CSE237C Final Project: Efficient Accelerations for Attention Model on HLS

Weihong Xu
wexu@ucsd.edu

December 21, 2020

1 Introduction

Attention models are emerging as one types of record-breaking models to model long-term dependencies in sequential data. The success of attention models have been shown in many applications, such as machine translation [8]. The recently proposed transformer architecture [8] has been considered as a better substitute for recurrent neural networks (RNN) [7] and demonstrated the state-of-the-art performance for natural language processing (NLP) applications.

Attention models have also shown the potential of performing image analysis as the form of a non-local layer which is beneficial for its ability to capture relations of image features that span the whole input image [9]. Recently, the transformer architecture has also been applied to computer vision tasks for image classification and object detection [2, 3] where the input images are cropped into patches and assumed dependencies like sequences. For vision tasks, in contrast to convolution which involves the operation of a convolution kernel on each local position, self-attention operates on the whole image and models the feature relations in all the positions across the image.

The biggest challenge to accelerate attention models is the expensive memory cost due to the large memory space consumption to store the score matrix in the case of long sequences. Though generally being more memory-efficient as compared to RNNs, the memory footprint increases quadratically with the sequence length in self-attention models. Compared to other popular models like convolutional neural networks (CNNs) which are computation-bounded, the attention models are dominated by memory because of the lower computational density in terms of the large amount of weights. Therefore, conventional accelerators optimized for convolution layers may not fit well for attention models. Moreover, the finely optimized accelerators for fully-connected layers also show low efficiency due to the existence of long-term data dependency of attention models.

This report addresses the aforementioned challenges in the following steps. First, the bottleneck of baseline attention model is analyzed. The original attention model in floating-point arithmetic are converted into the fixed-point format to obtain higher efficiency. Second, the Softmax operation that requires complex arithmetic units are simplified using Taylor

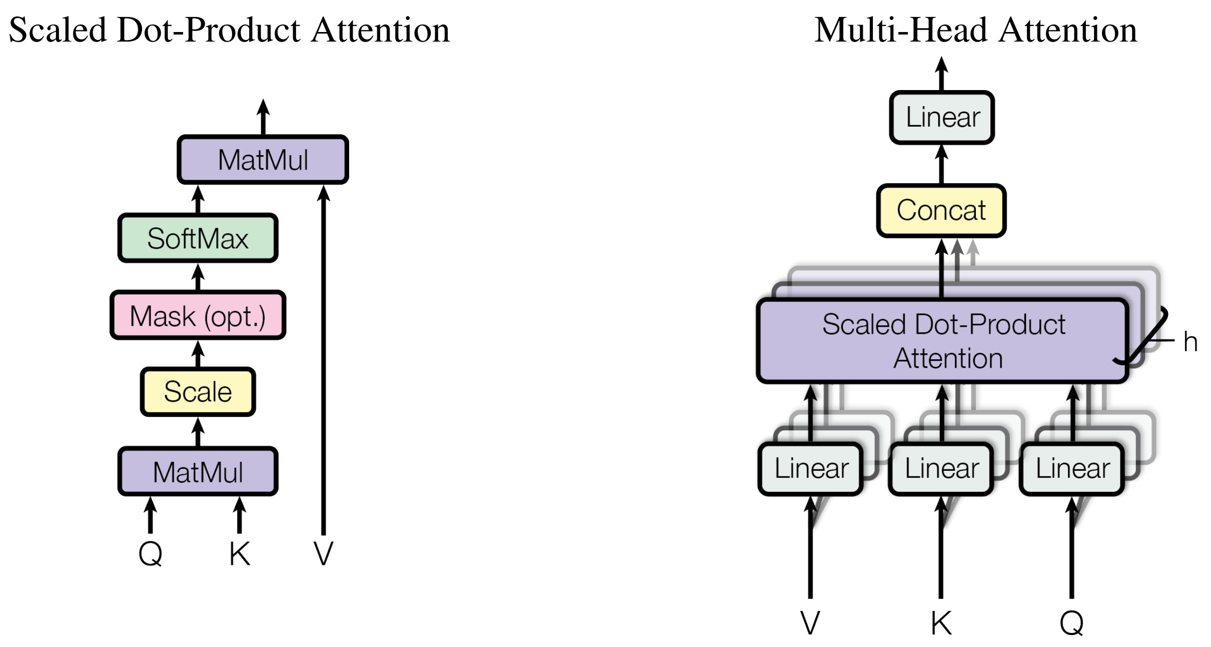

Figure 1: Two key operations in attention models: Scaled Dot-Product Attention and Multi-Head Attention.

series expansion. Finally, the excessive memory consumption of attention models are reduced by optimizing the dataflow and the computing order of matrix multiplication.

