PaLM-Style Parallel Transformer Blocks to Speed Up NanoGPT Training

Alain Welliver

9 December 2025

1 Hypothesis

Standard transformer blocks process attention and MLP layers sequentially: the MLP cannot begin until attention completes. This creates a computational bottleneck and extends gradient path length to $2L$ for L layers. Google's PaLM architecture reformulates this by computing attention and MLP in parallel from a shared normalized input, then summing their outputs into the residual stream.

I hypothesized that applying PaLM-style parallel blocks to NanoGPT would: (1) reduce training time by enabling better GPU kernel fusion and eliminating one LayerNorm operation per block, and (2) improve gradient flow by halving effective path length to L. After verifying this modification was not present in the NanoGPT speedrun records, I implemented and tested it through a three-stage experimental pipeline.

2 Methodology

2.1 Implementation

The modification replaces the sequential `Block.forward()` with a parallel formulation:

$$y = x + \frac{1}{\sqrt{2}}\big(\mathrm{Attn}(\mathrm{LN}(x)) + \mathrm{MLP}(\mathrm{LN}(x))\big).$$

The $1/\sqrt{2}$ scaling factor is critical—it dampens the combined update to match the variance dynamics of the original sequential block, preventing optimizer instability. A single RMSNorm is computed once and shared by both branches, eliminating one normalization per block.

2.2 Three-Stage Experimental Design

Stage A (Screening): Quick validation on 1×L40 GPU with abbreviated training. Three modifications tested: PaLM-Parallel, Depth Warming (gradual block activation), and Curriculum Learning (progressive sequence length). Goal: identify promising candidates.

Stage B (Validation): Top performer(s) from Stage A tested on 4×A100 GPUs with longer training to verify speedup generalizes to more compute.

Stage C (Final): Full-scale comparison on 8×H100 GPUs with 5100 iterations. Three runs per condition for statistical analysis. Originally planned five runs, but CUDA initialization errors consumed compute credits.

3 Results

3.1 Stage A Screening Results

Table 1: Stage A screening on 1×L40 GPU. PaLM-Parallel showed 5% speedup with comparable loss.

Modification	Validation Loss	Time (s)	Outcome
Baseline	3.8965	3108	Reference
PaLM-Parallel	3.9007	2952 (−5.0%)	✓ Promising speedup
Depth Warming	3.9244	3778 (+21.6%)	× Slowdown, rejected
Curriculum Learning	—	—	× Implementation failed

3.2 Stage C Final Results (8×H100, $n = 3$ per condition)

Table 2: Stage C results. Time improvement significant ($p = 0.046$), validation loss regression significant ($p < 0.001$).

Metric	Baseline	PaLM-Parallel	Change
Validation Loss	3.3005 ± 0.001	3.3348 ± 0.002	+1.04% (worse)
Training Time (s)	1588 ± 12	1546 ± 18	−2.66% (faster)
Peak Memory (MiB)	30,602	28,295	−7.5% (less)

4 Conclusion

The PaLM-style parallel transformer modification achieved a statistically significant 2.66% training speedup and 7.5% memory reduction, but at the cost of a 1.04% validation loss regression. This represents a speed–quality tradeoff similar to Part 1 of this project.

The modification is suitable for scenarios prioritizing throughput over final model quality, such as hyperparameter sweeps or rapid prototyping. The memory savings may enable larger batch sizes on memory-constrained systems. However, for accuracy-critical applications, the baseline sequential architecture remains preferable.

This work validates that architectural parallelization can yield meaningful speedups even in highly-optimized training pipelines, while demonstrating the importance of rigorous multi-stage validation before committing expensive H100 compute to full-scale experiments.

A Experimental Journey

Phase 1: Ideation and Implementation

I began by brainstorming modifications with potential for training speedup, focusing on ideas not already present in the NanoGPT speedrun records. Three candidates emerged:

- **PaLM-Parallel:** Based on Google's PaLM paper, replacing sequential attention→MLP with parallel computation. Verified not present in speedrun records.
- **Depth Warming:** Start with fewer active transformer blocks (e.g., 6 of 12), gradually enabling deeper layers with stochastic depth. Hypothesis: early training learns shallow features; reducing active depth reduces FLOPs per step.
- **Curriculum Learning:** Progressive sequence length starting at 256 tokens, increasing based on loss stability. Inspired by "train long, think short" philosophy for sample-efficient early training.

