RANDOMLLM: HOW LITTLE RANDOMNESS DOES A STRONG LOTTERY TICKET NEED?

Anonymous authors
Paper under double-blind review

ABSTRACT

Strong-ticket training freezes a network's weights at random values and learns only a mask over them. The standard construction for ticket models samples independent random weights for every projection, using as many random scalars as there are connections. We propose RandomLLM, which instead slices every random weight from a shared random pool generated by two frozen rank-r random sources and keeps a separate multivalued mask for every projection. We find that the randomness of the random weight has little influence on the strong ticket that mask learning produces: on a 3B model, random rank 1 cuts the random budget by a factor of about 10^5 relative to per-projection independent weights while matching their performance (16.00 against 15.95 perplexity, and 38.12 against 37.21 mean zero-shot accuracy), and a controlled three-seed experiment finds no resolvable effect from either sharing the pool or constraining its rank. Rather than accelerating this paradigm with dedicated hardware, we provide a general-purpose GPU implementation that avoids rebuilding the mask at every forward pass. Across four model scales, Mask packing accelerates prefill and decode by $1.42\times$–$6.25\times$ and $1.38\times$–$6.16\times$, respectively, while Threshold caching accelerates training by $1.18\times$–$1.59\times$. The source code is available at https://anonymous.4open.science/status/randomllm-8586.

1 INTRODUCTION

Learning a language model is usually identified with learning its weight matrices. Strong-ticket training offers another construction: hold the random weights fixed and learn the connections through which they act (Zhou et al., 2019; Ramanujan et al., 2020; Malach et al., 2020). The trained mask decides which connections act and how strongly, while the initialized random sources are never overwritten during training: they define the coefficients available to the mask, determine what must be stored for deployment, and shape how the model can be executed efficiently in hardware.

The standard construction for language models samples an independent random weight for every projection (Okoshi et al., 2025), so a decoder whose projections contain P connections rests on P independent random numbers. That source is far too large to keep resident, so prior work generates it on the fly, and the accelerators built around this paradigm generate it on chip, in dedicated hardware rather than on a general-purpose processor (Hirose et al., 2022; Yan et al., 2025). The random source is thereby treated as a cost to be engineered around, not as a design choice to be questioned.

We therefore ask: *how much randomness does a strong ticket actually need?* The budget has two knobs: the rank of each random weight and whether projections share one source. If both can be turned down far enough, the source becomes small enough to keep resident, the generation machinery disappears, and the paradigm runs on commodity hardware. Existing theory bounds this budget only from the side of sufficiency: logarithmic over-parameterization of an independent and identically distributed (i.i.d.) source suffices for a strong ticket to exist and is essentially optimal (Malach et al., 2020; Pensia et al., 2020), and structured sources have begun to be analyzed (da Cunha et al., 2023). These results concern the existence of worst-case targets and do not say how small a source can become before mask learning stops finding useful solutions.

We propose RandomLLM, which compresses the random source along both axes of this question at once (Figure 1): two frozen Gaussian factors define one shared pool $R = AB/\sqrt{r}$ of random rank r, every projection slices its random weight out of that pool at fixed offsets and learns its own

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

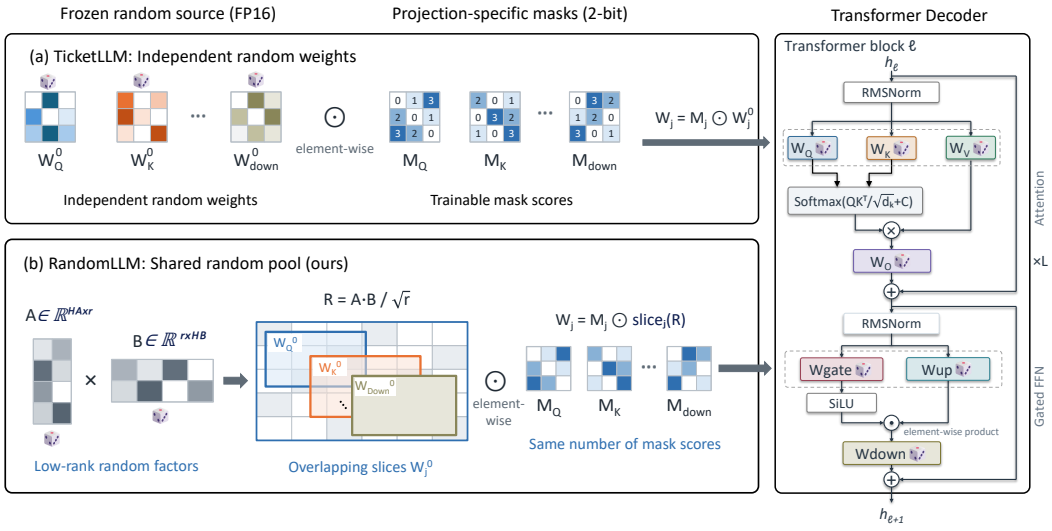


Figure 1: Random-weight construction in TicketLLM and RandomLLM. TicketLLM assigns an independently sampled full-rank random weight to each projection, whereas RandomLLM forms projection-specific slices from one shared random pool $R = AB/\sqrt{r}$, defined by two low-rank Gaussian factors. Within a fixed backbone, both constructions learn the same number of projection-specific mask scores and deploy the same number of two-bit mask values. RandomLLM therefore changes the random-source cost while leaving the mask cost unchanged.

multivalued mask. The number of trainable mask scores therefore stays at P for any r. Element-wise masking can break the rank bound of the random weight. If the mask were allowed to take arbitrary real values, then for any entrywise-nonzero random weight and any target Z, the mask $M = Z \oslash W^0$ would reproduce Z exactly. Our multivalued mask is restricted to $\{0, 1, 2, 3\}$, so this observation removes the rank obstruction but does not imply unrestricted expressivity. At $r = 1$, the projection can also be evaluated without materializing the random weight, since $[M \odot (ab^\top)]x = a \odot (M(b \odot x))$ requires only two element-wise scaling operations in addition to the multiplication by M.

We evaluate the construction on FineWebEdu and ten zero-shot tasks with backbones from 0.110B to 3B, and the results run against the intuition that a strong ticket needs a rich random source. At 3B, a rank-1 source of $24{,}576$ independent random scalars, roughly 10^{-5} of the projection connections, reaches 16.00 perplexity against 15.95 for per-projection independent weights, and 38.12 against 37.21 mean zero-shot accuracy; a controlled 2×2 experiment at 0.335B with three paired seeds finds that sharing one pool and constraining its rank each cost less than 0.3 perplexity, within seed-to-seed variation. The comparisons are largely single runs and LLaMA remains stronger at every scale, but across the configurations we tested the randomness of the random weight has little influence on the strong ticket that mask learning produces.

