FLASHFFTCONV: Efficient Convolutions for Long Sequences with Tensor Cores

Daniel Y. Fu[*,1], **Hermann Kumbong**[*,1], **Eric Nguyen**[2], **Christopher Ré**[1]
[*]Equal contribution. [1]Department of Computer Science, Stanford University.
[2]Department of Biongineering, Stanford University.
{danfu,kumboh,etnguyen,chrismre}@stanford.edu

Abstract

Convolution models with long filters have demonstrated state-of-the-art reasoning abilities in many long-sequence tasks but lag behind the most optimized Transformers in wall-clock time. A major bottleneck is the Fast Fourier Transform (FFT)—which allows long convolutions to run in $O(N \log N)$ time in sequence length N but has poor hardware utilization. In this paper, we study how to optimize the FFT convolution. We find two key bottlenecks: the FFT does not effectively use specialized matrix multiply units, and it incurs expensive I/O between layers of the memory hierarchy. In response, we propose FLASHFFTCONV. FLASHFFTCONV uses a matrix decomposition that computes the FFT using matrix multiply units and enables kernel fusion for long sequences, reducing I/O. FLASHFFTCONV speeds up exact FFT convolutions by up to 6.54× over PyTorch and achieves up to 4.4× speedup end-to-end. Given the same compute budget, FLASHFFTCONV allows Hyena-GPT-s to achieve 2.3 points better perplexity and M2-BERT-base to achieve 3.3 points higher GLUE score—matching models with twice the parameter count.

1 Introduction

A key challenge in machine learning is to efficiently reason over long sequences. Recently, convolutions have emerged as a key primitive for sequence modeling, underpinning state-of-the-art performance in language modeling [42, 76, 94, 110], time-series analysis [36, 46, 103, 115], computer vision [74, 81, 109], DNA modeling [82], and more [27, 55, 61, 71, 77, 80]. Despite these strong quality results—and other benefits ranging from better scaling in sequence length [46] to greater stability [9, 106]—convolutional sequence models still lag behind Transformers in wall-clock time.

A major reason is poor hardware support. Unlike classical convolutions used in vision applications, which often have short filters (e.g., 3×3 or 7×7 [53, 63]), convolutions for sequence modeling often use filters as long as the input sequence [71, 97]. Such long filters necessitate the use of the FFT convolution algorithm, which computes the convolution between an input u and convolution kernel k via a conversion to frequency space:

$$(u*k)[i] = \sum_{j}^{i} u[i]k[j-i] \quad \cong \quad u*k = \mathcal{F}^{-1}(\mathcal{F}u \odot \mathcal{F}k), \tag{1}$$

where $\mathcal{F}$ is the FFT, which can be computed in $O(N \log N)$ time in sequence length N, and $\odot$ is elementwise multiplication. Despite its asymptotic efficiency, the FFT convolution algorithm has poor wall-clock time on modern accelerators. In contrast, systems advances have pushed Transformers to the limits of modern accelerators—achieving more than 72% FLOP utilization end-to-end with FlashAttention-v2 [22, 24].

In this paper, we study how to optimize the FFT convolution algorithm on modern accelerators, to enable longer-context abilities. Just as systems advances such as FlashAttention yielded improvements in modeling quality [1, 70] and the development of new attention algorithms [2, 66, 73, 92], we hope

37th Conference on Neural Information Processing Systems (NeurIPS 2023).

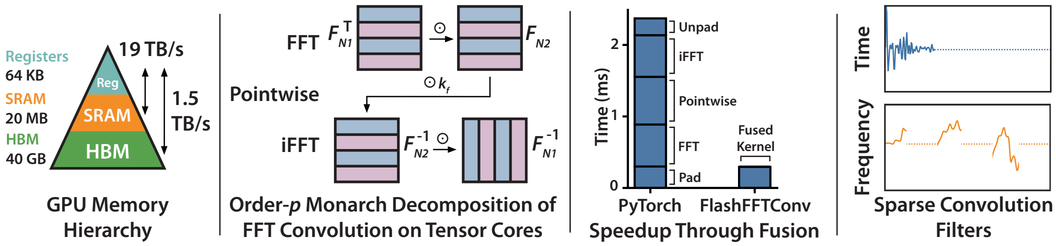

Figure 1: **Left:** GPU memory hierarchy. **Middle left:** Order-p Monarch decomposition of FFT convolution, with $p = 2$. **Middle right:** Kernel fusion for end-to-end speedup. **Right:** FLASHFFTCONV introduces analogues of sparsity for convolutions.

that understanding how to optimize the FFT convolution can also inspire algorithmic innovation, thus improving the quality of convolutional sequence models.

For short sequences, the FFT convolution is relatively easy to optimize. Kernel filters are often shared across many batches, which allows pre-computing the FFT of the filter $k_f = \mathcal{F}k$ and re-using it in a batch: $(u * k) = \mathcal{F}^{-1}(\mathcal{F}u \odot k_f)$. Thus the FFT convolution is pleasantly parallel across batches and filters, and intermediate outputs of the convolution can be cached in SRAM or registers via kernel fusion.

However, as sequence length increases, we find that two key bottlenecks emerge. First, FFT convolutions do not effectively use the specialized matrix-matrix multiply units available on modern accelerators—e.g., the H100 can use tensor cores to compute matrix-matrix multiply at 1.0 PetaFLOP/s compared to 67 TeraFLOP/s for general arithmetic. Second, sequences become too large to fit in SRAM, and kernel fusion fails, resulting in expensive I/O costs (Figure 1 middle right). These I/O costs can be exacerbated by padding operations for causality, and conversions from real-valued inputs/outputs to complex-valued FFT intermediates.

In response (Section 2), we propose FLASHFFTCONV, a new system that optimizes the FFT convolution for long sequences using a Monarch decomposition of the FFT. An order-p Monarch decomposition rewrites the FFT as a series of p matrix-matrix multiply operations (Figure 1 middle left), which can be efficiently mapped onto hardware [23]. The order p controls the number of matrix multiply operations and introduces a tradeoff: higher values of p incur lower FLOP cost via smaller matrices, but require more I/O to communicate intermediate results. Using a simple GPU cost model, we show how to adjust p based on the sequence length to balance the FLOP cost and I/O cost. This decomposition introduces a second benefit: a reduction in the amount of the sequence that needs to be kept in SRAM, which makes kernel fusion viable at longer sequence lengths. As a result, FLASHFFTCONV scales across four orders of magnitude in sequence length, from 256 to 4 million. FLASHFFTCONV also exploits a real-valued FFT algorithm to cut the length of the FFT operation in half [102], and selectively skips portions of the matrix-multiply operations when the input is zero-padded.

We evaluate FLASHFFTCONV in Section 3. We show that FLASHFFTCONV allows language models to improve in quality compared to a PyTorch model with the same compute budget. These improvements come from improvements in efficiency in the convolution, and we show that FLASHFFTCONV can speed up convolutional sequence models end-to-end across multiple modalities and sequence length, including language, vision, speech, and DNA. We even show that FLASHFFTCONV can outperform FlashAttention-v2 at sequence length 2K, due to a reduction in FLOP costs compared to attention. Finally, the Appendix gives more details of the method, extensions to sparsity, and additional experiments.

2 FLASHFFTCONV

We provide a broad overview of FLASHFFTCONV. Algorithm 1 shows an overview. Appendix C provides the cost model, additional extensions, and domain-specific optimizations.

For $N = N_1 N_2$, an order-2 Monarch FFT decomposition rewrites $\mathcal{F}_N = \mathbf{P}(\mathbf{I}_{N_2} \otimes \mathcal{F}_{N_1})\mathbf{D}\mathbf{P}^{-1}(\mathbf{I}_{N_1} \otimes \mathcal{F}_{N_2})\mathbf{P}$, where $\otimes$ denotes the Kronecker product, $\mathcal{F}_N$ is the $N \times N$ discrete Fourier matrix, $\mathbf{P}$ is a permutation matrix that reshapes the input to $N_1 \times N_2$, transposes it to $N_2 \times N_1$, and then reshapes it back to N, and $\mathbf{D} \in \mathbb{C}^{N \times N}$ is a diagonal matrix containing correctional values called Twiddle factors [6]. Higher-order Monarch decompositions recursively apply the order-2 decomposition to $\mathcal{F}_{N_1}$ or $\mathcal{F}_{N_2}$, which reduces FLOP costs but increases the number of permutation operations, increasing I/O cost.

Algorithm 1 FLASHFFTCONV core algorithm, with order-2 Monarch decomposition. We assume $N = N_1^2$ for simplicity here.

Input: Input $u \in \mathbb{R}^{B \times H \times N}$, convolution kernel $k_f \in \mathbb{C}^{H \times N}$, FFT matrices $\mathbf{F} \in \mathbb{C}^{N_1 \times N_1}$, $\mathbf{F^{-1}} \in \mathbb{C}^{N_1 \times N_1}$, Twiddle factors $t \in \mathbb{C}^N$, $t_{inv} \in \mathbb{C}^N$, B tile size B_{tile}, H tile size H_{tile}.
Output: Output $y \in \mathbb{R}^{B \times H \times N}$.

- **for** SMs in parallel across $B/B_{tile} \times H/H_{tile}$ **do**
  - Load $\mathbf{F}$, $\mathbf{F^{-1}}$, t, t_{inv} from HBM.
  - **for** $h \leftarrow 1$ to H_{tile} **do**
    - Load $\mathbf{K_f} \leftarrow k_f[h]$ from HBM, reshaped to $N_1 \times N_1$.
    - **for** $b \leftarrow 1$ to B_{tile} **do**
      - Load $\mathbf{X} \leftarrow u[b,h]$ from HBM, reshaped to $N_1 \times N_1$.
      - $\mathbf{X} \leftarrow ((\mathbf{F}^\top \mathbf{X}) * t)\mathbf{F}$ ▷ FFT, decomposed into two steps
      - $\mathbf{X} \leftarrow \mathbf{X} * \mathbf{K_f}^\top$ ▷ Elementwise multiply with k_f
      - $\mathbf{Y} \leftarrow ((\mathbf{X}\mathbf{F}^{-1})^\top * t_{inv})\mathbf{F}^{-1}$ ▷ Inverse FFT, decomposed into two steps
      - Write $\mathbf{Y}^\top$ to HBM.

Table 1: Improvement in quality given a fixed compute budget.

Model (Metric)	PyTorch	FLASHFFTCONV
M2-BERT-base-110M (GLUE Score ↑)	77.6	**80.9**
Hyena-s-155M (PPL ↓)	13.4	**11.1**

Traditionally, the decomposition broadcasts the matrix operation against the batch dimension and the hidden dimension, as shown in Figure 2 top left, which allows each $\mathcal{F}_{N_1}$ operation in the $\mathbf{I}_{N_2} \otimes \mathcal{F}_{N_1}$ matrix to run independently. However, it also makes kernel fusion difficult; fusing across the matrix multiply and permutation operations requires loading at least 16 sequences at once into SRAM to fill out the matrix multiply unit.

Instead, we broadcast the matrix operation across the entire sequence, as shown in Figure 2 top right, and run the algorithm in parallel across the batch and hidden dimensions. This reduces the SRAM requirements for kernel fusion. It also allows us to use kernel fusion for longer kernels by fusing the innermost matrix operations and elementwise multiplications, and taking an I/O each for the outermost matrix operations. Broadcasting along the sequence has an added benefit: the permutations simply become matrix transposes (Figure 2 bottom), which can be done quickly using well-established routines on-chip [84]. Finally, we also tile the computation across the B and H dimensions to reduce the cost of loading k_f, $\mathcal{F}$, and the twiddle factors from HBM.

