Parallelizing GPT-2 — "It works on my GPU"

Jack "Siggy" Sigler
Colorado School of Mines
Golden, CO, USA

Sam Abderholden
Colorado School of Mines
Golden, CO, USA

Dawson Matthews
Colorado School of Mines
Golden, CO, USA

1 INTRODUCTION OF THE PAPER AND THE TARGETED PROBLEM

In this project, we discuss the techniques behind parallelizing the GPT-2 machine learning architecture. Generative pre-trained transformer (GPT) models are the machine learning models that power modern large language models (LLMs) and were the breakthrough that sparked the worldwide obsession with language models. One of the challenges in machine learning is finding ways to reduce the time it takes to train models. One of the reasons the GPT model was so popular was due to how much of its process could be parallelized. Parallelizing GPT is what enables modern language models to be trained and used by anyone with fast results.

In our implementation, we will convert the forward pass portion of GPT-2 training from a serial Python script into a parallel C++ program utilizing CUDA to harness GPU parallelization. The goal of our project is to introduce as much speedup as possible through parallelization. We attempt to reach this goal by implementing batched processing of tokens, tiled matrix multiplication, floating point precision reduction, and other techniques to be discussed in this paper.

Due to the highly parallelizable nature of GPT-2, our results show extremely high speedup of the parallel CUDA approach over the serial approach. We achieved speedup factors of over 10,000 when comparing against our serial Python implementation, and over 500 when comparing against our serial C++ implementation. Our results showcase why GPUs have become so important in the machine learning industry, and why they will continue to be important for the foreseeable future.

2 SERIAL IMPLEMENTATION AND ANALYSIS

Problem Background

In this project, we implement an optimized CUDA version of the forward pass of Andrej Karpathy's *microgpt* [1]. *Microgpt* is a minimal, dependency-free Python implementation of a GPT-style model. It is based on the GPT-2 architecture [2], with a few changes made for simplicity. Both GPT-2 and Karpathy's *microgpt* implement:

- Decoder-only Transformer architecture
- Token + positional embeddings
- Stacked Transformer blocks
- Causal (masked) multi-head self-attention
- Residual connections around attention and MLP sublayers
- Feedforward MLP with expansion and projection
- Autoregressive next-token prediction

Since *microgpt* was intended to showcase the GPT-2 architecture in a single unoptimized Python file, the proposed model has only 4,192 parameters compared to GPT-2's 1.6 billion parameters. In addition to the reduced model size, *microgpt* introduces several simplifications, including the use of RMSNorm instead of LayerNorm, removal of bias terms, and ReLU in place of GELU [1].

In order to keep the scope of our implementation to 10 hours, we focus only on speeding up the forward pass of *microgpt*. Since we believed that comparing a CUDA C++ version to a Python version would introduce speedups due to both our parallelization choices and language differences (C++ vs Python), we first reimplemented *microgpt* in C++, preserving the following:

- Identical layer structure
- Same ordering of operations (norm → attention → residual → norm → MLP → residual)
- Same KV caching mechanism
- Same attention computation (scaled dot-product + softmax)
- Same MLP structure (`fc1` → activation → `fc2`)

Our C++ version is nearly identical, with differences limited to implementation details such as replacing dynamic Python data structures with pre-allocated caches, removing autograd in favor of raw floating-point computation, and expressing all operations as explicit loop-based computations over contiguous memory.

C++ Serial Baseline

Our implementation follows the training setup used in *microgpt*, which operates on a character-level dataset consisting of a set of names $\mathcal{D} = \{s^{(1)}, s^{(2)}, \ldots, s^{(N)}\}$. Each name $s^{(k)} \in \mathcal{D}$ is treated as a sequence of tokens, $s^{(k)} = [t_1, t_2, \ldots, t_T]$, where T denotes the sequence length. The model is trained autoregressively to predict the next token given all previous tokens. Our C++ implementation preserves this setup exactly, performing the same forward-pass computation and next-token prediction task. The prompt used to generate the C++ baseline can be found under **Appendix B**

Given a tokenized sequence (i.e., a single name) $[t_1, t_2, \ldots, t_T]$, the model computes a sequence of logits $[y_1, y_2, \ldots, y_{T-1}]$, where each y_i represents the model's prediction of token t_{i+1} conditioned on the prefix $[t_1, \ldots, t_i]$. In the baseline implementation, this computation is performed sequentially, iterating over the dataset one name at a time and, within each name, one token at a time, with each position attending over all previous tokens in the sequence. A high level pseudocode of our serial implentation is shown under **Algorithm 1**.

