Efficiently Dispatching Flash Attention For Partially Filled Attention Masks

Agniv Sharma[1,2] **Jonas Geiping**[2,3,4]

[1]University of Tübingen [2] Tübingen AI Center
[3] ELLIS Institute Tübingen
[4]Max Planck Institute for Intelligent Systems

Abstract

Transformers are widely used across various applications, many of which yield sparse or partially filled attention matrices. Examples include attention masks designed to reduce the quadratic complexity of attention, sequence packing techniques, and recent innovations like tree masking for fast validation in MEDUSA. Despite the inherent sparsity in these matrices, the state-of-the-art algorithm Flash Attention still processes them with quadratic complexity as though they were dense. In this paper, we introduce **Binary Block Masking**, a highly efficient modification that enhances Flash Attention by making it mask-aware. We further propose two optimizations: one tailored for masks with contiguous non-zero patterns and another for extremely sparse masks. Our experiments on attention masks derived from real-world scenarios demonstrate up to a 9x runtime improvement. The implementation will be publicly released to foster further research and application.

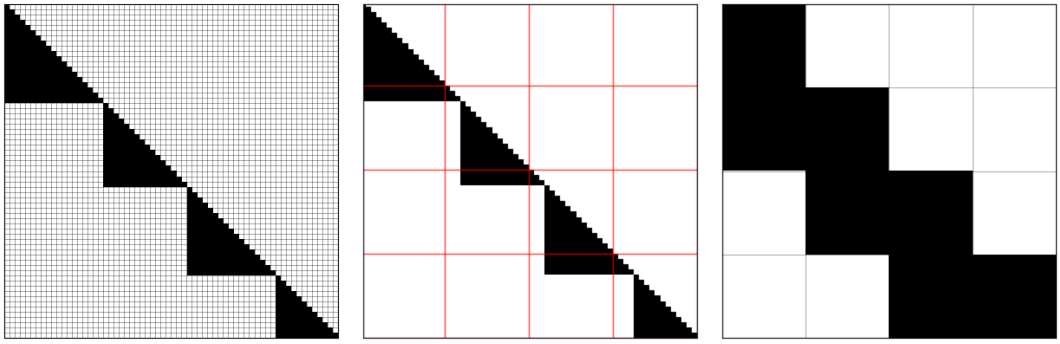
1 Introduction

Transformers [26] revolutionized sequence modeling by using self-attention mechanisms, enabling efficient parallelization and handling of long-range dependencies. A key component of transformers is the attention mechanism. However, in many practical scenarios—such as when processing very long sequences, reducing computational costs, or working with structured data like graphs — it becomes advantageous to restrict the range of elements each item can attend to. This leads to the development of masked attention mechanisms, which limit interactions to a subset of elements.

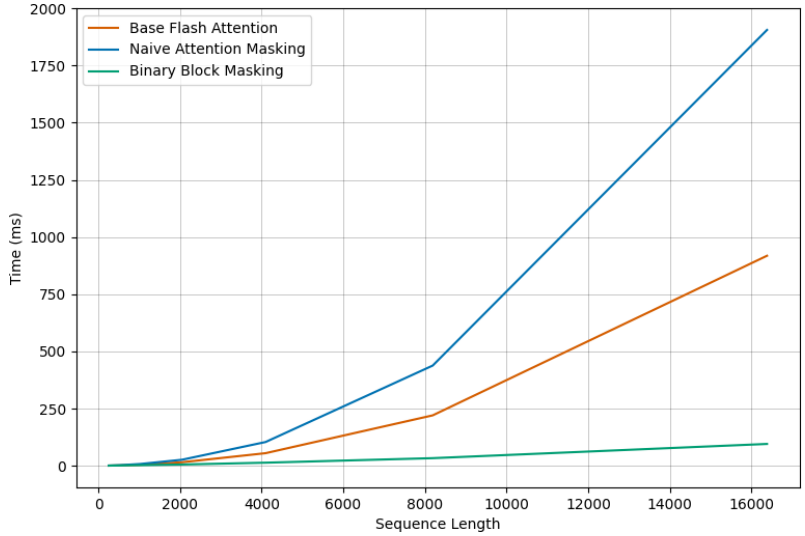
One application that leads to such partially filled masks is the creation of efficient transformers. Approaches like *LongFormer* and *BigBird* [2, 29] employ fixed sparsity patterns to alleviate the quadratic complexity inherent in self-attention, while other approaches exploit low-rank approximations to achieve similar objectives [27, 9]. Despite the theoretically lower computational complexity of these methods, many results fail to translate to reduced wall-clock runtimes. The currently prevalent practical solution is instead to simply implement dense attention more efficiently. Flash Attention [10, 8, 21] is a such a hardware-aware algorithm sufficient to boost the speed of dense attention and obviate the need for a more efficient attention in many applications.

However, because the initial flash attention implementation was written with only causal masks in mind, research using alternative attention patterns has slowed down. A previous approach to remedy this, [18], enables a few types of mask in flash attention, but only considers a small subset of cases and requires the user to carefully change the GPU kernel parameters for every new mask. To overcome these limitations, we introduce **Binary Block Masking** (`BinBlkMsk`) - a novel algorithm that extends Flash Attention to support any attention masks while also being user-friendly. Our approach builds on two key insights: (1) Flash Attention processes attention matrices in blocks, so we only need to handle blocks containing at least one non-zero mask value, and (2) these blocks are independent

4th NeurIPS Efficient Natural Language and Speech Processing Workshop (ENLSP-IV 2024).



(a) Original Attention Mask (b) Mask with Binary Blocks (c) Binary Block Matrix



(d) Performance Comparison

Figure 1: Proposed algorithm for masking flash attention

and can be preprocessed in parallel with minimal runtime overhead and then shared across multiple heads, layers, or runs depending on the use case. We further enhance the algorithm's efficiency for masks with contiguous non-zero blocks. Finally, we discuss how off-the-shelf graph optimization algorithms allow these gains to be realized even for relatively unstructured attention patterns, which can be preprocessed into low bandwidth layouts to improve block sparsity.

Our method introduces efficient support for custom masks in flash attention. With this improvement, we achieve up to a 9x reduction in runtime for real-world sparse and partially filled attention matrices. We validate the performance gains of our method using masks derived from three practical applications: packed fine-tuning on the ALPACA dataset [23], tree attention masks for MEDUSA [4], and sparse attention matrices from Longformer[2].

Our method's key advantages can be summarized as:

1. **Universal Mask Compatibility**: Our algorithm is the first to support any custom mask, providing a generalized solution without the need for user adjustments.

2. **Performance Consistency**: Our implementation significantly outperforms Flash Attention for partially filled or sparse attention masks, while matching its performance for almost filled attention masks.

We implement our algorithm using Triton [25], leveraging its device agnosticism, readability, and ease of integration. Our implementation and scripts for processing various attention masks will be made available to support further research.

2 Related Work

Since their inception, transformers have employed masked attention for various applications. The original transformer[26] uses causal masking to maintain the sequential nature of text, and standard techniques, such as in [20], utilize attention masks to 'pack' multiple sequences into one block during pretraining. Masking has also been utilized to create efficient transformers [15, 24]. Models like Star Transformer[11], Longformer[2], Big Bird[29], and ETC[1] use attention masks with global and local token windows to speed up attention calculations while preserving information flow. Sparse transformers [6] use factorized attention mask to attend to different parts of input across different heads and Reformer[13] uses locality-sensitive-hashing (LSH) to select keys which limit attention to keys and queries that collide to the same hash.

However, masked attention is a natural building block in a number of applications across domains, such as graph learning with transformers [3] or selective perception in vision transformers [28]. On the other hand, inference techniques to speed up generation in text generation, such as [4, 16, 22] all also use some form of a tree attention mask to increase the number of candidates validated at each step of speculative decoding[14, 5]. All these approaches utilize masked attention, but ours is the first method to combine the advantages of state-of-the-art Flash Attention with masking, achieving the best of both techniques.

