Sanger: A Co-Design Framework for Enabling Sparse Attention using Reconfigurable Architecture

Liqiang Lu*
School of Electronics Engineering and Computer Science, Peking University, Beijing, China
liqianglu@pku.edu.cn

Yicheng Jin*
School of Electronics Engineering and Computer Science, Peking University, Beijing, China
yicheng.jin@pku.edu.cn

Hangrui Bi
School of Electronics Engineering and Computer Science, Peking University, Beijing, China
henryb@pku.edu.cn

Zizhang Luo
School of Electronics Engineering and Computer Science, Peking University, Beijing, China
semiwaker@pku.edu.cn

Peng Li
Institute of Software, Chinese Academy of Sciences, Beijing, China
lipeng@iscas.ac.cn

Tao Wang
School of Electronics Engineering and Computer Science, Peking University, Beijing, China
wangtao@pku.edu.cn

Yun Liang†
School of Electronics Engineering and Computer Science, Peking University, Beijing, China
ericlyun@pku.edu.cn

ABSTRACT

In recent years, attention-based models have achieved impressive performance in natural language processing and computer vision applications by effectively capturing contextual knowledge from the entire sequence. However, the attention mechanism inherently contains a large number of redundant connections, imposing a heavy computational burden on model deployment. To this end, sparse attention has emerged as an attractive approach to reduce the computation and memory footprint, which involves the sampled dense-dense matrix multiplication (SDDMM) and sparse-dense matrix multiplication (SpMM) at the same time, thus requiring the hardware to eliminate zero-valued operations effectively. Existing techniques based on irregular sparse patterns or regular but coarse-grained patterns lead to low hardware efficiency or less computation saving.

This paper proposes Sanger, a framework that harvests sparsity in the attention mechanism through synergistic hardware and software co-design. The software part prunes the attention matrix into a dynamic structured pattern, and the hardware part features a reconfigurable architecture that exploits such patterns. Specifically, we dynamically sparsify vanilla attention based on a quantized prediction of the attention matrix. Then, the sparse mask is rearranged into structured blocks that are more amenable to hardware implementation. The hardware design of Sanger features a score-stationary dataflow that keeps sparse scores stationary in the PE to avoid decoding overhead. Using this dataflow and a reconfigurable systolic array design, we can unify the computation of SDDMM and SpMM operations. Typically, the PEs can be configured during runtime to support different data access and partial sum accumulation schemes. Experiments on BERT show that Sanger can prune the model to 0.08 - 0.27 sparsity without accuracy loss, achieving 4.64X, 22.7X, 2.39X, and 1.47X speedup compared to V100 GPU, AMD Ryzen Threadripper 3970X CPU, as well as the state-of-the-art attention accelerators A^3 and SpAtten.

CCS CONCEPTS

• **Computer systems organization** → *Systolic arrays*; • **Computing methodologies** → **Natural language processing**.

KEYWORDS

Transformer, attention, sparse, reconfigurable architecture, systolic array, hardware-software co-design

ACM Reference Format:
Liqiang Lu, Yicheng Jin, Hangrui Bi, Zizhang Luo, Peng Li, Tao Wang, and Yun Liang. 2021. Sanger: A Co-Design Framework for Enabling Sparse Attention using Reconfigurable Architecture. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '21), October 18–22, 2021, Virtual Event, Greece.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3466752.3480125

1 INTRODUCTION

Attention-based neural networks have significantly advanced the progress of machine learning in recent years, setting new state-of-the-art in domains including natural language processing and

*These authors contributed equally.
†Corresponding author.

MICRO '21, October 18–22, 2021, Virtual Event, Greece

ACM ISBN 978-1-4503-8557-2/21/10…$15.00
https://doi.org/10.1145/3466752.3480125

computer vision. Specifically, Transformer [55] uses only attention mechanisms as its building blocks instead of recurrence or convolutions in the traditional models. With such revolutionary model design, Transformer and its variants [16, 46] dominate a wide range of NLP tasks including machine translation [55], text classification [16], text generation [46], etc. In the field of computer vision, attention is widely used, together with convolution, to allow a global receptive field, leading to improved performance on various computer vision tasks such as image captioning [63], image generation [66], image segmentation [59], etc.

Attention takes three matrices as its inputs, namely the query (Q), the key (K), and the value (V). The first step of computing attention is to obtain a score matrix (S) by multiplying Q and K. The score matrix is often referred to as the attention matrix. Then, the score matrix goes through the softmax function for normalization. The second step is to multiply the normalized score matrix with the value matrix (V) to generate the final output. Unlike convolution and RNN that aggregates information locally, vanilla self-attention computes attention for every pair of queries and keys. The benefit of a large accessible context comes at the cost of an overwhelming computational burden, which increases quadratically with the sequence length. For example, for tasks involving images or long texts, the input sequence length could go as large as 16K [53]. For a single input containing 16K tokens, the computation of one self-attention module of BERT-Base reaches a formidable amount of 861.9 GFLOPs.

In the face of such overwhelming computation pressure when scaling attention to long sequences, sparse attention emerges as a promising solution. For sparse attention, the involved sparsity turns the first step into a general sampled dense–dense matrix multiplication (SDDMM) computation, where the queries and keys are dense and the score matrix is sparse according to a sample mask. In the second step we generate the output by multiplying the sparse score matrix with a dense value matrix, which is also a sparse computation called sparse-dense matrix multiplication (SpMM). Such sparsity leads to irregular data access of keys and values, making the computation hard to parallelize. Besides, the distribution of nonzeros determines the workload of PEs, which will cause a load imbalance problem when the nonzeros are not distributed evenly.

We divide existing sparse attention mechanisms into two categories, namely static sparsity and dynamic sparsity. Static sparsity [2, 11, 18, 31, 65] is pre-determined and independent of the content of queries and keys. It is usually designed to be structured to facilitate parallelization on GPUs. For example, [11, 65] employ block sparsity. However, by imposing restrictions on the distribution of non-zeros, these patterns ignore the inherent relationship between different words, resulting in limited computation saving at the same level of accuracy. In contrast, dynamic sparsity [12, 13, 19, 25, 48, 52, 57, 68] achieves a better trade-off between accuracy and computation savings because its patterns are dynamically generated from the input queries and keys, thus harvesting the redundancy in attention more effectively and achieving higher sparsity. For example, [12, 13] perform sparsification after normalizing the full attention matrix with the softmax function, which can learn arbitrary patterns and fully utilize the intrinsic structure of input data. But unfortunately, these patterns can only be determined at runtime and are usually highly unstructured. Such irregularity can lead to workload imbalance and poor data locality, resulting in degraded hardware performance.

This paper proposes Sanger, a software-hardware co-design framework that accelerates the sparse attention models by combining dynamic sparsity patterns and reconfigurable architecture. The software part provides sparsity patterns, which can achieve high performance and a balanced workload. The architecture is designed with reconfigurability to support the dynamic characteristics of sparsity, which helps to improve the compression ratio. More clearly, we design an algorithm to predict the dynamic sparsity by computing a low-bit version of the attention matrix and zeroing out small attention weights via binary thresholding. To ensure load balance for efficient hardware implementation, we encode the resulting attention mask with unstructured sparsity into multiple fine-grained structured blocks. The prediction and encoding are performed on the fly as we determine the sparsity patterns dynamically on a per-sample basis.

At the hardware level, identifying that the sparse score matrix is both the output of the first step and the input of the second step, we propose a score-stationary dataflow that seamlessly unifies the computation of SDDMM and SpMM operations. Using this dataflow, we keep the sparse scores stationary in PEs until the computation is finished, which effectively avoids the decoding overhead. To allow more flexibility in the sparsity patterns, we design a reconfigurable systolic array based on this dataflow. Since the queries, keys, and values are stored as dense matrices, we decouple the input data registers from the PE arrays, where the connection between registers and PEs is dynamically configured by the attention mask.

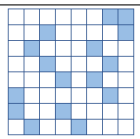
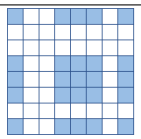
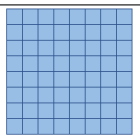
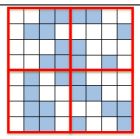
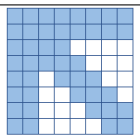
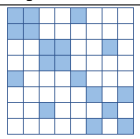
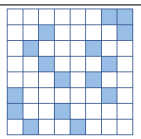
The contribution of this paper can be summarized as follows.

- We propose Sanger, a hardware and software co-design framework that exploits the dynamic sparsity in attention via a reconfigurable architecture. The hardware flexibility enables the software to achieve high sparsity.
- We propose a dynamic and fine-grained structured pruning technique with high flexibility and sparsity. Our algorithm uses low-bit computation to predict sparse attention masks and encodes them through packing and splitting to keep the workload balanced.
- We propose a score-stationary dataflow and a reconfigurable architecture. The dataflow effectively eliminates sparsity decoding overhead and memory transfer overhead by unifying the SDDMM and SpMM operations. The architecture is reconfigurable to support a wide range of sparsity patterns.

Experiments show that Sanger can prune BERT to 0.08 - 0.27 sparsity[1] without accuracy loss. For GPT-2, BART, we can prune it to 0.15 - 0.35, 0.23 - 0.54 sparsity within 0.5% accuracy loss, respectively. For BERT, GPT-2, and BART, Sanger achieves 4.71X, 6.45X, 3.76X speedup over V100 GPU-FP32, 4.64X, 6.88X, 3.94X speedup over V100 GPU-FP16, and 22.7X, 13.3X, 13.2X speedup over AMD Ryzen Threadripper 3970X CPU. For energy-efficiency, the average improvement of Sanger is 48X, 35X, 113X compared to GPU-FP32, GPU-FP16, and Ryzen CPU. Compared to the state-of-the-art attention accelerators A^3 and SpAtten, Sanger shows 2.39X and 1.47X speedup.

[1] In this paper, we term sparsity as the proportion of non-zeros.

Table 1: Existing sparse attention patterns.

