Accelerating the Attention Layer of the Transformer Model using an Optimized Dataflow

Zeyu Chen, Kevin Hutto, Jonathan Nativ
Georgia Institute of Technology
ECE 8893 Spring 2022 Project Report

***Abstract*—The Transformer model introduced by Google is the leading Natural Language Processing Model. The multi-headed self-attention mechanism emphasized the semantic correlation between words in a sentence. However, as the input length and batch group keep increasing, large intermediate matrix footprints generated within the attention layer result in severe memory costs. Activation-activation operators here have space complexity $O(BNH^2)$, which far exceeds the memory bandwidth most machine supports. To address this bottleneck, Kao proposed Fused Logic Attention Tiling (FLAT) [1]. The high-dimensional matrix multiplication in the attention layer is performed with low granularity tiling. This approach stages the piece-wise intermediate tensor on chip and leverages the higher on-chip bandwidth. However, FLAT is still a software level optimization. In our research, we realize there are great hardware optimization opportunities in FLAT. We accelerate it with three HLS techniques – Loop Unroll/Pipeline/Array Partition, Systolic Array, and Data Quantization – and deploy it on FPGA.**

I. Introduction

A recent development in machine learning architectures is the Transformer model, making use of the attention mechanism. The attention mechanism was first used to attempt Natural Language Processing (NLP) for tasks such as translation in a manner more akin to human language processing. Original processing focused on analysis at the granularity of entire sentences or blocks of words. The attention mechanism instead focuses on a moving window, focusing word by word, with weights for surrounding tokens based on the distance from the current word. With this paradigm shift, the attention mechanism is able to translate much larger input strings than previous attempts. While software improvements have made the attention mechanism viable, hardware acceleration has not been fully explored. In FLAT [1], Kao proposed a novel solution, Fused Logic Attention Tiling (FLAT), tiling activation-activation operators in the attention layer with low-level granularity. This method has been proven in both improving computational latency and memory usage. It enables GPU platforms to support large input sequence calculations without running into resource allocation limitations. However, we then realize there are great opportunities in optimizing the nested inner loops proposed in the first version FLAT. We want to optimize FLAT, the Attention Layer and then Transformer Model from a hardware perspective. Three main techniques are utilized in this paper:

This paper makes the following key contributions:

- *We design a back-to-back systolic array architecture for FLAT, which fuses three operations in the attention layer and maximizes the data reuse and throughput.*
- *We implement our design in Vitis HLS 2021 on FPGA and achieve both performance and resource improvement compared to the baseline result.*

Note that since both the encoder and decoder in the original Transformer model contains the attention layer, any performance improvement we achieve here is actually 2x speed up.

II. Problem Description

As shown in Fig.1, there are two kinds of operators in multi-headed attention mechanism, activation-weight operators Q/K/V/O and activation-activation operators L/S/A. Activation-activation operators are computational expensive and has strict memory requirements. All input matrices are four-dimensional and as the sequence length grows, intermediate tensor size and number of memory accesses will all increase quadratically. The operation intensity here far exceeds what on-chip resources can support and has to rely on off-chip resources, which eventually lowers the model performance.

A recent approach in utilizing the attention mechanism for transformer models is the FLAT algorithm [1]. FLAT implements a tiled version of the attention mechanism shown in Fig. 2. It performs the tiling as shown in Fig. 1. However, in the first version code of FLAT, there exists multiple nested loops for both breaking large input into small tiles and performing complex high dimension matrix operation, which is not hardware friendly. Hardware optimization opportunities are hidden and wasted in the original FLAT. In this paper, we propose a hardware implementation of FLAT leveraging the optimization opportunities. We want to build a hardware-friendly pipeline model that maximizes the instruction-level parallelism on limited resources. After that, we will show the evaluation result we get comparing the optimized FLAT, original FLAT, and original Transformer code published by Google.

III. Background

In this section we will discuss previous work in implementing the transformer model and provide an overview of Systolic Array designs.