We denote the embedding dimension as d, the number of Transformer layers as L, and the number of attention heads as H. For each token position i, the model produces a hidden representation $x_i \in \mathbb{R}^d$, which is transformed through attention and MLP operations across all layers.

The computational cost of the forward pass over N sequences is $O\Big(N \cdot L \cdot (C_{\text{attn}} + C_{\text{mlp}})\Big)$, where $C_{\text{attn}} = O(T^2 \cdot d)$ and $C_{\text{mlp}} = O(T \cdot d^2)$. Since the names have approximately constant sequence lengths, we can simplify this to $O(N \cdot L \cdot d^2)$.

Algorithm 1 High-Level Forward Pass

loss ← 0
for each sequence $s^{(k)} \in \mathcal{D}$ **do**
  for $i = 1$ to $T - 1$ **do**
    $x_i \leftarrow \text{TokenEmbedding}(t_i) + \text{PositionalEmbedding}(i)$
    $x_i \leftarrow \text{RMSNorm}(x_i)$
    for $\ell = 1$ to L **do**
      $x_i \leftarrow x_i + \text{Attention}(\text{RMSNorm}(x_i), \{k_j, v_j\}_{j \le i})$
      $x_i \leftarrow x_i + \text{MLP}(\text{RMSNorm}(x_i))$
    end for
    $y_i \leftarrow \text{Linear}(x_i)$
    $\text{loss} \leftarrow \text{loss} - \log(\text{Softmax}(y_i)[t_{i+1}])$
  end for
end for
loss ← loss/total_tokens

Using the *perf time* command we were able to get high level timing characteristics of the C++ baseline program, especially how much time our C++ baseline was performing OS overhead versus useful compute work. With nearly zero system time (0.008s out of 2.74s total) and only 269 context switches, the program runs almost entirely in userspace with negligible OS overhead. An IPC of 3.06 with 0.999 CPUs utilized confirms the single core is saturated with compute for the full duration. Next, we used *perf record* in order to perform a call stack analysis of where the program was spending its CPU time. We found that **99.4%** of runtime is spent in linear matrix multiplications, with **70.1%** occurring in the MLP and **28.3%** in attention (note the program spends around 0.4% performing rmnsnorm, Softmax, and embedding operations). While attention is often considered the bottleneck in Transformer models, this aligns with our earlier complexity analysis, as the approximately constant sequence length causes the d^2 term to dominate, making the MLP the primary contributor to runtime. Treating the matrix multiplications together with the remaining targeted primitives as parallelizable gives $p \approx 0.998$, so Amdahl's Law places the theoretical end-to-end speedup ceiling at $\frac{1}{1-p} \approx 500\times$. If the serial baseline runs for longer, closer to an hour, we expect that the parallelizable region could increase beyond 99.8%, leading to even larger speedups.

3 PARALLEL IMPLEMENTATION

Based on the serial workload analysis, our parallelization strategy focuses on exposing more independent work to the GPU. The original *microgpt* structure processes one name at a time and then one token at a time within that name. This creates very little parallel work per operation, especially because the names are short.

While this CPU-side analysis identifies the dominant computational patterns, it does not capture GPU-specific bottlenecks such as memory access and kernel efficiency. To guide implementation decisions at the GPU level, we additionally performed a bottleneck analysis using nvprof on an initial CUDA implementation (see **Section 5.2**). The impact of the resulting design choices is evaluated through an ablation study in **Section 5.2**.

The nvprof analysis directly informed three key optimizations: introducing batching, implementing tiled matrix multiplication, switching from double to float precision. To increase available parallelism, we batch multiple names together so each kernel operates over many sequences and token positions at once, as shown in **Algorithm 2**. We denote the batch size as $B = |\mathcal{B}|$ and represent each batch as a tensor of shape $(|\mathcal{B}|, T, d)$. To reduce CUDA kernel complexity, we preprocess the names into batches where all sequences in a batch have the same length.

