Center of Attention: Investigating the Effect of Alternate Attention Mechanisms on Efficiency and Model Utility on GPT-2

Stanford CS224N Default Project

James Ding
Department of Electrical Engineering
Stanford University
jamesdin@stanford.edu

Anisha Palaparthi
Department of Computer Science
Stanford University
anishapv@stanford.edu

Hollie Zheng
Department of Computer Science
Stanford University
holliez@stanford.edu

Abstract

In this project, we investigated how alternative attention mechanisms can improve the performance of the GPT-2 model in the tasks of Sentiment Analysis, Paraphrase Detection, and Sonnet Generation. In particular, we explored the tradeoffs between training/evaluation time and model utility in these three downstream tasks when each of the attention mechanisms (standard dense self-attention, FlashAttention [1], Sliding Window Attention [2], and Attention Sink [3]) are applied. We find that in these downstream tasks, FlashAttention provides the most speedup and is able to maintain relatively high performance, while Sliding Window and Attention Sink achieve a lower runtime compared to the baseline but suffer on performance as a result.

1 Key Information

- **Mentor**: None for default project
- **External Collaborators**: None
- **Sharing project**: None

2 Introduction

General Pretrained Transformer (GPT) models have demonstrated their effectiveness in NLP tasks like question answering and sentiment analysis. GPTs follow the paradigm of "pretrain, then fine-tune" to achieve strong performance in down-stream tasks. The release of GPT-2 highlighted that scaling up model and data size significantly improves performance, ushering in the era of Large Language Models (LLMs) [4]. While scaling unlocks many opportunities, it also introduces new challenges for model size and hardware.

As LLMs grow, transformer models are also getting larger and more popular due to their ability to capture contextual information through self-attention. Longer sequences can enable self-attention models to capture more context, but it comes at a cost of the models' memory and operation complexity increasing quadratically with sequence length [5] . Several attention-optimization models have been proposed, including sparse-approximation, low-rank approximation, and partitioning long

contexts. But those approaches either fail to resolve memory bottlenecks or they lose key contextual information during partitioning [1] [2].

More recent models, such as FlashAttention, Sliding Window, and Attention Sink, claim to address these shortcomings. To assess their effectiveness in practice, we evaluate the tradeoffs between runtime and performance of these attention models in GPT-2 across three downstream tasks: Sentiment Analysis, Paraphrase Detection, and Sonnet Generation.

3 Related Work

3.1 Self-Attention in Transformer Models

Self-attention plays a pivotal role in the performance of transformer models, especially in NLP tasks. In particular, the attention mechanism allows each the model to relate between different words within the input text through a scaled-dot-product calculation. This operation is crucial to the Transformer's ability to focus on relevant parts of the input text as well as model the dependencies between different parts of text [6]. The typical causal multi-head self-attention, referred to as "dense" self-attention, entails multiple scaled-dot-product computations to achieve the benefits discussed above. However, these operations are computationally very expensive, with a complexity of $O(n^2)$ (see "Approach" for further explanation). As a result, dense attention does not scale with larger context lengths and proves to a runtime bottleneck in Transformer models. This problem has motivated substantial research on improving the runtime of the attention computation in order to improve overall model efficiency.

3.2 Accelerating Attention

The primary research direction for improving transformer models focuses on how to reduce the runtime of the crucial yet expensive attention calculation. We observe that the prior work that has sought to improve the runtime of the attention can be categorized into three main approaches: (1) Sparse Attention [7] [2] , (2) Kernel Methods [8] (both of which aim to reduce the computational complexity), and (3) Hardware Optimization [1].

Sparse Attention methods aim to reduce the computational complexity of the attention computation from the $O(n^2)$ standard dense attention requires; they achieve this by selecting certain tokens within the full attention matrix for the softmax computations. Kernel methods instead attempt to replace the softmax operation with a feature map approximation, such as an Exponential Linear Unit (ELU) [8], to reduce the computational complexity. Hardware-optimized attention techniques instead aim to take advantage of advancements in parallel hardware and utilize GPUs more efficiently rather than reduce the computational complexity of dense attention.

We will investigate alternate attention mechanisms that both reduce the computational complexity through sparse attention techniques (Sliding Window and Attention Sink, also referred to as "Slide" and "Sink," respectively, for conciseness) the hardware-optimized method FlashAttetion of computing dense attention (referred to as "flash").

4 Approach

4.1 Baseline GPT-2

We implemented the baseline GPT-2 model using the GPT-2 code template provided by CS224n course staffs. GPT-2 is a decoder-only transformer model with 1.5 billion parameters [4]. It uses byte pair encoding (BPE) to decompose words into tokens. After converting each tokens to ids, GPT-2 uses a trainable embedding layer across each token.

We applied dense causal multi-head self-attention in our baseline GPT-2 model [6]. The attention computation involves the query, key, and value matrices $\mathbf{Q}, \mathbf{K}, \mathbf{V}$, respectively, and is mathematically denoted as:

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}})\mathbf{V}$$

Multi-head attention then allows the model to focus attention on multiple regions and connect different dependencies by concatenating multiple attention "heads". This is represented as:

$$\text{MultiHead}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{Concat}(\text{head}_1, ..., \text{head}_\text{h})\mathbf{W}^O$$

where $\text{head}_\text{i} = \text{Attention}(\mathbf{Q}\mathbf{W}_i^Q, \mathbf{K}\mathbf{W}_i^\mathbf{K}, \mathbf{V}\mathbf{W}_i^V)$ and $\mathbf{W}_i^Q, \mathbf{W}_i^K, \mathbf{W}_i^V, \mathbf{W}^O$ are the corresponding weight matrices.