Our method closely relates to two previous approaches: Block Sparse Attention[10] and Faster Causal Attention[18]. In Block Sparse Attention[10], authors skip processing some blocks randomly, but unlike our approach, they do not use a mask to guide this, resulting in a mask-agnostic method that provides only approximate attention, whereas ours offers exact calculations. Faster Causal Attention[18] handles only two specific masks—QK masks (dropping random query keys) and hash sparse masks—but it lacks flexibility for general masks and requires significant adaptation for new methods. In contrast, our method is generalizable and intuitive across different masking strategies.

This method we described was developed as part of a course project in Spring 2024 and is contemporary with PyTorch's new FlexAttention [12]. Both approaches rely on a similar principle of reducing mask attention computations through block structures that are dispatched intelligently. However, while FlexAttention is a JIT-compiled "functional" approach in which the mask is provided as a function over block indices, our method is a conceptually more straightforward tensor-to-tensor interface that directly consumes inputs and masks as tensors. We provide a detailed comparisons of the performance characteristics of both approaches in Appendix B, where we show that our tensor-to-tensor approach is conceptually simpler, yet competitive in both dispatch speed and runtime.

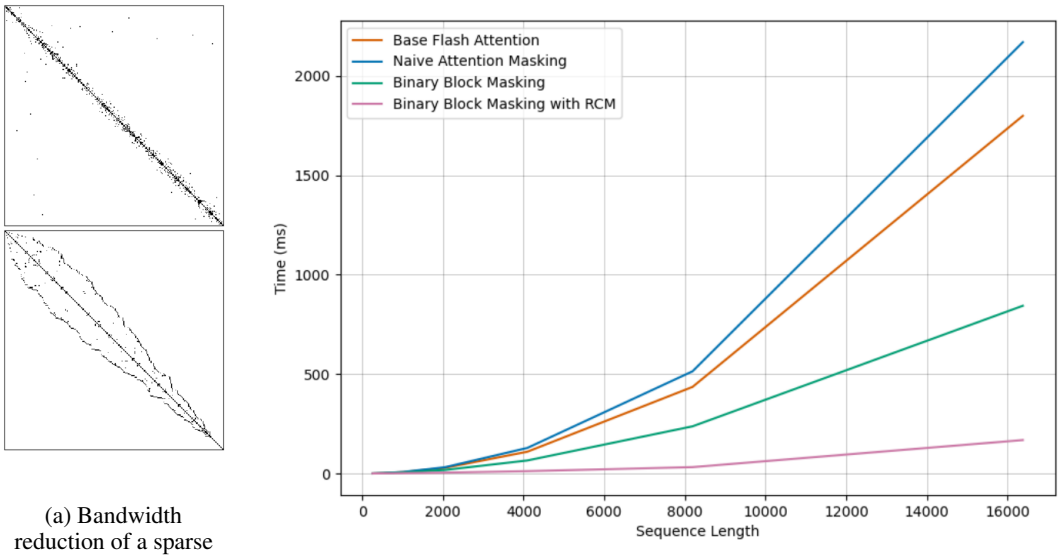
3 Method

3.1 Background

Standard Attention In standard self-attention[26], the Query (Q) and Key (K) matrices, both of size $N \times D$ (where N is the number of tokens and D is the dimensionality), are multiplied to produce an $N \times N$ attention matrix. This matrix is normalized using softmax to form a probability distribution, which is then multiplied by the Value (V) matrix ($N \times D$) to generate the final output. In multi-head attention, this process is repeated across multiple heads, and typically processed over batches to enhance parallelism and capture diverse patterns.

Flash Attention Flash Attention[10, 8] improves self-attention efficiency through two key optimizations. First, it computes the attention matrix in blocks using a running softmax, reducing data transfers from High Bandwidth Memory (HBM). Second, it recomputes the attention matrix during the backward pass, saving memory and optimizing IO operations. These techniques make Flash Attention particularly effective for large models and long sequence lengths.

Flash Attention with *naive* masking. Flash Attention currently only supports standard causal masks. Extending Flash Attention to handle custom masks using a naive approach would involve reading the corresponding mask block for each Flash Attention block. While this method would correctly apply the mask, it introduces additional memory accesses for reading the mask and extra computation for applying it, resulting in a slower runtime than the base Flash Attention. We refer to this method as *Naive Attention Masking*, and benchmark its performance to highlight the trade-offs.



(a) Bandwidth reduction of a sparse mask through RCM

(b) Performance Gains achieved through RCM preprocessing

Figure 2: The result and performance of RCM bandwidth reduction for a sparse attention mask.

3.2 Our Approach

Binary Block Masking Our method focuses on optimizing Flash Attention by processing only the blocks of the attention matrix that have non-zero entries in their corresponding mask blocks. We first preprocess the attention mask to create a binary matrix, termed the "Binary Block Matrix" (`BinBlkMat`). This matrix has dimensions `N//BLOCKSIZE_I` $\times$ `N//BLOCKSIZE_J` where N is the size of the attention mask and `BLOCKSIZE_I` and `BLOCKSIZE_J` represent the block sizes in the `row` and `column` dimensions respectively. Each entry in `BinBlkMat` is set to 1 if any value in the corresponding mask block is non-zero.

During the attention step, the relevant block is processed only if its corresponding entry in `BinBlkMat` is non-zero, reducing unnecessary computations. When masks are fixed, `BinBlkMat` can be precomputed. For dynamically changing masks, `BinBlkMat` can be calculated in parallel for all blocks and shared across transformer layers and heads. Thus, `BinBlkMat` introduces minimal runtime overhead while significantly improving the efficiency.

Dense Binary Block Masking In natural language processing, masks often contain contiguous non-zero blocks. To optimize for such cases, we leverage the insight that only the edges of these contiguous blocks need to be checked, as the center is entirely ones. We introduce two additional arrays, "`total_ones`" and "`offset`", both of size `N//BLOCKSIZE_I` $\times 1$. "`total_ones`" stores the number of consecutive blocks filled with ones, while "`offset`" stores the position of the first such block.

During the attention step, we parallelize across the Q dimension and iterate over blocks of K. Thus, while processing `i`$^{\texttt{th}}$ block of Q, the mask is read only if the current iteration is less than the `offset_i` or greater than the `offset_i + total_ones_i`, and if the corresponding `BinBlkMat` value is 1. For iterations within the range of the contiguous block, the mask read is skipped, as it consists entirely of ones. This significantly reduces the number of mask reads from HBM, speeding up processing. Similar to `BinBlkMat`, the `total_ones` and `offset` arrays can be computed in parallel with minimal overhead and shared across layers and heads.

Binary Block Masking with Reverse Cuthill-McKee (RCM) In highly sparse scenarios, such as masked transformers for graph datasets [3], processing entire blocks due to a single non-zero entry can lead to inefficiencies, as every block may be required, but only filled with a few non-sparse entries. To mitigate this, we can make use of the Reverse Cuthill-McKee (RCM) [7] algorithm as a preprocessing step. RCM is a well-established graph reordering technique that minimizes the bandwidth of sparse matrices by treating them as connected graphs, effectively increasing the matrix's *"fill-in"*.

While standard Binary Block Masking provides a performance improvement, RCM significantly amplifies this benefit in cases of extreme sparsity. By reorganizing the matrix structure, RCM reduces

the number of blocks that must be processed. Figure 2a illustrates a case where RCM reduces the number of blocks by 50%. Figure 2b shows results on a synthetic mask where RCM preprocessing reduces the number of blocks by 90%, resulting in a significant performance gain. However, it is important to note that the extent of the performance gain is dependent on both the sparsity and the structure of the mask.

Although we focus on the RCM algorithm in this work, other reordering techniques that increase matrix *"fill-in"* could also yield similar improvements in performance when working with sparse matrices. This approach is especially advantageous when the sparsity pattern of the attention mask is either known in advance or can be efficiently computed.

