GFormer: Accelerating Large Language Models with Optimized Transformers on Gaudi Processors

Chengming Zhang
University of Houston
czhang59@cougarnet.uh.edu

Xinheng Ding
Indiana University
xinhding@iu.edu

Baixi Sun
Indiana University
sunbaix@iu.edu

Xiaodong Yu
Stevens Institute of Technology
xyu38@stevens.edu

Weijian Zheng
Argonne National Laboratory
wzheng@anl.gov

Zhen Xie
Binghamton University
zxie3@binghamton.edu

Dingwen Tao
Indiana University
ditao@iu.edu

***Abstract*—Heterogeneous hardware like Gaudi processor has been developed to enhance computations, especially matrix operations for Transformer-based large language models (LLMs) for generative AI tasks. However, our analysis indicates that Transformers are not fully optimized on such emerging hardware, primarily due to inadequate optimizations in non-matrix computational kernels like Softmax and in heterogeneous resource utilization, particularly when processing long sequences. To address these issues, we propose an integrated approach (called GFormer) that merges sparse and linear attention mechanisms. GFormer aims to maximize the computational capabilities of the Gaudi processor's Matrix Multiplication Engine (MME) and Tensor Processing Cores (TPC) without compromising model quality. GFormer includes a windowed self-attention kernel and an efficient outer product kernel for causal linear attention, aiming to optimize LLM inference on Gaudi processors. Evaluation shows that GFormer significantly improves efficiency and model performance across various tasks on the Gaudi processor and outperforms state-of-the-art GPUs.**

I. Introduction

Transformers [1] have revolutionized the field of natural language processing (NLP) and beyond, becoming the backbone of numerous state-of-the-art (SOTA) machine learning applications across machine translation [2], question answering, and computer vision [3]. Despite Transformers' broad applicability and potential to catalyze advancements across various fields, the computation efficiency of Transformers presents a significant challenge that hampers their broader adoption and scalability. At the heart of this issue is the self-attention mechanism requires quadratic memory and time complexity $O(N^2)$ for processing contexts of N inputs [4]. As the ambition to process longer sequences and build larger, more comprehensive models grows, this quadratic bottleneck becomes increasingly prohibitive. The situation is further exacerbated by the trend towards ever-increasing model sizes and sequence lengths in pursuit of enhanced performance and generalization capabilities.

To address these challenges, exploring specialized hardware accelerators, such as Intel Gaudi processors [5], AMD Versal ACAP AI Engines (AIEs), SambaNova Reconfigurable Dataflow Units (RDUs) [6], and Cerebras's wafer-scale engine (WSE) [7], has emerged as a promising avenue for mitigating the computation demands of training and inference of large Transformer-based models. Gaudi processors stand out for their innovative architecture designed specifically to accelerate deep learning (DL) workloads and offer a heterogeneous compute architecture comprising a Matrix Multiplication Engine (MME) and a cluster of fully programmable Tensor Processing Cores (TPCs). This combination allows Gaudi to efficiently handle various DL operations, both matrix-based and non-matrix-based, with high performance and flexibility.

Despite the potential showcased by Gaudi, Softmax operations in Transformers become a performance bottleneck when processing long sequence inputs [8]. The main reasons are ❶ The computational complexity of Softmax operations in a Transformer is $\mathcal{O}(N^2)$. ❷ Softmax operations are mapped into TPC, but reduction operations in Softmax are not well-suited for single instruction multiple data (SIMD) architectures like TPC (see more details about TPC architecture in Section II). Long sequences further exacerbate this problem especially when the sequence length exceeds 2048. Overall, the limited computational capability of TPC combined with complexities of Softmax operations in Transformers hinders Gaudi's overall performance and efficiency.

Existing algorithmic approaches to optimize Transformers fall into three categories: ❶ Exploiting the sparsity of attention matrices, exemplified by methods such as Reformer [9] and Big Bird [10]. ❷ Applying kernel methods [11], including Performer [12] and Transformers as RNNs [4], to approximate and eliminate Softmax operations. This attention mechanism is referred to as *linear attention* because its complexity reduces to $\mathcal{O}(N)$ upon the removal of Softmax. ❸ Combining diverse attention methods to enhance Transformers' performance. An example is [13], which successfully integrates sparse and kernel methods to improve model quality. While such integrations often lead to models of higher quality, they may also result in increased computational load and slower processing speeds.

Moreover, challenges arise when directly adapting these efficient Transformer techniques to Gaudi processors. Specifically, Gaudi processors feature a heterogeneous compute architecture comprising Matrix Multiplication Engines (MME) and Tensor Processing Cores (TPC). However, the sparse attention

mechanism, which introduces irregular memory access and computation, is primarily mapped onto TPCs, leaving MMEs, which are not programmable and only support dense matrix-matrix operations, idle in scenarios requiring sparse attention. Conversely, linear attention, which is fundamentally based on matrix multiplication, can utilize almost all calculations on MMEs due to their stronger computational capabilities, but this leaves TPCs idle in such cases. This situation raises a critical question: *Can we effectively combine sparse and dense attention mechanisms in a way that fully leverages both MME and TPC, while maintaining the quality of the model?*

To this end, we propose an optimized Gaudi-based Transformer (called GFormer) for large language models (LLMs) acceleration on the Gaudi processor. GFormer synergistically combines sparse and linear attention mechanisms to enhance computational efficiency by fully leveraging both MME and TPC while preserving model quality. Key components of our framework include: ❶ The integration of diverse attention mechanisms to optimize both computation efficiency and model fidelity. ❷ The implementation of a windowed local-context self-attention kernel utilizing the vector units in TPC, aimed at maximizing computational throughput. ❸ The development of an efficient outer product TPC kernel for handling a subset of the outer product operations in causal linear attention, effectively balancing the workload between MME and TPC. ❹ The introduction of an optimal workload partitioning algorithm to ensure balanced utilization of TPC and MME resources. To the best of our knowledge, *this is the first work that facilitates high-performance and high-utilization LLM inference on heterogeneous hardware like Gaudi processors*. This exploration aims to harness the computational capabilities of Gaudi and the characteristics of LLMs, inspiring future innovative ML hardware designs.

The main contributions of this paper are summarized below:

- We introduce an innovative approach to integrate disparate sparse and linear attention mechanisms. This strategy is designed to fully utilize the computational capabilities of MME and TPC on the Gaudi processor.
- We develop a windowed local-context self-attention kernel that is specifically tailored for TPC. This kernel is optimized to leverage TPC's local memory and vectorized load and store operations.
- We present an efficient outer product kernel for TPC, employing the vector unit (SIMD) to optimize the processing of causal linear attention operations.
- We introduce a performance modeling technique for TPC and MME. This model is instrumental in balancing workloads between TPC and MME.
- We evaluate GFormer on GPT and ViT models and find that it achieves up to 2× and 2.2× speedups, respectively.

II. Background and Motivation

In this section, we present background information for Transformers, the Gaudi processor architecture, the TPC programming model, and our motivation.

A. Transformers

Transformer [1] architecture departs from previous sequence-to-sequence models by relying on self-attention to draw global dependencies among inputs, which allows it to handle sequences of data in parallel and capture long-range dependencies more effectively. Figure 1 presents the architecture of a Transformer, which typically consists of an encoder, a decoder, and other operations such as position embedding. We describe the decoder for simplicity. A decoder contains masked multi-head self-attention and a fully connected feed-forward network. The masked multi-head self-attention mechanism prevents the current token from attending to tokens in masked positions. The feed-forward network provides further transformation of the attention-aggregated information.

