Fast Inference for Quantized Transformer Attention Heads

[Presented in Class]

CSE 237C Project Report
Carmen Dyck A59016086
Sanjayan Sreekala A59020260

Abstract

With the upsurge in large language model (LLM) applications, the demand for efficient inference methods for transformers is pressing. Current techniques focus largely on training acceleration, yet inferencing—calculating outputs with fixed weights—is critical for serving commercial LLMs at scale and for creating quality synthetic text data for training future versions of LLMs. This project develops an FPGA-accelerated inferencing method by implementing a 4-bit quantized attention head component in transformers using HLS. We explore optimization techniques aiming for minimal memory and resource usage.

Note: Implementation is at multiple branches. *Please see* ***APPENDIX*** *for code overview and information on branches.*

Introduction

In the realm of natural language processing, the emergence of large language models (LLMs) has marked a significant milestone, offering enhanced capabilities in generating and interpreting complex language patterns. However, the practical deployment of these models, especially in real-time applications, poses substantial challenges, primarily due to the computational and memory-intensive nature of their underlying architectures, particularly the transformer models.

This project addresses the critical need for efficient and scalable inference methods for transformers, an essential component of LLMs. While significant strides have been made in accelerating the training phase of these models, the inference phase—where the model utilizes fixed weights to compute outputs—remains a key area for improvement. Efficient inference is crucial not only for deploying commercial LLMs at scale but also for generating high-quality synthetic text data, vital for training future iterations of LLMs.

---

* This report includes sections refined with the assistance of GPT-4-V for enhanced clarity and grammar.

The focus of this project is on the development of an FPGA-accelerated inferencing method, specifically by implementing a 4-bit quantized attention head component in transformers using High-Level Synthesis (HLS).

This project does not implement a transformer nor does it implement computing gradients for training. The focus of this project, however, is to create repeatable attention layers that can take inputs of ‘N’ token vectors each ‘DMODEL’ size long and compute attention on this similar to how it would be in a encoder layer as introduced in [1] by Vaswani, Ashish, et al.

The project focuses on using 4-bit integer values for inputs, weights, and outputs, instead of the traditional floating-point numbers. This approach, including methods for 4-bit and 8-bit quantization, is gaining popularity. It's been shown that these techniques, as highlighted in sources [7] and [8], can be developed without significantly compromising performance.

Background

Modern transformers contain encoder and decoder layers that take inputs of ‘N’ token vectors each ‘DMODEL’ size long (N x DMODEL), computes attention and outputs N vectors of the same size. The salient parts of the encoder/decoder layers is as follows:

Projection

In a traditional encoder/decoder layer, each input vector is converted to Query (Q), Key (K), and Value (V) vectors by weights. These vectors can be of different sizes but in this project we restrict them to the same size as of the token vectors. The projection of the token inputs into these Q, K, and V matrices are given in **Figure 1** below.

---

* This report includes sections refined with the assistance of GPT-4-V for enhanced clarity and grammar.

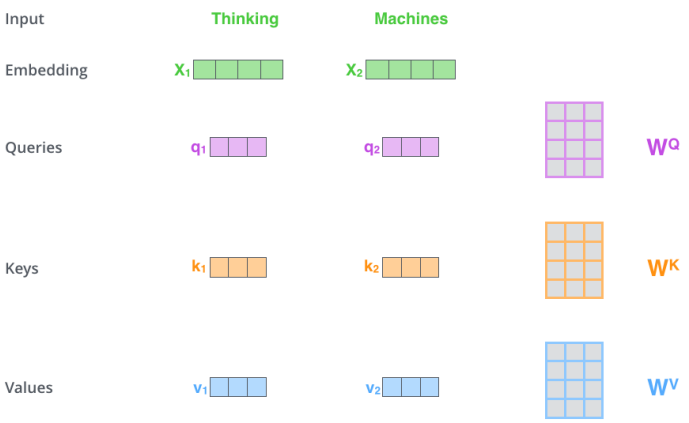

Figure 1. Projection. Image via Alammar, J. [2]

Attention with QKV

At the core of the transformer's architecture is the attention mechanism, specifically the Query-Key-Value (QKV) model. The attention mechanism computes the dot products of the Queries with Keys to determine the attention scores, which are scaled and normalized using the softmax function. These scores are used to weigh the corresponding Values, aggregating them to produce the final output of the attention layer. This mechanism allows the model to dynamically focus on different parts of the input sequence, crucial for understanding the context and relationships within the data. A visual representation of the QKV attention model is given in **Figure 2** below.

