Transformer tricks: Precomputing the first layer

Nils Graef*
OpenMachine

Abstract

This micro-paper [1] describes a trick to speed up inference of transformers with RoPE [2] (such as LLaMA, Mistral, PaLM, and Gemma [3]). For these models, a large portion of the first transformer layer can be precomputed, which results in slightly lower latency and lower cost-per-token. Because this trick optimizes only one layer, the relative savings depend on the total number of layers. For example, the maximum savings for a model with only 4 layers (such as Whisper tiny [4]) is limited to 25%, while a 32-layer model is limited to 3% savings. See [5, 6, 7, 8, 9] for code and more transformer tricks.
The next two sections detail the precompute for transformers with parallel attention/FFN [10] (such as GPT-J, Pythia, and PaLM [10, 11, 12]) and without (such as Llama 2, Mistral, and Mixtral [13, 14, 15, 16]).

1 Precompute for parallel transformers

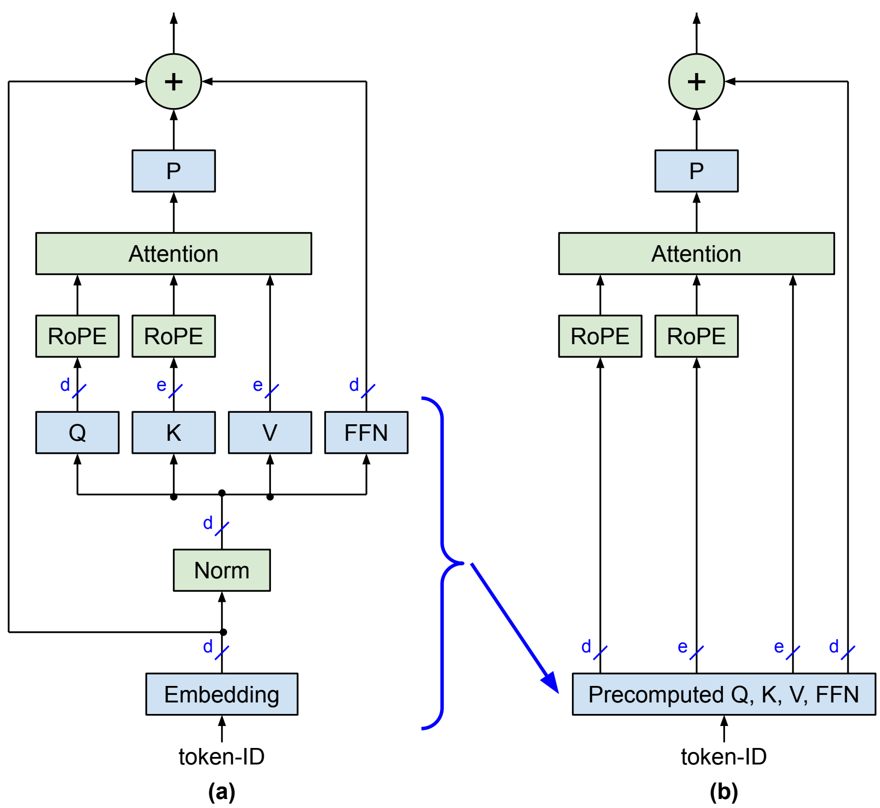

Figure 1: First layer of parallel transformer (a) without precompute; and (b) with precompute of FFN and linear layers Q, K, and V.

*info@openmachine.ai

Figure 1(a) shows the first layer of a transformer with RoPE and parallel attention/FFN. Because the inputs of Q, K, V, and FFN only depend on the embedding, we can precompute their outputs and store them in memory instead of the input embeddings, see Figure 1(b). Figure 1 uses the following dimensions, based on the type of attention such as multi-head attention (MHA) [17], multi-query attention (MQA) [18], and grouped-query attention (GQA) [19]:

- d: embedding dimension.
- e: $e = d$ for MHA. For MQA, $e = d/n_{heads}$. And for GQA, $e = d \cdot n_{kv\_heads}/n_{heads}$.
- Q, K, V, P are the linear layers for query, keys, values, and post-attention projection.
- FFN (feedforward network) is usually a two-layer MLP (multi-layer perceptron). Mistral and Llama2 use a two-layer MLP with a GLU variant [20] for the first layer. And MoE models (mixture-of-experts) [21] such as Mixtral use a switch FFN.
- The embedding layer is implemented by a simple memory read operation, where the token-ID provides the read-address to read d values from memory.

