Fast Causal Attention with Dynamic Sparsity

Matteo Pagliardini [*1] **Daniele Paliotta** [*2] **Martin Jaggi** [1] **François Fleuret** [2]

Abstract

Transformer-based language models have found many diverse applications requiring them to process sequences of increasing length. For these applications, the causal self-attention—which is the only component scaling quadratically w.r.t. the sequence length—becomes a central concern. While many works have proposed schemes to sparsify the attention patterns and reduce the computational overhead of self-attention, those are often limited by implementation concerns and end up imposing a simple and static structure over the attention matrix. Conversely, implementing more dynamic sparse attention often results in runtimes significantly slower than computing the full attention using the Flash implementation. We extend FlashAttention to accommodate a large class of attention sparsity patterns that, in particular, encompass key/query dropping and hashing-based attention. This leads to implementations with no computational complexity overhead and a multi-fold runtime speedup on top of FlashAttention. Even with relatively low degrees of sparsity, our method improves visibly upon FlashAttention as the sequence length increases. Without sacrificing perplexity, we increase the training speed of a transformer language model by $2.0\times$ for sequences of $8k$ tokens.

1. Introduction

Many methods have been developed to mitigate the quadratic cost of self-attention in Transformers (Vaswani et al., 2017). A promising line of work consists in computing a dynamic modulation of a sub-part of the attention matrix. These methods are based, for instance, on dropping keys and queries (Kim et al., 2022) or using geometric hashing of the keys and queries to identify, at linear cost, sub-blocks of the attention matrix that carry most of the weight (Kitaev et al., 2020).

However, today's most successfully deployed models still rely on vanilla attention, in part thanks to the efficiency of FlashAttention (Dao et al., 2022). This implementation is mathematically equivalent to the vanilla attention proposed by Vaswani et al. (2017), but trades in additional compute for less memory I/O. While still avoiding a memory footprint quadratic with the sequence length, it delivers practical speedups of over $\times 5$ compared to a naive implementation.

Using an attention layer in an autoregressive model requires making it causal. This is achieved by applying a mask to the attention matrix, so that information cannot flow from the future to the past during training. While FlashAttention can deal with vanilla causal masks, it does not provide enough flexibility to be used for situations where the mask is not perfectly regular, that is, lower triangular. This in particular prevents using it for models that dynamically drop keys and queries or rely on geometric hashing, which results in irregular causal structures as illustrated in Fig. 1 and Fig. 4.

We propose an extension of FlashAttention—Sparse Causal Flash Attention (SCFA)— that addresses this constraint. Our contribution is threefold:

- We present the SCFA kernel, which relaxes the constraint that the causal mask has to be triangular. This kernel can handle any sparsity pattern that can be expressed with a range of keys per query, and any causal masking in the resulting sub-blocks. See § 3.
- We show that SCFA permits to revisit the promising paradigm of dynamic hash-based attention. We devise an algorithm that builds upon the fundamental idea of Reformer (Kitaev et al., 2020) to restrict the computation of the attention matrix over "hash collision blocks", but avoids both the high computational cost, and the approximate coverage of the hash collisions. See § 3.2.
- We propose a new approach implemented with SCFA that reduces computation by dynamically selecting, for each head, keys and queries to be removed from the attention operation, superseding existing methods that limited pruning to entire heads or entire queries/keys,

[*]Equal contribution [1]EPFL [2]University of Geneva. Correspondence to: Matteo Pagliardini <matteo.pagliardini@epfl.ch>, Daniele Paliotta <daniele.paliotta@unige.ch>.

ICML 2023 workshop on Efficient Systems for Foundation Models, Honolulu, Hawaii, USA. PMLR 202, 2023. Copyright 2023 by the author(s).

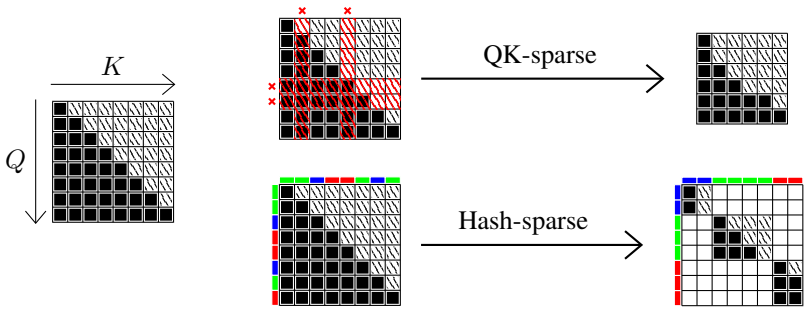


Figure 1. **Sparsification of the attention matrix.** In each depicted attention matrix, black areas indicate coefficients to compute, patterned areas those forced to zero due to the causal masking, and white areas coefficients that are ignored. We consider two dynamic strategies to sparsify the left attention matrix. The QK-sparse attention consists of dropping some keys and queries (top, the discarded keys and queries are indicated in red), and the Hash-sparse attention computes a hash code for each key and each query, and restricts the attention matrix to blocks of keys and queries of same hash code (bottom, the three hash values are indicated for each key or query with the colors blue/green/red). In both cases, the attention operation must be able to deal with sub-blocks of the attention matrix with a non-triangular causal mask.

due to the lack of an efficient fine-grained kernel implementation. See § 3.1.

Experimental evaluations show that SCFA can efficiently be used for a variety of sequence modeling tasks, and that our open-source implementation in the Triton language and compiler (Tillet et al., 2019) significantly outperforms FlashAttention as we increase the sparsity and for longer sequences. Moreover, unlike the hash-based attention introduced in Reformer by Kitaev et al. (2020), our hash-based SCFA not only implements the exact computation but also has a smaller runtime (see, § 3.2). Finally, we show that a prototype of queries and keys dropping can be implemented thanks to SCFA, and that the computational reduction is proportional to the fraction of query-key pairs dropped (see § 3.1).

2. Related Work

State-of-the-art sequence models have very high computational requirements. As a consequence, a lot of effort has been invested into developing methods to reduce the memory footprint in Transformers. Many efficient Transformer variants have been developed, with the main goal of taming the quadratic complexity of the attention mechanism (Tay et al., 2020). Several methods rely on kernelized attention (Katharopoulos et al., 2020; Choromanski et al., 2020), while others endow the Transformer with some auxiliary memory to increase the context (Wu et al., 2022; Borgeaud et al., 2021).

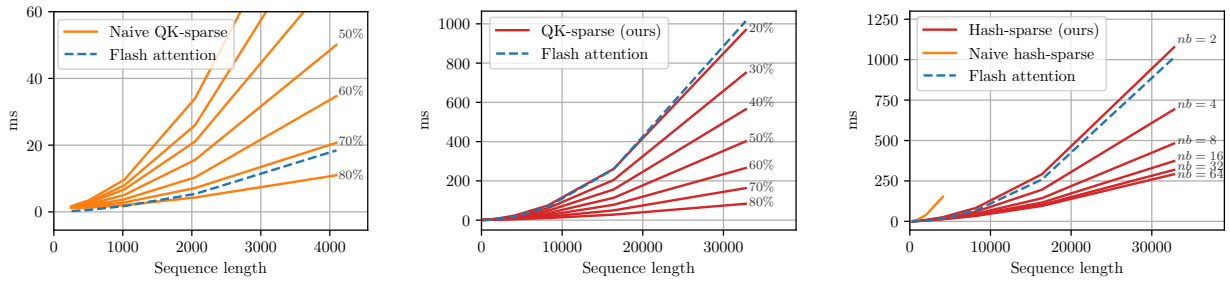
In many cases, leveraging sparsity in the attention matrix has proven useful. The Sparse Transformer (Child et al., 2019) works with a factorized sparse representation of the attention. They employ several sparse attention patterns, where each output position only computes weightings from a subset of input positions.

The Reformer (Kitaev et al., 2020) uses locality-sensitive-hashing (LSH) to sparsify the attention matrix and allow queries to restrict their context window to keys that collide with the same hash. However, to allow GPU-efficient processing, complex machinery has to be developed where the queries and keys are split into fixed-sized chunks, with the attention being applied only within the chunk and the immediate neighbor.

FlashAttention introduced by Dao et al. (2022) has recently gained a lot of popularity as an efficient, IO-aware exact attention implementation. FlashAttention uses tiling to avoid materializing the full attention matrix on slow GPU HBM, splitting the computation over blocks of query, key, and value vectors. FlashAttention has already reached wide adoption, as it's now available directly in Pytorch as of version 2.0. Additionally, FlashAttention supports very efficient block-sparse structures.

Hash Attention. When computing the attention matrix for a $T \times D$ query tensor $\boldsymbol{Q}$ and a $T \times D$ key tensor $\boldsymbol{K}$, we consider the matrix of dot-products $\boldsymbol{Q}\boldsymbol{K}^\top$, which can become impractical to compute for very long sequences. However, we are only interested in the row-wise $\mathrm{softmax}(\boldsymbol{Q}\boldsymbol{K}^\top)$, meaning that the contribution of the keys to every query is dominated by the ones with the highest similarity. Thus, restricting the attention computation to queries and keys with high similarity is a natural choice to reduce the computation.

Hash attention, introduced in the Reformer (Kitaev et al., 2020), allows to quickly select the closest key vectors for each query using locality-sensitive-hashing (LSH). In general, the LSH mechanism assigns a hash code to vectors with the requirement that vectors that are close in space are



(a) Naive QK-sparse (b) QK-sparse (c) Hash-sparse

Figure 2. **Fig.(a) & Fig.(b): Runtimes of full Flash-attention (Dao et al., 2022) and Query/Key dropping based sparsity.** We show the forward + backward runtimes. For sparse methods, we drop at random a percentage of keys and queries. **Fig.(a):** A naive implementation consisting in creating compact representations of the key, value, and query tensors by removing dropped keys and queries. As a result, the attention matrix is no longer triangular (see Fig. 1). We call the PyTorch `scaled_dot_product_attention` method with a custom but still causal mask. The non-triangular mask prevents FlashAttention to be used and only dropping more than 70% of the keys and queries seems to improve the runtime over attending the entire sequence using FlashAttention. **Fig.(b):** Our modification of FlashAttention improves the runtime. Reshaping the tensor induces an overhead which compensates the speed gain for shorter sequences. However, this offset is compensated by a strong margin as the sequence length increases. Our implementation allows significant gains over FlashAttention even for low levels of sparsity. The detailed runtimes for the forward and backward passes can be found in App. C. **Fig.(c): Runtimes of the full Flash-attention and hash-based sparsity.** Similarly to our QK-sparse method we show how a naive implementation fails to provide any speed gain. On the other hand, our modification of the basic FlashAttention method computes only what is required. While there is a cost to reordering the tensors based on the hash buckets, this cost is largely compensated for as the number of buckets nb increases, and as the sequence length increases.

mapped to the same hash with high probability. For the hash attention, the Reformer assumes a shared query-key space ($\boldsymbol{Q} = \boldsymbol{K}$). After computing the hashes, the queries are sorted according to their hash bucket. In the sorted attention matrix, pairs that fall into the same bucket cluster near the diagonal. In order to implement the LSH-attention scheme efficiently on GPU, the Reformer splits the queries into fixed-sized chunks. Queries belonging to the same chunk can attend to each other and one chunk back. This results in a suboptimal mechanism where there is no guarantee that the attention will capture exactly all of the elements that belong to the same bucket (See Fig. 3.c).

