Hardware Accelerator for Transformer Attention Mechanism

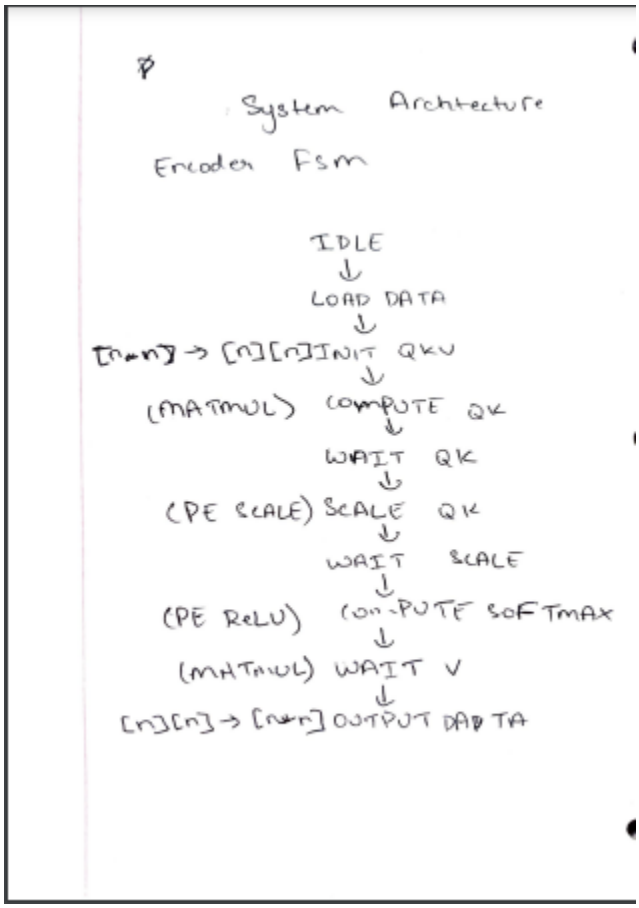
Chirag Maheshwari
ECE 498
Fall 2024

Architecture Description

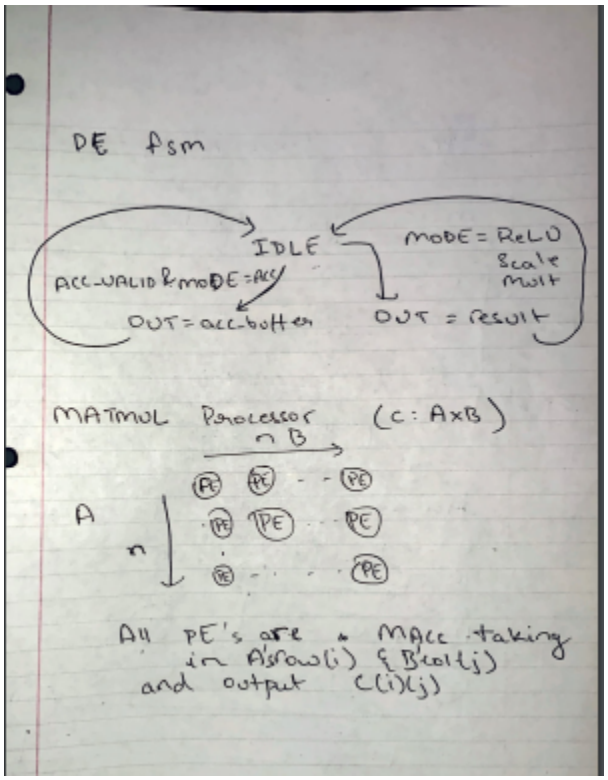
For my final project for ECE 498, I implemented a hardware accelerator with a specialized architecture for computing Transformer attention mechanisms. The design focuses on efficient matrix operations and parallel processing while maintaining configurable parameters to support different precision requirements, matrix dimensions, and processing element configurations.

