Adaptable Butterfly Accelerator for Attention-based NNs via Hardware and Algorithm Co-design

Hongxiang Fan
Imperial College London
London, UK
h.fan17@imperial.ac.uk

Thomas Chau
Samsung AI Center
Cambridge, UK
thomas.chau@samsung.com

Stylianos I. Venieris
Samsung AI Center
Cambridge, UK
s.venieris@samsung.com

Royson Lee
University of Cambridge
Cambridge, UK
dsr12@cam.ac.uk

Alexandros Kouris
Samsung AI Center
Cambridge, UK
a.kouris@samsung.com

Wayne Luk
Imperial College London
London, UK
w.luk@imperial.ac.uk

Nicholas D. Lane
Samsung AI Center &
University of Cambridge
Cambridge, UK
nic.lane@samsung.com

Mohamed S. Abdelfattah
Cornell University
New York, NY, United States
mohamed@cornell.edu

Abstract—Attention-based neural networks have become pervasive in many AI tasks. Despite their excellent algorithmic performance, the use of the attention mechanism and feed-forward network (FFN) demands excessive computational and memory resources, which often compromises their hardware performance. Although various sparse variants have been introduced, most approaches only focus on mitigating the quadratic scaling of attention on the algorithm level, without explicitly considering the efficiency of mapping their methods on real hardware designs. Furthermore, most efforts only focus on either the attention mechanism or the FFNs but without jointly optimizing both parts, causing most of the current designs to lack scalability when dealing with different input lengths. This paper systematically considers the sparsity patterns in different variants from a hardware perspective. On the algorithmic level, we propose FABNet, a hardware-friendly variant that adopts a unified butterfly sparsity pattern to approximate both the attention mechanism and the FFNs. On the hardware level, a novel adaptable butterfly accelerator is proposed that can be configured at runtime via dedicated hardware control to accelerate different butterfly layers using a single unified hardware engine. On the Long-Range-Arena dataset, FABNet achieves the same accuracy as the vanilla Transformer while reducing the amount of computation by $10 \sim 66 \times$ and the number of parameters $2 \sim 22 \times$. By jointly optimizing the algorithm and hardware, our FPGA-based butterfly accelerator achieves $14.2 \sim 23.2 \times$ speedup over state-of-the-art accelerators normalized to the same computational budget. Compared with optimized CPU and GPU designs on Raspberry Pi 4 and Jetson Nano, our system is up to $273.8 \times$ and $15.1 \times$ faster under the same power budget.

Keywords—Adaptable Butterfly Accelerator, Attention-based Neural Networks, Butterfly Sparsity, Algorithm and Hardware Co-Design

I. INTRODUCTION

Recent years have witnessed a great success of attention-based neural networks (NNs) on many AI tasks [1]. The attention mechanism [2] that captures long-range information from sequences of data has demonstrated its excellent algorithmic performance in various natural language processing [3], [4] and computer vision [5] applications. However, the advances of attention-based NNs come at a cost: the use of attention and linear layers significantly increases the computational load, resulting in a large overhead on their speed and power consumption [6]. Figure 1 shows an operation breakdown on four mainstream attention-based models. For short input sequences, linear layers occupy over 80% of operation counts. As the input sequence increases, the computation is gradually dominated by the attention layer. Since both attention and linear layers are memory- and compute-intensive, it is challenging to achieve high hardware performance on attention-based NNs across input sequences of various lengths.

So far, various approaches and designs have been introduced to accelerate attention-based DNNs. On the algorithmic level, several efficient sparse variants have attempted to reduce the computational complexity [7]–[13]. However, most of these approaches only focus on reducing the number of parameters and operations without considering the real hardware performance, such as end-to-end latency. Furthermore, the hardware efficiency of implementing these sparsity patterns on real hardware designs is often overlooked. On
the hardware level, although various highly-optimized accelerators (Table I) have been proposed \cite{6, 14–19}, several issues still remain unresolved:

- Most of current accelerators only focus on optimizing either FFNs \cite{19} or the attention mechanism \cite{6, 14–18}. Without jointly optimizing both parts, these hardware designs lack scalability when accelerating the end-to-end attention-based NNs with different input lengths.
- While optimizing the attention mechanism, most of the existing designs dynamically detect and prune the redundant computation at runtime to achieve high sparsity on specific datasets and networks. However, the generality of these dynamic approaches needs to be further tested as their performance gain may vary across different datasets and network architectures.
- The sparsity patterns introduced by these dynamic approaches are often unstructured, requiring dynamic hardware controllers to exploit sparsity. Such complicated controllers often contain larger numbers of clocking elements, and their hardware overhead increases as the transistor size reduces \cite{21}. As such, the performance or energy gain of these dynamic methods may be diminished.

To address the aforementioned issues, this paper adopts butterfly sparsity to accelerate attention-based models with three novel aspects (Table I): \textit{i)} fine-grained structured regularity, which possesses regular data accesses to optimize both memory and compute efficiency; \textit{ii)} static sparsity pattern, which avoids the need of designing a dynamic controller in hardware; \textit{iii)} sparsity exploitation on both attention and linear layers, which allows scalable end-to-end acceleration of attention-based NNs. We therefore propose \textit{FABNet}, a hardware-friendly model for FFT, \textit{ATTention and BUTterfly-Net}. To fully exploit the sparsity in hardware, we propose an adaptable butterfly accelerator that can be configured at runtime via dedicated hardware control to accelerate different layers using a single unified engine, significantly improving hardware efficiency. To push the performance limit, we jointly optimize the model and hardware via a co-design approach. Overall, this work makes the following contributions:

- A hardware-friendly attention-based model, \textit{FABNet}, that adopts the butterfly sparsity pattern on both attention and linear layers for end-to-end acceleration (Section III).
- A novel adaptable butterfly accelerator configurable at runtime via dedicated hardware control to accelerate different layers using a single unified engine (Section IV).
- Several hardware optimizations to improve the hardware efficiency and a co-design approach to jointly optimize both algorithmic and hardware parameters (Section V).
- A comprehensive evaluation on different datasets that demonstrates the advantages of our approach over CPU, GPU and state-of-the-art accelerators (Section VI).

### II. BACKGROUND AND MOTIVATION

#### A. Attention-Based Neural Networks

Based on their network structure, attention-based NNs can be classified into three categories: \textit{i)} encoder-decoder, \textit{ii)} encoder-only, and \textit{iii)} decoder-only networks. The encoder-decoder NNs are mainly designed for sequence-to-sequence tasks, such as machine translation \cite{2}. One of the most widely used encoder-decoder network is the Transformer, which is constructed by a stack of encoder and decoder blocks. Figure 2 illustrates the structure, where \( N_e, D_{hid} \) and \( R_{ffn} \) represent input length, hidden size and FFN expand ratio respectively. Each encoder starts with a multi-head attention module, followed by a feed-forward network (FFN) consisting of two linear (fully-connected) layers. Finally, residual addition \cite{22} and layer normalization (LN) \cite{23} are used after FFN. Within each multi-head attention, the inputs are first mapped to query (Q), key (K) and value (V) matrices through three different linear layers. The query matrix is then multiplied with \( K^T \), followed by a softmax operation to get the score matrix (S). The generated S is multiplied with V and the resultant matrix will flow into another linear layer, which generates the final output matrix of the multi-head attention. Similar to the encoder, the decoder employs two multi-head attention modules and one FFN, where the difference is that the inputs of the query and key matrices in the second attention module come from the last encoder.