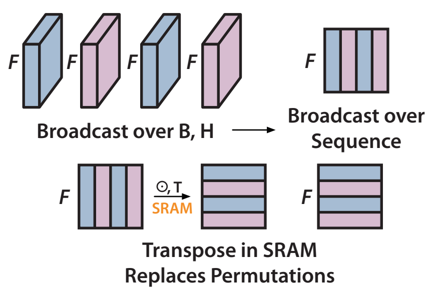

Figure 2: **Top:** FLASHFFTCONV adapts the Monarch FFT decomposition to broadcast matrix multiply operations over the sequence instead of over the batch and hidden dimensions. **Bottom:** This converts HBM permutations simple matrix transpose operations in SRAM.

3 Experiments

We evaluate FLASHFFTCONV in terms of quality and efficiency. We show that FLASHFFTCONV enables models to achieve better quality for the same compute budget, speeds up convolutions compared to other implementations, speeds up models end-to-end, and even outperforms FlashAttention-v2.

Improvement in Quality with Fixed Compute Budget We train M2-BERT-base [42] and Hyena-s [94] from scratch. We compare the quality of models trained with the same compute budget but different implementations of the convolution—either FLASHFFTCONV or a PyTorch implementation of the FFT convolution. FLASHFFTCONV achieves higher pretraining throughput, which allows the models to see more data during pretraining—with improvements in quality similar in magnitude to the effect of doubling the number of parameters in the model (see Appendix D for reference results).

FLASHFFTCONV Speeds up Convolutions Table 2 benchmarks the speed of the convolution compared against an FFT convolution implemented in PyTorch. The baseline of kernel fusion without

Table 2: Top: Time (↓) to compute the forward pass of a convolution with FLASHFFTCONV in milliseconds on one H100-SXM. Bottom: Ablations removing specific optimizations. Batch size 64, hidden dimension 768. p indicates the order of the Monarch decomposition.

	$p=2$		$p=3$				$p=4$		
Sequence Length	**256**	**1K**	**4K**	**8K**	**16K**	**32K**	**1M**	**2M**	**4M**
PyTorch	0.43	1.57	6.65	13.7	28.6	62.1	2,346.3	4,892.1	10,127.6
FLASHFFTCONV	**0.09**	**0.24**	**1.37**	**3.19**	**9.27**	**21.8**	**1,492.8**	**2,695.1**	**5,939.0**
Fusion-Only/cuFFTdx	0.21	0.67	3.51	7.71	21.4	45.5	–	–	–
Speedup over PyTorch	4.78×	6.54×	4.85×	4.29×	3.09 ×	2.85×	1.57×	1.82×	1.71×

Table 3: End-to-end throughput (↑) of convolutional sequence models against PyTorch on A100.

Model (size, seqlen, unit)	**PyTorch**	**FlashFFTConv**	**Speedup**
M2-BERT-base (110M, 128, seqs/s)	594	**2610**	4.4×
Hyena-s-4K (155M, 4K, seqs/s)	47.5	**162**	3.4×
Long convs, Path-X (102M, 16K, images/s)	126	**308**	2.4×
SaShiMi (5.4M, 64K, audio clips/s)	38.7	**50.3**	1.3×
HyenaDNA (1M, seqs/s)	3.26	**10.1**	3.1×

Table 4: End-to-end throughput (↑) in thousands of tokens per second, FLOP utilization, and speedup of Hyena against GPT running FlashAttention-v2 [22] across sequence lengths for A100.

Model	**2K**	**8K**	**16K**
GPT-2.7B, FA-v2 [22]	33.8	27.8	21.6
Hyena-2.7B, FLASHFFTCONV	**35.2**	**35.2**	**32.3**
FA-v2 FLOP Utilization	65.7	72.1	78.5
FLASHFFTCONV FLOP Utilization	62.3	61.9	56.5
FLASHFFTCONV Speedup	1.1×	1.3×	1.5×

tensor cores recovers the strong baseline of using Nvidia's cuFFTdx kernel fusion library [87]. Speedups are greatest for short sequences (up to 6.54×), where PyTorch is dominated by I/O costs, but more modest for longer sequences, which incur additional I/O costs from higher-order Monarchs.

FLASHFFTCONV Speeds Up Convolutional Sequence Models Table 3 benchmarks end-to-end throughput of convolutional sequence models across various modalities and sequence lengths spanning four orders of magnitude. Speedup varies by the size of the models and the relative amount of time spent computing the convolution compared to other parts of the models. For example, FLASHFFTCONV only speeds up the SaShiMi model by 1.3×, since other operations such as SSM-based filter generation, pooling layers, and MLPs take up more time.

FLASHFFTCONV is Faster than FlashAttention-v2 Table 4 shows throughput, end-to-end FLOP utilization, and speedup of a 2.7B-parameter Hyena model using FLASHFFTCONV against a 2.7B-parameter GPT model using FlashAttention-v2 [22] at three sequence lengths. FLASHFFTCONV achieves lower end-to-end FLOP utilization than FlashAttention-v2 but achieves higher throughput, since convolutions incur fewer overall FLOPs.

4 Conclusion

We present FLASHFFTCONV, a new system for optimizing FFT convolutions for long sequences. FLASHFFTCONV uses a Monarch decomposition of the FFT to map the FFT convolution efficiently onto tensor cores and enable kernel fusion across four orders of magnitude in sequence length. We show that FLASHFFTCONV improves quality under a fixed compute budget improves the efficiency of long convolutions, speeds up convolutional sequence models end-to-end, and enables higher-quality models given a fixed compute budget. We hope that our work will help support further adoption of convolutional sequence models, and that our insights can help inform the design of future architectures.

Acknowledgments

We gratefully acknowledge the support of DARPA under Nos. FA86501827865 (SDH) and FA86501827882 (ASED); NIH under No. U54EB020405 (Mobilize), NSF under Nos. CCF1763315 (Beyond Sparsity), CCF1563078 (Volume to Velocity), and 1937301 (RTML); ONR under No. N000141712266 (Unifying Weak Supervision); the Moore Foundation, NXP, Xilinx, LETI-CEA, Intel, IBM, Microsoft, NEC, Toshiba, TSMC, ARM, Hitachi, BASF, Accenture, Ericsson, Qualcomm, Analog Devices, the Okawa Foundation, American Family Insurance, Google Cloud, Microsoft Azure, Swiss Re, Brown Institute for Media Innovation, Department of Defense (DoD) through the National Defense Science and Engineering Graduate Fellowship (NDSEG) Program, Fannie and John Hertz Foundation, National Science Foundation Graduate Research Fellowship Program, Texas Instruments Stanford Graduate Fellowship in Science and Engineering, and members of the Stanford DAWN project: Teradata, Facebook, Google, Ant Financial, NEC, VMWare, and Infosys. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views, policies, or endorsements, either expressed or implied, of DARPA, NIH, ONR, or the U.S. Government.

References

[1] Gustaf Ahdritz, Nazim Bouatta, Sachin Kadyan, Qinghui Xia, William Gerecke, Timothy J O'Donnell, Daniel Berenberg, Ian Fisk, Niccolò Zanichelli, Bo Zhang, et al. Openfold: Retraining alphafold2 yields new insights into its learning mechanisms and capacity for generalization. *bioRxiv*, pages 2022–11, 2022.

[2] Ben Athiwaratkun, Sujan Kumar Gonugondla, Sanjay Krishna Gouda, Haifeng Qian, Hantian Ding, Qing Sun, Jun Wang, Liangfu Chen, Jiacheng Guo, Parminder Bhatia, et al. On io-efficient attention mechanisms: Context-aware bifurcated attention and the generalized multi-group attention. In *Workshop on Efficient Systems for Foundation Models@ ICML2023*, 2023.

[3] Žiga Avsec, Vikram Agarwal, Daniel Visentin, Joseph R Ledsam, Agnieszka Grabska-Barwinska, Kyle R Taylor, Yannis Assael, John Jumper, Pushmeet Kohli, and David R Kelley. Effective gene expression prediction from sequence by integrating long-range interactions. *Nature methods*, 18(10):1196–1203, 2021.

[4] Manohar Ayinala, Michael Brown, and Keshab K Parhi. Pipelined parallel fft architectures via folding transformation. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 20(6):1068–1081, 2011.

[5] Jun Ho Bahn, Jung Sook Yang, Wen-Hsiang Hu, and Nader Bagherzadeh. Parallel fft algorithms on network-on-chips. *Journal of Circuits, Systems, and Computers*, 18(02):255–269, 2009.

[6] David H Bailey. Ffts in external of hierarchical memory. In *Proceedings of the 1989 ACM/IEEE conference on Supercomputing*, pages 234–242, 1989.

[7] AJAA Bekele. Cooley-tukey fft algorithms. *Advanced algorithms*, 2016.

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

[9] Alberto Bietti and Julien Mairal. Invariance and stability of deep convolutional representations. *Advances in neural information processing systems*, 30, 2017.

[10] Mikołaj Bińkowski, Jeff Donahue, Sander Dieleman, Aidan Clark, Erich Elsen, Norman Casagrande, Luis C Cobo, and Karen Simonyan. High fidelity speech synthesis with adversarial networks. In *International Conference on Learning Representations*, 2019.

[11] Rishi Bommasani, Drew A Hudson, Ehsan Adeli, Russ Altman, Simran Arora, Sydney von Arx, Michael S Bernstein, Jeannette Bohg, Antoine Bosselut, Emma Brunskill, et al. On the opportunities and risks of foundation models. *arXiv preprint arXiv:2108.07258*, 2021.

[12] E Oran Brigham. *The fast Fourier transform and its applications*. Prentice-Hall, Inc., 1988.

[13] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[14] Beidi Chen, Tri Dao, Kaizhao Liang, Jiaming Yang, Zhao Song, Atri Rudra, and Christopher Ré. Pixelated butterfly: Simple and efficient sparse training for neural network models. 2021.

[15] Beidi Chen, Tri Dao, Eric Winsor, Zhao Song, Atri Rudra, and Christopher Ré. Scatterbrain: Unifying sparse and low-rank attention. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2021.

[16] Krzysztof Choromanski, Valerii Likhosherstov, David Dohan, Xingyou Song, Andreea Gane, Tamas Sarlos, Peter Hawkins, Jared Davis, Afroz Mohiuddin, Lukasz Kaiser, et al. Rethinking attention with performers. *arXiv preprint arXiv:2009.14794*, 2020.

[17] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311*, 2022.

[18] Eleanor Chu and Alan George. *Inside the FFT black box: serial and parallel fast Fourier transform algorithms*. CRC press, 1999.

[19] James W Cooley and John W Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of computation*, 19(90):297–301, 1965.

[20] Fiona Cunningham, James E Allen, Jamie Allen, Jorge Alvarez-Jarreta, M Ridwan Amode, Irina M Armean, Olanrewaju Austine-Orimoloye, Andrey G Azov, If Barnes, Ruth Bennett, et al. Ensembl 2022. *Nucleic acids research*, 50(D1):D988–D995, 2022.