Phase 2: Stage A Screening

Set up experiment infrastructure on a single L40 GPU ($0.99/hr) for rapid iteration. Curriculum Learning implementation encountered bugs in the data loader modification that prevented training from completing; abandoned due to time constraints. Depth Warming ran successfully but produced a 21.6% slowdown; the overhead of managing dynamic depth scheduling and stochastic block skipping exceeded any computational savings from reduced effective depth. PaLM-Parallel showed a promising 5% speedup with minimal loss increase, advancing to Stage B.

Phase 3: Stage B Validation

Moved PaLM-Parallel to 4×A100 80GB PCIe GPUs for longer training (targeting `val_loss < 3.35`). Results: 2 successful runs achieving val loss 3.3955 ± 0.0002 in $2598s \pm 45s$. One run failed due to CUDA errors (see Phase 4). The speedup pattern held with more compute, validating progression to Stage C.

Phase 4: CUDA Initialization Errors

During Stage C setup on 8×H100 pods, encountered persistent CUDA initialization failures:

```
CUDA initialization: CUDA unknown error - this may be due to an incorrectly
set up environment, e.g. changing env variable CUDA_VISIBLE_DEVICES after program
start.
```

Root cause: The experiment runner script imported PyTorch before spawning distributed training processes. Once CUDA initializes in a corrupted state in the parent process, child processes inherit the broken context;causing all 8 GPU workers to fail. Resolution: set `CUDA_VISIBLE_DEVICES=0,1,2,3,4,5,6,7` before any Python import, restart shell with `exec bash` to clear corrupted state. In containerized environments like RunPod, full pod restart required if corruption persists at driver level. This consumed $20 in failed restarts and forced reduction from planned 5 runs to 3 runs per condition.

Phase 5: Stage C Final Experiments

Successfully ran 3 baseline and 3 PaLM-Parallel experiments on 8×H100 SXM for 5100 iterations each. Results showed the modification achieved speedup but with slight accuracy degradation;a tradeoff pattern consistent with Part 1 findings. Total Stage C compute time: approximately 4.5 hours of 8×H100 ($21.50/hr $\times$ 4.5hr $\approx$ $97).

B GPU Usage and Compute Allocation

Hardware Configurations

Table 3: Hardware configurations across experimental stages.

Stage	Hardware	Purpose	Runs
A	1×L40 (48GB)	Quick screening	1–2 per variant
B	4×A100 80GB PCIe	Validation	2–3 per variant
C	8×H100 SXM	Final comparison	3 per condition

Runpod Billing Summary

Table 4: Runpod billing for Part 2 experiments. Majority spent on Stage C H100 runs.

Date	GPU Cost	Total Cost
Dec 7, 2025	\$8.11	\$8.20
Dec 8, 2025	\$98.33	\$98.45
Total	\$106.44	\$106.65

C Complete Statistical Analysis

Stage C Individual Run Data

Table 5: Raw data from all Stage C runs.

Run	Val Loss	Train Loss	Time (s)	Memory (MiB)
Baseline 1	3.2996	3.6320	1574.8	30,602
Baseline 2	3.3016	3.6400	1598.6	30,602
Baseline 3	3.3004	3.6380	1590.6	30,602
PaLM 1	3.3336	3.6666	1532.5	28,295
PaLM 2	3.3352	3.6720	1568.9	28,295
PaLM 3	3.3356	3.6710	1535.7	28,295

Statistical Tests

Validation Loss: Baseline mean 3.3005, PaLM mean 3.3348, difference +0.0343 (+1.04%). Independent t-test: $t = -37.7$, $p < 0.001$. Cohen's $d = -30.8$ (large effect). Permutation test $p = 0.103$ (non-significant with $n = 3$, reflecting small sample size).

Training Time: Baseline mean 1588.0s, PaLM mean 1545.7s, difference −42.3s (−2.66%). Independent t-test: $t = 3.11$, $p = 0.046$. Cohen's $d = 2.54$ (large effect).

Peak Memory: Baseline 30,602 MiB, PaLM 28,295 MiB, difference −2,307 MiB (−7.54%). Zero variance within conditions indicates deterministic memory allocation.

Normality & Variance: Shapiro-Wilk tests: Baseline `val_loss` $p = 0.235$, PaLM `val_loss` $p = 0.503$ (both approximately normal). Levene's test for equal variances: $p = 0.622$ (homogeneous).