Based on the original encoder-decoder structure of Transformer, different variants have been proposed. The encoder-only networks, such as BERT \cite{3} and XLM \cite{24}, are autoencoding models that have been widely applied to NLP tasks, such as sequence classification \cite{25}. The Vision Transformer (ViT) \cite{5} also lies in this category. An extra linear projection layer is introduced at the beginning, while its encoder layers correspond to the encoder part of the original Transformer. Finally, the decoder-only networks represent the autoregressive models designed for NLP tasks, such as language modeling \cite{26}. GPT \cite{4} is a typical decoder-only

### Table I

**COMPARISON OF EXISTING ACCELERATORS FOR ATTENTION-BASED NNs IN TERMS OF SPARSITY REGULARITY, PATTERN AND LOCATION.**

| Accelerators | Pattern Regularity | Sparsity Pattern | Sparsity Location |
|--------------|--------------------|------------------|------------------|
| A² \cite{14} | unstructured       |                  |                  |
| SpAtten \cite{6} | coarse-grained structured |                  |                  |
| Sanger \cite{15} | load-balanced unstructured | dynamic | attention |
| Energon \cite{16} | unstructured       |                  |                  |
| DOTA \cite{18} | unstructured       |                  |                  |
| FTRANS \cite{19} | None              | static           | FFN              |
| EdgeBERT \cite{20} | None              | dynamic          | layer            |
| **Our work** | **fine-grained structured** | **static**       | **attention & FFN** |
Figure 2. The structure of a Transformer. Shortcut addition and layer normalization are omitted for simplicity.

model that corresponds to the decoder part of the original Transformer. Although we focus on encoder-only networks in this work, our hardware design is flexible and applicable to decoders too.

B. Butterfly Matrices and FFT

Despite the impressive accuracy attained using attention-based NNs, these models are expensive and not scalable, e.g. the self-attention mechanism in the Transformer scales quadratically in terms of computation and memory as a function of the input sequence length. As a result, numerous works [7], [8], [13], [27] adopt structured linear mappings, such as sparse and low-rank matrices, to approximate the attention matrices and/or the weight matrices in the feed-forward layers. Choosing an appropriate structure for each linear mapping, however, is application-dependent, often requiring domain expertise and entailing an arduous process of hand-picking solutions as different structures have different trade-offs in accuracy and speed.

To counteract this, recent work has utilized butterfly matrices [38], [39], which are universal representations of structured matrices that have a simple recursive structure. Specifically, each butterfly matrix \( W_{\text{bfly}} \) of size \( N \) encodes the recursive divide-and-conquer structure with butterfly patterns and, hence it can be expressed as the product of sparse butterfly factor matrices [30] as follows:

\[
W_{\text{bfly}} = \begin{pmatrix}
  W'_N & \cdots & W'_2 & \cdots & \cdots & \cdots & W'_2 & \cdots & W'_N
\end{pmatrix}
\]

where each \( W'_N \), a butterfly factor, is a \( 2 \times 2 \) block matrix of diagonal matrices, \( B^i \) with size \( N/2 \), whose entries can be trained via gradient-based methods:

\[
W'_N = \begin{bmatrix}
B^2_{N/2} & B^4_{N/2} & \cdots & B^2_{N/2} & B^4_{N/2}
\end{bmatrix}
\]

Due to their expressiveness in representing structured matrices and approximating unstructured data, butterfly matrices and their variants [31], [32] have found success in compressing attention and weight matrices, considerably improving the accuracy and efficiency of attention-based NNs. For instance, applying butterfly factorization to a linear layer with an \( M \times M \) weight matrix can reduce the computational and memory complexity from \( O(M^2) \) to \( O(M \log M) \).

Besides attention and weight matrices, some designs have explored replacing the entire attention mechanism with more efficient counterparts [33]. A prominent example is FNet [34], in which the self-attention modules are replaced by 2D Discrete Fourier Transform (DFT) operations. Specifically, for each input, 1D DFT is applied along the sequence and the hidden dimension independently, keeping only the real component of the resulting outputs. To reduce DFT computation time, the Cooley-Tukey Fast Fourier Transform (FFT) algorithm [35] is used. As the use of DFT facilitates information flow across all embeddings, it results in a similar performance compared to the use of vanilla self-attention layers, but at a significant reduction in latency and memory.

On the algorithmic front, our proposed FABNet utilizes a mixture of these techniques – FFT and butterfly matrices – to outperform relevant approximation approaches in terms of accuracy. Notably, since FFT matrices can be considered a special case of butterfly matrices with \( B^2_{N/2} \), \( B^4_{N/2} \) being identity matrices and \( B^2_{N/2} \), \( B^4_{N/2} \) acting as twiddle factors, both the FFT and butterfly matrices possess the recursive butterfly structure. Therefore, it is possible to use a unified computational and data access pattern and then devise a single hardware engine to accelerate both FFT and butterfly-based operations with high hardware efficiency.

C. Latency Breakdown and Motivation

The operation counts in Figure 1 reveal that the computation of attention-based NNs is dominated by different components when the length of input sequences changes. To further investigate the real hardware performance of each subcomponent, we profile the execution time of the BERT-Large model on the Nvidia V100 GPU and Intel Xeon Gold 6154 CPU. The length of input sequences is set to 256, 1024 and 2048 on both devices, and the batch size for GPU and CPU is 8 and 1, respectively. Figure 3 shows the latency breakdown. We split the latency consumption into three main subcomponents: attention layers, linear layers, and other operations, e.g. layer normalization, residual connections, matrix transformations and IO operations. Notably, on both CPU and GPU, linear layers take up a significant portion of execution time, 67.9% and 79.34% respectively, when the input length is small. As the input length becomes larger, the execution time of attention layers increases gradually and becomes dominant. As such, the latency is dominated by different components depending on the length of the input sequence. According to Amdahl’s law [36], to achieve high hardware performance across different input lengths, it is necessary to optimize both attention and linear layers.
The majority of previous accelerators for attention-based NNs focused on optimizing a single component of the entire model (either attention or FFN as shown in Table I), leading to suboptimal end-to-end performance gains. The execution time of these accelerators is heavily dependent on the input length which varies across different applications, reducing the scalability of these hardware designs and thus narrowing their deployability in real-world scenarios. Naively adopting a combination of previous works on optimizing the linear layers \cite{21} and attention layers \cite{6,14–18}, however, would result in low hardware efficiency as they adopt different sparsity patterns. As a result, designing an end-to-end accelerator for scalable attention-based NNs remains an open problem. In this work, we address this challenge by adopting an algorithm and hardware co-design approach. On the algorithmic level, a hardware-friendly model called FABNet is proposed, which adopts a unified butterfly sparsity pattern to compress both attention and linear layers. On the hardware level, we propose an adaptable butterfly design that can be configured at runtime to accelerate different layers in FABNet using one unified hardware engine.

III. ALGORITHM OPTIMIZATION

A. Computational Analysis of Sparsity Patterns

Various pruning schemes have been proposed to reduce the computational complexity of attention-based NNs, leading to different efficient models \cite{7–13,31,32,34,37}. By analysing the computational and data access patterns of these variants, we define five basic sparsity patterns shown in Figure 4: i) low rank, ii) sliding window, iii) butterfly, iv) random, and v) block-wise pattern. As low-rank approximation of an attention matrix requires both sequential row and column reads but the data are usually only stored in either a row-major or column-major, the hardware efficiency of low-rank sparsity is inherently diminished. Random sparsity also demonstrates low hardware efficiency due to its random read pattern. Furthermore, we observe that the sparsity in various sparse variants can be expressed as different combinations of the basic sparsity patterns, as summarized in Table II. As some basic sparsity patterns can only capture either long-range global or short-range local information (Figure 4), the rationale behind using multiple sparsity patterns within each variant is mainly to compensate for the underlying accuracy loss. For example, Pixelfly \cite{31} introduces an additional low-rank sparsity pattern to increase the expressiveness of their flat block-wise butterfly pattern and improve accuracy.

Different sparsity patterns exhibit diverse data access patterns, which calls for custom hardware support. However, supporting multiple sparsity patterns may complicate the hardware design. For instance, in order to fully utilize the sparsity in the random pattern, complex dynamic controllers are required to achieve a load-balanced execution on different hardware engines \cite{15,38}. The extra overhead of such controllers may counteract the improvement brought by skipping sparse operations \cite{21}.

In this work, we aim to find a hardware-friendly sparsity pattern that: 1) has structured data access patterns to simplify the memory design, 2) captures both local and global range information with a single sparsity pattern, and 3) is applicable to both the attention mechanism and FFNs to sustain its performance improvement across both long and short input sequences. To meet these requirements, we adopt the butterfly sparsity as a basis for constructing our efficient algorithm.