Prior work makes this paradigm practical with dedicated generation hardware; because our source fits in memory, we instead give an implementation for general-purpose GPUs, on which no random weight has to be regenerated during a forward pass. A straightforward implementation nevertheless rebuilds the mask at every projection evaluation. Across four model scales, Mask packing accelerates prefill and decode by $1.42\times$–$6.25\times$ and $1.38\times$–$6.16\times$, respectively, while Threshold caching accelerates training by $1.18\times$–$1.59\times$.

Our contributions are fourfold:

- We show that the randomness of the random weight has little effect on the strong-ticket model that mask learning produces: across 0.110B–3B backbones, shrinking the random source by up to five orders of magnitude leaves mean perplexity within 1.4 of per-projection independent weights, and a controlled four-arm experiment on three paired seeds finds no resolvable effect from either axis.

- We introduce RandomLLM, which compresses both the rank of each random weight and its independence across projections while leaving the number of trainable mask scores unchanged.
- We explain why the compression is possible: element-wise masking can lift the rank bound of the random weight, so a low-rank random weight need not yield a low-rank masked weight. At random rank one, a projection admits a materialization-free evaluation with unchanged matrix-multiplication FLOPs and only two additional element-wise scalings.
- We provide a general-purpose GPU implementation that removes repeated mask computation. Across four model scales, Mask packing accelerates prefill and decode by $1.42\times$–$6.25\times$ and $1.38\times$–$6.16\times$, while Threshold caching accelerates training by $1.18\times$–$1.59\times$.

2 Problem Formulation

We first define the random weight, mask scores, multivalued mask, and masked weight, then specify the mask rule used by TicketLLM and RandomLLM.

Let j index a projection in a decoder-only transformer, including its block and projection type. For input $x \in \mathbb{R}^{n_j}$ and output dimension m_j, projection j computes

$$y_j = W_j x, \qquad W_j = M_j \odot W_j^0, \tag{1}$$

where W_j^0 is a frozen random weight, $S_j \in \mathbb{R}^{m_j \times n_j}$ contains trainable mask scores, $M_j = M(S_j)$ is the multivalued mask, and W_j is the masked weight. Write $P = \sum_j m_j n_j$ for the number of projection connections. The random weight, mask scores, and multivalued mask are distinct objects: sharing the random pool does not remove the P mask scores used during training, and these real-valued scores are replaced by discrete masks for deployment. Embeddings and normalization parameters follow their training recipes and are accounted for separately.

Independent and shared random weights. TicketLLM generates W_j^0 separately for every j. RandomLLM instead slices every W_j^0 from one shared random pool generated by two frozen random factors. Our empirical question is whether mask learning remains effective under this restriction at fixed backbone dimensions. We study this random-weight construction rather than claiming a new universal approximation theorem for strong tickets.

Multivalued mask. We use the three-threshold multivalued mask from multicoated masking (Okoshi et al., 2022):

$$\begin{aligned} M_j \equiv M(S_j) &= \sum_{t=1}^{3} \mathbf{1}\{|S_j| \geq \tau_{j,t}\}, \\ \tau_{j,t} &= \mathrm{Quantile}_{\rho_t}(|S_j|), \\ (\rho_1, \rho_2, \rho_3) &= (0.50, 0.67, 0.83). \end{aligned} \tag{2}$$

Specifically, let $s_{j,(0)} \leq \cdots \leq s_{j,(N_j-1)}$ be the sorted entries of $|S_j|$, where $N_j = m_j n_j$. The implementation uses $\tau_{j,t} = s_{j,(k_t)}$ with the zero-based index $k_t = 1 + \lfloor \rho_t (N_j - 1) \rfloor$, and the non-strict inequality in Eq. (2) retains all ties at a threshold. The nested supports retain approximately 50%, 33%, and 17% of entries, producing values in $\{0, 1, 2, 3\}$ and approximately 50% zeros. The same multivalued mask and thresholding rule are used in RandomLLM and TicketLLM.

3 RandomLLM

RandomLLM retains projection-specific mask scores while replacing independently generated random weights with slices from one shared random pool, as illustrated in Figure 1. We first describe the shared random pool and mask training, then analyze rank and storage, and finally present the training and inference implementation.

3.1 SHARED RANDOM POOL

Choose pool dimensions $H_A \geq \max_j m_j$, $H_B \geq \max_j n_j$, and random rank r. Sample two independent standard Gaussian random factors

$$A \in \mathbb{R}^{H_A \times r}, \qquad B \in \mathbb{R}^{r \times H_B}, \tag{3}$$

and freeze them for the duration of training. Their product defines the shared random pool

$$R = \frac{AB}{\sqrt{r}} \in \mathbb{R}^{H_A \times H_B}. \tag{4}$$

The full R is a mathematical definition and need not be stored. Projection j has fixed row and column offsets (u_j, v_j) and slices the random weight

$$W_j^0 = \frac{R[u_j : u_j + m_j,\ v_j : v_j + n_j]}{\sqrt{n_j}} = \frac{A[u_j : u_j + m_j, :]B[:, v_j : v_j + n_j]}{\sqrt{r n_j}}. \tag{5}$$

Offsets are generated deterministically from the random-weight seed and projection index within the valid slice ranges. They are not optimized. Overlapping slices intentionally reuse entries of R; different projection indices do not guarantee disjoint slices. The two random factors contain $r(H_A + H_B)$ stored values, independent of the number of projections.

Combining Eqs. (1) and (5) gives the complete projection:

$$\boxed{y_j = \left[M_j \odot W_j^0\right] x = W_j x.} \tag{6}$$

This construction is used for query, key, value, attention output, and the gate, up, and down projections in the feed-forward module. The decoder retains its attention, residual, and gated feed-forward organization. Projection normalization follows the sparse-model recipes described in Appendix B.

3.2 MASK TRAINING

Training minimizes the autoregressive language-modeling loss through the masked weights. Each mask-score matrix is initialized with Kaiming uniform initialization. The random factors and random weights remain frozen, while each S_j is updated with a straight-through estimator that treats the discrete order-statistic thresholds as constants in the backward pass. If $G_j = \partial \mathcal{L} / \partial W_j$, the implemented mask-score gradient is

$$\widetilde{\nabla}_{S_j} \mathcal{L} = 3\ \mathrm{sign}(S_j) \odot G_j \odot W_j^0. \tag{7}$$

The factor three is the backward scaling used by the three-threshold implementation. The estimator supplies gradients even to connections whose current mask value is zero. Changing the random rank leaves the $m_j n_j$ trainable mask scores intact, but changes their fixed random-weight-dependent gradient multipliers. At deployment, each mask value can be stored in two bits; this bit width describes the mask, not the precision or number of distinct values of the masked weight.

