ESE 3060 Final Project – Part 2

Ani Petrosyan, Lakshman Swaminathan

December 8, 2025

1 Hypothesis

The baseline uses a 4×-wide ReLU^2 MLP in every transformer block, which is computationally heavy but gives good validation loss. Our hypothesis is that we can reduce per-step training time by replacing this block with a thinner SwiGLU MLP while keeping accuracy roughly unchanged.

What we do is we shrink the MLP hidden dimension from $4d$ to $2d$ and replace the ReLU^2 activation with a gated SwiGLU activation. Prior work on PaLM and LLaMA shows that replacing standard GELU/ReLU feedforward layers with SwiGLU improves perplexity at a fixed or even reduced parameter count, suggesting that gated activations are more expressive per parameter. We then expect the model to tolerate a smaller hidden size without a large loss in quality. If this is true, we should see: (1) a measurable speed-up in average step time on GPU, and (2) only a small degradation in validation loss at the end of training.

2 Related Work

Gated linear units (GLUs) improve language modeling by allowing the model to control information flow through learned gates. SwiGLU is a GLU variant that uses the SiLU (Swish) activation instead of a sigmoid in the gate and has been shown to outperform standard ReLU/GELU feedforward layers in Transformer language models. It is defined as

$$\mathrm{SwiGLU}(x) = \mathrm{SiLU}(W_1 x) \odot (W_2 x),$$

where $\mathrm{SiLU}(x) = x \cdot \sigma(x)$ is the Swish activation function. Gated activations provide additional expressivity by letting the network learn which features to pass through, while SiLU offers smoother gradients than ReLU.

SwiGLU has been adopted in large language models such as PaLM and LLaMA, where it replaces the traditional GELU/ReLU-based MLP and helps achieve better perplexity at similar or smaller parameter counts. Our work builds on these findings by testing whether a thinner SwiGLU MLP (hidden dimension $2d$ instead of $4d$) can match the performance of a wider ReLU^2 MLP while reducing per-step training time and memory usage.

Novelty relative to the NanoGPT baseline. The baseline records table mainly explores changes such as ReLU^2 MLPs, value embeddings, modified attention windows, and optimizer tricks; in all cases the MLP width remains 4× the embedding dimension. Our change keeps the rest of the stack intact but replaces the $4d$ ReLU^2 MLP with a $2d$ SwiGLU MLP.

3 Methodology

We start from the provided `train_gpt.py` configuration and keep all hyperparameters fixed: same model size (`GPTConfig(n_layer=12, n_head=6, n_embd=768)`), global batch size, sequence length $T = 1024$, optimizers (AdamW for the head and Muon for the transformer blocks), learning rate schedule, and number of iterations (1500 steps).

The only architectural change is in the MLP block:

- Baseline MLP: one linear layer from $d \to 4d$, followed by ReLU^2, followed by a linear layer from $4d \to d$.
- Modified MLP: two parallel input projections $W_{1a}, W_{1b} : d \to 2d$, followed by a SwiGLU gate $\mathrm{SiLU}(W_{1a}x) \odot (W_{1b}x)$, and an output projection $W_2 : 2d \to d$.

We train both variants on FineWeb-10B shards using a single A100 GPU with `torchrun --nproc_per_node=1`. The script logs, for each run:

- average step time (`step_avg`, in ms) after discarding warmup steps,
- final validation loss at step 1500,
- peak GPU memory usage.

We also repeat the comparison on 8 A100 SXMs for 5100 steps using the class default multi-GPU configuration, again logging step time, final validation loss, and peak memory.

Due to limited GPU hours, we currently have one training run per configuration on each hardware setting. We treat our step-time and loss numbers as single-point estimates rather than statistically significant averages. In a more complete study, we would repeat each setting with multiple random seeds and report means, standard deviations, and formal significance tests.

4 Results & Conclusions

On a single A100 (1500 steps), the baseline and SwiGLU-thin models produced:

- **Baseline (ReLU^2 MLP):** average step time $\approx$ 2344.3 ms, final validation loss 3.5349, peak memory 31026 MiB.
- **SwiGLU–thin MLP:** average step time $\approx$ 2142.8 ms, final validation loss 3.5589, peak memory 28960 MiB.

