Slim attention: cut your context memory in half without loss of accuracy — *K-cache is all you need for MHA*

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

Abstract

Slim attention shrinks the context memory size by 2x for transformer models with MHA (multi-head attention), which can speed up inference by up to 2x for long context. Slim attention is an exact, mathematically identical implementation of the standard attention mechanism and therefore doesn't compromise the model accuracy. In other words, slim attention losslessly compresses the context memory by a factor of 2.
For encoder-decoder transformers, the context memory size can be reduced even further. For the Whisper models for example, we can reduce the context memory by 8x, which can speed up token generation by 5x for batch size 64. And for rare cases where the MHA projections have non-square weight matrices, the cache can be reduced by a factor of 32 for the T5-11B model for example.

Fig. 1 illustrates how slim attention computes the value (V) projections from the key (K) projections in a mathematical equivalent way without any impact on the model accuracy. Therefore, we only need to store the keys in memory, instead of storing both keys and values (aka KV-cache). This reduces the size of the context memory (KV-cache) by half. Alternatively, slim attention can double the context window size without increasing context memory. However, calculating V from K on-the-fly requires additional compute, which we will discuss in the next sections.

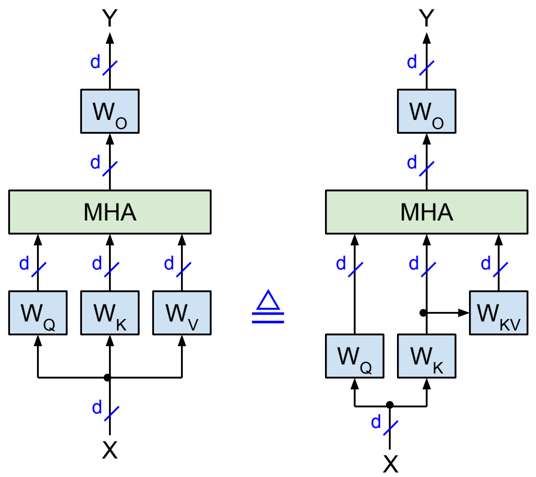

Figure 1: Mathematically identical implementations of multi-headed self-attention with square weight matrices of dimension $d \times d$. *Left*: vanilla version. *Right*: proposed version where values V are computed from keys K with $W_{KV} = W_K^{-1} W_V$. The $\triangleq$ symbol denotes mathematical identity.

Slim attention is applicable for transformers that use MHA (multi-head attention [1]) instead of MQA (multi-query attention [2]) or GQA (grouped query attention [3]), which includes LLMs such as CodeLlama-7B and Aya-23-35B; SLMs such as Phi-3-mini and SmolLM2-1.7B; VLMs (vision language models) such as LLAVA; audio-language models such as Qwen2-Audio-7B; and encoder-decoder transformer models such as Whisper [4] and T5 [5]. Table 1 lists various MHA transformer models ranging from 9 million to 35 billion parameters. The last column of Table 1 specifies the KV-cache size (in number of activations) for each model to support its maximum context length, where the KV-cache size equals $2hd_k \cdot$ layers $\cdot$ context_length.

Year	**publisher**	**model**	**params**	d	**layers**	h	d_k	**context length**	**context memory**
2024	Meta	CodeLlama-7B	7B	4,096	32		128	16k	4.3B
		CodeLlama-13B	13B	5,120	40				6.7B
	Google	CodeGemma-7B	8.5B	3,072	28	16	256	8k	1.9B
	Cohere	aya-23-35B	35B	8,192	40	64	128		5.4B
	HuggingFace	SmolLM2-1.7B	1.7B	2,048	24	32	64		0.8B
		SmolVLM	2.3B					16k	1.6B
	Microsoft	Phi-3-mini-128k	3.8B	3,072	32		96	128k	25.8B
		bitnet_b1_58-3B	3.3B	3,200	26	32	100	2k	0.3B
	Allen AI	OLMo-1B	1.3B	2,048	16		128	4k	0.3B
		OLMo-2-1124-7B	7.3B	4,096	32				1.1B
		OLMo-2-1124-13B	13.7B	5,120	40				1.7B
	Amazon	Cronos-Bolt-tiny	9M	256	4		64	0.5k	1M
		Cronos-Bolt-base	205M	768	12				9.4M
	Alibaba	Qwen2-Audio-7B	8.4B	4,096	32		128	8k	2.1B
	UW–Madison	llava-NeXT-video-7B	7.1B					4k	1.1B
2023		llava-v1.6-vicuna-13B	13.4B	5,120	40				1.7B
	LMSYS	Vicuna-7B-v1.5-16k	7B	4,096	32			16k	4.3B
		Vicuna-13B-v1.5-16k	13B	5,120	40				6.7B
2022	Google	Flan-T5-base	248M	768	12		64	0.5k	9.4M
		Flan-T5-XXL	11.3B	4,096	24	64			101M
	OpenAI	Whisper-tiny	38M	384	4	6		enc: 1500 dec: 448	6M
		Whisper-small	242M	768	12				36M
		Whisper-large-v3	1.5B	1,280	32	20			160M
2019		GPT-2 XL	1.6B	1,600	48	25		1k	157M

Table 1: Various transformers with MHA (instead of MQA or GQA) and their maximum KV-cache sizes (in number of activations) based on their respective maximum context length. h is the number of attention heads, d is the embedding dimension (aka hidden size), and d_k is the head dimension.

