Flash Attention Final Report

Adhitya Polavaram (ap977), Alkis Boukas (ab2453), JT Klenke (jtk96)

Link to code

Introduction

Attention is a ubiquitous computation used in transformers. It processes a sequence of embeddings that represent the tokens. Mathematically, it involves 3 sequence length by embedding dimension (shortened to $N \times d$) matrices, Q, K, V. The attention computation, $\mathrm{softmax}(\frac{QK^T}{\sqrt{d}})V$, involves computing $QK^T = S$ a large $N \times N$ matrix and then computing a softmax operation over the rows of S before finally multiplying by V to yield a $N \times d$ output matrix. The softmax operation is defined for an n dimensional vector x as $\frac{\exp x_i}{\sum_{j=1}^{n} \exp x_j}$, an exponentiation of all entries vectors (in our case rows of S) and then a normalization. However, in order to maintain numerical stability in practice, we want to first subtract all values by the max of all values in the row, $m = \max_i(x_i)$. Note that this doesn't change the value of the softmax, and provides numerical stability by making sure that no exponent becomes too large. Given that the softmax operation involves two global values, the maximum and this normalization factor, it would seem that we'd be forced to materialize all of S at once, but with clever updating we can instead keep a running value for the maximum and normalization factor and update past values. The final answer will be mathematically equivalent (although not equivalent in practice because of floating point rounding error).

Since $N >> d$, with even longer sequence lengths being used by latest architectures, attentions matrices are typically tall and skinny which means that we want to avoid having to materialize all of S. We do this by tiling Q, K, V into $B_r \times B_c$ sized tiles. Then we can make use of the tiles Q_i, K_j to compute a tile of S, we keep a running value for the normalization factor and the maximum value for each row (two B_r sized vectors), we can then use these values to update past tiles of the output matrix O. Since $\exp(x_i - m_{new}) = \exp(x_i) / \exp(m)$ we can update old values to reflect the new maximum value and similarly update the normalization factor, multiplying by the old factor and dividing by the new one. We do this for each tile, iteratively constructing the final output.

The advantage of this tiling technique comes from kernel fusion and a drastic reduction in the number of high bandwidth memory accesses (HBM, the larger but slower memory on the GPU). Naive computation of attention would use matrix multiplication that essentially only reads from HBM, which makes it memory bound. Instead, we use tiling to load a portion of Q, K, V, and write a portion of O

with all of these values residing in much faster shared memory (SRAM). The problem of having to correctly compute the softmax of the entire row even though we only have access to a tile of memory at a time is solved my the aforementioned exponent trick, where we keep track of the running maximum value and use them to update the row. This turns attention from a memory bottlenecked operation to one that (with the right tile size) is bottlenecked by the computations required in matrix multiplication.

