Approximate attention: infinite context length with constant complexity per token [work in progress]

Nils Graef, Aarush Gupta, TBD
OpenMachine, San Francisco Bay Area, `info@openmachine.ai`

Abstract

This micro-paper [1] details a hybrid between exact and approximate dot-product attention for transformers that use rotary position embedding (RoPE) [2] (such as LLaMA, Mistral, and Gemma [3, 4, 5, 6]). The approximation is done by polynomials. For polynomial degree 1, this scheme is a hybrid between local attention [7] and linear transformer [8]. The goal is to reduce the computational complexity and memory size of attention for a given LLM without changing its weights (especially for long context lengths). See [9, 10] for code and more transformer tricks.

1 Scaled dot-product attention

Recall that the scaled dot-product attention of the vanilla transformer [11] is defined as

$$\mathbf{O} = \text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}}\right)\mathbf{V} \tag{1.1}$$

with the following variables:

- d_k is the dimension of keys and queries; d_v is the dimension of values.
- s is the current sequence length (aka context length).
- $\mathbf{Q}$ and $\mathbf{K}$ are matrices of dimension $s \times d_k$. $\mathbf{V}$ and output $\mathbf{O}$ are $s \times d_v$ matrices.

Let's make the following assumptions:

- For simplicity, we assume that $d = d_k = d_v$, which is the case for many popular LLMs.
- At the output of the transformer decoder stack, we only need to compute the last token of the output matrix $\mathbf{O}$, i.e. the last row of $\mathbf{O}$, which is the d-dimensional vector $\vec{o}$. Therefore, during the autoregressive next-token-generation process, we only need to compute the last row of $\mathbf{O}$ for each layer of the transformer stack.
- Similarly, we only need the last row of $\mathbf{Q}$, which is the d-dimensional vector $\vec{q}$.
- We also need matrices $\mathbf{K}$ and $\mathbf{V}$, which we organize (in a KV-cache) as s pairs of KV-vectors called $\vec{k}_i$ and $\vec{v}_i$.
- Furthermore, we assume that the scaling factor $1/\sqrt{d}$ is already merged into $\vec{q}_i$ or $\vec{k}_i$. For example, this scaling factor could be included into the weights (and biases) of the linear layer K by scaling its weights (and biases) by $1/\sqrt{d}$.

We can now express the output vector $\vec{o}$ as follows, where $\langle . \rangle$ denotes the dot-product:

$$\vec{o} = \sum_{i=1}^{s} \text{softmax}(\langle \vec{q}, \vec{k}_i \rangle)\vec{v}_i = \sum_{i=1}^{s} \frac{\exp(\langle \vec{q}, \vec{k}_i \rangle)}{\sum_{i=1}^{s} \exp(\langle \vec{q}, \vec{k}_i \rangle)}\vec{v}_i \tag{1.2}$$

2 Hybrid between exact and approximate attention with RoPE

RoPE is used by many popular LLMs such as LLaMA and Mistral. RoPE modifies the vectors $\vec{q}$ and $\vec{k}$, which results in scaling of the dot-products. This dot-product scaling decays with the relative distance between keys as illustrated in Figure 1, which is copied from the RoPE paper [2].

For example, only the 200 most recent keys have high scaling factors (closer to 1) while the older keys have scaling closer to 0. Therefore, we can split the key-value pairs (KV-pairs) into the following two groups, where N is 200 for example and $g = s - N$ is a shortcut variable:

Group	KV-pairs	Index range	Dot-product magnitudes	Calculation of exp(x)
1	N most recent	$i = (g+1), .., s$	large	exact
2	older than N steps	$i = 1, .., g$	small (e.g. [-1 .. +1])	polynomial approximation

Specifically, the first group contains the N most recent key-value pairs, and the second group contains the remaining g pairs. The N most recent KV-pairs (first group) are processed explicitly, while the remaining g KV-pairs are used for polynomial approximation. Derived from equation (1.2), below equation shows how this hybrid scheme combines these two groups:

$$\vec{o} = \sum_{i=1}^{s} \frac{\exp(\langle \vec{q}, \vec{k}_i \rangle)}{\sum_{i=1}^{s} \exp(\langle \vec{q}, \vec{k}_i \rangle)} \vec{v}_i = \left(\sum_{i=g+1}^{s} \exp(\langle \vec{q}, \vec{k}_i \rangle) \vec{v}_i + \sum_{i=1}^{g} \exp(\langle \vec{q}, \vec{k}_i \rangle) \vec{v}_i \right) \frac{1}{D_1 + D_2} = \frac{\vec{t} + \vec{u}}{D_1 + D_2} \tag{2.1}$$

where

$$\begin{aligned} \vec{t} &= \sum_{i=g+1}^{s} \exp(\langle \vec{q}, \vec{k}_i \rangle) \vec{v}_i \\ D_1 &= \sum_{i=g+1}^{s} \exp(\langle \vec{q}, \vec{k}_i \rangle) \\ \vec{u} &= \sum_{i=1}^{g} \exp(\langle \vec{q}, \vec{k}_i \rangle) \vec{v}_i \\ D_2 &= \sum_{i=1}^{g} \exp(\langle \vec{q}, \vec{k}_i \rangle) \end{aligned} \tag{2.2}$$

We can now compute vector $\vec{t}$ of equation (2.2) explicitly using the $\exp(x)$ function. And vector $\vec{u}$ can be approximated by polynomials. Note that the exact softmax attention is only calculated for the N first KV-pairs (e.g. $N = 200$), which reduces the size of the KV-cache to only 200 KV-pairs irrespectively of the actual context length s. The calculation of $\vec{t}$ is similar (or identical) to the sliding window attention (SWA) [7]. However, here we can set the window length to a smaller value (e.g. 200 instead of 4096 as used in Mistral-7B).

