

Accelerating Attention for GPT-2 Using FLASHATTENTION, Longformer, and cosFORMER

Stanford CS224N Default Project

Diego Sierra

Department of Computer Science
Stanford University
dds7ae@stanford.edu

Thomas Sarda

Department of Computer Science
Stanford University
tsarda@stanford.edu

Tom-Eliot Jullien

Department of Computer Science
Stanford University
jullien@stanford.edu

Abstract

Modern language models are often limited by the quadratic complexity and memory overhead of Transformer self-attention. We evaluate the trade-off between systems-level optimizations, such as FlashAttention, and algorithmic modifications, including cosFormer and Longformer. By integrating these variants into a GPT-2 fine-tuning pipeline for paraphrase detection and sonnet generation, we analyze their impact on both task quality and hardware efficiency.

Our findings show that PyTorch Scaled Dot-Product Attention (SDPA), which utilizes fused kernels, is the most effective approach, reducing end-to-end training time by $\approx 8\%$ and peak GPU memory by $\approx 26\%$ without degrading accuracy. Conversely, our implementations of cosFormer and Longformer failed to yield speedups. This performance gap is attributed to the short average sequence length of our tasks (≈ 48 tokens) and the high IO overhead of unfused operations. We conclude that theoretical complexity gains are only realized on modern GPUs when paired with hardware-aware, IO-optimized implementations.

1 Key Information to include

- Mentor: Wei Liu; No external collaborators; No project sharing
- Late Days:

Team member	Late days remaining
Tom-Eliot Jullien	4
Thomas Sarda	4
Diego Sierra	4
Total pooled	12
Team size	3
Late days applied for the report	3

We have not used any late days for the assignment and we only turned in our milestone report 1 day and 23 hours late on Gradescope. So for the final report, we would like to use the max 3 days.

2 Introduction

Transformer self-attention is the computational core of modern language models, but it is also a major performance bottleneck. A common way to describe computational cost is in terms of *FLOPs* (floating-point operations), i.e., the number of arithmetic operations such as additions and multiplications performed on floating-point numbers. However, FLOPs alone often fail to predict real runtime on modern accelerators. In practice, many Transformer workloads are *memory/IO-bound*, meaning that performance is limited primarily by *data movement* (reading and writing tensors to GPU memory) rather than by the GPU’s peak arithmetic throughput. Standard attention implementations create and repeatedly read/write large intermediate tensors (e.g., attention score and probability matrices) from GPU *high-bandwidth memory* (HBM), which can dominate runtime and inflate peak memory usage [1]. A complementary systems perspective emphasizes that performance can improve substantially when we reduce memory traffic (bytes transferred between compute units and memory) and fuse kernels (combine multiple operations into a single GPU kernel) so that intermediate results stay in fast on-chip memory rather than being written back to HBM [2].

In response, prior work has pursued two complementary directions. A systems-oriented line of work accelerates *exact* attention by improving implementation efficiency, for example by fusing operations and reducing memory traffic so that large intermediate tensors need not be repeatedly materialized in GPU HBM, as in FlashAttention-style approaches [1]. In parallel, an algorithmic line of work reduces the cost of long-context attention by changing the attention computation itself. Broadly, these methods aim to (i) avoid constructing dense $T \times T$ interaction matrices, and/or (ii) restrict which token pairs are allowed to interact. For instance, *linear attention* variants rewrite the attention computation so that it can be evaluated using compact running summaries, which can reduce memory growth as sequence length increases (e.g., cosFormer) [3]. *Sparse attention* variants impose structured sparsity patterns—such as local sliding windows—so each token attends to only a subset of positions, reducing both computation and memory by avoiding dense all-pairs interactions (e.g., Longformer-style attention) [4]. Together, these approaches highlight a practical design space of systems optimizations versus algorithmic approximations, motivating a controlled empirical comparison within a shared fine-tuning pipeline.

In this project, we study how swapping the causal self-attention operator in a GPT-2 fine-tuning pipeline affects (i) downstream quality and (ii) practical efficiency on two CS224N default-project tasks: paraphrase detection on Quora Question Pairs (QQP) and sonnet generation. Following the default-project extensions track, we focus on attention-mechanism modifications and compare four causal variants: (i) standard masked softmax attention, (ii) a fused exact attention path using PyTorch Scaled Dot-Product Attention (SDPA), (iii) cosFormer, and (iv) Longformer-style causal sliding-window attention. Because QQP fine-tuning runs for ~ 90 –190 minutes in our setup, we use it as the primary benchmark for *end-to-end* efficiency.

