AttentionLib: A Scalable Optimization Framework for Automated Attention Acceleration on FPGA

Zhenyu Liu*, Xilang Zhou, Faxian Sun, Jianli Chen, Jun Yu and Kun Wang†
State Key Lab of Integrated Chips & Systems, and School of Microelectronics, Fudan University, Shanghai, China
*23212020007@m.fudan.edu.cn, †kun.wang@ieee.org

***Abstract*—The self-attention mechanism is a fundamental component within transformer-based models. Nowadays, as the length of sequences processed by large language models (LLMs) continues to increase, the attention mechanism has gradually become a bottleneck in model inference. The LLM inference process can be separated into two phases: *prefill* and *decode*. The latter contains memory-intensive attention computation, making FPGA-based accelerators an attractive solution for acceleration. However, designing accelerators tailored for the attention module poses a challenge, requiring substantial manual work. To automate this process and achieve superior acceleration performance, we propose *AttentionLib*, an MLIR-based framework. *AttentionLib* automatically performs fusion dataflow optimization for attention computations and generates high-level synthesis code in compliance with hardware constraints. Given the large design space, we provide a design space exploration (DSE) engine to automatically identify optimal fusion dataflows within the specified constraints. Experimental results show that *AttentionLib* is effective in generating well-suited accelerators for diverse attention computations and achieving superior performance under hardware constraints. Notably, the accelerators generated by *AttentionLib* exhibit at least a 25.1× improvement compared to the baselines solely automatically optimized by Vitis HLS. Furthermore, these designs outperform GPUs in *decode* workloads, showcasing over a 2× speedup for short sequences.**

***Index Terms*—Transformer, FPGA, MLIR, Fusion, HLS**

I. Introduction

The self-attention mechanism is a critical component in the transformer-based models [4], which has achieved remarkable performance in natural language processing [5] and computer vision [8]. Nowadays, transformer-based large language models (LLMs) have demonstrated exceptional performance across various domains [6, 17]. As the length of the input sequence increases, the attention computation accounts for an increasing proportion of the computational load in model inference [15]. Expediting attention computations has emerged as a pivotal factor in accelerating transformer model inference, particularly for LLMs.

The inference process of LLMs typically comprises two distinct phases: *prefill* and *decode*. These phases can be effectively offloaded to separate devices to enhance overall throughput [20, 23]. The *prefill* phase necessitates extensive attention computation, which is a computationally intensive workload. In contrast, the *decode* phase entails low-batch, long-sequence attention computations, characterized by low parallelism yet high memory bandwidth requirements. Given these considerations, FPGAs emerge as an optimal choice for handling *decode* workloads owing to their low power consumption and adaptable reconfiguration capabilities. Hence, leveraging FPGA-based accelerators to expedite attention computations proves to be a judicious and cost-effective strategy, particularly in enhancing the efficiency of LLM inference processes.

Several spatial accelerators have been proposed to enhance the efficiency of self-attention based models by leveraging data reuse and parallelism through arrays of processing elements (PEs). These PEs are interconnected to facilitate various data reuse strategies.

Dataflow is vital in determining the design parameters of spatial accelerators, as it determines how data and computations are scheduled over space and time within the hardware resources. Different dataflow strategies are primarily categorized by specifying the tensor temporarily reused inside each PE. For instance, the Google Tensor Processing Unit (TPU) [3] employs a weight-stationary systolic array to accelerate matrix multiplication and convolution. In this case, the weights are retained inside the PE during execution. Similarly, there are output-stationary dataflow [1] and row-stationary dataflow [2].

However, these dataflows are primarily designed for individual operators, such as matrix multiplication and convolution. As the length of sequences in the attention-based model increases, the transfer of data between the PE array and the off-chip memory introduces a significant performance bottleneck. In order to address this bottleneck, fusion dataflow approaches have been proposed, which involve buffering intermediate results on-chip to mitigate the overhead of accessing off-chip memory. In this regard, FLAT [16] proposes distinct fusion dataflows for the self-attention layer, while TileFlow [18] models the fusion dataflow using a tree structure to find the optimal solutions.

While these methodologies have demonstrated considerable efficacy in mitigating the memory access overhead in attention-based models [13], current high-level synthesis (HLS) tools that aim to implement accelerators based on fusion dataflow still heavily rely on manual design of parallelization strategies, tiling strategies, memory hierarchy, and other optimization factors. Because of the vast design space constituted by these design parameters, manual design process leads to lengthy design cycles and increases the likelihood of sub-optimal designs.

Recently, frameworks have emerged offering automated generation for accelerators [9, 12, 14, 21, 22, 19]. Nevertheless, these tools often overlook the optimization of the attention mechanism itself and fail to consider scenarios with limited resources.

