Improving GPT-2 Throughput for Lossless Text Compression

Undergraduate Thesis

Alison Zhong
Advised by Dr. Yu Su

The Ohio State University
Department of Computer Science and Engineering
Spring 2024

Thesis Committee:
Yu Su, CSE (advisor)
Hanqi Guo, CSE

© Copyright by
Alison Zhong
2024

ABSTRACT

Compression helps with handling the enormous amount (in the hundreds of millions of terabytes) of data generated daily. We can compress data with redundancies at higher rates with better models of data. Knowing large language models' (LLM) impressive performance at modeling text data, they seem well suited for lossless text compression. We implement a lossless text compressor that uses arithmetic coding with GPT-2. A naive GPT-2-based compressor is slow—it compresses 200-500 bytes per second with a GPU compared to almost two million bytes per second by 7-Zip (a general-purpose compressor). Though such a compressor's outputs are $\approx 30\%$ smaller than 7-Zip's, the poor compression speed limits its practicality. Hence, we investigate how various LLM optimizations impact our compressor's performance and speed. We achieve a $1.5\times$ speedup without significant performance degradation. We achieve further speedup (over $2\times$) if we accept sacrificing performance. Additionally, we find that increasing model size improves compression performance more than increasing context size and that distilled models compress better than pruned models.

ACKNOWLEDGEMENTS

Thank you to Dr. Yu Su for giving me the opportunity to work on this interesting and challenging project and for your unwavering support. Thank you to Dr. Hanqi Guo for agreeing to serve on my honors research distinction defense committee. Thank you to Samuel Stevens for being my research mentor, guiding me through this journey, and providing insightful suggestions and feedback.

Table of Contents

List of Figures

List of Algorithms

CHAPTER 1

INTRODUCTION

Compression addresses two problems: storing and transferring large amounts of data. Data is generated every moment, and compressing that data reduces the space needed to preserve all that collected information. Since the transfer rate is bounded by bytes per second, reducing the amount of data transferred leads to faster transfers.

Compression reduces the amount of bits needed to represent information by eliminating redundancy. Data compression can be lossless or lossy. Lossy methods sacrifice additional information that is necessary to recover data perfectly [32]. Lossless compression is standard with text data, as changing one character can change the data's interpretation.

No lossless data compression methods can efficiently compress all input data as only 2^N values can be stored in N bits. The four main types of compression methods are run-length encoding, statistical methods, dictionary-based methods, and transforms [32].

Data is composed of strings of symbols. Statistical methods leverage statistical models to encode symbols efficiently. Huffman coding constructs a code for symbols based on their probabilities. More likely symbols receive shorter codes, minimizing redundancy in the code [19]. Arithmetic coding represents data as an interval: each symbol narrows the interval, and more likely symbols narrow the interval less. The output from arithmetic coding is a number in the range of the final interval [22].

An algorithm's compression rate is calculated as the input data size divided by the output data size. Bits-per-character (BPC) measures the average number of bits representing a character from the original data. Higher compression rates correspond to lower BPC.

Shannon defines entropy as the negative expectation of log of probability for symbols in the data ($\Sigma_i - p_i * \log(p_i)$) and shows that a data source's entropy lower bounds its average coding length [35]. Huffman and arithmetic coding approach that lower bound. Arithmetic coding is more optimal as it represents symbols with a fractional number of bits [22].

Large language models (LLMs) based on the transformer architecture [38] have achieved state-of-the-art results on many natural language tasks [3, 36, 20]. LLMs take tokens as input, convert the token sequence into embeddings, and apply multi-head attention and a feed-forward network with normalization in layers. Tokenizers convert inputs into tokens; Byte Pair Encoding (BPE) tokenizers are trained by greedily merging possible byte pairs [34]. An embedding layer converts token sequences into positional embeddings. The attention mechanism connects all input and output positions, allowing transformer models to learn global dependencies [38].

Autoregressive language models are trained to estimate the probability distribution of a token sequence [44]. Generative Pre-trained Transformer 2 (GPT-2) is an autoregressive model released by OpenAI in 2019 that achieved state-of-the-art results on tasks without being trained on such tasks [30].

Arithmetic coding with a probability model that predicts the next tokens with higher probabilities should achieve a lower BPC. Thus, an autoregressive LLM like GPT-2 should compress text at a higher compression rate than general-purpose compressors.

A typical trade-off with compression algorithms is compression rate and compression speed: what percent increase in compression rate is worth a magnitude increase in compression time [32]. In practice, lower compression rate methods like 7-Zip and gzip are used since speed is desirable. Modern, transformer-based LLMs are slow due to having millions to billions of parameters and the attention's quadratic complexity [14]. We investigate accelerating transformer throughput while retaining good probability modeling, which translates to retaining compression performance. Accelerating transformer throughput allows more people to achieve higher compression rates.

CHAPTER 2

RELATED WORKS

Much research has been done on using neural networks to compress data [21, 1, 43]. DZip [15] was a neural network-based compressor that achieved better compression rates on datasets than domain-specific neural network compressors. The authors of DZip demonstrate the compression ability of neural networks and warn that the "practicality of DZip is currently limited due to the required encoding/decoding time" [15].

Recently, explorations of LLM-based compressors have been conducted. The authors of LLMZip show lower upper bounds on the entropy of the English language with LLaMa-7B [37]. They use LLaMa-7B [36] to calculate asymptotic upper bounds and use arithmetic coding to demonstrate compression rates close to those bounds. There has also been work exploring how compression and prediction relate. When discounting for model parameters, LLM Chinchilla 70B achieves better compression than domain-specific compressors on text, image, and audio data [11]. An adjacent work shows a correlation between compression and intrinsic dimension: OPT models with better compression rates have lower intrinsic dimension [5]. The authors also show that intrinsic dimension positively correlates to how easily an OPT model can be trained for tasks.

There's a range of methods to make transformer inference more efficient. These methods are classified as time-efficient, computation-efficient, memory-efficient, and storage-efficient based on which resource they reduce the requirements for while running transformer model inference [14]. From a recent survey, popular methods to accelerate transformer inference include knowledge distillation, model pruning, quantization, approximating multi-headed

attention, selecting optimal architecture for hardware, and hardware accelerators [7].

Knowledge distillation, model pruning, and quantization methods all seek to reduce the model size. There are several techniques for knowledge distillation, which include training on a larger model's output logits [46], training on a transfer set with an ensemble of specialized models [18], training on an intermediate model [26], and training with noisy output from a larger model [41]. Distilled models can have comparable performance to the original model. Model pruning also reduces the number of parameters. Model pruning can be unstructured, structured, or a mix. Unstructured pruning methods zero weights based on an estimation of their contribution, such as their magnitude or change in magnitude during training [17, 33]. Structured pruning methods remove weights in groups based on the model's architecture. Layers can be dropped after pre-training [31] with comparable performance to those that train for layer pruning [13]. Quantization reduces the number of bits used to represent model parameters. Standard quantization reduces parameters from 32-bit floating point to 8-bit integer [42]. There are more extreme quantization methods, such as reducing parameters to ternary values $\{-1, 0, 1\}$ [45]. LLM.int8() is a method to quantize a model's weights without significant performance drops on models with less than 175 billion parameters. Speedups with LLM.int8() are observed when a large enough batch size is fit into a smaller number of GPUs [12].

Automatic mixed precision (AMP) reduces the precision of the weights. With AMP, GPU operations that can be run with half-precision are run with 16 bits and other operations are run with full 32 bits ([28]). AMP has minimal impact on model performance ([14]).

Two attention approximation strategies based on the attention weight matrix being sparse are sparse attention and factorized attention [14]. Star-Transformer, Sparse Transformer, and Linformer are examples of sparse attention approximations as they achieve subquadratic complexity by choosing which connections to keep [16, 6, 2]. Factorized attention methods, like Linformer, factorize matrices to reduce their ranks and the complexity of the attention operation without noticeably impacting performance [39].