We further applied a "causal" mask, which masks the attention values on later tokens in the sequence and ensures that attentions is only calculated with respect to previous tokens. Finally, we apply an additional "padding" mask which ensures that the attention calculation does not take into account padding tokens. While multi-head and causal aspects of the attention remain fixed throughout all of our approaches, we vary the softmax computation across the the subsequent three attention mechanisms. Figure 1 shows a visualization comparing all the self-attention mechanisms used.

4.2 FlashAttention

As recent technological innovations have accelerated computations drastically, compute speed has substantially out-paced memory speed; in turn, the majority of operations Transformers rely on are bottlenecked by slow memory accesses. This is why compute-focused approaches like sparse-approximation and low-rank approximation can fail to yield real world speedups, or come at reduced performance for downstream tasks [1] [7] [2]. FlashAttention addresses the memory bottleneck in long sequences by minimizing costly off-chip access through an IO-aware approach. FlashAttention achieves this Io-Aware attention computation through two techniques: Tiling and Recomputation (see Appendix for full FlashAttention algorithm).

Tiling relies on the insight that we can compute attention by blocks by keeping track of two statistics: $m(x)$ and $l(x)$. To see this, we define the softmax operation for a vector $x \in \mathbb{R}^B$ as follows:

$$m(x) \coloneqq \max_i \; x_i, \quad f(x) \coloneqq \begin{bmatrix} e^{x_1 - m(x)} & \dots & e^{x_B - m(x)} \end{bmatrix}, \quad \ell(x) \coloneqq \sum_i f(x)_i, \quad \text{softmax}(x) \coloneqq \frac{f(x)}{\ell(x)}.$$

Note that the softmax operation couples columns of $\mathbf{K}$, so we can use the above definitions to find the softmax of two such vectors $x^{(1)}, x^{(2)}$, namely $x = [x^{(1)} x^{(2)}] \in \mathbb{R}^{2B}$ as follows:

$$m(x) = m(\begin{bmatrix} x^{(1)} & x^{(2)} \end{bmatrix}) = \max(m(x^{(1)}), m(x^{(2)})), \quad f(x) = \begin{bmatrix} e^{m(x^{(1)}) - m(x)} f(x^{(1)}) & e^{m(x^{(2)}) - m(x)} f(x^{(2)}) \end{bmatrix},$$

$$\ell(x) = \ell(\begin{bmatrix} x^{(1)} & x^{(2)} \end{bmatrix}) = e^{m(x^{(1)}) - m(x)} \ell(x^{(1)}) + e^{m(x^{(2)}) - m(x)} \ell(x^{(2)}), \quad \text{softmax}(x) = \frac{f(x)}{\ell(x)}.$$

The equations above show that we can split the matrices $\mathbf{Q}, \mathbf{K}, \mathbf{V}$ into blocks, compute the softmax for the blocks individually, calculate the additional $m(x)$ and $l(x)$ statistics, and combine the results to get the overall softmax. Thus, this can be highly parallelized while limiting the IO operations to GPU kernel; once the blocks of $\mathbf{Q}, \mathbf{K}, \mathbf{V}$ are loaded to the fast SRAM for the GPU computations, IO operations are not needed to until the complete computation is complete.

The key insight of Recomputation is the observation that the entire $O(N^2)$ intermediate attention values do not need to be stored for the backward pass. Instead, by only storing and using the final output attention matrix $\mathbf{O} = \mathbf{P}\mathbf{V} \in \mathbb{R}^{N \times d}$ (where N is the context length and d is the head dimension) and the statistics m, l, we can quickly recover the intermediate matrices $\mathbf{S} = \mathbf{Q}\mathbf{K}^\top \in \mathbb{R}^{N \times N}, \mathbf{P} = \text{softmax}(\mathbf{S}) \in \mathbb{R}^{N \times N}$.

While the traditional attention computation would require reading/writing each of $\mathbf{O}$, $\mathbf{S}$, and $\mathbf{P}$ for every computation and backward pass, FlashAttention minimizes the IO of the attention computation and avoids the expensive IO runtime-bottleneck of GPU computation.

While we use the Dao AI Lab's official paper implementation to leverage CUDA GPU-optimizations, significant implementation was required to integrate this API in a way that was compatible with our larger GPT-2 model that differed from their implementation. This applies to all attention extensions as well (see Appendix for further explanation these implementation details).

4.3 Sliding-Window Attention

A simplified version of the LongFormer Sliding-Window Attention was implemented in the model. This implementation, which takes in one parameter, namely the window overlap w, divides the $\mathbf{K}$ and $\mathbf{Q}$ matrices, which have dimensions (batch size bs, number of heads h, sequence length s, attention head size hs), into chunks with sequence length twice the window overlap.

$$\mathbf{K}_i = \mathbf{K}[:, :, iw : (i+2)w, :] \qquad \mathbf{Q}_i = \mathbf{Q}[:, :, iw : i(i+2)w, :] \qquad \forall i \in \left\{0, \ldots, \left\lfloor \frac{s}{w} \right\rfloor - 1\right\}$$

From this process, each subsequent chunk overlaps with the previous chunk by the specified window overlap w and that these chunks span the entire sequence length s. After the chunking process is complete, each chunk $\mathbf{K}_i$ is multiplied with their respective chunk $\mathbf{Q}_i$ to produce a diagonal component of the final attention score matrix $\mathbf{A}$. Since each chunk has sequence length twice the window overlap, this results in a time complexity of $O(s)$, which is a dramatic speed up compared to normal self-attention, wich has a time complexity of $O(s^2)$.

$$\mathbf{A_i} = \mathbf{Q}_i \mathbf{K}_i^\top \qquad \forall i \in \left\{0, \ldots, \left\lfloor \frac{s}{w} \right\rfloor - 1\right\}$$

After the $\mathbf{A}_i$ are computed, they are aggregated and superimposed in the same sliding manner as $\mathbf{K}_i$ and $\mathbf{Q}_i$ were produced to give the final attention score matrix $\mathbf{A}$.

$$\mathbf{A}[:, :, iw : (i+2)w, iw : (i+2)w] = \mathbf{A}_i, \qquad \mathbf{A}[:, :, j, k] = 0 \text{ otherwise}$$
$$\forall i \in \left\{0, \ldots, \left\lfloor \frac{s}{w} \right\rfloor - 1\right\}$$

This attention score matrix $\mathbf{A}$ is then processed as normal, with the result being scaled, passed through softmax, and then being multiplied by $\mathbf{V}$.