Compared to other sparsity patterns, the butterfly sparsity provides a number of favorable properties. As shown in Figure 4, although random sparsity is able to capture both

---

**Figure 3.** Execution time breakdown of Transformer with different input lengths on GPU and CPU.

**Figure 4.** Basic sparsity patterns in existing variants.

**Table II** COMBINATION OF SPARSITIES IN DIFFERENT VARIANTS.

| Model       | Sparsity pattern | Att. FFN | Unified Sparsity | Co-Design |
|-------------|------------------|---------|------------------|-----------|
| Performer   | Low-Rank (Extra kernels) | ✓       | ✗                | ✗         |
| Reformer    | Block-wise (Extra kernels) | ✓       | ✗                | ✗         |
| Sparse Sinkhorn | Block-wise + Random | ✓       | ✗                | ✗         |
| Longformer  | Sliding-Window + Low-Rank | ✓       | ✓                | ✓         |
| BigBird     | Random + Sliding-Window + Low-Rank | ✓       | ✓                | ✓         |
| FNet        | Butterfly        | ✓       | ✓                | ✓         |
| Kaleidoscope| Butterfly        | ✗       | ✓                | ✓         |
| Sparse Trans| Low-Rank + Butterfly + Sliding-Window | ✓       | ✓                | ✓         |
| PixelFly    | Butterfly + Block-Wise + Low-Rank | ✓       | ✓                | ✓         |
| Our work    | Butterfly        | ✓       | ✓                | ✗         |

---

![Image](image-url)
local and global information, it has two drawbacks compared to butterfly sparsity: 1) it requires complicated controllers with excessive hardware overhead [21], and 2) its performance gain cannot be guaranteed as the sparsity may vary substantially among different datasets and tasks. Compared with random sparsity, the sliding-window pattern is more hardware-friendly. However, Table [1] shows that it often requires low-rank sparsity to compensate for the accuracy loss, as sliding-window sparsity only captures the local relationship within each window. Moreover, although some variants adopt a single low-rank or block-wise sparsity pattern with satisfactory algorithmic performance, they require extra algorithmic operations and dedicated computational kernels during inference (e.g. the locality-sensitive hashing (LSH) in Reformer [9]) during inference, resulting in large hardware overhead. In contrast, this paper treats the butterfly sparsity as a promising method due to its regular data access pattern and the ability of capturing both global and local information.

B. Unified Butterfly Pattern for Attention and Linear Layers

The butterfly pattern has demonstrated its effectiveness and generality in approximating linear transformations [32]. Furthermore, Lee-Thorp et al. [34] have shown the potential of simplifying the computation by replacing the entire attention layer with Fourier transform, which effectively mixes tokens without explicitly approximating the attention mechanism. To maximize the ability to reduce the computation with acceptable algorithmic performance, we start by proposing two basic building blocks for scalable inference: 1) the Attention Butterfly (ABfly), and 2) Fourier Butterfly (FBfly) blocks.

In the ABfly block, we retain the backbone of the attention module and compress all the linear layers using butterfly factorization. Specifically, the ABfly block starts with three butterfly linear layers to generate Q, K and V matrices. The results are fed into a vanilla multi-head attention layer and another butterfly linear layer to obtain the relationships among different tokens. A butterfly FFN that consists of two butterfly linear layers is placed at the end of the ABfly block for additional processing. To further reduce the amount of computation and number of parameters, we replace the attention module with a 2D Fourier transform layer, implemented using FFT, resulting in a more compute-efficient block called FBfly. The use of FFT effectively mixes different input tokens, which allows the following butterfly FFN to process a longer sequence. More importantly, all computation in the FBfly block, which use the FFT’s twiddle factors and the butterfly linear layers’ weights, is performed using a unified butterfly pattern, resulting in higher hardware efficiency over previous works.

Although FBfly is less compute- and memory-intensive than ABfly, the use of the Fourier transform layer may degrade accuracy [34]. To preserve high accuracy, we propose a novel butterfly-based network called FABNet that introduces a hybrid of the ABfly and FBfly blocks, as depicted in Figure 5. There are $N_{ABfly}$ ABfly blocks at the beginning and $N_{FBfly}$ ABfly blocks stacked on top. With this setup, we expose both $N_{FBfly}$ and $N_{ABfly}$ as hyperparameters, enabling a trade-off between algorithmic and hardware performance. To optimize this trade-off, we develop a co-design method (Section V-C) that explores the design space of both neural architecture and hardware design.

IV. HARDWARE ACCELERATOR

A. Architecture Overview

Figure 6 shows the proposed hardware accelerator consisting of: a Butterfly Processor (BP), an Attention Processor (AP), a Post-processing Processor (PostP), the off-chip memory, and several on-chip buffers. BP consists of $P_{BE}$ number of Butterfly Engines (BEs), which are used to accelerate the computations that involve butterfly patterns, including both FFT and butterfly linear transformations. AP contains $P_{AE}$ number of Attention Engines (AEs), and each AE is composed of one QK unit and one SV unit. The QK unit is designed to implement the softmax and the matrix multiplication between queries and keys. The SV receives the outputs from the QK unit and multiplies the results with value vectors to generate the final results of the attention layer. The PostP module is responsible for executing the layer normalization and shortcut (SC) addition. To ease the on-chip memory consumption, the intermediate results between different FFT and butterfly operations are transferred back to the off-chip memory. Although doing so increases the bandwidth requirement, this ensures our accelerator is scalable on hardware platforms with limited on-chip memory. To improve the overall hardware performance, all the on-chip buffers utilize double-buffering to overlap the data transfer with the computation.

B. Adaptable Butterfly Engine

Figure 7 shows the hardware architecture of BE. Each BE is mainly composed of a butterfly memory system and $P_{BU}$ number of adaptable Butterfly Units (BUs). To improve the hardware efficiency and enable the use of a single unified engine, the BE module is designed with a focus on
adaptable butterfly unit.

2) Butterfly Memory System: Our butterfly memory system comprises an input manager, a serial-to-parallel (S2P) module, a parallel-to-serial (P2S) module and butterfly buffers. As shown in Figure 8a, the butterfly pattern requires different data access at different stages. The conventional column-major or row-major order will cause bank conflicts while reading the data. For instance, accessing index pair \(x_0\) and \(x_8\) of the first stage causes a read conflict in the column-major order as shown in Figure 8b, in which each row represents a memory bank. The row-major order also suffers from the same issue while reading \(x_0\) and \(x_2\) in the third stage.

To avoid such bank conflict, we introduce a custom data layout strategy and implement it using the S2P module shown in Figure 9. We permute each column \(i\) using a starting position \(P_1\), which indicates how many rows the first element in the current column should be shifted down. We
define the starting position using the following formula:

\[ P_0 = 0, \quad P_{2^n-1} = P_{2^{n-1}-1} - 1 \]

For each \( 2^n - 1 \sim (2^n - 1) \), the starting positions \( P_{2^n-1} \) are obtained by shifting \( P_{2^{n-1}-1} \) one position down, as shown in Figure 9. The starting positions are generated using a counter, and a bit-count and addition operations (Figure 9b). After packing the serial data together, S2P permutes them based on the starting positions.

Figure 10 presents an example with 16 inputs, where the data required by the first and second stage of the butterfly pattern are read from the buffers without bank conflicts. However, as the butterfly units receive data in pairs, an extra pairing is required after the S2P module. An example is the second output column \( \langle x_{11}, x_1, x_9, x_3 \rangle \) of the first stage in Figure 11. To pair indices, we design an index coalescing module before the butterfly units (Figure 11). Based on the index of each input, a bit-count and addition operation is used to calculate the corresponding shift position. Then, a crossbar coalesces the index pairs based on the indices and shift positions. To ensure the outputs generated from the butterfly units preserve the original order, a recover module is used before the data is written back.

V. OPTIMIZATIONS AND CO-DESIGN

A. Memory Sharing in Butterfly Buffers

We employ butterfly buffers to allow the overlap between data transfer and computation. To reduce the memory consumption and improve the hardware efficiency, the butterfly buffers are shared between both FFT and butterfly linear transformation. Nonetheless, as the data width of FFT is twice that of the butterfly linear transformation, different address mapping and overlapping strategies are required.