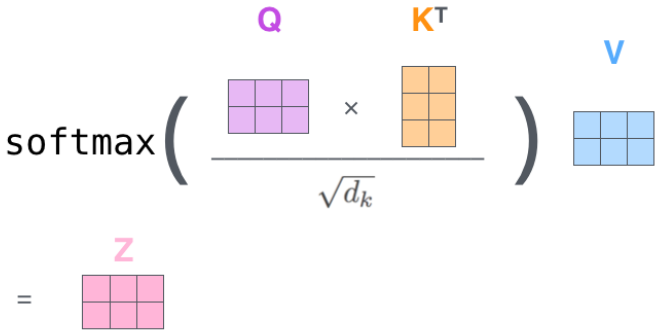

Figure 2. Attention. Image via Alammar, J. [2]

Softmax and LayerNorms

The softmax function [3], which transforms attention scores into a probability distribution, is a key component in the attention mechanism. Softmax is a resource-intensive part of the transformer model implementation as it involves multiple

---

* This report includes sections refined with the assistance of GPT-4-V for enhanced clarity and grammar.

exponentiations, summations and divisions. As described on Wikipedia, “The softmax function takes as input a vector z of K real numbers, and normalizes it into a probability distribution consisting of K probabilities proportional to the exponentials of the input numbers” [4].

$$P(y = j \mid \mathbf{x}) = \frac{e^{\mathbf{x}^\mathsf{T}\mathbf{w}_j}}{\sum_{k=1}^{K} e^{\mathbf{x}^\mathsf{T}\mathbf{w}_k}}$$

Eqn 1. Softmax. [4]

Layer Normalization, applied across features for each data point, normalizes the values of an output from a layer. There are different kinds of normalization (e.g RMSnorm [5]), but the simplest normalization involves ensuring values are in the range [0, 1]. Of course, this is still a compute- and resource-intensive operation.

We explore alternatives to both these compute-intensive operations in our implementation.

MLP

While the Multi-Layer Perceptron (MLP) (denser layer), which provides additional non-linear processing capabilities, is a standard component of modern transformer models, it is not implemented in this project. The MLP typically consists of fully connected layers that further process the output of the attention mechanism. Its omission in our FPGA-accelerated inference method is a design choice to streamline the focus on optimizing the QKV attention mechanism.

Quantization

Popular quantization techniques for transformers [7] [8] involve quantization post pre-training and identifying appropriate scaling for the quantized integers; however, this is beyond the scope of this project. In this project, we assume all integers are in the same scale and also the values are already quantized.

Deep Learning in HLS

Different frameworks and libraries, like HLS4ML [9], support deep learning training and inference on FPGA. Recently, implementations for transformers, as seen in [10], have become more popular. Building on this trend, we aim to implement an attention layer designed exclusively for inference, incorporating quantization and various optimization techniques.

---

* This report includes sections refined with the assistance of GPT-4-V for enhanced clarity and grammar.

Implementation

The project was implemented in C++ and synthesized using the Vitis HLS 2023 tool.

Code Structure

Please see the appendix for detailed information on the files. The code structure of the top-level *attention()* function is given in **Figure 3** below.