4 Experiments

In the following section, we evaluate the performance of our "Binary Block Masking" algorithm using attention masks from three real-world use cases: (1) tree attention masks for Speculative Decoding in MEDUSA[4], (2) masks from packed finetuning[20] of an LLM on the ALPACA dataset[23], and (3) fixed masks from the LongFormer paper[2].

4.1 Experimental Setup

Evaluation Metrics All experiments report total runtime (forward and backward pass) averaged over 100 runs, with a batch size of 4 and 32 attention heads. The `BLOCKSIZE` for Flash Attention[10, 8]is fixed to $(128, 32)$, which is fine-tuned to achieve optimal performance on our hardware configuration.

Baselines We compare against two baselines: the Triton implementation[25] of Flash Attention[10, 8] and a naive masking algorithm that applies masks block by block, as previously described. Although Flash Attention is mask-unaware (and hence does not actually produce a correct result), it serves as a useful baseline to display performance gains with our method. We use Flash Attention's Triton implementation (instead of the CUDA implementation [17]) for fairer comparisons, as Triton introduces its own overhead.

Hardware and Precision Experiments were conducted on an RTX 3060 (6GB memory) using `bfloat16` precision for query, key and value vectors. For a more detailed analysis on different hardware, please consult Appendix D.

4.2 Preprocessing

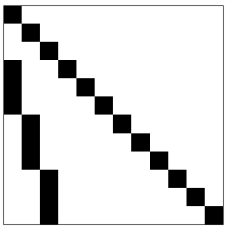
We implement a Triton kernel to preprocess the attention mask into the binary block mask format. The preprocessing runtime is comparable to the forward pass of a single attention head. However, this overhead is minimal as the Flash Attention runtime is dominated by the backward pass. Additionally, preprocessing is done once and shared across multiple heads and transformer layers. The following table provides runtime comparisons for various mask dimensions.

Table 1: Binary Block Preprocessing Time vs Total Runtimes of Flash Attention Algorithm For Varying Head and Batch Sizes

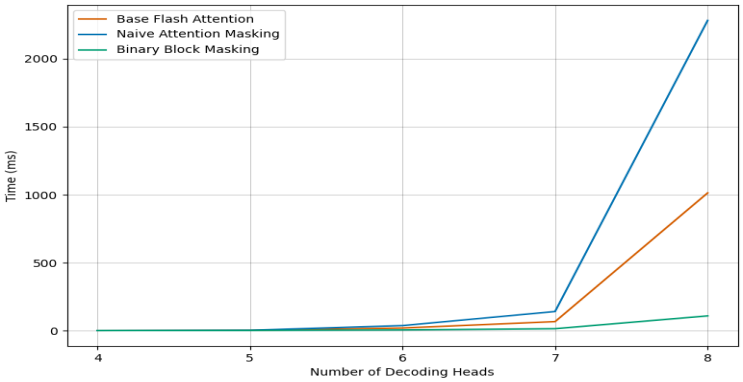
Mask Size	Prepro. Time (ms)	Forward Pass (ms) (B=1, H=1)	Total Runtime (ms) (B=1, H=1)	Total Runtime (ms) (B=4, H=32)
4096	0.176	0.306	1.348	96.076
8192	0.687	0.886	3.986	377.625
16384	2.598	2.935	13.239	1524.763

4.3 Evaluation on MEDUSA Tree Masks

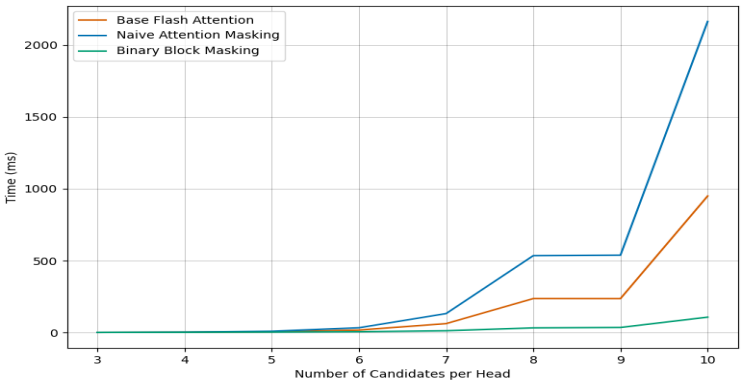
Figure 3 shows the results for tree masks derived from the MEDUSA architecture[4], a speculative decoding approach that speeds up inference by using multiple heads to predict future tokens in parallel. MEDUSA generates several candidate tokens per head, verified through a tree-based attention mechanism. We evaluate performance gains from using these tree attention masks.



(a) Tree Attention Mask(K=3, s_k=3)



(b) Performance with varying Heads and fixed Candidates(s_k=3)



(c) Performance with varying Candidates and fixed Heads(K=4)

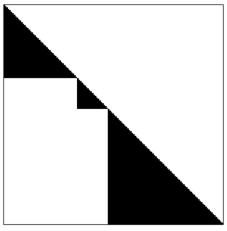
Figure 3: Mask Visualization and Performance Comparison for MEDUSA tree mask

The size of a MEDUSA tree mask is given by $\sum_{k=1}^{K} \prod_{i=1}^{k} s_i$, where K is the number of decoding heads and s_k is the number of candidates per head. Our current configuration uses 4 heads with 4 candidates each, resulting in a 340x340 attention matrix. At this scale, Flash Attention shows no significant speedup, so we use the native PyTorch implementation as baseline.

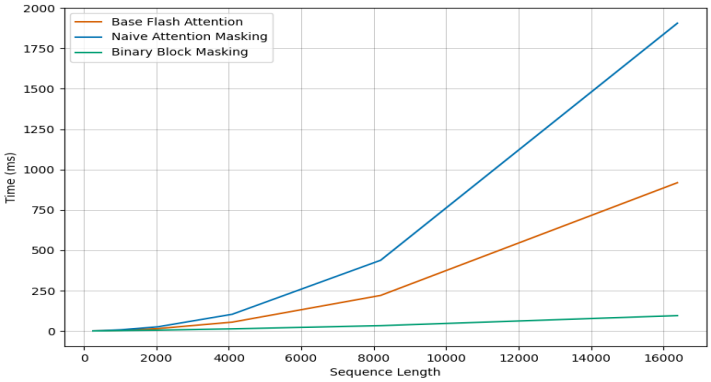
Our experiments demonstrate scalability with minimal computational overhead, allowing more decoding heads and candidates per head. Figures 3b, and 3c show two scenarios: one with fixed number of candidates per-head(s_k=3) and varying decoding heads(K), and another with fixed decoding heads($K = 4$) and varying number of candidates(s_k). It is important to note that in the first scenario, the size of the attention matrix grows exponentially with K. Therefore, while in other cases the attention matrix typically doubles at each step, here it quadruples in the final step, resulting in a sudden increase in compute time for naive masking and base flash attention cases. In both, our method outperforms base Flash Attention, showing promise for applications requiring longer sequences and greater variety.

4.4 Evaluation on packed ALPACA finetuning

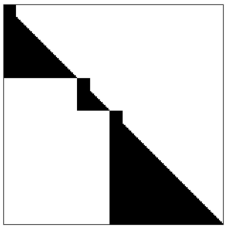
Alpaca[23] is an instruction fine-tuning dataset comprising input, instruction, and output components. We combine the input and instruction into a single input and pack multiple sequences into longer ones, using masks to prevent cross-sequence attention. Figure 4 demonstrates our algorithm's performance on two instruction tuning workloads: (1) **Causal Masks**: Input and output are concatenated, with tokens attending only to previous tokens. And (2) **Input-Bidirectional Mask**: Input tokens attend to each other, while output tokens attend to input and preceding output tokens. Both masks consist of contiguous non-zero blocks, enabling the use of our `Dense BinBlkMsk` algorithm variant. Packing improves upon padding, where shorter sequences in a batch are extended with a special token to match the longest sequence's length. While padding wastes GPU resources, it is more compatible with Flash Attention. Our experiments show that our algorithms efficiently combine the benefits of Flash Attention with packing, resulting in approximately a 9x performance increase in both cases.