Features	Software-hardware co-design				Software only		
	A^3 [19]	SpAtten [57]	FTRANS [31]	Sanger	[2, 11, 65]	[25, 48, 52]	[12, 13, 68]
Sparsity pattern	dynamic	dynamic	static	dynamic	static	data-dependent	dynamic
Pattern visualization							
Pattern regularity	unstructured	coarse-grained structured	-	fine-grained structured	coarse-grained structured	coarse-grained structured	unstructured
Sparsity	medium	low	-	high	low	medium	high
Hardware speedup	low	medium	low	high	low	low	low

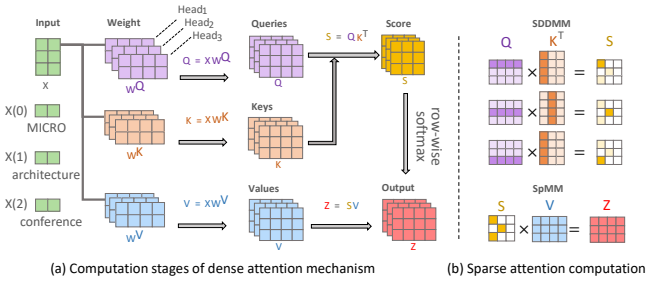


Figure 1: Computation of multi-head attention.

2 BACKGROUND AND MOTIVATION

2.1 Primer on Attention

The attention mechanism is the key operation in the Transformer models, which can be described as mapping a query vector and a set of key-value vector pairs to an output vector. As the input sequence contains multiple tokens, their corresponding query vectors, key vectors, and value vectors form the query matrix, key matrix, and value matrix. Figure 1 (a) depicts the computation stages of the self-attention (abbreviated as attention) mechanism. At first, we obtain the query (Q), key (K), value (V) matrices by multiplying the embedded input sequence and the corresponding weight matrices. Next, the score matrix is calculated by multiplying the query matrix and key matrix, which represents the importance of each input token when producing an output element. We then normalize the score matrix with a row-wise softmax function. Finally, the score matrix is multiplied with the value matrix to generate the output matrix.

The attention mechanism is essentially a content-based similarity search. In general, most tokens in the sequence are irrelevant to the current query, which makes the attention mechanism inherently sparse. Unlike ReLU or dropout that explicitly zeroes out elements, operators like softmax and GELU assign near-zero values to a fair portion of elements, which can be pruned through rounding or thresholding. We refer to such sparsification opportunities as implicit sparsity. In particular, the sparse score matrix is the output of $Q \times K$ and the input of $S \times V$, which turns the attention operation into SDDMM and SpMM. As shown in Figure 1 (b), the computation of SDDMM takes dense matrices (Q, K) as inputs and calculates the outputs based on a sample mask. Here, the sample mask refers to a binary matrix that indicates which elements need to be calculated in the output matrix. Besides, we define the SpMM operation as the matrix multiplication between the sparse score matrix and the dense value matrix.

2.2 Existing Sparse Attention Designs

Table 1 lists the recent studies on exploiting the sparsity in attention. Most of them are algorithm-level improvements without hardware optimization, which we classify as software-only methods. Here we further divide these works into three sub-categories. The *dynamic sparsity* pattern [12, 13, 68] can achieve high sparsity conditioned on individual input samples during runtime, but require computing the full attention matrix at first. The *static pattern* [2, 11, 65] is predetermined and independent of data. These patterns usually adopt coarse-grained structures for efficient training on common accelerators such as dropping entire blocks or rows, but achieve limited sparsity in the comparison with others. The *data-dependent pattern* [25, 48, 52] can be seen as a trade-off between dynamic sparsity and static sparsity, which tries to derive constrained sparse patterns from input with an additional clustering or sorting step.

As shown in Table 1, a few attention accelerators have been proposed in recent years that apply co-designed sparsity patterns [19, 31, 57]. FTRANS [31] compresses weights of Transformer-based models with a block circulant matrix-based representation. It does not involve any attention-specific sparsity and still computes the full attention matrix. As FTRANS requires computation in the frequency domain that involves complex values, the actual hardware speedup is low. A^3 [19] approximates attention by greedily searching for key vectors that are likely to be relevant to the current query. However, this approach provides limited computation saving because it gives a rather rough prediction which harms accuracy at high sparsity. SpAtten [57] performs structured activation pruning by removing entire attention heads and tokens. However, the remaining heads and tokens may still contain redundant attention, leading to low sparsity.

In summary, all these works achieve limited speedup because of low sparsity or hardware inefficiency. The unstructured dynamic

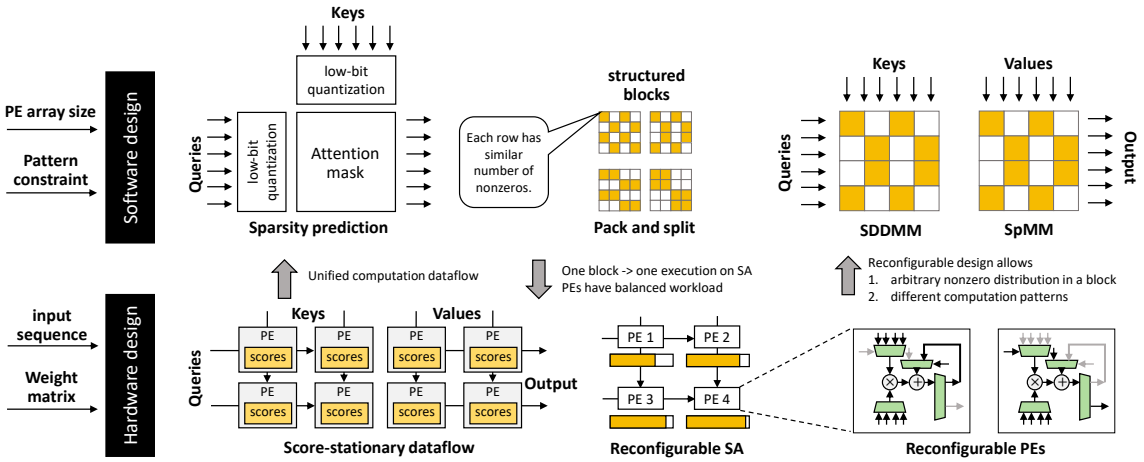


Figure 2: Sanger framework overview.

patterns of software-only approaches exhibit higher sparsity compared to co-design approaches, but their irregularity makes it hard to achieve adequate acceleration. On the other hand, existing hardware accelerators adopt fixed architecture design, which strictly limits the sparsity patterns they support and leads to low hardware efficiency. For example, SpAtten [57] only supports coarse-grained token-level pruning because its computation engine is optimized for dense vector-matrix multiplication. In this paper, we propose a fine-grained pruning approach that poses minimal limitations on sparse patterns and a matrix encoding strategy that performs load balancing dynamically. We also propose a reconfigurable architecture design to flexibly support such sparsity.

3 SANGER OVERVIEW

Figure 2 presents the overview of our proposed framework Sanger, which consists of an attention pruning algorithm and a reconfigurable attention accelerator. At the software level, we use quantized queries and keys to calculate a sketchy prediction of the attention matrix, which is then pruned via binary thresholding. The resulting attention mask exhibits an unstructured sparsity pattern as it is dynamically determined by the input queries and keys. To regularize the computation and improve hardware efficiency, we propose an encoding scheme to pack and split the attention mask into multiple structured blocks with balanced workloads. Finally, the dense attention mechanism is transformed to SDDMM and SpMM operations with the sparse blocks serving as their sample masks. Section 4 presents the details.

At the hardware level, Sanger features a unified score-stationary dataflow that supports both SDDMM and SpMM operations on one systolic array. Based on this dataflow, we design a reconfigurable systolic array that processes one block at a time. The input entry of PEs can be dynamically configured to receive queries, keys, values with different indices. Such reconfigurability allows flexible nonzero distribution in a block, which helps to improve the compression ratio during pruning. Section 5 and 6 present the details.

In the end-to-end implementation, we first train the model weights under sparsity pattern constraints, including PE array size, sparse block size, and the maximum number of nonzeros in each row. The weights are dense and fine-tuned to make it easier to form our sparsity pattern. In the inference phase, Sanger accelerator takes weight matrices from training and token sequence as its input to calculate the queries, keys and values. Sanger generates sparse masks dynamically with a hardware pipeline which implements all the steps in the pruning algorithm, including quantization, thresholding, and encoding. Then, the sparse mask is executed efficiently on the reconfigurable systolic array.

Algorithm 1: Sparse Attention Algorithm Overview

Input: Q, K, V are queries, keys and values
T is the threshold for binarization
Output: Z is the output of the attention mechanism

1. Quantized prediction of the attention matrix
$\hat{Q} = 4bit - quant(Q)$
$\hat{K} = 4bit - quant(K)$
$\hat{S} = softmax\left(GEMM\left(\hat{Q}, \hat{K}\right)\right)$
2. Generate attention mask with binary thresholding
$M = binary - thres\left(\hat{S}, T\right)$
3. Packing and splitting mask into structured blocks
$\hat{M} = pack - split(M)$
4. Compute sparse attention
$S = softmax\left(SDDMM\left(Q, K, \hat{M}\right)\right)$
$Z = SpMM(S, V)$

4 SOFTWARE PRUNING FOR SPARSE ATTENTION

To exploit the dynamic sparsity in the attention mechanism, we face a few challenges. First, dynamic sparsity cannot be pre-determined before inference, and its pattern varies drastically among samples. Second, different from the ReLU sparsity which explicitly zeroes out activations, the softmax operation induces implicit sparsity, where all positions are assigned nonzero values. Third, dynamic sparsity usually exhibits highly unstructured patterns, posing another challenge for hardware implementation. To address these issues, we devise an algorithm to predict the sparse pattern for each input sample via quantized approximation, extract the implicit sparsity

via binary thresholding, and re-arrange the unstructured pattern into a hardware-friendly format via packing and splitting.

Algorithm 1 presents a step-by-step overview of our approach. Given queries Q, keys K, and values V, we first quantize the Q and K matrices to 4-bit and compute a low-bit prediction $\hat{S}$ of the attention matrix via dense general-purpose matrix multiplication (GEMM) and softmax. In the second step, we extract implicit sparsity from the dense attention matrix $\hat{S}$ by zeroing out small attention weights according to a threshold T. The output of this step is a binarized attention mask M with unstructured sparsity. Next, through packing and splitting, we transform M into small blocks with balanced structures for the sake of hardware efficiency, with each of them corresponding to one execution on the systolic array. Finally, the inference of attention is transformed to SDDMM and SpMM operations.