3. Method

We develop an efficient CUDA kernel written in Triton (Tillet et al., 2019) that maintains the careful memory management of FlashAttention but can handle a causal structure defined through an arbitrary indexing of the keys and the queries. In the case where this indexing consists of a binary decision to drop or not the head of a query/key, this corresponds to our QK-sparse kernels as described in § 3.1. In the case where the indexing corresponds to bucket indices e.g. obtained from hashing, this corresponds to our Hash-sparse kernel described in § 3.2.

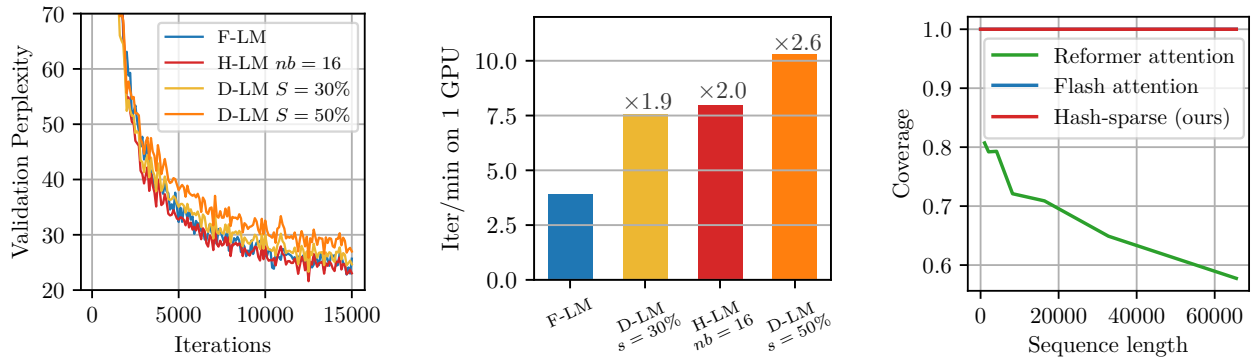
3.1. QK-Sparse Attention

Shrinking the attention matrix. Our QK-sparse attention kernel is best summarized in the first row of Fig. 1. Independently for each head, we decide to keep or drop keys and queries. We then remove dropped keys and queries to create smaller $\boldsymbol{Q}^c$, $\boldsymbol{K}^c$, and $\boldsymbol{V}^c$ tensors. Through this reduction we are left with a smaller attention matrix $\boldsymbol{A}^c$ which still has a causal structure in that indices for the queries and keys are increasing monotonically.

Leveraging non-triangular causal attention structure. Despite the advantageous structure of the smaller attention matrix, existing implementations fail to take advantage of it. Especially, as shown in Fig. 4 bottom-left, FlashAttention can leverage the causal structure when the causal mask is triangular but does not support any other shape. See App. B for more details on the kernel implementation.

3.2. Hash-Sparse Attention

Restructuring attention based on hashes. Independently for each head, we associate a bucket identifier to each key and query. We then need to reorder $\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}$ by sorting them along the sequence length dimension. As shown in the bottom row of Fig.1, this results in clusters of keys and queries with a similar hash index close to the diagonal. If the sorting is stable, then those blocks have a local causal structure in which the original indices (original position in the sequence) of keys and queries is a monotonic function within the block. This brings us to a case very similar to the previous one in section § 3.1, in that we now have the same structure but scattered by blocks within the full attention matrix.



(a) Iter to reach perplexity (b) Speed gains (c) Reformer coverage

Figure 3. **Fig.(a) & Fig.(b): Training Language Models (LM) on OpenWebText2 (Gao et al., 2020) using Hash-sparse (H-LM), QK-sparse (D-LM), or FlashAttention over the entire sequence (F-LM).** We train on sequences of length 8192 and use 16 buckets for our H-LM model and a sparsity of 30% and 50% for our D-LM models. We show that it is possible to use our proposed dynamic sparsity schemes to significantly gain in speed (Fig.(b)) without compromising the perplexity (Fig.(a)). In **Fig.(c)** we observe the impact of the fixed attention structure in Reformer: an increasing fraction of hash collisions are missed. Our approach maintains 100% exact coverage of collisions for all sequence lengths. See App. C for more details.

Taking advantage of the new structure. We would like to take advantage of the block structure and only compute attention for queries and keys falling within the same block while at the same time respecting causality. We adapt the FlashAttention kernel in a very similar way to our QK-sparse kernel. See Fig. 4 for a visual representation. See App. B for more details on the kernel implementation.

Overhead. Sorting and re-ordering $\boldsymbol{Q}$, $\boldsymbol{K}$ and $\boldsymbol{V}$ introduces some overhead. As shown in our experiments in § 3.2, this overhead is largely compensated for as the sequence length increases.

4. Experiments

In this section, we present our experimental setup and results. We show that (i) unlike naive implementations using existing libraries, our dynamic sparsity attention schemes can significantly improve over the FlashAttention runtime, (ii) this still holds in real-world sequence modeling tasks after factoring in all the non-attention operations, and (iii) it is possible to match—and sometimes outperform—the baselines in terms of perplexity while significantly gaining in speed.

Experimental setup We test our hash-based sparsity scheme on OpenWebText2 (Gao et al., 2020), enwik8 (Hutter, 2012), and MNIST (LeCun et al., 1998) for autoregressive image generation . We experiment with QK-dropping-based sparsity on OpenWebText2. See App. B.3 for details on models, baselines and hardware.

Hashing mechanism We adopt the same hashing procedure as (Kitaev et al., 2020). Namely, we use a shared query-key space, and we disallow queries to attend to themselves. We also adopt the LSH scheme from (Andoni et al., 2015). This allows us to pick the number of unique hash codes. We refer to *bucket* as the set of vectors that map to a certain hash.

Q/K-dropping mechanism used. We show that naively dropping heads for each key and query at random can already yield competitive results while significantly improving the runtime. While better dropping schemes could be devised, they are outside of the scope of this work.

Runtime performances in a vacuum. We test our implementation with different numbers of buckets nb and percentages of dropped keys, queries, and values. Buckets and dropped entries are randomly sampled. We compare with causal FlashAttention over the entire sequence. Importantly, to ensure a fair comparison, we take into account pre-processing and post-processing steps required to reshape the tensors for both methods. For our method this includes stable sorting by bucket index and transposing tensors, for the baseline only the transposition is required, see App. B.2 for detailed code. Fig. 2 summarises our findings. We observe large improvements in runtime as the number of buckets nb and the sequence length increases.

Language modeling on OpenWebText2. For sequences of length $T = 8192$ we train transformer LMs using FlashAttention (F-LM), and identical models replacing only the FlashAttention by our Hash-sparse and QK-sparse attention (H-LM and D-LM). Our findings are summarized in Fig. 3.

Comparison with Reformer. We compare our Hash-sparse implementation with the Reformer hashed attention. For all comparisons, we always equalize the average bucket size. Fig. 3.c shows that the fixed attention structure imposed by the Reformer does not allow to capture all of the hash colli-

sions, with the coverage decreasing steeply as the sequence length increases. On the contrary, our method is exact and covers every bucket collision in the attention matrix.

5. Conclusions

We develop and validate an efficient kernel that can make sparse attention based on dynamic patterns very fast. We hope that our contribution will inspire the community to research dynamic attention patterns in a way that is less constrained by a tight computational budget.

The computational cost of large attention models remains both a practical issue in scaling up to very large contexts, and a fundamental research question to close the gap between the energy usage of biological systems to that of GPU systems able to run very large models. Dynamically modulating the computation is an obvious direction to address this challenge.

References

Andoni, A., Indyk, P., Laarhoven, T., Razenshteyn, I., and Schmidt, L. Practical and optimal lsh for angular distance, 2015.

Behnke, M. and Heafield, K. Losing heads in the lottery: Pruning transformer attention in neural machine translation. In *EMNLP (1)*, pp. 2664–2674. Association for Computational Linguistics, 2020.

Bender, E. M., Gebru, T., McMillan-Major, A., and Shmitchell, S. On the dangers of stochastic parrots: Can language models be too big? In *Proceedings of the 2021 ACM Conference on Fairness, Accountability, and Transparency*, FAccT '21, pp. 610–623, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383097. doi: 10.1145/3442188.3445922. URL https://doi.org/10.1145/3442188.3445922.

Borgeaud, S., Mensch, A., Hoffmann, J., Cai, T., Rutherford, E., Millican, K., van den Driessche, G., Lespiau, J., Damoc, B., Clark, A., de Las Casas, D., Guy, A., Menick, J., Ring, R., Hennigan, T., Huang, S., Maggiore, L., Jones, C., Cassirer, A., Brock, A., Paganini, M., Irving, G., Vinyals, O., Osindero, S., Simonyan, K., Rae, J. W., Elsen, E., and Sifre, L. Improving language models by retrieving from trillions of tokens. *CoRR*, abs/2112.04426, 2021. URL https://arxiv.org/abs/2112.04426.

Child, R., Gray, S., Radford, A., and Sutskever, I. Generating long sequences with sparse transformers. *CoRR*, abs/1904.10509, 2019. URL http://arxiv.org/abs/1904.10509.

Choromanski, K., Likhosherstov, V., Dohan, D., Song, X., Gane, A., Sarlós, T., Hawkins, P., Davis, J., Mohiuddin, A., Kaiser, L., Belanger, D., Colwell, L. J., and Weller, A. Rethinking attention with performers. *CoRR*, abs/2009.14794, 2020. URL https://arxiv.org/abs/2009.14794.

Dao, T., Fu, D. Y., Ermon, S., Rudra, A., and Ré, C. Flashattention: Fast and memory-efficient exact attention with io-awareness. In *NeurIPS*, 2022.