For long contexts, the KV-caches can be larger than the parameter memory. For example, assuming batch size 1 and 1 byte (int8 or fp8) per parameter and activation, the Phi-3-mini-128k model has a 3.8GB parameter memory and requires 25GB of memory capacity for its KV-cache to support a context length of 128K tokens. For a batch size of 16 for example, the KV-cache size grows to $16 \cdot 25\text{GB} = 400\text{GB}$. Therefore, memory bandwidth and capacity become a bottleneck for supporting large context windows.

For a memory bound system with batch size 1, the time to generate each token takes as long as it takes reading from memory all model parameters and KV-caches. Therefore, slim attention can speed up the token generation by close to 2x for long contexts. For the Phi-3-min-128k model with 3.8GB parameters for example, slim attention reduces the KV-cache size from 25GB to 12.5GB, which reduces the total memory from 28.8GB to 16.3GB, and speeds up the token generation by up to 1.8x for batch size 1 (the maximum speedup happens for the generation of the very last token of the 128K tokens). And for batch size 16 for example, the speedup is (400+3.8) / (200+3.8) = 2x.

1 Calculate V from K

The vanilla transformer [1] defines the self-attention Y of input X as follows, where h is the number of heads:

$$Q = XW_Q = \text{concat}\left(Q_1, \ldots, Q_h\right) \qquad K = XW_K = \text{concat}\left(K_1, \ldots, K_h\right) \tag{1.1}$$

$$V = XW_V = \text{concat}\left(V_1, \ldots, V_h\right) \qquad Y = \text{concat}\left(\text{head}_1, \ldots, \text{head}_h\right) W_O \tag{1.2}$$

$$\text{head}_i = \text{attention}\left(Q_i, K_i, V_i\right) = \text{softmax}\left(\frac{Q_i K_i^T}{\sqrt{d_k}}\right) V_i \tag{1.3}$$

with $W_Q = \text{concat}(W_{Q,1}, \dots, W_{Q,h})$, $W_K = \text{concat}(W_{K,1}, \dots, W_{K,h})$, and $W_V = \text{concat}(W_{V,1}, \dots, W_{V,h})$. The matrices Q, K, V, W_Q, W_K, W_V are split into h submatrices, one for each attention head. Input X, output Y, queries Q, keys K, and values V are $n \times d$ matrices, where n is the current sequence length (in tokens) and $d = d_{model}$ is the dimension of the embeddings.

For MHA, the weight matrices W_K and W_V are usually square matrices $d \times d$, which allows us to calculate V from K as follows: Refactoring equation (1.1) as $X = KW_K^{-1}$ lets us reconstruct X from K, which we can then plug into equation (1.2) to get

$$V = K(W_K^{-1} W_V) = KW_{KV} \text{ and } V_i = KW_{KV,i} \text{ where } W_{KV} = \text{concat}(W_{KV,1}, \dots, W_{KV,h}), \tag{1.4}$$

and $W_{KV,i}$ are $d \times d_v$ matrices. Fig. 1 illustrates the modified attention mechanism where V is calculated from K according to equation (1.4). For inference, $W_{KV} = W_K^{-1} W_V$ can be precomputed offline and stored in the parameter file instead of W_V. This requires that W_K is invertible (i.e. non-singular). In general, any square matrix can be inverted if its determinant is non-zero. It's extremely unlikely that a random matrix has a determinant that is exactly 0. In fact, all weight matrices we encountered are invertible.

2 K-cache is all you need

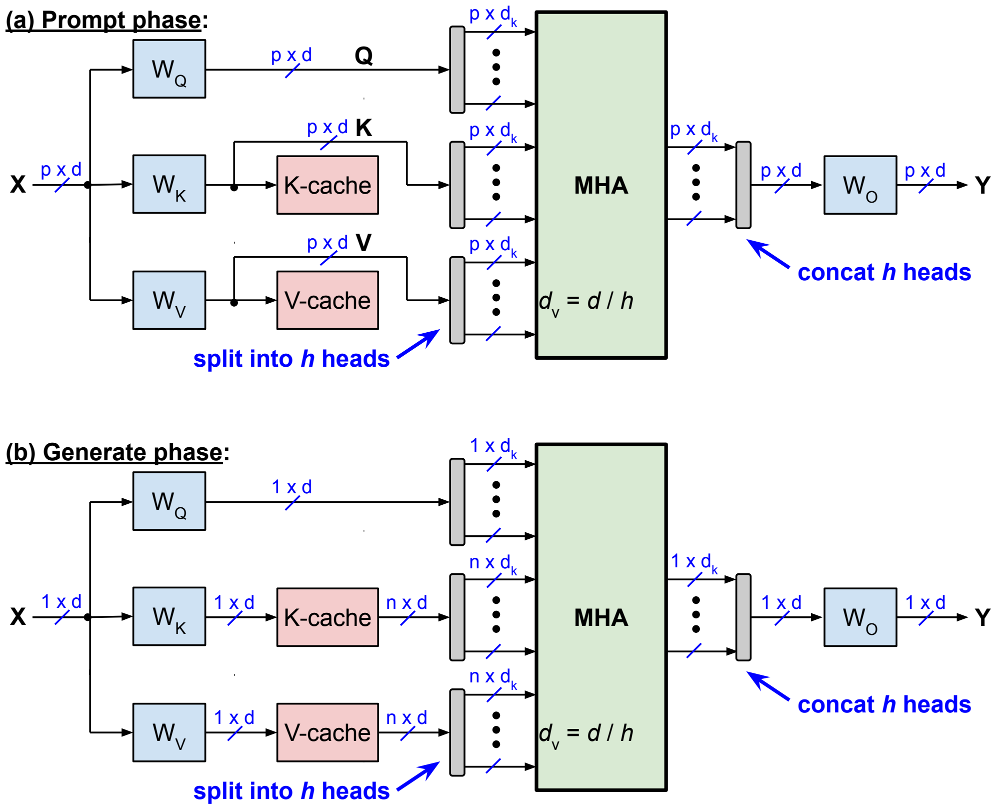

Figure 2: Standard MHA with KV-cache during (a) prompt phase and (b) generate phase.