4.1 Predicting Attention Matrix

Unlike weights and input activations, the attention matrix is the intermediate results of the attention mechanism that cannot be determined before execution. We propose to derive dynamic sparsity from a quantized prediction of the attention matrix, thereby avoiding computing the full version. More clearly, we calculate the quantized prediction of the attention matrix S as follows

$$\hat{S} = \text{Softmax}\left(Q^{-1}\left(Q(Q)Q(K)^{\top}\right)/\sqrt{d}\right)$$

where $Q(\cdot)$ is a quantization operator that maps input buffers to low-bit while $Q^{-1}(\cdot)$ is the corresponding de-quant operator that transforms low-bit outputs back into high precision. We choose symmetric linear quantization as our quantization operator Q to strike a balance between accuracy and efficiency: it provides a higher precision than directly fetching the MSBs of fixed-point buffers and is much easier to implement than k-means quantization. With its computation shown below, symmetric linear quantization can be implemented as a multiplication by a scaling factor γ and a rounding bit-shift. We implement the de-quant operator Q^{-1} in the same way.

$$Q(x) = \text{round}(\gamma x)$$

Following the quantization-aware training scheme, we simulate quantization during training to accommodate quantization error and collect range statistics of Q, K from training data. During training, we approximate gradients of the non-differentiable operator Q using Straight-Through Estimator [3]. In our experiments, we quantize all Q and K matrices to 4-bit, so the computational overhead is around 1/16 of the original 16-bit dense attention.

4.2 Generating Attention Mask

As shown in Figure 3, after obtaining a quantized approximation $\hat{S}$ of the attention matrix, we generate a binary attention mask M according to the sparsity pattern it exhibits. While most of the existing works on leveraging output sparsity focus on ReLU-induced sparsity [5, 50], we deal with the implicit sparsity induced by softmax in the case of attention mechanisms. Typically, vanilla softmax assigns near-zero values to a large part of attention scores because it is a smoothened version of the argmax operator. However, unlike ReLU that explicitly zeroes out activations, softmax never produces

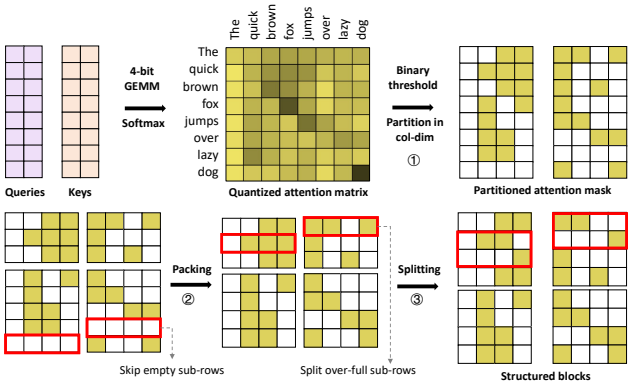


Figure 3: Encode the attention mask into fine-grained structured blocks.

exact zeros. This feature of vanilla softmax makes its output sparsity less easy to utilize directly and harms the prediction accuracy of models based on soft attention.

To extract the implicit sparsity, we apply a binary threshold to the predicted attention matrix. We empirically observe that this simple technique achieves a high compression ratio and overall accuracy. As shown below, the binarization operator is a shifted Heaviside step function. In our experiments, the threshold $T \in [0, 1]$ is a global hyper-parameter applied to all attention heads and can be tuned to trade-off between sparsity and accuracy.

$$M_{ij} = \text{binarize}\left(\hat{S}_{ij}, T\right) = \begin{cases} 1, & \text{if } \hat{S}_{ij} \geq T \\ 0, & \text{otherwise} \end{cases}$$

4.3 Packing and Splitting Attention Mask

Given the attention mask M from binary thresholding, the original GEMM of $Q \times K^{\top}$ becomes a sparse SDDMM operation, which means we can skip computing the output elements that M masks out. Unfortunately, the sparsity in SDDMM could not directly translate into actual speedup because the unstructured pattern of M makes it not amenable to efficient hardware implementation. In contrast to weight pruning [26, 70], it is non-trivial to enforce fine-grained structural constraints on activations. Besides, the overhead of packing/encoding sparse activation matrices occurs at every input sample rather than once before deployment.

Facing these challenges, we refrain from directly regularizing activation patterns and design an efficient encoding strategy that re-arranges the unstructured attention mask into a load-balanced format. As shown in Figure 3, our encoding strategy consists of three steps: partitioning, packing and splitting. Given an attention mask, we first partition it into sub-matrices along the column dimension, with each having the same width as the number of vertical input ports of the PE array. Then we pack each sub-matrix by skipping the sub-rows that contain no nonzero elements. For example, we skip the empty sub-rows as highlighted in the bottom left. Next, to make sure the amount of nonzeros in each sub-row does not exceed the PE number in each row of the PE array, we split the overfull sub-row into multiple rows. For example, we split the second sub-row in the left-top block into two sub-rows. Finally, the attention mask is transformed into multiple structured blocks, with each block having a similar number of nonzeros in each row.

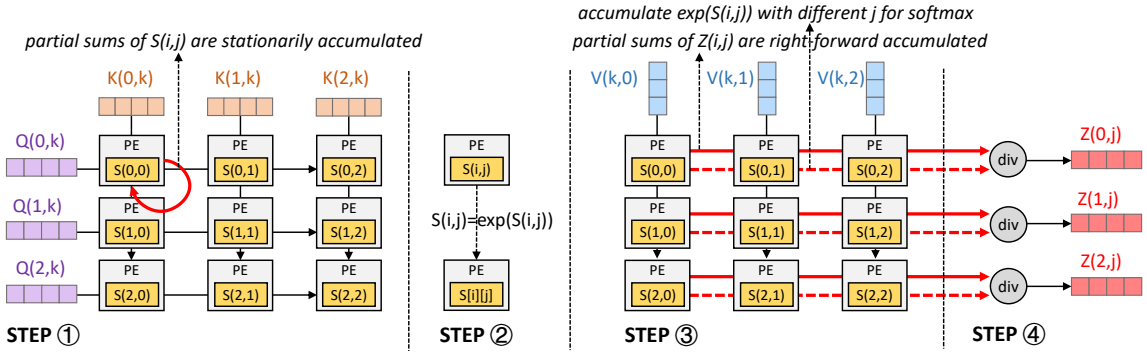


Figure 4: Dense score-stationary dataflow.

5 HARDWARE DATAFLOW

Our software pruning technique only sets the constraints of the proportion of nonzeros in each row. However, the actual nonzero distribution can be arbitrary, leading to irregular data access of SDDMM and SpMM operations. Besides, the sparsity occurs in the score matrix, which is both the output of SDDMM operation and the input of SpMM operation, making it challenging to decode the sparsity to cooperate with dense queries, keys, and values. Our hardware design uses a unified dataflow, which takes queries, keys, values, and a mask that indicates the sparsity pattern as inputs. This dataflow unifies the SDDMM and SpMM operations by keeping sparse scores stationary in the PE until the computation is finished, namely score-stationary dataflow. This design can effectively eliminate the decoding overhead by keeping the score in place. In the next, we first present the dense design of this dataflow in Section 5.1 and then the sparse score-stationary dataflow in Section 5.2.

5.1 Dense Score Stationary Dataflow

As shown in Figure 4, the complete dataflow is divided into four steps, where the first step and the third step are responsible for the matrix multiplication of $Q \times K$ and $S \times V$, respectively. For simplicity, we assume that each PE has the functionality of one multiply-and-accumulate (MAC) operation. In **STEP 1**, the query matrix and key matrix are multiplied to generate a score matrix using a systolic array. In this step, each row of the query matrix is horizontally mapped to one PE row, while the row of the key matrix is vertically sent to the PEs of the systolic array. The computation of the first step can be regarded as a vector outer-product between one row from the query matrix and one row from the key matrix. Each PE calculates the partial sum of the same score element and iteratively accumulates it until the calculation is finished (iterative accumulation). In **STEP 2**, each PE conducts an exponential operator $e^{S(i,j)}$ to prepare for the softmax operation. In **STEP 3**, the systolic array is reused to multiply the score matrix with the value matrix. As the scores have been stored in the PE, only the column of the value matrix vertically traverses across the PE array. Different from STEP 1, each row of the PE array here is responsible for a vector dot-product between one row from the score matrix and one column from the value matrix. Scores are kept stationary in the PEs in STEP 3, while the results of multiplying scores and values are accumulated to the right (forward accumulation). For example, the result of S(0,0)*V(0,0) is moved to the right PE to be accumulated with S(0,1)*V(1,0). The final result of dot-product is gathered at the rightmost PE and sent to the division module. Note that we also calculate the sum of $e^{S(i,j)}$ in each row of the score matrix in this step. In **STEP 4**, the results of the PE array are divided by the sum of $e^{S(i,j)}$ for the normalization in softmax function.

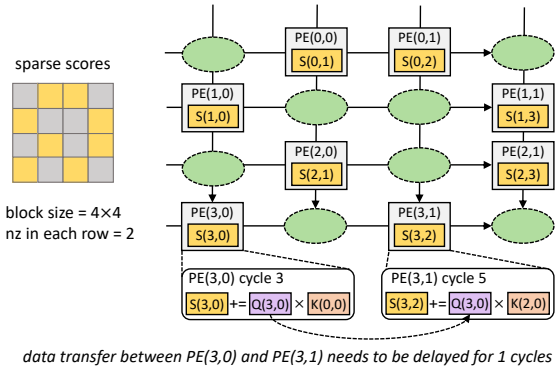
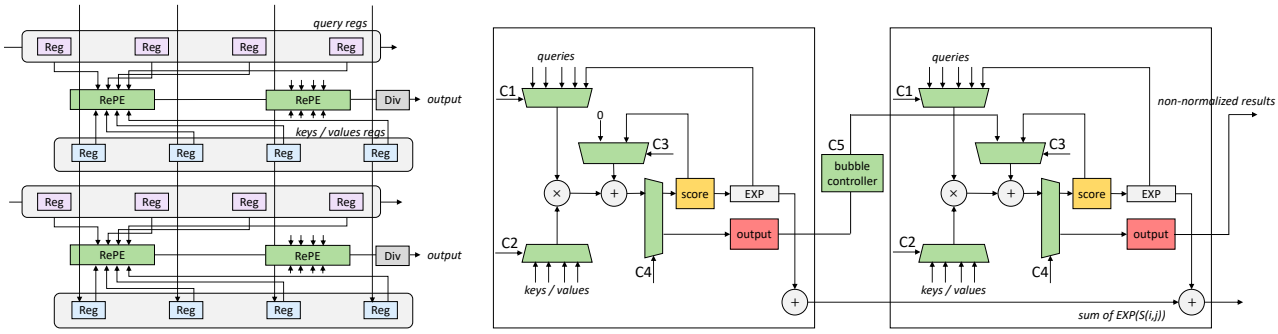


Figure 5: Sparse score-stationary dataflow.