Gao, L., Biderman, S., Black, S., Golding, L., Hoppe, T., Foster, C., Phang, J., He, H., Thite, A., Nabeshima, N., Presser, S., and Leahy, C. OpenWebText2 dataset, as part of 'the Pile: An 800gb dataset of diverse text for language modeling'. *arXiv preprint arXiv:2101.00027*, 2020.

Goyal, S., Choudhury, A. R., Raje, S., Chakaravarthy, V. T., Sabharwal, Y., and Verma, A. Power-bert: Accelerating BERT inference via progressive word-vector elimination. In *ICML*, volume 119 of *Proceedings of Machine Learning Research*, pp. 3690–3699. PMLR, 2020.

Hutter, M. The human knowledge compression contest. *URL http://prize. hutter1. net*, 6, 2012.

Katharopoulos, A., Vyas, A., Pappas, N., and Fleuret, F. Transformers are RNNs: Fast autoregressive transformers with linear attention. In *Proceedings of the International Conference on Machine Learning (ICML)*, pp. 5294–5303, 2020. URL https://fleuret.org/papers/katharopoulos-et-al-icml2020.pdf.

Kim, S., Shen, S., Thorsley, D., Gholami, A., Kwon, W., Hassoun, J., and Keutzer, K. Learned token pruning for transformers. In *KDD*, pp. 784–794. ACM, 2022.

Kitaev, N., Kaiser, L., and Levskaya, A. Reformer: The efficient transformer. In *ICLR*. OpenReview.net, 2020.

LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. Gradient-based learning applied to document recognition. *Proc. IEEE*, 86(11):2278–2324, 1998.

Li, J., Cotterell, R., and Sachan, M. Differentiable subset pruning of transformer heads. *Trans. Assoc. Comput. Linguistics*, 9:1442–1459, 2021.

Loshchilov, I. and Hutter, F. Decoupled weight decay regularization. In *ICLR (Poster)*. OpenReview.net, 2019.

Michel, P., Levy, O., and Neubig, G. Are sixteen heads really better than one? In *NeurIPS*, pp. 14014–14024, 2019.

Peng, H., Schwartz, R., Li, D., and Smith, N. A. A mixture of h - 1 heads is better than h heads. In *ACL*, pp. 6566–6577. Association for Computational Linguistics, 2020.

Peng, H., Pappas, N., Yogatama, D., Schwartz, R., Smith, N. A., and Kong, L. Random feature attention. In *ICLR*. OpenReview.net, 2021.

Qin, Z., Sun, W., Deng, H., Li, D., Wei, Y., Lv, B., Yan, J., Kong, L., and Zhong, Y. cosformer: Rethinking softmax in attention. In *ICLR*. OpenReview.net, 2022.

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., and Sutskever, I. Language models are unsupervised multitask learners, 2019.

Raganato, A., Scherrer, Y., and Tiedemann, J. Fixed encoder self-attention patterns in transformer-based machine translation. In *EMNLP (Findings)*, volume EMNLP 2020 of *Findings of ACL*, pp. 556–568. Association for Computational Linguistics, 2020.

Tay, Y., Dehghani, M., Bahri, D., and Metzler, D. Efficient transformers: A survey. *CoRR*, abs/2009.06732, 2020. URL https://arxiv.org/abs/2009.06732.

Tay, Y., Bahri, D., Metzler, D., Juan, D., Zhao, Z., and Zheng, C. Synthesizer: Rethinking self-attention for transformer models. In *ICML*, volume 139 of *Proceedings of Machine Learning Research*, pp. 10183–10192. PMLR, 2021a.

Tay, Y., Dehghani, M., Abnar, S., Shen, Y., Bahri, D., Pham, P., Rao, J., Yang, L., Ruder, S., and Metzler, D. Long range arena : A benchmark for efficient transformers. In *ICLR*. OpenReview.net, 2021b.

Tillet, P., Kung, H. T., and Cox, D. Triton: An intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, MAPL 2019, pp. 10–19, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450367196. doi: 10.1145/3315508.3329973. URL https://doi.org/10.1145/3315508.3329973.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. Attention is all you need. *CoRR*, abs/1706.03762, 2017. URL http://arxiv.org/abs/1706.03762.

Voita, E., Talbot, D., Moiseev, F., Sennrich, R., and Titov, I. Analyzing multi-head self-attention: Specialized heads do the heavy lifting, the rest can be pruned. In *ACL (1)*, pp. 5797–5808. Association for Computational Linguistics, 2019.

Wang, H., Zhang, Z., and Han, S. Spatten: Efficient sparse attention architecture with cascade token and head pruning. In *HPCA*, pp. 97–110. IEEE, 2021.

Weidinger, L., Mellor, J., Rauh, M., Griffin, C., Uesato, J., Huang, P., Cheng, M., Glaese, M., Balle, B., Kasirzadeh, A., Kenton, Z., Brown, S., Hawkins, W., Stepleton, T., Biles, C., Birhane, A., Haas, J., Rimell, L., Hendricks, L. A., Isaac, W., Legassick, S., Irving, G., and Gabriel, I. Ethical and social risks of harm from language models. *CoRR*, abs/2112.04359, 2021.

Wu, Y., Rabe, M. N., Hutchins, D., and Szegedy, C. Memorizing transformers, 2022.

Xia, M., Zhong, Z., and Chen, D. Structured pruning learns compact and accurate models. In *ACL (1)*, pp. 1513–1528. Association for Computational Linguistics, 2022.

Zheng, L., Wang, C., and Kong, L. Linear complexity randomized self-attention mechanism. In *ICML*, volume 162 of *Proceedings of Machine Learning Research*, pp. 27011–27041. PMLR, 2022.

Zhou, W., Ge, T., Wei, F., Zhou, M., and Xu, K. Scheduled drophead: A regularization method for transformer models. In *EMNLP (Findings)*, volume EMNLP 2020 of *Findings of ACL*, pp. 1971–1980. Association for Computational Linguistics, 2020.

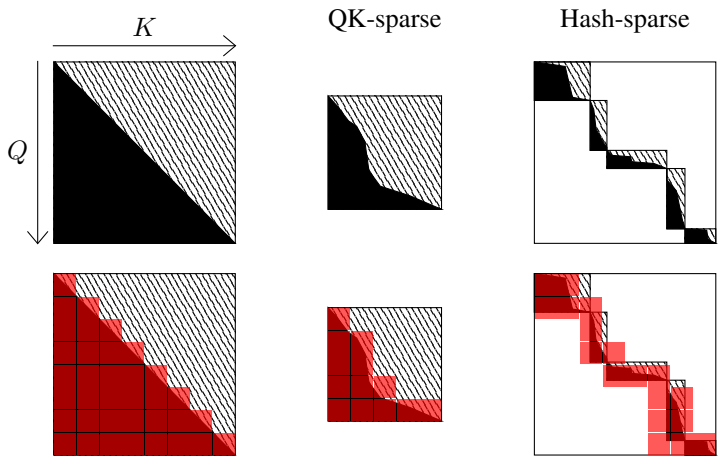


Figure 4. **SCFA computation patterns.** In each depicted attention matrix, black areas indicate coefficients to compute, patterned areas are those forced to zero due to the causal masking, and white areas coefficients that are ignored. The red squares in the bottom matrices show the tiles actually computed by our SCFA kernel. In the regular case (left), this coincides with the behavior of FlashAttention. However, in the case of irregular causal masking due to keys/queries dropping (center) or in the case of irregular causal masking and band block sparsity due to hashing (right), FlashAttention does not provide means to compute a fine-grain subset of the attention matrix.

A. Additional Background

A.1. Structured attention

With a focus on performances on downstream applications, Raganato et al. (2020) show that it is possible to replace all but one attention head with static attentive patterns only relying on the position, e.g. attending to the previous token. Similarly, Tay et al. (2021a) investigate removing entirely the dot product attention and instead let the input token alone predict its attention pattern over the entire sequence, or use a random attention pattern. While they show their approach can be competitive on certain downstream task, results on language modeling seems to indicate those attention patterns are not expressive enough to model natural language. Peng et al. (2021) propose Random Feature Attention (RFA) which relies on kernel methods to approximate the softmax atttention and achieve a linear computational complexity w.r.t. the sequence length. A follow up work by Zheng et al. (2022) augments RFA with randomized attention to build an unbiased estimator of the softmax attention. Instead of looking for a linear approximation of the softmax, Qin et al. (2022) propose to replace the softmax by a linear projection kernel and a cos-based re-weighting mechanism which scales linearly with the sequence length. While kernel based method seem like a good compromise in terms of speed vs. performance, they have been shown to underperform on certain downstream tasks (Tay et al., 2021b). Compared to those methods, we are not trying to replace or find an estimator of the softmax attention, we instead provide an efficient way to leverage different forms of sparsity of the softmax attention matrix. Our speed gains do not come from a conceptually different way to compute the attention but simply deciding not to compute certain regions of the softmax attention matrix.

A.2. Pruning

The large body of works on pruning mostly focus on faster inference on downstream tasks. As a result, many methods are modifying the training process to facilitate pruning, often resulting in slower training.

Pruning heads in transformers. Many works have investigated dropping entire attention heads in transformer architectures. Let the entire attention matrix $\boldsymbol{A}$ be of shape $B \times H \times T \times T$ with B the batch size, H the number of heads, and T the sequence length. Dropping entire heads imposes an implicit structure over $\boldsymbol{A}$ which is now of shape $B \times H' \times T \times T$, with H' the number of non-dropped heads. Michel et al. (2019) and Voita et al. (2019) both observe—in the context of Neural Machine Translation (NMT)—that a large fraction of heads can be dropped without significantly impacting performance. While their goal is not primarily to speed up training, many of methods following on this insight try to incorporate a sparsifying mechanism during training which facilitate dropping heads at test time (Behnke & Heafield, 2020; Peng et al.,

2020; Xia et al., 2022; Li et al., 2021). Still limited to NMT and downstream classification tasks, Zhou et al. (2020) found a regularizing effect of dropping heads during training. In comparison, less works have investigated dropping head in the context of language modeling, with text generations applications in mind. In comparison to those works, we propose a way to take advantage of dynamic sparsity structures, which is much more general but still includes head dropping. We show the potential of our work to enable speedup *during training*. Our work is orthogonal to head-dropping mechanisms and could be used in addition to those.

Pruning tokens in transformers. While dropping heads can be seen as dropping the model parameters generating keys, queries and values for those heads, another more recent line of work looked into dropping tokens. Dropping entire tokens imposes an implicit structure over the attention matrix $\boldsymbol{A}$ which is now of shape $B \times H \times T_Q \times T_{KV}$, with T_Q and T_{KV} the number remaining queries and keys. Goyal et al. (2020) obtain faster inference on downstream tasks by using an attention based scoring mechanism to eliminate redundant input vectors. Wang et al. (2021) develop a joint algorithm-architecture framework which speeds up inference for downstream classifications tasks and language generation tasks. Their method includes head and token pruning along with specific hardware optimizations. Our work can be used to implement those approaches and does not require custom hardware accelerators. We moreover allow dropping individual heads instead of entire tokens.