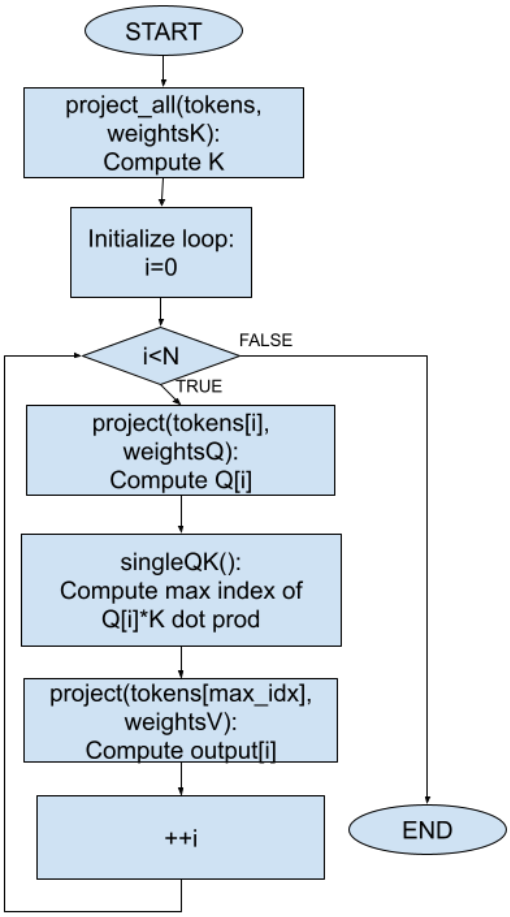

Figure 3. Top level *attention()* function code structure.

The top-level *attention()* function accepts four inputs: one (streamed) *tokens[][]* input matrix of size *NxDMODEL*, and three weights matrices–one each for the Query, Key, and Value intermediate matrices. It has one output: the streamed *outputs[][]* transformed matrix of size *NxDMODEL*. All values (tokens, weights, and outputs) are given as four-bit fixed width values. Inputs and outputs are streamed using HLS interface AXIs.

As is demonstrated in **Figure 3**, there are three main functions called in the C++ implementation of the top-level *attention()* function. The *project()* function is used to

---

* This report includes sections refined with the assistance of GPT-4-V for enhanced clarity and grammar.

calculate the Query, Key, and Value matrices from the initial *tokens* input. It accepts a single row from the *tokens* input matrix and a full weight matrix, then multiplies and normalizes the *tokens* row and transpose of each column of the weight matrix to stream a row vector output. The *project_all()* function accepts the full *tokens* matrix and calls *project()* on each row. This is used to calculate the Key matrix before the main loop of the *attention()* function begins. Finally, the *singleQK()* function multiplies a (streamed) row of the Query matrix with the K matrix, and returns the index of the max value in the resultant vector.

Note that intermediate values are stored with higher bit widths for improved accuracy.

Softmax versus Max

As detailed in the Background section, the softmax function is an exponential function that is used to select the indexes of the Values matrix to place in the output matrix [4]. Because the softmax requires computation of exponents and divisions, it is very complicated to implement in hardware. As a result, we elected to implement the attention using the max function. In the *singleQK()* function, the index of the maximum value in the matrix is calculated and returned. This decreases the accuracy of the overall model, but it significantly decreases latency and resource usage as well.

The accuracy of replacing softmax with max was tested empirically in python simulations with 4 bit quantization. We saw an average error rate increase tending to ~8.5% with this change. This might be significant for some transformer applications, but we postulate that increased model sizes and additional training with max may alleviate some of the precision issues.

Normalization

As normalization to [0, 15] involves scaling with addition/subtraction and then division, we opted to simplify this by replacing the division with bit shifts. The subtraction is done as it would be in a regular normalization step, and then the amount of shifting required to shift the max value in the vector to be less than 16 is identified. This shifting is then applied to all values in the vector.

HLS Optimization

Several HLS optimizations were applied to help meet resource and latency constraints. Primarily, complete array_partition and loop unrolling were applied for the computation of Query and Value matrices. This allowed for multiple dot products to be performed simultaneously, minimizing the latency of the matrix multiplications, at the expense of

---

* This report includes sections refined with the assistance of GPT-4-V for enhanced clarity and grammar.

increased resource consumption. The loops to be unrolled were carefully considered (only the simplest loops were unrolled), since unrolling can cause drastic increases in resource consumption, and as the input size scales, this is unsustainable.

Inputs and outputs to the top-level *attention()* function as well as the Query, Key, and final output matrices to the *project()* and *singleQK()* functions were streamed. Streaming inputs and outputs alongside the dataflow pragma can allow multiple operations to occur at once, decreasing the overall latency.

The entire project was implemented using 4-bit fixed point data values. This allowed the HLS tool to automatically optimize any multiplications to use shifts and adds, which eliminates the need for DSPs and decreases project latency.

Testbench

Sanity tests for all functions were created simultaneously with the model. These ensure each function is compilable and functional.

