Optimized Linear Attention for TPU Hardware

Gabrael Levine

March 2024

1 Abstract

Recent works on kernel-based attention such as [8] offer an alternative to classical attention with linear rather than quadratic time complexity, enabling transformer-like models to efficiently process long sequences. However, when implemented naively, these algorithms greatly underperform classical attention in speed and memory usage. In this work, we demonstrate a hardware-aware linear attention algorithm, specially optimized for the Tensor Processing Unit (TPU) chip, that is substantially faster than all baselines.

2 Related work

Many approaches to subquadratic sequence modeling have been explored recently. State-space models (SSMs) have been heavily explored for NLP in recent years, with works such as S4 [2] and S5 [5] demonstrating perplexity on language-modeling tasks that compares to Transformers, but with far faster inference.

Long-context convolutional networks using Fourier transforms. Classical convolution kernels are limited to a fixed context (due to the fixed dimensions of the kernel), making them unsuitable on their own for language modeling, which involves long-horizon dependencies between words and phrases. Works such as Hyena [4] take advantage of Fourier transforms to apply convolution kernels with effectively unlimited context.

While linear attention has historically underperformed relative to traditional dot-product attention, some new approaches apply better feature representations to boost performance. This includes the Based architecture, which approximates softmax using a Taylor series, and Porcupine [8], which applies an exponential activation function to obtain "spikier" and thus better-performing features.

Due to the simplicity of its computations, we believe linear attention plays to the strengths of TPU hardware (very fast matrix multiplication). In constrast, state-space models and long-convolutional models depend on specific mathematical operations (associative scans and Fourier transforms) that TPU hardware is not optimized for.

3 Approach

Kernel-based attention is commonly computed two different ways: using an inner product between queries and keys, or using an outer product between keys and values. Although these two algorithms produce the same output when given the same inputs, the time and space complexity is not the same.

3.1 Inner-product formulation

$y_i = \frac{\sum_{j=1}^{i}(q_i^T k_j)v_j^T}{\sum_{j=1}^{i} q_i^T k_j}$

The inner product algorithm requires $O(n^2)$ computation.

3.2 Outer-product formulation

$y_i = \frac{q_i^T \sum_{j=1}^{i}(k_j v_j^T)}{q_i^T \sum_{j=1}^{i} k_j}$

The outer product algorithm requires $O(nd^2)$ computation.

3.3 Optimization 1: Block-wise outer-product formulation

TPU programs operate in a cycle of load $\rightarrow$ process $\rightarrow$ store. We can take a simple matrix multiplication $C = AB$ as an example. While these devices can process matrix multiplications with extremely large dimensions, they are conducted in a block-wise manner (figure), transferring blocks of A and B into memory and storing the output for a block of C.

Memory transfers account for the majority of the computation time when training large neural networks [3], so we want to make as much of the computation happen in fast SRAM as possible, minimizing transfers to and from the comparatively slower HBM main memory. This motivates recent kernel fusion approaches such as FlashAttention [1].

Additionally, TPUs are optimized for matrix-matrix products. The bulk of their computational capability comes from a dedicated matrix multiplication unit (citation), which multiplies two $m \times m$ matrices to produce an $m \times m$ output (where $m = 128$ on recent-generation TPUs). While they are also capable of vector and scalar operations, the available computation power is severely limited compared to the matrix product.

Therefore, an optimized implementation for TPU has two main goals:

1. Minimize the amount of data transferred between HBM and SRAM.

2. Whenever possible, formulate all mathematical operations as products of two 128×128 matrices.

Note that the output at step i only depends on the cumulative sum of the outer products up to i. This opens up the possibility to compute the output recurrently without having to keep the entire $O(nd^2)$ outer product result in memory. Hence, the first optimization we make is to reformulate the summation in a recurrent block-wise manner.

w The algorithm is as follows. Let b be the block size. i_b is the index of the start of the current block. Initialize the carry variables $(c_{kv})_{i_b}$ and $(c_k)_{i_b}$ to zeroes.

$i_b = \text{floordiv}(i, b)$

We split up the summations in the numerator and denominator into an intra-block term and an inter-block (carry) term.