2 Analysis of Attention Models

The attention models consist of an encoder and a decoder, where both the encoder and the decoder are constructed by stacking multiple self-attention layers. The key operations in a self-attention layer are the scaled dot-product attention and multi-head attention as shown in Figure 1. For each piece of data in the input sequence, we generate an embedding vector of dimension d_m. For a sequence of length N, all the embedding vectors can be regarded as an embedding matrix of dimension $d_m \times N$. For each word, a query vector Q of dimension d_q and a key vector K of dimension d_k, and a value vector (V) of dimension d_v are generated through multiplying the embedding vector with three weight matrices, respectively. In this report, the values of d_k, d_q and d_v are set to 16. The Q vectors, K vectors and V vectors of all the words in the input sequence can be stacked together as matrices Q, K and V. In the scaled dot-product attention, the dependencies between words in the sequence can be captured via the following equation:

$$\text{Self-attention}(Q, K, V) = \text{Softmax}(\frac{QK^T}{\sqrt{d_k}})V, \tag{1}$$

where Softmax$(\cdot)$ denotes the Softmax function. The $\frac{QK^T}{\sqrt{d_k}}$ is defined as the score matrix S that measures how much focus to put on various parts of the input sequence. We use SA to denote the output of the attention layer.

To achieve higher accuracy and performance, the training of large attention models is typically conducted on long sequences [4–6]. The long sequences pose a severe challenge for accelerating attention models due to the prohibitive memory cost. The attention layer requires $O(N^2)$ memory space to fully parallelize the computation and store intermediate results, where N is the sequence length. In this case, the memory footprint increases quadratically with the sequence length. Concretely, the bottleneck in self-attention is the computation of the score matrix $S = \frac{QK^T}{\sqrt{d_k}}$ in Equation 1, where S has a size of $N \times N$. For typical natural language processing tasks, the score matrix consumes 16GB of memory when the sequence length N is $64K$ and using 32-bit floating-point precision. When further scaling the sequence length, the memory footprint is beyond the capacity of most commodity GPUs.