B. Details On The Implementation

We here give implementation details for experiments of § 4. First, in App. B.1, we describe in more details the implementations of our custom Triton kernels introduced in § 3. Secondly in App. B.2 we provide the python code used in our runtime benchmarks of § 4, including pre and post processing steps required to reshape and re-order tensors. Lastly in § B.3 and § B.4 we give more details on the experimental setup and hyperparameters used in our sequence modeling experiments of § 4. The code for all our experiments can be found via the following link: `https://anonymous.4open.science/r/dynamic-sparse-flash-attention-1C84`.

B.1. Triton Kernels

QK-Sparse Triton Kernel. In Alg. 1 we detail the core of the QK-sparse algorithm from § 3.1. This algorithm is computing the softmax attention result only for one block of queries, corresponding to one head. In practice this algorithm would be run in parallel for all blocks of queries for all heads. We find the index of the last relevant tile to compute by iterating over block of key indices and comparing them with the largest query index for the current block of queries. This works as $\boldsymbol{q}_{idx}$ and $\boldsymbol{k}_{idx}$ have a monotonic structure thanks to the stable sort used when reshaping the tensors (see the pre-processing code in App. B.2 for more details). We apply causal masking locally by looking at $\boldsymbol{q}_{idx}$ and $\boldsymbol{k}_{idx}$, q query can only attend to past keys: `mask = q_idx[:, None] >= k_idx[None, :]` . The backward pass relies exactly on the same trick, we first iterate over query indices to find the starting block of queries.

Algorithm 1 Forward Pass for the QK-sparse kernel

Require: Matrix $\boldsymbol{Q} \in \mathbb{R}^{N_Q \times d}$, matrices $\boldsymbol{K}, \boldsymbol{V} \in \mathbb{R}^{N_{KV} \times d}$, index matrices $\boldsymbol{Q}_{idx} \in \mathbb{R}^{N_Q \times d}$ and $\boldsymbol{K}_{idx} \in \mathbb{R}^{N_{KV} \times d}$,softmax scaling constant $\tau \in \mathbb{R}$, softmax statistics vectors $\boldsymbol{M} \in \mathbb{R}^{N_Q}$ and $\boldsymbol{L} \in \mathbb{R}^{N_Q}$, output tensor $\boldsymbol{O} \in \mathbb{R}^{N_Q \times d}$, query block size B_m, key block size B_n, starting query index start_m.

Initialize $\boldsymbol{o} \leftarrow (0)_{B_m \times d} \in \mathbb{R}^{B_m \times d}, \ell \leftarrow (0)_{B_m} \in \mathbb{R}^{B_m}, \boldsymbol{m} \leftarrow (-\infty)_{B_m} \in \mathbb{R}^{B_m}$
Load current block of queries $\boldsymbol{q} \leftarrow \boldsymbol{Q}[\text{start}_m : \text{start}_m + B_m, :]$
Load current block of query indices $\boldsymbol{q}_{idx} \leftarrow \boldsymbol{Q}_{idx}[\text{start}_m : \text{start}_m + B_m]$
end $\leftarrow 0$
for start_n in range(0, N_{KV}, B_m) **do**
  Load block of key indices: $\boldsymbol{k}_{idx} \leftarrow \boldsymbol{K}_{idx}[\text{start}_n : \text{start}_n + B_n]$
  if $\min(\boldsymbol{k}_{idx}) \leq \max(\boldsymbol{q}_{idx})$ **then**
    end $\leftarrow$ end $+ 1$
  end if
end for
for n in range(0, end) **do**
  Start of current key block $\text{start}_n \leftarrow n * B_n$
  Load current block of values $\boldsymbol{v} \leftarrow \boldsymbol{V}[\text{start}_n : \text{start}_n + B_n, :]$
  Load current block of keys $\boldsymbol{k} \leftarrow \boldsymbol{K}[\text{start}_n : \text{start}_n + B_n, :]$
  Load current block of key indices $\boldsymbol{k}_{idx} \leftarrow \boldsymbol{K}_{idx}[\text{start}_n : \text{start}_n + B_n]$
  Compute inner product $\mathbf{qk} \leftarrow \tau \boldsymbol{q}.\boldsymbol{k}^\top$
  Apply causal mask $\mathbf{qk} \leftarrow \text{MASK}(\mathbf{qk}, \boldsymbol{q}_{idx}, \boldsymbol{k}_{idx})$
  Update softmax statistics $\ell, \boldsymbol{m}, \boldsymbol{o} \leftarrow \text{UPDATE_STATS}(\ell, \boldsymbol{m}, \mathbf{qk}, \boldsymbol{o}, \boldsymbol{v})$
end for
Write $\boldsymbol{O}[\text{start}_m : \text{start}_m + B_m, :] \leftarrow \boldsymbol{o}$
Write $\boldsymbol{L}[\text{start}_m : \text{start}_m + B_m] \leftarrow \ell$
Write $\boldsymbol{M}[\text{start}_m : \text{start}_m + B_m] \leftarrow \boldsymbol{m}$

Hash-Sparse Triton Kernel. In Alg. 2 we detail the core of the Hash-sparse algorithm from § 3.2. This algorithm is computing the softmax attention result only for one block of queries, corresponding to one head. In practice this algorithm would be run in parallel for all blocks of queries for all heads. We find the index of the first and last relevant tiles to compute by iterating over block of key hashes and comparing them with the largest and smallest query hashes for the current block of queries. This works as we reshaped our $\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}$ tensors by sorting them by their hash values (see the pre-processing code in App. B.2 for more details). In addition to causal masking, we also enforce attention to happen within the same bucket: `mask = (q_idx[:, None] >= k_idx[None, :]) & (q_hash[:, None] == k_hash[None, :])`. In our experiments we often replace >= by > to prevent a query to attend to itself as in the Reformer. As a side note, for most application it would also be fine to only enforce causal masking and allow attention across buckets within a tile. While this could add some serendipity in the attention computation, some applications might require masking based on hash. The backward pass relies exactly on the same trick, we first iterate over query indices to find the starting and end blocks of queries.

Algorithm 2 Forward Pass for the Hash-sparse kernel

Require: Matrix $\boldsymbol{Q} \in \mathbb{R}^{N_Q \times d}$, matrices $\boldsymbol{K}, \boldsymbol{V} \in \mathbb{R}^{N_{KV} \times d}$, index matrices $\boldsymbol{Q}_{idx} \in \mathbb{R}^{N_Q \times d}$ and $\boldsymbol{K}_{idx} \in \mathbb{R}^{N_{KV} \times d}$, matrices containing the hash values $\boldsymbol{Q}_{hash} \in \mathbb{R}^{N_Q \times d}$ and $\boldsymbol{K}_{hash} \in \mathbb{R}^{N_{KV} \times d}$,softmax scaling constant $\tau \in \mathbb{R}$, softmax statistics vectors $\boldsymbol{M} \in \mathbb{R}^{N_Q}$ and $\boldsymbol{L} \in \mathbb{R}^{N_Q}$, output tensor $\boldsymbol{O} \in \mathbb{R}^{N_Q \times d}$, query block size B_m, key block size B_n, starting query index start_m.

Initialize $\boldsymbol{o} \leftarrow (0)_{B_m \times d} \in \mathbb{R}^{B_m \times d}, \ell \leftarrow (0)_{B_m} \in \mathbb{R}^{B_m}, \boldsymbol{m} \leftarrow (-\infty)_{B_m} \in \mathbb{R}^{B_m}$
Load current block of queries $\boldsymbol{q} \leftarrow \boldsymbol{Q}[\text{start}_m : \text{start}_m + B_m, :]$
Load current block of query indices $\boldsymbol{q}_{idx} \leftarrow \boldsymbol{Q}_{idx}[\text{start}_m : \text{start}_m + B_m]$
Load current block of query hashes $\boldsymbol{q}_{hash} \leftarrow \boldsymbol{Q}_{hash}[\text{start}_m : \text{start}_m + B_m]$
start $\leftarrow 0$
$\text{end}_{hash} \leftarrow 0$
for start_n in range(0, N_{KV}, B_m) **do**
  Load block of key hashes: $\boldsymbol{k}_{hash} \leftarrow \boldsymbol{K}_{hash}[\text{start}_n : \text{start}_n + B_n]$
  if $\min(\boldsymbol{k}_{hash}) \leq \max(\boldsymbol{q}_{hash})$ **then**
    $\text{end}_{hash} \leftarrow \text{end}_{hash} + 1$
  end if
  if $\max(\boldsymbol{k}_{hash}) < \min(\boldsymbol{q}_{hash})$ **then**
    start $\leftarrow$ start $+ 1$
  end if
end for
end $\leftarrow \text{end}_{hash}$
for j in range(start, end_{hash}) **do**
  Load block of key indices: $\boldsymbol{k}_{idx} = \boldsymbol{K}_{idx}[jB_n : (j+1)B_n]$
  if $\min(\boldsymbol{k}_{idx}) \leq \max(\boldsymbol{q}_{idx})$ **then**
    end $\leftarrow j + 1$
  end if
end for
for n in range(start, end) **do**
  Start of current key block $\text{start}_n \leftarrow n * B_n$
  Load current block of values $\boldsymbol{v} \leftarrow \boldsymbol{V}[\text{start}_n : \text{start}_n + B_n, :]$
  Load current block of keys $\boldsymbol{k} \leftarrow \boldsymbol{K}[\text{start}_n : \text{start}_n + B_n, :]$
  Load current block of key indices $\boldsymbol{k}_{idx} \leftarrow \boldsymbol{K}_{idx}[\text{start}_n : \text{start}_n + B_n]$
  Load current block of key hashes $\boldsymbol{k}_{hash} \leftarrow \boldsymbol{K}_{hash}[\text{start}_n : \text{start}_n + B_n]$
  Compute inner product $\mathbf{qk} \leftarrow \tau \boldsymbol{q}.\boldsymbol{k}^\top$
  Apply causal and bucket mask $\mathbf{qk} \leftarrow \text{MASK}(\mathbf{qk}, \boldsymbol{q}_{idx}, \boldsymbol{k}_{idx}, \boldsymbol{q}_{hash}, \boldsymbol{k}_{hash})$
  Update softmax statistics $\ell, \boldsymbol{m}, \boldsymbol{o} \leftarrow \text{UPDATE_STATS}(\ell, \boldsymbol{m}, \mathbf{qk}, \boldsymbol{o}, \boldsymbol{v})$
end for
Write $\boldsymbol{O}[\text{start}_m : \text{start}_m + B_m, :] \leftarrow \boldsymbol{o}$
Write $\boldsymbol{L}[\text{start}_m : \text{start}_m + B_m] \leftarrow \ell$
Write $\boldsymbol{M}[\text{start}_m : \text{start}_m + B_m] \leftarrow \boldsymbol{m}$