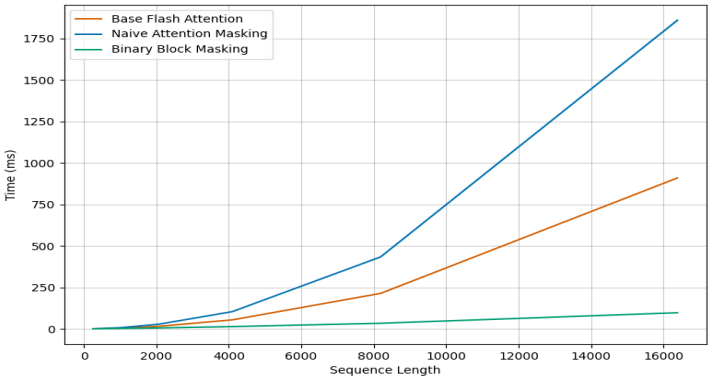
(a) Packed Causal Mask



(b) Performance comparison on Causal Mask



(c) Packed Input-Bidirectional Mask



(d) Performance comparison on Input-Bidirectional Mask

Figure 4: Mask Visualization and Performance Comparison for packed ALPACA dataset

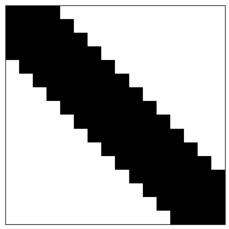
4.5 Evaluation on LongFormer Attention Masks

We evaluate our algorithm across three LongFormer[2] masking patterns: Windowed, Dilated, and Global. For the Windowed mask, we use the `Dense BinBlkMsk`, while the base algorithm is used for the other two masks. The performance results, along with the corresponding masks, are presented in Figure 5. Our findings demonstrate that our method consistently outperforms the baseline Flash Attention algorithm across all three masks. Moreover, these results highlight that the performance improvements gained through sparsity are complementary to those achieved via FlashAttention. By leveraging our mask-aware algorithm, we can construct more efficient architectures capable of processing longer sequences with faster processing.

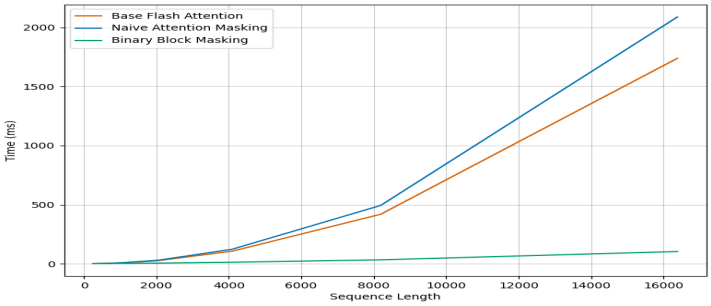
5 Discussion, Limitations, and Future Work

Our experimental results demonstrate that the proposed algorithm effectively leverages the mask structure to achieve notable speedups, though the extent of these improvements is highly dependent on the mask's fill pattern. Specifically, as the mask becomes increasingly filled, the processing time grows correspondingly. However, even in extreme cases—such as a fully dense mask (composed entirely of ones) or a causal mask—our algorithm maintains performance that is competitive with Flash Attention[10, 8]. This is primarily due to the optimization introduced by the `Dense BinBlkMsk` representation, which ensures that the only additional overhead compared to the base implementation is the reading of binary values for the Binary Block Mask (Appendix E).

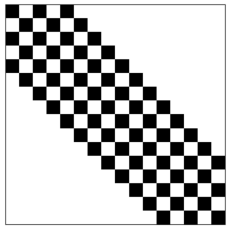
One scenario where our method may exhibit slower performance than Flash Attention arises when the mask is nearly full but consists of non-contiguous non-zero blocks. In such cases, Flash Attention may outperform our algorithm in raw speed. However, it is important to note that Flash Attention is mask-agnostic, meaning the output it generates does not inherently respect the mask constraints. To



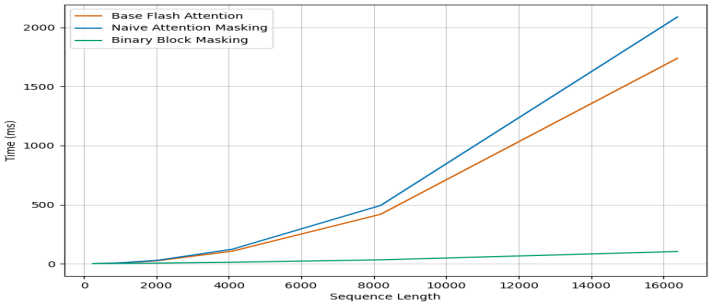
(a) Longformer - Windowed Mask



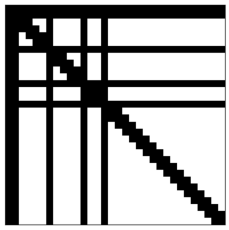
(b) Performance comparison on Windowed Mask



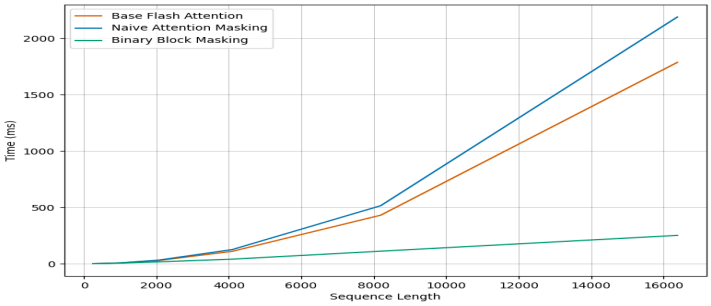
(c) Longformer - Dilated Mask



(d) Performance comparison on Dilated Mask



(e) Longformer - Global Mask



(f) Performance comparison on Global Mask

Figure 5: Mask Visualization and Performance Comparison for Longformer

apply masking with Flash Attention, a post-processing step is required, which can negate its initial speed advantage and make the overall process slower.

For future work, a key direction would be migrating our custom Triton kernels[25] to CUDA[17] and integrating them with the state-of-the-art Flash Attention framework to further enhance performance. Additionally, it is crucial to evaluate the algorithm on real-world, end-to-end tasks rather than focusing solely on isolated mask cases. Another avenue is exploring alternatives to Reverse Cuthill-McKee[7] for increasing matrix fill-in, enabling us to extend the approach to more general sparse matrices, including asymmetric matrices or matrices without underlying connected graph structures.

6 Conclusion

In this paper, we present our implementation of **Binary Block Masking** that optimizes Flash Attention for partially filled attention masks. By preprocessing the mask blocks in parallel, our method identifies non-zero blocks and restricts attention computation to these blocks, enhancing efficiency. This approach is both general-purpose and easy to integrate, making it broadly applicable. Our experiments demonstrate substantial performance improvements over base Flash Attention and highlight the versatility of our method across diverse applications. We will release our Triton kernels and mask-processing code to encourage further research in this area.

Acknowledgments and Disclosure of Funding

We appreciate Madhavi Sen for her help with formatting of figures and tables, Anna Manasyan for proof reading, and Dr. Philipp Hennig for inspiring this collaboration. Dr. Jonas Geiping acknowledges the support of the Hector Foundation II, the Max Planck Society, and the Tübingen AI Center in Tübingen, Germany. Lastly, Agniv Sharma's work is supported by his hopes and dreams.

References

[1] Joshua Ainslie, Santiago Ontanon, Chris Alberti, Vaclav Cvicek, Zachary Fisher, Philip Pham, Anirudh Ravula, Sumit Sanghai, Qifan Wang, and Li Yang. ETC: Encoding long and structured inputs in transformers. In Bonnie Webber, Trevor Cohn, Yulan He, and Yang Liu, editors, *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 268–284, Online, November 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.emnlp-main.19. URL https://aclanthology.org/2020.emnlp-main.19.