3.3 WEIGHT PROPERTIES

The sliced product in Eq. (5) has inner dimension r. Thus the random weight W_j^0 has rank at most the random rank r. Element-wise masking changes this constraint because a general mask is not a low-rank matrix product. Writing $a_{j,q}$ for column q of the sliced factor A and $b_{j,q}^\top$ for row q of the sliced factor B gives

$$W_j = \frac{1}{\sqrt{r n_j}} \sum_{q=1}^{r} \mathrm{diag}(a_{j,q}) M_j \,\mathrm{diag}(b_{j,q}). \tag{8}$$

For Gaussian factors, the diagonal matrices in each summand are invertible almost surely, so an individual summand has the same rank as M_j. Their sum need not preserve that rank, but Eq. (8) shows why the random rank of W_j^0 does not directly bound the rank of W_j. This does not imply unrestricted expressivity: the masked weight remains constrained by fixed random coefficients and discrete nonnegative mask values.

Random-weight statistics. For the independent standard Gaussian factors in Eq. (5), the second moment of each factor entry is one and the fourth moment is three. Substituting these moments into the normalized product gives

$$\mathbb{E}[W_{ik}^0] = 0, \qquad \mathbb{E}[(W_{ik}^0)^2] = \frac{1}{n}, \qquad \frac{\mathbb{E}[(W_{ik}^0)^4]}{\mathbb{E}[(W_{ik}^0)^2]^2} = 3 + \frac{6}{r}. \tag{9}$$

Thus normalization fixes the entry-wise variance across ranks, while higher moments depend on r and shared row and column factors induce dependencies among coefficients even when pairwise covariances vanish. These statistics motivate the random rank ablations but do not predict monotonic performance or identify a single causal mechanism. The moment derivation is given in Appendix A.

3.4 STORAGE

Let random-factor values use b_f bits, masks use b_m bits, and offsets and other metadata use B_{meta} bits. RandomLLM's total projection storage is

$$B_{\text{proj}} = b_m P + b_f r(H_A + H_B) + B_{\text{meta}}. \tag{10}$$

With $b_f = 16$ and $b_m = 2$, ignoring the small metadata term, its storage in 16-bit-equivalent values is $P/8 + r(H_A + H_B)$. The corresponding LLaMA weights cost P, and explicitly storing every TicketLLM random weight in 16 bits together with its two-bit mask costs $9P/8$. Table 1 reports both the number of independently sampled random parameters and the projection storage from Eq. (10) in MB or GB. For the evaluated pool dimensions, mask storage dominates random-factor storage; the count includes zero mask entries, so the 50% sparsity is not applied as a second discount.

This advantage concerns explicit projection storage; seed-regenerated TicketLLM can also avoid storing random weights, and our current implementation does not yet reduce peak training memory.

3.5 HIGH-PERFORMANCE IMPLEMENTATION

A direct PyTorch implementation of Eq. (6) repeatedly sorts each mask-score tensor to obtain the thresholds, constructs the multivalued mask in floating point, materializes the masked weight W_j, and then invokes a matrix multiplication; we use this implementation as the PyTorch baseline. During gradient accumulation, the mask scores do not change until the optimizer step, and at inference they remain fixed for the entire request. We exploit this structure with a Triton backend (Tillet et al., 2019) that preserves the discrete order-statistic convention of Eq. (2) while moving repeated mask work out of the projection loop.

Training. Threshold caching computes the three thresholds once for the current mask scores and invalidates them after an optimizer update. A Triton kernel materializes the multivalued masked weight, while cuBLAS performs the forward, input-gradient, and weight-gradient matrix multiplications. Weight caching additionally casts and reuses the masked weight in the active mixed-precision datatype across accumulation microbatches. The custom backward maps the masked-weight gradient to the mask-score gradient using Eq. (7), preserving the straight-through rule.

Inference. The PyTorch baseline recomputes the thresholds, multivalued mask, and masked weight from FP mask scores during each projection evaluation. Mask packing stores four two-bit mask values per byte; a tiled Triton kernel unpacks them while loading a random-weight tile and accumulates the matrix product in FP32. Weight caching materializes and stores every masked weight once at model load. Mask packing removes FP mask-score reads and threshold sorting from the token loop, whereas Weight caching trades additional memory for a conventional matrix multiplication. At model load, an automatic policy selects among Weight caching, Mask packing, and the PyTorch baseline according to available device memory; explicit selections remain available for reproducible measurement. Stored masks and weights are checked against the current mask scores to prevent stale execution.

Mask packing and Weight caching currently consume the materialized W_j^0 produced at initialization rather than generating random-weight tiles directly from A and B. They therefore accelerate masked-weight execution but do not yet keep only the random factors in live GPU memory.

4 EXPERIMENTS

We organize the evaluation around three questions: whether one shared random pool remains effective across model scales, how a shared pool and low-rank factors separately affect performance, and whether masked weights can be executed efficiently.

4.1 EXPERIMENTAL SETUP

Data and metrics. We train LLaMA-style decoder backbones from scratch on packed 2,048-token sequences from FineWebEdu sample-10BT-derived streams (Touvron et al., 2023; Penedo et al., 2024). Perplexity is measured on a FineWebEdu held-out split disjoint from the training stream. Downstream evaluation covers ARC-Easy and ARC-Challenge (Clark et al., 2018), HellaSwag (Zellers et al., 2019), OpenBookQA (Mihaylov et al., 2018), BoolQ (Clark et al., 2019), PIQA (Bisk et al., 2020), WinoGrande (Sakaguchi et al., 2020), COPA (Roemmele et al., 2011), MMLU (Hendrycks et al., 2021), and LAMBADA (Paperno et al., 2016). We report the unweighted mean of the ten task accuracies. ARC option-count partitions are merged within each task by sample count. Table 1 reports every task score; Appendix B specifies the available training and evaluation details.

Models and comparisons. We compare LLaMA, TicketLLM, and RandomLLM. TicketLLM denotes our implementation of independently generated random weights with a multivalued mask under the common transformer recipe; it does not exactly reproduce the published Ada-Sup optimizer or binary random weights of Okoshi et al. (2025). TicketLLM and RandomLLM use the same multivalued mask and backbone dimensions, while their random-weight constructions and initialization distributions differ; all RandomLLM runs use $H_A = H_B = 12{,}288$. The archived TicketLLM and RandomLLM checkpoints use different random-weight initializations, so the cross-scale table compares complete configurations; Appendix B.1 audits this difference, and the controlled ablation below supplies the distribution-matched comparison.

Training and uncertainty. The 0.335B RandomLLM rank sweep uses 8,000 steps at global batch size 64, while the separate controlled 0.335B ablation uses the same 8,000-step budget with three paired training/random-weight seed pairs. The 0.7B models use 4,172 steps at global batch size 512 and sequence length 2,048. At 3B TicketLLM and RandomLLM both use 23,000 steps, while LLaMA uses approximately 21,500 steps with optimizer restarts at continuation boundaries. The 0.110B LLaMA and TicketLLM runs also include weight-only continuations, and some other archived histories are incomplete. We therefore treat the cross-scale results as descriptive observations rather than a controlled scaling law; the matched ablation below supports causal comparisons. All main-table entries are individual observed runs, and all RandomLLM entries use global seed 42.