Inference consists of the following two phases, which are illustrated in Fig. 2 for the vanilla MHA with KV-cache, where p is the number of prompt-tokens (aka input-tokens) and n is the total number of current tokens including input-tokens and generated tokens, so $n = p + 1, \dots, n_{max}$ and n_{max} is the context window length:

- During the **prompt phase** (aka prefill phase), all p input-tokens are batched up and processed in parallel. In this phase, the K and V projections are stored in the KV-cache.

- During the **generate phase** (aka decoding phase), each output token is generated sequentially (aka autoregressively). For each iteration of the generate phase, only one K and V vector is calculated and stored in the KV-cache, while all the previously stored KV-vectors are read from the cache.

Fig. 3 illustrates slim attention, which only has a K-cache because V is now calculated from K. Plugging equation (1.4) into (1.3) yields

$$\text{head}_i = \text{softmax}\left(\frac{Q_i K_i^T}{\sqrt{d_k}}\right) K W_{KV,i} = \underbrace{\text{softmax}\left(\frac{Q_i K_i^T}{\sqrt{d_k}}\right)[K W_{KV,i}]}_{\text{Option 1}} = \underbrace{\left[\text{softmax}\left(\frac{Q_i K_i^T}{\sqrt{d_k}}\right) K\right] W_{KV,i}}_{\text{Option 2}} \quad (2.1)$$

Equation (2.1) can be computed in two different ways:

- Option 1: Compute $V_i = K W_{KV,i}$ first, and then multiply it with the softmax attention scores. This option is used by Fig. 3(a) and 3(b).
- Option 2: First multiply softmax$(\cdot)$ with K, and then multiply the result by $W_{KV,i}$. This option is illustrated in Fig. 3(c). During the generate phase, this option has lower compute complexity than option 1.

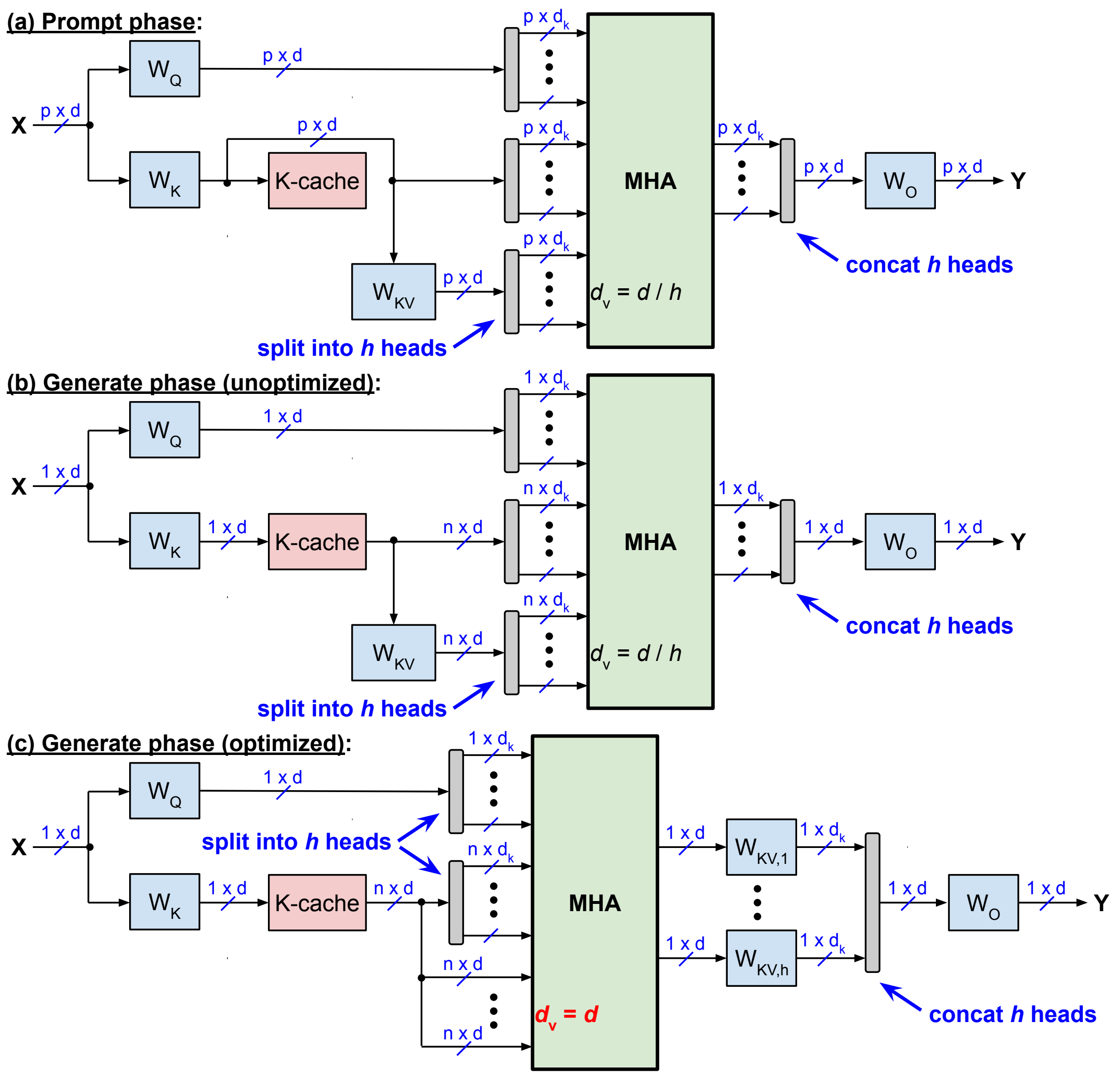

Figure 3: Proposed slim attention without V-cache during (a) prompt phase; (b) unoptimized and (c) optimized generate phase.