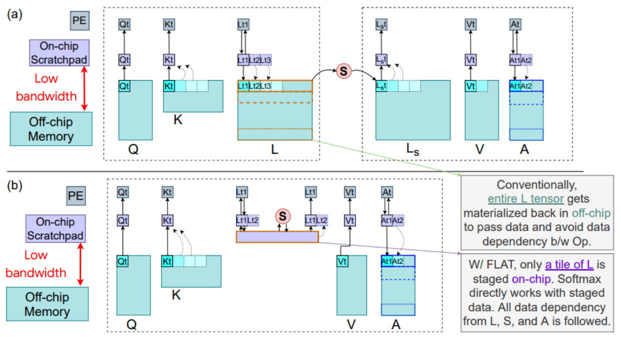
(a)
PE
On-chip Scratchpad
Low bandwidth
Off-chip Memory
Q
K
L
S
L_s
V
A
(b)
Conventionally, entire L tensor gets materialized back in off-chip to pass data and avoid data dependency b/w Op.
W/ FLAT, only a tile of L is staged on-chip. Softmax directly works with staged data. All data dependency from L, S, and A is followed.

Fig. 1. Diagram of how FLAT works. Taken from [1]

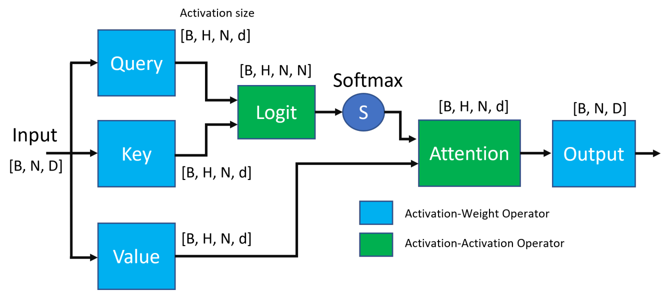

Fig. 2. Diagram of how FLAT works. Taken from [1]

A. Transformer Model and Attention Mechanism

The transformer model neural network, utilizing the attention mechanism, was proposed to enable language processing on large input data streams [2]. While prior work in language processing for tasks such as translation made use of convolutional networks, the transformer model instead utilizes a moving window, with larger "attention" weights given for words that appear closer to the current target word. The convolutional neural networks attempted to act concurrently on the entire input, which created bottlenecks on the size an input could be.

B. Fused Logit Attention Tiling (FLAT)

Multi-headed attention mechanism often requires high-dimensional tensor operations, which generate large intermediate memory footprints. Fetching data from off-chip can create a memory bottleneck and severely lower overall performance. FLAT, on the other hand, is a tiled version of the attention mechanism, shown in Fig. 2. It performs tiling on large tensors and fuses three expensive Activation-to-Activation operators–Logit, SoftMax, and Attend–in the attention layer. A small batch of data can be fully processed and produce the final data without requiring intermediate storage, while all operations can be kept on-chip and benefits from high bandwidth. This can leverage the limited memory resources of many ASIC chips and reduces the memory requirements.

C. Systolic Arrays

Systolic arrays are an efficient hardware design technique for achieving high throughput while mitigating issues that arise with high levels of parallelism, such as routing complexity, fan-out, clock frequency, and energy consumption [3][4]. Systolic array architectures have been proposed for problems with high arithmetic intensity, such as matrix multiplication and convolution. As a result, they have become a significant backbone of recent accelerator designs for machine learning workloads, an example of which is Google's Tensor Processing Unit (TPU).

IV. Approach

Now we will discuss our approach to providing latency improvement to the FLAT attention model by utilizing an FPGA. Original multi-headed attention mechanism requires three operations: Logit function, Softmax, and the Attention(L/S/A). These three functions are performed sequentially at first and has large intermediate footprint. FLAT tiles the matrix size and fuses these three operations, so our main focus for improvement is to create a corresponding hardware architecture for FLAT, before speeding up each of the three computations separately.