[2] Iz Beltagy, Matthew E. Peters, and Arman Cohan. Longformer: The long-document transformer. *arXiv:2004.05150*, 2020.

[3] David Buterez, Jon Paul Janet, Dino Oglic, and Pietro Lio. Masked attention is all you need for graphs. *arXiv preprint arXiv:2402.10793*, 2024.

[4] Tianle Cai, Yuhong Li, Zhengyang Geng, Hongwu Peng, Jason D. Lee, Deming Chen, and Tri Dao. Medusa: Simple llm inference acceleration framework with multiple decoding heads. *arXiv preprint arXiv: 2401.10774*, 2024.

[5] Charlie Chen, Sebastian Borgeaud, Geoffrey Irving, Jean-Baptiste Lespiau, Laurent Sifre, and John Jumper. Accelerating large language model decoding with speculative sampling. *arXiv preprint arXiv:2302.01318*, 2023.

[6] Rewon Child, Scott Gray, Alec Radford, and Ilya Sutskever. Generating long sequences with sparse transformers. *arXiv preprint arXiv:1904.10509*, 2019.

[7] E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. In *Proceedings of the 1969 24th National Conference*, ACM '69, page 157–172, New York, NY, USA, 1969. Association for Computing Machinery. ISBN 9781450374934. doi: 10.1145/800195.805928. URL https://doi.org/10.1145/800195.805928.

[8] Tri Dao. Flashattention-2: Faster attention with better parallelism and work partitioning. *arXiv preprint arXiv:2307.08691*, 2023.

[9] Tri Dao, Albert Gu, Matthew Eichhorn, Atri Rudra, and Christopher Ré. Learning fast algorithms for linear transforms using butterfly factorizations. In *International conference on machine learning*, pages 1517–1527. PMLR, 2019.

[10] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, editors, *Advances in Neural Information Processing Systems*, volume 35, pages 16344–16359. Curran Associates, Inc., 2022. URL https://proceedings.neurips.cc/paper_files/paper/2022/file/67d57c32e20fd0a7a302cb81d36e40d5-Paper-Conference.pdf.

[11] Qipeng Guo, Xipeng Qiu, Pengfei Liu, Yunfan Shao, Xiangyang Xue, and Zheng Zhang. Star-transformer, 2022. URL https://arxiv.org/abs/1902.09113.

[12] Horace He, Driss Guessous, Yanbo Liang, and Joy Dong. FlexAttention: The Flexibility of PyTorch with the Performance of FlashAttention. https://pytorch.org/blog/flexattention/, 2024. PyTorch Blog.

[13] Nikita Kitaev, Lukasz Kaiser, and Anselm Levskaya. Reformer: The efficient transformer. In *International Conference on Learning Representations*, 2020. URL https://openreview.net/forum?id=rkgNKkHtvB.

[14] Yaniv Leviathan, Matan Kalman, and Yossi Matias. Fast inference from transformers via speculative decoding. In *International Conference on Machine Learning*, pages 19274–19286. PMLR, 2023.

[15] Tianyang Lin, Yuxin Wang, Xiangyang Liu, and Xipeng Qiu. A survey of transformers. *AI Open*, 3:111–132, 2022. ISSN 2666-6510. doi: https://doi.org/10.1016/j.aiopen.2022.10.001. URL https://www.sciencedirect.com/science/article/pii/S2666651022000146.

[16] Xupeng Miao, Gabriele Oliaro, Zhihao Zhang, Xinhao Cheng, Zeyu Wang, Zhengxin Zhang, Rae Ying Yee Wong, Alan Zhu, Lijie Yang, Xiaoxiang Shi, et al. Specinfer: Accelerating large language model serving with tree-based speculative inference and verification. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 932–949, 2024.

[17] NVIDIA, Péter Vingelmann, and Frank H.P. Fitzek. Cuda, release: 10.2.89, 2020. URL https://developer.nvidia.com/cuda-toolkit.

[18] Matteo Pagliardini, Daniele Paliotta, Martin Jaggi, and François Fleuret. Fast attention over long sequences with dynamic sparse flash attention. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. URL https://openreview.net/forum?id=UINHuKeWUa.

[19] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library, 2019. URL https://arxiv.org/abs/1912.01703.

[20] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of machine learning research*, 21(140):1–67, 2020.

[21] Jay Shah, Ganesh Bikshandi, Ying Zhang, Vijay Thakkar, Pradeep Ramani, and Tri Dao. Flashattention-3: Fast and accurate attention with asynchrony and low-precision. *arXiv preprint arXiv:2407.08608*, 2024.

[22] Benjamin Spector and Chris Re. Accelerating llm inference with staged speculative decoding. *arXiv preprint arXiv:2308.04623*, 2023.

[23] Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. Stanford alpaca: An instruction-following llama model. https://github.com/tatsu-lab/stanford_alpaca, 2023.

[24] Yi Tay, Mostafa Dehghani, Dara Bahri, and Donald Metzler. Efficient transformers: A survey. *ACM Comput. Surv.*, 55(6), dec 2022. ISSN 0360-0300. doi: 10.1145/3530811. URL https://doi.org/10.1145/3530811.

[25] Philippe Tillet, Hsiang-Tsung Kung, and David Cox. 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*, pages 10–19, 2019.

[26] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017. URL https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.

[27] Sinong Wang, Belinda Z Li, Madian Khabsa, Han Fang, and Hao Ma. Linformer: Self-attention with linear complexity. *arXiv preprint arXiv:2006.04768*, 2020.

[28] Cong Wei, Brendan Duke, Ruowei Jiang, Parham Aarabi, Graham W Taylor, and Florian Shkurti. Sparsifiner: Learning sparse instance-dependent attention for efficient vision transformers. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 22680–22689, 2023.

[29] Manzil Zaheer, Guru Guruganesh, Kumar Avinava Dubey, Joshua Ainslie, Chris Alberti, Santiago Ontanon, Philip Pham, Anirudh Ravula, Qifan Wang, Li Yang, et al. Big bird: Transformers for longer sequences. *Advances in Neural Information Processing Systems*, 33, 2020.

APPENDIX

A Algorithm For Binary Block Masking

In this section, we present the algorithm for the forward pass of Binary Block Masking. In the following algorithm, the Query vectors are already read into the SRAM. Thus, the given algorithm will be executed in parallel across the Query blocks. The number of such blocks is given by $\lceil \frac{M}{B_m} \rceil$, where M is the total number of Query vectors.

A key distinction from the base Flash Attention mechanism lies in the reading and verification of pre-processed binary block values, followed by the application of the mask on a block-by-block basis. A similar change can also be made for the backward pass.