†Corresponding author

978-3-9826741-0-0/DATE25/© 2025 EDAA

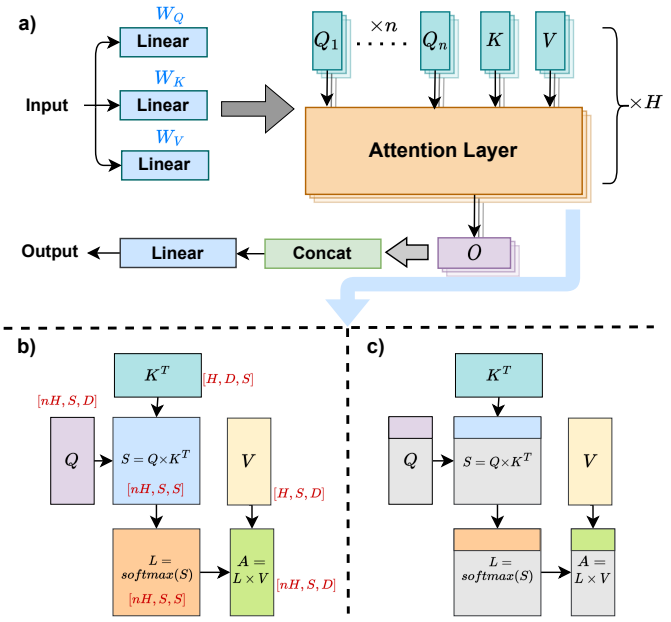

Fig. 1. (a) Self-attention mechanism in the transformer model. (b) Computation of the attention layer. The data enclosed in square brackets signifies the data shape, where S denotes the length of the sequence and $n \times H \times D$ is the hidden dimension size of the transformer model. (c) Fusion dataflow of FLAT. The colored portion indicates the data stored in on-chip memory.

To address this gap, this paper introduces *AttentionLib*, a dataflow-based framework that aims to automate the design process of accelerators for attention modules. *AttentionLib* builds upon and extends the Multi-Level Intermediate Representation (MLIR) framework [10], which enables defining IRs at multiple levels of abstraction, making the best use of different information exposed at each step of the compilation process. Through the fusion dataflow modeling and compilation techniques in the MLIR framework, *AttentionLib* facilitates automated dataflow analysis, optimization, and hardware generation.

In summary, the contributions of *AttentionLib* are as follows:

- We provide an automated flow that transforms the attention computation into hardware implementations through MLIR framework and HLS tools.
- We propose a set of IRs specifically designed for modeling the fusion dataflow, enabling the analysis of operator fusion and loop tiling strategies.
- *AttentionLib* incorporates fusion dataflow optimization passes which connect with a design space exploration (DSE) engine, allowing for tuning the optimization parameters to different hardware constraints.

To the best of our knowledge, *AttentionLib* is the first framework dedicated to optimizing and generating accelerators tailored for the attention module. In evaluation, *AttentionLib* can successfully generate suitable accelerators for a spectrum of attention computation scales, culminating in a notable minimum speedup ratio of 25.1× after DSE optimization. Furthermore, *AttentionLib* exhibits commendable flexibility and scalability in effectively managing long sequences, thereby enhancing the applicability of FPGAs in inference tasks of transformer-based models.

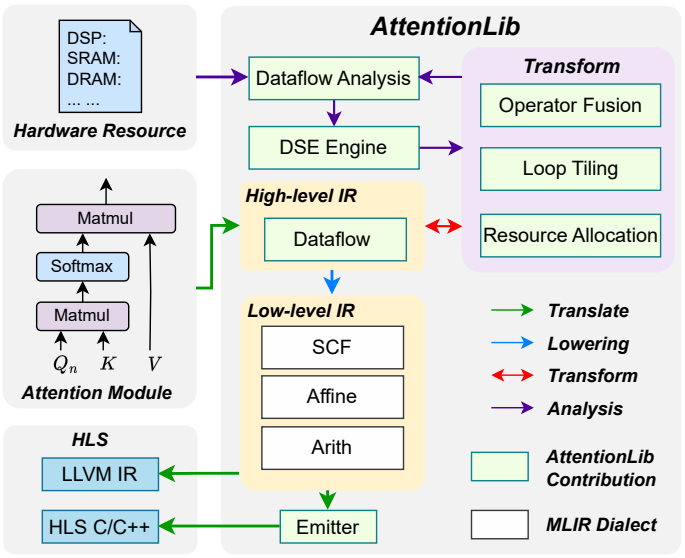

Fig. 2. Overview of *AttentionLib* framework.

II. Background

A. Attention Mechanism

Self-attention is a mechanism used to process sequential data and has a wide range of applications in the field of deep learning, especially in transformer models. As shown in Fig. 1 (a), self-attention takes each element of the input sequence as a query, key, and value, and computes the attention relationship within the sequence.