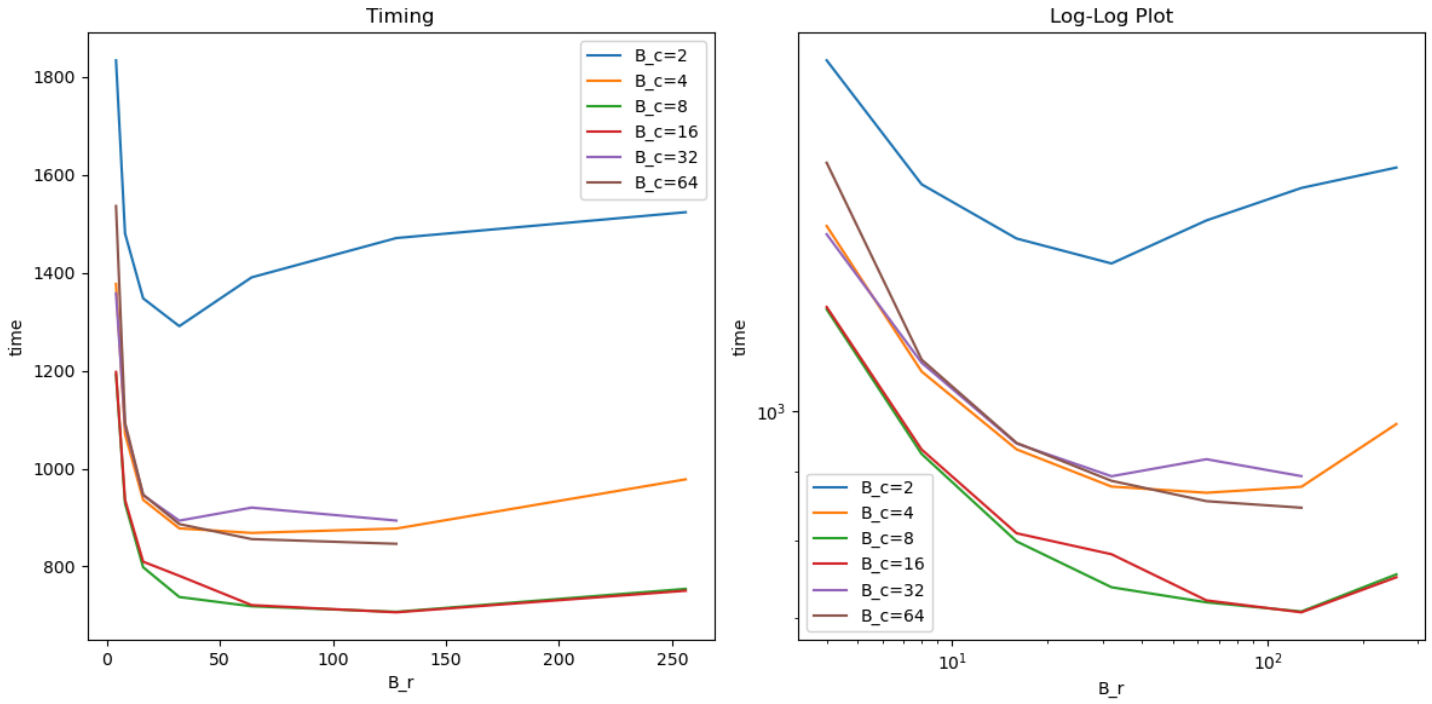
Since attention is typically multiheaded (where the input matrix is instead a 3d tensor of independent, smaller attention computations) and batched (a series of independent instantiation of each 3d tensor), we included that in our implementation. These are both embarrassingly parallel because they offer independent computations of attention.

In order to implement this algorithm in CUDA, we exploited the embarrassing parallelism across heads and batches by making our grid includes the number of heads and batch size as dimensions. This leaves each block having to handle T_r (the number of tiles that span Q) tiles of computation. Then, we create a thread to handle each row of the tile. This allows each thread to work on building a row of O_i independently and requires synchronization only when moving to the next block to make sure that all threads have finished with the memory in SRAM before it is replaced. This worked quite well, but we saw one further area of optimization - to have each block handle only one tile, by adding T_r as the third dimension of the grid. This way, we add another dimension of parallelization that leads to each thread only needing to tend to exactly one row of the output matrix.

This strategy explicitly ties our launch parameters to our problem size. Specifically, our grid dim is ParseError: KaTeX parse error: Expected 'EOF', got '_' at position 13: (\text{batch_size}, \text{nu… and our block dim is B_r. The main area for further improvements would be to allocate more threads per block to assist with matrix multiply, however, we did not have enough time to get cutlass working (which uses all threads in the block to perform the matrix multiply) or to implement a more parallel version of matrix multiply.

