SWAT: Scalable and Efficient Window Attention-based Transformers Acceleration on FPGAs

Zhenyu Bai, Pranav Dangi, Huize Li✉, Tulika Mitra✉
School of Computing, National University of Singapore, 119077, Singapore
{zhenyu.bai, dangi, huizeli}@nus.edu.sg, tulika@comp.nus.edu.sg

ABSTRACT

Efficiently supporting long context length is crucial for Transformer models. The quadratic complexity of the self-attention computation plagues traditional Transformers. Sliding window-based static sparse attention mitigates the problem by limiting the attention scope of the input tokens, reducing the theoretical complexity from quadratic to linear. Although the sparsity induced by window attention is highly structured, it does not align perfectly with the microarchitecture of the conventional accelerators, leading to suboptimal implementation. In response, we propose a dataflow-aware FPGA-based accelerator design, SWAT, that efficiently leverages the sparsity to achieve scalable performance for long input. The proposed microarchitecture is based on a design that maximizes data reuse by using a combination of row-wise dataflow, kernel fusion optimization, and an input-stationary design considering the distributed memory and computation resources of FPGA. Consequently, it achieves up to 22× and 5.7× improvement in latency and energy efficiency compared to the baseline FPGA-based accelerator and 15× energy efficiency compared to GPU-based solution.

1 INTRODUCTION

Transformer-based models [17], known for their self-attention mechanisms, are leading advancements in artificial intelligence. A typical transformer model contains linear layers, multi-head attention layers, and Feed-Forward Networks (FFN). The self-attention process involves first transforming each input token into the Query (Q), Key (K), and Value (V) vectors. Then it computes the dot products of Q and K for each token $S = Q \times K^T$, determining the similarity scores S between them. These scores are then normalized using a softmax function to form probabilities $S' = SoftMax(S)$. Finally, the V vectors are multiplied by these probabilities and summed up to produce the final output $Z = S' \times V$. This mechanism allows each token to contextually relate to **every** other token in the input, which is crucial for tasks requiring complex contextual understanding.

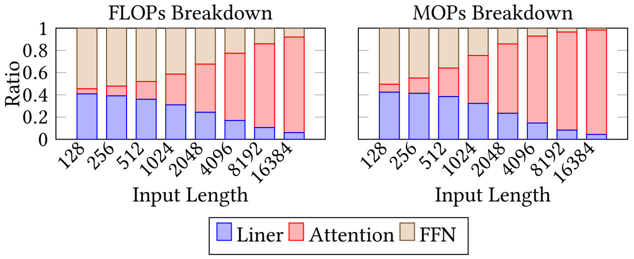
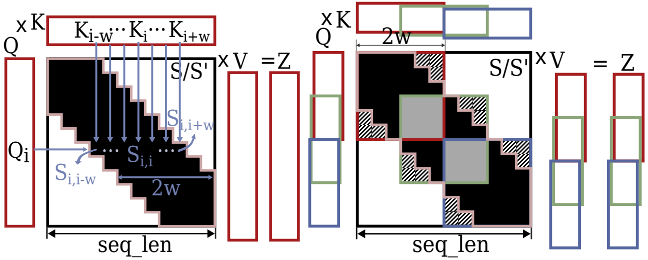


Figure 1: Floating point operations (FLOPs) and memory operations (MOPS) breakdown for different input lengths



(a) Window Attention **(b) sliding chunks implementation**

Figure 2: Sliding window attention and its SOTA sliding chunks implementation

However, a notable impediment of this model is its quadratic complexity, which necessitates each sequence element to be compared against every other element. This complexity becomes particularly intractable in tasks with long contexts, such as document-level translation or long-form questions, due to the computational demands [8]. This issue is illustrated in Figure 1 where the floating point operations (FLOPs) and memory operations (MOPs) for attention computation grow with increasing input length and become a critical performance bottleneck.

To address this, the sliding window attention has been introduced [1]. This method limits the attention of each token to a fixed subset of **adjacent** tokens, thereby reducing the theoretical complexity from quadratic to linear. From the computational perspective, this method introduces a structured pattern of sparsity in the attention computation, as shown in Figure 2a where each token attends to w tokens before and after, forming a diagonal sparsity pattern of width $2w$. This sparsity manifests as a mask applied to the S and S' matrices, resulting in a Sampled Dense-Dense Matrix Multiplication between Q and K, and a Sparse-Dense Matrix Multiplication between S' and V. Despite the structured nature of this sparsity pattern, unlike dense operations, it is not perfectly aligned to be seamlessly filled into the vector/matrix math lanes of conventional accelerators. Instead, it requires fine-grain control of the microarchitecture for optimal performance that is not provided by existing accelerators. For instance, the Tensor Cores in Nvidia GPUs for accelerating dense matrix multiplications can only be accessed through higher-level programming models such as CUDA C++ API or C libraries, limiting micro-architecture level

Acknowledgement: this research is partially supported by the National Research Foundation, Singapore under its Competitive Research Program Award NRF-CRP23-2019-0003 and AMD Research gift.
The corresponding authors are Huize Li and Tulika Mitra.