In the case of multi-head attention, represented by $n = 1$ in Fig. 1 (a), the input matrix undergoes several steps as shown in Fig. 1 (a), (b). Initially, key (K), query (Q), and value (V) vectors are generated for each token's vector through matrix multiplications with learnable weight matrices. These components are then grouped into H sets, where the dot product between query and key vectors is computed within each set, capturing inter-element relationships. Subsequently, a normalization operation like *softmax* is applied to the resulting attention scores. Finally, the value vectors are linearly combined using the weights obtained from the attention scores, resulting in the output of the attention layer. The outputs from the H attention heads are then concatenated and fed through a linear layer to obtain the final output.

When n is greater than 1 and H is equal to 1 in Fig. 1 (a), it signifies multi-query attention, where all queries share the same key and value pairs. On the other hand, when both n and H are greater than 1, it represents grouped-query attention. In this case, each attention head consists of n queries that share a common set of key and value pairs.

B. Fusion Dataflow

As the sequence length increases, the performance bottleneck of transformer models shifts from computation to memory bandwidth. To alleviate this bottleneck, various fusion dataflows have been proposed. The general approach of fusion dataflow is to fuse multiple operators into a single one and buffer the intermediate results in on-chip memory, thus reducing off-chip

```
%0 = "dataflow.fused"(%Q, %K){
  %1 = "dataflow.matmul"(%Q, %K){tile = [4, 8, 8]} : (tensor<8x128x64xf32>, tensor<8x128x64xf32>) → tensor<8x128x128xf32>
  %2 = "dataflow.softmax"(%1){ tile = [4, 8, 8]} : tensor<8x128x128xf32> → tensor<8x128x128xf32>
  dataflow.yield(%2) : tensor<8x128x128xf32>
}{allocation = pipeline, tile = [16, 16, 8]} : (tensor<8x128x64xf32>, tensor<8x128x64xf32>) → tensor<8x128x128xf32>
```
(a)

```
%0 = "dataflow.fused"(%Q, %K, %V){
  %1 = "dataflow.fused"(%Q, %K){... ...}{...} : (...) → ...
  %2 = "dataflow.matmul"(%1, %V){tile = [4, 8, 8]} : (tensor<8x128x128xf32>, tensor<8x128x64xf32>) → tensor<8x128x64xf32>
  dataflow.yield(%2) : tensor<8x128x64xf32>
}{allocation = sequential, tile = [4, 8, 8]} : () → tensor<8x128x64xf32>
```
(b)

Fig. 3. Examples of the `dataflow` dialect. (a) Fusion of two operators. (b) Nested fusion of three operators.

memory access. Since the intermediate results are sometimes too large, loop tiling is necessary. Fig. 1 (c) illustrates the row-based fusion dataflow of FLAT [16]. FLAT fuses the computations of the attention layer into a single operator and performs loop tiling at the minimum granularity of one row, which is required for the softmax operation.

TileFlow takes a step further by tiling the rows of softmax as well, and employs a tree structure to model the fusion dataflow. This approach aims to find the optimal solution within the design space, which includes compute ordering, resource binding, and loop tiling. However, TileFlow does not delve into grouped-query attention and multi-query attention.

III. Overview of AttentionLib

Fig. 2 illustrates the overview of the *AttentionLib* framework. The framework takes the user-defined attention network (number of attention heads, number of queries, etc.) and hardware resources (number of DSPs and size of memory) as input. It performs IR transformation and optimization using the MLIR framework, ultimately generating LLVM IR or synthesizable HLS C/C++ code.

Specifically, the framework first translates the attention network structure definitions into the `dataflow` dialect, which serves as the high-level IR of *AttentionLib* and models the fusion dataflow (Section IV-A). Following this, optimizations are applied to the high-level IR for fusion dataflow, including operator fusion, loop tiling, and hardware resource allocation (Section IV-B). The optimization passes are connected to a DSE engine to get the optimal performance and meet the hardware resource constraints (Section IV-D).

Once the optimization process is complete, the `dataflow` dialect is lowered to the dialects of MLIR, such as the structured control dialect `scf` and the polyhedral dialect `affine` (Section IV-C). Finally, the *AttentionLib* framework translates the low-level IR into LLVM IR or HLS C/C++ code, enabling interaction with high-level synthesis tools.

IV. AttentionLib Framework

A. Dataflow Dialect

The `dataflow` dialect is the high-level IR of the *AttentionLib* framework, specifically designed to represent various parameters involved in the design of fusion dataflow, thus enabling the analysis and optimization of fusion dataflow. The operations defined in the `dataflow` dialect are described in Table I. Within this dialect, `dataflow.fused` and `dataflow.yield` operations are used to represent the fused operators, while `dataflow.matmul` and `dataflow.softmax` operations are used to represent specific computations. In order to provide additional information for analysis and optimization, we have introduced attributes `tile` and `allocation` to some operations. To provide a visual representation, Fig. 3 showcases two examples of the `dataflow` dialect, illustrating its usage within the framework.