Figures

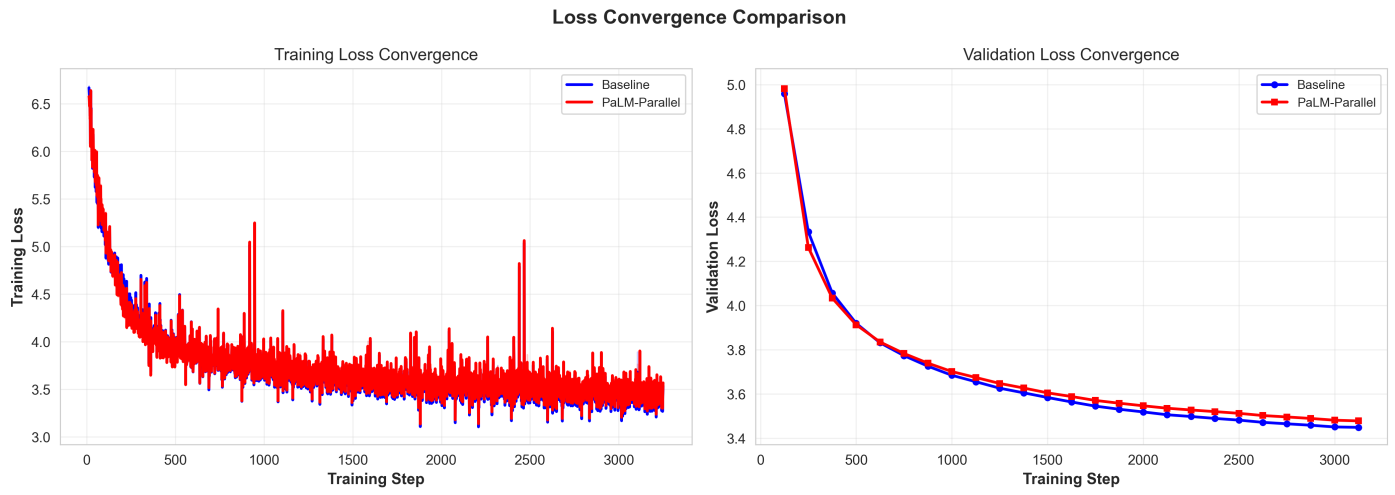

Figure 1: Loss convergence comparison for training (left) and validation (right) on Stage C runs. Baseline (blue) vs. PaLM-Parallel (red).