DAC '24, June 23–27, 2024, San Francisco, CA, USA
© 2024 Association for Computing Machinery.
ACM ISBN xxxx...$15.00
https://doi.org/10.1145/000000.0000000

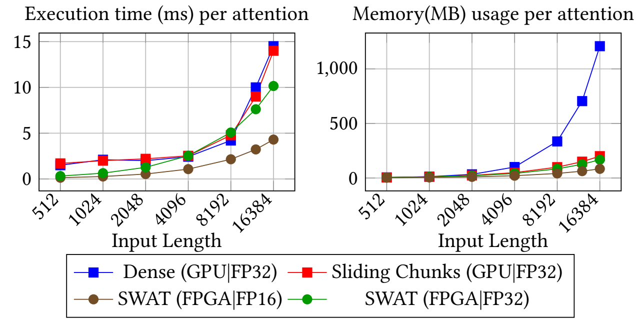


Figure 3: Execution time and memory usage of existing approaches Dense and Sliding Chunks compared to SWAT

control. The current state-of-the-art implementation[1], known as *sliding chunks*, addresses this limitation by dividing the sparse operation into smaller dense operations across chunks of width $2w$, as illustrated in Figure 2b.

Yet, this approach leads to redundant computations in the form of the overlapping regions (gray regions in figure 2b) and corner areas (dashed regions in the figure) of each chunk. The ratio of these redundant computation is given by $\frac{1}{2} - \frac{1}{4|chunks|}$ where $|chunks|$ is the number of chunks. This ratio increases and approaches rapidly to $\frac{1}{2}$, i.e., 50% redundancy. Moreover, eliminating these redundant calculations is challenging as the correctness of the results must be ensured, which further increases the computational overhead. In Figure 3, We compare the execution time and memory usage[2] of the sliding chunks with the naïve dense operations approach on an AMD MI210 GPU. The result indicates that while the sliding chunks approach significantly reduces memory usage, the computational time remains similar to the dense method, primarily due to the redundant computation but also due to the overhead for increased frequency of small kernel launches on GPU.

Motivation & Contribution: We aim to refine the implementation of sliding window attention by focusing on the efficient computation of its structured, yet imperfectly aligned sparsity, which requires precise control over computation and memory operations. We propose a novel hardware design using Field-Programmable Gate Arrays (FPGAs). FPGAs are favored over ASICs because their programmability allows for the support of various attention mechanisms such as global attention and random attention, enhancing the model accuracy across different tasks. Additionally, FPGAs are readily available on cloud platforms, e.g., Microsoft Azure[2] and Amazon AWS, offering a cost-effective deployment solution.

The implementation of the sliding window attention has to consider the computation workload and how it is mapped onto the distributed memory and computation resources of the FPGA fabric for optimal performance. Therefore, we deeply analyze the data flow of the sliding window attention, which reveals that a combination of window attention, row-major dataflow, and kernel fusion can significantly enhance off-chip transfer efficiency, ensuring each data item is loaded just once. We propose employing a fixed-size First-In-First-Out (FIFO) buffer to manage the sliding window input, leading to an input-stationary data flow. Here, the input data remains in the buffers after being loaded while necessary computational resources are placed around them by the micro-architecture design. This approach better utilizes the on-chip memory blocks' bandwidth and minimizes on-chip data movement, aligning well with the distributed memory and computation units of the underlying FPGA fabric and therefore achieving higher performance.

As demonstrated in Figure 3, SWAT exhibits linear scaling of memory use with input length. SWAT achieves 6× energy efficiency to conventional GPU-based solutions for comparable execution time for input length below 8K tokens and shows superior performance for longer input. When compared to a baseline FPGA accelerator, SWAT achieves 22× and 5.7× improvements in latency and energy efficiency, respectively (with 16384 tokens). Moreover, SWAT shows better scalability compared to both GPU and FPGA solutions.

2 BACKGROUND & RELATED WORKS

2.1 Efficient Transformers

Efficient Transformers [16] are the variants of the vanilla Transformer model, aiming at improving the computational efficiency. Two main strategies underlie these improvements. The first seeks to approximate the traditional SoftMax attention mechanisms with algorithms of lower computational complexity. For instance, FNet [9] and Linformer [18] substitute the standard attention calculations with Fourier Transforms and linear projections, respectively. The second pathway, sparse attention, aims to reduce attention operations while preserving the SoftMax-based attention formulation.