TABLE I
THE DATAFLOW DIALECT OPERATIONS

Operation	Description
`dataflow.func`	Defines a fusion dataflow function.
`dataflow.fused`	Represents the operator resulting from the fusion of two operators.
`dataflow.yield`	The return value of the fusion operator.
`dataflow.matmul`	Represents tiled matrix multiplication.
`dataflow.softmax`	Represents tiled softmax computation.

As depicted in Fig. 3, the `dataflow.fused` operation typically contains two operations of the `dataflow` dialect, representing the fusion of two operators. Nested fusion, supporting the fusion of more than two operators, is also possible. The `tile` attribute of the computation operation indicates the tile size of loop, while the `tile` attribute of the `dataflow.fused` operation denotes the tile size of input data, influencing computation parallelism and scale, respectively. In addition, two resource allocation strategies are supported: `pipeline` and `sequential`. With the `pipeline` strategy, operators pass results immediately to the next, enhancing parallelism. On the other hand, the `sequential` strategy involves the first operator computing one tile at a time, storing it in on-chip memory for subsequent utilization by the second operator in the computation process.

B. Fusion Dataflow Analysis

In order to evaluate the performance of the accelerator and verify that the hardware resource constraints are met, the fusion dataflow needs to be analyzed.

Single Operation. The compute resource usage for a single operation is determined by the type of operator and the tile size. In the case of matrix multiplication, for example, if a compute unit can compute one multiply-add at a time, the total number of compute units used is the product of the sizes of the individual loop tile. Similarly, the amount of on-chip memory

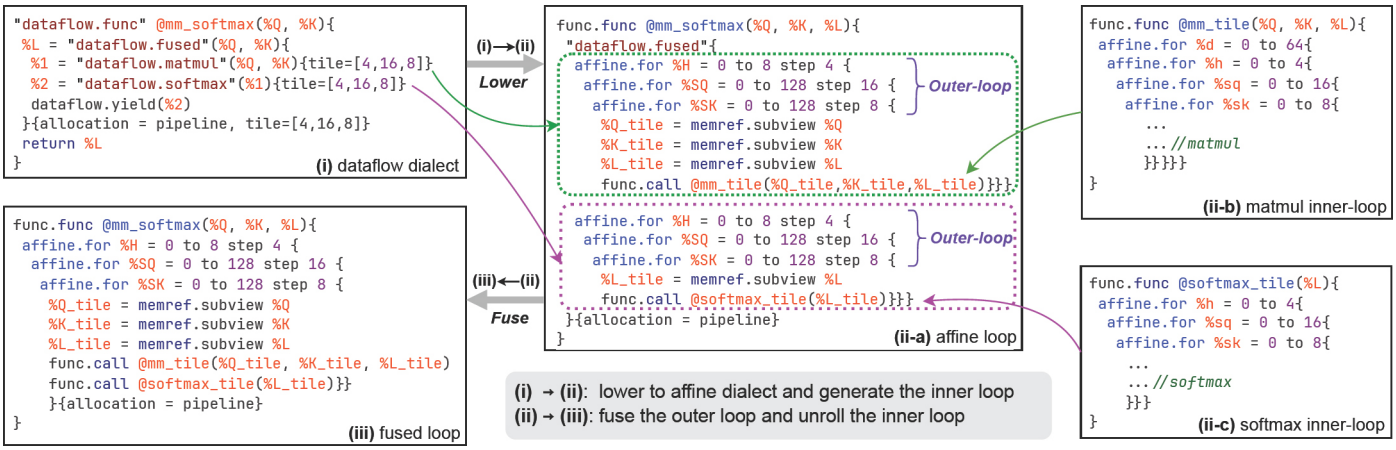

Fig. 4. Lowering process of the `dataflow` dialect. Some operation attributes or types are omitted for simplicity.

TABLE II
RESOURCE USAGE OF THE FUSED OPERATOR

Allocation Strategy	Compute Resource Usage	Memory Usage
`pipeline`	$Op_1 + Op_2$	$Op_1 \cup Op_2$
`sequential`	$\max\{Op_1, Op_2\}$	$\max\{Op_1, Op_2\}$

required is determined by the data width, operator type, and tile size [16, 18].