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}(\frac{\mathbf{A}}{\sqrt{d_k}})\mathbf{V}$$

4.4 Attention with Attention Sinks

This method operates similarly to Sliding-Window Attention but with the additional augmentation of each token attending to the first couple tokens. This augmentation comes from the insight that the first couple of tokens in an attention computation often have high attention scores even if the tokens are not significant. This is due to the softmax function requiring attention scores to sum to 1 even if the token does not have a match with any of the prior tokens. Thus, the attention is offloaded into the first couple tokens [2]. Because of this, it was off interest to see if incorporating attention sinks would improves the performance of the model especially compared to using Sliding-Window Attention.

The implementation of attention with attention sinks follows very similarly to the implementation for the simplified LongFormer Sliding-Window Attention above. The main augmentation is that in addition to each corresponding chunk $\mathbf{K}_i$ and $\mathbf{Q}_i$ being multiplied together to get $\mathbf{Q}_i$, all the chunks $\mathbf{K}_i$ are also multipled by the first chunk for $\mathbf{Q}$, namely $\mathbf{Q}_0$, to produce $\mathbf{S}_i$. Or in other words, the first n tokens where n is twice the window overlap w attends to all other tokens in the query. Note that this still maintains a linear time complexity $O(s)$ for computing the attention score matrix $\mathbf{A}$, while improving the performance from Sliding-Window Attention.

$$\mathbf{S_i} = \mathbf{Q}_0 \mathbf{K}_i^\top \qquad \forall i \in \left\{0, \ldots, \left\lfloor \frac{s}{w} \right\rfloor - 1\right\}$$

Once these new $\mathbf{S}_i$ and $\mathbf{A}_i$ are computed as before, these are then aggregated in a similar manner as for sliding window attention to produce the attention score matrix, but with the first $2w$ columns having entries corresponding to $\mathbf{S}_i$.

$$\mathbf{A}[:, :, iw : (i+2)w, iw : (i+2)w] = \mathbf{A}_i, \qquad \mathbf{A}[:, :, iw : (i+2)w, 0 : 2w] = \mathbf{S}_i$$
$$\mathbf{A}[:, :, j, k] = 0 \text{ otherwise} \qquad \forall i \in \left\{0, \ldots, \left\lfloor \frac{s}{w} \right\rfloor - 1\right\}$$

And as before, this attention score matrix $\mathbf{A}$ is processed as normal.

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}(\frac{\mathbf{A}}{\sqrt{d_k}})\mathbf{V}$$

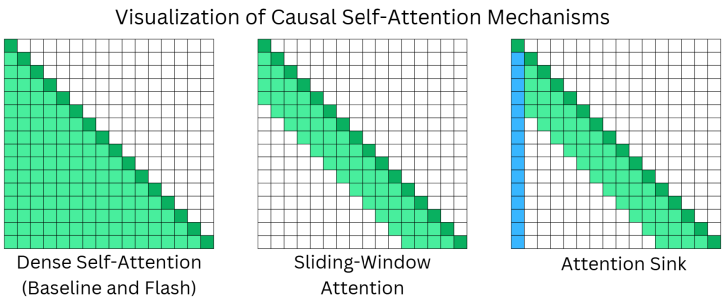


Figure 1: Visualization of Attention Mechanisms

5 Experiments

5.1 Data

There are three tasks for the default project: sentiment analysis, paraphrase detection, and sonnet generation [9]. Sentiment analysis uses two movie review datasets: SST and CFIMDB. SST contains 12k single-sentence reviews labeled by three human judges on a five-point sentiment scale, while CFIMDB includes 2k highly polar, multi-sentence reviews with binary sentiment labels. The inputs of the Sentiment Analysis task are the reviews and the output is the sentiment as categorized in one of 5 classes: "Negative", "Somewhat Negative", "Neutral", "Somewhat Positive", and "Positive". Paraphrase detection relies on the Quora dataset, which comprises 400k question pairs with binary labels indicating whether the pairs are paraphrases. The inputs of the Paraphrase Detection task are the question pairs, and the output is "Yes" or "No" depending on if they are paraphrases of each other. Finally, sonnet generation is based on 154 14-line sonnets written by Shakespeare. Those sonnets are in iambic pentameter, and most have the traditional rhyme scheme of abab cdcd efef gg. The input of the Sonnet Generation task is a starting text used as a context seed, and the output is a sonnet.

5.2 Evaluation method

We used task-specific evaluation metrics to assess model performance [9]. For sentiment analysis, we measured accuracy by comparing GPT-2's final token embedding predictions to the true development set labels. Paraphrase detection was similarly evaluated based on prediction accuracy against the true labels. We also calculated the confusion matrix as well as f1-score, precision, and recall to evaluate the performance of these classification tasks (see Appendix for further explanation of these metrics). For sonnet generation, we employed the CHRF score to quantify how closely GPT-2's generated sonnets aligned with Shakespeare's language distribution. During evaluation, the first three lines of each of the 12 held-out test sonnets were provided as prompts, and the model generated the remaining lines. Besides the model's performance, we also recorded the model's runtime for training and evaluation. Doing so can help determine the efficiency gains of the new attention mechanisms.