A. Systolic Array Design

1) Optimal Design: To handle the three operations we propose our pipelined back-to-back systolic array design, with the SoftMax function executed in-between Logit and Attention. We break the four-dimensional input into two-dimensional row granularity and further tile them into multiple iterations. The first systolic array performs a Logit operator and employs a weight stationary strategy as shown in Fig. 3. The operation performed here can be represented using the Tensorflow Einsum function as $tf.einsum("FH, TH->FT", Query, Key)$, where F and T are input Query and Key length and H is the split size per head. Key values are pre-loaded and fixed at each processing element(PE), so the PE array size is $T * H$. Elements of Query tensor and Bias tensor will pass through columns and rows of PEs, respectively, in the F iterations. For each iteration each active PE will pass the partial sum to its horizontal neighbor and the Query input to its vertical neighbor, indicated by the red and yellow arrow in the graph. Notice that Bias inputs only get added once at the first column of PEs and then get discarded, so we can also say the partial sums of PEs in the first column are initialized to Bias values. The rightmost column of PEs will write the output of the Logit operator to an intermediate buffer so that the Softmax can begin to be computed in a pipeline fashion. Value at this time should be

$$Logit = Bias_{A1} + KEY1 * a1 + KEY2 * a2 + KEY3 * a3...$$

The Softmax operation is then performed inside the intermediate Logit buffer without other data movement or storage. Next, the output is fed into the second systolic array, which performs the Attend operator and employs a weight stationary mechanism. The operation here is a simple matrix multiplication as $tf.einsum("FT, TH->FH", Logit, Value)$. We initially designed this to be output stationary, however, when we reviewed the synthesis report, we found the fan-out due to moving data from each PE to the output buffer to be too large. We decided to implement this as weight stationary to mitigate this problem. Shown in Fig. 4, elements of the Value tensor will be fixed in the PEs and Logit, from last two operations, will horizontally pass though the PEs. Outputs of the Attend operator accumulate along the rows of PEs and are finally shifted into the output in memory.

We tried to optimize our design before starting the implementation. For all three operators L/S/A, we expected Softmax(S) will be the performance bottleneck. We chose to use row-level stable Softmax, which requires the normalization process before doing natural exponential and division on each row. Though we included ”hls math” for optimization, we were still concerned Softmax would use most resources as division and exponential are expensive operations on an FPGA and dependency exists in all three loops. We realized the maximum element of each row could be found by first systolic array. Elements written to the same row of FT always arrive at the rightmost column in a diagonal shape, with one cycle slack. While the output is written to the Logit intermediate buffer, we perform one more data movement in this column. Outputs are vertically passed down to next PE and two outputs can thus be compared. Temporary max is latched in the max buffer and the final max can be obtained once the last element in the row arrives at the last PE. This optimization reduces one dependent loop in Softmax and as we pipeline the remaining two loops, the latency of Softmax reduces by a large margin.

2) Realistic Design: The ideal systolic array design described above requires almost full loop unrolling for Logit and Attend operations, and full array partitions of the input and output matrices. This is not realistic considering the limited resources and the performance will degrade as the input matrices scales up. To gain more control over resource utilization, we decided to break the systolic arrays into two levels, as shown in Fig. 5. We unroll the outer tiles as they are different actual PEs and we pipeline the inner tiles as they are ”pseudo” PEs. We tried multiple configurations in our testing environment and found out the configuration with 4x4 inner tiles and 16x16 outer tiles gives the best balance of latency and resource utilization.

B. Other Design Optimizations

A second improvement we believe is easily applicable to the FLAT algorithm is data quantization. We believe the attention mechanism can function with minimal reduction in translation performance when utilizing smaller data sizes. We quantized the original floating point data to 16-bit fixed point, and with our chosen reduced data size adjust the hardware architecture to take advantage of the improvement in data loading and processing times. DRAM burst read is successfully implemented and the implementation fetches 64 elements in one DRAM access. This reduces the loading latency by approximately 50 percent.

To achieve full parallelism of the program, we tried to double buffer all the functions, as shown in Fig. 6 so that once program is fully loaded, all operations can be executed in parallel with no dependency inside. However, further design explorations are needed here. As we continue optimizing the design, the performance bottleneck changes, and blindly paralleling functions may adversely affect the performance. Also, the current double buffer design is missing the data streaming model. We are actively working on this.