Scaling results

Block scaling



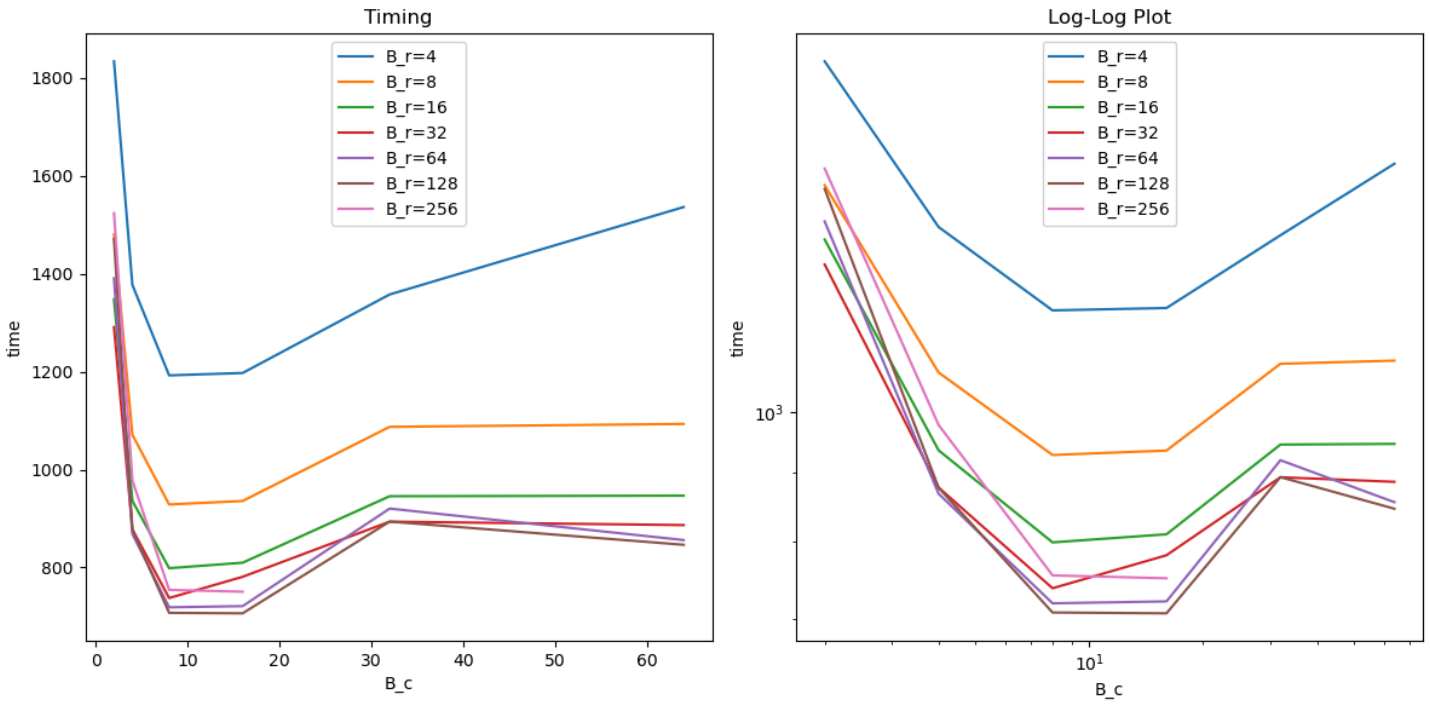
This graph shows the effect of changing the block size on the speed of Flash Attention. B_r is the height of the block (and the axis where we parallelize) and B_c is the width of the block which obviously effects the total size of the block as well as the SRAM utilization. You can see that as B_r increases the implementation gets faster (up to a point), this makes sense as the profiler results suggest that we are compute bound and have relatively low thread utilization, thus spawning more threads to do work on the blocks speed up the computation.

We see that when B_r becomes a multiple of 32, that the improvements level off, probably because then we achieve warp efficiency where all the threads are executing the same instructions.

The reason that B_r eventually trends up seems to be cache effects because the l2 throughput doubles and the number of available registers to each thread halves from the 128×16 to the 256×8 (shown below),

```
Metric Name                          Metric Unit    Metric Value
-------------------------------- -------------- ---------------
L1/TEX Cache Throughput                      %           98.56
L2 Cache Throughput                          %           12.09
Block Limit Registers                    block              16
Waves Per SM                                                4.94
-------------------------------- -------------- ---------------
L1/TEX Cache Throughput                      %           98.78
L2 Cache Throughput                          %           26.38
Block Limit Registers                    block               8
Waves Per SM                                                3.70
```

the issue is that the registers per thread decreases (since the blocks have more threads) and it seems that there are fewer blocks allocated per SM (since the number of waves per SM is higher). This suggests that the memory access patterns (of the registers) are a significant bottleneck given that even for a higher number of waves per SM, the smaller block size is still faster.



We seem there is a clear sweet spot for B_c. We would expect larger B_c to perform better until SRAM has been fully utilized, this is because there is a roughly fixed cost to moving data from HBM to SRAM. As SRAM is quite small, moving data to it is dominated more by latency than throughput, therefore we would expect having more SRAM utilization to be strictly better as it reduces the total number of HBM accesses without trading the total number of computations. However, we don't see this. We hypothesize the reason is that our SMs are limited by memory, not thread count, this means that as we increase the size of the tile (without also increasing the thread count) our blocks resource requirement

gets even more out of balance which means that each SM is using all of its memory but not all of its threads. For a theoretically compute bound operation like attention (which is largely matrix multiplies), this is a huge bottleneck.