[21] Zihang Dai, Guokun Lai, Yiming Yang, and Quoc Le. Funnel-transformer: Filtering out sequential redundancy for efficient language processing. *Advances in neural information processing systems*, 33:4271–4282, 2020.

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

[23] Tri Dao, Beidi Chen, Nimit S Sohoni, Arjun Desai, Michael Poli, Jessica Grogan, Alexander Liu, Aniruddh Rao, Atri Rudra, and Christopher Ré. Monarch: Expressive structured matrices for efficient and accurate training. In *International Conference on Machine Learning*. PMLR, 2022.

[24] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. FlashAttention: Fast and memory-efficient exact attention with IO-awareness. In *Advances in Neural Information Processing Systems*, 2022.

[25] Tri Dao, Albert Gu, Matthew Eichhorn, Atri Rudra, and Christopher Ré. Learning fast algorithms for linear transforms using butterfly factorizations. *arXiv preprint arXiv:1903.05895*, 2020.

[26] Tri Dao, Nimit S. Sohoni, Albert Gu, Matthew Eichhorn, Amit Blonder, Megan Leszczynski, Atri Rudra, and Christopher Ré. Kaleidoscope: An efficient, learnable representation for all structured linear maps. *arXiv preprint arXiv:2012.14966*, 2021.

[27] Shmuel Bar David, Itamar Zimerman, Eliya Nachmani, and Lior Wolf. Decision s4: Efficient sequence-based rl via state spaces layers. In *The Eleventh International Conference on Learning Representations*, 2022.

[28] Christopher De Sa, Albert Gu, Rohan Puttagunta, Christopher Ré, and Atri Rudra. A two-pronged progress in structured dense matrix vector multiplication. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1060–1079. SIAM, 2018.

[29] Tim Dettmers and Luke Zettlemoyer. Sparse networks from scratch: Faster training without losing performance. *arXiv preprint arXiv:1907.04840*, 2019.

[30] Prafulla Dhariwal, Heewoo Jun, Christine Payne, Jong Wook Kim, Alec Radford, and Ilya Sutskever. Jukebox: A generative model for music. *arXiv preprint arXiv:2005.00341*, 2020.

[31] Sander Dieleman, Aaron van den Oord, and Karen Simonyan. The challenge of realistic music generation: modelling raw audio at scale. *Advances in neural information processing systems*, 31, 2018.

[32] Xin Dong, Shangyu Chen, and Sinno Pan. Learning to prune deep neural networks via layer-wise optimal brain surgeon. *Advances in Neural Information Processing Systems*, 30, 2017.

[33] Nan Du, Yanping Huang, Andrew M Dai, Simon Tong, Dmitry Lepikhin, Yuanzhong Xu, Maxim Krikun, Yanqi Zhou, Adams Wei Yu, Orhan Firat, et al. Glam: Efficient scaling of language models with mixture-of-experts. In *International Conference on Machine Learning*, pages 5547–5569. PMLR, 2022.

[34] Yuli Eidelman and Israel Gohberg. On a new class of structured matrices. *Integral Equations and Operator Theory*, 34(3):293–324, 1999.

[35] Murali Emani, Venkatram Vishwanath, Corey Adams, Michael E Papka, Rick Stevens, Laura Florescu, Sumti Jairath, William Liu, Tejas Nama, and Arvind Sujeeth. Accelerating scientific applications with sambanova reconfigurable dataflow architecture. *Computing in Science & Engineering*, 23(2):114–119, 2021.

[36] Yassir Fathullah, Chunyang Wu, Yuan Shangguan, Junteng Jia, Wenhan Xiong, Jay Mahadeokar, Chunxi Liu, Yangyang Shi, Ozlem Kalinli, Mike Seltzer, et al. Multi-head state space model for speech recognition. *arXiv preprint arXiv:2305.12498*, 2023.

[37] William Fedus, Barret Zoph, and Noam Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *The Journal of Machine Learning Research*, 23(1):5232–5270, 2022.

[38] Quentin Fournier, Gaétan Marceau Caron, and Daniel Aloise. A practical survey on faster and lighter transformers. *ACM Computing Surveys*, 2021.

[39] Jonathan Frankle and Michael Carbin. The lottery ticket hypothesis: Finding sparse, trainable neural networks. *arXiv preprint arXiv:1803.03635*, 2018.

[40] Jonathan Frankle, Gintare Karolina Dziugaite, Daniel Roy, and Michael Carbin. Linear mode connectivity and the lottery ticket hypothesis. In *International Conference on Machine Learning*, pages 3259–3269. PMLR, 2020.

[41] Jonathan Frankle, Gintare Karolina Dziugaite, Daniel M Roy, and Michael Carbin. Stabilizing the lottery ticket hypothesis. *arXiv preprint arXiv:1903.01611*, 2019.

[42] Daniel Y. Fu, Simran Arora, Jessica Grogan, Isys Johnson, Sabri Eyuboglu, Armin W. Thomas, Benjamin F. Spector, Michael Poli, Atri Rudra, and Christopher Ré. Monarch Mixer: A simple sub-quadratic GEMM-based architecture. In *Advances in Neural Information Processing Systems*, 2023.

[43] Daniel Y. Fu, Tri Dao, Khaled K. Saab, Armin W. Thomas, Atri Rudra, and Christopher Ré. Hungry Hungry Hippos: Towards language modeling with state space models. In *International Conference on Learning Representations*, 2023.

[44] Daniel Y. Fu, Elliot L. Epstein, Eric Nguyen, Armin W. Thomas, Michael Zhang, Tri Dao, Atri Rudra, and Christopher Ré. Simple hardware-efficient long convolutions for sequence modeling. *International Conference on Machine Learning*, 2023.

[45] Karan Goel, Albert Gu, Chris Donahue, and Christopher Ré. It's raw! audio generation with state-space models. *arXiv preprint arXiv:2202.09729*, 2022.

[46] Albert Gu, Karan Goel, and Christopher Re. Efficiently modeling long sequences with structured state spaces. In *International Conference on Learning Representations*, 2021.

[47] Albert Gu, Ankit Gupta, Karan Goel, and Christopher Ré. On the parameterization and initialization of diagonal state space models. In *Advances in Neural Information Processing Systems*, 2022.

[48] Albert Gu, Isys Johnson, Aman Timalsina, Atri Rudra, and Christopher Ré. How to train your hippo: State space models with generalized orthogonal basis projections. *arXiv preprint arXiv:2206.12037*, 2022.

[49] Ankit Gupta, Albert Gu, and Jonathan Berant. Diagonal state spaces are as effective as structured state spaces. In *Advances in Neural Information Processing Systems*, 2022.

[50] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.

[51] Song Han, Jeff Pool, John Tran, and William Dally. Learning both weights and connections for efficient neural network. *Advances in neural information processing systems*, 28, 2015.

[52] Ramin Hasani, Mathias Lechner, Tsun-Huang Wang, Makram Chahine, Alexander Amini, and Daniela Rus. Liquid structural state-space models. *arXiv preprint arXiv:2209.12951*, 2022.

[53] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[54] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[55] Md Mohaiminul Islam, Mahmudul Hasan, Kishan Shamsundar Athrey, Tony Braskich, and Gedas Bertasius. Efficient movie scene detection using state-space transformers. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 18749–18758, 2023.

[56] Yanrong Ji, Zhihan Zhou, Han Liu, and Ramana V Davuluri. DNABERT: pre-trained bidirectional encoder representations from transformers model for DNA-language in genome. *Bioinformatics*, 37(15):2112–2120, 2021.

[57] Norm Jouppi, George Kurian, Sheng Li, Peter Ma, Rahul Nagarajan, Lifeng Nai, Nishant Patil, Suvinay Subramanian, Andy Swing, Brian Towles, et al. Tpu v4: An optically reconfigurable supercomputer for machine learning with hardware support for embeddings. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, pages 1–14, 2023.

[58] Thomas Kailath, Sun-Yuan Kung, and Martin Morf. Displacement ranks of matrices and linear equations. *Journal of Mathematical Analysis and Applications*, 68(2):395–407, 1979.

[59] Nal Kalchbrenner, Erich Elsen, Karen Simonyan, Seb Noury, Norman Casagrande, Edward Lockhart, Florian Stimberg, Aaron Oord, Sander Dieleman, and Koray Kavukcuoglu. Efficient neural audio synthesis. In *International Conference on Machine Learning*, pages 2410–2419. PMLR, 2018.

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

[61] Sanghyeon Kim and Eunbyung Park. Smpconv: Self-moving point representations for continuous convolution. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 10289–10299, 2023.

[62] Nikita Kitaev, Łukasz Kaiser, and Anselm Levskaya. Reformer: The efficient transformer. *arXiv preprint arXiv:2001.04451*, 2020.

[63] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25, 2012.

[64] Kundan Kumar, Rithesh Kumar, Thibault De Boissiere, Lucas Gestin, Wei Zhen Teoh, Jose Sotelo, Alexandre De Brebisson, Yoshua Bengio, and Aaron C Courville. Melgan: Generative adversarial networks for conditional waveform synthesis. *Advances in neural information processing systems*, 32, 2019.

[65] Mitsuru Kusumoto, Takuya Inoue, Gentaro Watanabe, Takuya Akiba, and Masanori Koyama. A graph theoretic framework of recomputation algorithms for memory-efficient backpropagation. *Advances in Neural Information Processing Systems*, 32, 2019.

[66] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023.

[67] Kushal Lakhotia, Eugene Kharitonov, Wei-Ning Hsu, Yossi Adi, Adam Polyak, Benjamin Bolte, Tu-Anh Nguyen, Jade Copet, Alexei Baevski, Abdelrahman Mohamed, et al. On generative spoken language modeling from raw audio. *Transactions of the Association for Computational Linguistics*, 9:1336–1354, 2021.

[68] Adam Lavely. Powering extreme-scale hpc with cerebras wafer-scale accelerators. *Cerebras White Paper*, 2022.

[69] Binrui Li, Shenggan Cheng, and James Lin. tcfft: Accelerating half-precision fft through tensor cores. *arXiv preprint arXiv:2104.11471*, 2021.

[70] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*, 2023.

[71] Yuhong Li, Tianle Cai, Yi Zhang, Deming Chen, and Debadeepta Dey. What makes convolutional models great on long sequence modeling? *arXiv preprint arXiv:2210.09298*, 2022.

[72] Ji Lin, Yongming Rao, Jiwen Lu, and Jie Zhou. Runtime neural pruning. *Advances in neural information processing systems*, 30, 2017.

[73] Hao Liu and Pieter Abbeel. Blockwise parallel transformer for long context large models. *arXiv preprint arXiv:2305.19370*, 2023.

[74] Zhuang Liu, Hanzi Mao, Chao-Yuan Wu, Christoph Feichtenhofer, Trevor Darrell, and Saining Xie. A convnet for the 2020s. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 11976–11986, 2022.

[75] Xuezhe Ma, Xiang Kong, Sinong Wang, Chunting Zhou, Jonathan May, Hao Ma, and Luke Zettlemoyer. Luna: Linear unified nested attention. *Advances in Neural Information Processing Systems*, 34:2441–2453, 2021.

[76] Xuezhe Ma, Chunting Zhou, Xiang Kong, Junxian He, Liangke Gui, Graham Neubig, Jonathan May, and Luke Zettlemoyer. Mega: moving average equipped gated attention. *arXiv preprint arXiv:2209.10655*, 2022.