2.2 Sparse attention

Sparse attention can be further classified into two categories. Dynamic mathods [10, 12, 13] evaluate or predict attention scores in real-time, prioritizing the most significant ones for subsequent computation. These methods, while adaptable, introduce irregular sparsity patterns that hinder efficient computation. In contrast, static methods [1, 3, 20] pre-define attention patterns, achieving structured sparsity at the cost of some accuracy. The structured sparsity can be leveraged for predictable performance gains with static optimization and dedicated hardware support.

Sliding window attention [1], is the key component of nearly all static sparse attention approaches. This technique limits each token's attention to a predetermined number of adjacent tokens, based on the research findings that show the substantial impact of the local context within the attention mechanism [11, 19].

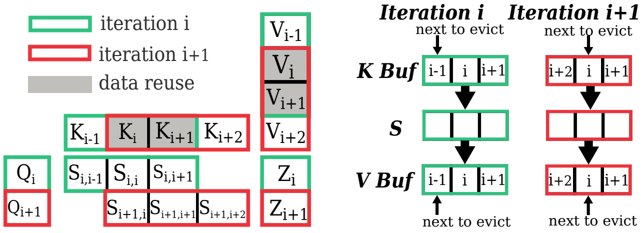
2.3 Accelerators for static structured attention

ASIC-based accelerator SALO [14], designed explicitly for the Longformer model, utilizes structured sparsity in window attention with a 2D square systolic array. However, it is limited to the basic Longformer setup. Furthermore, the systolic array's square structure requires square tiling of the input and the intermediate matrices. This tiling is suboptimal for row-wise SoftMax operations, necessitating supplementary computations outside the accelerator's capabilities.

Butterfly [7] is an FPGA-based accelerator for efficient transformer models that utilizes a static butterfly sparsity pattern. Notably, this pattern can be approximated through Discrete Fourier Transformations. However, this FFT-based approximation of the original SoftMax attention has not consistently demonstrated reliable accuracy across various downstream tasks. Butterfly attempts

[1] Hugging Face's Longformer implementation, as per the original Longformer paper[1].
[2] The experimental setup will be presented in Section 5.

to mitigate this by combining vanilla SoftMax attention layers with FFT-based attention layers. Our analysis in Section 5.2 reveals that incorporating at least one conventional attention mechanism is crucial for acceptable accuracy. Unfortunately, this traditional attention's quadratic complexity leads to suboptimal performance for the accelerator when managing long input sequences.



(a) Data-reuse (b) Fixed-length FIFO

Figure 4: Data management for window attention

3 DATAFLOW ANALYSIS & OPTIMIZATION

3.1 Enabling kernel fusion

The standard implementation of Transformer models involves a sequential three-step computation- QK multiplication, SoftMax, and SV multiplication- primarily due to the row-wise data dependency of the SoftMax operation. As on-chip memory is typically insufficient for handling the entire computation in one go, each step is broken down into smaller tile-wise operations, leading to redundant off-chip data transfers for loading and storing the tile-wise intermediate results (the tiles of S and S').

Kernel fusion [5], as an optimization technique, aims to consolidate these steps into a single operation for each input tile, thereby reducing off-chip data transfers. However, the inherent row-wise dependency in the SoftMax operation presents a significant challenge to this fusion. By reinterpreting the SoftMax operation, we can divide it into two components: the numerator that does not depend on the other elements of the row and the denominator that depends on the sum of the exponential of all elements of the same row. By viewing the denominator as a scaling factor, it can be placed after the third step $Z = S' \times V$ as shown in Equation 1. This restructuring allows for the fusion of the three operations into a unified **row-wise** kernel.

$$\begin{aligned} Z_{i,j} &= \sum_{n=0}^{H} S'_{i,n}V_{n,j} = \sum_{n=0}^{H} \frac{exp(S_{i,n})}{\sum_{l=0}^{H} exp(S_{i,l})} V_{n,j} \\ &= (\frac{1}{\sum_{l=0}^{H} exp(S_{i,l})})(\sum_{n=0}^{H} exp(S_{i,n})V_{n,j}) \end{aligned} \quad (1)$$

3.2 Row-major dataflow & Data reuse

In the standard three-step computation of transformers, independent execution of each step limits the benefits gained from tiling strategies or dataflow optimization. However, in the context of sliding window attention, these three computations share a common sparsity pattern, as depicted in Figure 2a. When considering the attention computation for a given input row of Q, say Q_i, and the subsequent row vector Q_{i+1}, we observe significant data reuse of the attended rows of K (columns of K^T) and V, as shown in Figure 4a for the window width $w = 1$. The most effective way to harness this data reuse is by adopting a row-major dataflow. Although kernel fusion *postpones* the row-wise dependencies of SoftMax, it does not *eliminate* them. A row-major approach, therefore, becomes advantageous, minimizing the memory needed for storing intermediate results S and S', which now can be stored in on-chip memories.