Across our experiments, system-level fusion via SDPA yields consistent efficiency gains with negligible impact on accuracy, whereas algorithmic modifications (linear or sparse attention) exhibit more pronounced trade-offs between speed, memory, and task performance in our setting. We detail these trade-offs in Results Section.

3 Related Work

Transformers and masked softmax attention. SDPA computes

$$\text{Attn}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d}} + M\right)V, \quad (1)$$

where (per head) $Q, K, V \in \mathbb{R}^{T \times d}$, d is the head dimension, and M is an additive mask that enforces causality and ignores padding tokens by assigning large negative values to disallowed positions [5]. GPT-2 is an autoregressive Transformer that uses causal masking during both training and generation so that token t only attends to tokens $\leq t$ [6].

FlashAttention and IO-aware exact attention. FlashAttention emphasizes a systems view of attention efficiency: on modern GPUs, runtime is often limited by *memory traffic* (reads/writes to GPU HBM) rather than by arithmetic throughput. Standard implementations typically materialize

large intermediate tensors, such as the $T \times T$ attention score and probability matrices, which must be repeatedly read and written. FlashAttention reduces this overhead via tiling and kernel fusion, combining score computation, softmax, and value aggregation while keeping intermediates in fast on-chip memory (SRAM/registers) and using an online softmax procedure to avoid storing the full $T \times T$ matrix in HBM; the backward pass similarly uses recomputation to reduce memory writes [1]. Practitioner-oriented performance guides echo these principles, highlighting the benefits of reducing memory traffic and fusing kernels to increase arithmetic intensity [2].

Sparse and non-softmax attention variants. A complementary line of work reduces attention cost by modifying the attention pattern or the normalization itself. *Sparse attention* methods restrict each query position t to attend only to a structured subset of keys $\mathcal{N}(t) \subseteq \{1, \dots, T\}$. In this view, masked attention can be written as

$$\text{Attn}_{\text{sparse}}(Q, K, V)_t = \sum_{j \in \mathcal{N}(t)} \alpha_{t,j} V_j, \quad \alpha_{t,j} = \frac{\exp\left(\frac{Q_t K_j^\top}{\sqrt{d}}\right)}{\sum_{k \in \mathcal{N}(t)} \exp\left(\frac{Q_t K_k^\top}{\sqrt{d}}\right)}. \quad (2)$$

Longformer instantiates $\mathcal{N}(t)$ using a local sliding window of size w (and optionally a small set of global tokens), yielding an attention cost that scales as $O(Tw)$ rather than $O(T^2)$ when $w \ll T$ [4].

Non-softmax / linear attention methods instead replace the softmax interaction with alternative normalizations that admit factorized computation. A common template writes attention using a nonnegative feature map $\phi(\cdot)$:

$$\text{Attn}_{\text{lin}}(Q, K, V) = \frac{\phi(Q)(\phi(K)^\top V)}{\phi(Q)(\phi(K)^\top \mathbf{1})}, \quad (3)$$

which can be evaluated without forming a $T \times T$ matrix by first computing the summaries $S = \phi(K)^\top V$ and $z = \phi(K)^\top \mathbf{1}$. cosFormer refines this idea by introducing a cosine reweighting of queries/keys (implemented via a deterministic trigonometric decomposition) together with a non-softmax normalization, enabling linear-time attention under its factorization [3]. In practice, realized speedups depend strongly on implementation details: without carefully optimized kernels, methods with better asymptotic complexity may not outperform highly engineered dense attention on GPUs.

4 Approach

We isolate the impact of the attention operator by keeping the GPT-2 architecture, tokenizer, data pipeline, and optimizer fixed, and swapping only the causal self-attention implementation inside each Transformer block. Unless otherwise stated, all experiments use the gpt2 configuration (12 layers, 12 heads, head dimension $d=64$, hidden size 768) [6]. We compare four causal attention backends that share the same input/output interface and can be selected at runtime.

Baseline (standard) masked softmax attention. Our baseline implements standard SDPA with an additive padding mask and a causal (upper-triangular) mask, followed by softmax (with dropout during training) and value aggregation as in (1).