Causal language models are just concerned with the previous context (tokens on the left) when predicting the next token in a sequence of tokens. In Softmax-based self attention, we add an attention mask matrix into the raw attention matrix to mask attention on the right of the current position. All elements of the lower triangular part of the attention mask matrix are 0, and the other elements of the attention mask matrix are set to $-\infty$. Here we refer to such an attention mechanism as **causal attention**. Generative Pre-trained Transformer (GPT) models [14] are based on the Transformer decoder architecture. GPT models are typical causal language models. GPT models are characterized by their large scale, extensive pre-training on diverse text corpora, and their ability to adapt to a wide range of tasks with minimal task-specific modifications.

Vision Transformer (ViT) [3] adapts the Transformer for image classification tasks. By treating images as sequences of patches (akin to words in a sentence), ViT applies the **self-attention** mechanism across these patches to capture global dependencies within the image. ViT can attend to image tokens bidirectionally (full access to the image tokens on the left and right). This approach has demonstrated competitive or superior performance to conventional convolutional neural networks (CNNs) on image classification benchmarks.

B. Efficient Attention Mechanism

Efficient attention mechanisms aim to reduce the computational complexity traditionally associated with the Softmax-based attention in Transformers, which scales quadratically with the sequence length. Specifically, sparse attention selectively focuses on a subset of key positions for each query in the sequence, rather than attending to all positions. This selective focus drastically reduces the number of computations and memory requirements, as the attention matrix is no longer fully dense but sparse. For example, Longformer [15] introduces a sparse attention mechanism that judiciously selects a subset of positions to attend to, blending local with a few global attention patterns. Big Bird [10] incorporates a unique mix of random, global, and sliding window attention. This approach not only maintains the model's ability to grasp complex

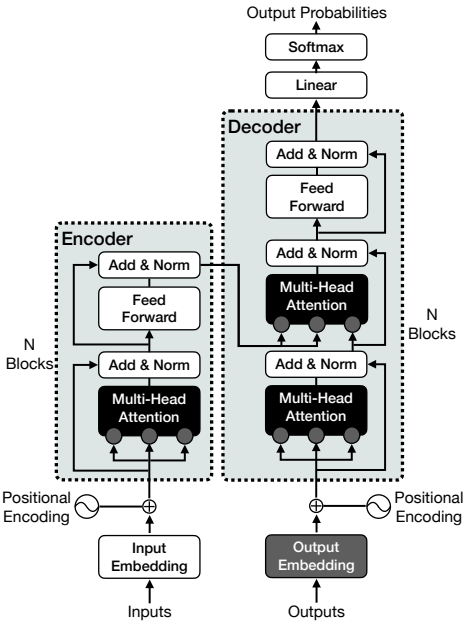

Fig. 1. Overview of Transformer architecture.

dependencies over vast stretches of text but also does so with enhanced flexibility and efficiency.

The equation $\text{softmax}(QK^T)V \approx \phi(Q)(\phi(K)^T V)$ expresses the idea of linear attention. Linear attention first uses the kernel method ϕ to project query Q and key K matrices into feature spaces to approximate and remove Softmax operations. We can first compute $(\phi(K)^T V)$ to avoid explicit calculation of attention matrix $\text{softmax}(QK^T)$. Thus the computation complexity of linear attention becomes $O(N)$. For example, "Transformers are RNNs" [4] employs a simple feature map defined below:

$$\phi(x) = elu(x) + 1 \tag{1}$$

The Performer [12] employs a randomized feature map to approximate the Softmax attention.

C. Gaudi Processor Architecture

Gaudi processor is a specialized hardware accelerator designed for deep learning training workloads [5]. As shown in Figure 2, it features a heterogeneous compute architecture with a Matrix Multiplication Engine (MME), eight fully programmable Tensor Processing Cores (TPC), and fast memory and network units [16]. The MME is specifically tuned for doing all operations that can be lowered to matrix multiplication, such as fully connected layers, convolutions, and batched GEMM. The TPC is a very long instruction word (VLIW) single instruction multiple data (SIMD) processor crafted for deep learning nonlinear operations.

The fast memory and network units enhance intra-/inter-processor data transfers. Four high-bandwidth memory (HBM) devices provide 32 GB of capacity with one terabyte-per-second of memory bandwidth. Shared memory can be used to streamline the data exchange between MME and TPC. On-chip ten 100 gigabit integrated remote direct memory access (RDMA) over converged Ethernet (RoCE) ports facilitate efficient inter-processor communication.

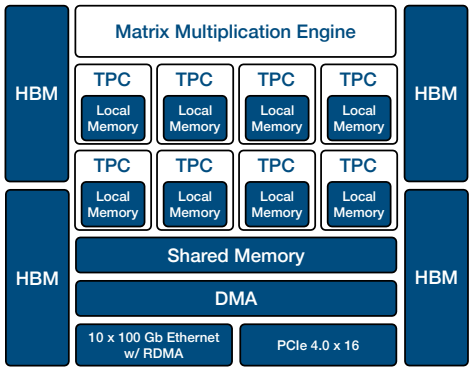

Fig. 2. A high-level view of Gaudi architecture, which consists of Matrix Multiplication Engine (MME), Tensor Processing Cores (TPC), Memory Units (Local Memory, Shared Memory, DMA, HBM), and Connection Units (Ethernet, PCIe).

D. TPC Programming

a) ***TPC architecture****:* TPC is responsible for executing non-linear deep learning operators. Its wide SIMD vector unit supports 2048-bit SIMD operations with data types such as float, bfloat16, INT16, INT32, and INT8. The TPC's arithmetic logic unit can execute up to 64 floats/INT32, 128 INT16, or 256 INT8 operations per cycle. Multiple TPC cores in Gaudi can be executed in parallel.

TPC processor includes four distinct memory spaces: scalar local memory, vector local memory, global memory, and configuration space. Global memory is accessed through specialized access points termed tensors. A 2,048-bit vector can be loaded from or written to global memory every four cycles, on average. Local memory of each TPC processor is divided into scalar local memory (1 KB) and vector local memory (80 KB). Local memory can be either read from or written to on every cycle with no bandwidth constraint [17].

b) ***TPC programming****:* TPC is programmed via TPC-C, a derivative of C language. A TPC program contains TPC code (kernel) and host glue code. TPC code is the actual kernel implementation. TPC CLANG compiler is based on LLVM and is used for TPC kernels' compilation, simulation, and debugging. Host glue code is executed on the host machine and controls TPC kernels' execution. A TPC kernel only accepts tensors as inputs or outputs with dimensions ranging from 1 to 5. Index spacing, similar to threads in CUDA programming, efficiently divides workloads among TPC processors. Each index space member corresponds to an independent unit of work executed on a single TPC. TPC CLANG compiler also provides intrinsic functions for optimized kernel implementation. Intrinsics encompass arithmetic, bitwise, logical, load, store, et al, operations.