To capitalize on this data reuse opportunity, SWAT employs fixed-size on-chip FIFO buffers for the K and V inputs, while the Q input changes for each row. This setup, illustrated in Figure 4b, features a buffer with a moving pointer that indicates the next element to be replaced, ensuring data is loaded exactly once and achieving 100% off-chip memory transfer efficiency.

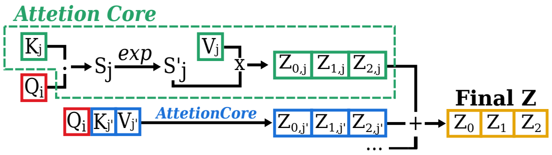


Figure 5: input-stationary dataflow

3.3 Input-Stationary dataflow for FPGA

The previous sections discussed SWAT's dataflow at the algorithmic level. Now, it's important to consider how this dataflow is mapped onto the FPGA's microarchitectural design. The following key aspects are considered. First, FPGAs have distributed memory (BRAMs and URAMs) and computing elements (LUTs, DSP slices) across the chip. Secondly, by utilizing fixed-size FIFO buffers for input data, as exhibited in Figure 4b, input data mostly remains stationary. Finally, kernel fusion ensures a consistent pairing of (K_j, V_j) for each Key/Value row $j \in [i - w, i + w]$. This coherency is apparent both in the fusion equation (see Equation 1) and the FIFO eviction process (Figure 4b). From the algorithm level, this is because the same input sequence indexes Key and $Value$ matrices according to the self-attention mechanism.

Consequently, we adopt an *input-stationary* dataflow. In this design, input data remain in their respective buffers, and computational units are positioned nearby. This is different from conventional accelerator designs, where the data is brought to the computational units. Figure 5 illustrates this dataflow for one row of input Q within the input stationary paradigm. An *Attention Core*—our terminology for the minimal computational unit—consists of a buffer holding one row of K (K_j), and one row of V (V_j). Upon the arrival of a new row of Q, denoted as Q_i, the multiplication with K_j is performed locally within each Attention Core: $S_{i,j} = Q_i \cdot K_j$. Subsequently, the numerator of the SoftMax computation is performed: $S'_{i,j} = exp(S_{i,j})$ according to the kernel fusion. For the multiplication of S' with V, we adhere to the input-stationary paradigm, where each S' element multiplies with the corresponding V row stored in the **same** attention core. This operation yields one slice of Z per attention core. The slices produced by all attention cores are summed up outside of the attention cores to form the final result Z.

3.4 Dataflow compatibility for ASIC

The dataflow optimization techniques we have developed, particularly row-major dataflow and kernel fusion, are also applicable to ASIC-based implementations, which, unlike FPGAs, are not constrained by the distribution of computation and memory resources. While ASICs can potentially offer superior performance, they lack the flexibility of FPGAs, which is crucial for adapting to the evolving landscape of Transformer models.