If the approximate attention uses a polynomial degree of 1, then this scheme is a hybrid of SWA (aka local attention) and linear attention [8] (which can be considered as a recurrent network [8]). Griffin [12] also mixes recurrences with local attention. However, here we only modify the attention mechanism (see Figure 2) without changing any weights. Our goal is post-training approximation of a given transformer without chaging its weights.

Better numerical stability. For higher numerical stability, many softmax implementations such as flash attention [13] subtract the maximum m across all softmax-arguments so that the arguments x of $\exp(x)$ are $x \leq 0$, and thus $\exp(x)$ is between 0 and 1. After subtracting the maximum m from the softmax arguments, we get $\vec{t}^*$ and D_1^* (instead of $\vec{t}$ and D_1) as follows (note that e^{-m} is identical to $\exp(-m)$):

$$\begin{aligned} \vec{t}^* &= \sum_{i=g+1}^{s} \exp(\langle \vec{q}, \vec{k}_i \rangle - m) \vec{v}_i = e^{-m} \cdot \vec{t} \\ D_1^* &= \sum_{i=g+1}^{s} \exp(\langle \vec{q}, \vec{k}_i \rangle - m) = e^{-m} \cdot D_1 \end{aligned} \tag{2.3}$$

We can now calculate output vector $\vec{o}$ by scaling $\vec{u}$ and D_2 by the same scaling factor e^{-m} as

$$\vec{o} = \frac{e^{-m}(\vec{t} + \vec{u})}{e^{-m}(D_1 + D_2)} = \frac{\vec{t}^* + e^{-m}\vec{u}}{D_1^* + e^{-m} D_2} \tag{2.4}$$

Block diagram. Figure 2 shows an implementation of this hybrid scheme:

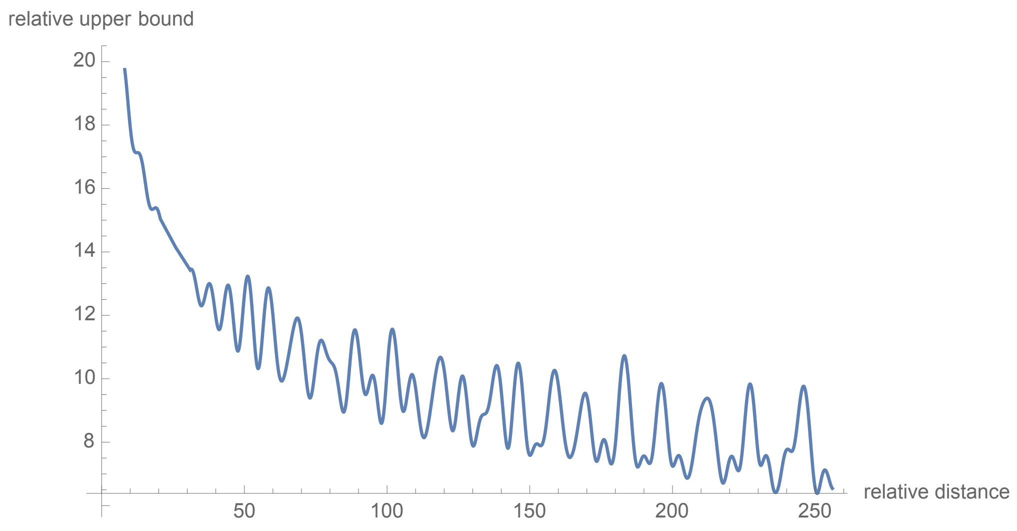

Figure 1: Long-term decay of RoPE, copied from [2]

- The block labeled SWA (sliding window attention) consists of a sliding window KV-cache of depth N (implemented by a circular buffer) followed by “exact attention.”
- The “exact attention” block calculates vector $\vec{t}$ from the N KV-pairs it receives from the KV-cache and from the latest query vector $\vec{q}_s$ (e.g. by using flash attention [13]). If the sequence length s is smaller than N, then it only processes s KV-pairs.
- The “approximate attention” block receives the N-th KV-pair from the KV-cache if s is greater than or equal to N. The approximate attention block uses this KV-pair and the latest query vector $\vec{q}_s$ to approximate $\vec{u}$ and D_2.
- Note that the query and key vectors ($\vec{q}$ and $\vec{k}$) shown in the figure already include the rotations performed by RoPE.
- Finally, adders and multipliers are used to compute the output vector $\vec{o}$ according to equation (2.4).