During the prompt phase, Fig. 3(a) has the exact same computational complexity as the vanilla scheme shown in Fig. 2(a). However, during the generate phase, the proposed scheme has a slightly higher complexity than the vanilla scheme (that's the cost of eliminating the V-cache). Using big-O notation, Table 2 specifies the complexity per token per layer during the generate phase for batch size 1. The columns labeled "compute", "reads", and "intensity" specify the computational complexity, the number of memory reads, and the arithmetic intensity, resp. We define the arithmetic intensity here as big-O complexity per each activation or parameter read from memory. Specifically, the projection complexity includes calculating XW_Q, XW_K, XW_V, and the W_O linear layer. And the memory reads for projection include reading all four weight matrices; while the memory reads of the MHA include reading the K-cache and the V-cache (only for Fig. 2(b)).

	Projection complexity			**MHA complexity**		
	compute	**reads**	**intensity**	**compute**	**reads**	**intensity**
Vanilla, see Fig. 2(b)	$O(4d^2)$	$4d^2$	1	$O(2nd)$	$2nd$	1
Unoptimized, Fig. 3(b)	$O((n+3)d^2)$	$4d^2$	$(n+3)/4$	$O(2nd)$	nd	2
Optimized, Fig. 3(c)	$O(4d^2)$	$4d^2$	1	$O((h+1)nd)$	nd	$h+1$

Table 2: Complexity per token per layer during the generate phase for batch size 1

Table 3 shows the arithmetic intensity (now defined as Ops per memory byte) of various SoCs, TPUs, and GPUs, which vary from 70 to 583. A system is memory bound (i.e. limited by memory bandwidth) if the arithmetic intensity of the executed program is below the chip's arithmetic intensity. Here, the arithmetic intensity of slim attention is 1 or $h+1$, see Table 2, where h is the number of attention heads, which ranges between 16 and 64 for the models listed in Table 1. So the arithmetic intensity (up to 65 for $h=64$) is usually less than the system's intensity, which means that the system is still memory bound during the token generation phase. Therefore, slim attention speeds up the processing by up to 2x as it reduces the context memory reads by half. Furthermore, slim attention enables processing all heads in parallel as a single matrix-matrix multiplication instead of multiple vector-matrix multiplications, which is usually more efficient and faster on many machines.

Chip	**TOPS (int8)**	**Theoretical memory bandwidth (GB/s)**	**Arithmetic intensity (Ops per byte)**
Rockchip RK3588	6	19	316
Apple A18	35	60	583
Apple M4 Max	38	546	70
Google TPU v4	275	1,200	229
Google TPU v5p	918	2,765	332
NVIDIA H200	1,980	4,800	413
NVIDIA B200	4,500	8,000	563

Table 3: 8-bit integer TOPS (tera-operations-per-second), memory bandwidth, and arithmetic intensity of popular SoCs, TPUs, and GPUs

3 Taking advantage of softmax sparsities

In this section we describe how we can take advantage of softmax sparsities (i.e. sparsities in the attention scores) to reduce the computational complexity of the attention blocks. In some applications, many attention scores are 0 or close to zero. For those attention scores (i.e. attention scores smaller than a threshold), we can simply skip the corresponding V vector, i.e. we don't have to add those skipped vectors to the weighted sum of V-vectors. This reduces the complexity of calculating the weighted sum of V-vectors. For example, for a sparsity factor $S = 0.8$ (i.e. 80% of scores are 0), the complexity is reduced by 5x, or by $\frac{1}{1-S}$ in general.

By the way, taking advantage of these softmax sparsities is also applicable for systems with KV-cache where V is not computed from K. In this case, skipping V vectors with zero scores means that we don't have to read those V-vectors from the KV-cache, which speeds up the autoregressive generate phase for memory bound systems. However, this will never speed it up more than slim attention's removal of the entire V-cache. Furthermore, for MQA and GQA, each V-vector is shared among multiple (e.g. 4 or more) queries so we can only skip reading a V-vector from memory if all 4 (or more) attention scores are zero for this shared V-vector, which reduces the savings significantly. For example, if the V-vectors are shared among 4 queries and the attention scores have sparsity $S = 0.8$, then the probability of all four queries being 0 is only $S^4 = 0.41$, so we can only skip 41% of the V-vectors.

4 Support for RoPE

Many transformers nowadays use RoPE (rotary positional embedding) [6], which applies positional encoding to the Q and K projections, but not the V projections. In general, RoPE can be applied to the K projections either before storing them in K-cache or after reading them from K-cache. The former is preferred because of lower computational complexity during the generate phase (so that each K-vector is RoPE'd only once instead of multiple times). However, if the RoPE'd keys are stored in K-cache, then we first need to un-RoPE them before we can compute V from K. The following details two options to support RoPE.

Option 1 is for the case where we don't take advantage of softmax sparsities. In this case, we apply RoPE to the K-vectors after reading them from K-cache during the generate phase. That way we can use the raw K-vectors for computing V from K.