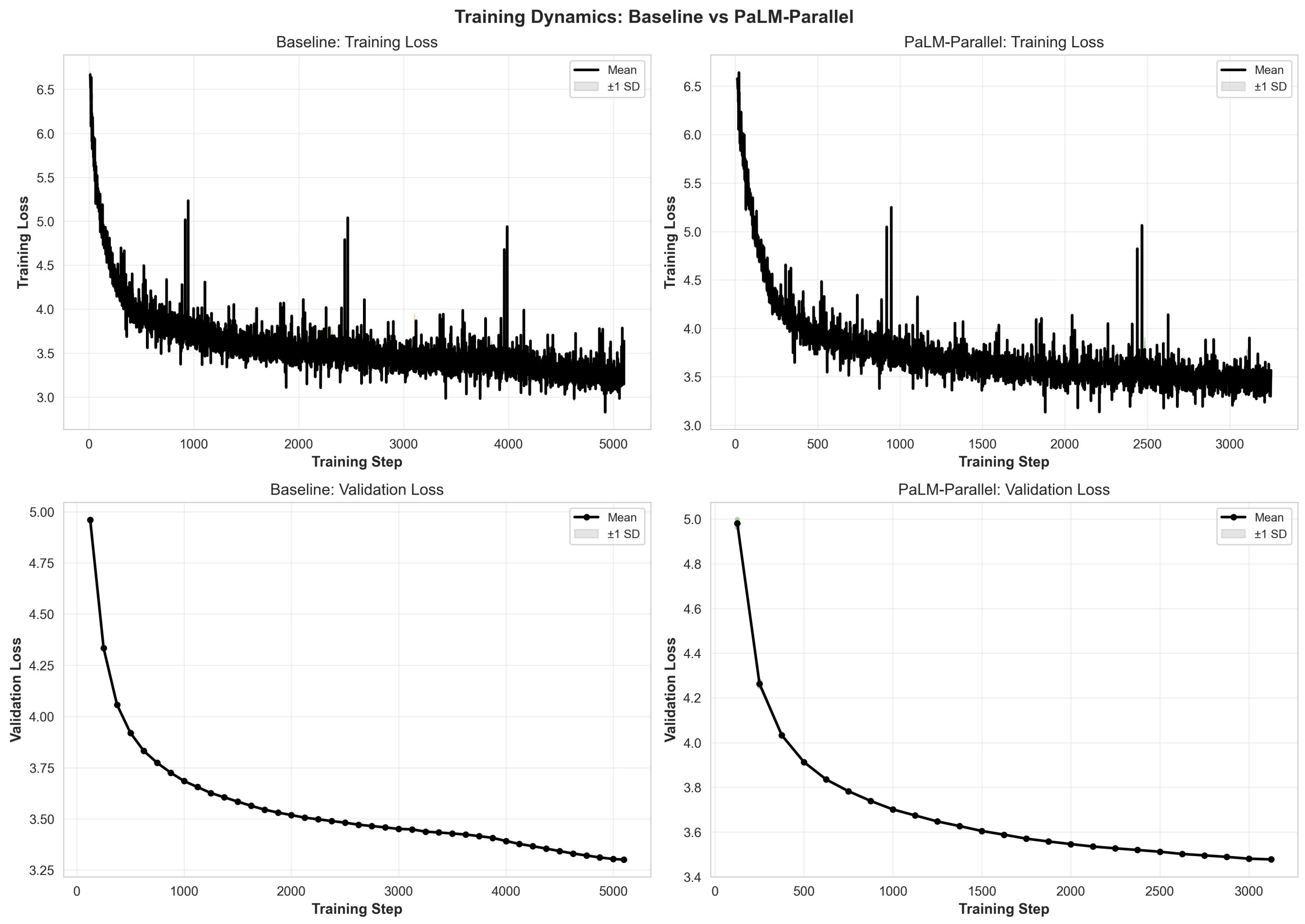

Figure 2: Training dynamics for baseline (left column) and PaLM-Parallel (right column): training loss (top) and validation loss (bottom) with mean and ± 1 standard deviation bands across seeds.