The final top level test case is divided into 2 parts: a python script that creates test cases with random input and expected outputs with softmax, and a cpp testbench that tests the attention implementation against these test cases.

The outputs and the expected outputs are both normalized to range (0-1) and then the accuracy is recorded as average RMSE over 10 iterations.

Results

The accuracy of the model for different numbers of tokens are given in **Figure 4** below.

---

* This report includes sections refined with the assistance of GPT-4-V for enhanced clarity and grammar.

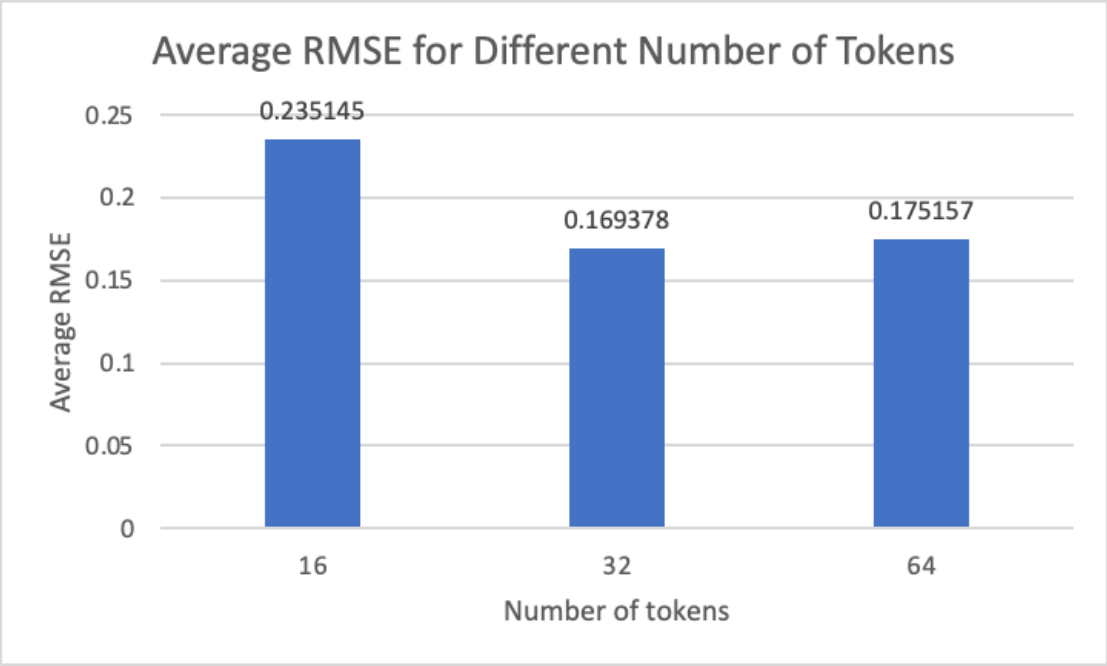

Figure 4. Average RMSE over 10 runs for different numbers of tokens are given above. Note that N=DMODEL, meaning that the number of tokens equals the number of values in the token vector (token matrix input is square).

As will be discussed in the “Challenges and Future Work” section, the model with N=DMODEL=64 was not synthesizable. The following synthesis results are thus given for just N=16 and N=32. **Figure 5** gives the percent usage of each resource on the PYNQ board for each N value.

---

* This report includes sections refined with the assistance of GPT-4-V for enhanced clarity and grammar.