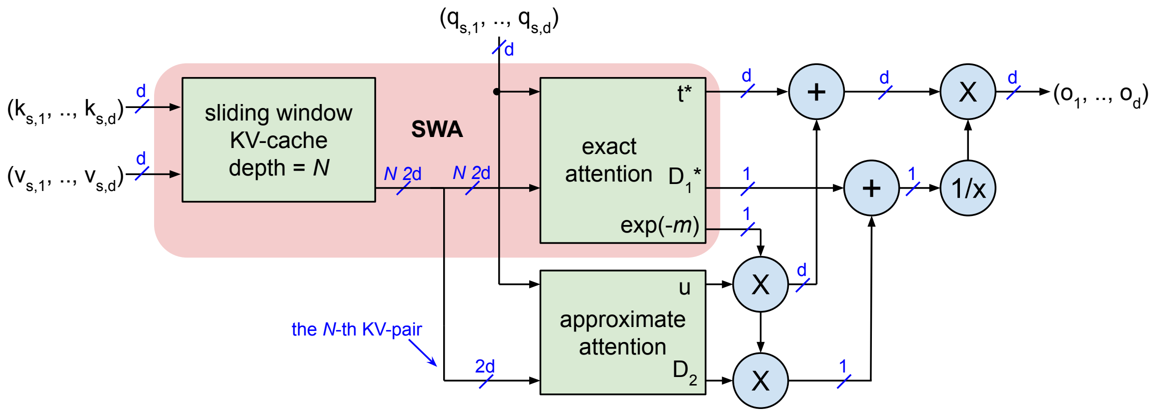

Figure 2: Block diagram of hybrid between exact and approximate attention

The rest of this paper details the polynomial approximation of $\vec{u}$ and D_2.

3 Polynomial approximation of $\exp(x)$

In general, the exponential function can be approximated by a polynomial of degree n as follows:

$$\exp(x) \approx a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n \tag{3.1}$$

Many computers (or maybe all computers) implement the floating-point exponential function by a combination of argument-reduction and polynomial approximation over a small range, see [14] for example.

The polynomial coefficinets a_i can be calculated by using Taylor, Chebyshev [15], or minimax approximation algorithms [16]. The polynomial approximation of $\exp(x)$ is only accurate for a small range, e.g. only for $x = -1 \dots 1$. If x falls outside this range, then the approximation error becomes large. Even worse, the polynomial might not be monotonic outside this range, which violates a key requirement for softmax to work properly as a way to generate a probability distribution. For the Taylor approximation, the table below lists the absolute and relative errors for polynomial degrees $n = 1$ to 4 for various values of x.

		error for $n=1$		error for $n=2$		error for $n=3$		error for $n=4$	
x	$\exp(x)$	abs.	rel.	abs.	rel .	abs.	rel.	abs.	rel.
0.25	1.28	0.03	**3%**	0.00	**0%**	0.00	**0%**	0.00	**0%**
0.5	1.65	0.15	9%	0.02	**1%**	0.00	**0%**	0.00	**0%**
1.0	2.72	0.72	26%	0.22	8%	0.05	**2%**	0.01	**0%**
1.5	4.48	1.98	44%	0.86	19%	0.29	7%	0.08	**2%**
2.0	7.39	4.39	59%	2.39	32%	1.06	14%	0.39	5%
2.5	12.18	8.68	71%	5.56	46%	2.95	24%	1.33	11%
3.0	20.09	16.09	80%	11.59	58%	7.09	35%	3.71	18%

The scaling factor $1/\sqrt{d}$ helps to reduce the range of the dot-product. As mentioned in the previous section, RoPE essentially scales down dot-products with increasing distance. And the recently proposed "clipped softmax" [17] is another method that limits the range of dot-products, especially for outliers that are often correlated with delimiter tokens such as [SEP], ".", and "," [17].

4 Approximation for case $d = 2$ with quadratic polynomial

For simplicity, let's assume that $d = 2$ and we approximate the exponential function by a polynomial of degree two (with coefficients $a_0 = a, a_1 = b, a_2 = c$) as follows:

$$\exp(x) \approx a + bx + cx^2 \tag{4.1}$$

Let's plug the two-dimensional dot-product $\langle \vec{q}, \vec{k}_i \rangle = q_1 k_{i,1} + q_2 k_{i,2}$ into equation (4.1):

$$\exp(q_1 k_{i,1} + q_2 k_{i,2}) \approx a + b(q_1 k_{i,1} + q_2 k_{i,2}) + c(q_1 k_{i,1} + q_2 k_{i,2})^2 \tag{4.2}$$

Plugging above into equation (2.2) yields

$$\begin{aligned} \vec{u} &\approx \sum_{i=1}^{g} \left(a + b(q_1 k_{i,1} + q_2 k_{i,2}) + c(q_1 k_{i,1} + q_2 k_{i,2})^2\right) \vec{v}_i \\ D_2 &\approx \sum_{i=1}^{g} a + b(q_1 k_{i,1} + q_2 k_{i,2}) + c(q_1 k_{i,1} + q_2 k_{i,2})^2 \end{aligned} \tag{4.3}$$

Similar to the linear transformer [8], we can now expand equation (4.3) as follows

$$\begin{aligned} \vec{u} &\approx a \sum_{i=1}^{g} \vec{v}_i + bq_1 \sum_{i=1}^{g} k_{i,1}\vec{v}_i + bq_2 \sum_{i=1}^{g} k_{i,2}\vec{v}_i + cq_1^2 \sum_{i=1}^{g} k_{i,1}^2 \vec{v}_i + 2cq_1q_2 \sum_{i=1}^{g} k_{i,1}k_{i,2}\vec{v}_i + cq_2^2 \sum_{i=1}^{g} k_{i,2}^2 \vec{v}_i \\ D_2 &\approx ag + bq_1 \sum_{i=1}^{g} k_{i,1} + bq_2 \sum_{i=1}^{g} k_{i,2} + cq_1^2 \sum_{i=1}^{g} k_{i,1}^2 + 2cq_1q_2 \sum_{i=1}^{g} k_{i,1}k_{i,2} + cq_2^2 \sum_{i=1}^{g} k_{i,2}^2 \end{aligned} \tag{4.4}$$