5.2 Sparse Score Stationary Dataflow

The sparsity of attention occurs in the intermediate matrix (scores) while the input matrices are still dense (queries, keys, values). The irregular sparsity makes it hard to leverage the parallelism of the systolic array, as systolic arrays exhibit highly structured data access. To effectively exploit the sparsity, we propose a sparse score stationary dataflow extended from the dense version. The sparse dataflow shares the same data access of queries, keys, values as the dense dataflow and also keeps each score element stationary in the PE.

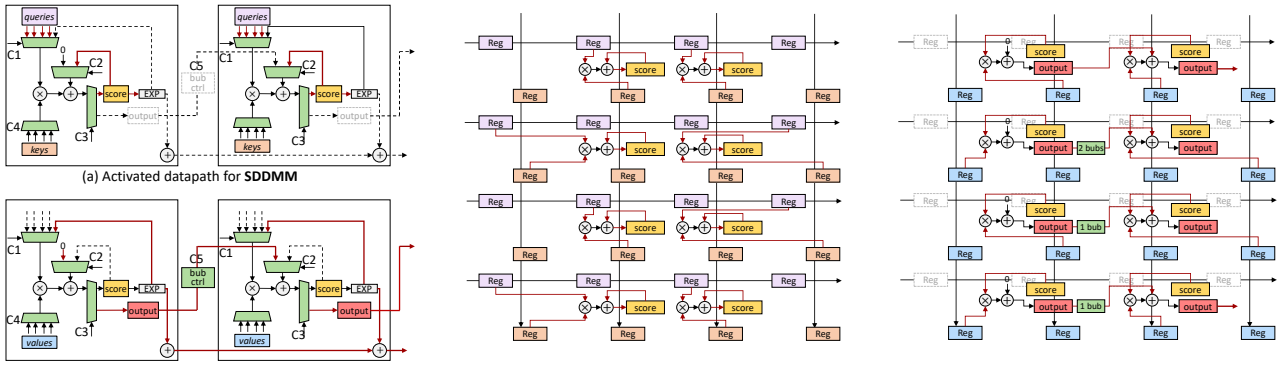
In the dense dataflow, the score coordinates for each PE are derived from the indices in the matrix. However, in the sparse dataflow, the sparse scores in the same row are packed and then sent to one row of PEs. To rectify the data movement, we insert bubbles between PEs, as shown in Figure 5. Each bubble can be regarded as a virtual PE, which only makes the data stall for one cycle. The number of bubbles between PEs follows the distribution of zeros in the sparse block. For example, in a 4×4 sparse block with each row has 2 nonzero scores, the scores are mapped to a 4×2 PE array, as shown in Figure 5. In the access pattern of systolic arrays, the data of Q(3,0) is required to multiply with K(0,0) in PE(3,0) at



(a) Flexible PE array and data access of queries, keys, values

(b) The reconfigurability of RePE design

Figure 6: Reconfigurable systolic array. The C1-C5 are reconfigurable modules during runtime. C1 determines the choice between queries and scores. C2 alternates the data source between keys and values. C3 is responsible for iterative accumulation (for SDDMM) and forward accumulation (for SpMM). C4 decides the result destination of the adder. C5 controls the access delay of output. RePE: reconfigurable PE.



(a) Activated datapath for **SDDMM**

(b) Activated datapath for **SpMM**

(c) An end-to-end datapath configuration for the sparsity pattern in Figure 5

Figure 7: Configuration details and examples of RePE.

cycle 3 and multiply with K(2,0) in PE(3,1) at cycle 5. Therefore, one bubble is inserted to delay the data transfer for 1 cycle.

One of the objectives of score-stationary dataflow is to maximize the reuse opportunities of the score matrix on chip during computation. By doing this, we can avoid the data transfer of scores, which further eliminates the decoding overhead when the score matrix is sparse. The queries, keys, and values are still dense matrices, which can be directly streamed to the PE array using simple FIFO logic. Another advantage of score-stationary dataflow is to unify the computation of SDDMM and SpMM on a single systolic array, which helps to save the total area of the accelerator. Prior approaches use multiple pipeline stages to separately implement each step of attention mechanism on different hardware modules [19, 57].

6 ARCHITECTURE OPTIMIZATION

The score-stationary dataflow requires different data access and accumulation schemes at different stages. Therefore, it is necessary to design a reconfigurable architecture. There are two challenges for a reconfigurable architecture design. *1) To enable the sparsity pattern with uncertain nonzero distribution, the connection between input data (Q, K, V) and PEs needs to be flexible.* For example, in Figure 5, PE (0,0) receives the second column of the key matrix. However, when the pattern changes, it may receive other key columns. *2) SDDMM and SpMM operations differ in the data transfer and accumulation scheme of partial sums.* The streaming data is queries and keys for SDDMM, but partial sum of the output for SpMM. Besides, SDDMM iteratively accumulates partial sums in the same PE, while SpMM requires to forward accumulate results from different PEs in one row.

6.1 Reconfigurable Systolic Array Design

As data (for Q, K, V) are stored in a dense matrix that is inherently different from sparse scores, we decouple the data registers of them from the PE. These registers are connected one by one, which transfer the data in traditional systolic array manner. To support flexible distribution of nonzeros, each PE is connected to all the registers in one row via a multiplexer controller. This multiplexer is configured by the sparsity pattern to determine which query / key / value is sent to the PE, as shown in Figure 6 (a). On the other hand, the number of PEs in each row limits the maximum number of nonzeros in each sparse row. For example, Figure 6 (a) shows a 2×2 PE array with two horizontal ports and four vertical ports, which can handle a 2×4 sparse block with each row has less than 50% nonzeros.

The second challenge is addressed by inserting reconfigurable control modules in the PE. As shown in Figure 6 (b), modules are reconfigurable when the systolic array alternates between SDDMM

and SpMM. Clearly, multiplexer C1 passes queries in SDDMM of $Q \times K$ while it sends the score (after exponential operator) to the multiplier in the SpMM of $S \times V$. Similarly, multiplexer C2 is used to choose the data source between keys and values. C3 is inserted to choose the input data of the adder. C3 selects the partial sum of the score in SDDMM operation to enable iterative accumulation. In SpMM stage, it selects the multiplication results from the previous PE to conduct forward accumulation. C4 decides the result destination of the adder.

As systolic arrays need to follow strict data access rules to ensure correctness, we introduce a bubble controller C5 in Figure 6 (b), whose operation is to set the data transfer delay between adjacent PEs in one PE row. In the third step of Figure 4 (S×V), the bubble controller is activated. Since the values are dense, the transfer of PE results is delayed by a proper number of bubbles to ensure correct data access.

Figure 7 presents a detailed example. Figure 7 (a) and (b) draw the activated datapath when performing SDDMM and SpMM operations. For SDDMM, queries and keys are selected according to the sparsity pattern. Then the multiplication results are gathered in the same PE. After a complete score is computed, we conduct an exponential operator for the softmax function. For SpMM, values are loaded to multiply with scores in each PE, where the results are accumulated across different PEs. Note that the exponential results of scores are also accumulated at the same time for the final normalization. Figure 7 (c) shows an end-to-end datapath configuration for the sparsity in Figure 5. Here, we only picture the activated connection between registers and PEs. In the first row of the sparse block, the first nonzero is located at the second element, making the second query register connected with the first PE. Such nonzero distribution also determines that the first PE is connected with the second register when calculating the output. Besides, the configuration of the bubble controller depends on the number of zeros in between. For example, there are two zeros between nonzero scores in the second row in Figure 5, which sets a 2-cycle delay in the bubble controller when accumulating the output in the second PE row in Figure 7 (c).

6.2 Implementation Details

The first stage of the attention mechanism is to calculate queries, keys, and values using three weight matrices. This stage is a dense matrix multiplication, which is directly mapped on our systolic array design. As mentioned in Section 4.2, the attention mask is generated based on the MSB of queries and keys. We implement this generation in a *pre-process module* that contains a 4-bits multiplier array. There is an encode module to pack and split the attention mask into sparse blocks. As *pack* removes an entire row in the attention block, the corresponding query vector will be skipped in SDDMM operation. *Split* copies the queries in the PE temporal registers before the computation of SDDMM ($Q \times K$). At the end of SpMM ($S \times V$), each two adjacent PE rows are connected with an adder to merge the results from the split query vectors, which only requires very small hardware logic. Modules are fully-pipelined to achieve high throughput. We use a similar implementation of the exponential operator as [58], which firstly converts e^x to $2^{x/\ln 2}$ and uses a look-up table for approximation.

Table 2: Sanger area and power breakdown.

Modules	Parameter	Area (mm^2)	Power (mW)
Pre-process module	64 × 64 4-bits multiplier	1.467	461.9
Pack & split module	64 × 64 1bit mask 4KB mask FIFO	0.056	1.610
Systolic array	64 vertical ports 64 × 16 RePE array 64 EXP modules	8.47	2243
Memory	128KB query buffer 128KB key buffer 128KB value buffer 128KB output buffer	6.912	57.81
Total	UMC 55nm: Area = 16.9 mm^2,Power = 2.76W		

7 EXPERIMENTS

7.1 Experimental Settings

Benchmarks. We evaluate our method on BERT [16], GPT-2 [45], and BART [30]. Our evaluation datasets include eight tasks from the GLUE benchmark [56] (text classification), SQuAD v1.1 [47] (question answering), and CLOTH [62] (long-context cloze which is deliberately introduced to evaluate our method on longer sequences). We exclude the problematic WNLI task from GLUE following [16]. Similar to [47, 56, 62], we evaluate models using Matthew's correlation for CoLA, Pearson/Spearman correlation for STS-B and EM/F1 for SQuAD. For other tasks, we report accuracy or accuracy/F1 on the validation set. All these metrics are positively correlated with model accuracy.

Software pruning implementation. Our code is based on the BERT implementation by NVIDIA [39] and the evaluation code is from Hugging Face's Transformers library [60]. All models are implemented and executed using PyTorch v1.7.1 [42]. We use MKL-DNN [23] and CuDNN [10] as operator libraries for Intel CPUs and NVIDIA GPUs. Since our method does not require pre-training, we directly fine-tune a pre-trained checkpoint on downstream tasks. We use a pruning threshold of 2e-3 for SQuAD and CLOTH, and 2e-2 for the remaining GLUE tasks. For all tasks in GLUE, we use a batch size of 32 and a learning rate of 2e-5. For SQuAD, we use a batch size of 12 and a learning rate of 3e-5. For CLOTH, we use a batch size of 8 and a learning rate of 1e-5. We train all models with enough epochs until convergence.