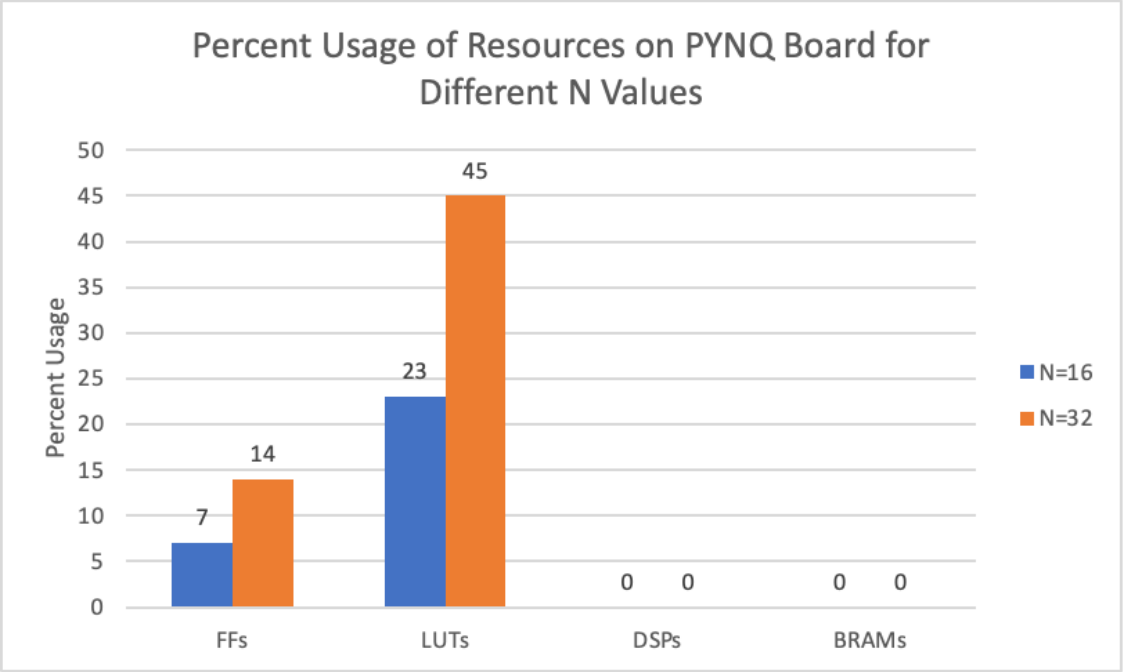

Figure 5. Percent usage of resources on the PYNQ board for N=DMODEL=16 and N=DMODEL=32. As the number of tokens and token size increases, the usage of flip-flops and lookup tables increases approximately linearly with N, and the number of DSPs stays at 0. The number of BRAMs for N=16 is 0, and for N=32 is 1.

The design does not use any DSPs because all multiplications within the matrix multiplies are optimized to use shifts and adds. Also, BRAM usage is low because arrays are completely partitioned to allow for simultaneous accesses during unrolled loops. A change to use matrix multiplication libraries, as detailed in the Future Work section, would decrease the LUT and FF usage and instead make use of DSPs.

The latency of the design as the number and size of the tokens increases is given in **Figure 6** below. As expected, the latency of the design scales quadratically with the number of tokens.

---

* This report includes sections refined with the assistance of GPT-4-V for enhanced clarity and grammar.