Algorithm 2 High-Level Batched Forward Pass

loss ← 0
batches ← CreateBatches($\mathcal{D}$)
for each batch $\mathcal{B} \in$ batches **do**
  $X \leftarrow \text{TokenEmbedding}(\mathcal{B}) + \text{PositionalEmbedding}(0{:}T-1)$
  $X \leftarrow \text{RMSNorm}(X)$
  for $\ell = 1$ to L **do**
    $X \leftarrow X + \text{Attention}(\text{RMSNorm}(X))$ ▷ independent per sequence
    $X \leftarrow X + \text{MLP}(\text{RMSNorm}(X))$
  end for
  $Y \leftarrow \text{Linear}(X)$
  $\text{loss} \leftarrow \text{loss} + \text{CrossEntropy}(Y, \text{Targets}(\mathcal{B}))$
end for
loss ← loss/total_tokens

Furthermore, our strategy focuses on exposing parallelism at the level of individual tensor elements, rather than at the level of tokens or sequences. This allows each CUDA kernel to operate over $|B| \cdot T \cdot d$ elements simultaneously, significantly increasing available work per launch compared to the serial implementation.

We achieve this by decomposing the forward pass into a sequence of CUDA kernels, each operating at a finer granularity over the batched tensor. Rather than treating attention and MLP as monolithic operations, we explicitly map each sub-operation (RMSNorm, linear projections, attention, and residual additions) to separate GPU kernels. This decomposition is shown in **Algorithm 3**, where each step corresponds directly to a kernel launch in our implementation. Since each operation is implemented as its own kernel, we were able to reduce the granularity of the linear matrix multiplication from per-token to per-output-element, and further optimize it using tiling (as motivated by our GPU bottleneck analysis).

Algorithm 3 Layer-by-Layer GPU Forward Pass

for $\ell = 1$ to L **do**

$X_{\text{norm}} \leftarrow \text{RMSNorm}(X)$ ▷ **Attention Block**

$Q \leftarrow \text{Linear}(X_{\text{norm}}, W_Q^{(\ell)})$

$K^{(\ell)} \leftarrow \text{Linear}(X_{\text{norm}}, W_K^{(\ell)})$

$V^{(\ell)} \leftarrow \text{Linear}(X_{\text{norm}}, W_V^{(\ell)})$

$A \leftarrow \text{SelfAttention}(Q, K^{(\ell)}, V^{(\ell)})$

$A_{\text{proj}} \leftarrow \text{Linear}(A, W_O^{(\ell)})$

$X_{\text{mid}} \leftarrow X + A_{\text{proj}}$

$X_{\text{norm2}} \leftarrow \text{RMSNorm}(X_{\text{mid}})$ ▷ **MLP Block**

$H \leftarrow \text{Linear}(X_{\text{norm2}}, W_1^{(\ell)})$

$H \leftarrow \text{ReLU}(H)$

$H_{\text{proj}} \leftarrow \text{Linear}(H, W_2^{(\ell)})$

$X \leftarrow X_{\text{mid}} + H_{\text{proj}}$

end for

Each operation in **Algorithm 3** is implemented as a dedicated CUDA kernel, as summarized in **Table 1**.

From a systems perspective, the full model is transferred to the GPU once at initialization, while a reusable workspace is allocated once per batch. This avoids repeated host-device transfers and ensures that all intermediate activations remain resident on the GPU throughout the forward pass.

Lastly, the original *microgpt* implementation in Python uses FP64 precision by default, by simply changing this in our CUDA version to use floats (fp32) instead, we were able to speed up all of our compute intensive operations.

With this strategy in place, we will next evaluate how effectively the batched GPU implementation reduces wall-clock time compared to serial and PyTorch baselines.

4 EVALUATION STRATEGY

In this section, we compare a serial C++ baseline, unbatched and batched PyTorch baselines, and our custom parallel CUDA implementation to analyze forward-pass runtime across configurations. We evaluate these methods under three scaling regimes: (i) fixed model size with increasing number of names processed, (ii) fixed number of names with increasing model size, and (iii) scaling both simultaneously.

We do not directly compare performance between our CUDA implementation and *microgpt*, as the pure Python implementation is prohibitively slow (our CUDA implementation is over 10,000× faster on a medium benchmark). However, we use *microgpt* to validate correctness by comparing logits from a single-sequence forward pass against those produced by the original implementation.

For analyzing results, we will abbreviate methods as UT = unbatched_torch, BT = batched_torch, SC = serial_cpp, and PC = parallel_cpp. We abbreviate presets as S, M, L, VL, and XL for the standard size sweep, and MS5, MM5, ML5, MVL5, and MXL5 for the 5k-step model sweep.