Fused Operation. The resource usage of the fused operation is determined by both the resource requirements of the individual operators being fused and the resource allocation strategy employed. The relationships between these factors are summarized in Table II. When two operators are fused together using the `pipeline` strategy, the compute resource usage of the fused operator is the sum of the compute resources required by the two individual operators. Simultaneously, the on-chip memory usage is a union of the memory blocks used by the two operators. Conversely, when the `sequential` strategy is employed, both the compute resource usage and memory requirement are determined by the maximum values of the two individual operators.

Performance Estimation. In the execution of each tile within the fused operator, there are three phases: loading the input of the current tile from off-chip memory to on-chip memory, performing the computation, and storing the output to off-chip. The performance bottleneck is determined by the maximum duration among these three phases, where the latency of data loading and storage can be obtained from the volume of data movement and the bandwidth of memory [18], and the latency of computation can be obtained through polyhedron-based analysis [7, 11]. Once the latency of a single tile is obtained, the overall latency of the operator can be calculated, allowing for an estimation of the accelerator's performance.

C. Lowering to Low-level IR

Once the optimal fusion dataflow is obtained, the `dataflow` dialect undergoes a lowering process to dialects of MLIR, as shown in Fig. 4. Each operation in `dataflow.fused` is divided into two components—an outer loop and an inner loop—due

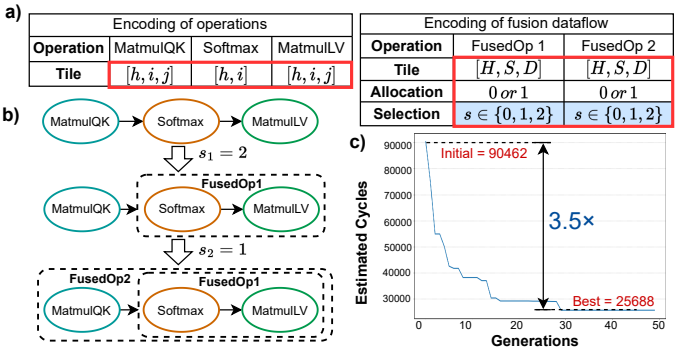

Fig. 5. (a) Genetic algorithm gene encoding. (b) Building fusion dataflow from the gene encoding. (c) Effectiveness of DSE process on the multi-head attention module with head = 8, hidden_size = 512 and seq_len = 128.

to loop tiling, with each component generated independently. The outer loop remains consistent across both operations, serving as a shared structure, while the inner loop embodies the tiled computing kernel, crucial in determining the utilization of compute resources. Subsequently, the outer loops of both operations are fused while retaining the `allocation` attribute, which proves instrumental in streamlining the generation of the corresponding HLS code. Meanwhile, the inner loop undergoes an unrolling process to enhance parallelism.

D. Design Space Exploration

We can estimate the performance of the fusion dataflow through the tile sizes for each operator and the `allocation` attribute of the fusion operator. However, achieving optimal performance is challenging due to the myriad parameters influencing performance, resulting in a vast design space. Conducting an exhaustive search becomes impractical, with the number of iterations surpassing 10^{15}, rendering it inefficient.

To streamline the quest for a high-performance fusion dataflow, we have devised a Design Space Exploration engine using genetic algorithms (GA). The encoding of fusion dataflow is depicted in Fig. 5 (a). Tile sizes are directly encoded, while the `allocation` attribute employs 0 for `pipeline` and 1 for `sequential`. For operator fusion, a selection variable s is introduced, which indicates which two operators are selected for fusion. Considering an attention module comprising three

TABLE III
PERFORMANCE OF ACCELERATORS GENERATED BY ATTENTIONLIB

Attention Module	Optmization	Cycles	Speedup	DSP (Util. %)	BRAM (Util. %)	URAM (Util. %)
MHA	Baseline	6.312×10^5	1.0×	396 (5.8%)	847 (39.2%)	0 (0%)
	DSE	2.513×10^4	25.1×	6720 (98.2%)	2140 (99.1%)	512 (53.3%)
GQA	Baseline	6.292×10^5	1.0×	396 (5.8%)	815 (37.7%)	0 (0%)
	DSE	2.308×10^4	27.2×	6720 (98.2%)	2140 (99.1%)	896 (93.3%)
MQA	Baseline	6.274×10^5	1.0×	396 (5.8%)	799 (37.0%)	0 (0%)
	DSE	2.112×10^4	29.7×	6720 (98.2%)	2124 (98.3%)	576 (60.0%)

operators sequentially, each fusion presents up to three choices: no fusion, fusion of the 1st and 2nd operators, and fusion of the 2nd and 3rd operators, denoted as 0, 1, and 2, respectively. Upon making two fusion selections, a comprehensive fusion dataflow is constructed. Fig. 5 (b) illustrates the dataflow configuration with $s_1 = 2$ and $s_2 = 1$.