[77] Temesgen Mehari and Nils Strodthoff. Towards quantitative precision for ecg analysis: Leveraging state space models, self-supervision and patient metadata. *IEEE Journal of Biomedical and Health Informatics*, 2023.

[78] Harsh Mehta, Ankit Gupta, Ashok Cutkosky, and Behnam Neyshabur. Long range language modeling via gated state spaces. *arXiv preprint arXiv:2206.13947*, 2022.

[79] Maxim Milakov and Natalia Gimelshein. Online normalizer calculation for softmax. *arXiv preprint arXiv:1805.02867*, 2018.

[80] Koichi Miyazaki, Masato Murata, and Tomoki Koriyama. Structured state space decoder for speech recognition and synthesis. In *ICASSP 2023-2023 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 1–5. IEEE, 2023.

[81] Eric Nguyen, Karan Goel, Albert Gu, Gordon Downs, Preey Shah, Tri Dao, Stephen Baccus, and Christopher Ré. S4nd: Modeling images and videos as multidimensional signals with state spaces. In *Advances in neural information processing systems*, 2022.

[82] Eric Nguyen, Michael Poli, Marjan Faizi, Armin Thomas, Callum Birch-Sykes, Michael Wornow, Aman Patel, Clayton Rabideau, Stefano Massaroli, Yoshua Bengio, et al. Hyenadna: Long-range genomic sequence modeling at single nucleotide resolution. In *Advances in Neural Information Processing Systems*, 2023.

[83] NVIDIA. Nvidia Tesla V100 GPU architecture, 2017.

[84] NVIDIA. Nvidia A100 tensor core GPU architecture, 2020.

[85] NVIDIA. Nvidia H100 tensor core GPU architecture, 2022.

[86] NVIDIA. Cuda c++ programming guide, 2023. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[87] NVIDIA. cufftdx v1.1.0 documentation, 2023. https://docs.nvidia.com/cuda/cufftdx/index.html.

[88] NVIDIA. Cutlass 3.2, 2023. https://github.com/NVIDIA/cutlass.

[89] Aaron van den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew Senior, and Koray Kavukcuoglu. Wavenet: A generative model for raw audio. *arXiv preprint arXiv:1609.03499*, 2016.

[90] Alan V Oppenheim. Applications of digital signal processing. *Englewood Cliffs*, 1978.

[91] Alan V Oppenheim, John R Buck, and Ronald W Schafer. *Discrete-time signal processing. Vol. 2*. Upper Saddle River, NJ: Prentice Hall, 2001.

[92] Daniele Paliotta, Matteo Pagliardini, Martin Jaggi, and François Fleuret. Fast causal attention with dynamic sparsity. In *Workshop on Efficient Systems for Foundation Models@ ICML2023*, 2023.

[93] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

[94] Michael Poli, Stefano Massaroli, Eric Nguyen, Daniel Y Fu, Tri Dao, Stephen Baccus, Yoshua Bengio, Stefano Ermon, and Christopher Ré. Hyena hierarchy: Towards larger convolutional language models. *Proceedings of the 40th International Conference on Machine Learning (ICML 2023)*, 2023.

[95] Ryan Prenger, Rafael Valle, and Bryan Catanzaro. Waveglow: A flow-based generative network for speech synthesis. In *ICASSP 2019-2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 3617–3621. IEEE, 2019.

[96] David W Romero, Robert-Jan Bruintjes, Jakub M Tomczak, Erik J Bekkers, Mark Hoogendoorn, and Jan C van Gemert. Flexconv: Continuous kernel convolutions with differentiable kernel sizes. *arXiv preprint arXiv:2110.08059*, 2021.

[97] David W Romero, Anna Kuzina, Erik J Bekkers, Jakub Mikolaj Tomczak, and Mark Hoogendoorn. Ckconv: Continuous kernel convolution for sequential data. In *International Conference on Learning Representations*, 2021.

[98] Victor Sanh, Thomas Wolf, and Alexander Rush. Movement pruning: Adaptive sparsity by fine-tuning. *Advances in Neural Information Processing Systems*, 33:20378–20389, 2020.

[99] Jonathan Shen, Ruoming Pang, Ron J Weiss, Mike Schuster, Navdeep Jaitly, Zongheng Yang, Zhifeng Chen, Yu Zhang, Yuxuan Wang, Rj Skerrv-Ryan, et al. Natural tts synthesis by conditioning wavenet on mel spectrogram predictions. In *2018 IEEE international conference on acoustics, speech and signal processing (ICASSP)*, pages 4779–4783. IEEE, 2018.

[100] Vikas Sindhwani, Tara Sainath, and Sanjiv Kumar. Structured transforms for small-footprint deep learning. *Advances in Neural Information Processing Systems*, 28, 2015.

[101] Jimmy TH Smith, Andrew Warrington, and Scott Linderman. Simplified state space layers for sequence modeling. In *The Eleventh International Conference on Learning Representations*, 2023.

[102] H V Sorensen, D Jones, Michael Heideman, and C Burrus. Real-valued fast fourier transform algorithms. *IEEE Transactions on acoustics, speech, and signal processing*, 35(6):849–863, 1987.

[103] Siyi Tang, Jared A Dunnmon, Liangqiong Qu, Khaled K Saab, Christopher Lee-Messer, and Daniel L Rubin. Spatiotemporal modeling of multivariate signals with graph neural networks and structured state space models. *arXiv preprint arXiv:2211.11176*, 2022.

[104] Yi Tay, Mostafa Dehghani, Samira Abnar, Yikang Shen, Dara Bahri, Philip Pham, Jinfeng Rao, Liu Yang, Sebastian Ruder, and Donald Metzler. Long range arena: A benchmark for efficient transformers. In *International Conference on Learning Representations*, 2020.

[105] Yi Tay, Mostafa Dehghani, Dara Bahri, and Donald Metzler. Efficient transformers: A survey. *ACM Computing Surveys*, 55(6):1–28, 2022.

[106] Yi Tay, Mostafa Dehghani, Jai Prakash Gupta, Vamsi Aribandi, Dara Bahri, Zhen Qin, and Donald Metzler. Are pretrained convolutions better than pretrained transformers? In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 4349–4359, 2021.

[107] Yi Tay, Vinh Q Tran, Sebastian Ruder, Jai Gupta, Hyung Won Chung, Dara Bahri, Zhen Qin, Simon Baumgartner, Cong Yu, and Donald Metzler. Charformer: Fast character transformers via gradient-based subword tokenization. *arXiv preprint arXiv:2106.12672*, 2021.

[108] Aaron Van Den Oord, Oriol Vinyals, et al. Neural discrete representation learning. *Advances in neural information processing systems*, 30, 2017.

[109] Jue Wang, Wentao Zhu, Pichao Wang, Xiang Yu, Linda Liu, Mohamed Omar, and Raffay Hamid. Selective structured state-spaces for long-form video understanding. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 6387–6397, 2023.

[110] Junxiong Wang, Jing Nathan Yan, Albert Gu, and Alexander M Rush. Pretraining without attention. *arXiv preprint arXiv:2212.10544*, 2022.

[111] Qipeng Wang, Mengwei Xu, Chao Jin, Xinran Dong, Jinliang Yuan, Xin Jin, Gang Huang, Yunxin Liu, and Xuanzhe Liu. Melon: Breaking the memory wall for resource-efficient on-device machine learning. In *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services*, pages 450–463, 2022.

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

[113] 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*, 33:17283–17297, 2020.

[114] Dan Zhang, Safeen Huda, Ebrahim Songhori, Kartik Prabhu, Quoc Le, Anna Goldie, and Azalia Mirhoseini. A full-stack search technique for domain optimized deep learning accelerators. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 27–42, 2022.

[115] Michael Zhang, Khaled Kamal Saab, Michael Poli, Tri Dao, Karan Goel, and Christopher Re. Effectively modeling time series with simple discrete state spaces. In *International Conference on Learning Representations*, 2022.

[116] Chen Zhu, Wei Ping, Chaowei Xiao, Mohammad Shoeybi, Tom Goldstein, Anima Anandkumar, and Bryan Catanzaro. Long-short transformer: Efficient transformers for language and vision. *Advances in Neural Information Processing Systems*, 34:17723–17736, 2021.

Appendix

We present extended related work (Appendix A), additional algorithmic details (Appendix C), additional experimental results (Appendix D), and experimental details (Appendix E).

A Related Work

Long Convolutions in Sequence Modeling Long convolutional models have emerged as a promising alternative to Transformers for sequence modeling [42–44, 46–48, 52, 76, 82, 94, 96, 97, 101]. These methods differ in how they generate the convolutional kernels; for example, the S4 line of work uses learned state space models [46, 49, 76, 78], while other works [94, 96, 97] parameterize the convolution using an MLP from positional encodings. However, all the models operate by taking a convolution over the input sequence with a kernel as long as the input: $y = u * k$, where $u \in \mathbb{R}^{B \times H \times N}, k \in \mathbb{R}^{H \times N}$, and the kernel k is broadcast along the B dimension. When used for language modeling, these models often incorporate elementwise multiplicative gating as well: $y = f(u) \odot ((g(u) \odot h(u)) * k)$, where f, g, and h are linear maps along the H dimension [42, 43, 78, 94, 110].

Long-Context Applications Long convolutional models have especially been helpful for long-context applications, such as DNA modeling and speech synthesis. In DNA modeling, most longer-context genomic models have relied on either tokenization [56, 107, 113] or downsampling [3, 38]. However, recent work has suggested that modeling DNA directly from base pairs can yield downstream improvements in quality, which requires long sequence lengths [82].

Like DNA modeling, speech synthesis has also benefited from long-context modeling. While traditional speech synthesis pipelines use intermediate representations such as spectrograms [64, 95, 99], linguistic features [10, 59, 89], or discrete audio codes [30, 31, 67, 108], recent work has shown that modeling the speech directly from the raw waveform can yield downstream improvements in quality [45]. Again, such models require long sequences to model audio at the rate at which it is naturally sampled, necessitating long-sequence modeling.

FFT Algorithms There is a long history of efficient FFT algorithms, ranging from the Cooley-Tukey FFT algorithm published in 1965 [19] to parallel FFT algorithms [4] and more [5, 6, 18]. These algorithms have enabled fundamental progress in a range of disciplines, from control theory [7, 12] to signal processing [90, 91]. As FFTs prove more useful for modern deep learning applications, such as long convolutions, new techniques are required to run them efficiently on modern accelerators. Our work continues a line of work exploring how to use tensor cores for the FFT convolution [43, 44, 69], and extends the algorithmic capabilities to much longer sequences.

Sparsity in Deep Learning As deep learning models have grown larger and deeper [11, 13, 17], there is increasing interest in reducing the cost of training and running models. Sparsity in particular has received a great deal of attention, and has a long history in machine learning, including work in pruning neural networks [32, 50, 51, 72, 98] and finding lottery tickets [39–41]. Our work in partial convolutions and frequency-sparse convolutions relates to this line of work, as an analogue of sparsity in convolutional filters. The Monarch decomposition is also closely related to structured matrices. Structured matrices have subquadratic ($o(n^2)$ for dimension $n \times n$) parameters and runtime, such as sparse and low-rank matrices, and fast transforms (Fourier, Chebyshev, sine/cosine, orthogonal polynomials) [23]. Structured matrices can often be computed with simple divide-and-conquer schemes, and can be used to represent many fast transforms [28, 34, 58, 100].