Option 2 is for the case where we take advantage of softmax sparsities as detailed in the previous section. In this case, RoPE is applied to the K-vectors before writing them into the K-cache. And when they are read from K-cache during the generate phase, then we have to revert (or undo) the RoPE-encoding before we can use the K-vectors to compute the V-vectors (i.e. multiplying the K-vectors with the attention scores). However, we only need to do this for a portion of the K-vectors, depending on the sparsity factor S. For example, for $S = 0.8$, we only need to revert the RoPE-encoding for 20% of the K-vectors. The RoPE encoding can be reverted (aka RoPE-decoding) by simply performing a rotation in the opposite direction by the same amount as shown below for the 2D case.

RoPE encoding:

$$\begin{aligned} y_1 &= x_1 \cos m\theta + x_2 \sin m\theta \\ y_2 &= -x_1 \sin m\theta + x_2 \cos m\theta \end{aligned}$$

RoPE decoding:

$$\begin{aligned} x_1 &= y_1 \cos m\theta - y_2 \sin m\theta \\ x_2 &= y_1 \sin m\theta + y_2 \cos m\theta \end{aligned}$$

Note that the RoPE decoding uses the same trigonometric coefficients (such as $\cos m\theta$) as the RoPE encoding. Therefore, we only need one look-up table that can be used for both RoPE encoding and decoding.

5 Support for bias

Since PaLM's removal of bias terms from all its projection layers [7], most transformer models nowadays do the same. However, some models are still using biases today (especially older models that are still relevant today such as Whisper). In this section, we briefly discuss how projection layers with bias can be supported. We show how the biases of two of the four attention projection layers can be eliminated in a mathematically equivalent way.

Removal of the bias in the V projection layer: This bias can be combined with the bias of the output projection layer as follows. Recall that all value vectors v_i plus their constant bias b are multiplied by the attention scores s_i (i.e. the softmax outputs) and summed up, such as

$$\sum_{i=1}^{n} s_i(v_i + b) = \sum_{i=1}^{n} s_i v_i + \sum_{i=1}^{n} s_i b = \sum_{i=1}^{n} s_i v_i + b$$

The last equal sign holds because the sum over all attention-scores s_i is always 1 as per softmax definition (because the softmax generates a probability distribution that always adds up to 1). We can now merge the bias b with bias c of the preceding output projection layer (O) as follows: $y = (x + b)W_O + c = xW_O + (bW_O + c) = xW_O + c'$, with the new bias $c' = bW_O + c$. This new bias vector c' can be computed offline, before inference time. Or simply remove the V-bias already during training.

Removal of the bias in the K projection layer: The bias of the K projection cancels out due to the constant invariance of the softmax function. For example, say we have 2-dimensional heads, then the dot-product p between query vector $q = (q_1 + b_1, q_2 + b_2)$ with bias b and key vector $k = (k_1 + c_1, k_2 + c_2)$ with bias c is as follows:

$$\begin{aligned} p &= (q_1 + b_1)(k_1 + c_1) + (q_2 + b_2)(k_2 + c_2) = [q_1k_1 + q_2k_2] + [q_1c_1 + q_2c_2] + [b_1k_1 + b_2k_2] + [b_1c_1 + b_2c_2] \\ &= q_1k_1 + q_2k_2 + f(q) + b_1k_1 + b_2k_2 + \text{constant}, \end{aligned}$$

where $f(q) = q_1c_1 + q_2c_2$ is a function of the query vector only; and "constant" is a constant that only depends on the two biases b and c. Now recall that the softmax function doesn't change if a constant is added to all its arguments. Because all arguments of the attention softmax use the same single query vector q, $f(q)$ is the same for all arguments and is therefore constant and can be removed from all softmax arguments. As a result, we can remove the entire bias vector c from the keys. But we still need the bias vector b for the queries. However, this assumes that there is no RoPE applied between the projections and the dot-product calculation, which is fortunately the case for Whisper for example.

6 Support for non-square weight matrices

Some transformers with MHA use non-square weight matrices for their K and V projections. Specifically, these models do not satisfy $d = d_kh$, where h is the number of heads. The table below shows three exemplary models where $e = d_kh > d$. Let's also define the aspect ratio r as $r = e/d$. For example, Google's T5-11B model has a large aspect ratio of $r = 16$, i.e. dimension e is 16 times larger than d.

Model	d	d_k	h	$e = d_kh$	aspect ratio $r = e/d$
CodeGemma-7B	3,072	256	16	4,096	1.3
T5-3B	1,024	128	32	4,096	4
T5-11B	1,024	128	128	16,384	16

There are two options to reduce the KV-cache by 2x or more, which are compared in the table below and summarized as follows:

- **Option 1**: Because the K weight matrix is non-square, inverting this matrix is not straight forward. And the resulting W_{KV} matrix is now an $e \times e$ matrix, which has r-times more parameters than the $d \times e$ matrix W_V.
- **Option 2**: Instead of storing V in cache and then calculating V from K, we can store the smaller d-element vectors X before the projection and then on-the-fly calculate both projections (V and K) from X. The cache is now r-times smaller than option 1, and $2r$ times smaller than the baseline, for example 32 times smaller for the T5-11B model. However, this comes at a slightly higher computational cost.