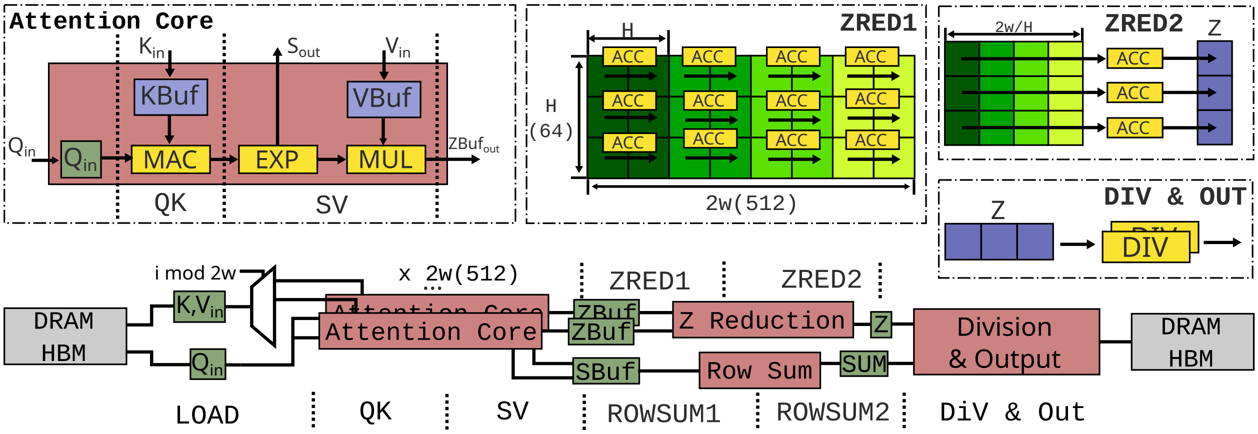


Figure 6: SWAT Microarchitecture design

4 ARCHITECTURE DESIGN

Figure 6 shows the architecture design of SWAT by following the dataflow design outlined in the previous section. The architecture makes use of a pipeline execution to improve resource usage efficiency. The functions of the pipeline stages are as follows:

LOAD Stage: Data from the main memory is fetched and loaded into the K/V buffers of the attention cores. For the standard window width configuration ($2w = 512$), 512 attention cores are instantiated. Each K/V buffer uses one BRAM block, storing a full row of K or V of size H(head dimensionality). According to the K/V buffer replacement policy, the entire K/V buffer of **one** attention core is refreshed per attention of one row. The selection signal is computed by the row index i modulo the window size according to the FIFO policy. The Q row is loaded during this stage and distributed across all attention cores.

QK Stage: This stage calculates the dot product between the K row and the Q row in the attention cores. Due to FPGA constraints, the FP16 multiply-accumulate (MAC) operation is pipelined at an Initial Interval (II) of 3 cycles. Forcing the MAC to be pipelined at fewer cycles will significantly increase resource usage.

SV Stage: Following the QK stage, the SV stage computes the exponential of the S values and multiplies these with the corresponding V elements within the **same** attention core, generating a slice of Z per attention core, stored in $ZBuf$. The FP16 multiplications are executed over an II=3 pipeline. While a lower II is feasible, it does not improve overall performance due to the II=3 of QK stage and would lead to increased resource usage for pipelining.

Z Reduction: This two-phase stage sums the individual Z slices from each attention core to form the complete output Z vector. For a standard configuration of $H = 64$, parallel accumulation over H channels (because Z has H elements) would result in a stage duration of approximately $3 \times 2w$ which is 8x that of QK and SV stages of $3 \times H$ cycles. To maintain pipeline balance, the reduction is split into two substages, ZRED1 and ZRED2. In ZRED1, Z slices are grouped by each H of them and processed with H accumulation channels per group, which results in approximately an overall latency of $3 \times H$ cycles. ZRED2 then combines the outputs from ZRED1 into the final Z vector.

Row sum: Operating in parallel to Z reduction, the Row Sum stage computes the sum of S' values from the attention cores. With $2w$ elements of S', this stage employs a similar two-stage approach as Z reduction for timing balance, comprising ROWSUM1 and ROWSUM2.

Division and Output: The final stage divides each Z element by the corresponding sum of the S' row, as per the post-fusion algorithm. The division is pipelined at a 2-cycle interval because better throughput is unnecessary. The output vector is then written back to HBM or DRAM.

Table 1 presents the timing for each pipeline stage based on the Xilinx Vitis HLS synthesis tool report. The design uses half-precision 16-bit floating-point data, with default settings of head dimension $H = 64$ and window width $2w = 512$. The overall pipeline is well balanced and timed at 201 cycles, predominantly due to the longer stage, QK.

Table 1: The timing (in cycles) of the pipeline stages

LOAD	QK	SV	ZRED1	ZRED2	DIV&OUT
66	201	197	195	66	179
			ROWSUM1	ROWSUM2	
			195	27	

4.1 Parameterized design

SWAT's architecture integrates basic window attention with additional attention mechanisms to improve accuracy across various tasks. One such mechanism is *global attention*, which designates important global tokens to be attended by all input tokens. Models like Longformer [1] and ViL [21] have demonstrated the effectiveness of global attention in enhancing accuracy for text classification and vision tasks, respectively. Another mechanism, *random attention*, introduced by the BigBird model [20], generally improves model accuracy by incorporating randomly (but statically) selected additional tokens for each input token to attend to.