Many attention approximations fail to produce speedups as they fail to address memory

complexity. FlashAttention restructures the attention to reduce the number of reads and writes as I/O is a bottleneck for GPU operations, achieving $2-4\times$ speedup [10]. FlashAttention2 improves work sharing, increasing GPU block occupation and reducing shared memory access for $2\times$ more speedup [8]. Flash-Decoding makes similar observations and optimizes for decoding by parallelizing key value computations and reducing them to achieve $20-50\times$ speedup [9].

A similar bottleneck with attention is the GPU waiting for instructions from Python. A solution is `torch.compile`, which just-in-time compiles Pytorch models and functions into kernels [40]. `torch.compile` aims to reduce the time the GPU spends waiting for instructions and the number of reads and writes.

An unrelated method to accelerate inference is speculation. Speculative decoding, also known as speculative sampling, leverages a smaller model to generate guesses and validate them with the larger model [4, 23]. This guarantees at least one additional token will be generated with each run of the larger model.

Some optimizations may lead to worse performance depending on the hardware used. One solution is to dynamically select architecture and optimizations to balance performance and throughput given the hardware used [7]. More hardware-related optimizations include designing hardware for specialized operations, skipping redundant operations, pipeline operations, and optimizing loops [7].

CHAPTER 3

METHODOLOGY

We investigate how various optimizations impact bits-per-character (BPC), encoding time, and decoding time. Since the goal is lossless text compression, we extract the first 5 10^5-byte excerpts from text8 [24] to test our compressor on. The text8 dataset is composed of 10^8 bytes of text from Wikipedia and is used as a compression benchmark.

The size of the input data, size of the compressed data, time to encode the data, and time to decode the data are measured. We test how automatic mixed precision (AMP), FlashAttention, FlashAttention2, `torch.compile`, 8-bit quantization, and various layer dropping strategies perform individually. From those results, we select one subset of optimizations to combine and test on a larger excerpt (10^6-bytes) from text8.

BPC	compressed number of bits / input number of bytes
encoding speed	input number of bytes / encoding time
decoding speed	input number of bytes / decoding time
compression rate	compressed number of bytes / input number of bytes
compression rate ratio	compression rate / baseline compression rate
coding speed	(encoding + decoding speed) / 2
coding time	encoding + decoding time
speedup	coding time / baseline coding time

Table 3.1: Measurements

3.1 Compressor

We implement a baseline compressor that converts text into tokens, feeds those tokens into OpenAI's GPT-2 small model, and encodes the input with arithmetic coding based on the logits output from the model. We use an open-source implementation of arithmetic coding [27]. The following pseudocode explains how coding and decoding take place. One simplification from the pseudocode is that the models have an input token limit. The solution is to choose a window size and adjust the input window of tokens as needed. The pseudocode of arithmetic coding is also simplified.

Algorithm 1 Encoding

```
Read text from input file
Tokenize text
N = number of tokens
Encode N
Give the first token and cumulative uniform probability table to the encoder
while i ≤ N do
    Input the tokens before token i to GPT-2
    Transform the logits from GPT-2 into probabilities with softmax
    Construct a cumulative probability table with the probabilities
    Give the ith token and the cumulative probability table to the encoder
```

Algorithm 2 Decoding

Read bytes from the compressed input
Decode N
Give the cumulative uniform probability table to the decoder
while $i \leq N$ **do**
    Input the decoded tokens before token i to GPT-2
    Transform the logits from GPT-2 into probabilities with softmax
    Construct a cumulative probability table with the probabilities
    Give the cumulative probability table to the decoder to get token i
Convert tokens to text and output decompressed input

Algorithm 3 Arithmetic Encoder

Set high to half of max integer value
Set low to 0
while not finished **do**
    Take symbol and cumulative probability table
    Use the symbol to index into the cumulative probability table
    Update range's high and low with the symbol's high and low values
    Shift bits to the buffer and write any underflow bits until high and low have different top bit values
    Delete the second highest bit and increment underflow bit count until low's top two bits $\neq 01$ or high's top two bits $\neq 10$
Flush buffered bits

Algorithm 4 Arithmetic Decoder

Set high to half of max integer value
Set low to 0
while not finished **do**
    Take cumulative probability table
    Use cumulative probability table to convert coding range
    Calculate a value with the converted range
    Shift until find the highest symbol with low less than the value
    Update range's high and low with the symbol's high and low values
    Shift bits from the buffer until high and low have different top bit values
    Transform second highest bit with a bit from the buffer until low's top two bits $\neq 01$ or high's top two bits $\neq 10$
    Return symbol

3.2 Determinism

We follow PyTorch's guidelines on enforcing determinism [29]. This is necessary as when the model's output is nondeterministic, arithmetic coding may impact decode a token incorrectly. Once one token is decoded incorrectly, the probability model becomes incorrect, leading to subsequent tokens being decoded incorrectly.

3.3 Optimizations

We implement several optimizations: AMP [28], FlashAttention [10, 8], layer dropping [31], `torch.compile` [40], and 8-bit quantization [12]. For layer dropping, we test the dropping top, symmetric, old alternating, even alternating, and bottom layer strategies [31]. We drop layers using a ratio instead of a fixed number of layers. For a model with 12 layers like GPT-2, ratio 2 drops $12 * \frac{1}{2} = 6$ layers, ratio 3 drops $12 * \frac{1}{3} = 4$ layers, and ratio 6 drops

$12 * \frac{1}{6} = 2$ layers. This will hopefully lead to similar speedups across models with different numbers of layers.

We measure the BPC, the encoding time, and the decoding time for each optimization. We also check if the decompressed output matches the input. We select a combination of optimizations that are likely to preserve low BPC and increase throughput. We run an ablation study with DistilGPT2, GPT-2, and GPT-2 medium and compare the performance of the optimized compressor with the baseline on a larger excerpt.

3.4 Hardware

We run our experiments on RTX 2080 and RTX A6000 GPUs. At the time of experimentation, FlashAttention optimizations did not support running on RTX 2080.

3.5 Window Sizes

Generally, BPC should decrease as window size increases - having more context allows GPT-2 to model the data better. However, there is a bug with DistilGPT2 and GPT-2 around 1024 tokens (the maximum allowed number of tokens), so we use 1023 tokens as our maximum window size. We test our compressor with window sizes of 128, 256, 512, 768, and 1023.

CHAPTER 4

EXPERIMENTS

In preliminary experiments, we investigate automatic mixed precision (AMP), `torch.compile`, 8-bit quantization, 15 layer dropping strategies, FlashAttention, and FlashAttention2. From our results, we select AMP, `torch.compile`, symmetric layer dropping, and FlashAttention2 for further experimentation. We test AMP with `torch.compile`, AMP with `torch.compile` and symmetric layer dropping, and AMP with `torch.compile` and FlashAttention2 for our final experiments. The compressor with `torch.compile` decompresses output unequal to the input once in the preliminary, which could have been due to chance. The compressor with `torch.compile` fails multiple times in the final experiments. Our compressor does not fail otherwise. Numbers are available in the Appendix.

4.1 Preliminary Experiments

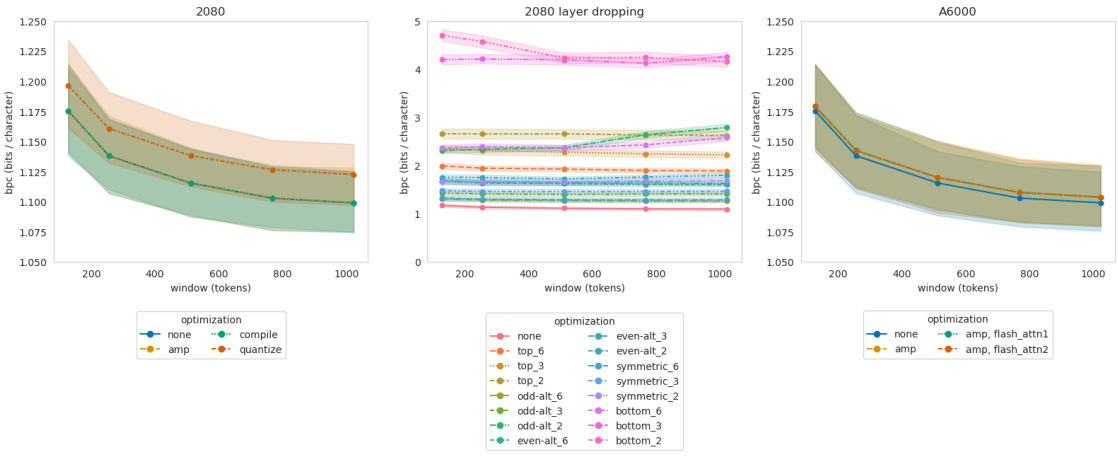

Figure 4.1: BPC vs window size

Bits-per-character (BPC) generally decreases as the model is given more context. There are exceptions with layer dropping, likely due to information lost with model pruning. Non-pruning optimizations result in negligible BPC increases, and pruning more layers results in more significant BPC increases.