Optimization of deep learning primitives There is a rich history of optimizing deep learning primitives. Many techniques, such as kernel fusion, aim to reduce data movement. Recently, libraries such as PyTorch 2.0 [93] have added kernel fusion automatically. Other techniques include checkpointing, wherein one stores fewer intermediate results and recomputes the others on-the-fly where they are needed, trading additional compute for memory [65, 111]. Many algorithms also have hand-optimizations that can remove unnecessary computation or memory accesses [79].

Another line of optimization techniques aims to reduce FLOPs. MLPs and attention are particularly popular targets of FLOP reduction, via sparse factorizations of weights [14, 19, 23, 25, 26, 29, 39, 116], or sparse/low-rank approximations of attention [8, 16, 21, 33, 37, 60, 62, 75, 112, 116] and their combinations [15, 105].

B Background

We provide some background on the FFT convolution and the Monarch FFT decomposition, and discuss the performance characteristics of GPUs.

B.1 FFT Convolution

Recall the definition of a convolution operation: $(u*k)[i] = \sum_j^i u_j k_{i-j}$. Computing this formula directly incurs $O(NN_k)$ FLOPs in sequence length N and kernel length N_k. For long convolutions, where $N_k = N$, a popular strategy is to use the Fourier transform to convert the signal u and kernel k to the frequency domain, and compute the convolution using pointwise multiplication in frequency domain, using Equation 1. Critically, a Fourier transform $\mathcal{F}_N$ over an input of length N can be computed in $O(N\log N)$ time using the FFT—bringing the overall cost of the long convolution from $O(N^2)$ to $O(N\log N)$. We note that the FFT convolution technically computes a circular convolution $\sum_j^N u_j k_{i-j}$, where $i-j<0$ loops back to the end of k. For this reason, u and k are often padded with zeros to compute a causal convolution.

Monarch FFT Decomposition For $N = N_1 N_2$, an order-2 Monarch FFT decomposition rewrites $\mathcal{F}_N = \mathbf{P}(\mathbf{I}_{N_2} \otimes \mathcal{F}_{N_1})\mathbf{D}\mathbf{P}^{-1}(\mathbf{I}_{N_1} \otimes \mathcal{F}_{N_2})\mathbf{P}$, where $\otimes$ denotes the Kronecker product, $\mathcal{F}_N$ is the $N\times N$ discrete Fourier matrix, $\mathbf{P}$ is a permutation matrix that reshapes the input to $N_1 \times N_2$, transposes it to $N_2 \times N_1$, and then reshapes it back to N, and $\mathbf{D} \in \mathbb{C}^{N\times N}$ is a diagonal matrix containing correctional values called Twiddle factors [6]. Higher-order Monarch decompositions recursively apply the order-2 decomposition to $\mathcal{F}_{N_1}$ or $\mathcal{F}_{N_2}$, which reduces FLOP costs but increases the number of permutation operations, increasing I/O cost.

B.2 GPU Performance Characteristics

We provide some background on the GPU memory hierarchy and available compute units, as well as compute-bound vs. memory-bound operations. We focus on GPU programming in this paper, but the general principles extend to most modern hardware accelerators [35, 57, 68, 114].

GPU Compute Model and Memory Hierarchy GPUs have a memory hierarchy consisting of global memory (HBM), shared memory (SRAM), and registers, as shown in Figure 1 Left. Lower/larger levels of the memory hierarchy have more space but are much slower, whereas higher/smaller levels of the memory hierarchy have less space but are much faster [83–85]. The memory hierarchy is closely tied to the GPU compute model. A GPU is composed of many independent streaming multiprocessors (SMs), each of which is composed of independent threads. HBM is shared among all SMs, but each SM has an independent SRAM. The SRAM is shared among all the threads in the SM. Each thread has access to its own registers, but cannot access the registers of other threads. Thus, performing global operations between SMs requires moving data to and from HBM, whereas independent work in each SM can remain local to SRAM.

GPU Compute Units Modern GPUs (since the V100 [83]) have specialized matrix multiply units called tensor cores, which can compute matrix-matrix multiply operations with much higher TFLOPs than the general-purpose compute units. For example, the H100 tensor core can compute matrix multiplication between 16×16 matrices at 1.0 PFLOPs, whereas the general-purpose compute units can only compute at 67 TFLOPs [85].

Memory-Bound vs. Compute-Bound Operations GPU operations can be memory-bound or compute-bound. Memory-bound operations are bottlenecked by the amount of I/O between HBM and registers they need to perform, and are limited by the bandwidth of the memory hierarchy. Examples include simple pointwise operations such as addition or multiplication, as well as most traditional FFT implementations. Compute-bound operations are bottlenecked by the amount of FLOPs they need to execute, and are limited by the speed of the compute units. Examples include large matrix multiply operations.

Kernel Fusion A popular method for reducing I/O costs is kernel fusion—loading data for multiple operations into SRAM, computing them independently in each SM, and then writing the final results back to HBM. Kernel fusion is common (and can be automated) for pointwise operations [93], but is more challenging for complex operations that require referencing multiple pieces of data. For example, fusing the operations in attention was not common until the development of FlashAttention [24].

C Algorithm Details

C.1 Domain-Specific Optimizations

We use a few domain-specific optimizations to adapt the convolution specifically for the sequence learning workload. First, since the convolutions used in sequence learning are real-to-real convolutions (with real kernel weights), we can use a classic algorithm called one-stage decimation in time to compute the FFT of a sequence of length N using a complex FFT of length $N/2$ (see Appendix C)—cutting the FFT cost in half. Second, inputs and outputs are often padded with zeros in the convolution to compute a causal convolution [42, 46, 94]. We special-case this padding, and use it to eliminate half of the outermost matrix multiply operations in the FFT and iFFT. We also fuse in additional operations around the convolution, such as elementwise-gating, to further reduce I/O.

We review the details of how to compute a real-to-real FFT of size N using a complex FFT of size $N/2$, following a tutorial by [102].

For this section, we adopt notation common in describing FFT algorithms. Let $x(n)$ be an input sequence of length N, and let $X(k)$ be the result of its discrete Fourier transform. Recall that:

$$X(k)=\sum_{n=0}^{N-1}x(n)W_N^{nk}, \tag{2}$$

for $k=0,1,...,N-1$, where $W_N=e^{-2\pi i/N}$ is the Nth root of unity.

First, if $x(n)$ is real, then symmetries emerge in $X(k)$. In particular, we have $X(k)=X^*(-k)=X^*(N-k)$, where * denotes complex conjugation. These symmetries allow us to have an algorithm for computing $X(k)$ using a single complex DFT of size $N/2$.

In particular:

$$\begin{aligned}X(k)&=\sum_{n=0}^{N-1}x(n)W_N^{nk}\\&=\sum_{n=0}^{N/2-1}x(2n)W_{N/2}^{nk}+W_N^k\sum_{n=0}^{N/2-1}x(2n+1)W_{N/2}^{nk},\end{aligned}$$

for $k=0,1,\ldots,N-1$. The DFT is now decomposed into two parts: a DFT over the even-indexed elements of $x(n)$, and over the odd-indexed elements of $x(n)$.

We can now create a third complex sequence, of length $N/2$, and put the even-indexed elements of $x(n)$ in the real part, and the odd-indexed elements of $x(n)$ in the imaginary part. Let:

$$z(n)=x(2n)+ix(2n+1),$$

for $n=0,1,...,N/2-1$. Then, we compute the $N/2$-sized DFT $Z(k)$, and we can recover the DFT over the even and odd parts of $x(n)$ ($X_e[k]$ and $X_o[k]$, respectively):

$$\begin{aligned}X_e[k]&=\frac{Z[k]+Z^*[N/2-k]}{2}\\X_o[k]&=-i\frac{Z[k]-Z^*[N/2-k]}{2i}.\end{aligned}$$

We can now recover $X[k], k=0...,N-1$ using:

$$X[k]=X_e[k \mod N/2]+X_o[k \mod N/2]W_N^k.$$

The inverse FFT proceeds similarly. The goal is to recover $x(n)$ given an input $X[k]$, using a simple complex inverse DFT of length $N/2$.

First, we recover $X_e[k]$ and $X_o[k]$:

$$\begin{aligned}X_e[k]&=\frac{X[k]+X^*[N/2-k]}{2}\\X_o[k]&=\frac{X[k]-X^*[N/2-k]}{2}W_N^k,\end{aligned}$$

for $k=0,...,N/2-1$. Then, we construct $Z[k]$:

$$Z[k]=X_e[k]+iX_o[k], k=0...,N/2-1.$$

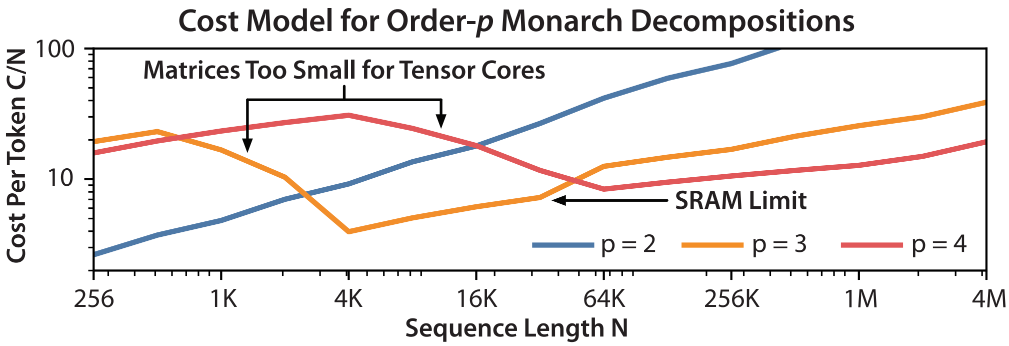

Figure 3: Compute costs of different order-p Monarch decompositions as sequence length increases on A100. Tradeoff points correspond to when the matrices in the Monarch decomposition reach the size of tensor cores on A100 and when the sequence becomes too long for SRAM.

We use the inverse DFT to recover $z(n)$, and then recover $x(n)$ from the real and imaginary parts of $z(n)$:

$$x(2n) = \text{Re}(z_n)$$
$$x(2n+1) = \text{Im}(z_n),$$

for $n = 0,...,N/2-1$.

To implement these in our kernels, we perform the bookkeeping after reading the inputs or before writing the output, and then use the FFT/iFFT implementations as detailed in Algorithm 1 and others.

C.2 Cost Model of order-p Monarch Decomposition

We present a formal cost model for an order-p Monarch decomposition of the convolution based on sequence length. The cost model accounts for both the cost of compute and I/O, similar to a roofline analysis [54]. Let B and H be the batch size and model hidden dimension, respectively, and assume that we compute the convolution in half precision. Let N be the sequence length, and let $N = \Pi_{i=1}^{p} N_i$ be the product of p factors. For simplicity, we will assume that N is a power of 2. Let μ be the size of the matrix-matrix multiply unit on the GPU (e.g., 16 for A100 [84] and H100 [85]). Let τ_G and τ_M be the empirically-achievable FLOPs on the GPU for general-purpose arithmetic, and matrix-matrix multiply arithmetic, respectively. For convenience, define $\gamma(N_i)$ as a helper function that returns τ_G if $N_i < \mu$, and τ_M if $N_i \geq \mu$. Finally, let σ_H and σ_S be empirically-achievable bandwidth for HBM and SRAM, respectively. Sample values for these constants are given in Appendix E.

