ProTEA: Programmable Transformer Encoder Acceleration on FPGA

Ehsan Kabir[1], Jason D. Bakos[2], David Andrews[1], Miaoqing Huang[1],

[1]Department of Electrical Engineering and Computer Science University of Arkansas, Fayetteville, Arkansas

[2] Department of Computer Science and Engineering, University of South Carolina, Columbia, USA

ekabir@uark.edu

Introduction

- Transformers are crucial for NLP, Translation, and CV applications.
- Internal parallelism making them suitable for hardware acceleration.
- Existing accelerators focus on sparse or custom architectures.
- Lack of flexibility and parallelism for different TNN applications.
- High computational complexity and memory demand require efficient tiling and coding.
- GPUs have high power consumption, low computational efficiency, and underutilized memory bandwidth for dense computations.
- ***ProTEA*** is a runtime programmable accelerator tailored for the dense computations of most state-of-the-art transformer encoders.

Objectives

Novel Accelerator Architecture: High DSP utilization for high parallelism. and low latency.

Efficient Tiling Strategy: Supports large models in on-chip memory by applying tiling in MHA and FFN layers.

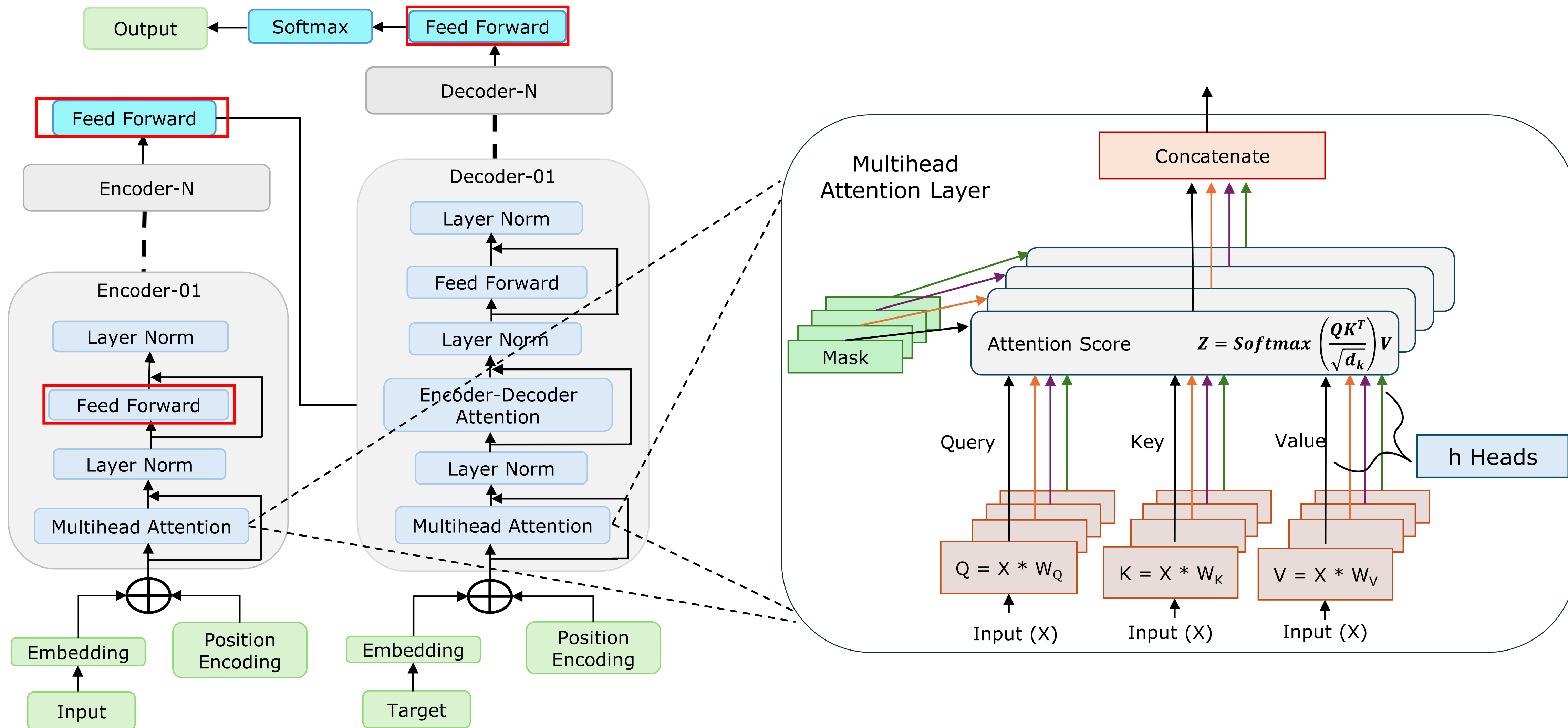
Optimized HLS Code: Efficient HLS coding for better performance within limited resources and compilation time.