E. Motivation

The impressive ability of Transformer-based models comes from complex computational operations and the huge number of parameters (340 million in BERT, 1.5 billion in GPT-3) [2], [18], which results in intensive computations during training and inference. Consequently, training and inference of Transformer-based models is both time-consuming and

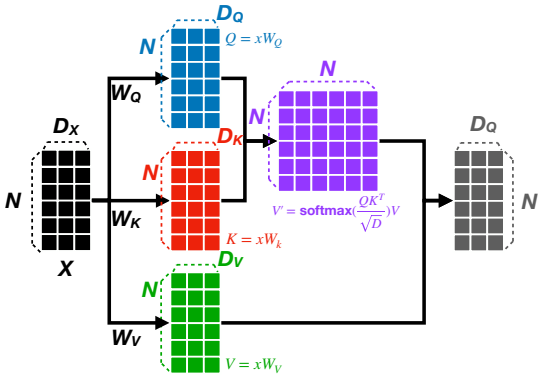

Fig. 3. Matrix Computation workflow of each self-attention. Q, K and V are query, key, value matrices of dimension size N by D_Q,D_K, D_V, respectively.

resource-intensive. Utilizing a new efficient Transformer architecture is a possible solution to reduce computation complexity. However, the Gaudi-specific optimizations on Transformer architecture are not well studied. Additionally, Figure 3 shows the computation flow of Softmax-based self-attention. Specifically, The input sequence $x \in \mathbb{R}^{N \times D_x}$ is projected by three weight matrices W_Q, W_K, W_V to corresponding representations Q, K and V. Following common terminology, the Q, K, and V are referred to as the "queries", "keys", and "values" respectively. Then Softmax is used to normalize the attention matrix QK^T into a probability distribution. As indicated in [8], The Softmax operation is only executed on TPC and becomes a performance bottleneck when processing long sequence inputs. But there is no existing approach to breaking this bottleneck on Gaudi processors. Furthermore, Gaudi processors feature heterogeneous compute architecture comprising MME and TPC. It is worthwhile to balance workloads between MME and TPC to fully utilize the computation resources of both MME and TPC. However, there is no previous method to investigate balancing workloads on Gaudi processors.

$$\begin{aligned} Q &= xW_Q \\ K &= xW_k \\ V &= xW_V \\ V' &= \text{softmax}(\frac{QK^T}{\sqrt{D}})V \end{aligned} \tag{2}$$

III. Design Methodology

In this section, we propose our optimized Transformer design and optimized TPC kernels.

A. Overview of This Work

Figure 2 shows the overview of GFormer. The Gaudi processor is a heterogeneous architecture comprising a cluster of TPCs, as well as configurable MMEs. To maximize the utilization of both TPC and MME, our proposed design effectively combines sparse and linear attention approximations in the following ways. Inputs of self-attention are Q, K, and V. They are referred to as the "queries", "keys" and "values" respectively. $Q, K, V \in \mathbb{R}^{B \times N \times H \times E}$, where B, N, H, and E are batch size, sequence length, the number of heads, head size, respectively. We split Q, K, V along the head dimension (H) into two groups, as inputs of sparse attention and inputs of linear attention. The partition is according to a hyperparameter τ. $H \times \tau$ is the number of heads for sparse attention and $H \times (1 - \tau)$ is the number of heads for linear attention.

For the sparse attention part, to take full advantage of capabilities of SIMD architecture in TPC, we adopt a windowed local-context self-attention and implement an efficient TPC kernel. Window attention avoids irregular data access and enables task partition across multiple TPCs. For the linear attention part, inspired by the Performer, we use positive orthogonal random features to approximate the Softmax operation in the Transformer [12]. Specifically, $\text{softmax}(QK^T)V \approx \phi(Q)(\phi(K)^T V)$, where ϕ is the feature map. Most calculations of linear attention are matrix-matrix multiplication and can be mapped to MME, which brings two benefits. (1) it takes advantage of powerful MME. (2) it avoids the data movement between MME and TPC. Our mixed approach not only maximizes hardware utilization in the Gaudi processor but also helps reduce the accuracy loss caused by the Softmax approximation. We expect that executions of TPC and MME will overlap through our optimization.

B. TPC Best Fitted Sparse Attention

Problems: Softmax applies the standard exponential function to each element of the input tensor and normalizes these values by dividing by the summation of all these exponentials along a specific dimension. The computational complexity of Softmax operations is $\mathcal{O}(N^2)$. Sparse attention, for example, Longformer [15], Big Bird [10], is proposed to reduce computational complexity.

Challenges: However, we face two challenges when performing Softmax on an irregularly sparse attention matrix. First, irregular data access leads to high data reading latency and low TPC utilization. Second, it is not able to perform index space mapping (divide workloads) between TPC processors evenly. The TPC programming only supports linear transformations for mapping, but the random number of non-zero elements in each row of the sparse attention matrix causes these linear transformations to fail.

Proposed design: To overcome these challenges and benefit the computation pattern of TPC, we adopt a windowed local-context self-attention. Our attention pattern employs a fixed-size window attention surrounding each token. Using multiple stacked layers of such windowed attention results in a large receptive field, where top layers have access to all input locations and have the capacity to build representations that incorporate information across the entire input, similar to CNNs. Given a fixed window size w, each token attends to w previous (left) tokens. Besides, the window size is a multiple of 64 to fully utilize vector units in TPC. Since a TPC's wide SIMD vector unit supports 2048-bit SIMD operations. The TPC can execute up to 64 float operations in parallel in one cycle. Additionally, the TPC prefers directly loading a vector of values from global memory.

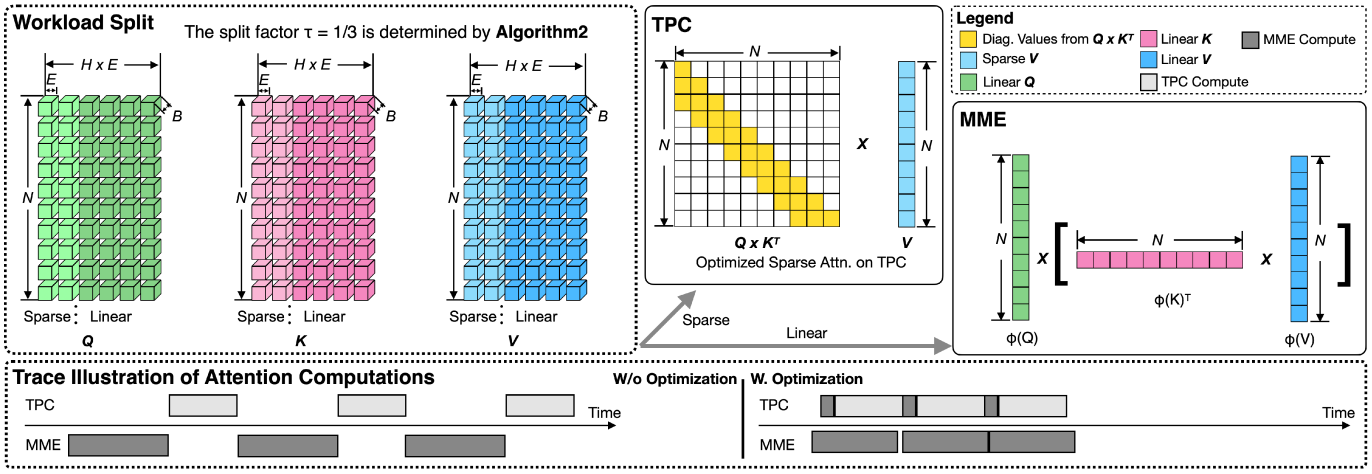

Fig. 4. Overview of GFormer workflow. τ of the heads is for sparse attention.