4.2 MAIN RESULTS

Within the reported runs, at least one RandomLLM configuration attains lower PPL than TicketLLM at every model scale. Selecting the lowest-PPL RandomLLM configuration at each scale, the reduction relative to TicketLLM ranges from 0.09 to 2.29, while the corresponding mean zero-shot accuracy difference ranges from -0.55 to $+1.19$ percentage points. The 3B results illustrate the compression directly: random rank 1 uses only $24{,}576$ random scalars instead of 2.72 billion and nearly matches TicketLLM in PPL (16.00 versus 15.95), while improving mean zero-shot accuracy from 37.21 to 38.12. Increasing the random rank to 8 still uses only $196{,}608$ random scalars and reaches 13.66 PPL and 38.40 mean accuracy. Under the projection-storage accounting of Section 3.4, the selected RandomLLM configurations use about one-tenth as much storage as explicitly storing TicketLLM random weights, because RandomLLM stores only the random factors and masks; this comparison does not imply the same advantage over seed-regenerated TicketLLM storage. LLaMA remains the strongest reference at every scale. The best observed PPL occurs at random ranks 2, 8, 4, and 8 from 0.110B through 3B, respectively; although PPL improves with rank at 0.335B and 3B, the 0.110B and 0.7B sweeps are non-monotonic. Because the random rank sweeps are predominantly single runs, these values are descriptive observations rather than estimates from a prespecified rank-selection procedure and do not establish a universal random rank or a broad capability advantage.

Table 1: Main results and projection storage across four model scales. Avg. is the mean accuracy over ten zero-shot tasks; bold marks the best observed PPL and Avg. within each scale.

Model	Random Rank r	Random scalars	Proj. storage	PPL	ARCe	ARCc	HS	OQ	BQ	PQ	WGe	COPA	MMLU	LAMB	Avg.
0.110B															
LLaMA	–	–	169.87 MB	**21.13**	36.14	22.74	27.12	27.20	58.23	57.94	52.49	41.00	22.89	15.77	**36.15**
TicketLLM	–	84.93M	191.10 MB	24.53	34.21	20.40	26.54	24.60	61.19	56.26	52.57	40.00	22.99	11.93	35.07
RandomLLM	1	24,576	21.28 MB	25.20	34.74	21.74	26.68	23.60	47.89	56.53	51.46	40.00	22.96	13.66	33.93
RandomLLM	2	49,152	21.33 MB	23.59	32.63	23.08	26.39	24.20	58.07	55.39	50.91	42.00	22.89	13.06	34.86
RandomLLM	4	98,304	21.43 MB	24.96	34.21	23.08	26.35	24.80	53.30	56.47	50.91	40.00	22.98	12.71	34.48
RandomLLM	8	196,608	21.63 MB	24.04	33.33	24.08	26.71	25.40	58.47	57.51	49.41	44.00	22.95	13.14	35.50
0.335B															
LLaMA	–	–	604.03 MB	**17.66**	38.95	23.75	29.00	25.80	60.70	59.58	52.64	45.00	22.92	17.66	**37.60**
TicketLLM	–	302.01M	679.53 MB	21.77	36.84	23.41	27.29	25.20	61.65	57.07	52.96	44.00	22.89	14.42	36.57
RandomLLM	1	24,576	75.55 MB	22.63	34.21	20.74	27.06	27.20	58.90	56.96	50.20	37.00	22.82	13.12	34.82
RandomLLM	2	49,152	75.60 MB	22.02	35.26	21.40	27.10	26.40	58.44	56.80	50.83	40.00	22.87	14.38	35.35
RandomLLM	4	98,304	75.70 MB	21.87	35.09	22.07	27.15	28.00	56.61	57.07	49.64	41.00	22.90	13.82	35.34
RandomLLM	8	196,608	75.90 MB	21.32	34.74	22.07	27.23	24.80	59.51	57.73	50.99	44.00	22.87	16.25	36.02
0.7B															
LLaMA	–	–	1.36 GB	**14.35**	39.82	24.08	30.65	29.20	51.53	61.81	50.75	46.00	22.84	19.31	**37.60**
TicketLLM	–	679.48M	1.53 GB	17.57	37.19	23.75	28.35	26.00	60.18	59.47	51.70	45.00	22.89	14.46	36.90
RandomLLM	1	24,576	169.92 MB	19.97	35.44	22.07	27.41	26.40	56.79	57.83	50.99	40.00	23.05	12.57	35.26
RandomLLM	2	49,152	169.97 MB	18.38	35.61	23.41	27.73	26.60	55.63	58.65	51.62	45.00	22.91	15.16	36.23
RandomLLM	4	98,304	170.07 MB	17.48	37.19	22.74	28.49	24.60	59.69	59.30	52.01	48.00	22.92	16.80	37.18
RandomLLM	8	196,608	170.26 MB	18.16	37.89	21.40	28.12	25.20	61.96	59.63	51.38	44.00	22.92	16.10	36.86
3B															
LLaMA	–	–	5.44 GB	**12.35**	44.04	27.42	35.47	30.60	54.10	64.96	50.83	58.00	23.62	23.08	**41.21**
TicketLLM	–	2.72B	6.12 GB	15.95	40.70	21.40	30.49	27.20	51.10	61.48	53.59	46.00	22.88	17.25	37.21
RandomLLM	1	24,576	679.53 MB	16.00	38.95	21.07	30.01	27.00	57.49	61.92	52.96	51.00	22.91	17.85	38.12
RandomLLM	2	49,152	679.58 MB	14.49	39.65	22.41	30.87	28.60	50.86	62.89	52.25	46.00	22.91	17.85	37.43
RandomLLM	4	98,304	679.67 MB	14.05	42.11	22.41	31.00	28.00	53.82	62.62	52.01	47.00	22.97	19.45	38.14
RandomLLM	8	196,608	679.87 MB	13.66	42.46	23.41	31.70	29.80	48.26	63.76	51.62	49.00	22.94	21.05	38.40

Random scalars counts independently sampled, frozen random values and excludes masks. Total projection storage is analyzed in Section 3.4 (MB $= 10^6$ bytes; GB $= 10^9$ bytes). ARCe/ARCc denote ARC-Easy/Challenge; HS, OQ, BQ, PQ, WGe, and LAMB denote HellaSwag, OpenBookQA, BoolQ, PIQA, WinoGrande, and LAMBADA. Training-budget qualifications are given in Section 4.1.