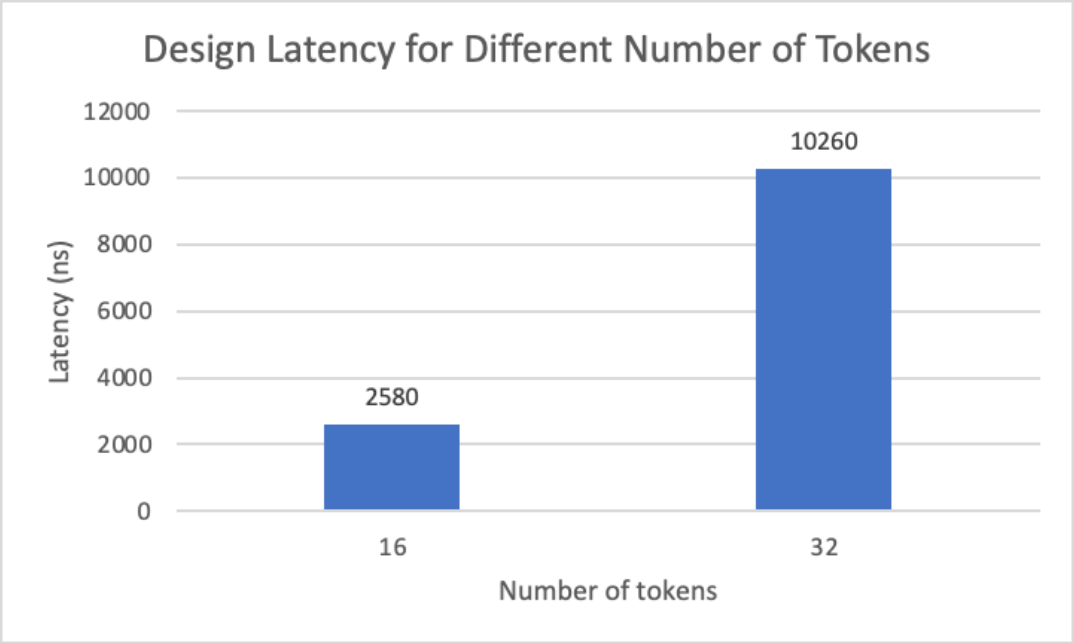

Figure 6. The latency of the design for N=DMODEL=16 and N=DMODEL=32. The latency increases significantly as the number of tokens increases.

Vivado Design & PYNQ DEMO

The design integration into the overall system is given in **Figure 7** below. Tokens are streamed in as inputs, and the transformed tokens are streamed out.

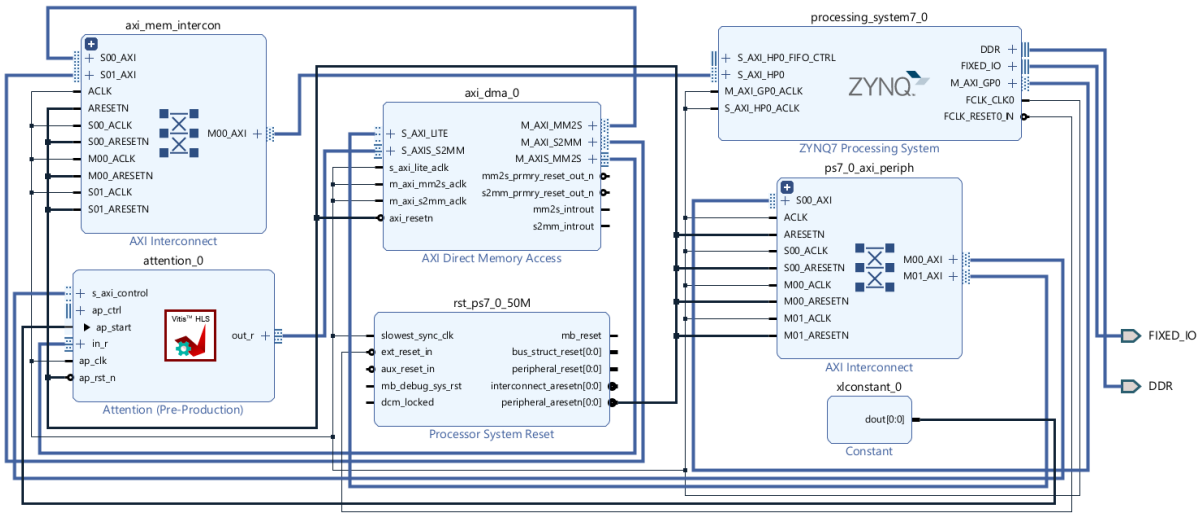

Figure 7. Design integration into the overall system

For demonstration purposes the code was adapted to use without internal streaming and to only include AXIS streaming for input and output (tokens) to the attention block.

---

* This report includes sections refined with the assistance of GPT-4-V for enhanced clarity and grammar.

Further we used N = 16 and DMODEL = 16 for the demo. The inputs and outputs to the PYNQ board are given in **Figures 8** and **9** below.

Please see branch *streaming_for_PYNQ* for this implementation. The .hwh, .bit and notebook files are also present in this branch.

```
In [23]: NUM_SAMPLES = 16*16

         tokens = np.array([np.random.randint(0, 16) for i in range(NUM_SAMPLES)])
         print(tokens.reshape(16, 16))
```

```
[[ 3 10  5 15  5  0  7 10  2  3 15 15  1 11 12  5]
 [ 3  9 10  0  0  8  9 12 14  7  4 14  2  2  9 12]
 [ 3 13 15 10  3 12  6  8 15 12  0  8  1 11  8  6]
 [ 5 14 10 14  9  6  0  9  8  6  3  0  1  9 13  8]
 [14  3  9  2 15  9 10  0  6  8  9  3 13  8  8 15]
 [ 2 10  9  4  7  6 15 15  8 14  8 12 13  3  3  3]
 [13  8  0  8  0 14 15  1 11  0 11  9 10 11 15 10]
 [ 6 11 12  6  7 10  9  0 15 10  7 11 13 13  3  0]
 [11  0 12 15 12  3  0  5 14 12 12 11  2  8 13  2]
 [11 13  7  3 12  1  2 15  5  9  3 14 14 14  9 11]
 [ 1  4 12  8 10  2 11  8  0  6  7  3  0 11  8  7]
 [ 2  8 15  3  4  8  0  2  4  0  7  0  1 15  5  8]
 [ 6  0  2 14  3 14  0 11 12 10 10  5  9  4  0 15]
 [ 2  9  0  6 15  8  0 11 11 11  9  6  5  3  8  9]
 [ 3  8  8  0  0 15  9  3  8  1  5 15 15  3 12  9]
 [ 0  5  8 12  5  9 14  9 15 10  2  9  3 14 12  7]]
```