Listing 1 illustrates the pseudocode of windowed attention on TPC. Specifically, we declare exp_x in local memory (Line 1) to enable high-bandwidth reading and writing. We directly load a vector of data into x from the input tensor (Line 22). We apply the exponential function to each element of x to generate y (Line 23). We store y into exp_x (Line 24). We avoid writing y into global memory to speed up intermediate data write and read since window size is typically less than 256, in which all exponential data y can be held in local memory. Then we obtain reciprocal of the sum of exponents and multiply all exponents by the reciprocal value (Lines 28-40). The advantages of this TPC kernel are (1) utilizing local memory. (2) vectorized load and store.

```
1  __local__ float64 exp_x[C];
2  void main(tensor in, tensor out, int window_start,
       int window_size) {
3  const int5 index_space_start =
       get_index_space_offset();
4  const int5 index_space_end = get_index_space_size
       () + index_space_start;
5  const int width_step  = 64;
6  int w_s = window_start;
7  int w_e   = window_start + window_size;
8  const int height_step  = 1;
9  const int h_s=index_space_start[1]*height_step;
10 const int h_e=index_space_end[1]*height_step;
11 int5 coords = {w_s, h_s, 0, 0, 0};
12 float64 x, y, sum;
13 int e_i = 0;
14 for (int h = h_s; h < h_e; h += height_step)
15 {
16   coords[1] = h;
17   sum = 0.f;
18   for (int w = w_s; w < w_e; w += width_step)
19   {
20     coords[0] = w;
21     // Load input tensors
22     x = v_f32_ld_tnsr_b(coords, in);
23     y = v_exp_cephes_fast_f32(x);
24     exp_x[e_i] = y;
25     e_i = e_i + 1;
26     sum = sum + y;
27   }
28  // Sum across the vector
29   sum = v_f32_reduce_add(sum);
30  // 1/(sum_of_exponents)
31   sum = v_reciprocal_f32(sum);
32   e_i = 0;
33   for (int w = w_s; w < w_e; w += width_step)
34   {
35     coords[0] = w;
36     x = exp_x[e_i];
37     e_i = e_i + 1;
38     // Multiply exp(x) * 1/(sum_of_exponents)
39     y = x * sum;
40     v_f32_st_tnsr(coords, out, y);
41   }
42 } }
```

Listing 1. Pseudocode of windowed attention on TPC.

C. Efficient Outer Product on TPC

Problems: For causal linear attention, we need manually to let the current token only pay attention to previous tokens since there is no attention mask matrix in the causal linear attention scenario. Algorithm 1 describes computation procedures in causal linear attention. Inputs of the procedure are Q, K, and V. The Q, K, and V are referred to as the "queries", "keys" and "values" respectively. The output of the procedure is H (hidden state). Given that subscripting a matrix with i returns the i-th row as a vector. Within the loop, each pair of Q_i and K_i is transformed using a random feature map ϕ, resulting in Q_i' and K_i'. A normalized output vector H_i is computed via accumulation of outer product of K_i' and V_i from 1 to i.

Algorithm 1: Causal linear attention.

Inputs : $\{Q_i\}_{i=1}^{N}, \{K_i\}_{i=1}^{N}, \{V_i\}_{i=1}^{N}$
Outputs: $\{H_i\}_{i=1}^{N}$

$A \in \mathbb{R}^{E\times E}, Z \in \mathbb{R}^{E}$
$A, Z \leftarrow 0, 0$
for $i = 1$ **to** N **do**
  # Random feature maps
  ${Q_i}', {K_i}' \leftarrow \phi(Q_i), \phi(K_i)$
  $A \leftarrow A + {K_i}' \bigotimes V_i$
  $Z \leftarrow Z + {K_i}'$
  ${H_i}^T \leftarrow {{K_i}'}^T S/({Q_i}' \cdot Z)$
end

Listing 2 describes the Pytorch implementation of the causal linear attention on Gaudi. Specifically, MME is not

programmable and only supports matrix-matrix multiplication. To achieve outer product operation, we first insert a dimension of size one into k and v at the specified dimension. Then we perform a batch matrix-matrix multiplication of k_prime and v. We then perform the cumulative sum of att_raw and att_norm over the sequence length dimension. Figure 5 shows the profiling result of causal linear attention of such implementation. We can find the MME is overwhelmed by matrix-matrix multiplication. But TPC is quite idle. The reason is that the last dimensions of k and v (head dimension) are usually less than 64, Lines 7 and 8 in Listing 2 are actually $B \times H \times N$ small matrix-matrix multiplication $(1 \times E \times E \times 1)$, where B, H, N, and E are batch size, the number heads, sequence length, head size, respectively. The small size of matrix-matrix multiplication is not very efficient in MME.

```
1  def linear_causal_attention(q, k, v):
2      # Project key and queries onto the feature
           map space
3      k_prime = feature_map(k)
4      q_prime = feature_map(q)
5
6      ref_v = torch.ones_like(v.unsqueeze(2))
7      out_k_v=k_prime.unsqueeze(3)@v.unsqueeze(2)
8      norm = k_prime.unsqueeze(3) @ ref_v
9      # Consolidate against the feature dimension
10     att_raw = q_prime.unsqueeze(2) @ out_k_v
11     att_norm = q_prime.unsqueeze(2) @ norm
12     # Cumulative sum over the sequence
13     att_raw = att_raw.cumsum(1)
14     att_norm = att_norm.cumsum(1)
15     att_raw = att_raw.squeeze(2)
16     att_norm = att_norm.squeeze(2)
17     # Normalize
18     attn = att_raw / att_norm
19     return attn
```

Listing 2. Pseudocode of causal linear attention.

Challenges: To balance the utilization of both MME and TPC. We propose to map a proportion of outer product operations onto TPC. However, the challenge is to implement an efficient outer product kernel on TPC, which achieves similar performance as MME. Otherwise, the outer products on TPC will instead become a bottleneck. Specifically, we need to solve the following issues: (1) We need to evenly distribute outer product operations among TPC to avoid load imbalance problems. (2) As aforementioned, we need the For loop to implement $B \times H \times N$ outer product operations, but serial execution characteristics of the For loop will hinder instruction parallelism on TPC. (3) In the outer product between vector 1 and vector 2. One element from vector 1 is multiplied by all elements of vector 2. Additionally, we prefer to directly load a vector of data into TPC for higher data reading bandwidth. However, TPC intrinsics do not support accessing a specific element in the vector data, which prevents us from achieving the outer product.

Proposed implementation: To this end, we proposed an

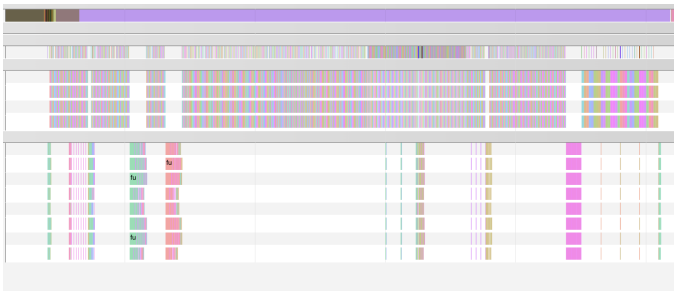

Fig. 5. Profiling result of original causal linear attention.

Fig. 6. Profiling result of optimized causal linear attention.

efficient outer product kernel design in Listing 3 to solve the aforesaid problems. Specifically, we assume the head size is 64. To solve the first issue, we partition computation using index space (Lines 3-10) to evenly distribute outer product operations among TPCs. We load a row of data from a_mat and b_mat into a_v, and b_v, respectively (Lines 17-18). For the second issue, We unroll the For loop to increase the instruction level parallelism (Line 19). For the third issue, we design a function, v_broadcast_element_f32 to enable access to the element in vector and convert the element into vector. The idea of v_broadcast_element_f32 is bit-level shuffle and shift. We then can broadcast the element in a_v at w position into a vector a_x. Then b_v is multiplied by a_x to generate an output vector (Line 20-24). The advantages of this TPC kernel are (1) vectorized load and store. (2) efficient vector multiplication using SIMD. (3) without insertion of an extra dimension (avoiding memory movement). Figure 6 depicts the profiling result of causal linear attention after optimization. Both MME and TPC have relatively balanced utilization.

```
1  #define A_LEN 64
2  void main(tensor a_mat, tensor b_mat, tensor o_mat
       ) {
3    const int5 index_space_start =
         get_index_space_offset();
4    const int5 index_space_end =
         get_index_space_size() + index_space_start;
5    int5 a_coords = {0};
6    int5 b_coords = {0};
7    int5 o_coords = {0};
8    const int height_step  = 1;
9    const int h_s=index_space_start[1]*height_step;
10   const int h_e=index_space_end[1]*height_step;
11   for (int h = h_s; h < h_e; h += height_step) {
12     a_coords[1] = h;
13     a_coords[0] = 0;
14     b_coords[1] = h;
15     o_coords[1] = h;
16     // Load a row from a_mat, b_mat
17     float64 a_v=v_f32_ld_tnsr_b(a_coords, a_mat);
```

```
18      float64 b_v=v_f32_ld_tnsr_b(b_coords, b_mat);
19      #pragma unroll (8)
20      for(int w = 0; w < A_LEN; w += 1) {
21       float64 a_x=v_broadcast_element_f32(a_v, w);
22        float64 out_v = v_f32_mul_b(a_x, b_v);
23        o_coords[0] = w * A_LEN;
24        v_f32_st_tnsr(o_coords, o_mat, out_v);
25      }
26    } }
```

Listing 3. Pseudocode of outer product on TPC.