5 PERFORMANCE ANALYSIS

5.1 Performance Analysis

We begin by examining the case of a fixed model size with increasing input name counts, shown in **Figure 1** below.

Our PC method outperforms all other baselines across every input size benchmarked. For both SC and UT, this result is unsurprising, as PC is more heavily optimized for this specific workload. More notably, PC also outperforms BT, which is a highly optimized and widely used ML framework. This advantage can largely be attributed to Python overhead in BT, including costs associated with object management and data movement between Python and underlying implementations.

However, a closer examination of the results shows that, while PC currently outperforms BT, its runtime increases more rapidly with input size, whereas BT exhibits near-ideal scaling behavior. It is therefore reasonable to expect that as input sizes continue to grow, particularly to those typical of modern ML workloads, PC will eventually lose its performance advantage.

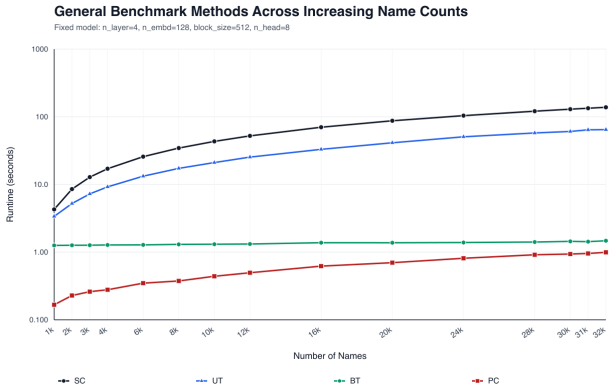

Figure 1: Results from Input Scaling

Next we will look at the results from the other two scaling regimes: holding input size constant while increasing model size (ii), and increasing both input size and model size together (iii). The preset definitions for these experiments cam be found in **Appendix A** and the results are shown in **Table 2**.

Across both regimes, PC consistently outperforms all baselines, maintaining the largest speedup across almost every preset. This trend holds even as model size increases, demonstrating that the implementation effectively exploits parallelism in the dominant compute paths.

To better interpret these results, we define the forward-pass complexity as $O\big(N \cdot L \cdot (C_{\text{attn}} + C_{\text{mlp}})\big)$, where $C_{\text{attn}} = T^2 \cdot d$ and $C_{\text{mlp}} = T \cdot E^2$. Under the assumption that T remains approximately constant, this simplifies to $\approx O(N \cdot L \cdot d^2)$.

The empirical trends in **Table 2** align with this analysis. In regime (ii), where N is fixed and model size increases, runtime grows rapidly with d and L, consistent with the d^2 dependence. In regime (iii), where N, L, and d all increase, the combined effect leads to the largest runtimes, reflecting the multiplicative $O(N \cdot L \cdot d^2)$ scaling. Despite this growth, PC maintains a consistent advantage.

Operation	Thread Granularity	Description
Embedding + Positional	element (B, T, d)	each thread loads one embedding value
RMSNorm	token (B, T)	each thread normalizes one token by looping over its d values
SelfAttention	element (B, T, d)	one thread computes one weighted scalar within its attn head
Linear (Q, K, V, MLP)	element $(T$, out_dim$)$	tiled matix mul, one thread owns one output cell
ReLU	element (B, T, d)	independent element-wise activation
Vector Add	element (B, T, d)	independent element-wise vector addition
CrossEntropy	single thread	one thread loops over all $(B, T,$ vocab$)$ to compute cross-entropy

Table 1: Kernel decomposition with thread granularity and execution details.

Preset	UT	BT	SC	PC
MS5	4.33	0.92	1.23	0.20
	0.28×	1.34×	1.00×	6.22×
MM5	6.81	1.10	10.76	0.26
	1.58×	9.80×	1.00×	41.38×
ML5	12.45	2.37	93.53	0.50
	7.51×	39.45×	1.00×	185.50×
MVL5	20.20	5.79	327.28	1.07
	16.21×	56.50×	1.00×	307.03×
MXL5	32.29	12.34	784.64	2.10
	24.30×	63.59×	1.00×	373.60×