Figure 8. The input to the PYNQ board was a randomized token array of size NxDMODEL.

---

* This report includes sections refined with the assistance of GPT-4-V for enhanced clarity and grammar.

```
In [26]: print(out_r.reshape(16, 16))
```

```
[[ 0 10  8  3  4  4  8  3  6  3  8  5  5  9  1  4]
 [ 0 10  8  3  4  4  8  3  6  3  8  5  5  9  1  4]
 [ 0  0  5  3  4  6  6  5 12  4 11  5 10  8  0  1]
 [ 0 10  8  3  4  4  8  3  6  3  8  5  5  9  1  4]
 [ 0 10  8  3  4  4  8  3  6  3  8  5  5  9  1  4]
 [ 0 10  8  3  4  4  8  3  6  3  8  5  5  9  1  4]
 [ 0 10  8  3  4  4  8  3  6  3  8  5  5  9  1  4]
 [ 0 10  8  3  4  4  8  3  6  3  8  5  5  9  1  4]
 [ 0 12  8  7  8  5 10  9 11  5 14  6 14 11  1  2]
 [ 0 15 11  9  6  7 15  1 12  3 10 11 11  9  4  3]
 [ 0 12  8  7  8  5 10  9 11  5 14  6 14 11  1  2]
 [ 0  0  5  3  4  6  6  5 12  4 11  5 10  8  0  1]
 [ 0 10  8  3  4  4  8  3  6  3  8  5  5  9  1  4]
 [ 0 10  8  3  4  4  8  3  6  3  8  5  5  9  1  4]
 [ 0 10  8  3  4  4  8  3  6  3  8  5  5  9  1  4]
 [ 0 10  8  3  4  4  8  3  6  3  8  5  5  9  1  4]]
```

Figure 9. The output of the PYNQ was a normalized array of transformed tokens.

The code for this project can be found at this link, and branch structure is given in the appendix.

Challenges and Future Work

One of the largest challenges in this project was optimizing for N=DMODEL=64. When N=DMODEL=64, the synthesis time significantly increased, taking several hours on our standard systems. This is understandable, given the extensive number of resources required for synthesis, (memory scales as approximately N*N*DMODEL), resulting in around 200,000 multiplications.

Another challenge was the implementation of softmax. Softmax is a complicated generalized logistic function that was difficult to implement in hardware, so this project was implemented using the max function instead. This decreased the accuracy of the project. Future work would include improving the accuracy of the implementation by using softmax instead of max. Alternatively, the model could continue to be trained using an implementation with max, which could improve the accuracy of results without needing to implement softmax.

An option for future work would be to use libraries for matrix multiplication. Since matrix multiplication libraries are optimized for performance, this would be a useful option for

---

* This report includes sections refined with the assistance of GPT-4-V for enhanced clarity and grammar.

decreasing resource usage and increasing latency. A useful library would be the AMD DSPLib [6], which offers a matrix multiply. It optimizes memory usage by using a tiling pattern, which would be a useful option for the larger N=DMODEL=64 (and higher) implementations.

Finally, due to time constraints, this project was scoped to be tested only on random data. Future work could include using real datasets and multiple attention layers to build a usable model.