Accumulating softmax statistics while avoiding NaNs. The following is a brief summary of how the FlashAttention algorithm (Dao et al., 2022) proposes to accumulate softmax statistics when iterating over blocks of keys. Given an input vector $\boldsymbol{x}$, our goal is to compute $\text{softmax}(\boldsymbol{x}) \triangleq e^{\boldsymbol{x} - \max(\boldsymbol{x})} / \sum_i e^{\boldsymbol{x}_i - \max(\boldsymbol{x})}$. Let $f(\boldsymbol{x}, m) \triangleq e^{\boldsymbol{x} - m}$, and $\ell(\boldsymbol{x}, m) \triangleq \sum_i f(\boldsymbol{x}, m)$. Hence:

$$\text{softmax}(\boldsymbol{x}) = \frac{f(\boldsymbol{x}, \max(\boldsymbol{x}))}{\ell(\boldsymbol{x}, \max(\boldsymbol{x}))}$$

Given a vector $\boldsymbol{x} \triangleq [\boldsymbol{x}_1, \boldsymbol{x}_2]$, let $m_g = \max(\boldsymbol{x})$ (global max), and $m_1 = \max(\boldsymbol{x}_1)$, we notice:

$$\begin{aligned}\text{softmax}(x) &= \frac{f(\boldsymbol{x}, m_g)}{\ell(\boldsymbol{x}, m_g)} \\ &= \frac{f(\boldsymbol{x}_1, m_1)}{e^{m_1 - m_g}\ell(\boldsymbol{x}_1, m_1) + \ell(\boldsymbol{x}_2, m_g)} + \frac{f(\boldsymbol{x}_2, m_g)}{e^{m_1 - m_g}\ell(\boldsymbol{x}_1, m_1) + \ell(\boldsymbol{x}_2, m_g)}\end{aligned}$$

Therefore, if we have m_1, $\ell(\boldsymbol{x}_1, m_1)$, and $\boldsymbol{r} = \frac{f(\boldsymbol{x}_1, m_1)}{l(\boldsymbol{x}_1, m_1)}$, we can update the softmax statistics for a new block of entries $\boldsymbol{x}_2$ by following the following steps:

1. Compute new global max: $m_g = \max(m_1, \max(\boldsymbol{x}_2))$
2. Compute $f(\boldsymbol{x}_2, m_g) = e^{\boldsymbol{x}_2 - m_g}$
3. Compute $\ell(\boldsymbol{x}_2, m_g) = \sum_i f(\boldsymbol{x}_2, m_g)$
4. Compute new $\ell(\boldsymbol{x}, m_g)$: $\ell(\boldsymbol{x}, m_g) = e^{m_1 - m_g}\ell(\boldsymbol{x}_1, m_1) + \ell(\boldsymbol{x}_2, m_g)$
5. Correct running softmax result: $\boldsymbol{r} = \boldsymbol{r}\frac{\ell(\boldsymbol{x}_1, m_1)}{\ell(\boldsymbol{x}, m_g)}$
6. Add contribution from $\boldsymbol{x}_2$ to $\boldsymbol{r}$: $\boldsymbol{r} = \boldsymbol{r} + \frac{f(\boldsymbol{x}_2, m_g)}{\ell(\boldsymbol{x}, m_g)}$

In case all the keys are masked for a given query, we would have $\max(\boldsymbol{x}_2) = -\infty$, given that the first m_1 is initialized to $-\infty$ as well (see Alg. 1 and Alg. 2) the fourth and second steps above would be undefined and result in NaN values. We solve the problem by replacing $-\infty$ values in m_g by 0s when doing those two steps. Another potential issue is in step three: when a query has no matching key in $\boldsymbol{x}_2$ then $\ell(\boldsymbol{x}_2, m_g)$ is now 0, which generate $+\infty$ in step five. This is an issue as we process keys by blocks, and if there are no keys for a query in the current block, we might find matching keys in following blocks. Adding ∞ values to r would prevent us to accumulate statistics later on. To get our desired behaviour and have 0s when queries have no matching key we replace ∞ values by 1s in $\ell(\boldsymbol{x}, m_g)$ during steps five and six. We summarize those steps in Alg. 3.

Algorithm 3 UPDATE_STATS method

Require: Vector $\ell \in \mathbb{R}^N$, vector $\boldsymbol{m} \in \mathbb{R}^N$, matrix of masked inner products $\mathbf{qk} \in \mathbb{R}^{N \times N}$, output buffer $\boldsymbol{o} \in \mathbb{R}^{B_m \times d}$, block of values $\boldsymbol{v} \in \mathbb{R}^{B_m \times d}$

Compute new global max (step 1) $\boldsymbol{m}_{new} \leftarrow \max(\text{rowmax}(\mathbf{qk}), \boldsymbol{m})$
Replace $-\infty$ by 0: $\hat{\boldsymbol{m}}_{new} \leftarrow \text{WHERE}(\boldsymbol{m}_{new} == -\infty, 0, \boldsymbol{m}_{new})$
Compute step 2: $\boldsymbol{p} \leftarrow e^{\mathbf{qk} - \hat{\boldsymbol{m}}_{new}[:, None]}$ # row-wise subtraction
Compute step 3: $\ell_2 \leftarrow \text{rowsum}(\boldsymbol{p})$
Compute step 4: $\ell_{new} \leftarrow e^{\boldsymbol{m} - \hat{\boldsymbol{m}}}\ell + \ell_2$
Compute $\boldsymbol{z} \leftarrow \frac{1}{\ell_{new}}$
Replace ∞ by 1: $\boldsymbol{z} \leftarrow \text{WHERE}(\boldsymbol{z} == \infty, 1, \boldsymbol{z})$
Update $\boldsymbol{p} \leftarrow \boldsymbol{p} \times \boldsymbol{z}[:, None]$ # row-wise multiplication
Correct running softmax output (step 5) $\boldsymbol{o} \leftarrow \boldsymbol{o} \times (\ell\boldsymbol{z})[:, None]$ # row-wise multiplication
Add contribution from current block (step 6) $\boldsymbol{o} \leftarrow \boldsymbol{o} + \boldsymbol{p}.\boldsymbol{v}\ \ell_{new}, \boldsymbol{m}_{new}, \boldsymbol{o}$

Hyperparameters. We extend the implementation of FlashAttention available in the Triton tutorial. In our benchmarks, we use a batch size $B = 4$, 48 heads of 64 dimensions each.

B.2. Runtimes in a Vacuum

Baseline implementation. We use Pytorch's FlashAttention implementation provided by the `torch.nn.functional.scaled_dot_product_attention` function. To ensure fairness, we assume that all benchmarked functions receive a tensor $\boldsymbol{Q}$ of shape `(BATCH, CTX_Q, H, D_HEAD)`, and tensors $\boldsymbol{K}, \boldsymbol{V}$ of shapes `(BATCH, CTX_KV, H, D_HEAD)`, where `BATCH` is the batch size, `CTX_Q` is the number of queries, `CTX_KV` is the

number of keys and values, H is the number of heads, and D_HEAD is number of dimensions per head. For this reason the only pre and post processing steps required are transposing the input and output tensors.

Listing 1. pytorch_full_flashattention function applying the FlashAttention algorithm on the entire sequence (no sparsity).

```
def pytorch_full_flashattention(q, k, v):

    BATCH, N_CTX, H, D_HEAD = q.shape

    q = q.transpose(1, 2) # (BATCH, H, N_CTX_Q, D_HEAD)
    k = k.transpose(1, 2) # (BATCH, H, N_CTX_KV, D_HEAD)
    v = v.transpose(1, 2) # (BATCH, H, N_CTX_KV, D_HEAD)

    y = torch.nn.functional.scaled_dot_product_attention(q, k, v, dropout_p=0.0, attn_mask=None, is_causal=True)
    return y.transpose(1,2).contiguous()
```

Our proposed interface. We propose the following interface to orchestrate between the Hash-sparse and the QK-sparse implementations:

Listing 2. dynamic_sparse_attention interface

```
def dynamic_sparse_attention(q, k, v, q_idx, k_idx, sm_scale=None, sparsity_mode='hash'):
    """
    Keyword arguments:
    q: query tensor of shape (BATCH, N_CTX_Q, H, D_HEAD)
    k: key tensor of shape (BATCH, N_CTX_KV, H, D_HEAD)
    v: value tensor of shape (BATCH, N_CTX_KV, H, D_HEAD)
    q_idx: tensor of shape (BATCH, N_CTX_Q, H) for each sequence in the batch, for each query in the sequence, for each head,
        represents either the bucket index if sparsity_mode=='hash' or the whether to keep that given head if sparsity_mode=='qk'.
        The type should be torch.int32 if sparsity_mode=='hash' and torch.float if sparsity_mode=='qk'.
    k_idx: tensor of shape (BATCH, N_CTX_KV, H) for each sequence in the batch, for each key in the sequence, for each head,
        represents either the bucket index if sparsity_mode=='hash' or the whether to keep that given head if sparsity_mode=='qk'.
        The type should be torch.int32 if sparsity_mode=='hash' and torch.float if sparsity_mode=='qk'
    sm_scale: normalization constant, 1/sqrt(D_HEAD) unless specified
    sparsity_mode: 'hash' to select the hash-sparse implementation and 'qk' for the qk-sparse implementation
    """

    if sm_scale is None:
        sm_scale = 1.0 / math.sqrt(q.size(-1))

    if sparsity_mode == 'hash':
        return hash_sparse_attention(q, k, v, q_hash=q_idx, k_hash=k_idx, sm_scale=sm_scale)
    elif sparsity_mode == 'qk':
        return qk_sparse_attention(q, k, v, q_keep=q_idx, k_keep=k_idx, sm_scale=sm_scale)
    else:
        raise KeyError(f"Unknown sparsity_mode: '{sparsity_mode}', should be in  ['hash', 'qk']")
```

Pre & post processing steps for Hash-sparse. In addition to having to transpose the $\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}$ tensors. The preprocessing steps consist in re-ordering the $\boldsymbol{Q}$, $\boldsymbol{K}$ and $\boldsymbol{V}$ tensors based on bucket indices in q_hash and k_hash. We keep track of the original position of the queries, keys and values by storing the indices given by the sorting operations. Importantly, we use stable sorts to ensure the queries, keys and values are sorted within each bucket. The following code is showing how we implemented all those steps using Pytorch:

