Flash normalization: simplified RMSNorm for LLMs

Nils Graef, Matthew Clapp, Andrew Wasielewski
OpenMachine, San Francisco Bay Area, `info@openmachine.ai`

Abstract

RMSNorm [1] is used by many LLMs such as Llama, Mistral, and OpenELM [2, 3, 4]. This paper details FlashNorm, which is an exact but simpler implementation of RMSNorm followed by linear layers. See [5, 6, 7] for code and more transformer tricks.

1 Flash normalization

RMSNorm [1] normalizes the elements a_i of vector $\vec{a}$ as $y_i = \frac{a_i}{\text{RMS}(\vec{a})} \cdot g_i$ with $\text{RMS}(\vec{a}) = \sqrt{\frac{1}{n}\sum_{i=1}^{n} a_i^2}$ and normalization weights g_i. In transformer [8] and other neural networks, RMSNorm is often followed by a linear layers as illustrated in Fig. 1(a), which we optimize as follows:

- **Weightless normalization**: We merge the normalization weights g_i into the linear layer with weights $\mathbf{W}$, resulting in a modified weight matrix $\mathbf{W}^*$ with $W^*_{i,j} = g_i \cdot W_{i,j}$ as illustrated in Fig. 1(b). This works for linear layers with and without bias.
- **Deferred normalization**: Instead of normalizing before the linear layer, we normalize after the linear layer, as shown in Fig. 1(c). This only works if the linear layer is bias-free, which is the case for many LLMs such as Llama, Mistral, and OpenELM.

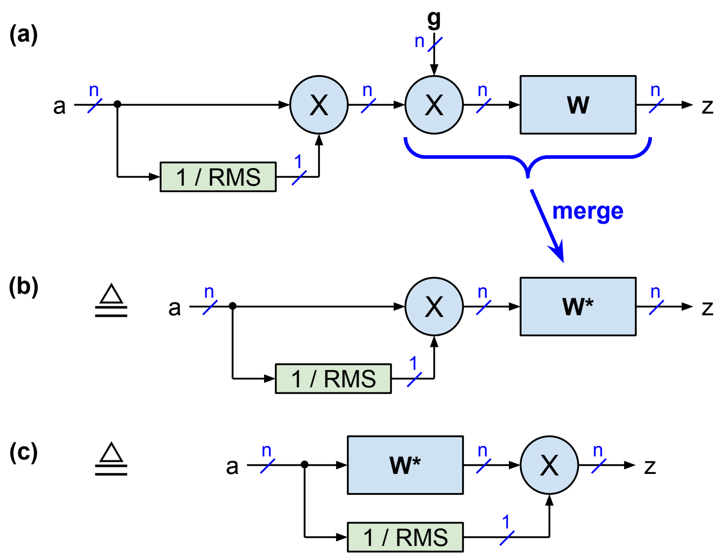

Figure 1: Mathematically identical implementations of RMSNorm followed by a bias-free linear layer: (a) unoptimized version with weight matrix $\mathbf{W}$; (b) optimized version with normalization weights g_i merged into the linear layer with new weights $\mathbf{W}^*$; (c) optimized version with deferred normalization. The $\triangleq$ symbol denotes mathematical identity.

In summary, FlashNorm eliminates the normalization weights and defers the normalization to the output of the linear layer, which removes a compute bottleneck described at the end of this paper. Deferring the normalization is similar to Flash Attention [9], where the normalization by the softmax denominator is done after the multiplication of softmax arguments with value projections (V) (so that keys and values can be processd in *parallel*). Therefore, we call our implementation *flash* normalization (or FlashNorm), which allows us to compute the linear layer and $\text{RMS}(\vec{a})$ in *parallel* (instead of sequentially).

Mehta et al. report significant changes in the overall tokens-per-second throughput when they modify the layer normalization implementation, which they attribute to a lack of kernel fusion for the underlying GPU. The simplifications presented here reduce the number of operations and thus the number of the individual kernel launches mentioned in [4].