Figure 12 shows the proposed address mapping strategies for butterfly linear transformation and FFT. Assuming the bitwidth of real numbers is 16 bits, each input buffer is 16-bit wide. While processing butterfly linear transformations, input buffers A and B are used as two independent ping-pong banks with separate read and write ports (top right in Figure 12). In this manner, when input buffer A is used for computation, buffer B can start the input data transfer for the next batch, leading to the overlapping strategy shown in Figure 13. While processing FFT, since the data include both real and imaginary parts which require 32-bit read and write ports, we concatenate the lower parts of input buffer A and B as the first ping-pong bank for the storage of the data.
complex numbers. To improve the hardware efficiency, we further reuse the higher parts of both buffers as the second ping-pong bank. As the computation requires both read and write accesses, we adopt a different overlapping strategy that pipelines the output data transfer only with the input data load of the next batch (Figure 13b). By employing different address mapping and overlapping strategies for FFT and butterfly linear transformation, we maximise the hardware efficiency and performance.

B. Fine-Grained Pipelining between BP and AP

While executing the ABfly block, BP and AP are in use, performing butterfly linear transformation and attention matrix multiplication, respectively. To further improve performance when executing the ABfly block, we employ fine-grained pipelining between BP and AP.

Figure 14 illustrates the dataflow of BP and AP. In the naive implementation, the key (K), value (V) and query (Q) matrices are generated sequentially from BP. After Q, K and V are computed, AP starts the computation of $Q \times K^T$ and $S \times V$. To optimize this process, we reorder the execution sequence of linear layers such that $BP$ computes $K$ and $V$ at the beginning (Figure 14b). As $Q \times K^T$ can be decomposed into multiple vector matrix multiplications that multiply different rows of $Q$ with the entire matrix $K^T$, we can actually start the computation of $Q \times K^T$ once the first few rows of $Q$ become available. As such, the $Q \times K^T$ in AP can be pipelined with the computation of $Q$ in BP. At the same time, since $S$ is generated from the $QK$ unit in a row-by-row fashion, we can further pipeline the $Q \times K^T$ with $S \times V$, as the computation of $S \times V$ can start once the first few rows of $S$ are generated from the $QK$ unit. Assuming there are $M$ and $L$ rows in $Q$ and $K$ matrices, it takes $\frac{T(QK)}{M}$ and $\frac{T(SV)}{L}$ to compute one row in the $SV$ and $QK$ units, respectively. As such, the total latency reduction achieved is $\frac{M}{M-1}T(QK) + \frac{L}{L-1}T(SV)$ compared to the unoptimized non-pipelined implementation.

C. Algorithm and Hardware Co-Design

The overall design space of our end-to-end system is formed by FABNet’s hyperparameters and the butterfly accelerator’s hardware parameters. Specifically, the joint design space consists of: 1) the algorithm parameters, i.e. the hidden size ($D_{hid}$), the expand ratio of FEN ($R_{fn}$), the total number of blocks ($N_{total}$) and the number of ABfly blocks ($N_{ABfly}$) in FABNet, and 2) the hardware parameters, i.e. the parallelism of $BU$ ($P_{bu}$) and $BE$ ($P_{be}$) in BP, and the parallelism of the $QK$ ($P_{qk}$) and $SV$ ($P_{sv}$) units in AP.

To assess the trade-off provided by each design point, we need to evaluate its algorithmic performance (e.g. an accuracy metric), its latency and its resource consumption. During search, the algorithmic performance is obtained by training and evaluating FABNet, while the latency is estimated by utilizing a custom simulator built for our butterfly accelerator. To verify whether the design can be accommodated by the target FPGA device, we developed an analytical model to estimate the consumption of DSP blocks and on-chip memory (BRAMs). As DSPs are mainly consumed by the multipliers in AP and BP, we formulate its resource usage as:

$$DSP_{usage} = P_{be} \times P_{bu} \times 4 + P_{head} \times (P_{qk} + P_{sv})$$

where the value of 4 reflects the number of multipliers in each $BU$. The consumption of BRAM is mainly occupied by the shortcut buffer, query buffer, key buffer and different buffers in $BU$ including butterfly buffer and weight buffers, which can be formulated as:

$$BRAM_{usage} = (BRAM_{bfly} + BRAM_{weight}) \times P_{be} + BRAM_{key} + BRAM_{se} + BRAM_{query}$$
The proposed analytical resource model is only used during the design space exploration stage. At the end of the co-design process, the final performance is obtained by running synthesis and place-&-route on our design with the optimized configurations.

Figure 15 illustrates the proposed co-design approach. Given a target dataset, FPGA device and both algorithmic and hardware performance constraints, we employ exhaustive grid search to traverse the joint design space and find the Pareto-optimal set of algorithmic and hardware parameters. Each individual design point corresponds to a different compression ratio of FABNet and level of parallelism of the butterfly accelerator, and provides different accuracy, latency and resource consumption. The final output is the Pareto front of parameters for both FABNet and our butterfly accelerator that satisfies a given set of constraints.

VI. EVALUATION

A. Experimental Setup

Benchmarks. To evaluate the algorithmic and hardware performance of our approach on workloads with long sequences, we choose five tasks from Long-Range-Arena [39], including hierarchical data classification (ListOPs), byte-level text classification (Text), byte-level document retrieval (Retrieval), image classification for sequences of pixels (Image), classification of long-range spatial dependency (Pathfinder). The input sequences of these datasets range from 1024 to 4096.

Software Implementation. We implement the vanilla Transformer [3], FNet [34] and our FABNet models using PyTorch (v1.10) [40]. The pretrained models are obtained from Huggingface 4.16 [41]. The batch size is 256 for both Image and Pathfinder tasks, and 32 for the rest of datasets during training. The learning rate is set to 0.0001, except for the Image and Pathfinder tasks where we use 0.01 and 0.0005 respectively. Multiple Nvidia A100 and V100 GPUs are used for training. To use FFT cores on Nvidia GPUs, the PyTorch API “rfft” is used to implement the FFT operation required in both FNet and FABNet. The high-performance CUDA implementation [32] of butterfly linear transformation is adopted to accelerate both GPU training and inference. We define two models with different default settings: FABNet-Base (\(D_{\text{hid}} = 768, R_{\text{ffn}} = 4, N_{\text{total}} = 12, N_{\text{ABfly}} = 0\)) and FABNet-Large (\(D_{\text{hid}} = 1024, R_{\text{ffn}} = 4, N_{\text{total}} = 24, N_{\text{ABfly}} = 0\)).

Hardware Implementation. We implement our hardware accelerators using Verilog. To evaluate performance in different scenarios, two Xilinx FPGA boards are used in our experiments: VCU128 for cloud/server scenarios and Zynq 7045 for edge/mobile settings. Xilinx Vivado 2019.1 is used for synthesis and implementation. While the maximum clock frequencies of our designs depend on the particular FPGA board and resource consumption, all the FPGA designs are clocked at 200 MHz which is below the maximum. We obtain power consumption values using the Xilinx Power Estimator (XPE) tool and develop a cycle-accurate performance model to evaluate the speed performance, which is cross-validated with our RTL simulation results generated by Vivado. The memory accesses to external memory are also considered. We use 16-bit half-precision floating-point in our hardware designs. We deploy four multipliers in each BU. As the hidden dimension \(D_{\text{hid}}\) is usually at most 1024, we set the depth of butterfly, query and key buffers as 1024. Finally, the size of shortcut buffers is the same as butterfly buffers.

B. Algorithmic Performance

The FBfly introduced in Section III-B is an efficient alternative to the vanilla attention block. To evaluate its algorithmic impact on end-to-end models, we take a six-layer Transformer as an example and compress it with different numbers of FBfly blocks, starting from the last block to the first block. Figure 16 shows the accuracy results on LRA-Text and LRA-Image. Although the accuracy fluctuates with different numbers of compressed layers, FBfly shows higher accuracy than the non-compressed Transformer with 4 and 1 compressed layers on LRA-Text and LRA-Image, respectively, demonstrating the improved algorithmic performance of our approach on end-to-end models.

To obtain the best possible algorithmic performance of each model, we use the optimized configuration specified in [42] for both vanilla Transformer and FNet. We perform a sample grid search to optimize the hyperparameters of our FABNet. Table III presents the optimized accuracy of different models. FABNet achieves higher accuracy than both Transformer and FNet on three out of five tasks, including ListOPs, Retrieval and Image. On average, FABNet achieves the same accuracy as Transformer. To investigate the efficiency of FABNet, Figure 17 shows the compression

\[\text{Figure 16. Accuracy with different number of compressed layers.}\]
rate of our optimized FABNet over the vanilla Transformer and FNet in terms of floating-point operations (FLOPs) and model size (number of parameters). Compared with the vanilla Transformer, FABNet achieves around 10 ~ 60× reduction in FLOPs and 2 ~ 22× reduction in model size, depending on the target task. Furthermore, compared with FNet, FABNet reduces FLOPs by 2 ~ 10× and model size by 2 ~ 32×.