Now, we can present the cost of an FFT convolution with an order-p Monarch decomposition. Let $\omega(i)$ be a helper function that returns the bandwidth of the memory where the intermediate results of decomposition step i is stored. The overall cost of the convolution using an order-p Monarch decomposition is given by the following:

$$C = BH \sum_{i=1}^{p} \frac{16NN_i}{\gamma(N_i)} + \frac{4N}{\omega(i)} \tag{3}$$

Figure 3 graphs Equation 3 for different order-p decompositions on different sequence lengths for A100, for $p \in \{2,3,4\}$. For cases where $N_1 = \cdots = N_p$, the total FLOP cost of an order-p decomposition grows with $O(N^{(p+1)/p})$. However, for shorter sequences, higher-order decompositions are actually *more expensive*, since they decompose to matrices that are smaller than the matrix-matrix multiply unit (corresponding to the early bumps). Note also the bump in cost for $p = 3$ between 32K and 64K, which is a result of running out of SRAM but which is mediated by an extra decomposition for $p = 4$.

C.3 Architectural Extensions: Sparsity in Convolutions

We present 2 architectural extensions to FlashFFTConv: partial convolutions and frequency-sparse convolutions. These can be thought of as convolutional analogues to sparse attention and present opportunities for further optimization.

Partial Convolutions In partial convolutions, we zero out later portions of the convolution kernel, analogous to local attention. This has two benefits. First, it reduces the memory footprint, since it requires fewer elements to be held in GPU memory at once. Second, it allows for natural extensions of a pretrained convolutional model to longer sequences (i.e., via a sliding window approach).

Frequency-Sparse Convolutions In frequency-sparse convolutions, we zero out portions of the convolution kernel in frequency space, i.e. zeroing out portions of k_f. This can be thought of as a variant of partial convolutions in frequency space. Here, the specific sparsity pattern can yield computational benefits. Zeroing out the right portions of the kernel can obviate the need to compute portions of the matrix-matrix multiplies in the Monarch decomposition.

We present some examples of sparsity patterns for the full 4-way decomposition case, since the algorithms generalize to lower-order decompositions.

Let $N = N_1^4$, and consider a kernel $k_f \in \mathbf{C}^N$. Consider the matrix multiply and looping operations that occur when computing the FFT portions of FLASHFFTCONV (u, k_f) (the iFFT portions are the same, in the opposite order):

1. In Algorithm 4, there is one FFT operation over the columns of u, reshaped to $N_1 \times N/N_1$, and a Twiddle correction..
2. Then, Algorithm 3 iterates over the rows of u for $\alpha := N_1$ steps.
3. Let u' be the row in a specific iteration. In Algorithm 3, there is an FFT over the columns of u', reshaped to $N_1 \times N_1^2$, and a Twiddle correction.
4. Then, the inner loop iterates over the rows of u' for $\beta := N_1$ steps.
5. In each loop, u' has one FFT operation with a twiddle factor correction. Let the matrix of this FFT operation be denoted $\mathbf{A}$.
6. Then there is a second FFT operation. Let the matrix of this FFT operation be denoted $\mathbf{B}$.

Now, reshape k_f to $N_1 \times N_1 \times N_1 \times N_1$. Let us consider how sparsity along the each of the four dimensions of k_f lets us skip operations in the above steps.

- Sparsity in the first dimension allows us to skip computation in $\mathbf{B}$, exactly in proportion to how much of the first dimension we eliminate. This can result in cost savings, as long as $\mathbf{B}$ can still be expressed using the tensor cores on-chip after skipping the computation. For example, if $\mathbf{B}$ is 32×32, then $N_1 = 32$, and it does not make sense to eliminate more than half of the first dimension.
- Sparsity in the second dimension works exactly the same way, except it allows us to skip computation in $\mathbf{A}$.
- Sparsity in the third dimension lets us reduce β. Each row of the third dimension that we remove lets us skip one iteration of the inner loop in step 4 above.
- Sparsity in the fourth dimension lets us reduce α. Each row of the fourth dimension that we remove lets us skip one iteration of the outer loop in step 2 above.

As an example, we reveal the sparsity dimensions that we applied in the experiment detailed in Table 10 in the main paper. Conceptually, we use the full 2-million length kernel k_f, and reshape it to $32 \times 32 \times 32 \times 64$. Let a, b, c, and d be variables describing how much of each dimension we set to zero. Specifically, we set $k_f[a:,:,:,:] = 0$, $k_f[:,b:,:,:] = 0$, $k_f[:,:,c:,:] = 0$, and $k_f[:,:,:,d:] = 0$ sequentially. The formula the sparsity fraction S given a,b,c,d in this case is given by:

$$S = 1 - (32-a)(32-b)(32-c)(64-d),$$

or more generally, 1 minus the product of the fraction of each dimension that is removed. Table 6 lists the configurations of the sparsity patterns and the sparsity fractions used for the experiment in Table 10.

C.4 Low-level CUDA details

To ensure high performance, we implement CUDA kernels for each specific sequence length, allowing us to cater to specific performance nuances that arise from the decomposition at that sequence length. In this section, we dive into some of the low-level implementation details for FLASHFFTCONV.

Table 5: Sparsity patterns for k_f and sparsity fraction for the frequency-sparse convolution experiment in Table 10.

Sparsity Pattern	S
a=0,b=0,c=0,d=0	0
a=16,b=0,c=0,d=0	50
a=16,b=16,c=0,d=0	75
a=16,b=16,c=4,d=4	79
a=16,b=16,c=8,d=8	84
a=16,b=16,c=16,d=16	91

Matrix Multiplication Using CUDA Tensor cores CUDA Tensor cores can perform the multiplication of two $m \times k$ and $k \times n$ matrices for bfloat16 or float16 elements, using around the same number of cycles as is required for the multiplication of two scalars. $m \times k \times n$ must be of one of the following: $16 \times 16 \times 16$, $32 \times 8 \times 16$, $8 \times 32 \times 16$. This informs our choice of radix for decomposition when performing the FFT and iFFT. In particular our implementation breaks down matrix-matrix multiplications into blocked matrix-matrix multiplications where $m \times k \times n = 16 \times 16 \times 16$. We note the following about matrix-matrix multiplication on tensor cores [86]:

- Tensor cores are utilized at the level of the warp and programmatic access of the tensor cores is via the Warp Level Matrix Multiply Accumulate (WMMA) API.
- Tensor core operands are held in register fragments ($wmma :: matrix\_a$, and $wmma :: matrix\_b$) and results are written to a register fragment ($wmma :: accumulator$).
- The operand fragments can hold data in row-major or column-major format and data in the $wmma :: accumulator$ fragment can be written to memory in row-major or column-major format.
- The specific mapping of items in a fragment to threads in warp is unspecified, however, the mapping of items to threads in the $wmma :: accumulator$ fragment exactly matches that for the $wmma :: matrix\_a$ fragment read row-major, allowing us to directly copy the results of a matrix-matrix multiplication and use as the operand for another matrix-matrix multiply.

To perform a matrix-matrix multiplication $C = A \times B$ using the tensor cores, a warp loads the contents of A and B into registers (WMMA fragments in CUDA parlance), performs the matrix-matrix multiplication, and writes the results which are stored in an accumulator fragment back to memory.

Register Reuse A key part of ensuring high performance is minimizing I/O across different levels of the memory hierarchy: from HBM to SRAM and from SRAM to registers. To ensure this, we move the output from the $accumulator$ fragment directly into $matrix\_a$ fragment for use in subsequent matrix multiplications, avoiding an extra trip to SRAM. However, this is only possible if the output from the previous matrix-matrix multiply does not need to be transposed before using it as an operand for the next one. When this is not the case, we need to make a trip to SRAM and back. In Algorithm 2 we detail I/O from SRAM to registers.

Locality and Tiling The algorithm is trivially parallelizable across B and H, allowing us to tile in both dimensions at the threadblock level. In Algorithm 3 , all loops from $i \leftarrow 1$ to N_1 are warp-tiled.

Miscellaneous optimizations In addition to the above optimizations, we also perform some other optimizations that provide marginal speedup. These include: utilizing vector intrinsics/types for performing memory reads/writes and arithmetic for 16-bit floating point (fp16) and brain float point (bf16), allowing non-tensor core operations on these types to be performed at around twice the normal speed. Furthermore, we double buffer I/O movements across all levels of the memory hierarchy, reducing warp stalls. We also aggressively tune our kernel hyperparameters such as block and tile dimensions, and loop unrolling factors for the best performance on the specific underlying hardware.

C.5 Generalization to 3-way and 4-way Monarch Decompositions

We provide algorithm listings for 3-way and 4-way Monarch Decompositions.

Algorithm 2 Detailed Annotation of FLASHFFTCONV core algorithm showing I/O from SRAM to register fragments, with two-way Monarch decomposition. We assume $N = N_1^2$ for simplicity here.

Input: Input $u \in \mathbb{R}^{B \times H \times N}$, convolution kernel $k_f \in \mathbb{C}^{H \times N}$, FFT matrices $\mathbf{F} \in \mathbb{C}^{N_1 \times N_1}$, $\mathbf{F^{-1}} \in \mathbb{C}^{N_1 \times N_1}$, Twiddle factors $t \in \mathbb{C}^N$, $t_{inv} \in \mathbb{C}^N$, B tile size B_{tile}, H tile size H_{tile}.
Output: Output $y \in \mathbb{R}^{B \times H \times N}$.

```
for SMs in parallel across B/B_tile × H/H_tile do
    Load F, F^{-1}, t, t_inv from HBM.
    for h ← 1 to H_tile do
        Load K_f ← k_f[h] from HBM, reshaped to N_1 × N_1.
        for b ← 1 to B_tile do
            Load X ← u[b,h] from HBM, reshaped to N_1 × N_1.
            X ← F^⊤ X                ▷ F^⊤ (matrix_a), X (matrix_b) output to accumulator
            Load X from accumulator to matrix_a
            X ← X ∗ t                ▷ Elementwise multiply directly in matrix_a
            X ← XF                   ▷ X (matrix_a), F (matrix_b) output to accumulator
            Load X from accumulator to matrix_a
            X ← X ∗ K_f^⊤            ▷ Elementwise multiply with k_f directly in matrix_a
            X ← XF^{-1}              ▷ X (matrix_a), F^{-1} (matrix_b) output to accumulator
            Write X from accumulator fragment to SRAM
            Load X^⊤ from SRAM to matrix_a fragment
            X ← X^⊤ ∗ t_inv          ▷ Elementwise multiply with t_inv directly in matrix_a
            Y ← XF^{-1}              ▷ X (matrix_a), F^{-1} (matrix_b) output to accumulator
            Write Y^⊤ to HBM.
```

3-Way Decomposition Algorithm 3 shows the algorithm for a 3-way Monarch decomposition. It involves one extra matrix multiply operation on either side of the FFT and iFFT, and proceeds over the algorithm in Algorithm 1 in an inner loop.

4-way Decomposition For the 4-way decomposition, we assume that we need to write intermediate outputs to HBM. Here, we treat the 3-way decomposition as a sub-routine, and assume it has a fused kernel (i.e., Algorithm 3). We compute one matrix multiply for the FFT and one for the iFFT, and then call the kernel for the 3-way decomposition over the rows of the output. The algorithm is listed in Algorithm 4.