Preset	UT	BT	SC	PC
S	2.30	0.93	0.50	0.14
	0.22×	0.54×	1.00×	3.50×
M	6.66	1.07	10.76	0.25
	1.62×	10.04×	1.00×	42.40×
L	23.32	2.41	189.75	0.76
	8.14×	78.82×	1.00×	250.21×
VL	64.90	6.14	1342.22	3.06
	20.68×	218.60×	1.00×	438.44×
XL	127.71	13.46	4771.83	9.18
	37.37×	354.55×	1.00×	519.91×

Table 2: Runtime (top)/speedup (bottom) per method. Runtime is in seconds; Speedup is relative to SC

Overall, PC consistently outperforms all baselines while exhibiting scaling behavior that aligns with the theoretical $O(N \cdot L \cdot E^2)$ complexity. However, BT demonstrates more favorable scaling at larger sizes, suggesting that PC's performance advantage may diminish for sufficiently large workloads.

5.2 Ablation + Bottleneck

To perform the bottleneck analysis, we utilized nvprof to perform a kernel-level profiling of our program. What we wanted to discover was what portions or kernel calls were responsible the largest portion of program time. **Table 3** shows the configuration for the model and training set size used in the profiling. The batch size parameter is only applicable to the second profiling result, after optimizations were introduced.

Parameter	Value
Names	10,000
Batch Size	512
Number of Layers	8
Embedding Dimension	512
Block Size	64
Number of Heads	16

Table 3: Profiling Configuration Parameters

What we noticed in **Table 4** were two things:

(1) Over 80% of execution time was spent in the `linear_kernel`.
(2) The average time per kernel was reasonable, but we had many invocations of them.

Kernel	Time (%)	Total (s)	Calls	Avg (μs)	Max (μs)
linear	80.86	70.32	490k	143.51	464.80
RMSNorm	10.33	8.99	170k	52.86	82.27
self-attn	7.33	6.38	80k	79.70	162.91
cross-ent.	0.85	0.74	10k	73.68	149.54
resid. add	0.25	0.22	160k	1.36	1.95
ReLU	0.13	0.11	80k	1.37	2.14
embed. look.	0.02	0.02	10k	1.61	2.18

Table 4: CUDA Kernel Profiling Results for Parallel Implementation Pre-Optimizations

From these investigations, we concluded that the most effective path forward was to adopt optimization strategies standard with modern large language model (LLM) training practices. Our first focus was improving the performance of the linear layer, which is dominated by matrix multiplication. By implementing a tiling strategy, as introduced in lecture, and transitioning from double-precision to single-precision (float) arithmetic, we reduced the average runtime of the `linear_kernel` by approximately 81.87×.

Building on this improvement, we next targeted batching, a cornerstone of modern training systems. Processing individual tokens or sequences underutilizes the parallel compute capabilities of GPUs. By restructuring our implementation to process multiple sequences simultaneously in a single batch, we were able to more effectively leverage GPU parallelism and further improve overall throughput.

Figure 2 presents the ablation study used to evaluate the impact of our optimizations. Overall, the results demonstrate that these changes effectively reduce the runtime of the forward pass. One notable observation is that the tiling optimization, in isolation, led

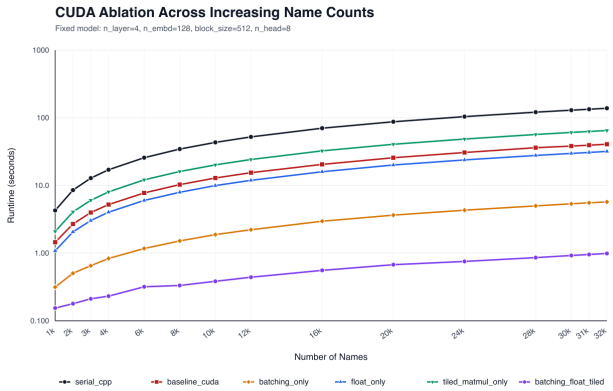

Figure 2: Ablation Study of Implemented Optimizations

to an increase in runtime. This behavior is expected for smaller problem sizes where batching is not applied. In such cases, the overhead associated with thread synchronization and shared memory transfers outweighs the benefits of improved data locality, resulting in a net slowdown.

Our profiling results for the optimized implementation can be found in **Appendix C**.

6 CONCLUSION

In this project, we successfully transitioned the GPT-2 forward pass from a sequential Python implementation to a high-performance, parallelized CUDA C++ application. By identifying and addressing the bottlenecks inherent in serial execution, we demonstrated the immense computational advantages of GPUs for transformer-based architectures.