As was already shown in the linear transformer paper [8], the advantages of equation (4.4) include lower memory and computational complexities:

- We don't need to store the g KV-pairs anymore. Instead, we only need to store the sum terms such as $\sum_{i=1}^{g} k_{i,1}\vec{v}_i$, which we update in each iteration of the autoregressive next-token-generation process. Specifically, for $d = 2$ and quadratic polynomial approximation, we only need to store 6 KV-sums (each such sum-vector has dimension 2) and 5 sums (or sum-scalars) for the denominator D_2. So in total, we only need to store 12 + 5 = 17 values. Let's call this storage "KV-state" (as opposed to KV-cache).

- This also reduces the computational complexity significantly.
- More importantly, the memory size for the KV-state is constant; and the computational complexity per token is constant, too. Specifically, memory size and compute complexity per token are independent from the current sequence length s, which allows us to support infinitely long sequences.
- Furthermore, for applications where you want to store a conversation history (e.g. for a personal assistant or an AI companion), it would be more efficient (in terms of compute and perhaps also memory) to store the KV-state of that conversation history instead of storing and reprocessing a very long input sequence of tokens.

Figures 3 and 4 show block diagrams to approximate the output vector $\vec{u}$ and denominator D_2 according to equation (4.4). There are two steps:

- Step 1: Update the KV-state, i.e. all KV-sums (or sum-vectors and sum-scalars).
- Step 2: Calculate the output-vector $\vec{u}$ and denominator D_2 by multiplying elements of $\vec{q}$ with the KV-sums.

The blocks labeled "reg" in the figures denote registers that hold the KV-state. These registers, together with their preceding adders, form accumulators to implement the sums of equation (4.4). Alternatively, the KV-state can be implemented in SRAM or off-chip DRAM.

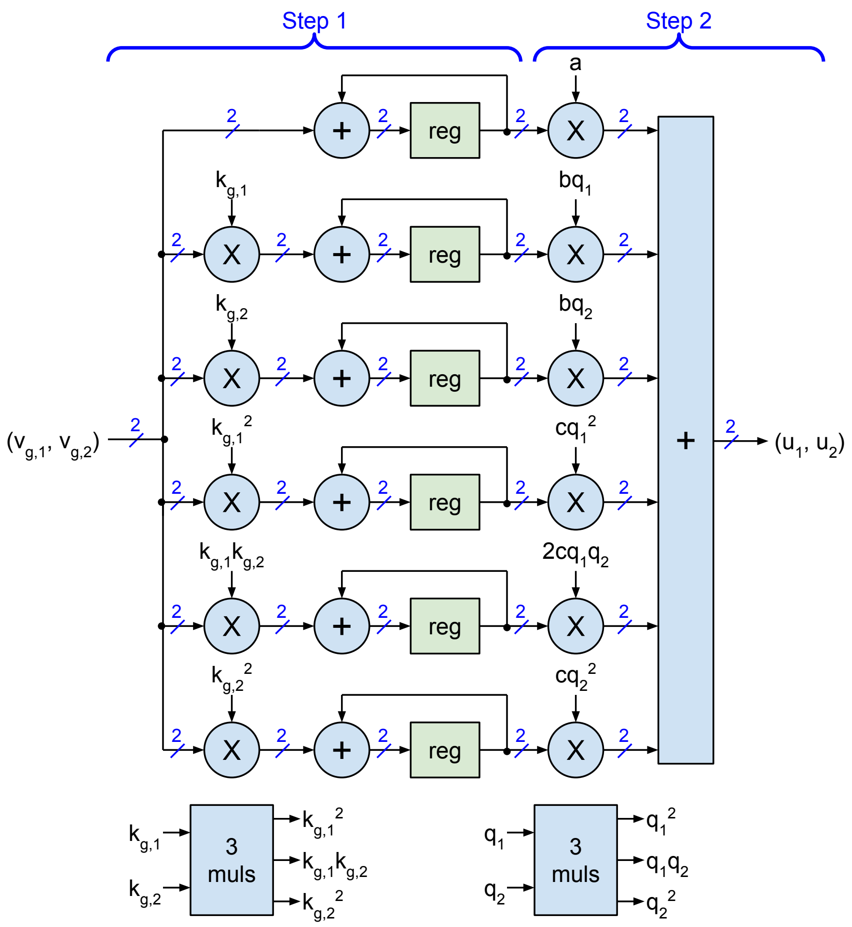

Figure 3: Block diagram for approximating vector $\vec{u}$ for $d = 2$

5 General case

The two-dimensional example ($d = 2$) is not very realistic: Many LLMs use $d = 64, 128,$ or 256. Therefore, the equations for arbitrary dimension d and arbitrary polynomial degree n are given in this section.