4.3 Ablation Study

Table 2 compares four configurations. TicketLLM gives each projection its own random weight and does not use low-rank factors. Shared full-rank source slices each random weight from one shared random pool, also without low-rank factors. Independent rank-4 source gives each projection its own random factors with random rank 4. Rank-4 shared source, or RandomLLM, uses one pair of random factors with random rank 4 to define the shared random pool from which all random weights are sliced. The four-arm protocol fixes the 0.335B backbone and samples the entries used by TicketLLM and Shared full-rank source from the same marginal Gaussian-product distribution produced at random rank 4 (Appendix B.3). Within this ablation, TicketLLM denotes this distribution-matched configuration; the archived TicketLLM checkpoint remains a separate reference because its initialization differs. Multivalued masks, training streams, budgets, and seeds are matched across all four configurations.

We compute the effect of each design choice within each paired training/random-weight seed and then average the effects across the three pairs. Averaged over the two choices in the low-rank factors column, using a shared pool changes PPL by $+0.124 \pm 0.287$ and mean zero-shot accuracy by $+0.076 \pm 0.621$ percentage points. Averaged over the two pool choices, using low-rank factors changes PPL by $+0.285 \pm 0.144$ and mean zero-shot accuracy by -0.200 ± 0.432 percentage points. In this experiment, the larger mean PPL penalty is therefore associated with low-rank factors rather than the shared pool, whereas neither choice shows a stable mean zero-shot accuracy advantage with three seed pairs; Appendix Table 4 reports the paired main effects and interaction.

4.4 System Efficiency

We benchmarked the PyTorch baseline and Weight caching in both training and inference, with Mask packing as the intermediate inference strategy and Threshold caching as the intermediate training strategy, on rank-1 RandomLLM architectures at four scales using one NVIDIA H20 GPU. Each

Table 2: Four random-weight constructions at 0.335B. Entries report the mean and sample standard deviation over three paired training/random-weight seeds. Within this table, TicketLLM is the distribution-matched configuration without low-rank factors or a shared pool; its random-weight coefficients match the marginals produced at random rank 4.

Configuration	low-rank factors	Shared pool	PPL ↓	Avg. ↑
TicketLLM	No	No	20.784 ± 0.249	36.189 ± 0.393
Shared full-rank source	No	Yes	21.117 ± 0.326	35.850 ± 0.321
Independent rank-4 source	Yes	No	21.278 ± 0.320	35.573 ± 0.559
Rank-4 shared source (RandomLLM)	Yes	Yes	21.192 ± 0.229	36.065 ± 0.632

inference run used a 128-token prompt followed by 16 fixed incremental decode tokens in FP16; Figure 2 reports medians of three runs after one warmup, excluding model-load preparation and first-time compilation. The 0.110B, 0.335B, and 0.7B measurements use freshly initialized, shape-matched models because their trained checkpoints are not available locally; the 3B measurements use the trained checkpoint. Thus, the first three scales characterize execution shapes and numerical consistency rather than model quality.

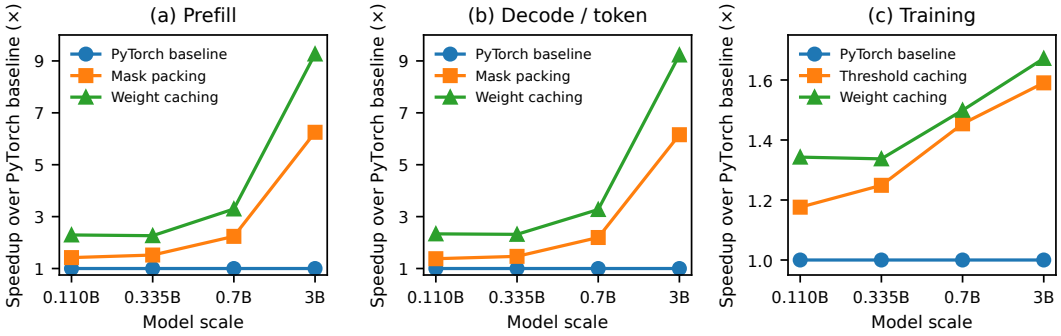


Figure 2: Kernel speedups across model scales on one NVIDIA H20. Panels (a) and (b) compare the PyTorch baseline, Mask packing, and Weight caching for prefill and incremental decode; panel (c) compares the PyTorch baseline, Threshold caching, and Weight caching for training throughput. All measurements use FP16 and exclude preparation and first-time compilation; the first three scales are shape-matched models, while 3B uses the trained checkpoint.

Inference speedup increases sharply with scale. At 0.110B, Mask packing improves prefill and decode by $1.42\times$ and $1.38\times$, while Weight caching reaches $2.29\times$ and $2.34\times$; at 3B these become $6.25\times$, $6.16\times$, $9.27\times$, and $9.23\times$, respectively. The corresponding packed-mask and cached-weight artifacts grow from 20.25 and 162 MiB at 0.110B to 648 and 5,184 MiB at 3B. For training, Threshold caching increases the speedup from $1.176\times$ at 0.110B to $1.590\times$ at 3B, while the speedup from Weight caching rises from $1.343\times$ to $1.672\times$; each three-way benchmark has the same forward loss, and the 3B runs differ by less than 1 MiB in peak allocated memory. On the full FineWebEdu held-out split for the trained 3B checkpoint, all three inference methods produced byte-identical metric files, while Mask packing reduced end-to-end evaluation time by $1.14\times$. These measurements establish an implementation tradeoff on the tested checkpoint and GPU, not a hardware-general speedup claim. Appendix C gives the complete timing protocol and numerical-consistency checks.

5 Related Work

Strong tickets and low-bit models. Lottery-ticket work studies sparse subnetworks trainable from their initial weights, while strong-ticket work learns masks over frozen random networks and analyzes mask search, approximation, and resampling (Frankle & Carbin, 2019; Zhou et al., 2019; Ramanujan et al., 2020; Malach et al., 2020; Chijiwa et al., 2021). TicketLLM develops Ada-Sup for sparse low-bit transformers, builds on nested multivalued mask selection, and explicitly discusses seed-based weight regeneration (Okoshi et al., 2022; 2025). Slot Machines lets each connection select among fixed random values (Aladago & Torresani, 2021); RandomLLM instead slices random

weights from a shared pool. A matched Ada-Sup comparison remains necessary, and attention-specific SLTH guarantees (Otsuka et al., 2026) do not automatically transfer to our shared-pool construction.

Weight sharing and low-rank methods. Hidden-Fold Networks combine recurrent weight reuse with mask learning (López García-Arias et al., 2021), while SupSup shares a frozen network across tasks with task-specific masks (Wortsman et al., 2020). RandomLLM instead shares random factors across projections and retains a separate multivalued mask for each. LoRA learns low-rank updates (Hu et al., 2022), and VeRA shares frozen low-rank matrices with learned scaling vectors (Kopiczko et al., 2024); RandomLLM uses its frozen factors to generate random weights, so the masked weight follows a different rank construction.