$y_i = \frac{q_i^T \sum_{j=i_b}^{i}(k_j v_j^T) + q_i^T (c_{kv})_{i_b}}{q_i^T \sum_{j=i_b}^{i} k_j + q_i^T (c_k)_{i_b}}$

The carry variables are updated as following:

$(c_{kv})_{i_b+b} = \sum_{j=i_b}^{i_b+b-1}(k_j v_j^T) + (c_{kv})_{i_b}$

$(c_k)_{i_b+b} = \sum_{j=i_b}^{i_b+b-1} k_j + (c_k)_{i_b}$

Since we only cache the carry variables instead of the full outer product at all sequence indices, the memory requirement for the outer product is reduced from $O(nd^2)$ to $O(d^2)$.

3.4 Optimization 2: Block-wise hybrid formulation

While the above approach greatly reduces the memory usage (satisfying goal 1), it still can't be implemented efficiently on TPUs.

When compiled for a TPU matrix multiplication core, the outer product $k_i v_i^T$ looks like this:

$[k_i 0_m s 0_m][v_i 0_m s 0_m]^T$

All of the zero columns/rows represent wasted computation, an artifact of padding the vectors to make them into square matrices for the TPU matrix multiplication unit. This results in a 128× speed penalty when $m = 128$. Ideally, we would avoid computing any outer products due to this inefficiency. This motivates the next optimization, which brings back parts of the original inner-product, quadratic-time algorithm.

In our proposed algorithm, the intra-block computations are performed with the inner-product formula ($O(b^2)$ memory, $O(1)$ time), and the inter-block computations are performed with the recurrent outer-product formula ($O(d^2)$ memory, $O(\frac{n}{b})$ time). To ensure square matrices, we set the block size b equal to the head dimension d. Thus, the overall computational complexity is $O(d^2)$ in memory and $O(\frac{n}{d})$ in time.

Algorithm 1 Forward Pass

```
1:  Let $M$ be a lower-triangular $b \times b$ matrix with all nonzero elements equal to 1.
2:  Initialize carry variables $(C_{KV})_{i_b}$ and $(C_K)_{i_b}$ to zeroes.
3:  Let $\odot$ denote elementwise multiplication and $\oslash$ denote elementwise division.
4:  For each block $Q_{i_b}$, $K_{i_b}$, $V_{i_b}$ of the input:
5:      Attention scores computation for the current block:
6:      $A_{i_b} = M \odot (Q_{i_b} K_{i_b}^T)$
7:      Compute output for the block:
8:      $Y_{i_b} = (A_{i_b} V + Q_{i_b} (C_{KV})_{i_b}) \oslash (A_{i_b} \mathbf{1}_{b \times b} + Q_{i_b} (C_K)_{i_b})$
9:      Update the carry variables:
10:     $(C_{KV})_{i_b + b} = K_{i_b}^T V_{i_b} + (C_{KV})_{i_b}$
11:     $(C_K)_{i_b + b} = K_{i_b}^T \mathbf{1}_{b \times b} + (C_K)_{i_b}$
```

As these computations consist only of matrix-matrix products and elementwise operations, they can be performed efficiently on TPUs.

3.5 Computing the backward pass

We use recomputation, similar to [1], to avoid caching large tensors in HBM for intermediate gradient computations. Like the forward pass, we operate over tiles of the input and transmit information along the sequence through carry variables.

4 Experiments

We compare the following attention algorithms using a single attention head with dimension 128.

Optimized kernel attention: The main proposed approach of this work (described in the Optimization 2 section above). Implemented in Pallas, a low-level language built on JAX that offers finer-grained control over how computations are carried out on the TPU hardware.