2 Flash normalization for FFN

For the feed-forward networks (FFN) of LLMs, the linear layers at the FFN input usually have more output channels than input channels. In this case, deferring the normalization requires more scaling operations (i.e. more multiplications). This section details ways to reduce the number of scaling operations for bias-free FFNs.

2.1 Flash normalization for FFNs with ReLU

Even though ReLU is a nonlinear function, multiplying its argument by a non-negative scaling factor s is the same as scaling its output by s, i.e. $\text{ReLU}(s \cdot \vec{a}) = s \cdot \text{ReLU}(\vec{a})$ for $s \geq 0$ [10]. Because of this scale-invariance, we can defer the normalization to the output of the FFN as illustrated in Fig. 2(b), which saves $f - n$ multipliers.

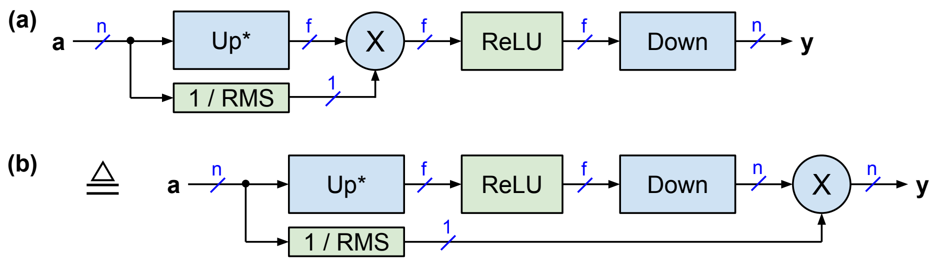

Figure 2: FFN with ReLU and preceding flash normalization: (a) unoptimized version; (b) optimized version where the normalization is deferred to the output of the FFN. Up and Down denote the linear layers for up and down projections.

2.2 Flash normalization for FFNs with GLU variant

Fig. 3(a) shows an FFN with a GLU variant [11] and flash normalization at its input. The flash normalization requires two sets of f multipliers at the outputs of the Gate and Up linear layers in Fig. 3(a). One set can be deferred to the FFN output in Fig. 3(b), which saves $f - n$ multipliers.

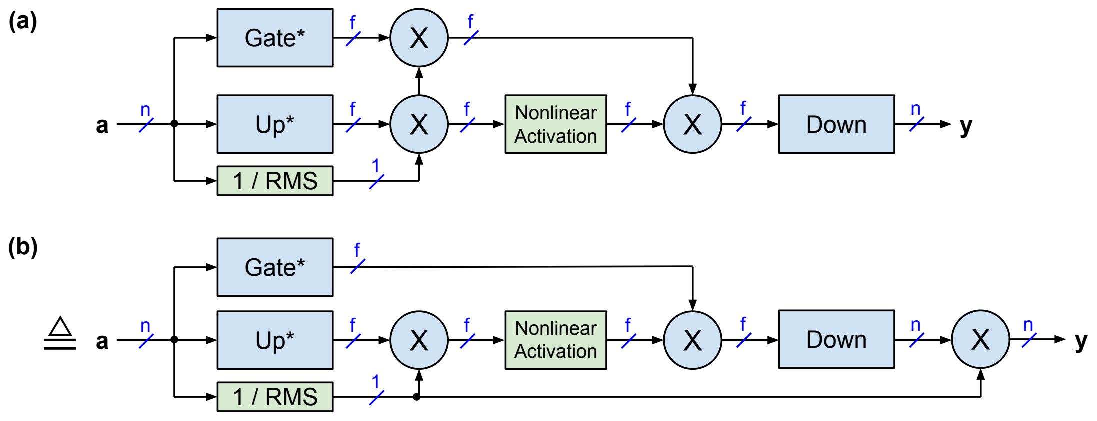

Figure 3: FFN with GLU variant and preceding flash normalization: (a) unoptimized version; (b) optimized version with fewer scaling multipliers. Gate, Up, and Down denote the linear layers for gate, up, and down projections.