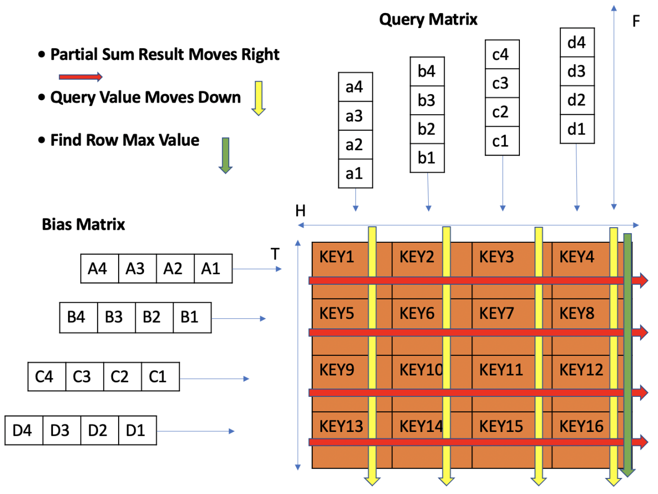

Fig. 3. Implementation of the Logit Function

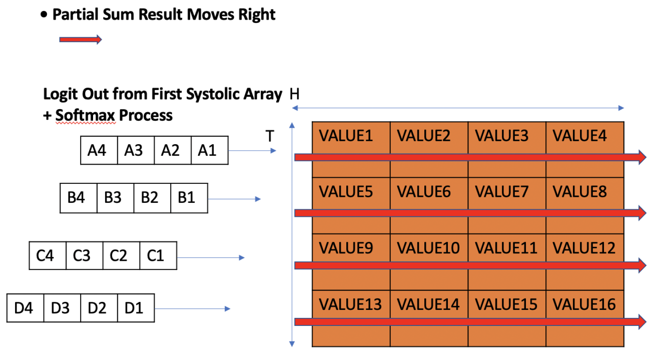
• Partial Sum Result Moves Right
Logit Out from First Systolic Array
+ Softmax Process
H
T
A4 A3 A2 A1
B4 B3 B2 B1
C4 C3 C2 C1
D4 D3 D2 D1
VALUE1 VALUE2 VALUE3 VALUE4
VALUE5 VALUE6 VALUE7 VALUE8
VALUE9 VALUE10 VALUE11 VALUE12
VALUE13 VALUE14 VALUE15 VALUE16

Fig. 4. Implementation of the Attention Mechanism

C. Design Summary

Our design perfectly follows FLAT algorithm and is further optimized to be hardware-friendly. Large input matrices are tiled into smaller components that fit in on-chip memory. Three operators L/S/A are fused together to allow input data to flow through all operations without generating a large intermediate memory footprint. However, we still believe there are many further optimization opportunities, which will be discussed in detail in the future work section.

V. Results

In this section we will discuss the results for our implementation. Our FLAT implementation was written in synthesizeable C++, and synthesized using Vitis HLS 2021. We

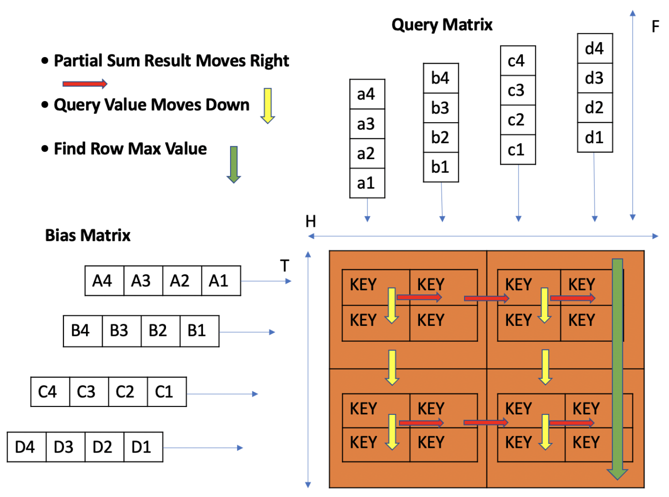

Fig. 5. Alternative Systolic Array to reduce resource utilization

synthesized our solutions to our target FPGA, the xc7z020-clg400-1, on the Pynq-Z2 development board.