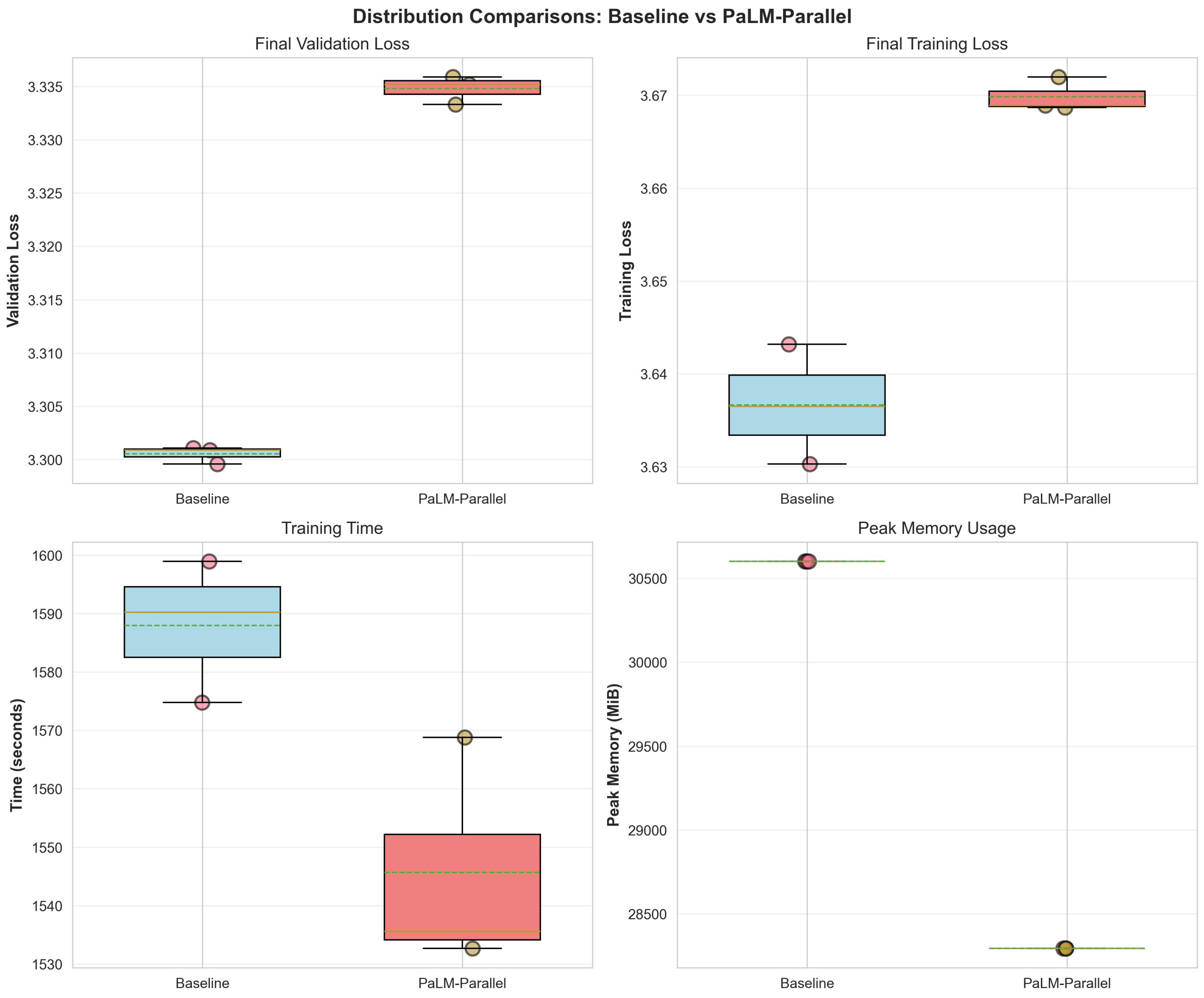

Figure 3: Distribution comparisons across Stage C runs: final validation loss, final training loss, training time, and peak memory for Baseline vs. PaLM-Parallel. Boxplots show median, mean (dashed line), and individual runs.

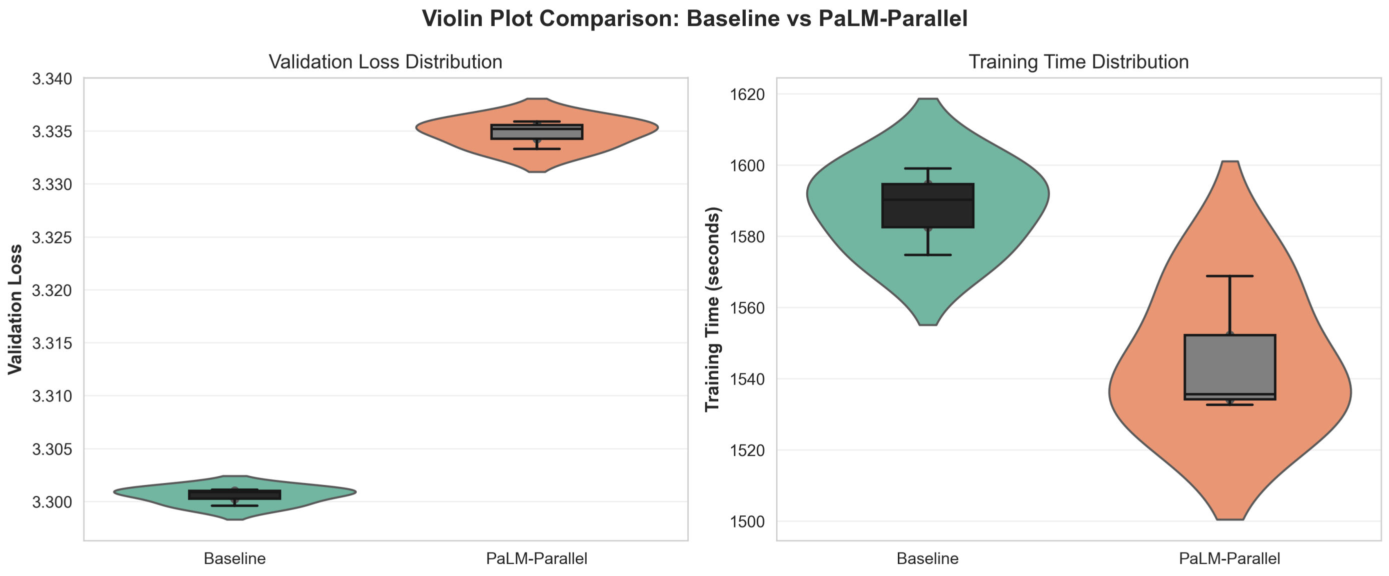

Figure 4: Violin plot comparison of validation loss (left) and training time (right) for Baseline vs. PaLM-Parallel, highlighting the speed–quality tradeoff.