The precompute is done as follows: For each token stored in the embedding table, perform the calculations needed for the first layer normalization, FFN, skip-connection, and linear layers Q, K, V, and store the results in memory instead of the original input-embeddings. This precompute is done offline only once and stored in the parameter memory (along with weights, biases, and output-embeddings).

The benefits of precompute include:

- **Lower computational complexity per token**: For each token, we save the operations needed for FFN and the linear layers Q, K, V. This can speed up inference if the system is limited by compute.
- **Fewer memory reads for low batch sizes**: This can speed up inference for systems that are memory bandwidth limited, especially during the autoregressive next-token-prediction phase, see the table below and section 3 for examples.

	Without precompute	**With precompute**
	1) For each token, read d embedding values 2) Plus, for each batch, read weights for Q, K, V, FFN	For each token, read $2(d+e)$ precomputed values
Reads per batch: (B is batch-size)	$B \cdot d + \texttt{num\_weights\_Q\_K\_V\_FFN}$	$B \cdot 2(d+e)$

Notes on batch size:

- During the prefill phase, many implementations use a batch size larger than 1, because the input tokens can be processed in parallel.
- During the autoregressive next-token-generation phase, single-user implementations often use a batch size of `num_beams` (i.e. the width of the beam search, such as `num_beams` = 4), while multi-user implementations use larger batch sizes. However, the maximum batch size for multi-user applications can be limited by the total memory capacity as the number of KV-caches increases linearly with the batch size.

However, precomputing the first layer can increase (or decrease) the total memory size, which depends on the vocabulary size and the number of eliminated weights as shown in the table below. For example, the total memory size of Mistral-7B only increases by 2%, see section 3 for more details.

Without precompute	**With precompute**
1) Store embeddings: $d \cdot \texttt{vocab\_size}$	Store precomputed values: $2(d+e) \cdot \texttt{vocab\_size}$
2) Store weights for Q, K, V, and FFN	

2 Precompute for serial transformers

Transformers without the parallel attention/FFN scheme can also benefit from precompute, but the savings are smaller: As shown in Figure 2(c), we can only precompute Q, K, and V, but not the FFN. For reference, Figure 2(a) shows the vanilla transformer with absolute positional encoding (PE) instead of RoPE and with pre-normalization [22]. The PE is located right after the embedding layer, which prevents us from precomputing the first layer. But replacing the PE by RoPE, as done in Figure 2(b), allows us to precompute the linear layers Q, K, and V and store the precomputed values along the embeddings in memory as illustrated in Figure 2(c).