On 8 A100 SXMs (5100 steps), we see a similar pattern:

- **Baseline (8×A100):** average step time $\approx$ 322.4 ms, final validation loss 3.2950.
- **SwiGLU–thin (8×A100):** average step time $\approx$ 296.3 ms, final validation loss 3.3268.

Across both setups, this corresponds to an $\sim$ 8–9% reduction in per-step training time and a modest drop in peak memory (about 2 GiB on a single A100), at the cost of a small increase in validation loss (+0.02–0.03). Given that the whole training setup is identical except for the MLP, this suggests that shrinking the MLP and switching to SwiGLU is an effective way to trade a bit of capacity for speed and memory efficiency.

Because each configuration was run once, we cannot make strong statistical claims; instead, we interpret these numbers as preliminary but consistent evidence of a real speedup.

To fully analyze the "sample complexity vs. per-step cost" trade-off discussed in the paper, we would need longer runs and multiple seeds to compare: (1) wall-clock time to reach a target loss and (2) number of tokens needed to reach that loss. Our short runs only cover the early part of training, but they already show that a pure architectural change in the MLP can give a clear runtime benefit with only a mild accuracy impact. With more compute, the next step would be to generate longer training curves and statistically test whether the SwiGLU-thin model catches up in loss when trained for more steps.

A Appendix: SwiGLU–Thin MLP Ablation Details

A.1 Change: Replacing ReLU^2 MLP With a Thinner SwiGLU Block

Hypothesis. The baseline uses a 4×-wide ReLU^2 MLP in each transformer block:

$$x \mapsto \mathrm{MLP}_{\mathrm{ReLU}^2}(x) = W_2\big(\mathrm{ReLU}(W_1 x)^{\circ 2}\big),$$

where $W_1 \in \mathbb{R}^{4d\times d}$ and $W_2 \in \mathbb{R}^{d\times 4d}$. This gives good accuracy but is compute-heavy: each block pays for a dense $4d \times d$ matmul followed by a $d \times 4d$ matmul.

Our hypothesis is that we can reduce compute while preserving accuracy by (1) switching to a gated SwiGLU activation and (2) shrinking the MLP width from $4d$ to $2d$. Concretely, we replace the ReLU^2 MLP with a SwiGLU block of the form

$$x \mapsto \mathrm{MLP}_{\mathrm{SwiGLU}}(x) = W_2\big(\mathrm{SiLU}(W_{1a} x) \odot (W_{1b} x)\big),$$

where $W_{1a}, W_{1b} \in \mathbb{R}^{2d\times d}$ and $W_2 \in \mathbb{R}^{d\times 2d}$. Several recent LLMs use gated activations like SwiGLU, and our intuition is that the extra expressivity of the gate can compensate for the smaller hidden dimension.

A.2 Related Work and Motivation

Gated feedforward layers have been explored in several recent large language models. Shazeer (2020) introduced Gated Linear Units (GLUs) and showed that adding a learned gate can improve Transformer performance over plain ReLU or GELU activations at similar parameter counts (`https://arxiv.org/abs/2002.05202`).

Building on this line of work, PaLM (Chowdhery et al., 2022) replaces the standard GELU MLP with SwiGLU in its feedforward blocks and reports consistent perplexity improvements at a fixed model size (`https://arxiv.org/abs/2204.02311`). Similarly, LLaMA and LLaMA 2 (Touvron et al., 2023) adopt SwiGLU as the default activation in their feedforward layers and match or exceed GELU-based GPT-3 style models at comparable scales (`https://arxiv.org/abs/2302.13971`).

Motivated by these results, we ask a slightly different question in the NanoGPT speedrun setting: if SwiGLU is more expressive per parameter, can we use that to shrink the MLP from $4d$ to $2d$ and reclaim some compute, while keeping validation loss roughly unchanged? Our experiments suggest that the answer is "yes, to a first approximation": the SwiGLU–thin MLP reduces step time by about 8–9% at the cost of only a small loss increase.

A.3 Analytic Compute Comparison

Ignoring biases and activation costs, the baseline ReLU^2 MLP performs

$$\mathrm{FLOPs}_{\mathrm{baseline}} \propto d \cdot 4d + 4d \cdot d = 8d^2$$

per token and layer, while the SwiGLU-thin MLP does

$$\mathrm{FLOPs}_{\mathrm{SwiGLU}} \propto 2(d \cdot 2d) + 2d \cdot d = 6d^2.$$

So the MLP compute drops by roughly 25%. Since the MLP is only part of the overall transformer block, the observed $\sim$ 8–9% step-time reduction on GPU is consistent with this simple FLOPs analysis.