Listing 3. hash_sparse_attention function showing pre and post processing steps for the Hash-sparse algorithm.

```
def hash_sparse_attention(q, k, v, q_hash, k_hash, sm_scale):
    assert q_hash.dtype == torch.int32 and k_hash.dtype == torch.int32

    BATCH, N_CTX_Q, H, D_HEAD = q.shape

    q = q.transpose(1, 2) # (BATCH, H, N_CTX_Q, D_HEAD)
    k = k.transpose(1, 2) # (BATCH, H, N_CTX_KV, D_HEAD)
    v = v.transpose(1, 2) # (BATCH, H, N_CTX_KV, D_HEAD)
    q_hash = q_hash.transpose(1, 2).contiguous() # (BATCH, H, N_CTX_Q)
```

```
    k_hash = k_hash.transpose(1, 2).contiguous() # (BATCH, H, N_CTX_KV)

    # Re-order the queries,keys,values according q_hash and k_hash
    q_hash = q_hash.sort(dim=-1, stable=True) # q_hash.shape = (BATCH, H, N_CTX_Q), stable sort to keep time ordering within a bucket
    k_hash = k_hash.sort(dim=-1, stable=True) # k_hash.shape = (BATCH, H, N_CTX_KV)

    q_idx = q_hash.indices
    k_idx = k_hash.indices

    q_hash = q_hash.values
    k_hash = k_hash.values

    q_idx_extended = q_idx.unsqueeze(-1).expand_as(q)
    k_idx_extended = k_idx.unsqueeze(-1).expand_as(k)

    q = torch.gather(q, dim=-2, index=q_idx_extended).contiguous()
    k = torch.gather(k, dim=-2, index=k_idx_extended).contiguous()
    v = torch.gather(v, dim=-2, index=k_idx_extended).contiguous()

    y = hash_sparse_attention_kernel(q, k, v, q_idx, k_idx, q_hash, k_hash, sm_scale)
    y = torch.zeros((BATCH, H, N_CTX_Q, D_HEAD), dtype=q.dtype, device=q.device).scatter(dim=2, index=q_idx_extended, src=y).transpose(1,2).contiguous()
    return y
```

Pre & post processing steps for QK-sparse. In addition to having to transpose the $\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}$ tensors. The preprocessing steps consist in removing dropped keys, values and queries from $\boldsymbol{K}$, $\boldsymbol{V}$ and $\boldsymbol{Q}$. The sorting operations need to be stable to keep the original time ordering within the remaining keys and queries. Moreover, the index tensor has to be padded so our kernel can rely on those indices to compute which tile it should and shouldn't compute.

Listing 4. `qk_sparse_attention` function showing pre and post processing steps for the QK-sparse algorithm.

```
def compact(keep_tensor, x, index=None):
  """ Build a compact representation of x
  Keyword arguments:
  x: input tensor to compact, x.shape = (BATCH, N_CTX, H, D_HEAD)
  keep_tensor: float tensor of shape (BATCH, N_CTX, H) containing a 1 when the head is kept, else 0
  """
  BATCH, T, H, D_HEAD = x.shape
  if index is None:
    with torch.no_grad():
        indices_per_head = keep_tensor.sum(dim=-2)
        buffer_size = indices_per_head.max().int() # first sum computes the num of non-killed elem per head, we take to max of that
        # sorting: it is very important that the sorting is stable, else we cannot use causal masking
        sorted = keep_tensor.sort(dim=-2, descending=True, stable=True) # sorted.indices.shape == (BATCH x T x H) , now sorted over sequence T
        index = sorted.indices[:,:buffer_size,:] # (BATCH x buffer_size x H) expand indices to cover all the dimensions for each heads
  else:
    indices_per_head = None
  compact_x = x.gather(dim=-3, index=index.unsqueeze(-1).expand(-1,-1,-1,D_HEAD)) # (BATCH x buffer_size x H x D_HEAD) / expand indices to cover all the dimensions for each heads
  return compact_x, index, indices_per_head


@torch.no_grad()
def pad_index(index, indices_per_head, pad_idx=-1):
  """ Pad the index tensor to comply with the kernel, returns a copy.
  Keyword arguments:
  index: original index tensor to pad given by `compact`, index.shape = (BATCH, buffer_size, H). For each batch and timestep, reprsents the head idx it's originating from.
  indices_per_head: of shape (BATCH, H), for each head, contains how many indices have not been dropped.
  """
  BATCH, buffer_size, H = index.shape
  index_copy = torch.clone(index).type(torch.int32)
  mask = torch.arange(buffer_size, device=index.device).view(1,-1,1).expand(BATCH,buffer_size,H) >= indices_per_head.view(BATCH,1,-1)
  index_copy[mask] = pad_idx
  return index_copy


def qk_sparse_attention(q, k, v, q_keep, k_keep, sm_scale):
    assert q_keep.dtype == torch.float and k_keep.dtype == torch.float

```

```
    BATCH, N_CTX_Q, H, D_HEAD = q.shape

    # Building compact representations
    q_c, q_idx, iph_q = compact(q_keep, q) # q_c.shape = (BATCH, compact_N_CTX_Q, H)
    k_c, k_idx, iph_k = compact(k_keep, k) # k_c.shape = (BATCH, compact_N_CTX_KV, H)
    v_c, _, _ = compact(k_keep, v, index=k_idx) # v_c.shape = (BATCH, compact_N_CTX_KV, H)
    q_idx_padded = pad_index(q_idx, iph_q, pad_idx=-1) # (B, compact_N_CTX_Q, H)
    k_idx_padded = pad_index(k_idx, iph_k, pad_idx=1e9) # (B, compact_N_CTX_KV, H)

    # We need to transpose everything
    q_c = q_c.transpose(1, 2).contiguous() # (BATCH, H, compact_N_CTX_Q, D_HEAD)
    k_c = k_c.transpose(1, 2).contiguous() # (BATCH, H, compact_N_CTX_KV, D_HEAD)
    v_c = v_c.transpose(1, 2).contiguous() # (BATCH, H, compact_N_CTX_KV, D_HEAD)
    q_idx_padded = q_idx_padded.transpose(1, 2).contiguous() # (BATCH, H, compact_N_CTX_Q)
    k_idx_padded = k_idx_padded.transpose(1, 2).contiguous() # (BATCH, H, compact_N_CTX_KV)

    y_c = qk_sparse_attention_kernel(q_c, k_c, v_c, q_idx_padded, k_idx_padded, sm_scale).transpose(1,2)
    y = torch.zeros_like(q).scatter(dim=1, index=q_idx.long().view(BATCH,-1,H,1).expand(BATCH, -1, H, D_HEAD),
     src=y_c)
    return y
```

B.3. Experimental Setup

Models & Baselines. For our language modeling experiments on OpenWebText2, we use a base autoregressive transformer architecture with 12 layers, a hidden size of 768, 12 heads of 64 dimensions each. For experiments on sequence length $T = 8192$, we use a batch size of $96 = 4 \times 8 \times 2$ (batch size 4 with 8 accumulation steps and data parallelism over 2 node). When $T = 16384$ we use a batch size of $30 = 2 \times 5 \times 3$. The resulting models are of around 122M parameters. The goal not being to outperform the state-of-the-art perplexity, we train for $15k$ iterations. The attention modules used are either using FlashAttention for the baselines or one of our sparse kernels for our methods. To ensure a fair comparison, and similarly to Kitaev et al. (2020), we set the keys equal to normalized queries for all of our models. See App. B for more details.

Hardware. All of our timing experiments with random tensors are done on NVIDIA A100 GPUs, using `bfloat16`. For our language modeling tasks on OpenWebText2, we trained using data-parallelism on two or three A100s for experiments with sequence lengths of respectively 8192 and 16384. When comparing runtimes in LM training, we normalize the times by multiplying by the number of GPUs used. Comparisons with the Reformer are performed on a single A100 or a single NVIDIA RTX 4090 GPU.

B.4. Sequence Modeling Experiments

Language modeling on OpenWebText2. Our implementation is based on NanoGPT (github.com/karpathy/nanoGPT). We use the AdamW optimizer (Loshchilov & Hutter, 2019). We used `bfloat16` and NVIDIA A100-40GB GPUs for all our experiments. Here is a list of hyperparameters shared by all our language models (F-LM, H-LM, and D-LM):

- Weight-decay: 0.1
- Depth (number of transformer blocks): 12
- Number of heads: 12
- Dropout: 0.0
- Learning rate: 0.001
- Percentage of iterations for warmup: 2%. We use a cosine learning rate scheduler.
- Adam beta1: 0.9
- Adam beta2: 0.95
- Tokenizer: We use the GPT2 tokenizer provided by the tiktoken library (github.com/openai/tiktoken).
- Hidden dimensions: 768
- Dimensions per head: 64

Sequential MNIST and enwik8. For the comparisons with Reformer, we use a standard GPT2 (Radford et al., 2019) implementation. For the language modeling on enwik8, the Transformer has 12 blocks with 768 hidden dimensions, 8 attention heads, and 64 dimensions per head. Dropout is set to 0.1 and the batch size is 8 with 2 gradient accumulation steps. The sequence length is 4096. For autoregressive image generation on MNIST, we use a smaller model with 8 transformer blocks and a hidden dimension of 256. Dropout is set to 0.1 and the batch size is 10. We train for 25 epochs.

C. Additional Details and Analysis

Quadratic computational cost of attention in transformers. In Fig. 5 we show the runtime (forward + backward) of a transformer language model as a function of the sequence length. We separate the time taken by the attention operations from the time taken by the rest of the model. We see how the attention computation ends up dominating the runtime as the sequence length increases.