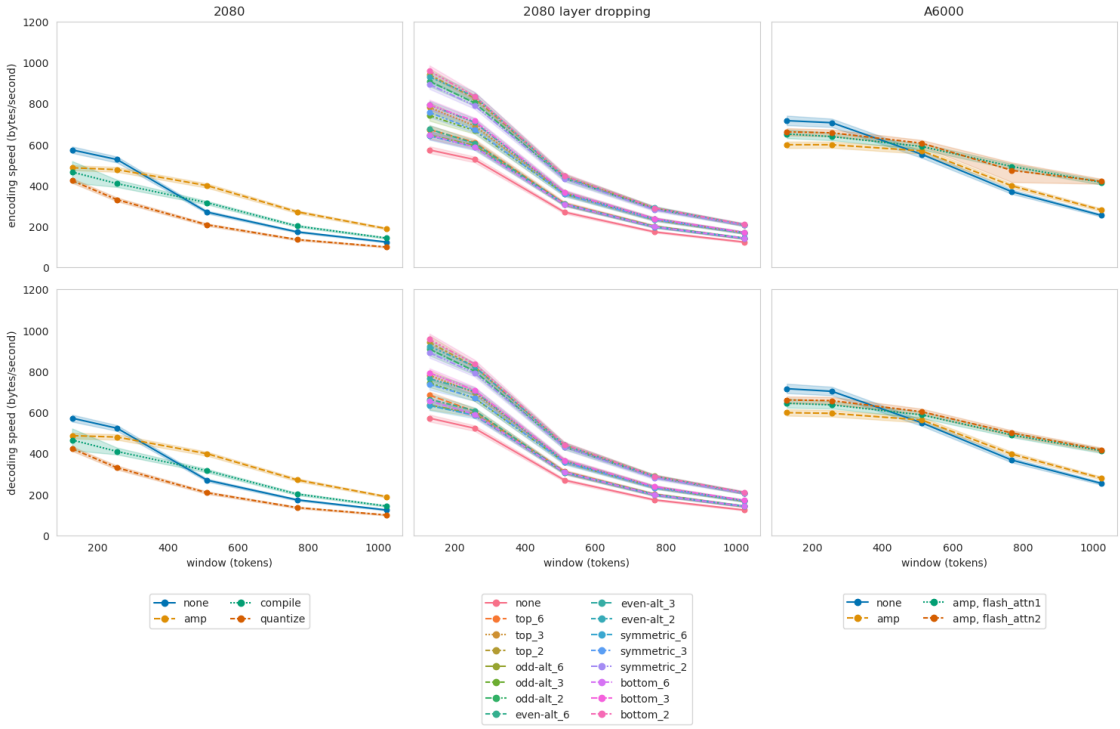

Figure 4.2: speed vs window size

As window size increases, encoding and decoding speeds decrease. Non-pruning optimizations are slower than the baseline with the smallest window size but become faster than the baseline with larger window sizes. 8-bit quantization is the exception, likely slower due to using single batch inference. Also, the encoding and decoding speed for pruning optimizations seem to be grouped by the ratio of layers dropped.

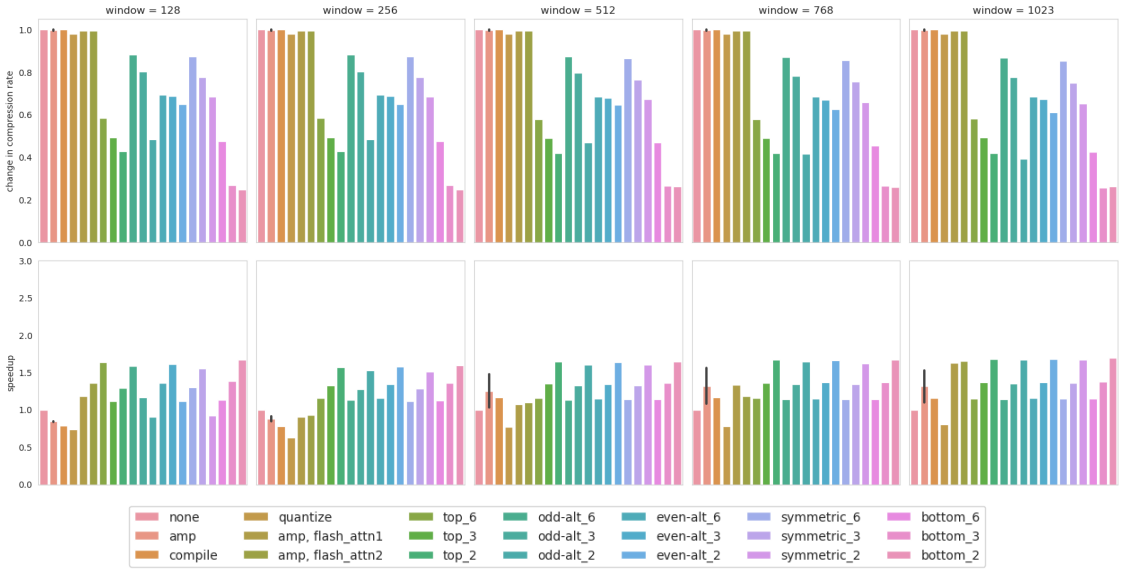

Figure 4.3: compression rate ratio and speedup

Change in compression rate ratio seems stable for optimizations over different window sizes. Speedup varies for AMP, `torch.compile`, 8-bit quantization, FlashAttention, and FlashAttention2.

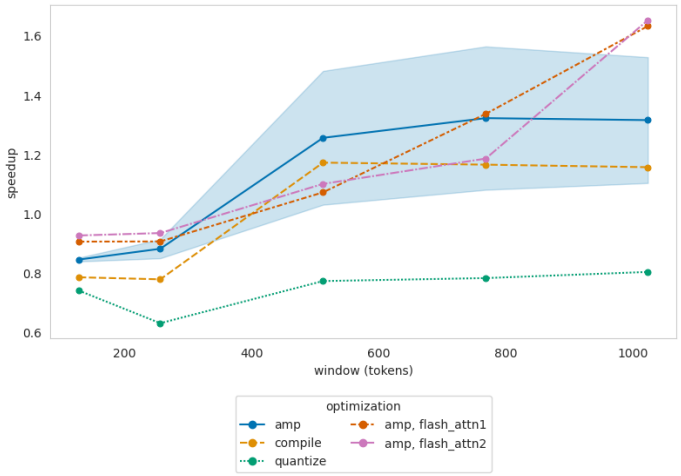

Figure 4.4: speedup vs window size

Generally, speedups increase with window size.