Fused SDPA ("flash") attention. Our "flash" mode routes attention through PyTorch's SDPA operator with `is_causal=True`. In our float32 runs, SDPA does not dispatch to the FlashAttention (fp16/bf16) kernel, but it still provides a more fused/optimized exact-attention path than our baseline implementation. This collapses multiple steps (masking/softmax/dropout/value weighting) into a smaller set of kernels, reducing intermediate tensor materialization and kernel-launch overhead (the same general principle FlashAttention exploits at the kernel level) [1, 2]. In our paraphrase setup, we ignore the explicit padding mask inside the fused operator to preserve performance; empirically, this does not measurably affect accuracy for our evaluation protocol.

cosFormer attention. To study an algorithmic alternative to masked softmax attention, we implement a causal cosFormer-style attention backend [3]. Concretely, we apply a non-negative feature map $\phi(\cdot)$ (ReLU) to queries and keys, and introduce a deterministic cosine/sine reweighting over token positions. This yields a factorized formulation in which causal attention can be computed

using prefix (running) summaries rather than explicitly forming a dense $T \times T$ score matrix: at each position t , the output depends on accumulated key–value statistics from positions $j \leq t$, normalized by an accumulated key mass. We integrate this operator as a drop-in replacement for GPT-2 causal self-attention while keeping the rest of the model and training protocol unchanged.

Longformer-style causal sliding-window attention. We implement the sparsity pattern as an additive mask over the dense attention matrix, setting blocked entries to -10^4 before softmax. This is numerically identical to a true sparse kernel and yields valid quality measurements, but does not reduce wall-clock FLOPs. We forgo a custom sparse kernel for the following reasons. First, at our sequence lengths (under 50 tokens on average for our tasks) we do not expect the substantial gains from linear cost attention observed above 2000 tokens [4]. Secondly, this method still allows us to investigate whether sparse attention affects generation quality relative to full attention. Our results isolate the inductive bias of local attention.

We search over window sizes $w \in 32, 64, 96$ and global token counts $g \in 1, 3, 8, 16$, selecting $w = 64, g = 1$ as the best efficiency–accuracy tradeoff (CHRF 42.60 vs. 42.22 for standard attention on dev set). This is also implemented as a drop-in replacement for GPT-2 causal self-attention.

5 Experiments

5.1 Data

We evaluate on two tasks from the CS224N default GPT-2 project.

- **Paraphrase detection:** We use the Quora Question Pairs (QQP) dataset provided in the default project. Our training logs indicate 283,003 training examples and 40,429 development examples. Inputs are tokenized with the GPT-2 tokenizer and truncated/padded to a fixed maximum sequence length. We construct padding masks in the standard additive form (0 for valid tokens, large negative values for padded positions).
- **Sonnet generation:** We use the default-project sonnet dataset and training pipeline. The dataset consists of Shakespeare-style sonnets and the provided preprocessing/tokenization pipeline produces fixed-length training sequences consistent with GPT-2 input formatting.

5.2 Evaluation method

We evaluate each attention variant along two axes: task quality and computational efficiency. To ensure a controlled comparison, all settings are held fixed across runs except for the attention backend.

- **Paraphrase detection:** We report development accuracy and submit predictions to the CS224N test leaderboard to obtain test accuracy.
- **Sonnet generation:** We measure generation quality using chrF [7] on the development set and submit to the CS224N test leaderboard for test chrF. chrF measures similarity between a generated text and reference text using a character n -gram F-score (precision/recall), and higher chrF means the generation matches the reference more closely.
- **End-to-end efficiency:** From training logs, we report total wall-clock time, throughput (steps/sec and tokens/sec), and peak GPU memory usage (allocated and reserved).
- **Attention-only efficiency:** To isolate the attention kernel, we run an attention microbenchmark at fixed batch size and sequence length, reporting forward latency (inference) and forward+backward latency (training).

5.3 Experimental details

We use the default GPT-2 (gpt2) configuration described in the Approach section.

Paraphrase detection hyperparameters. For paraphrase detection, we fine-tune for 5 epochs with batch size 8, maximum sequence length 512, learning rate 9.36×10^{-6} , and weight decay 10^{-3} .

To study attention extensions under a controlled setting, we keep the model and training pipeline fixed and swap only the causal self-attention backend: standard masked softmax attention (baseline),

PyTorch SDPA with `is_causal=True` (“Flash/SDPA”), cosFormer, and Longformer-style sliding-window attention. Unless otherwise stated, attention-only microbenchmarks are run in float32.