Parameterized HLS Code: Allows design-time customization of key parameters.

Runtime Programmability: Enables dynamic parameter adjustments without hardware re-synthesis.

ekabir@uark.edu

Background

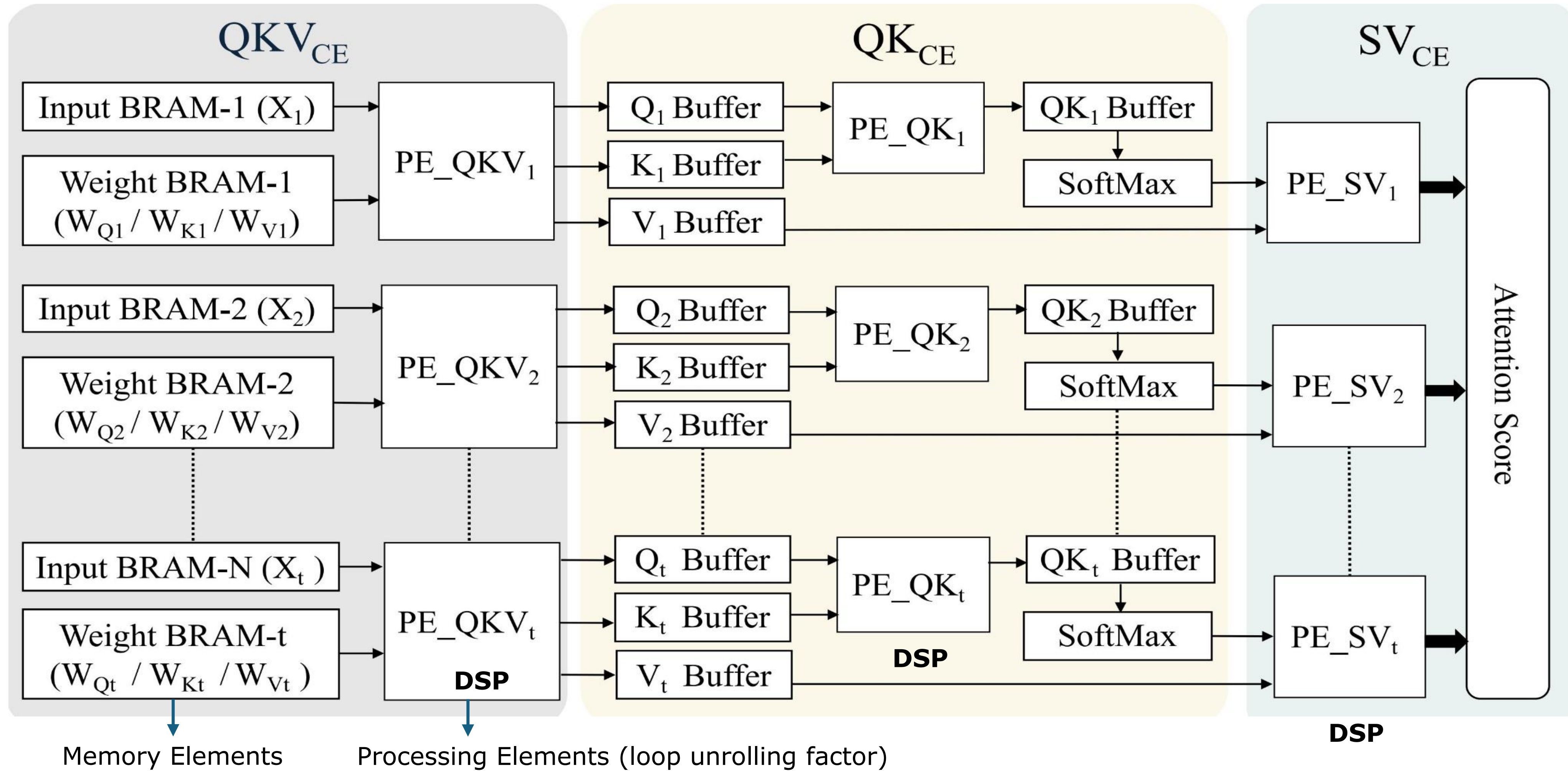


Overall Architecture

- Designed accelerator in C language using Vitis HLS 2022.2.1.
- Two main modules- Attention and Feedforward Network modules.
- The system was designed in Vivado 2022.1.2 using the accelerator IP exported from HLS and other IPs (timer, UART, processor, etc.).
- Inputs and weights fetched from off-chip high-bandwidth memory (HBM) via AXI4 master interfaces.
- MicroBlaze (μB) processor enables programming TNN hyperparameters dynamically without hardware re-synthesis.
- Processor controls with the accelerator through AXI-lite slave interface.
- Software running on the processor was written in C on the Vitis IDE tool.