4.2 Final Experiments

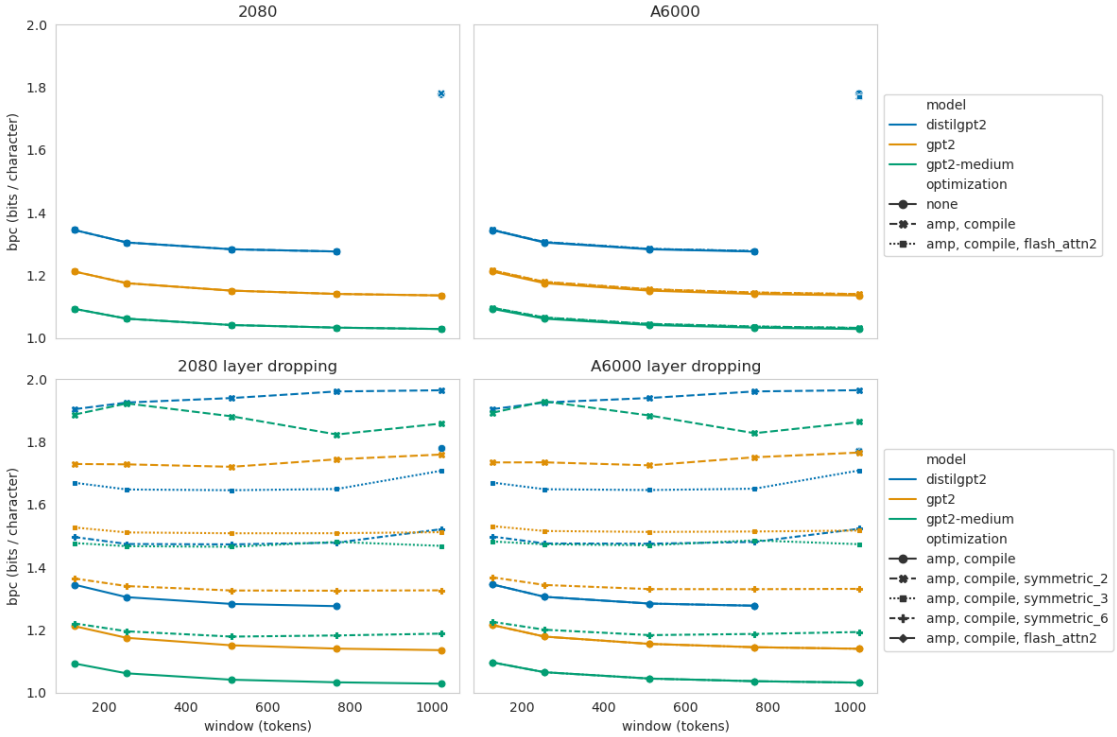

Figure 4.5: Final: BPC vs window size

The line between 768 tokens and 1023 tokens is removed since we expect BPC to continue decreasing until some point. As expected, larger models achieve lower BPC, though that no longer holds after pruning.

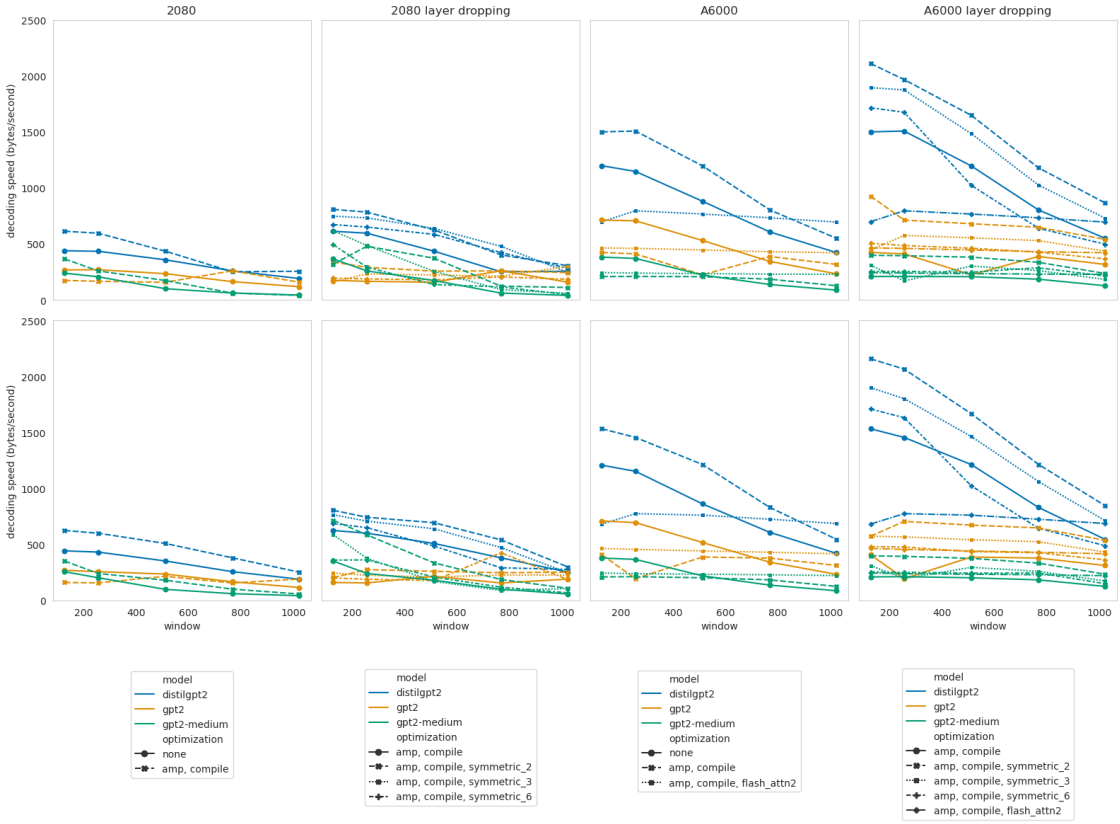

Figure 4.6: speed vs window size

Encoding and decoding speeds generally decrease as window size increases. The combined optimization of AMP, `torch.compile`, and FlashAttention2 is an exception as its speed is fixed. This might be due to `torch.compile` with FlashAttention2 needing to recompile more due to hitting cache limits.

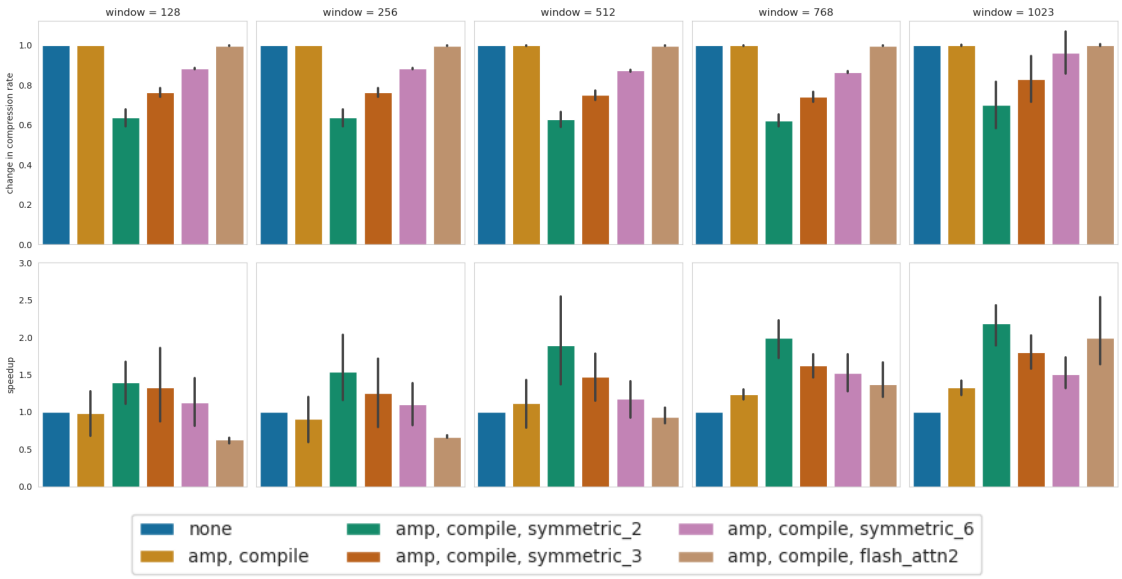

Figure 4.7: compression rate ratio and speedup

Changes in compression rate ratios vary for symmetric layer droppings over different models and window sizes. Speedup also varies.