In the DSE algorithm, an initial set of fusion dataflows adhering to hardware constraints is randomly generated, encoded, and fed into the GA as the starting population. The fitness function of GA is as follows:

$$Fit(x) = -cL(x) - \max\{0, H(x) - R\},$$

where $L(x)$ and $H(x)$ denote the latency and resource usage of the fusion dataflow, respectively; c is a constant used to correct the order of magnitude, and R is the resource constraint. Through crossovers, mutations, and natural selection, the population evolves to yield a fusion dataflow with superior performance. Following experimentation, the GA typically converges to an optimized solution within fewer than 50 generations. The entire DSE process concludes within minutes, showcasing accelerated performance optimization, as illustrated in Fig. 5 (c).

V. EXPERIMENTAL EVALUATION

AttentionLib automatically can generate HLS C++ code to create accelerators based on the attention computation structure and available hardware resources. To evaluate the efficacy of *AttentionLib* framework, we undertake a series of comprehensive experiments, scalability analyses and comparative studies in this section. Our experiments demonstrate the performance improvement of operator fusion for attention computation and also validate the effectiveness of our DSE algorithm.

The generated HLS C++ code is synthesized using AMD Vitis 2023.2, and all results are obtained from the Vitis synthesis report. For our experimental platform, we employ an AMD Alveo U200 featuring 6840 DSPs, 2160 Block RAMs (BRAMs), and 960 Ultra RAMs (URAMs), producing accelerators operating at a frequency of 100MHz.

A. Performance

We evaluate *AttentionLib*'s efficacy with multi-head attention (MHA), grouped-query attention (GQA), and multi-query attention (MQA) configurations, maintaining a 512 hidden size, 8 queries, and 4 groups for GQA. Input sequence length remained constant at 128, processed in 32-bit precision. During the evaluation phase, we implement operator fusion and activate the DSE engine to optimize performance. For the baselines, we only enable automatic loop pipeling in Vitis to improve parallelism, and do not incorporate operator fusion or dataflow optimization techniques. The comparative results are presented in Table III. In the tests, the accelerators generated by *AttentionLib* obtain a speedup of $25.1\times$ to $29.7\times$ compared to the baselines.

It is noteworthy that the speedup achieved with the DSE engine escalates progressively from MHA to GQA to MQA. This phenomenon can be attributed to the diminishing sizes of keys and values, expanding available BRAMs and URAMs. The DSE engine effectively leverages these augmented hardware resources to capitalize on heightened parallelism, thereby enhancing the speedup achieved in the optimization process.

Upon thorough analysis, we attribute the performance enhancement to several key factors: (1) centralized access to off-chip memory (DDR) and storage on-chip facilitates reaching the maximum bandwidth of off-chip memory; (2) efficient on-chip memory usage mitigates the memory dependence bottleneck thereby enhancing the throughput of the compute process; (3) the application of loop tiling and unrolling strategies improves parallel execution capabilities; (4) operator fusion minimizes the frequency of off-chip memory accesses, leading to reduced latency and increased efficiency; (5) The DSE engine makes it possible to maximize the utilization of resources.

B. Scalability Study

In the current AI landscape, transformer-based models are experiencing a surge in parameter counts, notably evidenced by the expanding hidden dimensions. There is also a trend towards longer input sequences to enhance contextual understanding. To assess the effectiveness of the *AttentionLib* framework across models of diverse sizes and input dimensions, we conduct a series of tests across different scales using MHA as a prime example. The cycle count and throughput data are visually represented in Fig. 6. Leveraging advanced loop tiling techniques, our framework demonstrates the capability to efficiently generate accelerators tailored for attention computations of varying sizes, showcasing adaptability across a broad spectrum of model requirements.

In Fig. 6 (a) and (b), the accelerator's throughput rises with increasing sequence length and hidden dimension size. This upward trend is directly attributable to the accelerator's ability to process a larger volume of inputs concurrently, effectively harnessing the inherent data parallelism within the model. Nonetheless, as the input sizes continue to escalate, there is a notable strain on the FPGA's on-chip memory resources,

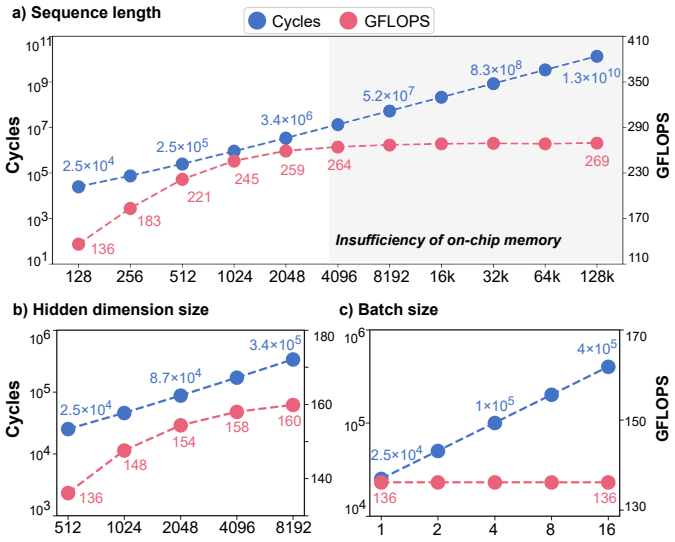

Fig. 6. Performance of accelerators generated by *AttentionLib* with varying hidden size and input size. Basic hidden_size = 512 and seq_len = 128.