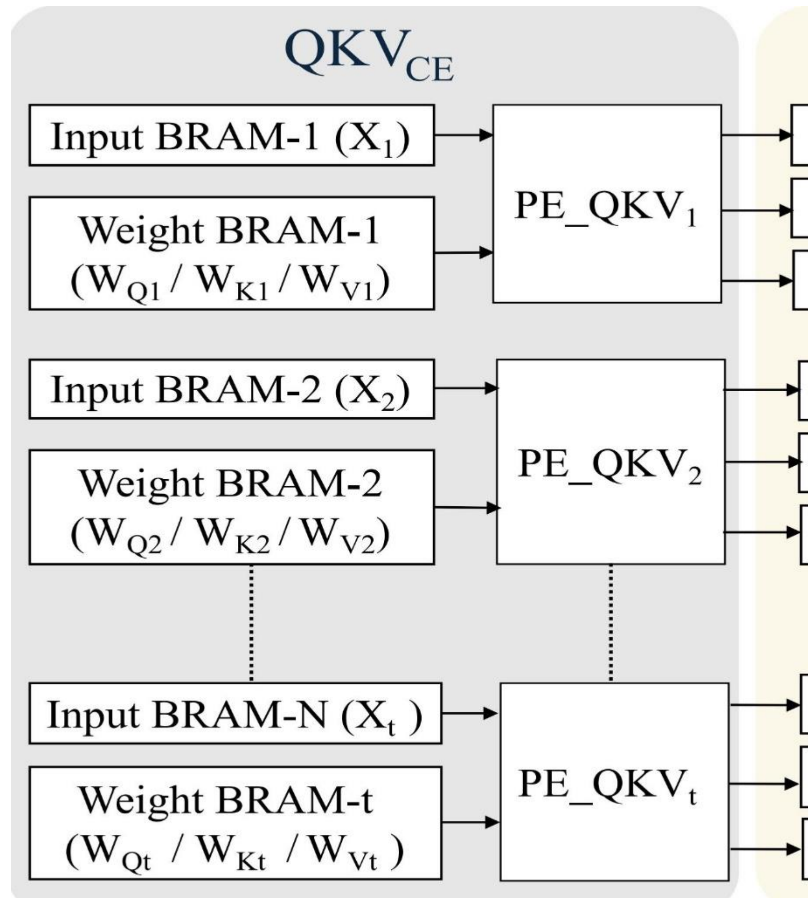
ekabir@uark.edu

Attention Module



ekabir@uark.edu

QKV Engine



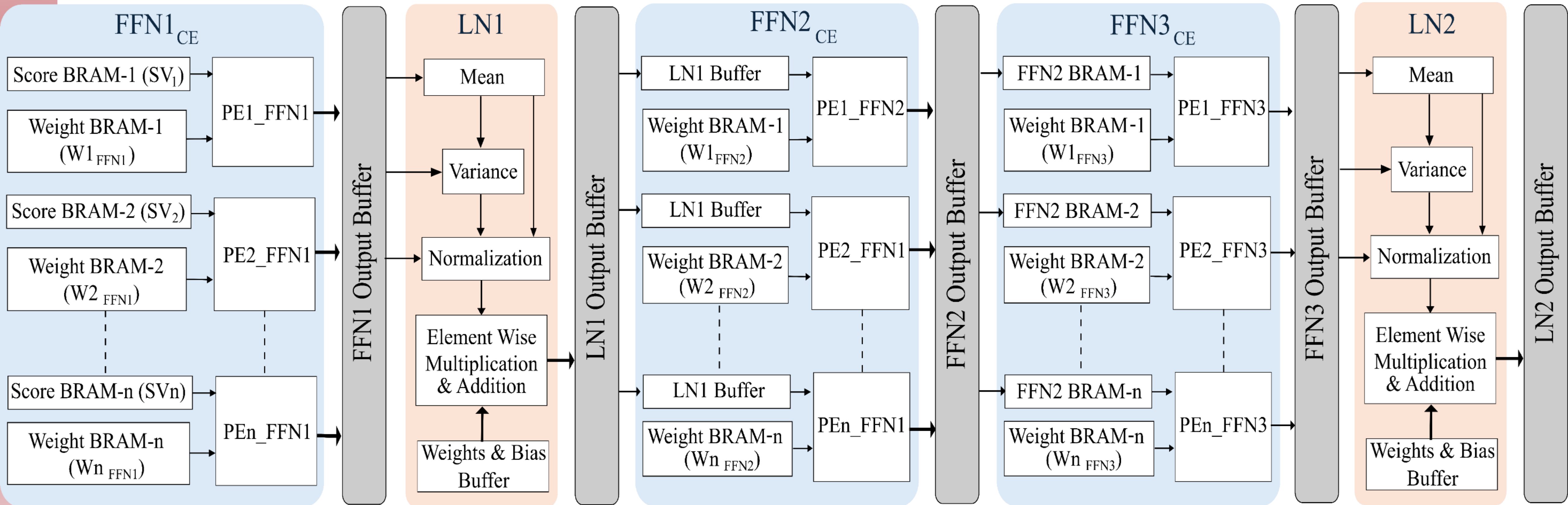
(embedding dimension/tile size) no. of iterations

Algorithm 1 Q, K, V Calculation

1: **for** $i \leftarrow 1$ to $Sequence\ Length$ **do**
2:     **#pragma HLS pipeline off**
3:     $S_q \leftarrow 0$
4:     $S_k \leftarrow 0$
5:     $S_v \leftarrow 0$
6:     **for** $k \leftarrow 1$ to $\frac{Embedding\ Dimension}{Number\ of\ Heads}$ **do**
7:         **#pragma HLS pipeline II = 1**
8:         **for** $j \leftarrow 1$ to $Tiles\ in\ MHA$ **do**
9:             $S_q \leftarrow S_q + x[i][j] \times w_q[k][j];$
10:             $S_q \leftarrow S_q + x[i][j] \times w_k[k][j];$
11:             $S_q \leftarrow S_q + x[i][j] \times w_v[k][j];$
12:         **end for**
13:         $Q[i][k] \leftarrow Q[i][k] + S_q;$
14:         $K[i][k] \leftarrow K[i][k] + S_k;$
15:         $V[i][k] \leftarrow V[i][k] + S_v;$
16:     **end for**
17: **end for**