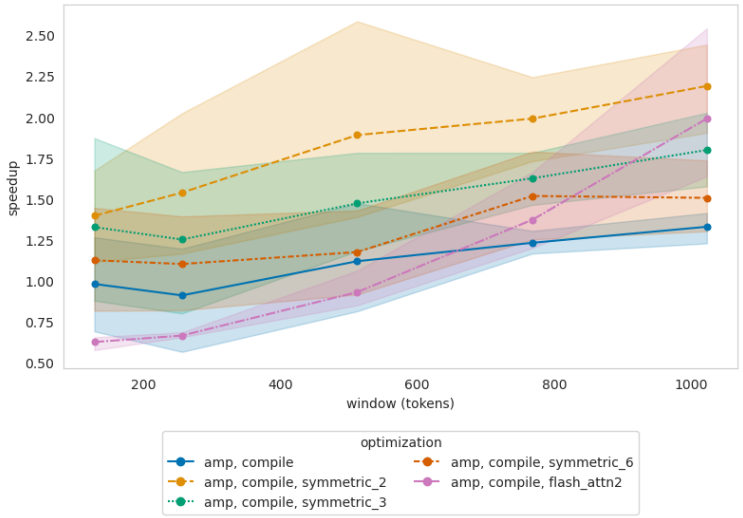

Figure 4.8: speedup vs window size

When measuring speedups across models and GPUs, the 95% confidence intervals overlap, making it difficult to conclude which optimization achieves higher speedups. Overall, as window size increases, speedups increase.

4.3 General Results

All calculations use GPT-2 with a window size of 128 tokens and no optimizations as the baseline.

Figure 4.9: Increasing context vs model size

Increasing model size leads to a greater increase in compression rate for a smaller decrease in speed.

model	**number of parameters**
distilGPT2	82M
GPT-2, symmetric drop $\frac{1}{6}$	103.3M
GPT-2	124M
GPT-2 medium, symmetric drop $\frac{1}{6}$	295.8M
GPT-2 medium	355M

Table 4.1: Model Size

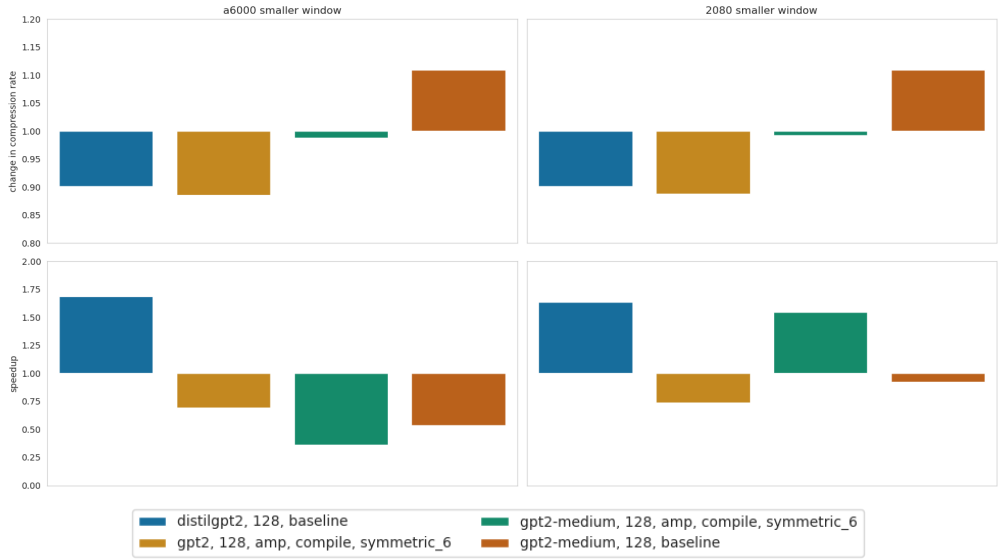

Figure 4.10: Smaller window: distillation vs pruning

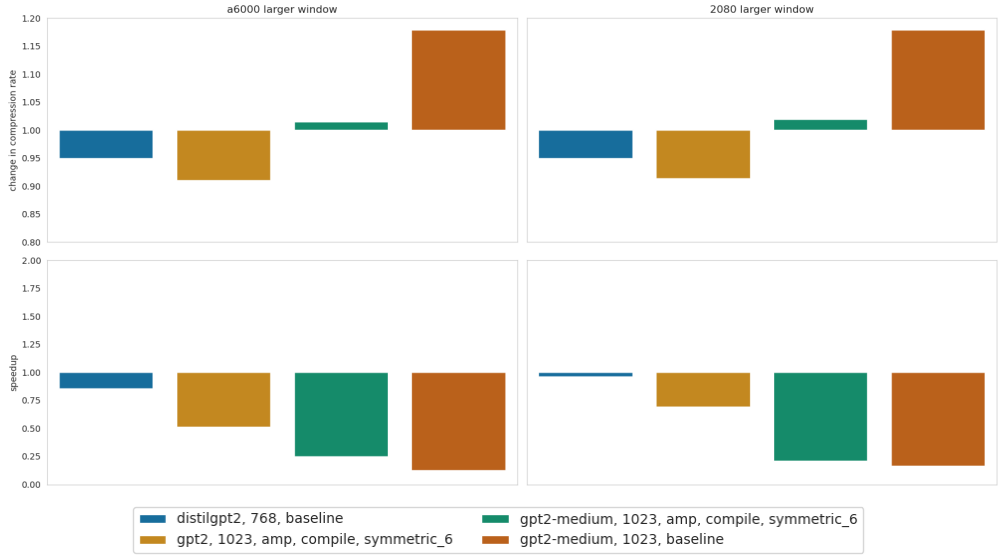

Figure 4.11: Larger window: distillation vs pruning

At small and large contexts, a distilled or smaller model does better than dropping $\frac{1}{6}$ of the layers of the larger model. In most cases, the smaller model is also faster than the pruned model.