5.3 Experimental details

All fine-tuning and testing were conducted on a GCP L4 GPU. The project consisted of two main phases. The first phase involved implementing a GPT-2 baseline and tuning it to meet performance targets for SST, CFIMDB, paraphrase detection, and sonnet generation. This phase required debugging and parameter optimization to achieve the desired results. For SST and CFIMDB, we settled on the default parameter (epoch=10, lr=1e-3) with batch size increased to 64. Additionally, full model trials used a lower learning rate of 1e-5. For paraphrase detection, we used the default parameter (epoch=10, lr=1e-5) with batch size modified to 32, which is the max batch size a L4 GPU can handle. For sonnet generation, we used the default parameter (epoch=10, lr=1e-5, batch size=8) with model size increased to gpt2-large. The max batch size a L4 GPU can handle in this case is 8. The second phase focused on integrating three attention extensions: flash, slide, and sink. Since tuning the extra parameters (overlap and sink size) of slide and sink attention yielded no significant improvement, we used slide and sink's default values for all trials. Slide attention's overlap is 4, and sink attention's overlap is 4 and sink size is 4.

Metric	Parameters	Target	Baseline
SST Last Linear Dev	Default + batch size=64	0.462	0.457
SST Full Model Dev	Default + batch size=64 + lr=1e-5	0.513	0.545
CFIMDB Last Linear Dev	Default + batch size=64	0.861	0.865
CFIMDB Full Model Dev	Default + batch size=64 + lr=1e-5	0.971	0.984
Paraphrase Dev	Default + batch size=32	0.893	0.886
Paraphrase Test	Default + batch size=32	0.851	0.858
Sonnet Dev	Default + model size=gpt2_large	41.103	41.968
Sonnet Test	Default + model size=gpt2_large	41.297	42.177

Figure 2: Our GPT-2 implementation results vs the target for each task provided on Ed and the final project document. Default parameter is 10 epochs, learning rate (lr) of 1e-3, and batch size of 8.

For all tasks, we measured the runtime per training epoch and evaluation step. For binary classification tasks, we tracked validation accuracy and used a confusion matrix to capture true positives, true negatives, precision, recall, and F1 score. For the 5-labels SST, we recorded validation accuracy and its confusion matrix. For sonnet generation, we evaluated performance using the CHRF dev score. Since sonnet generation produces a set of 12 sonnets after each epoch, we measured the time taken for their generation instead of the standard evaluation time. We recorded the average generation time over ten epochs due to the fluctuation of sonnet generation time for each epoch.

Paraphrase detection had a constraint of three test submission attempts, which was fewer than the number of models we aimed to compare (one baseline and three extensions). As a result, we did not use test accuracy for baseline vs extension comparisons.

5.4 Results

After parameter finetuning, our GPT-2 implementation came close or surpassed the target for all the dev and test cases, as shown in 2. After we got our baseline set up, we implemented the attention extensions and ran them on the three downstream tasks. To accommodate for space, we only included the table of results for our leader board results, Paraphrase Detection, and Sonnet Generation in figures 2, 3, and 4 of the main paper; the remaining data, including confusion matrices, Sentiment Analysis results, and runtime graphs are located in the Appendix as Figures 5-12 for further review.

- **SST and CFIMDB**: In Fig. 6, the last linear layer shows no significant runtime difference between attention mechanisms, and this is expected for the smaller model. In the full model, flash has the most decrease in runtime while retaining performance close to Baseline, as shown in Fig. 10 and Fig. 12. While slide and sink has slightly lower runtime than Baseline, both models have worse performance than baseline for last-linear layer (Fig. 9 and Fig. 11) and full-model (Fig. 10 and Fig. 12). Fig. 5 is a plot summarizing the accuracy for each model in sentiment analysis.
- **Paraphrase Detection**: Flash, Slide, and Sink run slightly faster than the baseline as expected Fig. 3. While Flash maintained baseline accuracy and Slide performed slightly worse, Sink's accuracy and F1 score were significantly lower than expected. An interesting observation is that all attention models had similar true negative scores, but the most notable difference was in their true positive rates Fig. 7.
- **Sonnet Generation**: In Fig. 4, there is a very small training runtime difference between the four attention models, probably because the baseline runtime is already low. But surprisingly the average sonnet generation time slightly differs between attention models. Flash and Slide performed close to Baseline, while Sink is behind the rest.

6 Analysis

6.1 On Model Utility

We observe that across the downstream tasks, FlashAttention had the best performance compared to the alternate attention mechanisms of Sliding Window and Attention Sink, closely matching the

Attention	Training Time per Epoch	Dev Time per Epoch	Final Training Loss	Final Dev Accuracy	Precision	Recall	F1 Score
Baseline	45 mins	2 mins	0.39	0.884	0.82	0.87	0.85
Flash	42 mins	85 secs	0.386	0.885	0.84	0.84	0.84
Slide	43 mins	90 secs	0.533	0.752	0.75	0.48	0.59
Sink	43 mins	90 secs	0.631	0.635	0.5	0.16	0.25

Figure 3: Our GPT-2 implementation vs the three attention extensions for paraphrase detection. Since paraphrase detection is a binary classification, we can evaluate with the F1 score. "Dev Time" refers to the evaluation time after the epoch

Attention	Final Training Loss	Runtime per Epoch	Avg. Gen Time per Epoch	Dev CHRF Score
Baseline	2.31	19 secs	50 secs	41.618
Flash	2.413	17 secs	75 secs	41.968
Slide	4.417	18 secs	45 secs	40.78
Sink	3.266	18 secs	37 secs	37.05

Figure 4: Our GPT-2 implementation vs the three attention extensions for sonnet generation.

baseline performance of standard dense self-attention. Sliding Window Attention and Attention Sink both had reduced performance on the three downstream tasks; this outcome is inline with our expectations, as both these approaches compute attention between fewer tokens and are therefore unable to model the dependencies as well as dense self-attention as implemented in our baseline and FlashAttention. This reflected across all our performance metrics, including precision, recall and F1-scores, that match the closely to the baseline for FlashAttention but fall short for Sliding Window Attention and Attention Sink.