Special case for ReGLU and Bilinear GLU: If the activation function is ReLU (aka ReGLU [11]) or just linear (aka bilinear GLU [11]), then we can also eliminate the scaling before the activation function and combine it with the scaling at the output as illustrated in Fig. 4(b), which saves $2f - n$ multipliers. Now the output scaling is using the reciprocal of the squared RMS as scaling value, which is the same as the reciprocal of the mean-square (MS):

$$\frac{1}{(\mathrm{RMS}(\vec{a}))^2} = \frac{1}{\mathrm{MS}(\vec{a})} = \frac{1}{\frac{1}{n}\sum_{i=1}^{n} a_i^2} = \frac{n}{\sum_{i=1}^{n} a_i^2}$$

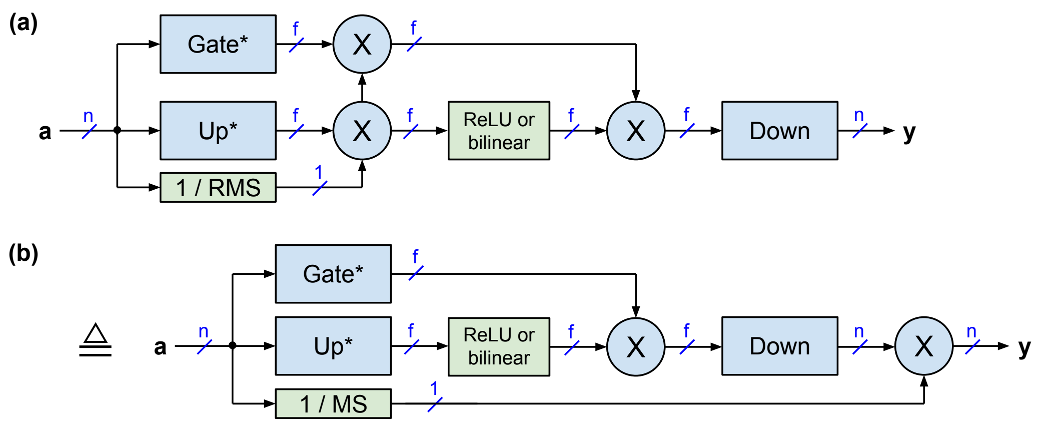

Figure 4: FFN with ReGLU (or bilinear GLU) and preceding flash normalization: (a) unoptimized version; (b) optimized version with fewer scaling multipliers.

3 Flash normalization for attention with RoPE

TODO: fix text and figures with RoPE: RoPE is not just scaling the channels with twiddle factors, but there is also an add operation after the scaling such as $y_1 = x_1 \cos(\alpha) - x_2 \sin(\alpha)$ and $y_2 = x_1 \sin(\alpha) + x_2 \cos(\alpha)$. So we would fuse the normalization scaling with these twiddle factors and then perform the multiply and add operations.

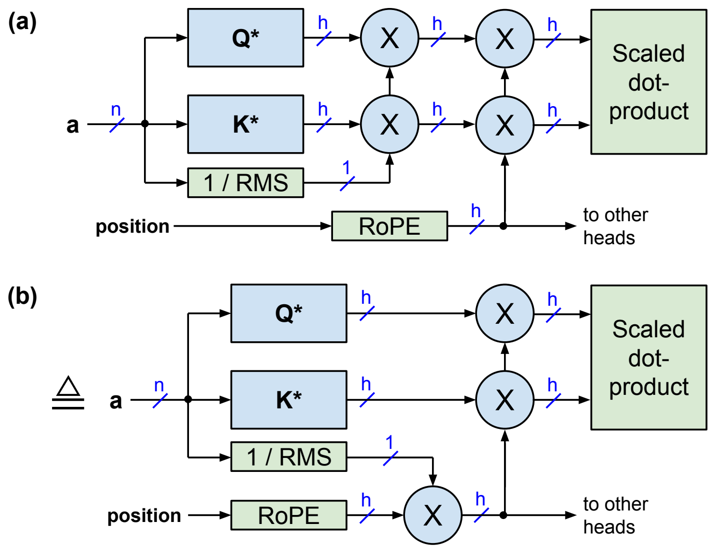

Figure 5: Flash normalization for scaled dot-product attention with RoPE: (a) unoptimized version; (b) optimized version where the normalization is fused with RoPE.