Hardware implementation. The Sanger accelerator consists of a pre-process unit to generate the attention mask, an encoding module to generate structured blocks, and a reconfigurable systolic array for computation. The design is written in Chisel hardware description language [1]. We use Chisel to generate Verilog RTL. Then we use Synopsys Design Compiler to estimate the chip area and total power under the UMC 55nm technology. The synthesized frequency is 500MHz. Table 2 provides the detailed design parameter and area/power breakdown for Sanger. We use Cadence Innovus 16.13 for placement and routing. Each PE row supports up to 0.25 (16/64) sparsity in rows of a 64 × 64 sparse block. To evaluate the performance of Sanger, we developed a cycle-accurate performance model with the assumption of a 128 GB/s HBM bandwidth.

Platforms for comparison. We compare our framework with modern hardware accelerators, including server GPU (NVIDIA Tesla V100 PCIe 32GB), server CPU (AMD Ryzen Threadripper

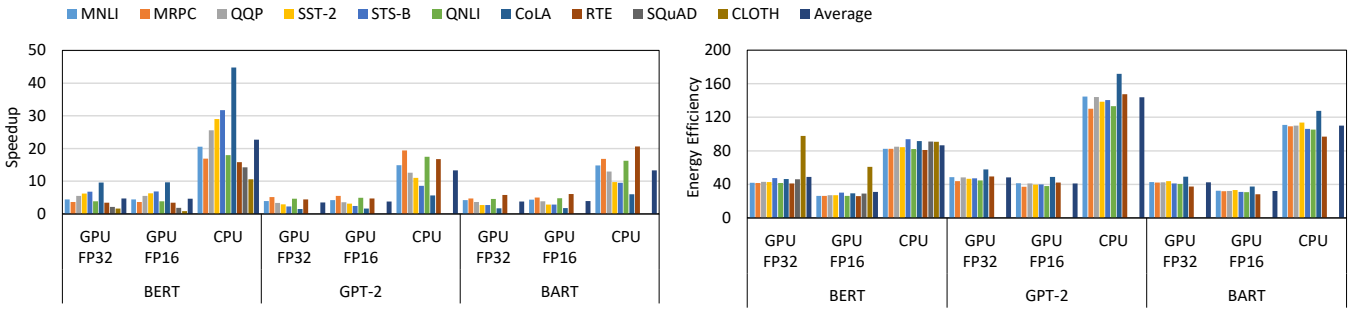


Figure 8: Speedup and energy efficiency improvement over GPU and CPU on BERT, GPT-2, BART benchmarks.

Table 3: Pruning results on different tasks. Acc. and S means accuracy and sparsity.

			MNLI	MRPCacc/F1	QQPacc/F1	SST-2	STS-B	QNLI	CoLA	RTE	SQuADEM/F1	CLOTH
BERT[16]	**Baseline**	**Acc.**	83.1	**86.0/90.2**	**90.6/87.3**	90.6	**89/88.6**	90.7	**54.2**	67.5	**81.4/88.6**	**79.1**
	Longformer[2]	**Acc.**	82.8	82.8/87.9	90.4/87.0	90.7	87.8/87.5	89.9	50.5	64.3	80.7/87.6	77.9
		S	-	-	-	-	-	-	-	-	0.34	0.22
	Bigbird[65]	**Acc.**	82.5	79.7/85.8	90.3/86.9	90.9	88.1/87.6	89.8	49.7	**68.3**	80.1/87.3	76.6
		S	-	-	-	-	-	-	-	-	0.34	0.22
	Sanger	**Acc.**	**83.2**	85.0/89.6	90.5/87.1	**91.1**	88.8/88.4	**90.8**	**54.2**	68.2	81.0/88.4	78.8
		S	0.109	0.115	0.172	0.229	0.313	0.088	0.331	0.087	0.127	0.158

3970X). We measure the performance of GPUs using PyTorch with cuBLAS 11.2, and CPUs using PyTorch MKL 2020.0.2. For Nvidia GPU and AMD CPU, we measure power consumption via nvidia-smi and reading RAPL counters, and execution latency using CUDA Event and native Python. On GPU platforms, we choose to evaluate the performance using CuBLAS not CuSparse on sparse models for two reasons. First, CuSparse library is designed for high performance computing (HPC) workloads with extremely high sparsity (<0.0001), e.g., Netflix [4] (5.7e-6 sparsity, recommendation system), Amazon0312 [38] (1.9e-5 sparsity, Co-purchase network). The sparsity in Transformers is only around 0.1, and thus CuSparse is inefficient for this case. For example, for the CLOTH task (sequence length of 512 and 0.15 sparsity), CuSparse takes similar latency as CuBLAS. Second, we deal with dynamic sparsity in this work. If we use CuSparse, it requires to transform the sparse data to sparse format (e.g. CSR) on the fly, which takes extra time. In many cases, the transformation overhead is as long as executing the kernel.

We also compare to the state-of-the-art sparse attention accelerators, including A^3 [19], SpAtten [57], FTRANS [31]. For a fair comparison, we scale the number of multipliers of all accelerators to 128 (16×8 RePE array with 16 vertical ports of Sanger) and assume the frequency is 1GHz. For a fair comparison, we take both pruning technique and architecture design into account, we calculate the effective throughput of the accelerators, which we define as *architectural throughput* $\times$ *computation saving*.

7.2 Pruning Results

We obtain the attention mask S by applying a binary threshold on a low-precision estimation $\hat{P}$ of the attention matrix. Compared to static and data-dependent patterns, the sparsity patterns we generate are conditioned on individual input samples. Such dynamic patterns can better adapt to data, thereby improving the sparsity-accuracy curve.

The results on BERT [16] are shown in Table 3. The column headings are NLP tasks. For a given Transformer network, it can be applied to multiple tasks. The row labeled "baseline" corresponds to the original BERT-base (with dense attention) for these tasks. we compare our method with two common sparse patterns: longformer [2] and bigbird [65]. While these patterns are originally proposed for processing long sequences (e.g., text length 4096), we scale them for standard benchmarks with shorter contexts. Both patterns have 1 global block and 3 sliding window blocks per row in our reproduction, and bigbird has 1 additional random block. We also scale down their block size to 32 to accommodate reduced sequence lengths. We evaluate all three methods using the same initial checkpoint and dataset as in our experiments. Longformer and bigbird are fine-tuned on the same pre-training dataset as BERT to recover accuracy before training on downstream tasks. For all tasks, we report task-specific metrics and the corresponding pruning ratio of each pattern.

For BERT, we achieve the same level of accuracy as the dense baseline while achieving 3.0 - 11.5X computation saving. For some tasks such as SST-2 and RTE, Sanger even outperforms the dense baseline slightly, which can be partly attributed to better concentration of attention [68] because of thresholding. We also notice that tasks of longer sequence length involve more redundancy, e.g., SQuAD and CLOTH compared to CoLA and STS-B. In the comparison, longformer and bigbird obtain a lower compression ratio on SQuAD and CLOTH, and fail to produce sparsity on the GLUE tasks because the typical sequence lengths of these tasks are almost comparable to their block size. Besides, they also incur more significant accuracy degradation in most tasks.

We also apply our pruning technique on GPT-2 [45] and BART [30] benchmarks on GLUE tasks. Similar results are observed. Overall, Sanger achieves 0.15 - 0.35 and 0.23 - 0.54 sparsity within 0.5% accuracy loss for GPT-2 and BART, respectively. The lower sparsity

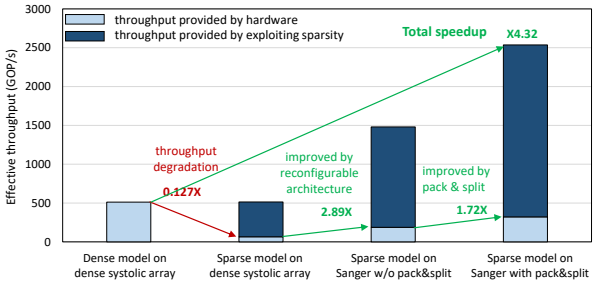


Figure 9: Performance breakdown

in GPT-2 and BART is probably due to their structural features, requiring additional tuning compared to BERT. For example, half of the attention matrix in the decoder is masked off, leaving less room for compression.

7.3 Comparison with CPUs and GPUs

Figure 8 shows the speedup of Sanger over AMD Ryzen CPU and V100 GPU on 10 language processing tasks. On the GPU platform, we evaluate FP32 data precision and FP16, where FP16 uses tensor cores for acceleration. For BERT, GPT-2, BART benchmarks, on average, Sanger achieves 4.71X, 6.45X, 3.76X speedup over GPU-FP32, 4.64X, 6.88X, 3.94X speedup over GPU-FP16, and 22.7X, 13.3X, 13.2X speedup over AMD Ryzen CPU. For energy-efficiency, Sanger is 48X, 35X, 113X better compared to GPU-FP32, GPU-FP16, and Ryzen CPU. The high speedup comes from the reduced computation and the optimized architecture of Sanger that can effectively leverage the sparsity in the attention mechanism. The performance of GPU-FP16 is worse than GPU-FP32 in some tasks due to the limited input sequence length and batch size. The performance with tensor core improves with larger workloads. For example, for the first 8 tasks, the input sequence length is up to 128, leading to the under-utilization of GPU tensor cores. For SQuAD and CLOTH tasks, the sequence length is 384 and 512. Therefore, GPU-FP16 outperforms GPU-FP32 in these two tasks. Sanger processes 64 queries and 64 keys in parallel, which means the computing resources are nearly fully-utilized in these tasks. We also observe that the speedup of Sanger does not perfectly scale with the sparsity. For example, on BERT, the sparsity in the MNLI task is 0.208, which is 2.51X over the CoLA task (0.083 sparsity). However, the speedup of CoLA (9.58X) is only 2.18X of MNLI (4.39X). This is because a higher sparsity means less nonzeros in each row, which further reduces the PE utilization.

Figure 9 gives the performance breakdown of Sanger. The baseline is executing a dense model on a dense systolic array. After pruning, the dense architecture cannot effectively leverage the sparsity, leading to throughput degradation. With the score-stationary dataflow and reconfigurable design, Sanger achieves a 2.89X speedup. Packing and splitting lead to a further 1.72X speedup due to balanced workloads. The software and hardware together achieve 4.32X speedup compared to baseline.