Our final implementation leverages several important parallel computing techniques: batched processing of sequences, tiled matrix multiplication, and a transition from double-precision to single-precision floating point accuracy. The results show the effectiveness of these strategies, achieving a speedup of over 10,000× compared to the serial Python baseline and over 500× compared to the serial C++ implementation on large-scale model configurations. Notably, our CUDA implementation outperformed the PyTorch implementation in several benchmarks, especially in smaller to mid-sized workloads where Python overhead is more pronounced.

However, our performance analysis also revealed that while our CUDA implementation is very powerful, its scaling behavior for extremely large datasets and model sizes slightly lags behind that of industry-standard frameworks like PyTorch. This suggests that further optimizations, such as more sophisticated memory coalescing, reducing global memory traffic between operations, and the use of specialized hardware features like Tensor Cores, would be necessary to maintain a performance advantage given larger models.

Future work could extend this project by implementing the backward pass to support full model training, reducing memory usage by further reducing floating point precision, and exploring multi-GPU parallelism. Ultimately, this project serves as a clear demonstration of how parallel computing principles are the driving forces behind the modern era of large language models.

7 LINK TO SOURCE AND OTHER RESOURCES

The full project source, including the implementations, benchmark scripts, and generated results, is available at https://github.com/sigggy/CUDA-parallel-gpt.

8 FEEDBACK

No feedback provided.

A BENCHMARK PRESETS

Preset	n_layer	n_embd	block_size	n_head	steps
S	1	64	128	4	2000
M	2	128	128	8	5000
L	4	256	256	8	10000
VL	6	384	512	12	20000
XL	8	512	512	16	30000
MS5	1	64	512	4	5000
MM5	2	128	512	8	5000
ML5	4	256	512	8	5000
MVL5	6	384	512	12	5000
MXL5	8	512	512	16	5000

Table 5: Model Configurations Used in Experiments

B SERIAL IMPLEMENTATION PROMPT

The initial serial C++ baseline discussed in Section 2 was generated from the prompt “Convert the provided microGPT-style Python implementation into a clean serial C++ program,” with the following requirements:

- Keep the model architecture equivalent to the Python version, including token embeddings, positional embeddings, stacked transformer blocks, RMSNorm, causal self-attention, MLP feedforward layers, residual connections, and final logits with cross-entropy loss.
- Implement only the forward pass.
- Do not use CUDA, OpenMP, Eigen, BLAS, or any external numerical library.
- Use simple C++ data structures such as `std::vector` over `double` values and explicit loops.
- Process one input sequence at a time.
- Preserve the same tensor shapes and operation ordering as the Python version.
- Include clear helper functions for embedding lookup, RMSNorm, linear projection, attention score computation, softmax, MLP projection, ReLU, residual addition, and logits/loss computation.
- Prioritize readability and correctness over optimization.
- Use deterministic parameter loading from fixture files so the C++ implementation can be checked against the Python baseline.
- Organize the code so later versions can reuse the same structure for batching and CUDA parallelization.

C OPTIMIZED PROFILING RESULTS

Kernel	Time (%)	Total Time	Calls	Avg	Max
linear_kernel	85.88%	2.40485 s	1372	1.7528 ms	6.9377 ms
rmsnorm_kernel	4.52%	126.59 ms	476	265.95 μs	424.22 μs
self_attn_kernel	4.45%	124.61 ms	224	556.29 μs	1.1746 ms
cross_entropy_loss_kernel	3.73%	104.36 ms	28	3.7271 ms	6.7604 ms
relu_kernel	0.58%	16.380 ms	224	73.123 μs	133.66 μs
add_vec_kernel	0.42%	11.762 ms	448	26.254 μs	50.016 μs
embedding_lookup_kernel	0.02%	481.57 μs	28	17.198 μs	31.776 μs

Table 6: CUDA Kernel Profiling Results for the Optimized Parallel Implementation

REFERENCES

[1] Andrej Karpathy. 2026. microgpt. Blog post. https://karpathy.github.io/2026/02/12/microgpt/.

[2] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. *Language Models are Unsupervised Multitask Learners.* Technical Report. OpenAI. https://cdn.openai.com/better-language-models/language_models_are_unsupervised_multitask_learners.pdf.