Fig. 5(a) shows the Q and K linear layers with flash normalization followed by RoPE [12] and scaled dot-product attention [8]:

- Q* and K* are the linear layers for Q (queries) and K (keys) fused with the normalization weights of the activation vector $\vec{a}$ (according to flash normalization).

- h is the dimension of the attention heads.
- Note that the RoPE scaling vector only depends on the position of activation vector $\vec{a}$ and is shared among all attention heads. Therefore, it's more efficient to first scale the RoPE scaling vector by $1/\text{RMS}(\vec{a})$ and then share this fused scaling vector among all attention heads as illustrated in Fig. 5(b). This saves $h \cdot (H - 1)$ multipliers, where H is the number of attention heads.
- Furthermore, we can fuse the scaling factor $1/\sqrt{h}$ of the scaled dot-product with the $1/\text{RMS}(\vec{a})$ factor (note that we need to use $\sqrt{1/\sqrt{h}}$ as a scaling factor for this).
- Unfortunately, the V linear layer (value projection) still needs the normalization at its output.

4 Optimizations for QK-normalization with RoPE

Some LLMs use query-key normalization [13]. For example, each layer of OpenELM [4] has the following two sets of normalization weights:

- `q_norm_weight`: query normalization weights for all heads of this layer
- `k_norm_weight`: key normalization weights for all heads of this layer

Unfortunately, FlashNorm can't be applied for QK-normalization. But for the type of QK-normalization used in OpenELM, we can apply the following two optimizations detailed in the next sections:

1. Eliminate the RMS calculation before the Q and K linear layers.
2. Fuse the normalization weights with RoPE.

4.1 Eliminate RMS calculation before QK linear layers

Fig. 6(a) shows a linear layer with flash normalization followed by QK-normalization. The weights of the first normalization are already merged into the linear layer weights $\mathbf{W}^*$. Note that $\text{RMS}(s \cdot \vec{a}) = s \cdot \text{RMS}(\vec{a})$ where s is scalar and $\vec{a}$ is a vector. Due to this scale-invariance of the RMS function, the second multiplier (scaler s_c) in the pipeline of Fig. 6(a) cancels out the first multiplier (scaler s_a). Fig. 6(b) takes advantage of this property. We can express this by using the vectors $\vec{a}, \vec{b}, \vec{c}$ along the datapath in Fig. 6 as follows:

- Note that $s_c = \frac{1}{\text{RMS}(\vec{c})} = \frac{1}{\text{RMS}(\vec{b} \cdot s_a)} = \frac{1}{s_a \cdot \text{RMS}(\vec{b})} = \frac{s_b}{s_a}$.
- With above, we can show that the y outputs of figures 6(a) and 6(b) are identical:

$$y = \vec{a} \cdot \mathbf{W}^* \cdot s_a \cdot s_c \cdot \vec{g} = \vec{a} \cdot \mathbf{W}^* \cdot s_a \cdot \frac{s_b}{s_a} \cdot \vec{g} = \vec{a} \cdot \mathbf{W}^* \cdot s_b \cdot \vec{g}$$

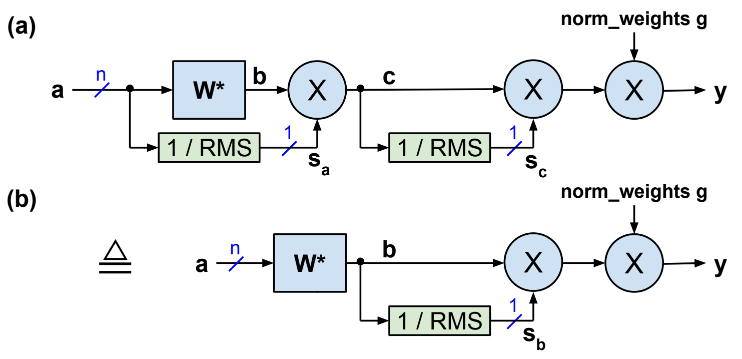

Figure 6: Linear layer with flash normalization followed by another normalization: (a) unoptimized version; (b) optimized version.