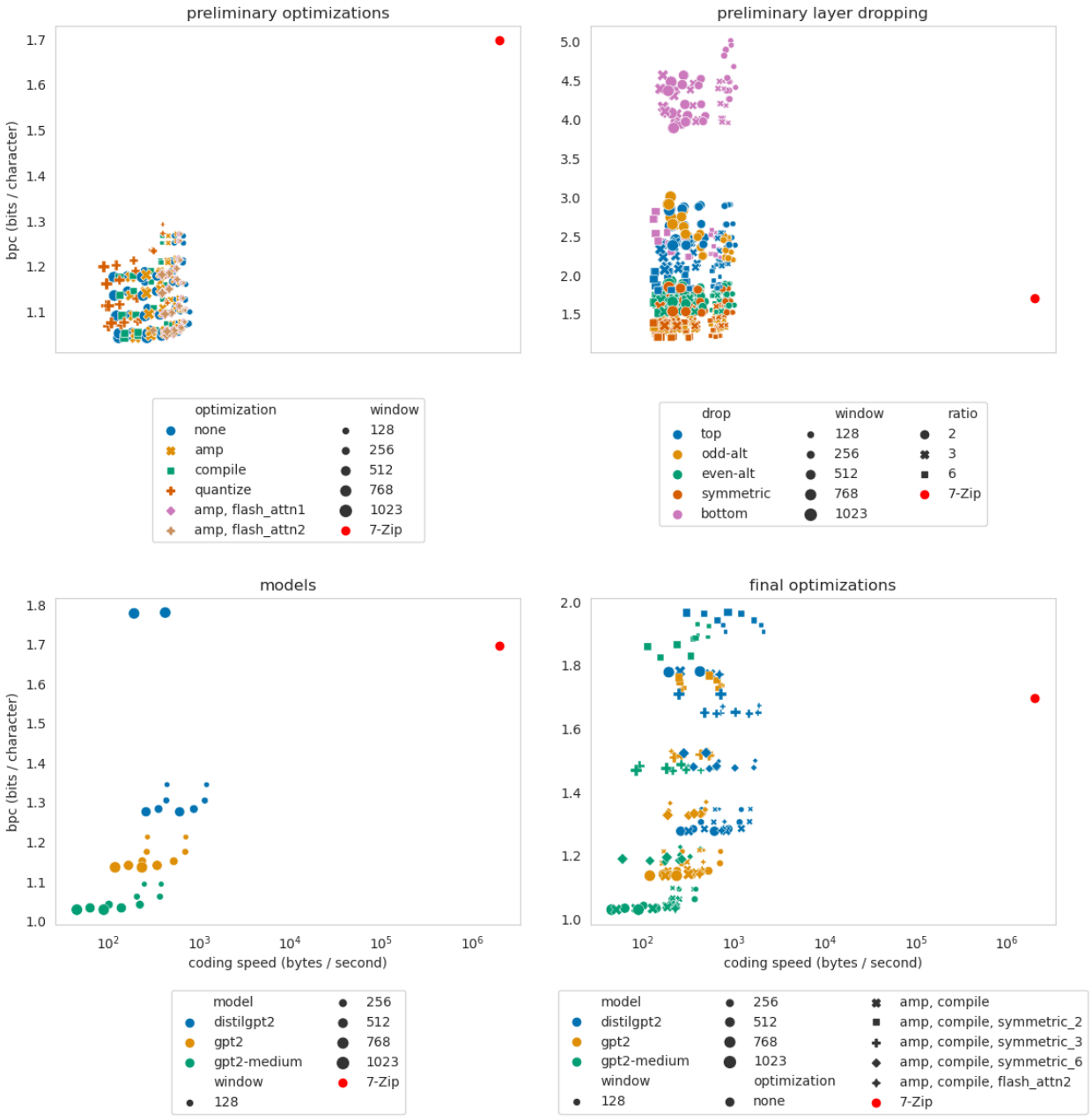

Figure 4.12: BPC and coding speed trade-off with 7-Zip

7-Zip has a BPC of 1.696 and a coding speed of 1988071.57 bytes/second on enwik8 [25]. While arithmetic coding with GPT-2 achieves a lower BPC, it is much slower. Even with optimizations, 7-Zip is at least $900 - 10,000\times$ faster.

CHAPTER 5

CONCLUSION

Increasing GPT-2's model size yields a greater decrease in BPC than increasing GPT-2's context window with a smaller speed decrease. Similarly, switching to a smaller or distilled model leads to less increase in BPC than pruning a sixth of a model's layers. Since increasing model size is more effective than increasing context, non-pruning optimizations are only sometimes useful, as most speedups are achieved at larger window sizes.

Future work includes exploring distillation, pruning, memory-based optimizations, and multi-batch compression. Comparing how different distillation methods impact compression and throughput may lead to improved performance and higher speedups. More sophisticated model pruning methods may yield smaller compression rate losses for similar speedups. Flash-Decoding and memory-aware, sparse attention approximations have the potential for significant speedups. For multi-batch compression, there would be a trade-off between speed and compression as each division of the input loses context at the chunk's start. Finally, it remains to test how optimizations impact autoregressive models other than GPT-2.

REFERENCES

[1] Fabrice Bellard. "Lossless Data Compression with Neural Networks". In: (2019).

[2] Iz Beltagy, Matthew E. Peters, and Arman Cohan. *Longformer: The Long-Document Transformer*. 2020. arXiv: 2004.05150 [cs.CL].

[3] Tom Brown et al. "Language Models are Few-Shot Learners". In: *Advances in Neural Information Processing Systems*. Ed. by H. Larochelle et al. Vol. 33. Curran Associates, Inc., 2020, pp. 1877–1901. URL: https://proceedings.neurips.cc/paper_files/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf.

[4] Charlie Chen et al. *Accelerating Large Language Model Decoding with Speculative Sampling*. 2023. arXiv: 2302.01318 [cs.CL].

[5] Emily Cheng, Corentin Kervadec, and Marco Baroni. "Bridging Information-Theoretic and Geometric Compression in Language Models". In: *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*. Ed. by Houda Bouamor, Juan Pino, and Kalika Bali. Singapore: Association for Computational Linguistics, Dec. 2023, pp. 12397–12420. DOI: 10.18653/v1/2023.emnlp-main.762. URL: https://aclanthology.org/2023.emnlp-main.762.

[6] Rewon Child et al. *Generating Long Sequences with Sparse Transformers*. 2019. arXiv: 1904.10509 [cs.LG].

[7] Krishna Teja Chitty-Venkata et al. "A survey of techniques for optimizing transformer inference". In: *J. Syst. Archit.* 144.C (Nov. 2023). ISSN: 1383-7621. DOI: 10.1016/j.sysarc.2023.102990. URL: https://doi.org/10.1016/j.sysarc.2023.102990.

[8] Tri Dao. "FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning". In: (2023).

[9] Tri Dao et al. "Flash-Decoding for long-context inference". In: (2023).

[10] Tri Dao et al. "FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness". In: *Advances in Neural Information Processing Systems*. 2022.

[11] Grégoire Delétang et al. *Language Modeling Is Compression*. 2024. arXiv: 2309.10668 [cs.LG].

[12] Tim Dettmers et al. *LLM.int8(): 8-bit Matrix Multiplication for Transformers at Scale*. 2022. arXiv: 2208.07339 [cs.LG].

[13] Angela Fan, Edouard Grave, and Armand Joulin. *Reducing Transformer Depth on Demand with Structured Dropout*. 2019. arXiv: 1909.11556 [cs.LG].

[14] Quentin Fournier, Gaétan Marceau Caron, and Daniel Aloise. "A Practical Survey on Faster and Lighter Transformers". In: 55.14s (July 2023). ISSN: 0360-0300. DOI: 10.1145/3586074. URL: https://doi.org/10.1145/3586074.

[15] Mohit Goyal et al. *DZip: improved general-purpose lossless compression based on novel neural network modeling*. 2020. arXiv: 1911.03572 [cs.LG].

[16] Qipeng Guo et al. *Star-Transformer*. 2022. arXiv: 1902.09113 [cs.CL].

[17] Song Han, Huizi Mao, and William J. Dally. *Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding*. 2016. arXiv: 1510.00149 [cs.CV].

[18] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. *Distilling the Knowledge in a Neural Network*. 2015. arXiv: 1503.02531 [stat.ML].

[19] David A. Huffman. "A Method for the Construction of Minimum-Redundancy Codes". In: *Proceedings of the IRE* 40.9 (1952), pp. 1098–1101. DOI: 10.1109/JRPROC.1952.273898.

[20] Albert Q. Jiang et al. *Mistral 7B*. 2023. arXiv: 2310.06825 [cs.CL].

[21] Byron Knoll. "CMIX". In: (2014). URL: https://www.byronknoll.com/cmix.html.

[22] G. G. Langdon. "An Introduction to Arithmetic Coding". In: *IBM Journal of Research and Development* 28.2 (1984), pp. 135–149. DOI: 10.1147/rd.282.0135.

[23] Yaniv Leviathan, Matan Kalman, and Yossi Matias. *Fast Inference from Transformers via Speculative Decoding*. 2023. arXiv: 2211.17192 [cs.LG].

[24] Matt Mahoney. "About the Test Data". In: (2011). URL: https://www.mattmahoney.net/dc/textdata.html.

[25] Matt Mahoney. "Large Text Compression Benchmark". In: (2023). URL: https://www.mattmahoney.net/dc/text.html.

[26] Seyed Iman Mirzadeh et al. "Improved Knowledge Distillation via Teacher Assistant". In: *Proceedings of the AAAI Conference on Artificial Intelligence* 34.04 (Apr. 2020), pp. 5191–5198. DOI: 10.1609/aaai.v34i04.5963. URL: https://ojs.aaai.org/index.php/AAAI/article/view/5963.

[27] nayuki. "Reference arithmetic coding". In: (2018). URL: https://www.nayuki.io/page/reference-arithmetic-coding.

[28] PyTorch. "Automatic Mixed Precision Package - torch.amp". In: (2023). URL: https://pytorch.org/docs/stable/amp.html.

[29] PyTorch. "Reproducibility". In: (2023). URL: https://pytorch.org/docs/stable/notes/randomness.html.

[30] Alec Radford et al. "Language Models are Unsupervised Multitask Learners". In: (2019).

[31] Hassan Sajjad et al. "On the effect of dropping layers of pre-trained transformer models". In: *Computer Speech & Language* 77 (2023), p. 101429. ISSN: 0885-2308. DOI: https://doi.org/10.1016/j.csl.2022.101429. URL: https://www.sciencedirect.com/science/article/pii/S0885230822000596.

[32] David Salomon. *Data Compression: The Complete Reference*. Berlin, Heidelberg: Springer-Verlag, 2006. ISBN: 1846286026.