An interesting result, upon analysis of the confusion matrices, is that the different attention mechanisms create different points of failure for the model. While our baseline GPT model and FlashAttention model had a fairly distributed errors in evaluation for Paraphrase detection (misclassifying as "Yes" or "No" in a somewhat balanced manner), Attention Sink was more prone to false positives. On the other hand, Sliding Window Attention had a somewhat larger tendency to report false positives than the baseline, though not to the extent of Attention Sink. This disagrees with the principle behind Attention Sink, that including earlier context in the attention calculation could provide better insight into dependencies between tokens with the sequence that Sliding Window - however, our experiments suggest that this feature of Attention Sink is actually detrimental to performance in the downstream task of Paraphrase Detection. We hypothesize this effect is due to format of questions, which may have similar ways of starting the sentence even if two questions are dissimilar in their meaning.

In the task of Sonnet Generation, Attention Sink also achieved the lowest CHRF score, though not by a substantial margin compared to Sliding Window Attention. Overall, we noticed that all three alternate attention mechanisms achieve similar, if not slightly lower, performance compared to the standard. This suggests that optimized attention mechanisms may be better suited to downstream generation tasks such as Sonnet Generation as opposed to the classification tasks like Sentiment Analysis and Paraphrase Detection.

6.2 On Model Runtime

We observe that the alternate attention mechanisms (FlashAttention, Sliding Window, and Attention Sink) achieve a faster training and evaluation time across the downstream tasks (Paraphrase Detection and Sonnet Generation) when compared to the baseline dense self-attention. Across Fig. 8, we saw the greatest speedup with FlashAttention, while Slide and Sink have a smaller speedup over the baseline.

An unexpected result emerged in the Sonnet Generation task: FlashAttention took longer to generate sonnets than the other attention mechanisms, making it an outlier in our experiments. This may be

due to the overhead associated with parsing and generating longer token sequences, which increases evaluation time. In particular, FlashAttention may incur higher I/O costs per access compared to other downstream tasks, which offsets the efficiency gained from Flash's lower number of I/O access.

Overall, these results are in line with our expectations, since the mechanisms of Sliding Window Attention and Attention Sink primarily focus on reducing the computational complexity of the attention calculation while FlashAttention primarily focuses on making better use of GPU hardware to compute standard dense self-attention faster. Though all three attention mechanisms make use of the GPU, FlashAttention's insight into reducing read/write GPU operations allows it to eliminate IO overheads that other attention mechanisms suffer from. Doing so enables Flash to more effectively realize the speedups achieved by GPU matrix computations. Our experiments indicate that the GPU IO overhead has a greater impact on the training and evaluation time of the model compared to the complexity of the $O(n^2)$ attention calculation. As a result, we find that on these downstream tasks reducing computational complexity of the attention mechanism matters less than reducing the IO bottleneck, as both Sliding Window Attention and Attention Sink rely on slow memory read/writes that failed to achieve much speedup.

6.3 Tradeoffs

Our results show that reducing the computational complexity of attention calculations comes at a significant cost to performance. Since attention scores consider fewer tokens, calculations are faster, but the model struggles to capture dependencies needed for downstream tasks. Sliding Window Attention and Attention Sink introduce a tradeoff that sacrificed model utility for faster training/evaluation time. In contrast, standard dense self-attention achieves the best performance but at the cost of significantly longer training and evaluation.

FlashAttention best mitigates the tradeoff between runtime and utility. It matches the baseline's performance across multiple metrics while showing substantial speedup in training and evaluation. Among the attention mechanisms we evaluated, FlashAttention offers the best overall performance considering both model utility and efficiency.

7 Conclusion

We find that alternate attention mechanisms pose a tradeoff between model utility and efficiency across the three downstream tasks of Sentiment Analysis, Paraphrase Detection, and Sonnet Generation. The highest performing model used standard dense self-attention, though it also took longer to train and evaluate. Approaches that reduced the computational complexity of the $O(n^2)$ attention operation, such as Sliding Window Attention and Attention Sink, did provide speedup but suffered in performance relative to the baseline. FlashAttention, which implemented dense self-attention in an IO aware manner than took advantage of GPU computation, best mitigated this tradeoff by performing close to the baseline across multiple metrics and achieving the most speedup. While our work successfully compares the impact of multiple attention mechanisms on model utility and efficiency, we recognize that the scope of the downstream tasks these methodologies were applied to were limited. For future work, we would explore the tradeoffs between model utility and runtime across various other downstream tasks and compare our findings with other attention mechanisms.

Team contributions

James implemented the experimental setup for the three downstream tasks and integrated the attention extensions into the baseline model. He also implemented the hyperparameter tuning and data collection we needed to produce the data and figures in the paper. Anisha implemented the base GPT-2 model, FlashAttention extension, and sanity checks to test the attention modules separately prior to running the experiments. Hollie implemented the Sliding Window attention and Attention Sink attention modules. All three team members equally contributed to the conceptualization, research, and paper writing for the project.

References

[1] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. *arXiv*, 2022.

[2] M. E. Peters I. Beltagy and A. Cohan. Longformer: The long-document transformer. *arXiv preprint arXiv:2004.05150*, 2020.

[3] A. Radford R. Child, S. Gray and I. Sutskever. Generating long sequences with sparse transformers. *arXiv preprint arXiv:1904.10509*, 2019.

[4] Rewon Child David Luan Dario Amodei Ilya Sutskever et al. Alec Radford, Jeffrey Wu. Language models are unsupervised multitask learners. *OpenAI blog, 1(8):9,*.