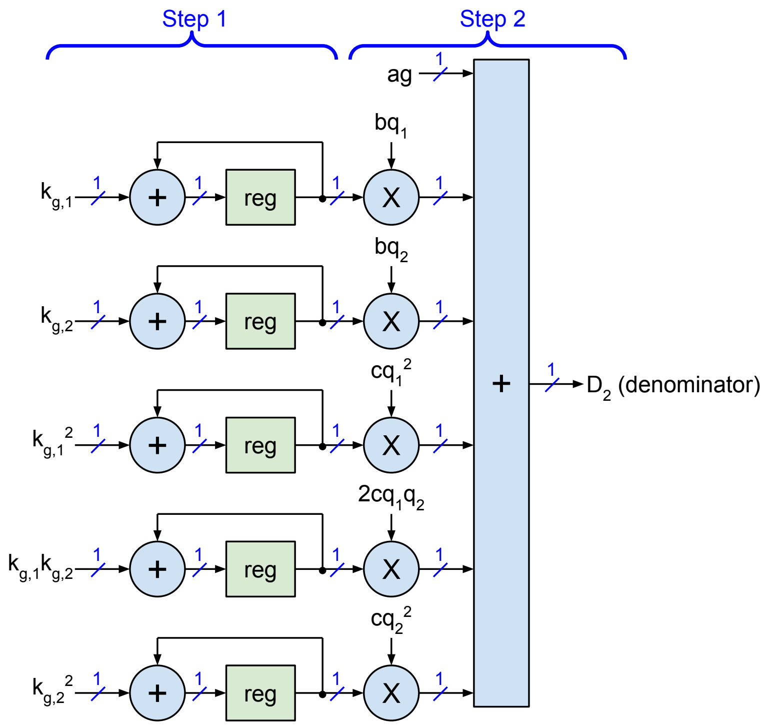

Figure 4: Block diagram for approximating denominator D_2 for $d = 2$

5.1 Arbitrary d with quadratic polynomial

Let's plug the d-dimensional dot-product $\langle \vec{q}, \vec{k}_i \rangle$ into equation (4.1):

$$\exp\left(\sum_{j=1}^{d} \vec{q}_j \vec{k}_{i,j}\right) \approx a + b \sum_{j=1}^{d} \vec{q}_j \vec{k}_{i,j} + c \left(\sum_{j=1}^{d} \vec{q}_j \vec{k}_{i,j}\right)^2 \tag{5.1}$$

Note that the last term of equation (5.1) can be expanded to $1 + 2 + ... + d$ terms, i.e. $\sum_{i=1}^{d} i = \frac{d(d+1)}{2}$ terms, as follows

$$\left(\sum_{j=1}^{d} \vec{q}_j \vec{k}_j\right)^2 = q_1^2 k_1^2 + 2q_1q_2k_1k_2 + \cdots + 2q_1q_dk_1k_d + q_2^2k_2^2 + 2q_2q_3k_2k_3 + \cdots + 2q_2q_dk_2k_d + \cdots + q_d^2k_d^2$$

which we will shorten as follows for use in equation (5.3) below:

$$\left(\sum_{j=1}^{d} \vec{q}_j \vec{k}_j\right)^2 = q_1^2 k_1^2 + 2q_1q_2k_1k_2 + \cdots + q_d^2k_d^2$$

Plugging equation (5.1) into equation (2.2) yields

$$\begin{aligned} \vec{u} &\approx \sum_{i=1}^{g} \left[a + b \sum_{j=1}^{d} \vec{q}_j \vec{k}_{i,j} + c \left(\sum_{j=1}^{d} \vec{q}_j \vec{k}_{i,j}\right)^2 \right] \vec{v}_i \\ D_2 &\approx \sum_{i=1}^{g} \left[a + b \sum_{j=1}^{d} \vec{q}_j \vec{k}_{i,j} + c \left(\sum_{j=1}^{d} \vec{q}_j \vec{k}_{i,j}\right)^2 \right] \end{aligned} \tag{5.2}$$

We can now expand equation (5.2) as follows

$$\vec{u} \approx a \sum_{i=1}^{g} \vec{v}_i + bq_1 \sum_{i=1}^{g} k_{i,1}\vec{v}_i + \cdots + bq_d \sum_{i=1}^{g} k_{i,d}\vec{v}_i + cq_1^2 \sum_{i=1}^{g} k_{i,1}^2\vec{v}_i + 2cq_1q_2 \sum_{i=1}^{g} k_{i,1}k_{i,2}\vec{v}_i + \cdots + cq_d^2 \sum_{i=1}^{g} k_{i,d}^2\vec{v}_i$$
$$D_2 \approx ag + bq_1 \sum_{i=1}^{g} k_{i,1} + \cdots + bq_d \sum_{i=1}^{g} k_{i,d} + cq_1^2 \sum_{i=1}^{g} k_{i,1}^2 + 2cq_1q_2 \sum_{i=1}^{g} k_{i,1}k_{i,2} + \cdots + cq_d^2 \sum_{i=1}^{g} k_{i,d}^2 \tag{5.3}$$

The total number of KV-sums of equation (5.3) is $1 + d + \frac{d(d+1)}{2}$.