thereby imposing limitations on the extent to which substantial throughput gains can be achieved.

Moreover, Fig. 6 (c) illustrates that augmenting batch size does not enhance throughput. This limitation arises from compute resource constraints, impeding simultaneous operations across batches and impeding additional parallelization efforts. This also suggests that FPGAs are more suitable for neural network inference scenarios at the edge.

C. Comparison with CPU and GPU

To facilitate a comprehensive comparison, we scrutinize the runtime performance of sequences varying in length for both *prefill* and *decode* workloads against CPU and GPU platforms, as detailed in Table IV. Note that *prefill* workloads entail complete MHA calculations, whereas *decode* workloads involve MHA computations with query sequences limited to a single element. We use the just-in-time compilation technology of PyTorch to accelerate the inference speed on CPU and GPU.

Remarkably, the accelerator generated by *AttentionLib* consistently outperforms the CPU in runtime efficiency. However, when subjected to *prefill* workloads involving long sequences, the accelerator demonstrates inferior performance compared to the GPU. This performance gap can be attributed to GPUs boasting superior floating-point resources and higher frequency, while the limited number of DSPs in FPGAs imposes constraints on computational scalability. On the other hand, when attention computations for lengthy sequences are performed directly on CPUs and GPUs, out-of-memory errors occur due to the necessity of storing massive intermediate tensors. In contrast, *AttentionLib* avoids this problem through the implementation of tiling techniques and on-chip storage of tiled intermediate tensors.

Contrastingly, in the context of the *decode* workload, the generated accelerator significantly outpaces GPU in processing short sequences and is only surpassed by the GPU in processing very long sequences. Further analysis reveals that in *decode* workloads with long sequences, over 80% of the accelerator's runtime is dedicated to reading K and V data. Operating at 100MHz, the bandwidth of the generated accelerator while reading data from DDR stands at 23.8GB/s. An effective strategy to further enhancing speed involves augmenting the bandwidth. In this vein, we evaluate the efficacy of the *AttentionLib* framework on *decode* workloads using the Alveo U280, an FPGA with less on-chip memory than the Alveo U200 but equipped with high-bandwidth memory (HBM). At the same frequency, the bandwidth between the accelerator and the HBM reaches 190.7 GB/s. Results show that increasing the bandwidth with an HBM can improve operational speeds, surpassing those of the GPU in most cases. This implies that FPGAs are a viable alternative to GPUs for executing such memory-intensive computations.

TABLE IV
RUNTIME PERFORMANCE COMPARED TO CPU AND GPU

Part a) Prefill. Input size: [8, seq_len, 64].						
Platform	**Runtime(ms)**					
	128	**256**	**1k**	**4k**	**16k**	**64k**
CPU	0.43	1.47	14.84	205.27	3097.88	OOM
GPU	0.38	0.39	0.45	1.96	36.38	OOM
Alveo U200	**0.25**	**0.75**	**8.91**	**132.57**	**2089.21**	**33299.51**
Part b) Decode. Q: [8, 1, 64], K/V: [8, seq_len, 64].						
Platform	**Runtime(ms)**					
	128	**256**	**1k**	**4k**	**16k**	**64k**
CPU	0.19	0.20	0.36	1.15	9.14	41.57
GPU	0.32	0.33	0.33	0.37	0.74	1.61
Alveo U200	**0.02**	**0.04**	**0.18**	**0.70**	**2.79**	**11.16**
Alveo U280	**0.01**	**0.01**	**0.05**	**0.18**	**0.70**	**2.81**

CPU: Intel Xeon Gold 5218R; GPU: NVIDIA Tesla A100.
OOM: Out of memory.

VI. CONCLUSION

In this paper, we propose *AttentionLib*, an MLIR-based optimization framework designed for accelerating attention mechanisms. The *AttentionLib* framework incorporates fusion dataflow analysis for attention computation and facilitates the generation of FPGA accelerators. Furthermore, we have developed a DSE engine to expedite the search for optimized fusion dataflow configurations. Experimental results underscore the exceptional scalability of *AttentionLib*, showcasing its proficiency in generating accelerators tailored for attention computations across various dimensions, all while delivering commendable acceleration outcomes. Furthermore, the accelerators generated by *AttentionLib* demonstrate superior performance to GPUs in most cases involving *decode* workloads. When processing extremely long sequences, the generated designs also have comparable performance to GPUs.