[5] Samira Abnar Yikang Shen Dara Bahri Philip Pham1 Jinfeng Rao Liu Yang Sebastian Ruder Donald Metzler Yi Tay, Mostafa Dehghani. Long range arena: A benchmark for efficient transformers. *arXiv*, 2020.

[6] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[7] Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. Efficient streaming language models with attention sinks. *arXiv*, 2023.

[8] Angelos Katharopoulos, Apoorv Vyas, Nikolaos Pappas, and François Fleuret. Transformers are rnns: Fast autoregressive transformers with linear attention. In *International conference on machine learning*, pages 5156–5165. PMLR, 2020.

[9] CS224n Course Staff. Cs 224n: Default final project: Build gpt-2. *Stanford University*, 2024.

A Appendix

A.1 Precision, Recall, F1 Scores

Using the confusion matrix, we are able to get precision, recall, and the f1 score for tasks with binary classification.

- Precision: Out of all positive predictions, how many of them are correct? Equation expressed as $TP/(TP + FP)$
- Recall: Out of all actual positive cases, how many did my model predict correctly? Equation expressed as $TP/(TP + FN)$
- F1 Score: Balancing precision with recall. Equation expressed as $2TP/(2TP + FP + FN)$

A.2 Implementation Details

Though we were using APIs from other libraries for the implementation of the attention operation itself across FlashAttention, Sliding Window, and Attention Sink, substantial wrapper implementation we developed from scratch was required to integrate these APIs into our GPT-2 model. This is because the APIs made different assumptions to the input structure, output structure, masking requirements, and presence of padding tokens that diverged significantly from our baseline model. For instance, FlashAttention requries unpadded queries, keys, and values, which needed to be carefully extracted from our padded inputs. Along a similar vein, Sliding Window Attention and Attention Sink had required wrapper implementation to take into account the causal and padding requirements. All three implementations also required differently structuring our wrapper classes to fit both our GPT-2 model and the APIs, which can be observed in the codebase.

A.3 Extra Plots from Results

This section contains extra plots summarizing the runtime, accuracy, and confusion matrix of the three downstream tasks. They are still referenced in the results section, but are included here due to space constraints.

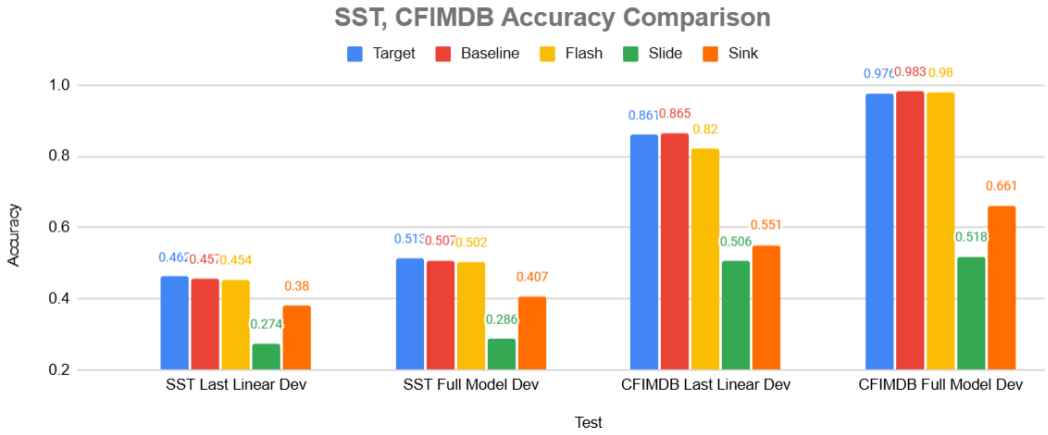


Figure 5: Performance comparison of the four attention models vs target in SST and CFIMDB tasks.

	Last Linear Layer				Full Model			
	SST	**CFIMDB**	**SST**	**CFIMDB**	**SST**	**CFIMDB**	**SST**	**CFIMDB**
Attention	**Train Time per Epoch**		**Dev Time per Epoch**		**Train Time per Epoch**		**Dev Time per Epoch**	
Baseline	11 secs	57 secs	10 secs	11 secs	22 secs	69 secs	20 secs	20 secs
Flash	11 secs	54 secs	10 secs	10 secs	15 secs	50 secs	15 secs	14 secs
Slide	11 secs	57 secs	10 secs	11 secs	20 secs	64 secs	19 secs	20 secs
Sink	11 secs	57 secs	10 secs	11 secs	20 secs	63 secs	19 secs	20 secs

Figure 6: Comparing the runtime between our Baseline vs the three attention extensions on all trials in sentiment analysis.

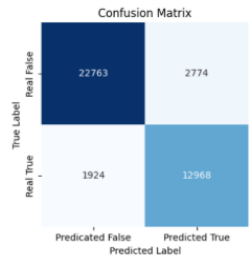
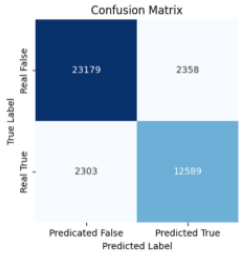
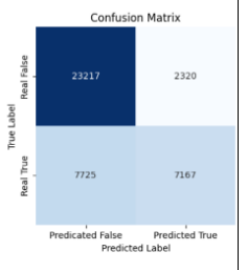
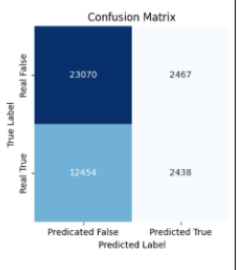
Baseline	**Flash**	**Slide**	**Sink**
			
True Pos: 0.87 **True Neg:** 0.89	**True Pos:** 0.84 **True Neg:** 0.9	**True Pos:** 0.48 **True Neg:** 0.9	**True Pos:** 0.16 **True Neg:** 0.9

Figure 7: Comparing the paraphrase detection confusion matrix between our Baseline vs the three attention extensions.