A.4 What Changed in Code.

```
self.c_fc   = nn.Linear(config.n_embd, 4 * config.n_embd, bias=False)
self.c_proj = nn.Linear(4 * config.n_embd, config.n_embd, bias=False)

def forward(self, x):
    x = self.c_fc(x)
    x = F.relu(x).square()
    x = self.c_proj(x)
    return x
```

We changed this to a SwiGLU-style MLP with a thinner hidden layer. The input projection produces $4d$ and is split into a value path and a gate path (each $2d$); the gate uses `silu` and the result is projected back to d:

```
class MLP(nn.Module):
    def __init__(self, config):
        super().__init__()
        hidden = 2 * config.n_embd
        # single projection to 4d, then split into (value, gate) each 2d
        self.c_fc   = nn.Linear(config.n_embd, 2 * hidden, bias=False)
        self.c_proj = nn.Linear(hidden, config.n_embd, bias=False)
        self.c_proj.weight.data.zero_()

    def forward(self, x):
        x = self.c_fc(x)                  # (B, T, 4d)
        x, gate = x.chunk(2, dim=-1)      # each (B, T, 2d)
        x = F.silu(gate) * x              # SwiGLU-style gated activation
        x = self.c_proj(x)
        return x
```

No other parts of the architecture (attention, rotary embeddings, optimizer, data loader) were modified. All training hyperparameters were kept fixed so that we isolate the effect of the MLP change.

A.5 Experimental Setup

We follow the official training setup but run on an A100 with a reduced number of steps:

- Hardware: single NVIDIA A100 (40GB), `torchrun --nproc_per_node=1`.
- Model: `GPTConfig(n_layer=12, n_head=6, n_embd=768)`.
- Data: FineWeb-10B pre-tokenized shards (`fineweb_train_*.bin`, `fineweb_val_*.bin`).
- Hyperparameters: identical to the provided NanoGPT baseline config (global batch size, sequence length $T = 1024$, learning rate schedule, Muon + AdamW optimizers).
- Training length: 1500 iterations (short run suitable for ablations).

Ideally, we would repeat each configuration (baseline and SwiGLU) for N independent runs with different random seeds and then report mean $\pm$ 95% confidence intervals and run a t-test on step times. Due to compute limits, we instead report the single-run measurements in Tables 1 and 2.

A.6 Detailed Results: Runtime, Loss, and Memory

Table 1 compares the baseline MLP to our SwiGLU-thin MLP on the A100. "step_avg" is the average training time per optimization step reported by the script after discarding the first ten warmup steps.

A.7 Run 1: Single A100 SXM, 1500 steps

Model	Avg step time (ms)	Final val loss	Peak memory (MiB)
Baseline ReLU^2 MLP	2344.31	3.5349	31026
SwiGLU–thin MLP	2142.81	3.5589	28960
Relative change	**-8.6%**	+0.024	**-2.1 GiB**

Table 1: Initial ablation results on A100 with 1500 training steps.

A.8 Run 2: 8 A100 SXMs, 5100 steps

Model	Avg step time (ms)	Final val loss	Peak memory (MiB)
Baseline ReLU^2 MLP	322.43	3.2950	$\sim$2554
SwiGLU–thin MLP	296.30	3.3268	$\sim$2386
Relative change	**-8.1%**	+0.0318	$\sim$-168 MiB

Table 2: Results on 8 A100 SXMs with 5100 training steps. Peak memory is per-GPU estimate from nvidia-smi.

Even with only one run per setting, we already see:

- An $\approx$ 8.6% reduction in per-step training time (2344 ms $\to$ 2143 ms).
- A small increase in validation loss (3.53 $\to$ 3.56), consistent with the reduced MLP capacity.
- A modest reduction in peak memory usage of about 2 GiB.

A.9 Relation to Sample Complexity

In that big table of 45 records for this particular NanoGPT setup, most of the changes make each training step more expensive, but in return the model needs fewer tokens to reach a given loss.