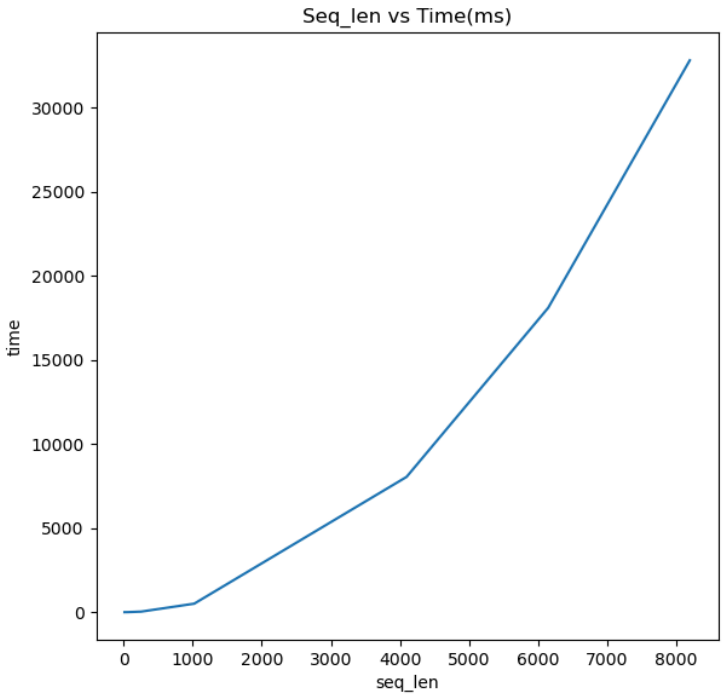
Consider the following profiling results, both for $B_r = 32$ but with $B_c = 8, 32$ respectively (the larger is slower).

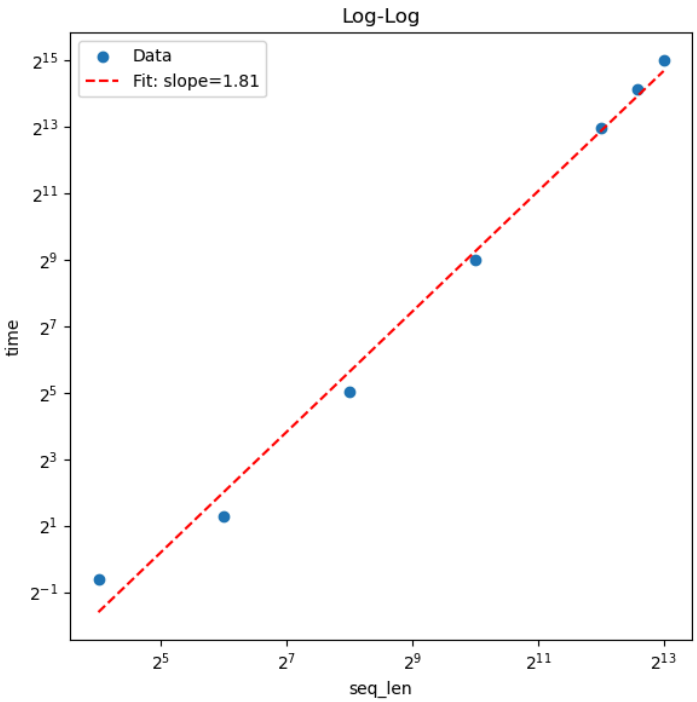
```
Metric Name                          Metric Unit    Metric Value
-------------------------------- --------------- ---------------
Dynamic Shared Memory Per Block      Kbyte/block           13.31
Achieved Active Warps Per SM                warp           10.58
Waves Per SM                                               5.39
-------------------------------- --------------- ---------------
Dynamic Shared Memory Per Block      Kbyte/block           28.67
Achieved Active Warps Per SM                warp            4.94
Waves Per SM                                              11.85
```

Unsurprisingly, we see that the shared memory per block is much lower for smaller tiles, this allows more active warps per SM which is where the speedup comes from (since this better balanced the resources in an SM). We can see that despite having fewer tiles (since each is larger), the number of waves per SM is larger, this suggests that for a smaller tile size we are able to schedule enough more concurrent blocks that it makes up for the fact that each block has to load values from K_j, V_j more frequently.