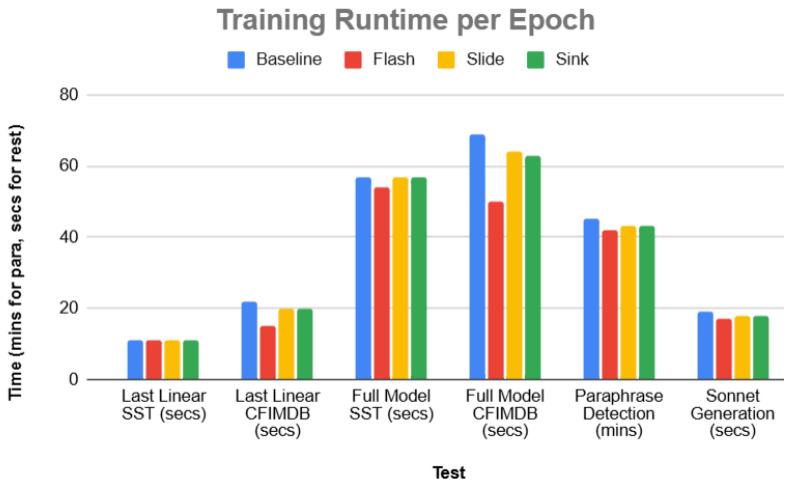


Figure 8: Training runtime comparing the four attention models on all tasks.

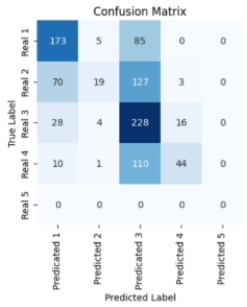
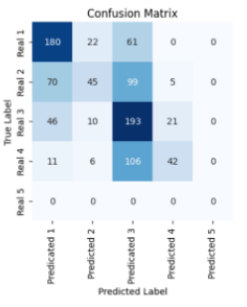
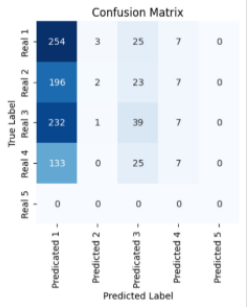
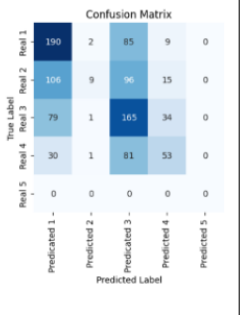
Baseline	**Flash**	**Slide**	**Sink**
			
Dev Accuracy: 0.46	**Dev Accuracy:** 0.45	**Dev Accuracy:** 0.27	**Dev Accuracy:** 0.38

Figure 9: Our Baseline vs the three attention extensions on last linear layer SST.

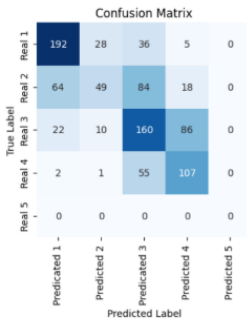
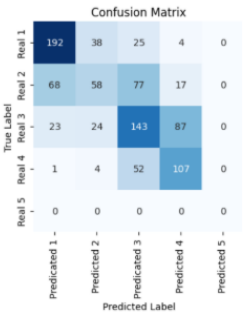
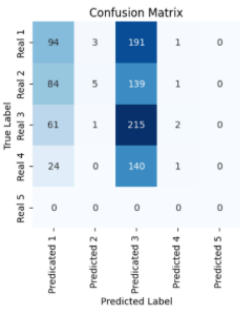
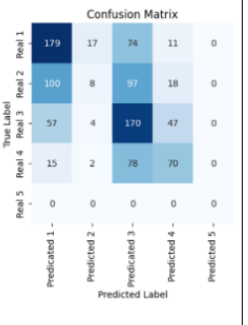
Baseline	**Flash**	**Slide**	**Sink**
			
Dev Accuracy: 0.51	**Dev Accuracy:** 0.50	**Dev Accuracy:** 0.29	**Dev Accuracy:** 0.41

Figure 10: Our Baseline vs the three attention extensions on full model SST.

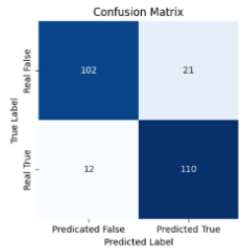
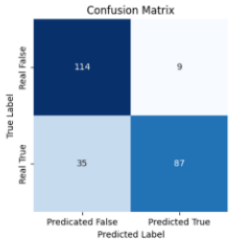
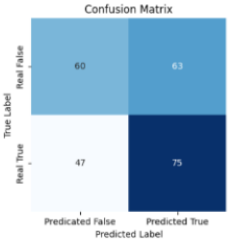
Baseline	Flash	Slide	Sink
			
True Pos: 0.90 **True Neg:** 0.83 **Dev Accuracy:** 0.86 **Precision:** 0.90 **Recall:** 0.87 **F1:** 0.87	**True Pos:** 0.71 **True Neg:** 0.93 **Dev Accuracy:** 0.82 **Precision:** 0.91 **Recall:** 0.71 **F1:** 0.79	**True Pos:** 0.64 **True Neg:** 0.57 **Dev Accuracy:** 0.51 **Precision:** 0.50 **Recall:** 0.89 **F1:** 0.64	**True Pos:** 0.58 **True Neg:** 0.68 **Dev Accuracy:** 0.55 **Precision:** 0.54 **Recall:** 0.61 **F1:** 0.58

Figure 11: Our Baseline vs the three attention extensions on last linear layer CFIMDB. Since CFIMDB is a binary classification, we can use the F1 score.