D. Optimal Partition Algorithm

As discussed in Section III-A, to fully utilize both MME and TPC, we partition Q, K, and V along head dimension according to a hyperparameter τ for sparse attention and linear attention. The overall system's runtime is determined by the maximum runtime between the TPC and MME, so it is crucial to balance their respective workloads. We are required to carefully determine the hyperparameter τ to make MME and TPC has balanced workloads. To achieve this balance, we model computation latencies of both the TPC and MME, which allows us to obtain the relationship between workload and runtime for each engine. By estimating the runtime of TPC and MME based on their respective workloads, we can obtain the optimal partition when their runtime is equal or similar.

Specifically, we first analyze workloads' floating point of operations (FLOPs) quantitatively. We then develop a set of micro-benchmarks to measure the average performance of workloads on the TPC and MME. These micro-benchmarks cover a wide range of sizes and sparsity levels. Algorithm 2 reveals how to obtain the optimal partition using FLOPs and average performance. Inputs are initial partition (p), problem size (size), FLOPs per unit of partition 0 (FLOPs_per0), FLOPs per unit of partition 1 (FLOPs_per1), average performance of partition 0 (perf0), and average performance of partition 1 (perf1). The output (p') is an optimal partition. We estimate the computation latency of partitions 0 and 1 (Lines 3-4). We find the optimal partition when latency0 and latency1 are equal or similar (Line 5).

Algorithm 2: Optimal partition algorithm.

Inputs : p, size, FLOPs_per0, FLOPs_per1, perf0, perf1
Outputs: p'

1 FLOPs_p0 = p × size × FLOPs_per0
2 FLOPs_p1 = (1-p) × size × FLOPs_per1
3 latency0 = $\frac{\text{FLOPs\_p0}}{\text{perf0}}$
4 latency1 = $\frac{\text{FLOPs\_p1}}{\text{perf1}}$
5 Find p' let latency0 = latency1

By estimating and balancing their runtime, we can achieve better overall performance on the Gaudi processor. Specifically, assuming H_0 is the number of sparse heads, H_1 is the number of linear heads, and $\frac{H_0}{H_1} = \frac{\tau}{(1-\tau)}$. From the perspective of linear attention, the number of FLOPs for applying random features onto Q and K is $4BN(H_1E)^2$. The number of FLOPs for $C = \phi(K)^T V$ is $2BN(H_1E)^2$. The number of FLOPs for $\phi(Q)C$ is $2BN(H_1E)^2$. Then the number of FLOPs for linear attention is:

$$4BN(H_1E)^2 + 2BN(H_1E)^2 + 2BN(H_1E)^2 = 8BN(H_1E)^2 \quad (3)$$

Thus the computation latency for linear attention on MME can be estimated as

$$MME\_latency = \frac{8BN(H_1E)^2}{MME\_perf} \quad (4)$$

From the perspective of sparse attention, the number of FLOPs for $R = QK^T$ is $2BH_0ENW$, where W is the window size. The number of FLOPs for sparse Softmax $A = \text{sparse softmax}(R)$ operations is $3BH_0NW$, where W is the window size. Then the number of FLOPs for sparse attention is:

$$2BH_0ENW + 3BH_0NW = 5BH_0NW \quad (5)$$

Thus the computation latency for sparse attention on TPC can be estimated as

$$TPC\_latency = \frac{5BH_0NW}{TPC\_perf} \quad (6)$$

We can obtain a suitable partition τ when letting $TPC\_latency$ is close to $MME\_latency$.

IV. Performance Evaluation

In this section, we present our experimental setup and demonstrate the effectiveness of GFormer compared with other solutions using different models.

A. Experimental Setup

a) ***Platforms****:* We perform our experiments on one Habana Labs System 1 (HLS-1) [5] AI training system. We implement HLS-1 using AWS EC2 DL1 instances [19]. The HLS-1 incorporates eight Gaudi processors and two Gen 4.0 PCIe switches. External host CPU is used to manage HLS-1 via PCIe switches. Each Gaudi processor is equipped with 1 MME, 8 TPC, and 32 GB on-chip memory. All experiments are on a single Gaudi processor.

b) ***Implementation details****:* We implement our proposed models based on PyTorch. Gaudi software stack version is 1.14.0. The Gaudi software stack includes the graph compiler, Gaudi driver, Gaudi firmware, and corresponding PyTorch. The PyTorch version is 2.1.1.

c) ***Models****:* For GPT model, we adopt GPT-Neo [20] architecture in Huggingface [21]. Other representative LLMs such as Llama [22] have similar structures. So for simplicity, we just evaluate the GPT model. For ViT model, we use the Vision Transformer [23] architecture in Huggingface. We set batch size, the number of layers, the number of heads, and head size as 4, 12, 16, and 64, respectively. These are realistic scenarios of parameter configurations found in GPT-3 with 125 million parameters [20] and Vision Transformer (ViT) base model [23].

B. Evaluation on Speedup of Sparse Attention Kernel

Figure 7 depicts the speedup of the windowed attention kernel on different window sizes and sequence lengths. We vary sequence length from 1k to 4k. Further increasing sequence length causes memory errors. The baseline is Softmax-based attention. Window 64 and Window 128 are short for windowed attention with window sizes 64 and 128. As the sequence length increases, the speedups of windowed attention over baseline are up to 1.7×.

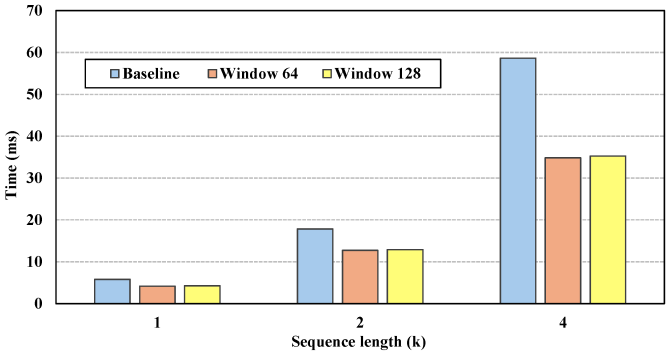

Fig. 7. Speedup of windowed attention kernel.

C. Evaluation on Performance of Outer Product kernel

As illustrated in Figure 8, we compare the performance of outer product kernels on both MME and TPC. As discussed in §III-C we use batch matrix-matrix multiplication to implement the outer product on MME. We directly implement our outer product kernel on TPC using TPC intrinsic. To compare the performance of outer product operations on CPU and GPU, we vary sequence length from 1k to 4k and the corresponding number of outer products varies from 64k to 512k. Even though TPC is less computationally powerful than the MME, the outer product kernel on TPC achieves a better performance (1.2×) than the outer product on MME.