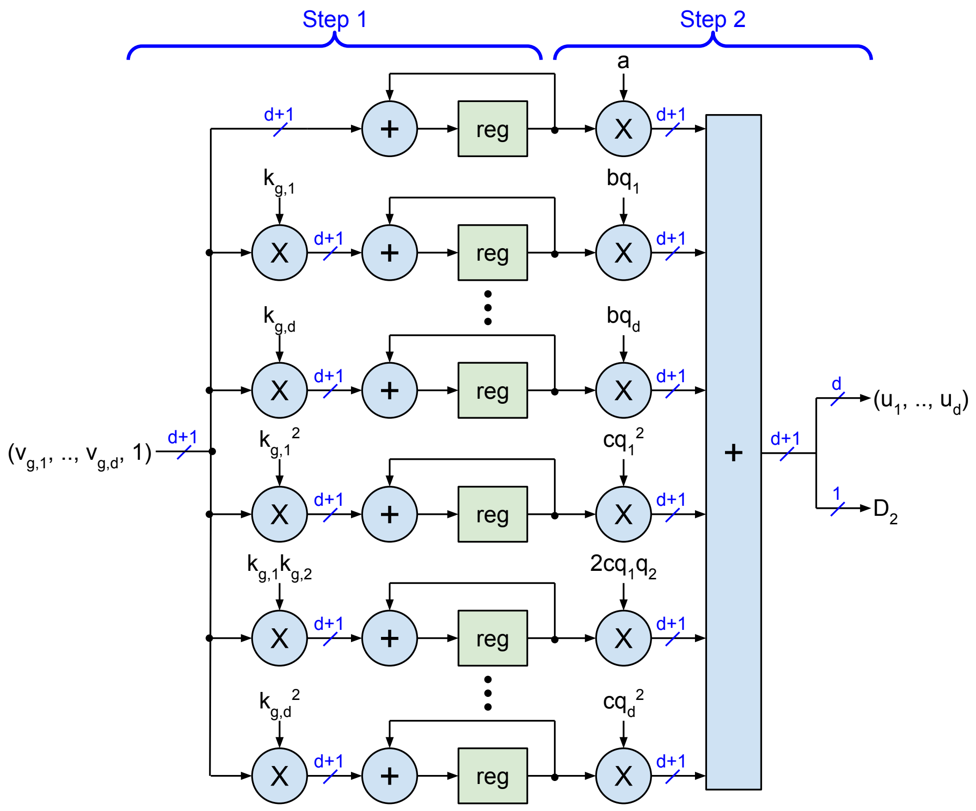

Figure 5: Block diagram for approximating vector $\vec{u}$ and denominator D_2

Figure 5 shows an implementation that approximates the output vector $\vec{u}$ and denominator D_2 according to equation (5.3): The arithmetic blocks for $\vec{u}$ and D_2 are merged into one block diagram by appending a constant 1 to the input vector $(v_{g,1}, .., v_{g,d})$ to form a new vector $(v_{g,1}, .., v_{g,d}, 1)$. This constant 1 is used for computing the denominator D_2.

5.2 Arbitrary d with cubic polynomial ($n = 3$)

Let's plug the d-dimensional dot-product $\langle \vec{q}, \vec{k}_i \rangle$ into equation (4.1) and let's add a cubic term with the polynomial coefficient e as follows

$$\exp\left(\sum_{j=1}^{d} \vec{q}_j \vec{k}_{i,j}\right) \approx a + b\sum_{j=1}^{d} \vec{q}_j \vec{k}_{i,j} + c\left(\sum_{j=1}^{d} \vec{q}_j \vec{k}_{i,j}\right)^2 + e\left(\sum_{j=1}^{d} \vec{q}_j \vec{k}_{i,j}\right)^3 \tag{5.4}$$

Plugging equation (5.4) into equation (2.2) yields

$$\begin{aligned}\vec{u} &\approx \sum_{i=1}^{g}\left[a + b\sum_{j=1}^{d}\vec{q}_j\vec{k}_{i,j} + c\left(\sum_{j=1}^{d}\vec{q}_j\vec{k}_{i,j}\right)^2 + e\left(\sum_{j=1}^{d}\vec{q}_j\vec{k}_{i,j}\right)^3\right]\vec{v}_i \\ D_2 &\approx \sum_{i=1}^{g}\left[a + b\sum_{j=1}^{d}\vec{q}_j\vec{k}_{i,j} + c\left(\sum_{j=1}^{d}\vec{q}_j\vec{k}_{i,j}\right)^2 + e\left(\sum_{j=1}^{d}\vec{q}_j\vec{k}_{i,j}\right)^3\right]\end{aligned} \tag{5.5}$$

Similar to equation (4.4), expanding equation (5.5) yields a series of KV-sums. To determine the total number of KV-sums, we use the multinomial theorem [18], which lets us expand a sum with d terms raised to the n-th power as follows:

$$(x_1 + x_2 + \cdots + x_d)^n = \sum_{k_1+k_2+\cdots+k_d=n;k_1,k_2,\ldots,k_d\geq 0}\binom{n}{k_1,k_2,\ldots,k_d}\prod_{t=1}^{d}x_t^{k_t} \tag{5.6}$$

where $\binom{n}{k_1,k_2,\ldots,k_d} = \frac{n!}{k_1!k_2!\cdots k_d!}$ is the multinomial coefficient. As stated in [18], "the sum is taken over all combinations of nonnegative integer indices k_1 through k_d such that the sum of all k_i is n." Furthermore, [18] gives the number of terms in a multinomial sum, $\#_{n,d}$, as

$$\#_{n,d} = \binom{d+n-1}{d-1} \tag{5.7}$$

With $n = 3$, we get $\#_{n=3,d} = \binom{d+2}{d-1} = \frac{(d+2)!}{(d-1)!3!} = \frac{d(d+1)(d+2)}{3!} = \frac{d(d+1)(d+2)}{6}$. Note that this number of terms is in addition to the terms we already have for linear and quadratic. So the total number of KV-sums is

$$\#_{total,n=3} = 1 + d + \frac{d(d+1)}{2} + \frac{d(d+1)(d+2)}{6} \tag{5.8}$$

5.3 Arbitrary d with arbitrary polynomial degree n

The previous sections can be extended to polynomials of degree four and higher. We only give the number of KV-sums needed for this. Per equation (5.7), the additional number of terms for degree n is given as

$$\#_{n,d} = \binom{d+n-1}{d-1} = \frac{(d+n-1)!}{(d-1)!n!} = \frac{1}{n!}\prod_{i=d}^{d+n-1} i = \frac{d(d+1)(d+2)\cdots(d+n-1)}{n!} \tag{5.9}$$