C.6 Frequency-Sparse Patterns

We describe frequency-sparse patterns and the matmul savings in more detail here. We use the full 4-way decomposition case, since the algorithms generalize to lower-order decompositions.

Let $N = N_1^4$, and consider a kernel $k_f \in \mathbf{C}^N$. Consider the matrix multiply and looping operations that occur when computing the FFT portions of FLASHFFTCONV (u, k_f) (the iFFT portions are the same, in the opposite order):

1. In Algorithm 4, there is one FFT operation over the columns of u, reshaped to $N_1 \times N/N_1$, and a Twiddle correction..
2. Then, Algorithm 3 iterates over the rows of u for $\alpha := N_1$ steps.
3. Let u' be the row in a specific iteration. In Algorithm 3, there is an FFT over the columns of u', reshaped to $N_1 \times N_1^2$, and a Twiddle correction.
4. Then, the inner loop iterates over the rows of u' for $\beta := N_1$ steps.
5. In each loop, u' has one FFT operation with a twiddle factor correction. Let the matrix of this FFT operation be denoted $\mathbf{A}$.
6. Then there is a second FFT operation. Let the matrix of this FFT operation be denoted $\mathbf{B}$.

Now, reshape k_f to $N_1 \times N_1 \times N_1 \times N_1$. Let us consider how sparsity along the each of the four dimensions of k_f lets us skip operations in the above steps.

- Sparsity in the first dimension allows us to skip computation in $\mathbf{B}$, exactly in proportion to how much of the first dimension we eliminate. This can result in cost savings, as long as $\mathbf{B}$ can

Algorithm 3 FLASHFFTCONV algorithm for 3-way decomposition. We assume $N = N_1^3$ for simplicity here.

Input: Input $u \in \mathbb{R}^{B \times H \times N}$, convolution kernel $k_f \in \mathbb{C}^{H \times N}$, FFT matrices $\mathbf{F} \in \mathbb{C}^{N_1 \times N_1}$, $\mathbf{F^{-1}} \in \mathbb{C}^{N_1 \times N_1}$, Twiddle factors $t_1 \in \mathbb{C}^{N_1^2}$, $t_{1,inv} \in \mathbb{C}^{N_1^2}$, $t_2 \in \mathbb{C}^N$, $t_{2,inv} \in \mathbb{C}^N$, B tile size B_{tile}, H tile size H_{tile}.
Output: Output $y \in \mathbb{R}^{B \times H \times N}$.

- **for** SMs in parallel across $B/B_{tile} \times H/H_{tile}$ **do**
  - Load $\mathbf{F}$, $\mathbf{F^{-1}}$, t, t_{inv} from HBM.
  - **for** $h \leftarrow 1$ to H_{tile} **do**
    - Load $\mathbf{K_f} \leftarrow k_f[h]$ from HBM, reshaped to $N_1^2 \times N_1$.
    - $\mathbf{K_f} \leftarrow K_f^T$. ▷ Transpose last two dimensions.
    - Reshape $\mathbf{K_f}$ to $N_1 \times N_1^2$.
    - **for** $b \leftarrow 1$ to B_{tile} **do**
      - Load $\mathbf{X} \leftarrow u[b,h]$ from HBM, reshaped to $N_1 \times N_1 \times N_1$.
      - **for** $i \leftarrow 1$ to N_1 **do**
        - $\mathbf{X'} \leftarrow \mathbf{F}\mathbf{X}[:, i*N_1 : (i+1)*N_1]$
        - $\mathbf{X}[:, \mathbf{i}*\mathbf{N_1} : (\mathbf{i}+\mathbf{1})*\mathbf{N_1}] \leftarrow \mathbf{X'}$ ▷ Transpose, matmul, transpose.
      - $\mathbf{X} \leftarrow \mathbf{X} * t_2$
      - **for** $i \leftarrow 1$ to N_1 **do** ▷ Loop over rows
        - $\mathbf{X'} \leftarrow \mathbf{F}\mathbf{X}[i]$
        - Reshape $\mathbf{X'}$ to $N_1 \times N_1$
        - $\mathbf{X'} \leftarrow ((\mathbf{F}^\top \mathbf{X'}) * t)\mathbf{F}$ ▷ FFT, decomposed into two steps
        - $\mathbf{X'} \leftarrow \mathbf{X'} * \mathbf{K_f}[i]^\top$ ▷ Elementwise multiply with k_f
        - $\mathbf{Y'} \leftarrow ((\mathbf{X'}\mathbf{F}^{-1})^\top * t_{inv})\mathbf{F}^{-1}$ ▷ Inverse FFT, decomposed into two steps
        - $\mathbf{Y'} \leftarrow \mathbf{Y'}^\top$
        - $\mathbf{Y}[i] \leftarrow \mathbf{Y'}$ ▷ Finish inner loop
      - $\mathbf{Y} \leftarrow \mathbf{Y} * t_{2,inv}$
      - **for** $i \leftarrow 1$ to N_1 **do**
        - $\mathbf{Y'} \leftarrow \mathbf{F}\mathbf{Y}[:, i*N_1 : (i+1)*N_1]$
        - $\mathbf{Y}[:, \mathbf{i}*\mathbf{N_1} : (\mathbf{i}+\mathbf{1})*\mathbf{N_1}] \leftarrow \mathbf{Y'}$ ▷ Transpose, matmul, transpose.
      - Write $\mathbf{Y}$ to HBM.

still be expressed using the tensor cores on-chip after skipping the computation. For example, if $\mathbf{B}$ is 32×32, then $N_1 = 32$, and it does not make sense to eliminate more than half of the first dimension.

- Sparsity in the second dimension works exactly the same way, except it allows us to skip computation in $\mathbf{A}$.
- Sparsity in the third dimension lets us reduce β. Each row of the third dimension that we remove lets us skip one iteration of the inner loop in step 4 above.
- Sparsity in the fourth dimension lets us reduce α. Each row of the fourth dimension that we remove lets us skip one iteration of the outer loop in step 2 above.

As an example, we reveal the sparsity dimensions that we applied in the experiment detailed in Table 10 in the main paper. Conceptually, we use the full 2-million length kernel k_f, and reshape it to $32 \times 32 \times 32 \times 64$. Let a, b, c, and d be variables describing how much of each dimension we set to zero. Specifically, we set $k_f[a:,:,:,:] = 0$, $k_f[:,b:,:,:] = 0$, $k_f[:,:,c:,:] = 0$, and $k_f[:,:,:,d:] = 0$ sequentially. The formula the sparsity fraction S given a,b,c,d in this case is given by:

$$S = 1 - (32 - a)(32 - b)(32 - c)(64 - d),$$

or more generally, 1 minus the product of the fraction of each dimension that is removed. Table 6 lists the configurations of the sparsity patterns and the sparsity fractions used for the experiment in Table 10.

C.7 Hardware Support

FLASHFFTCONV was developed on A100 GPUs, and tested on A100 and H100 GPUs. Older generations of GPU such as V100 are not supported, since the sizes of the tensor cores are different.

Algorithm 4 FLASHFFTCONV algorithm for 4-way decomposition. We assume $N = N_1^4$ for simplicity here.

Input: Input $u \in \mathbb{R}^{B \times H \times N}$, convolution kernel $k_f \in \mathbb{C}^{H \times N}$, FFT matrices $\mathbf{F} \in \mathbb{C}^{N_1 \times N_1}$, $\mathbf{F^{-1}} \in \mathbb{C}^{N_1 \times N_1}$, Twiddle factors $t \in \mathbb{C}^N$, $t_{inv} \in \mathbb{C}^N$, $t_2 \in \mathbb{C}^N$, $t_{2,inv} \in \mathbb{C}^N$.
Output: Output $y \in \mathbb{R}^{B \times H \times N}$.
  Reshape u to $B \times H \times N_1 \times (N/N_1)$.
  Reshape k_f to $H \times N_1 \times (N/N_1)$.
  $k_f \leftarrow k_f^\top$. ▷ Transpose last two dimensions.
  Reshape k_f to $HN_1 \times N/N_1$.
  $u \leftarrow \mathbf{F}u$ ▷ Computes the FFT over the columns of u.
  Reshape u to $B \times (HN_1) \times (N/N_1)$. ▷ Move N_1 into H dimension.
  Reshape k_f to $(HN_1) \times (N/N_1)$.
  Call FLASHFFTCONV (u, k_f). ▷ Call 3-way FLASHFFTCONV.
  Reshape u to $B \times H \times N_1 \times (N/N_1)$.
  $y \leftarrow \mathbf{F^{-1}}u$ ▷ Computes the iFFT over the columns of u.
  Return y.

Table 6: Sparsity patterns for k_f and sparsity fraction for the frequency-sparse convolution experiment in Table 10.

Sparsity Pattern	**S**
a=0,b=0,c=0,d=0	0
a=16,b=0,c=0,d=0	50
a=16,b=16,c=0,d=0	75
a=16,b=16,c=4,d=4	79
a=16,b=16,c=8,d=8	84
a=16,b=16,c=16,d=16	91

Table 7: Classification accuracy (↑) on Path-X and Path-512 from the long range arena benchmark [104]. FLASHFFTCONV allows for higher-resolution classification. ✗ indicates out of memory.

Task (seq. len.)	PyTorch	FLASHFFTCONV
Path-X (16K)	**96.9**	**96.9**
Path-512 (256K)	✗	**96.1**

We look forward to integrating more general libraries such as Cutlass [88] to support a wider range of GPUs, and developing support for non-GPU accelerators.

D Additional Results

D.1 Additional Quality Results

Longer Sequence Models We show how increased efficiency can lead to higher quality via longer sequence lengths. We evaluate long convolution models on Path-X and Path-512, high-resolution imaging tasks from the long range arena (LRA) benchmark [104].[1] These tasks take an image (128×128 for Path-X and 512×512 for Path-512), flatten it out, and require a sequence model to classify whether two dots in the image are connected by a path.

Existing PyTorch implementations of convolutional sequence models (or even prior optimized implementations [43]) fail to achieve better-than-random (50%) accuracy on Path-512 due to out of memory errors and a lack of support for such long sequences. However, Table 7 shows that FLASHFFTCONV allows a convolutional sequence model to solve Path-512 for the first time simply by increasing the available sequence length and reducing the memory footprint of the model through fusion.

[1] We refer to Path-512 as a scaled-up version of Path-256.

Table 8: Quality and memory footprint of partial convolutions during training across sequence lengths.

Hyena-s-8K	**8K**	**4K**	**2K**	**1K**	**512**	**256**
PPL (↓)	13.8	13.8	13.8	13.9	14.0	14.2
Memory Footprint (↓)	32.5G	15.3G	11.8G	8.4G	6.1G	5.8G

Table 9: PPL (↓) from using partial convolutions to extend the sequence length of HyenaDNA to longer sequences. At 4M sequence length, the models are able to embed the longest human genes.

Base Filter Length	**1M**	**2M**	**4M**
HyenaDNA-450K	2.91	2.91	2.91
HyenaDNA-1M	2.91	2.91	2.90

Table 10: Applying frequency-sparsity to the filters of a pretrained HyenaDNA-1M model.

Sparsity Fraction	**0%**	**50%**	**75%**	**79%**	**84%**	**91%**
PPL (↓)	2.91	2.91	2.90	2.91	2.93	2.98
Convolution Speedup (↑)	1.0×	1.2×	1.3×	1.4×	1.5×	1.8×

D.2 Partial and Frequency-Sparse Convolutions

We evaluate the impact of partial convolutions on downstream quality and memory footprint and on how well they can extend the sequence length of existing models. We evaluate the impact of frequency-sparse convolutions on downstream quality, and we show that frequency-sparse convolutions can yield up to 1.4× additional speedup in the convolution without impacting quality.