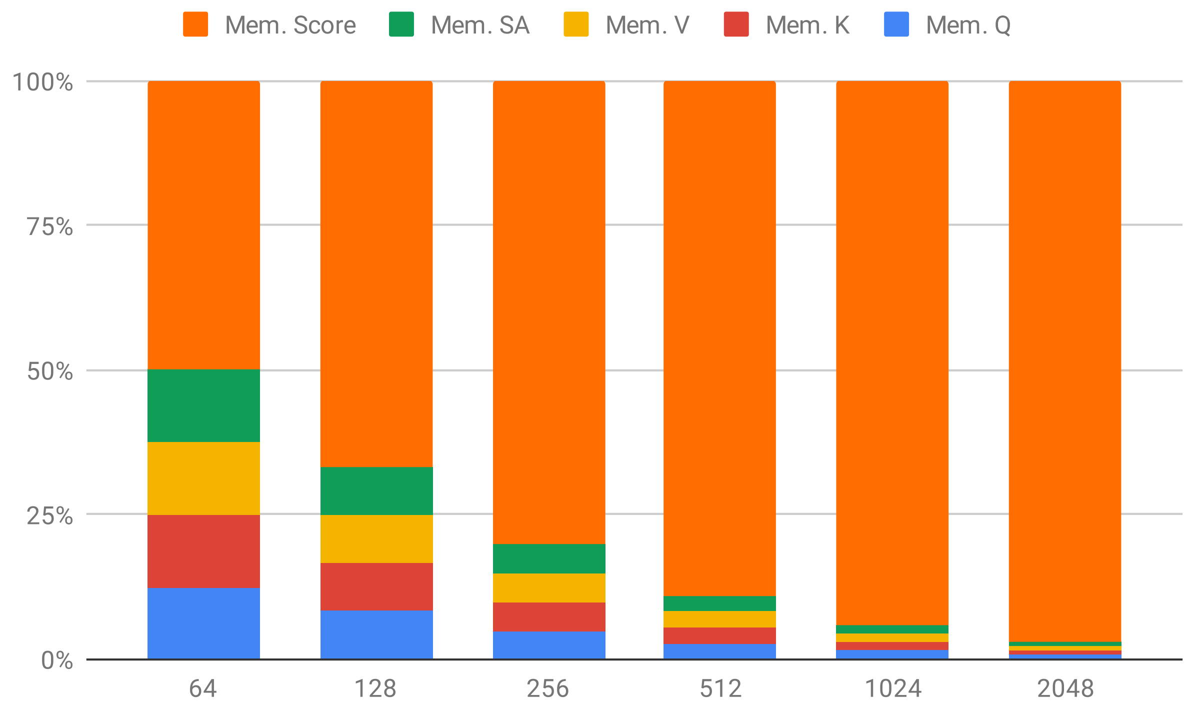

Figure 2: Memory usage breakdown under various sequence lengths.

It is costly to load and process the entire input embeddings since the intermediate matrices would consume huge amount of memory space. We plot the memory usage breakdown for six types of matrices for attention models in Figure 2. It shows that the score matrix S accounts for the majority of total memory usage as the sequence length is greater than 256. Hence, reducing the required memory space of the score matrix is a key design factor.

3 Testbench Development

The ground truth results are generated by referring to the Pytorch implementation in [1]. The code re-implemented using Numpy library is put in Folder **attention_ground_truth**. The model with parameters $d_m = 16$ and $N = 512$ is selected and used for random test data generation. The following code shows part of the forward function that performs the inference of single attention layer:

```python
def forward(self, seq_in):
    v = np.matmul(seq_in, self.weight_v)

    q = np.matmul(seq_in, self.weight_q)

    k = np.matmul(seq_in, self.weight_k)

    scores = np.matmul(q, np.transpose(k))/math.sqrt(self.d_k)
    scores = softmax(scores, axis=-1)

    output = np.matmul(scores, v)

    return output
```

The HLS test code is a modification from the previous projects. Instead of using the MSE metric, the mean-percentage-error is used in this project. The implementation code is as follows:

```cpp
struct Rmpe
{
        int num;
        float sum;
        float error;

        Rmpe()
        {
                num = 0;
                sum = 0;
                error = 0;
        }

        float add_value(float truth, float d_n)
        {
                num++;
                sum += abs(d_n / truth);
                error = sqrtf(sum / num);
                return error;
        }
};
```

The other parts of the test code in HLS basically follow the standard test flow as the previous projects. More details can be found in the file **attention_test.cpp**.

4 Naive HLS Implementation of Attention Model

Figure 3 depicts an intuitive illustration of an attention layer. The computation flow can be divided into four parts. The first part is the calculation of V, Q, and K matrices. Three matrix multiplications are needed in this step. The second part involves the multiplication of Q and K matrices then yielding the score matrix S. In the third step, the score matrix passes a Softmax computation unit to obtain its corresponding matrix in the range between 0 and 1. The final step is a matrix multiplication between matrix V and the Softmax of score matrix. The output is called the SA matrix.

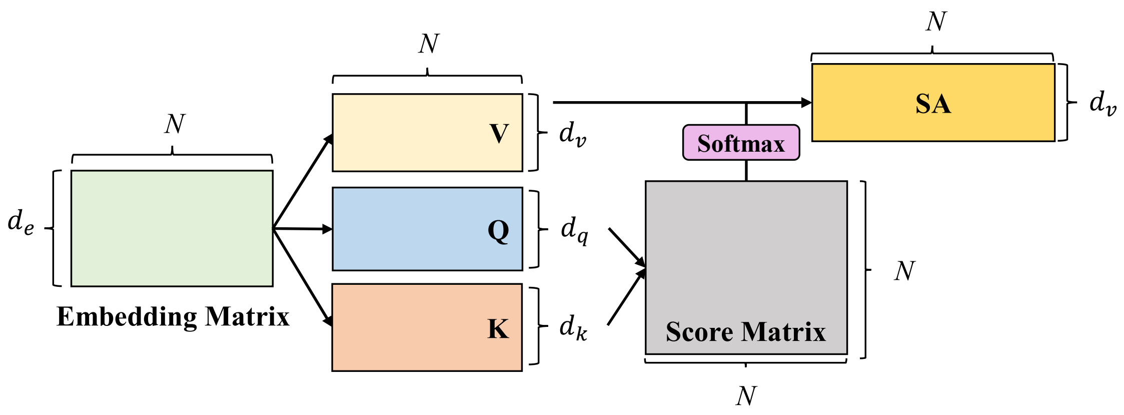

Figure 3: Illustration of an attention layer.