Algorithm 2 Backward Pass

```
1: Let $M$ be a lower-triangular $b \times b$ matrix with all nonzero elements equal to 1.
2: Let $(C_{KV})_{i_f}$ and $(C_K)_{i_f}$ be the final values of the carry variables returned by the forward pass. Their gradients $(dC_{KV})_{i_b}$, $(dC_K)_{i_b}$ are initialized to zero.
3: Let $\odot$ denote elementwise multiplication and $\oslash$ denote elementwise division.
4: For each block $Q_{i_b}$, $K_{i_b}$, $V_{i_b}$, $dY_{i_b}$ in reverse order of the input:
5:     Update carry variables:
6:     $(C_{KV})_{i_b} = (C_{KV})_{i_b} - K_{i_b}^T V_{i_b}$
7:     $(C_K)_{i_b} = (C_K)_{i_b} - K_{i_b}^T \mathbf{1}_{b\times b}$
8:     Compute attention scores:
9:     $A_{i_b} = M \odot (Q_{i_b} K_{i_b}^T)$
10:    Compute intermediate variables for gradient computation:
11:    Numerator: $N = A_{i_b} V + Q_{i_b} (C_{KV})_{i_b}$
12:    Denominator: $D = A_{i_b} \mathbf{1}_{b\times b} + Q_{i_b} (C_K)_{i_b} + \text{EPS}$
13:    Compute gradients for numerator and denominator:
14:    $dN = dY_{i_b} \oslash D$
15:    $dD = -dY_{i_b} \odot (N \oslash D^2)$
16:    Compute gradients for attention scores, $V$, $Q$, $(C_{KV})_{i_b}$, and $(C_K)_{i_b}$:
17:    $dA_N = dN V^T$
18:    $dA_D = dD \mathbf{1}_{b\times b}$
19:    $dA_{i_b} = M \odot (dA_N + dA_D)$
20:    $dV = (A_{i_b}^T dN) + (K_{i_b} (dC_{KV})_{i_b})$
21:    $dQ = (dA_{i_b} K_{i_b}) + (dN (C_{KV})_{i_b}^T) + (dD (C_K)_{i_b}^T)$
22:    Update gradients of carry variables:
23:    $(dC_{KV})_{i_b} = Q_{i_b}^T dN$
24:    $(dC_K)_{i_b} = Q_{i_b}^T dD$
```

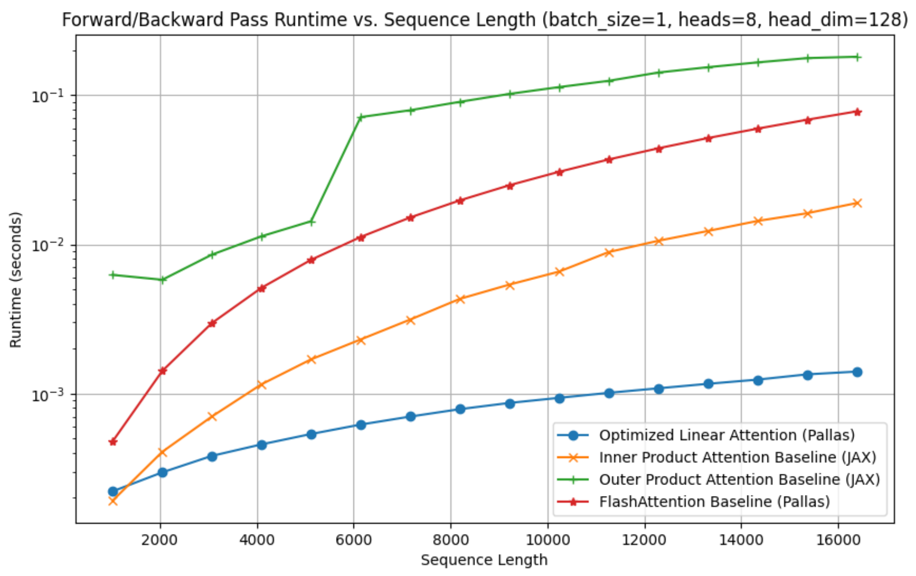

Figure 1: Runtime comparison of the forward and backward pass with baseline attention algorithms. The experiments are carried out on a single TPUv4-8 node.

Inner-product kernel attention: While *linear* attention is seldom implemented like this in practice due to the quadratic time complexity, it serves as an additional baseline. Implemented in plain JAX and JIT-compiled.

Outer-product kernel attention: Baseline representative of common linear attention implementations. Implemented in plain JAX and JIT-compiled.

FlashAttention [1]: The current state-of-the-art for hardware-optimized classical attention. Implemented in Pallas.