Partial Convolutions Reduce Memory Footprint and Increase Sequence Length Partial convolutions reduce the memory footprint of models, in both language modeling and DNA modeling. A large proportion of the convolution filters can be pruned without impacting downstream quality. Table 8 shows that a Hyena-s-8K model can be pretrained with a much shorter convolution kernel—as short as 2K—without negatively impacting quality.

Partial convolutions yield another benefit: we can naturally extend the sequence length of existing pretrained models. We extend a pretrained HyenaDNA-1M model to 4M sequence length with promising PPL results (Table 9)—yielding the first model that can embed the longest human genes at single-nucleotide resolution (2.3M base pairs) (See Appendix D for a visualization of gene embeddings).

Frequency-Sparse Convolutions Increase Throughput Frequency-sparse convolutions can increase the speed of convolutions—and may also have positive effects on quality. Table 10 shows that we can set up to 79% of the entries of the kernel k_f to zero without losing quality. Sparsification in frequency space may even improve the quality of pretrained models slightly; the PPL of a pretrained HyenaDNA-1M model improves by 0.01 points after its kernels are 75% sparsified in frequency space—potentially as a result of removing high-frequency noise. Sparsification also yields up to 1.4× speedup in the convolution via skipping entire blocks of the matrix-matrix multiplies in the Monarch decomposition.

We use our 4M-sequence length HyenaDNA model to generate embeddings for various DNA segments following the procedure from [82]. The DNA classes include human genes corresponding to different biological function annotations from the Ensembl genome dataset known as biotypes [20]. The longest human gene, the dystrophin gene, is annotated.

D.3 Backward Pass Benchmark

Table 11 gives the time to compute the backward pass of a convolution with FlashFFTConv compared to PyTorch in milliseconds. We use batch size 64, and hidden dimension 768. When the input is too large to fit in memory, or the PyTorch implementation runs out of memory from storing intermediates for the backward pass, we split the call up into multiple calls and aggregate the time.

D.4 Reference Larger Models

Table 12 gives performance numbers for larger models trained for the same number of tokens and steps as the reference PyTorch models in Table 1 in the main paper.

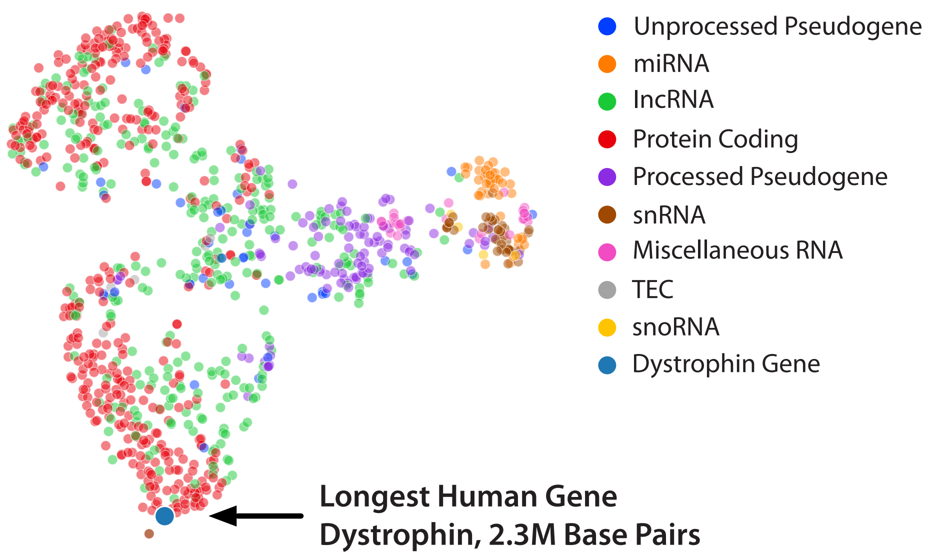

Figure 4: t-SNE visualization of various genes and DNA segments using our new HyenaDNA-4M. The longest human gene, Dystrophin, is annotated.

Table 11: Top: Time (↓) to compute the backward pass of a convolution with FLASHFFTCONV in milliseconds on A100. Bottom: Ablations removing specific optimizations. Batch size 64, hidden dimension 768. p indicates the order of the Monarch decomposition.

	$p=2$		$p=3$				$p=4$		
Sequence Length	**256**	**1K**	**4K**	**8K**	**16K**	**32K**	**1M**	**2M**	**4M**
PyTorch	1.19	2.92	17.4	34.8	71.2	161.3	5,999.4	12,065.3	25,113.6
FLASHFFTCONV	**0.18**	**0.67**	**4.1**	**8.9**	**32.0**	**86.8**	**3,429.8**	**7,181.7**	**14,499.8**

Table 12: Reference quality numbers for models when trained for the same number of steps and training data.

Model (Metric)	
M2-BERT-base-110M (GLUE Score ↑)	77.6
M2-BERT-large-260M (GLUE Score ↑)	81.0
Hyena-s-155M (PPL ↓)	13.4
Hyena-m-355M (PPL ↓)	11.1

The GPT-style PyTorch models are trained for 5B tokens, with batch size 512K tokens. The BERT-style PyTorch models are trained for 16000 steps, with batch size 64K tokens. In contrast, the FLASHFFTCONV models, with higher training throughput, are trained for 15B tokens and 70000 steps in the same compute budget, respectively.

E Experiment Details

E.1 Compute

All experiments were conducted on a box with 8xA100-40GB GPUs, except for long sequence HyenaDNA-1M to 4M experiments, which were conducted on a box with 8xA100-80GB GPUs.

E.2 Fixed Compute Budget Experiment

For the experiment in Table 1, we train an M2-BERT-base model from scratch, and a Hyena-s-155M model from scratch.

We train the M2-BERT-base model using masked language modeling of 30% on the C4 dataset, and fine-tune it on GLUE using the protocol from [42]. The FLASHFFTCONV model has higher training throughput, so it trains for more tokens; we train the FLASHFFTCONV model for 70,000 steps with a batch size of 64K tokens. The PyTorch model, with lower training throughput, only trains for 16,000 steps, with the same batch size. The M2-BERT-base model we use is parameter-matched with a Transformer BERT-base. It has 12 hidden layers, with a model dimension of 960, and an expansion factor of four. It also uses a block-diagonal MLP with four blocks. The M2 Hyena filter has embedding dimension 5, filter order 128, and initial sine activation factor of 10. We train with learning rate 8e-4, weight decay 1e-5, and 6% warmup with a cosine decay.

We train the Hyena-s-155M model using a causal language modeling objective on the Pile. We train the FLASHFFTCONV model for 15M tokens, and the PyTorch model for 5M tokens. The Hyena-s-155M model matches the configuration from [94] and has 18 layers, with a hidden dimension of 864, and an expansion factor of 4. The Hyena filter has embedding dimension 33, filter order 64, and initial sine activation factor of 14. We train with learning rate 6e-4, with 1% warmup time and a cosine decay.

E.3 Path-X and Path-512 Experiments

For the experiment in Table 7, we use simple convolutional language models, as in [44].

For Path-X, we use the same model and hyperparameters as the convolutional model from [44]. We use a convolutional model with 6 layers, prenorm batch norm, and hidden dimension of 256. For the convolution filter parameters, we use kernel dropout 0.3, kernel learning rate 0.0005, λ factor 0.001, and two channels on the filter. We use an overall learning rate of 0.0005 and weight decay 0.05. We train for 500000 steps, with 10000 steps of warmup with a cosine decay, and global batch size 16.

For Path-512, we scale up the resolution of Path-256. We train for 200000 steps, with 10000 steps warmup, learning rate 0.0005, and weight decay 0.05. For the model, we train with 4 layers, and hidden dimension 256. We use kernel dropout 0.1, kernel learning rate 0.0005, λ factor 0.001, and two channels on the filter. We keep the filter length to be 65536.

E.4 Convolution Benchmarks

For the experiments in Table 2, we time the forward pass of a convolution with batch size 64, hidden dimension 768, and varying sequence length. If we run out of memory for a sequence length, we split the batch and hidden dimension and call the forward pass multiple times. We time each call 30 times and take the average of the runs. We use the same protocol for the backward pass in Table 11.

E.5 End-to-End Modeling Details

For the experiments in Table 3, we run forward pass of each model, and use it to compute throughput. Batch sizes vary by model, and we check throughput calculations with a few batch sizes to make sure the result is consistent. For the M2-BERT-base model, we use the pretrained 110M checkpoint from [44]. For the Hyena-s-4K model, we use an identical model to the one in Table 1, but with a filter length of 4K. For the long convs Path-X model, we use the same model as in Table 7. For the SaShiMi model, we use the standalone SaShiMi model from the official implementation [45], and we use 8 layers with hidden dimension 64, and 4 up pool and down pool layers. For the HyenaDNA model, we use the official 1M-sequence length checkpoint from [82]. For M2-BERT-base, Hyena-s-4K, and HyenaDNA, we additionally fuse element-wise gating into our kernel, and short depthwise convolutions. For M2-BERT-base, we also fuse in a long convolution that runs in parallel to the gated convolution.

E.6 Comparison to Transformers

For the comparison against Transformers in Table 4, we use the official implementations with the FlashAttention-v2 release [22]. We use a Hyena model, and match the number of layers, hidden dimension, and expansion factor to the 2.7B Transformer model. To compute the FLOP usage, we take the formula:

$$2 * \text{num tokens} * \text{num parameters}$$

for the parametric FLOPs. For the non-parameter FLOPs, we add the raw FLOP count from our cost model in Equation 3 (without the adjustment for speed of tensor core FLOPs).

Table 13: Measured Constants for Cost Model for A100-40GB.

Constant	A100-40GB
σ_H	1.35 TB/s
σ_S	9.5 TB/s
τ_M	234 TFLOPs
τ_G	17.6 TFLOPs

E.7 Partial Convolutions for Hyena

For the measurement of memory footprint reduction in Table 8, we use the same Hyena-s model as in Tables 1 and 3, except we cut the filter short. This lets us offload parts of the input, which reduces the memory footprint.

E.8 Extending HyenaDNA-1M

In Table 9, we use a sliding window approach to extend the HyenaDNA-1M and HyenaDNA-450K models to longer sequences. This mimics training a 4M-sequence HyenaDNA with a short filter.

E.9 Frequency-Sparse Convolutions

To evaluate frequency-sparse convolutions, we take the pretrained HyenaDNA-1M model, and sparsify k_f using the strategy described in Appendix C.6. We then run standard validation using the validation set from [82].

E.10 Empirical GPU Profiling

Table 13 gives empirically-measured GPU stats for an A100-40GB, which we used to generate Figure 3. The statistics are specialized to the Monarch decomposition workload. To measure the achievable tensor core FLOPs, we measured the utilization of real fp16 matrix multiply. To measure achievable general arithmetic FLOPs, we measured the utilization of continuously applying Twiddle factors. To measure the achievable HBM bandwidth, we measured the speed of `torch.clone` of a tensor. To measure the achievable SRAM bandwidth, we measured the slow down from writing intermediate results to SRAM between matrix multiply instructions.