The results for the simple optimization techniques are shown in Table I. We first implemented a basic hardware solution of the FLAT attention model. Due to the number of calculations required, this created a very long latency for the baseline solution. The baseline solution was also too large to fit on the Pynq-Z2 board, with 163% BRAM usage. These results are shown in Table I (b). We first attempted to improve latency by simply pipelining the loading of data from the DRAM. These results are shown in Table I (c). The large majority of the work in the circuit is the performance of the multiplication itself, so as expected this did not provide much of a decrease in latency but did increase LUT usage. We separately attempted to improve latency by pipelining and unrolling the matrix multiplications completed in the transformer. The results for pipelining the Fused Logit Operator function are shown in Table I (d), softmax shown in Table I (e), and the fused attention operator is shown in Table I (f). In Table I (e) we show the results for combining both the data loading and multiplication optimizations. With this basic combination without using the systolic array, we receive a latency speedup from our baseline of 88.7 times.

Next, in Table II we show our results for the Systolic array and double buffering optimizations. Note that Vitis HLS cannot accurately estimate latency with the double buffering optimization: double buffering introduces variable latency in each loop iteration, so the estimation is given as the range of the minimum iteration latency to maximum iteration, multiplied by number of loop executions and other loop overhead. Due to the size limitations of Pynq-z2, we were unable to receive a latency result for running on an actual board. However, we tried to manually compute the whole latency by summing up latency from each individual module. We believed the latency for non-partition case should be around 12s, and for 8x8 partition case, this number should be reduced

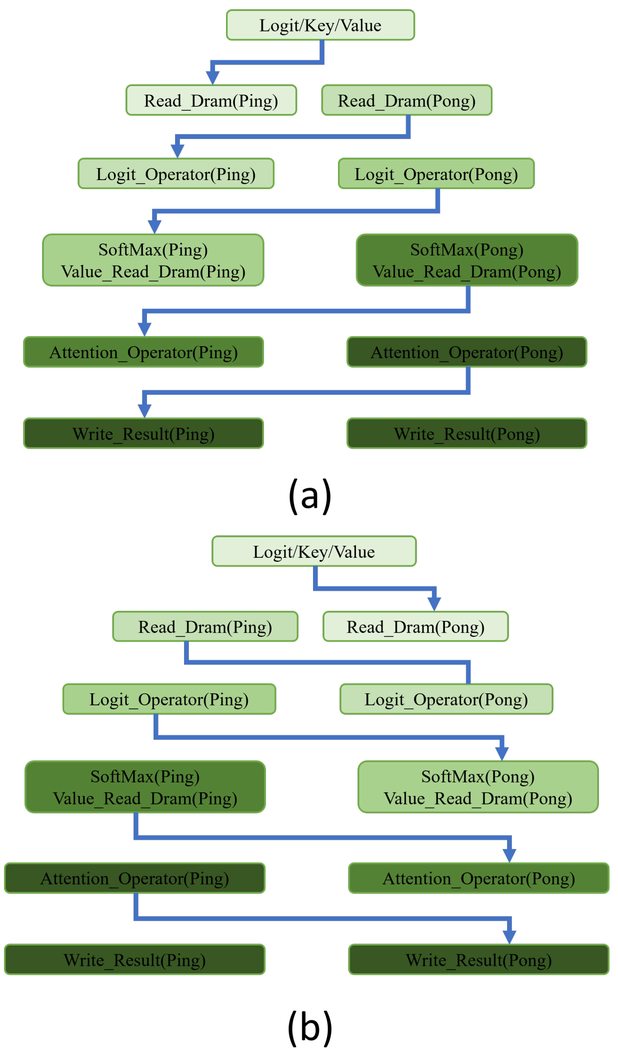

Fig. 6. Double buffer design showing how the two dataflows ((a) and (b)) are combined to decrease latency

to around 1.1s. We are actively running cosim design to verify our calculation.