The scale-invariance property of $\text{RMS}(\vec{a})$ doesn't hold exactly true for RMS with epsilon (see appendix). This should not matter because the epsilon only makes an impact if the RMS (or energy) of the activation vector is very small, in which case the epsilon limits the upscaling of this low-energy activation vector.

4.2 Fuse normalization weights with RoPE

Fig. 7(a) illustrates QK-normalization with RoPE. If the QK-normalization weights are the same for all heads of a layer, as is the case for OpenELM [4], then we can fuse them with RoPE: multiply the RoPE scaling vector with the normalization weights and then share the fused scaling vectors across all heads of the LLM layer as shown in Fig. 7(b).

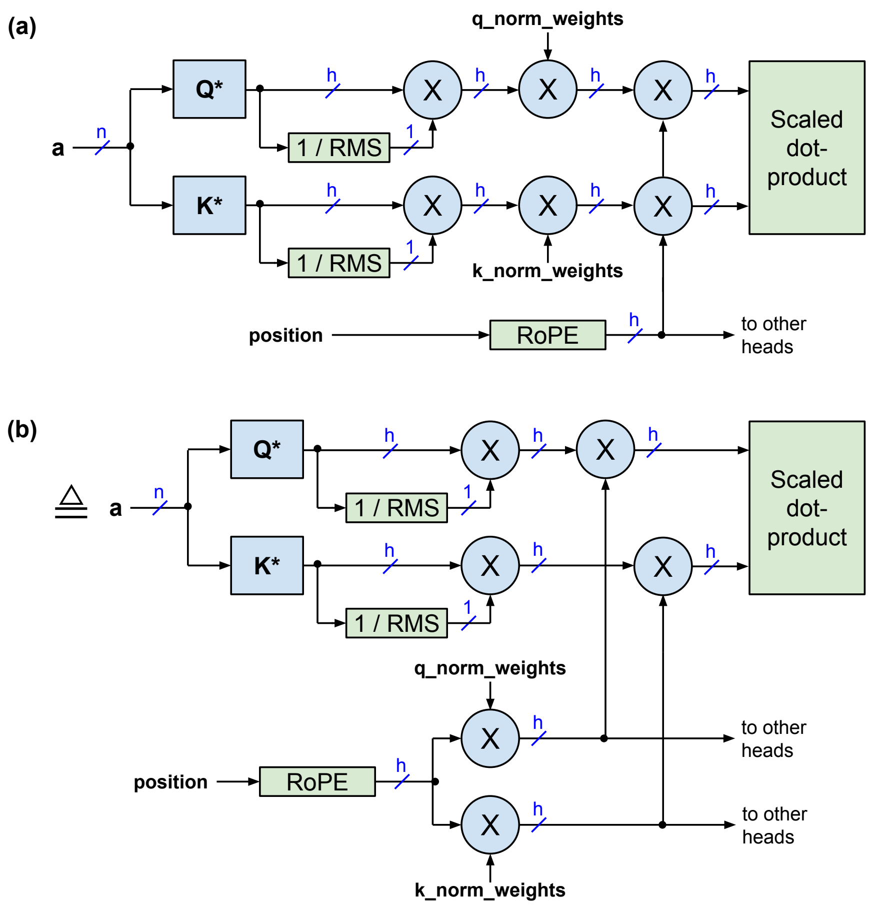

Figure 7: QK-normalization with RoPE: (a) unoptimized version; (b) optimized version.

