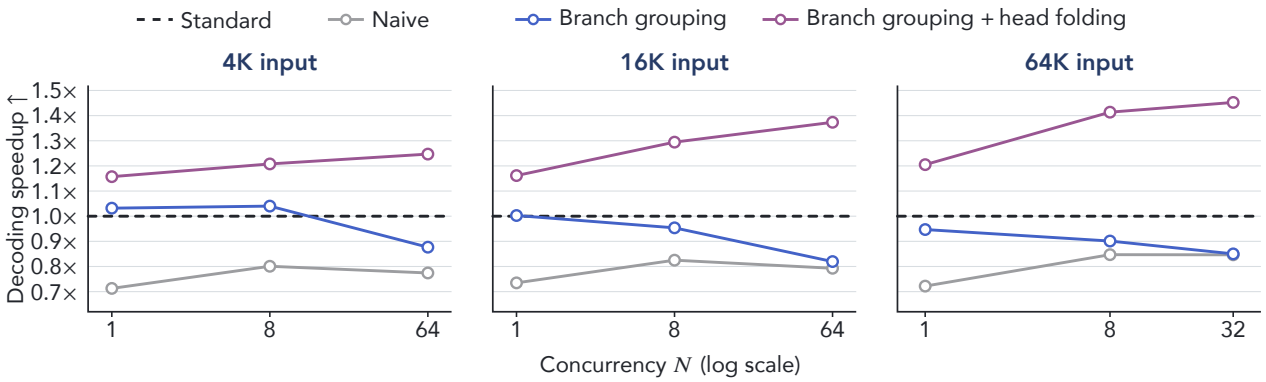

Figure 14. Contribution of each engine mechanism. Decode speedup over Standard of the 4-branch W3 Joint-KV MBT under three engines on one H200 GPU, for N concurrent requests that each generate 1K tokens.

with the input length and concurrency. Only the two mechanisms together make the MBT faster than Standard in every setting. Each mechanism reduces one of the two costs in Eq. 7.