To sum up the realization of attention model consists of four matrix multiplications and one Softmax calculation. The naive HLS is implemented as follows. The data format is floating-point and all matrix multiplications as the code:

```
for (i = 0; i < SEQ_LENGTH; i++)
{
  for (j = 0; j < D_MODEL; j++)
  {
        DTYPE psum = 0.0;
        for (k = 0; k < D_MODEL; k++)
        {
          psum += input_seq[i * D_MODEL + k] * weight_q[k * D_MODEL + j];
        }
        buf_Q[i * D_MODEL + j] = psum;
  }
}
```

The matrix multiplication consists of three loops and follows the inner-product computation flow. In this case, each output psum pixel will be reused by k times.

The Softmax unit is responsible for the 2D Softmax calculation as follows. First, the exponential value of each pixel in the score matrix is computed using **expf** math function. Then each pixel in a row is divided by the accumulated exponential value of this row.

```
CALC_SOFTMAX:
  DTYPE row_sum[SEQ_LENGTH];
  for (i = 0; i < SEQ_LENGTH; i++)
  {
    DTYPE temp_exp;
        row_sum[i] = 0.0;
```

```
      for (j = 0; j < SEQ_LENGTH; j++)
      {
        temp_exp = expf(buf_S[i*SEQ_LENGTH+j]);
        buf_S[i * SEQ_LENGTH + j] = temp_exp;
        row_sum[i] += temp_exp;
      }
  }

  for (i = 0; i < SEQ_LENGTH; i++)
    for (j = 0; j < SEQ_LENGTH; j++)
        buf_S[i * SEQ_LENGTH + j] = buf_S[i * SEQ_LENGTH + j] / row_sum[i];
```

The baseline implementation is general and can be applied to arbitrary sequence lengths. The functionality of the designs is verified against the testbench. Given the clock period constraint of 10 ns, the synthesis results are shown as follows: The BRAM consumption

Table 1: Synthesis results and throughput for the baseline design.

Design	Est. Clk (ns)	Clock Cycles		Utilization			
		Latency	Interval	BRAM	DSP48E	FF	LUT
Baseline	8.427	106,905,095	106,905,095	561	12	2,793	4,515

has exceeded the available resources of the Zynq board. The overall clock cycles are long, limiting the processing throughput. From the report, it is observed that two loops, the calculation of score matrix and the calculation of output matrix, account for the majority of total processing time (around 87%). This is because the size of these two matrices are $N \times N$, where N is much larger than d_m. Hence, the speedup can be achieved by improving the processing time of these two matrices.

5 Optimizations of Efficient Accelerations

5.1 Fixed-point and Simplified Exponential Function

One optimization is done towards transferring the floating-point implementation into fixed-point format. Moreover, the simplification of complex exponential computation in Softmax function will also bring about substantial gain in terms of lower processing latency and less hardware resource consumption. Folder **attention_opt1** contains the optimized code. Instead of implementing the exponential computation using the original **exp** math function, the simplified version is based the Taylor series expansion which can be explained as follows:

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + ... \tag{2}$$

In the experiment, the expansion order is set to two since it can provide sufficient arithmetic precision. In this case, the calculation of exponential function can be done using two additions, two multiplications, and one shift operation. The code implementation of Taylor expansion is given as follows:

```
void CORDIC_exp(data_fixed *x, data_fixed *exp_x)
{
  data_fixed temp = *x;
  *exp_x = data_fixed(1.0) + temp;

  temp *= *x;
  *exp_x += temp << 1;
}
```

Table 2 summarizes the MPE values for different expansion orders. It shows that increasing the expansion order cannot provide further arithmetic precision gain as the expansion order is greater than 2. Hence, the order of 2 is sufficient.

Table 2: The MPE for different expansion orders.

Expansion order	1	2	3
MPE	0.1100	0.1096	0.1096

The other optimization is to convert the arithmetic format into fixed-point, which is realized through adding **ap_fixed** header and typedef primitives. It is found that the intermediate values of attention models have various dynamic ranges. The accumulation of exponential values when calculating Softmax function would consume more integer bits while the matrix multiplication requires more fractional bits. Thus, single type of fixed-point format can hardly guarantee the arithmetic precision. To address this issue, two types of fixed-point formats are used. The information are given in Table , which results in negligible performance degradation and significant hardware consumption reduction.