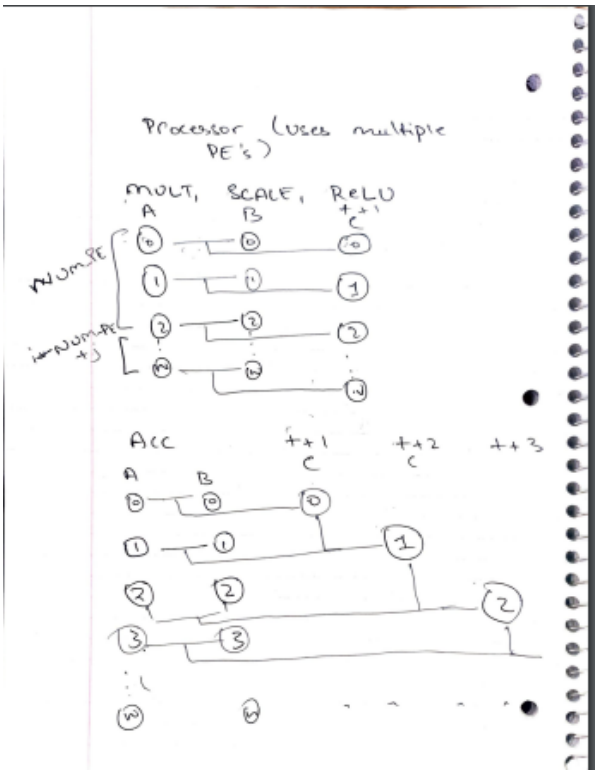
A. Core Architecture Components

The system architecture consists of four primary computational blocks, each optimized for specific operations within the attention mechanism:



- Initialize QKV Vectors: These vectors are initialized using input data. Since my design has access to limited resources, I initialize these vectors using the input matrix itself instead of weight matrices from BRAM.
- Query-Key (Q-K) Multiplication Unit: This block implements the QK^T matrix multiplication.
- Scaling Unit: Following the QK multiplication, this dedicated unit performs the critical scaling operation (division by $\sqrt{dk}$) required in attention mechanisms. The scaling helps prevent the dot products from growing too large, maintaining numerical stability.
- Softmax Processing Block: This specialized unit handles the normalization of attention scores. It implements an optimized version of the softmax function through parallel exponential approximation units and a normalization circuit.
- Attention-Value (A-V) Multiplication Unit: The final computational block performs matrix multiplication between the normalized attention scores and the value matrix, producing the output attention vectors.





The system implements a straightforward data flow architecture that minimizes data movement and maximizes throughput. Input data streams through a series of carefully orchestrated stages:

- Initial data buffering occurs in dedicated input registers, where incoming serial data is assembled into matrix format.
- The QKV matrices are formed in parallel storage elements, enabling simultaneous access during computation.
- Intermediate results flow through a pipeline of computational units, with each stage operating concurrently when possible.
- Output data is serialized through a streaming interface, allowing continuous operation with minimal stalling.

Design Methodology

- I applied principles learned from the course, including fixed-point representations, pipelining, and parallel processing to create hardware with lower latency and high throughput.
- I applied principles learned from the course, including fixed-point representations, pipelining, and parallel processing to create hardware with lower latency and high throughput.

A) Pipelining

My design uses a mostly scalar pipeline for ease of development and simplicity.

- Data Loading Stage: Implements efficient data ingestion through a serial-to-parallel conversion mechanism. This stage buffers incoming data and organizes it into the required matrix format.
- Computation Stage: Comprises multiple sub-stages that operate in parallel where possible:
    - Matrix multiplication pipelines
    - Scaling operations
    - Softmax computation
    - Final multiplication and accumulation
- Result Generation Stage: Manages the conversion of computed results back into a serial stream for output.

Each pipeline stage includes handshaking mechanisms to ensure proper data synchronization and prevent data hazards. The top-level module also implements an interface similar to AXI-Lite for proper data synchronization.