5 Bottleneck of RMS normalization for batch 1

This section describes the compute bottleneck of RMS normalization that exists for batch size 1. This bottleneck is different from the bottleneck detailed in [4]. Let's consider a processor with one vector unit and one matrix unit:

- The matrix multiplications of the linear layers are performed by the matrix unit, while the vector unit performs vector-wise operations such as RMSNorm and FlashNorm.
- Let's assume that the vector unit can perform m operations per cycle and the matrix unit can perform m^2 operations per cycle, where m is the processor width. Specifically:
  - Multiplying an n-element vector with an $n \times n$ matrix takes n^2 MAD (multiply-add) operations, which takes n^2/m^2 cycles with our matrix unit.
  - Calculating $1/\text{RMS}(\vec{a})$ takes n MAD operations (for squaring and adding) plus 3 scalar operations (for $1/\sqrt{x/n}$), which takes n/m cycles with our vector unit if we ignore the 3 scalar operations.
  - Scaling an n-element vector by a scaling factor takes n multiply operations, which takes n/m cycles.

For the example $n = 512, m = 128$ and batch 1, Fig. 8 shows timing diagrams without and with deferred normalization:

- Without deferred normalization, the matrix unit has to wait for 8 cycles until the vector unit has calculated the RMS value and completed the scaling by $1/\text{RMS}(\vec{a})$ as illustrated in Fig. 8(a).

- As shown in Fig. 8(b), it is possible to start the matrix unit 3 cycles earlier if the weight matrix $\mathbf{W}$ is processed in row-major order for example. But the RMS calculation still presents a bottleneck.
- FlashNorm eliminates this bottleneck: With deferred normalization, the matrix unit computes the vector-matrix multiplication in parallel to the vector unit's RMS calculation as shown in Fig. 8(c). The scaling at the end can be performed in parallel to the matrix unit if $\mathbf{W}$ is processed in column-major order for example.

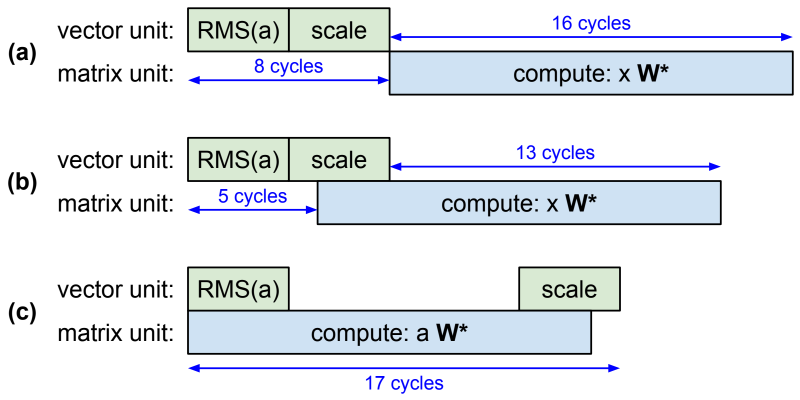

Figure 8: Timing diagrams for $n = 512, m = 128$: (a) without deferred normalization; (b) with interleaved scaling and vector-matrix multiplication; (c) with deferred normalization.

6 Experiments and conclusions

Refer to [5] for Python code that demonstrates the mathematical equivalency of the optimizations presented in this paper. The speedup of FlashNorm is modest: We measured a throughput of 204 tokens per second for OpenELM-270M with 4-bit weight quantization using the MLX framework on an M1 MacBook Air. This throughput increases to only 225 tokens per second when we remove RMSNorm entirely. Therefore, the maximum possible speedup of any RMSNorm optimization is $\leq 10\%$ for this model.

For many applications, the main advantage of FlashNorm is simplification. This is similar to the simplifications we get from using RMSNorm over LayerNorm [14], and from Llama's removal of biases from all linear layers.

Future work should investigate which of the presented optimizations are applicable for training of LLMs.