[33] Victor Sanh, Thomas Wolf, and Alexander M. Rush. *Movement Pruning: Adaptive Sparsity by Fine-Tuning*. 2020. arXiv: 2005.07683 [cs.CL].

[34] Rico Sennrich, Barry Haddow, and Alexandra Birch. *Neural Machine Translation of Rare Words with Subword Units*. 2016. arXiv: 1508.07909 [cs.CL].

[35] C. E. Shannon. "A mathematical theory of communication". In: *The Bell System Technical Journal* 27.3 (1948), pp. 379–423. DOI: 10.1002/j.1538-7305.1948.tb01338.x.

[36] Hugo Touvron et al. *Llama 2: Open Foundation and Fine-Tuned Chat Models*. 2023. arXiv: 2307.09288 [cs.CL].

[37] Chandra Shekhara Kaushik Valmeekam et al. *LLMZip: Lossless Text Compression using Large Language Models*. 2023. arXiv: 2306.04050 [cs.IT].

[38] Ashish Vaswani et al. "Attention is All you Need". In: *Advances in Neural Information Processing Systems*. Ed. by I. Guyon et al. Vol. 30. Curran Associates, Inc., 2017. URL: https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.

[39] Sinong Wang et al. *Linformer: Self-Attention with Linear Complexity*. 2020. arXiv: 2006.04768 [cs.LG].

[40] William Wen. "Introduction to torch.compile". In: (2024). URL: https://pytorch.org/tutorials/intermediate/torch_compile_tutorial.html.

[41] Qizhe Xie et al. "Self-Training With Noisy Student Improves ImageNet Classification". In: *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 2020, pp. 10684–10695. DOI: 10.1109/CVPR42600.2020.01070.

[42] Canwen Xu and Julian McAuley. *A Survey on Model Compression and Acceleration for Pretrained Language Models*. 2022. arXiv: 2202.07105 [cs.CL].

[43] Yibo Yang, Stephan Mandt, and Lucas Theis. "An Introduction to Neural Data Compression". In: *Found. Trends. Comput. Graph. Vis.* 15.2 (Apr. 2023), pp. 113–200. ISSN: 1572-2740. DOI: 10.1561/0600000107. URL: https://doi.org/10.1561/0600000107.

[44] Zhilin Yang et al. *XLNet: Generalized Autoregressive Pretraining for Language Understanding*. 2020. arXiv: 1906.08237 [cs.CL].

[45] Wei Zhang et al. "TernaryBERT: Distillation-aware Ultra-low Bit BERT". In: *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Ed. by Bonnie Webber et al. Online: Association for Computational Linguistics, Nov. 2020, pp. 509–521. DOI: 10.18653/v1/2020.emnlp-main.37. URL: https://aclanthology.org/2020.emnlp-main.37.

[46] Haoran Zhao et al. "Highlight Every Step: Knowledge Distillation via Collaborative Teaching". In: *IEEE Transactions on Cybernetics* 52.4 (2022), pp. 2070–2081. DOI: 10.1109/TCYB.2020.3007506.

APPENDIX

Failures

Trials in which the decompressed output is different from the input are counted as failures. There are 75 trials per preliminary experiment and 5 trials per final experiment.

Optimization	Model	Failures
`torch.compile`	GPT-2	1

Table 5.1: Preliminary Experiment Failures on 2080

Optimization	Model	Failures
AMP, `torch.compile`	DistilGPT2	1
AMP, `torch.compile`	GPT-2	**3**
AMP, `torch.compile`	GPT-2 Medium	2
AMP, `torch.compile`, symmetric 2	GPT-2	1
AMP, `torch.compile`, symmetric 2	GPT-2 Medium	**3**
AMP, `torch.compile`, symmetric 3	DistilGPT2	1
AMP, `torch.compile`, symmetric 3	GPT-2	2
AMP, `torch.compile`, symmetric 3	GPT-2 Medium	**3**
AMP, `torch.compile`, symmetric 6	DistilGPT2	1
AMP, `torch.compile`, symmetric 6	GPT-2	2
AMP, `torch.compile`, symmetric 6	GPT-2 Medium	**3**

Table 5.2: Final Experiment Failures on 2080

Optimization	Model	Failures
AMP, `torch.compile`	DistilGPT2	1
AMP, `torch.compile`	GPT-2	1
AMP, `torch.compile`, symmetric 2	GPT-2	1
AMP, `torch.compile`, symmetric 3	DistilGPT2	1
AMP, `torch.compile`, symmetric 3	GPT-2	1
AMP, `torch.compile`, symmetric 6	DistilGPT2	1
AMP, `torch.compile`, symmetric 6	GPT-2	1
AMP, `torch.compile`, symmetric 6	GPT-2 Medium	**2**
AMP, `torch.compile`, FlashAttention2	DistilGPT2	1

Table 5.3: Final Experiment Failures on A6000

BPC & Coding Speed

Optimization	Model	BPC 128	BPC 256	BPC 512	BPC 768	BPC 1023
baseline	GPT-2	**1.18**	**1.14**	**1.12**	**1.10**	**1.10**
AMP	GPT-2	**1.18**	**1.14**	**1.12**	**1.10**	**1.10**
`torch.compile`	GPT-2	**1.18**	**1.14**	**1.12**	**1.10**	**1.10**
quantize	GPT-2	1.20	1.16	1.14	1.13	1.12
top 6	GPT-2	2.00	1.95	1.93	1.90	1.89
top 3	GPT-2	2.36	2.31	2.28	2.24	2.22
top 2	GPT-2	2.66	2.66	2.66	2.64	2.63
odd-alt 6	GPT-2	**1.31**	**1.29**	**1.27**	**1.27**	**1.27**
odd-alt 3	GPT-2	**1.43**	**1.42**	**1.40**	**1.41**	**1.41**
odd-alt 2	GPT-2	2.32	2.34	2.37	2.64	2.79
even-alt 6	GPT-2	1.68	1.64	1.63	1.61	1.61
even-alt 3	GPT-2	1.68	1.66	1.64	1.64	1.63
even-alt 2	GPT-2	1.76	1.76	1.72	1.77	1.80
symmetric 6	GPT-2	1.33	1.30	1.29	1.29	1.29
symmetric 3	GPT-2	1.48	1.47	1.46	1.47	1.47
symmetric 2	GPT-2	**1.67**	**1.67**	**1.66**	**1.68**	**1.69**
bottom 6	GPT-2	2.37	2.39	2.37	2.43	2.58
bottom 3	GPT-2	4.20	4.21	4.20	4.13	4.26
bottom 2	GPT-2	4.71	4.58	4.23	4.24	4.16

Table 5.4: Preliminary BPC on 2080

Optimization	Model	Speed 128	Speed 256	Speed 512	Speed 768	Speed 1023
baseline	GPT-2	**572.31**	**524.28**	269.21	173.19	123.60
AMP	GPT-2	486.24	478.20	**398.44**	**270.68**	**188.69**
`torch.compile`	GPT-2	465.09	408.91	315.18	201.58	142.82
quantize	GPT-2	422.66	329.91	207.78	135.48	99.26
top 6	GPT-2	**678.03**	**607.84**	**311.28**	**200.80**	142.18
top 3	GPT-2	780.44	693.28	363.89	236.37	169.09
top 2	GPT-2	940.43	824.15	443.33	**289.48**	207.42
odd-alt 6	GPT-2	640.91	593.90	305.64	198.30	141.49
odd-alt 3	GPT-2	742.32	668.26	357.88	232.92	167.51
odd-alt 2	GPT-2	908.20	801.65	432.44	285.60	207.22
even-alt 6	GPT-2	668.95	607.34	308.28	199.88	142.73
even-alt 3	GPT-2	778.42	705.57	362.48	236.61	169.47
even-alt 2	GPT-2	925.13	829.51	440.67	288.79	207.65
symmetric 6	GPT-2	637.66	586.59	307.08	197.39	141.89
symmetric 3	GPT-2	746.10	671.63	356.56	233.28	167.87
symmetric 2	GPT-2	891.15	790.72	431.21	281.11	206.49
bottom 6	GPT-2	650.43	588.76	307.86	198.24	**142.14**
bottom 3	GPT-2	**792.54**	**711.10**	**365.86**	**237.54**	**170.75**
bottom 2	GPT-2	**956.89**	**835.49**	**444.69**	289.19	**209.75**