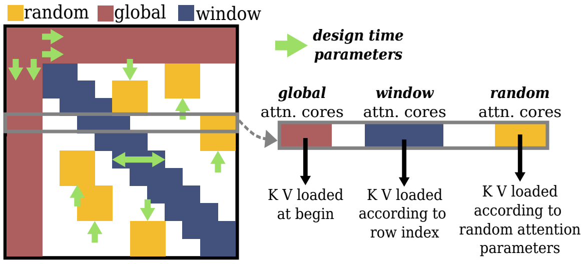


Figure 7: Parameterized Design of SWAT

As illustrated in Figure 7, SWAT supports these mechanisms through design-time parameters: the indices of the random and global attention tokens, as well as the width of the sliding window, are set as synthesis parameters. SWAT's architectural design can adapt to these additional attention patterns with minimal changes. Specific subsets of attention cores are allocated for computing global and random attention with respect to the parameters. Attention cores dedicated to global attention have fixed K and V buffers, aligning with the consistent nature of global tokens. These buffers are pre-loaded prior to the attention computation, minimizing performance impact. In contrast, attention cores handling random attention update their K and V buffers dynamically, which increases the latency of the LOAD stage to 195 cycles from the initial 66. However, thanks to the pipelined design of our system, this increase in latency does not hamper overall execution speed.

4.2 FPGA resources utilization

Table 2 provides a detailed account of resource usage on the Alveo U55C FPGA post-synthesis. We present four configurations: the standard Longformer setup of pure window attention with FP16 datatype and 512 attention cores; the BigBird configuration of 192 sliding window tokens, 192 random attention tokens, 128 global tokens, i.e. total 512 tokens per row with FP16; the same BigBird configuration but with dual pipelines for parallel processing two heads (which also demonstrates the potential of handling 1024 tokens per row in different attention configurations); and an FP32 version for later comparative analysis with GPUs.

Table 2: Resources usage on U55C/VCU128

Design	DSP	LUT	FF	BRAM
FP16 (512 attn)	19%	38 %	11%	25%
FP16 (BigBird 512 attn)	19%	33 %	11	25%
FP16 (BigBird 2 x 512 attn)	38%	66 %	22	50%
FP32 (512 attn)	49%	67%	23%	25%
Butterfly (FP16, 120-BE)	32%	79%	63%	49%

5 EVALUATION

5.1 Butterfly accelerator baseline

The Butterfly Accelerator[7] is the only FPGA-based accelerator for static sparse attention–the butterfly sparsity[3]– and serves as our baseline. It incorporates two key hardware components: the FFT-BTF (Fast Fourier Transform-Butterfly) engine for **approximating** the standard SoftMax attention using Fourier transform; and the ATTN-BTF (Attention-Butterfly) engine that behaves just as the standard SoftMax attention. The FFT-BTF offers increased speed at the expense of some accuracy, whereas the ATTN-BTF ensures accuracy with slower operation. The hybrid use of FFT and SoftMax layers in Butterfly's software model is tuned for specific datasets to achieve a balance between speed and accuracy through design space exploration. However, the performance study of the Butterfly Accelerator in [7] focuses only on the full-FFT version.

5.2 Accuracy comparison with Butterfly

To draw a fair comparison with SWAT, we delve into the accuracy-performance tradeoff in Butterfly's adaptable design. We evaluated model accuracies using the Long-Range Arena (LRA) benchmark datasets [15], which are tailored for efficient transformer models. Table 3 shows the **accuracy advantage** of SWAT implementations over the purely FFT-layered Butterfly model, particularly in vision tasks. Additionally, we observe an accuracy improvement in the Butterfly model when replacing one or two last FFT layers with traditional SoftMax attention layers (respectively noted as configuration BTF-1 and BTF-2), underscoring the accuracy benefit of incorporating even a single layer of traditional SoftMax attention. However, Longformer and Bigbird still show better average accuracy compared to BTF-1 and BTF2, especially in vision tasks. For the accuracy consideration, we will use BTF-1 and BTF-2 in the performance analysis in the next section.

Table 3: Accuracy gain of window attention-based models (Longformer and BigBird supported by SWAT) and baseline Butterfly models with one or two layers (BTF-1, BTF-2) replaced by the vanilla SoftMax attention on LRA datasets compared to Butterfly's full-FFT attention

	Vision based		Text based		
Model	Image	PathFinder	Text	ListOps	AVG.
Longformer	+15.26%	+3.03%	+0.17%	+1.61%	+5.02%
Bigbird	+13.87%	+8.16%	+1.34%	+2.03%	+6.35%
BTF-1	+6.26%	+2.85%	+0.01%	+2.4%	+3.01%
BTF-2	+8.95%	+2.14%	+1.05%	+2.42%	+3.64%

Table 4: Top-1 accuracy of PixelFly (butterfly model) against ViL (supported by SWAT) on ImageNet-1K [6]