C. Effectiveness of Co-design

We evaluate the effectiveness of our co-design approach in finding the optimal algorithm and hardware designs. For demonstration, we use LRA-Text as the target dataset and VCU128 FPGA as the target device. We select $D_{\text{hid}}$, $R_{\text{ffn}}$, $N_{\text{ABfly}}$ and $N_{\text{total}}$ from $\{64, 128, 256, 512, 1024\}$, $\{1, 2, 4\}$, $\{0, 1\}$ and $\{1, 2\}$ respectively. Parameters for hardware parallelism ($P_{\text{be}}, P_{\text{bu}}, P_{\text{qk}}$ and $P_{\text{sv}}$) are chosen from $\{0, 4, 8, 16, 32, 64, 128\}$. Figure 18 shows the points in the accuracy-latency design space. The orange line represents the accuracy loss, which is constrained to be less than 1% compared with the vanilla Transformer. The Pareto front is indicated by the brown line and the other blue points represent designs with less optimized software-related hyperparameters (Figure 16) or hardware design parameters. Among the design points that satisfy the accuracy constraint, we choose the point with the lowest latency in the Pareto front as our point of comparison. Within our design space, the selected point is up to 10% more accurate than the points in the same latency range and up to 130× faster than points in the same accuracy range, underlining the advantages of our co-design approach. The runtime of the co-design process is around 10 hours on our GPU server. To get the configurations for the rest of the datasets in LRA, we constrain the overall accuracy loss to be less than 0.5% compared to the vanilla Transformer. The final models and designs are chosen as the configurations with the highest hardware performance ($\langle P_{\text{be}}, P_{\text{bu}}, P_{\text{qk}}, P_{\text{sv}} \rangle = \langle 64, 4, 0, 0 \rangle$) without violating the accuracy constraints. Unless mentioned otherwise, the remaining the sections report the algorithmic and hardware performance using these optimized configurations.

D. Comparison with Baseline Design

To evaluate the speedup brought by our algorithm (FABNet) and hardware (butterfly accelerator), we use a baseline design for comparison [3]. The baseline hardware is designed with multiple multiply-accumulate (MAC) units to accelerate the linear transform and the matrix multiplications between query, key and value vectors. Each MAC is composed of a multiplier array followed by an adder tree. The fine-grained intra- and inter-layer pipeline techniques [43] are used to optimize the hardware performance. We allocate the parallelism of each MAC unit according to its workload in order to achieve load-balanced execution between different pipeline stages. For a fair comparison,
we implement both baseline and butterfly accelerators on a VCU128 FPGA using 2048 multipliers. The high bandwidth memory (HBM) is used as the external memory. Both designs are clocked at 200 MHz. We evaluate both base (12 layers) and large (24 layers) versions of each model using four different input sequences (128, 256, 512 and 1024).

A speedup breakdown is shown in Figure 19. To demonstrate the improvement brought by our algorithm, we first evaluate both BERT-Base and FABNet on the baseline design. As the FFT is not supported in the baseline design, we implement the Fourier layers as linear layers by multiplying the input sequences with DFT matrices. Since the operation reduction brought by the algorithm is not fully utilized by the baseline design, FABNet results in a 1.6 $\sim$ 2.3× speedup compared to BERT. To further evaluate the improvement brought by hardware optimizations, we evaluate FABNet on our butterfly accelerator, showing 19.5 $\sim$ 53.3× speedup when compared to the baseline design. By combining both algorithm and hardware optimizations, the overall speedup of our approach is 30.8 $\sim$ 87.3× over the baseline design.

E. Comparison with GPU and CPU

We compare our butterfly accelerator against GPU and CPU in both edge and server scenarios. In the edge scenario, our butterfly accelerator is implemented on a Xilinx Zynq 7045 FPGA. DDR4 is used as external memory and 512 multipliers are used for computation. Nvidia Jetson Nano GPU and Raspberry Pi4 are used as the GPU and CPU platforms, respectively. In the server scenario, the butterfly accelerator is implemented on a Xilinx VCU128 FPGA. HBM is used as external memory and the design consumes 1920 multipliers. We use Nvidia V100 and TITAN Xp GPUs for comparison, with highly-optimized CUDA implementations [32]. FPGA designs are clocked at 200 MHz.

We evaluate both FABNet-Base and FABNet-Large using 128, 256, 512 and 1024 input sequences. Figure 20 shows the results in terms of speedup and energy efficiency. We represent energy efficiency using Giga operations per second (GOPS/Watt). In the edge scenario, our design on Zynq 7045 FPGA achieves 3.5 $\sim$ 8× speedup over Jetson Nano GPU and 36.6 $\sim$ 342.3× speedup over Raspberry Pi4\footnote{On FABNet-Large with long input sequences greater than 768, Raspberry Pi 4 suffers from out-of-memory (OOM) issues.}. At the same time, our design yields 7.0 $\sim$ 15.1× and 162.8 $\sim$ 273.8× higher energy efficiency than Jetson Nano and Raspberry Pi4, respectively. In the server scenario, our design on VCU128 is up to 8.0 and 9.0× faster and up to 74.0 and 79.4× more energy-efficient than the V100 and TITAN Xp GPU, respectively. In summary, the end-to-end speedup and energy efficiency gains on both edge and server scenarios under different input sequences highlight the scalability of our butterfly accelerator.

F. Comparison with SOTA Accelerators

Table 4 compares our butterfly accelerator with existing state-of-the-art (SOTA) accelerators in terms of speed and power consumption. Instead of comparing the effective throughput [6], [15], we use the end-to-end latency to represent the actual execution speed of the hardware. The energy efficiency is represented by the number of predictions per Joule (Pred./J). Following the experimental setting of [18], we compare all other SOTA accelerators on LRA-Image dataset with one-layer vanilla Transformer. Among these accelerators, only SpAtten [6] and DOTA [18] report the end-to-end performance. For the rest of the accelerators that
only support attention, we estimate their performance by reusing their available multipliers to accelerate FFN. Furthermore, in both [6] and [15], the authors compare different ASIC and FPGA designs based on the assumption that all the ASIC designs are clocked at 1 GHz with 128 multipliers. For a fair comparison, we follow the same assumption in our experiments. For designs with more than 128 multipliers, we follow the scaling approach of [6], [15] to linearly scale down its throughput to get their end-to-end performance. For instance, DOTA [18] achieves 11.4× speedup over Nvidia V100 using 12,000 multipliers with 12 TOPS throughput. We scale down its throughput by 12,000/128 = 93.75, which leads to 0.123× speedup over V100. To obtain the power consumption, we use the same linear scaling approach. For instance, Sanger [15] reports the power consumption of a design with 1024 multipliers. We divide the power consumption of their systolic array (2243 mW) by 1024/128 = 8, which leads to 280.375 mW. Together with the power of other modules such as pre-processing and memory, their total power consumption is 0.801 W. To match the computational capacity of ASIC designs, we use 640 DSPs in the VCU128 FPGA. As our FPGA-based design is clocked at 200 MHz, this ensures that we have the same 640 × 200M = 128 GOPS theoretical peak performance as ASIC designs (128 × 1G = 128 GOPS). While this is a simple approximation, it allows us to compare different hardware architectures regardless of their underlying target platforms.

As shown in Table V, our butterfly accelerator achieves 25.6× speedup over the FPGA-based FTRANS [19] while using nearly 10× fewer DSPs. At the same time, we achieve 62.3× higher energy efficiency than FTRANS. Compared with ASIC designs, our accelerator achieves 14.2 × 23.2× speedup under the same computational capacity. Although our FPGA-based butterfly design consumes more power than ASIC designs, it yields 1.1 × 4.3× higher energy efficiency than the other SOTA ASIC accelerators. We expect further speedup and energy efficiency improvements when our design is implemented as an ASIC.