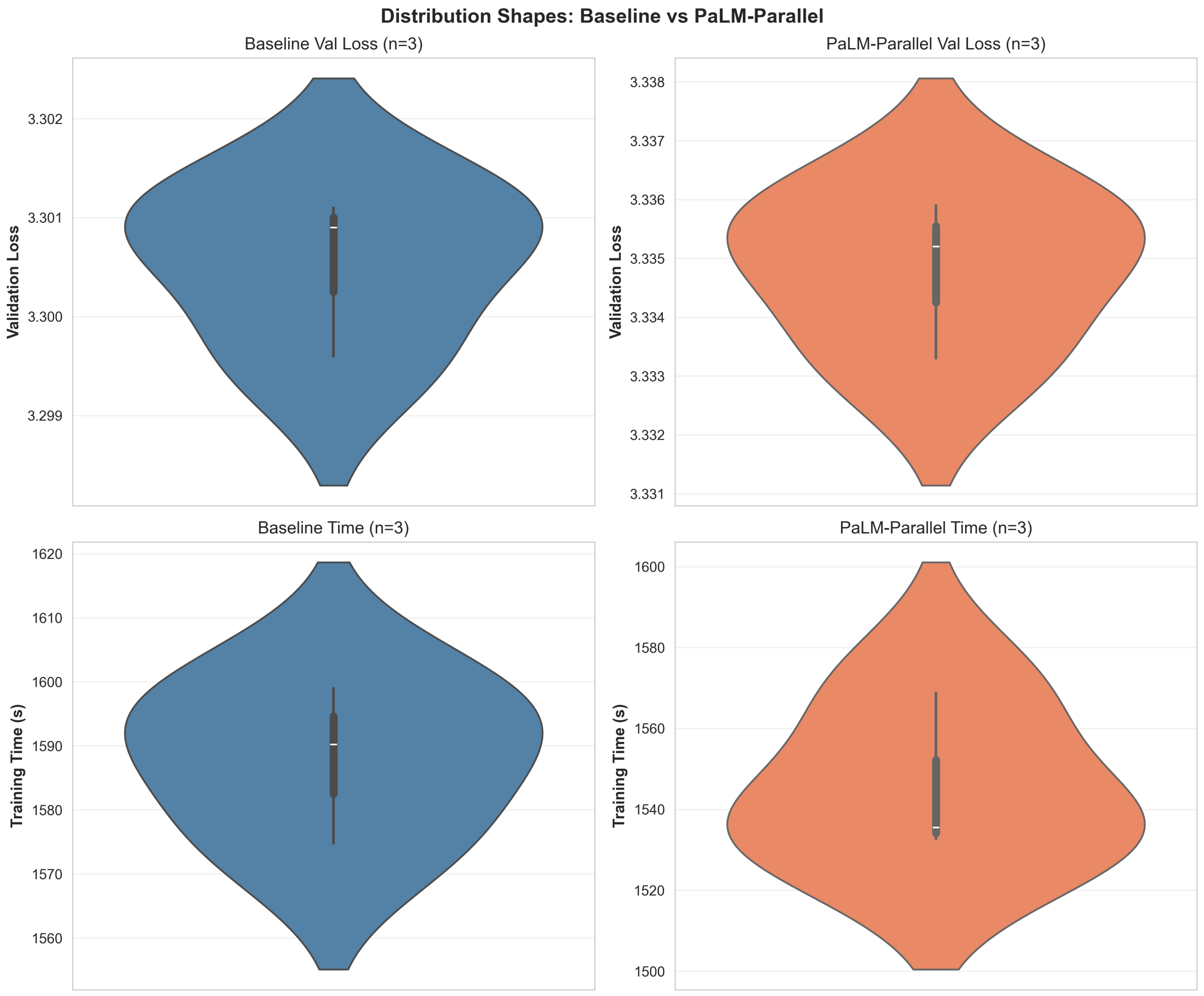

Figure 5: Individual violin plots (2×2 grid) showing distribution shapes for validation loss and training time for each condition separately (Baseline vs. PaLM-Parallel, $n = 3$ per condition).