VI. Future Work

There are many further improvements that can be made to accelerate in hardware the FLAT algorithm and, more generally, the Transformer model. The current implementation requires too many resources to fit on the Pynq-Z2 board. To provide flexibility to accelerate the Transfomer model on a small board, smaller tiling can be explored to reduce resource utilization. Currently the Logit function is the bottleneck of the design, with the greatest resource usage. Thus, the Logit function would be the prime focus for potential improvements. Also, during our experiment, we found that the Logit output matrix mentioned in the previous design has high sparsity. Using sparse matrix multiplication, we could further optimize the design, especially the second systolic array. If we can

TABLE I
FGPA UTILIZATION OF FLAT TRANSFORMER MODEL WITH BASIC OPTIMIZATIONS

	Latency	BRAM	DSP	FF	LUT
(a)	2.735e+11 ns	457(163%)	28(12%)	6353(5%)	10545(19%)
	Data Loading Optimizations				
(b)	2.696e+11 ns	457(163%)	28(12%)	14398(13%)	177192(333%)
	Multiplication Optimizations				
(c)	1.519e+11 ns	457(163%)	91 (41%)	8005 (7%)	14207 (26%)
(d)	1.519e+11 ns	457(163%)	28 (12%)	8347 (7%)	11713 (22%)
(e)	1.523e+11 ns	457(163%)	91 (41%)	7969 (7%)	14729 (27%)
	Combined Pipeline Optimizations				
(f)	3.081e+9 ns	457(163%)	154 (70%)	10148 (9%)	45304 (85%)

(a) Baseline hardware results. (b) Data loading optimizations (c) Fused Logit Operator optimizations (d) Softmax Operator optimizations (e) Fused Attention Operator optimizations (f) Combined optimizations

TABLE II
FGPA UTILIZATION FOR FLAT TRANSFORMER MODEL WITH ADVANCED OPTIMIZATIONS

	Latency	BRAM	DSP	FF	LUT
	Baseline Hardware results				
	273 s	457 (163%)	28 (12%)	6353 (5%)	10545 (19%)
	Compute Logic Function (No partition)				
	1.18e+7 ns	4	16	6934	8049
	Compute Attention Function (No partition)				
	8.02e+6 ns	4	16	6840	8249
	Compute Softmax Function (No partition)				
	2.14e+4 ns	0	52	164651	203838
	Systolic Array with Double Buffering				
(a)	0.190 s - 257.76 s	242 (86%)	84 (38%)	189423 (178%)	240349 (451%)
(b)	0.025 s - 258.63 s	1313 (468%)	880 (400%)	401730 (377%)	711396 (1337%)
(c)	0.031 s - 254.01 s	593 (211%)	464 (210%)	369197 (346%)	525846 (988%)
	Latency	BRAM	DSP	FF	LUT

Results for the Systolic array implementation of FLAT. (a) No partitioning. (b) PE partitioning 8x8. (c) PE partitioning 4x4.

only perform S/A operators on non-zero elements after L, we believe the latency and memory usage will further improve.

We also would explore data streaming rather than double buffering to lower BRAM usage, which is significant for the full parallelism of the program. This additionally benefits our systolic array design, as the peak bandwidth is currently only achieved in one cycle per systolic array execution. Using a dataflow would enable peak bandwidth for much longer without the wind-up and wind-down phases. Last, we only briefly explored data quantization in this project, and further improvements can be made with greater exploration of possible quantizations.

REFERENCES

[1] S.-C. Kao, S. Subramanian, G. Agrawal, and T. Krishna, "Flat: An optimized dataflow for mitigating attention performance bottlenecks," 2021.

[2] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, ser. NIPS'17. Red Hook, NY, USA: Curran Associates Inc., 2017, p. 6000–6010.

[3] R. Linderman and W. Ku, "A three dimensional systolic array architecture for fast matrix multiplication," in *ICASSP '84. IEEE International Conference on Acoustics, Speech, and Signal Processing*, vol. 9, 1984, pp. 782–785.

[4] J. de Fine Licht, G. Kwasniewski, and T. Hoefler, "Flexible communication avoiding matrix multiplication on fpga with high-level synthesis," in *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 244–254. [Online]. Available: https://doi.org/10.1145/3373087.3375296