Low-bit computation. Methods such as BitNet b1.58 constrain learned weights to a small value set (Ma et al., 2024). RandomLLM assigns two bits to a mask multiplying a frozen random weight; this does not generally make the masked weight a conventional two-bit quantized weight tensor. A fair efficiency comparison must include the random factors, masks, normalization scales, and regeneration work and use matched execution conditions. The LLaMA and TicketLLM results here establish the present empirical scope; we do not claim a state-of-the-art comparison against low-bit training methods.

6 Conclusion

RandomLLM slices projection-specific random weights from a shared pool defined by two frozen factors and trains a separate multivalued mask for each. Across the reported scales, it approaches TicketLLM with substantially less explicit projection storage, although LLaMA remains stronger. The controlled 0.335B experiment finds a modest PPL cost and no stable mean zero-shot accuracy advantage over distribution-matched TicketLLM. Our analysis shows that the masked weight need not inherit the source rank bound, but the single-run sweeps do not establish a universal random rank. Mask packing, Threshold caching, and Weight caching provide speed–memory tradeoffs for inference and training.

Limitations. The evidence is limited to the reported data streams, training budgets, and mostly single-run configurations; restarts, seeds, random-weight distributions, and 3B scheduling limit cross-scale interpretation. LLaMA remains ahead, and the random rank sweeps do not establish a universal rank rule. The controlled 0.335B comparison has only three paired runs and combines optimization, data-order, and random-weight-seed variation. Reduced random-factor storage does not demonstrate lower total training memory or an advantage over seed-only serialization. The kernels were tested on one H20 with materialized random weights; factor-only execution and broader hardware evaluation remain open.

Reproducibility statement

Appendices B and C and the supplementary artifacts document protocols, manifests, integrity checks, results, tests, and metric hashes. Incomplete cross-scale observations are marked and excluded from controlled claims.

AI use statement

Qwen assisted with proofreading and manuscript refinement, and DeepSeek with code debugging. The authors reviewed and revised all AI-generated output and take full responsibility for the paper.

References

Maxwell M. Aladago and Lorenzo Torresani. Slot machines: Discovering winning combinations of random weights in neural networks. In *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pp. 163–174, 2021. URL https://proceedings.mlr.press/v139/aladago21a.html.

Yonatan Bisk, Rowan Zellers, Ronan Le Bras, Jianfeng Gao, and Yejin Choi. PIQA: Reasoning about physical commonsense in natural language. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 2020. URL https://arxiv.org/abs/1911.11641.

Daiki Chijiwa, Shin'ya Yamaguchi, Yasutoshi Ida, Kenji Umakoshi, and Tomohiro Inoue. Pruning randomly initialized neural networks with iterative randomization. In *Advances in Neural Information Processing Systems*, volume 34, 2021. URL https://proceedings.neurips.cc/paper/2021/hash/23e582ad8087f2c03a5a31c125123f9a-Abstract.html.

Christopher Clark, Kenton Lee, Ming-Wei Chang, Tom Kwiatkowski, Michael Collins, and Kristina Toutanova. BoolQ: Exploring the surprising difficulty of natural yes/no questions. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, 2019. URL https://aclanthology.org/N19-1300/.

Peter Clark, Isaac Cowhey, Oren Etzioni, Tushar Khot, Ashish Sabharwal, Carissa Schoenick, and Oyvind Tafjord. Think you have solved question answering? try ARC, the AI2 reasoning challenge, 2018. URL https://arxiv.org/abs/1803.05457.

Arthur da Cunha, Francesco d'Amore, and Emanuele Natale. Polynomially over-parameterized convolutional neural networks contain structured strong winning lottery tickets. In *Advances in Neural Information Processing Systems*, volume 36, pp. 25929–25957, 2023. doi: 10.52202/075280-1128. URL https://papers.nips.cc/paper_files/paper/2023/hash/525338e0d98401a62950bc7c454eb83d-Abstract-Conference.html.

Jonathan Frankle and Michael Carbin. The lottery ticket hypothesis: Finding sparse, trainable neural networks. In *International Conference on Learning Representations*, 2019. URL https://arxiv.org/abs/1803.03635.

Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. Measuring massive multitask language understanding. In *International Conference on Learning Representations*, 2021. URL https://arxiv.org/abs/2009.03300.

Kazutoshi Hirose, Jaehoon Yu, Kota Ando, Yasuyuki Okoshi, Ángel López García-Arias, Junnosuke Suzuki, Thiem Van Chu, Kazushi Kawamura, and Masato Motomura. Hiddenite: 4k-pe hidden network inference 4d-tensor engine exploiting on-chip model construction achieving 34.8-to-16.0tops/w for cifar-100 and imagenet. In *2022 IEEE International Solid-State Circuits Conference (ISSCC)*, volume 65, pp. 252–254, 2022. doi: 10.1109/ISSCC42614.2022.9731668. URL https://doi.org/10.1109/ISSCC42614.2022.9731668.

Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations*, 2022. URL https://arxiv.org/abs/2106.09685.

Dawid J. Kopiczko, Tijmen Blankevoort, and Yuki M. Asano. VeRA: Vector-based random matrix adaptation. In *International Conference on Learning Representations*, 2024. URL https://arxiv.org/abs/2310.11454.

Ángel López García-Arias, Masanori Hashimoto, Masato Motomura, and Jaehoon Yu. Hidden-fold networks: Random recurrent residuals using sparse supermasks. In *British Machine Vision Conference*, 2021. URL https://arxiv.org/abs/2111.12330.

Shuming Ma, Hongyu Wang, Lingxiao Ma, Lei Wang, Wenhui Wang, Shaohan Huang, Li Dong, Ruiping Wang, Jilong Xue, and Furu Wei. The era of 1-bit LLMs: All large language models are in 1.58 bits, 2024. URL https://arxiv.org/abs/2402.17764.

Eran Malach, Gilad Yehudai, Shai Shalev-Schwartz, and Ohad Shamir. Proving the lottery ticket hypothesis: Pruning is all you need. In *International Conference on Machine Learning*, 2020.

Todor Mihaylov, Peter Clark, Tushar Khot, and Ashish Sabharwal. Can a suit of armor conduct electricity? a new dataset for open book question answering. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, 2018. URL https://aclanthology.org/D18-1260/.

Yasuyuki Okoshi, Ángel López García-Arias, Kazutoshi Hirose, Kota Ando, Kazushi Kawamura, Thiem Van Chu, Masato Motomura, and Jaehoon Yu. Multicoated supermasks enhance hidden networks. In *Proceedings of the 39th International Conference on Machine Learning*, volume 162 of *Proceedings of Machine Learning Research*, pp. 17045–17055, 2022. URL https://proceedings.mlr.press/v162/okoshi22a.html.

Yasuyuki Okoshi, Hikari Otsuka, Daichi Fujiki, and Masato Motomura. TicketLLM: Next-generation sparse and low-bit transformers with supermask-based method. *Transactions on Machine Learning Research*, 2025. URL https://openreview.net/forum?id=sE69HKykQw.