We attribute the performance gain of our approach over ASIC designs to two main factors: 1) the use of FFT and butterfly factorization which significantly reduces the computational complexity at the algorithmic level; 2) the adaptable butterfly design that adopts a single unified hardware engine to accelerate both FFT and butterfly linear transformation, which significantly improves the hardware efficiency; and 3) the co-design process which jointly optimizes both algorithm and hardware parameters.

G. Off-Chip Memory Bandwidth Analysis

In order to investigate the sensitivity of our design to off-chip memory bandwidth, we vary the bandwidth from 6, 12, 25, 50, 100 and 200 GB/s, and evaluate its latency based on our performance model. For these experiments, we use five different designs with 16, 32, 64 and 128 BEs executing FABNet-Large with 24 layers. To understand the bandwidth requirements under both short and long input lengths, we evaluate each design using three input sequences (128, 1024,
and 4096). The results are shown in Figure 21. For a small-scale design of 16 BEs, a bandwidth of 50 GB/s is enough for the design to reach its peak performance under different input sequences. For the largest design of 128 BEs, the achieved performance saturates once the bandwidth reaches 100 GB/s.

H. Power and Resource Analysis

Table VI shows the power consumption breakdown based on the report generated from the Vivado XPE tool. We implement two designs with 120 BEs (BE-120) and 40 BEs (BE-40) on a VCU128 FPGA, which have been used in Section VI-E and Section VI-F respectively. In both designs, the dynamic power accounts for more than 70% of the total power consumption. The memory resources, including both BRAM and HBM, consume more than 25% of the dynamic power. Furthermore, when the number of BEs scales from 40 to 120, the power of clocking, logic & signal and DSPs increases from 2.688 W, 2.381 W and 0.338 W to 6.882 W, 7.732 W and 1.437 W, respectively.

Table VII presents the resource consumption of both BE-40 and BE-120 designs on the same VCU128 FPGA. Due to the use of FFT and butterfly matrices, our FABNet becomes less memory-intensive than the vanilla attention-based NNs. Since the theoretical memory bandwidth of a single HBM (450 GB/s) can already satisfy the requirement of our accelerator (Section VI-G), we use one HBM in both designs to reduce the resource and power consumption. When the number of BEs decreases from 120 to 40, the BRAM usage is reduced from 978 to 338. This reduction can also be observed on the LUT and register resources.

Table VI

| Design | Dynamic (W) | Static (W) |
|--------|-------------|------------|
|        | Clocking | Logic & Signal | DSP | Memory (BRAM & HBM) |        |
| BE-40  | used  | 2.668 | 2.381 | 0.338 | 5.325 | 3.368 |
|        | pct.  | 18.8% | 16.7% | 2.9%  | 37.5% | 23.7% |
| BE-120 | used  | 6.882 | 7.732 | 1.437 | 6.142 | 3.665 |
|        | pct.  | 26.4% | 29.7% | 5.5%  | 23.6% | 14.1% |

Table VII

| Resource Usage of Our Designs on VCU128. |
|----------------------------------------|
| Design | LUTs | Registers | DSP48s | BRAMs | HBM |
| BE-40  | 1,303,680 | 2,607,360 | 9,024 | 2,016 | 2 |
|        | 358,609 | 536,810 | 640 | 338 | 1 |
| BE-120 | 1,034,610 | 1,648,695 | 2,880 | 978 | 1 |
|        | 978 | 63.2% | 31.9% | 48.5% | 50.0% |

1Power of I/O is not included as it occupies less than 1% of the total power.

VII. RELATED WORK

Efficient Approaches for Attention. As the algorithmic complexity of the self-attention mechanism scales quadratically with respect to the input sequence length, many sparse variants have been introduced to approximate the attention-based NNs [45]. The sparsity patterns in these approaches are determined either dynamically [7], [8], [10], [12] or statically [11], [31], [34]. Although these methods achieve high compression rate on the number of operations and parameters, the hardware cost and efficiency of their mappings on real hardware designs are not considered in these works.

Domain-Specific Accelerators for Attention-based NNs. To better utilize existing efficient attention-based algorithmic approaches on hardware, various domain-specific hardware designs have been introduced. Ham et al. [14] propose a hardware architecture called A³, which dynamically prunes entries based on their softmax importance. By leveraging the sparsity in both head and token levels, Wang et al. [6] propose SpAtten that dynamically prunes entire rows and columns from the attention matrix. EdgeBERT [20] explores the layer sparsity via an entropy-based early-exit approach, which significantly reduces the computation and memory footprint. To detect the weak relationship in attention, DOTA [18] uses low-rank linear transformations to detect and omit the weak connections. ELSA [17] approximates the attention mechanism using a sign random projection approach. To further exploit the sparsity in attention-based NNs, Energon [19] adopts a low-precision NN to predict the sparsity in the attention matrix. However, the generated sparsity patterns from these approaches are always unstructured, which may lead to hardware inefficiency. Sanger [15] propose pack-and-split modules to distribute the non-zero computation to each computation engine, achieving a load-balanced execution. Although these accelerators achieve notable speedup over GPUs and CPUs when executing the attention mechanism, their end-to-end hardware performance is limited as the approximation and acceleration of the FFN part are not considered in their design.

Comparison to Previous Work. As spatial-domain convolution corresponds to frequency-domain multiplication, various FFT-based hardware accelerators have been introduced to accelerate CNNs [46], [47], LSTMs [48], [49] and attention-based NNs [19]. However, these designs only use FFT as a domain transfer approach, while the main computation is still performed in another processing engine. In contrast, this paper adopts the butterfly sparsity, a generalized pattern of FFT, for the main computation of the network. Based on the proposed method, all the computations are performed in a single unified butterfly accelerator, resulting in a higher hardware efficiency over previous designs.

Although butterfly sparsity has been explored in recent
literature, most approaches only focus on algorithm-level optimizations. Dao et al. [32] demonstrate the potential of the butterfly matrix in approximating linear transformations, but its efficiency on attention matrices is not explored in their work. On the other hand, Pixelated Butterfly [31] and Sparse Transformer [13] focus on adopting the butterfly pattern for attention matrices, neglecting the linear layers. Moreover, their designs require the use of multiple sparsity patterns to compensate for the accuracy loss, which significantly complicates the hardware design. Lee-Thorp et al. [34] show the effectiveness of Fourier transforms in accelerating attention layers, but did not consider the linear layers, leading to scalability issues (Section II-C). Different from all these efforts, this paper exploits the use of butterfly sparsity for both attention and linear layers via an algorithm-hardware co-design approach. A novel butterfly-based algorithm and an adaptable hardware accelerator are jointly designed to overcome existing limitations to push the performance limit.

VIII. CONCLUSION

This paper proposes the end-to-end acceleration of attention-based NNs via algorithm and hardware co-design. On the algorithmic level, we propose FABNet, a hardware-friendly attention-based NN. Both the attention and linear layers are compressed using a unified butterfly sparsity pattern allowing for scalable end-to-end acceleration. On the hardware level, an adaptable butterfly accelerator is proposed that can be configured at runtime to accelerate different layers based on a unified hardware engine to achieve high hardware efficiency. Both algorithm and hardware design parameters are jointly optimized to push the performance limit. Our experiments demonstrate that our co-design approach yields up to $23.2 \times$ speedup over state-of-the-art accelerators. Furthermore, our design achieves up to $273.8 \times$ higher energy efficiency compared to optimized GPU implementations.

ACKNOWLEDGEMENT

The support of the UK EPSRC grants (UK EPSRC grant number EP/V028251/1, EP/L016796/1, EP/S030069/1 and EP/N031768/1), AMD and Intel is gratefully acknowledged. We also thank Alexander Mathiasen for insightful discussions on model compression.