7.4 Comparison with Other Accelerators

Table 4 compares Sanger with three state-of-the-art sparse attention accelerators in both software design and hardware design. The comparison is based on the BERT model that is supported by all accelerators (SQuAD task). Sanger shows the highest computation saving at the software level thanks to our fine-grained pruning technique and reconfigurable architecture. In comparison, A^3 introduces a pre-processing step and uses a rough sparsity prediction technique which harms model accuracy under aggressive pruning. SpAtten uses a coarse-grained approach for attention pruning where it prunes entire columns and rows progressively. Such structural constraint limits the level of sparsity it can exploit in a similar way to traditional weight pruning. FTRANS [31] leverages the sparsity in the weight matrix used to generate queries, keys, and values with input sequence. We calculate the computation saving of FTRANS under 4×4 circulant matrix as it shows no accuracy loss. With a circulant matrix representation, the vector-matrix multiplication is equivalent to element-wise vector multiplication in the frequency domain. It takes at least 3 multipliers to perform complex value multiplication. Therefore, the computation saving of FTRANS is 1.33X (16/12).

From the hardware design aspect, Sanger outperforms A^3, SpAtten, FTRANS with 2.39X, 1.47X, 3.11X speedup respectively. SpAtten, and FTRANS apply fixed computing modules, limiting the pattern of sparsity strictly. While Sanger is composed of a reconfigurable systolic array with more flexibility during pruning, allowing higher effective throughput. SpAtten generates sparsity by a top-K engine that makes the token indices irregular. Therefore, SpAtten requires a large reconfigurable adder tree to gather the partial sums. A^3 applies candidate selection to find the most important attentions, which requires more area than computing modules. FTRANS needs to inversely transform the results from the frequency domain to the time domain, which is designed as an inverse Fast Fourier Transformation (IFFT) module.

7.5 Pattern Visualization and Impact of Packing

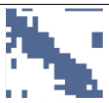
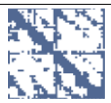
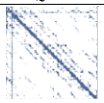
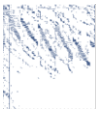
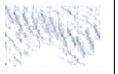
In this section, we study how packing and splitting impact hardware efficiency. Table 5 visualizes the sparsity pattern on each downstream task with or without our encoding scheme. These patterns exhibit different pixel granularity due to different input sequence lengths. For example, the attention matrix of MNLI is 16×16, while it is 384×384 for CLOTH. To demonstrate the benefit of our encoding scheme, we also list the PE utilization for each sparsity.

The encoded pattern is generated under the constraint that the maximum percentage of nonzeros in each sub-row of blocks is limited to 25%. We observe that all original sparsity patterns show uneven nonzero distribution, leading to low PE utilization (0.36 - 0.56). After packing and splitting, the mask becomes more structured, making the utilization increase to 0.56 - 0.72 with an average improvement of 1.5X. Because the packing and splitting are performed in the row dimension, the resulting attention masks have the same width as the original matrix while varies in their heights. Encoded attention masks of the first nine tasks exhibit thin-tall shapes because their short sequence lengths lead to denser patterns with fewer empty sub-rows to be packed. In comparison, the masks of CLOTH show short-fat shapes because their sequence lengths are much longer and have more zeros to be skipped.

Table 4: Comparison with existing sparse attention accelerators.

Attention accelerators	Software design			Hardware design				
	Sparse matrix	Pruning technique	Computation saving	Computing modules	Sparsity encoding	Sparsity decoding	Target device	Effective throughput
A^3	attention (unstructured)	greedy iterative approximation	1.72X	dot-product unit	Candidate Selection	Post-Scoring Selection	ASIC(40nm)	221 GOP/s
SpAtten	attention (structured)	head pruning token pruning	3.00X	parallel matrix mul	top-k engine	reconfigurable adder tree	ASIC(55nm)	360 GOP/s
FTRANS	weight (structured)	circulant matrix	1.33X	element-wise vec-mul	FFT	IFFT	FPGA(16nm)	170 GOP/s
Sanger	attention (structured)	prediction + thresholding	5.90X	reconfigurable systolic array	Prediction & pack/split	none	ASIC(55nm)	529 GOP/s

Table 5: Sparsity pattern visualization.

	MNLI	MRPC	QQP	SST-2	STS-B	QNLI	CoLA	RTE	SQuAD	CLOTH
No packing										
PE utilization	0.521	0.565	0.446	0.429	0.360	0.558	0.330	0.587	0.367	0.452
Use packing & splitting (constraint: 25% nonzeros)										
PE utilization (25% constraint)	0.719	0.721	0.690	0.697	0.603	0.721	0.625	0.735	0.629	0.563
Improvement (25% constraint)	1.38X	1.28X	1.55X	1.63X	1.68X	1.29X	1.90X	1.25X	1.72X	1.25X

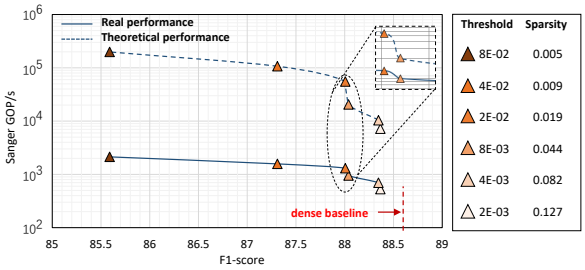


Figure 10: Trade-off between Sanger performance and F1-score

7.6 Impact of Binary Thresholding

As described in Section 7.5, we obtain the attention mask by applying a binary threshold T to the predicted attention matrix. Naturally, the larger T becomes, the more connections are pruned in the attention mechanism. We empirically choose 4-bit (the MSB of queries and keys) to predict the sparse attention mask. This is because 4-bit is the sweet point between accuracy and hardware efficiency.

Figure 10 shows how model accuracy and effective throughput of Sanger are affected by the choice of the threshold. We collect data from a BERT-based model fine-tuned on SQuAD v1.1. We achieve more computation reduction with a higher compression ratio, which boosts the effective throughput of Sanger. Meanwhile, the F1 score only drops moderately thanks to our fine-grained structured pruning scheme. In other words, the speedup compared to SpAtten [57] and A^3 [19] can be further improved with higher sparsity as shown. The users can use Sanger to explore the trade-off between speedup, sparsity, and accuracy. Besides, the gap between actual and theoretical performance grows more significant as the threshold increases, which echoes with our observation that a highly sparse attention matrix leads to low hardware efficiency.

8 RELATED WORK

Sparsity in attention mechanism. There are two lines of work trying to exploit the sparsity in the attention mechanism. The first line of work [12, 13, 68] aims to improve either accuracy or interpretability by concentrating attention on a few important tokens rather than assigning non-zero credits to all tokens. More specifically, these works perform sparsification via explicit top-k selection [68] or sparse variants of softmax [12, 13] after the full attention matrix has been computed. In this way, they can learn arbitrary patterns and fully utilize the intrinsic structure of input data. In contrast, the second line of work uses sparsity to improve algorithmic efficiency [2, 11, 22, 25, 48, 52, 65]. We refer readers to [54] for a more comprehensive survey on this topic.

Deep learning sparse accelerators. Architecture, modeling, optimization, and auto-generation techniques for dense DNN accelerators have been widely explored [6–8, 17, 28, 34, 35, 43, 61].

As sparsification is an effective approach for computation reduction, a number of studies focus on supporting sparse computation with efficient architecture designs [9, 14, 15, 18–21, 24, 26, 28, 29, 31–33, 36, 37, 40, 41, 44, 49, 51, 57, 64, 67, 69]. Among them, [14, 15, 18, 20, 24, 26, 69, 70] apply software-hardware co-design approaches where a specialized sparsity pattern is designed in synergy with the accelerator. These sparse accelerators primarily focus on static sparsity. One of the main contributions of these works is their novel sparse format designs (e.g. Bitmap in SIGMA[44], CISS in Tensaurus[51], hierarchical bitmap in SMASH[24]) for storing the sparse data. However, the sparsity involved in the attention mechanism is dynamic, which is generated by multiplying queries and keys. It will incur a very high overhead to conduct such complex format transformation on chip for an end-to-end attention implementation. Sanger avoids sparsity decoding overhead with a score-stationary dataflow that unifies SDDMM and SpMM via a reconfigurable architecture.

In recent years, a few designs have also been proposed to accelerate attention mechanisms [19, 27, 31, 57]. However, these accelerators are limited in either efficiency or generality. A^3 [19] mainly focuses on approximating attention with a reduced amount of computation. At the architecture level, it only designs several dot-product modules for acceleration, lacking detailed architecture optimizations. FTRANS [31] is an FPGA accelerator which applies Fast Fourier Transformation to the attention mechanism. However, it restricts the weight matrices to be block-circulant, losing generality for other applications. SpAtten [57] applies token pruning technique that removes entire rows and columns of the attention matrix, but its computation engine is inefficient for accelerating dynamic sparsity.

9 CONCLUSION

We propose Sanger, a co-design framework for enabling sparse attention on a reconfigurable systolic array. We first apply quantized prediction to obtain the sparse attention mask. Then, to ensure a balanced workload of hardware implementation, we propose an encoding scheme that transforms the sparse mask into multiple structured blocks. In hardware aspects, we design a score-stationary dataflow that unifies the computation of SDDMM and SpMM. We implement this dataflow on a reconfigurable systolic array, which supports various sparsity patterns by dynamically configuring the datapath. Sanger achieves 2.39X and 1.47X speedup over A^3 and SpAtten accelerators.

ACKNOWLEDGMENTS

This work was supported in part by the Beijing Natural Science Foundation (No. JQ19014) and PCL lab.

A ARTIFACT APPENDIX

A.1 Abstract

In this letter we provide detailed information that will facilitate the artifact evaluation process. In the artifact checklist section, we present brief information about this artifact, and outline basic requirements to reproduce the experiment results. Then we describe the directory tree of our codebase and go into more detail about the requirements. Finally, in the experiment workflow section we explain step by step how to reproduce the experiments.

A.2 Artifact check-list (meta-information)

- **Algorithm:** Sanger, a framework that harvests sparsity in the attention mechanism through synergistic hardware and software co-design. The software part prunes the attention matrix into a dynamic structured pattern, and the hardware part features a reconfigurable architecture that exploits such a pattern.
- **Program:** Python
- **Model:** We use BERT-Base-Uncased (420.1MB), GPT2-Small (522.7MB) and BART-Base (532.1MB). Our scripts download pre-trained checkpoints from the Hugging Face Model Hub https://huggingface.co/models automatically.
- **Data set:** We evaluate models on GLUE (1.2GB), SQuAD v1.1 (120MB) and CLOTH (60MB). Our scripts can automatically download the GLUE and the SQuAD dataset. As for the CLOTH dataset, you can download it from this link: https://www.cs.cmu.edu/~glai1/data/cloth/
- **Run-time environment:** CentOS 7, CUDA SDK 10.1 or higher.
- **Hardware:** NVIDIA Tesla V100-PCIE-16GB GPU, AMD Ryzen Threadripper 3970X CPU
- **Execution:** Internet connection is required to download the GLUE and the SQuAD dataset.
- **Experiments:** We provide shell scripts that can be used to reproduce the results of the experiments.
- **How much disk space are required (approximately)?:** The codebase and downloaded datasets take up about 1.5GB in total, but note that you need to make sure you have enough space for the checkpoints. Each checkpoint takes up about 500 MB of space.
- **How much time is needed to prepare workflow (approximately)?:** Less than an hour
- **How much time is needed to complete experiments (approximately)?:** Training a model could take from 1 minute to 3 hours, depending on the task. Evaluating a model on any task should finish within 5 minutes. The time required to conduct other experiments is negligible.