All end-to-end paraphrase fine-tuning runs were executed on an NVIDIA H100 GPU. Attention microbenchmarks and additional speed tests (using only 2 epochs) were executed on an NVIDIA L4 GPU. We use an H100 for end-to-end training because it better represents high-throughput training hardware, and we use an L4 for microbenchmarks and quick speed tests because it is more cost-efficient while still providing a realistic GPU environment.

Sonnet generation hyperparameters. For sonnet generation, we fine-tune GPT-2 for 10 epochs with batch size 8 and learning rate 8.4996×10^{-5} while freezing the first four Transformer layers. During periodic evaluation, we generate using nucleus sampling ($\text{top-}p=0.9165$) with temperature 1.024 and compute chrF; we select the checkpoint with the best development chrF.

5.4 Results

We report (i) default-project leaderboard scores for paraphrase detection and sonnet generation, and (ii) controlled efficiency measurements when swapping only the causal self-attention backend in the same GPT-2 fine-tuning pipeline. End-to-end results are reported on an H100 GPU, while attention microbenchmarks are reported on an L4 GPU.

Default-project leaderboard scores. We first report task-quality results for our GPT-2 (gpt2) submissions. Table 1 summarizes dev and test leaderboard scores for paraphrase detection and sonnet generation.

Task (metric)	Dev	Test
Paraphrase detection (accuracy)	0.899	0.885
Sonnet generation (chrF)	42.222	41.666

Table 1: CS224N default-project dev and test leaderboard scores for GPT-2 (gpt2).

End-to-end efficiency on QQP paraphrase fine-tuning (H100). We next compare attention backends under fixed hyperparameters to measure end-to-end efficiency in a controlled setting. Table 2 reports development accuracy, wall-clock time, throughput, and peak GPU memory during QQP fine-tuning.

Attention backend	Dev acc	Wall (min)	Tokens/s	Peak (GB)	Reserved (GB)
Standard	0.894	99.12	11984	4.62	5.33
Flash/SDPA	0.895	91.23	13072	3.42	3.86
cosFormer	0.864	192.88	6146	5.01	5.66
Longformer	0.895	154.05	7948	4.62	5.45

Table 2: End-to-end QQP fine-tuning results on a single H100. Peak and Reserved denote peak allocated and peak reserved GPU memory from training logs; Tokens/s is averaged over training steps (excluding evaluation). Wall-clock time includes training, periodic evaluation, and checkpointing.

Attention-only microbenchmarks (H100). To isolate the attention kernel from the full training pipeline, we also run an attention-only microbenchmark on the H100. Table 3 reports forward latency (inference) and forward+backward latency (training), together with peak allocated GPU memory. Consistent with the end-to-end results, SDPA achieves the lowest forward latency and substantially reduces peak memory compared to the standard baseline. The longformer-style backend exhibits high variance and much higher latency, consistent with additional masking overhead and the absence of specialized sparse kernels in our current implementation.

Attention backend	Fwd (ms)	Total (ms)	Peak inf (MB)	Peak train (MB)
Standard	1.126 ± 0.118	3.045 ± 0.052	303.0	527.0
Flash/SDPA	0.597 ± 0.035	2.545 ± 2.667	98.8	191.3
cosFormer	1.346 ± 0.549	3.895 ± 0.542	435.2	767.4
Longformer	10.965 ± 8.052	16.198 ± 10.641	311.8	526.8

Table 3: Attention-only microbenchmark on H100. Fwd is eval-mode forward latency under no_grad; Total denotes forward+backward latency in train mode. Memory columns show peak allocated GPU memory during the benchmark.

Key takeaway (QQP, end-to-end). Tables 2 and 3 shows that fused SDPA provides the best efficiency/accuracy trade-off in our setting: it reduces wall-clock time and peak memory relative to standard attention while preserving development accuracy. In contrast, cosFormer is slower and degrades accuracy, and our Longformer-style variant does not improve end-to-end runtime, consistent with the fact that practical speedups often require specialized kernels in addition to improved asymptotic structure.

Attention-only microbenchmarks (L4). To isolate the attention kernel from the rest of the model, we also run an attention-only microbenchmark. Table 4 reports forward and backward latency, along with peak allocated and reserved GPU memory, on an NVIDIA L4.