D Ablation Studies

Depth Warming (Rejected in Stage A)

Concept: Start training with only first N transformer blocks active (e.g., 6 of 12), gradually enabling deeper blocks over training with light stochastic depth (randomly skipping some late blocks) during warmup. By end of training, always use full 12-block stack. Hypothesis: Early training learns shallow features; deeper layers underutilized but fully computed. Reducing active depth should reduce FLOPs per step while regularization from stochastic depth improves sample efficiency.

Implementation: Added per-block `layer_enabled(step)` function returning 0/1. Schedule: 0–1/3 of steps use depth=6 with $p_{\text{skip}} = 0.5$ for layers >6; 1/3–2/3 use depth=9 with $p_{\text{skip}} = 0.25$ for layers >9; 2/3–1 use full depth=12 with no stochastic skipping.

Results: Stage A showed `val_loss` 3.9244 (comparable to baseline 3.8965) but training time 3778s vs 3108s;a 21.6% slowdown. The overhead of managing dynamic depth scheduling exceeded any FLOP savings from reduced effective depth. Rejected before Stage B.

Curriculum Learning (Implementation Failed)

Concept: Training curriculum starting with short sequences (256 tokens), dynamically increasing effective sequence length based on loss stability rather than fixed schedule. Early training on short sequences allows model to quickly learn local dependencies with larger effective batch size; as loss plateaus, increasing sequence length forces integration of longer-range context.

Results: Implementation encountered bugs in the data loader modification that prevented training from completing. Due to time constraints (Dec 9 deadline), abandoned without successful runs. Future work could revisit with more debugging time;the theoretical motivation remains sound.

PaLM-Parallel Scaling Factor

The $1/\sqrt{2}$ scaling factor is critical; it dampens the combined update to match the variance dynamics of the original sequential block, preventing optimizer instability. The scaling was not ablated due to prior work establishing its necessity for stable training;ablating it would likely cause divergence rather than provide useful signal.

E Reproducibility

Git Repository

Repository: https://github.com/alainwelliver/deep-learning-speedrun-project

Key commits:

- Stage A baseline: `a8ac360eb39e396a9452c517c223bdcf70d6259f`
- Stage A PaLM: `cd40143b96a3a1c807d2c224ed9f844c72af324f`
- Stage A Depth Warming: `7271ad3faae6abb942f7c413fda978bf82d87138`
- Stage B validation: `2ac3a57e8aff853f2c37d5c696839c4737337296`

Reproduction Commands

Baseline (8×H100):

```
torchrun --standalone --nproc_per_node=8 experiments/train_gpt.py
```

PaLM-Parallel (8×H100):

```
torchrun --standalone --nproc_per_node=8 experiments/train_gpt_palm.py
```

Environment

PyTorch: 2.8.0+cu128
CUDA: 12.8
Python: 3.12.3
Platform: Linux (Ubuntu, glibc 2.39)
Cloud: Runpod.io

F FAQ

Q: Did you verify this modification wasn't already tried in the speedrun?

A: Yes. I reviewed the NanoGPT speedrun records table linked in the project description, the modded-nanogpt repository commit history and pull requests, and the LessWrong "20% in 3 months" writeup documenting 45+ attempted techniques. PaLM-style parallel attention/MLP blocks were not present in any prior submissions.

Q: Why only 3 runs instead of the planned 5?

A: CUDA initialization errors during Stage C setup required multiple pod restarts, consuming approximately $20 in compute credits. The reduction from 5 to 3 runs was necessary to stay within budget while still providing statistically meaningful comparison. With $n = 3$, we achieved $p = 0.046$ for the time improvement, meeting the $p < 0.05$ significance threshold.

Q: Is the speedup worth the accuracy loss?

A: It depends on the use case. For hyperparameter search, rapid prototyping, or memory-constrained scenarios, the 2.66% speedup and 7.5% memory savings may justify the 1.04% loss increase. For final model training where accuracy is paramount (e.g., targeting the ≤ 3.28 `val_loss` threshold), the baseline is preferred.

Q: Do you have training curve plots?

A: Yes. Training curves, loss distributions, and violin plots comparing baseline vs PaLM-Parallel are included as attached figures in Appendix C. The curves show both modifications track similarly through early training but diverge slightly in final convergence, with PaLM-Parallel exhibiting marginally higher loss variance.