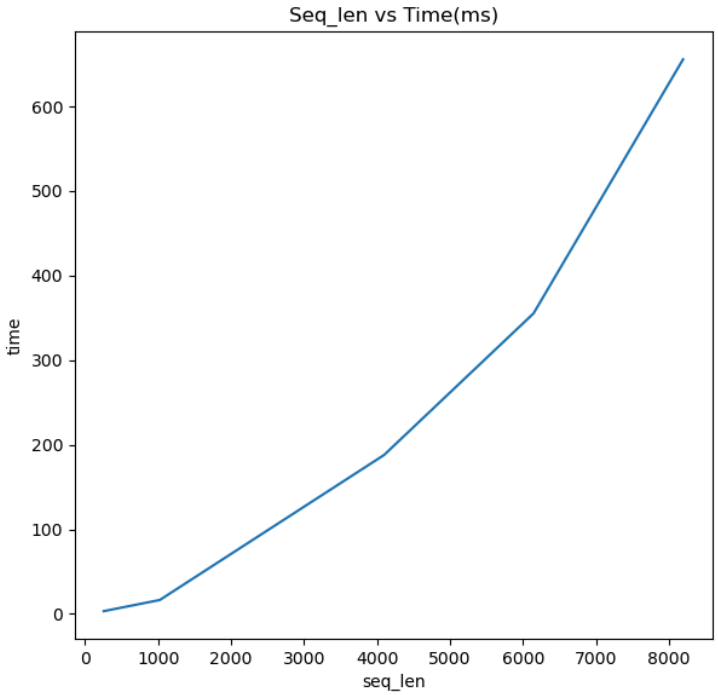
Sequence length scaling

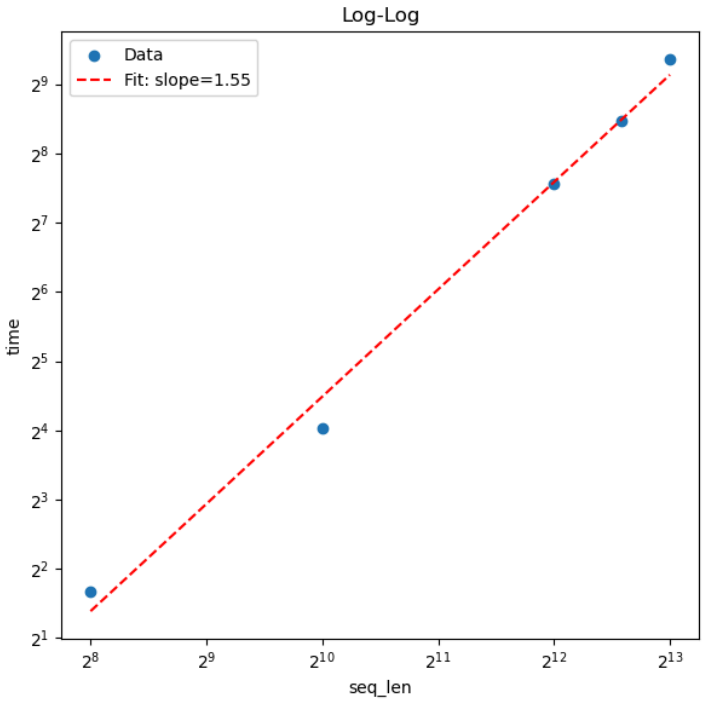
Sequence length scaling before parallelizing over Q_i