Conclusion

The attention layer is a critical component of a transformer neural network that helps to detect contextual relationships. Current work on transformers focuses on training model acceleration, but this project involves interference optimization on hardware. This project models a 4-bit quantized attention head on an FPGA using HLS, which is critical for the creation and speedup of transformers like LLMs. Optimizations include the usage of max instead of softmax, using fixed-point input and weights, and HLS optimization including array partitioning and loop unrolling. The final output is synthesized for 16 and 32 token inputs, and further resource optimization is required to synthesize for 64 token inputs. Future work includes further training the model to improve accuracy using max instead of softmax, making use of HLS libraries, adding further layers, and testing with real datasets.

---

* This report includes sections refined with the assistance of GPT-4-V for enhanced clarity and grammar.

REFERENCES

[1] Vaswani, Ashish, et al (2017). "Attention is all you need." *Advances in neural information processing systems* 30
[2] Alammar, J. (n.d.). The Illustrated Transformer. Retrieved December 10, 2023, from https://jalammar.github.io/illustrated-transformer/
[3] Goodfellow, Ian; Bengio, Yoshua; Courville, Aaron (2016). "6.2.2.3 Softmax Units for Multinoulli Output Distributions". Deep Learning. MIT Press. pp. 180–184. ISBN 978-0-26203561-3.
[4] Wikipedia. (2023, December 12). Softmax function. Retrieved December 10, 2023, from "https://en.wikipedia.org/wiki/Softmax_function"
[5] Zhang, Biao, and Rico Sennrich (2019). "Root mean square layer normalization." *Advances in neural information processing systems* 32
[6] AMD (2023, October 19). "Matrix Multiply." AMD Adaptive Computing Documentation Portal. Retrieved December 10, 2023, from https://docs.xilinx.com/r/en-US/Vitis_Libraries/dsp/user_guide/L2/func-matmul.html.
[7] Dettmers, T., Pagnoni, A., Holtzman, A., & Zettlemoyer, L. (2023). Qlora: Efficient finetuning of quantized llms. arXiv preprint arXiv:2305.14314.
[8] Dettmers, T., Lewis, M., Belkada, Y., & Zettlemoyer, L. (2022). Llm. int8 (): 8-bit matrix multiplication for transformers at scale. arXiv preprint arXiv:2208.07339.
[9] Duarte, J., Han, S., Harris, P., Jindariani, S., Kreinar, E., Kreis, B., ... & Wu, Z. (2018). Fast inference of deep neural networks in FPGAs for particle physics. Journal of Instrumentation, 13(07), P07027.
[10] Peng, H., Huang, S., Chen, S., Li, B., Geng, T., Li, A., ... & Ding, C. (2022, July). A length adaptive algorithm-hardware co-design of transformer on fpga through sparse attention and dynamic pipelining. In Proceedings of the 59th ACM/IEEE Design Automation Conference (pp. 1135-1140).

APPENDIX

The code is divided into different branches. The repository can be found at https://github.com/spsanps/CSE237C-Project .

The main branches are:

- main_with_tests
    - Main codebase with all the files and test cases for N and DMODEL = 16, 32, and 64 with hls stream implemented internally
- streaming_for_PYNQ

---

* This report includes sections refined with the assistance of GPT-4-V for enhanced clarity and grammar.

    - Codebase with axis stream implementation at top level for PYNQ demo at N, DMODEL = 16. No HLS streaming internally. ***.hwh, .bit*** and ***.ipynb*** files are checked in here.

Important Files

- QKV.cpp
    - Implements a single Q dot product on K
- QKVProj.cpp
    - Implements Conversion of tokens into Q K and V
- dotProd.cpp
    - Implements dot Product
- attention.cpp
    - Implements attention
- attention.h
    - Contains definition for data types and sizes
- weight*.h
    - Weights of projection matrices at each DMODEL
- softmaxVsMax.ipynb
    - Estimates error of using max instead of softmax with quantized values
- create_testcases.ipynb
    - Generate .txt expected input output tokens with softmax and normalization for use in test2.cpp
- test_*.cpp
    - Sanity checks for each function
- test.cpp
    - Sanity checks for attention.cpp
- test2.cpp
    - Error check for attention.cpp

PYNQ_DEMO (only present in streaming_for_PYNQ branch)

- .hwh, .bit and .ipynb for running attention on PYNQ

---

* This report includes sections refined with the assistance of GPT-4-V for enhanced clarity and grammar.