A.3 Description

Below we introduce some important files and directories in the codebase. For a more detailed description, please refer to the **README** in the GitHub repository.

- **configs/**. This sub-directory contains configuration files for dense models, sparse models, and static sparsity (BigBird, Longformer, etc.).
- **scripts/**. This sub-directory holds the shell scripts for running experiments.
- **bench_cpu_gpu.py**. This script benchmarks dense attention on CPU and GPU.
- **modeling_<model>.py**. These files contain implementations of the BERT, GPT2 and BART models, supporting both dense and sparse attention.

- **modeling_sanger_attn.py**. This file contains an implementation of the sparse attention algorithm of Sanger, and some helper functions for measuring sparsity and load balance.
- **modeling_static_spattn.py**. This file implements some attention mechanisms with static sparsity, e.g. BigBird and Longformer.
- **run_<task>.py**. These files are intended for training or evaluating models on GLUE, SQuAD or CLOTH.

A.3.1 How to access. You can access our codebase from here: https://github.com/pku-liang/Sanger The unpacked artifact takes approximately 10MB of disk space.

A.3.2 Hardware dependencies. Benchmarking experiments on CPU and GPU require an NVIDIA Tesla V100-PCIE-16GB GPU and an AMD Ryzen Threadripper 3970X CPU.

A.3.3 Software dependencies. Software experiments require CUDA SDK 10.1 or higher and Python 3.7 or higher. Other dependent Python packages are listed in **requirements.txt**.

A.3.4 Data sets. We evaluate models on three datasets, namely GLUE, SQuAD, and CLOTH. They correspond to three different NLP tasks, namely text classification, question answering, and long-context cloze. Our script automatically downloads the GLUE and SQuAD datasets before training or evaluation, so you do not need to download them manually. As for the CLOTH dataset, you need to download and extract it to the **data/cloth** sub-directory.

A.3.5 Models. We evaluate the efficiency of Sanger on three models, namely BERT-Base-Uncased, GPT2-Small, and BART-Base. Our scripts can automatically download pre-trained models from the Model Hub, so there is no need to download them manually. Alternatively, you can also download fine-tuned checkpoints and evaluate them directly.

A.4 Installation

- Install all software dependencies.
- Create a virtual environment with a Python version of at least 3.7.
- Install all dependent Python packages in **requirements.txt** using pip.

A.5 Experiment workflow

Evaluate Sanger performance.

- **Train a model with Sanger sparse attention**. We provide scripts for training in the **scripts/** sub-directory. For example, to train a Sanger-pruned BERT-Base model on SQuAD, you can execute **scripts/train_sparse_on_squad.sh**. Note that you have to pass in an appropriate configuration file, which you can find in **configs/**. You can skip this step if you choose to load a fine-tuned checkpoint directly.
- **Evaluate the fine-tuned model**. We also provide scripts for evaluation in **scripts/**. For example, to evaluate the sparse model from the last step, you can execute **scripts/eval_sparse_on_squad.sh**. If you need to load a checkpoint from a non-standard location, be sure to change the path in the script. When the evaluation is complete, the script should print out the accuracy.
- **Measure sparsity and load balance**. Each evaluation script contains a flag that enables measuring the sparsity level of attention and calculating the load balance of the PE array. If you set this flag in the previous step, the script will log the results to a CSV file named **load_balance.csv** during evaluation.

Comparison with dense attention and static sparse attention.

- **Train a model with dense or static sparse attention**. We provide dedicated scripts for train models with dense attention (e.g. **scripts/train_dense_on_squad.sh**). To train a model with static sparse attention, you can use the same script as Sanger and pass in an appropriate configuration file (e.g. **bert_base_longformer.json**).
- **Evaluate the fine-tuned model**. The process is similar to evaluating Sanger models. Note that you also need to use different scripts when evaluating dense models.

Comparison with CPU and GPU. You can measure the latency of dense attention on CPU and GPU by executing **bench_cpu_gpu.py**.

A.6 Evaluation and expected results

On the software side, following the workflow described above, you should be able to reproduce the results in Figure 8, Figure 10, Table 3, and Table 5 of the paper.

REFERENCES

[1] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. 2012. Chisel: constructing hardware in a scala embedded language. In *Proceedings of the Design Automation Conference (DAC)*.

[2] Iz Beltagy, Matthew E Peters, and Arman Cohan. 2020. Longformer: The long-document transformer. *arXiv preprint arXiv:2004.05150* (2020).

[3] Yoshua Bengio, N. Léonard, and Aaron C. Courville. 2013. Estimating or Propagating Gradients Through Stochastic Neurons for Conditional Computation. *ArXiv* abs/1308.3432 (2013).

[4] James Bennett and Stan Lanning. 2007. The netflix prize. In *Proceedings of KDD cup and workshop*.

[5] Shijie Cao, Lingxiao Ma, W. Xiao, Chen Zhang, Yunxin Liu, L. Zhang, L. Nie, and Z. Yang. 2019. SeerNet: Predicting Convolutional Neural Network Feature-Map Sparsity Through Low-Bit Quantization. In *Proceedings of Conference on Computer Vision and Pattern Recognition (CVPR)*.

[6] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. 2014. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. *Sigplan Notices* (2014).

[7] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and Oliver Temam. 2014. Dadiannao: A machine-learning supercomputer. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*.

[8] Yu-Hsin Chen, Tushar Krishna, Joel S Emer, and Vivienne Sze. 2016. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *Journal of Solid-State Circuits* (2016).

[9] Yu-Hsin Chen, Tien-Ju Yang, Joel Emer, and Vivienne Sze. 2018. Eyeriss v2: A Flexible Accelerator for Emerging Deep Neural Networks on Mobile Devices. *arXiv preprint arXiv:1807.07928* (2018).

[10] Sharan Chetlur, C. Woolley, Philippe Vandermersch, J. Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cuDNN: Efficient Primitives for Deep Learning. *ArXiv* abs/1410.0759 (2014).

[11] Rewon Child, Scott Gray, Alec Radford, and Ilya Sutskever. 2019. Generating long sequences with sparse transformers. *arXiv preprint arXiv:1904.10509* (2019).

[12] Gonçalo M. Correia, Vlad Niculae, and André F. T. Martins. 2019. Adaptively Sparse Transformers. In *Proceedings of Conference on Empirical Methods in Natural Language Processing/International Joint Conference on Natural Language Processing*.

[13] Baiyun Cui, Y. Li, Ming Chen, and Z. Zhang. 2019. Fine-tune BERT with Sparse Self-Attention Mechanism. In *Proceedings of Conference on Empirical Methods in*

Natural Language Processing/International Joint Conference on Natural Language Processing.

[14] Alberto Delmas Lascorz, Patrick Judd, Dylan Malone Stuart, Zissis Poulos, Mostafa Mahmoud, Sayeh Sharify, Milos Nikolic, Kevin Siu, and Andreas Moshovos. 2019. Bit-tactical: A software/hardware approach to exploiting value and bit sparsity in neural networks. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[15] Chunhua Deng, Siyu Liao, Yi Xie, Keshab K Parhi, Xuehai Qian, and Bo Yuan. 2018. PermDNN: Efficient Compressed DNN Architecture with Permuted Diagonal Matrices. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*.

[16] J. Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics – Human Language Technologies*.

[17] Zidong Du, Ro Fasthuber, Tianshi Chen, Paolo Ienne, Ling Li, Tao Luo, Xiaobing Feng, Yunji Chen, and Olivier Temam. 2015. ShiDianNao: Shifting vision processing closer to the sensor. In *Proceedings of SIGARCH Computer Architecture News*.

[18] Cong Guo, Bo Yang Hsueh, Jingwen Leng, Yuxian Qiu, Yue Guan, Zehuan Wang, Xiaoying Jia, Xipeng Li, Minyi Guo, and Yuhao Zhu. 2020. Accelerating sparse DNN models without hardware-support via tile-wise sparsity. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.

[19] Tae Jun Ham, Sung Jun Jung, Seonghak Kim, Young H Oh, Yeonhong Park, Yoonho Song, Jung-Hun Park, Sanghee Lee, Kyoung Park, Jae W Lee, et al. 2020. A 3: Accelerating Attention Mechanisms in Neural Networks with Approximation. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*.

[20] Song Han, Junlong Kang, Huizi Mao, Yiming Hu, Xin Li, Yubin Li, Dongliang Xie, Hong Luo, Song Yao, and Yu Wang. 2017. ESE: Efficient Speech Recognition Engine with Sparse LSTM on FPGA. In *Proceedings of the International Symposium on Field Programmable Gate Arrays (FPGA)*.

[21] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J Dally. 2016. EIE: efficient inference engine on compressed deep neural network. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*.

[22] Weizhe Hua, Yuan Zhou, Christopher De Sa, Zhiru Zhang, and G Edward Suh. 2019. Boosting the performance of CNN accelerators with dynamic fine-grained channel gating. In *Proceedings of the 52nd International Symposium on Microarchitecture (MICRO)*.

[23] Intel. 2021. Oneapi-Src/oneDNN. https://github.com/oneapi-src/oneDNN.

[24] Konstantinos Kanellopoulos, Nandita Vijaykumar, Christina Giannoula, Roknoddin Azizi, Skanda Koppula, Nika Mansouri Ghiasi, Taha Shahroodi, Juan Gomez Luna, and Onur Mutlu. 2019. SMASH: Co-designing Software Compression and Hardware-Accelerated Indexing for Efficient Sparse Matrix Operations. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*.

[25] Nikita Kitaev, Lukasz Kaiser, and Anselm Levskaya. 2020. Reformer: The Efficient Transformer. *ArXiv* abs/2001.04451 (2020).