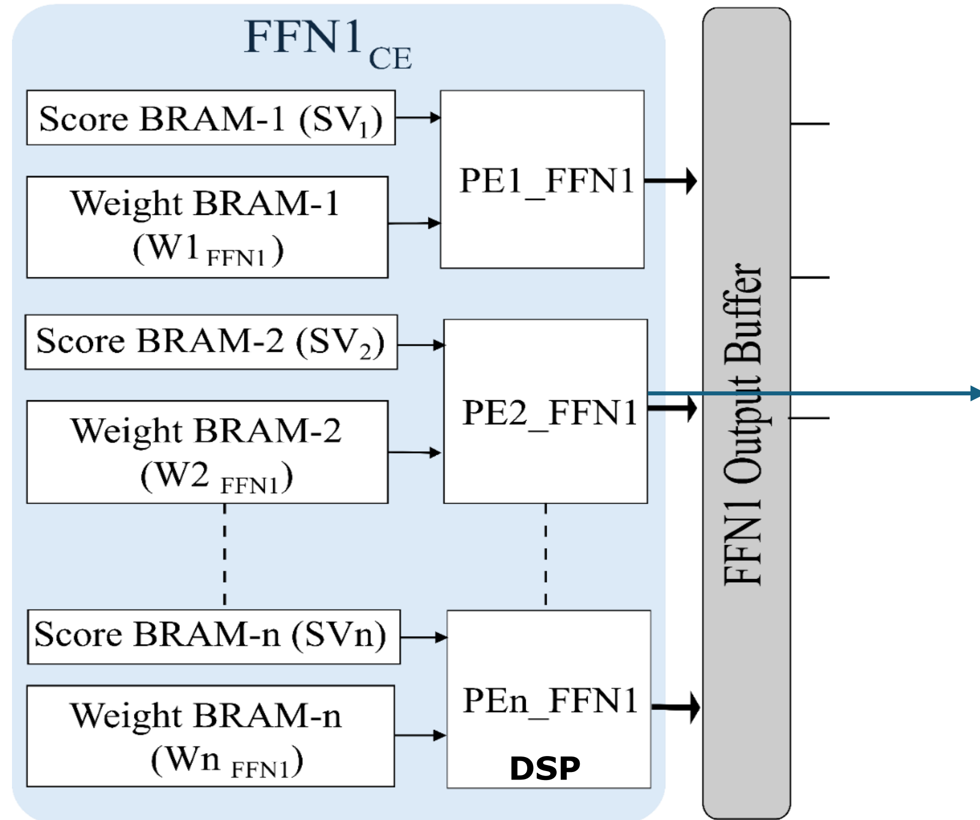
ekabir@uark.edu

FFN Module



ekabir@uark.edu

FFN1 Engine

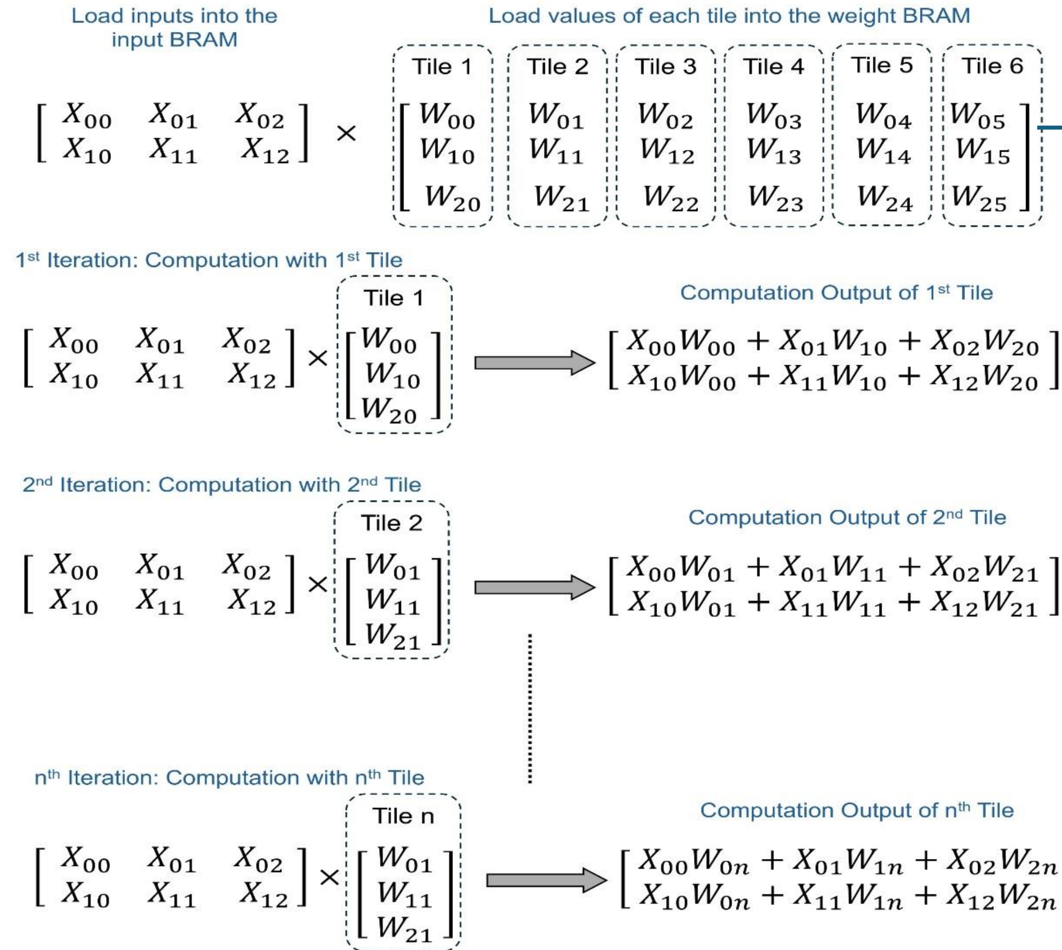


(tile size x tile size) no. of iterations

Algorithm 4 FFN Computation Example

1: **for** $i \leftarrow 1$ to *Sequence Length* **do**
2: **#pragma HLS pipeline off**
3: $m \leftarrow index \times \frac{Embedding\ Dimension}{Tile\ no.\ FFN}$
4: **for** $j \leftarrow 1$ to $\frac{Embedding\ Dimension}{Tile\ no.\ FFN}$ **do**
5: **#pragma HLS pipeline II = 1**
6: $sum \leftarrow 0$
7: **for** $k \leftarrow 1$ to $\frac{Embedding\ Dimension}{Tile\ no.\ FFN}$ **do**
8: $sum \leftarrow sum + inputs[i][k] \times weights[k][j]$;
9: **end for**
10: $output[i][m] \leftarrow output[i][j] + sum$;
11: $m \leftarrow m + 1$;
12: **end for**
13: **end for**

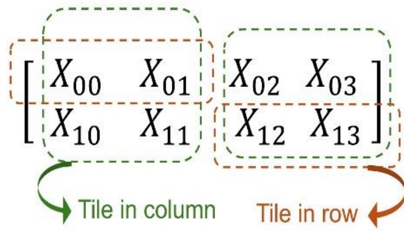
Tiling on MHA



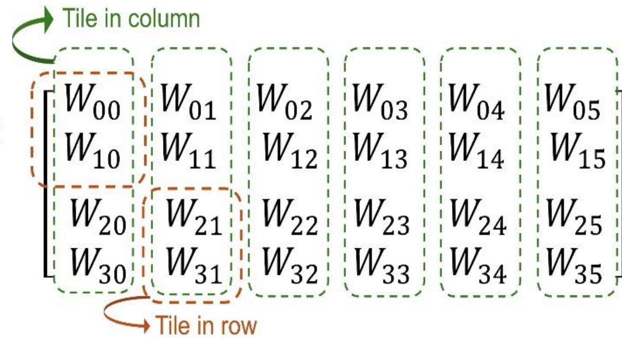
- Tiling is applied only along the columns of the weight matrix.
- Each matrix is loaded (embedding dimension/tile_size) times.
- Output of each tile is stored in intermediate buffers.
- Final output is the cumulative sum of the results computed across all tiles.

Tiling on FFN

Load inputs into the input BRAM



Load values of each tile into the weight BRAM



1st Iteration: Computation with 1st Tile

Tile - 01, Column

$[X_{00}\ X_{01}] \times \begin{bmatrix} W_{00} \\ W_{10} \end{bmatrix} \Rightarrow [X_{00}W_{00} + X_{01}W_{10}]$

2nd Iteration: Computation with 2nd Tile

Tile - 02, Column

$[X_{02}\ X_{03}] \times \begin{bmatrix} W_{20} \\ W_{30} \end{bmatrix} \Rightarrow [X_{02}W_{20} + X_{03}W_{30}]$

Output Column = Output for 1st tile + Output for 2nd tile + …+ Output for nth tile in column

3rd Iteration: Computation with 1st Tile

Tile - 01, Row

$[X_{10}\ X_{11}] \times \begin{bmatrix} W_{00} \\ W_{10} \end{bmatrix} \Rightarrow [X_{10}W_{00} + X_{11}W_{10}]$