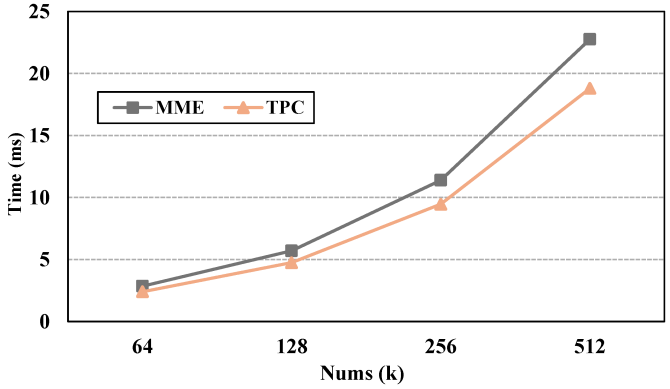

Fig. 8. Performance of Outer Product Kernel.

D. Evaluation on Partition

We evaluate our partition algorithm on both causal and self-attention. Here we fix the sequence length to 4k. Here we set MME_perf and TPC_perf as 13.37 TFLOPS and 2.31 TFLOPS according to extensive micro-benchmarks. Figure 9 shows the speedup over baseline using different percentages of sparse attention. For causal attention, we find the ideal number of sparse attention heads is 3 according to our partition algorithm. The experiment also shows that there is maximum speedup when the number of sparse attention heads is 3. For self-attention, we find the ideal number of sparse attention heads is 4 according to our partition algorithm. As shown in the Figure 9. There is maximum speedup when the number of sparse attention heads is 4. This experiment proves the effectiveness of our partition algorithm.

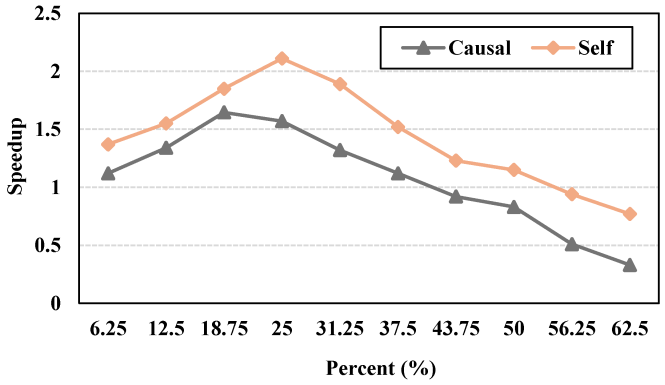

Fig. 9. Performance of different partition. Causal and self are short for causal attention and self-attention.

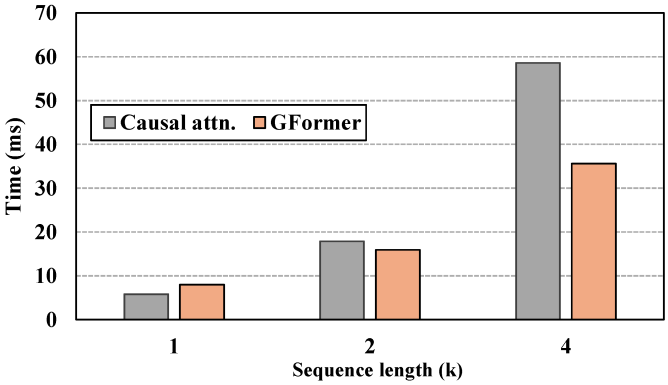

Fig. 10. Performance of causal attention.

E. Evaluation on Mixed Attention

GPT or ViT is composed of multiple identical attention layers. Hence we evaluate the performance of a single attention layer across different sequence lengths as shown in Figure 10 11. For causal attention, GFormer achieves up to 1.6× speedup over the baseline. For self-attention, GFormer achieves up to 2.1× speedup over the baseline.

F. Evaluation on Speedup and Accuracy

We evaluate our method's speedup and approximation accuracy in LLM and ViT. We compare it with baseline, Performer, and Big Bird. The baseline uses Softmax attention. The Performer adopts linear attention. The Big Bird uses sparse attention. Our approach mixes sparse attention and linear attention. All models contain 12 layers and 16 attention heads in each layer.

a) ***LLM Speedup****:* Figure 12 shows the performance of GPT with different attention mechanisms on Gaudi. Compared with the baseline, The Performer takes 16% more run time since causal linear attention in the GPT model has low computation efficiency. GPT with GFormer achieves 2.0 × speedup over baseline when sequence length is 6k due to efficiently balancing computation of linear causal attention on both MME and TPC.

b) ***LLM Accuracy****:* As shown in Table II, we compare perplexity between different models across wikitext-103 [24] and bookcorpus [25]. We pre-train all models on wikitext-103

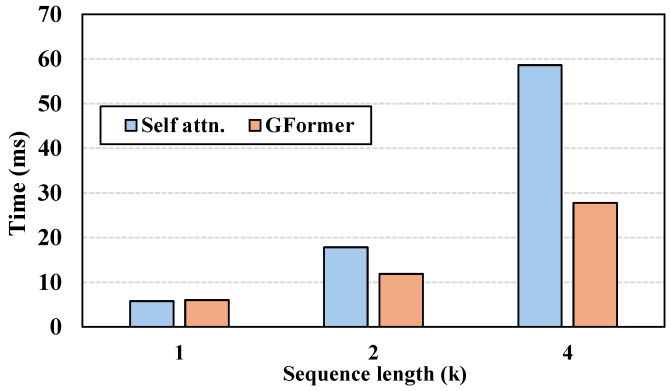

Fig. 11. Performance of self attention.

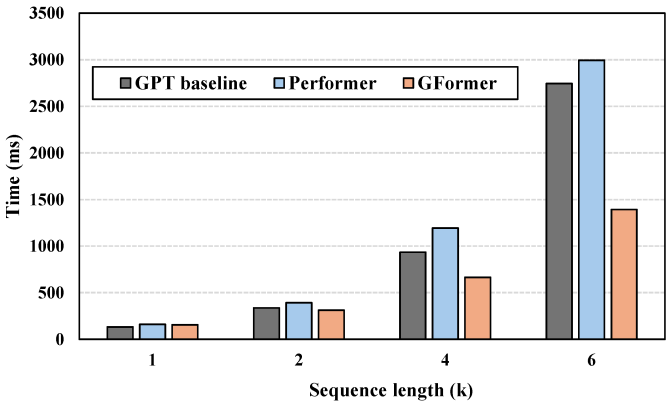

Fig. 12. Performance of GPT with our attention on Gaudi.

and bookcorpus for 120k steps. Lower Perplexity means better model performance. Perplexity (PPL) is one of the most common metrics for evaluating autoregressive or causal language models. Perplexity is defined as the exponentiated average negative log-likelihood of a sequence. For a tokenized sequence $X = (x_0, x_1, , x_t)$, $PPL(x) = exp(-\frac{1}{t}\sum_i^t logp(x_i|x_{<i}))$. The perplexity of our method is only 1.2 higher than the baseline but lower than other methods.

To evaluate the performance of our method on real NLP tasks, in Table I, we compare the performance of different models using the General Language Understanding Evaluation benchmark (GLUE) [26]. GLUE is a collection of resources for training, evaluating, and analyzing natural language understanding systems. Here we follow the BERT architecture [2]. We pre-train all our models on wikitext-103 for 120k steps and then fine-tune on GLUE. The average score of our method is only 5 lower than the baseline but higher than other methods.

*c) **ViT Speedup**:* As shown in Figure 13, the ViT model with Performer only achieves up to 1.1 × speedup over baseline. ViT model with GFormer achieves up to 2.2 × speedup over baseline since GFormer fully utilizes both MME and TPC.