[26] HT Kung, Bradley McDanel, and Sai Qian Zhang. 2019. Packing sparse convolutional neural networks for efficient systolic array implementations: Column combining under joint optimization. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[27] HT Kung, Bradley McDanel, Sai Qian Zhang, Xin Dong, and Chih Chiang Chen. 2019. Maestro: A memory-on-logic architecture for coordinated parallel use of many systolic arrays. In *Proceedings of International Conference on Application-specific Systems, Architectures and Processors (ASAP)*.

[28] Hyoukjun Kwon, Ananda Samajdar, and Tushar Krishna. 2018. Maeri: Enabling flexible dataflow mapping over dnn accelerators via reconfigurable interconnects. In *SIGPLAN Notices*, Vol. 53. 461–475.

[29] Alberto Delmás Lascorz, Sayeh Sharify, Isak Edo, Dylan Malone Stuart, Omar Mohamed Awad, Patrick Judd, Mostafa Mahmoud, Milos Nikolic, Kevin Siu, Zissis Poulos, et al. 2019. Shapeshifter: Enabling fine-grain data width adaptation in deep learning. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*.

[30] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Ves Stoyanov, and Luke Zettlemoyer. 2020. Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. *Proceedings of Annual Conference of the Association for Computational Linguistics* (2020).

[31] Bingbing Li, Santosh Pandey, Haowen Fang, Yanjun Lyv, Ji Li, Jieyang Chen, Mimi Xie, Lipeng Wan, Hang Liu, and Caiwen Ding. 2020. FTRANS: energy-efficient acceleration of transformers using FPGA. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED)*.

[32] Yun Liang, Liqiang Lu, Yicheng Jin, Jiaming Xie, Ruirui Huang, Jiansong Zhang, and Wei Lin. 2021. An Efficient Hardware Design for Accelerating Sparse CNNs with NAS-based Models. *Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* (2021).

[33] Yun Liang, Liqiang Lu, and Jiaming Xie. 2020. OMNI: A framework for integrating hardware and software optimizations for sparse CNNs. *Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* (2020).

[34] Daofu Liu, Tianshi Chen, Shaoli Liu, Jinhong Zhou, Shengyuan Zhou, Olivier Teman, Xiaobing Feng, Xuehai Zhou, and Yunji Chen. 2015. Pudiannao: A polyvalent machine learning accelerator. In *Proceedings of SIGARCH Computer Architecture News*.

[35] Liqiang Lu, Naiqing Guan, Yuyue Wang, Liancheng Jia, Zizhang Luo, Jieming Yin, Jason Cong, and Yun Liang. 2021. TENET: A Framework for Modeling Tensor Dataflow Based on Relation-centric Notation. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*.

[36] Liqiang Lu and Yun Liang. 2018. SpWA: an efficient sparse winograd convolutional neural networks accelerator on FPGAs. In *Proceedings of the Design Automation Conference (DAC)*.

[37] Liqiang Lu, Jiaming Xie, Ruirui Huang, Jiansong Zhang, Wei Lin, and Yun Liang. 2019. An efficient hardware accelerator for sparse convolutional neural networks on FPGAs. In *Proceedings of International Symposium on Field-Programmable Custom Computing Machines (FCCM)*.

[38] Julian McAuley and Jure Leskovec. 2013. Hidden factors and hidden topics: understanding rating dimensions with review text. In *Proceedings of The conference on Recommender systems (RecSys)*.

[39] NVIDIA. 2021. NVIDIA/DeepLearningExamples. https://github.com/NVIDIA/DeepLearningExamples.

[40] Subhankar Pal, Jonathan Beaumont, Dong-Hyeon Park, Aporva Amarnath, Siying Feng, Chaitali Chakrabarti, Hun-Seok Kim, David Blaauw, Trevor Mudge, and Ronald Dreslinski. 2018. Outerspace: An outer product based sparse matrix multiplication accelerator. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*.

[41] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W Keckler, and William J Dally. 2017. Scnn: An accelerator for compressed-sparse convolutional neural networks. In *Proceedings of SIGARCH Computer Architecture News*.

[42] Adam Paszke, S. Gross, Francisco Massa, A. Lerer, J. Bradbury, G. Chanan, Trevor Killeen, Z. Lin, N. Gimelshein, L. Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, B. Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *NeurIPS*.

[43] Wajahat Qadeer, Rehan Hameed, Ofer Shacham, Preethi Venkatesan, Christos Kozyrakis, and Mark A Horowitz. 2013. Convolution engine: balancing efficiency & flexibility in specialized computing. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA)*.

[44] Eric Qin, Ananda Samajdar, Hyoukjun Kwon, Vineet Nadella, Sudarshan Srinivasan, Dipankar Das, Bharat Kaul, and Tushar Krishna. 2020. SIGMA: A Sparse and Irregular GEMM Accelerator with Flexible Interconnects for DNN Training. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*.

[45] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language models are unsupervised multitask learners. *OpenAI blog* (2019).

[46] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog* (2019).

[47] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. 2016. SQuAD: 100, 000+ Questions for Machine Comprehension of Text. In *Proceedings of Conference on Empirical Methods in Natural Language Processing*.

[48] Aurko Roy, M. Saffar, Ashish Vaswani, and David Grangier. 2020. Efficient Content-Based Sparse Attention with Routing Transformers. In *Proceedings of Transactions of the Association for Computational Linguistics (TACL)*.

[49] Shaden Smith and George Karypis. 2015. Tensor-matrix products with a compressed sparse tensor. In *Proceedings of the Workshop on Irregular Applications: Architectures and Algorithms*.

[50] Mingcong Song, J. Zhao, Y. Hu, Jiaqi Zhang, and Tao Li. 2018. Prediction Based Execution on Deep Neural Networks. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*.

[51] Nitish Srivastava, Hanchen Jin, Shaden Smith, Hongbo Rong, David Albonesi, and Zhiru Zhang. 2020. Tensaurus: A versatile accelerator for mixed sparse-dense tensor computations. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*.

[52] Yi Tay, Dara Bahri, L. Yang, Donald Metzler, and D. Juan. 2020. Sparse Sinkhorn Attention. In *Proceedings of International Conference on Machine Learning (ICML)*.

[53] Yi Tay, M. Dehghani, Samira Abnar, Y. Shen, Dara Bahri, Philip Pham, J. Rao, Liu Yang, Sebastian Ruder, and Donald Metzler. 2020. Long Range Arena: A Benchmark for Efficient Transformers. *ArXiv* abs/2011.04006 (2020).

[54] Yi Tay, Mostafa Dehghani, Dara Bahri, and Donald Metzler. 2020. Efficient transformers: A survey. *arXiv preprint arXiv:2009.06732* (2020).

[55] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems (NeurIPS)*.

[56] Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R. Bowman. 2018. GLUE: A Multi-Task Benchmark and Analysis Platform for Natural Language Understanding. In *Proceedings of the Workshop: Analyzing and Interpreting Neural Networks for NLP*.

[57] Hanrui Wang, Zhekai Zhang, and Song Han. 2021. SpAtten: Efficient Sparse Attention Architecture with Cascade Token and Head Pruning. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*.

[58] Meiqi Wang, Siyuan Lu, Danyang Zhu, Jun Lin, and Zhongfeng Wang. 2018. A high-speed and low-complexity architecture for softmax function in deep learning. In *Proceedings of 2018 Asia Pacific Conference on Circuits and Systems*.

[59] X. Wang, Ross B. Girshick, A. Gupta, and Kaiming He. 2018. Non-local Neural Networks. In *Proceedings of Conference on Computer Vision and Pattern Recognition (CVPR)*.

[60] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, R'emi Louf, Morgan Funtowicz, and Jamie Brew. 2020. Transformers: State-of-the-Art Natural Language Processing. In *Proceedings of Conference on Empirical Methods in Natural Language Processing*.

[61] Qingcheng Xiao, Size Zheng, Bingzhe Wu, Xu Pengcheng, Xuehai Qian, and Yun Liang. 2021. HASCO: Towards Agile HArdware and Software CO-design for Tensor Computation. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*.

[62] Qizhe Xie, Guokun Lai, Zihang Dai, and E. Hovy. 2018. Large-scale Cloze Test Dataset Created by Teachers. In *Proceedings of Conference on Empirical Methods in Natural Language Processing*.

[63] Kelvin Xu, Jimmy Ba, Ryan Kiros, Kyunghyun Cho, Aaron C. Courville, R. Salakhutdinov, R. Zemel, and Yoshua Bengio. 2015. Show, Attend and Tell: Neural Image Caption Generation with Visual Attention. In *Proceedings of International Conference on Machine Learning (ICML)*.

[64] Tzu-Hsien Yang, Hsiang-Yun Cheng, Chia-Lin Yang, I-Ching Tseng, Han-Wen Hu, Hung-Sheng Chang, and Hsiang-Pang Li. 2019. Sparse reram engine: Joint exploration of activation and weight sparsity in compressed neural networks. In *Proceedings of the 46th International Symposium on Computer Architecture (ISCA)*.

[65] Manzil Zaheer, Guru Guruganesh, Avinava Dubey, Joshua Ainslie, Chris Alberti, Santiago Ontanon, Philip Pham, Anirudh Ravula, Qifan Wang, Li Yang, et al. 2020. Big bird: Transformers for longer sequences. *arXiv preprint arXiv:2007.14062* (2020).

[66] Han Zhang, I. Goodfellow, Dimitris N. Metaxas, and Augustus Odena. 2019. Self-Attention Generative Adversarial Networks. In *Proceedings of International Conference on Machine Learning (ICML)*.

[67] Shijin Zhang, Zidong Du, Lei Zhang, Huiying Lan, Shaoli Liu, Ling Li, Qi Guo, Tianshi Chen, and Yunji Chen. 2016. Cambricon-X: An accelerator for sparse neural networks. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*.

[68] Guangxiang Zhao, Junyang Lin, Zhiyuan Zhang, Xuancheng Ren, Qi Su, and X. Sun. 2019. Explicit Sparse Transformer: Concentrated Attention Through Explicit Selection. *ArXiv* abs/1912.11637 (2019).

[69] Xuda Zhou, Zidong Du, Qi Guo, Shaoli Liu, Chengsi Liu, Chao Wang, Xuehai Zhou, Ling Li, Tianshi Chen, and Yunji Chen. 2018. Cambricon-S: Addressing Irregularity in Sparse Neural Networks through A Cooperative Software/Hardware Approach. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*.

[70] Maohua Zhu, Tao Zhang, Zhenyu Gu, and Yuan Xie. 2019. Sparse Tensor Core: Algorithm and Hardware Co-Design for Vector-wise Sparse Neural Networks on Modern GPUs. In *Proceedings of the 52nd International Symposium on Microarchitecture (MICRO)*.