Attention backend	Fwd (ms)	Total (ms)	Peak inf (MB)	Peak train (MB)
Standard	7.253 ± 0.432	19.933 ± 0.183	279.15	570.00
Flash/SDPA	1.938 ± 0.196	7.470 ± 0.230	74.90	188.00
cosFormer	7.017 ± 0.519	23.004 ± 0.739	411.36	942.00
Longformer	19.585 ± 0.612	30.331 ± 0.252	287.90	678.00

Table 4: Attention-only microbenchmark on L4. Fwd is eval-mode forward latency under no_grad; Total denotes forward+backward latency in train mode. Memory columns show peak allocated GPU memory during the benchmark.

Interpreting the L4 microbenchmarks. Table 4 highlights large kernel-level gains for SDPA on L4: total attention time drops from 19.933 ms (standard) to 7.470 ms and peak allocated memory drops from 279.15 MB to 74.90 MB. cosFormer remains in the same order of magnitude as standard attention and increases memory usage, while the Longformer-style backend is slower, consistent with additional masking overhead and the lack of sparse kernels in our current implementation.

Summary. Overall, fused SDPA improves both end-to-end efficiency on H100 and attention-kernel efficiency on L4 while preserving accuracy on QQP. In our current setting, linear/sparse variants do not translate theoretical savings into wall-clock improvements without specialized kernels (e.g., fused scans or sparse attention kernels).

6 Analysis

Why does Flash/SDPA help here? The primary benefit comes from reducing intermediate tensor materialization and kernel-launch overhead. A standard attention pipeline is typically decomposed into several large steps (score matmul, masking, softmax, dropout, and value aggregation), each of which may read/write large tensors to GPU HBM. In contrast, SDPA provides a more fused exact-attention path that collapses multiple stages into fewer kernels, reducing memory traffic and improving arithmetic intensity, consistent with IO-aware attention principles. This effect is visible both end-to-end and at the kernel level: on H100, SDPA reduces QQP fine-tuning wall time by $\sim 8\%$ and peak allocated memory by $\sim 26\%$; in attention-only microbenchmarks, SDPA reduces total (forward+backward) attention time by $\sim 16\%$ on H100 and by $\sim 62\%$ on L4, alongside a substantial drop in peak allocated memory on both GPUs.

Why are cosFormer and Longformer slower in our implementation? The main takeaway is that neither backend currently benefits from the kind of GPU kernel fusion and maturity that SDPA enjoys. Two factors dominate.

- **cosFormer: dense execution and unfused operators.** Although cosFormer can be *linear-time* on paper, achieving that speedup requires computing attention without ever forming the full pairwise score matrix. The linear version works by keeping *running summaries* as we scan from left to right (e.g., cumulative key–value statistics and cumulative normalization terms under the cosine reweighting), and then producing the output for token t from these summaries. In our GPT-2 codebase, the fast path (flash) provided by the framework is a fused *dense* attention operator (SDPA), but there is no comparable fused kernel for this scan-based cosFormer computation. Implementing it efficiently would require custom CUDA/Triton kernels (to fuse the scan, normalization, and value aggregation) and careful numerical handling under causal masking. Instead, for reliability and tight integration with the existing training loop, we implemented a faithful reference version using standard PyTorch ops: a dense $T \times T$ score matmul, explicit causal masking, and row-wise normalization. This keeps the math correct, but it necessarily remains $O(T^2)$ in compute and memory and incurs extra overhead from additional unfused operations (ReLU, cosine/sine reweighting, normalization), leading to substantial memory traffic and kernel-launch overhead.
- **Longformer: no sparse kernels yet.** Our Longformer-style implementation enforces a sliding-window pattern, but it does so by constructing a dense $T \times T$ structural mask and then running standard matmul-based attention. This provides numerically equivalent results, but does not realize the sparse $O(Tw)$ compute pattern. In the absence of optimized sparse attention kernels, masking overhead and dense matmuls dominate. However this implementation allows us to isolate the effect of the attention pattern itself from kernel-level optimizations. Flash Attention, for instance, achieves its speedups almost entirely through memory access optimization rather than changes to the attention pattern; CosFormer replaces softmax with a linear kernel approximation. By keeping all variants on the same dense matmul backbone, we ensure that differences in generation quality reflect the inductive bias of each attention design.

Why are asymptotic improvements muted at a maximum length of 512? Even though we set the maximum length to 512, the effective sequence lengths in QQP are substantially shorter. The measured throughput implies an average of ≈ 48 tokens per example (since tokens/sec/steps/sec ≈ 384 tokens per step at batch size 8). With relatively short effective sequences, optimized dense attention can already perform well, and the overheads of alternative attention mechanisms (additional transforms, masking logic, or lack of fusion) are not amortized.