*d) **ViT Accuracy**:* To evaluate different models' accuracy in Vision Transformer. We pre-train all models on ImageNet 2012 [27] (1 million images, 1,000 classes) at a resolution 384×384. We set the patch size to 16×16. As illustrated in Table III, ViT with GFormer has only 1.4% accuracy drop compared with baseline.

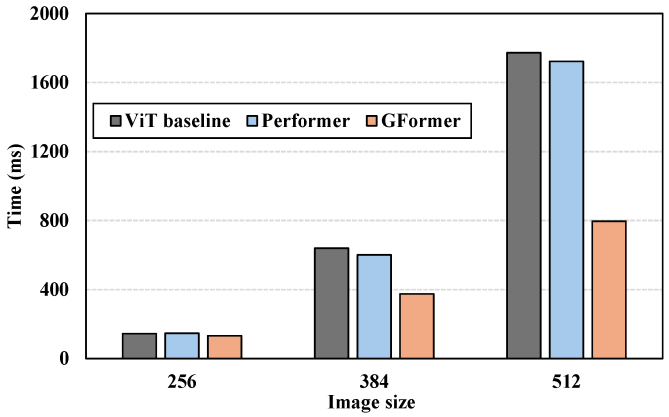

Fig. 13. Performance of ViT with our attention on Gaudi.

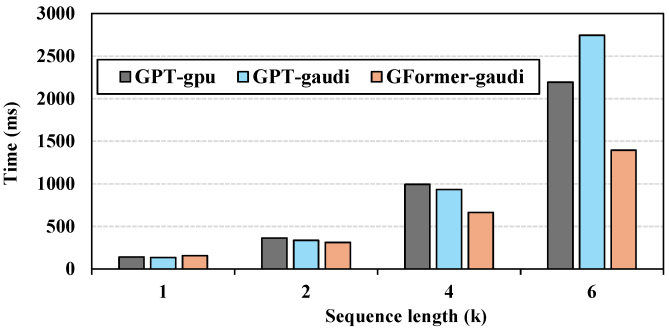

Fig. 14. Performance of GPT on GPU and Gaudi.

G. Comparison with GPUs.

We evaluate the performance of GPT and ViT on both GPU and Gaudi. The GPU type is V100 GPU with 32GB of memory on Bridges-2 [28]. As shown in Figure 14, we compare the GPT model with GFormer on Gaudi (GFormer-gaudi) with the GPT model with Softmax attention on Gaudi (GPT-gaudi) and the GPT model with Softmax attention on Gaudi on GPU (GPT-gpu). GPT-gaudi only has a similar performance to GPT-gpu. But it has worse performance when the sequence length reaches 6k. But our proposed GFormer-gaudi always has speedup over GPT-gpu, the speedup is up to 1.5 × when the sequence length is 6k.

Figure 15 shows the performance comparison among ViT with Softmax attention on GPU (ViT-gpu), ViT with Softmax attention on Gaudi (ViT-gaudi), and proposed ViT with GFormer on Gaudi (GFormer-gaudi). ViT-gaudi is 1.5 × on average slower than ViT-gpu. However, our proposed GFormer-gaudi achieves up to 1.2× speedup over Soft-gpu. These two experiments prove that transformer-based models on Gaudi can achieve speedup over GPU when we fully utilize its hardware resource.

V. Related Work

Existing works exemplify the ongoing efforts to redesign attention mechanisms from both algorithm and algorithm-hardware co-deign aspects to balance computational efficiency

TABLE I
A COMPARISON OF THE GLUE SCORES BETWEEN THE BASELINE (SOFTMAX ATTENTION), KERNEL METHODS (PERFORMER), SPARSE ATTENTION (BIG BIRD), AND OUR METHOD.

	GLUE								
Model	COLA (m)	SST-2 (a)	MRPC (f1/a)	STS-B (p/s)	QQP (f1/a)	MNLI (a)	QNLI (a)	RTE (a)	Average
Baseline	48.7	90.8	89.2/84.6	85.7/85.8	85.9/89.8	81.1	87.9	65.8	81.3
Performer	39.3	90.1	84.7/75.6	81.0/80.7	83.4/88.1	76.6	83.5	61.3	76.4
Big Bird	30.2	90.0	83.3/78.7	81.9/81.6	83.5/87.5	76.3	83.3	59.7	76.1
Our	40.1	90.4	84.6/75.9	81.7/81.8	83.7/88.2	76.9	83.4	62.6	**76.8**

TABLE II
A COMPARISON OF PERPLEXITY BETWEEN ALL THE MODELS ON WIKITEXT-103 AND BOOKCORPUS

Model	wikitext-103 (ppl)	bookcorpus (ppl)
Baseline	5.665	7.723
Performer	7.364	8.957
Big Bird	7.798	9.132
GFormer	6.837	8.558

TABLE III
TOP-1 ACCURACY OF PRE-TRAINED VISION TRANSFORMER BASE ON IMAGENET WITH DIFFERENT ATTENTION REPLACEMENTS. ACC. Δ REPRESENTS THE AVERAGE ACCURACY DROP TO BASELINE.

Model	Top-1 Acc	Acc. Δ
Baseline	81.5%	-
Performer	79.9%	-1.6%
Big Bird	79.6%	-1.9%
GFormer	80.1%	-1.4%

with the powerful representational capabilities of Transformers. Specifically, Scatterbrain [13] presents a hybrid attention algorithm that combines the benefits of sparse and low-rank attention mechanisms. This approach aims to capture the advantages of both methods, offering a more flexible and efficient solution for approximating attention in Transformers. Even though this method showcases the higher attention approximation accuracy, it doesn't consider hardware efficiency.

Works such as DOTA [29] and ViTALiTy [30] exemplify specific hardware designs to meet the unique demands of Transformers. DOTA introduces a method to dynamically identify and omit attentions with minimal impact on performance, allowing for significant acceleration by reducing computational load. ViTALiTy, on the other hand, combines low-rank and sparse approximation techniques within a novel hardware design to accelerate Vision Transformers, showcasing the potential of custom hardware solutions to optimize performance efficiently.

On the other hand, several works leverage existing hardware to accelerate Transformer models effectively. Fang, et al. [31] presents a co-optimized framework that intelligently adjusts

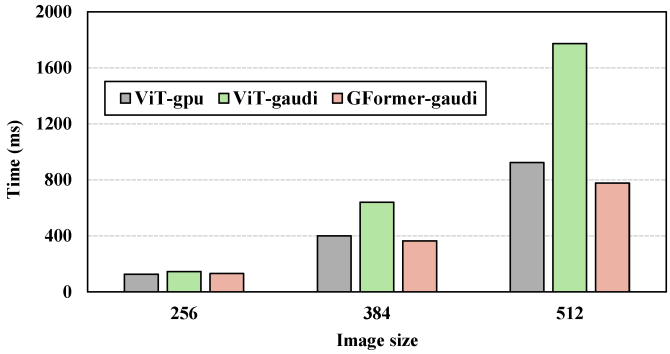

Fig. 15. Performance of ViT on GPU and Gaudi.

both the algorithmic and hardware aspects to exploit the sparsity of transformers, making it possible to achieve substantial acceleration without the need for specialized hardware. Yu, et al [32] discuss GPU-friendly 2:4 fine-grained structured sparsity and quantization for Transformers. Liu et al. [33] exploit the dynamic sparsity in the attention of Transformers and provide corresponding GPU optimization. Zhao, et al. [34] introduce an FPGA-based Transformer accelerator with an output block stationary dataflow to minimize the repeated memory access by block-level and vector-level broadcasting.