Hikari Otsuka, Daiki Chijiwa, Yasuyuki Okoshi, Daichi Fujiki, Susumu Takeuchi, and Masato Motomura. The strong lottery ticket hypothesis for multi-head attention mechanisms. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 2026. URL https://arxiv.org/abs/2511.04217.

Denis Paperno, Germán Kruszewski, Angeliki Lazaridou, Ngoc Quan Pham, Raffaella Bernardi, Sandro Pezzelle, Marco Baroni, Gemma Boleda, and Raquel Fernández. The LAMBADA dataset: Word prediction requiring a broad discourse context. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics*, 2016. URL https://aclanthology.org/P16-1144/.

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. In *Advances in Neural Information Processing Systems*, 2024. URL https://arxiv.org/abs/2406.17557.

Ankit Pensia, Shashank Rajput, Alliot Nagle, Harit Vishwakarma, and Dimitris Papailiopoulos. Optimal lottery tickets via subset sum: Logarithmic over-parameterization is sufficient. In *Advances in Neural Information Processing Systems*, volume 33, pp. 2599–2610, 2020. URL https://proceedings.neurips.cc/paper/2020/hash/1b742ae215adf18b75449c6e272fd92d-Abstract.html.

Vivek Ramanujan, Mitchell Wortsman, Aniruddha Kembhavi, Ali Farhadi, and Mohammad Rastegari. What's hidden in a randomly weighted neural network? In *IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2020. URL https://arxiv.org/abs/1911.13299.

Melissa Roemmele, Cosmin Adrian Bejan, and Andrew S. Gordon. Choice of plausible alternatives: An evaluation of commonsense causal reasoning. In *AAAI Spring Symposium on Logical Formalizations of Commonsense Reasoning*, 2011. URL https://commonsensereasoning.org/2011/papers/Roemmele.pdf.

Keisuke Sakaguchi, Ronan Le Bras, Chandra Bhagavatula, and Yejin Choi. WinoGrande: An adversarial winograd schema challenge at scale. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 2020. URL https://arxiv.org/abs/1907.10641.

Philippe Tillet, H. T. Kung, and David Cox. Triton: An intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pp. 10–19, 2019. doi: 10.1145/3315508.3329973. URL https://doi.org/10.1145/3315508.3329973.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. LLaMA: Open and efficient foundation language models, 2023. URL https://arxiv.org/abs/2302.13971.

Mitchell Wortsman, Vivek Ramanujan, Rosanne Liu, Aniruddha Kembhavi, Mohammad Rastegari, Jason Yosinski, and Ali Farhadi. Supermasks in superposition. In *Advances in Neural Information Processing Systems*, volume 33, 2020. URL https://proceedings.neurips.cc/paper/2020/hash/ad1f8bb9b51f023cdc80cf94bb615aa9-Abstract.html.

Jiale Yan, Hiroaki Ito, Yuta Nagahara, Kazushi Kawamura, Masato Motomura, Thiem Van Chu, and Daichi Fujiki. Bingogcn: Towards scalable and efficient gnn acceleration with fine-grained partitioning and slt. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture (ISCA '25)*, pp. 1910–1924, 2025. doi: 10.1145/3695053.3731115. URL https://dl.acm.org/doi/10.1145/3695053.3731115.

Rowan Zellers, Ari Holtzman, Yonatan Bisk, Ali Farhadi, and Yejin Choi. HellaSwag: Can a machine really finish your sentence? In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, 2019. URL https://aclanthology.org/P19-1472/.

Hattie Zhou, Janice Lan, Rosanne Liu, and Jason Yosinski. Deconstructing lottery tickets: Zeros, signs, and the supermask. In *Advances in Neural Information Processing Systems*, volume 32, 2019. URL https://proceedings.neurips.cc/paper/2019/hash/1113d7a76ffceca1bb350bfe145467c6-Abstract.html.

A Random-Weight Moments

Standard Gaussian entries have zero mean, unit second moment, and fourth moment three. Hence, for $Z_q = A_{iq}B_{qk}$, $\mathbb{E}[Z_q] = 0$, $\mathbb{E}[Z_q^2] = 1$, and $\mathbb{E}[Z_q^4] = \mathbb{E}[A_{iq}^4]\mathbb{E}[B_{qk}^4] = 3 \times 3 = 9$. The products are independent across q; expanding the fourth power and retaining the nonzero even-order terms gives

$$\mathbb{E}\left[\left(\sum_{q=1}^{r} Z_q\right)^2\right] = r, \tag{11}$$

$$\mathbb{E}\left[\left(\sum_{q=1}^{r} Z_q\right)^4\right] = 9r + 6\binom{r}{2} = 3r^2 + 6r. \tag{12}$$

Division by $\sqrt{rn}$ gives Eq. (9). This calculation concerns the random-weight coefficients; the multivalued mask depends on both the random weight and training, so these moments do not transfer directly to W.

B Experimental Details

B.1 Models and Training

The model scales are nominal backbone labels. The repository implements LLaMA-style decoders with rotary position embeddings, RMS normalization, bias-free projections, and a gated SiLU feed-forward block. Table 3 gives the dimensions used for projection accounting. Input and output vocabulary weights and all normalization parameters are excluded from P. The labels do not denote pretrained LLaMA checkpoints or the number of trainable parameters after compression.

Table 3: Backbone dimensions and exact projection connection counts. $P = L(4d^2 + 3df)$, where f is the feed-forward hidden dimension. The vocabulary size is 32,000 and sequence length is 2,048.

Scale	Blocks L	Width d	Heads	FFN width f	P
0.110B	12	768	12	2,048	84,934,656
0.335B	24	1,024	16	2,731	302,014,464
0.7B	24	1,536	24	4,096	679,477,248
3B	24	3,072	48	8,192	2,717,908,992

LLaMA learns its projection weights. TicketLLM and RandomLLM freeze their random weights and learn full-size mask scores, with additional input normalization on attention-output and feed-forward projections. Embeddings and normalization parameters remain trainable. The query, key, and value projections use the corresponding unnormalized sparse projection class. RandomLLM samples A with its random-weight seed and B with the next seed, and derives projection offsets from deterministic module identifiers. The random factors, offsets, and random weights remain fixed during training, while mask scores use Kaiming uniform initialization.

Archived TicketLLM initialization. The inspected archived TicketLLM configuration applies a model-level truncated-normal initializer after constructing its projection weights; the inspected 0.7B checkpoint contains continuous frozen values with standard deviation approximately 0.02. RandomLLM restores its factor-generated random weights after generic initialization. The main cross-scale table therefore compares complete archived configurations, while the controlled ablation matches coefficient marginals across its four configurations.