Table 3: Two types of fixed-point format in attention models.

Type	Total bits	Integer part	Fractional part
Matrix multiplication	21	2	19
Accumulation in Softmax	14	11	3

The functionality of the associated implementation is verified using the testbench. The synthesis results of **Opt. 1** and the baseline designs are shown as follows:

Table 4: Synthesis results and throughput for various designs.

Design	Est. Clk (ns)	Clock Cycles		Utilization				Speedup
		Latency	Interval	BRAM	DSP48E	FF	LUT	
Baseline	8.427	106, 905, 095	106, 905, 095	561	12	2, 793	4, 515	1×
Opt. 1	8.552	55, 811, 591	55, 811, 591	370	12	5, 072	13, 222	1.91×

Due to the shortened quantization bits, the BRAM consumption declines by 34%. The invesitigation of fixed-point arithmetic units also reduces the complexity. As a result, the required clock cycles are reduced and a 1.91× speedup factor is achieved.

5.2 Dataflow Optimization

The mentioned optimization schemes have saved the hardware consumption while improving the throughput. However, the bottleneck of the accelerator is still the large storage space of socre matrix. To alleviate this problem, I chose to re-organize and optimize the dataflow by increasing data reuse opportunities of multiplication between score matrix and output matrix.

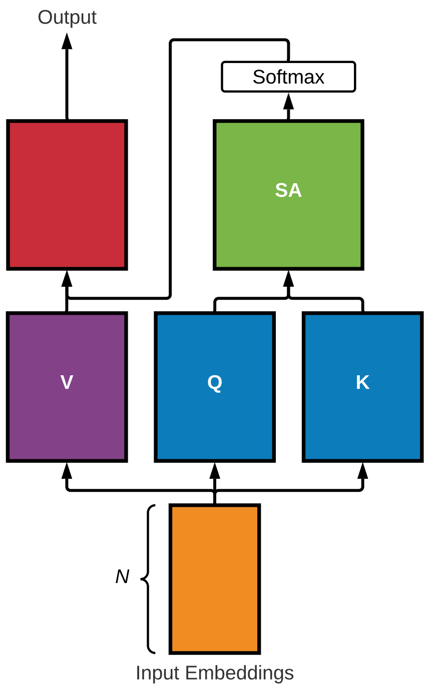

Figure 4: Redrawed dataflow of an attention layer.

Figure 4 redraws the dataflow in Figure 3. The score matrix requires a memory space of $\mathcal{O}(N^2)$. The memory requirement is reduced to a linear complexity by using the scheme in Figure 5. The score matrix is split into single row and the output matrix is calculated in a form of outer product. More specifically, each row of Q multiplies with K to obtain s specific row of score matrix. Then the row of score matrix is multiplied with the entire V and the partial sum of output matrix is yield. In this case, the partial sum of output can be reused for multiple times and each row of score matrix is used only once. The required memory space of score matrix becomes $\mathcal{O}(N)$.

The following code gives the detailed HLS implementation of optimized dataflow. The two loops, calculation of SS and calculation of output matrix, in design Opt. 1 are merged together. The outer product of output partial sum is realized by exchanging the loop order of the two inner most loops. One minor optimization is replacing the division operation of Softmax accumulation in design Opt. 1 by multiplying a constant factor $\frac{1}{512}$, which leads to no performance degradation. This because the accumulated values are around 512 and are much bigger than single exponential value of matrix S.