While specialized hardware designs offer increased flexibility, they necessitate knowledge of the hardware domain and pose challenges in real-world implementation. Our GFormer leverages existing hardware, enabling large-scale applications in real-world scenarios without additional effort. Unlike FPGA-based designs, which are complex to program, GFormer is user-friendly. Moreover, GFormer provides an economical alternative to GPU designs, delivering comparable functionality and performance at a lower cost. Targeting commercial, emerging heterogeneous hardware, *GFormer underscores the potential of heterogeneous computing as an effective path for accelerating Transformer-based DL tasks*.

VI. CONCLUSION AND FUTURE WORK

Softmax operation is one of the major performance bottlenecks on the Gaudi processor, particularly when processing long sequences. In this work, we introduce an integrated approach that combines sparse and linear attention mechanisms. Our approach includes a windowed local-context self-attention TPC kernel and an efficient outer product TPC kernel

for processing causal linear attention operations. We evaluate our proposed solution in GPT and ViT models on a Gaudi processor. The evaluation shows that our solution achieves up to 2 $\times$ and 2.2 $\times$ speedups in GPT and ViT compared to the original models using Softmax attention.

In the future, we intend to initially extend our work to enable distributed LLM acceleration across multiple Gaudi cards, focusing on optimized communication. Subsequently, we will adapt our mixed attention mechanism for use with various heterogeneous AI accelerators, including Versal ACAP. Lastly, we aim to investigate the application of Gaudi in other emerging ML tasks, such as graph neural networks.

REFERENCES

[1] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[2] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[3] 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," *arXiv preprint arXiv:2010.11929*, 2020.

[4] A. Katharopoulos, A. Vyas, N. Pappas, and F. Fleuret, "Transformers are rnns: Fast autoregressive transformers with linear attention," in *International conference on machine learning*, pp. 5156–5165, PMLR, 2020.

[5] E. Medina and E. Dagan, "Habana labs purpose-built ai inference and training processor architectures: Scaling ai training systems using standard ethernet with gaudi processor," *IEEE Micro*, vol. 40, no. 2, pp. 17–24, 2020.

[6] M. Emani, V. Vishwanath, C. Adams, M. E. Papka, R. Stevens, L. Florescu, S. Jairath, W. Liu, T. Nama, and A. Sujeeth, "Accelerating scientific applications with sambanova reconfigurable dataflow architecture," *Computing in Science & Engineering*, vol. 23, no. 2, pp. 114–119, 2021.

[7] S. Lie, "Cerebras architecture deep dive: First look inside the hw/sw co-design for deep learning: Cerebras systems," in *2022 IEEE Hot Chips 34 Symposium (HCS)*, pp. 1–34, IEEE Computer Society, 2022.

[8] C. Zhang, B. Sun, X. Yu, Z. Xie, W. Zheng, K. A. Iskra, P. Beckman, and D. Tao, "Benchmarking and in-depth performance study of large language models on habana gaudi processors," in *Proceedings of the SC'23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*, pp. 1759–1766, 2023.

[9] N. Kitaev, Ł. Kaiser, and A. Levskaya, "Reformer: The efficient transformer," *arXiv preprint arXiv:2001.04451*, 2020.

[10] M. Zaheer, G. Guruganesh, K. A. Dubey, J. Ainslie, C. Alberti, S. Ontanon, P. Pham, A. Ravula, Q. Wang, L. Yang, *et al.*, "Big bird: Transformers for longer sequences," *Advances in neural information processing systems*, vol. 33, pp. 17283–17297, 2020.

[11] T. Hofmann, B. Schölkopf, and A. J. Smola, "Kernel methods in machine learning," 2008.

[12] 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," *arXiv preprint arXiv:2009.14794*, 2020.

[13] B. Chen, T. Dao, E. Winsor, Z. Song, A. Rudra, and C. Ré, "Scatterbrain: Unifying sparse and low-rank attention," *Advances in Neural Information Processing Systems*, vol. 34, pp. 17413–17426, 2021.

[14] A. Radford, K. Narasimhan, T. Salimans, I. Sutskever, *et al.*, "Improving language understanding by generative pre-training," 2018.

[15] I. Beltagy, M. E. Peters, and A. Cohan, "Longformer: The long-document transformer," *arXiv preprint arXiv:2004.05150*, 2020.

[16] H. Labs, "Gaudi training platform white paper," tech. rep., November 2020.

[17] H. Labs, "Tpc programming." https://docs.habana.ai/en/latest/TPC/index.html, 02 2024.

[18] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, *et al.*, "Language models are few-shot learners," *arXiv preprint arXiv:2005.14165*, 2020.

[19] P. Kokkinos, T. A. Varvarigou, A. Kretsis, P. Soumplis, and E. A. Varvarigos, "Cost and utilization optimization of amazon ec2 instances," in *2013 IEEE Sixth International Conference on Cloud Computing*, pp. 518–525, IEEE, 2013.

[20] S. Black, G. Leo, P. Wang, C. Leahy, and S. Biderman, "GPT-Neo: Large Scale Autoregressive Language Modeling with Mesh-Tensorflow," 03 2021. If you use this software, please cite it using these metadata.

[21] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, *et al.*, "Huggingface's transformers: State-of-the-art natural language processing," *arXiv preprint arXiv:1910.03771*, 2019.

[22] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale, *et al.*, "Llama 2: Open foundation and fine-tuned chat models," *arXiv preprint arXiv:2307.09288*, 2023.

[23] B. Wu, C. Xu, X. Dai, A. Wan, P. Zhang, Z. Yan, M. Tomizuka, J. Gonzalez, K. Keutzer, and P. Vajda, "Visual transformers: Token-based image representation and processing for computer vision," 2020.

[24] S. Merity, C. Xiong, J. Bradbury, and R. Socher, "Pointer sentinel mixture models," 2016.

[25] Y. Zhu, R. Kiros, R. Zemel, R. Salakhutdinov, R. Urtasun, A. Torralba, and S. Fidler, "Aligning books and movies: Towards story-like visual explanations by watching movies and reading books," in *The IEEE International Conference on Computer Vision (ICCV)*, 12 2015.

[26] 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," *arXiv preprint arXiv:1804.07461*, 2018.

[27] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE conference on computer vision and pattern recognition*, pp. 248–255, Ieee, 2009.

[28] S. T. Brown, P. Buitrago, E. Hanna, S. Sanielevici, R. Scibek, and N. A. Nystrom, "Bridges-2: A platform for rapidly-evolving and data intensive research," in *Practice and Experience in Advanced Research Computing*, pp. 1–4, 2021.

[29] Z. Qu, L. Liu, F. Tu, Z. Chen, Y. Ding, and Y. Xie, "Dota: detect and omit weak attentions for scalable transformer acceleration," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 14–26, 2022.

[30] J. Dass, S. Wu, H. Shi, C. Li, Z. Ye, Z. Wang, and Y. Lin, "Vitality: Unifying low-rank and sparse approximation for vision transformer acceleration with a linear taylor attention," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 415–428, IEEE, 2023.

[31] C. Fang, A. Zhou, and Z. Wang, "An algorithm–hardware co-optimized framework for accelerating n: M sparse transformers," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 30, no. 11, pp. 1573–1586, 2022.

[32] C. Yu, T. Chen, Z. Gan, and J. Fan, "Boost vision transformer with gpu-friendly sparsity and quantization," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 22658–22668, 2023.

[33] L. Liu, Z. Qu, Z. Chen, Y. Ding, and Y. Xie, "Transformer acceleration with dynamic sparse attention," *arXiv preprint arXiv:2110.11299*, 2021.

[34] Z. Zhao, R. Cao, K.-F. Un, W.-H. Yu, P.-I. Mak, and R. P. Martins, "An fpga-based transformer accelerator using output block stationary dataflow for object recognition applications," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 70, no. 1, pp. 281–285, 2022.