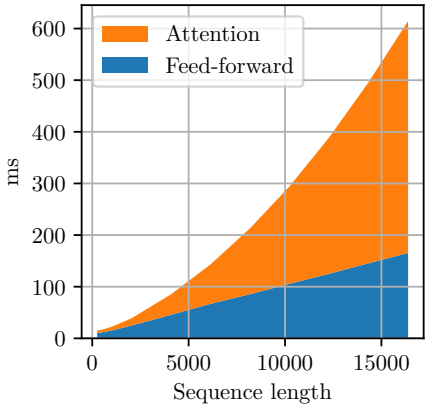
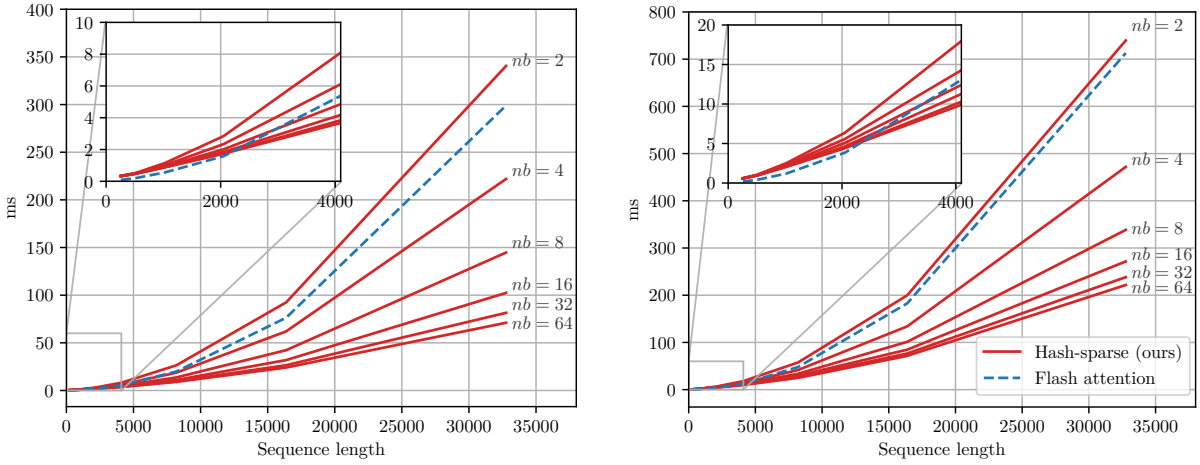


Figure 5. **Quadratic computational cost of self attention dominates for larger sequences.** We measure the forward and backward runtimes for a 12 layers transformer (see App. B.4 for implementation details). Except for the attention operations, the rest of the transformer runtime grows linearly with the sequence length.

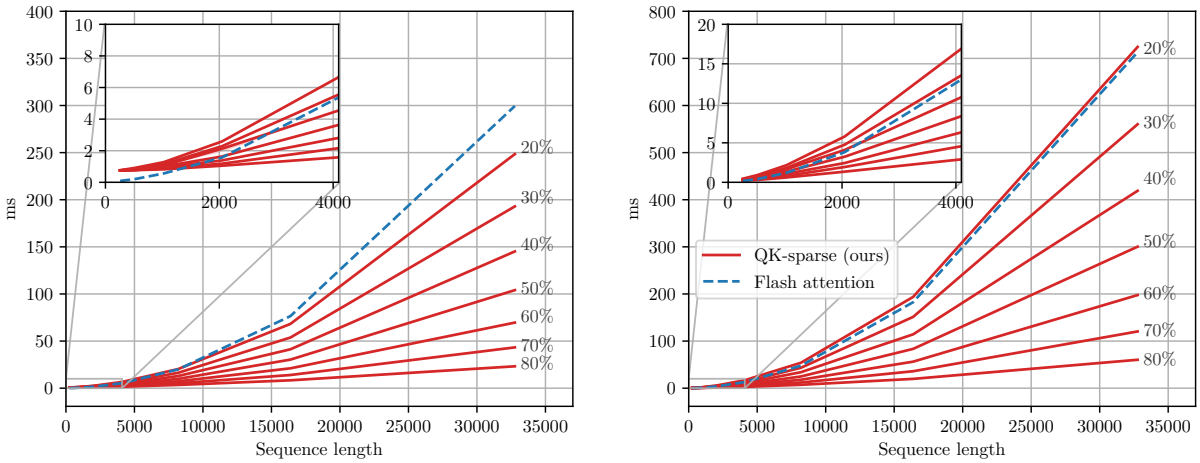
Additional runtimes performances in a vacuum. In Fig. 6 and Fig. 7 we show the runtime details for the forward and backward methods separately for respectively the Hash-sparse and QK-sparse methods. We also measure runtimes of the forward and backward passes when we assume the pre and post-processing steps aree free, see Fig. 8.



(a) Forward

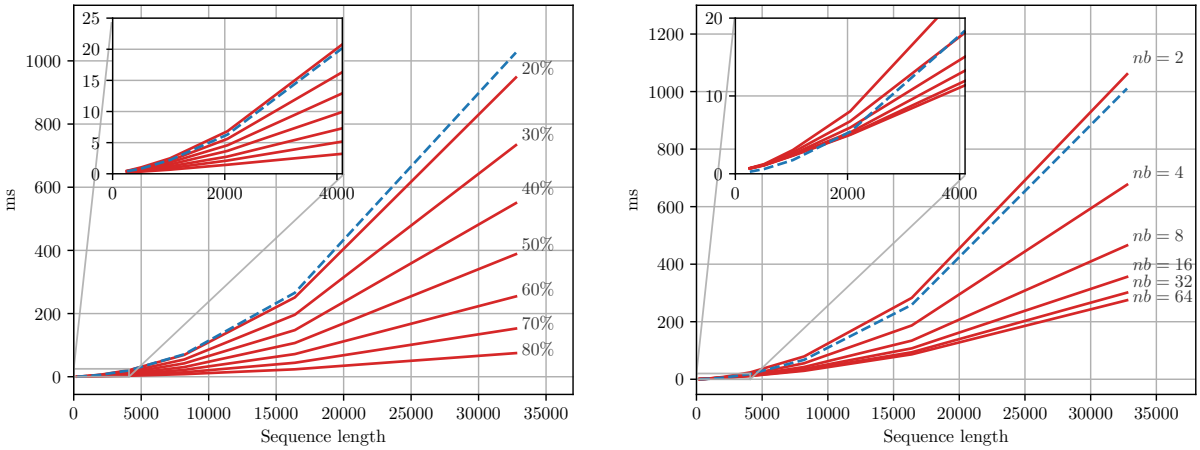
(b) Backward

Figure 6. **Forward and Backward runtimes in a vacuum for our Hash-sparse method. Fig.(a)** Forward runtimes for our Hash-sparse method. **Fig.(b):** Backward runtimes for our Hash-sparse method.



(a) Forward (b) Backward

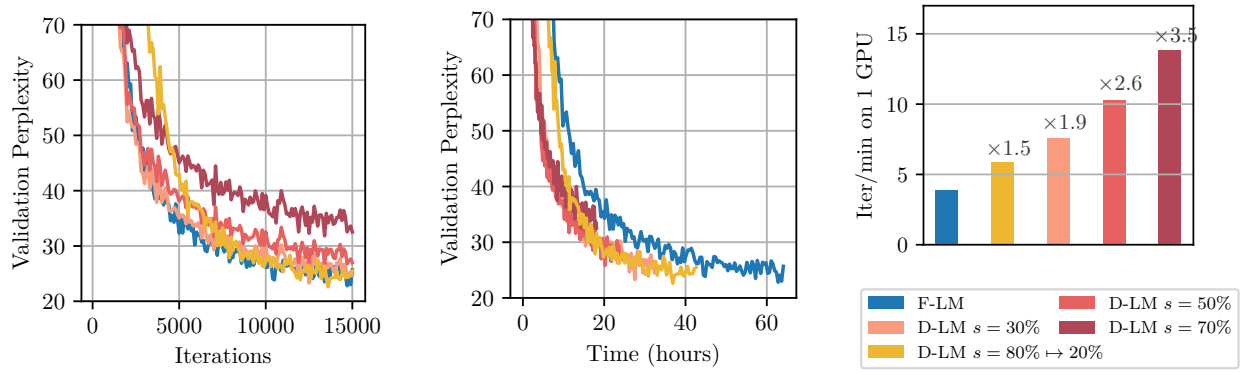
Figure 7. **Forward and Backward runtimes in a vacuum for our QK-sparse method. Fig.(a)** Forward runtimes for our QK-sparse method. **Fig.(b):** Backward runtimes for our QK-sparse method.



(a) QK-sparse (b) Hash-sparse

Figure 8. **Forward+Backward runtimes in a vacuum when we assume the preprocessing is *free*. Fig.(a)** Forward+backward runtimes for our QK-sparse method when the pre and postproceessing steps are free. While the runtimes for large sequences stays relatively the same compared to Fig. 2, the offset for short sequences is now much smaller. **Fig.(b):** Forward+backward runtimes for our Hash-sparse method when the pre and postproceessing steps are free. While the runtimes for large sequences stays relatively the same compared to Fig. 2, the offset for short sequences is now smaller. Intrestingly, unlike for the QK-sparse method, the offset for smaller sequences stays significant. We believe this might be due to the influence of the block size used (128): when the sequence length is not large enough in comparison to the block size, the block structure of the hash-sparse attention matrix cannot be efficiently leveraged and the number of tiles processed by the Hash-sparse method is larger than that of the FlashAttention method. In contrast, for the QK-sparse method the good tiles all start from 0 (we know the first tile(s) will be efficiently packed).

Linear QK-dropping scheduler. In the main paper we show results dropping keys and queries at random with a fixed pre-defined probability. In an additional experiment we start by dropping 80% of keys and queries at random and linearly decay this probability to 20%. Our intuition is earlier iterations just aim to learn contextual cues which are very redundant (and therefore quite immune to random dropping) before requiring more fine-grained representations. In Fig. 9 we show and analyse the results of that experiment.



(a) Iter to reach perplexity (b) Time to reach perplexity (c) Speed gains

Figure 9. **Training Language Models (LM) on OpenWebText2 using random Query/Key dropping based sparsity (D-LM) or FlashAttention over the entire sequence (F-LM).** We include the D-LM model with linear decay of s from 80% to 20%. In **Fig.(b) and Fig.(c)**, we observe the additional D-LM model in yellow is slower than other D-LM models we experimented with, but, as seen in **Fig.(a)**, reaches a slightly better perplexity.

H-LM models speeding up during training. The speed of Hash-sparse attention is conditioned on the distribution of bucket indices over keys and queries—e.g. if all the keys and queries were to fall in the same bucket then there would be no speedup over FlashAttention. Interestingly, when training on real data such as OpenWebText2, we observe our Hash-sparse based models are speeding up during training. In Fig. 10 we plot the number of iterations reached after x hours of training (normalizing the time by the number of GPUs used for training). In Fig. 10.(a) we see a speedup for H-LM models early in the training.