	Baseline	**Option 1**	**Option 2**
Cache reduction factor	1	2	$2r$
Size of W_V or W_{KV}	de	e^2 (r-times larger)	de
Computational complexity	baseline	higher	even higher
Support for RoPE?	Yes	Yes	No

Option 1: The standard matrix inverse is defined only for square matrices, and the matrix inversion functions in NumPy and SciPy are limited to such matrices. We want to compute the inverse of W_K, which is a $d \times e$ matrix, such that $W_KW_K^{-1} = I$, where I is the identity matrix, and $e > d$ and W_K^{-1} is the so-called right inverse of W_K. We compute W_K^{-1} by using a trick that inverts the term $W_KW_K^T$ instead of W_K as follows:

$$I = W_KW_K^T(W_KW_K^T)^{-1} = W_KW_K^{-1}$$

In the equation above, everything right of W_K has to be the inverse of W_K. Therefore, $W_K^{-1} = W_K^T(W_KW_K^T)^{-1}$. We can now use the matrix inversion function of NumPy to compute the inverse of the term $W_KW_K^T$, which is a square $d \times d$ matrix. Now we can calculate $W_{KV} = W_K^{-1}W_V$. However, storing W_{KV} instead of the original W_V takes r times more space in memory, which is an issue for large aspect-ratios r.

Option 2 caches the X-matrix instead of KV or just K, where the X-matrix contains the input activations of the attention layer (before the projections). Recomputing all K-vectors from X by multiplying X with weight matrix W_K would require $2nde$ operations and would be very expensive. A lower complexity option is illustrated in Fig. 4, which is similar to the trick illustrated in Fig. 3(c). Recall that for the i-th head ($i = 1, \ldots, h$), the softmax argument (without

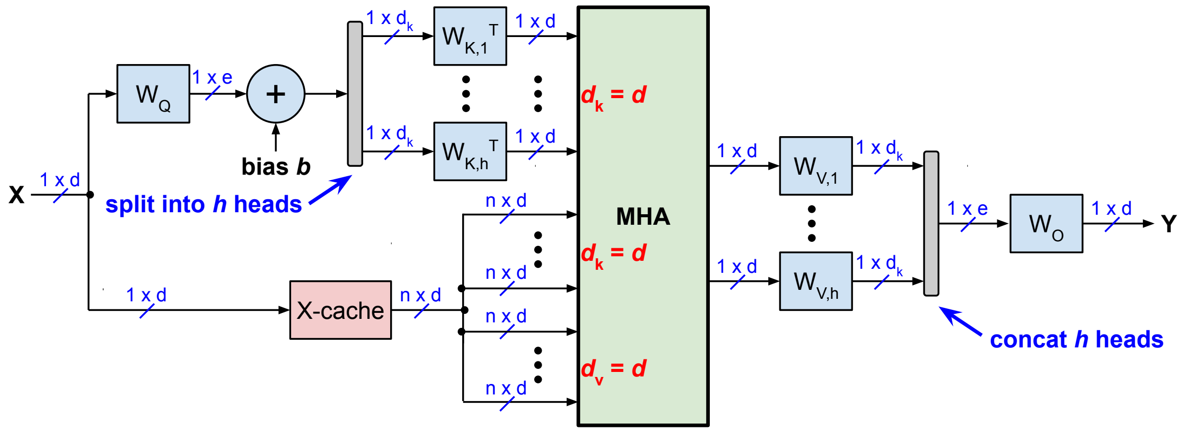

Figure 4: Slim attention with X-cache (instead of KV or V-cache) for the generate-phase of transformers with non-square weight matrices with $e > d$.

the scaling factor $\sqrt{d_k}$) is $A_i = Q_i K_i^T$, where $Q_i = XW_{Q,i}$ and $K_i = XW_{K,i}$. For the generate phase, there is only one input vector x_n for the query, but there are n input vectors X for the key and value projections. We can take advantage of this and modify A as follows (which uses the trick $(BC)^T = C^T B^T$ for transposing the product of arbitrary matrices B and C):

$$A_i = Q_i K_i^T = x_n W_{Q,i}(XW_{K,i})^T = (x_n W_{Q,i} W_{K,i}^T) X^T$$

For each iteration of the generate phase, we now have to calculate the term $x_n W_{Q,i} W_{K,i}^T$ only once for each attention head, which is independent of the sequence length. Calculating this term involves multiplying the d-dimensional vector x_n with the $d \times d_k$ matrix $W_{Q,i}$ and the $d_k \times d$ matrix $W_{K,i}^T$, which requires $2de$ multiplications for the h heads, so $4de$ operations in total (where we count a multiply-add operation as 2 operations).

This scheme also works for projection layers with biases (as used by the Whisper models for example). Recall from the previous section that we can eliminate the biases from the key and value projections, but not from the query projection. Adding a constant query bias-vector b to the equation above is straightforward and also illustrated in Fig. 4:

$$A_i = Q_i K_i^T = (x_n W_{Q,i} + b)(XW_{K,i})^T = ((x_n W_{Q,i} + b) W_{K,i}^T) X^T$$

However, this scheme doesn't work if there is a positional encoding such as RoPE located between the projection layers and the dot-product calculation. But option 2 fully supports other relative position encoding (PE) schemes such as RPE of the T5 model, Alibi, Kerple and FIRE [8] which add a variable bias to the softmax arguments (instead of modifying the queries and keys before the dot-product calculation). See for example the FIRE paper [8], which shows that FIRE and even NoPE can outperform RoPE for long context.