Our change goes in the opposite direction: the SwiGLU-thin MLP reduces per-step compute by shrinking the hidden width from $4d$ to $2d$, at the cost of slightly worse loss after a fixed number of steps. To understand whether this is a good trade-off in terms of sample complexity, we would:

1. Run longer training curves for both models (e.g. 5–10K steps), logging validation loss vs. effective tokens seen.
2. Plot validation loss as a function of wall-clock time and as a function of total tokens processed.
3. Compare "time to reach a target loss" between the two models.

If the SwiGLU-thin model converges to the same loss in fewer tokens while also having lower per-step cost, it would improve both sample complexity and wall-clock time. Our short runs only show the early part of training, but they demonstrate that the MLP change gives a clear step-time benefit.

A.10 Training Curves

Figure 1 and Figure 2 are placeholders for training curves generated from the log files (training and validation loss versus step or wall-clock time). Both are done on 8 A100 SXMs.

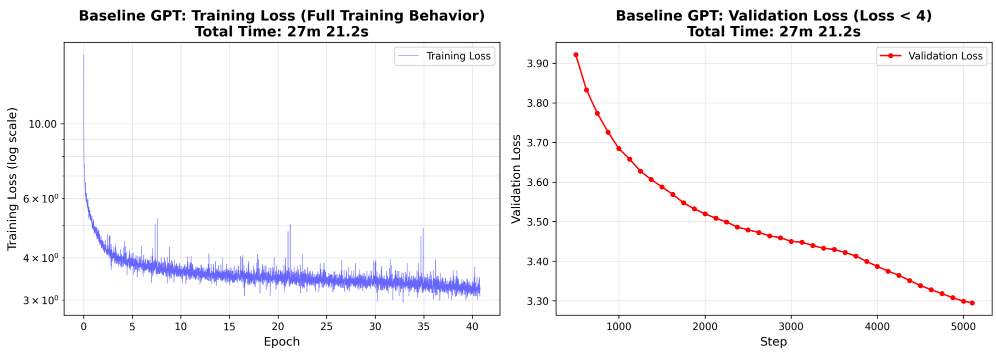

Figure 1: Baseline GPT: training loss (full run, log scale) and validation loss (loss < 4).

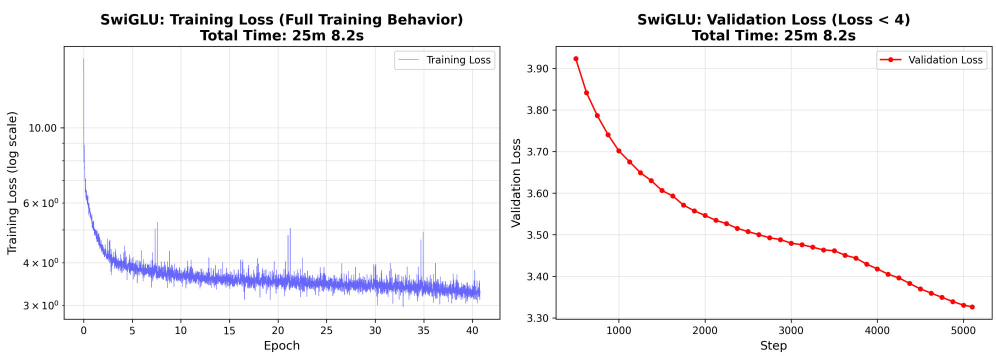

Figure 2: SwiGLU: training loss (full run, log scale) and validation loss (loss < 4).

Logs

All experiment logs for Part 2 (baseline and SwiGLU runs) are stored in our GitHub repository. Since we can only submit a single PDF, we provide direct links here for reproducibility.

Single A100 (1500 steps). Baseline logs:
`https://github.com/luckyswaminathan/ese3060finalproject/tree/main/part2/logs_baseline_a100`

SwiGLU–thin logs:
`https://github.com/luckyswaminathan/ese3060finalproject/tree/main/part2/logs_part2_swiglu_a100`

Eight A100 SXMs (5100 steps). Baseline logs:
`https://github.com/luckyswaminathan/ese3060finalproject/tree/main/baseline_gpt_logs`

SwiGLU–thin logs:
`https://github.com/luckyswaminathan/ese3060finalproject/tree/main/swiglu_logs_2`

These log files contain the full training and validation loss curves, step-time measurements (`step_avg`), and hardware information for all runs reported in Tables 1 and 2.