B. Verification

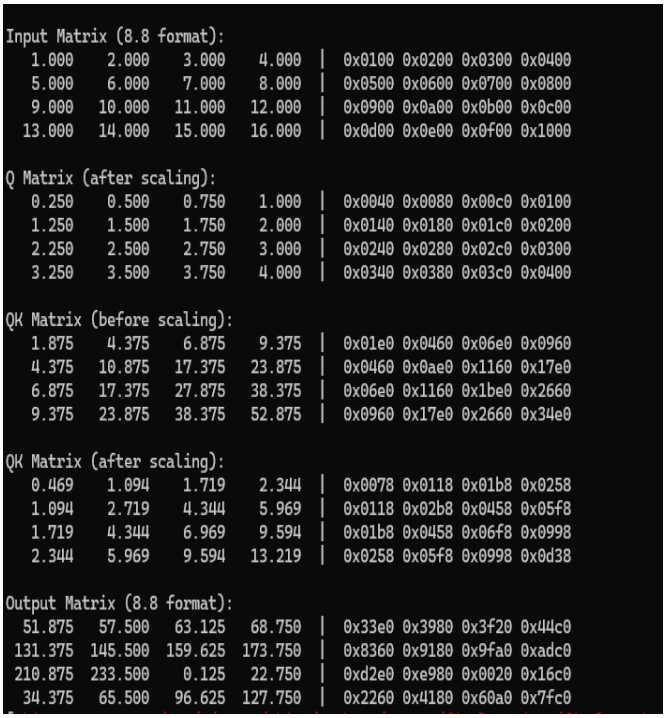
I verified the design, particularly the matrix multiplication and MAC processors, with a testbench and against a reference script in Python. This script accounted for the accelerator's behavior, the limitations of the fixed-point representation, and overflow handling with ReLU. All relevant design and simulation files are included in my submission. I ran into issues with my installation of Vivado when trying to use and configure Zynq SoCs such as those on the Pynq Z2. I also realised that using the Zynq SoC IP/block design would make it harder for me to test my actual RTL. But I nonetheless created an AXI IP and wrapper for my accelerator.

```
Processing completed at time 1365000
Starting output sequence at time 1365000
Received output[0][0] = 33e0 at time 1385000
Received output[0][1] = 3980 at time 1395000
Received output[0][2] = 3f20 at time 1405000
Received output[0][3] = 44c0 at time 1415000
Received output[1][0] = 8360 at time 1425000
Received output[1][1] = 9180 at time 1435000
Received output[1][2] = 9fa0 at time 1445000
Received output[1][3] = adc0 at time 1455000
Received output[2][0] = d2e0 at time 1465000
Received output[2][1] = e980 at time 1475000
Received output[2][2] = 0020 at time 1485000
Received output[2][3] = 16c0 at time 1495000
Received output[3][0] = 2260 at time 1505000
Received output[3][1] = 4180 at time 1515000
Received output[3][2] = 60a0 at time 1525000
Received output[3][3] = 7fc0 at time 1535000

Test Results:
Input Matrix (Fixed Point 8.8 format):
   1.000    2.000    3.000    4.000
   5.000    6.000    7.000    8.000
   9.000   10.000   11.000   12.000
  13.000   14.000   15.000   16.000

Output Matrix (Fixed Point 8.8 format):
  51.875   57.500   63.125   68.750
 131.375  145.500  159.625  173.750
 210.875  233.500    0.125   22.750
  34.375   65.500   96.625  127.750
```

Testbench



Reference

C. Parallelism Implementation

The design maximizes parallel operation through several key strategies:

- Matrix Multiplication Units: Matrix multiplication is done using systolic arrays where each unit contains a MAC processor operating in parallel. With n^2 MACs, we effectively achieve a complexity of O(n).
- Processors and Processing Elements: Processors use a set number of processing elements. They take in two input streams and process them in bursts of NUM_PE elements until all elements are processed. This parallelism cannot be fully exploited when computing a multiply and accumulate. For this, the processor makes execution sequential where each PE executes only when it receives its accumulated input.

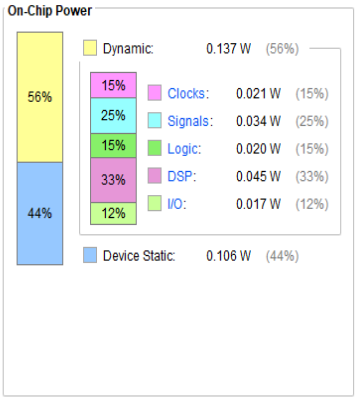
Utilization, Power and Timing

Summary

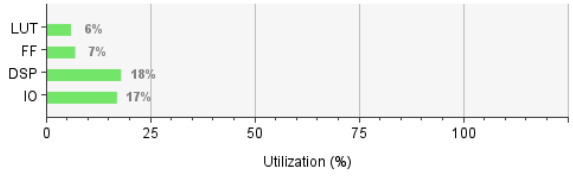
Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