7 Support for encoder-decoder transformers

In general, calculating K from V is not only possible for self-attention (see Fig. 1) but also for cross-attention. In this section, we present two context memory options for encoder-decoder transformers such as Whisper (speech-to-text), language translation models such as Google's T5, and time series forecasting models such as Amazon's Chronos models. One option is not limited to MHA only, but is also applicable to MQA and GQA. The table below compares the options, which are summarized as follows:

- The **baseline** implementation uses complete KV-caches for both self-attention and cross-attention of the decoder stack. We refer to these caches as self KV-cache and cross KV-cache, resp.
- **Option 1** is an optimized implementation where the V-caches for both self-attention and cross-attention are eliminated, which reduces the total cache size by 2x.

	Baseline	**Option 1**	**Option 2**
Self KV-cache size	100%	50%	50%
Cross KV-cache size	100%	50%	0
Need to read cross W_K and W_V during the generate phase?	No	W_{KV} only	W_K and W_V
Support for RoPE?	Yes	Yes	No
Complexity of cross phase	full	half	0

- **Option 2** eliminates the entire cross KV-cache and also eliminates the V-cache of the self-attention.

The **baseline** implementation consists of the following three phases:

- During the **prompt phase**, only the encoder is active. All p prompt tokens are batched up and processed in parallel. This phase is identical to the prompt phase of a decoder-only transformer albeit without a causal mask (and is also identical to the entire inference of an encoder-only transformer such as BERT).
- During the **cross phase**, we take the output of the encoder (which is a $p \times d$ matrix) and precompute the KV projections for the decoder's cross attention and store them in cross context memory (aka cross KV-cache).
- The **generate phase** is similar to the generate phase of a decoder-only transformer with the following difference: There is a cross-attention block for each layer, which reads keys and values from the cross KV-cache. In addition, the self-attention blocks have their own self KV-cache (which is not the same as the cross KV-cache).

Option 1 calculates V from K for both self-attention and cross-attention of the decoder stack (note that the encoder stack doesn't have a KV-cache because it is not autoregressive). This requires reading the cross W_{KV} parameters from memory during the generate phase. So if the W_{KV} matrices are larger than the cross V-cache, then calculating V from K for the cross-attention doesn't make sense for batch size 1 (but for larger batch sizes). For the Whisper models for example, the cross V-cache is always larger than the W_{KV} matrices, because the number of encoder tokens p is always 1500. And for batch sizes B larger than 1, the reading of the cross W_{KV} parameters can be amortized among the B inferences.

Option 2 efficiently recomputes the cross KV projections from the encoder output instead of storing them in the cross KV-cache as follows:

- At the end of the prompt phase, the encoder output is a $p \times d$ matrix, which is then used by all layers of the decoder stack. We call this $p \times d$ matrix E the encoder-cache (or E-cache) and we assume that it resides in on-chip SRAM (such as L2 cache) at the end of the prompt phase, because it's usually very small (less than 1 million values for Whisper tiny and base for example).
- Recomputing all K-vectors could be done by multiplying the $p \times d$ matrix E with the $d \times d$ weight matrix W_K, which requires $2pd^2$ operations and would be very expensive. A lower complexity option is illustrated in Fig. 5, which is similar to Fig. 4. The main difference is that all cross attention layers share the same E-cache. And on many machines, this E-cache might fit into on-chip SRAM so that it doesn't need to be re-read for each layer during the generate phase.
- As with Fig. 4 in the previous section, Fig. 5 doesn't support RoPE, but other (and potentially better) relative PE schemes such as RPE, FIRE, and NoPE.
- This scheme doesn't calculate V from K and therefore is not limited to MHA only or to projection matrices that can be inverted.
- Similar to option 1 (and unlike the baseline), the cross W_V and W_K parameters need to be read from memory for each generated token during the generate phase. Therefore for batch size 1, this scheme only makes sense if the KV-cache is larger than the number of cross W_V and W_K parameters, which is the case for all Whispers models (because they use $p = 1500$ prompt-tokens, and d of the largest Whisper model is smaller than 1500). And for batch sizes larger than 1, this option usually makes sense because the parameter reads are amortized among all the inferences of the batch.

Time-to-first-token (TTFT): Options 1 and 2 speed up or eliminate the cross phase. Specifically, option 1 speeds up the cross phase by 2x. And option 2 completely eliminates the cross phase. This speeds up the time-to-first-token latency (TTFT). And in certain cases, the overall compute complexity can be reduced compared to the baseline. For cases where the number of encoder tokens is larger than the decoder tokens (such as for Whisper or for text summarization) and for large d dimensions, these options can reduce the overall complexity.

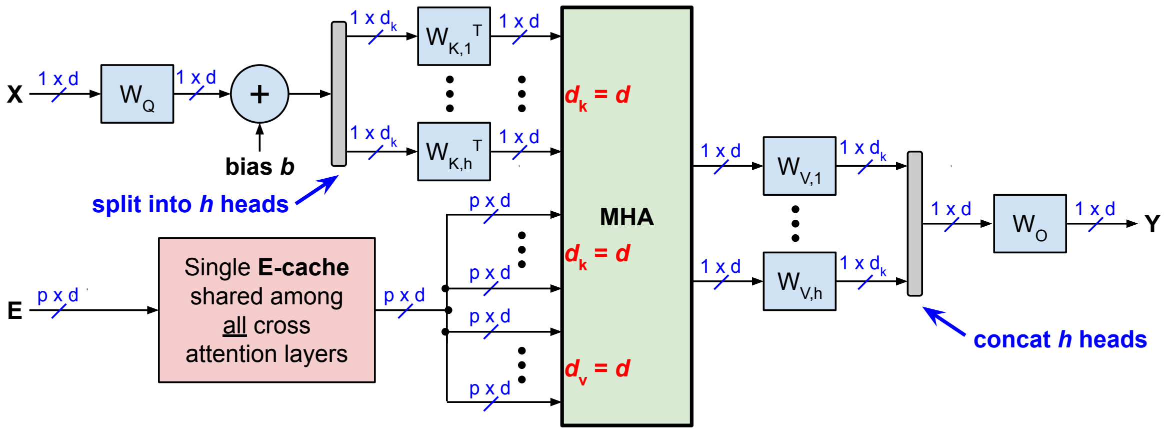

Figure 5: Slim attention with a single X-cache that is shared among all cross attention layers for the generate-phase of encoder-decoder transformers.