Limitations and next steps. First, our SDPA “flash” mode is evaluated in float32; in this dtype, PyTorch does not dispatch to the FlashAttention kernel in our environment. Enabling mixed precision (bf16/fp16) should allow SDPA to use FlashAttention and may further improve speed and memory. Second, we will complete and report test leaderboard results for all attention variants once we finalize stable runs across configurations. Third, both cosFormer and Longformer would benefit from kernel-level implementations that match their intended structure (fused scan-style kernels for linear attention and true sparse kernels for sliding-window attention).

7 Conclusion

We studied the quality–efficiency trade-off of swapping the causal self-attention operator in a GPT-2 (gpt2) fine-tuning pipeline while keeping the model, data processing, and optimization procedure fixed. Across CS224N default-project tasks (QQP paraphrase detection and sonnet generation), fused exact attention via PyTorch SDPA provided the most consistent practical gains: on QQP end-to-end fine-tuning, SDPA reduced training time by $\sim 8\%$ and peak allocated GPU memory by $\sim 26\%$ while preserving development accuracy. Attention-only microbenchmarks further supported this conclusion, showing substantial kernel-level improvements for SDPA relative to a standard unfused attention implementation.

In contrast, our current cosFormer and Longformer-style backends did not improve wall-clock performance in this setting. For cosFormer, the theoretical advantages of linear-time formulations are

difficult to realize without scan-based fusion and kernel-level support; for Longformer, structured sparsity requires specialized sparse attention kernels to avoid falling back to dense masking and matmul. Overall, our results reinforce a central systems lesson emphasized by FlashAttention-style work: algorithmic complexity improvements are not sufficient on their own. Achieving real speedups on modern GPUs typically requires IO-aware implementations that reduce memory traffic and fuse the full attention computation into optimized kernels. Future work should evaluate mixed-precision SDPA (to enable FlashAttention dispatch), incorporate true sparse kernels for sliding-window attention, and explore fused scan-style implementations for linear attention to better match the intended computational structure of these methods.

Team contributions (Required for multi-person team)

- **Diego Sierra:** Repository setup; initial project plan draft; brainstorming project directions and conducting literature review; implementing GPT-2 and Adam optimizer integration; paraphrase detection training/fine-tuning and course submissions; collecting and organizing training timing logs; writing evaluation/benchmark scripts to report training and inference latency from saved .pt checkpoints and log files; FlashAttention research and implementation (SDPA-based attention backend); drafting the initial milestone report and initial final report; running and benchmarking all attention variants on an H100 GPU.
- **Thomas Sarda:** Brainstorming project directions and literature review; finalizing and submitting the project milestone; GPT-2 code reviews; sonnet generation training/fine-tuning and course submissions; Longformer attention implementation; helping investigate cosFormer behavior and attempting performance improvements; finalizing and submitting the final report; coordinating poster printing.
- **Tom-Eliot Jullien:** Brainstorming project directions and literature review; contributing to milestone finalization; GPT-2 code reviews; sentiment analysis training/fine-tuning and course submissions; cosFormer attention implementation and performance investigation; contributing to final report finalization; poster creation and submission; benchmarking every attention mechanisms on L4 GPU.

References

- [1] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness, 2022.
- [2] Horace He. Making deep learning go *Brrrr* from first principles. Blog post, 2022.
- [3] Zhen Qin, Weixuan Sun, Hui Deng, Dongxu Li, Yunshen Wei, Baohong Lv, Junjie Yan, Lingpeng Kong, and Yiran Zhong. cosformer: Rethinking softmax in attention, 2022.
- [4] Iz Beltagy, Matthew E. Peters, and Arman Cohan. Longformer: The long-document transformer, 2020.
- [5] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2023.
- [6] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. 2019.
- [7] Maja Popović. chrF: character n-gram F-score for automatic MT evaluation. In Ondřej Bojar, Rajan Chatterjee, Christian Federmann, Barry Haddow, Chris Hokamp, Matthias Huck, Varvara Logacheva, and Pavel Pecina, editors, *Proceedings of the Tenth Workshop on Statistical Machine Translation*, pages 392–395, Lisbon, Portugal, September 2015. Association for Computational Linguistics.