Model	Params	Top-1
ViL-Tiny	6.7M	76.7%
Pixelfly-M-S	5.9M	72.6%
ViL-Small	24.6M	82.4%
Pixelfly-V-S	16.9M	77.5%
Pixelfly-M-B	17.4M	76.3%
Pixelfly-V-B	28.2M	78.6%
ViL-Med	39.7M	83.5%

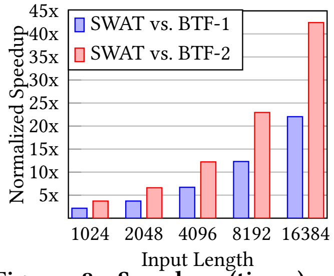


Figure 8: Speedup (times) of SWAT over Butterfly versions

In addition to the Butterfly model comparison, we explored the effectiveness of the sliding window attention versus FFT-based approximations in vision-specific tasks. Our analysis, detailed in Table 4, compares the accuracy of the state-of-the-art ViL (Vision Longformer) model [21], which is supported by SWAT, against the SOTA FFT-attention-based Pixelfly model [4]. This comparison is particularly insightful as both models operate with a similar number of parameters. The results highlight that the ViL model achieves superior accuracy on the ImageNet-1K dataset, underscoring the effectiveness of sliding window attention in vision applications.

5.3 Performance comparison with Butterfly

Both SWAT and Butterfly accelerators were synthesized on FPGAs of the same characteristics[3]. They are also using a similar number of FPGA resources in FP16 precision, as shown in Table 2. Using the cycle-accurate simulator provided by Butterfly, we independently evaluated the performance of the FFT-BTF engine and ATTN-BTF engine. As the original performance evaluation of Butterfly only considered the full-FFT configuration, we project its performance by computing the optimal ratio of resource distribution for FFT-BTF and ATTN-BTF engines at different input lengths.

[3] U55C (SWAT) and VCU128 (Butterfly) have the same number of logical resources

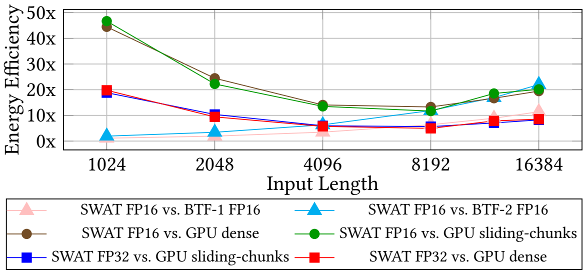


Figure 9: Energy Efficiency of SWAT against SOTA GPU and FPGA implementations

SWAT's latency is based on its pipeline latency as stated earlier in Table 1. For FPGA implementations, both accelerators produce consistent operation latencies regardless of the concrete values of input data, number of heads, layers, and batches. Total attention time is proportional to the execution time of a single head, which has an arbitrarily chosen dimensionality of 64. Figure 8 shows the speedup of SWAT in Longformer or Bigbird configuration against the Butterfly accelerator in BTF-1 (1 softmax layer) and BTF-2 (2 softmax layers) configurations across various sequence lengths, from 1024 to 16384 tokens. Due to the poor scalability of the vanilla SoftMax attention, the Butterfly accelerator exhibits declining performance with long input sequences. At the standard Longformer configuration of 4096 input tokens (the accuracies in Table 3 are obtained at this configuration), SWAT performs 6.7× and 12.2× better respectively over BTF-1 and BTF-2. Due to the faster computation of SWAT, it also outperforms Butterfly from the energy perspective. We evaluate the power consumption of SWAT using the Xilinx Power Estimator. Figure 9 shows the energy efficiency (energy consumption per attention) of SWAT against BTF-1 and BTF-2. SWAT shows an increasing energy efficiency advantage along the input length, attaining 11.4× and 21.9× over BTF-1 and BTF-2 at 16384 context length, respectively.

5.4 Comparison with GPU

We benchmarked SWAT against GPU implementations with the same Transformer model, using AMD's rocBLAS and MIOpen libraries for tensor multiplication and SoftMax operation in the sliding chunks and the naïve dense approach which have been discussed in Section 1. We measured the execution time while excluding the overhead due to the first kernel launch and averaged the latency over 100 attention computations for consistency. To compare fairly with GPU implementation, we synthesized an FP32 version of SWAT, which exhibits a higher pipeline latency of 264 cycles due to the FPGA's limitation on the FP32 MAC operation. The execution time comparison has been shown previously in Figure 3. At short input length, SWAT demonstrates better latency, which can be partly ascribed to the underutilization of the GPU in our single-batch experimental setup. However, as the input length reaches 4k, the GPU's execution time begins to rise sharply, indicating its full utilization. In contrast, SWAT exhibits a linearly increasing execution time in relation to input length, with similar performance of GPU between 4k and 8k input length but much better scalability for longer input length.