Algorithm 1 Masked Flash Attention Forward Pass

Require: Query block $Q \in \mathbb{R}^{B_m \times d}$, Key blocks $K \in \mathbb{R}^{B_n \times d}$, Value blocks $V \in \mathbb{R}^{B_n \times d}$
Require: Attention mask $M \in \mathbb{R}^{B_m \times B_n}$, Binary block mask $B \in \{0,1\}^{\lceil \frac{N}{B_n} \rceil}$
Require: Block sizes B_m, B_n, sequence length N, scaling factor α
Ensure: Output $O \in \mathbb{R}^{B_m \times d}$

Initialize $O \leftarrow 0$, $l \leftarrow 0$, $m \leftarrow -\infty$
for $j \leftarrow 0$ **to** $\lceil \frac{N}{B_n} \rceil - 1$ **do**
  if $B[j] = 1$ **then** ▷ Process block only if binary mask indicates non-zero values
    $S \leftarrow QK^T/\sqrt{d}$ ▷ Compute scaled dot product
    $S \leftarrow \text{APPLYMASK}(S, M)$ ▷ Apply attention mask
    $m' \leftarrow \max(m, \max(S))$ ▷ Update maximum value
    $P \leftarrow \exp(S - m')$ ▷ Compute attention weights
    $l' \leftarrow \sum P$ ▷ Compute row sums
    $\alpha \leftarrow \exp(m - m')$ ▷ Compute scaling factor
    $O \leftarrow \alpha O + PV$ ▷ Update output
    $l \leftarrow \alpha l + l'$ ▷ Update normalizing factor
    $m \leftarrow m'$ ▷ Update maximum value
  end if
end for
return O, l, m ▷ Return normalized output

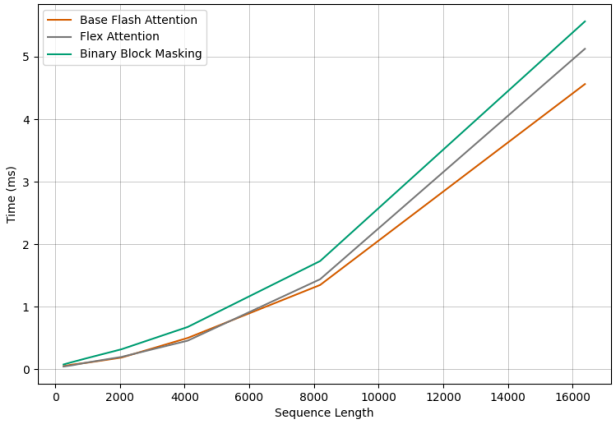
B Comparison with FlexAttention

In this section, we compare our method with the FlexAttention method and show their performance on two masking scenarios—causal and bidirectional input-output masking—using the ALPACA dataset. Both methods preprocess the mask to enable blockwise sparsity in attention computation. However, our method has distinct advantages:

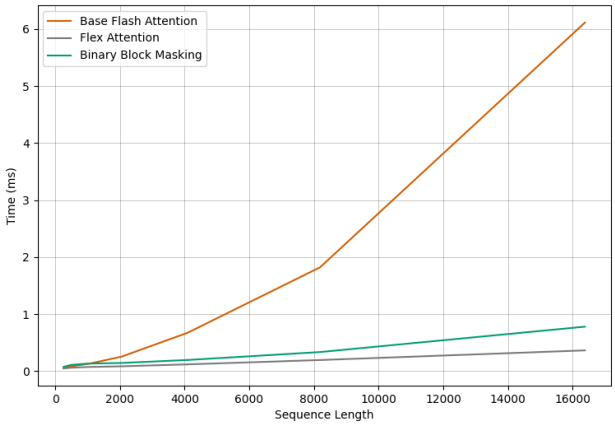
1. **Faster Preprocessing Runtimes:** As seen in Table 2, our preprocessing step is easily parallelized across blocks, incurring minimal runtime overhead, while FlexAttention's preprocessing is significantly slower.
2. **Optimized Masking Capabilities:** Our method introduces optimization for the "Sparse Attention Mask" cases by using the RCM algorithm.
3. **Ease of Use:** Since our method utilizes a direct tensor-to-tensor approach, it is much more user-friendly as it does not require the explicit `torch.compile()` step for every different instantiation, and the user does not need to consider the functional representation of the masks.

We also see in Figure 6 that our method has comparable performance with the FlexAttention method.

FlexAttention has two practical advantages: it is integrated into the PyTorch library and supports direct modification of attention scores.



(a) Performance comparison for Causal Mask



(b) Performance comparison for ALPACA input-bidirectional mask

Figure 6: Runtime Comparisons FlexAttention vs Binary Block Masking

Table 2: Flex Attention vs BinBlkMat: Preprocessing Time

Mask Size	**BinBlkMat (ms)**	**Flex Attention (ms)**
256	0.06	0.19
512	0.06	0.26
1024	0.06	0.45
2048	0.08	5.05
4096	0.19	16.92
8192	0.67	67.45
16384	2.92	265.51

C Preprocessing BinBlkMat: Method Description and Analysis

We explored several different kernel implementations to compute `BinBlkMat`. In the following descriptions, `N` refers to the sequence length or the dimension of the original mask, `BLOCKSIZE_I` is the block size along the rows, and `BLOCKSIZE_J` is the block size along the columns. The output

BinBlkMat has a shape of $N//$BLOCKSIZE_I $\times$ $N//$BLOCKSIZE_J. The approaches we tested are as follows:

1. **Convolutional approach**: We used `torch.nn.functional.conv2d` with a stride of BLOCKSIZE.
2. **Direct kernel grid**: We launch a grid of N $\times$ N kernels, where each kernel K_{ij} checks whether the corresponding entry (i,j) in the original mask was 0 or 1. If the value was 1, it assigned 1 to the index (i//BLOCKSIZE_I, j//BLOCKSIZE_J) in BinBlkMat.
3. **Block-based summation kernel**: We launched a grid of size N//BLOCKSIZE_I $\times$ N//BLOCKSIZE_J, where each kernel operated over a block of size BLOCKSIZE_I $\times$ BLOCKSIZE_J. The kernel computed the sum of the block elements, assigning a 1 to the corresponding block in BinBlkMat if the sum was greater than zero.
4. **In-built Torch operations**: Similar to the third approach, but using native PyTorch[19] operations for block summation and assignment.

Among these methods, the third approach provided the best performance. Moreover, the sum output from this approach is also used to calculate total_ones and offset for Dense Binary Block Masking cases, enabling the efficient computation of two outputs with minimal overhead. The runtime for the third approach is reported in Table 3

Table 3: Preprocessing runtime for BinBlkMat

Mask Size	BinBlkMat (ms)	BinBlkMat with total_ones and offset (ms)
256	0.06	0.12
512	0.06	0.12
1024	0.06	0.12
2048	0.08	0.14
4096	0.19	0.26
8192	0.67	0.73
16384	2.92	2.96

D Performance Comparison On Different GPUs

In the main paper, we use an RTX 3060 GPU with 6GB of VRAM for benchmark evaluations. In this section, we extend the analysis to include runtimes on four additional GPUs: A100, H100, T4, and RTX 6000 Ada Generation. We begin by presenting the runtimes of our preprocessing kernels, followed by the performance results on the Input-Bidirectional ALPACA mask[23].

D.1 Preprocessing Runtime Comparisons

Tables 4,5,6,7 give the performance comparison for A100, H100, T4, and RTX 6000 Ada Generation respectively. Apart from T4, the preprocessing times decrease even further for more powerful GPUs.

Table 4: Preprocessing runtime for BinBlkMat: RTX 3060 vs. A100

Seq. Length	RTX 3060		A100	
	BinBlkMsk	BinBlkMsk with total_ones and offset	BinBlkMsk	BinBlkMsk with total_ones and offset
256	0.06	0.12	0.07	0.16
512	0.06	0.12	0.07	0.16
1024	0.06	0.12	0.07	0.16
2048	0.08	0.15	0.07	0.16
4096	0.19	0.25	0.10	0.19
8192	0.67	0.73	0.20	0.29
16384	2.92	2.96	0.64	0.73

Table 5: Preprocessing runtime for BinBlkMat: RTX 3060 vs. H100

Seq. Length	RTX 3060		H100	
	BinBlkMsk	BinBlkMsk with total_ones and offset	BinBlkMsk	BinBlkMsk with total_ones and offset
256	0.06	0.12	0.05	0.11
512	0.06	0.12	0.05	0.11
1024	0.06	0.12	0.05	0.11
2048	0.08	0.15	0.05	0.11
4096	0.19	0.25	0.07	0.13
8192	0.67	0.73	0.15	0.21
16384	2.92	2.96	0.47	0.53

Table 6: Preprocessing runtime for BinBlkMat: RTX 3060 vs. T4

Seq. Length	RTX 3060		T4	
	BinBlkMsk	BinBlkMsk with total_ones and offset	BinBlkMsk	BinBlkMsk with total_ones and offset
256	0.06	0.12	0.11	0.21
512	0.06	0.12	0.11	0.24
1024	0.06	0.12	0.11	0.23
2048	0.08	0.15	0.15	0.27
4096	0.19	0.25	0.30	0.41
8192	0.67	0.73	1.05	1.17
16384	2.92	2.96	4.66	4.80