APPENDIX

A. Abstract

This Appendix summarizes the necessary information and instructions to evaluate our artifacts. The functionality of our hardware accelerator can be evaluated by running Verilog HDL designs and System Verilog testbenches on Vivado design suite. The accuracy results can be obtained by running our PyTorch programs and the associated Bash scripts. The power and resource utilization can be obtained by running Synthesis and Implementation using our RTL code and constraint files. The latency can be obtained by running our custom Python-based performance model. We also provide all our training log files and Vivado design reports in the link: https://drive.google.com/drive/folders/1jaR8gDX-zO1Hu83xFg_JOwRgoBMPnjY?usp=sharing

B. Artifact check-list (meta-information)

- **Algorithm**: FABNet, an efficient model that adopts a unified butterfly sparsity pattern to approximate both the attention mechanism and the FFNs.
- **Program**: Python, PyTorch, Verilog HDL
- **Model**: We evaluate three models for comparison, including Transformer, FNet and FABNet.
- **Data set**: Long-Range-Arena (LRA) dataset, which is a well-known long sequence natural language processing (NLP) dataset. The zip file can be downloaded from the link: https://storage.googleapis.com/long-range-arena/lra_release.gz. The required disk space of the unzip file is around 33 GB.
- **Run-time environment**: Ubuntu 20.04, CUDA SDK 11.3 or higher.
- **Hardware**: Nvidia V100 GPU, Nvidia TITAN Xp GPU, Nvidia Jetson Nano GPU, Intel Xeon Gold 6154 CPU, Raspberry Pi 4.
- **Metrics**: Accuracy, simulated latency, resource and power consumption.
- **Experiments**: Bash scripts and detailed instructions are provided to run experiments.
- **How much disk space required (approximately)?**
- **How much time is needed to prepare workflow (approximately)?** 1 ~ 2 hours.
- **How much time is needed to complete experiments (approximately)?** Accuracy results: hundreds of GPU hours to obtain. Power and resource consumption: around 70 hours. Functionality of Verilog design: around 5 hours.
- **Publicly available?** Yes.
- **Code licenses (if publicly available)?** Yes.
- **Archived (provide DOI)?** We will update this later.

C. Description

1) **How to access**: You can access our codebase from the link: https://zenodo.org/record/7010800#.YwQKCOzMJhF or https://github.com/os-hxfan/Butterfly_Acc.git

2) **Hardware dependencies**: A GPU server is required to run the training of our models. A CPU server is needed to run simulation, synthesis and place&route. Different GPUs and CPUs, such as Nvidia Jetson Nano GPU and Raspberry Pi 4, are also required to evaluate the hardware performance of different models.

3) **Software dependencies**: Vivado Design Suite 2019.2, PyTorch 1.10.2, CUDA SDK 11.3 or higher, Python 3.8 or higher. Other dependencies are listed in requirements.txt.
4) Data sets: Five tasks in the LRA dataset including ListOPs for hierarchical data classification, Text for byte-level text classification, Retrieval for byte-level document retrieval, Image for image classification for sequences of pixels and Pathfinder for classification of long-range spatial dependency.

D. Installation

We provide a detailed installation guide in the README.md of the root directory.

E. Experiment workflow

To evaluate the functionality of our hardware, perform the following steps:

- Follow the instruction of experimental setup in the root directory to install software dependencies. Install Vivado 2019.2.
- Generate the test data using our Python programs.
- Create a Vivado project for our hardware design.
- Import all the Verilog source code and System Verilog testbenches.
- Include all the necessary IPs from Vivado IP library.
- Provide Vivado Tcl scripts and step-by-step instructions in ./hardware/npu_design/verilog/README.md to automate the whole process.

To reproduce the algorithmic and hardware performance, we provide all the scripts under the directory ./script_figs to generate figures and tables. The detailed instructions are provided in ./script_figs/README.md.

F. Evaluation and expected results

We provide scripts under ./script_figs to generate all the figures and tables related to accuracy performance and hardware performance including power consumption, resource utilization and simulated latency performance. As running all the experiments requires a few hundred GPU/CPU hours, to facilitate the artifact evaluation, we refer to the following key results that can be obtained within a reasonable time:

- Vivado simulation to run different layers, such as fast Fourier transform, butterfly matrix multiplication and layer normalization, on our Verilog hardware design. We provide System Verilog testbenches under ./hardware/npu_design/verilog/functionality/testbench/ and a detailed workflow in the first paragraph of Section E.
- Power breakdown in Table \[\text{VI}\] and resource utilization in Table \[\text{VII}\]. We provide detailed instructions and Vivado Tcl scripts under ./hardware/npu_design/verilog/ to run synthesis and place & route on both VCU128 and Zynq 7045 FPGAs.

Although it takes longer to run other experiments, all the results are reproducible using our provided scripts. We provide all the GPU training log files and Vivado design reports in the link: [https://drive.google.com/drive/folders/1jaR8gDX-zO1Hu83xFg_JfOwRgoBMPnjY?usp=sharing](https://drive.google.com/drive/folders/1jaR8gDX-zO1Hu83xFg_JfOwRgoBMPnjY?usp=sharing).

G. Methodology

Submission, reviewing and badging methodology:

- [https://www.acm.org/publications/policies/artifact-review-badging](https://www.acm.org/publications/policies/artifact-review-badging)
- [http://cTuning.org/ae/submission-20201122.html](http://cTuning.org/ae/submission-20201122.html)
- [http://cTuning.org/ae/reviewing-20201122.html](http://cTuning.org/ae/reviewing-20201122.html)

REFERENCES

[1] G. Brauwers and F. Frasincar, “A General Survey on Attention Mechanisms in Deep Learning,” IEEE Transactions on Knowledge and Data Engineering, 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,” Advances in Neural Information Processing Systems (NeurIPS), 2017.

[3] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding,” in ACL, 2019.

[4] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, “Language Models are Unsupervised Multitask Learners,” OpenAI blog, vol. 1, no. 8, p. 9, 2019.

[5] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly et al., “An image is worth 16x16 words: Transformers for image recognition at scale,” in International Conference on Representation Learning (ICLR), 2021.

[6] H. Wang, Z. Zhang, and S. Han, “SpAtten: Efficient Sparse Attention Architecture with Cascade Token and Head Pruning,” IEEE International Symposium on High Performance Computer Architecture (HPCA), 2021.

[7] K. Choromanski, V. Likhosherstov, D. Dohan, X. Song, A. Gane, T. Sarlos, P. Hawkins, J. Davis, A. Mohiuddin, L. Kaiser et al., “Rethinking Attention with Performers,” in International Conference on Learning Representations (ICLR), 2021.

[8] S. Wang, B. Z. Li, M. Khabsa, H. Fang, and H. Ma, “Linformer: Self-Attention with Linear Complexity,” arXiv preprint arXiv:2006.04768, 2020.

[9] N. Kitaev, L. Kaiser, and A. Levskaya, “Reformer: The Efficient Transformer,” in International Conference on Learning Representations (ICLR), 2020.

[10] Y. Tay, D. Bahri, L. Yang, D. Metzler, and D.-C. Juan, “Sparse Sinkhorn Attention,” in International Conference on Machine Learning (ICML), 2020.

[11] I. Beltagy, M. E. Peters, and A. Cohan, “Longformer: The Long-Document Transformer,” arXiv preprint arXiv:2004.05150, 2020.

[12] M. Zaheer, G. Guruganesh, K. A. Dubey, J. Ainslie, C. Alberti, S. Onotanu, P. Pham, A. Ravula, Q. Wang, L. Yang et al., “Big Bird: Transformers for Longer Sequences,” in Advances in Neural Information Processing Systems (NeurIPS), 2020.
[10] R. Child, S. Gray, A. Radford, and I. Sutskever, “Generating Long Sequences with Sparse Transformers,” arXiv preprint arXiv:1904.10509, 2019.

[11] T. J. Ham, S. J. Jung, S. Kim, Y. H. Oh, Y. Park, Y. Song, J.-H. Park, S. Lee, K. Park, J. W. Lee et al., “A²: Accelerating Attention Mechanisms in Neural Networks with Approximation,” in IEEE International Symposium on High Performance Computer Architecture (HPCA), 2020.

[12] L. Lu, Y. Jin, H. Bi, Z. Luo, P. Li, T. Wang, and Y. Liang, “Sanger: A Co-Design Framework for Enabling Sparse Attention Using Reconfigurable Architecture,” in 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 2021.

[13] Z. Zhou, J. Liu, Z. Gu, and G. Sun, “Energon: Towards Efficient Acceleration of Transformers Using Dynamic Sparse Attention,” IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD), 2021.

[14] T. J. Ham, Y. Lee, S. H. Seo, S. Kim, H. Choi, S. J. Jung, and J. W. Lee, “ELSA: Hardware-Software Co-Design for Efficient, Lightweight Self-Attention Mechanism in Neural Networks,” in ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA), 2021.