So the total number of KV-sums is:

$$\#_{total} = 1 + \sum_{j=1}^{n}\left(\frac{1}{j!}\prod_{i=d}^{d+j-1} i\right) = 1 + d + \frac{d(d+1)}{2} + \frac{d(d+1)(d+2)}{6} + \cdots + \frac{d(d+1)(d+2)\cdots(d+n-1)}{n!} \tag{5.10}$$

5.4 Sharing KV-pairs

Many LLMs use multi-query attention (MQA) [19] or grouped-query attention (GQA) [20], which reduces the size of the KV-cache significantly. It also reduces the sizes of the linear layers K and V, which reduces their computational complexity and weight storage size. The same savings are also applicable to the polynomial approximation scheme detailed here. Specifically, for MQA, one set of KV-state is shared among all heads (or group of heads for GQA). Additionally, we can also share the computations needed for step 1 (i.e. for updating the KV-state) among all heads (or group of heads).

6 Splitting dot-products into partial products

This section details a way to reduce the required range of the polynomial approximation: We can split the dot-product into two or more partial products and then approximate each partial product by a lower-degree polynomial.

6.1 Case $d = 2$ with two partial products and linear approximation

Let's first consider the simple case with $d = 2$ where we split the dot-product into two partial products and approximate each by a linear polynomial as

$$\exp(q_1 k_{i,1} + q_2 k_{i,2}) = \exp(q_1 k_{i,1}) \cdot exp(q_2 k_{i,2}) \approx (a + bq_1 k_{i,1})(a + bq_2 k_{i,2}) \tag{6.1}$$

Plugging above into equation (2.2) yields

$$\begin{aligned} \vec{u} &\approx \sum_{i=1}^{g} (a + bq_1 k_{i,1})(a + bq_2 k_{i,2}) \vec{v}_i \\ D_2 &\approx \sum_{i=1}^{g} (a + bq_1 k_{i,1})(a + bq_2 k_{i,2}) \end{aligned} \tag{6.2}$$

We can now expand equation (6.2) as follows

$$\begin{aligned} \vec{u} &\approx a^2 \sum_{i=1}^{g} \vec{v}_i + abq_1 \sum_{i=1}^{g} k_{i,1} \vec{v}_i + abq_2 \sum_{i=1}^{g} k_{i,2} \vec{v}_i + b^2 q_1 q_2 \sum_{i=1}^{g} k_{i,1} k_{i,2} \vec{v}_i \\ D_2 &\approx a^2 g + abq_1 \sum_{i=1}^{g} k_{i,1} + abq_2 \sum_{i=1}^{g} k_{i,2} + b^2 q_1 q_2 \sum_{i=1}^{g} k_{i,1} k_{i,2} \end{aligned} \tag{6.3}$$

Figure 6 shows a block diagram that approximates the vector $\vec{u}$ and denominator D_2 according to equation (6.3).

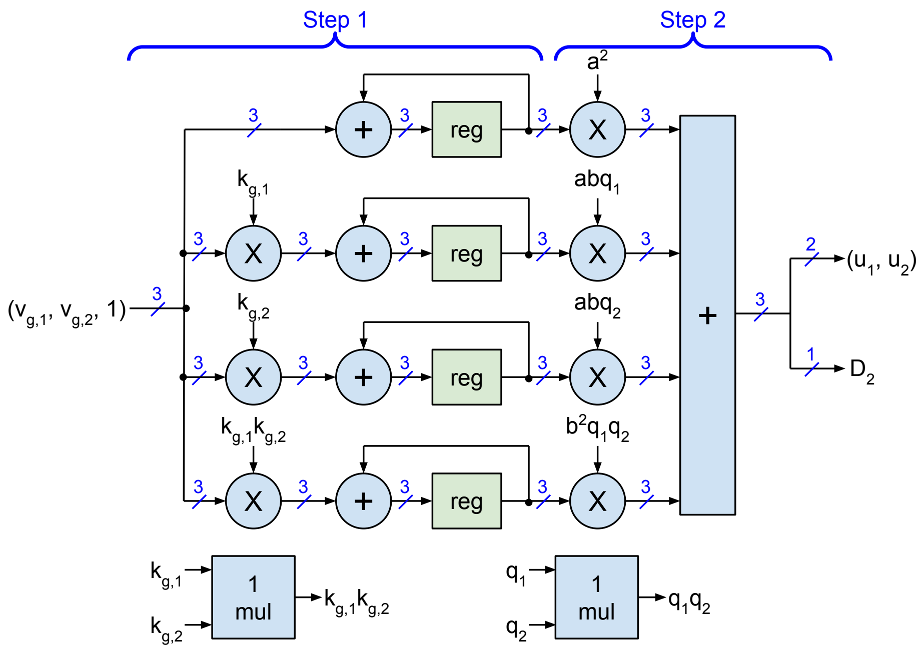

Figure 6: Block diagram for approximating vector $\vec{u}$ and denominator D_2 with two partial products for $d = 2$

6.2 General case and examples

This section derives $\#_{total}$ (the number of KV-sums) for arbitrary dimension d, number of partitions p, and polynomial degree n:

- **Polynomial degree** $n = 1$ (i.e. we approximate each partition by a linear polynomial): In this case, the number of KV-sums is $\#_{total,n=1} = (1 + \frac{d}{p})^p$.