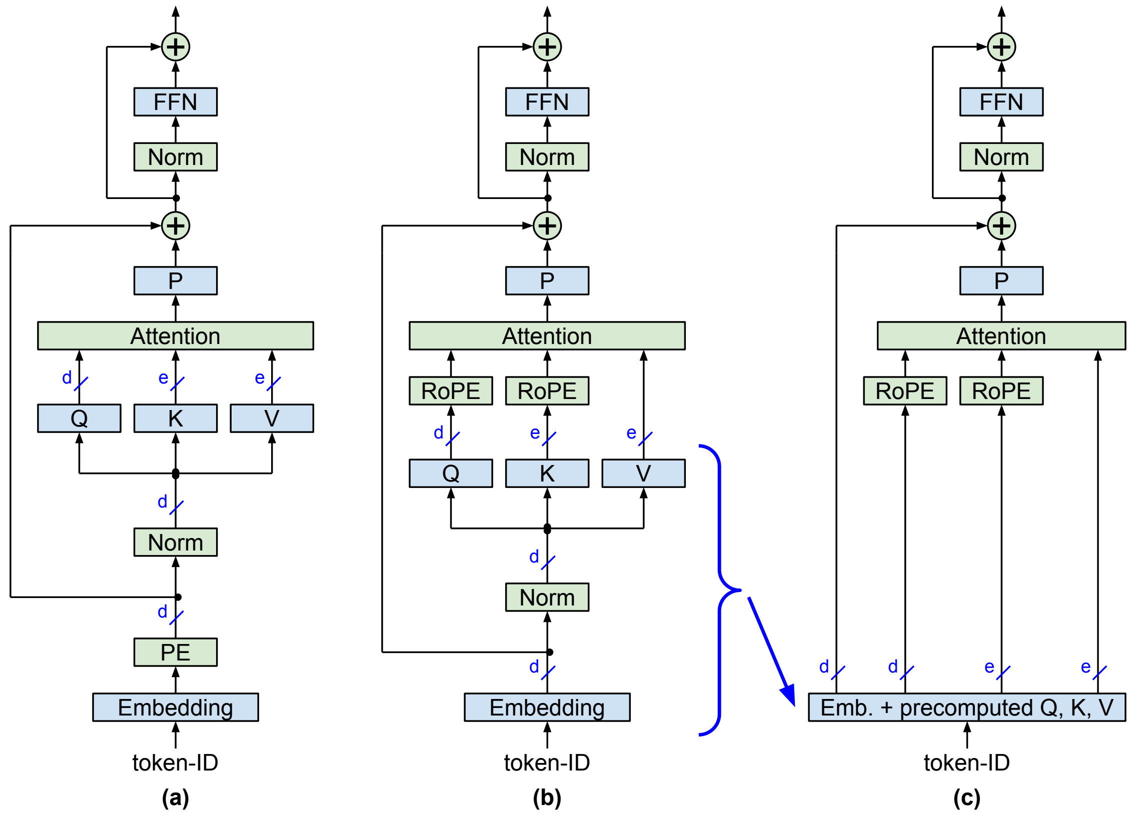

Figure 2: First transformer layer. (a) Vanilla with pre-normalization and vanilla PE; (b) Vanilla with RoPE; (c) Precomputing linear layers Q, K, V.

3 Examples

Parameter	**Pythia-6.9B**	**Mistral-7B**	**Mixtral-8x7B**	**Notes**
Parallel attention/FFN?	parallel	serial		[10]
MHA, MQA, or GQA?	MHA	GQA		[17, 18, 19]
`dim` (aka d)	4,096			embedding dimension
`n_layers`	32			number of layers
`n_heads`, `n_kv_heads`	32, 32	32, 8		number of heads, KV-heads
`e` (output dim. of K, V)	4,096	1,024		`e = d * n_kv_heads / n_heads`
FFN type	2-layer MLP	SwiGLU *)	SwiGLU MoE	*) MLP with SwiGLU (GLU variant) [20, 21]
FFN `hidden_dim`	16,384	14,336		FFN hidden dimension
FFN `n_experts`	1		8	FFN number of experts
`vocab_size`	50,400	32,000		vocabulary size
Number of weights (calculated from above parameters):				
Q+P weights per layer	33,554,432			`2 * dim * dim`
K+V weights per layer	33,554,432	8,388,608		`2 * dim * dim / n_heads * n_kv_heads`
FFN weights per layer	134,217,728	176,160,768	1,409,286,144	`(2 or 3) * dim * hidden_dim * n_exp.`
Input+output embed.	412,876,800	262,144,000		`2 * dim * vocab_size`
Total weights:	6.9B	7.2B	46.7B	

The table above compares the configurations and number of weights of Pythia-6.9B, Mistral-7B, and Mixtral-8x7B. The next table shows the memory read savings and memory size increases for Pythia-6.9B, Mistral-7B, and a hypothetical Mixtral-8x7B with parallel attention/FFN layers.

	Pythia-6.9B	**Mistral-7B**	**Hypothetical Mixtral-8x7B with parallel attn./FFN**
Number of weights that can be eliminated	184,549,376	25,165,824	1,434,451,968
Number of reads w/o precompute for batch 1	184,553,472	25,169,920	1,434,456,064
Number of reads with precompute for batch 1	16,384	10,240	10,240
First layer reduction factor for batch size 1:	**11,264x**	**2,458x**	**140,084x**
First layer reduction factor for batch size 16:	704x	154x	8,756x
First layer reduction factor for batch size 256:	44x	10x	548x
First layer reduction factor for batch size 1,024:	11x	3x	137x
Increase (or decrease) of total weight memory size:			
Increase embedding memory by $(2e + d) \cdot \texttt{vocab\_size}$	619,315,200	196,608,000	
Memory decrease due to elimination of weights	–184,549,376	–25,165,824	-1,434,451,968
Total absolute memory increase (or decrease):	434,765,824	171,442,176	**-1,237,843,968**
Total relative memory increase (or decrease):	6%	**2%**	**–3%**

Acknowledgments

We would like to thank James Martens (DeepMind) for his generous support and endorsement for the arXiv submission process.