The table below lists the cache size savings and speedups during the generate phase for all 5 Whisper models, assuming a fixed encoder context length of $p = 1500$, a decoder output context length of 448, and a vocab_size of 51,865. Option 2 reduces the cache sizes by 8.7x, assuming that the entire encoder output (matrix E) is kept in on-chip SRAM (the E-cache), which speeds up the generate phase by over 5x for a memory bound system with batch size 64. Note that the speedups listed in the table are in addition to speeding up the cross phase (i.e. 2x cross phase speedup for option 1, and entire elimination of the cross phase for option 2).

There is a further optimization, which is a hybrid of options 1 and 2, where one layer of the decoder stack uses option 1 and the other layers use a modified version of option 2 as follows: Say the first layer implements option 1, which entails calculating and storing the cross K-cache for the first layer, i.e. $K = EW_K$. The trick is now to use this K-cache instead of the E-cache for all the other layers and then we can calculate E from K as $E = KW_{K\_layer1}^{-1}$. So the other layers will now use K instead of E, which means that they need to use modified versions of their W_K and W_V weight matrices, which are offline computed as $W_{V\_new} = W_{K\_layer1}^{-1} W_{V\_old}$ and $W_{K\_new} = W_{K\_layer1}^{-1} W_{K\_old}$. The savings of this hybrid versus option 2 are not huge: It only speeds up the first layer because we only need to read one of the two cross weight matrices for the first layer for each generated token (i.e. we only need to read W_{KV_layer1} instead of reading both W_{K_layer1} and W_{V_layer1}). So this hybrid makes only sense for shallow models, e.g. Whisper tiny which only has 4 layers.

References

[1] 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*.

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

[3] 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*.

[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] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. October 2019. *arXiv:1910.10683*.

[6] 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*.

[7] 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, Vinodkumar Prabhakaran,

	Whisper models					
	tiny	**base**	**small**	**medium**	**large**	**Notes**
Params	38M	73M	242M	764M	1.5B	
Layers	4	6	12	24	32	
d	384	512	768	1,024	1,280	embedding dimension
d_{ffn}	1,536	2,048	3,072	4,096	5,120	hidden dimension of FFN
Cache sizes (in M):						
Encoder E-cache	0.6	0.8	1.2	1.5	1.9	$1500 \cdot d$
Cross KV-cache	4.6	9.2	27.6	73.7	122.9	$2 \cdot 1500 \cdot d \cdot$ layers
Self KV-cache	1.4	2.8	8.3	22.0	36.7	$2 \cdot 448 \cdot d \cdot$ layers
Baseline cache	6.0	12.0	35.9	95.7	159.6	cross KV + self KV
Option 1 cache	3.0	6.0	18.0	47.9	79.8	half of baseline
Option 2 cache	0.7	1.4	4.1	11.0	18.4	no cross KV + half of self KV-cache
Option 2 savings	8.7x	8.7x	8.7x	8.7x	8.7x	cache savings vs. baseline
Number of parameters (in M) for generate phase:						
Baseline params	28.2	48.6	138.9	405.4	800.4	$d \cdot$ vocab + layers $\cdot (6d^2 + 2d \cdot d_{ffn})$
Option 1 params	28.8	50.1	146.0	430.6	852.8	baseline + cross K ($d^2 \cdot$ layers)
Option 2 params	29.4	51.7	153.1	455.8	905.2	baseline + cross KV ($2d^2 \cdot$ layers)
Memory reads (in M) per token for batch size 1:						
Baseline	34.2	60.5	174.8	501.2	960.0	baseline cache + baseline params
Option 1	31.8	56.1	164.0	478.5	932.6	option 1 cache + option 1 params
Option 2	30.0	53.1	157.2	466.8	923.6	option 2 cache + option 2 params
Option 1 speedup	1.08x	1.08x	1.07x	1.05x	1.03x	speedup vs. baseline
Option 2 speedup	1.14x	1.14x	1.11x	1.07x	1.04x	speedup vs. baseline
Memory reads (in M) per token for batch size 64:						
Baseline	6.4	12.7	38.1	102.1	172.1	baseline cache + $1/64\cdot$ params
Option 1	3.4	6.8	20.2	54.6	93.1	option 1 cache + $1/64\cdot$ params
Option 2	1.1	2.2	6.5	18.1	32.5	option 2 cache + $1/64\cdot$ params
Option 1 speedup	1.9x	1.9x	1.9x	1.9x	1.8x	speedup vs. baseline
Option 2 speedup	5.6x	5.8x	5.8x	5.6x	5.3x	speedup vs. baseline

Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier Garcia, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathy Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. PaLM: Scaling language modeling with Pathways. April 2022. *arXiv:2204.02311*.

[8] Shanda Li, Chong You, Guru Guruganesh, Joshua Ainslie, Santiago Ontanon, Manzil Zaheer, Sumit Sanghai, Yiming Yang, Sanjiv Kumar, and Srinadh Bhojanapalli. Functional interpolation for relative positions improves long context Transformers. October 2023. *arXiv:2310.04418*.