Sequence length scaling after parallelizing Q_i

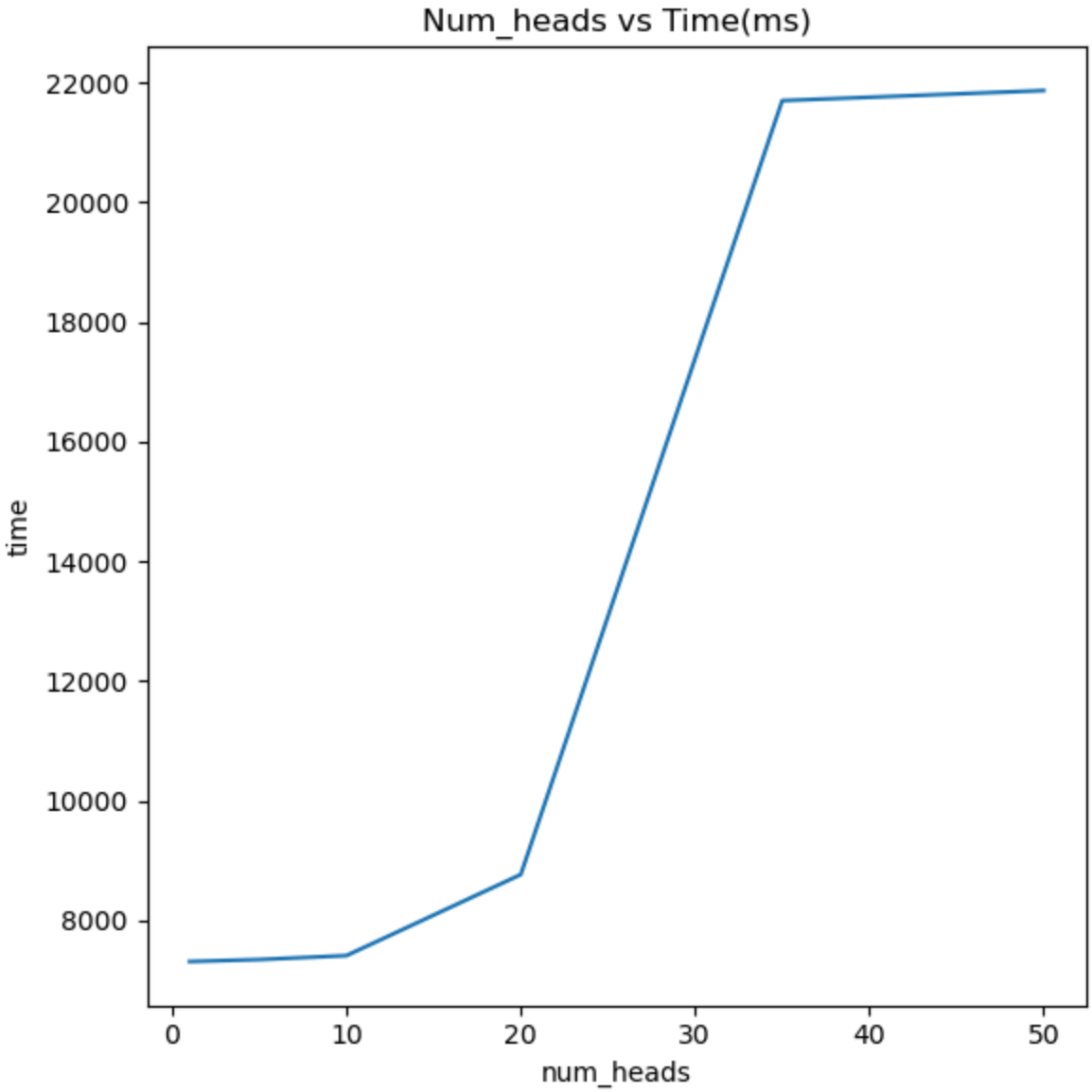




We see that the timing scales sub-quadratically in both cases but that scaling is better in the case where we spawn more blocks, this makes sense because it involves more parallelism along the scaling dimension. In both cases the scaling tends towards quadratic because attention is a quadratic operation in sequence length, thus, since parallelism eventually becomes constrained by GPU resources we would expect eventual convergence to quadratic (we can see in the log-log plot that the higher points have a larger slope, closer to quadratic).