References

[1] Frank Elavsky. The Micro-Paper: Towards cheaper, citable research ideas and conversations. February 2023. *arXiv:2302.12854*.

[2] Jianlin Su, Yu Lu, Shengfeng Pan, Ahmed Murtadha, Bo Wen, and Yunfeng Liu. RoFormer: Enhanced transformer with Rotary Position Embedding. April 2021. *arXiv:2104.09864*.

[3] Gemma Team, Google DeepMind. Gemma: Open Models Based on Gemini Research and Technology. 2024.

[4] Alec Radford, Jong Wook Kim, Tao Xu, Greg Brockman, Christine McLeavey, and Ilya Sutskever. Robust speech recognition via large-scale weak supervision. December 2022. *arXiv:2212.04356*.

[5] OpenMachine. Transformer tricks. 2024. URL `https://github.com/OpenMachine-ai/transformer-tricks`.

[6] Nils Graef and Andrew Wasielewski. Slim attention: cut your context memory in half without loss – K-cache is all you need for MHA. March 2025. *arXiv:2503.05840*.

[7] Nils Graef, Andrew Wasielewski, and Matthew Clapp. FlashNorm: fast normalization for LLMs. July 2024. *arXiv:2407.09577*.

[8] Nils Graef. Transformer tricks: Removing weights for skipless transformers. April 2024. *arXiv:2404.12362*.

[9] Nils Graef. MatShrink: Lossless weight compression for back-to-back linear layers. 2025. URL `https://github.com/OpenMachine-ai/transformer-tricks/blob/main/doc/matShrink.pdf`.

[10] Ben Wang and Aran Komatsuzaki. GPT-J-6B: A 6 billion parameter autoregressive language model. 2021. *Github repo*.

[11] Stella Biderman, Hailey Schoelkopf, Quentin Anthony, Herbie Bradley, Kyle O'Brien, Eric Hallahan, Mohammad Aflah Khan, Shivanshu Purohit, Usvsn Sai Prashanth, Edward Raff, Aviya Skowron, Lintang Sutawika, and Oskar van der Wal. Pythia: A suite for analyzing large language models across training and scaling. April 2023. *arXiv:2304.01373*.

[12] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, et al. PaLM: Scaling language modeling with Pathways. April 2022. *arXiv:2204.02311*.

[13] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. LLaMA: Open and efficient foundation language models. February 2023. *arXiv:2302.13971*.

[14] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, et al. Llama 2: Open foundation and fine-tuned chat models. July 2023. *arXiv:2307.09288*.

[15] Albert Q Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. Mistral 7B. October 2023. *arXiv:2310.06825*.

[16] Albert Q Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Emma Bou Hanna, Florian Bressand, Gianna Lengyel, Guillaume Bour, Guillaume Lample, Lélio Renard Lavaud, Lucile Saulnier, Marie-Anne Lachaux, Pierre Stock, Sandeep Subramanian, Sophia Yang, Szymon Antoniak, Teven Le Scao, Théophile Gervet, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. Mixtral of Experts. January 2024. *arXiv:2401.04088*.

[17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. June 2017. *arXiv:1706.03762*.

[18] Noam Shazeer. Fast Transformer Decoding: One Write-Head is All You Need. November 2019. *arXiv:1911.02150*.

[19] Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. GQA: Training generalized multi-query transformer models from multi-head checkpoints. May 2023. *arXiv:2305.13245*.

[20] Noam Shazeer. GLU Variants Improve Transformer. February 2020. *arXiv:2002.05202*.

[21] Omar Sanseviero, Lewis Tunstall, Philipp Schmid, Sourab Mangrulkar, Younes Belkada, and Pedro Cuenca. Mixture of Experts Explained. December 2023. *HuggingFace blog*.

[22] Ruibin Xiong, Yunchang Yang, Di He, Kai Zheng, Shuxin Zheng, Chen Xing, Huishuai Zhang, Yanyan Lan, Liwei Wang, and Tie-Yan Liu. On Layer Normalization in the Transformer Architecture. February 2020. *arXiv:2002.04745*.