Table 5.5: Preliminary Coding Speed on 2080

Optimization	Model	BPC 128	BPC 256	BPC 512	BPC 768	BPC 1023
baseline	GPT-2	**1.18**	**1.14**	**1.12**	**1.10**	**1.10**
AMP	GPT-2	**1.18**	**1.14**	**1.12**	1.11	**1.10**
AMP, FlashAttention	GPT-2	**1.18**	**1.14**	**1.12**	1.11	**1.10**
AMP, FlashAttention2	GPT-2	**1.18**	**1.14**	**1.12**	1.11	**1.10**

Table 5.6: Preliminary BPC on A6000

Optimization	Model	Speed 128	Speed 256	Speed 512	Speed 768	Speed 1023
baseline	GPT-2	**715.56**	**704.23**	550.15	368.12	253.49
AMP	GPT-2	598.47	596.87	565.55	397.01	279.29
AMP, FlashAttention	GPT-2	647.62	637.24	588.66	**491.23**	413.51
AMP, FlashAttention2	GPT-2	660.99	656.86	**604.27**	486.41	**418.65**

Table 5.7: Preliminary Coding Speed on A6000

Optimization	Model	BPC 128	BPC 256	BPC 512	BPC 768	BPC 1023
baseline	DistilGPT2	**1.34**	**1.30**	**1.28**	**1.28**	1.78
baseline	GPT-2	**1.21**	**1.17**	**1.15**	**1.14**	**1.14**
baseline	GPT-2 Medium	**1.09**	**1.06**	**1.04**	**1.03**	**1.03**
AMP, `torch.compile`	DistilGPT2	**1.34**	**1.30**	**1.28**	**1.28**	1.78
AMP, `torch.compile`	GPT-2	**1.21**	**1.17**	**1.15**	**1.14**	**1.14**
AMP, `torch.compile`	GPT-2 Medium	**1.09**	**1.06**	**1.04**	**1.03**	**1.03**
symmetric 2	DistilGPT2	1.90	1.93	1.94	1.96	1.97
symmetric 2	GPT-2	1.73	1.73	1.72	1.74	1.76
symmetric 2	GPT-2 Medium	1.89	1.92	1.88	1.82	1.86
symmetric 3	DistilGPT2	1.67	1.65	1.65	1.65	1.71
symmetric 3	GPT-2	1.53	1.51	1.51	1.51	1.51
symmetric 3	GPT-2 Medium	1.48	1.47	1.47	1.48	1.47
symmetric 6	DistilGPT2	1.50	1.47	1.47	1.48	**1.52**
symmetric 6	GPT-2	1.36	1.34	1.33	1.33	1.33
symmetric 6	GPT-2 Medium	1.22	1.20	1.18	1.18	1.19

Table 5.8: Final BPC on 2080

Optimization	Model	Speed 128	Speed 256	Speed 512	Speed 768	Speed 1023
baseline	DistilGPT2	443.21	435.16	357.61	260.46	192.48
baseline	GPT-2	270.50	265.58	237.15	167.30	119.16
baseline	GPT-2 Medium	249.74	206.86	101.90	63.32	45.35
AMP, `torch.compile`	DistilGPT2	620.87	600.15	474.25	318.19	256.46
AMP, `torch.compile`	GPT-2	169.62	163.51	188.81	210.07	176.65
AMP, `torch.compile`	GPT-2 Medium	360.89	252.22	179.12	83.26	50.97
symmetric 2	DistilGPT2	**808.68**	**765.21**	**661.87**	473.24	**304.53**
symmetric 2	GPT-2	**273.56**	**285.53**	**260.56**	255.47	**250.31**
symmetric 2	GPT-2 Medium	520.03	**534.58**	**356.46**	**157.00**	**112.14**
symmetric 3	DistilGPT2	758.38	722.24	641.42	**477.34**	249.52
symmetric 3	GPT-2	205.46	231.00	218.28	221.45	265.04
symmetric 3	GPT-2 Medium	**603.67**	433.70	215.02	92.46	84.89
symmetric 6	DistilGPT2	681.50	652.30	537.63	359.43	281.67
symmetric 6	GPT-2	199.53	190.91	180.36	**317.53**	187.83
symmetric 6	GPT-2 Medium	428.99	329.68	175.52	119.60	59.72

Table 5.9: Final Coding Speed on 2080

Optimization	Model	BPC 128	BPC 256	BPC 512	BPC 768	BPC 1023
baseline	DistilGPT2	**1.34**	**1.30**	**1.28**	**1.28**	1.78
baseline	GPT-2	**1.21**	**1.17**	**1.15**	**1.14**	**1.14**
baseline	GPT-2 Medium	**1.09**	**1.06**	**1.04**	**1.03**	**1.03**
AMP, `torch.compile`	DistilGPT2	1.35	1.31	**1.28**	**1.28**	1.77
AMP, `torch.compile`	GPT-2	1.22	1.18	1.16	1.15	**1.14**
AMP, `torch.compile`	GPT-2 Medium	1.10	1.07	**1.04**	1.04	**1.03**
symmetric 2	DistilGPT2	1.90	1.93	1.94	1.96	1.97
symmetric 2	GPT-2	1.73	1.74	1.73	1.75	1.77
symmetric 2	GPT-2 Medium	1.89	1.93	1.88	1.83	1.86
symmetric 3	DistilGPT2	1.67	1.65	1.65	1.65	1.71
symmetric 3	GPT-2	1.53	1.52	1.51	1.51	1.52
symmetric 3	GPT-2 Medium	1.48	1.47	1.47	1.49	1.47
symmetric 6	DistilGPT2	1.50	1.48	1.48	1.48	**1.52**
symmetric 6	GPT-2	1.37	1.34	1.33	1.33	1.33
symmetric 6	GPT-2 Medium	1.23	1.20	1.18	1.19	1.19
FlashAttention2	DistilGPT2	1.35	1.31	**1.28**	**1.28**	1.77
FlashAttention2	GPT-2	1.22	1.18	1.16	**1.14**	**1.14**
FlashAttention2	GPT-2 Medium	1.10	1.07	**1.04**	1.04	**1.03**

Table 5.10: Final BPC on A6000

Optimization	Model	Speed 128	Speed 256	Speed 512	Speed 768	Speed 1023
baseline	DistilGPT2	1204.61	1152.00	873.00	610.07	423.30
baseline	GPT-2	713.30	702.88	525.70	345.08	234.94
baseline	GPT-2 Medium	382.39	370.72	223.03	139.60	89.42
AMP, `torch.compile`	DistilGPT2	1518.13	1483.36	1206.50	819.37	549.54
AMP, `torch.compile`	GPT-2	417.78	305.34	308.15	385.16	316.79
AMP, `torch.compile`	GPT-2 Medium	211.97	212.95	207.10	186.62	128.31
symmetric 2	DistilGPT2	**2135.53**	**2017.88**	**1659.49**	**1199.47**	**857.58**
symmetric 2	GPT-2	**751.93**	**711.20**	**678.37**	**650.12**	**540.68**
symmetric 2	GPT-2 Medium	**400.61**	**396.38**	**381.00**	**336.63**	**237.13**
symmetric 3	DistilGPT2	1899.06	1840.12	1475.98	1046.32	722.87
symmetric 3	GPT-2	511.46	573.83	550.64	528.95	436.19
symmetric 3	GPT-2 Medium	313.24	185.15	300.99	263.21	180.65
symmetric 6	DistilGPT2	1714.02	1655.07	1025.30	644.92	494.86
symmetric 6	GPT-2	495.06	483.50	450.78	428.16	366.16
symmetric 6	GPT-2 Medium	257.08	254.93	247.70	268.37	183.54
FlashAttention2	DistilGPT2	692.11	787.41	765.99	731.31	692.77
FlashAttention2	GPT-2	466.15	460.28	446.11	432.45	421.10
FlashAttention2	GPT-2 Medium	246.58	241.77	236.52	232.00	227.23

Table 5.11: Final Coding Speed on A6000