A key aspect of SWAT is its energy efficiency, which is notably remarkable when compared to MI210, which has a power consumption of 300 watts. This efficiency is highlighted in Figure 9, where SWAT's energy efficiency is compared against MI210 in both FP32 and FP16 precision. In FP32 precision, particularly considering the under-utilization of the GPU at shorter input lengths, SWAT achieves an impressive 20× energy efficiency advantage at an input length of 1k. However, as the GPU becomes better utilized with longer input sequences, SWAT's relative energy efficiency advantage decreases, reaching a minimum of 4.2× at an input length of 8k. SWAT's scalability, however, becomes increasingly pronounced with longer context lengths, and hence SWAT's superior energy efficiency grows, reaching up to approximately 8.4× that of GPU-based implementations at 16k input length.

6 CONCLUSION

We introduced SWAT, an FPGA-based accelerator specifically designed for window attention-based transformer models. Our approach is rooted in a comprehensive analysis of window attention workloads, leading to an input-stationary dataflow. This dataflow combines the advantages of FPGA's inherent distributed memory-and-computation architecture with a row-major dataflow and kernel fusion optimization. This unique combination effectively leverages the diagonal-structured sparsity inherent in sliding window attention, resulting in significantly improved performance. For long context lengths, SWAT stands out by delivering superior performance and energy efficiency over both the current SOTA FPGA-based static sparse attention accelerator and server-class GPUs.

REFERENCES

[1] Iz Beltagy et al. 2020. Longformer: The long-document transformer. *arXiv preprint arXiv:2004.05150*.

[2] Adrian M Caulfield et al. 2016. A cloud-scale acceleration architecture. In *MICRO-49*. IEEE, 1–13.

[3] Tri Dao et al. 2019. Learning fast algorithms for linear transforms using butterfly factorizations. In *ICML*. 1517–1527.

[4] Tri Dao et al. 2021. Pixelated butterfly: Simple and efficient sparse training for neural network models. *arXiv*.

[5] Tri Dao et al. 2022. Flashattention: Fast and memory-efficient exact attention with io-awareness. *NeurIPS-35*.

[6] Jia Deng et al. 2009. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*. Ieee, 248–255.

[7] Hongxiang Fan et al. 2022. Adaptable Butterfly Accelerator for Attention-based NNs via Hardware and Algorithm Co-design. In *MICRO-55*. IEEE, 599–615.

[8] Sehoon Kim et al. 2023. Full Stack Optimization of Transformer Inference. In *Architecture and System Support for Transformer Models (ASSYST@ ISCA 2023)*.

[9] James Lee-Thorp et al. 2021. Fnet: Mixing tokens with fourier transforms. *arXiv preprint arXiv:2105.03824*.

[10] Liqiang Lu et al. 2021. Sanger: A co-design framework for enabling sparse attention using reconfigurable architecture. In *MICRO-54*.

[11] Niki Parmar et al. 2018. Image transformer. In *International conference on machine learning*. PMLR, 4055–4064.

[12] Yubin Qin et al. 2023. FACT: FFN-Attention Co-optimized Transformer Architecture with Eager Correlation Prediction. In *ISCA-50*. 1–14.

[13] Zheng Qu et al. 2022. Dota: detect and omit weak attentions for scalable transformer acceleration. In *ASPLOS-27*. 14–26.

[14] Guan Shen et al. 2022. SALO: an efficient spatial accelerator enabling hybrid sparse attention mechanisms for long sequences. In *DAC-59*. 571–576.

[15] Yi Tay et al. [n. d.]. Long range arena: A benchmark for efficient transformers. *arXiv preprint arXiv:2011.04006*.

[16] Marcos Treviso et al. [n. d.]. Efficient methods for natural language processing: A survey. *TACL* 11, 826–860.

[17] Ashish Vaswani et al. 2017. Attention is all you need. *NeurIPS* 30.

[18] Sinong Wang, , et al. 2020. Linformer: Self-attention with linear complexity. *arXiv preprint arXiv:2006.04768*.

[19] Haoran You et al. [n. d.]. Vitcod: Vision transformer acceleration via dedicated algorithm and accelerator co-design. In *HPCA 2023*. 273–286.

[20] Manzil Zaheer et al. 2020. Big bird: Transformers for longer sequences. *Advances in neural information processing systems* 33, 17283–17297.

[21] Pengchuan Zhang et al. [n. d.]. Multi-scale vision longformer: A new vision transformer for high-resolution image encoding. In *ICCV 2021*. 2998–3008.