4.1 Results and Analysis

In terms of speed, our approach is more performant than every baseline for sequence lengths of 2048 and greater. This demonstrates the advantages of using hardware-aware algorithms for computing kernel attention.

The outer-product algorithm is by far the slowest, at least two orders of magnitude slower than our optimized algorithm. This validates our hypothesis that the naive outer-product attention algorithm is unsuitable for TPUs.

We had also intended to gather data on memory usage. However, we encountered difficulties getting the Jax profiling tools to output memory usage data for the on-device TPU memory. Despite this, we can anecdotally confirm from our testing that our algorithm permits a far larger batch size before running out of memory, in comparison to the baselines, pointing to its improved memory efficiency.

4.2 Future work

We highlight two possible improvements that could potentially boost performance on language modeling tasks.

4.2.1 Hardware-aware state expansion

The experiments in [6] show that model performance on challenging in-context learning benchmarks such as MQAR scales with the hidden state dimensionality. Intuitively this makes sense, because a model with a larger hidden state can store more information. The authors demonstrate a computationally-efficient way to accomplish this for linear attention, where the q and k vectors are up-projected to higher dimensionality after they have been transferred from HBM to SRAM. This would be straightforward to add to our block-wise linear attention algorithm, and could result in significant performance increases on recall-intensive tasks.

4.2.2 Forgetting

The work in [7] demonstrates a Mamba-like state-space-model architecture optimized for TPU. As in Mamba, their model can selectively forget or remember parts of the hidden state at each timestep. The authors show that this allows for better length generalization on language modeling and synthetic benchmarks. Again, this intuitively makes sense, because without forgetting past information, the hidden state will become increasingly noisy/high-entropy along the sequence as more information is added to it.

While forgetting could be accomplished with a simple, fixed exponential decay of the hidden state (solving the increasing entropy issue), this would hurt in-context learning, as the influence of the prompt on the model output would become lower and lower the more tokens are generated. Instead, we believe an approach like [7] could work well, where the model predicts a "forget gate" value at each sequence step, selectively controlling how much the hidden state decays at that particular step.

Further work would also include measuring the real-world impact on training and inference speed when our linear attention algorithm is used to construct a language model. We have implemented a proof-of-concept for language model training using a modified version of the Levanter framework (https://github.com/G-Levine/levanter), laying the groundwork to evaluate the performance of multibillion-parameter-scale language models that use our attention mechanism.

4.3 Conclusion

In this work, we have presented a highly-efficient hardware-aware algorithm for computing linear attention on TPU. Based on the orders-of-magnitude speed improvement over the baselines demonstrated in our experiments, we believe this is a promising method for training future large language models on TPU.

5 Acknowledgements

Research supported with Cloud TPUs from Google's TPU Research Cloud (TRC)

References

[1] Tri Dao et al. *FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness*. 2022. arXiv: `2205.14135 [cs.LG]`.

[2] Albert Gu, Karan Goel, and Christopher Ré. *Efficiently Modeling Long Sequences with Structured State Spaces*. 2022. arXiv: `2111.00396 [cs.LG]`.

[3] Albert Gu and Tri Dao. *Mamba: Linear-Time Sequence Modeling with Selective State Spaces*. 2023. arXiv: `2312.00752 [cs.LG]`.

[4] Michael Poli et al. *Hyena Hierarchy: Towards Larger Convolutional Language Models*. 2023. arXiv: `2302.10866 [cs.LG]`.

[5] Jimmy T. H. Smith, Andrew Warrington, and Scott W. Linderman. *Simplified State Space Layers for Sequence Modeling*. 2023. arXiv: `2208.04933 [cs.LG]`.

[6] Simran Arora et al. *Simple linear attention language models balance the recall-throughput tradeoff*. 2024. arXiv: `2402.18668 [cs.CL]`.

[7] Soham De et al. *Griffin: Mixing Gated Linear Recurrences with Local Attention for Efficient Language Models*. 2024. arXiv: `2402.19427 [cs.LG]`.

[8] Michael Zhang et al. "The Hedgehog & the Porcupine: Expressive Linear Attentions with Softmax Mimicry". In: *The Twelfth International Conference on Learning Representations*. 2024. URL: `https://openreview.net/forum?id=4g02l2N2Nx`.