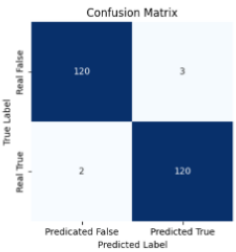
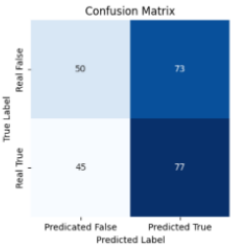
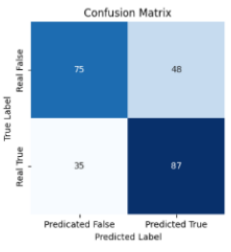
Baseline	Flash	Slide	Sink
			
True Pos: 0.98 **True Neg:** 0.98 **Dev Accuracy:** 0.98 **Precision:** 0.98 **Recall:** 0.98 **F1:** 0.98	**True Pos:** 0.98 **True Neg:** 0.98 **Dev Accuracy:** 0.98 **Precision:** 0.98 **Recall:** 0.98 **F1:** 0.98	**True Pos:** 0.63 **True Neg:** 0.41 **Dev Accuracy:** 0.52 **Precision:** 0.51 **Recall:** 0.63 **F1:** 0.57	**True Pos:** 0.71 **True Neg:** 0.61 **Dev Accuracy:** 0.66 **Precision:** 0.64 **Recall:** 0.71 **F1:** 0.68

Figure 12: Our Baseline vs the three attention extensions on full model CFIMDB.

A.4 FlashAttention Algorithm

This is the algorithm for standard dense self-attention, followed by the IO-aware FlashAttention algorithm

Algorithm 1 Standard Attention Implementation

Require: Matrices $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N \times d}$ in HBM.

1: Load $\mathbf{Q}, \mathbf{K}$ by blocks from HBM, compute $\mathbf{S} = \mathbf{Q}\mathbf{K}^\top$, write $\mathbf{S}$ to HBM.
2: Read $\mathbf{S}$ from HBM, compute $\mathbf{P} = (\mathbf{S})$, write $\mathbf{P}$ to HBM.
3: Load $\mathbf{P}$ and $\mathbf{V}$ by blocks from HBM, compute $\mathbf{O} = \mathbf{P}\mathbf{V}$, write $\mathbf{O}$ to HBM.
4: Return $\mathbf{O}$.

Algorithm 2 FlashAttention Algorithm

Require: Matrices $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N \times d}$ in HBM, on-chip SRAM of size M.

1: Set block sizes $B_c = \left\lceil \frac{M}{4d} \right\rceil, B_r = \min\left(\left\lceil \frac{M}{4d} \right\rceil, d\right)$.
2: Initialize $\mathbf{O} = (0)_{N \times d} \in \mathbb{R}^{N \times d}, \ell = (0)_N \in \mathbb{R}^N, m = (-\infty)_N \in \mathbb{R}^N$ in HBM.
3: Divide $\mathbf{Q}$ into $T_r = \left\lceil \frac{N}{B_r} \right\rceil$ blocks $\mathbf{Q}_1, \ldots, \mathbf{Q}_{T_r}$ of size $B_r \times d$ each, and divide $\mathbf{K}, \mathbf{V}$ in to $T_c = \left\lceil \frac{N}{B_c} \right\rceil$ blocks $\mathbf{K}_1, \ldots, \mathbf{K}_{T_c}$ and $\mathbf{V}_1, \ldots, \mathbf{V}_{T_c}$, of size $B_c \times d$ each.
4: Divide $\mathbf{O}$ into T_r blocks $\mathbf{O}_i, \ldots, \mathbf{O}_{T_r}$ of size $B_r \times d$ each, divide ℓ into T_r blocks $\ell_i, \ldots, \ell_{T_r}$ of size B_r each, divide m into T_r blocks $m_1, \ldots, m_{T_r}$ of size B_r each.
5: **for** $1 \le j \le T_c$ **do**
6: Load $\mathbf{K}_j, \mathbf{V}_j$ from HBM to on-chip SRAM.
7: **for** $1 \le i \le T_r$ **do**
8: Load $\mathbf{Q}_i, \mathbf{O}_i, \ell_i, m_i$ from HBM to on-chip SRAM.
9: On chip, compute $\mathbf{S}_{ij} = \mathbf{Q}_i \mathbf{K}_j^T \in \mathbb{R}^{B_r \times B_c}$.
10: On chip, compute $\tilde{m}_{ij} = \mathrm{rowmax}(\mathbf{S}_{ij}) \in \mathbb{R}^{B_r}$, $\tilde{\mathbf{P}}_{ij} = \exp(\mathbf{S}_{ij} - \tilde{m}_{ij}) \in \mathbb{R}^{B_r \times B_c}$ (pointwise), $\tilde{\ell}_{ij} = \mathrm{rowsum}(\tilde{\mathbf{P}}_{ij}) \in \mathbb{R}^{B_r}$.
11: On chip, compute $m_i^{\mathrm{new}} = \max(m_i, \tilde{m}_{ij}) \in \mathbb{R}^{B_r}$, $\ell_i^{\mathrm{new}} = e^{m_i - m_i^{\mathrm{new}}} \ell_i + e^{\tilde{m}_{ij} - m_i^{\mathrm{new}}} \tilde{\ell}_{ij} \in \mathbb{R}^{B_r}$.
12: Write $\mathbf{O}_i \leftarrow \mathrm{diag}(\ell_i^{\mathrm{new}})^{-1}(\mathrm{diag}(\ell_i) e^{m_i - m_i^{\mathrm{new}}} \mathbf{O}_i + e^{\tilde{m}_{ij} - m_i^{\mathrm{new}}} \tilde{\mathbf{P}}_{ij} \mathbf{V}_j)$ to HBM.
13: Write $\ell_i \leftarrow \ell_i^{\mathrm{new}}, m_i \leftarrow m_i^{\mathrm{new}}$ to HBM.
14: Return $\mathbf{O}$.