Table 7: Preprocessing runtime for BinBlkMat: RTX 3060 vs. RTX 6000 Ada

Seq. Length	RTX 3060		RTX 6000 Ada	
	BinBlkMsk	BinBlkMsk with total_ones and offset	BinBlkMsk	BinBlkMsk with total_ones and offset
256	0.06	0.12	0.06	0.13
512	0.06	0.12	0.06	0.13
1024	0.06	0.12	0.06	0.13
2048	0.08	0.15	0.06	0.14
4096	0.19	0.25	0.12	0.19
8192	0.67	0.73	0.41	0.49
16384	2.92	2.96	1.48	1.54

D.2 ALPACA Input-Bidirectional Mask Runtime Comparison

The tables 8,9,10,11 present the runtime comparisons for the A100, H100, T4, and RTX 6000 Ada Generation GPUs, respectively. We employed a `BLOCKSIZE` of $(128, 32)$, originally fine-tuned for the RTX 3060. While these performance metrics are not optimized for the specific hardware configurations tested, the results still clearly demonstrate the advantages of using the Binary Block Masking algorithm. Further optimization could potentially yield even better runtimes across different hardware.

Table 8: Performance comparison on ALPACA Input-Bidirectional mask: RTX 3060 vs. A100

Seq. Length	**RTX 3060**		**A100**	
	Base Flash Attention	**BinBlkMsk**	**Base Flash Attention**	**BinBlkMsk**
256	0.58	0.59	0.17	0.23
512	1.50	1.32	0.33	0.44
1024	4.45	2.64	0.86	0.80
2048	15.06	5.68	2.26	1.74
4096	54.32	13.82	7.69	4.22
8192	213.92	33.80	28.38	10.01
16384	909.90	97.34	107.92	33.75

Table 9: Performance comparison on ALPACA Input-Bidirectional mask: RTX 3060 vs. H100

Seq. Length	**RTX 3060**		**H100**	
	Base Flash Attention	**BinBlkMsk**	**Base Flash Attention**	**BinBlkMsk**
256	0.58	0.59	0.19	0.16
512	1.50	1.32	0.18	0.24
1024	4.45	2.64	0.42	0.43
2048	15.06	5.68	1.16	0.90
4096	54.32	13.82	3.82	2.16
8192	213.92	33.80	13.78	5.85
16384	909.90	97.34	50.99	18.44

Table 10: Performance comparison on ALPACA Input-Bidirectional mask: RTX 3060 vs. T4

Seq. Length	**RTX 3060**		**T4**	
	Base Flash Attention	**BinBlkMsk**	**Base Flash Attention**	**BinBlkMsk**
256	0.58	0.59	0.97	1.19
512	1.50	1.32	1.84	2.52
1024	4.45	2.64	5.15	3.36
2048	15.06	5.68	15.95	7.57
4096	54.32	13.82	57.14	19.30
8192	213.92	33.80	215.30	49.26
16384	909.90	97.34	869.90	137.02

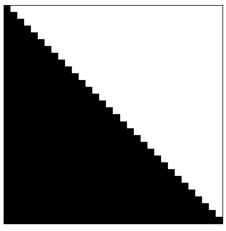
Table 11: Performance comparison on ALPACA Input-Bidirectional mask: RTX 3060 vs. RTX 6000 Ada

Seq. Length	RTX 3060		RTX 6000 Ada	
	Base Flash Attention	BinBlkMsk	Base Flash Attention	BinBlkMsk
256	0.58	0.59	0.14	0.26
512	1.50	1.32	0.28	0.29
1024	4.45	2.64	0.71	0.56
2048	15.06	5.68	2.30	1.16
4096	54.32	13.82	8.03	2.61
8192	213.92	33.80	30.69	6.48
16384	909.90	97.34	113.76	18.06

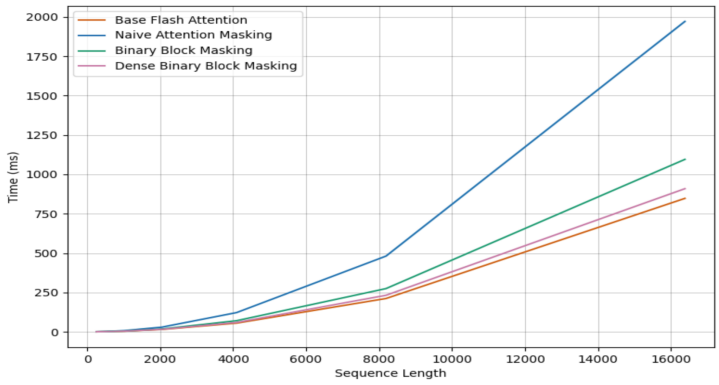
E Performance Comparison on Causal and All-Ones mask

In this section, we present the performance of our algorithm under two extreme masking conditions: the Causal Mask (Figure 7) and the All-Ones Mask (Figure 8). Our approach leverages partial mask fill to optimize computational efficiency. As the mask becomes denser, the performance of the algorithm can degrade. However, even under these extreme cases, our "Dense Binary Block Masking" technique effectively exploits the contiguous structure of the masks, resulting in comparable performance.

It is important to note that, for the purpose of this evaluation, we treat our method as mask-agnostic. While we skip most computations for the masked parts of the input, some overhead remains due to the need to read binary mask values to determine which parts of the matrix to bypass. In practical applications, our algorithm can be integrated directly with kernels optimized for causal masks. In such cases, we would achieve optimal performance by eliminating the unnecessary mask value reads entirely. Therefore, the results presented here are intended to demonstrate the efficiency gains achieved by "Dense Binary Block Masking" under extreme conditions, rather than serve as a performance baseline for causal masks.