Acknowledgements

We would like to thank Dmitry Belenko for helpful feedback on this work.

References

[1] Biao Zhang and Rico Sennrich. Root mean square layer normalization. October 2019. *arXiv:1910.07467*.

[2] 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*.

[3] 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*.

[4] Sachin Mehta, Mohammad Hossein Sekhavat, Qingqing Cao, Maxwell Horton, Yanzi Jin, Chenfan Sun, Iman Mirzadeh, Mahyar Najibi, Dmitry Belenko, Peter Zatloukal, and Mohammad Rastegari. OpenELM: An efficient language model family with open-source training and inference framework. April 2024. *arXiv:2404.14619*.

[5] OpenMachine. Transformer tricks. 2024. *Github repository*.

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

[7] Nils Graef. Transformer tricks: Precomputing the first layer. February 2024. *arXiv:2402.13388*.

[8] 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*.

[9] Tri Dao, Daniel Y Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. FlashAttention: Fast and memory-efficient exact attention with IO-awareness. May 2022. *arXiv:2205.14135*.

[10] Wikipedia. Rectifier (neural networks), 2024. Accessed June-2024.

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

[12] 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*.

[13] Alex Henry, Prudhvi Raj Dachapally, Shubham Pawar, and Yuxuan Chen. Query-key normalization for transformers. October 2020. *arXiv:2010.04245*.

[14] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. Layer Normalization. July 2016. *arXiv:1607.06450*.

Appendix

RMS with epsilon

Many implementations add a small epsilon ϵ to the RMS value to limit the resulting scaling factor $1/\text{RMS}(\vec{a})$ and to avoid division by 0 as follows:

$$\text{RMSe}(\vec{a}) = \sqrt{\epsilon + \frac{1}{n}\sum_{i=1}^{n} a_i^2} = \sqrt{\epsilon + (\text{RMS}(\vec{a}))^2}$$

$\text{RMSe}(\vec{a})$ can be used as a drop-in-replacement for RMS. The popular HuggingFace transformer library calls this epsilon `rms_norm_eps`, which is set to 10^{-5} for Llama3.

Eliminating $1/n$

This section details a small optimization that eliminates the constant term $1/n$ from the RMS calculation. First, we factor out $1/n$ as follows:

$$\text{RMS}(\vec{a}) = \sqrt{\frac{1}{n}\sum_{i=1}^{n} a_i^2} = \sqrt{\frac{1}{n}}\sqrt{\sum_{i=1}^{n} a_i^2} = \sqrt{\frac{1}{n}} \cdot \text{RSS}(\vec{a})$$

where we define the function RSS (Root Squared Sum) as

$$\text{RSS}(\vec{a}) = \sqrt{\sum_{i=1}^{n} a_i^2}$$

We can now merge the constant term into the normalization weights g_i as follows:

$$y_i = \frac{a_i}{\text{RMS}(\vec{a})} \cdot g_i = \frac{a_i}{\text{RSS}(\vec{a})}\sqrt{n} \cdot g_i = \frac{a_i}{\text{RSS}(\vec{a})} \cdot g_i^*$$

with new normalization weights $g_i^* = \sqrt{n} \cdot g_i$. These new normalization weights can now be merged with the weights $\mathbf{W}$ of the following linear layer as shown in the previous sections. This optimization also applies for the case where we add an epsilon as detailed in the previous section. In this case, we factor out $1/n$ as follows:

$$\text{RMSe}(\vec{a}) = \sqrt{\epsilon + \frac{1}{n}\sum_{i=1}^{n} a_i^2} = \sqrt{\frac{1}{n}\left(n\epsilon + \sum_{i=1}^{n} a_i^2\right)} = \sqrt{\frac{1}{n}} \cdot \text{RSSe}(\vec{a})$$

where we define the function $\mathrm{RSSe}(\vec{a})$ as

$$\mathrm{RSSe}(\vec{a}) = \sqrt{n\epsilon + \sum_{i=1}^{n} a_i^2}$$