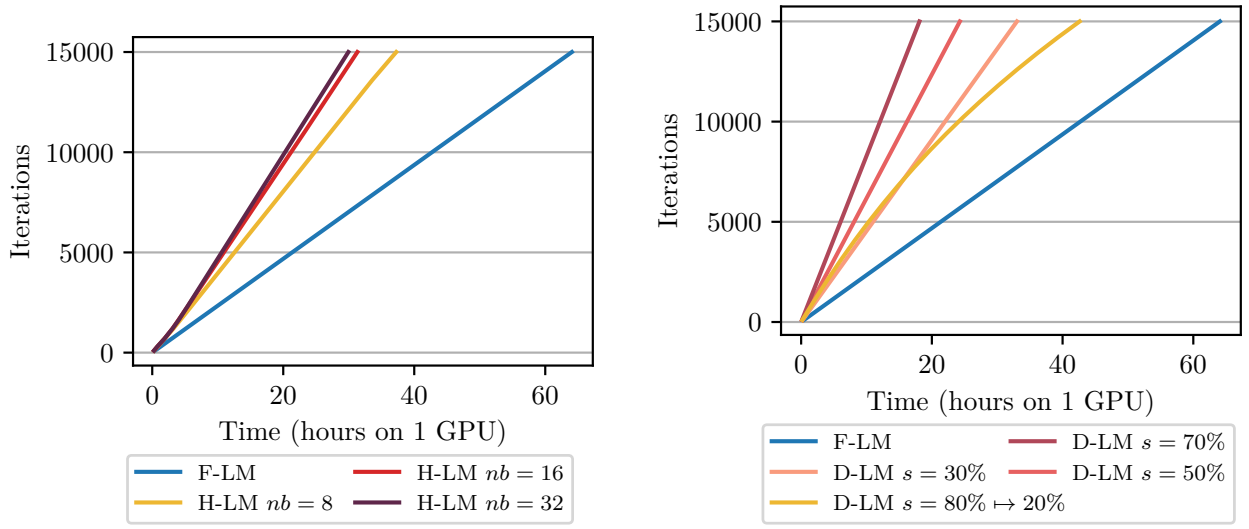
Different bucket sizes nb for H-LM. For sequences of 8192 tokens, we show the influence of increasing the number of buckets nb in Fig. 11. As nb increases the runtime decreases.

Training H-LMs for more iterations. In Fig. 3 we show results of language models trained on OpenWebText2 for 15k iterations. To verify whether our finding are consistent when you train for more iterations we also try training for 50k iterations. For sequences of 8192 tokens, using the same hyperparameters described in App. B.4, we show in Fig. 12 that our findings for our Hash-sparse based models do hold when training for more iterations—we match the perplexity per iterations of the baseline model using FlashAttention over the entire sequence while being significantly faster.

Comparisons with Reformer. Fig. 13 shows runtime comparison between our hash-sparse implementation and the Reformer LSH attention. Table 14 shows the performance on two autoregressive tasks. In Fig. 15 we show the attention matrices for two different heads and show how the coverage—the percentage of key-query interactions actually computed vs. what should be computed according to the hash indices computed for keys and queries—of the Reformer LSH algorithm can be low.

D. Limitations and Societal Impacts

Limitations. The aim of our work is to develop a method for efficiently computing attention with several sparsity structures. We don't focus on developing the best method for sparsification, although, for example, we improve the hashing-based mechanism. Moreover, as already explained, on very small sequences we incur in some constant overhead which limits our gains.



(a) Time comparison H-LM vs. F-LM

(b) Time comparison D-LM vs. F-LM

Figure 10. **Training speed as a function of time for H-LM, D-LM and F-LM. All models trained on sequences of** 8192 **tokens.** In **Fig.(b)** we observe D-LM models having a fixed speed throughout the experiments except for the one corresponding to the linear decaying of the sparsity ratio s which slows down during training. In **Fig.(a)** H-LM models are speeding up early during training and then seem to keep a constant speed.

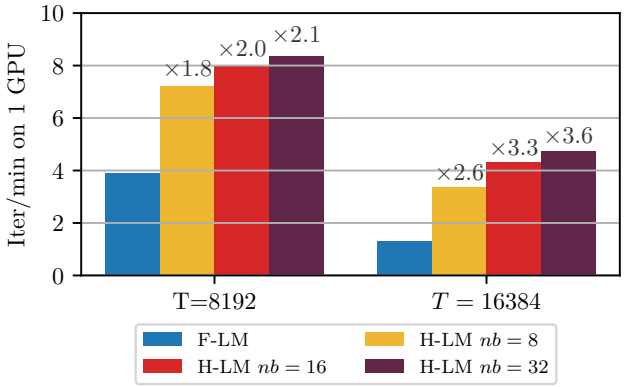
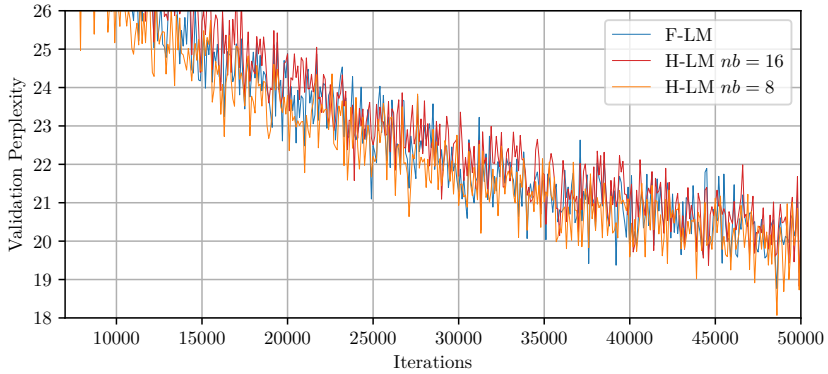
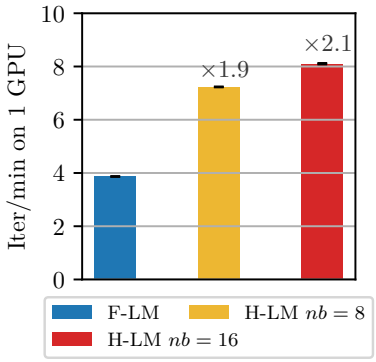


Figure 11. **Influence of the number of buckets** nb **on H-LM training speed for** $T = 8192$ **and** $T = 16384$**.** In accordance with Fig. 2 increasing nb speeds up the training but the speed gain from doubling nb decreases as nb is increasing.

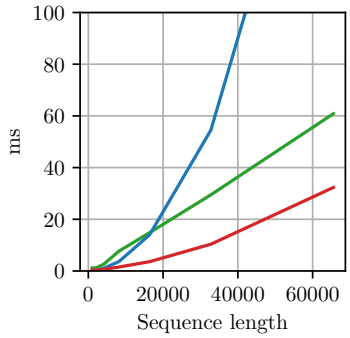


(a) Iter to reach perplexity

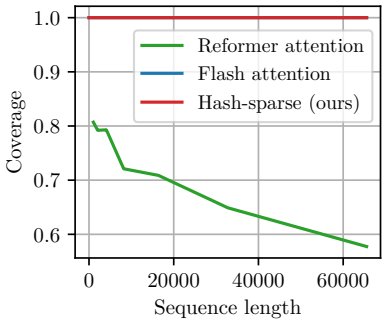


(b) Speed gains

Figure 12. **Comparison between H-LM and F-LM when training for** 50**k iterations on sequences of** 8192 **tokens.** Those curves are averaged over two seeds. In **Fig.(a)** we zoom in on the perplexity per iteration for the baseline F-LM and two of our H-LM models using our Hash-sparse attention with 8 and 16 buckets. We observe that our methods are matching the perplexity of the F-LM model which uses FlashAttention over the entire sequence. **Fig.(b):** We match the perplexity but gain in speed, our methods are $1.4\times$ and $1.8\times$ faster than the baseline for respectively 8 and 16 buckets.



(a) Forward attn. runtimes



(b) Coverage of Reformer attn.

Figure 13. **Comparing forward runtimes of attention modules alone. Fig.(a):** Reformer attention ensures a linear computational complexity w.r.t. the sequence length, outperforming FlashAttention for longer sequences. **Fig.(b):** However, due to the fixed attention structure, the Reformer misses an increasing fraction of hash collisions. Our approach outperforms both methods and maintains 100% exact coverage of collisions for all sequence lengths. See App. B.4 and App. C for more details.

Figure 14. **Comparing models using Reformer attention vs our Hash-sparse attention.** On the simple sequential MNIST task (predicting pixels as a sequence), we obtain a comparable perplexity as the Reformer. On enwik8 character language modeling, with T=4096, we outperform the Reformer model by a margin.

Attention	MNIST (ppl ↓)	enwik8 (bits/c ↓)
Reformer	1.76	3.32
Hash-sparse	1.67	2.29

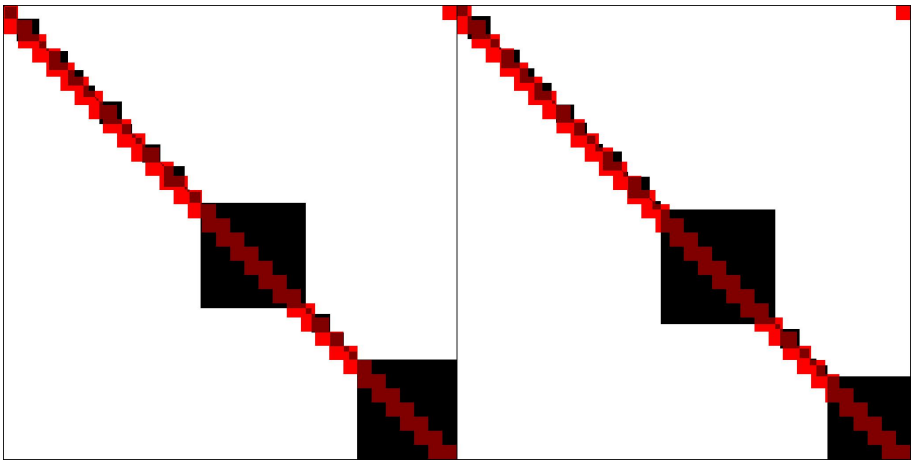


Figure 15. **Attention pattern of Reformer.** The two images show the attention pattern for the Reformer LSH attention after reordering queries and keys according to the bucket. The red squares are the positions for which attention is computed in the Reformer with bucket size = 32. The back squares are positions for which the queries and keys map to the same hash bucket. Queries are sampled from a normal distribution with $\mu = 3$ and $\sqrt{\sigma} = 5$. Each image refers to a different attention head. Black regions which are not covered by red tiles are key-query interactions which should be computed but are missed by the Reformer LSH attention.

Societal impacts. The attention mechanism is central to Large Language Models (LLMs). Moreover, efficient attention mechanisms can make these models more powerful, by making them faster to train and by increasing their context length. The social impact and risks associated with our work, therefore, are included in the risks associated with the deployment of such systems (Bender et al., 2021; Weidinger et al., 2021).