Total On-Chip Power:	**0.243 W**
Design Power Budget:	**Not Specified**
Process:	typical
Power Budget Margin:	**N/A**
Junction Temperature:	**27.8°C**
Thermal Margin:	57.2°C (4.8 W)
Ambient Temperature:	25.0 °C
Effective ϑJA:	11.5°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Low

Launch Power Constraint Advisor to find and fix invalid switching activity



Resource	Utilization	Available	Utilization %
LUT	3441	53200	6.47
FF	6959	106400	6.54
DSP	40	220	18.18
IO	21	125	16.80



Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0.788 ns	Worst Hold Slack (WHS):	0.010 ns	Worst Pulse Width Slack (WPWS):	3.500 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	14433	Total Number of Endpoints:	14433	Total Number of Endpoints:	6992

All user specified timing constraints are met.

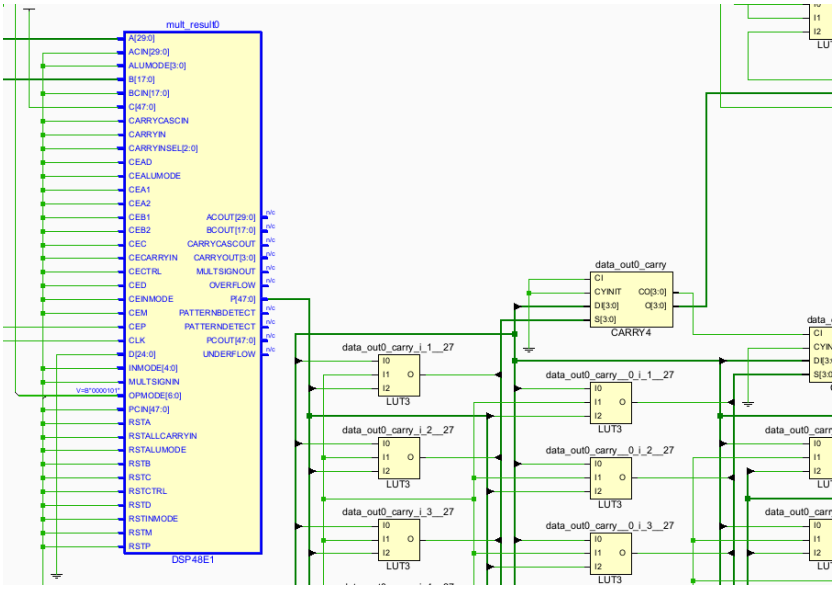
Conclusions and Future Power Considerations

I have several ideas for how this work can be improved upon.

- The top level can be parallelized by making the pipeline superscalar. The computations for QK^T and V matrices don't overlap meaning both processes can work in parallel and get synchronised for the final matrix multiplication before the softmax. If weight matrices were being used, the initialisation of Q,K,V would be

‘parallel’ as it is now (it is currently simply manipulating the input matrix using bitwise operators).

- A more bare-bones systolic architecture could be created instead of full MACs for each PE, enabling better weight and partial sum reuse. MACs were chosen since the MAC operation had been tested on the processor module and could easily be implemented for matrix multiplications without having to worry about data movement between PEs since all the MACs could do their computations indepenently.



- I
- For added optimization, I can better utilize on-board resources based on the target hardware. On inspection of my synthesized design, while Vivado correctly infers DSP48E1 units for multiplication, it performs accumulation using adders and registers instead of integrating all operations into the slice. For MAC processors used in matrix multiplication, slightly more efficiency can be achieved by using a tree structure for partial sum reductions. We can perform element-wise multiplication first, then accumulate separately.
- This design needs to be more rigorously tested for throughput, latency and performance on all kinds of matrix sizes and clock speeds. While the design works in simulation, the implemented version might run into timing issues.

Conclusion

The journey of working on this project has been extremely rewarding and fulfilling. It has given me a basic understanding of how to map algorithms into hardware, and has left me with an appetite to understand how modern ASICs work. It was very satisfying to see a working matrix multiply after much trial and error, and I look forward to implementing and learning about more advanced hardware designs and architectures in the future.