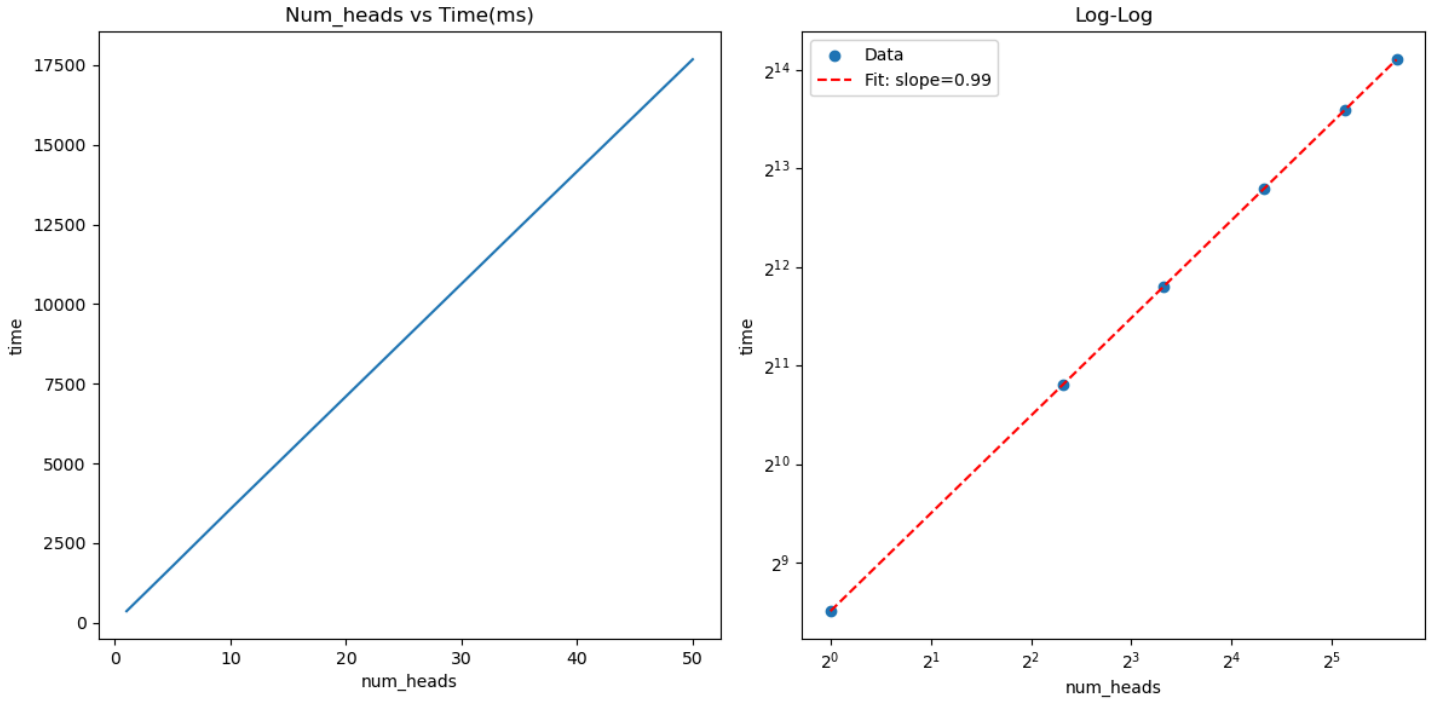
Attention Head & Batch Size Scaling

Since both the attention head and batch size are embarrassingly parallel, we would expect to see a scaling law that is roughly constant (ie ideal weak scaling). However, we see two different behaviors depending on the implementation. (For clarity the graphs will only show scaling of attention heads as it looks nearly identical to scaling for batch size.) Before implementing the optimization that spawned threadblocks for each Q_i we saw the following scaling plot.