4th Iteration: Computation with 2nd Tile

Tile - 02, Row

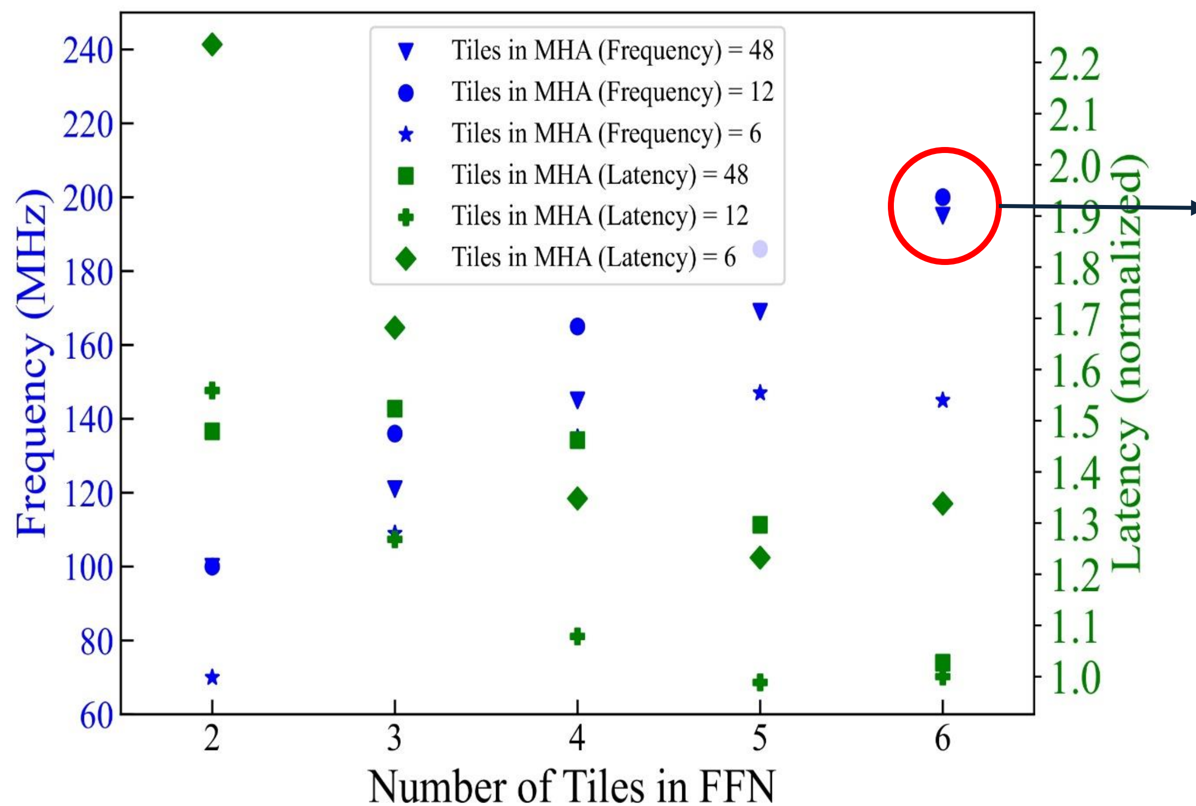
$[X_{10}\ X_{11}] \times \begin{bmatrix} W_{20} \\ W_{30} \end{bmatrix} \Rightarrow [X_{10}W_{20} + X_{11}W_{30}]$

Output Row = Output for 1st tile + Output for 2nd tile + …+ Output for nth tile in Row

- Tiling is applied along both the rows and columns of the weight matrix.
- Two loops of iterations, each iterates (embedding dimension/tile_size) times.
- Each matrix is loaded 4*(embedding dimension/tile_size) times.
- Output of each tile is stored in intermediate buffers.
- Outputs are first accumulated along the columns and then along the rows for all tiles.

ekabir@uark.edu

Choosing Tile Size



- The number of tiles in MHA was varied from 6 to 48, and for each MHA tile count, the number of tiles in FFN was varied from 2 to 6.
- The optimal configuration for achieving the highest frequency (blue color) and lowest latency (green color) was 12 tiles in MHA and 6 tiles in FFN.
- Maximum frequency of 200 MHz for 12 tiles in MHA and 6 tiles in FFN.
- This is optimal for HLS, allowing for efficient array partitioning within a reasonable compilation time (approximately 36 hours) for a state-of-the-art transformer encoder.

ekabir@uark.edu

Results

Overall Result Showing Runtime Programmability

Test no.	Sequence Length	Embedding Dimension	Number of Heads	Number of Layers	Data Format	DSPs	LUTs	FFs	Latency (ms)	GOPS
#1	64	768	8	12	8bit fixed	3612 (40%)	993107 (76%)	704115 (27%)	279	53
#2			4						285	51
#3			2						295	49
#4	64	768	8	8	8bit fixed	3612 (40%)	993107 (76%)	704115 (27%)	186	80
#5				4					93	159
#6	64	512	8	12	8bit fixed	3612 (40%)	993107 (76%)	704115 (27%)	186	36
#7		256							95	18
#8	128	768	8	12	8bit fixed	3612 (40%)	993107 (76%)	704115 (27%)	560	54
#9	32								165	44