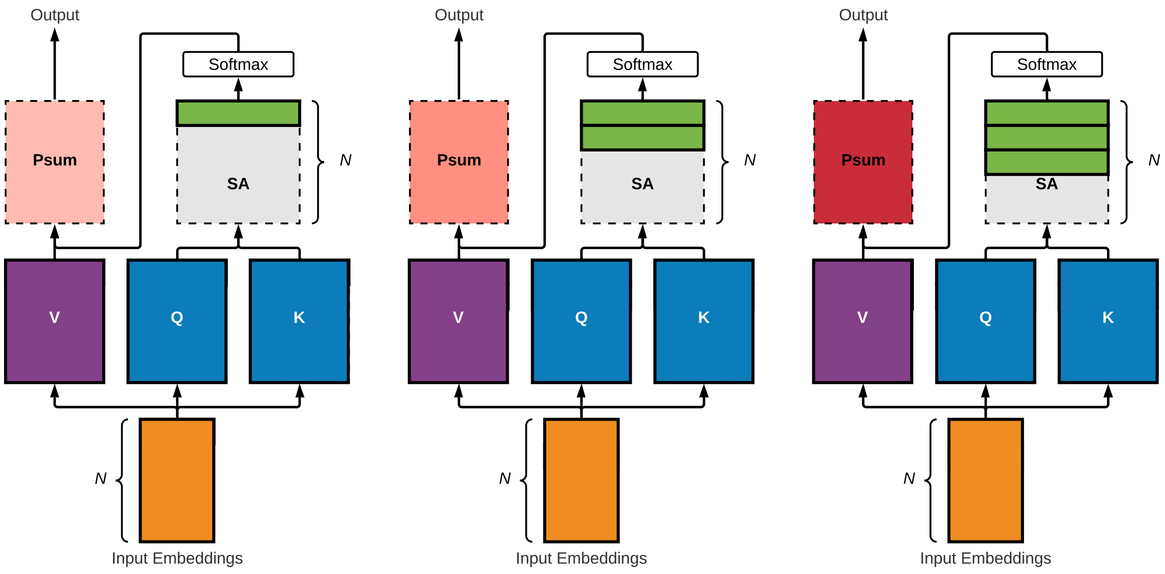

Figure 5: Optimized dataflow of an attention layer.

```
CALC_S_OUT:
  for (i = 0; i < SEQ_LENGTH; i++)
  {
        // Calculate one row of mat. S
        for (j = 0; j < SEQ_LENGTH; j++)
        {
          data_fixed psum = 0.0;
          for (k = 0; k < D_MODEL; k++)
          {
                psum += buf_Q[i * D_MODEL + k] * buf_K[j * D_MODEL + k];
          }
          data_fixed temp_exp;
          data_fixed temp_scaled_psum = psum * scaling;
          CORDIC_exp(&temp_scaled_psum, &temp_exp);
          buf_S_row[j] = temp_exp * data_fixed(0.001953125);
    }

        // Calculate partial sum of mat. output
        for (k = 0; k < SEQ_LENGTH; k++)
          for (j = 0; j < D_MODEL; j++)
            output_psum[i * D_MODEL + j] += buf_S_row[k] * buf_V[k * D_MODEL + j];
  }
```

The functionality of the associated implementation is verified using the testbench. The synthesis results of **Opt. 2** and the baseline designs are shown in Table 5. The optimized dataflow significantly shrinks down the BRAM consumption by 87.8%. The simplified Softmax calculation slightly reduces the FF and LUT consumption by 11% and 5%, respectively.The processing latency is reduced to 85% of design **Opt. 1**. As a result, the speedup factor increases to 2.24× compared to the baseline design.

Table 5: Synthesis results and throughput for various designs.

Design	Est. Clk (ns)	Clock Cycles		Utilization				Speedup
		Latency	Interval	BRAM	DSP48E	FF	LUT	
Baseline	8.427	106, 905, 095	106, 905, 095	561	12	2, 793	4, 515	1×
Opt. 1	8.552	55, 811, 591	55, 811, 591	370	12	5, 072	13, 222	1.91×
Opt. 2	8.552	47, 691, 269	47, 691, 269	45	12	4, 514	12, 529	2.24×

References

[1] How to code the transformer in pytorch. https://towardsdatascience.com/how-to-code-the-transformer-in-pytorch-24db27c8f9ec.

[2] Nicolas Carion et al. End-to-end object detection with transformers. *arXiv preprint arXiv:2005.12872*, 2020.

[3] Alexey Dosovitskiy et al. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020.

[4] Maximilian Ilse et al. Attention-based deep multiple instance learning. *ICML'18*, 2018.

[5] Nikita Kitaev et al. Reformer: The efficient transformer. *arXiv preprint arXiv:2001.04451*, 2020.

[6] Bin Li et al. Dual-stream multiple instance learning network for whole slide image classification with self-supervised contrastive learning. *arXiv preprint arXiv:2011.08939*, 2020.

[7] Zachary C Lipton et al. A critical review of recurrent neural networks for sequence learning. *arXiv preprint arXiv:1506.00019*, 2015.

[8] Ashish Vaswani et al. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[9] Xiaolong Wang et al. Non-local neural networks. In *CVPR'18*, 2018.