ACKNOWLEDGMENT

This work was financially supported in part by Project of Shanghai EC (24KXZNA12). The computaions in this research were performed using the CFFF platform of Fudan University.

References

[1] Zidong Du, Robert Fasthuber, Tianshi Chen, Paolo Ienne, Ling Li, Tao Luo, et al. ShiDianNao: Shifting Vision Processing Closer to the Sensor. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*. 2015.

[2] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks. *ACM SIGARCH Computer Architecture News* 44.3 (2016).

[3] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, et al. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*. 2017.

[4] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, et al. Attention Is All You Need. In *Advances in Neural Information Processing Systems*. Ed. by I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, et al. Vol. 30. 2017.

[5] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-Training of Deep Bidirectional Transformers for Language Understanding. 2019. arXiv: 1810.04805.

[6] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, et al. Language Models Are Few-Shot Learners. 2020. arXiv: 2005.14165.

[7] Hyoukjun Kwon, Prasanth Chatarasi, Vivek Sarkar, Tushar Krishna, Michael Pellauer, and Angshuman Parashar. MAESTRO: A Data-Centric Approach to Understand Reuse, Performance, and Hardware Cost of DNN Mappings. *IEEE Micro* 40.3 (2020).

[8] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, et al. An Image Is Worth 16x16 Words: Transformers for Image Recognition at Scale. 2021. arXiv: 2010.11929.

[9] Liancheng Jia, Zizhang Luo, Liqiang Lu, and Yun Liang. TensorLib: A Spatial Accelerator Generation Framework for Tensor Algebra. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*. 2021.

[10] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, et al. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2021.

[11] Liqiang Lu, Naiqing Guan, Yuyue Wang, Liancheng Jia, Zizhang Luo, Jieming Yin, et al. TENET: A Framework for Modeling Tensor Dataflow Based on Relation-Centric Notation. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. 2021.

[12] Nicolas Bohm Agostini, Serena Curzel, Vinay Amatya, Cheng Tan, Marco Minutoli, Vito Giovanni Castellana, et al. An MLIR-Based Compiler Flow for System-Level Design and Hardware Acceleration. In *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design*. 2022.

[13] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness. 2022. arXiv: 2205.14135.

[14] Hanchen Ye, Cong Hao, Jianyi Cheng, Hyunmin Jeong, Jack Huang, Stephen Neuendorffer, et al. ScaleHLS: A New Scalable High-Level Synthesis Framework on Multi-Level Intermediate Representation. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 2022.

[15] Gianni Brauwers and Flavius Frasincar. A General Survey on Attention Mechanisms in Deep Learning. *IEEE Transactions on Knowledge and Data Engineering* 35.4 (2023). arXiv: 2203.14263.

[16] Sheng-Chun Kao, Suvinay Subramanian, Gaurav Agrawal, Amir Yazdanbakhsh, and Tushar Krishna. FLAT: An Optimized Dataflow for Mitigating Attention Bottlenecks. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 2023.

[17] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, et al. LLaMA: Open and Efficient Foundation Language Models. 2023. arXiv: 2302.13971.

[18] Size Zheng, Siyuan Chen, Siyuan Gao, Liancheng Jia, Guangyu Sun, Runsheng Wang, et al. TileFlow: A Framework for Modeling Fusion Dataflow via Tree-Based Analysis. In *56th Annual IEEE/ACM International Symposium on Microarchitecture*. 2023.

[19] Yang Liu, Tianchen Wang, Yuxuan Dong, Zexu Zhang, Shun Li, Jun Yu, et al. TransLib: an extensible graph-aware library framework for automated generation of transformer operators on FPGA. In *Proceedings of the 43rd IEEE/ACM International Conference on Computer-Aided Design*. 2024.

[20] Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, et al. Splitwise: Efficient Generative LLM Inference Using Phase Splitting. 2024. arXiv: 2311.18677.

[21] Hanchen Ye, Hyegang Jun, and Deming Chen. HIDA: A Hierarchical Dataflow Compiler for High-Level Synthesis. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*. 2024.

[22] Weichuang Zhang, Jieru Zhao, Guan Shen, Quan Chen, Chen Chen, and Minyi Guo. An Optimizing Framework on MLIR for Efficient FPGA-Based Accelerator Generation. 2024. arXiv: 2401.05154.

[23] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, et al. DistServe: Disaggregating Prefill and Decoding for Goodput-Optimized Large Language Model Serving. 2024. arXiv: 2401.09670.