- **Polynomial degree** $n = 2$ (i.e. we approximate each partition by a quadratic polynomial): In this case, the total number of KV-sums is $\#_{total,n=2} = (1 + \frac{d}{p} + \frac{d/p(d/p+1)}{2})^p$.
- **Arbitrary polynomial degree** n: In this general case, the total number of KV-sums equals equation (5.10) raised to the p-th power and where d is replaced by d/p as

$$\#_{total} = \left(1 + \sum_{j=1}^{n} \left(\frac{1}{j!} \prod_{i=d/p}^{d/p+j-1} i\right)\right)^p \tag{6.4}$$

For dimensions $d = 4, 8, 16, 32, 64, 128,$ and 256, the table below lists a few examples for p and n.

d	p	n	$\#_{total}$
4	1	1	5
		2	15
		3	35
	2	1	9
		2	36
		3	100
	4	1	16
		2	81
		3	256
8	1	1	9
		2	45
		3	165
	2	1	25
		2	225
		3	1,225
	4	1	81
		2	1,296
		3	10,000
	8	1	256
		2	6,561
		3	65,536

d	p	n	$\#_{total}$
16	1	1	17
		2	153
		3	969
	2	1	81
		2	2,025
		3	27,225
	4	1	625
		2	50,625
		3	1,500,625
	8	1	6,561
		2	1,679,616
	16	1	65,536
32	1	1	33
		2	561
		3	6,545
	2	1	289
		2	23,409
		3	938,961
	4	1	6,561
		2	4,100,625
	8	1	390,625

d	p	n	$\#_{total}$
64	1	1	65
		2	2,145
		3	47,905
	2	1	1,089
		2	314,721
	4	1	83,521
128	1	1	129
		2	8,385
		3	366,145
	2	1	4,225
		2	4,601,025
	4	1	1,185,921
256	1	1	257
		2	33,153
		3	2,862,209
	2	1	16,641

Here is an example for head dimension $d = 64$: TODO: the numbers in this example seem unrealistic, the dot-products have larger ranges and RoPE doesn't scale down by this much (RoPE scales down by about 2x rather than 15x, and oftentimes RoPE is applied to only half of the channels).

- Let's assume the outputs of the linear layers Q and K have range [-1 .. +1], so the dot-products would have range [-64 .. +64]. After scaling by $1/\sqrt{d} = 1/8$, the dot-products have range [-8 ... +8].
- Let's further assume that RoPE reduces the range by about 15x down to [-0.53 .. +0.53].
- **Option 1**: We can use polynomial degree $n = 2$ for this range and one partition ($p = 1$), which requires 2,145 KV-sum states (see table).
- **Option 2**: Alternatively, we can use 2 partitions ($p = 2$), where each partition now has a range of only [-0.27 .. +0.27]. Because of this smaller range, we can now pick a lower-degree polynomial $n = 1$, which requires only 1,089 KV-sums.

7 Experiments

This section is TBD. Implement this scheme for an LLM and measure the resulting perplexity to compare approximate versus exact attention. This is a post-training optimization and doesn't require any training.

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] 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*.

[4] 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, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. Llama 2: Open foundation and fine-tuned chat models. July 2023. *arXiv:2307.09288*.

[5] 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*.

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

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

[8] Angelos Katharopoulos, Apoorv Vyas, Nikolaos Pappas, and François Fleuret. Transformers are RNNs: Fast autoregressive transformers with linear attention. June 2020. *arXiv:2006.16236*. And code.

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

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

[11] 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*.

[12] Soham De, Samuel L Smith, Anushan Fernando, Aleksandar Botev, George Cristian-Muraru, Albert Gu, Ruba Haroun, Leonard Berrada, Yutian Chen, Srivatsan Srinivasan, Guillaume Desjardins, Arnaud Doucet, David Budden, Yee Whye Teh, Razvan Pascanu, Nando De Freitas, and Caglar Gulcehre. Griffin: Mixing gated linear recurrences with local attention for efficient language models. February 2024. *arXiv:2402.19427*.

[13] 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*.

[14] Sun Microsystems. exp.c library. 2004. URL `https://netlib.org/fdlibm/e_exp.c`.

[15] Wikipedia. Chebyshev polynomials, 2024. Accessed Feb-2024.

[16] Wikipedia. Minimax approximation algorithm, 2024. Accessed Feb-2024.

[17] Yelysei Bondarenko, Markus Nagel, and Tijmen Blankevoort. Quantizable transformers: Removing outliers by helping attention heads do nothing. June 2023. *arXiv:2306.12929*.

[18] Wikipedia. Multinomial theorem, 2024. Accessed Feb-2024.

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

[20] 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*.