[15] Z. Qu, L. Liu, F. Tu, Z. Chen, Y. Ding, and Y. Xie, “DOTA: Detect and Omit Weak Attentions for Scalable Transformer Acceleration,” in 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2022.

[16] B. Li, S. Pandey, H. Fang, Y. Lyv, J. Li, J. Chen, M. Xie, L. Wan, H. Liu, and C. Ding, “FTRANS: Energy-Efficient Acceleration of Transformers using FPGA,” in ACM/IEEE International Symposium on Low Power Electronics and Design (ISLPED), 2020, pp. 175–180.

[17] T. Tambe, C. Hooper, L. Pentecost, T. Jia, E.-Y. Yang, M. Donato, V. Sanh, P. Whatmough, A. M. Rush, D. Brooks et al., “EdgeBERT: Sentence-Level Energy Optimizations for Latency-Aware Multi-Task NLP Inference,” in 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 2021.

[18] K. He, X. Zhang, S. Ren, and J. Sun, “Deep Residual Learning for Image Recognition,” in IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2016, pp. 770–778.

[19] J. L. Ba, J. R. Kiros, and G. E. Hinton, “Layer Normalization,” arXiv preprint arXiv:1607.06450, 2016.

[20] G. Lample and A. Conneau, “Cross-lingual language model pretraining,” Advances in Neural Information Processing Systems (NeurIPS), 2019.

[21] A. Wang, A. Singh, J. Michael, F. Hill, O. Levy, and S. R. Bowman, “GLUE: A Multi-Task Benchmark and Analysis Platform for Natural Language Understanding,” Proceedings of the 2018 EMNLP Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP, 2018.

[22] X. Ma, P. Zhang, S. Zhang, N. Duan, Y. Hou, M. Zhou, and D. Song, “A Tensorized Transformer for Language Modeling,” in Advances in Neural Information Processing Systems (NeurIPS), 2019.

[23] Z. Qu, L. Liu, F. Tu, Z. Chen, Y. Ding, and Y. Xie, “DOTA: Detect and Omit Weak Attentions for Scalable Transformer Acceleration,” in 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2022.

[24] D. S. Parker, “Random Butterfly Transformations with Applications in Computational Linear Algebra,” 1995.

[25] T. Dao, A. Gu, M. Eichhorn, A. Rudra, and C. Ré, “Learning Fast Algorithms for Linear Transforms Using Butterfly Factorizations,” in International Conference on Machine Learning (ICML), 2019.

[26] C. De Sa, A. Cu, R. Puttagunta, C. Ré, and A. Rudra, “A two-pronged progress in structured dense matrix vector multiplication,” in Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms. SIAM, 2018, pp. 1060–1079.

[27] B. Chen, T. Dao, E. Winsor, Z. Song, A. Rudra, and C. Ré, “Scatterbrain: Unifying Sparse and Low-rank Attention,” in Advances in Neural Information Processing Systems (NeurIPS), 2021.

[28] T. Dao, N. S. Sohoni, A. Gu, M. Eichhorn, A. Blonder, M. Leszczynski, A. Rudra, and C. Ré, “Kaleidoscope: An Efficient, Learnable Representation for All Structured Linear Maps,” in International Conference on Learning Representations (ICLR), 2020.

[29] J. Lee-Thorp, J. Ainslie, I. Eckstein, and S. Ontanon, “FNet: Mixing Tokens with Fourier Transforms,” arXiv preprint arXiv:2105.03824, 2021.

[30] J. W. Cooley and J. W. Tukey, “An algorithm for the machine calculation of complex fourier series,” Mathematics of Computation, vol. 19, no. 90, pp. 297–301, 1965.

[31] J. W. Cooley and J. W. Tukey, “An algorithm for the machine calculation of complex fourier series,” Mathematics of Computation, vol. 19, no. 90, pp. 297–301, 1965.

[32] X. Ma, P. Zhang, S. Zhang, N. Duan, Y. Hou, M. Zhou, and D. Song, “A Tensorized Transformer for Language Modeling,” in Advances in Neural Information Processing Systems (NeurIPS), 2019.

[33] B. Chen, T. Dao, K. Liang, J. Yang, Z. Song, A. Rudra, and C. Ré, “Pixelated Butterfly: Simple and Efficient Sparse training for Neural Network Models,” in International Conference on Learning Representations (ICLR), 2022.

[34] B. Chen, T. Dao, K. Liang, J. Yang, Z. Song, A. Rudra, and C. Ré, “Pixelated Butterfly: Simple and Efficient Sparse training for Neural Network Models,” in International Conference on Learning Representations (ICLR), 2022.

[35] B. Chen, T. Dao, K. Liang, J. Yang, Z. Song, A. Rudra, and C. Ré, “Pixelated Butterfly: Simple and Efficient Sparse training for Neural Network Models,” in International Conference on Learning Representations (ICLR), 2022.

[36] T. Dao, M. Eichhorn, A. Blonder, M. Leszczynski, A. Rudra, and C. Ré, “Kaleidoscope: An Efficient, Learnable Representation for All Structured Linear Maps,” in International Conference on Learning Representations (ICLR), 2020.

[37] T. Dao, M. Eichhorn, A. Blonder, M. Leszczynski, A. Rudra, and C. Ré, “Kaleidoscope: An Efficient, Learnable Representation for All Structured Linear Maps,” in International Conference on Learning Representations (ICLR), 2020.
[38] T. Geng, A. Li, R. Shi, C. Wu, T. Wang, Y. Li, P. Haghi, A. Tumeo, S. Che, S. Reinhardt et al., “AWB-GCN: A Graph Convolutional Network Accelerator with Runtime Workload Rebalancing,” in 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 2020.

[39] Y. Tay, M. Dehghani, S. Abnar, Y. Shen, D. Bahri, P. Pham, J. Rao, L. Yang, S. Ruder, and D. Metzler, “Long Range Arena: A Benchmark for Efficient Transformers,” in International Conference on Learning Representations (ICLR), 2020.

[40] A. Paszke et al., “PyTorch: An Imperative Style, High-Performance Deep Learning Library,” in Advances in Neural Information Processing Systems (NeurIPS), 2019.

[41] T. Wolf et al., “Huggingface’s Transformers: State-of-the-art Natural Language Processing,” arXiv preprint arXiv:1910.03771, 2019.

[42] Y. Xiong, Z. Zeng, R. Chakraborty, M. Tan, G. Fung, Y. Li, and V. Singh, “Nyströmformer: A Nyström-based Algorithm for Approximating Self-Attention,” in Proceedings of the AAAI Conference on Artificial Intelligence (AAAI), vol. 35, no. 16. NIH Public Access, 2021, p. 14138.

[43] L. Song, J. Mao, Y. Zhuo, X. Qian, H. Li, and Y. Chen, “HyPar: Towards Hybrid Parallelism for Deep Learning Accelerator Array,” in 2019 IEEE International Symposium on High Performance Computer Architecture (HPCA), 2019.

[44] M. Alwani, H. Chen, M. Ferdman, and P. Milder, “Fused-Layer CNN Accelerators,” in 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 2016.

[45] Y. Tay, M. Dehghani, D. Bahri, and D. Metzler, “Efficient Transformers: A Survey,” ACM Computing Surveys (CSUR), 2020.

[46] C. Zhang and V. Prasanna, “Frequency Domain Acceleration of Convolutional Neural Networks on CPU-FPGA Shared Memory System,” in Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA’17), 2017, pp. 35–44.

[47] T. Abtahi, C. Shea, A. Kulkarni, and T. Mohsenin, “Accelerating Convolutional Neural Network With FFT on Embedded Hardware,” IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 26, no. 9, pp. 1737–1749, 2018.

[48] S. Wang, Z. Li, C. Ding, B. Yuan, Q. Qiu, Y. Wang, and Y. Liang, “C-LSTM: Enabling Efficient LSTM using Structured Compression Techniques on FPGAs,” in Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA’18), 2018, pp. 11–20.

[49] Z. Li, C. Ding, S. Wang, W. Wen, Y. Zhuo, C. Liu, Q. Qiu, W. Xu, X. Lin, X. Qian et al., “E-RNN: Design Optimization for Efficient Recurrent Neural Networks in FPGAs,” in 2019 IEEE International Symposium on High Performance Computer Architecture (HPCA). IEEE, 2019, pp. 69–80.