Training. The 0.7B families use matched 4,172-step runs; the TicketLLM–RandomLLM comparison at 3B uses 23,000 steps. The 0.110B LLaMA and TicketLLM runs and the 3B LLaMA run include weight-only continuations that do not restore optimizer or scheduler state. Documented RandomLLM learning rates are 3×10^{-3} for the 0.110B and 0.335B sweeps, 3.9×10^{-4} for the 0.7B runs, and 3×10^{-3} for the archived 3B run. Warmup lasts 300, 42, and 400 steps at 0.335B,

0.7B, and 3B, respectively. The recorded 0.335B and 0.7B runs use AdamW, gradient clipping, BF16 mixed precision, and microbatches of two sequences. Incomplete archived manifests make the remaining cross-scale rows observational references rather than controlled comparisons.

B.2 Data and Evaluation

Training streams are prepared from FineWebEdu `sample-10BT` using the repository's LLaMA tokenizer and packed 2,048-token chunks. The configured full held-out split contains 4,096 chunks (8,388,608 tokens); it is disjoint from the training stream and evaluated without shuffling. Multiple-choice tasks use the repository's likelihood-based in-context evaluation loaders. Options are scored by mean continuation loss; MMLU scores answer labels, and LAMBADA uses the final word as the continuation. ARC-Easy and ARC-Challenge are each split internally by option count and then recombined by sample count. The reported Avg. gives equal weight to the ten final task scores, irrespective of task size. The controlled ablation records the selected data ranges, tokenizer, preprocessing configuration, task files, and file hashes.

B.3 Ablation Setup

The completed four-arm experiment fixes the 0.335B backbone, the three multivalued-mask thresholds, normalization, mask-score initialization, optimizer, and token budget. For TicketLLM and Shared full-rank source, each unscaled coefficient is an independent draw $r^{-1/2}\sum_{q=1}^{r} u_q v_q$, where $u_q, v_q \sim \mathcal{N}(0,1)$ independently and $r = 4$. Shared full-rank source slices from one $12{,}288 \times 12{,}288$ shared random pool, whereas TicketLLM samples each random weight independently. Independent rank-4 source uses independent Gaussian random factors for each projection, whereas RandomLLM uses the shared random factors A and B to define one shared random pool; both use random rank 4. All four configurations scale coefficients by $1/\sqrt{n}$, matching the single-entry distribution while varying row/column dependence and cross-projection reuse. The independent random-weight generators leave the initial mask-score RNG unchanged across configurations.

The fixed training set contains 268,800 chunks, whose contents were checked against the complete prefix of the larger archived stream. Validation uses subsequent chunk indices 268,800–272,895; test uses 272,896–276,991, with zero-based inclusive indexing. Each arm trains for 8,000 steps (1,048,576,000 processed tokens) at global batch 64, microbatch 2, learning rate 0.003, and 300 warmup steps; final evaluation uses the preselected last step, without selecting checkpoints on test results. The paired training/random-weight seeds are $(42, 20240624)$, $(43, 20240625)$, and $(44, 20240626)$. Every configuration uses the same optimized training implementation and BF16 mixed precision. The validation and test chunks in this new protocol differ from the held-out split of the archived main-table runs; their PPL values must not be pooled across protocols.

Figure 3 complements the fixed-budget final metrics in Table 2: TicketLLM and Shared full-rank source learn faster early in training, while Independent rank-4 source and RandomLLM follow similar trajectories throughout.

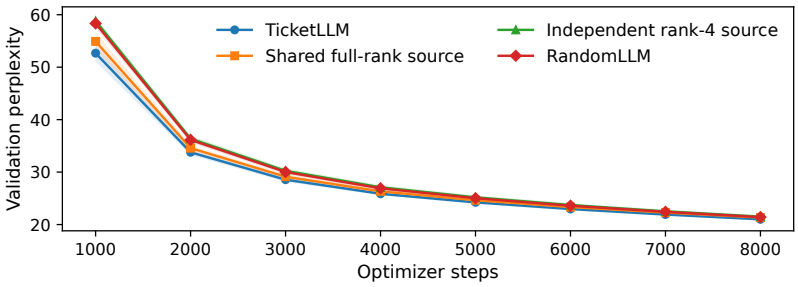


Figure 3: Validation PPL in the controlled 0.335B ablation. Lines and bands give the mean and sample standard deviation over the three paired training/random-weight seeds in Table 2. TicketLLM and Shared full-rank source learn faster early in training; Independent rank-4 source and RandomLLM track one another throughout.

Table 4: Paired ablation effects at 0.335B. Each effect is computed within a training/random-weight seed pair before reporting its mean and sample standard deviation across three pairs. Positive ΔPPL is worse; positive ΔAvg. is better.

Ablation effect	ΔPPL	ΔAvg. (pp)
Shared pool, averaged over weight types	$+0.124 \pm 0.287$	$+0.076 \pm 0.621$
Low-rank factors, averaged over pool choices	$+0.285 \pm 0.144$	-0.200 ± 0.432
Shared pool $\times$ low-rank factors	-0.419 ± 0.830	$+0.831 \pm 1.125$

C Kernel Benchmarks

C.1 Timing Setup

The benchmarked 3B checkpoint with random rank 1 contains 168 masked weights. Inference latency was measured on one NVIDIA H20 with the same checkpoint and FP16 datatype for every method. The prompt contained 128 tokens and greedy decoding produced 16 fixed incremental tokens. Reported values are medians of three runs after one warmup and exclude model loading, mask or weight preparation, and first-time Triton compilation.

Training used the same 3B checkpoint with FP32 parameters, FP16 autocast, sequence length 2,048, microbatch size two, and 32 accumulation microbatches. The timed region included the once-per-window exact-threshold construction.

Table 5: Detailed 3B training comparison on one NVIDIA H20. One accumulation window contains 32 microbatches of two 2,048-token sequences.

Method	Window (s) ↓	Microbatch (ms) ↓	Tokens/s ↑	Peak (MiB)
PyTorch baseline	42.14	1,316.81	3,111	65,047.83
Threshold caching	26.49	827.95	4,947	65,048.41
Weight caching	25.20	787.64	5,200	65,048.41

C.2 Numerical Consistency

The three training methods produced the same final training loss, 12.8182535; the maximum sampled mask-score-gradient difference across the optimized methods was 0.01218 under mixed precision. On the complete FineWebEdu held-out split, the PyTorch baseline, Mask packing, and Weight caching produced byte-identical metric files. Their evaluation times were 1,039.67, 909.41, and 525.51 seconds, respectively; the same three methods also produced identical metrics on the tested downstream subset.

A batched greedy-generation check used 32 prompts drawn equally from HellaSwag, BoolQ, MMLU, and LAMBADA and generated 16 tokens per prompt under BF16 autocast. Weight caching matched the PyTorch baseline on all 32 sequences and all 512 generated tokens. Mask packing matched 31 of 32 sequences and 505 of 512 tokens; the first difference occurred when mixed-precision rounding changed a near-tied argmax. Thus the tested aggregate metrics were unchanged, but Mask packing is not guaranteed to be token-identical near decision boundaries.