Runtime programmable parameters

Synthesized once, Fixed Resource

Results

Comparison with FPGA Accelerators

Accelerator	Precision	FPGA	DSP	Latency (ms)	GOPS	(GOPS/DSP)× 1000	Method	Sparsity
Peng et al. [21]	–	Alveo U200	3368	0.32	555	164	HLS	90%
ProTEA	Fix8	Alveo U55C	3612	4.48	79	22		0%
Wojcicki et al. [23]	Float32	Alveo 250	4351	1.2	0.0006	0.00013	HLS	0%
ProTEA	Fix8	Alveo U55C	3612	0.425	0.0017	0.00045		
EFA-Trans. [25]	Int8	ZCU102	1024	1.47	279	272	HDL	0%
ProTEA	Fix8	Alveo U55C	3612	5.18	83	23	HLS	
Qi et al. [28]	–	Alveo 200	4145	15.8	75.94	18	HLS	0%
ProTEA	Fix8	Alveo U55C	3612	9.12	132	37		
FTrans [29]	Fix16	VCU118	5647	2.94	60	11	HLS	93%
ProTEA	Fix8	Alveo U55C	3612	4.48	79	22		0%

Normalized throughput

- ***ProTEA*** achieved 2.8× and 1.7× improvements in speed and GOPS, respectively, compared to **Wojcicki et al**. [23] and **Qi et al**. [28].
- EFA-Trans [25] used HDL methods, resulted in more efficient hardware with a lower level of abstraction, making it 3.5× faster than ***ProTEA***.
- Peng et al. [21] applied a high sparsity of 90% to their model, achieving a 14× speedup over ***ProTEA***. The same level of sparsity on ***ProTEA*** will make it 1.4x faster.
- FTRANS [29] compressed the model by 93%. The same compression would make ***ProTEA*** 9.4× faster.

ekabir@uark.edu

Results

Cross Platform Comparison

TNNs	Works	Platform	Frequency	Latency (ms)	Speed Up
Peng et al. #1	[21]	INTEL I5-5257U CPU	2.7 GHz	3.54 (Base)	1
		JETSON TX2 GPU	1.3 GHz	0.673	5.3×
		ProTEA FPGA	0.2 GHz	4.48	0.79×
Wojcicki et al. #2	[23]	NVIDIA TITAN XP GPU	1.4 GHz	1.062 (Base)	1
		ProTEA FPGA	0.2 GHz	0.425	2.5×
EFA-Trans. #3	[25]	INTEL I5-4460 CPU	3.2 GHz	4.66 (Base)	1
		NVIDIA RTX 3060 GPU	1.3 GHz	0.71	6.5×
		ProTEA FPGA	0.2 GHz	5.18	0.89×

- ***ProTEA*** is 2.5× faster than the NVIDIA TITAN XP GPU for model #2.
- ***ProTEA*** is 0.79× and 6.65× slower than the Intel I5-5257U CPU and JETSON TX2 GPU respectively for model #1 because this study [21] applied a pruning technique.
- For model #3, ***ProTEA*** performed slower than the Intel I5-4460 CPU and NVIDIA RTX 3060 GPU, potentially due to the use of aggressive sparsity and omission of certain computations in the referenced work.

Conclusion

- A flexible FPGA-based accelerator for a transformer neural network (TNN) encoder layer using a high-level synthesis (HLS) tool.
- On the Alveo U55C platform, high utilization of resources such as DSPs for enhanced parallelism and minimized latency.
- Supports runtime programmability, allowing it to adapt to various topologies without requiring re-synthesis.
- An efficient tiling technique was implemented to accommodate large models in on-chip memory while preventing the overutilization of computational resources.
- Outperforms some CPUs and GPUs in terms of speed and throughput despite operating at a lower frequency and lacking sparsity optimizations.
- Achieved 1.3 to 2.8× speed up compared to the fastest state-of-the-art FPGA-based accelerators.

ekabir@uark.edu

Thanks

Q/A