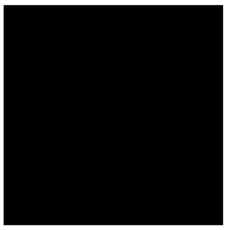


(a) Causal Mask

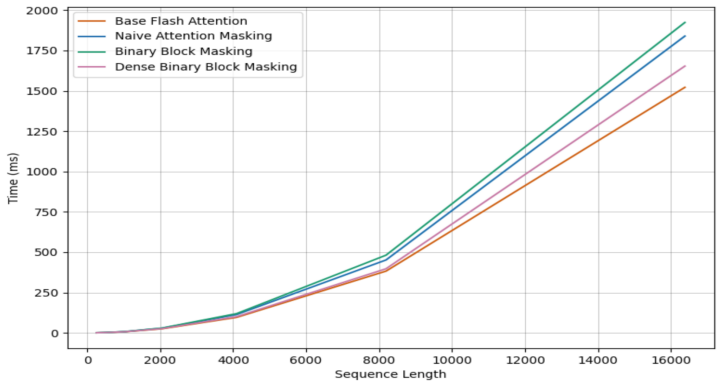


(b) Performance comparison on Causal Mask

Figure 7: Mask Visualization and Performance Comparison for Causal Masks



(a) All-Ones Mask

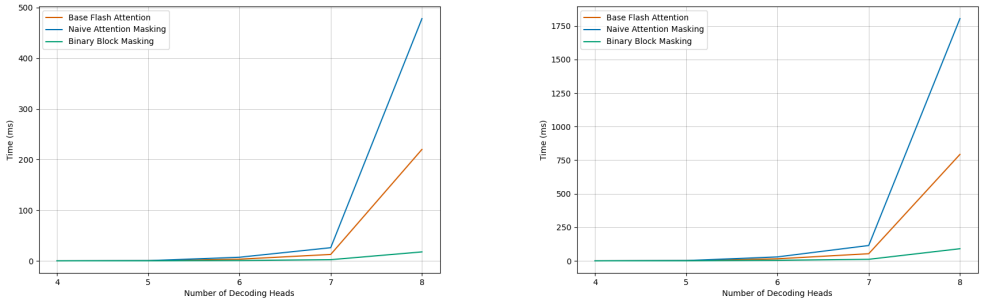


(b) Performance comparison on All-Ones Mask

Figure 8: Mask Visualization and Performance Comparison for Causal Masks

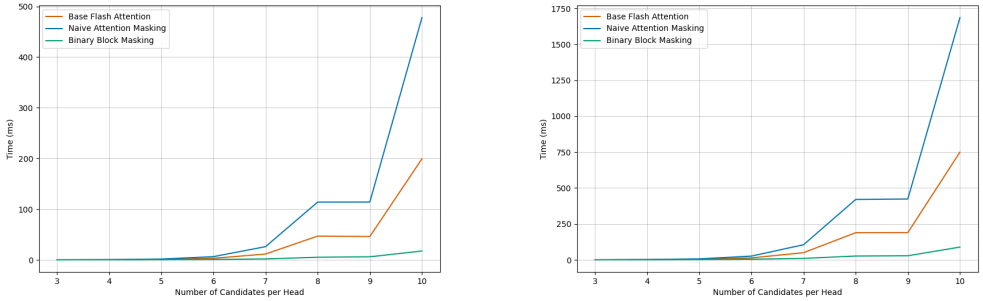
F Forward and Backward Pass Performance Comparisons

In the main paper, we reported performance benchmarks based on total runtimes. In this section, we provide a detailed breakdown of the forward and backward pass runtimes for MEDUSA masks[4] (Figures 9 and 10), ALPACA masks[23] (Figures 11 and 12), and Longformer masks[2] (Figures 13, 14, and 15)



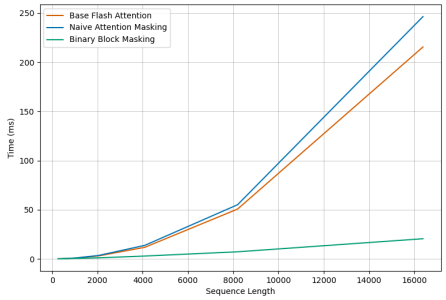
(a) Forward Pass (b) Backward Pass

Figure 9: MEDUSA: Varying Decoding Heads, Number of Candidates Fixed

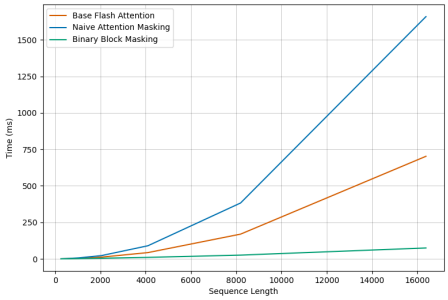


(a) Forward Pass (b) Backward Pass

Figure 10: MEDUSA: Varying Number of Candidates, Fixed Decoding Heads

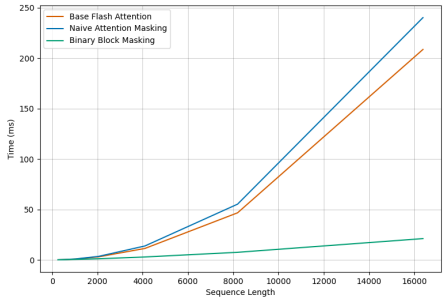


(a) Forward Pass

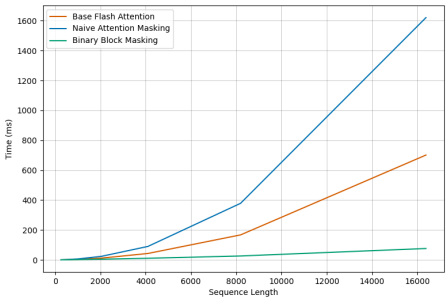


(b) Backward Pass

Figure 11: ALPACA: Sequential Mask

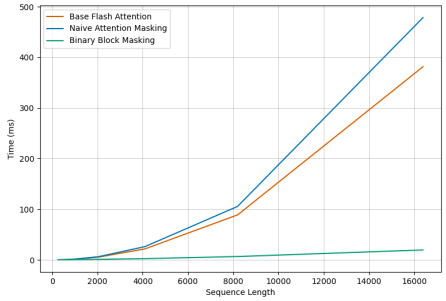


(a) Forward Pass

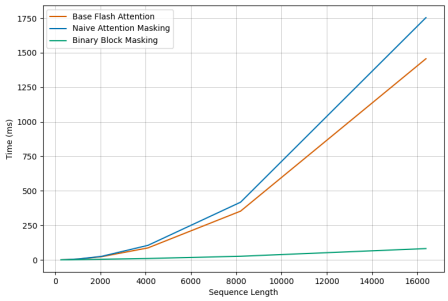


(b) Backward Pass

Figure 12: ALPACA: Input-Bidirectional Mask

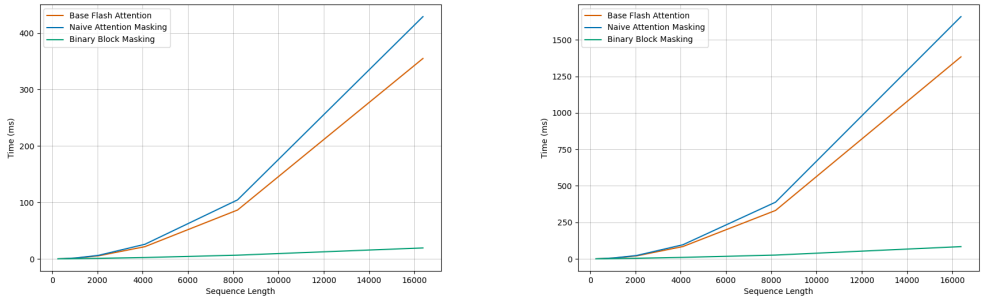


(a) Forward Pass



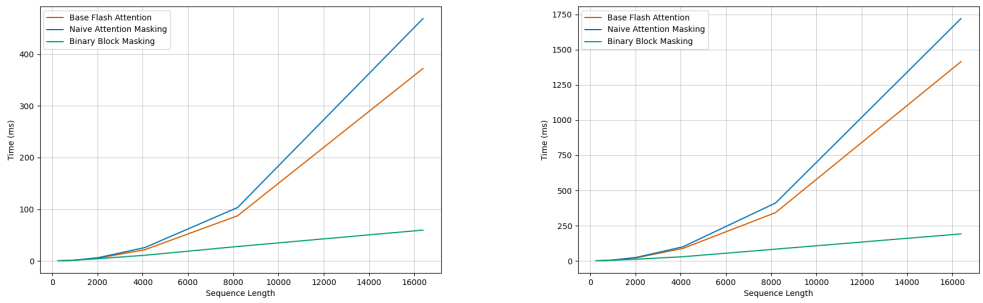
(b) Backward Pass

Figure 13: LongFormer: Windowed Mask



(a) Forward Pass (b) Backward Pass

Figure 14: LongFormer: Dilated Mask



(a) Forward Pass (b) Backward Pass

Figure 15: LongFormer: Global Mask

G Dataset and Code License Declaration

This section provides details on the licenses and sources of the code and datasets utilized in this study.

1. **ALPACA**: We obtain the ALPACA dataset from Hugging Face (URL: `https://huggingface.co/datasets/tatsu-lab/alpaca`) [23]. The dataset is licensed under the Creative Commons NonCommercial (CC BY-NC 4.0) License.
2. **MEDUSA**: We have developed our own code to generate MEDUSA masks based on the guidelines provided in the official report [4].
3. **Longformer**: We implement our own code for generating Longformer masks, as described in the original research paper [2].
4. **Flash Attention**: For our baseline, we use the public implementation of Flash Attention 2, available at URL: `https://github.com/triton-lang/triton/blob/main/python/tutorials/06-fused-attention.py`.

All other code used in this paper was developed by us.