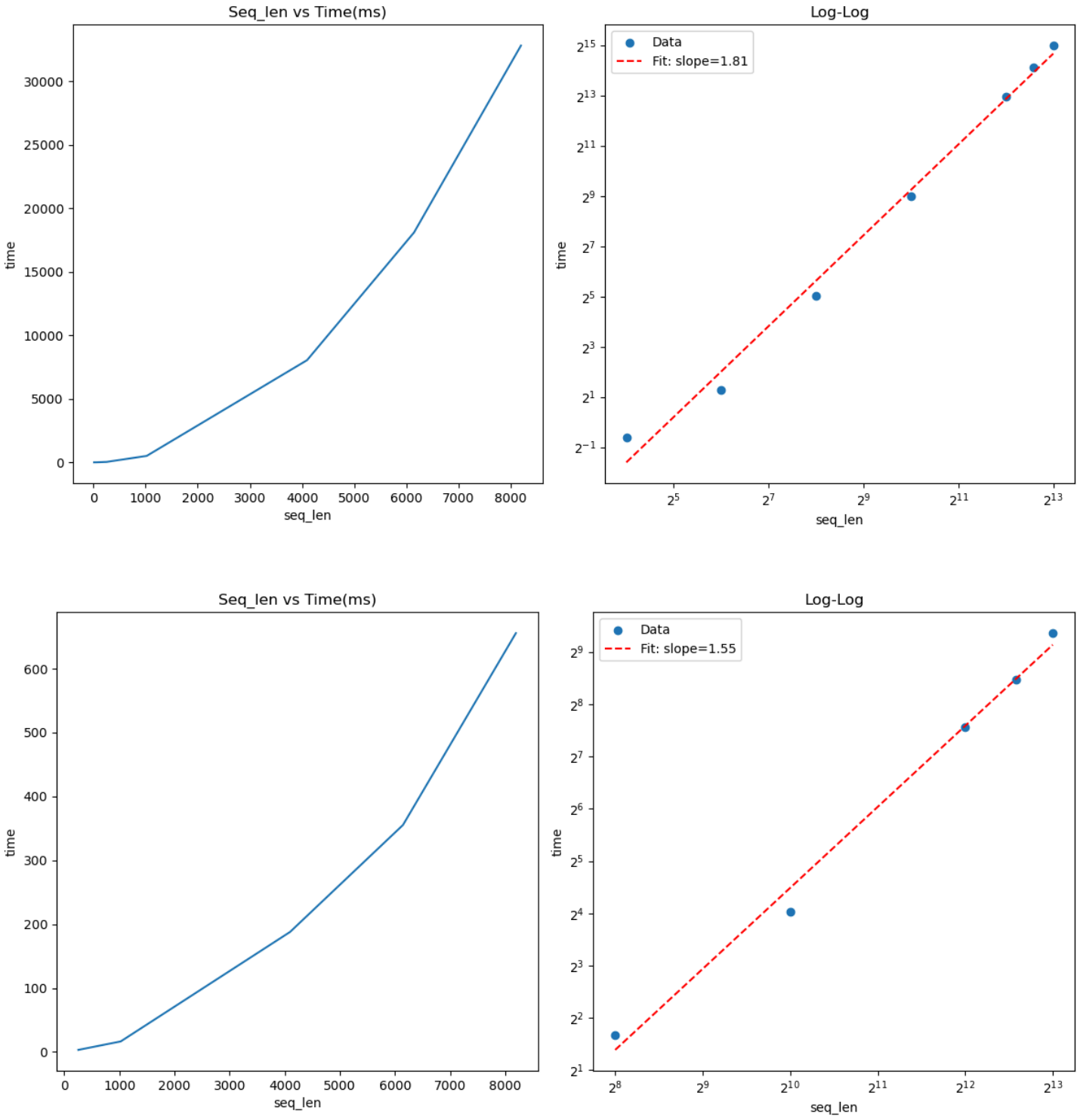
This shows roughly constant time until it jumps significantly, we hypothesize this is caused by having fewer blocks that could be maximally scheduled by the SMs, this means that adding more blocks directly creates opportunities for embarrassing parallelism. However, once the number of blocks gets high enough, there are more than can be simultaneously run which means that things begin to scale linearly. This also explains why it levels off again as the number of blocks will not lead to significant

slowdowns until it triggers another wave. The comparatively small scaling is likely because of costs of moving memory to HBM that scale loosely with problem size.



After spawning blocks for each Q_i, there are now significantly more blocks than can be simultaneously run on the SMs, this means that as we scale up the number of blocks, parallelism doesn't increase. This creates the perfect linear scaling we see as the number of waves per SM scales with the problem size.

Attempted Optimizations



Spawning a threadblocks for each Q_i sped up the implementation significantly, we can see in the scaling law of the sequence length

We attempted to use static shared memory rather than dynamic static memory, this involved changing the block of memory we were using inside the kernel to load Q_i, K_j, V_j, O_i from

```
extern __shared__ float sram[];
```

to

```
constexpr int size = 2 * B_r * d + 2 * B_c * d;
__shared__ float sram[size];
```

This set the size of the SRAM at compile time allowing it to be static shared memory. We thought this would be faster because the